

MAKALAH STRUKTUR DATA DAN ALGORITMA

“Algorithm Design (Quick Sort, Counting Sort dan Binary Heaps).”



Disusun Oleh :

Kelompok 8

- | | |
|-----------------------------|---------------|
| 1. Aisyah Azzahrah | (G1A024033) |
| 2. Alfarian Adisaputra | (G1A024039) |
| 3. Lovien Najla Dhafiyah | (G1A024055) |
| 4. Algedian Syahtria Wangsa | (G1A024071) |
| 5. Achmad Rifky Akbar | (G1A024091) |
| 6. Pandu Pratama Putra | (G1A024097) |
| 7. Habiby Irawan Halim | (G1A024107) |

Dosen Pengampu :

Arie Vatesia, S.T., M.T.I., Ph.D.

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS BENGKULU

2025

STRUKTUR DATA DAN ALGORITMA

Kelompok - 2

Program Studi Informatika

Universitas Bengkulu, Jl. Wr. Supratman Kandang Limun, Bengkulu Bengkulu 38371 A, Indonesia

Telp: (0736) 344087, 22105 - 227

Abstrak— Kenaikan pesat media sosial atau teknologi dalam kehidupan sehari-hari sering dikaitkan dengan algoritma dan desainnya, yang mempunyai kelebihan dan mempermudah dalam aktifitas manusia. Seperti halnya dalam urusan antrian yang kadang perlu algoritma atau pemrograman untuk bisa mempersingkat waktu dan mengurangi tenaga manusia. Karena itulah diperlukannya suatu program yang bisa memuat semua hal itu dalam satu kesatuan perangkat kerja yang bisa dimanfaatkan dengan mudah.

Kata Kunci— Algorithm Design (Quick Sort, Counting Sort dan Binary Heaps.

I. PENDAHULUAN

Pemrograman terstruktur memiliki peranan yang sangat penting dalam desain, pemeliharaan, dan pengembangan perangkat lunak. Konsep ini pertama kali diperkenalkan oleh Edsger Dijkstra, seorang profesor di Universitas Eindhoven, pada tahun 1960. Dalam pandangannya, Dijkstra mengkritik penggunaan instruksi lompatan bebas (GOTO) yang dapat menyebabkan program menjadi tidak terstruktur dan sulit dipahami. Untuk mengatasi masalah ini, pendekatan pemrograman terstruktur dikembangkan dengan tujuan untuk menyusun program secara lebih sistematis, teratur, dan mudah untuk dikembangkan.

Standarisasi dalam pemrograman sangat penting untuk menciptakan program yang berkualitas, fleksibel, dan efektif, baik dalam perancangan maupun pemeliharaannya. Beberapa aspek krusial yang harus diperhatikan dalam penetapan standar ini mencakup teknik pemecahan masalah, penyusunan program, perawatan program, serta prosedur penyelesaian masalah. Program yang baik seharusnya memiliki struktur logika yang terorganisir, struktur data yang jelas, dan dokumentasi yang memadai agar mudah dipahami, diuji, dan dikembangkan lebih lanjut. Selain itu, penting untuk memastikan program dapat beroperasi pada berbagai jenis sistem operasi dan perangkat keras yang berbeda, guna meningkatkan fleksibilitas penggunaannya.

Dalam menyelesaikan permasalahan, seorang programmer harus menguasai teknik penyusunan solusi yang efisien. Dua pendekatan yang umum digunakan adalah metode Top-Down dan Bottom-Up. Meskipun programmer memiliki kebebasan dalam mengembangkan ide, penerapan prosedur standar yang telah terbukti efektif sangat dianjurkan untuk menjamin kualitas dan konsistensi program yang dihasilkan. Dalam praktiknya, seorang pemrogram sering menggabungkan berbagai algoritma yang telah ada untuk menyelesaikan masalah dengan lebih efisien. Sebagai contoh, dalam proses pengurutan data, algoritma seperti Quick Sort sering dipilih karena kecepatannya. Algoritma-algoritma tersebut banyak dibahas dalam literatur

mengenai algoritma, struktur data, dan teknik pemrograman. Oleh karena itu, selain menguasai pembuatan algoritma secara mandiri, seorang programmer juga perlu memahami berbagai algoritma yang telah dikembangkan untuk mempercepat proses penyelesaian masalah.

II. LANDASAN TEORI

1. Desain Algoritma

1) Definisi Desain Algoritma

Dalam aktivitas sehari-hari, kita sering kali tanpa sadar menerapkan algoritma. Secara sederhana, algoritma dapat dipahami sebagai serangkaian langkah logis yang harus diikuti untuk menyelesaikan suatu permasalahan. Langkah-langkah ini harus disusun secara sistematis dan mengikuti urutan yang tepat. Sebagai contoh, ketika kita ingin merebus air: kita mulai dengan menyiapkan panci, mengisinya dengan air, menutup panci, meletakkannya di atas kompor, menyalakan kompor dengan api sedang, dan ketika air mulai mendidih, kita mematikan kompor dan mengangkat pancinya. Urutan ini jelas dan logis, sehingga dapat dianggap sebagai algoritma dalam kegiatan sehari-hari.

Dalam bidang ilmu komputer dan matematika, algoritma dapat diartikan sebagai serangkaian prosedur yang teratur yang digunakan untuk melakukan perhitungan, penalaran otomatis, dan pemrosesan data. Algoritma dimulai dengan input dan kondisi awal, diikuti oleh penjabaran instruksi yang sistematis, eksekusi, serta pemrosesan data sesuai urutan yang telah ditentukan, dan akhirnya menghasilkan output sebelum berhenti pada kondisi akhir. Algoritma berfungsi untuk membantu menyelesaikan masalah sehari-hari. Dengan penerapan algoritma, pembuatan program menjadi lebih terstruktur dan logis, sehingga dapat mengurangi kemungkinan terjadinya kesalahan atau error.

Selain itu, algoritma dapat diterapkan secara berulang, sehingga penulisan kode dalam program menjadi lebih efisien. Ciri-ciri algoritma yang baik mencakup:

- Memiliki titik awal dan akhir, di mana algoritma harus berhenti setelah menyelesaikan serangkaian tugas dengan langkah yang terbatas.
- Setiap langkah harus didefinisikan dengan jelas untuk menghindari ambiguitas dan kebingungan.
- Memiliki masukan (input) atau kondisi awal.
- Memiliki keluaran (output) atau kondisi akhir.
- Algoritma harus efektif, sehingga jika diikuti dengan benar, dapat menyelesaikan suatu masalah

f)

2) Sifat-Sifat Algoritma

Berdasarkan definisi dan ciri algoritma yang telah dijelaskan, dapat disimpulkan bahwa sifat algoritma adalah sebagai berikut:

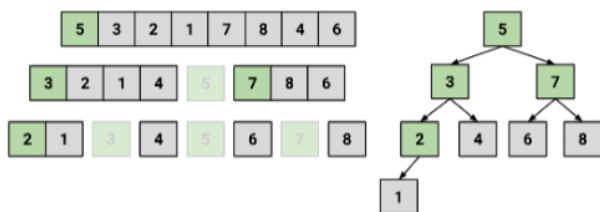
- Input
- Sebuah algoritma memiliki input atau kondisi awal

sebelum dieksekusi, yang dapat berupa nilai-nilai variabel yang diambil dari himpunan tertentu.

- Output
- Algoritma akan menghasilkan output setelah dieksekusi, atau mengubah kondisi awal menjadi kondisi akhir. Nilai output diperoleh dari proses nilai input melalui algoritma.
- Finiteness
- Algoritma harus menghasilkan kondisi akhir atau output setelah sejumlah langkah terbatas dilakukan pada setiap kondisi awal atau input yang diberikan.
- Effectiveness
- Setiap langkah dalam algoritma dapat dilaksanakan dalam waktu tertentu sehingga menghasilkan solusi yang diharapkan.
- Generality
- Langkah-langkah dalam algoritma berlaku untuk semua himpunan input yang relevan dengan masalah yang diberikan, bukan hanya untuk himpunan tertentu saja.

2. Quick Sort

Sistem pengurutan stabil menggunakan algoritma Timsort yang berbasis pada merge sort, dengan waktu eksekusi terburuk yang sama, yaitu linearithmic. Sementara itu, sistem pengurutan tidak stabil menggunakan quicksort untuk mengurutkan larik angka atau boolean, dengan dua varian utama yang akan dibahas: quicksort poros tunggal yang isomorfik dengan pohon pencarian biner, dan quicksort poros ganda yang menyerupai pohon 2-3 yang hanya memiliki node. Quicksort memanfaatkan teknik pemartisian secara rekursif di sekitar elemen pivot, di mana elemen-elemen di sebelah kiri pivot harus lebih kecil atau sama dengan pivot, dan elemen di sebelah kanan harus lebih besar atau sama dengan pivot. Proses pemartisian ini mirip dengan pemilihan elemen akar dalam pohon pencarian biner, di mana semua elemen di subtree kiri lebih kecil dan di subtree kanan lebih besar dari elemen akar.



Quicksort di sebelah kiri selalu memilih elemen paling kiri sebagai elemen pivot dan menggunakan partisi ideal yang mempertahankan urutan relatif elemen yang tersisa. Pohon pencarian biner di sebelah kanan menunjukkan hasil penyisipan setiap elemen dalam urutan input kiri-ke-kanan yang diberikan oleh array.

Quicksort dual-pivot memilih dua poros pada setiap panggilan rekursif, mirip dengan cara node 3-anak dalam pohon 2-3 yang menyimpan dua kunci dan tiga anak. Memilih satu elemen poros serupa dengan memilih satu elemen akar dalam node biner, sedangkan memilih dua elemen poros mirip dengan memilih dua elemen akar dalam node 3-anak. Secara ketat, quicksort dual-pivot tidak isomorfik dengan pohon 2-3 karena tidak ada korespondensi satu-ke-satu. Pertimbangkan pohon 2-3 yang hanya terdiri dari node 2-anak: quicksort yang sesuai adalah quicksort single-pivot, bukan quicksort dual-pivot. Proses mempartisi sebuah array di sekitar dua elemen pivot, p_1

dan p_2 di mana $p_1 \leq p_2$, mengatur ulang elemen-elemen dengan mensyaratkan urutan sebagai berikut:

- Semua elemen yang kurang dari p_1 .
- Elemen pivot p_1 .
- Semua elemen x di mana $p_1 \leq x \leq p_2$.
- Elemen pivot p_2 .
- Semua elemen yang lebih besar dari p_2 .

Quicksort dual-pivot adalah algoritma yang relatif baru yang diperkenalkan pada tahun 2009. Analisis eksperimental menunjukkan bahwa quicksort dual-pivot secara signifikan lebih cepat dibandingkan quicksort single-pivot pada komputer modern. Para ilmuwan komputer mengaitkan peningkatan kinerja ini dengan kemajuan dalam caching CPU dan hierarki memori sejak tahun 1960-an dan 1970-an ketika quicksort single-pivot pertama.

➤ Studi Kasus

Contoh studi kasus dalam Quick Sort dengan menggunakan bahasa pemrograman C++ ialah sebagai berikut :

1. Latar Belakang

Dalam dunia nyata, sering kali kita membutuhkan proses pengurutan data agar lebih mudah diolah atau dianalisis. Misalnya, dalam mengurutkan nilai ujian, daftar harga produk, atau data transaksi berdasarkan jumlah tertentu. Oleh karena itu, diperlukan algoritma pengurutan yang efisien dan cepat. Salah satu algoritma tersebut adalah Dual-Pivot QuickSort, yang mampu mengurutkan data lebih cepat dibandingkan metode pengurutan tradisional.

2. Permasalahan

Diberikan sekumpulan angka acak, yaitu:

{24, 8, 42, 75, 29, 77, 38, 57}

Bagaimana cara mengurutkan angka-angka tersebut dari yang terkecil ke yang terbesar dengan menggunakan algoritma Dual-Pivot Quick Sort?

3. Penyelesaian

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 void swap(int& a, int& b) {
6     int temp = a;
7     a = b;
8     b = temp;
9 }
10
11 void dualPivotQuickSort(vector<int>& arr, int low, int high) {
12     if (low < high) {
13         // Pastikan pivot1 lebih kecil dari pivot2
14         if (arr[low] > arr[high])
15             swap(arr[low], arr[high]);
16
17         int pivot1 = arr[low];
18         int pivot2 = arr[high];
19
20         int i = low + 1;
21         int lt = low + 1;
22         int gt = high - 1;
23
24         while (i <= gt) {
25             if (arr[i] < pivot1) {
26                 swap(arr[i], arr[lt]);
27                 lt++;
28             }
29             else if (arr[i] > pivot2) {
30                 swap(arr[i], arr[gt]);
31                 gt--;
32                 // setelah swap, cek lagi elemen yg ketukar
33             }
34             i++;
35         }
36
37         lt--;
38         gt++;
39
40         swap(arr[low], arr[lt]);
41         swap(arr[high], arr[gt]);
42
43         // Rekursif untuk 3 bagian
44         dualPivotQuickSort(arr, low, lt - 1);
45         dualPivotQuickSort(arr, lt + 1, gt - 1);
46         dualPivotQuickSort(arr, gt + 1, high);
47     }
48 }
49

```

Untuk menyelesaikan masalah ini, kita membuat sebuah program C++ yang:

Menggunakan library vector untuk menampung data secara dinamis. Serta mengimplementasikan Dual-Pivot QuickSort, di mana:

- Dua elemen pertama dan terakhir dari array dipilih sebagai pivot
- Array dipartisi menjadi tiga bagian:
- Elemen yang lebih kecil dari pivot pertama.
- Elemen di antara pivot pertama dan pivot kedua.
- Elemen yang lebih besar dari pivot kedua.
- Proses ini dilakukan secara rekursif hingga seluruh elemen tersusun rapi.

4. Langkah Eksekusi

```
50 int main() {
51     vector<int> arr = {24, 8, 42, 75, 29, 77, 38, 57};
52     int n = arr.size();
53
54     cout << "data sebelum diurutkan: ";
55     for (int i = 0; i < n; i++)
56         cout << arr[i] << " ";
57     cout << endl;
58
59     dualPivotQuicksort(arr, 0, n - 1);
60
61     cout << "Hasil setelah diurutkan: ";
62     for (int i = 0; i < n; i++)
63         cout << arr[i] << " ";
64     cout << endl;
65
66     return 0;
67 }
```

- 1) Program mencetak data sebelum diurutkan.
- 2) Program menjalankan fungsi dualPivotQuicksort untuk mengurutkan elemen.
- 3) Program mencetak data setelah diurutkan.

5. Hasil

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS Z:\tugas informatika\SDA\quicksort-cpp> cd "z:\tugas i
data sebelum diurutkan: 24 8 42 75 29 77 38 57
Hasil setelah diurutkan: 8 24 29 38 42 57 75 77
PS Z:\tugas informatika\SDA\quicksort-cpp> █
```

Setelah algoritma dijalankan, data yang semula tidak terurut akan menjadi: 8 24 29 38 42 57 75 77

6. Kesimpulan

Dengan menggunakan algoritma Dual-Pivot QuickSort dan struktur data vector, proses pengurutan data menjadi lebih cepat dan efisien. Algoritma ini sangat cocok diterapkan pada dataset berukuran besar karena memiliki performa lebih baik dibandingkan metode pengurutan sederhana seperti bubble sort atau insertion sort.

3. Counting Sort

Counting Sort merupakan algoritma pengurutan yang tidak melibatkan perbandingan dalam ilmu komputer, yang beroperasi dengan menghitung jumlah elemen yang memiliki nilai tertentu dalam array, serta menyimpan frekuensi kemunculan setiap nilai. Algoritma ini mengadopsi pendekatan penghitungan kemunculan nilai.

a. Sorting Decision Tree

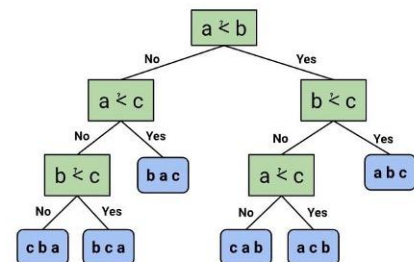
Dalam algoritma merge sort, kita memanfaatkan fakta bahwa subarray yang terdiri dari elemen tunggal sudah terurut dengan sendirinya. Dengan kata lain:

- a) Mengurutkan array yang memiliki 1 elemen tidak

memerlukan perbandingan.

- b) Mengurutkan array yang terdiri dari 2 elemen memerlukan 1 perbandingan.
- c) Mengurutkan array yang berisi 3 elemen memerlukan 2 atau 3 perbandingan, tergantung pada pertanyaannya.

Kita dapat menggambarkan pohon keputusan pengurutan untuk memahami pertanyaan-pertanyaan yang perlu diajukan guna menentukan urutan yang benar untuk 3 elemen. Setiap daun pada pohon keputusan melambangkan kemungkinan urutan yang terurut untuk elemen-elemen tersebut, sementara setiap cabang menunjukkan pilihan untuk menjawab. Agar lebih jelas, lihatlah visualisasi dari pohon keputusan di bawah ini



b. Counting Sort and Enumeration

Pengurutan penghitungan mengatur elemen array dengan menggunakan enumerasi alih-alih perbandingan. Elemen dianggap sebanding jika dapat dibandingkan satu sama lain. Elemen dianggap dapat dihitung jika dapat disusun dalam urutan dari yang pertama hingga yang terakhir. Proses ini dapat dilakukan dengan membuat array hitungan yang berfungsi untuk menyimpan frekuensi kemunculan setiap elemen dalam array input. Lakukan iterasi pada array masukan, dan perbarui array hitungan untuk mencerminkan jumlah kemunculan setiap elemen. Selanjutnya, lakukan iterasi pada array hitungan untuk mengembalikan frekuensi kemunculan setiap elemen ke dalam array input.

c. Radix Sorts

Pengurutan dengan metode hitungan efektif untuk nilai integer kecil jika kita mengetahui rentang antara integer terkecil dan terbesar. Namun, tipe data lain seperti string lebih kompleks untuk dihitung. Sebagai contoh, jika kita memiliki string "a" dan menempatkannya pada indeks 0 dalam array hitungan, di mana posisi string "b" dalam array tersebut? Kita tahu bahwa string "b" muncul setelah "a", tetapi jarak pastinya tidak dapat ditentukan. Kita mungkin menemukan string seperti "aa", "aaa", "aaaa", dan seterusnya, tanpa mengetahui berapa banyak ruang yang perlu disediakan untuk elemen-elemen ini. Untuk mengatasi tantangan ini, kita dapat terinspirasi oleh struktur data trie. Seperti halnya trie yang memecah string menjadi huruf-huruf penyusunnya dan memproses setiap huruf secara terpisah, kita juga dapat menerapkan metode counting sort pada setiap huruf. Radix sort adalah jenis counting sort yang membagi string (atau objek serupa) menjadi subunit yang dapat diurutkan secara independen.

Urutan Radix Digit Paling Signifikan (MSD) dimulai dari karakter paling kiri dan bergerak ke kanan. Proses penghitungan secara rekursif mengurutkan karakter satu per satu, kemudian melanjutkan ke indeks berikutnya dalam string. Sementara itu, Urutan Radix Angka Paling Tidak Penting (LSD) dimulai dari karakter paling kanan dan bergerak ke kiri. Pada setiap indeks dalam string, proses penghitungan berulang akan mengurutkan semua elemen pada indeks tersebut.

➤ Studi Kasus

Contoh studi kasus dalam Counting Sort dengan

menggunakan bahasa pemrograman C++ ialah sebagai berikut :

1. Latar Belakang

Dalam dunia pendidikan, proses pengolahan data nilai mahasiswa menjadi sangat penting untuk melakukan evaluasi dan pemerinkatan. Salah satu kebutuhan umum adalah mengurutkan nilai dari yang terkecil hingga yang terbesar, misalnya untuk membuat daftar ranking atau analisis prestasi. Ketika data yang diolah berukuran kecil dan nilai-nilainya memiliki rentang terbatas, metode pengurutan tradisional seperti bubble sort atau insertion sort bisa saja digunakan. Namun, untuk efisiensi yang lebih tinggi, terutama saat data bersifat integer kecil dan terbatas, teknik Counting Sort sangat cocok digunakan.

Counting Sort adalah algoritma pengurutan yang tidak membandingkan antar elemen, melainkan menghitung jumlah kemunculan tiap nilai dan menentukan posisi akhirnya berdasarkan jumlah tersebut. Algoritma ini sangat cepat untuk data integer kecil, karena kompleksitas waktunya adalah $O(n + k)$, di mana n adalah jumlah data dan K adalah rentang nilai maksimum.

2. Permasalahan

Seorang mahasiswa memiliki data nilai tugas dari 7 temannya, yaitu:

8, 1, 2, 2, 0, 0, 5.

Mahasiswa tersebut ingin mengurutkan nilai dari yang terkecil hingga yang terbesar untuk membuat daftar ranking.

Buatlah program sederhana untuk mengurutkan data tersebut menggunakan teknik counting sort !

3. Penyelesaian



```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     int arr[] = {8, 1, 2, 2, 0, 0, 5};
7     int n = 7;
8
9     int maxVal = 8;
10    vector<int> count(maxVal + 1, 0);
11
12    for(int i = 0; i < n; i++)
13        count[arr[i]]++;
14
15    for(int i = 0; i <= maxVal; i++) {
16        while(count[i]--) {
17            cout << i << " ";
18        }
19    }
20 }
21
```

1. Inisialisasi array dan variabel:
arr adalah array input: {8, 1, 2, 2, 0, 0, 5}.
n adalah jumlah elemen di array, yaitu 7.
maxVal adalah nilai maksimum yang mungkin ada di array, yaitu 8.
2. Membuat array count:
vector<int> count(maxVal + 1, 0);
Ini membuat sebuah vektor count dengan ukuran 9 (dari 0 sampai 8), semua elemen diinisialisasi ke 0.
Tujuannya adalah untuk mencatat berapa banyak setiap angka muncul di arr.
3. Mengisi array count:
Loop for(int i = 0; i < n; i++)
Untuk setiap elemen arr[i], kita tambahkan count[arr[i]]++.
Artinya: berapa kali angka tertentu muncul akan dihitung dan disimpan.
Setelah proses ini, count akan berisi:
count[0] = 2 // angka 0 muncul 2 kali

```
count[1] = 1 // angka 1 muncul 1 kali
count[2] = 2 // angka 2 muncul 2 kali
count[3] = 0 // angka 3 tidak ada
count[4] = 0 // angka 4 tidak ada
count[5] = 1 // angka 5 muncul 1 kali
count[6] = 0 // angka 6 tidak ada
count[7] = 0 // angka 7 tidak ada
count[8] = 1 // angka 8 muncul 1 kali
```

4. Mencetak hasil dengan urutan:

Loop for(int i = 0; i <= maxVal; i++)

Untuk setiap i dari 0 sampai 8:

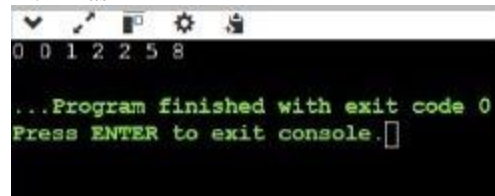
Selama count[i] masih lebih dari 0, cetak i, lalu kurangi count[i] satu per satu (while(count[i]--)).

Cara kerja while(count[i]--):

Jika count[i] > 0, cetak i, lalu kurangi count[i] satu.

Terus ulang sampai count[i] == 0.

4. Hasil



```
0 0 1 2 2 5 8
...Program finished with exit code 0
Press ENTER to exit console.
```

Dari count di atas:

0 dicetak 2 kali

1 dicetak 1 kali

2 dicetak 2 kali

5 dicetak 1 kali

8 dicetak 1 kali

Sehingga hasilnya:

0 0 1 2 2 5 8

5. Binary Heaps

Tumpukan biner merupakan pilihan terakhir dalam desain struktur data 3-choose-2. Ini adalah struktur data pohon yang memenuhi kriteria pohon lengkap dan pohon biner, namun tidak termasuk dalam kategori pohon pencarian. Lalu, apa yang dapat kita lakukan dengan pohon biner yang tidak memiliki sifat pohon pencarian? Pohon 2-3 dan pohon LLRB menawarkan implementasi yang efisien untuk himpunan dan peta, berkat kombinasi sifat pohon pencarian dan invarian yang tinggi. Sebaliknya, tumpukan biner menerapkan tipe data abstrak yang berbeda, yaitu antrian prioritas.

a. Priority Queue Abstract Data Type

Priority Queue adalah struktur data abstrak di mana elemen-elemen diatur berdasarkan nilai prioritas yang relevan. Priority Queue memiliki aplikasi yang signifikan dalam kehidupan sehari-hari. Contohnya, dapat digunakan untuk mengurutkan pasien di ruang gawat darurat rumah sakit. Alih-alih melayani pasien berdasarkan urutan kedatangan seperti dalam antrian biasa, Priority Queue memastikan bahwa pasien dengan kondisi paling kritis atau yang memerlukan penanganan segera mendapatkan perhatian lebih dulu, meskipun ada pasien lain yang tiba lebih awal. Ciri khas dari Priority Queue adalah bahwa beberapa elemen dapat memiliki nilai prioritas yang sama. Sebagai contoh, dua pasien mungkin memerlukan perawatan yang setara.

b. Heap Invariant

Dalam penerapan Priority Queue, tumpukan biner harus mempertahankan Heap Invariant yang tergantung pada jenis tumpukan yang digunakan, apakah itu MinPQ atau MaxPQ. o Min-Heap Invariant Nilai prioritas setiap simpul harus lebih kecil atau sama dengan nilai prioritas dari semua anaknya. o Max-Heap

Invariant Nilai prioritas setiap simpul harus lebih besar atau sama dengan nilai prioritas dari semua anaknya.

c. Array Representation

Meskipun demikian, tumpukan biner ternyata tidak lebih efisien secara asimtotik dibandingkan dengan pohon pencarian seimbang seperti pohon 2-3 atau pohon merah-hitam yang condong ke kiri. Dalam praktiknya, keuntungan utama dari penggunaan tumpukan biner untuk mengimplementasikan antrian prioritas terletak pada kemampuan kita untuk merepresentasikan pohon tersebut menggunakan array. Representasi array adalah bentuk representasi yang umum dan diasumsikan untuk tumpukan biner.

Contoh persoalan kasus pemrograman dengan bahasa c++ Dalam kehidupan sehari-hari, khususnya dalam dunia perkuliahan, mahasiswa sering dihadapkan dengan banyak tugas dari berbagai mata kuliah. Tugas-tugas ini memiliki tingkat urgensi yang berbeda-beda: ada yang perlu segera diselesaikan, ada pula yang bisa dikerjakan belakangan. Oleh karena itu, diperlukan suatu sistem yang dapat membantu mahasiswa mengelola daftar tugas berdasarkan tingkat prioritas.

Solusi kasus pemrograman dengan Binary Heaps:

Menggunakan algoritma binary heaps, lebih tepatnya berbasis antrian prioritas atau priority queue untuk mengatur dan menyelesaikan tugas berdasarkan prioritas tertinggi terlebih dahulu.

- 1) Menyimpan semua tugas ke dalam sebuah struktur data priority_queue.
- 2) Mengatur tugas-tugas secara otomatis berdasarkan prioritas (antrian prioritas akan menjaga elemen dengan prioritas terbesar di paling atas).
- 3) Mengeluarkan tugas satu per satu dari antrian, mulai dari tugas prioritas tertinggi ke terendah.
- 4) Menampilkan nama dan prioritas tugas saat tugas dikeluarkan.

➤ Studi Kasus

Contoh studi kasus dalam Binary Heaps dengan menggunakan bahasa pemrograman C++ ialah sebagai berikut :

1. Latar Belakang

Dalam kehidupan sehari-hari, khususnya dalam dunia perkuliahan, mahasiswa sering dihadapkan dengan banyak tugas dari berbagai mata kuliah. Tugas-tugas ini memiliki tingkat urgensi yang berbeda-beda: ada yang perlu segera diselesaikan, ada pula yang bisa dikerjakan belakangan. Oleh karena itu, diperlukan suatu sistem yang dapat membantu mahasiswa mengelola daftar tugas berdasarkan tingkat prioritas. Salah satu teknik yang efektif untuk kebutuhan ini adalah menggunakan antrian prioritas (priority queue), yang berbasis pada struktur data Binary Heap.

Priority queue memastikan bahwa tugas dengan prioritas tertinggi selalu tersedia untuk diproses terlebih dahulu.

2. Permasalahan

Diberikan sekumpulan tugas dengan prioritas masing-masing:

```
{"SDA", 1}
{"PBO", 3}
{"Agama", 4}
{"Kewarganegaraan", 2}
```

Bagaimana cara menyusun dan menampilkan daftar tugas tersebut berdasarkan urutan prioritas tertinggi ke terendah menggunakan Priority Queue di dalam program C++?

3. Penyelesaian

```
1 #include <iostream>
2 #include <queue>
3 #include <string>
4 using namespace std;
5
6 struct Task {
7     string name;
8     int priority;
9
10     bool operator<(const Task& other) const {
11         return priority > other.priority;
12     }
13 };
14
15 int main() {
16     priority_queue<Task> Matkul;
17
18     Matkul.push({"SDA", 1});
19     Matkul.push({"PBO", 3});
20     Matkul.push({"Agama", 4});
21     Matkul.push({"Kewarganegaraan", 2});
22
23     cout << "Daftar tugas berdasarkan prioritas:\n";
24     while (!Matkul.empty()) {
25         Task tugas = Matkul.top();
26         Matkul.pop();
27         cout << tugas.name << " (Prioritas: " << tugas.priority << ")\n";
28     }
29
30     return 0;
31 }
```

Deklarasi Struktur Data:

Membuat struct Task dengan atribut name dan priority.

Override operator < untuk membalik urutan sehingga prioritas lebih tinggi dianggap lebih kecil dalam konteks antrian.

Inisialisasi Priority Queue:

Membuat priority queue Matkul bertipe Task.

Menambahkan Tugas:

Menambahkan semua tugas ke priority queue menggunakan push().

Mencetak Tugas Berdasarkan Prioritas:

Selama antrian tidak kosong:

Ambil tugas dengan top().

Tampilkan nama dan prioritas tugas.

Hapus tugas dari antrian dengan pop().

4. Hasil

```
Daftar tugas berdasarkan prioritas:
SDA (Prioritas: 1)
Kewarganegaraan (Prioritas: 2)
PBO (Prioritas: 3)
Agama (Prioritas: 4)
```

Setelah algoritma dijalankan, daftar tugas yang ditampilkan berdasarkan prioritas tertinggi ke terendah adalah:

```
Agama (Prioritas: 4)
PBO (Prioritas: 3)
Kewarganegaraan (Prioritas: 2)
SDA (Prioritas: 1)
```

III. KESIMPULAN

Signifikansi pemrograman terstruktur dan desain algoritma, serta penerapan tiga algoritma utama, yaitu Quick Sort, Counting Sort, dan Binary Heaps. Pemrograman terstruktur berkontribusi pada pembuatan program yang sistematis, mudah untuk dikembangkan, dan mengurangi kemungkinan kesalahan. Quick Sort adalah algoritma pengurutan yang efisien dengan menggunakan teknik partisi, meskipun stabilitasnya bergantung pada cara implementasinya. Counting Sort memberikan metode pengurutan alternatif yang tidak memerlukan perbandingan, sangat efektif untuk data dengan rentang nilai yang terbatas, dan konsep ini dapat diperluas melalui Radix Sort. Di sisi lain, Binary Heaps digunakan untuk membangun Priority Queue, yang memungkinkan pengelolaan elemen berdasarkan prioritas dengan efisiensi tinggi berkat representasi array. Dengan memahami dan menguasai berbagai algoritma ini, seorang programmer akan lebih siap menghadapi berbagai tantangan dalam pengembangan perangkat lunak, memilih metode yang paling tepat untuk setiap masalah, serta mampu menciptakan solusi yang lebih optimal dan fleksibel. Semoga laporan ini bermanfaat dan menjadi landasan dalam pengembangan ilmu di bidang algoritma dan struktur data.

REFERENSI

- [1] Kevin Lin, Nathan Brunelle (2024). *Algorithm-design (Quicksort, counting sorts, and binary heaps)*. Courses.Cs.Washington.Edu.
- [2] Samodra, J., Haq, S. T. N., & Widayanti, E. (2025). *Analisis dan Desain Algoritma*. Media Nusa Creative (MNC Publishing).
- [3]

