Link List II

Delete element from front

- Delete element from the front of link list is very easy • Step1 includes to set the head link to node 2^{nd} •

Head = temp1 (Currently)

  - Head = temp1->link;

- Delete the node from memory
- delete(temp1);

∅

Head

Delete element from front

```
struct Node {  int data;
 Node* next; };
Node* head = nullptr;

bool deleteFront()
{
 if (head == nullptr)
 {
 cout << "List is empty. Cannot delete front
element." <<endl;  return false; // Return false to
indicate failure  }

 Node* temp = head;
 head = head->next; // Update head to the next

node  delete temp; // Deallocate memory of the old

head


 return true; // Return true to indicate success
}
```


Delete element from front

- node* temp1=head;
- head = temp1->next;
- delete(temp1);
- disp();

Delete from end

- Traverse the list till n-1 node
  - Goto second last node
  - Save the address of last node in a pointer for deletion purpose
  - Assign value of NULL to pointer part of second last element

Delete element from end

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

Delete at end Code1

```
Node* head = nullptr;

bool deleteEnd() {
 if (head == nullptr) {
 cout << "List is empty. Cannot delete from the end." << endl;
return false; // Return false to indicate failure  }

 if (head->next == nullptr) {
 // Special case: Only one element in the list
delete head;
 head = nullptr;
 return true; // Return true to indicate success  }

Node* current = head;

Node* previous = nullptr;

 while (current->next != nullptr) {
 previous = current;
 current = current->next;
 }

 // Previous now points to the second-to-last node
delete current; // Deallocate memory of the last node
 previous->next = nullptr; // Update the next pointer of the second-to-last node
 return true; // Return true to indicate success
}
```

```
if (head->next==NULL){
                head=NULL;
        }
        node* previous = head;
        node* temp1=head;
        while(temp1->next!=NULL)
        {
                previous = temp1;
                temp1=temp1->next;
        }
        node* temp2 = previous->next;
        previous->next=NULL;
        delete(temp2);
```


Delete from end code 2

```
node* temp1=head;
        while(temp1->next->next!=NULL)
        {
                temp1=temp1->next;
        }
        node* temp2 = temp1->next;
        temp1->next=NULL;
        delete(temp2);
```


- Deletion in middle (**at nth point or delete by Number**)

previousNode **nodePtr** Contents of node to be deleted: 13 OR node 2 want to delete

head

NULL

5 13 19

list

- Deletion in middle (**at nth point or delete by Number**)

head

previousNode nodePtr

NULL

5 13 19

list

Home work

- Deletion in middle (**at nth point or delete by Number**)

head

previousNode nodePtr

5 19

NULL

list

Deletion in middle

```
Node* head = nullptr;

bool deleteNode(int value) {
 if (head == nullptr) {
 cout << "List is empty. Cannot delete." << endl;
return false; // Return false to indicate failure  }

 if (head->data == value) {
 // Special case: Deleting the first node
Node* temp = head;
 head = head->next;
 delete temp; // Deallocate memory of the first
node
 return true; // Return true to indicate success  }

 Node* current = head;
 Node* previous = nullptr;

 while (current != nullptr && current->data != value)
{  previous = current;
 current = current->next;
 }

 if (current == nullptr) {
 cout << "Value not found in the list. Cannot delete." << endl;
return false; // Return false to indicate failure  }

 // Bypass the node to delete it
 previous->next = current->next;
 delete current; // Deallocate memory of the node being
removed
 return true; // Return true to indicate success

}
```

Doubly Liked Lists

Definition

•A **Doubly Linked List** is a data structure where each node contains references to both the next and the previous nodes.

•Each node has three parts: **data, a reference to the next node**, and a **reference to the previous node.**

Characteristics

•Can be traversed **both ways** (forward and backward).

•More memory overhead due to an extra pointer for the previous node. **Applications**

•**Browser History:** Navigate forward and backward through pages.

•**Undo-Redo functionality:** Text editors or programs supporting undo/redo actions.

•**Music playlist navigation:** Move forward or backward between songs.

Doubly Liked Lists

- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers

forward traversal

Doubly Linked List.

next

head x_3

x_1 x_2 x_4

prev

backward traversal

Doubly Linked List

- Every node contains the address of the previous node except the first node
  - Both forward and backward traversal of the list is possible

next

a b c

head
tail prev

Node Class

- DoubleListNode class contains three data members – data: double-type data in this example
  - next: a pointer to the next node in the list
  - Prev: a pointer to the pervious node in the list

```
class DoubleListNode {
```

```
public:
    double data; // data
    DoubleListNode * next; // pointer to next
DoubleListNode * prev; // pointer to previous };
```


List Class

- List class contains two pointers
  - `head`: a pointer to the first node in the list
  - `tail`: a pointer to the last node in the list
  - Since the list is empty initially, `head` and `tail` are set to `NULL`

```
class List {
    public:
        List(void) { head = NULL; tail = NULL; } // constructor
```

```
    ~List(void); // destructor

    . . .

  private:
    DoubleListNode * head;
    DoubleListNode * tail;
};
```


Adding First Node

head

•

```
// Adding first node
```

•tail

```
head = new DoubleListNode;
head->next = null;
head->prev = null;
tail = head;
```


Inserting a Node in Doubly Linked List (1)

- To add a new item after the linked list node pointed by `current`
-

a c

•

head tail current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List

(2) • To add a new item after the linked list node pointed by `current`

a c

b

head tail

current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
```

```
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List (3)

- To add a new item after the linked list node pointed by `current`

•

a c

•

head

b tail current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
```

```
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List (3)

- To add a new item after the linked list node pointed by `current`

•

a c

•

head

b tail current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
```

```
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List

(3) • To add a new item after the linked list node pointed by

current

•

a c

•

head

b tail

current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
                                         //Current->next = newNode
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List (3)

- To add a new item after the linked list node pointed by `current`

a c

head
b tail current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```


Inserting a Node in Doubly Linked List (3)

- To add a new item after the linked list node pointed by `current`

a c

head
b tail current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c

head b

current

```
oldNode = current;
oldNode->prev->next                    =
```

```
oldNode->next;   oldNode->next->prev
=    oldNode->prev;    current    =
oldNode->prev;
delete oldNode;
```


Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c

head b

current oldNode

```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```


Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c

head b

current

oldNode

```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```


Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c

head b

current

oldNode

```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c

head b

current

oldNode

```
oldNode = current;
```

```
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```


Deleting a Node From Doubly Linked List

- Suppose `current` points to the node to be deleted from the list

a c current

head

```
oldNode = current;
oldNode->prev->next                          =
oldNode->next;   oldNode->next->prev
```

```
= oldNode->prev; current = oldNode->prev;
delete oldNode;
```


Applications of Doubly Linked List:

❖ **Browser history** for forward and backward

navigation. ❖ **Undo and redo** functionality in text

editors.

❖ **Implementation of music playlists**, enabling forward/backward song navigation.

❖ **Navigation systems** like file directories.

❖ **LRU (Least Recently Used)** cache implementation.

❖ **Deque (Double-ended queue)** implementation.

Circular Linked Lists

Definition

- A **Circular Linked List** is similar to a singly or doubly linked list, except the last node points back to the first node.
- Can be either **singly** or **doubly** circular, depending on whether there's one or two pointers per node.

Characteristics

- Circular structure (continuous looping).
- Allows continuous traversal through the list.

Applications

•**Round-robin scheduling:** Used in CPU scheduling algorithms. •**Buffer implementation:** Circular buffers in streaming data.
•**Token passing in networks:** Used in ring-based protocols for data transfer.

Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- *Solution* : Have the last node point to the first node

Circular Linked List.

head

x_1 x_2 x_n

. . .

Circular Linked List

- A linked list in which the last node points to the first node

A0 A1 A2 A3

head

Simple (singly) linked list

A0 A1 A2 A3 head

Circular linked list

Advantages of Circular Linked List

- Whole list can be traversed by starting from any point
  - Any node can be starting point
  - What is the stopping condition?

- Fewer special cases to consider during implementation
  - All nodes have a node before and after them

- Used in the implementation of other data structures
  - Circular linked lists are used to create circular queues
  - Circular doubly linked lists are used for implementing Fibonacci heaps

Disadvantages of Circular Linked List

- Finding end of list and loop control is harder
  - No NULL to mark beginning and end

Applications of Circular Linked List:

- ❖ **Round-robin scheduling** in CPU task scheduling.

❖ **Circular queues** for real-time systems.

❖ **Buffer management** in circular buffers for streaming data. ❖ **Token passing in network communication** (e.g., token ring protocol). ❖ **Game development** to manage continuous turn-taking. ❖ **Data structures for musical chairs** or ring-based applications.

Linked List – Advantages

- Access any item as long as external link to first item maintained
- Insert new item without shifting
- Delete existing item without shifting
- Can expand/contract (flexible) as necessary

Linked List – Disadvantages (1)

- Overhead of links
  - Used only internally, pure overhead

- If dynamic, must provide
  - Destructor
  - Copy constructor
  - Assignment operator

- No longer have direct access to each element of the list
  - Many sorting algorithms need direct access
  - Binary search needs direct access

- Access of n^{th} item now less efficient
  - Must go through first element, then second, and then third, etc.

Linked List – Disadvantages (2)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	

Linked List – Disadvantages (3)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List

`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = null_value

Linked List – Disadvantages (4) •

List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists

- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	**Get** a new node; **Set** data part = value next part = null_value **If** list is empty Set head to point to new node

Linked List – Disadvantages (5)

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;` This is the inefficient part	**Get** a new node; **Set** data part = value next part = null_value **If** list is empty Set head to point to new node **Else** Traverse list to find last node Set next part of last node to point to new node

Some Applications

- Applications that maintain a Most Recently Used (MRU) list
  - For example, a linked list of file names
- Cache in the browser that allows to hit the BACK button
  - A linked list of URLs
- Undo functionality in Photoshop or Word
  - A linked list of state
- A list in the GPS of the turns along your route **Can we**

Thank you