MAKALAH

TENTANG SORTING

Disusun oleh :

Nama	: L Hafid Alkhair
NIM	: 2023903430060
Kelas	: TRKJ 1.C
Jurusan	: Teknologi Informasi dan Komputer
Program Studi	: Teknologi Rekayasa Komputer Jaringan
Dosen Pengajar	: Indrawati, SST. MT

JURUSAN TEKNOLOGI INFORMASI KOMPUTER

PRODI TEKNOLOGI REKAYASA KOMPUTER JARINGAN

POLITEKNIK NEGERI LHOKSEUMAWE

TAHUN AJARAN 2023/2024

DAFTAR ISI

BAB I

PEMBAHASAN

A. Pengertian Searching

Pencarian (searching) dalam konteks pemrograman merujuk pada proses mencari nilai tertentu di dalam kumpulan data atau struktur data. Tujuan utama dari pencarian adalah menemukan lokasi atau keberadaan elemen yang dicari dengan efisien. Pencarian menjadi aspek kritis dalam pengembangan perangkat lunak, terutama ketika berurusan dengan set data yang besar.

Pada dasarnya, ada dua jenis pencarian utama:

- Pencarian Sekuensial (Linear): Mencari elemen satu per satu dari awal hingga akhir data.
- Pencarian Binari: Digunakan pada data terurut dan membagi data menjadi dua bagian untuk mencari nilai secara efisien.

Pengertian tentang metode pencarian, algoritma, dan struktur data terkait pencarian akan menjadi dasar bagi pengembang untuk memilih pendekatan yang sesuai sesuai dengan kebutuhan spesifik aplikasi atau skenario pemrograman.

B. Pentingnya Searching dalam Pemrograman

Pencarian memainkan peran kunci dalam pengembangan perangkat lunak dan pemrograman komputer. Berikut adalah beberapa alasan mengapa pencarian menjadi aspek penting:

- Efisiensi Pencarian: Dalam kasus pengolahan data yang besar, efisiensi pencarian menjadi krusial. Algoritma pencarian yang tepat dapat mengoptimalkan waktu eksekusi dan sumber daya komputasi.
- Keamanan Data: Pencarian digunakan untuk memverifikasi keberadaan atau ketiadaan data dalam struktur tertentu. Ini penting untuk memastikan integritas dan keamanan data.

- Fungsionalitas Databases: Dalam sistem database, pencarian memungkinkan pengguna untuk mengambil informasi yang dibutuhkan dari dataset yang besar. Ini mencakup pencarian berbasis teks, pencarian kriteria, dan banyak lagi.
- Optimisasi Algoritma: Pemahaman yang baik tentang algoritma pencarian memungkinkan pengembang untuk memilih pendekatan terbaik berdasarkan sifat data dan kebutuhan aplikasi.
- Pengembangan Aplikasi Web dan Mobile: Dalam aplikasi modern, pencarian diperlukan untuk berbagai tujuan seperti pencarian produk, filter data, dan penelusuran konten.
- Kecepatan Pencarian di Internet: Dalam konteks pencarian web, efisiensi pencarian berdampak langsung pada pengalaman pengguna. Mesin pencari seperti Google mengandalkan algoritma pencarian yang canggih untuk menyajikan hasil dengan cepat.
- Analisis Data: Pencarian digunakan dalam analisis data untuk menemukan pola, tren, atau anomali yang mungkin tidak terlihat tanpa penggunaan algoritma pencarian.

Pemahaman yang baik tentang konsep pencarian dan penerapannya memungkinkan pengembang untuk mengoptimalkan kinerja aplikasi dan sistem secara keseluruhan. Oleh karena itu, penting untuk menguasai berbagai teknik pencarian dan memilih pendekatan yang sesuai dengan konteks pengembangan yang sedang dihadapi.

C. Metode Pencarian Sederhana

1. Pencarian Linear

Pencarian linear, juga dikenal sebagai pencarian sekuensial, adalah metode pencarian yang sederhana dan langsung. Prinsip dasarnya adalah memeriksa setiap elemen satu per satu hingga elemen yang dicari ditemukan atau sampai akhir set data dicapai. Dalam konteks array atau daftar, pencarian ini dimulai dari elemen pertama dan terus berlanjut hingga elemen terakhir.

Langkah-langkah Pencarian Linear:

1. Mulai dari elemen pertama dalam set data.
2. Bandingkan elemen saat ini dengan elemen yang dicari.
3. Jika elemen ditemukan, kembalikan indeks atau nilai elemen tersebut.
4. Jika tidak ditemukan, lanjutkan pencarian ke elemen berikutnya.
5. Ulangi langkah-langkah 2-4 sampai elemen yang dicari ditemukan atau seluruh set data telah diperiksa.

Implementasi dalam Bahasa C

```
#include <stdio.h>


int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i; // Mengembalikan indeks elemen yang dicari
      }
   }
```

```
    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

int main() {
    int data[] = {10, 25, 30, 42, 50, 62, 73};
    int target = 42;
    int n = sizeof(data) / sizeof(data[0]);

    int result = linearSearch(data, n, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan dalam set data.\n", target);
    }

    return 0;
}
```

Implementasi ini menciptakan fungsi linearSearch untuk melakukan pencarian linear pada array tertentu dan kemudian menampilkan hasilnya dalam program utama.

2. Pencarian Binary

Pencarian binary, juga dikenal sebagai pencarian biner, merupakan metode pencarian yang efisien untuk set data yang sudah terurut. Prinsip dasarnya melibatkan pembagian set data menjadi dua bagian dan membandingkan elemen tengah dengan elemen yang dicari. Jika elemen yang dicari kurang dari elemen tengah, pencarian dilanjutkan di setengah kiri, dan sebaliknya. Proses ini terus berlanjut hingga elemen ditemukan atau set data menyusut menjadi kosong.

Langkah-langkah Pencarian Binary:

1. Tentukan elemen tengah dari set data.
2. Bandingkan elemen tengah dengan elemen yang dicari.
3. Jika elemen tengah sama dengan elemen yang dicari, pencarian selesai.
4. Jika elemen yang dicari kurang dari elemen tengah, lanjutkan pencarian di setengah kiri.
5. Jika elemen yang dicari lebih besar dari elemen tengah, lanjutkan pencarian di setengah kanan.
6. Ulangi langkah-langkah 1-5 sampai elemen ditemukan atau set data menyusut menjadi kosong.

Contoh implementasi pencarian binary dalam bahasa C:

```
#include <stdio.h>


int binarySearch(int arr[], int low, int high, int target) {
   while (low <= high) {
      int mid = low + (high - low) / 2;
```

```
        if (arr[mid] == target) {
            return mid; // Mengembalikan indeks elemen yang dicari
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

int main() {
    int data[] = {10, 25, 30, 42, 50, 62, 73};
    int target = 42;
    int n = sizeof(data) / sizeof(data[0]);

    int result = binarySearch(data, 0, n - 1, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d.\n", target, result);
    } else {
```

```
        printf("Elemen %d tidak ditemukan dalam set data.\n", target);
    }

    return 0;
}
```

Implementasi ini menciptakan fungsi binarySearch untuk melakukan pencarian binary pada array terurut dan kemudian menampilkan hasilnya dalam program utama.

D. Struktur Data Terkait Searching

1. Array

Kelebihan Array:

- **Akses Cepat:** Array menyediakan akses langsung ke elemen berdasarkan indeksnya. Operasi akses memiliki kompleksitas waktu O(1), membuatnya efisien untuk pencarian dan manipulasi elemen.
- **Pendefinisian Ukuran Tetap:** Ukuran array didefinisikan pada saat deklarasi, memberikan kejelasan dan kontrol pada penggunaan memori. Ini terutama bermanfaat dalam aplikasi yang memerlukan alokasi memori statis.
- **Implementasi Pencarian Linear dan Binary:** Array mendukung metode pencarian linear dan binary, tergantung pada sifat data (terurut atau tidak terurut).

Kekurangan Array:

- Ukuran Tetap: Ukuran array ditentukan saat deklarasi dan sulit diubah selama runtime. Hal ini dapat menyulitkan dalam menangani dataset yang dinamis.
- Pencarian Linear yang Lambat: Pencarian linear pada array besar dapat menjadi lambat, terutama jika elemen yang dicari berada di akhir array atau tidak ada dalam array.
- Penambahan atau Penghapusan Elemen Tidak Efisien: Menambah atau menghapus elemen dari array seringkali memerlukan penggeseran elemen-elemen lain, yang dapat memakan waktu.
- Pemborosan Memori: Jika ukuran array lebih besar dari yang dibutuhkan, ini dapat mengakibatkan pemborosan memori.
- Tidak Mendukung Elemen Berbeda Tipe: Array di C biasanya hanya dapat menyimpan elemen dengan tipe data yang sama.

Implementasi Pencarian Linear dalam Array:

```
#include <stdio.h>


int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i; // Mengembalikan indeks elemen yang dicari
      }
   }
   return -1; // Mengembalikan -1 jika elemen tidak ditemukan
```

```
}

int main() {
    int data[] = {10, 25, 30, 42, 50, 62, 73};
    int target = 42;
    int n = sizeof(data) / sizeof(data[0]);

    int result = linearSearch(data, n, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan dalam set data.\n", target);
    }

    return 0;
}
```

Implementasi Pencarian Binary dalam Array (syarat: array terurut):

```
#include <stdio.h>


int binarySearch(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;


        if (arr[mid] == target) {
            return mid; // Mengembalikan indeks elemen yang dicari
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }


    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}
```

```
int main() {
    int data[] = {10, 25, 30, 42, 50, 62, 73};
    int target = 42;
    int n = sizeof(data) / sizeof(data[0]);

    int result = binarySearch(data, 0, n - 1, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan dalam set data.\n", target);
    }

    return 0;
}
```

Dalam implementasi pencarian di atas, array digunakan sebagai struktur data untuk menyimpan elemen dan dilakukan pencarian linear atau binary tergantung pada sifat data.

2. Linked List

Linked List adalah struktur data dinamis yang terdiri dari simpul-simpul yang saling terhubung. Dalam konteks pencarian, linked list memerlukan pencarian linear karena tidak menyediakan akses langsung ke elemen berdasarkan indeks.

Pencarian Linear dalam Linked List:

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* next;

};


int searchLinkedList(struct Node* head, int target) {

   struct Node* current = head;

   int index = 0;


   while (current != NULL) {

      if (current->data == target) {

         return index; // Mengembalikan indeks elemen yang dicari

      }

      current = current->next;

      index++;
```

```
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

int main() {
    // Contoh implementasi pencarian dalam linked list
    struct Node* head = NULL;
    // ... inisialisasi linked list
    int target = 42;

    int result = searchLinkedList(head, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan dalam linked list.\n", target);
    }

    return 0;
}
```

Kelebihan dan Kekurangan Linked List untuk Searching:

Kelebihan Linked List:

- **Dinamis:** Memungkinkan penambahan atau penghapusan elemen dengan lebih efisien dibandingkan array.
- **Fleksibilitas:** Struktur linked list dapat diubah dengan mudah selama runtime.
- **Alokasi Memori Dinamis:** Alokasi memori hanya dilakukan ketika diperlukan.

Kekurangan Linked List:

- **Pencarian Linear:** Pencarian elemen memerlukan traversal penuh dari awal hingga akhir linked list, yang dapat menjadi lambat untuk linked list yang besar.
- **Tidak Ada Akses Langsung ke Indeks:** Tidak menyediakan akses langsung ke elemen berdasarkan indeks seperti array.

Linked list merupakan pilihan yang baik ketika operasi penambahan atau penghapusan elemen sering terjadi, tetapi kelemahan utamanya terletak pada pencarian linear yang dapat menjadi tidak efisien.

3. Tree

Tree adalah struktur data hirarkis yang terdiri dari simpul-simpul yang terhubung. Pencarian dalam tree umumnya menggunakan konsep rekursi atau iterasi untuk menemukan elemen.

Pencarian dalam Binary Search Tree (BST)

Binary Search Tree (BST) adalah bentuk khusus dari tree yang memungkinkan pencarian elemen dengan efisien.

Pencarian dalam Binary Search Tree:

```
#include <stdio.h>
#include <stdlib.h>

struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

int searchBST(struct TreeNode* root, int target) {
    if (root == NULL || root->data == target) {
        return -1; // Mengembalikan -1 jika elemen tidak ditemukan
    }

    if (target < root->data) {
        return searchBST(root->left, target); // Pencarian di subtree kiri
    } else {
        return searchBST(root->right, target); // Pencarian di subtree kanan
    }
}
```

```
int main() {
    // Contoh implementasi pencarian dalam binary search tree
    struct TreeNode* root = NULL;
    // ... inisialisasi binary search tree
    int target = 42;

    int result = searchBST(root, target);

    if (result != -1) {
        printf("Elemen %d ditemukan dalam binary search tree.\n", target);
    } else {
        printf("Elemen %d tidak ditemukan dalam binary search tree.\n", target);
    }

    return 0;
}
```

Kelebihan dan Kekurangan Binary Search Tree (BST) untuk Searching:

Kelebihan BST:

- **Pencarian Efisien:** Memiliki kecepatan pencarian logaritmik, membuatnya efisien untuk dataset besar.
- **Struktur Terurut:** Elemen diurutkan sehingga pencarian binary dapat digunakan.

Kekurangan BST:

- **Tergantung pada Keseimbangan:** Jika tidak seimbang, BST dapat menjadi tidak efisien dan mirip dengan pencarian linear.
- **Operasi Penambahan dan Penghapusan Rumit:** Menjaga keseimbangan BST saat operasi penambahan dan penghapusan dapat rumit.

Binary Search Tree efisien untuk pencarian, terutama ketika data disusun secara teratur. Namun, perlu diperhatikan agar BST tetap seimbang untuk memastikan kinerja pencarian yang optimal.

E. Algoritma Pencarian Lanjutan

1. Algoritma Interpolasi

Algoritma interpolasi adalah metode pencarian yang digunakan pada set data yang terurut dan merinci langkah-langkahnya sebagai berikut:

- **Hitung Prediksi Lokasi:** Menggunakan formula interpolasi, prediksi lokasi dihitung berdasarkan nilai kunci dan elemen pertama dan terakhir dalam set data.
- **Bandingkan dengan Elemen di Lokasi Prediksi:** Bandingkan nilai kunci dengan elemen di lokasi prediksi.
  - Jika sama, elemen ditemukan.
  - Jika lebih kecil, cari di set data sebelah kiri prediksi.
  - Jika lebih besar, cari di set data sebelah kanan prediksi.

Kelebihan Algoritma Interpolasi:

- **Efisien untuk Data Terdistribusi Merata:** Bekerja lebih efisien jika data terdistribusi merata, memberikan kinerja yang lebih baik dibandingkan pencarian biner.

Kekurangan Algoritma Interpolasi:

- **Hanya untuk Data Terurut:** Hanya dapat digunakan pada set data yang sudah terurut.
- **Kinerja Tergantung Distribusi Data:** Kinerja algoritma sangat dipengaruhi oleh distribusi data. Jika data tidak terdistribusi merata, hasil mungkin tidak optimal.

2. **Algoritma Hashing**

Algoritma hashing melibatkan penggunaan fungsi hash untuk mengonversi nilai kunci menjadi indeks atau alamat di dalam struktur data yang disebut tabel hash. Ini memungkinkan pencarian konstan (O(1)) jika fungsi hash dirancang dengan baik.

Contoh Implementasi Tabel Hash dengan Fungsi Hash Sederhana:

```
#include <stdio.h>

#include <stdlib.h>


#define TABLE_SIZE 10


struct Node {

   int data;

   struct Node* next;

};
```

```
struct Node* hashtable[TABLE_SIZE] = {NULL};

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void insert(int key) {
    int index = hashFunction(key);

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = hashtable[index];
    hashtable[index] = newNode;
}

int search(int key) {
    int index = hashFunction(key);

    struct Node* current = hashtable[index];
    while (current != NULL) {
        if (current->data == key) {
```

```
            return index; // Mengembalikan indeks elemen yang dicari
        }
        current = current->next;
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

int main() {
    // Contoh implementasi tabel hash
    insert(42);
    // ... tambahkan lebih banyak elemen

    int target = 42;
    int result = search(target);

    if (result != -1) {
        printf("Elemen %d ditemukan dalam tabel hash pada indeks %d.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan dalam tabel hash.\n", target);
    }
```

```
    return 0;
}
```

Algoritma hashing sangat efisien untuk pencarian jika fungsi hashnya baik dan data terdistribusi merata di seluruh tabel hash. Namun, perlu perhatian dalam penanganan tabrakan hash dan pemilihan fungsi hash yang efektif.

F. Peningkatan Kinerja Pencarian

1. Penyotiran Data

Pencarian pada data yang sudah terurut dapat diakomodasi oleh algoritma pencarian biner, yang memiliki kompleksitas waktu logaritmik ((logf_0O(logn)). Oleh karena itu, menyortir data sebelum melakukan pencarian dapat meningkatkan kinerja pencarian, terutama jika pencarian dilakukan lebih dari sekali pada data yang sama.

Contoh Implementasi Pencarian Binary setelah Penyortiran:

```
#include <stdio.h>
#include <stdlib.h>

int binarySearch(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid; // Mengembalikan indeks elemen yang dicari
        } else if (arr[mid] < target) {
```

```
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

void sortArray(int arr[], int n) {
    // Algoritma pengurutan, misalnya QuickSort atau MergeSort
    // ...
}

int main() {
    int data[] = {73, 62, 50, 42, 30, 25, 10};
    int n = sizeof(data) / sizeof(data[0]);
    int target = 42;

    // Menyortir array sebelum melakukan pencarian
    sortArray(data, n);
```

```
    int result = binarySearch(data, 0, n - 1, target);

    if (result != -1) {
        printf("Elemen %d ditemukan pada indeks %d setelah penyortiran.\n", target, result);
    } else {
        printf("Elemen %d tidak ditemukan setelah penyortiran.\n", target);
    }

    return 0;
}
```

2. Penggunaan Struktur Data yang tepat.

Penggunaan struktur data yang tepat dapat signifikan dalam meningkatkan kinerja pencarian. Misalnya, untuk pencarian cepat, Hash Table dapat memberikan kompleksitas waktu O(1) dalam kondisi tertentu.

Contoh Implementasi:

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10
```

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

struct Node* hashtable[TABLE_SIZE] = {NULL};

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void insert(int key) {
    int index = hashFunction(key);

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = hashtable[index];
    hashtable[index] = newNode;
}

int search(int key) {
    int index = hashFunction(key);
```

```
    struct Node* current = hashtable[index];
    while (current != NULL) {
        if (current->data == key) {
            return index; // Mengembalikan indeks elemen yang dicari
        }
        current = current->next;
    }

    return -1; // Mengembalikan -1 jika elemen tidak ditemukan
}

int main() {
    // Contoh implementasi pencarian dalam tabel hash
    insert(42);
    // ... tambahkan lebih banyak elemen

    int target = 42;
    int result = search(target);

    if (result != -1) {
        printf("Elemen %d ditemukan dalam tabel hash pada indeks %d.\n", target, result);
```

```
    } else {
        printf("Elemen %d tidak ditemukan dalam tabel hash.\n", target);
    }

    return 0;
}
```

3. Implementasi Algoritma Paralel.

Pencarian paralel melibatkan eksekusi beberapa operasi pencarian secara bersamaan, yang dapat meningkatkan kinerja dalam beberapa situasi. Ini dapat diterapkan dengan menggunakan konsep pemrograman paralel atau menggunakan teknologi seperti multithreading.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define ARRAY_SIZE 1000000

int parallelSearch(int arr[], int n, int target) {
    int result = -1;

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
```

```
            if (arr[i] == target) {
                #pragma omp critical
                {
                    result = i; // Menggunakan critical section untuk menghindari race condition
                }
            }
        }

        return result;
    }

    int main() {
        int data[ARRAY_SIZE];
        // ... inisialisasi array dengan data

        int target = 42;
        int result = parallelSearch(data, ARRAY_SIZE, target);

        if (result != -1) {
            printf("Elemen %d ditemukan pada indeks %d secara paralel.\n", target, result);
        } else {
```

```
        printf("Elemen %d tidak ditemukan secara paralel.\n", target);
    }

    return 0;
}
```

Catatan: Implementasi paralel dapat memerlukan manajemen thread dan sinkronisasi yang cermat untuk menghindari race condition.

Penting untuk mempertimbangkan sifat data dan kebutuhan aplikasi saat memilih strategi untuk meningkatkan kinerja pencarian. Tidak ada satu pendekatan yang cocok untuk semua situasi, dan eksperimen serta analisis kinerja dapat membantu menentukan metode yang paling efektif.

BAB II

PENUTUP

KESIMPULAN

Pencarian data di C melibatkan beberapa metode dan struktur data. Linear Search cocok untuk data tak terurut, sementara Binary Search bagus untuk data terurut. Struktur data seperti Array, Linked List, dan Binary Search Tree memiliki kelebihan dan kekurangan masing-masing. Algoritma pencarian lanjutan seperti Interpolation Search dan Hashing juga bisa digunakan. Penting untuk memilih metode yang sesuai dengan sifat data dan mempertimbangkan peningkatan kinerja seperti penyortiran data sebelumnya. Kesimpulannya, pemahaman yang baik tentang berbagai metode dan struktur data membantu pengembang membuat keputusan yang tepat dalam mengatasi pencarian data.

DAFTAR PUSTAKA

https://www.sobatambisius.com/2021/09/belajar-bahasa-c-13-searching.html

https://www.duniailkom.com/latihan-kode-program-bahasa-c-pencarian-data-array-searching/

https://www.belajarstatistik.com/blog/2022/03/15/contoh-program-pencarian-dalam-bahasa-c/

https://www.belajarstatistik.com/blog/2022/03/15/contoh-program-pencarian-dalam-bahasa-c/

https://www.kopicoding.com/linear-search-bahasa-c/