Data Structures

Fall 2023

---

10. Doubly Linked List

Introduction

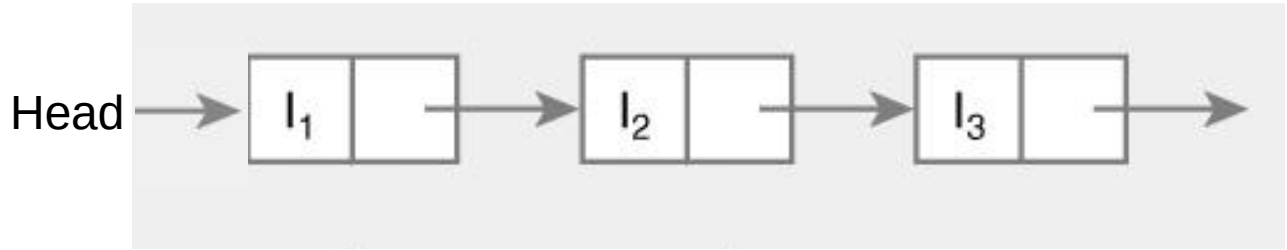
- The singly linked list contains only one pointer field i.e. every node holds an address of next node.
- The singly linked list is uni-directional i.e. we can only move from one node to its successor.
- This limitation can be overcome by **Doubly linked list.**

Doubly Linked List

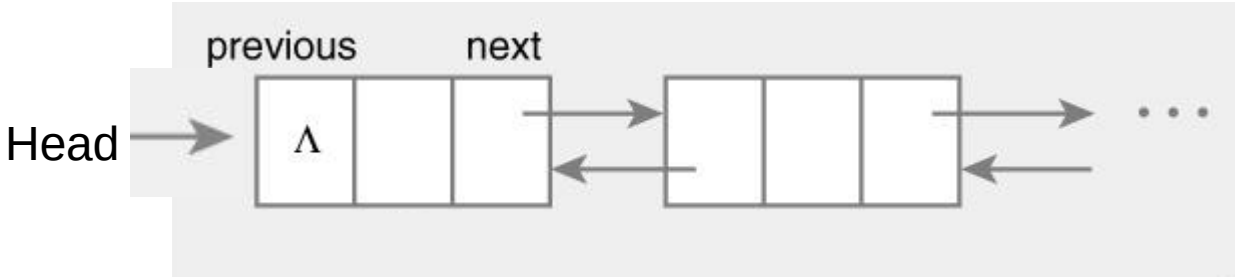
- In Doubly linked list, each node has two pointers.
- One pointer to its successor (NULL if there is none) and one pointer to its predecessor (NULL if there is none).
- These pointers enable bi-directional traversing.

Comparison of Linked List

A Singly Linked List



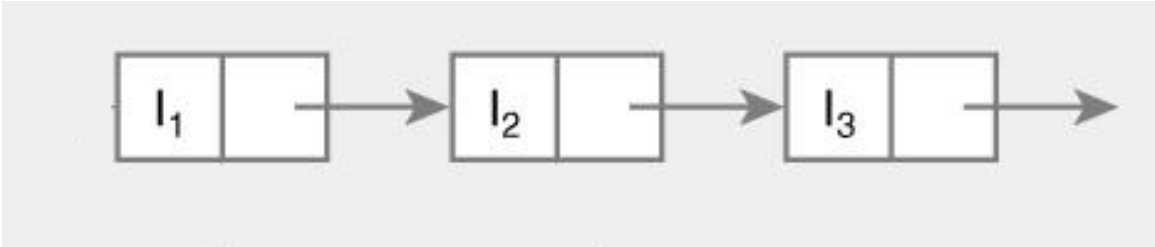
A Doubly Linked List



Comparison of Linked List

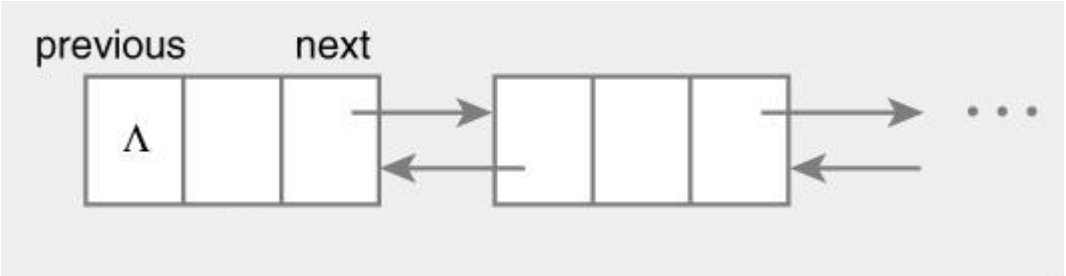
- Linked list

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



- Doubly linked list

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



Linked List Class

```
class list
{
    private:
        node *head,*tail;

    public:
        list()
        {
            head=NULL;
            tail=NULL;
        }
```

Insertion

- In insertion process, element can be inserted in three different places
  - At the beginning of the list
  - At the end of the list
  - At the specified position.

- To insert a node in doubly linked list, you must update pointers in both predecessor and successor nodes.

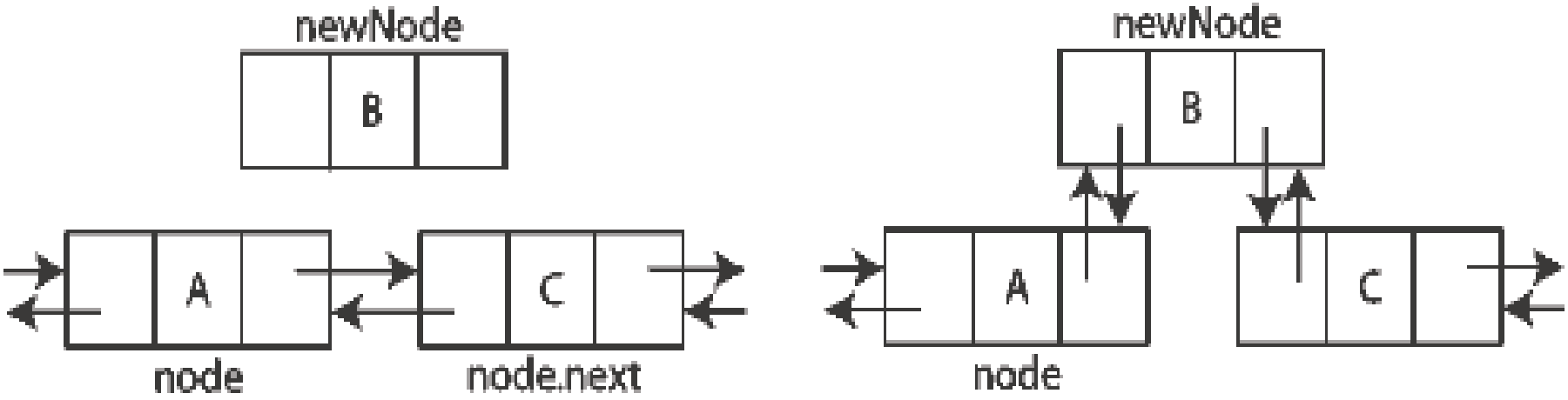
Insertion at head

```
void insert_at_head(int v)
{
    node *temp=new node;
    temp->data=v;
    temp->next=NULL;
    temp->prev=NULL;

    if(head==NULL&tail==NULL)
    {
        head=temp;
        tail=temp;
    }
    else
    {
        temp->next=head;
        head->prev=temp;
        head=temp;
    }
}
```

Insertion at end

```
void insert_at_end(int v)
{
    node *temp=new node;
    temp->data=v;
    temp->next=NULL;
    temp->prev=NULL;

    if(head==NULL&tail==NULL)
    {
        head=temp;
        tail=temp;
    }
    else
    {
        tail->next=temp;
        temp->prev=tail;
        tail=temp;
    }
}
```

Insertion at location



Insertion at location

```
void insert_at_pos(int loc,int v)
{
    node *temp=new node;
    temp->data=v;
    temp->next=NULL;
    temp->prev=NULL;

    if(head==NULL&tail==NULL)
    {
        head=temp;
        tail=temp;
    }
    else
    {
        node *p=head,*c=head;
        for(int i=1;i<loc;i++)
        {
            p=c;
            c=c->next;
        }
        p->next=temp;
        temp->prev=p;
        c->prev=temp;
        temp->next=c;
    }
}
```

Deletion

- In deletion process, element can be deleted from three different places
  - From the beginning of the list
  - From the end of the list
  - From the specified position in the list.

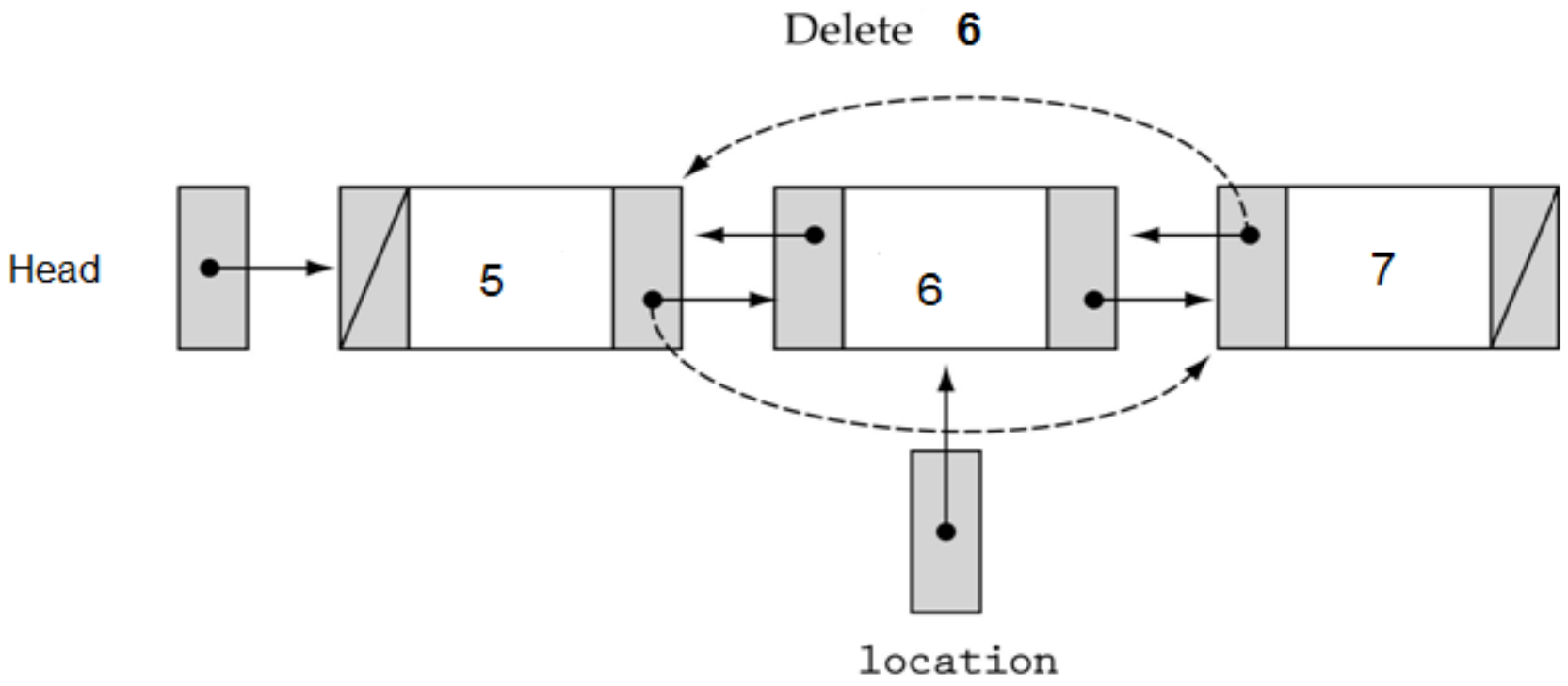
- When the node is deleted, the memory allocated to that node is released and the previous and next nodes of that node are linked

Deletion at head and tail

```
void delete_at_head()
{
    node *q=head;
    head=head->next;
    head->prev=NULL;
    q->next=NULL;
    delete q;
    q=NULL;
}
```

```
void delete_at_tail()
{
    node *q=tail;
    tail=tail->prev;
    tail->next=NULL;
    q->prev=NULL;
    delete q;
    q=NULL;
}
```

Deletion at location



Deletion at location

```
void delete_at_loc(int l)
{
    node *pre=head,*curr=head;
    for(int i=1;i<l;i++)
    {
        pre=curr;
        curr=curr->next;
    }
    pre->next=curr->next;
    curr->next->prev=pre;
    delete curr;
    curr=NULL;
}
```

Printing List

```
void display()
{
    cout<<"\nfrom head to tail"<<endl;
    node *p=head;
    while(p!=NULL)
    {
        cout<<p->data<<"\t";
        p=p->next;
    }
}
```

```
void reverse_display()
{
    cout<<"\nfrom tail to head"<<endl;
    node *q=tail;
    while(q!=NULL)
    {
        cout<<q->data<<"\t";
        q=q->prev;
    }
}
```

Advantages

- The doubly linked list is bi-directional, i.e. it can be traversed in both backward and forward direction.
- The operations such as insertion, deletion and searching can be done from both ends.
- Predecessor and successor of any element can be searched quickly

Disadvantages

- It consume more memory space.
- There is a large pointer adjustment during insertion and deletion of element.
- It consumes more time for few basic list operations.

Any Question So Far?