Chapter 4 Data Structures LinkedList

By: Dr. Aryaf A. Al-adwan

Faculty of Engineering Technology

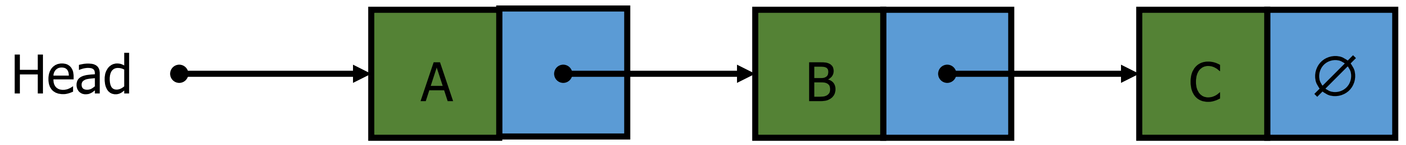
Computer and Networks Engineering Dept.

Data Structures Course

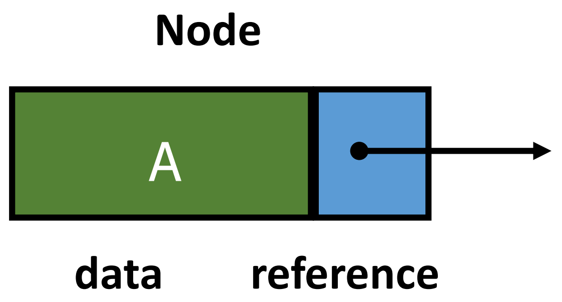
List Outline

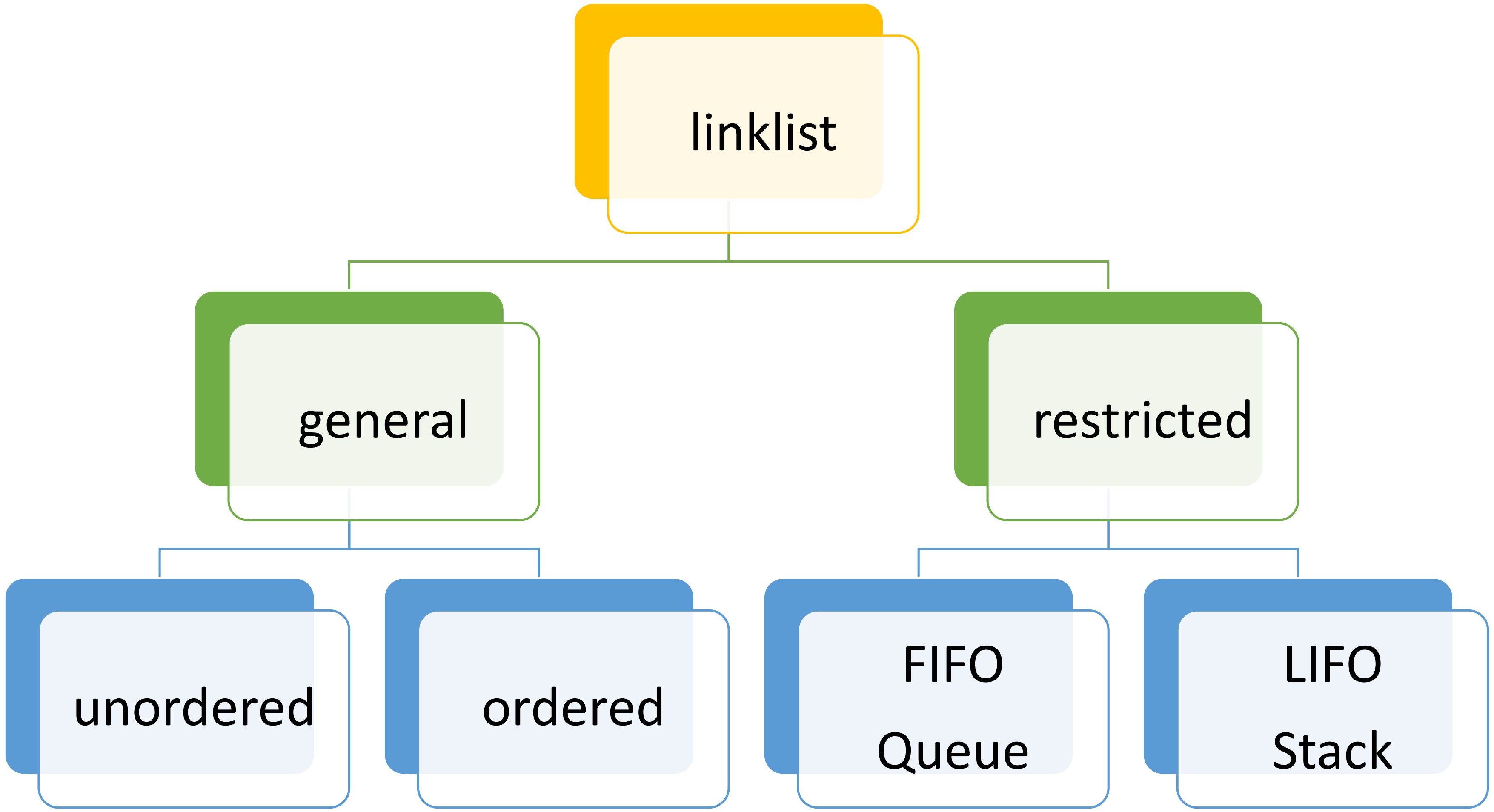
- Linked lists
    - Abstract data type (ADT)
- Basic operations of linked lists
    - Insert, find, delete, print, etc.
- Variations of linked lists
    - Ordered Linked lists
    - Circular linked lists
    - Doubly linked lists

Singly Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - reference to the next node in the list
- *Head*: reference to the first node
- Each node has link to its **successor** except last node
- The last node reference to `NULL`




linklist
general
restricted
unordered
ordered
FIFO
Queue
LIFO
Stack

Unordered Singly Linkedlist

Node

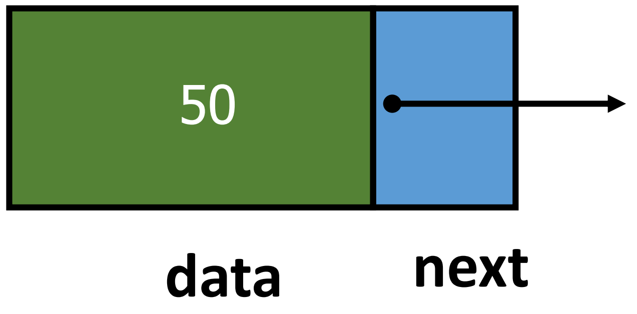
- Each Node of a linklist contains:

1. Data
2. Pointer to the next node

```
template <class T>
struct Node
{
T data;
Node <T> *next;
};
Node <T> *head;
```

head: a pointer to the first node in the list.
Since the list is empty initially, head is set to NULL

Node



Operations of SinglyLinkedList

1. **IsEmpty:** determine whether or not the list is empty
2. **Insert:**
   - insert a new node into empty list.
   - insert a new node to the end (append).
   - insert a new node at the beginning of the list.
   - insert a new node after a particular position (middle).
3. **Delete:** delete a node with a given value
   - Delete from the beginning of the list
   - Delete from middle and end of the list
4. **Display:** print all the nodes in the list
5. **Count** the number of elements in the list
6. **Searching** the list
7. **Displaying** the list

isEmpty

- Initially in the constructor
- Assign a NULL value to the head pointer

```
bool isEmpty()
{
if(head==NULL)
return true;
else
return false;
}
```

Insertion Steps

- Basic Algorithm:

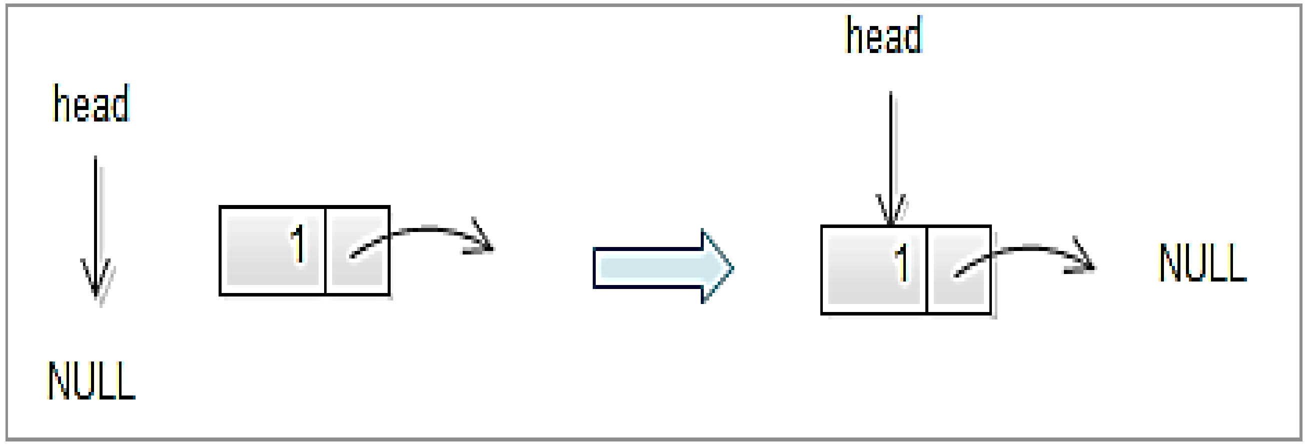
1. Check if the link list is empty or not
2. Allocate memory for the new node
3. Insert data into the new node
4. Point the next pointer of the new node to its successor

Inserting a new node

- Possible cases of `Insert` method
  1. Insert into an empty list
  2. Insert at the end (append)
  3. Add_as_first: insert at the beginning of the list
  4. Add_after: insert after a particular position

Insert Method

1) Insertion into empty linklist:

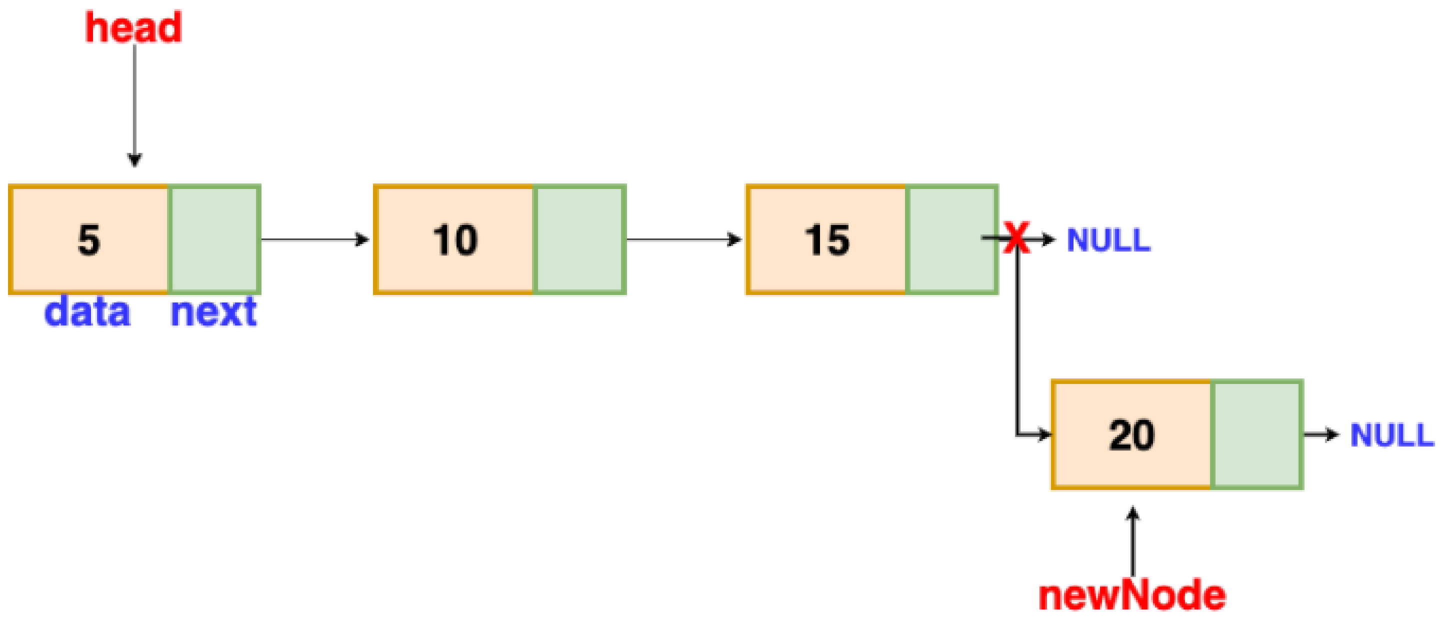


Add a node into an empty linked list

```
// insert into empty list
void insert(T num)
{
if( head == NULL )
 {
 head = new node<T>;
 head->data = num;
 head->next= NULL;
 }
```

Insert Method

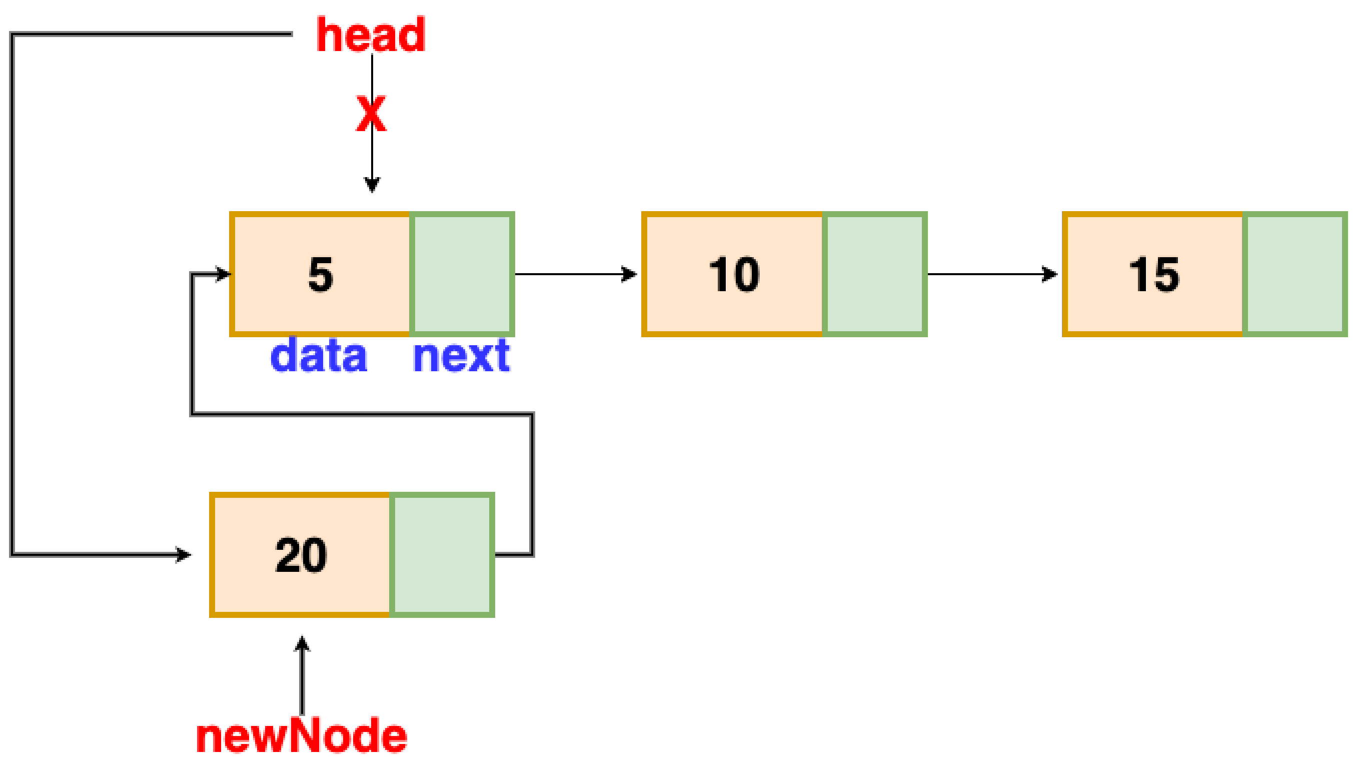
2) Insertion at the end (append):



Insertion at the end

```
else            // append
 {
 q = head;
 while( q->next!= NULL )
 q = q-> next;
 t = new node<T>;
 t->data = num;
 q-> next = t;
 t-> next = NULL;
}
}
```

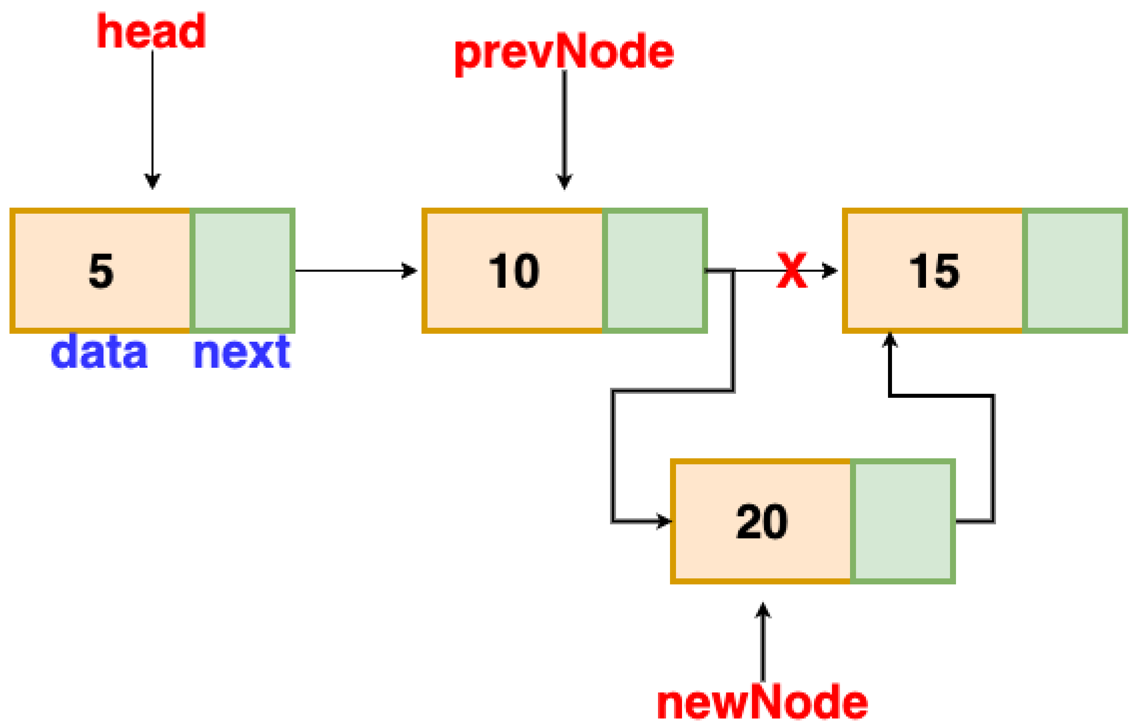
AddAsFirst method



Insertion at the beginning

```
void add_as_first(T num)
{
 node <T>*q;
 q = new node<T>;
 q->data = num;
 q->next= head;
 head = q;
}
```

AddAfter Method



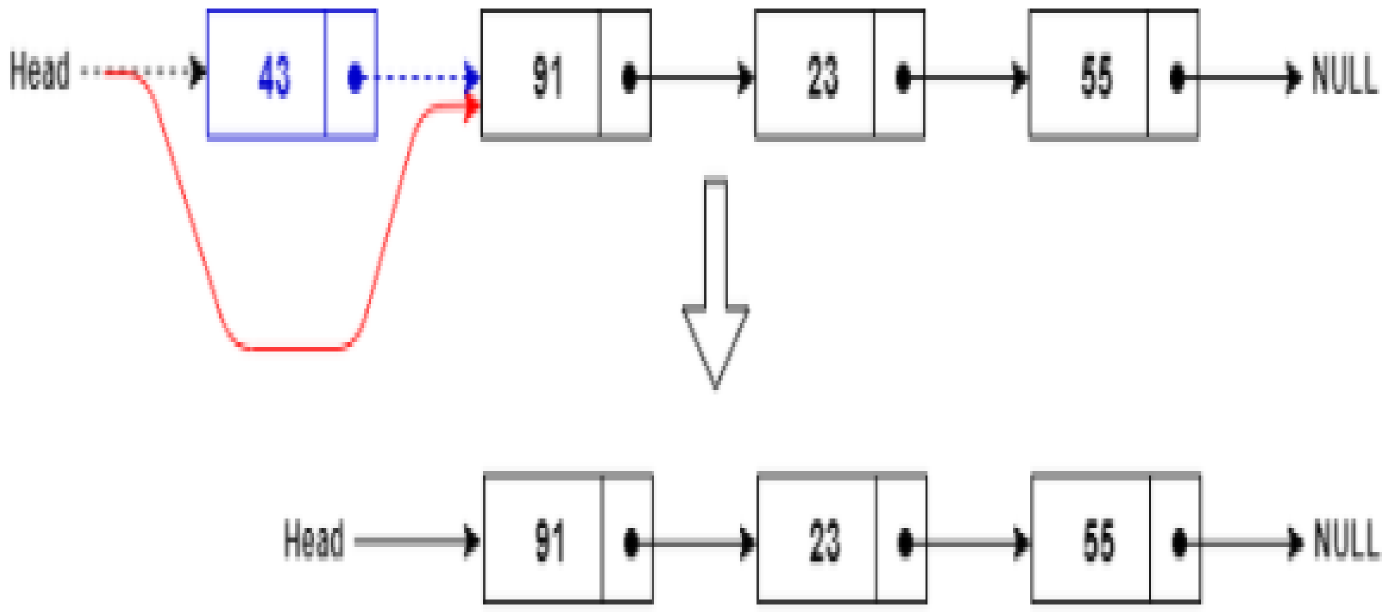
Insertion after a given node

```
void addafter( T c, T num)
{
 node <T>*q,*t;
 for(int i=1; q=head; i<c; i++)
        q = q->next;
 t = new node<T>;
 t->data = num;
 t->link = q->link;
 q->link = t;
}
```

Deleting a node

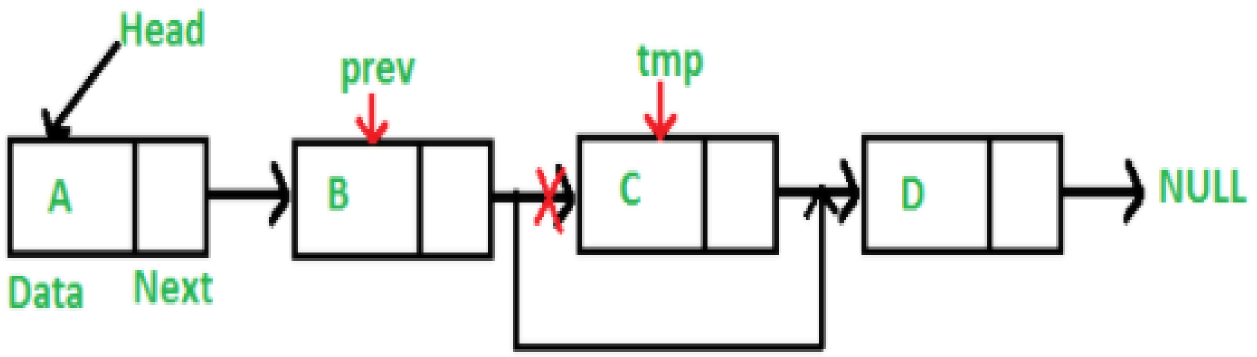
- Delete a node with the value equal to `num` from the list.
  - If such a node is found, return its position. Otherwise, return null.
- Possible cases of Delete method
  1. Delete from the beginning of the list (first node)
  2. Delete from middle or end of the list
- Steps
  - Find the desirable node
  - Set the reference of the node to null
  - Set the reference of the predecessor of the found node to the successor of the found node

Deleting from the beginning of the Linkedlist



```
T  del ( T num )
{
// delete from the beginning
 node <T>*q,*r;
 q = head;
 if( q->data == num )
 {
 head = q->link;
 delete q;
 return 0;
 }
```

Deleting from the middle and end of the Linkedlist



```
r = q;
 while( q!=NULL )
   {
        if( q->data == num )
          {
           r->link = q->link;
           delete q;
           return 0;
           }
     r = q;
    q = q->link;
   }
cout<<"\nElement "<<num<<" not Found.";
}
```

Printing the linkedlist

```
void print()
{
 node <T>*q;
 for ( q=head; q!=NULL; q=q->next)
       cout<<q->data<<endl;
}
```

```
// unordered singly link list implementation using templates
// programmed by Dr. Aryaf Aladwan
#include <iostream>
using namespace std;
template <class T>
class unorderedlinklist
{
 private:
 template <class T>
 struct node
 {
 T data;
 node <T> *link;
} ;
node <T> *head;
 public:
unorderedlinklist();
 void insert( T num );
 void add_as_first( T num );
 void addafter( T c, T num );
 T del( T num );
 void display();
 T count();
};
```

```
template <class T>
unorderedlinklist<T>::linklist()
{
 head = NULL;
}
template <class T>
void unorderedlinklist<T>::insert(T num)
{
 node <T>*q,*t;
 if( head == NULL ) // insert into empty list
 {
 head = new node<T>;
 head->data = num;
 head->link = NULL;
 }
 else // append
 {
 q = head;
 while( q->link != NULL )
 q = q->link;
 t = new node<T>;
 t->data = num;
 q->link= t;
 t->link = NULL;
 }
}
```

```
template <class T>
void unorderedlinklist<T>::add_as_first(T num) // insert in the beginning
{
 node <T>*q;
 q = new node<T>;
 q->data = num;
 q->link = head;
 head = q;
}
template <class T>
void unorderedlinklist<T>::addafter( T c, T num) // insert in the middle
{
 node <T>*q,*t;
 int i;
 for(i=1,q=head;i<c;i++)
 {
 q = q->link;
 if( q == NULL )
 {
 cout<<"\nThere are less than "<<c<<" elements.";
 return;
 }
 }
```

```
t = new node<T>;
 t->data = num;
 t->link = q->link;
 q->link = t;
}
```

```
template <class T>
T unorderedlinklist<T>::del( T num )
{
 node <T>*q,*r;
 q = head;
 if( q->data == num ) // delete from the beginning
 {
 head = q->link;
 delete q;
 return 0;
 }
 r = q;
 while( q!=NULL )
 {
 if( q->data == num )
 {
 r->link = q->link;
 delete q;
 return 0;
 }
 r = q;
 q = q->link;
 }
 cout<<"\nElement "<<num<<" not Found.";
}
```

```
void main()
{
 unorderedlinklist <int> ll;
ll.insert(12);
 ll.insert(13);
 ll.insert(23);
 ll.insert(43);
 ll.insert(44);
 ll.insert(50);
 ll.add_as_first(2);
 ll.add_as_first(111);
 ll.addafter(2,333);
 ll.addafter(6,666);
 ll.display();
ll.del(333);
 ll.del(12);
 ll.del(98);
ll.display();
}
```

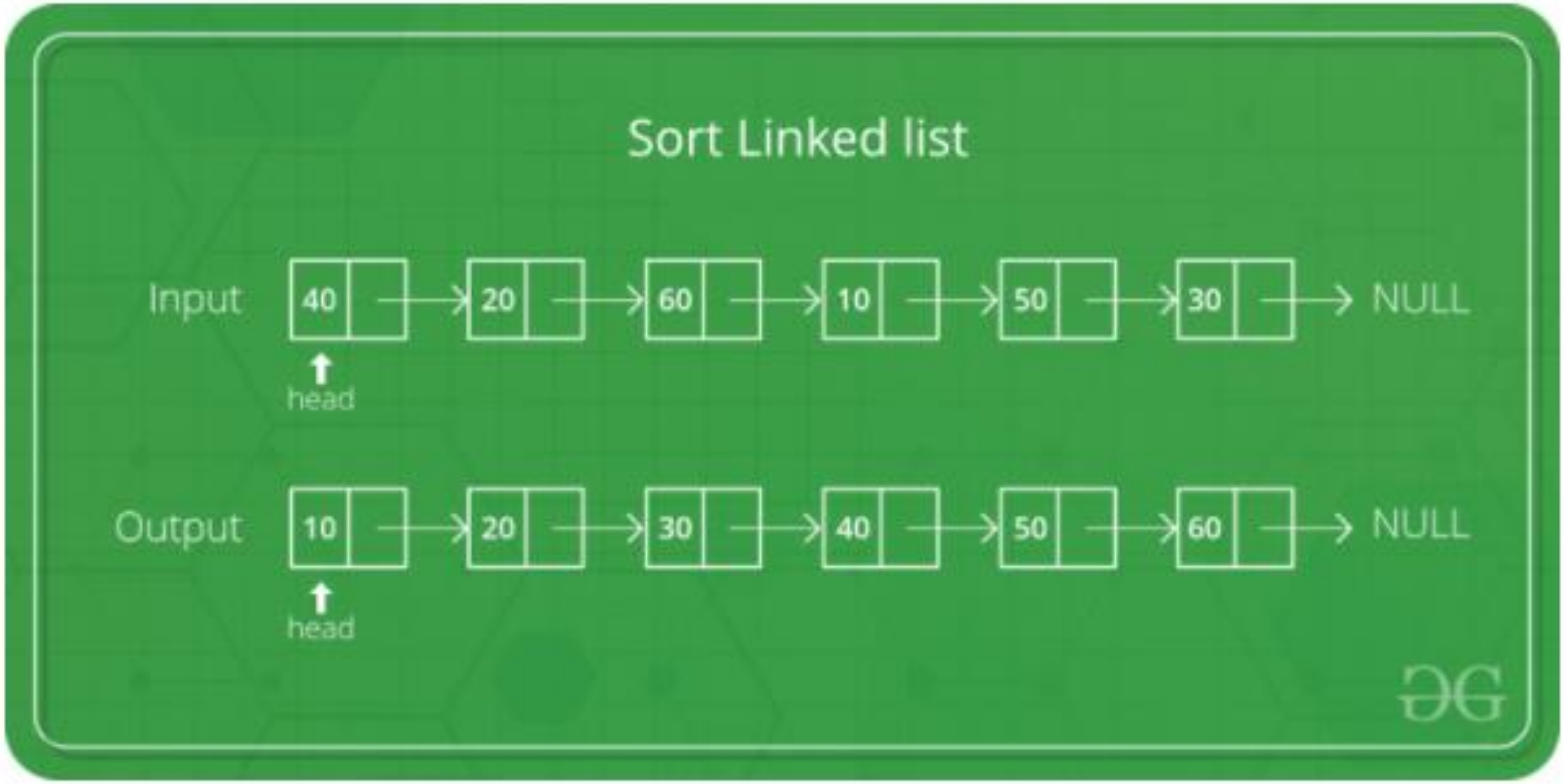
Ordered Linkedlist

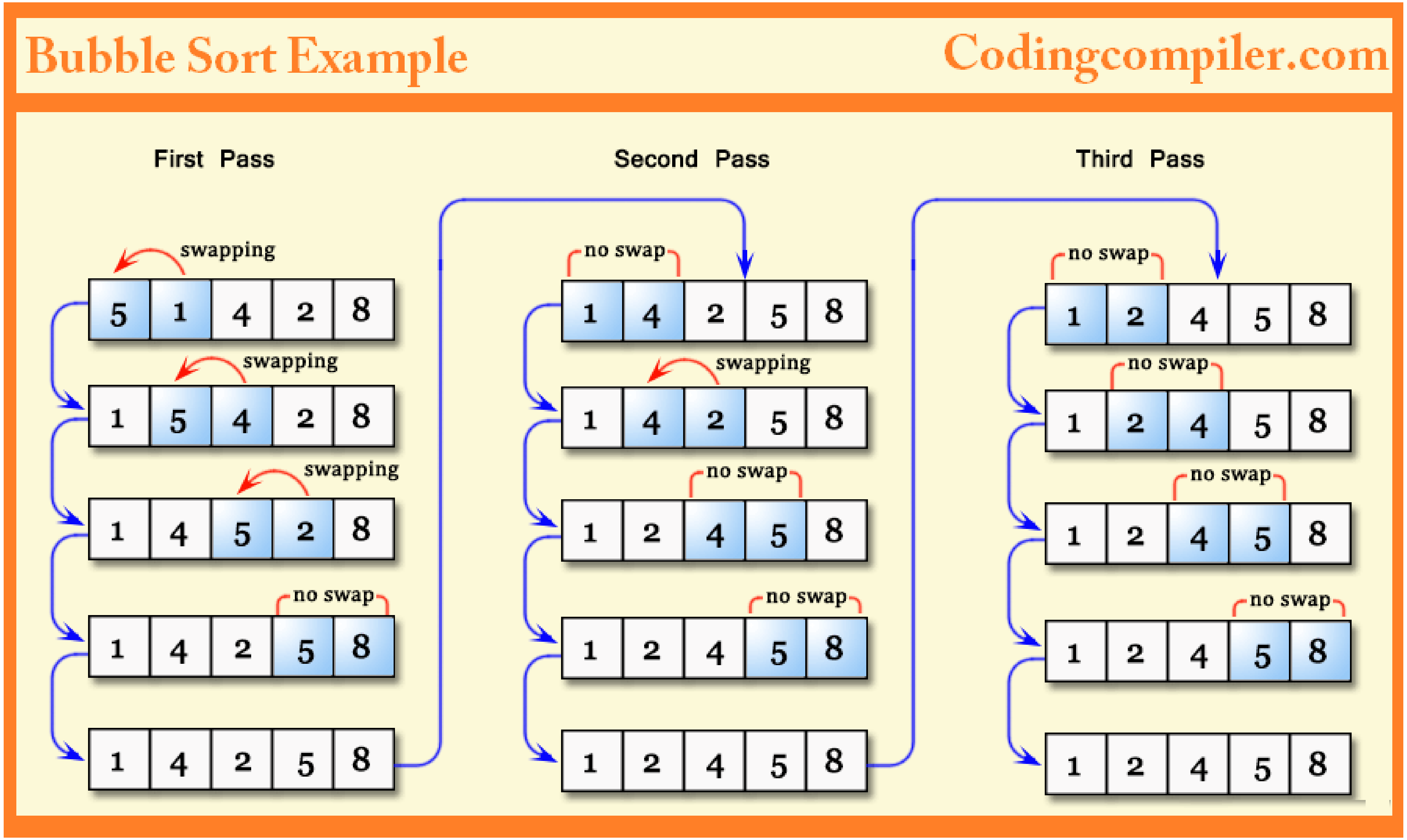
Ordered LinkedList

- Ordered singly linked list has the elements sorted in ascending or descending way.
- This kind of link lists can be created using two ways:
  1. Post creation: where the unordered link list is created, then a sorting function can be used to sort the elements of this link list.
  2. Upon creation: where the link list is created in an ordered fashion.

Post creation

- Create the linkedlist then sort it using any sorting algorithm




Bubble Sort Example
Codingcompiler.com
First Pass
Second Pass
Third Pass
swapping
swapping
swapping
no swap
5 1 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
no swap
swapping
no swap
no swap
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
no swap
no swap
no swap
no swap
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8

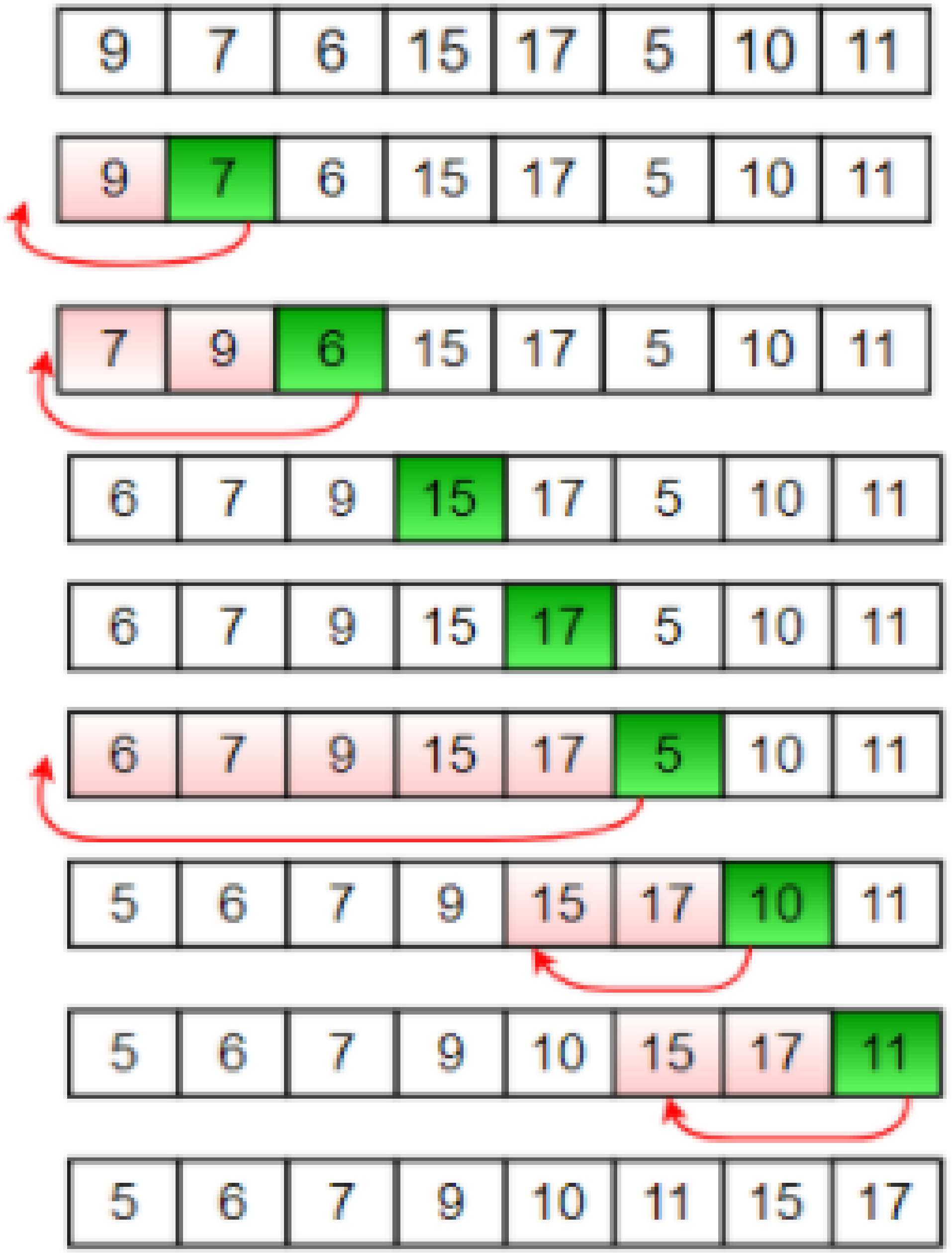
```
// Bubble Sort on array data structure
void main ()
{
int temp;
int a[10] = {10,2,0,14,43,25,18,1,5,45};
for(int i = 0; i<10; i++)
   for(int j = i+1; j<10; j++)
      if(a[j] < a[i])
   {
         temp = a[i];
         a[i] = a[j];
         a[j] = temp;
      }
}
```

Assignment 1

Implement the bubble sort algorithm to sort a linkedlist.

Upon creation

- During the creation of the linklist, create it in an ordered way.
- You can use the insertion sort.



```
template <class T>
void insert (T num)
{
 node <T>*q,*t,*n ;
 n= new node <T>;
n->data=num;
// insert into empty list
 if( head == NULL )
  {
    head=n;
    head->link = NULL;
  }
 else if(num<head->data)
  {
   q=head;
   while(q->link!=NULL)
          q=q->link;
    n->link=head;
    head=n;
    q->link=NULL;
 }
```

```
else
   {
    q=head;
     t=head->link;
     while(t!=NULL && num>t->data)
     {
         q=t;
          t=t->link;
      }
   q->link=n;
   n->link=t;
     }
}
```

Dr. Aryaf Aladwan

Linked Stack and Linked Queue

Linked Stack and Linked Queue

- Stacks can be implemented using link lists, where the insertion and deletion can be performed without memory limitations.

Linked stack has the following operations:

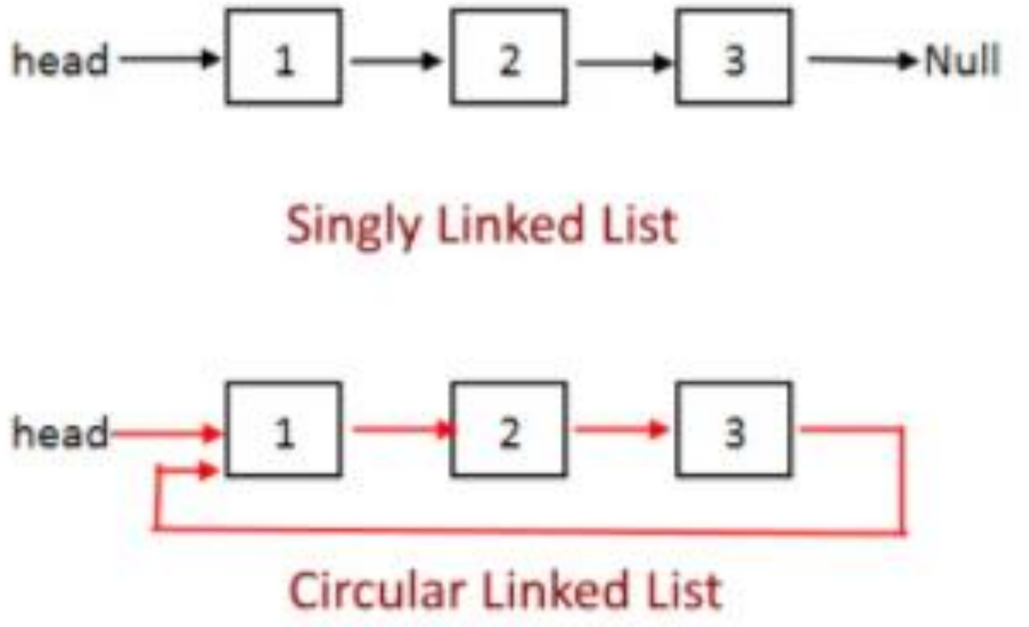
1. IsEmpty(): to check whether the stack is empty or not using the head pointer.
2. Push(): insert nodes at the end of the link list.
3. Pop(): remove nodes from the end of the link list.

- Also, queues can be implemented using link lists. Linked queue has the following operations:

1. IsEmpty(): to check whether the queue is empty or not using the head pointer.
2. Enqueue(): insert nodes at the end of the link list.
3. Dequeue(): remove nodes from the end of the link list.

Circular LinkedList

Circular Linked List is a variation of Linked list, in which the last node in the list is not pointing to NULL but its pointing the first node in the linked list and this makes a circle that is why it is called “Circular Linked List”.



Every time you insert a node you have to make its next pointer points to the head rather than NULL.

node->next=head

Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

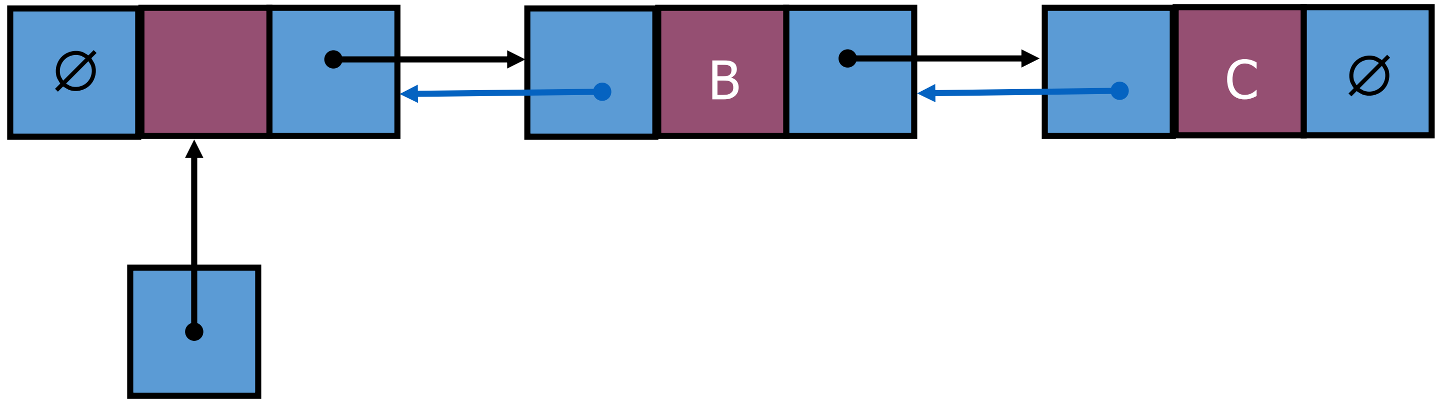
Doubly LinkedList

Variations of Linked Lists

- *Doubly linked lists*
  - Each node reference to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards

Doubly Linked List

- A linear connection of nodes where each node reference to both its predecessor and its successor.



Dummy Head

The Node in Doubly Linked List

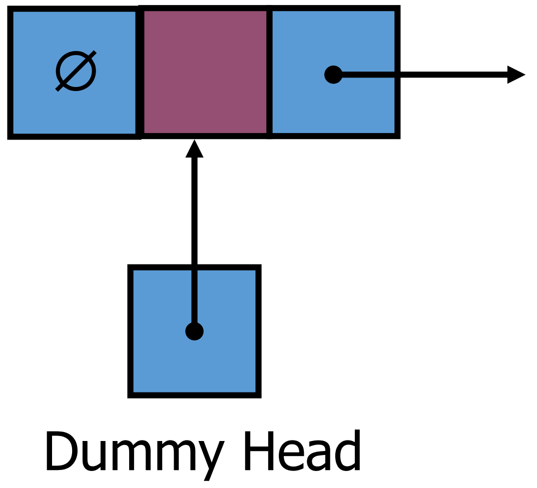
- Declare Node struct for the nodes
  1. data: Generic type data in this example
  2. next: a reference to the next node in the list
  3. prev: a reference to the previous node in the list

```
template <class T>
struct Node
{
T data;
Node <T> *next;
Node <T> *prev;
};
Node <T> *head;
```

Cont.

- Dummy Head Node: is a node without data and the previous reference in it equal to null.
- In constructor :

```
doublylinklist()
{
 head = new node<T>;
head->next=NULL;
head->prev=NULL;
}
```



isEmpty

- Initially in the constructor
- Assign a NULL value to the head pointer

```
bool isEmpty()
{
if(head->next ==NULL)
return true;
else
return false;
}
```

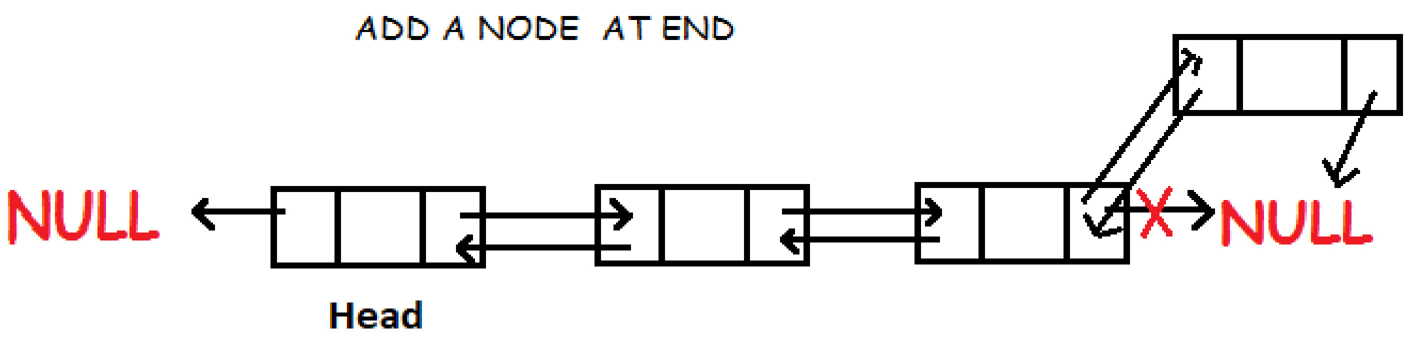
Insert Method

1) Insertion into empty linklist:

```
// insert into empty list
void insert(T num)
{
node <T>*q,*t;
 if( head ->next==NULL ) // insert into empty list
 {
 q = new node <T>;
 q->data = num;
 q->next = NULL;
 q->prev=head;
 head->next=q;
 }
```

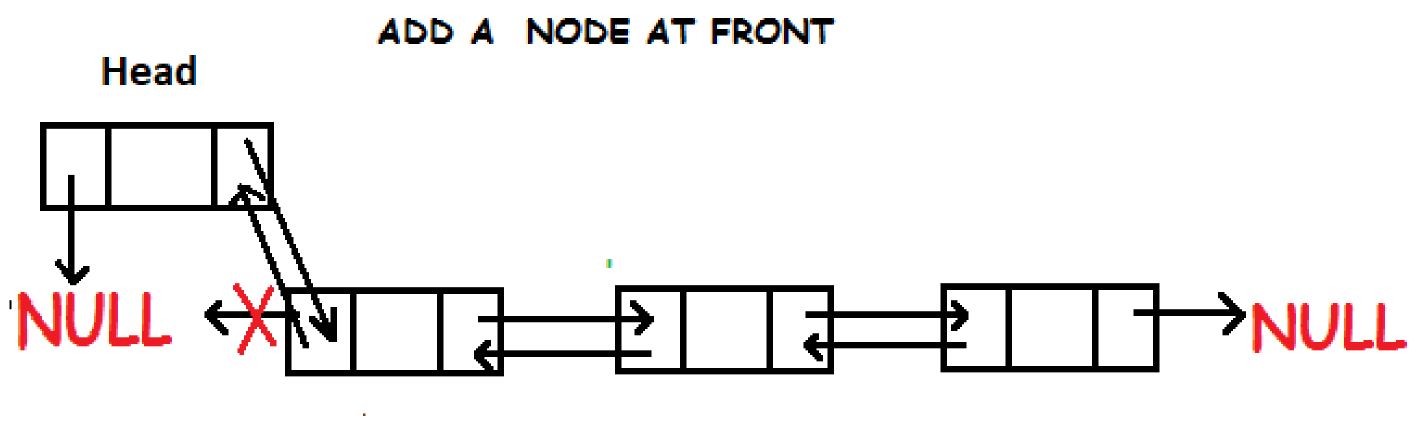
Insert Method

2) Insertion at the end (append):



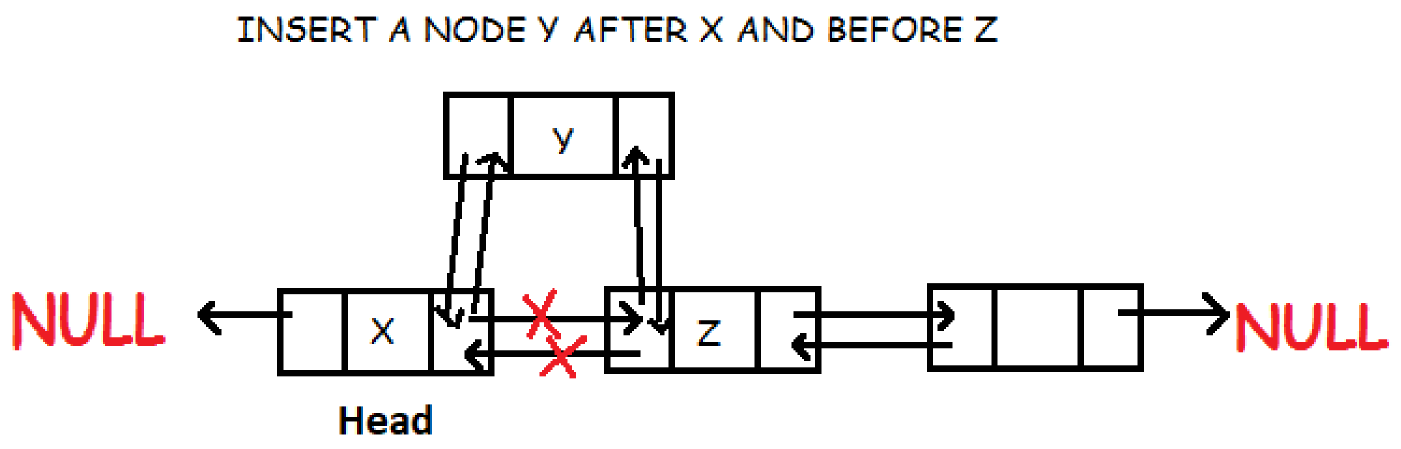
```
else // append
 {
 q = head;
 while( q->next != NULL )
 q = q->next;
 t = new node <T>;
 t->data = num;
 t->next= NULL;
t->prev=q;
 q->next = t;
 }
}
```

AddAsFirst method



```
void add_as_first(T num)
{
node <T>*q;
 q = new node <T>;
 q->data = num;
 q->prev=head;
 q->next=head->next;
 head->next->prev=q;
 head->next=q;
}
```

AddAfter Method

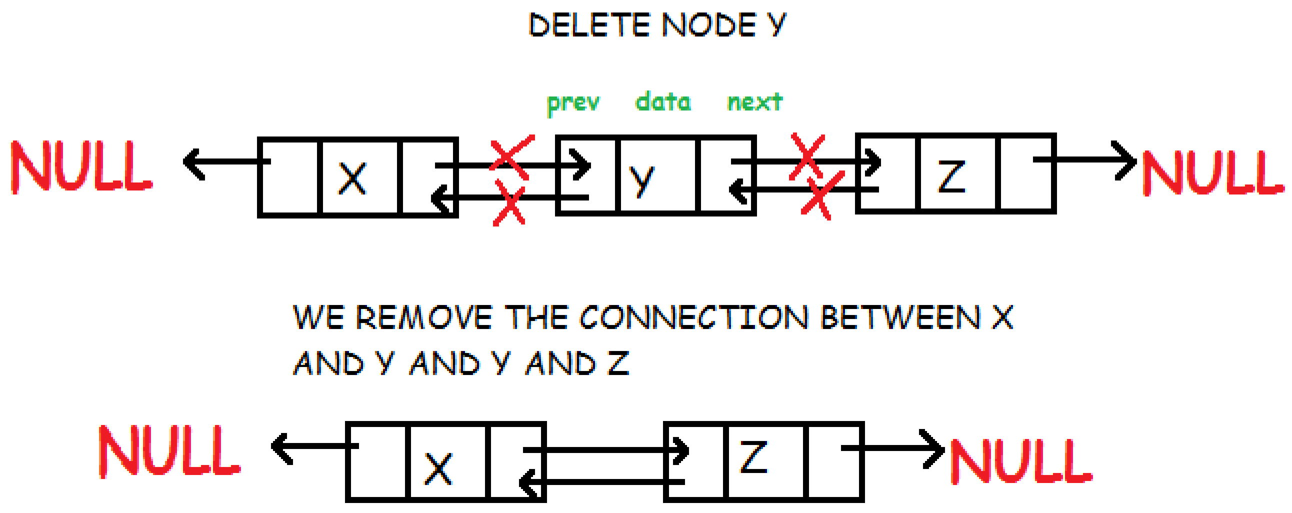


```
void addafter( T c, T num)
{
 node <T> *q,*t;
for(int i=1,q=head->next;i<c;i++)
 {
 q = q->next;
 }
 t = new node <T>;
 t->data = num;
 t->prev=q;
 t->next=q->next;
 q->next->prev=t;
 q->next=t;
}
```

Deleting a node

- Delete a node with the value equal to `num` from the list.
  - If such a node is found, return its position. Otherwise, return null.
- Possible cases of Delete method
  1. Delete from the beginning of the list (first node)
  2. Delete from middle or end of the list
- Steps
  - Find the desirable node
  - Set the reference of the node to null
  - Set the reference of the predecessor of the found node to the successor of the found node

Deleting from the beginning of the Linkedlist



```
T  del ( T num )
{
node <T>*q;
 q = head->next;
 if( q->data == num ) // delete
from the beginning
 {
 head->next = q->next;
 q->next->prev=q->prev;
 delete q;
 return 0;
}
```

Deleting from the middle and end of the Linkedlist

```
// delete from middle
while( q->next!=NULL )
 {
 if( q->data == num )
 {
 q->prev->next=q->next;
 q->next->prev=q->prev;
 delete q;
 return 0;
 }
 q=q->next;
```

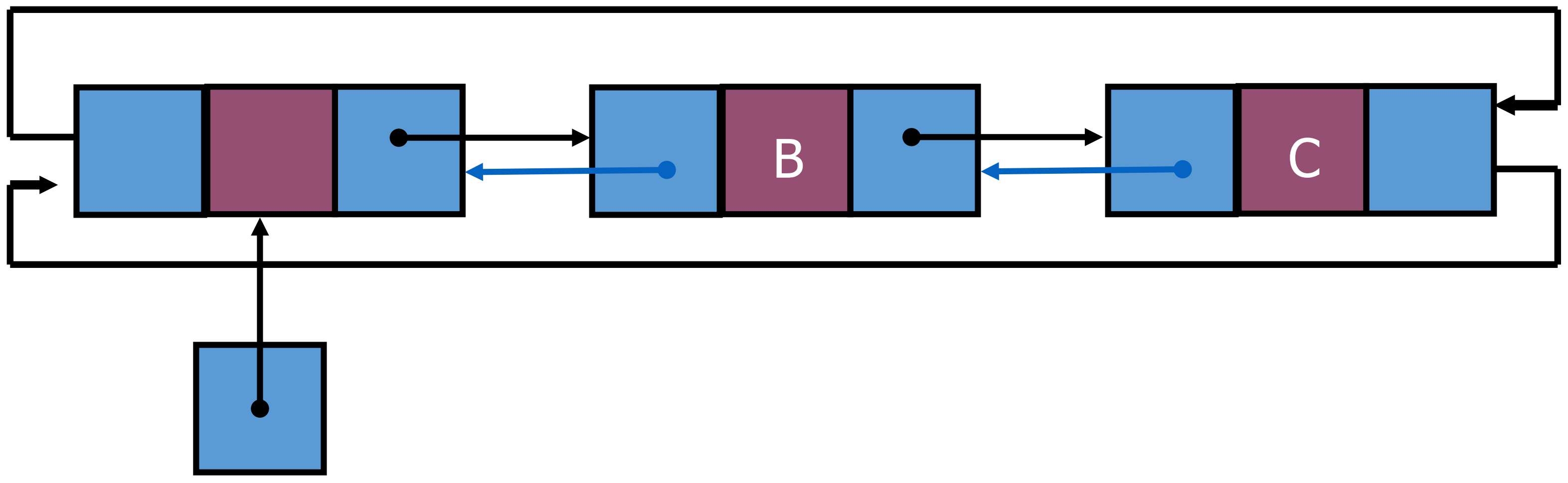
```
// delete from end
if(q->data==num && q->next==NULL)
{
T z;
z=q->data;
q->prev->next=NULL;
delete q;
return z;
}
 }
 cout<<"\nElement "<<num<<" not Found.";
}
```

Assignment 2

Write a full C++ program that implements the doubly linkedlist based on the previous codes.

Circular Doubly linked lists

- *Circular Doubly linked lists*
  - The last node points to the first node of the list



Dummy Head

Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic: a linked list can easily grow and shrink in size.**
    - **We don’t need to know how many nodes will be in the list. They are created in memory as needed.**
    - **In contrast, the size of an array is fixed at compilation time.**
  - **Easy and fast insertions and deletions**
    - **To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.**
    - **With a linked list, no need to move other nodes. Only need to reset some references.**

Applications of linkedlist

1. Implementation of dynamic stacks
2. Implementation of dynamic queues
3. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
4. Binary Search Tree (Doubly linked list).

End