PENGANTAR STRUKTUR DATA

Erick Fernando, Ahmad Faza, Aso Sudiarjo, Irmawati,
Wirawan Istiono, Samuel Ady Sanjaya, Risanto Darmawan

PENGANTAR STRUKTUR DATA

Erick Fernando
Ahmad Faza
Aso Sudiarjo
Irmawati
Wirawan Istiono
Samuel Ady Sanjaya
Risanto Darmawan

GETPRESS INDONESIA

PENGANTAR STRUKTUR DATA

Penulis : Erick Fernando
Ahmad Faza
Aso Sudiarjo
Irmawati
Wirawan Istiono
Samuel Ady Sanjaya
Risanto Darmawan

ISBN :

Editor : Hendra Nusa Putra,S.Kom, M.Kom
Penyunting: Yuliatri Novita, M.Hum.
Desain Sampul dan Tata Letak : Atyka Trianisa, S.Pd.

Penerbit: GETPRESS INDONESIA
Anggota IKAPI No. 033/SBA/2022

Redaksi:
Jl. Palarik RT 01 RW 06, Kelurahan Air Pacah
Kecamatan Koto Tangah, Padang, Sumatera Barat
website: www.getpress.co.id
email: adm.getpress@gmail.com

Cetakan pertama, Januari 2024

KATA PENGANTAR

Dengan mengucapkan puji syukur kehadirat Allah SWT, atas limpahan rahmat dan hidayahNya, maka Penulisan Buku dengan Judul Pengantar Struktur Data dapat diselesaikan dengan baik atas kerjasama tim penulis. Buku ini dibuat untuk mahasiswa yang mempelajari mengenai Data. Buku ini bertujuan untuk memahami struktur data, representasi data, array multi dimensi, tumpukan (*stack*), senarai berantai (*linked list*), pengurutan (*sorting*), pencarian (*searching*).

Buku ini masih banyak kekurangan dalam penyusunannya. Oleh karena itu, kami sangat mengaharapkan kritik dan saran demi perbaikan dan kesempurnaan buku ini selanjutnya. Kami mengucapkan terima kasih kepada berbagai pihak yang telah membantu dalam proses penyelesaian Buku ini. Semoga Buku ini dapat menjadi sumber referensi dan literatur yang mudah dipahami.

Padang, Januari 2024
Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
STRUKTUR DATA

Oleh Erick Fernando

1.1 Pendahuluan

Struktur data sangat penting dalam bidang ilmu komputer dan pengembangan perangkat lunak. Struktur data berfungsi sebagai komponen penting untuk meningkatkan efektivitas organisasi dan manipulasi data, dan signifikansinya tidak dapat dilebih-lebihkan (Goodrich, Tamassia and Goldwasser, 2013; Jamro, 2018; Malhotra and Malhotra, 2019). Pemrogram dapat menyimpan, mengambil, dan memodifikasi data secara efektif karena fungsionalitas yang disediakan oleh komponen-komponen ini, yang berfungsi sebagai entitas dasar. Hal ini penting, untuk memperoleh pemahaman yang komprehensif tentang struktur data, untuk menyelidiki berbagai kategori, relevansi yang melekat, dan penerapan klasifikasi tersebut secara luas dengan semakin memperhatikan detail dan evaluasi kritis.

Pilihan struktur data yang sesuai bergantung pada persyaratan spesifik yang terkait dengan tugas yang diberikan. Prosedur ini melibatkan evaluasi cermat terhadap berbagai kriteria, termasuk ukuran kumpulan data, prosedur yang diharapkan, dan batasan yang ditentukan oleh sumber daya memori yang tersedia. Untuk membuat penilaian yang tepat dan mengembangkan solusi tingkat lanjut, sangat penting untuk memiliki pemahaman menyeluruh tentang kelebihan dan kekurangan yang terkait dengan setiap tipe data.

Kesimpulannya, struktur data memegang posisi penting dalam bidang ilmu komputer dan pemrograman. Struktur data ini penting untuk pengelolaan dan manipulasi data, memungkinkan pengembang mengatasi kesulitan kompleks dan menciptakan solusi perangkat lunak yang sangat efisien.

1.2 Konsep dasar

Data adalah representasi dari realitas yang tersimpan dalam berbagai bentuk seperti tulisan, suara, gambar, atau simbol. Informasi ini mencakup fakta-fakta yang direkam, direpresentasikan, atau disimpan dalam berbagai jenis objek, baik itu konstan maupun variabel. Struktur data sendiri merujuk pada cara penyimpanan dan representasi data di dalam komputer. Ini menggambarkan cara di mana informasi diatur sehingga bisa diinterpretasikan, misalnya dalam bentuk array atau struktur, yang memungkinkan operasi-operasi tertentu untuk dilakukan pada data tersebut (Klein, 2016; Mehta and Sahni, 2016; Moshiri and Izhikevich, 2018).

Struktur data mempunyai tiga bagian utama yaitu

1. Penyimpanan informasi secara terorganisir inilah yang dimaksud dengan istilah "penyimpanan data terorganisir". Ini memerlukan penyimpanan informasi dengan cara tertentu melalui penerapan berbagai struktur penyimpanan, termasuk array, daftar, pohon, dan sejenisnya.
2. Kumpulan Variabel Terkait. Ini digunakan untuk menggambarkan hubungan yang ada antara banyak bagian atau variabel yang membentuk struktur data tersebut. Dalam konteks struktur data, terdapat keterkaitan atau keterkaitan yang terjalin antara satu variabel dengan variabel lainnya.

3. Organisasi Hubungan Tertentu: Aspek struktur data ini menyediakan seperangkat aturan atau regulasi yang mengatur cara variabel berinteraksi satu sama lain dan diatur di dalamnya. Untuk mengakses dan memanipulasi data dengan cara yang ditentukan dengan jelas, algoritma dan teknik operasi khusus diterapkan.

Penggunaan struktur data yang cocok dalam proses pemrograman memungkinkan pembuatan algoritma yang lebih jelas dan efisien, yang pada gilirannya membuat keseluruhan program menjadi lebih efisien dan mudah dipahami. Secara garis besar, struktur data terdiri dari beberapa elemen, termasuk koleksi nilai-nilai data serta serangkaian operasi dasar yang beroperasi pada data tersebut sesuai dengan algoritma yang telah ditentukan. Dengan menggunakan struktur data yang tepat, proses pengorganisasian dan manipulasi data menjadi lebih terstruktur dan efisien, memungkinkan penyusunan algoritma yang lebih efektif dan pemrograman yang lebih sederhana.

1.3. Mengapa Struktur Data Penting?

1.3.1 Organisasi Data yang Efisien:

Struktur data memainkan peran penting dalam menentukan organisasi dan penyimpanan data dalam memori komputer. Array memberikan banyak manfaat untuk akses acak karena pemanfaatan memori yang berdekatan. Daftar tertaut menggunakan alokasi memori dinamis untuk memfasilitasi pelaksanaan operasi penyisipan dan penghapusan yang fleksibel. *Trees* dan *graphs* berfungsi sebagai sarana untuk merepresentasikan data secara hierarki dan saling berhubungan.

Dalam beberapa keadaan, struktur data tertentu seperti *hash tables* atau *balanced trees*, seperti AVL trees, digunakan untuk mencapai tujuan spesifik terkait organisasi data.

1.3.2 Desain Algoritma

Membangun algoritma tidak mungkin dilakukan tanpa terlebih dahulu menyiapkan struktur data untuk menanganinya. Dengan memanfaatkan struktur data yang ditawarkan, seseorang dapat membangun berbagai algoritma, yang masing-masing memberikan tingkat tekanan komputasi yang berbeda pada sistem. Ketika digunakan dengan array atau array dinamis, algoritma pengurutan seperti *QuickSort* dan *MergeSort* secara signifikan lebih efektif dibandingkan ketika digunakan dengan daftar tertaut. Array dan array dinamis keduanya merupakan jenis struktur data. Karena adanya struktur data yang terurut seperti *sorted arrays* dan *binary search trees*, maka dimungkinkan untuk membuat algoritma pencarian seperti pencarian biner. Daftar tertaut dan tabel hash adalah dua jenis contoh struktur data terurut lainnya.

1.3.3 Manajemen memori

Proses pengelolaan memori dapat dibuat lebih efektif dan efisien dengan bantuan struktur data yang mampu memberikan dukungan terhadap proses tersebut. Dukungan ini dapat diberikan dalam beberapa cara berbeda. Tumpukan dan antrian adalah dua contoh struktur data dinamis yang secara dinamis mengalokasikan memori jika diperlukan dan membatalkan alokasinya saat tidak lagi diperlukan. Jenis alokasi dan dealokasi memori ini dikenal sebagai alokasi memori dinamis dan dealokasi memori dinamis. Proses mengalokasikan dan mengosongkan memori dinamis dikenal sebagai alokasi dan dealokasi memori dinamis. Situasi ini dapat dicirikan sebagai contoh tindakan alokasi memori dinamis dan mekanisme dealokasi. Memanfaatkan struktur data yang menggunakan manajemen memori khusus dapat membantu mengurangi fragmentasi memori dan meningkatkan pemanfaatan sumber daya. Ini adalah salah satu dari banyak manfaat menggunakan struktur tersebut. Pemanfaatan struktur data sedemikian rupa untuk membatasi

fragmentasi memori memungkinkan hal ini menjadi kenyataan, yang tidak mungkin terjadi tanpa adanya pendekatan seperti itu.

1.3.4 Abstraksi

Menyelidiki struktur data dapat menawarkan pemahaman konseptual yang lebih mendalam tentang data jika dipelajari. Pengembang tidak perlu mengelola memori programnya secara manual jika, misalnya, pengembang menggunakan tumpukan untuk mewakili struktur data masuk terakhir, keluar pertama (LIFO) dalam aplikasinya. Hasilnya, pengembang tidak perlu lagi melakukan manajemen memori langsung.

Struktur kode disederhanakan, keterbacaannya meningkat, dan bahaya kesalahan yang disebabkan oleh manajemen memori manusia diturunkan ketika kode diabstraksi. Karena meningkatnya risiko kesalahan terkait dengan manajemen memori manual, beberapa keuntungan diperoleh.

1.3.5 *Code Reusability*

Sebagian besar bahasa pemrograman sudah dilengkapi dengan struktur data yang cukup fleksibel untuk digunakan dalam berbagai jenis pekerjaan. Struktur ini cukup fleksibel untuk digunakan dalam segala hal mulai dari aplikasi web sederhana hingga program komputer yang rumit. Ada banyak struktur data siap pakai berbeda yang dapat diakses, beberapa contohnya adalah lists, arrays, and dictionaries.

Selain itu, perpustakaan dan kerangka kerja menyediakan *reusable data structures* untuk tujuan tertentu, seperti penyimpanan data, transversal grafik, atau pengembangan antrian prioritas. Struktur data ini dapat digunakan untuk berbagai aplikasi. Struktur data ini memiliki potensi penerapan yang luas dalam berbagai konteks. Struktur ini dapat digunakan dalam berbagai konteks karena luas dan dalamnya penerapannya.

1.3.6 *Performance Optimization*

Pilihan struktur data yang digunakan dalam suatu aplikasi dapat mempunyai pengaruh besar terhadap kinerja program secara keseluruhan. Sangat penting untuk memikirkan hal ini. Tabel hash banyak digunakan karena kemampuannya untuk melakukan operasi pencarian, penyisipan, dan penghapusan dalam kompleksitas *constant-time* (O(1)). Karakteristik ini menjadikannya sangat cocok untuk diimplementasikan dalam database. Fenomena ini merupakan faktor yang berkontribusi terhadap basis penggemar mereka yang besar dan setia. Struktur data yang dioptimalkan, seperti heap, memungkinkan pelaksanaan operasi pada antrean prioritas secara tepat waktu dan efisien. Fenomena ini menjadi mungkin terjadi karena proses optimasi.

1.3.7 *Problem Solving*

Pentingnya struktur data tidak dapat terlalu ditekankan dalam mengatasi beragam kesulitan secara efisien. Penerapan struktur data, seperti pohon percobaan dan sufiks, biasanya digunakan dalam manipulasi data tekstual untuk meningkatkan efisiensi pencocokan pola. Grafik sangat penting dalam beberapa bidang, termasuk namun tidak terbatas pada perencanaan rute, analisis jaringan sosial, dan sistem rekomendasi.

1.3.8 *Resource Management:*

Penggunaan sumber daya yang efisien dalam sistem operasi sangat bergantung pada pentingnya struktur data. Misalnya, bitmap digunakan untuk memantau status alokasi blok memori, membedakan antara blok yang sedang digunakan dan blok yang tersedia saat ini. Daftar tertaut adalah bentuk struktur data yang memfasilitasi pengorganisasian dan administrasi antrian untuk proses atau thread secara efektif.

1.3.8 Manajemen Basis Data

Sistem basis data sangat bergantung pada beragam struktur data untuk mencapai efisiensi maksimum dalam hal penyimpanan dan pengambilan data. B-tree dan indeks hash merupakan komponen penting dalam memungkinkan operasi database yang efisien, khususnya dalam konteks penanganan kumpulan data yang luas. Struktur data khusus, seperti filter mekar, digunakan dalam database dan sistem cache untuk tujuan melakukan verifikasi keanggotaan.

1.3.9 Representasi data

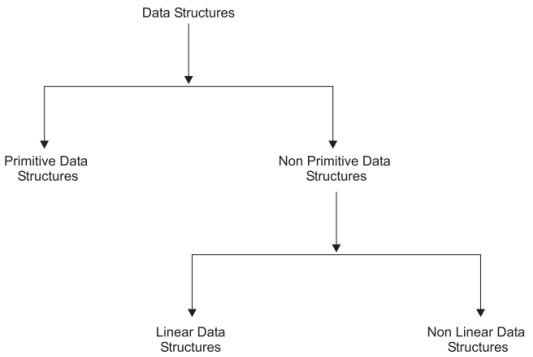
Struktur data memfasilitasi pengorganisasian dan manipulasi data dalam beragam format. Parser XML dan JSON menggunakan struktur hierarki untuk melintasi dan mengubah data. Bitmap umumnya digunakan di bidang pemrosesan gambar karena kemampuannya menyimpan dan mengambil data piksel secara efektif.

Kesimpulannya, struktur data tidak hanya menjadi dasar ilmu komputer dan proses pengembangan perangkat lunak, namun juga merupakan komponen fundamental dari sebagian besar prosedur komputasi yang berbeda. Mereka menyediakan bagi pengembang alat yang diperlukan untuk pengelolaan data yang efektif, penyelesaian masalah yang sulit, dan peningkatan kinerja program. Untuk dapat menulis kode yang efektif dan dapat diandalkan, diperlukan pemahaman komprehensif tentang struktur data, yang juga merupakan komponen penting dari pendidikan ilmu komputer.

1.4 Tipe Struktur data

Struktur data terbagi dalam dua kategori utama: primitif dan non-primitif (Klein, 2016; Balagurusamy, 2019; Knebl, 2020). Struktur primitif mencakup bentuk data dasar yang dapat

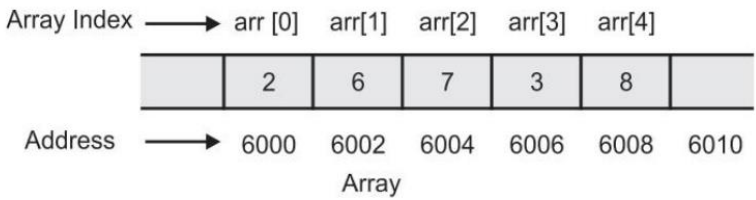
dimanipulasi secara langsung oleh instruksi tingkat mesin, seperti bilangan bulat, karakter, real, dan boolean. Struktur non-primitif, bagaimanapun, berasal dari satu atau lebih tipe primitif. Ini bertujuan untuk membuat kumpulan elemen data, yang sifatnya homogen atau heterogen. Selanjutnya, struktur non-primitif diklasifikasikan menjadi tipe linier dan non-linier. Dalam struktur linier, elemen mengikuti urutan berurutan, terlihat dalam array, tumpukan, antrian, dan daftar tertaut. Sebaliknya, struktur non-linier tidak memiliki tatanan yang tetap; mereka mungkin mengatur elemen secara hierarki, seperti yang terlihat pada pohon dan grafik.



Gambar 1. 1. Tipe Struktur data

1.4.1 Arrays

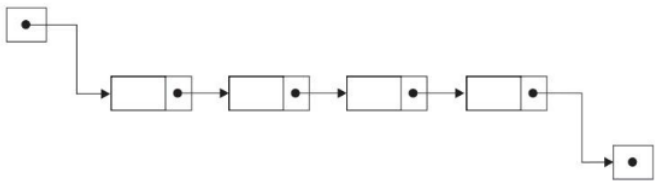
Arrays merupakan kumpulan komponen data dengan tipe yang sama yang disimpan di lokasi yang berurutan dalam memori disebut array. Daftar bilangan bulat, sekelompok nama, dan struktur data lainnya semuanya dapat berfungsi sebagai contoh array. Nama umum, yang disebut sebagai nama array, diberikan kepada kumpulan elemen array. Pengidentifikasi indeks digunakan untuk memfasilitasi akses ke elemen array pada tingkat yang lebih terperinci. Dalam bahasa pemrograman C, indeks array dimulai dengan 0. Misalnya, elemen keenam array 'daftar' dapat dirujuk menggunakan notasi 'daftar[5]'.



Gambar 1. 2. Tipe data Array

1.4.2 Linked Lists

Linked list adalah struktur data yang biasanya digunakan untuk tujuan menyimpan data dalam format seperti daftar. Ini terdiri dari banyak node yang saling berhubungan menggunakan pointer. Setiap node terdiri dari dua komponen. Salah satu komponen struktur menyimpan nilai data, sedangkan komponen lainnya menyimpan referensi ke node berikutnya dalam *Linked list*. *Linked list* mengatasi kelemahan utama yang umumnya dikaitkan dengan array, yaitu penggunaan ruang memori yang boros. *Linked list* menggunakan ruang memori yang sebanding dengan ukuran potongan data penyusunnya. Setiap kali elemen baru perlu ditambahkan ke daftar tertaut, node baru yang sesuai akan dihasilkan. Hal ini berlawanan dengan array, yang mengalokasikan sejumlah ruang memori terlepas dari kebutuhan spesifiknya.

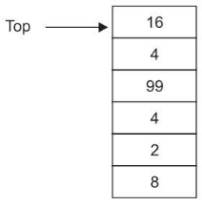


Gambar 1. 3. Linked list

Stacks

Stacks adalah struktur data linier yang mengatur kumpulan komponen sedemikian rupa sehingga memungkinkan operasi penyisipan dan penghapusan dilakukan hanya pada salah satu

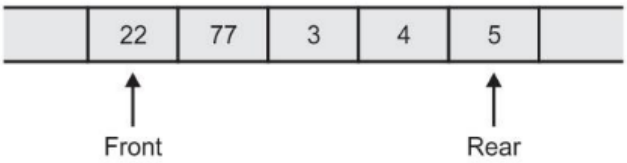
ujung struktur. Ujung *stacks* biasanya dilambangkan sebagai bagian atas. Struktur data Stack beroperasi berdasarkan prinsip Last-In-First-Out (LIFO), dimana bagian yang paling baru ditambahkan adalah bagian pertama yang dikeluarkan dari tumpukan. Tumpukan buku dapat disamakan dengan struktur data *stacks*, karena ia menunjukkan perilaku karakteristik yang memungkinkan penambahan atau penghapusan buku secara eksklusif dari posisi paling atas *stacks*, bukan dari lokasi perantara mana pun dalam *stacks*.



Gambar 1. 4. Representasi logis dari stacks

1.4.3 Queues

Queues adalah jenis struktur data linier yang mengatur elemen seperti daftar. Dalam struktur ini, elemen ditambahkan pada salah satu ujung antrian, yang disebut bagian belakang, dan dihilangkan dari ujung lainnya, yang disebut bagian depan. Struktur data *Queues* beroperasi berdasarkan prinsip First-In-First-Out (FIFO), dimana elemen yang awalnya *di-enqueued* juga merupakan elemen pertama yang *di-dequeued*. Susunan individu yang menunggu di halte dapat dianalogikan dengan struktur data antrian, karena individu masuk antrian di belakang dan keluar di depan.



Gambar 1. 5. Queues

1.5 Variabel dalam Struktur Data

Variabel biasanya mengacu pada ruang penyimpanan tertentu yang bertugas menyimpan data ketika membahas struktur data. Variabel bertanggung jawab untuk menyimpan data. Dalam batas-batas program atau struktur data, nilai dapat dipertahankan dalam bentuk variabel, yang kemudian dapat diedit oleh pemrogram. Proses ini terjadi dalam konteks program atau struktur data. Ada sejumlah faktor, seperti bahasa pemrograman di mana variabel pertama kali dibuat dan konteks penggunaannya selanjutnya, yang berpotensi berdampak pada karakteristik yang dimiliki variabel pada awalnya.

Dalam pemrograman komputer, variabel adalah lokasi penyimpanan jangka pendek di memori komputer yang mampu menyimpan data dan dimanipulasi oleh pemrogram saat program sedang dijalankan. Karena mereka diberi nama yang berbeda, pemrogram dapat membuat referensi tertentu ke lokasi memori dimana nilai atau data disimpan.

Pemrogram dapat mengakses dan memodifikasi data yang sesuai karena setiap variabel memiliki nama berbeda yang membedakannya dari variabel lain dalam sistem. Nama variabel ini berfungsi sebagai pengidentifikasi atau label yang mengarahkan komputer ke arah tempat memori yang tepat dimana data disimpan. Mereka digunakan untuk mengarahkan komputer ke lokasi ini.

Untuk mengakses dan bekerja dengan data yang telah disimpan dalam memori, pemrogram hanya perlu menggunakan nama variabel saat menggunakannya. Tidak perlu mengetahui secara pasti letak data di memori komputer agar dapat diakses dengan cepat dan efisien karena komputer akan secara otomatis mencari alamat memori yang sesuai dengan nama variabel yang diberikan.

Berikut ini adalah daftar beberapa jenis variabel yang ada dalam struktur data yang paling sering dilihat:

1.5.1 *Scalar Variables*

Scalar Variables adalah variabel yang dapat menampung satu nilai dalam satu waktu. Mereka digunakan untuk menyimpan potongan data individual, dan istilah "skalar" biasanya berarti bahwa variabel hanya dapat mewakili satu nilai (berbeda dengan array atau struktur data lain yang dapat menampung banyak nilai). Variabel skalar dapat terdiri dari berbagai tipe data, bergantung pada bahasa pemrogramannya, dan merupakan bagian penting dari sebagian besar bahasa pemrograman.

Tipe data umum untuk variabel skalar meliputi:

Integer: digunakan untuk menyimpan bilangan bulat. Mereka dapat ditandatangani (mengizinkan nilai positif dan negatif) atau tidak ditandatangani (hanya nilai positif). Contohnya termasuk int di C/C++, int di Java, dan int di Python.

Floating-Point: digunakan untuk menyimpan bilangan real dengan titik desimal. Tersedia dalam berbagai ukuran, seperti float, double, dan long double. Contohnya termasuk float dan double di C/C++, dan float dan double di Python.

Character: digunakan untuk menyimpan karakter tunggal, seperti huruf atau simbol. Mereka biasanya direpresentasikan sebagai char dalam bahasa seperti C/C++ dan sebagai karakter tunggal yang diapit tanda kutip tunggal atau ganda dalam banyak bahasa lainnya.

Boolean: dapat menampung satu dari dua nilai: benar atau salah. Mereka sering direpresentasikan sebagai bool dalam C/C++, dan sebagai bool atau boolean dalam bahasa lain.

Berikut adalah contoh variabel skalar dalam berbagai bahasa pemrograman:

```
// Integer scalar variable
int age = 30;

// Floating-point scalar variable
double temperature = 98.6;

// Character scalar variable
char grade = 'A';

// Boolean scalar variable
bool isStudent = true;
```

Gambar 1. 6. Contoh Penggunaan Variable

1.5.2 *Array Variables*

Variabel array mampu menyimpan sekelompok komponen yang semuanya bertipe data sama. Array adalah jenis struktur data yang memungkinkan beberapa nilai disimpan dalam satu nama variabel dan kemudian diambil menggunakan indeks. Mereka mempermudah pengorganisasian dan pengelolaan data dalam bahasa komputer dengan menyediakan fungsionalitas ini.

Berikut ini adalah karakteristik penting dari variabel array:

1. Homogenitas mengharuskan semua elemen yang membentuk array memiliki tipe data yang sama. Misalnya, satu tipe data, seperti angka, karakter, atau apa pun, bisa disimpan dalam array, tapi beberapa tipe data tidak bisa disimpan sekaligus.
2. *Index-based Access:* Indeks suatu elemen dalam array digunakan untuk mengakses elemen dalam array tersebut. Indeks digunakan untuk menunjukkan lokasi suatu elemen di dalam array dan biasanya dimulai pada 0 atau 1, bergantung pada bahasa pemrograman yang digunakan.
3. *Fixed Size:* Setelah array dibuat dalam banyak bahasa pemrograman yang berbeda, ukuran array tetap tidak

berubah setelah dibuat. Array dalam bahasa pemrograman pada umumnya memiliki ukuran yang telah ditentukan, namun bahasa lain menyertakan array dinamis atau dapat diubah ukurannya.

```c
// Declaration and initialization of an integer array
int numbers[5] = {1, 2, 3, 4, 5};

// Accessing elements using indices
int thirdElement = numbers[2]; // Accessing the third element (index 2)
```

```python
# Declaration and initialization of a string array
names = ['Alice', 'Bob', 'Charlie', 'David']

# Accessing elements using indices
third_name = names[2]  # Accessing the third element (index 2)
```

Gambar 1. 7. Array Variables

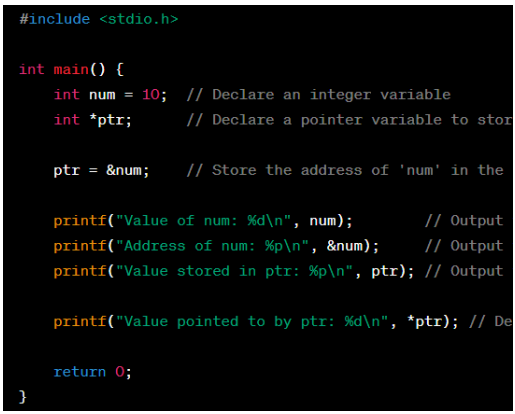
1.5.3 *Pointer variables*

Alamat memori dapat disimpan dalam variabel penunjuk, yang merupakan jenis variabel. Variabel tersebut tidak seperti variabel lain karena tidak membawa nilai secara langsung; sebaliknya, mereka menyimpan alamat ke suatu lokasi di memori komputer di mana suatu nilai disimpan. Pemrograman dibangun berdasarkan gagasan pointer, yang digunakan secara luas dalam bahasa tingkat rendah seperti C dan C++. Pointer adalah aspek penting dari proses pemrograman.

Berikut ini adalah karakteristik penting yang berhubungan dengan variabel pointer:

1. *Memory Address Storage :* menyimpan nilai sebenarnya, pointer bertanggung jawab untuk menyimpan alamat memori variabel atau struktur data lain.

2. Dereferensi pointer, seseorang dapat memperoleh akses ke nilai yang disimpan di alamat memori yang diarahkan oleh pointer.
3. Aritmatika dengan Pointer: Dalam bahasa pemrograman seperti C dan C++, operasi aritmatika dapat dilakukan pada pointer untuk menjelajahi wilayah memori dengan cara yang efisien.
4. Pointer sering digunakan dalam proses alokasi memori dinamis, yang melibatkan alokasi dan dealokasi memori selama eksekusi program.



Gambar 1. 8. *Pointer variables*

1.5.4 *Linked List Variables*

Struktur data linier yang dikenal sebagai *Linked List* terdiri dari node, yang masing-masing menyimpan elemen data selain referensi (juga dikenal sebagai penunjuk) ke node berikutnya dalam urutan daftar. Dalam *Linked List*, variabel digunakan untuk mewakili setiap simpul, dan urutan simpul dalam daftar ditentukan oleh referensi yang digunakan untuk menghubungkan variabel-variabel tersebut.

Pertimbangan penting untuk variabel daftar tertaut:

1. Struktur Node Setiap node dalam *Linked List* biasanya terdiri dari dua komponen: bidang data dan referensi (penunjuk) ke node berikut dalam urutan. Komponen-komponen ini secara kolektif disebut sebagai node.
2. Daftar tertaut memiliki alokasi memori dinamis, oleh karena itu ukurannya dapat berubah sepanjang pelaksanaan program. Ini memberikan *Linked List* struktur yang dinamis.
3. Traversal: Untuk mengakses elemen dalam *Linked List*, Anda harus memulai dari kepala, yang merupakan simpul pertama, dan menelusuri daftar dengan mengikuti referensi yang mengarahkan dari satu simpul ke simpul berikutnya. Proses ini dikenal sebagai traversal.

Ada beberapa variasi *Linked List*, termasuk *Linked List* tunggal (di mana setiap simpul hanya menunjuk ke simpul berikutnya), *Linked List* ganda (di mana setiap simpul mempunyai referensi ke simpul berikutnya dan sebelumnya), dan *Linked List* melingkar (dalam yang mana node terakhir menunjuk kembali ke node pertama).

Gambar 1. 9. *Linked List Variables*

1.5.5 Tipe Data

Tipe data adalah klasifikasi nilai yang dapat diproses oleh program komputer. Tipe data memungkinkan programmer untuk menentukan jenis data yang akan diolah oleh variabel atau fungsi dalam program. Dalam bahasa pemrograman, tipe data mewakili karakteristik dari data itu sendiri. Contohnya, tipe data bisa

berupa integer (bilangan bulat), float (bilangan pecahan), string (kumpulan karakter), dan sebagainya.

Konsep tipe data sangat penting dalam bidang pemrograman. Tipe data digunakan untuk menjelaskan kualitas atau jenis nilai yang mungkin diambil suatu variabel. Komputer dapat menentukan di mana data harus disimpan dalam memori dan jenis operasi apa yang dapat dilakukan pada data tersebut berkat adanya tipe data.

Komputer menyimpan dan memproses informasi dalam bentuk biner, yaitu rangkaian angka 0 dan 1. Namun, menggunakan 0 dan 1 secara langsung untuk menyelesaikan masalah dalam pemrograman akan sangat sulit dan tidak praktis. Oleh karena itu, bahasa pemrograman menyediakan tipe data yang memfasilitasi representasi data di dalam memori komputer. Misalnya, dalam bahasa pemrograman, integer dapat diwakili dengan 2 byte atau 16 bit, sementara float dapat diwakili dengan 4 byte atau 32 bit. Penggunaan tipe data ini memungkinkan programmer untuk berinteraksi dengan komputer menggunakan representasi yang lebih mudah dimengerti dan sesuai dengan kebutuhan program. Dengan demikian, penggunaan tipe data dapat mempermudah proses pengkodean dan memungkinkan penyelesaian masalah yang lebih efisien.

Berikut ini adalah daftar tipe data umum:

1. Bilangan bulat, sering disebut bilangan bulat, adalah bilangan yang tidak mengandung desimal. Ambil contoh ini: (–5), (0), dan (42).
2. Bilangan pecahan, sering disebut float, adalah bilangan yang direpresentasikan menggunakan desimal. 3.14, -0.001, dan 9.99 adalah beberapa contohnya.
3. String dapat digunakan untuk menyimpan teks, kata, atau rangkaian karakter. String dapat berupa teks atau karakter. "Halo, Dunia!", "Python", dan "123" adalah beberapa contohnya.

4. Boolean adalah suatu bentuk logika yang dapat mengambil nilai Benar (benar) atau nilai Salah (salah). Biasanya digunakan dalam konstruksi pernyataan logis. sebagai ilustrasi: benar dan salah.

Tipe data kompleks, seperti:

1. Array: Kumpulan elemen dengan tipe data yang sama. Contohnya: array bilangan bulat [1, 2, 3, 4, 5].
2. Object/Struktur: Kumpulan data yang terdiri dari berbagai tipe data yang berbeda. Contohnya: objek pengguna dengan nama (string), usia (integer), dan alamat (string).

Setiap tipe data memiliki ukuran dan representasi yang berbeda di dalam memori komputer. Misalnya, integer 2 byte (16 bit) dapat menyimpan nilai dari -32,768 hingga 32,767 pada tipe data signed integer, sementara float 4 byte (32 bit) dapat merepresentasikan bilangan pecahan dengan presisi tertentu.

Pemilihan tipe data yang tepat penting agar memori digunakan secara efisien dan operasi data dapat dilakukan dengan benar. Tipe data yang tepat juga membantu programmer dalam menulis kode yang lebih jelas, lebih efisien, dan lebih mudah dipelajari oleh orang lain yang membaca kode tersebut.

DAFTAR PUSTAKA

Balagurusamy, E. (2019) *Data Structures*. McGraw Hill Education.

Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2013) *Data Structures and Algorithms in Python*. John Wiley & Sons.

Jamro, M. (2018) *C# Data Structures and Algorithms*. Packt Publishing Ltd.

Klein, S.T. (2016) *Basic Concepts in Data Structures*, *Basic Concepts in Data Structures*. Available at: https://doi.org/10.1017/cbo9781316676226.

Knebl, H. (2020) *Algorithms and Data Structures*, *Algorithms and Data Structures*. Available at: https://doi.org/10.1007/978-3-030-59758-0.

Malhotra, D. and Malhotra, N. (2019) *DATA STRUCTURES AND PROGRAM DESIGN USING C*. MERCURY LEARNING AND INFORMATION LLC.

Mehta, D.P. and Sahni, S. (2016) *Handbook of and Applications*. Second Edi. CRC Press.

Moshiri, N. and Izhikevich, L. (2018) 'Design and Analysis of Data Structures'. Available at: https://www.amazon.com/dp/1981017232.

BAB 2
REPRESENTASI DATA

Oleh Ahmad Faza

2.1 Pendahuluan

Dalam dunia pemrograman dan pengolahan data, memahami representasi data merupakan hal dasar yang harus dikuasai. Data yang belum melewati proses pengolahan tidak memiliki makna dan memerlukan pemrosesan sehingga dapat memberikan wawasan atau informasi kepada pembacanya. Data dapat hadir dalam berbagai bentuk seperti keadaan, gambar, suara, huruf, angka, ekspresi matematika, bahasa, atau simbol lainnya. Penggunaan data sangat penting untuk memahami lingkungan, objek, peristiwa, atau konsep. Dalam bagian ini, kita akan membahas representasi data dan beberapa tipe data dalam struktur data

2.2 Data Primitif

Dalam struktur data, terdapat istilah **data primitif**. Data primitif ini merupakan jenis data dasar yang bisa ditemukan di hampir semua bahasa pemrograman (Foxwell, 2020; Singh, 2021). Data ini sangat penting karena berperan sebagai blok bangunan dasar untuk membuat program komputer. Mari kita bahas lebih lanjut tentang apa itu data primitif, fungsinya, dan karakteristiknya.

2.2.1. Definisi Data Primitif

Data primitif adalah jenis data yang paling sederhana dalam pemrograman. Mereka adalah jenis data dasar yang sudah tersedia di kebanyakan bahasa pemrograman. Contoh dari data primitif termasuk angka (seperti integer dan float), karakter (seperti huruf dan simbol), dan nilai logika (seperti benar atau salah).

2.2.2 Fungsi Data Primitif

Fungsi utama dari data primitif adalah untuk mewakili nilai tunggal (Singh, 2021). Misalnya, sebuah integer mewakili satu bilangan bulat, dan sebuah karakter mewakili satu huruf atau simbol. Data primitif ini digunakan untuk menyimpan dan mengolah informasi sederhana dalam program.

2.2.3 Karakteristik Data Primitif

Data primitif memiliki beberapa karakteristik khusus (Singh, 2021):

Sederhana : Mereka sederhana dan tidak terdiri dari jenis data lain. Ini berarti data primitif adalah bentuk data yang paling dasar dan tidak rumit.

Tidak Terkomposisi : Data primitif tidak dibuat dari jenis data lain. Mereka berdiri sendiri dan tidak bergantung pada struktur data yang lebih kompleks.

Langsung Dioperasikan oleh Prosesor : Data primitif dapat langsung diolah oleh prosesor komputer. Ini berarti komputer bisa langsung bekerja dengan data primitif tanpa perlu mengubahnya terlebih dahulu.

2.2.4 Contoh Data Primitif

***Integer* (Bilangan Bulat) :** *Integer* adalah jenis data yang digunakan untuk angka tanpa bagian desimal, alias bilangan bulat. Misalnya, 1, 2, 3, dan seterusnya. Integer sering dipakai untuk menghitung atau sebagai penanda posisi dalam sebuah daftar data. Ini adalah jenis data yang sangat umum dan penting dalam pemrograman.

***Float* (Bilangan Pecahan) :** *Float* adalah jenis data untuk angka yang punya bagian desimal. Jadi, kalau ada angka seperti 1.5 atau 3.14, itu termasuk float. Float berguna untuk perhitungan yang butuh ketelitian, seperti dalam matematika atau keuangan. Dengan float, komputer bisa mengolah angka yang lebih rumit daripada hanya bilangan bulat.

***Character* (Karakter) :** Character adalah jenis data yang mewakili satu huruf, angka, atau simbol. Misalnya, 'A', 'b', '3', atau '!'. Dalam pemrograman, karakter digunakan untuk membuat kata atau kalimat, dan juga bisa digunakan untuk mengatur bagaimana program bekerja.

***Boolean* (Logika) :** Boolean adalah jenis data yang sangat sederhana karena hanya punya dua nilai: benar atau salah. Boolean digunakan untuk membuat keputusan dalam program. Misalnya, jika suatu kondisi benar, maka program akan melakukan sesuatu, dan jika salah, program akan melakukan hal lain. Tabel 2.1 merupakan rangkuman perbandingan dari tipe data primitif beserta contohnya:

Tabel 2. 1 Rangkuman Data Primitif

Tipe Data	Definisi	Contoh Kontekstual
Integer	Tipe data yang merepresentasikan bilangan bulat, baik positif maupun negatif.	Usia: 25, Jumlah anak: 3
Float	Tipe data yang digunakan untuk menyimpan bilangan desimal atau bilangan real.	Suhu: 36.5°C, Berat: 70.2 kg
Character	Tipe data yang merepresentasikan karakter tunggal, biasanya disimpan dalam tanda kutip tunggal.	Inisial: 'A', Nilai: 'B'
Boolean	Tipe data yang hanya memiliki dua nilai, yaitu **true** (benar) dan **false** (salah).	Lampu menyala: true, Verifikasi berhasil: false

Jadi, data primitif adalah fondasi dari dunia pemrograman. Mereka seperti bata dasar yang digunakan untuk membangun rumah. Dengan memahami dan menggunakan data primitif, programmer dapat membuat program yang efektif dan efisien, yang mampu melakukan berbagai tugas dan operasi.

2.3 Data Non-Primitif

Data non-primitif ini berbeda dari data primitif karena lebih kompleks dan bisa melakukan lebih banyak hal. Mari kita jelajahi lebih dalam tentang apa itu data non-primitif, fungsinya, dan karakteristiknya.

2.3.1 Definisi Data Non-Primitif

Data non-primitif adalah jenis data yang dibuat dari data primitif. Ini berarti mereka tidak langsung tersedia dalam bahasa pemrograman seperti data primitif, tapi dibuat atau diturunkan dari data primitif. Contoh dari data non-primitif termasuk *array*, struktur, dan kelas (Singh, 2021).

2.3.2 Fungsi Data Non-Primitif

Fungsi utama dari data non-primitif adalah untuk menyimpan kumpulan nilai, bukan hanya satu nilai seperti data primitive (Singh, 2021). Misalnya, sebuah *array* bisa menyimpan banyak bilangan bulat, bukan hanya satu. Ini membuat data non-primitif sangat berguna untuk mengelola dan mengolah informasi yang lebih kompleks dalam program.

2.3.3 Karakteristik Data Non-Primitif

Data non-primitif memiliki beberapa karakteristik khusus:

Lebih Kompleks : Data non-primitif lebih kompleks daripada data primitif. Mereka bisa melakukan lebih banyak hal dan mewakili struktur data yang lebih rumit.

Dapat Disesuaikan : Programmer bisa membuat data non-primitif sesuai dengan kebutuhan mereka. Ini berarti data non-primitif bisa sangat bervariasi dan fleksibel.

Melibatkan Banyak Operasi atau Metode : Data non-primitif sering kali dilengkapi dengan berbagai operasi atau metode yang bisa digunakan untuk mengelola dan memanipulasi data.

Data non-primitif sangat penting dalam pemrograman karena mereka memungkinkan programmer untuk mengelola data yang lebih kompleks dan melakukan operasi yang lebih rumit. Tanpa data non-primitif, akan sulit untuk membuat program yang bisa mengelola banyak data atau melakukan tugas-tugas yang kompleks.

2.3.4 Struktur Data Linear

Definisi Struktur Data Linear

Struktur data linear memungkinkan elemen data dijelajahi secara berurutan (Singh, 2021). Ini berarti bahwa data disusun sedemikian rupa sehingga setiap elemen dapat diakses satu per satu, dalam urutan tertentu. Karakteristik utama dari struktur data linear adalah cara mereka mengatur elemen: setiap elemen terhubung secara linear, baik ke elemen sebelumnya maupun ke elemen berikutnya.

Aksesibilitas dalam Struktur Data Linear

Dalam struktur data linear, hanya satu elemen data yang dapat diakses langsung pada satu waktu. Ini berarti bahwa untuk mencapai elemen tertentu, kita mungkin perlu melalui serangkaian elemen lain dalam urutan tertentu. Aksesibilitas ini membuat struktur data linear ideal untuk aplikasi di mana urutan data penting dan di mana kita perlu memproses data dalam urutan tertentu.

Karakteristik Struktur Data Linear

Karakteristik utama dari struktur data linear adalah pengaturan elemen-elemen mereka dalam urutan linear. Dalam *array*, elemen-elemen disusun dalam urutan berurutan, tetapi tidak terhubung secara langsung satu sama lain. Sebaliknya, dalam struktur seperti *linked list, stack,* dan *queue,* setiap elemen terhubung ke elemen sebelum dan sesudahnya (kecuali elemen pertama dan terakhir). Hal ini memungkinkan operasi seperti penyisipan, penghapusan, dan akses ke elemen menjadi lebih terkontrol dan efisien.

2.3.5 Contoh Struktur Data Linear

Array: Menyimpan Elemen dalam Urutan Tetap

Array adalah kumpulan elemen yang sering digunakan dalam masalah matematika dan pemrograman. Karakteristik utama dari array adalah bahwa elemen-elemennya dirujuk menggunakan indeks. Ini berarti bahwa setiap elemen dalam array dapat diakses langsung dengan mengetahui posisinya dalam array. *Array* sangat berguna untuk menyimpan elemen dalam cara yang berurutan dan ukurannya tetap. Misalnya, dalam aplikasi yang memerlukan penyimpanan data dalam urutan tertentu, seperti nilai dalam tabel, *array* menjadi pilihan yang ideal.

Linked List: Fleksibilitas dalam Penyisipan dan Penghapusan

Linked List adalah kumpulan elemen data yang terdiri dari dua bagian: informasi (Info) dan tautan (Link). Setiap elemen, atau node, dalam linked list berisi data dan referensi ke node berikutnya. Keunggulan utama dari *linked list* adalah ukurannya yang dinamis dan kemampuannya untuk menyisipkan atau menghapus elemen dengan efisien. Ini membuat *linked list* ideal untuk aplikasi di mana jumlah elemen dapat berubah-ubah, seperti dalam manajemen antrian atau dalam implementasi struktur data yang lebih kompleks.

Stack: Prinsip Last-in-First-out

Stack adalah daftar elemen di mana penyisipan dan penghapusan terjadi hanya di satu ujung, yang disebut puncak (*Top*). *Stack* mengikuti prinsip *Last-in-First-out* (LIFO), di mana elemen terakhir yang ditambahkan adalah yang pertama dihapus. Operasi utama dalam *stack* adalah *Push* (menambah) dan *Pop* (menghapus). *Stack* sangat berguna dalam aplikasi seperti membalikkan data, mengurai ekspresi, dan dalam implementasi algoritma tertentu.

Queue: Mengatur Elemen dengan Prinsip *First-in-First-out*

Queue adalah daftar linear di mana elemen ditambahkan di satu ujung (belakang) dan dihapus dari ujung lainnya (depan). *Queue* mengikuti prinsip *First-in-First-out* (FIFO), di mana elemen pertama yang ditambahkan adalah yang pertama dihapus. Queue sangat berguna dalam skenario seperti penjadwalan, buffering, dan dalam pengelolaan sumber daya dalam komputasi. Tabel 2.2 merupakan rangkuman perbandingan dari struktur data linear beserta contohnya:

Tabel 2. 2. Contoh Struktur Data Linear

Struktur Data	Definisi	Contoh Kontekstual Lain
Array	Kumpulan elemen yang memiliki tipe data yang sama, disimpan secara berurutan dan dapat diakses melalui indeks.	Skor permainan: [450, 520, 300, 410]
LinkedList	Kumpulan elemen di mana setiap elemen terhubung ke elemen berikutnya melalui referensi.	Rute kereta: Stasiun A -> Stasiun B -> Stasiun C -> Stasiun D
Stack	Koleksi elemen dengan prinsip Last In First Out (LIFO), di mana elemen terakhir yang ditambahkan akan menjadi yang pertama dikeluarkan.	Tumpukan piring: Piring 3, Piring 2, Piring 1
Queue	Koleksi elemen dengan prinsip First In First Out (FIFO), di mana elemen pertama yang ditambahkan akan menjadi yang pertama dikeluarkan.	Antrian pesan: Pesan 1, Pesan 2, Pesan 3

Struktur data linear seperti *Array*, *Linked List*, *Stack*, dan *Queue* adalah komponen penting dalam pemrograman. Masing-masing memiliki karakteristik unik dan aplikasi praktis yang membuatnya cocok untuk berbagai skenario dalam pemrograman. Dari penyimpanan data sederhana dalam *array* hingga pengelolaan data yang lebih kompleks dalam *stack* dan *queue*, pemahaman yang baik tentang struktur data linear ini sangat penting bagi setiap programmer untuk merancang solusi yang efektif dan efisien.

2.3.6 Struktur Data Non-Linear

Berbeda dengan struktur data linear, struktur data non-linear tidak mengatur elemen data secara berurutan, melainkan dalam cara yang lebih kompleks dan sering kali hierarkis atau berbentuk jaringan. Dalam bagian ini, kita akan membahas definisi, organisasi data, efisiensi memori, dan karakteristik dari struktur data non-linear.

Definisi Struktur Data Non-Linear

Struktur data non-linear didefinisikan oleh pengaturan elemen data mereka yang tidak berurutan. Dalam struktur ini, elemen data tidak hanya terhubung secara linear satu sama lain, tetapi bisa terhubung ke beberapa elemen lain dalam berbagai cara. Ini memungkinkan representasi data yang lebih kompleks dan serbaguna dibandingkan dengan struktur data linear.

Organisasi Data dalam Struktur Data Non-Linear

Dalam struktur data non-linear, satu item data dapat terhubung ke beberapa item data lainnya, bukan hanya dalam urutan linear. Ini menciptakan struktur yang lebih kompleks seperti pohon (*trees*) dan grafik (*graphs*), di mana elemen dapat memiliki banyak hubungan dan tingkat. Misalnya, dalam struktur pohon, setiap *node* dapat memiliki beberapa anak, mewakili hubungan hierarkis. Dalam grafik, node dapat terhubung dengan banyak *node* lain, menciptakan jaringan hubungan yang luas.

Efisiensi Memori Struktur Data Non-Linear

Salah satu keunggulan utama struktur data non-linear adalah efisiensi penggunaan memori mereka. Berbeda dengan struktur data linear yang sering memerlukan alokasi memori secara rutin, struktur data non-linear dapat menggunakan memori lebih efisien karena mereka tidak memerlukan alokasi memori yang besar dan terus-menerus. Ini membuat struktur data non-linear ideal untuk aplikasi yang memerlukan pengelolaan data yang kompleks dan dinamis.

Karakteristik Struktur Data Non-Linear

Struktur data non-linear ideal untuk merepresentasikan hubungan yang kompleks. Mereka sangat berguna dalam aplikasi yang memerlukan representasi hierarkis, seperti dalam struktur pohon, atau dalam representasi jaringan, seperti dalam grafik. Pohon sering digunakan dalam aplikasi seperti struktur direktori file, pengolahan data XML, dan algoritma pencarian. Grafik digunakan dalam aplikasi seperti pemetaan jaringan, analisis jalur terpendek, dan pemodelan hubungan sosial.

2.3.7 Contoh Struktur Data Non-Linear

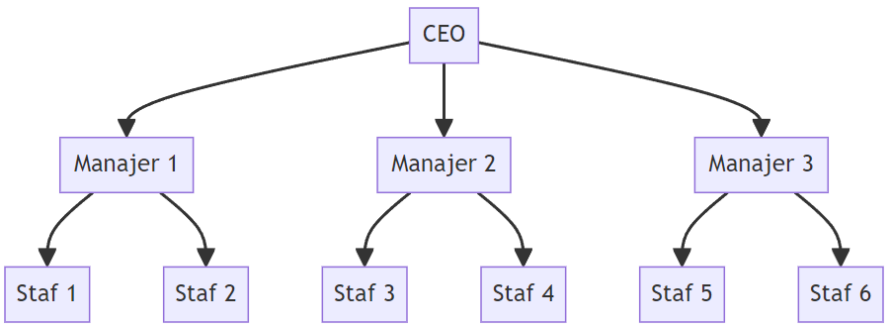
Struktur data seperti pohon (*tree*) dan grafik (*graph*) memainkan peran penting dalam representasi dan manipulasi data yang kompleks dan berhubungan. Kedua struktur data ini termasuk dalam kategori non-linear dan masing-masing memiliki karakteristik khas yang membuatnya cocok untuk berbagai aplikasi. Dalam bagian ini, kita akan menjelajahi deskripsi dan karakteristik kunci dari struktur data pohon dan grafik.

Pohon (*Tree*) : Representasi Hierarkis yang Fleksibel

Pohon adalah struktur data non-linear yang fleksibel, serbaguna, dan kuat. Karakteristik utama dari struktur data pohon adalah kemampuannya untuk merepresentasikan hubungan hierarkis. Contoh klasik dari penggunaan pohon adalah

dalam representasi hubungan keluarga, seperti antara kakek, anak-anaknya, dan cucu-cucunya. Setiap elemen dalam pohon, yang dikenal sebagai node, dapat memiliki hubungan orang tua-anak, di mana satu node orang tua dapat terhubung ke satu atau lebih node anak.

Pohon sangat ideal untuk merepresentasikan data yang memiliki struktur hierarkis atau berjenjang. Misalnya, dalam pengelolaan sistem file komputer, struktur direktori dan subdirektori dapat diwakili dengan pohon. Dalam bidang ilmu komputer, pohon digunakan dalam berbagai algoritma dan struktur data, seperti pohon pencarian biner, pohon AVL, dan pohon merah-hitam, yang masing-masing memiliki aplikasi khusus dalam pencarian dan pengurutan data. Gambar 2.1 merupakan contoh dari struktur data pohon

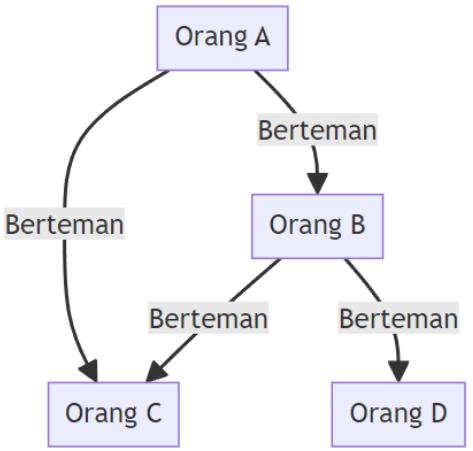


Gambar 2. 1 Contoh Struktur Data Pohon (Tree)

Grafik: Menghubungkan Dunia dengan Hubungan Kompleks

Grafik adalah struktur data non-linear yang terdiri dari himpunan terbatas pasangan terurut yang dikenal sebagai tepi (edges). Elemen-elemen dalam grafik, yang disebut simpul (vertices) atau node, dihubungkan oleh tepi. Karakteristik utama dari grafik adalah kemampuannya untuk merepresentasikan jaringan dan hubungan yang kompleks antara berbagai entitas.

Grafik sangat cocok untuk merepresentasikan jaringan dan hubungan yang kompleks, seperti dalam jaringan sosial, jaringan transportasi, dan grafik dependensi dalam proyek perangkat lunak. Dalam grafik, hubungan antara simpul dapat menggambarkan berbagai jenis koneksi, seperti jarak, kekuatan hubungan, atau arah (dalam grafik berarah). Grafik digunakan dalam berbagai aplikasi, mulai dari analisis jalur terpendek dalam pemetaan GPS hingga analisis jaringan sosial dan pencarian jalur dalam grafik. Gambar 2.2. merupakan contoh dari struktur data grafik.



Gambar 2. 2 Contoh Struktur Data Graph

Tabel 2.3 merangkum perbedaan antara struktur data pohon (tree) dengan graph.

Tabel 2. 3 Struktur Data Non-Linear

Struktur Data	Definisi	Contoh
Tree	Struktur data hierarkis yang terdiri dari *node* (simpul) dengan hubungan parent-*child*. Setiap node dapat memiliki beberapa *child*, tetapi hanya satu parent (kecuali *root* yang tidak memiliki *parent*).	Struktur organisasi perusahaan: CEO di puncak, di bawahnya ada manajer, di bawah manajer ada staf.
Graph	Kumpulan node (atau vertex) yang terhubung oleh edges (sisi). *Graph* dapat berupa directed (arah) atau undirected (tanpa arah), dan dapat memiliki weight (bobot) pada edges-nya.	Jaringan sosial: Orang A berteman dengan B dan C, B berteman dengan C dan D.

Pohon dan grafik adalah dua struktur data non-linear yang sangat penting dalam pemrograman dan analisis data. Pohon menyediakan cara yang efektif untuk merepresentasikan data hierarkis, sementara grafik sangat cocok untuk menggambarkan hubungan yang kompleks dan jaringan antara berbagai entitas. Pemahaman yang baik tentang kedua struktur data ini sangat penting bagi pengembang perangkat lunak dan ilmuwan data, karena mereka menyediakan alat yang kuat untuk mengorganisir, mengelola, dan memanipulasi data dalam cara yang efisien dan intuitif.

2.4 *Abstract Data Types*

Konsep Tipe Data Abstrak (*Abstract Data Types* - ADT) memainkan peran penting dalam pengembangan perangkat lunak yang efisien dan mudah dipahami. ADT adalah sebuah kelas atau tipe untuk objek yang didefinisikan oleh sekumpulan nilai dan operasi. ADT merupakan konsep penting dalam pemrograman yang menawarkan berbagai keuntungan. ADT tidak hanya mempermudah proses pengembangan perangkat lunak, tetapi juga meningkatkan kualitas dan keandalan kode. Bagian ini bertujuan untuk memberikan pemahaman yang lebih dalam tentang ADT, mulai dari definisinya, abstraksinya, hingga komponen-komponennya.

2.4.1 Definisi dan Abstraksi ADT

ADT didefinisikan sebagai tipe atau kelas untuk objek yang ditentukan oleh sekumpulan nilai dan operasi yang dapat dilakukan pada nilai-nilai tersebut. Karakteristik utama dari ADT adalah abstraksinya. ADT menyediakan pandangan yang independen dari implementasi, fokus pada operasi apa yang harus dilakukan, bukan bagaimana operasi tersebut diimplementasikan. Ini berarti bahwa ADT memberikan antarmuka yang memungkinkan pengguna untuk berinteraksi dengan data tanpa perlu memahami detail internal dari implementasinya.

2.4.2 Perspektif Pengguna dan Komponen ADT

Dari perspektif pengguna, mereka yang menggunakan ADT hanya perlu mengetahui operasi apa yang dapat dilakukan oleh ADT tersebut, tanpa harus memahami detail implementasi operasi tersebut. Contoh umum dari ADT adalah tipe data seperti int, float, dan char, di mana pengguna mengetahui nilai yang dapat diambil dan operasi yang dapat dilakukan pada mereka, tetapi tidak dengan implementasi internalnya.

ADT dapat dianggap sebagai "kotak hitam" yang menyembunyikan struktur internal dan desain dari tipe data tersebut. Ini adalah model matematis dari sebuah struktur data. Komponen utama dari ADT meliputi:

Data: Mendeskripsikan struktur dari data yang digunakan.

Operasi: Mendeskripsikan operasi yang valid dan antarmukanya.

Error: Mendeskripsikan mekanisme penanganan kesalahan.

2.4.3 Contoh Operasi ADT

Beberapa contoh operasi yang umum dalam ADT meliputi List, Insert, Delete, Search, dan Sort. Setiap operasi ini memiliki antarmuka yang ditentukan yang memungkinkan pengguna untuk melakukan manipulasi data tanpa perlu memahami detail teknis dari implementasi operasi tersebut.

2.4.4 Keuntungan ADT

Dapat Digunakan Kembali dan Keandalan

Salah satu keuntungan terbesar dari ADT adalah kemampuannya untuk digunakan kembali (*reusability*) di berbagai program. Ini berarti bahwa struktur data yang sama dapat digunakan dalam berbagai konteks tanpa perlu menulis ulang kode. Hal ini tidak hanya menghemat waktu dan upaya, tetapi juga meningkatkan keandalan (*robustness*) dari struktur data tersebut. Karena ADT telah diuji dan digunakan dalam berbagai situasi, keandalannya meningkat seiring waktu.

Pengurangan Koding

ADT menyederhanakan proses koding dengan mengabstraksi detail-detail kompleks dari struktur data. Programmer tidak perlu memikirkan tentang bagaimana data disimpan atau diatur dalam memori; mereka hanya perlu fokus pada operasi yang ingin mereka lakukan. Ini mengurangi beban kognitif pada programmer dan memungkinkan mereka untuk

fokus pada aspek yang lebih penting dari pengembangan perangkat lunak.

Enkapsulasi

Enkapsulasi adalah prinsip penting dalam pemrograman, dan ADT menerapkannya dengan baik. Dengan menyembunyikan struktur internalnya, ADT memastikan integritas data dengan mencegah akses eksternal yang tidak sah. Ini mengurangi risiko korupsi data dan membuat kode lebih aman dan lebih dapat diandalkan.

Prinsip Pemrograman Berorientasi Objek dan Rekayasa Perangkat Lunak

ADT berakar pada prinsip-prinsip Pemrograman Berorientasi Objek (OOP) dan Rekayasa Perangkat Lunak (SE). Ini berarti bahwa penggunaan ADT secara alami mendukung desain yang baik dan fungsionalitas yang ditingkatkan dalam pengembangan perangkat lunak. Prinsip-prinsip ini membantu dalam menciptakan kode yang lebih terstruktur, mudah dipelihara, dan fleksibel.

Spesifikasi Kesalahan

ADT mendefinisikan kondisi kesalahan spesifik yang terkait dengan operasinya, yang membantu dalam penanganan kesalahan yang lebih efektif. Dengan mengetahui jenis kesalahan apa yang dapat terjadi dan bagaimana mereka harus ditangani, programmer dapat menulis kode yang lebih tangguh dan tahan terhadap kesalahan.

2.5 Tantangan pada Representasi Data

Representasi data dalam konteks struktur data memang menghadirkan sejumlah tantangan yang signifikan. Tantangan-tantangan ini sering kali berkaitan dengan bagaimana data

disimpan, diakses, dan dikelola dalam berbagai skenario pemrograman. Berikut adalah beberapa poin penting yang merangkum tantangan-tantangan ini:

Kompleksitas Waktu dan Ruang

Setiap struktur data memiliki kelebihan dan kekurangan dalam hal kecepatan akses dan penggunaan memori. Misalnya, array mungkin cepat dalam akses data tetapi kurang efisien saat menambah atau menghapus elemen. Memilih struktur yang tepat memerlukan pemahaman tentang trade-off ini.

Scalability

Penting untuk memastikan bahwa struktur data yang dipilih dapat menyesuaikan diri dengan peningkatan volume data tanpa mengalami penurunan kinerja yang signifikan. Ini sering kali menjadi pertimbangan utama dalam aplikasi yang data-driven.

Fleksibilitas dan *Reusability*

Mendesain struktur data yang dapat diadaptasi untuk berbagai penggunaan tanpa mengorbankan efisiensi adalah tantangan. Fleksibilitas dan kemampuan untuk digunakan kembali adalah kunci dalam pengembangan perangkat lunak yang efisien.

Kesulitan dalam Implementasi

Implementasi struktur data yang lebih kompleks, seperti pohon merah-hitam atau heap Fibonacci, membutuhkan keahlian dan pemahaman mendalam tentang algoritma.

Pemilihan Struktur Data yang Tepat

Memilih struktur data yang paling cocok untuk kebutuhan spesifik suatu aplikasi sering kali memerlukan pertimbangan yang cermat tentang karakteristik data dan operasi yang akan dilakukan.

Pengelolaan Memori

Dalam bahasa pemrograman yang tidak mengelola memori secara otomatis, pengelolaan memori menjadi aspek kritis, terutama dalam menghindari kebocoran memori dan masalah lainnya.

Concurrency* dan *Synchronization

Dalam aplikasi yang menggunakan multithreading, memastikan akses dan modifikasi data yang aman dan efisien oleh berbagai thread adalah tantangan utama.

Serialisasi Data

Proses mengubah struktur data menjadi format yang dapat disimpan atau ditransmisikan, dan kemudian mengembalikannya ke bentuk aslinya, memerlukan pertimbangan khusus, terutama dalam aplikasi jaringan atau penyimpanan data.

Integritas Data dan Validasi

Menjaga konsistensi dan validitas data dalam struktur data adalah penting untuk mencegah kesalahan dan masalah integritas data.

Visualisasi Data

Untuk struktur data yang kompleks, menyajikan data secara visual yang mudah dipahami oleh pengguna atau pengembang bisa menjadi tantangan tersendiri.

2.5 Kesimpulan

Representasi data merupakan fondasi dari pemrograman dan teknologi informasi. Konsep ini tidak hanya penting untuk efisiensi dan efektivitas program, tetapi juga esensial dalam memahami bagaimana informasi diinterpretasikan dan digunakan. Dalam dunia pemrograman, pemahaman yang mendalam tentang berbagai tipe dan struktur data adalah kunci untuk menciptakan solusi yang canggih dan responsif. Representasi data melampaui sekadar penyimpanan; ini tentang bagaimana data diorganisir, diakses, dan dimanipulasi untuk mencapai tujuan tertentu.

Secara keseluruhan, pemahaman yang kuat tentang representasi data dan struktur data, baik linear maupun non-linear, serta penggunaan efektif ADT, adalah aspek kritis dalam pengembangan perangkat lunak. Mereka memberikan dasar yang kuat untuk membangun program yang tidak hanya berfungsi dengan baik tetapi juga mudah dipelihara dan diperluas, memenuhi tuntutan teknologi dan kebutuhan pengguna yang terus berkembang.

DAFTAR PUSTAKA

Foxwell, H.J. (2020) *Creating Good Data: A Guide to Dataset Structure and Data Representation*. Apress.

Singh, A. (2021) *Data Structure and Algorithm: Simply In Depth*.

BAB 3
ARRAY MULTI DIMENSI

Oleh Aso Sudiarjo

3.1 Pendahuluan

Dalam era perkembangan teknologi informasi saat ini, pemrograman komputer telah menjadi suatu yang sangat dibutuhkan. Dengan semakin bertambahnya kompleksitas dari tugas yang harus diselesaikan oleh program-program komputer. Salah satu struktur data yang memainkan peran penting dalam penyimpanan dan manipulasi data adalah array multi-dimensi.

Array multi-dimensi merupakan konsep yang memungkinkan dalam menyimpan dan mengorganisir data dalam bentuk tabel dua dimensi atau lebih. Konsep array multi-dimensi, mulai dari struktur dasarnya hingga implementasinya dalam berbagai bahasa pemrograman. Array multi-dimensi tidak hanya memberikan cara yang lebih terstruktur untuk menyimpan data, tetapi juga memungkinkan pengembang untuk mengatasi tantangan pemrograman yang lebih kompleks.

Pentingnya pemahaman terhadap array multi-dimensi terletak pada kemampuannya dalam menangani data dengan cara yang efisien, terutama ketika menyelesaikan tugas dengan data yang memiliki hubungan hierarkis atau terstruktur (Lucky E Santoso, 2004). Dengan memahami cara menggunakan dan memanipulasi array multi-dimensi, pengembang dapat meningkatkan kualitas dan efisiensi kode yang dibuat(Santoso, 2021).

3.2 Array Multi Dimensi

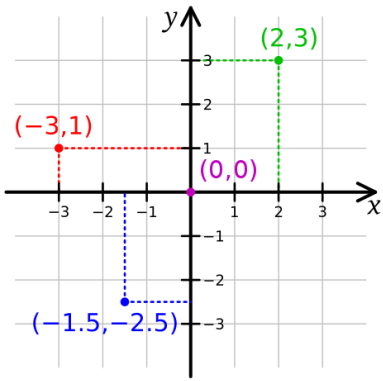
Array adalah kumpulan data dengan tipe integer dan key merupakan kata kunci atau data yang akan di cari pada kumpulan data tersebut (Utami, 2017). Array dapat dikatakan sebagai kumpulan dari nilai nilai data bertipe sama yang berurutan dengan menggunakan nama yang sama. Setiap data akan menempati penyimpanan memory yang berbeda atau disebut dengan elemen array. Elemen array tersimpan dalam indeks yang ada dalam variabel.

Contohnya adalah seperti penomoran sebuah gedung kelas dengan nomor kelas yang terstruktur dan berurutan. Dalam setiasp kelas akan digunakan untuk pembelajaran oleh mahasiswa berisi meja dan kursi. Gedung kelas sebagai array dan elemen dalam array tersebut bertipe sama yaitu meja dan kursi dan nomor kelas adalah sebagai indeks(Khairunnisa and Wulan, 2020).

Dalam penggunaan memori, Array dibagi menjadi dua jenis yaitu array dinamis dan array statis. Array dinamis berarti ukuran penyimpanan untuk elemen array pada array tersebut bisa berubah sedangkan array statis ukuran penyimpanan untuk elemen arraynya tidak bisa dirubah (Septian et al., 2021).

Array dimensi ganda atau multi dimensi merupakan array yang terdiri dari beberapa subskrip atau index array. Array multidimensi adalah struktur data penting dalam banyak aplikasi ilmiah (Suwandi, 2018).Contohnya pada array dua dimensi berarti memiliki dua index array, array tiga dimensi memiliki tiga index array.

Array dua dimensi memiliki penomoran setiap index-nya menggunakan dua angka analoginya seperti penguunaan diagram kartesius, matriks, dan titik koordinat (Boter et al., 2022).



Gambar 3. 1. Digram Kartesius

(Sumber : https://www.gurusiana.id/read/muhtadirajnabi/article/cartesius-3300462)

Penggunaan diagram kartesius digambarkan dalam bentuk grafik. Terdapat sumbu x sebagai array dimensi pertama dan sumbu y sebagai array dimensi kedua, Titik yang terbentuk seperti (2,3) merupakan index nilai dari x dan y.

Array dua dimensi dapat dianalogikan dalam penggunaan matriks. Matriks terdiri dari baris dan kolom. Nilai dalam tiap baris dan kolom memiliki nilai yang berpapasan. Misalnya kolom satu dan baris satu. Sehingga dari teori tersebut dapat diterapkan dalam penggunaan array dua dimensi (Hosseini and Atlasbaf, 2022).

Deklarasi array multi-dimensi melibatkan penentuan jenis data elemen, jumlah dimensi, dan ukuran setiap dimensi. Dalam C++, deklarasi array multi-dimensi melibatkan penentuan tipe data elemen, jumlah dimensi, dan ukuran setiap dimensi. Misalnya, dalam deklarasi array dua dimensi, data dinyatakan dengan jumlah baris dan kolom. Pada bagian ini, akan

menjelaskan sintaksis deklarasi untuk berbagai bahasa pemrograman dan bagaimana menyatakan ukuran dimensi.

Setelah array dideklarasikan, langkah berikutnya adalah memberikan nilai awal atau menginisialisasinya. Inisialisasi dapat dilakukan secara langsung dengan menyediakan nilai untuk setiap elemen atau menggunakan teknik khusus seperti inisialisasi bersarang. Bab ini akan membahas cara menginisialisasi array multi-dimensi dan bagaimana memastikan data awal sesuai dengan kebutuhan program (Hosseini and Atlasbaf, 2022).

Tipe_data nama_array [batas_baris][batas_kolom];

Keterangan :

- Tipe_data : Menyatakan jenis dari tipe data elemen array.
- Nama_array : Menyatakan nama variabel array yang digunakan.
- Batas_baris : Menyatakan jumlah maksimal baris untuk elemen array.
- Batas_kolom : Menyatakan jumal maksimal kolom untuk elemen array.

Contoh 1:

```
//inisialisasi array dua dimensi tanpa index
int matriks [3][3];
//inisialisasi array dua dimensi dengan index
int matriks [3][3]={{2,3,1},{4,6,1},{2,4,3}};
```

Gambar 3. 2. Inisialisasi array dua dimensi

Program tersebut merupakan contoh inisialisasi array dua dimensi dalam bahasa pemrograman C++. Ada dua pendekatan yang digunakan dalam program tersebut, yaitu inisialisasi tanpa indeks dengan hanya mendeklarasikan array dua dimensi tanpa

memberikan nilai awal. Array tersebut diinisialisasi dengan nilai default sesuai tipe datanya, yang pada contoh tersebut adalah int. Namun, nilai-nilai awal ini tidak ditentukan dengan jelas, sehingga mereka dapat mengandung nilai acak yang ada di lokasi memori yang dialokasikan untuk array. Kemudian inisialiasi dengan menggunakan indeks dengan memberikan nilai awal langsung pada saat deklarasi. Array ini memiliki ukuran 3x3, dan elemen-elemen diinisialisasi dengan nilai tertentu. Dalam contoh ini, elemen-elemen matriks diberikan nilai secara eksplisit sesuai dengan urutan baris dan kolom. Sehingga hasil dari inisialiasinya adalah sebagai berikut:

Contoh 2:

```cpp
#include <iostream>

using namespace std;

int main()
{
  int nilai [2][2];

  nilai[0][0] = 4;
  nilai[0][1] = 3;
  nilai[1][0] = 1;
  nilai[1][1] = 2;

  cout << "array nilai adalah:" << endl;
  cout << nilai[0][0] <<" "<< nilai[0][1] << endl;
  cout << nilai[1][0] <<" "<< nilai[1][1] << endl;

}
```

Hasil:

```
array nilai adalah:
4 3
1 2
```

Pada contoh 2, terdapat deklarasi array dengan batas baris 2 dan batas kolom 2. Kemudian pengisian nilai indeks secara manual dengan memanggil indeks yang berpapasan.

Contoh 3:

```
#include <iostream>

using namespace std;

int main() {

    int matriks[4][5]={{6,2,4,2,1},{3,2,6,4,5},{7,1,9,1,5},{5,1,3,5,1}};

    for (int i=0;i<4;i++){
        for (int j=0;j<5;j++){
            cout<<matriks[i][j]<<" ";
        }
        cout << endl;
    }

}
```

Hasil:

```
6 2 4 2 1
3 2 6 4 5
7 1 9 1 5
5 1 3 5 1
```

Contoh 3 merupakan contoh array dua dimensi untuk menampilkan baris matriks. Deklarasi indeks array dengan mengisi 4 baris dan 5 kolom. Kemudian menggunakan

perulangan (looping) untuk menampilkan tampilan matriks. Sehingga indeks nilai array tampil dengan variabel i mewakili perulangan untuk baris dan variabel j mewakili perulangan untuk kolom.

Contoh 4:

```
#include <iostream>

using namespace std;

int main()
{
    //deklarasi
    int a [2] [2] = {{4,3}, {1,2}};

    for(int i=0; i<2; i++){
        for(int j=0; j<2; j++){
            cout << "[" <<i<< " ][ " <<j <<"] =" <<a[i][j] << endl;
        }
    }
}
```

Hasil:

```
[0 ][ 0] =4
[0 ][ 1] =3
[1 ][ 0] =1
[1 ][ 1] =2
```

Pada contoh 4 merupakan array dua dimensi dengan ukuran indeks array adalah 2 untuk baris dan 2 untuk kolom. Dengan menggunakan perulangan (looping) for, variabel i untuk baris dan variabel j untuk kolom. Kemudian akan menampiilkan informasi nilai indeks array berada pada baris dan kolom yang berpapasan.

Contoh 5:

```
#include <iostream>

using namespace std;

int main() {

    int A[3][3];

    for (int b=0;b<3;b++){
        for (int k=0;k<3;k++){
            cout<<"A["<<b<<"]["<<k<<"] = ";
            cin>>A[b][k];
        }
        cout << endl;
    }

    for (int b=0;b<3;b++){
        for (int k=0;k<3;k++){
            cout<<A[b][k]<<" ";
        }
        cout << endl;
    }

}
```

Hasil:

```
A[0][0] = 1
A[0][1] = 5
A[0][2] = 3

A[1][0] = 0
A[1][1] = 9
A[1][2] = 6

A[2][0] = 4
A[2][1] = 8
A[2][2] = 9

1 5 3
0 9 6
4 8 9
```

Pada contoh 5 merupakan contoh untuk mengisi nilai indeks array pada layar output dengan menggunakan cin. Array dengan ukuran indeks 3 baris dan 3 kolom menggunakan perulangan (looping) for dengan b sebagai baris dan k sebagai kolom.

3.3 Kelebihan dan Kekurangan

Kelebihan Array Multi-Dimensi :

a. Representasi Struktur Terstruktur: Array multi-dimensi dapat merepresentasikan struktur data yang lebih kompleks dan terstruktur, memungkinkan penyimpanan data dalam format matriks atau tabel(Agustina et al., 2022).
b. Pemrosesan Data Tabular: Cocok digunakan untuk menyimpan dan memproses data dalam bentuk tabular, seperti data tabel, matriks, atau kubus, yang seringkali muncul dalam aplikasi pemrograman.
c. Kemudahan Akses dan Manipulasi: Memungkinkan akses dan manipulasi elemen-elemen dengan lebih mudah menggunakan indeks-indeks yang sesuai, memberikan efisiensi dalam operasi-operasi matriks dan struktur data terkait.
d. Implementasi Algoritma Matriks: Memudahkan implementasi algoritma yang melibatkan operasi matriks dan pemrosesan data berdimensi tingg
e. Peningkatan Keterbacaan Kode: Menyediakan cara yang jelas dan keterbacaan kode yang baik ketika digunakan dengan bijak, terutama dalam konteks matematika atau ilmu komputer yang memanfaatkan struktur data berdimensi tinggi (Nanda, 2020).

Kekurangan Array Multi-Dimensi:

a. Statis Ukuran: Array multi-dimensi memiliki ukuran yang tetap pada saat deklarasi, yang bisa menjadi keterbatasan jika ukuran matriks perlu berubah secara dinamis selama eksekusi program.
b. Penggunaan Memori Tetap: Matriks dengan ukuran besar dapat menggunakan banyak memori, terutama jika ukurannya sudah ditentukan pada saat kompilasi. Ini dapat menjadi masalah pada sistem dengan keterbatasan memori.
c. Keterbatasan Representasi Data: Meskipun bagus untuk data terstruktur, array multi-dimensi mungkin tidak efisien untuk merepresentasikan data dengan hubungan yang lebih kompleks atau tidak terstruktur.
d. Pengelolaan Elemen Bersarang: Jika elemen-elemen array multi-dimensi adalah objek kompleks atau struktur data bersarang, pengelolaannya dapat menjadi lebih rumit dan membutuhkan lebih banyak kode untuk mengelola memori dengan benar.
e. Ketidakefisienan pada Dimensi Tinggi: Penggunaan array multi-dimensi dengan dimensi yang sangat tinggi dapat menjadi kurang efisien dalam hal pengelolaan memori dan kinerja, terutama jika matriks besar digunakan.

3.4 Studi Kasus Array Multi Dimensi

Studi Kasus : Analisa Matriks dalam perhitungan matematika pada program C++ yang menngunakan array multidimensi dapat membantu dalam analisis matriks dalam konteks perhitungan matematika. Membuat fungsi untuk menampilkan matriks, menjumlahkan dua matriks, mengalikan dua matriks, dan mentransposisi matriks. Kemudian, kita menggunakan matriks contoh matrixA dan matrixB untuk

mendemonstrasikan operasi-operasi tersebut. Program ini memberikan gambaran tentang bagaimana array multi-dimensi dapat digunakan untuk melakukan perhitungan matematis dengan matriks.

```cpp
#include <iostream>
using namespace std;

// Fungsi untuk menampilkan matriks
void displayMatrix(int matrix[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

// Fungsi untuk penjumlahan dua matriks
void addMatrices(int matrix1[3][3], int matrix2[3][3], int result[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

// Fungsi untuk perkalian dua matriks
void multiplyMatrices(int matrix1[3][3], int matrix2[3][3], int result[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result[i][j] = 0;
            for (int k = 0; k < 3; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}

// Fungsi untuk transposisi matriks
void transposeMatrix(int matrix[3][3], int result[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result[j][i] = matrix[i][j];
        }
    }
}
```

```cpp
int main() {
    int matrixA[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int matrixB[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
    int resultMatrix[3][3];

    cout << "Matrix A:" << endl;
    displayMatrix(matrixA);

    cout << "\nMatrix B:" << endl;
    displayMatrix(matrixB);

    // Penjumlahan dua matriks
    addMatrices(matrixA, matrixB, resultMatrix);
    cout << "\nPenjumlahan Matrix A dan B:" << endl;
    displayMatrix(resultMatrix);

    // Perkalian dua matriks
    multiplyMatrices(matrixA, matrixB, resultMatrix);
    cout << "\nPerkalian Matrix A dan B:" << endl;
    displayMatrix(resultMatrix);

    // Transposisi matriks A
    int transposedMatrix[3][3];
    transposeMatrix(matrixA, transposedMatrix);
    cout << "\nTransposisi Matrix A:" << endl;
    displayMatrix(transposedMatrix);AA

    return 0;
}
```

Hasil:

```
Matrix A:
1 2 3
4 5 6
7 8 9

Matrix B:
9 8 7
6 5 4
3 2 1

Penjumlahan Matrix A dan B:
10 10 10
10 10 10
10 10 10

Perkalian Matrix A dan B:
30 24 18
84 69 54
138 114 90

Transposisi Matrix A:
1 4 7
2 5 8
3 6 9
```

Program ini adalah implementasi operasi matriks menggunakan array dua dimensi dalam C++. Fungsi-fungsi terpisah digunakan untuk menampilkan matriks, menjumlahkan, mengalikan, dan mentransposisi matriks. Program utama menginisialisasi dua matriks, melakukan operasi matriks, dan menampilkan hasilnya. Ini memberikan contoh praktis dari bagaimana array multi-dimensi dapat digunakan dalam konteks perhitungan matematis.

3.5 Kesimpulan

Array multi-dimensi adalah struktur data yang memberikan kerangka terstruktur untuk menyimpan data dalam bentuk matriks atau tabel. Dapat digunakan untuk pemrosesan data berdimensi tinggi, seperti matriks atau kubus, yang sering ditemui dalam bidang matematika, ilmu komputer, dan disiplin lainnya. Kelebihannya terletak pada kemudahan akses dan manipulasi elemen-elemen dengan menggunakan indeks, memungkinkan efisiensi dalam operasi-operasi matriks dan struktur data berdimensi tinggi. Array multi-dimensi juga mendukung implementasi algoritma yang melibatkan operasi matriks dengan cara yang efisien dan jelas. Meskipun demikian, array multi-dimensi memiliki keterbatasan, seperti ukuran yang ditentukan secara statis pada saat deklarasi, yang dapat menjadi kendala jika ukuran matriks perlu berubah secara dinamis. Penggunaan memori yang tetap dan keterbatasan dalam merepresentasikan data kompleks atau tidak terstruktur juga perlu diperhitungkan. Pengelolaan elemen bersarang, terutama jika elemen-elemen tersebut adalah objek kompleks atau struktur data bersarang, dapat menjadi lebih rumit. Keseluruhannya, array multi-dimensi adalah alat yang kuat untuk mengelola data berdimensi tinggi, dan pemilihan antara array multi-dimensi dan struktur data lainnya harus didasarkan pada kebutuhan khusus proyek dan sifat data yang dihadapi.

DAFTAR PUSTAKA

Agustina, S., Permanab, A.Y., Fazric, H.N., Hd, M.R.D., Robie, M., Ricky Firmansyah, 2022. Implementasi Struktur Data Tree Pada Web Blog Sebagai Media Pembelajaran. Jurnal Jitek 2.

Boter, J.M., Dehollain, J.P., Van Dijk, J.P.G., Xu, Y., Hensgens, T., Versluis, R., Naus, H.W.L., Clarke, J.S., Veldhorst, M., Sebastiano, F., Vandersypen, L.M.K., 2022. Spiderweb Array: A Sparse Spin-Qubit Array. Phys Rev Appl 18. https://doi.org/10.1103/PhysRevApplied.18.024053

Hosseini, S.M., Atlasbaf, Z., 2022. Analysis of Connected Arrays and Capacitively Coupled Arrays. IEEE Access 10. https://doi.org/10.1109/ACCESS.2022.3156594

Khairunnisa, K., Wulan, N., 2020. Perancangan Intelligent Tutoring System Sebagai Upaya Inovatif Pada Pembelajaran Algoritma dan Struktur Data. ALGORITMA : JURNAL ILMU KOMPUTER DAN INFORMATIKA 4. https://doi.org/10.30829/algoritma.v4i2.8515

Lucky E Santoso, 2004. Standard Template Library C++ untuk Mengajarkan Struktur Data. Jurnal Teknik 5.

Nanda, D.R., 2020. Algoritma, Tipe Data dan Struktur Data. Fakultas Komputer.

Santoso, J.T., 2021. STRUKTUR DATA dan ALGORITMA (Bagian 1), Penerbit Yayasan Prima Agus Teknik.

Septian, H., Suhartini, I., Pertrio, I., Jihad, L.A., 2021. IMPLEMENTASI STRUKTUR DATA TREE PADA GAME PACMAN DENGAN C. Jurnal Digit 11. https://doi.org/10.51920/jd.v11i2.198

Suwandi, A., 2018. PENGERTIAN STRUKTUR DATA. ARTIKELSTRUKTURDATA PENGERTIANSTRUKTURDATA.

Utami, E., 2017. Langkah Mudah Belajar Struktur Data Menggunakan C/C++. [WWW Document]. PT Alex Media Komputindo.sila.

BAB 4
TUMPUKAN (STACK)

Oleh Irmawati

4.1 Dasar-Dasar Tumpukan (Stack)

Tumpukan (Stack) adalah struktur data yang mengikuti prinsip Last In First Out (LIFO), yang berarti elemen yang terakhir dimasukkan ke dalam tumpukan akan menjadi yang pertama keluar. Prinsip dasar tumpukan melibatkan dua operasi utama: push, yang menambahkan elemen ke puncak tumpukan, dan pop, yang menghapus elemen dari puncak tumpukan (Weiss, 2014). Selain itu, operasi top (atau peek) memungkinkan kita melihat elemen teratas tanpa menghapusnya.

Operasi Utama:

- Push: Menambahkan elemen ke puncak tumpukan.
- Pop: Menghapus elemen dari puncak tumpukan.
- Top/Peek: Melihat elemen teratas tanpa menghapusnya.

Tumpukan mengikuti sifat LIFO, dimana elemen yang terakhir dimasukkan menjadi yang pertama keluar, dan sebaliknya. Implementasinya dapat menggunakan array atau linked list, dengan batasan kapasitas tumpukan jika diimplementasikan dengan array. Pemahaman dasar tumpukan memiliki aplikasi praktis dalam berbagai konteks, termasuk dalam manajemen memori, algoritma rekursif, dan parsing ekspresi matematika. Contoh Penggunaan:

- Pemanggilan fungsi (call stack) dalam pemrograman.
- Undo/Redo dalam aplikasi.

- Evaluasi ekspresi matematika.

Aplikasi Praktis:

- Dalam manajemen memori.
- Dalam algoritma rekursif.
- Parsing ekspresi matematika.

Meskipun tumpukan memiliki kelebihan, seperti kecepatan operasi push dan pop serta sederhananya, namun juga memiliki kekurangan, seperti kapasitas tumpukan yang terbatas dan ketidakefisienan dalam pencarian elemen di tengah tumpukan.

4.1.1 LIFO (Last In, First Out)

Metode LIFO (Last In First Out) adalah prinsip fundamental yang diterapkan dalam struktur data tumpukan. Prinsip ini mengatur cara elemen-elemen dikelola di dalam tumpukan, di mana elemen yang terakhir dimasukkan akan menjadi elemen yang pertama dihapus atau diakses.

1. Penambahan Elemen (Push):
    - Elemen baru ditambahkan ke puncak tumpukan.
    - Elemen yang baru ditambahkan menjadi elemen teratas atau yang paling baru di tumpukan.
    - Contoh: Jika elemen A, B, dan C ditambahkan secara berurutan, urutan tumpukan menjadi C (teratas) - B - A (terbawah).
2. Penghapusan Elemen (Pop):
    - Elemen di puncak tumpukan dihapus atau "dipop".
    - Operasi pop mengambil elemen teratas dan menghapusnya.
    - Contoh: Jika operasi pop dilakukan setelah menambahkan elemen A, B, dan C, elemen C akan dihapus, dan tumpukan menjadi B (teratas) - A (terbawah).

3. Melihat Elemen Teratas (Top/Peek):
    - Elemen teratas tumpukan dapat dilihat tanpa dihapus menggunakan operasi top atau peek.
    - Operasi top memberikan nilai dari elemen teratas tanpa mengubah tumpukan itu sendiri.
    - Dengan contoh di atas, operasi top akan memberikan nilai B.

Metode LIFO ini memiliki aplikasi luas, seperti dalam pemanggilan fungsi (call stack) dalam pemrograman, manajemen memori, dan implementasi fitur undo/redo dalam aplikasi. Prinsip "terakhir masuk, pertama keluar" ini menyederhanakan manipulasi data dalam konteks tumpukan dan memudahkan pemahaman perilaku tumpukan (Goodrich, Tamassia and Goldwasser, 2014).

Berikut adalah contoh sederhana penggunaan tumpukan (stack) dalam C++. Kita akan membuat program yang menggunakan tumpukan untuk menyimpan dan mengeluarkan elemen-elemen secara LIFO (Last In First Out):

```
#include <iostream>
#include <stack>
int main() {
   // Membuat tumpukan (stack) integer
   std::stack<int> myStack;
   // Menambahkan elemen ke dalam tumpukan (push)
   myStack.push(5);
   myStack.push(10);
   myStack.push(3);
  // Menampilkan elemen teratas tanpa menghapusnya (top/peek)
   std::cout << "Elemen teratas tumpukan: " << myStack.top() << std::endl;
   // Mengeluarkan elemen teratas dari tumpukan (pop)
   myStack.pop();
   // Menampilkan elemen teratas setelah pop
   std::cout << "Elemen teratas setelah pop: " << myStack.top() << std::endl;
   // Menampilkan jumlah elemen dalam tumpukan
   std::cout << "Jumlah elemen dalam tumpukan: " << myStack.size() << std::endl;
   // Memeriksa apakah tumpukan kosong
   if (myStack.empty()) {
      std::cout << "Tumpukan kosong." << std::endl;
   } else {
      std::cout << "Tumpukan tidak kosong." << std::endl;
   }
   return 0;
}
```

4.1.2 Operasi Dasar pada Tumpukan

Struktur data yang menggunakan prinsip Last In, First Out (LIFO), yang berarti elemen terakhir yang dimasukkan adalah yang pertama kali dikeluarkan. Operasi dasar pada tumpukan melibatkan empat operasi utama (Cormen *et al.*, 2022):

1. Push (Tambahkan) :
   Operasi ini digunakan untuk menambahkan elemen baru ke tumpukan. Elemen ini ditambahkan pada bagian atas tumpukan. Ketika sebuah elemen "dipush" ke tumpukan, ia menjadi elemen paling atas dan menjadi elemen yang dapat diakses paling awal.
2. Pop (Hapus) :
   Operasi ini digunakan untuk menghapus elemen teratas dari tumpukan. Saat elemen dihapus, elemen di bawahnya menjadi elemen teratas yang baru. Pop digunakan untuk mengakses dan menghapus elemen yang terakhir ditambahkan ke tumpukan.
3. Peek atau Top (Lihat atau Atas) :
   Operasi ini memungkinkan untuk melihat elemen teratas tumpukan tanpa menghapusnya. Ini berguna untuk melihat nilai atau mengakses informasi tanpa mengubah struktur tumpukan.
4. isEmpty (Apakah Kosong) :
   Operasi ini digunakan untuk memeriksa apakah tumpukan kosong atau tidak. Jika tidak ada elemen di dalam tumpukan, operasi ini mengembalikan nilai benar (true), dan sebaliknya.

Berikut adalah contoh penerapan operasi dasar pada tumpukan menggunakan *pseudocode*:

```
Inisialisasi tumpukan

Push(tumpukan, elemen1)

Push(tumpukan, elemen2)

Push(tumpukan, elemen3)

Tampilkan "Elemen teratas: " + Peek(tumpukan)

Pop(tumpukan)

Tampilkan "Setelah pop, elemen teratas: " +
Peek(tumpukan)

Jika isEmpty(tumpukan) {

   Tampilkan "Tumpukan kosong"

} else {

   Tampilkan "Tumpukan tidak kosong"

}
```

Dalam contoh ini, elemen1, elemen2, dan elemen3 ditambahkan ke tumpukan menggunakan operasi Push. Kemudian, operasi Peek digunakan untuk melihat elemen teratas tanpa menghapusnya, dan operasi Pop digunakan untuk menghapus elemen teratas. Operasi isEmpty digunakan untuk memeriksa apakah tumpukan kosong atau tidak.

4.2 Implementasi Tumpukan

Implementasi tumpukan (*stack*) melibatkan dua operasi utama: push (menambahkan elemen ke tumpukan) dan pop (menghapus elemen dari tumpukan). Tumpukan memiliki sifat LIFO (Last-In-First-Out), yang berarti elemen yang terakhir

dimasukkan adalah yang pertama kali dikeluarkan (Goodrich, Tamassia and Goldwasser, 2013).

Implementasi tumpukan dapat bervariasi tergantung pada bahasa pemrograman yang digunakan dan kebutuhan spesifik dari aplikasi atau algoritma yang menggunakan struktur data ini. Jika Anda mencari implementasi dalam bahasa pemrograman tertentu, saya sarankan untuk merujuk pada dokumentasi resmi bahasa tersebut atau sumber daya pembelajaran yang relevan.

4.2.1 Representasi Tumpukan dalam Memori

Tumpukan (stack) dapat direpresentasikan dalam memori menggunakan dua pendekatan utama : menggunakan array atau linked list.

1. Representasi dengan Array :
   Dalam representasi menggunakan array, elemen-elemen tumpukan disusun secara berurutan dalam suatu array. Setiap elemen memiliki indeks yang menunjukkan posisinya dalam tumpukan. Indeks 0 umumnya merujuk pada elemen paling atas tumpukan. Ketika elemen ditambahkan atau dihapus, indeks paling atas diperbarui sesuai. Media Sosial : Data dari platform seperti Facebook, Twitter, Instagram, dan lainnya termasuk dalam kategori ini. Mencakup teks, gambar, video, dan berbagai aktivitas pengguna.
2. Representasi dengan Linked List:
   Dalam representasi menggunakan linked list, setiap elemen tumpukan disimpan dalam suatu node. Node-node ini terhubung satu sama lain, dan node terakhir umumnya menunjuk ke elemen teratas tumpukan. Saat elemen ditambahkan atau dihapus, node-node dalam linked list disesuaikan untuk mempertahankan struktur tumpukan.

Referensi ini memberikan pandangan mendalam tentang bagaimana tumpukan direpresentasikan dalam memori menggunakan array atau linked list. Dokumentasi resmi bahasa pemrograman atau buku-buku tentang struktur data dan algoritma juga dapat memberikan informasi lebih lanjut tentang implementasi tumpukan dalam konteks pemrograman yang berbeda.

4.2.2 Operasi Push dan Pop

Operasi Push dan Pop umumnya merujuk pada operasi dasar yang dilakukan pada struktur data tumpukan (stack) dalam pemrograman komputer. Tumpukan adalah struktur data linier yang mengikuti prinsip LIFO (Last In, First Out), yang berarti elemen yang terakhir dimasukkan ke dalam tumpukan akan menjadi elemen pertama yang diambil keluar.

1. Push:
   Deskripsi: Operasi "Push" digunakan untuk menambahkan elemen baru ke dalam tumpukan.
   Implementasi: Elemen baru ditambahkan ke bagian atas tumpukan.
   Contoh: Ketika Anda menambahkan data baru ke dalam tumpukan, elemen tersebut menjadi elemen teratas tumpukan.
2. Pop:
   Deskripsi: Operasi "Pop" digunakan untuk menghapus elemen teratas dari tumpukan.
   Implementasi: Elemen teratas tumpukan dihapus.
   Contoh: Ketika Anda menghapus data dari tumpukan, elemen yang dihapus adalah elemen teratas tumpukan.

Berikut adalah contoh penggunaan operasi Push dan Pop pada tumpukan (stack) menggunakan bahasa pemrograman C++:

```
#include <iostream>
#include <stack>
int main() {
   // Inisialisasi tumpukan
   std::stack<int> myStack;
   // Push elemen ke dalam tumpukan
   myStack.push(10);
   myStack.push(20);
   myStack.push(30);
```

```
   // Pop elemen dari tumpukan
   int poppedElement = myStack.top();
   myStack.pop();
   // Menampilkan tumpukan setelah operasi Pop
   std::cout << "Tumpukan setelah operasi Pop:";
   while (!myStack.empty()) {
      std::cout << ' ' << myStack.top();
      myStack.pop();
   }
   std::cout << "\nElemen yang di-Pop: " << poppedElement << std::endl;
   return 0;
}
```

Pada contoh di atas, sebuah tumpukan (std::stack) diinisialisasi dan kemudian elemen-elemen (10, 20, dan 30) ditambahkan menggunakan operasi push. Setelah itu, operasi pop dilakukan untuk menghapus elemen teratas dari tumpukan, dan tumpukan ditampilkan setelah operasi pop.

Pastikan untuk menyertakan header <stack> dan menggunakan namespace std untuk menggunakan kelas tumpukan (stack) dari C++.

4.3 Aplikasi Tumpukan

Aplikasi tumpukan, atau stack, memiliki berbagai penerapan dalam dunia pemrograman dan pengembangan perangkat lunak. Struktur data tumpukan mengadopsi prinsip LIFO (Last In, First Out), di mana elemen terakhir yang dimasukkan menjadi elemen pertama yang diambil. Penerapan umum dari tumpukan mencakup manajemen memori dalam sistem operasi, pengelolaan fungsi panggilan dalam pemrograman, dan operasi undo/redo dalam aplikasi pengolah teks atau grafis.

Dalam manajemen memori, tumpukan digunakan untuk melacak alokasi memori dan mengelola pembebasan memori secara efisien. Setiap kali fungsi dipanggil dalam pemrograman, informasi kontekstualnya, seperti variabel lokal dan alamat kembali, dapat disimpan dalam tumpukan. Ini memungkinkan program untuk kembali ke pemanggil fungsi dengan benar setelah fungsi selesai dieksekusi.

Pada tingkat aplikasi, tumpukan juga digunakan untuk mendukung operasi undo/redo. Saat pengguna melakukan tindakan tertentu, status aplikasi pada titik waktu tersebut disimpan dalam tumpukan sehingga dapat dikembalikan jika diperlukan. Ini memberikan fleksibilitas dan kontrol kepada pengguna untuk membatalkan atau mengulangi tindakan dengan mudah.

Dalam pengembangan perangkat lunak, pemahaman yang baik tentang aplikasi tumpukan sangat penting untuk menyusun struktur data dengan efisien dan mengoptimalkan kinerja program. Pemrogram dapat menggunakan tumpukan untuk menangani sejumlah tugas yang melibatkan penyimpanan data

dengan prinsip LIFO, menciptakan solusi yang lebih elegan dan efektif dalam penyelesaian masalah tertentu.

1. Kompresi Data: Teknologi kompresi data digunakan untuk mengurangi ukuran data dan memungkinkan penyimpanan yang lebih efisien.
2. Manajemen Data: Sistem manajemen data Big Data mengatur, membersihkan, dan mengamankan data dalam lingkungan Big Data.
3. Otomatisasi: Otomatisasi proses penyimpanan dan manajemen data dapat membantu dalam manajemen skala yang sangat besar.
4. Analisis Data: Teknologi analisis data, seperti Apache Hadoop dan alat analisis lainnya, digunakan untuk mengekstrak wawasan dari Big Data.

4.3.1 Evaluasi Ekspresi Matematika

Evaluasi ekspresi matematika adalah proses menghitung atau mengevaluasi nilai dari suatu ekspresi matematika. Ekspresi matematika dapat berisi berbagai operasi aritmatika, seperti penjumlahan, pengurangan, perkalian, pembagian, eksponensiasi, dan fungsi matematika lainnya. Untuk menghitung nilai ekspresi tersebut, kita harus mengikuti aturan urutan operasi (operator precedence) dan aturan asosiatif.

Berikut adalah beberapa langkah umum yang dilakukan dalam evaluasi ekspresi matematika:

1. Identifikasi dan Aturan Prioritas Operator: Tentukan operator dalam ekspresi dan ikuti aturan prioritas operator. Contoh, perkalian dan pembagian memiliki prioritas lebih tinggi daripada penjumlahan dan pengurangan.
2. Aturan Asosiatif: Tentukan apakah ada operasi yang memiliki asosiatif kiri atau kanan. Misalnya, eksponensiasi

biasanya memiliki asosiatif kanan, yang berarti kita mengevaluasi dari kanan ke kiri jika ada beberapa operasi eksponensiasi berturut-turut.

3. Evaluasi Ekspresi di dalam Kurung: Hitung nilai ekspresi di dalam kurung terlebih dahulu, mengikuti aturan prioritas operator di dalam kurung.
4. Evaluasi Ekspresi Secara Berurutan: Lakukan evaluasi ekspresi dari kiri ke kanan, memperhatikan aturan prioritas operator.

4.3.2 Membalik Kalimat Menggunakan Tumpukan

Membalik kalimat menggunakan tumpukan adalah salah satu contoh penerapan struktur data tumpukan (stack) dalam pemrograman. Ide dasarnya adalah menggunakan tumpukan untuk menyimpan setiap kata dalam kalimat dan kemudian mengeluarkannya dari tumpukan untuk membentuk kalimat baru dengan urutan yang terbalik.

Berikut adalah contoh sederhana dalam C++ untuk membalik kalimat menggunakan tumpukan:

```
#include <iostream>
#include <stack>
#include <sstream>
int main() {
   std::stack<std::string> wordStack;
   std::string sentence, word;
```

```
// Meminta pengguna untuk memasukkan kalimat
    std::cout << "Masukkan kalimat: ";
    std::getline(std::cin, sentence);
    // Membaca kata-kata dari kalimat dan menambahkannya ke dalam tumpukan
    std::stringstream ss(sentence);
    while (ss >> word) {
        wordStack.push(word);
    }
    // Mengambil kata-kata dari tumpukan untuk membentuk kalimat terbalik
    std::cout << "Kalimat Terbalik: ";
    while (!wordStack.empty()) {
        std::cout << wordStack.top() << " ";
        wordStack.pop();
    }
    return 0;
}
```

4.3.3 Manfaat Tumpukan dalam Pemrograman

Manfaat tumpukan dalam pemrograman sangat signifikan karena memberikan pendekatan efisien dalam pengelolaan data dan pelaksanaan berbagai tugas. Struktur data tumpukan, yang mengikuti prinsip LIFO (Last In, First Out), sering digunakan dalam berbagai konteks pengembangan perangkat lunak. Salah satu manfaat utamanya adalah dalam manajemen memori, di mana tumpukan membantu melacak alokasi memori dan mengelola pembebasan memori secara sistematis. Selain itu, penggunaan tumpukan umumnya ditemukan dalam rekursi dan pemanggilan fungsi, memungkinkan pemanggilan fungsi bersarang untuk ditangani dengan efisien.

Selain itu, tumpukan juga memiliki peran krusial dalam penanganan undo/redo pada aplikasi pengolahan teks, grafis, dan sejenisnya. Kemampuan tumpukan untuk menyimpan status aplikasi pada setiap langkah, dan kemudian mengembalikannya atau mengulanginya, memberikan fleksibilitas kepada pengguna untuk mengelola dan memodifikasi perubahan.

4.4 Analisis Kompleksitas Tumpukan

Analisis kompleksitas tumpukan (stack) melibatkan evaluasi kinerja operasi dasar yang dapat dilakukan pada tumpukan. Tumpukan adalah struktur data linear yang mengikuti prinsip Last In, First Out (LIFO), di mana elemen terakhir yang dimasukkan adalah yang pertama kali diambil (Dasgupta and Papadimitriou, 2006). Operasi dasar pada tumpukan melibatkan penambahan elemen (push) dan penghapusan elemen (pop). Analisis kompleksitas membantu kita memahami seberapa efisien algoritma atau implementasi tumpukan dalam menangani sejumlah elemen.

1. Operasi Push:
   Waktu Kompleksitas: O(1)
   Penjelasan: Menambahkan elemen ke tumpukan melibatkan hanya satu operasi, tanpa perlu iterasi melalui elemen lainnya.
2. Operasi Pop:
   Waktu Kompleksitas: O(1)
   Penjelasan: Menghapus elemen dari tumpukan juga melibatkan satu operasi, dan tidak tergantung pada jumlah elemen di dalam tumpukan.
3. Operasi Peek (Melihat Elemen Teratas):
   Waktu Kompleksitas: O(1)
   Penjelasan: Melihat elemen teratas tumpukan juga dapat dilakukan dengan efisien, tanpa perlu menggeser atau memodifikasi elemen lainnya.

Analisis kompleksitas tumpukan sangat tergantung pada implementasi dan struktur data yang digunakan. Implementasi tumpukan dengan struktur dasar seperti array atau linked list memiliki pengaruh pada kinerja operasi. Meskipun kompleksitas waktu untuk operasi push, pop, dan peek pada umumnya adalah O(1), faktor konstanta dan kecepatan akses elemen dapat bervariasi.

4.4.1 Waktu Eksekusi Operasi Tumpukan

Analisis kompleksitas waktu pada operasi tumpukan (stack) membahas seberapa efisien operasi dasar pada struktur data ini dalam hal waktu eksekusi. Tiga operasi dasar yang umumnya dianalisis adalah push (penyisipan elemen), pop (pengambilan elemen), dan peek (melihat elemen puncak tanpa menghapusnya).

1. Push (Penyisipan Elemen ke Tumpukan):
   Kompleksitas Waktu Terbaik: O(1)
   Kompleksitas Waktu Terburuk: O(1)
   Pada operasi push, elemen baru ditambahkan di puncak tumpukan. Kompleksitas waktu terbaik dan terburuk adalah konstan O(1), karena penambahan elemen tidak tergantung pada jumlah elemen yang ada di dalam tumpukan.
2. Pop (Pengambilan Elemen dari Tumpukan):
   Kompleksitas Waktu Terbaik: O(1)
   Kompleksitas Waktu Terburuk: O(1)
   Operasi pop menghapus elemen puncak tumpukan. Seperti push, kompleksitas waktu pop adalah O(1) terbaik dan terburuk karena tidak tergantung pada jumlah elemen di dalam tumpukan.
3. Peek (Melihat Elemen Puncak Tanpa Menghapusnya):
   Kompleksitas Waktu Terbaik: O(1)
   Kompleksitas Waktu Terburuk: O(1)

Pada operasi peek, kita hanya melihat elemen puncak tanpa menghapusnya. Kompleksitas waktu peek juga tetap konstan O(1), tidak bergantung pada jumlah elemen.

Analisis kompleksitas waktu pada operasi tumpukan menunjukkan bahwa operasi dasar pada tumpukan dapat dilakukan dalam waktu konstan. Hal ini membuat tumpukan menjadi struktur data yang efisien untuk kasus penggunaan di mana operasi-operasi tersebut dominan. Penting untuk dicatat bahwa analisis kompleksitas ini didasarkan pada asumsi implementasi tumpukan yang efisien.

4.4.2 Ruang Memori yang Dibutuhkan oleh Tumpukan

Ruang memori yang dibutuhkan oleh tumpukan (stack) tergantung pada beberapa faktor, termasuk ukuran tumpukan, jenis data yang disimpan di dalamnya, dan implementasi tumpukan tersebut. Mari kita bahas secara umum dan kemudian mencari referensi yang relevan.

1. Elemen Tumpukan:
   Setiap elemen dalam tumpukan membutuhkan ruang memori untuk menyimpan nilainya. Ukuran ruang memori ini tergantung pada jenis data yang disimpan (misalnya, bilangan bulat, karakter, struktur data kompleks, dll.) dan ukuran memori yang dialokasikan untuk jenis data tersebut.
2. Overhead Struktur Data:
   Implementasi tumpukan biasanya memerlukan beberapa overhead struktur data. Ini bisa termasuk penandaan puncak tumpukan, penandaan dasar tumpukan, dan struktur data tambahan yang diperlukan untuk menjaga propertis LIFO (Last In, First Out).
3. Sistem Operasi dan Bahasa Pemrograman:
   Sistem operasi dan bahasa pemrograman yang digunakan juga dapat memengaruhi ruang memori yang dibutuhkan. Misalnya, beberapa bahasa pemrograman mungkin menyediakan mekanisme otomatis untuk mengelola

memori (seperti pengumpulan sampah), sementara yang lain memerlukan manajemen memori manual.

4.5 Penerapan Tumpukan

Penerapan tumpukan dalam struktur data lainnya mencakup sejumlah konsep yang memanfaatkan sifat tumpukan sebagai struktur data berbasis Last In, First Out (LIFO). Berikut adalah beberapa contoh penerapan tumpukan dalam berbagai konteks:

1. Ekspresi Matematika (Infix, Postfix, dan Prefix):
   Penerapan tumpukan dalam konversi ekspresi matematika memungkinkan transformasi notasi infix ke postfix atau prefix. Dengan demikian, evaluasi ekspresi matematika dapat dilakukan dengan lebih efisien.
2. Pemanggilan Fungsi (Function Call):
   Tumpukan digunakan untuk menyimpan alamat pengembalian dan variabel lokal saat suatu fungsi dipanggil. Hal ini memfasilitasi pemanggilan fungsi yang bersifat bertingkat dan pengelolaan nilai pengembalian.
3. Undo/Redo dalam Aplikasi:
   Dengan menyimpan status sebelumnya dalam tumpukan setiap kali perubahan dilakukan, fitur undo/redo dalam aplikasi dapat diimplementasikan. Pengguna dapat membatalkan atau mengulangi aksi dengan mengakses tumpukan tersebut.
4. Pengolahan Bahasa Pemrograman (Parsing):
   Tumpukan membantu dalam analisis sintaksis dan pemahaman struktur bahasa pemrograman. Informasi tentang blok kode dan hierarki struktur disimpan dalam tumpukan selama proses parsing.
5. Manajemen Memori:
   Dalam manajemen memori, tumpukan digunakan untuk melacak alokasi dan dealokasi memori. Alamat memori variabel atau objek dimasukkan ke dalam tumpukan,

memungkinkan pembebasan memori saat variabel atau objek tersebut tidak lagi diperlukan.

6. Rekursi:
   Pada pemrograman rekursif, tumpukan digunakan untuk menyimpan konteks panggilan fungsi yang bersifat rekursif. Setiap panggilan fungsi baru dimasukkan ke dalam tumpukan, dan tumpukan membantu dalam penanganan panggilan fungsi yang bersifat bertingkat.
7. Manajemen Operasi Histori (History Management):
   Dalam aplikasi seperti peramban web, tumpukan digunakan untuk melacak histori operasi pengguna, seperti navigasi mundur (back) dan navigasi maju (forward), memungkinkan pengelolaan riwayat dengan efisien.

Penerapan tumpukan dalam struktur data lainnya sangat bergantung pada kebutuhan khusus dan konteks penggunaannya. Keunggulan utama tumpukan terletak pada kemampuannya untuk melacak urutan operasi dengan efisien sesuai dengan prinsip LIFO.

4.5.1 Penggabungan Tumpukan dengan Antrian (Queue)

Penggabungan tumpukan (stack) dengan antrian (queue) adalah konsep yang mencoba mengintegrasikan dua struktur data tersebut untuk memanfaatkan kelebihan masing-masing. Tumpukan (stack) adalah struktur data yang mengikuti prinsip Last In First Out (LIFO), sedangkan antrian (queue) mengikuti prinsip First In First Out (FIFO). Penggabungan keduanya dapat berguna dalam beberapa konteks di mana perlu dilakukan manajemen data dengan dua pola akses yang berbeda.

Salah satu cara untuk menggabungkan tumpukan dan antrian adalah dengan membuat struktur data baru yang menyatukan karakteristik keduanya. Namun, perlu diingat bahwa penggabungan ini mungkin tidak selalu praktis atau diperlukan dalam setiap situasi, tergantung pada kebutuhan spesifik dari aplikasi atau masalah yang sedang dihadapi.

Berikut adalah contoh program C++ sederhana yang menggabungkan tumpukan dan antrian:

```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;
int main() {
   stack<int> myStack;
   queue<int> myQueue;
   // Menambahkan elemen ke tumpukan
   myStack.push(10);
   myStack.push(20);
   myStack.push(30);
   // Menambahkan elemen ke antrian
   myQueue.push(40);
   myQueue.push(50);
   myQueue.push(60);
   // Mengeluarkan elemen dari tumpukan
   cout << "Stack: ";
   while (!myStack.empty()) {
      cout << myStack.top() << " ";
      myStack.pop();
   } // Mengeluarkan elemen dari antrian
   cout << "\nQueue: ";
   while (!myQueue.empty()) {
      cout << myQueue.front() << " ";
      myQueue.pop();
   }   return 0
```

4.5.2 Penggunaan Tumpukan dalam Struktur Data Graf

Dalam struktur data graf, penggunaan tumpukan (stack) menjadi sangat penting, terutama dalam algoritma penelusuran kedalaman (DFS). DFS adalah metode penelusuran yang dimulai dari simpul awal dan kemudian secara rekursif mengunjungi semua simpul yang terhubung dengannya. Tumpukan digunakan untuk melacak jalur penjelajahan graf dan menyimpan simpul-simpul yang masih perlu dijelajahi.

Langkah-langkah Implementasi DFS dengan Tumpukan:

1. Inisialisasi Tumpukan: Mulai dari simpul awal dan tambahkan simpul tersebut ke tumpukan.
2. Proses DFS:
    - Pop dari Tumpukan: Ambil simpul teratas dari tumpukan.
    - Periksa Kepemilikan: Pastikan simpul tersebut belum dikunjungi sebelumnya.
    - Pencetakan dan Penandaan: Cetak simpul tersebut dan tandai sebagai sudah dikunjungi.
    - Ekspansi Tetangga: Tambahkan tetangga yang belum dikunjungi ke tumpukan.
3. Berulang Hingga Selesai: Ulangi proses ini sampai tumpukan kosong, yang menandakan bahwa semua simpul telah dikunjungi.

Berikut contoh sederhana program C++ yang menggunakan tumpukan dalam konteks struktur data graf. Program ini menyajikan graf berarah dengan beberapa simpul dan menunjukkan cara melakukan Depth First Search (DFS) menggunakan tumpukan.

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
class Graph {
public:
   int vertices;
   vector<vector<int>> adjacencyList;

   Graph(int v) : vertices(v), adjacencyList(v) {}

   void addEdge(int v, int w) {
      adjacencyList[v].push_back(w);
   }

   void depthFirstSearch(int startVertex) {
      vector<bool> visited(vertices, false);
      stack<int> stack;

      cout << "Depth First Search starting from vertex " << startVertex << ": ";

      stack.push(startVertex);
      visited[startVertex] = true;

      while (!stack.empty()) {
         int currentVertex = stack.top();
         stack.pop();
         cout << currentVertex << " ";

         for (const int &neighbor : adjacencyList[currentVertex]) {
            if (!visited[neighbor]) {
               stack.push(neighbor);
               visited[neighbor] = true;
            }
         }
      }
```

```
    cout << endl;
    }
};

int main() {
    // Membuat graf dengan 5 vertex
    Graph graph(5);

    // Menambahkan edge
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 4);

    // Melakukan Depth First Search dari vertex 0
    graph.depthFirstSearch(0);

    return 0;
}
```

Program ini membuat graf dengan 5 vertex dan beberapa edge ditambahkan. Program kemudian memanggil metode DFS dari vertex 0 dan mencetak hasilnya.

DAFTAR PUSTAKA

Cormen, T.H. *et al.* (2022) *Introduction to algorithms*. MIT press.

Dasgupta, S. and Papadimitriou, C.H. (2006) *Algorithms*.

Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2013) *Data structures and algorithms in Python*. John Wiley & Sons Ltd.

Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2014) *Data structures and algorithms in Java*. John wiley & sons.

Weiss, M.A. (2014) 'Data structures and algorithm analysis in C++'.

BAB 5
SENARAI BERANTAI (LINKED LIST)

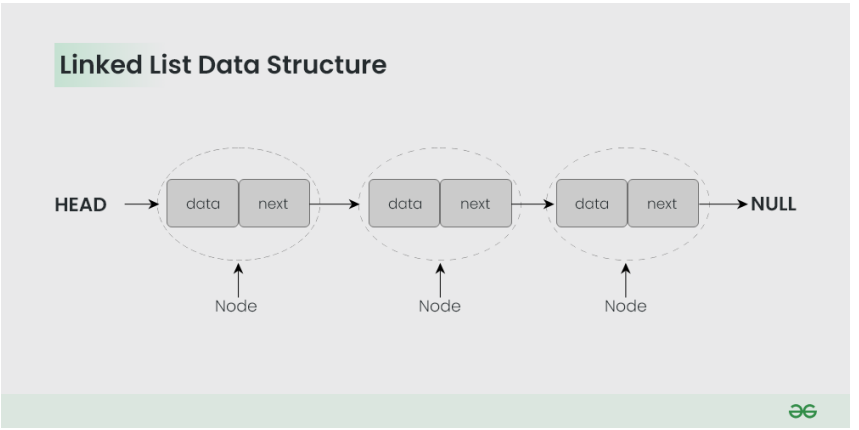
Oleh Wirawan Istiono

5.1 Apa itu Linked List?

Linked list secara luas dianggap sebagai struktur data pilihan untuk mengelola item data dinamis. Linked list terdiri dari sebuah node, yang berfungsi sebagai elemen data. Setiap node di dalam struktur terdiri dari dua bidang berbeda: satu bidang berisi data, dan bidang lainnya berisi alamat yang menyimpan referensi ke node berikutnya. Ringkasnya, linked list adalah (Ravikiran, 2023):

- Linked list adalah jenis struktur data linier yang digunakan untuk menyimpan sekelompok potongan data secara dinamis.
- Node melambangkan item data individual, sedangkan link atau pointer membangun koneksi antara setiap node.
- Setiap node di dalam linked list terdiri dari dua bidang berbeda: bidang pertama menyimpan informasi yang disimpan dalam node, sedangkan bidang kedua berisi penunjuk yang menyimpan alamat memori dari simpul berikutnya dalam linked list.
- Node terakhir dalam urutan diberi nilai null di kolom kedua, yang menunjukkan bahwa node tersebut tidak memiliki referensi ke node berikutnya.
- Ukuran linked list dapat disesuaikan secara dinamis untuk mengakomodasi berbagai kebutuhan.
- Pemanfaatan ruang memori tidak disia-siakan.

Linked list adalah jenis struktur data linier yang tidak menyimpan elemen-elemennya di lokasi memori yang berdekatan. Elemen-elemen linked list saling berhubungan dengan pemanfaatan pointer, seperti tergambar pada gambar terlampir (Parlante, 2002):



Gambar 5. 1. Struktur Data Linked List

(Sumber: Geeksforgeeks.org, 2023)

Struktur Node: Sebuah node dalam linked list biasanya terdiri dari dua komponen:

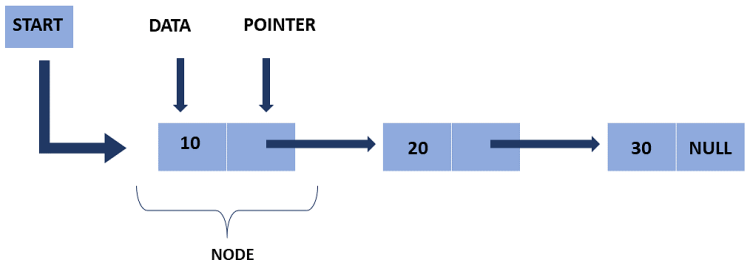
Data: Ini menyimpan nilai atau data aktual yang terkait dengan node.

Petunjuk Berikutnya: Ini menyimpan alamat memori (referensi) dari node berikutnya dalam urutan.

Kepala dan Ekor: Linked list diakses melalui node kepala, yang menunjuk ke node pertama dalam daftar. Node terakhir dalam daftar diberi nilai penunjuk NULL atau nullptr, yang berfungsi sebagai indikasi bahwa ini menandai titik penghentian daftar. Node terminal biasanya disebut sebagai node ekor.

Intinya, linked list adalah struktur data yang terdiri dari node, dimana setiap node menyertakan field data dan referensi, yang biasa disebut link, ke node berikutnya dalam daftar (Van Der Spek*dkk.*, 2008).

Ilustrasi linked list yang diberikan menunjukkan bahwa setiap node terdiri dari dua bidang. Bidang awal mencakup data, sedangkan bidang berikutnya terdiri dari penunjuk yang mereferensikan node lain (Parlante, 2002).



Gambar 5. 2. Representasi Linked List

(Sumber: Ravikiran, 2023)

Kode selanjutnya berkaitan dengan pembuatan node dan deklarasi linked list.

```
struct node {
   int data;
   struct node* next;
};


struct node* n = (struct node*)malloc(sizeof(struct node));
```

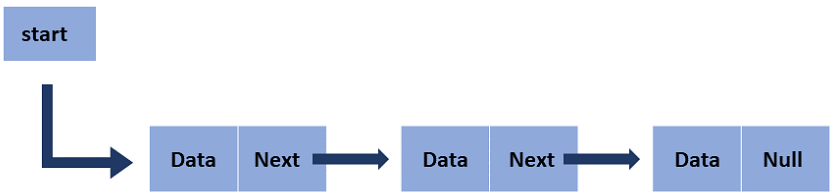
Deklarasi sebuah node memerlukan penyertaan variabel data dan variabel pointer, yang berfungsi untuk menyimpan alamat node berikutnya. Dalam konteks ini, fungsi malloc perlu digunakan untuk mengalokasikan memori secara dinamis untuk node.

5.2 Variasi Linked List

Linked list sebagian besar terdiri dari tiga kategori berbeda, yaitu (Ravikiran, 2023):

Linked List Tunggal

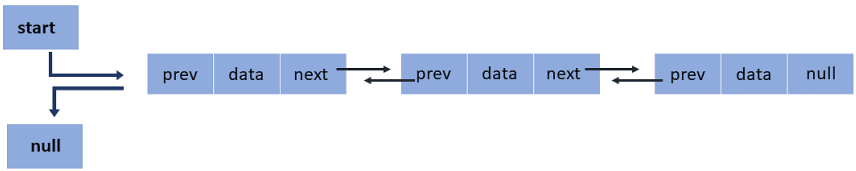
Linked list tunggal adalah struktur data yang terdiri dari urutan node, di mana setiap node berisi nilai dan referensi ke node berikutnya dalam urutan tersebut. Linked list tunggal secara luas diakui sebagai jenis linked list yang paling dominan. Dalam linked list, setiap node terdiri dari data ditambah kolom alamat yang menyimpan referensi ke node berikutnya (Van Der Spek*dkk.*, 2008).



Gambar 5. 3. Representasi Linked List
(Sumber: Ravikiran, 2023)

A. Doubly Linked List

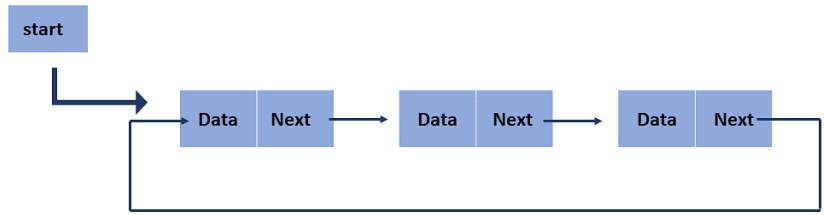
Doubly Linked List adalah struktur data yang terdiri dari urutan node, di mana setiap node memiliki nilai dan dua pointer. Pointer pertama menunjuk ke node sebelumnya. Linked list ganda terdiri dari dua blok penyimpanan penunjuk. Blok penunjuk awal dalam setiap node berisi lokasi memori dari node sebelumnya. Oleh karena itu, dalam inventaris tertaut ganda, terdapat tiga bidang yang berfungsi sebagai penunjuk sebelumnya, yang menyimpan referensi ke node sebelumnya. Selanjutnya, kumpulan data tersebut diikuti oleh penunjuk berikutnya, yang menunjukkan node berikutnya. Oleh karena itu, dimungkinkan untuk melanjutkan ke arah maju dan mundur.



Gambar 5. 4. Representasi Linked List
(Sumber: Ravikiran, 2023)

B. Circular Linked List

Circular Linked List adalah struktur data dalam ilmu komputer yang terdiri dari kumpulan node, di mana setiap node memiliki referensi ke node berikutnya dalam urutan. Linked list melingkar memiliki kemiripan yang mencolok dengan linked list tunggal. Perbedaan satu-satunya adalah pada hubungan antara simpul akhir dan simpul awal, sehingga menghasilkan terbentuknya lingkaran melingkar di dalam linked list melingkar (Van Der Spek*dkk.*, 2008).



Gambar 5. 5. Representasi Linked List
(Sumber: Ravikiran, 2023)

Linked list melingkar dapat diklasifikasikan menjadi linked list tunggal atau ganda. Penunjuk berikutnya dari node berikutnya akan diarahkan ke node awal, sehingga membentuk linked list tunggal. Dalam linked list ganda, penunjuk sebelumnya dari simpul pertama menyimpan lokasi memori dari simpul terakhir.

5.3 Apa perlunya menggunakan struktur data linked list?

Penghitungan berikut menguraikan beberapa keuntungan dari linked list, yang akan membantu dalam memahami pentingnya memperoleh pengetahuan tentang struktur data ini.

- Struktur Data Dinamis: Ukuran memori dapat dialokasikan atau dibatalkan pada saat run time berdasarkan operasi penyisipan atau penghapusan.
- Kemudahan Penyisipan/Penghapusan: Penyisipan dan penghapusan elemen lebih sederhana daripada array karena tidak ada elemen yang perlu digeser setelah penyisipan dan penghapusan, Hanya alamat yang perlu diperbarui.
- Pemanfaatan Memori yang Efisien: Diketahui bahwa Linked List adalah struktur data dinamis yang menyesuaikan ukurannya berdasarkan kebutuhan, sehingga mencegah pemborosan memori.
- Implementasi: Berbagai struktur data tingkat lanjut dapat diimplementasikan menggunakan linked list seperti tumpukan, antrian, grafik, peta hash, dll.

Contoh:

Dalam sistem tertentu, daftar ID yang diurutkan perlu dipertahankan. Hal ini dapat dicapai dengan memanfaatkan array, dinotasikan sebagai id[], yang memiliki elemen berikut: [1000, 1010, 1050, 2000, 2040].

Untuk menjaga urutan yang terurut, semua elemen setelah ID 1000 (tidak termasuk 1000) perlu dipindahkan saat memasukkan ID 1005 yang baru.

Operasi penghapusan dalam array bisa memakan biaya besar kecuali prosedur khusus diterapkan. Sebagai ilustrasi, untuk menghapus elemen 1010 dari array id[], semua elemen berikutnya harus digeser, sehingga memerlukan upaya

komputasi yang besar. Operasi ini berdampak buruk pada efisiensi kode secara keseluruhan (Parlante, 2002).

5.4 Operasi Diperlukan untuk Linked List

Pengoperasian Penting pada Linked List Adalah Sebagai Berikut:

- Traversing: Untuk melintasi semua node satu per satu.

  Dalam prosedur ini, pengguna akan dapat memvisualisasikan semua node yang terdapat dalam linked list.

  Ketika suhu nol, ini menandakan bahwa semua node telah dipindai, dan akhir dari linked list telah tercapai, mengakibatkan penghentian perulangan while.

```
struct node* temp = start;

printf("\nList elements are: ");

while (temp != NULL) {

   printf("%d ", temp->data);

   temp = temp->next;

}
```

- Penyisipan: Untuk menyisipkan node baru pada posisi tertentu.

Anda dapat menambahkan node ke awal, tengah, dan akhir jalur.

a. Insert di Awal

   Pada awalnya, penting untuk menghasilkan alokasi memori untuk node yang baru dibuat. Prosesnya melibatkan penyimpanan data dalam node yang baru dibuat. Ubah node yang berdekatan ke node yang baru

ditambahkan untuk membuat koneksi dengan node awal. Prosesnya dimulai dengan menginstruksikan node yang baru terbentuk.

```
struct node* NewNode = malloc(sizeof(struct node));

NewNode->data = 40;

NewNode->next = start;

start = NewNode;
```

b. Insert di Akhir

Tugas yang ada melibatkan penyisipan node baru ke dalam struktur data, dengan penyimpanan data selanjutnya di dalam node ini.

Untuk mengakses simpul terakhir dari linked list, seseorang harus menelusuri daftar hingga mencapai simpul terakhir.

Ubah penunjuk simpul terakhir untuk mereferensikan simpul yang baru saja dibuat.

```
struct node* NewNode = malloc(sizeof(struct node));

NewNode->data = 40;

NewNode->next = NULL;


struct node* temp = start;

while (temp->next != NULL) {

   temp = temp->next;

}

temp->next = NewNode;
```

c. Insert di Tengah

Cadangan ruang memori dan simpan data dalam node yang baru dibuat. Lintasi node sebelumnya sebelum penyisipan node baru. Ubah penunjuk berikutnya untuk memasukkan penyisipan simpul baru di posisi tengah.

```
struct node* NewNode;
NewNode = malloc(sizeof(struct node));
NewNode->data = 40;

struct node* temp = start;
int position = 3; // Assuming the position where the new node should be inserted

for (int i = 2; i < position; i++) {
   if (temp->next != NULL) {
      temp = temp->next;
   } else {
      // Handle the case when position is greater than the length of the list
      printf("Invalid position\n");
      return;
   }
}

NewNode->next = temp->next;
temp->next = NewNode;
```

- Hapus: Untuk menghapus node dari posisi tertentu.

Operasi penghapusan dalam linked list dapat dilakukan dengan tiga cara berbeda: penghapusan dari akhir, penghapusan dari awal, atau penghapusan dari lokasi yang ditentukan.

a. Delete dari Awal

Titik awalnya terletak pada node kedua.

```
start = start -> next;
```

b. Delete dari Akhir

Akses anggota kedua dari belakang dalam linked list.

Ubah penunjuk berikutnya menjadi nilai nol.

```
struct node* temp = start;
while (temp->next->next != NULL) {
   temp = temp->next;
}
temp->next = NULL;
```

c. Delete dari Tengah

Navigasikan melalui elemen sebelumnya sebelum ke elemen yang ingin dihapus. Ubah penunjuk berikutnya untuk menghapus simpul dari linked list.

```
for (int i = 2; i < position; i++) {
   if (temp->next != NULL) {
      temp = temp->next;
   } else {
      // Tangani kasus ketika posisi lebih besar dari panjang list
      printf("Invalid position\n");
      return;
   }
}

if (temp->next != NULL) {
   temp->next = temp->next->next;
} else {
   // Tangani kasus ketika posisi lebih besar dari panjang list
   printf("Invalid position\n");
}
```

- Pencarian: Untuk mencari elemen dari linked list.

  Operasi pencarian dilakukan untuk menemukan elemen tertentu dalam linked list. Jika elemen tersebut ada di lokasi tertentu, elemen tersebut akan dikembalikan. Alternatifnya, ini akan menghasilkan nilai nol.

```
void search(int item)
{
   struct node *NewNode;
   int i = 1, flag = 0;
   NewNode = start;
   if (NewNode == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      while (NewNode != NULL)
      {
         if (NewNode->data == item)
         {
            printf("Item found at position %d\n", i);
            flag = 1;
            break;
         }
         else
         {
            NewNode = NewNode->next;
            i++;
         }
      }
      if (flag == 0)
      {
         printf("Item not found\n");
      }
   }
}
```

5.5 Pemanfaatan linked list di berbagai domain dan bidang.

Bagian selanjutnya menguraikan berbagai penerapan linked list.

- Linked list biasanya digunakan dalam implementasi tumpukan dan antrian.
- Alat ini digunakan untuk banyak penggambaran pohon dan grafik.
- Alokasi memori dinamis melibatkan pemanfaatan daftar blok bebas yang tertaut.
- Matriks renggang biasanya digunakan untuk tujuan mendeskripsikan struktur data dengan sejumlah besar elemen nol.
- Alat ini digunakan untuk manipulasi polinomial.
- Selain itu, ini digunakan untuk menjalankan operasi aritmatika pada bilangan bulat yang diperluas.
- Teknik ini digunakan untuk tujuan mengidentifikasi rute dalam sistem jaringan.
- Sistem operasi memiliki berbagai aplikasi, termasuk manajemen memori, penjadwalan tugas, dan manajemen sistem file.
- Pemanfaatan linked list dapat meningkatkan efisiensi algoritme yang memerlukan operasi penyisipan atau penghapusan yang sering pada kumpulan data yang luas.
- Penerapan algoritme, seperti cache yang Paling Terakhir Digunakan (LRU), melibatkan pemanfaatan struktur data linked list untuk memantau dan mengelola item yang terakhir diakses dalam cache secara efektif.

Bagian selanjutnya membahas penerapan praktis linked list dalam berbagai skenario dunia nyata.

- Lagu-lagu di pemutar musik saling berhubungan dengan trek sebelumnya dan berikutnya.
- Dalam konteks browser web, URL halaman web sebelumnya dan berikutnya saling berhubungan melalui penggunaan tombol sebelumnya dan berikutnya.
- Keterkaitan antara foto sebelumnya dan berikutnya dalam penampil gambar difasilitasi melalui pemanfaatan tombol sebelumnya dan berikutnya.
- Proses peralihan antara dua aplikasi dilakukan dengan memanfaatkan kombinasi tombol "alt+tab" di sistem operasi Windows dan kombinasi tombol "cmd+tab" di MacBook. Pemanfaatan linked list melingkar diperlukan untuk fungsionalitas yang diinginkan.
- Kontak individu disimpan di ponsel. Informasi kontak yang baru saja dimasukkan akan disusun dalam urutan abjad yang sesuai. Salah satu pendekatan untuk menyelesaikan tugas ini adalah dengan memanfaatkan struktur data linked list untuk menempatkan kontak pada posisi alfabet yang sesuai.
- Perubahan yang dilakukan pada dokumen dihasilkan sebagai node dalam linked list ganda. Isinya dapat diubah dengan memanfaatkan opsi undo, yang dapat diakses dengan menekan kombinasi tombol Ctrl+Z. Tugas ini diselesaikan dengan pemanfaatan fitur linked list.

5.6 Keuntungan dan Kerugian Linked List

Linked list adalah struktur data yang banyak digunakan dalam bidang ilmu komputer, dan memiliki beberapa keunggulan penting jika dibandingkan dengan struktur data alternatif, seperti array. Linked list memiliki beberapa keunggulan penting.

- Salah satu keuntungan linked list adalah ukurannya yang dinamis, yang memungkinkan penambahan atau penghapusan elemen tanpa memperhatikan ukuran tetap daftar. Linked list adalah pilihan yang menguntungkan dalam situasi ketika ada persyaratan untuk memanipulasi sekelompok elemen yang berpotensi bervariasi ukurannya secara dinamis.
- Proses memasukkan atau menghapus elemen dalam linked list ditandai dengan efisiensi dan kecepatannya. Hal ini disebabkan oleh fakta bahwa hanya referensi ke node berikutnya yang perlu dimodifikasi, sehingga menghasilkan kompleksitas waktu yang konstan sebesar O(1).
- Efisiensi Memori: Linked list menunjukkan efisiensi memori yang lebih unggul dibandingkan dengan array karena kemampuannya mengalokasikan memori secara dinamis. Tidak seperti array, yang memiliki ukuran yang telah ditentukan dan dapat mengakibatkan pemborosan memori jika tidak digunakan sepenuhnya, linked list hanya menggunakan jumlah memori yang diperlukan.
- Kesederhanaan Implementasi: Linked list menunjukkan implementasi dan pemahaman yang lebih mudah dibandingkan dengan struktur data alternatif seperti pohon dan grafik.
- Pemanfaatan linked list memungkinkan implementasi beberapa struktur data abstrak, termasuk namun tidak terbatas pada tumpukan, antrian, dan array asosiatif.
- Kemudahan navigasi: Linked list memiliki tingkat kemampuan navigasi yang tinggi, memfasilitasi penjelajahan elemen yang efisien dan pelaksanaan berbagai tindakan dalam daftar.
- Singkatnya, linked list memberikan fleksibilitas dan kemanjuran yang signifikan sebagai struktur data, menunjukkan banyak keunggulan dibandingkan dengan struktur data alternatif. Oleh karena itu, mereka merupakan

pilihan yang sangat baik untuk mengatasi berbagai struktur data dan tantangan algoritmik.

Linked list banyak digunakan dalam bidang ilmu komputer, namun, serupa dengan struktur data lainnya, daftar ini memiliki kelemahan yang melekat. Linked list memberikan beberapa kelemahan penting.

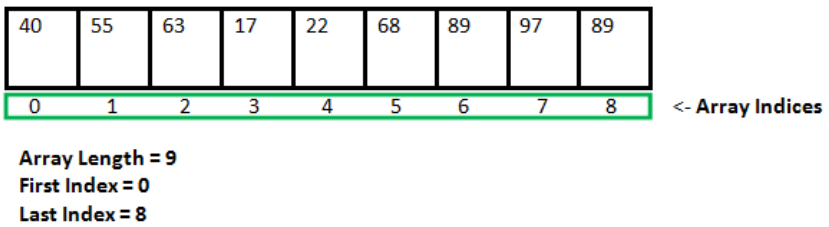
- Waktu akses dalam linked list bisa sangat lama karena keharusan menelusuri daftar untuk menemukan entri tertentu. Proses traversal ini memiliki kompleksitas waktu O(n), dimana n mewakili jumlah entri dalam daftar. Linked list bukanlah pilihan yang cocok dalam skenario yang memerlukan akses elemen yang efisien.
- Pointer memainkan peran penting dalam linked list karena digunakan untuk mereferensikan node berikutnya. Karakteristik linked list ini membuatnya lebih rumit untuk dipahami dan digunakan dibandingkan dengan array. Sifat rumit dari linked list dapat memberikan tantangan dalam hal debugging dan pemeliharaan.
- Linked list menunjukkan overhead yang lebih besar dibandingkan dengan array karena memori tambahan yang diperlukan untuk menyimpan referensi ke node berikutnya untuk setiap node dalam linked list.
- Inefisiensi Cache: Penggunaan linked list dapat mengakibatkan inefisiensi cache karena sifat alokasi memori yang tidak berdekatan. Hal ini menyiratkan bahwa selama penjelajahan linked list, kecil kemungkinannya untuk menemukan data yang diperlukan dalam cache, yang mengakibatkan cache hilang dan akibatnya, kinerja berkurang.
- Diperlukan memori tambahan: Linked list memerlukan penunjuk tambahan untuk setiap node, sehingga meningkatkan konsumsi memori. Masalah ini mungkin timbul ketika berhadapan dengan kumpulan data yang luas,

karena memori tambahan yang diperlukan untuk pointer mungkin terakumulasi dengan cepat.

- Singkatnya, linked list memiliki kemampuan dan kemampuan beradaptasi yang signifikan sebagai struktur data; namun, penting untuk mengetahui kelemahan bawaannya saat mengambil keputusan mengenai penggunaannya. Misalnya, dalam skenario yang membutuhkan waktu akses yang cepat, array mungkin dianggap lebih sesuai, sedangkan dalam situasi yang memerlukan penyisipan atau penghapusan elemen secara sering, linked list dapat dianggap sebagai pilihan yang lebih menguntungkan.

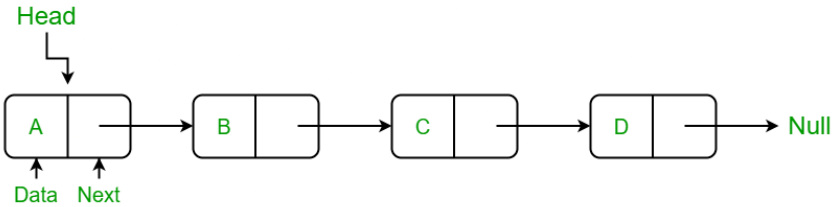
5.7 Linked List vs Array

Array adalah struktur data yang menyimpan elemen di tempat memori yang berdekatan, yang memfasilitasi penghitungan alamat untuk elemen yang disimpan. Karakteristik ini memungkinkan akses lebih cepat ke suatu elemen pada indeks tertentu (Geeksforgeeks.org, 2023).



Gambar 5. 6. Skema penyimpanan data array
(Sumber: Geeksforgeeks.org, 2023)

Struktur data linked list dicirikan oleh pengaturan penyimpanannya yang fleksibel, karena elemen-elemennya biasanya tidak disimpan di wilayah memori yang berurutan. Untuk menjaga ketertiban dan konektivitas anggotanya, linked list memerlukan tag tambahan yang memberikan referensi ke elemen berikutnya.



Gambar 5. 7. Representasi Linked List
(Sumber: Geeksforgeeks.org, 2023)

Berikut ini adalah perbedaan utama antara array dan linked list:

Tabel 8.1. Array vs Linked List

Himpunan	Linked list
Array disimpan di lokasi yang berdekatan	Linked list tidak disimpan di lokasi yang berdekatan
Ukurannya tetap	Ukurannya dinamis
Memori dialokasikan pada waktu kompilasi	Memori dialokasikan pada saat run time
Menggunakan lebih sedikit memori dibandingkan linked list	Menggunakan lebih banyak memori karena menyimpan data dan alamat node berikutnya
Elemen dapat diakses dengan mudah	Pengaksesan elemen memerlukan penelusuran seluruh linked list
Operasi penyisipan dan penghapusan membutuhkan waktu	Operasi penyisipan dan penghapusan lebih cepat

Sumber: (Geeksforgeeks.org, 2023)

5.8 Operasi dasar pada daftar Tertaut

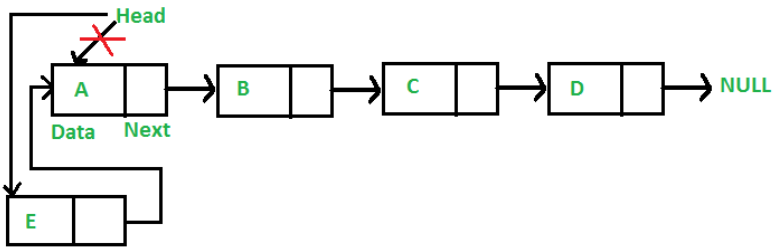
A. Penyisipan dalam Linked List

Tujuannya adalah untuk memasukkan node baru ke dalam Linked List tertentu di tempat tertentu (Geeksforgeeks.org, 2023).

- Di awal daftar
- Setelah simpul tertentu.
- Item terakhir dalam inventaris tertaut.

Prosedur berikut menguraikan langkah-langkah untuk menyisipkan node di depan atau awal linked list. Untuk menambahkan node di awal Linked List, langkah-langkah berikut perlu dilakukan:

- Buat koneksi antara node awal Linked List dan node yang baru dibuat.
- Kepala node awal dalam Linked List harus dihilangkan.
- Tetapkan node baru sebagai kepala linked list.



Gambar 5. 8. Sisipkan Node di Depan atau awal grafik Linked List

(Sumber: Geeksforgeeks.org, 2023)

Berikut ini cuplikan kode yang ditulis dalam JavaScript yang menunjukkan metodologi yang diusulkan.

```
function push(new_data) {
   var new_node = new Node(new_data);
   new_node.next = head;
   head = new_node;
}
```

Dalam analisis ini, kita akan menguji kompleksitas suatu algoritma tertentu.

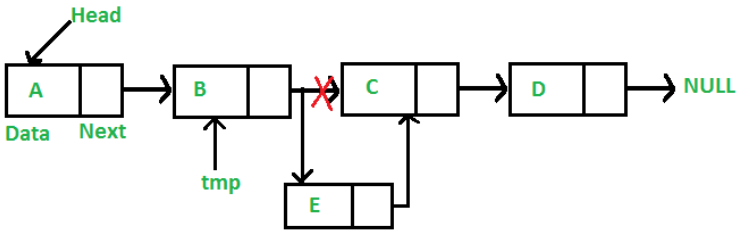
- Kompleksitas waktu dari operasi yang dijelaskan adalah O(1) karena adanya penunjuk ke node kepala, yang memungkinkan lampiran langsung dari node baru dan modifikasi penunjuk. Kompleksitas waktu untuk meletakkan simpul pada posisi kepala adalah O(1) karena jumlah tenaga kerja yang terlibat konstan.
- Kompleksitas ruang tambahan dari algoritma ini adalah O(1).

Instruksi selanjutnya menguraikan prosedur untuk memasukkan sebuah node setelah node tertentu dalam linked list (Parlante, 2002).

Untuk melakukan operasi penyisipan node setelah node tertentu dalam Linked List, langkah-langkah berikut harus diikuti:

- Verifikasi ada atau tidaknya node yang ditentukan.
- Jika tidak ada, hentikan prosedurnya.
- Jika node yang ditentukan ada,
- Elemen tersebut harus dipakai sebagai node baru untuk disisipkan.
- Ubah penunjuk dari node yang ditentukan untuk mereferensikan node yang baru disediakan.

- Tugas yang ada melibatkan menugaskan kembali penunjuk asli berikutnya dari simpul tertentu ke penunjuk berikutnya dari simpul baru.



Gambar 5. 9. Masukkan Node setelah Node yang Diberikan dalam Linked List
(Sumber: Geeksforgeeks.org, 2023)

Berikut ini cuplikan kode yang ditulis dalam JavaScript yang menunjukkan metodologi yang diusulkan.

```
function insertAfter(prev_node, new_data) {
   if (prev_node === null) {
      document.write("The given previous node cannot be null");
      return;
   }
   var new_node = new Node(new_data);
   new_node.next = prev_node.next;
   prev_node.next = new_node;
}
```

Tujuan dari analisis ini adalah untuk menguji kompleksitas suatu algoritma tertentu.

- Kompleksitas waktu dari prosedur ini adalah O(1) karena prev_node disediakan sebagai input, menghilangkan

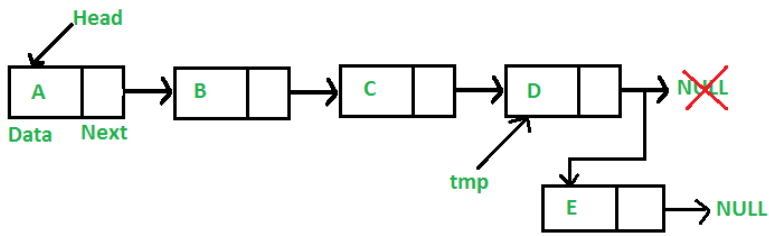
persyaratan untuk menelusuri daftar untuk menemukan prev_node.

- Kompleksitas ruang tambahan adalah O(1) karena menggunakan jumlah ruang yang konstan untuk mengubah penunjuk.

Sesuai dengan tata cara menyisipkan node pada akhir linked list

Untuk menambahkan node ke bagian ekor Linked List, perlu melakukan langkah-langkah berikut:

- Lanjutkan ke simpul terminal dari Linked List.
- Ubah penunjuk simpul terakhir dalam linked list, ganti nilai NULL saat ini dengan alamat simpul baru.
- Untuk menunjukkan akhir dari linked list, setel referensi berikutnya dari simpul baru ke NULL.



Gambar 5. 10. Insert Node di Akhir Linked List
(Sumber: Geeksforgeeks.org, 2023)

Berikut ini cuplikan kode yang ditulis dalam JavaScript yang menunjukkan metodologi yang diusulkan.

```
fungsi tambahkan(data_baru) {

var new_node = new Node(new_data);

if (head === null) {

        head = new_node;

        return;

}

var last = head;

        while (last.next !== null) {

                last = last.next;

        }

        last.next = new_node;

}
```

Dalam analisis ini, kita akan menguji kompleksitas suatu algoritma tertentu (Van Der Spek*dkk.*, 2008).

- Kompleksitas waktu dari algoritma dilambangkan sebagai O(N), dimana N adalah jumlah total node yang ada dalam linked list. Mengingat terdapat perulangan yang melakukan iterasi dari awal hingga akhir, dapat disimpulkan bahwa fungsi tersebut menjalankan pekerjaan dalam kompleksitas waktu O(n).
- Pendekatan ini dapat lebih dioptimalkan untuk mencapai kompleksitas waktu O(1) dengan mempertahankan penunjuk tambahan ke bagian ekor linked list. Dalam hal ruang tambahan, tidak diperlukan informasi tambahan. Kompleksitas waktu dari algoritma ini adalah O(1).

B. Pencarian berulang dan rekursif untuk elemen dalam Linked List (Geeksforgeeks.org, 2023)

Tujuannya adalah untuk menentukan apakah kunci 'X' tertentu ada dalam linked list.

Input: 14->22->15->31->11, X = 14

Output: Yes

Explanation: 14 is present in the linked list.

Input: 9->23->11->32->12->4, X = 13

Output: No

Dalam metodologi ini, vektor digunakan untuk menampung nilai semua node. Selanjutnya, keberadaan kunci tertentu dalam vektor diverifikasi, dan jika ditemukan, nilai kembalian 1 diberikan.

```
class Node {
        constructor(key) {
                this.key = key;
                this.next = null;
        }
}

class LinkedList {
        constructor() {
                this.head = null;
        }
        push(new_key) {
                let new_node = new Node(new_key);
                new_node.next = this.head;
                this.head = new_node;
        }
}
```

```
let llist = new LinkedList();

llist.push(10);
llist.push(30);
llist.push(11);
llist.push(21);
llist.push(14);

let x = 22;
let temp = llist.head;
let found = false;

while (temp) {
        if (temp.key === x) {
                found = true;
                break;
        }
        temp = temp.next;
}

if (found) {
        console.log("YES");
} else {
        console.log("NO");
}
```

Keluaran: Yes

Kompleksitas waktu untuk melintasi linked list adalah O(N).

Ruang tambahan yang diperlukan untuk menyimpan nilai adalah O(N).

Dalam pendekatan ini, kita akan mencari elemen dalam linked list menggunakan metode berulang.

Untuk mengatasi masalah yang ada, harap patuhi serangkaian petunjuk berikut:

- Buat penunjuk simpul, dinotasikan sebagai "saat ini", dan tetapkan nilai simpul kepala.
- Lakukan tindakan berikut selama nilai saat ini tidak sama dengan NULL.
- Jika nilai elemen saat ini (disebut sebagai kunci saat ini->) sama dengan kunci yang dicari, fungsi akan mengembalikan nilai true.
- Jika kondisi yang ditentukan tidak terpenuhi, program akan melanjutkan ke node berikutnya dengan memberikan nilai node berikutnya ke node saat ini (saat ini = saat ini->berikutnya).
- Jika kunci tidak ditemukan, fungsi tersebut akan menghasilkan nilai boolean false.

Berikut implementasi strategi diatas dengan menggunakan kode JavaScript.

```
class Node {
  constructor(d) {
    this.data = d;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
  }

  push(new_data) {
    var new_node = new Node(new_data);
    new_node.next = this.head;
    this.head = new_node;
  }
```

```
  search(x) {
    var current = this.head;
    while (current != null) {
      if (current.data === x) {
        return true; // data found
      }
      current = current.next;
    }
    return false; // data not found
  }
}

var linkedList = new LinkedList();
linkedList.push(10);
linkedList.push(30);
linkedList.push(11);
linkedList.push(21);
linkedList.push(14);

var x = 21;
if (linkedList.search(x)) {
  document.write("Yes");
} else {
  document.write("No");
}
```

Keluaran: Yes

Kompleksitas waktu dari algoritma ini adalah O(N), dimana N adalah jumlah node dalam LinkedList. Kompleksitas ruang tambahan adalah O(1) (Van Der Spek*dkk.*, 2008).

Dalam tugas ini, kami akan menerapkan pendekatan rekursif untuk mencari entri dalam Linked List.

- Untuk mengatasi masalah yang ada, harap ikuti petunjuk berurutan berikut:

- Jika node kepala sama dengan NULL, fungsinya harus mengembalikan false.
- Jika nilai node kepala sama dengan X, fungsi akan mengembalikan nilai true. Jika tidak, fungsi tersebut akan terus mencari secara rekursif di node berikutnya.

Berikut ini adalah implementasi rekursif dari algoritma tersebut di atas, yang ditulis dalam JavaScript.

```
class Node {
  constructor(val) {
    this.data = val;
    this.next = null;
  }
}

let head = null; // Head of list
function push(new_data) {
  let new_node = new Node(new_data);
  new_node.next = head;
  head = new_node;
}

function search(head, x) {
  if (head === null) {
    return false;
  }
  if (head.data === x) {
    return true;
  }
  return search(head.next, x);
}
push(10);
push(30);
push(11);
push(21);
```

```
push(14);
let x = 21;
if (search(head, x)) {
   document.write("Yes");
} else {
   document.write("No");
}
```

Keluaran: Yes

Kompleksitas waktu dari algoritma ini adalah O(N).

Ruang tambahan yang diperlukan untuk algoritma ini adalah O(N), yang mewakili ruang tumpukan yang digunakan oleh panggilan rekursif.

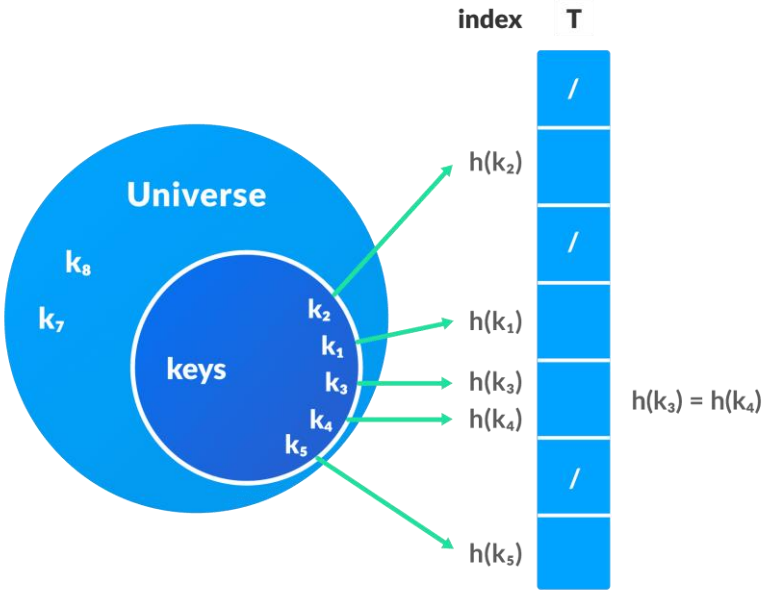
5.9 Tabel Hash

Struktur data tabel hash dirancang untuk menampung elemen dalam pasangan nilai kunci, di mana setiap elemen dikaitkan dengan kunci unik (Programiz.com, 2023).

- Kunci adalah bilangan bulat eksklusif yang digunakan untuk tujuan mengindeks data.
- Nilai mengacu pada informasi atau data yang ditautkan atau diasosiasikan dengan kunci tertentu.

Dalam konteks tabel hash, penghitungan indeks baru dilakukan dengan menerapkan fungsi hashing ke kunci terkait. Selanjutnya, elemen yang sesuai dengan kunci yang diberikan disimpan dalam indeks yang ditentukan. Prosedur di atas biasanya disebut sebagai hashing.

- Misalkan k melambangkan kunci dan h(x) melambangkan fungsi hash.
- Dalam konteks ini, fungsi h(k) digunakan untuk menentukan indeks berbeda untuk tujuan menyimpan elemen yang terkait dengan kunci k.



Gambar 5. 11. Representasi tabel hash
(Sumber: Programiz.com, 2023)

5.9.1 Tabrakan Hash

Dalam kasus di mana fungsi hash menghasilkan indeks yang identik untuk beberapa kunci, konflik akan muncul, sehingga memerlukan keputusan mengenai nilai yang harus dipertahankan pada indeks tertentu. Fenomena yang dimaksud ini biasa dikenal dengan istilah hash collion (Programiz.com, 2023).

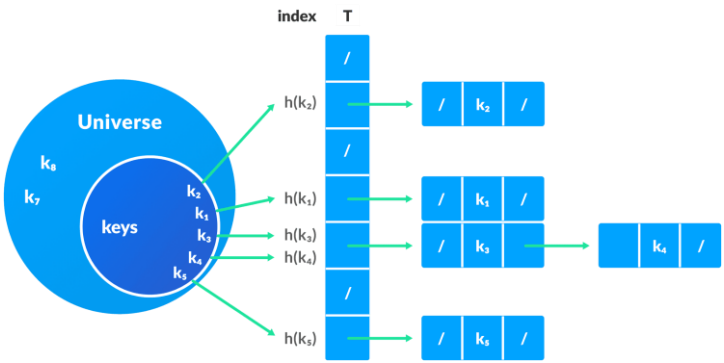
Penyelesaian tabrakan hash dapat dicapai dengan penerapan beberapa strategi.

- Teknik yang digunakan untuk menyelesaikan tabrakan dalam tabel hash melalui penggunaan rangkaian.
- Topik diskusinya adalah pengalamatan terbuka, secara khusus berfokus pada dua teknik: probing linier/kuadrat dan hashing ganda.

- Resolusi tabrakan dengan rantai

Dalam konteks hashing, ketika fungsi hash menghasilkan indeks identik untuk banyak elemen, elemen-elemen ini diakomodasi dalam indeks yang sama dengan menggunakan linked list ganda.

Dalam kasus di mana j mewakili slot yang dapat menampung banyak elemen, hal ini dikaitkan dengan pointer yang mereferensikan entri pertama dalam daftar elemen. Jika tidak ada elemen apa pun, variabel j diberi nilai NIL (Programiz.com, 2023)



Gambar 5. 12. Resolusi Tabrakan menggunakan rantai
(Sumber: Programiz.com, 2023)

Berikut ini adalah representasi pseudocode dari operasi tersebut.

```
chainedHashSearch(T, k)
 return T[hashFunction(k)]

chainedHashInsert(T, x)
 index = hashFunction(x.key)
 x.next = T[index] // Arahkan node baru ke kepala daftar saat ini
 T[index] = x // Perbarui head ke node baru

chainedHashDelete(T, x)
 index = hashFunction(x.key)
 if T[index] is not NIL
  if T[index] is x
   T[index] = T[index].next // Perbarui head ke node berikutnya
  else
   prev = T[index]
   current = T[index].next
   while current is not NIL and current is not x
    prev = current
    current = current.next
   if current is x
    prev.next = current.next // Hapus node dari linked list
```

- Buka Pengalamatan

Berbeda dengan metode chaining, pengalamatan terbuka tidak memungkinkan penyimpanan banyak komponen dalam satu slot. Dalam konteks ini, setiap slot ditempati oleh kunci tunggal atau tetap kosong, dilambangkan dengan NIL.

Berbagai pendekatan yang digunakan dalam pengalamatan terbuka meliputi (Programiz.com, 2023):

a. Linear Probing

Dalam konteks linear probing, tabrakan diselesaikan dengan memeriksa slot berikutnya.

$$h(k, i) = (h'(k) + i) \bmod m$$

Di mana

$i = \{0, 1, ...\}$

h'(k) is a new hash function

Jika terjadi tumbukan pada koordinat (k, 0), langkah selanjutnya adalah memeriksa koordinat (k, 1) pada tabel hash. Dengan cara ini, variabel i mengalami kenaikan linier.

Salah satu keterbatasan yang terkait dengan probing linier adalah terjadinya sekelompok slot berdekatan yang terisi. Selama proses pengenalan elemen baru, perlu untuk melintasi cluster yang lengkap. Hal ini berkontribusi pada peningkatan durasi yang diperlukan untuk menjalankan operasi pada tabel hash.

b. Quadratic Probing

Metode yang digunakan analog dengan probing linier, dengan perbedaan utama adalah penggunaan relasi yang meningkatkan jarak antar slot dengan nilai lebih besar dari satu.

$$h(k, i) = (h'(k) + c1i + c2i2) \bmod m$$

Di mana,

c1 dan c2 adalah konstanta bantu positif,

$i = \{0, 1, ...\}$

c. Double hashing

Jika terjadi tabrakan akibat penerapan fungsi hash h(k), fungsi hash alternatif dihitung untuk menentukan slot berikutnya.

$$h(k, i) = (h1(k) + ih2(k)) \bmod m$$

Meskipun fungsi hash tidak memberikan pencegahan mutlak terhadap tabrakan, fungsi hash memiliki kemampuan untuk secara efektif meminimalkan terjadinya tabrakan tersebut.

5.9.2. Fungsi Hash yang Baik

Dalam studi ini, kita akan mengkaji berbagai pendekatan untuk mengidentifikasi fungsi hash yang efektif.

- Metode Pembagian

  Dalam konteks tabel hash dengan kunci k dan ukuran m, fungsi hash h() dihitung sebagai berikut:

h(k) = k mod m

  Sebagai ilustrasi, mari kita pertimbangkan skenario di mana ukuran tabel hash adalah 10 dan nilai k adalah 112. Dalam kasus ini, fungsi hash h(k) dapat dihitung dengan mengambil modulus k dengan 10, sehingga menghasilkan dalam h(k) = 112 mod 10 = 2. Sangat penting bahwa nilai m tidak sesuai dengan pangkat 2 apa pun. Alasan fenomena ini adalah karena representasi biner pangkat 2 mengikuti pola yang memiliki awalan angka 1 diikuti barisan angka nol, misalnya 10, 100, 1000, dan seterusnya. Saat menentukan nilai k modulo m, nilai yang dihasilkan akan konsisten terdiri dari p-bit orde rendah (Programiz.com, 2023).

if m = 22, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 01 if m = 23, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 001 if m = 24, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 0001 if m = 2p, then h(k) = p lower bits of m

- Metode Perkalian

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Dalam istilah matematika, ungkapan "kA mod 1" mewakili bagian pecahan dari hasil kali k dan A. Notasi ⌊ ⌋ menunjukkan nilai dasar suatu besaran tertentu. Penting untuk dicatat bahwa A mewakili bilangan konstan. Variabel A dibatasi di dalam interval [0, 1]. Namun, Knuth menyarankan bahwa pilihan optimal kira-kira sama dengan $(\sqrt{5}-1)/2$.

Hashing Universal

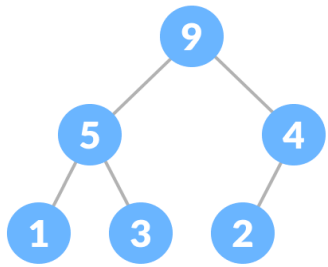
Dalam konteks hashing Universal, pemilihan fungsi hash dilakukan secara acak dan tidak bergantung pada kunci yang di-hash. Program Java selanjutnya berfungsi sebagai contoh fungsi operasional HashTable.

```
import java.util.HashMap;

class HashTable {
   public static void main(String args[]) {
      HashMap<Integer, Integer> ht = new HashMap<Integer,
Integer>();
      ht.put(123, 432);
      ht.put(12, 2345);
      ht.put(15, 5643);
      ht.put(3, 321);
      ht.remove(12);
      System.out.println(ht);
   }
}
```

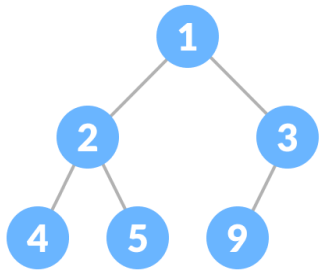
5.10 Heap Data Structure

Struktur data heap dicirikan sebagai pohon biner lengkap yang mematuhi properti heap, yang menyatakan bahwa setiap node di pohon memenuhi kondisi tertentu (Programiz.com, 2023).

- Nilai dari node induk secara konsisten lebih tinggi dibandingkan dengan node anaknya, dan kunci dari node akar adalah yang terbesar di antara semua node lainnya. Karakteristik ini juga disebut sebagai properti heap maksimum.
- Ukuran node induk secara konsisten lebih kecil dibandingkan node anak, dan nilai kunci dari node akar adalah yang terkecil di antara semua node lainnya. Karakteristik yang disebutkan di atas biasanya disebut sebagai "properti min heap".



Gambar 5. 13. Max-heap
(Sumber: Programiz.com, 2023)



Gambar 5. 14. Min-heap
(Sumber: Programiz.com, 2023)

Gambar 5.13 dan 5.14 menggambarkan bentuk struktur data yang biasa disebut dengan tumpukan biner. Bagian berikut menguraikan beberapa operasi penting yang biasanya dijalankan di heap, beserta algoritmanya masing-masing (Parlante, 2002).

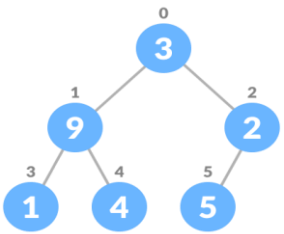
A. Menumpuk

Heapify adalah operasi prosedural yang melibatkan transformasi pohon biner menjadi struktur data heap. Tujuan dari alat ini adalah untuk menghasilkan Min-Heap atau Max-Heap.

1. Pertimbangkan array masukan

3	9	2	1	4	5
0	1	2	3	4	5

2. Buatlah pohon biner penuh dari array



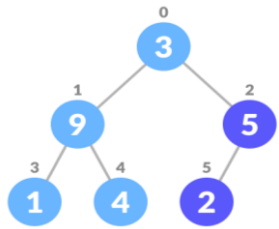
Gambar 5. 15. Pohon biner lengkap
(Sumber: Programiz.com, 2023)

3. Dimulai dengan indeks awal dari simpul bukan daun, yang dilambangkan dengan n/2-1.
4. Perbesar elemen sekarang i.
5. Indeks anak kiri diberikan oleh 2i + 1, sedangkan indeks anak kanan diberikan oleh 2i + 2.

   Setel leftChildIndex ke yang terbesar jika leftChild lebih besar dari currentElement (elemen pada indeks ke-i).

   Setel rightChildIndex ke terbesar jika rightChild lebih besar dari elemen terbesar.

6. Ganti elemen terbesar dengan elemen saat ini



Gambar 5. 16. Swap node atau element
(Sumber: Programiz.com, 2023)

7. Proses penumpukan subpohon harus diulangi secara iteratif, mengikuti langkah 3-7, hingga semua subpohon telah ditumpuk dengan benar.

```
def heapify(array, size, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < size and array[left_child] > array[largest]:
        largest = left_child

    if right_child < size and array[right_child] > array[largest]:
        largest = right_child

    if largest != i:
        array[i], array[largest] = array[largest], array[i]
        heapify(array, size, largest)
```

Kode ini untuk membuat Max-Heap:

```
def build_max_heap(array, size):
    # Start from the last non-leaf node and heapify all nodes in reverse order
    for i in range(size // 2 - 1, -1, -1):
        max_heapify(array, size, i)
```

Fungsi build_max_heap dirancang untuk menerima array dan ukurannya sebagai parameter, dan tujuannya adalah untuk menghasilkan max-heap dengan memanfaatkan fungsi max_heapify. Perulangan dimulai dari node non-leaf terakhir dan memanggil fungsi max_heapify pada setiap node dalam urutan terbalik, menjamin pelestarian karakteristik max-heap.

Dalam Min-Heap, node leftChild dan rightChild harus memiliki nilai yang lebih besar dari node induk untuk semua node di heap.

B. Tambahkan elemen ke heap.

Contoh algoritma kode untuk dimasukkan ke dalam Max Heap disediakan di bawah ini.

```
def insert_into_heap(array, new_node):
    # Append the new node to the end of the array
    array.append(new_node)
    index = len(array) - 1

    # Heapify the array to maintain the max-heap property
    while index > 0:
        parent_index = (index - 1) // 2
        if array[index] > array[parent_index]:
            # Swap the current node with its parent if it violates the
max-heap property
            array[index], array[parent_index] = array[parent_index], array[index]
            index = parent_index
        else:
            break
```

Dengan menggunakan contoh kode sebelumnya, maka akan muncul hasilnya sebagai berikut (Programiz.com, 2023):

Contoh penggunaan:

- array = [10, 8, 7, 6, 3, 5]

- new_node = 9
- insert_into_heap(array, new_node)
- Output: [10, 9, 7, 6, 8, 5, 3]

Fungsi insert_into_heap menerima array dan node baru sebagai parameter dalam kode yang disediakan. Prosesnya melibatkan penambahan node baru ke akhir array dan setelah itu melakukan pertukaran berulang antara node baru dan induknya hingga properti max-heap dibangun kembali. Prosedur yang disebutkan di atas menjamin pelestarian validitas heap biner sebagai max-heap setelah penyisipan (Parlante, 2002).

C. Hapus Item dari Heap

Contoh algoritma pseudocode untuk penghapusan Max Heap disediakan di bawah.

```
def delete_from_heap(array, node_to_delete):
  # Find the index of the node to be deleted
  index = array.index(node_to_delete)

  # Swap the node to be deleted with the last leaf node
  array[index], array[-1] = array[-1], array[index]

  # Remove the last leaf node (the node to be deleted)
  deleted_node = array.pop()

  # Heapify the array to maintain the max-heap property
  size = len(array)
  while True:
    largest = index
    left_child = 2 * index + 1
    right_child = 2 * index + 2

    if left_child < size and array[left_child] > array[largest]:
```

```
        largest = left_child

    if right_child < size and array[right_child] > array[largest]:
        largest = right_child

    if largest != index:
        # Swap the current node with the largest child if it violates the max-heap property
        array[index], array[largest] = array[largest], array[index]
        index = largest
    else:
        break

  return deleted_node
```

Dengan menggunakan contoh kode yang disajikan sebelumnya, hasilnya adalah sebagai berikut:

Contoh penggunaan:

- array = [10, 8, 7, 6, 3, 5]
- node_to_delete = 8
- deleted_node = delete_from_heap(array, node_to_delete)
- Output: [10, 6, 7, 5, 3], deleted_node = 8

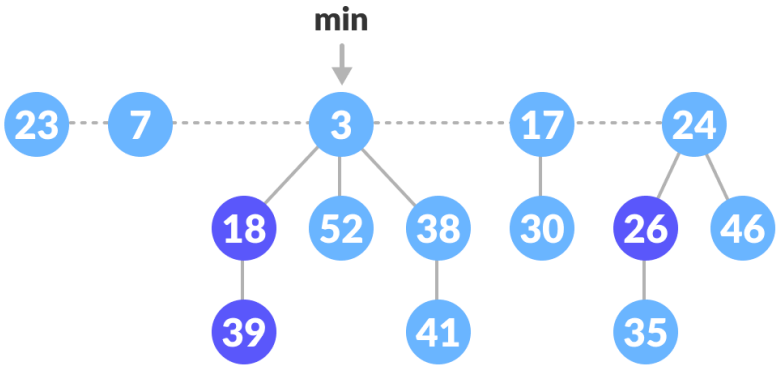
Fungsi delete_from_heap menerima array dan node untuk dihapus sebagai parameternya dalam kode yang disediakan. Algoritme pertama-tama menentukan indeks node yang perlu dihapus. Ia kemudian menukar node ini dengan node daun terakhir di heap. Dengan menghilangkan simpul daun terakhir, maka simpul yang diinginkan berhasil dimusnahkan. Terakhir, algoritme melakukan heapifikasi pada array untuk memastikan bahwa properti max-heap dipertahankan. Fungsi ini mengembalikan node yang dihapus sebagai outputnya.

5.11 Fibonacci Heap

Fibonacci Heap adalah struktur data yang terdiri dari sekumpulan pohon yang mematuhi kriteria tumpukan minimum atau tumpukan maksimum. Konsep properti min heap dan max heap telah dibahas sebelumnya dalam artikel tentang Struktur Data Heap. Sifat-sifat yang disebutkan di atas mencakup ciri-ciri pembeda pohon-pohon yang ada dalam tumpukan Fibonacci.

Dalam konteks tumpukan Fibonacci, sebuah node mungkin saja memiliki jumlah anak melebihi dua atau alternatifnya, tidak memiliki anak sama sekali. Selain itu, ia menunjukkan efisiensi yang unggul dalam hal operasi heap dibandingkan dengan heap binomial dan biner.

Istilah "Fibonacci heap" berasal dari konstruksi pohonnya, di mana pohon berorde n berisi minimal simpul Fn+2, dengan Fn+2 mewakili bilangan Fibonacci ke-(n + 2).



Gambar 5. 17. Fibonacci Heap
(Sumber: Programiz.com, 2023)

5.11.1 Properti Heap of Fibonacci

Bagian selanjutnya menyajikan analisis properti yang ditunjukkan oleh Fibonacci Heap. Beberapa aspek penting dari tumpukan Fibonacci meliputi:

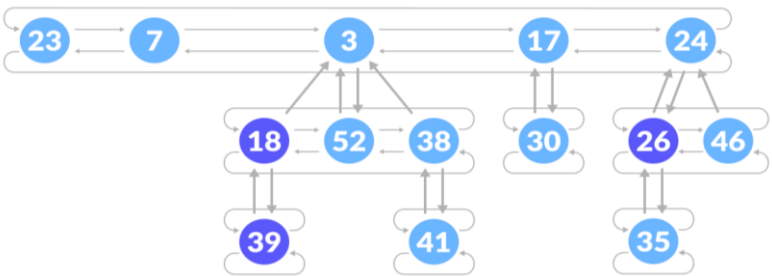
1. Strukturnya terdiri dari kumpulan pohon-pohon yang diurutkan dalam tumpukan minimum. Misalnya, dapat diamati bahwa ukuran induk selalu lebih kecil dibandingkan dengan ukuran keturunannya.
2. Referensi disimpan pada node yang berisi elemen minimum.
3. Strukturnya terdiri dari kumpulan node yang ditunjuk. Operasi penurunan kunci.
4. Pohon-pohon yang ada dalam tumpukan Fibonacci dicirikan oleh susunannya yang tidak teratur, tetapi setiap pohon berakar.

5.11.2 Representasi Node Tumpukan Fibonacci Berbasis Memori

Keterhubungan akar-akar pohon memudahkan akses yang lebih cepat. Node anak dari node induk saling berhubungan satu sama lain menggunakan linked list ganda melingkar, seperti yang digambarkan dalam diagram yang disediakan.

Terdapat dua manfaat utama yang terkait dengan pemanfaatan linked list ganda melingkar.

1. Kompleksitas temporal untuk menghapus sebuah node dari pohon adalah O(1).
2. Kompleksitas waktu untuk menggabungkan dua daftar seperti ini adalah O(1).



Gambar 5. 18. Struktur Tumpukan Fibonacci
(Sumber: Programiz.com, 2023)

5.11.3. Operasi pada Heap of Fibonacci

A. Insertion

```
FibonacciHeapInsertion(heap, node):
  if heap.min is None:
    // Jika heap kosong, setel node baru sebagai minimum
    heap.min = node
    node.left = node
    node.right = node
  else:
    // Sisipkan node baru di sebelah kanan node minimum
    node.left = heap.min
    node.right = heap.min.right
    heap.min.right.left = node
    heap.min.right = node
    // Perbarui node minimum jika perlu
    if node.key < heap.min.key:
      heap.min = node
  // Tingkatkan jumlah node di heap
  heap.size = heap.size + 1
```

Pseudocode yang diberikan menggunakan variabel "heap" untuk mewakili Fibonacci Heap, sedangkan variabel "node" digunakan untuk mewakili node yang akan ditempatkan. Operasi ini memverifikasi kekosongan heap. Jika hal ini terjadi, node yang baru ditambahkan mengambil peran sebagai node minimal dan membuat linked list ganda melingkar dengan dirinya sendiri. Jika

heap tidak kosong, node baru disisipkan di sebelah kanan node minimal dalam linked list melingkar. Node minimum mengalami pembaruan ketika nilai kunci dari node yang baru dimasukkan lebih kecil dari nilai kunci dari node minimum saat ini. Variabel "ukuran" digunakan untuk mempertahankan jumlah node di heap dan bertambah setelah operasi penyisipan.

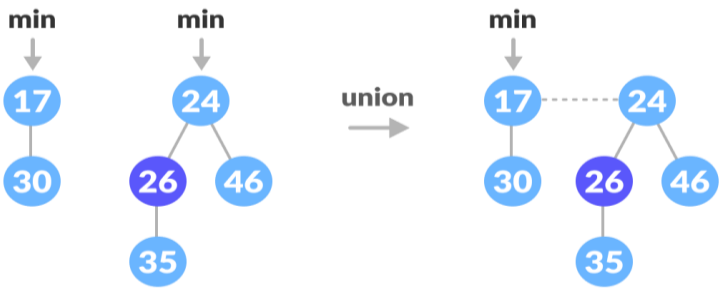
Proses memasukkan node ke heap yang sudah ada sebelumnya dapat dipecah menjadi beberapa bagian berikut.

1. Hasilkan node baru untuk mengakomodasi elemen tertentu.
2. Verifikasi apakah heap tidak memiliki elemen.
3. Jika heap tidak memiliki elemen, node baru harus ditetapkan sebagai node root dan kemudian diberi label sebagai minimum.
4. Alternatifnya, node dapat dimasukkan ke dalam daftar root dan nilai minimum dapat diperbarui.

5.11.4. Union

Operasi gabungan dua tumpukan Fibonacci melibatkan langkah-langkah berikut.

- Gabungkan akar kedua tumpukan dengan menggabungkan.
- Kunci minimal dari daftar root baru dipilih untuk memperbarui nilai min.



Gambar 5. 19. Persatuan dua tumpukan
(Sumber: Programiz.com, 2023)

5.11.5. Extract Min

Operasi yang dimaksud mempunyai arti penting dalam konteks tumpukan Fibonacci. Selama prosedur ini, node dengan nilai terendah diekstraksi dari heap dan pohon kemudian direstrukturisasi. Prosedur selanjutnya dijalankan:

1. Hapus node minimum.
2. Tetapkan penunjuk minimum ke akar berikutnya dalam daftar akar.
3. Buatlah array dengan panjang yang setara dengan derajat maksimum pohon yang ada di heap sebelum operasi penghapusan apa pun.
4. Lanjutkan menjalankan langkah-langkah di atas, khususnya langkah 5 hingga 7, secara berulang hingga kondisi memiliki beberapa akar dengan derajat yang sama tidak lagi terpenuhi.
5. Tujuannya adalah untuk membangun korelasi antara derajat akar saat ini (disebut sebagai min-pointer) dan derajat dalam array.
6. Tugas yang ada melibatkan pembuatan pemetaan antara derajat akar berikutnya dan derajat yang sesuai dalam larik.
7. Dalam kasus ketika terdapat banyak pemetaan pada tingkat tertentu, maka perlu menggunakan operasi gabungan untuk mengkonsolidasikan pemetaan ini sambil memastikan pemeliharaan properti min-heap, sehingga nilai minimum tetap berada di akar.

5.12 Kesimpulan

Linked list adalah struktur data serbaguna yang menyediakan alokasi memori dinamis serta operasi penyisipan dan penghapusan yang efisien. Memahami dasar-dasar linked list sangat penting bagi setiap programmer atau penggemar ilmu komputer. Dengan pengetahuan ini, Anda dapat menerapkan linked list untuk memecahkan berbagai masalah dan memperluas pemahaman Anda tentang struktur data dan algoritma.

DAFTAR PUSTAKA

Geeksforgeeks.org (2023) *Linked List Data Structure*, *geeksforgeeks.org*. Available at: https://www.geeksforgeeks.org/data-structures/linked-list/ (Accessed: 6 October 2023).

Parlante, N. (2002) 'Linked List Basics', *Review Literature And Arts Of The Americas*.

Programiz.com (2023) *Linked list Data Structure*, *programiz.com*. Available at: https://www.programiz.com/dsa/linked-list (Accessed: 6 October 2023).

Ravikiran, A. S. (2023) *Linked List in A Data Structure: All You Need to Know*, *simplilearn.com*. Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure (Accessed: 6 October 2023).

Van Der Spek, H. L. A. *et al.* (2008) 'A compile/run-time environment for the automatic transformation of linked list data structures', *International Journal of Parallel Programming*, 36(6), pp. 592–623. doi: 10.1007/s10766-008-0085-2.

BAB 6
PENGURUTAN (SORTING)

Oleh Samuel Ady Sanjaya

6.1 Pendahuluan

Struktur data merupakan landasan fundamental dalam pengembangan perangkat lunak, memainkan peran kunci dalam menyimpan dan mengelola data secara efisien. Salah satu aspek yang memperkuat integritas dan kinerja struktur data adalah kemampuannya dalam penyortiran data. Penyortiran, atau sorting, merupakan proses pengaturan elemen-elemen data dalam suatu urutan tertentu, dan algoritma sorting menjadi tulang punggung yang mendukung stabilitas dan kecepatan akses data dalam berbagai aplikasi (Pal and Halder, 2018). Pentingnya algoritma sorting dalam struktur data tidak dapat diabaikan.

Algoritma sorting memungkinkan data disusun sedemikian rupa sehingga mempermudah pencarian, penghapusan, dan penambahan elemen-elemen data. Kecepatan akses data yang dihasilkan oleh algoritma sorting yang efisien sangat penting dalam mengoptimalkan kinerja aplikasi, terutama pada dataset yang besar. Dalam konteks struktur data, penggunaan algoritma sorting juga dapat meningkatkan efisiensi penggunaan memori dan meminimalkan kompleksitas waktu operasi tertentu.

Selain itu, algoritma sorting memiliki dampak langsung pada fungsi-fungsi krusial dalam sistem basis data, seperti pencarian biner dan penggabungan data. Penggunaan struktur data yang terorganisir dengan baik melalui algoritma sorting akan mempercepat proses pencarian dan manipulasi data, yang

esensial dalam berbagai aplikasi, mulai dari basis data hingga pemrosesan citra. Oleh karena itu, pemahaman yang mendalam tentang algoritma sorting dan penerapannya dalam struktur data merupakan aspek kritis dalam pengembangan perangkat lunak yang efisien dan responsif. Dalam konteks ini, penelitian dan pemahaman lebih lanjut tentang algoritma sorting tidak hanya relevan untuk para pengembang perangkat lunak, tetapi juga menjadi kunci untuk menciptakan solusi teknologi informasi yang efektif dan handal.

6.2 Pengantar Sorting

Sorting, atau penyortiran, merupakan proses kritis dalam dunia pemrograman dan struktur data yang membawa dampak signifikan terhadap kinerja aplikasi dan efisiensi pengelolaan informasi. Konsep dasar sorting melibatkan pengaturan suatu rangkaian data secara teratur, memungkinkan akses yang lebih cepat dan efisien terhadap informasi yang disimpan. Algoritma sorting menjadi tulang punggung dari operasi ini, menentukan bagaimana elemen-elemen data diatur sehingga memaksimalkan efisiensi pencarian, penghapusan, dan penambahan data (Khan *et al.*, 2023). Melalui pemahaman algoritma sorting, kita dapat mendalami prinsip-prinsip yang mendasari kecepatan dan kompleksitas operasi penyortiran data. Pentingnya sorting dalam konteks struktur data dan pengembangan perangkat lunak membuatnya menjadi area yang mendalam untuk dijelajahi, dengan tujuan tidak hanya memahami cara data diatur, tetapi juga bagaimana efektivitas dan efisiensi algoritma sorting dapat ditingkatkan untuk memenuhi tuntutan aplikasi modern. Dalam konteks ini, pengantar ini akan menggali aspek-aspek kunci dari sorting, membuka pintu bagi eksplorasi lebih lanjut terhadap berbagai algoritma dan penerapannya dalam berbagai domain teknologi informasi.

6.2.1 Definisi Sorting

Sorting, atau penyortiran, merujuk pada proses pengaturan dan pengurutan elemen-elemen data dalam suatu urutan tertentu, baik itu secara meningkat (ascending) maupun menurun (descending), sesuai dengan suatu kriteria tertentu (Singh, 2021). Tujuan utama dari algoritma sorting adalah menyusun data sedemikian rupa sehingga mempermudah akses, pencarian, dan analisis data. Dengan kata lain, sorting memungkinkan penyusunan data menjadi suatu susunan yang terstruktur, memfasilitasi operasi-operasi berbasis urutan, seperti pencarian biner atau pencarian sekuen, untuk dilakukan dengan lebih efisien. Proses ini memiliki relevansi yang krusial dalam pengembangan perangkat lunak, basis data, dan berbagai aplikasi lainnya, memastikan bahwa data tersusun dengan cara yang memudahkan pemrosesan dan pengelolaan informasi dalam konteks penggunaan yang spesifik. Sorting menjadi dasar bagi berbagai algoritma dan struktur data, menghadirkan landasan yang diperlukan untuk operasi-operasi data yang optimal dan efisien dalam berbagai skenario pengembangan perangkat lunak.

6.2.2 Tujuan dan Manfaat Sorting dalam Struktur Data

Tujuan utama dari penggunaan algoritma sorting dalam struktur data adalah mengorganisir dan menyusun elemen-elemen data menjadi urutan tertentu, sehingga memungkinkan akses dan manipulasi data dengan lebih efisien. Salah satu tujuan pokoknya adalah meningkatkan kecepatan pencarian data. Dengan memiliki data yang tersusun, algoritma pencarian, seperti pencarian biner, dapat diaplikasikan dengan efisiensi yang tinggi karena memanfaatkan sifat data yang terurut. Ini mengurangi waktu yang diperlukan untuk menemukan elemen tertentu dalam kumpulan data besar.

Selain itu, sorting juga memberikan manfaat dalam proses penyisipan (insertion) dan penghapusan (deletion) data. Dalam struktur data seperti array atau linked list, penyisipan atau

penghapusan data dapat dilakukan lebih efisien ketika data sudah dalam keadaan terurut. Sebagai contoh, pada linked list terurut, operasi penyisipan dapat dilakukan dengan mempertahankan urutan tanpa memerlukan penyortiran ulang keseluruhan struktur data. Manfaat lainnya adalah kemampuan untuk memfasilitasi proses analisis dan pengolahan data secara umum. Data yang terurut dapat memberikan pemahaman yang lebih baik tentang struktur dan distribusi informasi, memudahkan identifikasi pola atau tren, dan mendukung pengambilan keputusan yang lebih cepat. Dalam aplikasi seperti basis data, sorting juga dapat meningkatkan efisiensi operasi seperti penggabungan data dari beberapa tabel.

Dengan demikian, tujuan dan manfaat sorting dalam struktur data tidak hanya terbatas pada penataan elemen-elemen data, tetapi juga melibatkan peningkatan kinerja berbagai operasi data, pengurangan kompleksitas pencarian, dan mempermudah pemrosesan informasi untuk keperluan analisis dan pengambilan keputusan.

6.2.3 Peran Sorting dalam Optimasi Kinerja Aplikasi

Sorting memiliki peran yang sangat krusial dalam optimasi kinerja aplikasi dengan membawa dampak positif pada berbagai aspek, termasuk efisiensi waktu eksekusi, penggunaan sumber daya, dan responsivitas keseluruhan aplikasi. Salah satu peran utamanya adalah dalam meningkatkan kecepatan akses data. Dengan mengurutkan data, operasi pencarian seperti pencarian biner dapat diimplementasikan dengan efisiensi tinggi, mengurangi waktu yang diperlukan untuk menemukan informasi yang diinginkan dalam kumpulan data yang besar. Hal ini memberikan kontribusi signifikan terhadap responsivitas aplikasi, terutama pada aplikasi yang melibatkan manipulasi dan penelusuran data secara intensif.

Pentingnya sorting juga terlihat dalam konteks pengoptimalan operasi penyisipan dan penghapusan data. Pada struktur data terurut, seperti pohon pencarian biner atau linked list terurut, penyisipan dan penghapusan dapat dilakukan dengan lebih efisien karena tidak memerlukan pengurutan ulang data secara menyeluruh. Ini mengurangi beban komputasional dan mempercepat waktu eksekusi operasi-operasi tersebut.

Selain itu, sorting memiliki dampak positif terhadap pengelolaan memori. Algoritma sorting yang efisien dapat mengoptimalkan penggunaan memori, terutama dalam konteks penyimpanan data terurut. Pada aplikasi dengan keterbatasan sumber daya, seperti perangkat mobile atau sistem terdistribusi, optimasi penggunaan memori dapat menghasilkan peningkatan kinerja yang signifikan.

Terakhir, peran sorting dalam optimasi kinerja aplikasi juga tercermin dalam kemampuannya mendukung operasi pengolahan data dan analisis. Data yang terurut mempermudah implementasi algoritma analisis data, seperti statistika atau pemodelan prediktif, dengan mengurangi kompleksitas operasi tersebut.

Dengan demikian, sorting bukan hanya sekadar langkah pra-pemrosesan data, tetapi juga sebuah strategi yang strategis dalam upaya meningkatkan efisiensi dan responsivitas keseluruhan aplikasi, menjadikannya unsur yang esensial dalam pengembangan perangkat lunak yang efektif dan optimal.

6.2.4 Komplesitas Waktu dan Ruang dalam Sorting

Komplesitas waktu dan ruang dalam sorting adalah dua aspek kritis yang harus dipertimbangkan dalam pemilihan dan implementasi algoritma sorting. Komplesitas waktu mencerminkan sejauh mana algoritma mampu menyelesaikan tugas sorting dalam konteks waktu tertentu, sementara

komplesitas ruang mencerminkan seberapa banyak sumber daya memori yang dibutuhkan oleh algoritma.

Dalam hal komplesitas waktu, algoritma sorting diukur berdasarkan waktu eksekusi relatif terhadap ukuran dataset yang diberikan. Algoritma dengan kompleksitas waktu O(n log n), seperti Merge Sort dan Quick Sort, sering dianggap efisien karena dapat menangani besar dataset dengan cepat dan memberikan kinerja yang baik pada berbagai kondisi input. Namun, algoritma dengan kompleksitas waktu O(n^2), seperti Bubble Sort atau Insertion Sort, cenderung kurang efisien dan dapat menjadi lambat ketika dihadapkan pada dataset yang besar.

Di sisi lain, komplesitas ruang mengacu pada jumlah memori yang dibutuhkan oleh algoritma sorting untuk menyimpan dan memanipulasi data. Algoritma in-place, seperti beberapa variasi Quick Sort, memiliki komplesitas ruang yang rendah karena mereka tidak memerlukan alokasi memori tambahan yang berskala dengan ukuran dataset. Sebaliknya, algoritma out-of-place, seperti Merge Sort, memerlukan alokasi memori tambahan untuk menyimpan hasil sorting, sehingga dapat memiliki komplesitas ruang yang lebih tinggi.

Pemilihan algoritma sorting harus memperhitungkan keseimbangan antara komplesitas waktu dan ruang tergantung pada kebutuhan spesifik aplikasi. Aplikasi yang membutuhkan kinerja tinggi pada dataset besar mungkin memilih algoritma dengan kompleksitas waktu yang rendah, meskipun mungkin memerlukan lebih banyak memori. Sebaliknya, aplikasi dengan batasan memori mungkin lebih memilih algoritma in-place, bahkan jika kompleksitas waktu sedikit lebih tinggi.

Dengan memahami komplesitas waktu dan ruang, pengembang dapat membuat keputusan yang terinformasi tentang pilihan algoritma sorting yang paling sesuai dengan karakteristik dan kebutuhan spesifik dari dataset dan aplikasi yang dihadapi.

6.2.5 Jenis-jenis Sorting Berdasarkan Pendekatan dan Metode

Dalam dunia algoritma dan struktur data, pemahaman terhadap berbagai jenis algoritma sorting menjadi esensial untuk mengoptimalkan kinerja aplikasi dan pengelolaan data. Pada bagian ini, kita akan menjelajahi beberapa algoritma sorting yang mencakup berbagai karakteristik dan kompleksitas. Algoritma Sorting Sederhana, seperti Bubble Sort, Insertion Sort, dan Selection Sort, memperkenalkan konsep dasar penyortiran dan sering digunakan pada dataset kecil. Sementara itu, Algoritma Sorting Efisien, seperti Merge Sort, Quick Sort, dan Heap Sort, memberikan solusi yang efektif untuk menyortir dataset besar dengan kompleksitas waktu yang lebih optimal. Adapun Algoritma Sorting Khusus, seperti Radix Sort dan Counting Sort, menghadirkan pendekatan khusus untuk kasus tertentu, seperti penyortiran data non-numerik. Dengan memahami karakteristik masing-masing algoritma, kita dapat membuat keputusan informasi tentang pemilihan yang sesuai dengan tuntutan aplikasi dan sifat dataset yang dihadapi.

6.3 Algoritma Sorting Sederhana

Algoritma Sorting Sederhana, seperti Bubble Sort, Insertion Sort, dan Selection Sort, memiliki kompleksitas waktu yang lebih rendah, namun mungkin kurang efisien untuk dataset yang besar. Bubble Sort membandingkan dan menukar elemen secara berurutan, Insertion Sort menyisipkan elemen satu per satu pada posisi yang benar, sedangkan Selection Sort memilih elemen terkecil dan menukarkannya dengan elemen pertama.

6.3.1 Bubble Sort

Bubble Sort adalah salah satu algoritma sorting sederhana yang bekerja dengan cara membandingkan dan menukar elemen-elemen data secara berurutan hingga data tersusun dengan

benar. Prosesnya dimulai dari elemen pertama hingga elemen terakhir, di mana dua elemen bersebelahan dibandingkan, dan jika elemen yang pertama lebih besar dari elemen yang kedua, keduanya ditukar posisinya. Proses ini terus berulang sampai tidak ada lagi perubahan yang diperlukan, menghasilkan data yang terurut. Meskipun Bubble Sort sederhana untuk dipahami, ia memiliki kompleksitas waktu yang tinggi, yaitu O(n^2), membuatnya kurang efisien untuk dataset besar. Bubble Sort sering digunakan dalam konteks pendidikan atau untuk dataset yang sudah hampir terurut, tetapi kurang umum digunakan dalam aplikasi produksi di mana efisiensi kinerja menjadi pertimbangan utama.

```
Procedure BubbleSort(arr: array of integer)
    n <- length(arr)
    for i from 0 to n-1 do
        swapped <- false
        for j from 0 to n-i-1 do
            if arr[j] > arr[j+1] then
                // Swap arr[j] and arr[j+1]
                temp <- arr[j]
                arr[j] <- arr[j+1]
                arr[j+1] <- temp
                swapped <- true
            end if
        end for
        // Jika tidak ada pertukaran dalam iterasi ini, array sudah terurut
        if not swapped then
            break
        end if
    end for
End Procedure

```

Gambar 6. 1. Pseudocode Algoritma Bubble Sort

Berikut adalah penjelasan langkah-langkah algoritma Bubble Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.1:

1. **Inisialisasi Variabel:**
   a. **n**: Panjang (jumlah elemen) array.
   b. **swapped**: Variabel boolean untuk menandai apakah ada pertukaran elemen dalam iterasi tertentu.
2. **Iterasi Outer (i):**
   Mengiterasi melalui setiap elemen array.
3. **Iterasi Inner (j):**
   a. Mengiterasi melalui elemen-elemen yang belum terurut dari awal hingga **(n-i-1)**.
   b. Membandingkan elemen ke-**j** dengan elemen ke**-(j+1).**
   c. Jika elemen ke-j lebih besar dari elemen ke**-(j+1)**, maka pertukaran dilakukan untuk menyusun elemen secara ascending.
4. **Pengecekan Pertukaran:**
   Setelah iterasi inner selesai, jika ada pertukaran elemen, swapped diubah menjadi true.
5. **Pengecekan Terurut:**
   Jika tidak ada pertukaran dalam iterasi inner, array sudah terurut, dan iterasi outer dihentikan.

Algoritma Bubble Sort bekerja dengan mengulang iterasi melalui array, membandingkan dan menukarkan elemen-elemen yang tidak terurut sampai seluruh array terurut. Pseudocode di atas menggambarkan implementasi Bubble Sort yang sederhana dengan kompleksitas waktu O(n^2).

6.3.2 Insertion Sort

Insertion Sort adalah algoritma sorting yang bekerja dengan cara menyisipkan elemen data satu per satu pada posisi yang tepat dalam urutan terurut. Prosesnya dimulai dengan memperlakukan elemen pertama sebagai himpunan terurut dan menyisipkan elemen berikutnya secara berurutan ke dalam himpunan tersebut. Ketika sebuah elemen baru disisipkan, algoritma membandingkannya dengan elemen-elemen sebelumnya dan menyusunnya pada posisi yang benar. Proses ini

berlanjut hingga semua elemen dalam dataset disusun dengan benar. Insertion Sort memiliki kompleksitas waktu O(n^2), membuatnya kurang efisien untuk dataset besar, namun, keuntungannya terletak pada implementasinya yang sederhana dan efisien untuk dataset yang hampir terurut atau memiliki ukuran kecil. Insertion Sort sering digunakan dalam praktik pengajaran dan dalam skenario di mana data sedikit atau sudah dekat dengan keadaan terurut.

```
Procedure InsertionSort(arr: array of integer)
    n <- length(arr)
    for i from 1 to n-1 do
        key <- arr[i]
        j <- i - 1
        while j >= 0 and arr[j] > key do
            // Geser elemen yang lebih besar ke posisi berikutnya
            arr[j + 1] <- arr[j]
            j <- j - 1
        end while
        // Tempatkan key pada posisi yang tepat dalam array yang sudah terurut
        arr[j + 1] <- key
    end for
End Procedure
```

Gambar 6. 2. Pseudocode Algoritma Insertion Sort

Berikut adalah penjelasan langkah-langkah algoritma Insertion Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.2:

1. **Inisialisasi Variabel:**
   a. **n**: Panjang (jumlah elemen) array.
   b. **key**: Variabel sementara untuk menyimpan nilai yang akan diurutkan.
   c. **i** dan **j**: Variabel loop untuk iterasi.
2. **Iterasi Outer (i):**
   Mengiterasi mulai dari elemen kedua hingga elemen terakhir array.

3. **Penyimpanan Nilai:**
   Menyimpan nilai dari elemen ke-i ke dalam key.
4. **Iterasi Inner (j):**
   a. Mengiterasi dari elemen sebelumnya hingga elemen pertama atau hingga ditemukan posisi yang tepat.
   b. Membandingkan key dengan elemen-elemen yang sudah terurut.
5. **Geser Elemen:**
   Jika nilai key lebih kecil dari elemen yang dibandingkan, geser elemen ke posisi berikutnya.
6. **Penempatan Key:**
   Tempatkan key pada posisi yang tepat dalam array yang sudah terurut.
7. **Pengulangan:**
   Iterasi berlanjut hingga seluruh array terurut.

Algoritma Insertion Sort bekerja dengan menyusun satu elemen pada setiap iterasi ke dalam bagian array yang sudah terurut sebelumnya. Pseudocode di atas mencerminkan langkah-langkah implementasi Insertion Sort yang sederhana dan efektif dengan kompleksitas waktu O(n^2).

6.3.3 Selection Sort

Selection Sort adalah algoritma sorting yang bekerja dengan cara secara berulang memilih elemen terkecil dari sisa dataset dan menukarnya dengan elemen pertama, kemudian memilih elemen terkecil dari sisa dataset yang lebih kecil, dan seterusnya. Proses ini terus berlanjut hingga seluruh dataset tersusun dengan benar. Meskipun Selection Sort sederhana untuk dipahami, algoritma ini memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam dataset. Oleh karena itu, Selection Sort mungkin tidak efisien untuk dataset yang besar. Keunggulan Selection Sort terletak pada kestabilannya, yaitu elemen-elemen dengan nilai yang sama akan tetap urutannya setelah disortir, dan pada implementasinya yang sederhana. Meskipun jarang

digunakan pada dataset besar, Selection Sort tetap bermanfaat untuk keperluan pendidikan dan di lingkungan di mana kompleksitas algoritma menjadi lebih kecil pertimbangan dibandingkan dengan kejelasan kode.

```
Procedure SelectionSort(arr: array of integer)
    n <- length(arr)
    for i from 0 to n-1 do
        minIndex <- i
        for j from i+1 to n-1 do
            // Temukan indeks elemen terkecil dari array yang belum terurut
            if arr[j] < arr[minIndex] then
                minIndex <- j
            end if
        end for
        // Tukar elemen terkecil dengan elemen pertama dalam array yang belum terurut
        temp <- arr[minIndex]
        arr[minIndex] <- arr[i]
        arr[i] <- temp
    end for
End Procedure
```

Gambar 6. 3. Pseudocode Algoritma Selection Sort

Berikut adalah penjelasan langkah-langkah algoritma Insertion Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.3:

1. **Inisialisasi Variabel:**
   a. **n**: Panjang (jumlah elemen) array.
   b. **minIndex**: Variabel untuk menyimpan indeks elemen terkecil.
   c. **i** dan **j**: Variabel loop untuk iterasi.
2. **Iterasi Outer (i):**
   Mengiterasi melalui seluruh array, dimulai dari indeks pertama.
3. **Inisialisasi minIndex:**
   Set minIndex sama dengan **i**.
4. **Iterasi Inner (j):**
   a. Mengiterasi dari elemen setelah **i** hingga elemen terakhir.

b. Mencari indeks elemen terkecil dalam array yang belum terurut.

5. **Perbandingan dan Pembaruan minIndex:**
   Jika elemen pada indeks **j** lebih kecil dari elemen pada minIndex, perbarui minIndex menjadi **j**.
6. **Tukar Elemen:**
   Setelah iterasi inner selesai, tukar elemen terkecil dengan elemen pertama dalam array yang belum terurut.
7. **Pengulangan:**
   Iterasi berlanjut hingga seluruh array terurut.

Algoritma Selection Sort bekerja dengan mencari elemen terkecil pada setiap iterasi inner dan menukarkannya dengan elemen pertama dalam array yang belum terurut. Pseudocode di atas mencerminkan langkah-langkah implementasi Selection Sort yang sederhana dengan kompleksitas waktu O(n^2).

6.4 Algoritma Sorting Efisien

Algoritma Sorting Efisien, termasuk Merge Sort, Quick Sort, dan Heap Sort, dirancang untuk menangani dataset besar dengan lebih efisien. Merge Sort menggunakan pendekatan divide-and-conquer untuk membagi dan menggabungkan dataset, Quick Sort menggunakan strategi pemilihan pivot, dan Heap Sort memanfaatkan struktur data heap untuk penyortiran efisien.

6.4.1 Merge Sort

Merge Sort adalah algoritma sorting yang berbasis pada konsep divide-and-conquer, dirancang untuk menyortir dataset dengan membaginya menjadi dua bagian, menyortir setiap bagian secara rekursif, dan kemudian menggabungkan dua bagian tersebut. Proses ini dilakukan sampai seluruh dataset tersusun dengan benar. Keunggulan utama Merge Sort terletak pada kompleksitas waktu yang relatif stabil, yaitu O(n log n),

membuatnya efisien untuk dataset besar. Selain itu, Merge Sort juga bersifat stabil, artinya elemen-elemen dengan nilai yang sama akan tetap urutannya setelah disortir. Namun, kekurangan utamanya adalah penggunaan ruang memori tambahan untuk penyimpanan sementara selama proses penggabungan, membuatnya memerlukan alokasi memori ekstra. Meskipun demikian, keandalan dan kinerja yang baik menjadikan Merge Sort menjadi pilihan yang populer untuk menyortir data, terutama pada aplikasi yang membutuhkan stabilitas dan efisiensi pada dataset yang besar.

```
Procedure MergeSort(arr: array of integer)
    if length(arr) <= 1 then
        return arr
    else
        mid <- length(arr) / 2
        left <- MergeSort(arr[0..mid-1])
        right <- MergeSort(arr[mid..end])

        // Panggil fungsi merge untuk menggabungkan dua bagian yang sudah diurutkan
        arr <- Merge(left, right)
    end if
End Procedure

Function Merge(left: array of integer, right: array of integer) -> array of integer
    result <- empty array
    i <- 0
    j <- 0

    while i < length(left) and j < length(right) do
        if left[i] <= right[j] then
            append left[i] to result
            i <- i + 1
        else
            append right[j] to result
            j <- j + 1
        end if
    end while

    // Menambahkan sisa elemen dari kedua array (jika ada)
    append remaining elements of left from i to end of result
    append remaining elements of right from j to end of result

    return result
End Function
```

Gambar 6. 4. Pseudocode Algoritma Merge Sort

Berikut adalah penjelasan langkah-langkah algoritma Merge Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.4:

1. **Fungsi MergeSort:**
   a. Jika panjang array kurang dari atau sama dengan 1, kembalikan array tersebut karena sudah terurut.
   b. Jika tidak, hitung nilai tengah (**mid**) dan pisahkan array menjadi dua bagian.
   c. Panggil rekursif **MergeSort** untuk dua bagian tersebut.
   d. Panggil fungsi **Merge** untuk menggabungkan dua bagian yang sudah diurutkan.
2. **Fungsi Merge:**
   a. Inisialisasi array kosong result untuk menyimpan hasil penggabungan.
   b. Gunakan dua variabel **i** dan **j** untuk mengiterasi melalui kedua bagian array yang sudah diurutkan.
   c. Selama masih ada elemen di kedua bagian array, bandingkan elemen-elemen tersebut.
   d. Tambahkan elemen yang lebih kecil ke dalam result dan perbarui indeks i atau j sesuai.
   e. Setelah satu bagian array habis, tambahkan sisa elemen dari bagian yang masih tersisa.

Algoritma Merge Sort bekerja dengan membagi array menjadi dua bagian, melakukan rekursi pada kedua bagian tersebut, dan menggabungkan hasilnya. Proses penggabungan (merge) dilakukan dengan membandingkan dan menyusun elemen-elemen dari dua bagian yang sudah diurutkan. Pseudocode di atas mencerminkan langkah-langkah implementasi Merge Sort dengan kompleksitas waktu O(n log n).

6.4.2 Quick Sort

Quick Sort adalah algoritma sorting yang mengadopsi strategi divide-and-conquer untuk menyortir dataset. Prosesnya dimulai dengan memilih suatu elemen sebagai pivot, kemudian membagi dataset menjadi dua bagian: satu bagian dengan elemen yang lebih kecil dari pivot, dan satu bagian dengan elemen yang lebih besar. Langkah ini diulangi secara rekursif pada kedua bagian tersebut, dan hasilnya digabungkan untuk menghasilkan dataset yang terurut. Kelebihan utama Quick Sort terletak pada kompleksitas waktu rata-rata O(n log n), menjadikannya salah satu algoritma sorting yang paling cepat. Namun, kinerjanya dapat bergantung pada pemilihan pivot dan kondisi dataset. Quick Sort juga bersifat in-place, artinya tidak memerlukan alokasi memori tambahan. Kombinasi antara kecepatan dan penggunaan memori yang efisien menjadikan Quick Sort populer di berbagai aplikasi dan lingkungan pengembangan perangkat lunak.

```
Procedure QuickSort(arr: array of integer, low: integer, high: integer)
    if low < high then
        // Panggil fungsi Partition untuk mendapatkan pivot
        pivotIndex <- Partition(arr, low, high)

        // Panggil rekursif QuickSort pada kedua sisi pivot
        QuickSort(arr, low, pivotIndex - 1)
        QuickSort(arr, pivotIndex + 1, high)
    end if
End Procedure

Function Partition(arr: array of integer, low: integer, high: integer) -> integer
    // Pilih pivot (dalam hal ini, ambil elemen terakhir)
    pivot <- arr[high]
    i <- low - 1

    for j from low to high - 1 do
        if arr[j] <= pivot then
            // Jika elemen ke-j lebih kecil atau sama dengan pivot, tukar dengan elemen di posisi i+1
            i <- i + 1
            swap arr[i] and arr[j]
        end if
    end for

    // Tukar pivot dengan elemen di posisi i+1
    swap arr[i+1] and arr[high]

    // Kembalikan indeks pivot yang baru
    return i + 1
End Function
```

Gambar 6. 5. Pseudocode Algoritma Quick Sort

Berikut adalah penjelasan langkah-langkah algoritma Quick Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.5:

1. **Fungsi QuickSort:**
    a. Jika **low** kurang dari **high**, artinya masih ada lebih dari satu elemen dalam array.
    b. Panggil fungsi **Partition** untuk mendapatkan pivot.
    c. Panggil rekursif **QuickSort** pada kedua sisi pivot.
2. **Fungsi Partition:**
    a. Pilih pivot (dalam hal ini, elemen terakhir dari array).
    b. Inisialisasi indeks i sebagai -1.
    c. Iterasi melalui array dari low hingga high - 1.
    d. Jika elemen ke-j lebih kecil atau sama dengan pivot, tukar dengan elemen di posisi i + 1.
    e. Setelah iterasi, tukar pivot dengan elemen di posisi i + 1.
    f. Kembalikan indeks pivot yang baru.

Algoritma Quick Sort bekerja dengan memilih pivot dari array dan mempartisi array menjadi dua bagian: elemen yang kurang dari atau sama dengan pivot, dan elemen yang lebih besar dari pivot. Proses ini diulangi secara rekursif untuk setiap bagian. Pseudocode di atas mencerminkan langkah-langkah implementasi Quick Sort dengan kompleksitas waktu rata-rata O(n log n).

6.4.3 Heap Sort

Heap Sort adalah algoritma sorting yang menggunakan struktur data heap untuk menyortir dataset. Prosesnya dimulai dengan membangun max heap atau min heap dari dataset, tergantung pada apakah kita ingin menyortir secara ascending atau descending. Setelah heap terbentuk, elemen terbesar (dalam max heap) atau terkecil (dalam min heap), yaitu elemen di akar heap, ditukar dengan elemen terakhir dalam array yang belum terurut. Setelah pertukaran dilakukan, ukuran heap dikurangi satu, dan langkah ini diulangi sampai seluruh dataset tersusun

dengan benar. Heap Sort memiliki kompleksitas waktu O(n log n), menjadikannya efisien untuk dataset besar. Kelebihan lainnya adalah bahwa algoritma ini bersifat in-place, artinya memerlukan ruang memori konstan selain array yang akan disortir. Meskipun Heap Sort mungkin kurang populer dibandingkan dengan beberapa algoritma lainnya, keefisienannya dan ketergantungannya terhadap ruang memori yang rendah membuatnya tetap relevan dalam beberapa konteks pengembangan perangkat lunak.

```
Procedure HeapSort(arr: array of integer)
    n <- length(arr)

    // Bangun max heap
    for i from n/2 - 1 to 0 do
        Heapify(arr, n, i)

    // Ekstrak elemen satu per satu dari heap
    for i from n-1 to 1 do
        // Tukar akar heap (elemen terbesar) dengan elemen terakhir
        swap arr[0] and arr[i]

        // Panggil Heapify untuk mengurutkan sisa heap
        Heapify(arr, i, 0)
    end for
End Procedure

Procedure Heapify(arr: array of integer, n: integer, i: integer)
    largest <- i
    left <- 2 * i + 1
    right <- 2 * i + 2

    // Bandingkan dengan anak kiri
    if left < n and arr[left] > arr[largest] then
        largest <- left

    // Bandingkan dengan anak kanan
    if right < n and arr[right] > arr[largest] then
        largest <- right

    // Jika largest bukan lagi root
    if largest != i then
        // Tukar root dengan elemen terbesar
        swap arr[i] and arr[largest]

        // Rekursif panggil Heapify pada subtree yang terpengaruh
        Heapify(arr, n, largest)
    end if
End Procedure
```

Gambar 6. 6. Pseudocode Algoritma Heap Sort

Berikut adalah penjelasan langkah-langkah algoritma Heap Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.6:

1. **Fungsi HeapSort:**
    a. Mendapatkan panjang array (**n**).
    b. Membangun max heap dengan memanggil fungsi Heapify pada setengah pertama elemen array.
2. **Fungsi Heapify:**
    a. **Heapify** digunakan untuk memastikan bahwa subtree dengan root di indeks **i** memenuhi properti heap maksimum.
    b. Inisialisasi **largest** sebagai root, **left** sebagai anak kiri, dan **right** sebagai anak kanan.
3. **Bandingkan dengan Anak Kiri dan Anak Kanan:**
   Jika anak kiri atau anak kanan lebih besar dari root, perbarui **largest**.
4. **Tukar dan Rekursif:**
   Jika **largest** bukan lagi root, tukar root dengan elemen terbesar (largest), lalu rekursif panggil **Heapify** pada subtree yang terpengaruh.
5. **Iterasi dan Ekstraksi Elemen:**
   Dalam iterasi kedua, ekstrak elemen satu per satu dari heap dengan menukar root dengan elemen terakhir dan panggil **Heapify** untuk memperbarui heap.

Algoritma Heap Sort bekerja dengan membangun max heap dari array dan secara berulang mengekstrak elemen terbesar dari heap (akar heap) hingga array terurut. Pseudocode di atas mencerminkan langkah-langkah implementasi Heap Sort dengan kompleksitas waktu O(n log n).

6.5 Algoritma Sorting Khusus

Algoritma Sorting Khusus, seperti Radix Sort dan Counting Sort, diadaptasi untuk kasus khusus. Radix Sort mengurutkan data berdasarkan digitnya, baik itu secara least significant digit (LSD) atau most significant digit (MSD). Counting Sort, sementara itu, bekerja dengan menghitung kemunculan setiap elemen unik dan menyusunnya berdasarkan frekuensinya. Kedua algoritma ini cocok untuk data non-numerik dan dapat memberikan kinerja yang sangat baik pada kondisi tertentu. Dengan pemahaman ini, pengembang dapat memilih algoritma sorting yang paling sesuai dengan kebutuhan spesifik aplikasi dan karakteristik dataset yang dihadapi.

6.5.1 Radix Sort

Radix Sort adalah algoritma sorting non-comparative yang berfokus pada pengurutan data berbasis digit atau karakter. Prosesnya dilakukan dengan memisahkan elemen dataset ke dalam beberapa grup berdasarkan digit atau karakter pada posisi tertentu. Kemudian, dataset diurutkan ulang secara rekursif berdasarkan digit atau karakter berikutnya, terus-menerus hingga seluruh dataset tersusun dengan benar. Radix Sort bisa berbasis Least Significant Digit (LSD) atau Most Significant Digit (MSD) tergantung pada digit mana yang diurutkan pertama kali. Kelebihan utama Radix Sort adalah kecepatan pada dataset besar dan kemampuannya mengurutkan data non-numerik. Namun, kekurangannya adalah memerlukan penyesuaian terhadap tipe data dan ukuran dataset, dan dapat menjadi kurang efisien pada dataset dengan rentang digit yang besar. Meskipun demikian, Radix Sort tetap menjadi pilihan yang relevan terutama pada kasus-kasus di mana pengurutan data berbasis digit menjadi kritis.

```
Procedure RadixSort(arr: array of integer)
    // Mencari nilai maksimum untuk menentukan jumlah digit
    maxVal <- findMax(arr)

    // Melakukan counting sort pada setiap digit, dimulai dari digit paling rendah
    exp <- 1
    while maxVal / exp > 0 do
        CountingSort(arr, exp)
        exp <- exp * 10
    end while
End Procedure

Function findMax(arr: array of integer) -> integer
    maxVal <- arr[0]
    for i from 1 to length(arr) - 1 do
        if arr[i] > maxVal then
            maxVal <- arr[i]
        end if
    end for
    return maxVal
End Function

Procedure CountingSort(arr: array of integer, exp: integer)
    n <- length(arr)
    output <- empty array
    count <- array of size 10 initialized with zeros

    // Hitung frekuensi setiap digit
    for i from 0 to n - 1 do
        index <- arr[i] / exp % 10
        count[index] <- count[index] + 1
    end for

    // Akumulasi frekuensi untuk mendapatkan posisi akhir setiap digit
    for i from 1 to 9 do
        count[i] <- count[i] + count[i - 1]
    end for

    // Bangun array output dengan mengurutkan berdasarkan digit yang sedang dihitung
    for i from n - 1 down to 0 do
        index <- arr[i] / exp % 10
        output[count[index] - 1] <- arr[i]
        count[index] <- count[index] - 1
    end for

    // Salin kembali array output ke array asal
    for i from 0 to n - 1 do
        arr[i] <- output[i]
    end for
End Procedure
```

Gambar 6. 7. Pseudocode Algoritma Radix Sort

Berikut adalah penjelasan langkah-langkah algoritma Radix Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.7:

1. **Fungsi RadixSort:**
   a. Menentukan nilai maksimum dari array untuk menentukan jumlah digit.
   b. Melakukan counting sort pada setiap digit, dimulai dari digit paling rendah hingga digit paling tinggi.
2. **Fungsi findMax:**
   Mencari nilai maksimum dalam array.
3. **Fungsi CountingSort:**
   a. Inisialisasi array count untuk menyimpan frekuensi setiap digit (0-9).
   b. Iterasi pertama untuk menghitung frekuensi setiap digit.
   c. Iterasi kedua untuk mengakumulasi frekuensi dan mendapatkan posisi akhir setiap digit.
   d. Iterasi ketiga untuk membangun array output berdasarkan digit yang sedang dihitung.
   e. Iterasi terakhir untuk mengembalikan hasil sorting ke array asal.

Algoritma Radix Sort bekerja dengan mengurutkan array berdasarkan digit-digitnya, dimulai dari digit paling rendah ke digit paling tinggi. Pseudocode di atas mencerminkan langkah-langkah implementasi Radix Sort dengan kompleksitas waktu O(d * (n + k)), di mana d adalah jumlah digit, n adalah jumlah elemen dalam array, dan k adalah basis digit (pada kasus ini, 10 karena sistem desimal).

6.5.2 Counting Sort

Counting Sort adalah algoritma sorting non-comparative yang efisien untuk menyortir elemen-elemen dengan kisaran nilai terbatas. Algoritma ini bekerja dengan membuat array "count" yang menyimpan jumlah kemunculan setiap nilai dalam

dataset. Setelah itu, array count diperbarui untuk mencerminkan posisi akhir setiap elemen dalam urutan terurut. Proses ini melibatkan akumulasi jumlah elemen yang lebih kecil atau sama pada array count. Counting Sort memiliki kompleksitas waktu O(n + k), di mana n adalah jumlah elemen dalam dataset dan k adalah rentang nilai unik dalam dataset. Kelebihan utamanya adalah kinerjanya yang sangat baik pada dataset dengan rentang nilai yang terbatas, membuatnya efisien untuk dataset kecil hingga menengah. Namun, Counting Sort tidak efisien untuk dataset dengan rentang nilai yang besar atau ketika nilai-nilai tersebut tidak terdistribusi merata. Algoritma ini sering digunakan sebagai alternatif ketika karakteristik dataset memenuhi syarat-syarat khusus yang mendukung keefisiennannya.

```
Procedure CountingSort(arr: array of integer)
    n <- length(arr)

    // Mencari nilai maksimum dan minimum dalam array
    maxVal <- findMax(arr)
    minVal <- findMin(arr)

    // Menentukan rentang (range) nilai dalam array
    range <- maxVal - minVal + 1

    // Inisialisasi array count dan output
    count <- array of size range initialized with zeros
    output <- empty array

    // Menghitung frekuensi setiap nilai dalam array
    for i from 0 to n - 1 do
        count[arr[i] - minVal] <- count[arr[i] - minVal] + 1
    end for

    // Akumulasi frekuensi untuk mendapatkan posisi akhir setiap nilai
    for i from 1 to range - 1 do
        count[i] <- count[i] + count[i - 1]
    end for

    // Bangun array output dengan mengurutkan berdasarkan frekuensi dan posisi akhir
    for i from n - 1 down to 0 do
        output[count[arr[i] - minVal] - 1] <- arr[i]
        count[arr[i] - minVal] <- count[arr[i] - minVal] - 1
    end for

    // Salin kembali array output ke array asal
    for i from 0 to n - 1 do
        arr[i] <- output[i]
    end for
End Procedure
```

Gambar 6. 8. Pseudocode Algoritma Counting Sort bagian 1

```
Function findMax(arr: array of integer) -> integer
    maxVal <- arr[0]
    for i from 1 to length(arr) - 1 do
        if arr[i] > maxVal then
            maxVal <- arr[i]
        end if
    end for
    return maxVal
End Function

Function findMin(arr: array of integer) -> integer
    minVal <- arr[0]
    for i from 1 to length(arr) - 1 do
        if arr[i] < minVal then
            minVal <- arr[i]
        end if
    end for
    return minVal
End Function
```

Gambar 6. 9. Pseudocode Algoritma Counting Sort bagian 2

Berikut adalah penjelasan langkah-langkah algoritma Counting Sort untuk mengurutkan array of integer pada Pseudocode di Gambar 6.8 dan Gambar 6.9:

1. **Fungsi CountingSort:**
   a. Mencari nilai maksimum dan minimum dalam array untuk menentukan rentang (range) nilai.
   b. Inisialisasi array count dan output.
   c. Iterasi pertama untuk menghitung frekuensi setiap nilai dalam array.
   d. Iterasi kedua untuk mengakumulasi frekuensi dan mendapatkan posisi akhir setiap nilai.
   e. Iterasi ketiga untuk membangun array output berdasarkan frekuensi dan posisi akhir.
   f. Iterasi terakhir untuk mengembalikan hasil sorting ke array asal.
2. **Fungsi findMax dan findMin:**
   Mencari nilai maksimum dan minimum dalam array.

Algoritma Counting Sort bekerja dengan menghitung frekuensi setiap nilai dalam array dan membangun array output berdasarkan frekuensi tersebut. Pseudocode di atas mencerminkan langkah-langkah implementasi Counting Sort dengan kompleksitas waktu O(n + k), di mana n adalah jumlah elemen dalam array dan k adalah rentang nilai.

6.6 Penerapan Sorting dalam Berbagai Konteks

Algoritma sorting memiliki peran penting dalam berbagai konteks dan aplikasi di dunia teknologi informasi. Salah satu penerapannya yang umum terjadi adalah dalam pengelolaan basis data. Sorting memungkinkan penyusunan data berbasis kriteria tertentu, mempercepat operasi pencarian dan penggabungan data. Dalam basis data relasional, misalnya, sorting dapat meningkatkan performa operasi query yang melibatkan klausa ORDER BY.

Selain itu, dalam pemrosesan citra, sorting dapat digunakan untuk mengurutkan nilai intensitas piksel, memudahkan analisis dan manipulasi citra. Contohnya adalah penggunaan algoritma sorting dalam filter citra untuk meningkatkan ketajaman atau mengidentifikasi elemen citra tertentu.

Penerapan sorting juga umum ditemui dalam pengembangan perangkat lunak untuk optimasi pencarian data. Dalam aplikasi pencarian atau aplikasi yang melibatkan struktur data besar, sorting memungkinkan implementasi algoritma pencarian biner atau pencarian sekuen dengan kompleksitas waktu yang lebih rendah.

Selain itu, dalam pengolahan log data atau analisis big data, sorting dapat digunakan untuk menyusun dan mengelompokkan data dengan efisien. Ini mempermudah ekstraksi pola atau tren, serta analisis data besar yang memerlukan akses cepat ke informasi terurut.

Dalam konteks pengembangan algoritma, sorting juga dapat menjadi langkah pra-pemrosesan yang penting. Penggunaan algoritma sorting dalam langkah pra-pemrosesan dapat meningkatkan efisiensi algoritma lain yang memanfaatkan data terurut, seperti algoritma pencarian atau algoritma penggabungan data.

Secara keseluruhan, penerapan sorting membuka pintu untuk meningkatkan efisiensi dan responsivitas dalam berbagai aspek pengembangan perangkat lunak dan analisis data. Dengan memahami konteks dan kebutuhan aplikasi, pengguna dapat memilih algoritma sorting yang paling sesuai untuk mencapai tujuan tertentu.

Selain beberapa contoh penerapan sorting untuk pemrograman di atas, algoritma sorting juga membuka peluang untuk meningkatkan keteraturan dan efisiensi dalam berbagai aspek kehidupan sehari-hari.

1. Pengurutan Barang di Supermarket:
   Di supermarket, barang-barang sering diatur secara terurut berdasarkan kategori atau jenisnya. Penerapan sorting mempermudah konsumen untuk menemukan barang yang mereka butuhkan dengan cepat.
2. Daftar Kontak di Ponsel:
   Daftar kontak pada ponsel umumnya dapat diurutkan berdasarkan abjad atau terakhir dihubungi. Sorting memudahkan pengguna untuk menemukan nomor kontak dengan lebih efisien.
3. Riwayat Pesan di Aplikasi Chat:
   Pesan dalam aplikasi chat sering diurutkan berdasarkan waktu atau abjad. Sorting membantu pengguna untuk melacak dan merespons pesan dengan lebih teratur.
4. Pengurutan Data dalam Aplikasi Keuangan:
   Transaksi keuangan dalam aplikasi perbankan sering diurutkan berdasarkan tanggal atau jenis transaksi. Sorting

memungkinkan pengguna untuk melihat riwayat transaksi secara terstruktur.

5. Daftar Playlist dalam Aplikasi Musik:
   Lagu-lagu dalam daftar playlist dapat diurutkan berdasarkan judul, artis, atau tahun rilis. Sorting memfasilitasi pencarian lagu dan menciptakan pengalaman mendengarkan yang lebih teratur.
6. Pengurutan Data pada Spreadsheet:
   Saat bekerja dengan data pada spreadsheet, pengurutan dapat digunakan untuk menyusun data berdasarkan kolom tertentu.
   Sorting mempermudah analisis data dan identifikasi pola.
7. Penjadwalan Acara pada Kalender:
   Acara-acara dalam kalender sering diurutkan berdasarkan tanggal dan waktu. Sorting membantu pengguna untuk mengelola jadwal dan memprioritaskan kegiatan.
8. Penyusunan Buku di Perpustakaan:
   Buku-buku di perpustakaan diurutkan berdasarkan genre, penulis, atau nomor urut. Sorting memungkinkan pengunjung untuk menemukan buku dengan mudah.
9. Pengurutan Nama dalam Daftar Hadir:
   Dalam pertemuan atau acara, daftar hadir dapat diurutkan berdasarkan abjad atau nomor registrasi. Sorting memudahkan pengelolaan kehadiran peserta.
10. Riwayat Pencarian pada Mesin Pencari:
    Riwayat pencarian di mesin pencari dapat diurutkan berdasarkan waktu atau frekuensi penggunaan. Sorting membantu pengguna untuk mengakses kembali hasil pencarian yang relevan.

Penerapan sorting dalam kehidupan sehari-hari membantu menciptakan keteraturan, efisiensi, dan kemudahan akses terhadap informasi yang diperlukan.

6.7 Kesimpulan

Metode-metode sorting yang telah dijelaskan, seperti Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, dan Counting Sort, memberikan pilihan yang bervariasi untuk pengurutan data tergantung pada kebutuhan spesifik dan kompleksitas dataset. Metode seperti Bubble Sort, Insertion Sort, dan Selection Sort, sederhana dan mudah dipahami, namun lebih cocok untuk dataset kecil atau hampir terurut. Di kehidupan sehari-hari, pengurutan barang di supermarket atau daftar kontak di ponsel merupakan contoh penerapan yang relevan.

Di sisi lain, metode seperti Merge Sort, Quick Sort, dan Heap Sort menawarkan efisiensi yang lebih tinggi, terutama pada dataset yang lebih besar, dengan kompleksitas waktu O(n log n). Penerapan dalam dunia nyata dapat ditemukan dalam pengurutan transaksi keuangan atau penyusunan buku di perpustakaan. Dalam pemrograman, penggunaan metode ini umumnya diutamakan pada aplikasi dengan dataset besar yang membutuhkan pengurutan cepat.

Metode Radix Sort dan Counting Sort efisien pada dataset dengan kisaran nilai terbatas. Pada mesin pencari, pengurutan riwayat pencarian atau dalam penyusunan menu makanan di restoran, contoh penerapannya dapat ditemui. Dalam pemrograman, keduanya bermanfaat untuk dataset dengan karakteristik tertentu, seperti data non-numerik.

Tabel di bawah ini menyajikan perkiraan kompleksitas waktu (time complexity) untuk beberapa metode sorting yang telah disebutkan. Perlu dicatat bahwa kompleksitas waktu dapat bervariasi tergantung pada implementasi spesifik dan karakteristik data yang diurutkan.

Table 6. 1. Perbandingan Time Complexity Algoritma Sorting

Algoritma	Rata-rata	Terbaik	Terburuk	Stabilitas
Bubble Sort	O(n^2)	O(n)	O(n^2)	Stabil
Insertion Sort	O(n^2)	O(n)	O(n^2)	Stabil
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Tidak Stabil
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Stabil
Quick Sort	O(n log n)	O(n log n)	O(n^2)	Tidak Stabil
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Tidak Stabil
Radix Sort	O(d * (n + k))	O(d * (n + k))	O(d * (n + k))	Stabil
Counting Sort	O(n + k)	O(n + k)	O(n + k)	Stabil

Berdasarkan tabel kompleksitas waktu yang disajikan, pemilihan algoritma sorting yang terbaik sangat tergantung pada karakteristik dan kebutuhan spesifik dari dataset yang diurutkan. Namun, untuk kasus umum, beberapa pertimbangan dapat diambil:

1. **Jika Kebutuhan Stabilitas Penting:**
   Jika stabilitas sangat penting, algoritma sorting seperti Merge Sort, Radix Sort, dan Counting Sort cenderung lebih diinginkan karena mereka dapat mempertahankan urutan relatif dari elemen-elemen dengan nilai yang sama.
2. **Untuk Dataset yang Kecil atau Hampir Terurut:**
   Bubble Sort, Insertion Sort, dan Selection Sort mungkin dapat dipertimbangkan, terutama jika dataset relatif kecil atau sudah hampir terurut, karena kompleksitas waktu terbaik mereka pada kasus tersebut relatif rendah.
3. **Kinerja Umum pada Dataset Besar:**
   Jika dataset besar dan kompleksitas waktu yang optimal diperlukan, algoritma sorting seperti Merge Sort, Quick Sort, dan Heap Sort cenderung lebih efisien karena memiliki kompleksitas waktu rata-rata O(n log n) pada kondisi umum.

4. **Dataset dengan Rentang Nilai Terbatas:**
   Counting Sort dan Radix Sort dapat menjadi pilihan yang baik untuk dataset dengan rentang nilai terbatas, terutama ketika rentang nilai tidak terlalu besar.

Penting untuk diingat bahwa keefektifan suatu algoritma sorting juga dapat dipengaruhi oleh faktor-faktor implementasi khusus, seperti overhead dalam pertukaran elemen, keberadaan nilai-nilai yang sama, dan pengaturan memori. Oleh karena itu, pemilihan algoritma sorting harus disesuaikan dengan karakteristik spesifik dari dataset yang dihadapi.

DAFTAR PUSTAKA

Khan, N.R. *et al.* (2023) *DATA STRUCTURE AND ALGORITHM*. Arcler Press.

Pal, D. and Halder, S. (2018) *Data Structures and Algorithms with C*. ALPHA SCIENCE INTERNATIONAL LTD.

Singh, A. (2021) *Data Structure and Algorithm: Simply In Depth*.

BAB 7
PENCARIAN (SEARCHING)

Oleh Risanto Darmawan

7.1 Pendahuluan

Dalam pengembangan perangkat lunak, kemampuan untuk menemukan informasi dengan cepat dan efisien adalah salah satu aspek kritis. Konsep pencarian (searching) dalam struktur data menjadi landasan utama yang mendukung operasi ini. Pencarian tidak hanya mencakup menemukan elemen tertentu, tetapi juga melibatkan pemahaman algoritma yang digunakan untuk memastikan pencarian tersebut dilakukan dengan optimal.

Pada pembahasan ada berbagai aspek pencarian dalam konteks struktur data. Mulai dari algoritma pencarian sederhana hingga konsep pencarian dalam struktur data khusus, pentingnya pemahaman yang mendalam tentang cara mencari informasi dalam kumpulan data yang kompleks.

Peran Pencarian dalam Pengembangan Perangkat Lunak

Pencarian bukan hanya sekadar operasi, tetapi juga fondasi dari banyak aplikasi yang digunakan sehari-hari. Mesin pencari web, basis data, aplikasi e-commerce, dan banyak lagi, semuanya memanfaatkan prinsip-prinsip pencarian untuk memberikan pengalaman yang memuaskan kepada pengguna. Oleh karena itu, memahami bagaimana algoritma pencarian bekerja dan bagaimana mereka dapat dioptimalkan adalah kunci untuk mengembangkan perangkat lunak yang efisien dan responsif.

Tujuan Pembahasan

Pembahasan ini bertujuan untuk memberikan pemahaman mendalam tentang berbagai aspek pencarian dalam struktur data. Berbagai algoritma pencarian, menganalisis efisiensi mereka, dan melihat aplikasi praktis dalam berbagai konteks pengembangan perangkat lunak. Dengan demikian, akan memiliki dasar yang kuat untuk mengambil keputusan cerdas dalam merancang dan mengimplementasikan operasi pencarian.

7.2 Konsep Pencarian (Searching)

Konsep pencarian dalam struktur data merujuk pada proses menemukan informasi yang dibutuhkan dari suatu kumpulan elemen yang disimpan dalam memori komputer. Kumpulan elemen ini dapat memiliki berbagai bentuk, seperti array, linked list, graf, atau pohon (Thareja Reenma, 2014).

Pencarian dalam struktur data bisa diartikan sebagai usaha untuk menemukan elemen tertentu yang memiliki karakteristik tertentu dalam kumpulan elemen tersebut.. Beberapa konsep kunci terkait pencarian dalam struktur data:

- **Pencarian Linear (Sequential Search)**
- **Pencarian Binari (Binary Search)**
- **Pencarian Hashing**
- **Pencarian dalam Struktur Data Khusus**

Proses ini mungkin melibatkan berbagai algoritma tergantung pada jenis struktur data yang digunakan.

7.3 Pencarian Linier (Sequential Search)

Pencarian Linear, juga dikenal sebagai Sequential Search, adalah metode pencarian dasar di mana elemen-elemen dalam kumpulan data diurutkan satu per satu hingga elemen yang dicari ditemukan atau sampai akhir kumpulan data (Barnett Granville, 2008).

Berikut adalah beberapa detail tentang Pencarian Linear:

Proses Pencarian Linear

Inisialisasi: Proses dimulai dengan menginisialisasi indeks pencarian (biasanya dimulai dari 0) dan elemen yang ingin dicari.

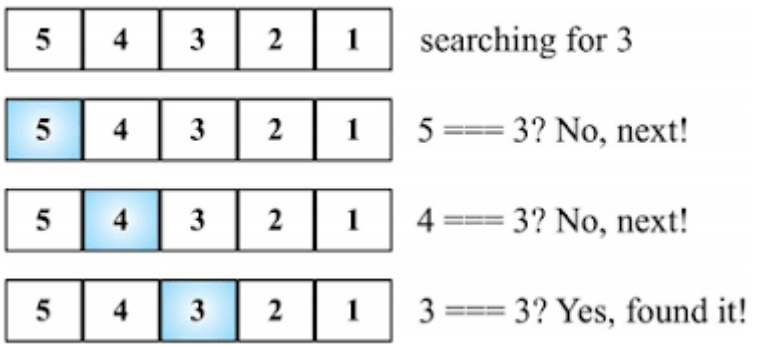
Pencarian Berulang: Kemudian, algoritma ini memeriksa setiap elemen dalam urutan sekuensial dari awal hingga akhir kumpulan data. Setiap elemen diuji untuk mencocokkan dengan elemen yang dicari.

Mencocokkan Elemen: Saat algoritma menemukan elemen yang cocok dengan elemen yang dicari, pencarian dihentikan, dan elemen tersebut dianggap ditemukan.

Akhir Pencarian: Pencarian linear akan berakhir dalam dua situasi:

elemen yang dicari ditemukan, dan indeksnya akan menunjuk ke elemen tersebut.

Jika algoritma telah memeriksa seluruh kumpulan data tanpa menemukan elemen yang dicari. Dalam hal ini, pencarian dianggap gagal, dan biasanya indeks pencarian akan memiliki nilai yang menunjukkan elemen tidak ditemukan.



Gambar 7. 1. Sequential Search
(Sumber : https://www.programmingboss.com/2021/12/sequential-search-in-Java.html)

Kelebihan dan Kekurangan Pencarian Linear

Kelebihan:

Sederhana dan mudah dipahami.

Dapat digunakan pada kumpulan data yang tidak diurutkan.

Kekurangan:

Memiliki kompleksitas waktu O(n), di mana n adalah jumlah elemen dalam kumpulan data. Ini berarti waktu pencarian meningkat secara linear dengan ukuran kumpulan data, yang membuatnya tidak efisien untuk kumpulan data besar.

Tidak efisien pada kumpulan data yang diurutkan, di mana algoritma seperti Pencarian Binari lebih baik.

Pencarian Linear sering digunakan dalam situasi di mana kumpulan data tidak besar atau ketika tidak ada informasi tambahan yang dapat digunakan untuk meningkatkan efisiensi pencarian. Namun, pada kumpulan data besar atau kumpulan data yang diurutkan, algoritma pencarian lain seperti Pencarian Binari biasanya lebih disukai karena mereka lebih efisien dalam hal waktu eksekusi.

Contoh sederhana implementasi Pencarian Linear (Sequential Search) dalam bahasa pemrograman Python:

```
def sequential_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Mengembalikan indeks elemen yang ditemukan
    return -1  # Mengembalikan -1 jika elemen tidak ditemukan

# Contoh penggunaan:
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target_element = 9

result = sequential_search(my_list, target_element)

if result != -1:
    print(f"Elemen {target_element} ditemukan di indeks {result}.")
else:
    print(f"Elemen {target_element} tidak ditemukan dalam kumpulan data.")
```

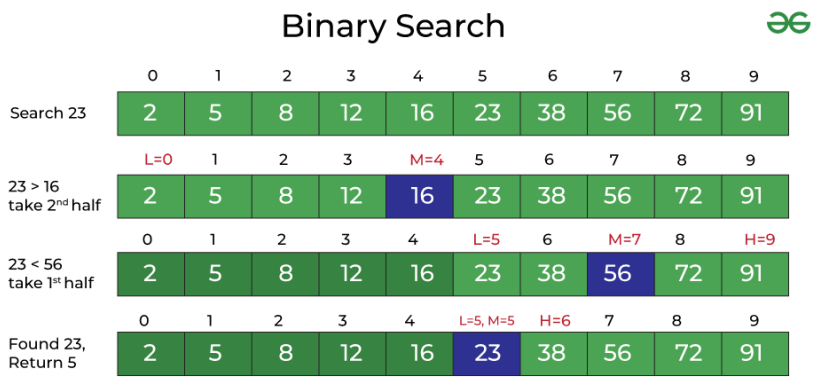
Dalam contoh di atas, sequential_search adalah sebuah fungsi yang melakukan pencarian linear dalam sebuah list (arr) untuk elemen yang sesuai dengan target. Fungsi ini memeriksa setiap elemen dalam list secara berurutan. Jika elemen yang dicari ditemukan, fungsi akan mengembalikan indeks elemen tersebut. Jika tidak ditemukan, fungsi akan mengembalikan -1.

Pada contoh penggunaan di atas, elemen 9 dicari dalam list my_list. Hasilnya adalah bahwa elemen 9 ditemukan di indeks ke-5 dalam list tersebut.

7.4 Pencarian Binari (Binary Search)

Pencarian Binari (Binary Search) adalah algoritma pencarian yang efisien yang digunakan pada kumpulan data yang telah diurutkan. Algoritma ini mengurangi jumlah elemen yang harus diperiksa setiap kali langkah pencarian dilakukan (Bullinaria John,

2019) .



Gambar 7. 2. Binary Search
(Sumber : https://www.geeksforgeeks.org/binary-search-in-java/)

Proses Pencarian Binari

Inisialisasi: Proses dimulai dengan menginisialisasi dua indeks, yaitu indeks awal (biasanya 0) dan indeks akhir (biasanya panjang kumpulan data dikurangi satu). Indeks ini akan menunjukkan rentang pencarian saat ini.

Pencarian Berulang: Selama rentang pencarian (indeks awal hingga indeks akhir) masih ada, algoritma ini mengambil elemen tengah dari rentang ini dan membandingkannya dengan elemen yang dicari.

Mencocokkan Elemen: Jika elemen tengah sama dengan elemen yang dicari, pencarian dihentikan, dan elemen tersebut dianggap ditemukan.

Update Rentang: Jika elemen tengah lebih kecil dari elemen yang dicari, indeks awal diperbarui untuk menjadi satu lebih besar dari indeks tengah. Jika elemen tengah lebih besar dari elemen yang dicari, indeks akhir diperbarui menjadi satu lebih kecil dari indeks tengah.

Akhir Pencarian: Pencarian binari akan berakhir ketika salah satu dari tiga kondisi terpenuhi:

Elemen yang dicari ditemukan, dan indeksnya akan menunjuk ke elemen tersebut.

Rentang pencarian tidak lagi ada, yang berarti elemen tidak ditemukan dalam kumpulan data yang diurutkan.

Rentang pencarian hanya memiliki satu elemen, dan elemen ini tidak sama dengan elemen yang dicari.

Kelebihan dan Kekurangan Pencarian Binari

Kelebihan:
Sangat efisien pada kumpulan data yang telah diurutkan karena membagi rentang pencarian menjadi setengah setiap kali langkah dilakukan.

Memiliki kompleksitas waktu O(log n), di mana n adalah jumlah elemen dalam kumpulan data. Ini membuatnya sangat cepat untuk kumpulan data besar.

Kekurangan:

Hanya dapat digunakan pada kumpulan data yang telah diurutkan. Jika kumpulan data tidak diurutkan, algoritma ini tidak akan berfungsi.

Pencarian Binari digunakan secara luas dalam pemrograman dan ilmu komputer karena efisiensinya. Itu cocok untuk pencarian dalam array yang besar seperti database, direktori, atau daftar nama yang telah diurutkan.

Contoh implementasi Pencarian Binari (Binary Search) dalam bahasa pemrograman Python

```
def binary_search(arr, target):

  left, right = 0, len(arr) - 1  # Inisialisasi indeks awal dan akhir
  while left <= right:
    mid = (left + right) // 2  # Menghitung indeks tengah
    if arr[mid] == target:
      return mid  # Mengembalikan indeks elemen yang ditemukan
    elif arr[mid] < target:
      left = mid + 1  # Perbarui indeks awal jika elemen tengah lebih kecil
    else:
      right = mid - 1  # Perbarui indeks akhir jika elemen tengah lebih besar
  return -1  # Mengembalikan -1 jika elemen tidak ditemukan

# Contoh penggunaan:
sorted_list = [1, 3, 5, 7, 9, 11, 13, 15, 17]
target_element = 7

result = binary_search(sorted_list, target_element)

if result != -1:
  print(f"Elemen {target_element} ditemukan di indeks {result}.")
else:
  print(f"Elemen {target_element} tidak ditemukan dalam kumpulan data.")
```

Dalam contoh di atas, binary_search adalah fungsi yang melakukan pencarian binari dalam sebuah list yang telah diurutkan (arr) untuk elemen yang sesuai dengan target. Fungsi ini menggunakan dua indeks, yaitu left dan right, untuk mengontrol rentang pencarian. Pada setiap langkah, indeks tengah (mid) dihitung, dan elemen tengah dibandingkan dengan elemen yang dicari.

Pada contoh penggunaan di atas, elemen 7 dicari dalam list yang telah diurutkan. Hasilnya adalah bahwa elemen 7 ditemukan di indeks ke-3 dalam list tersebut. Algoritma ini sangat efisien karena membagi rentang pencarian menjadi setengah setiap kali langkah dilakukan.

7.5 Pencarian Hashing

Pencarian Hashing adalah metode pencarian yang memanfaatkan fungsi hash untuk mengindeks dan mencari elemen dalam struktur data yang disebut tabel hash. Fungsi hash mengonversi kunci pencarian menjadi indeks dalam tabel hash, yang memungkinkan pencarian dengan waktu yang sangat cepat (Das Vinu V, 2006).

Berikut adalah detail tentang Pencarian Hashing

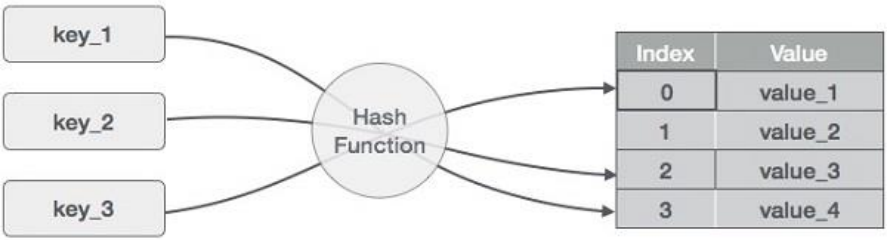
Proses Pencarian Hashing

Inisialisasi Tabel Hash, Proses dimulai dengan inisialisasi sebuah tabel hash yang digunakan untuk menyimpan data. Ukuran tabel hash harus dipilih secara hati-hati, dan biasanya berhubungan dengan jumlah data yang akan disimpan.

Fungsi Hash, Ketika elemen dimasukkan ke dalam tabel hash, fungsi hash digunakan untuk mengonversi kunci (misalnya, nilai yang ingin dicari) menjadi indeks dalam tabel hash. Fungsi hash harus dirancang untuk menghasilkan indeks yang unik untuk setiap kunci yang mungkin, dan ini harus dilakukan dengan cepat.

Penyimpanan Elemen, Elemen atau data yang ingin dicari dimasukkan ke dalam tabel hash menggunakan indeks yang dihasilkan oleh fungsi hash. Ini memungkinkan penyimpanan data yang efisien dan pencarian yang cepat.

Pencarian, Ketika ingin mencari elemen dalam tabel hash, dapat menggunakan fungsi hash yang sama untuk mengonversi kunci pencarian menjadi indeks. Kemudian, dapat langsung mengakses elemen pada indeks ini untuk melihat apakah elemen yang dicari ada di sana.



Gambar 7. 3. Hashing Search
(Sumber :
https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm)

Penanganan Tabrakan (Collision Handling), Terkadang, dua kunci yang berbeda dapat menghasilkan indeks yang sama dalam tabel hash, ini dikenal sebagai tabrakan (collision). Untuk menangani tabrakan, beberapa metode seperti chaining (menggunakan linked lists untuk menampung elemen dengan indeks yang sama) atau probing (mencari slot kosong lainnya dalam tabel) digunakan.

Kelebihan dan Kekurangan Pencarian Hashing

Kelebihan:

Pencarian dengan waktu konstan O(1) jika fungsi hash dirancang dengan baik. Artinya, waktu pencarian tidak bergantung pada jumlah elemen dalam tabel hash.

Efisien untuk kumpulan data besar jika fungsi hash yang baik digunakan.

Kekurangan:

Dibutuhkan desain dan pemilihan fungsi hash yang cermat. Jika fungsi hash tidak efisien atau jika terdapat tabrakan yang sering, kinerja pencarian dapat memburuk.

Tidak sesuai untuk semua jenis data. Pencarian hashing idealnya digunakan pada data yang dapat diindeks dengan baik menggunakan fungsi hash.

Pencarian Hashing banyak digunakan dalam pemrograman dan basis data untuk pencarian data yang cepat, seperti pencarian data dalam basis data yang diindeks atau pencarian kata kunci dalam mesin pencari.

Contoh sederhana implementasi Pencarian Hashing menggunakan bahasa pemrograman Python dengan menggunakan dictionary, yang pada dasarnya adalah struktur data yang menerapkan konsep hashing,

```
# Inisialisasi tabel hash (dictionary) dengan beberapa data
tabel_hash = {
   'John': 28,
   'Alice': 22,
   'Bob': 25,
   'Eve': 21,
   'Charlie': 30
}

# Fungsi hash sederhana untuk mengonversi nama menjadi indeks
def fungsi_hash(nama):
   # Menggunakan panjang nama sebagai indeks
   return len(nama)

# Contoh pencarian menggunakan fungsi hash
nama_dicari = 'Bob'
indeks = fungsi_hash(nama_dicari)

# Mencari nama dalam tabel hash
if nama_dicari in tabel_hash:
   print(f"{nama_dicari} ditemukan dengan umur {tabel_hash[nama_dicari]}
tahun.")
else:
   print(f"{nama_dicari} tidak ditemukan dalam tabel hash.")
```

Dalam contoh ini, tabel hash (dictionary) yang menyimpan nama sebagai kunci dan umur sebagai nilai, juga memiliki fungsi hash yang sangat sederhana yang menggunakan panjang nama sebagai indeks. Ketika mencari nama dalam tabel hash, akan menggunakan fungsi hash ini untuk menghasilkan indeks, lalu memeriksa apakah kunci (nama) ada dalam tabel hash. Jika ada, akan mencetak umur yang sesuai. Jika tidak, diberikan pesan bahwa nama tidak ditemukan.

Penting untuk dicatat bahwa contoh di atas adalah implementasi sederhana untuk tujuan pemahaman konsep. Dalam aplikasi nyata, fungsi hash akan dirancang secara lebih kompleks dan efisien, dan tabel hash mungkin digunakan untuk menyimpan data yang jauh lebih besar dan kompleks.

7.5 Pencarian dalam Struktur Data Khusus

Pencarian dalam Struktur Data Khusus mengacu pada penggunaan algoritma pencarian yang dikhususkan untuk struktur data tertentu, seperti pohon pencarian biner, pohon B, atau graf. Setiap jenis struktur data memiliki karakteristik unik, dan algoritma pencarian yang efisien biasanya dikembangkan untuk memanfaatkan karakteristik tersebut. Berikut adalah beberapa contoh pencarian dalam struktur data khusus:

a. Pencarian dalam Pohon Pencarian Biner (Binary Search Tree)

Dalam pohon pencarian biner, pencarian dapat dilakukan dengan membandingkan kunci yang dicari dengan kunci di setiap simpul.

Jika kunci yang dicari lebih kecil dari kunci simpul saat ini, pencarian dilanjutkan di subpohon kiri; jika lebih besar, pencarian dilanjutkan di subpohon kanan.

Ini adalah contoh dari pencarian dalam Binary Search Tree.

```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

```
def search_in_bst(root, key):
    # Basis: Root adalah None atau key ditemukan di root
    if root is None or root.val == key:
        return root

    # Jika key lebih kecil dari nilai root, cari di subpohon kiri
    if key < root.val:
        return search_in_bst(root.left, key)

    # Jika key lebih besar dari nilai root, cari di subpohon kanan
    return search_in_bst(root.right, key)

# Contoh penggunaan
root = TreeNode(10)
root.left = TreeNode(5)
root.right = TreeNode(15)
root.left.left = TreeNode(3)
root.left.right = TreeNode(7)

result = search_in_bst(root, 7)
if result:
    print(f"Elemen ditemukan: {result.val}")
else:
    print("Elemen tidak ditemukan")
```

b. Pencarian dalam Pohon B (B-Tree)

Pohon B adalah struktur data yang digunakan terutama untuk penyimpanan data dalam basis data dan sistem penyimpanan lainnya.

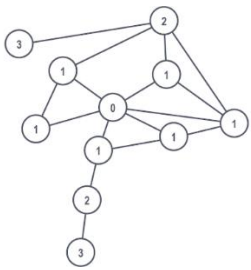
Pencarian dalam pohon B memanfaatkan kemampuan pohon B untuk menyusun data dalam beberapa tingkatan sehingga pencarian dapat dilakukan secara efisien.

Contoh dari pencarian dalam B-Tree :

```
# Library untuk B-Tree
from sortedcontainers import SortedDict

# Membuat B-Tree
btree = SortedDict({5: 'Nilai untuk kunci 5', 15: 'Nilai untuk kunci 15'})

# Pencarian dalam B-Tree
result = btree.get(5)
if result:
   print(f"Nilai ditemukan: {result}")
else:
   print("Nilai tidak ditemukan")
```

c. Pencarian dalam Graf

Dalam graf, pencarian dapat digunakan untuk menemukan lintasan atau hubungan antara dua simpul atau elemen dalam graf (Mehta Dinesh, 2005)



Gambar 7. 4. Breadth-first Search
(Sumber : https://neo4j.com/blog/graph-search-algorithm-basics/)

Seperti yang terlihat di atas, pencarian breadth-first search dimulai dari node yang diberikan (ditandai 0) dan kemudian melakukan perjalanan ke setiap node yang hanya berjarak satu lompatan (ditandai 1) sebelum melakukan perjalanan ke setiap node yang berjarak dua lompatan (2). Kemudian, hanya setelah

melakukan perjalanan ke semua 2 node, ia akan menjelajahi satu lompatan lagi ke node yang ditandai 3.

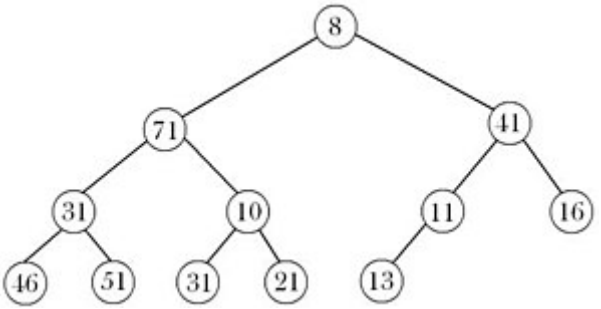
Berbagai algoritma pencarian digunakan dalam graf, seperti Depth-First Search (DFS) dan Breadth-First Search (BFS).

Berikut contoh dari pencarian dalam Graf

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbor):
        if node not in self.graph:
            self.graph[node] = []
        self.graph[node].append(neighbor)

    d, ef dfs_search(self, start, target, visited=None):
        if visited is None:
            visited = set()
        visited.add(start)

        if start == target:
            return True

        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                if self.dfs_search(neighbor, target, visited):
                    return True
        return False

# Contoh penggunaan
my_graph = Graph()
my_graph.add_edge(1, 2)
my_graph.add_edge(2, 3)
my_graph.add_edge(3, 4)

result = my_graph.dfs_search(1, 4)
if result:
    print("Node ditemukan dalam graf")
else:
    print("Node tidak ditemukan dalam graf")
```

d. Pencarian dalam Heap

Heap adalah struktur data yang dikelompokkan menjadi dua jenis: heap maksimum (max heap) dan heap minimum (min heap) (Barnett Granville, 2008).



Gambar 7. 5. Min-max heap
(Sumber : https://neo4j.com/blog/graph-search-algorithm-basics/)

Dalam konteks pencarian dalam heap, kita sering tertarik pada elemen terbesar (max heap) atau terkecil (min heap) dalam tumpukan.

Max Heap:

Heap maksimum mengatur elemen sehingga setiap elemen induk lebih besar dari kedua anaknya.

Operasi pencarian dalam max heap melibatkan mengambil elemen teratas, yang selalu elemen terbesar.

Min Heap:

Heap minimum mengatur elemen sehingga setiap elemen induk lebih kecil dari kedua anaknya.

Operasi pencarian dalam min heap melibatkan mengambil elemen teratas, yang selalu elemen terkecil.

Contoh Pencarian dalam Heap:

```
import heapq

# Membuat heap (min heap)
my_heap = [3, 1, 4, 1, 5, 9, 2, 6]

# Pencarian nilai terbesar dalam heap (max heap)
max_value = heapq.nlargest(1, my_heap)[0]
print(f"Nilai terbesar dalam heap: {max_value}")
```

Dalam contoh ini, heapq.nlargest digunakan untuk mencari nilai terbesar dalam heap (max heap).

e. Pencarian dalam Trie (Pohon Digital)

Trie, atau pohon digital, adalah struktur data yang digunakan untuk penyimpanan dan pencarian data dengan kunci yang dapat diurai menjadi karakter. Setiap node dalam trie mewakili satu karakter, dan jalur dari akar ke simpul tertentu membentuk kunci.

Trie sangat efisien untuk pencarian dan penyisipan string.

Setiap edge dari simpul ke simpul lain mewakili satu karakter.

Node yang mewakili akhir dari kata memiliki tanda khusus (misalnya, node memiliki atribut is_end_of_word).

Contoh Pencarian dalam Trie:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
# Contoh penggunaan
trie = Trie()
words = ["apple", "app", "banana", "orange"]

for word in words:
    trie.insert(word)

result = trie.search("app")
print(f"Kata 'app' {'ditemukan' if result else 'tidak ditemukan'} dalam trie.")
```

Dalam contoh ini, kita menggunakan trie untuk menyimpan kata-kata dan melakukan pencarian kata tertentu.

f. Pencarian dalam Tabel Hash

Tabel hash adalah struktur data yang menggunakan fungsi hash untuk mengonversi kunci menjadi indeks dan mengelola data di dalam array. Pencarian dilakukan dengan menggunakan fungsi hash untuk menemukan indeks kunci yang sesuai.

Fungsi hash mengonversi kunci menjadi indeks di dalam array.

Setiap indeks diisi dengan bucket yang dapat berisi beberapa elemen.

Operasi pencarian melibatkan menghitung fungsi hash dan mencari di dalam bucket yang sesuai.

Contoh Pencarian dalam Tabel Hash:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index] = value

    def search(self, key):
        index = self.hash_function(key)
        return self.table[index]

# Contoh penggunaan
hash_table = HashTable(10)
hash_table.insert(5, "Nilai untuk kunci 5")
hash_table.insert(15, "Nilai untuk kunci 15")

result = hash_table.search(5)
print(f"Nilai untuk kunci 5: {result}")
```

Dalam contoh ini, kita menggunakan tabel hash sederhana untuk menyimpan nilai dan melakukan pencarian untuk mendapatkan nilai berdasarkan kunci. Pencarian dalam struktur data ini dikustomisasi untuk memanfaatkan sifat-sifat unik dari masing-masing struktur data.

Setiap jenis pencarian dalam struktur data khusus memiliki algoritma dan metode pencarian yang unik, dan efisiensinya bergantung pada struktur data tersebut. Implementasi dan detailnya akan sangat bervariasi tergantung pada jenis struktur data yang digunakan. Biasanya, buku atau sumber referensi yang spesifik tentang struktur data tersebut akan memberikan informasi yang lebih mendalam tentang algoritma pencarian yang sesuai.

7.6 Analisis Efisiensi

Analisis Efisiensi dalam konteks struktur data mencakup penilaian kinerja algoritma pencarian yang digunakan dalam struktur data tertentu. Tujuan dari analisis efisiensi adalah untuk mengukur dan memahami seberapa baik atau buruk algoritma pencarian bekerja dalam berbagai kondisi, terutama dalam hal waktu yang diperlukan untuk menyelesaikan pencarian.

Beberapa aspek penting dalam analisis efisiensi pencarian dalam struktur data:

a. Kompleksitas Waktu (Time Complexity), Analisis efisiensi seringkali melibatkan perhitungan kompleksitas waktu dari algoritma pencarian. Ini mengukur berapa banyak operasi dasar yang diperlukan oleh algoritma dalam hubungannya dengan ukuran input (jumlah elemen dalam struktur data). Notasi kompleksitas waktu O() digunakan untuk menggambarkan seberapa cepat atau lambat algoritma tersebut. Misalnya, O(1) berarti waktu konstan (tidak peduli berapa banyak elemen dalam struktur data), sedangkan

O(n) berarti waktu linear (waktu tumbuh sebanding dengan jumlah elemen).

b. Worst-Case vs. Average-Case, Dalam analisis efisiensi, sering kali penting untuk mempertimbangkan kasus terburuk (worst-case) dan kasus rata-rata (average-case). Worst-case scenario adalah situasi di mana algoritma memerlukan waktu paling lama untuk menyelesaikan pencarian, sementara average-case adalah perkiraan waktu rata-rata berdasarkan sejumlah skenario yang mungkin. Memahami keduanya membantu dalam merencanakan penggunaan algoritma tertentu dalam aplikasi nyata.
c. Pengaruh Struktur Data, Struktur data yang digunakan untuk menyimpan elemen juga memiliki dampak besar pada analisis efisiensi. Sebagai contoh, pencarian dalam pohon pencarian biner (BST) memiliki kompleksitas waktu O(h), di mana h adalah tinggi pohon. Oleh karena itu, efisiensi pencarian dalam BST sangat tergantung pada seberapa seimbang pohon tersebut.
d. Pengaruh Data Terurut atau Tidak, Dalam beberapa kasus, analisis efisiensi harus mempertimbangkan apakah data dalam struktur data terurut atau tidak. Misalnya, pencarian binari efisien jika data terurut, tetapi akan menjadi O(n) jika data tidak terurut.
e. Perbandingan dengan Algoritma Lain, Dalam analisis efisiensi, seringkali perlu membandingkan algoritma pencarian yang berbeda untuk memilih yang paling efisien untuk kasus penggunaan tertentu. Ini dapat melibatkan perbandingan kompleksitas waktu, ruang memori, dan kinerja praktis.
f. Uji Kinerja Empiris, Selain analisis teoritis, uji kinerja empiris dengan data dunia nyata juga penting dalam memahami seberapa baik algoritma pencarian berfungsi dalam situasi nyata.

Analisis efisiensi adalah aspek penting dalam perancangan dan pemilihan algoritma untuk aplikasi praktis. Ini membantu dalam memastikan bahwa pencarian dalam struktur data dapat dijalankan dengan baik dalam berbagai situasi, terutama dalam aplikasi yang melibatkan data dalam skala besar.

7.7 Aplikasi Pencarian

Aplikasi Pencarian merujuk pada berbagai kasus penggunaan di mana operasi pencarian digunakan untuk menemukan data tertentu dalam kumpulan data yang lebih besar. Pencarian adalah operasi penting dalam komputer dan pemrograman, dan digunakan dalam berbagai konteks aplikasi. Beberapa contoh aplikasi pencarian:

a. Mesin Pencari (Search Engines)

Mesin pencari web seperti Google, Bing, dan Yahoo menggunakan algoritma pencarian untuk menemukan dan mengindeks berbagai halaman web di internet. Pengguna dapat memasukkan kata kunci, dan mesin pencari akan mencari dan mengembalikan hasil yang relevan.

Gambar 7. 6. Search Engine
(Sumber : https://www.reindigital.io/post/best-search-engine)

b. Basis Data (Databases)

Pencarian digunakan dalam basis data relasional untuk mengambil data yang sesuai dengan kriteria tertentu. Perintah SQL seperti SELECT digunakan untuk melakukan pencarian data dalam tabel basis data.

c. Pencarian dalam Aplikasi E-Commerce

Di situs web e-commerce seperti Amazon atau eBay, pencarian digunakan oleh pengguna untuk mencari produk yang mereka minati. Sistem ini kemudian mencari produk yang sesuai dalam database produk.

Gambar 7. 7. Amazon Search Engine
(Sumber : https://nozamasol.com/amazon-search-engine/)

d. Pencarian Kata Kunci dalam Dokumen (Document Retrieval)

Dalam sistem manajemen dokumen atau mesin pencari teks, pencarian digunakan untuk menemukan dokumen yang mengandung kata kunci atau frasa tertentu. Ini digunakan dalam sistem perpustakaan digital, arsip, atau platform berita.

e. Pencarian dalam Aplikasi Sosial Media

Aplikasi media sosial seperti Facebook, Twitter, atau Instagram memiliki fitur pencarian yang memungkinkan pengguna mencari pengguna lain, posting, atau konten yang relevan dengan minat mereka.

f. Pencarian Navigasi (GPS)

Dalam aplikasi GPS, pencarian digunakan untuk mencari lokasi atau alamat tertentu. Pengguna dapat memasukkan alamat atau nama tempat, dan sistem GPS akan menemukan rute atau koordinat yang sesuai.

g. Sistem Rekomendasi (Recommendation Systems)

Pencarian juga digunakan dalam sistem rekomendasi, seperti yang digunakan oleh Netflix atau Spotify, untuk mencari dan merekomendasikan konten atau produk yang relevan dengan preferensi pengguna.

h. Permainan (Games)

Dalam permainan video atau permainan komputer, pencarian digunakan untuk menemukan objek dalam lingkungan permainan atau mencari pemain lain dalam permainan multipemain.

i. Pencarian pada Situs Web (Website Search)

Pada banyak situs web, terdapat fasilitas pencarian yang memungkinkan pengguna mencari konten atau informasi tertentu di dalam situs web tersebut.

j. Pencarian dalam Aplikasi Mobile

Aplikasi mobile seringkali memiliki fitur pencarian yang memungkinkan pengguna mencari berbagai jenis konten, mulai dari kontak hingga pesan teks atau aplikasi lainnya.

Pencarian adalah fungsi yang mendasar dalam teknologi informasi dan digunakan dalam berbagai aspek kehidupan sehari-hari kita, baik dalam pencarian informasi di internet hingga navigasi di dalam aplikasi. Algoritma pencarian yang efisien dan tepat adalah kunci untuk pengalaman pengguna yang baik dalam banyak aplikasi ini.

7.8 Kesimpulan

Dalam dunia pemrograman dan ilmu komputer, konsep pencarian (searching) dalam struktur data memiliki peran yang sangat penting. Pencarian memungkinkan untuk menemukan data yang relevan dalam kumpulan data yang mungkin besar dan kompleks. Dalam pembahasan ini, berbagai aspek pencarian dalam struktur data,

Pencarian adalah operasi dasar yang digunakan dalam berbagai aplikasi komputer, mulai dari mesin pencari web hingga pencarian data dalam basis data. Berbagai algoritma pencarian, termasuk Pencarian Linear (Sequential Search) untuk pencarian sederhana dan Pencarian Binari (Binary Search) untuk kumpulan data yang diurutkan.

Terdapat berbagai algoritma pencarian yang dikustomisasi untuk struktur data tertentu, seperti pohon pencarian biner, pohon B, atau graf. Analisis efisiensi algoritma pencarian penting untuk memahami seberapa baik algoritma tersebut bekerja dalam berbagai situasi, baik dalam hal waktu eksekusi maupun penggunaan sumber daya lainnya. Berbagai aplikasi praktis dari konsep-konsep pencarian, mulai dari mesin pencari web hingga sistem rekomendasi.

Penting bahwa pemilihan algoritma pencarian yang sesuai dengan situasi tertentu sangat krusial dalam pengembangan perangkat lunak yang efisien dan aplikasi berkinerja tinggi. Dengan pemahaman yang baik tentang berbagai algoritma

pencarian dan analisis efisiensi, maka pengambilan keputusan yang lebih baik dalam pemrograman dan rekayasa sistem.

DAFTAR PUSTAKA

Barnett Granville and Del Tongo Luca, 2008, Data Structure and Algorithms: Annotated Reference with example First edition, dotnetslackers.com

Bullinaria John, 2019, Lecture Notes for Data Structures and Algorithms, school of Computer Science, University of Birmingham, UK

Cormen Thomas H. et al, Introduction to Algorithms third edition, MIT Press, Massachusetts

Das Vinu V, 2006, Principles of Data Structures using C and C++, New Age International, New Delhi

Goodrich Michael T, Tamassia Roberto, Mount David M, 2011, Data Structure and Algorithms in C++ second edition, John Wiley & Sons, Denver

Horowitz Ellis, Sahni Sartaj, 1982, Fundamentals of Data Structures, Computer Science Press, Maryland

ISRD Group, 2006, Data Structure using C, Tata McGraw-Hill Publishing

Lafore Robert, 1999, Sams Teach Yourself Data Structures and Algorithms in 24 Hours, Sams Publishing

Mehta Dinesh P, Sahni Sartaj, 2005, Handbook of Data Structures and Applications, Chapman & Hall/CRC

Mohd. Sabri Ahmad, 1982, DATA STRUCTURES, Politeknik Kuala Terengganu

Nievergelt Jurg, Hinrichs Klaus, 2011, Algorithms and Data Structures with Application to Graphics and Geometry, Global Text

Nwaocha Vivian, Reju, 2022, Fundamentals of Data Structures, NOUN Press National Open University of Nigeria

Thareja Reenma, 2014, Data Structure using C second edition, Oxford University Press, New Delhi

BIODATA PENULIS

Erick Fernando
Dosen Program Studi Sistem Informasi
Fakultas Teknik dan Informatik

Penulis saat ini merupakan dosen tetap pada Program Studi Sistem Informasi Fakultas Teknik dan Informatik, Universitas Multimedia Nusantara. Menyelesaikan pendidikan S1 pada Jurusan Teknik Informatika dan melanjutkan S2 pada Jurusan Magister Sistem Informasi. Serta Menyelesaikan S3 pada Doktor of Computer Science. Penulis menekuni bidang Basis data, Teknologi Blockchain, Adopsi Teknologi, Big Data, Sistem Informasi Geografis, *Emmbeded System* Dlll.

BIODATA PENULIS

Ahmad Faza, S.Kom., M.T.I.
Dosen Program Studi Sistem Informasi
Fakultas Teknik dan Informatika
Universitas Multimedia Nusantara

Penulis lahir pada 12 Januari 1995 di Bogor. Dengan latar belakang sebagai dosen tetap di Program Studi Sistem Informasi, Fakultas Teknik dan Informatika, Universitas Multimedia Nusantara. Penulis telah menyelesaikan pendidikan tinggi tingkat S1 di Jurusan Sistem Informasi, Universitas Multimedia Nusantara, serta melanjutkan studi ke jenjang S2 di program Magister Teknologi Informasi, Universitas Indonesia. Komitmen Penulis terhadap bidang Sistem Informasi tercermin dalam minat penelitiannya pada IT Governance. Keahliannya didukung oleh sertifikasi pada ITIL V3, yang memberikan landasan kuat dalam mengelola dan mengatur teknologi informasi dengan efektif. Penulis memiliki aspirasi untuk berperan aktif dalam mengembangkan penelitian di bidang sistem informasi di masa depan.

BIODATA PENULIS

Aso Sudiarjo, S.Kom., M.Kom.
Dosen Program Studi Teknik Informatika
Fakultas Teknik Universitas Perjuangan Tasikmalaya

Penulis lahir di Tasikmalaya tanggal 16 Januari 1989. Penulis adalah dosen tetap pada Program Studi Teknik Iformatika, fakultas Teknik Universitas Perjuangan Tasikmalaya. Menyelesaikan pendidikan S1 pada Jurusan Teknik Informatika STMIK Raharja dan melanjutkan S2 pada Jurusan Ilmu Komputer Universitas Budi Luhur. Penulis menekuni bidang Menulis.

BIODATA PENULIS

Dr. Irmawati, S.Kom., MMSI.
Dosen Program Studi Sistem Informasi
Fakultas Teknik dan Informatika
Universitas Multimedia Nusantara

Penulis adalah dosen tetap pada Program Studi Sistem Informasi Fakultas Teknik dan Informatika Universitas Multimedia Nusantara. Menyelesaikan pendidikan S1 dan S2 pada Program Studi Sistem Informasi Universitas Gunadarma dan S3 pada departemen Teknik Elektro Universitas Indonesia. Penulis menekuni bidang keilmuan meliputi *Artificial Intelligence*, *Image Processing*, *Machine Learning*, dan *Deep Learning*.

BIODATA PENULIS

Wirawan Istiono, S.Kom., M.Kom.
Dosen Informatika
Fakultas Teknik dan Informatika, Universitas Multimedia Nusantara

Penulis lahir di Jakarta tanggal 13 April 1983. Penulis adalah dosen pada Program Studi Informatika Fakultas Teknik dan Informatika, Universitas Multimedia Nusantara. Menyelesaikan pendidikan S1 pada Jurusan Teknik Informatika dan melanjutkan S2 pada bidang jurusan yang Teknik Informatika. Pada saat ini, penulis menekuni bidang Penelitian terkait komputer sains, sistem komputer dan teknologi informasi

BIODATA PENULIS

Samuel Ady Sanjaya, S.T., M.T.
Dosen Program Studi Sistem Informasi
Fakultas Teknik dan Informatika Universitas Multimedia Nusantara

Penulis lahir di Purworejo pada tanggal 5 April 1994. Penulis adalah dosen tetap pada Program Studi Sistem Informasi Fakultas Teknik dan Informatika, Universitas Multimedia Nusantara. Menyelesaikan S1 pada jurusan Teknik Informatika dan melanjutkan S2 pada jurusan Informatika. Memiliki fokus penelitian di bidang Big Data, Machine Learning dan juga Geographic Information Systems (GIS).

BIODATA PENULIS

Risanto Darmawan, MM, MKom.
Dosen Program Studi Teknik Infomatika
Fakultas Teknik, Universitas Krisnadwipayana

Penulis lahir di Jakarta, Januari 1967. Penulis menamatkan pendidikan Diploma D2 Informatika di Fakultas Politeknik Pertanian IPB Bogor di 1991. Program Sarjana (S1) di Universitas Budi Luhur, Jakarta, Fakultas Teknologi Informasi, Sistem Informasi di 2000 dan Universitas Terbuka, Tangerang Selatan, Fakultas Ekonomi, Prodi Manajemen di 2003. Penulis juga menyelesaikan program Pasca Sarjana (S2) di Universitas Pelita Harapan, Karawaci, Fakultas Business & Management, spesialisasi Marketing di 2003 dan di Sekolah Pascasarjanam, IPB Bogor, Program Studi Ilmu Komputer, di tahun 2006.