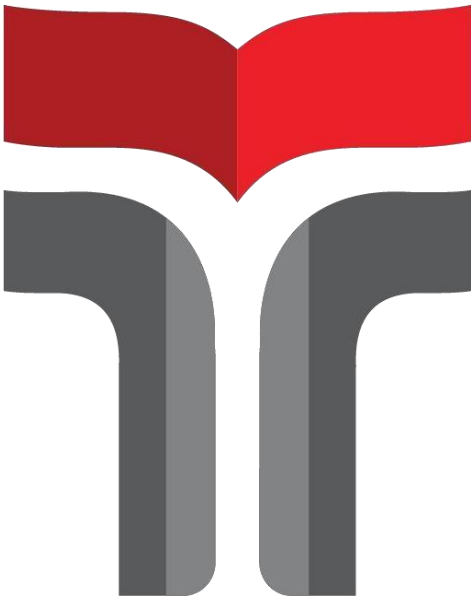
LAPORAN PRAKTIKUM
STRUKTUR DATA 6
"DOUBLE LINKED LIST"



Oleh:

NAMA: Ammar Dzaki Nandana

NIM: 2311104071

KELAS: SE 07 B

DOSEN: Wahyu Andi Saputra

PRODI S1 REKAYASA PERANGKAT LUNAK

FAKULTAS INFORMATIKA

INSTITUT TEKNOLOGI TELKOM PURWOKERTO

2023/2024

I. TUJUAN

- Memahami konsep dasar Double Linked List.
- Mengimplementasikan operasi dasar pada Double Linked List, yaitu:
- Menambah data di akhir list.
- Menghapus data dari posisi tertentu.
- Mencari data tertentu berdasarkan kriteria.
- Memahami cara kerja pointer dalam struktur data untuk menghubungkan elemen-elemen list secara dua arah.

II. DASAR TEORI

Double Linked List adalah tipe Linked List yang setiap elemennya memiliki dua pointer:

- Pointer Next: Mengarah ke elemen berikutnya.
- Pointer Prev: Mengarah ke elemen sebelumnya.

Keuntungan Double Linked List dibandingkan Single Linked List adalah kemampuannya untuk menelusuri list ke depan dan ke belakang, sehingga memudahkan manipulasi data pada posisi tertentu dalam list.

3.2 Operasi Dasar pada Double Linked List

1. Penambahan Elemen (Insert Last): Menambahkan elemen baru di akhir list.
2. Penghapusan Elemen:
    - deleteFirst: Menghapus elemen pertama.
    - deleteLast: Menghapus elemen terakhir.
    - deleteAfter: Menghapus elemen setelah elemen tertentu.
3. Pencarian Elemen (findElm): Mencari elemen di dalam list berdasarkan kriteria, misalnya nomor polisi kendaraan.

Double linked list juga menggunakan dua buah *successor* utama yang terdapat pada *list*, yaitu *first* (*successor* yang menunjuk elemen pertama) dan *last* (susesor yang menunjuk elemen terakhir *list*).
Komponen-komponen dalam *double linked list*:
1. *First* : *pointer* pada *list* yang menunjuk pada elemen pertama *list*.
2. *Last* : *pointer* pada *list* yang menunjuk pada elemen terakhir *list*.
3. *Next* : *pointer* pada elemen sebagai *successor* yang menunjuk pada elemen didepannya.
4. *Prev* : *pointer* pada elemen sebagai *successor* yang menunjuk pada elemen dibelakangnya.

III. GUIDED

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

class Node {
public:
    int data;
    Node* prev;
    Node* next;
};

class DoublyLinkedList {
public:
    Node* head;
    Node* tail;

    // Constructor untuk inisialisasi head dan tail
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
    }

    // Fungsi untuk menambahkan elemen di depan list
    void insert(int data) {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->prev = nullptr;
        newNode->next = head;

        if (head != nullptr) {
            head->prev = newNode;
        } else {
            tail = newNode; // Jika list kosong, tail juga mengarah ke node baru
        }
        head = newNode;
    }

    // Fungsi untuk menghapus elemen dari depan list
    void deleteNode() {
        if (head == nullptr) {
            return; // Jika list kosong
        }
        Node* temp = head;
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        } else {
            tail = nullptr; // Jika hanya satu elemen di list
        }
        delete temp; // Hapus elemen
    }

    // Fungsi untuk mengupdate data di list
    bool update(int oldData, int newData) {
        Node* current = head;
        while (current != nullptr) {
            if (current->data == oldData) {
                current->data = newData;
                return true; // Jika data ditemukan dan diupdate
            }
            current = current->next;
        }
        return false; // Jika data tidak ditemukan
    }

    // Fungsi untuk menghapus semua elemen di list
    void deleteAll() {
        Node* current = head;
        while (current != nullptr) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
        head = nullptr;
        tail = nullptr;
    }
```

```
    // Fungsi untuk menampilkan semua elemen di list
    void display() {
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    DoublyLinkedList list;
    while (true) {
        cout << "1. Add data" << endl;
        cout << "2. Delete data" << endl;
        cout << "3. Update data" << endl;
        cout << "4. Clear data" << endl;
        cout << "5. Display data" << endl;
        cout << "6. Exit" << endl;

        int choice;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: {
                int data;
                cout << "Enter data to add: ";
                cin >> data;
                list.insert(data);
                break;
            }
            case 2: {
                list.deleteNode();
                break;
            }
            case 3: {
                int oldData, newData;
                cout << "Enter old data: ";
                cin >> oldData;
                cout << "Enter new data: ";
                cin >> newData;
                bool updated = list.update(oldData, newData);
                if (!updated) {
                    cout << "Data not found" << endl;
                }
                break;
            }
            case 4: {
                list.deleteAll();
                break;
            }
            case 5: {
                list.display();
                break;
            }
            case 6: {
                return 0;
            }
            default: {
                cout << "Invalid choice" << endl;
                break;
            }
        }
    }
    return 0;
}
```

Kode di atas adalah implementasi dari *Double Linked List* dalam bahasa C++. Berikut adalah penjelasan rinci mengenai setiap bagian dari kode tersebut.

```
class Node {
public:
    int data;
    Node* prev;
    Node* next;
};
```

Kelas `Node` merepresentasikan satu elemen dalam *Double Linked List*, yang berisi:
- **data**: nilai yang disimpan dalam node.
- **prev**: pointer ke node sebelumnya.
- **next**: pointer ke node berikutnya.

```
class DoublyLinkedList {
public:
    Node* head;
    Node* tail;
```

Kelas `DoublyLinkedList` digunakan untuk mengelola seluruh *Double Linked List*, dengan atribut:
- **head**: pointer ke node pertama (kepala) dalam list.
- **tail**: pointer ke node terakhir (ekor) dalam list.

```
DoublyLinkedList() {
    head = nullptr;
    tail = nullptr;
}
```

Constructor ini menginisialisasi `head` dan `tail` menjadi `nullptr`, menunjukkan bahwa list kosong saat pertama kali dibuat.

```
void insert(int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->prev = nullptr;
    newNode->next = head;

    if (head != nullptr) {
        head->prev = newNode;
    } else {
        tail = newNode;
    }
    head = newNode;
```

```
}
```

Fungsi `insert` digunakan untuk menambahkan elemen baru di awal list. Prosesnya:
1. Membuat `newNode` dengan data yang diinginkan.
2. Menyetel pointer `next` dari `newNode` ke `head` (node pertama sebelumnya).
3. Jika `head` tidak `nullptr` (list tidak kosong), pointer `prev` dari `head` disetel ke `newNode`.
4. Jika list kosong (head `nullptr`), `tail` juga disetel ke `newNode`.
5. `head` di-update untuk menunjuk ke `newNode` sebagai node pertama.

```
void deleteNode() {
    if (head == nullptr) {
        return;
    }
    Node* temp = head;
    head = head->next;
    if (head != nullptr) {
        head->prev = nullptr;
    } else {
        tail = nullptr;
    }
    delete temp;
}
```

Fungsi `deleteNode` menghapus node pertama dari list:
1. Jika `head` adalah `nullptr`, list kosong, maka fungsi akan berhenti.
2. Jika ada node, `temp` menyimpan pointer ke `head` saat ini, lalu `head` di-update ke `head->next`.
3. Jika `head` baru tidak `nullptr`, pointer `prev` dari node pertama di-update menjadi `nullptr`.
4. Jika node yang dihapus adalah satu-satunya elemen (sehingga `head` menjadi `nullptr`), `tail` juga disetel `nullptr`.
5. `temp` dihapus untuk membebaskan memori.

```
bool update(int oldData, int newData) {
    Node* current = head;
    while (current != nullptr) {
        if (current->data == oldData) {
            current->data = newData;
            return true;
        }
        current = current->next;
    }
    return false;
}
```

Fungsi `update` mencari node dengan nilai `oldData` dan mengubahnya menjadi `newData`:

1. `current` berjalan dari `head` ke `tail`, mencari node dengan `data` yang cocok dengan `oldData`.
2. Jika ditemukan, `data` dari node tersebut di-update dan fungsi mengembalikan `true`.
3. Jika tidak ditemukan, fungsi mengembalikan `false`.

```
void deleteAll() {
    Node* current = head;
    while (current != nullptr) {
        Node* temp = current;
        current = current->next;
        delete temp;
    }
    head = nullptr;
    tail = nullptr;
}
```

Fungsi `deleteAll` menghapus semua node di dalam list:
1. `current` mulai dari `head` dan berjalan hingga `tail`, menghapus node satu per satu.
2. Setelah semua node dihapus, `head` dan `tail` disetel menjadi `nullptr`, menunjukkan list kosong.

```
void display() {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}
```

Fungsi `display` menampilkan semua elemen di dalam list, dimulai dari `head` hingga `tail`.

```
int main() {
    DoublyLinkedList list;
    while (true) {
        cout << "1. Add data" << endl;
        cout << "2. Delete data" << endl;
        cout << "3. Update data" << endl;
        cout << "4. Clear data" << endl;
        cout << "5. Display data" << endl;
        cout << "6. Exit" << endl;

        int choice;
        cout << "Enter your choice: ";
        cin >> choice;
```

```
        switch (choice) {
            case 1: {
                int data;
                cout << "Enter data to add: ";
                cin >> data;
                list.insert(data);
                break;
            }
            case 2: {
                list.deleteNode();
                break;
            }
            case 3: {
                int oldData, newData;
                cout << "Enter old data: ";
                cin >> oldData;
                cout << "Enter new data: ";
                cin >> newData;
                bool updated = list.update(oldData, newData);
                if (!updated) {
                    cout << "Data not found" << endl;
                }
                break;
            }
            case 4: {
                list.deleteAll();
                break;
            }
            case 5: {
                list.display();
                break;
            }
            case 6: {
                return 0;
            }
            default: {
                cout << "Invalid choice" << endl;
                break;
            }
        }
    }
    return 0;
}
```

Di dalam `main`:
1. Pilihan 1: Menambah data ke depan list.
2. Pilihan 2: Menghapus node pertama dari list.
3. Pilihan 3: Meng-update nilai node tertentu.
4. Pilihan 4: Menghapus semua node di list.
5. Pilihan 5: Menampilkan semua data dalam list.
6. Pilihan 6: Keluar dari program.

Setiap pilihan mengaktifkan fungsi terkait di `DoublyLinkedList`.

IV. UNGUIDED

```cpp
#include "doublelist.h"

// Membuat list kosong
void CreateList(List &L) {
    L.First = nullptr;
    L.Last = nullptr;
}

// Alokasi elemen baru
ElmList* alokasi(kendaraan x) {
    ElmList* P = new ElmList;
    P->info = x;
    P->next = nullptr;
    P->prev = nullptr;
    return P;
}

// Dealokasi elemen
void dealokasi(ElmList* &P) {
    delete P;
    P = nullptr;
}

// Menampilkan informasi kendaraan di list
void printInfo(const List &L) {
    ElmList* P = L.Last;
    cout << "DATA LIST 1" << endl;
    while (P != nullptr) {
        cout << "no polisi: " << P->info.nopol << endl;
        cout << "warna    : " << P->info.warna << endl;
        cout << "tahun    : " << P->info.thnBuat << endl;
        P = P->prev;
        if (P != nullptr) cout << endl;
    }
}

// Menambahkan elemen di akhir list
void insertLast(List &L, ElmList* P) {
    if (L.First == nullptr) { // List kosong
        L.First = P;
        L.Last = P;
    } else { // List tidak kosong
        L.Last->next = P;
        P->prev = L.Last;
        L.Last = P;
    }
}

```

```cpp
#include "doublelist.h"

// Membuat list kosong
void CreateList(List &L) {
    L.First = nullptr;
    L.Last = nullptr;
}

// Alokasi elemen baru
ElmList* alokasi(kendaraan x) {
    ElmList* P = new ElmList;
    P->info = x;
    P->next = nullptr;
    P->prev = nullptr;
    return P;
}

// Dealokasi elemen
void dealokasi(ElmList* &P) {
    delete P;
    P = nullptr;
}

// Menampilkan informasi kendaraan di list
void printInfo(const List &L) {
    ElmList* P = L.Last;
    cout << "DATA LIST 1" << endl;
    while (P != nullptr) {
        cout << "no polisi: " << P->info.nopol << endl;
        cout << "warna    : " << P->info.warna << endl;
        cout << "tahun    : " << P->info.thnBuat << endl;
        P = P->prev;
        if (P != nullptr) cout << endl;
    }
}

// Menambahkan elemen di akhir list
void insertLast(List &L, ElmList* P) {
    if (L.First == nullptr) { // List kosong
        L.First = P;
        L.Last = P;
    } else { // List tidak kosong
        L.Last->next = P;
        P->prev = L.Last;
        L.Last = P;
    }
}

```

```cpp
#include "doublelist.h"

// Membuat list kosong
void CreateList(List &L) {
    L.First = nullptr;
    L.Last = nullptr;
}

// Alokasi elemen baru
ElmList* alokasi(kendaraan x) {
    ElmList* P = new ElmList;
    P->info = x;
    P->next = nullptr;
    P->prev = nullptr;
    return P;
}

// Dealokasi elemen
void dealokasi(ElmList* &P) {
    delete P;
    P = nullptr;
}

// Menampilkan informasi kendaraan di list
void printInfo(const List &L) {
    ElmList* P = L.Last;
    cout << "DATA LIST 1" << endl;
    while (P != nullptr) {
        cout << "no polisi: " << P->info.nopol << endl;
        cout << "warna    : " << P->info.warna << endl;
        cout << "tahun    : " << P->info.thnBuat << endl;
        P = P->prev;
        if (P != nullptr) cout << endl;
    }
}

// Menambahkan elemen di akhir list
void insertLast(List &L, ElmList* P) {
    if (L.First == nullptr) { // List kosong
        L.First = P;
        L.Last = P;
    } else { // List tidak kosong
        L.Last->next = P;
        P->prev = L.Last;
        L.Last = P;
    }
}

```

V. KESIMPULAN

Kesimpulan dari implementasi *Double Linked List* ini adalah bahwa struktur data ini memungkinkan manipulasi data secara efisien dalam arah dua sisi—baik maju maupun mundur. Dalam penerapan ini, fungsi dasar seperti menambah, menghapus, mengupdate, dan menampilkan elemen berhasil diimplementasikan, sehingga memberikan fleksibilitas dalam pengelolaan data. Struktur data ini sangat bermanfaat ketika diperlukan navigasi bolak-balik melalui elemen-elemen list, serta memudahkan penghapusan dan penyisipan elemen di berbagai posisi dalam list. Melalui percobaan ini, kita memperoleh pemahaman yang lebih mendalam mengenai penggunaan pointer dalam mengelola memori secara dinamis, serta keuntungan menggunakan *Double Linked List* dalam situasi yang membutuhkan efisiensi dan fleksibilitas tinggi.