Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 3

Linked List

Doubly Linked List

Course Roadmap

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Linked Lists

2- Insertion Operation

3- Deletion Operation

4- Search Operation

5- Traverse Operation

6- Time Complexity & Space Complexity

Let's
START UP

Lecture Agenda

Section 1: Introduction to Linked Lists

Section 2: Insertion Operation

Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity

Introduction to Linked Lists

- Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last node's case a null pointer representing that there are no more nodes in the linked list.
- A linked list is a linear collection of data elements, whose order is not given by their physical placement in memory, Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing more efficient insertion or removal of nodes at arbitrary positions.
- The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation. Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.
- On the other hand, since simple linked lists by themselves do not allow random access to the data or any form of efficient indexing, many basic operations such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted may require iterating through most or all of the list elements. The advantages and disadvantages of using linked lists are given below. Linked list are dynamic, so the length of list can increase or decrease as necessary. Each node does not necessarily follow the previous one physically in the memory.

Introduction to Linked Lists

- **Advantages of Linked List:**

- Dynamic Data Structure: Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and de-allocating memory. So there is no need to give initial size of linked list.
- Insertion and Deletion: Insertion and deletion of nodes are really easier. Unlike array here we don’t have to shift elements after insertion or deletion of an element. In linked list we just have to update the address present in next pointer of a node.
- No Memory Wastage: As size of linked list can increase or decrease at run time so there is no memory wastage. In case of array there is lot of memory wastage, like if we declare an array of size 10 and store only 6 elements in it then space of 4 elements are wasted. There is no such problem in linked list as memory is allocated only when required.

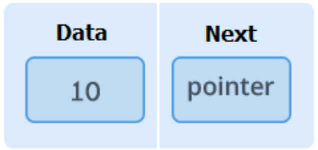
- **Disadvantages of Linked List:**

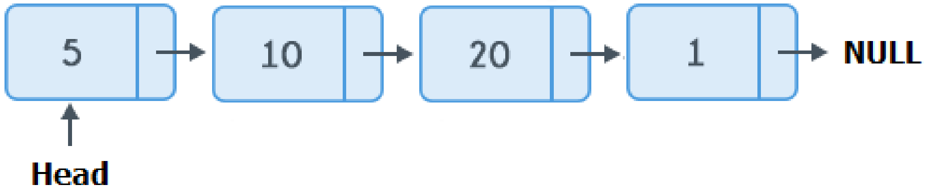
- Memory Usage: More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.
- Traversal: Elements or nodes traversal is difficult in linked list. We can not randomly access any element as we do in array by index. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large.

Introduction to Linked Lists

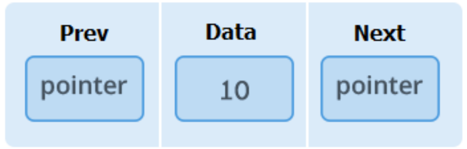
- Following are the basic operations supported by a linked list.
- Insertion: Adds an element at a specific position.
- Deletion: Removes an element at a specific position.
- Search: Searches an element using the given index or by the value.
- Traverse: Print all the array elements one by one.
- Link List Types

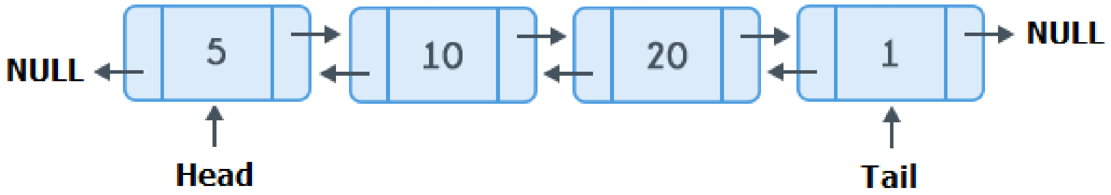
1. Single Linked List





2. Doubly Linked List





Doubly Linked List Node

- Initialize a global struct

```
#include <bits/stdc++.h>
using namespace std;

// A doubly linked list node
struct node {
    int data;
    node* prev;
    node* next;
};

// Initialize a global pointers for head and tail
node* head;
node* tail;
```

Doubly-Linked-List.cpp

Lecture Agenda

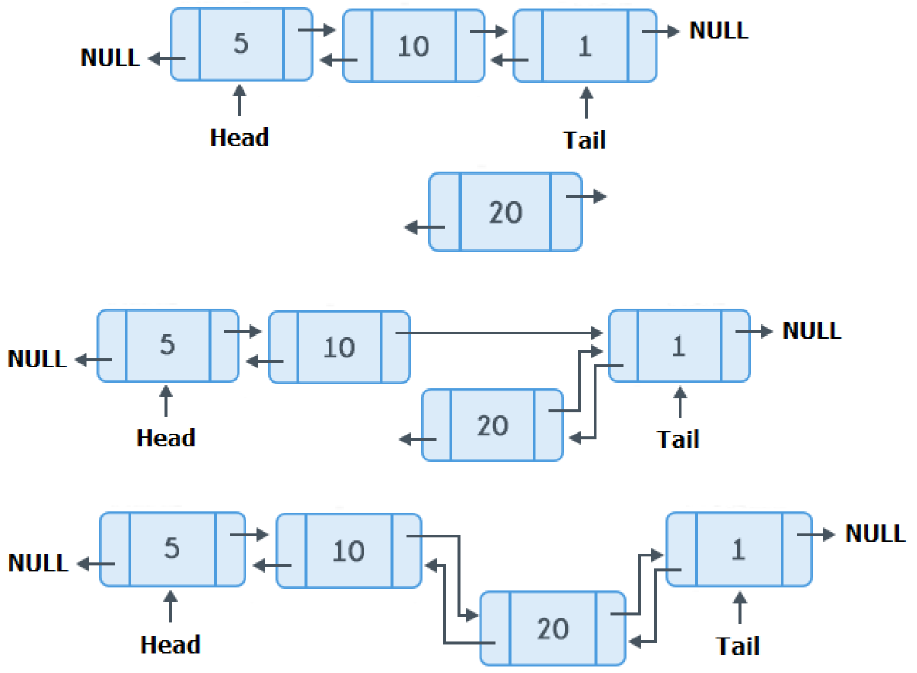
Insertion Operation - Doubly Linked List

- Insert operation is to insert one or more data elements into a linked list. Based on the requirement, a new element can be added at the beginning, end, or after any given node of linked list.

- *Insertion Algorithm*:

1. *create a new node*
2. *new node data = data*
3. *new node **next** = given node next*
4. *given node next **prev** = new node*
5. *given node **next** = new node*
6. *new node **prev** = given node*

- Insert 20 after 10



Insertion Operation - Doubly Linked List

```
// This function require a node to add the new node after it in the linked list
void insert_node(node* prev_node, int new_data) {
    // check if the given prev node is NULL
    if (prev_node == NULL)
        return;
    // allocate new node and put it's data
    node* new_node = new node();
    new_node->data = new_data;
    // set the next of the new node to be the next of the prev node
    new_node->next = prev_node->next;
    // check if the prev node is not the last node in the linked list
    if (prev_node->next != NULL)
        prev_node->next->prev = new_node;
    // move the next of the prev node to be the new node and vice versa
    prev_node->next = new_node;
    new_node->prev = prev_node;
}
```

Doubly-Linked-List.cpp

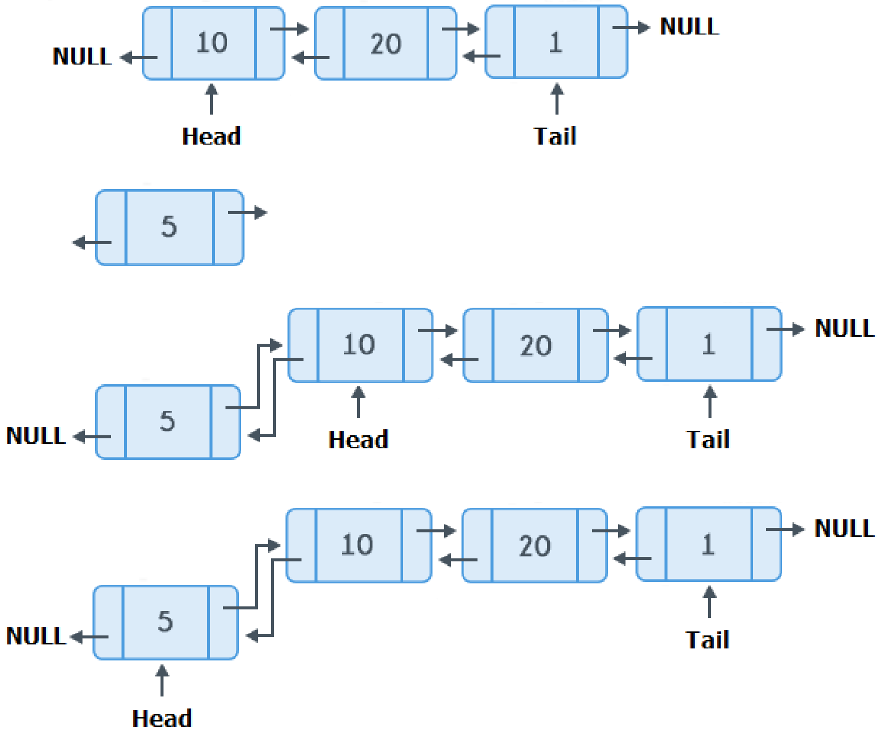
Insertion Operation - Doubly Linked List

- Insert operation is to insert one or more data elements into a linked list. Based on the requirement, a new element can be added at the beginning, end, or after any given node of linked list.

- *Insertion Algorithm*:

```
1. create a new node
2. new node data = data
3. if head == NULL then head = new node
4.                      tail = new node
5. otherwise new node next = head
6.           head prev = new node
7.           head = new node
```

- Insert at begin 5



Insertion Operation - Doubly Linked List

```
// This function inserts a node at the begin of the linked list
void insert_begin(int new_data) {
    // allocate new node and put it's data
    node* new_node = new node();
    new_node->data = new_data;
    // check if the linked list is empty
    if (head == NULL) {
        head = new_node;
        tail = new_node;
    }
    // otherwise insert the new node in the begin of the linked list
    else {
        // set next of the new node to be the head and vice versa
        new_node->next = head;
        head->prev = new_node;
        // set the new node as a head
        head = new_node;
    }
}
```

Doubly-Linked-List.cpp

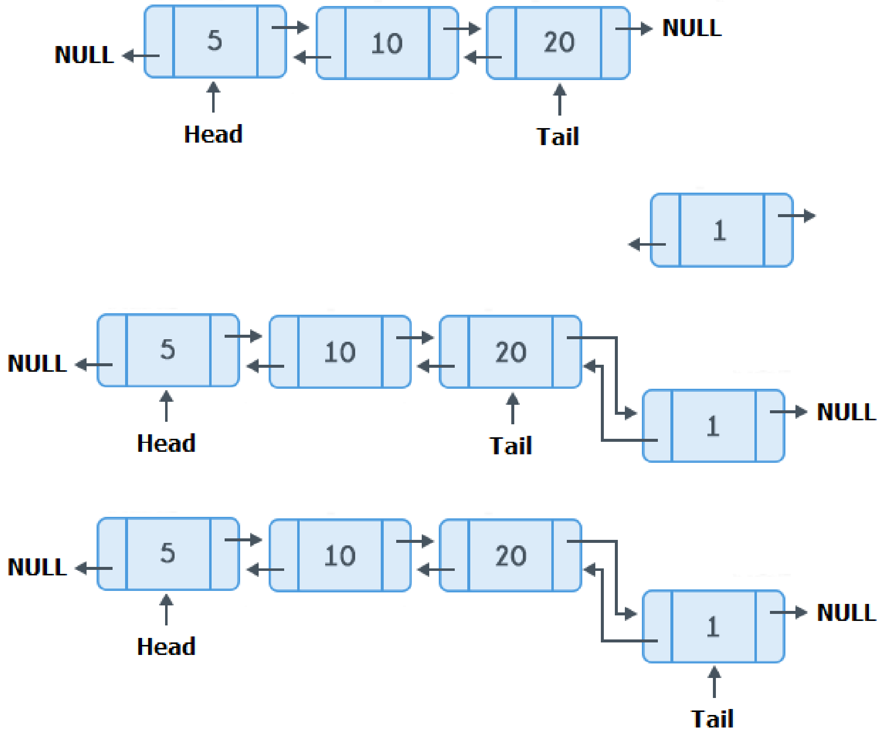
Insertion Operation - Doubly Linked List

- Insert operation is to insert one or more data elements into a linked list. Based on the requirement, a new element can be added at the beginning, end, or after any given node of linked list.

- *Insertion Algorithm*:

```
1. $create\ a\ new\ node$
2. $new\ node\ data = data$
3. $if\ head == NULL\ then\ head = new\ node$
4.                              $tail = new\ node$
5. $otherwise\ new\ node\ prev = tail$
6.                 $tail\ next = new\ node$
7.                 $tail = new\ node$
```

- Insert at end 1



Insertion Operation - Doubly Linked List

```
// This function inserts a node at the end of the linked list
void insert_end(int new_data) {
    // allocate new node and put it's data
    node* new_node = new node();
    new_node->data = new_data;
    // check if the linked list is empty
    if (head == NULL) {
        head = new_node;
        tail = new_node;
    }
    // otherwise reach the end of the linked list
    else {
        // set the next of the last node to be the new node and vice versa
        tail->next = new_node;
        new_node->prev = tail;
        // set the new node as a tail
        tail = new_node;
    }
}
```

Doubly-Linked-List.cpp

Lecture Agenda

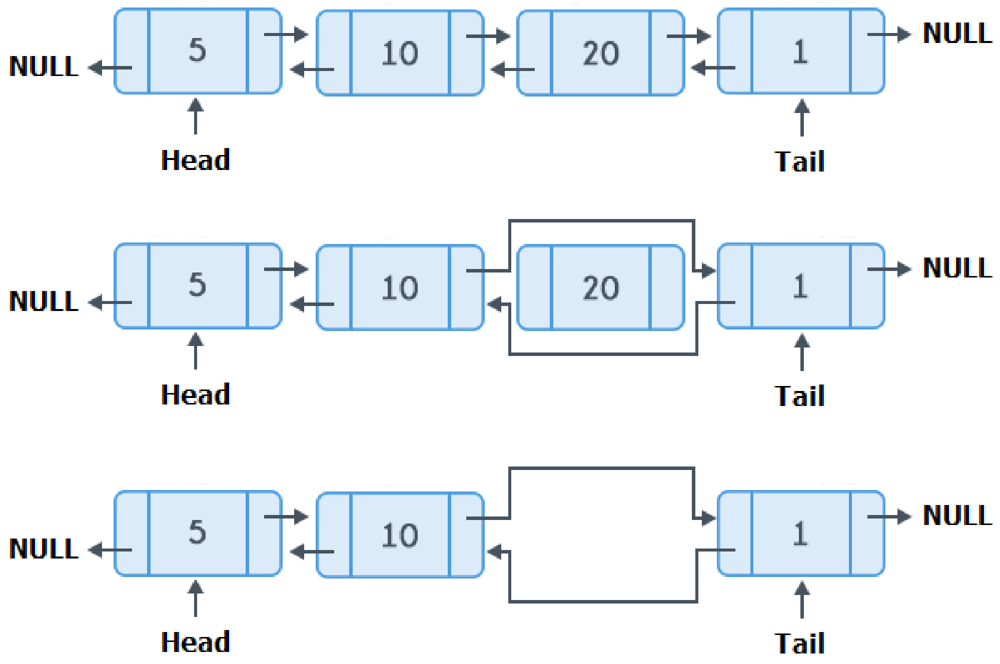
Deletion Operation - Doubly Linked List

- Delete operation removes an element from a linked list. It must be given a pointer to the element, and it then remove it out of the list by updating pointers.

- ***Deletion Algorithm***:

1. *temp node = given node **next***
2. *given node **next** = temp node next*
3. *temp node next **prev** = given node*
4. ***delete** temp node*

- Delete 20 after 10



Deletion Operation - Doubly Linked List

```
// This function require a node to delete the node after it in the linked list
void delete_node(node* prev_node) {
    // check if the given prev node is NULL
    if (prev_node == NULL || prev_node->next == NULL)
        return;
    // get the deleted node in the linked list
    node* temp_node = prev_node->next;
    // jump the deleted node
    prev_node->next = temp_node->next;
    // check if the temp node is not the last node in the linked list
    if (temp_node->next != NULL)
        temp_node->next->prev = prev_node;
    // delete the node which selected
    delete(temp_node);
}
```

Doubly-Linked-List.cpp

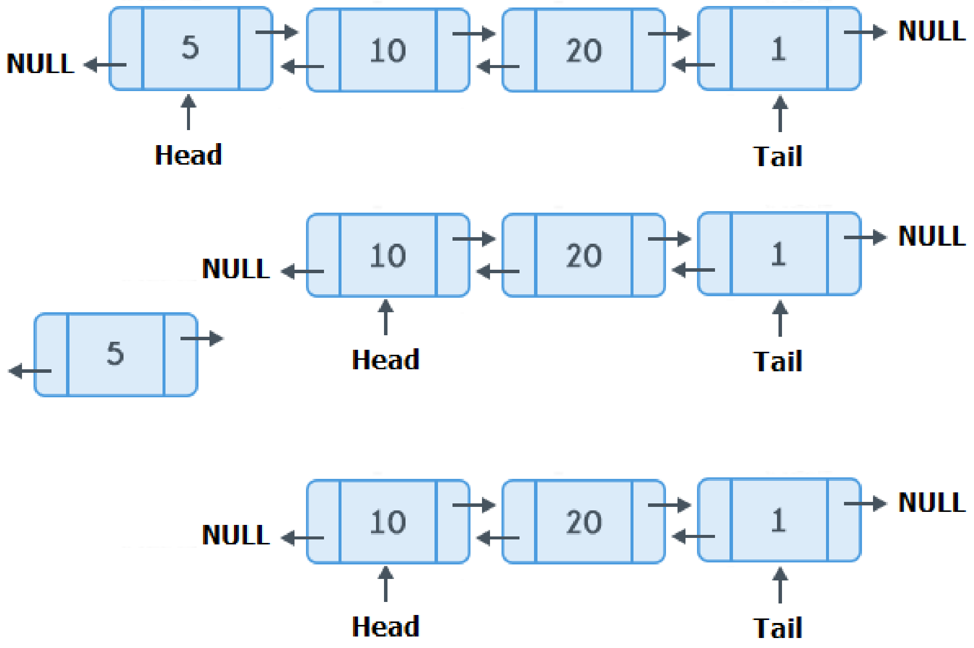
Deletion Operation - Doubly Linked List

- Delete operation removes an element from a linked list. It must be given a pointer to the element, and it then remove it out of the list by updating pointers.

- *Deletion Algorithm*:

```
1. $temp\ node = head$
2. $if\ head\ equal\ tail\ then$
3.        $delete\ temp\ node$
4.        $head = tail = NULL$
5. $otherwise\ head = head\ next$
6.                $head\ prev = NULL$
7.                $delete\ temp\ node$
```

- Delete at begin 5



Deletion Operation - Doubly Linked List

```
// This function deletes the first node in the linked lis
void delete_begin() {
    // check if the linked list is empty
    if (head == NULL)
        return;
    // get the node which it will be deleted
    node* temp_node = head;
    // check if the linked list has only one node
    if (head == tail) {
        delete(temp_node);// delete the temp node
        head = tail = NULL;
    }
    // otherwise the linked list has nodes more than one
    else {
        // shift the head to be the next node
        head = head->next;
        head->prev = NULL;
        delete(temp_node); // delete the temp node
    }
}
```

Doubly-Linked-List.cpp

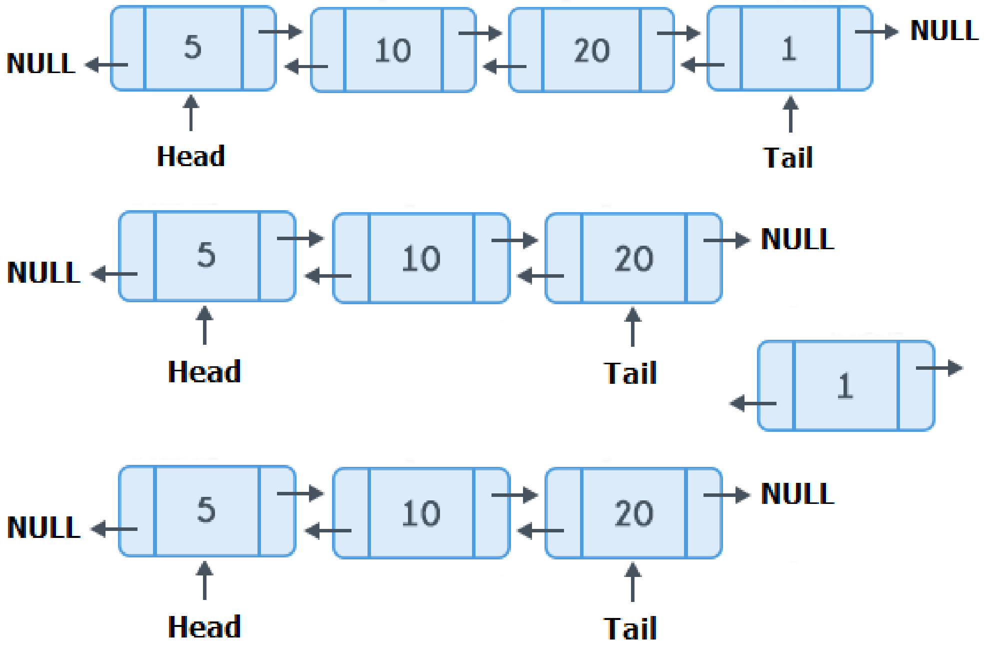
Deletion Operation - Doubly Linked List

- Delete operation removes an element from a linked list. It must be given a pointer to the element, and it then remove it out of the list by updating pointers.

- *Deletion Algorithm*:

```
1. temp node = tail
2. if head equal tail then
3.       delete temp node
4.       head = tail = NULL
5. otherwise tail = tail prev
6.               tail next = NULL
7.               delete temp node
```

- Delete at end 1



Deletion Operation - Doubly Linked List

```
// This function deletes the last node in the linked list
void delete_end() {
    // check if the linked list is empty
    if (head == NULL)
        return;
    // get the node which it will be deleted
    node* temp_node = tail;
    // check if the linked list has only one node
    if (head == tail) {
        delete(temp_node); // delete the temp node
        head = tail = NULL;
    }
    // otherwise the linked list has nodes more than one
    else {
        // jump the deleted node
        tail = tail->prev;
        tail->next = NULL;
        delete(temp_node); // delete the node which selected
    }
}
```

Doubly-Linked-List.cpp

Lecture Agenda

- ✓ Section 1: Introduction to Linked Lists
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- Section 4: Search Operation
- Section 5: Traverse Operation
- Section 6: Time Complexity & Space Complexity

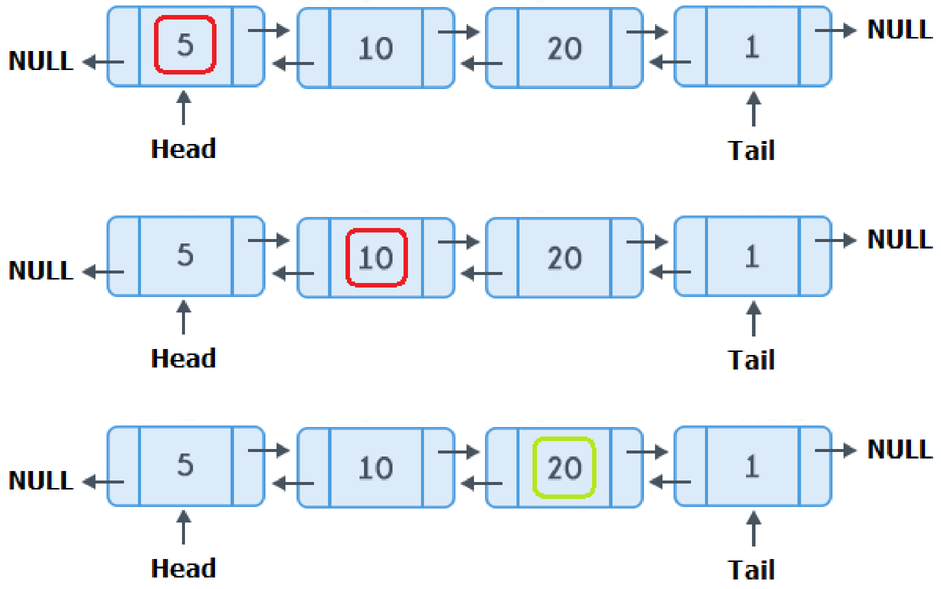
Search Operation - Doubly Linked List

- Search Operation finds the first element with key k in the linked list by a simple linear search, returning a pointer to this element. If no object with key k appears in the linked list, then the procedure returns NULL.

- *Search Algorithm*:

```
1. $curr = head$
2. $if\ curr\ data == item\ then\ item\ found$
3. $curr = curr\ next$
4. $Repeat\ Step\ 2\ if\ curr\ not\ equal\ NULL$
5. $item\ not\ found$
```

- Search for 20



Search Operation - Doubly Linked List

```
// This function searches for a node in the linked list
bool search_node(int key) {
    // iterate on the nodes till reach the last node in the linked list
    node* curr = head;
    while (curr != NULL) {
        // check if the given key exists in the linked list
        if (curr->data == key)
            return true;
        curr = curr->next;
    }
    return false;
}
```

Doubly-Linked-List.cpp

Lecture Agenda

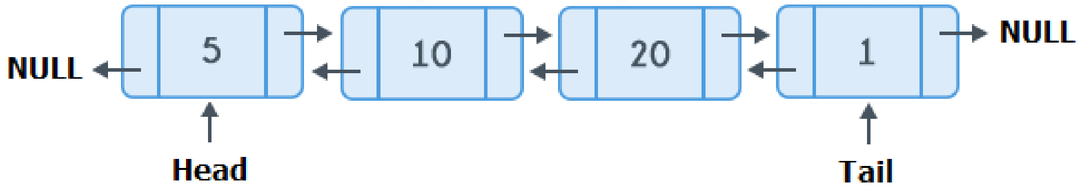
Traverse Operation - Doubly Linked List

- Traverse Operation You start at the head of the list and continue until you come across a node that is.

- *Traverse Algorithm*:

```
1. $curr = head$
2. $print\ curr\ data$
3. $curr\ = curr\ next$
4. $Repeat\ Step\ 2\ if\ curr\ not\ equal\ NULL$
```

- Traverse this linked list



Traverse Operation - Doubly Linked List

```cpp
// This function prints the contents of the linked list
void print_linked_list() {
    // print the data nodes starting from head till reach the last node
    node* curr = head;
    while (curr != NULL) {
        cout << curr->data << ' ';
        curr = curr->next;
    }
}

// This function prints the contents of the linked list
void print_linked_list_reverse() {
    // print the data nodes starting from tail till reach the first node
    node* curr = tail;
    while (curr != NULL) {
        cout << curr->data << ' ';
        curr = curr->prev;
    }
}
```

Doubly-Linked-List.cpp

Functionality Testing - Doubly Linked List

- Initialize a global struct

```cpp
#include <bits/stdc++.h>
using namespace std;

// A doubly linked list node
struct node {
     int data;
     node* prev;
     node* next;
};

// Initialize a global pointers for head and tail
node* head;
node* tail;
```

- In the Main function:

```cpp
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

- Expected Output:

```
Linked List items forward:
Linked List items backward:
```

Doubly-Linked-List.cpp

Functionality Testing - Doubly Linked List

➢ In the Main function:

```
cout << "adding the following elements 10 20 30 40 50\n";
insert_end(10);
insert_end(20);
insert_end(30);
insert_end(40);
insert_end(50);
cout << "the above elements have been added to the linked list\n";
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

➢ Expected Output:

```
adding the following elements 10 20 30 40 50
the above elements have been added to the linked list
Linked List items forward:  10 20 30 40 50
Linked List items backward: 50 40 30 20 10
```

Doubly-Linked-List.cpp

Functionality Testing - Doubly Linked List

- In the Main function:

```
cout << "add element 60 at the end of the linked list\n";
insert_end(60);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
cout << "add element 20 at the begin of the linked list\n";
insert_begin(20);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

- Expected Output:

```
add element 60 at the end of the linked list
Linked List items forward:  10 20 30 40 50 60
Linked List items backward: 60 50 40 30 20 10
add element 20 at the begin of the linked list
Linked List items forward:  20 10 20 30 40 50 60
Linked List items backward: 60 50 40 30 20 10 20
```

Functionality Testing - Doubly Linked List

➢ In the Main function:

```
cout << "add element 70 at position 4 : \n";
insert_node(head->next->next->next, 70);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
cout << "add element 90 at position 7 : \n";
insert_node(head->next->next->next->next->next->next, 90);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

➢ Expected Output:

```
add element 70 at position 4 :
Linked List items forward:  20 10 20 30 70 40 50 60
Linked List items backward: 60 50 40 70 30 20 10 20
add element 90 at position 7 :
Linked List items forward:  20 10 20 30 70 40 50 90 60
Linked List items backward: 60 90 50 40 70 30 20 10 20
```

Functionality Testing - Doubly Linked List

- In the Main function:

```
cout << "delete the first element \n";
delete_begin();
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
cout << "delete the last element \n";
delete_end();
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

- Expected Output:

```
delete the first element
Linked List items forward:  10 20 30 70 40 50 90 60
Linked List items backward: 60 90 50 40 70 30 20 10
delete the last element
Linked List items forward:  10 20 30 70 40 50 90
Linked List items backward: 90 50 40 70 30 20 10
```

Functionality Testing - Doubly Linked List

➢ In the Main function:

```
cout << "delete element at position 3 : \n";
delete_node(head->next->next);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
cout << "delete element at position 2 : \n";
delete_node(head->next);
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

➢ Expected Output:

```
delete element at position 3 :
Linked List items forward:  10 20 30 40 50 90
Linked List items backward: 90 50 40 30 20 10
delete element at position 2 :
Linked List items forward:  10 20 40 50 90
Linked List items backward: 90 50 40 20 10
```

Functionality Testing - Doubly Linked List

- In the Main function:

```
if (search_node(40))
    cout << "element " << 40 << " in the linked list\n";
else
    cout << "element " << 40 << " not in the linked list\n";

if (search_node(100))
    cout << "element " << 100 << " in the linked list\n";
else
    cout << "element " << 100 << " not in the linked list\n";
```

- Expected Output:

```
element 40 in the linked list
element 100 not in the linked list
```

- Linked List Diagram:

```
Linked List items: 10 20 40 50 90
```

Doubly-Linked-List.cpp

Functionality Testing - Doubly Linked List

- In the Main function:

```
cout << "deleting the following elements 10 20 40 50 90\n";
delete_end();
delete_end();
delete_end();
delete_end();
delete_end();
cout << "the above elements have been deleted from the linked list\n";
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

- Expected Output:

```
deleting the following elements 10 20 40 50 90
the above elements have been deleted from the linked list
Linked List items forward:
Linked List items backward:
```

Functionality Testing - Doubly Linked List

- In the Main function:

```
cout << "adding the following elements 30 20 10\n";
insert_begin(10);
insert_begin(20);
insert_begin(30);
cout << "the above elements have been added to the linked list\n";
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

- Expected Output:

```
adding the following elements 30 20 10
the above elements have been added to the linked list
Linked List items forward:  30 20 10
Linked List items backward: 10 20 30
```

Functionality Testing - Doubly Linked List

- In the Main function:

```
cout << "deleting the following elements 30 20 10\n";
delete_begin();
delete_begin();
delete_begin();
cout << "the above elements have been deleted from the linked list\n";
cout << "Linked List items forward:  ";
print_linked_list();
cout << '\n';
cout << "Linked List items backward: ";
print_linked_list_reverse();
cout << '\n';
```

Doubly-Linked-List.cpp

- Expected Output:

```
deleting the following elements 30 20 10
the above elements have been deleted from the linked list
Linked List items forward:
Linked List items backward:
```

Lecture Agenda

- ✓ Section 1: Introduction to Linked Lists
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- ✓ Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity

Time Complexity & Space Complexity

➢ Time Analysis

	Worst Case	Average Case
• Insert at the begin	Θ(1)	θ(1)
• Insert at the end	Θ(1)	θ(1)
• Insert at specific position	Θ(n)	θ(n)
• Delete at the begin	Θ(1)	θ(1)
• Delete at the end	Θ(1)	θ(1)
• Delete at specific position	Θ(n)	θ(n)
• Search	Θ(n)	θ(n)
• Traverse	Θ(n)	θ(n)

Lecture Agenda

- ✓ Section 1: Introduction to Linked Lists
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Search Operation
- ✓ Section 5: Traverse Operation
- ✓ Section 6: Time Complexity & Space Complexity

Practice

Practice

1- Count number of nodes in linked list by iterative/recursive way
2- Print linked list by iterative/recursive way
3- Print linked list in reversed order
4- Print element at middle of linked list
5- Print element at position i in linked list
6- Insert element at position i in linked list
7- Delete element at position i in linked list
8- Get node at position i in linked list
9- Delete all elements in linked list by iterative/recursive way
10- Search node in linked list by iterative/recursive way
11- Swap two nodes in linked list by index
12- Reverse linked list
13- Check if linked list has loop or not
14- Find length of loop in linked list
15- Count number of times which element occurs in linked list by iterative/recursive way

Practice

16- Remove duplication of nodes in sorted linked list
17- Remove duplication of nodes in unsorted linked list
18- Intersection of two sorted linked list
19- Intersection of two unsorted linked list
20- Union of two sorted linked list
21- Union of two unsorted linked list
22- Difference of two sorted linked list
23- Difference of two unsorted linked list
24- Segregate even and odd nodes in a linked list
25- Check if linked list is palindrome or not

Assignment

Implement STL List

- Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions. List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.
- Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Compared to other base standard sequence containers (array, and vector), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these.

More Info: cplusplus.com/reference/list/list/
More Info: en.cppreference.com/w/cpp/container/list
More Info: geeksforgeeks.org/list-cpp-stl/

Implement STL List

- Member functions:
  - (constructor) Construct vector (public member function)
  - (destructor) List destructor (public member function)
  - (operator=) Assign content (public member function)
- Iterators:
  - (begin) Return iterator to beginning (public member function)
  - (end) Return iterator to end (public member function)
  - (rbegin) Return reverse iterator to reverse beginning (public member function)
  - (rend) Return reverse iterator to reverse end (public member function)
  - (cbegin) Return const_iterator to beginning (public member function)
  - (cend) Return const_iterator to end (public member function)
  - (crbegin) Return const_reverse_iterator to reverse beginning (public member function)
  - (crend) Return const_reverse_iterator to reverse end (public member function)
- Element access:
  - (front) Access first element (public member function)
  - (back) Access last element (public member function)

Implement STL List

- Capacity:
  - (empty) Test whether vector is empty (public member function)
  - (size) Return size (public member function)
  - (max_size) Return maximum size (public member function)
- Modifiers:
  - (assign) Assign new content to container (public member function)
  - (push_front) Insert element at beginning (public member function)
  - (pop_front) Delete first element (public member function)
- Operations:
  - (unique) Remove duplicate values (public member function)
  - (merge) Merge sorted lists (public member function)
  - (sort) Sort elements in container (public member function)
  - (reverse) Reverse the order of elements (public member function)
  - (splice) Transfer elements from list to list (public member function)
  - (remove) Remove elements with specific value (public member function)

Implement STL List

- Modifiers:
  - (push_back) Add element at the end (public member function)
  - (pop_back) Delete last element (public member function)
  - (emplace) Construct and insert element (public member function)
  - (insert) Insert elements (public member function)
  - (erase) Erase elements (public member function)
  - (swap) Swap content (public member function)
  - (resize) Change size (public member function)
  - (clear) Clear content (public member function)

More Info: cplusplus.com/reference/list/list/
More Info: en.cppreference.com/w/cpp/container/list

DO
MORE.