**Data Structures Lab 03(a)**

**Course**: Data Structures (CL2001) **Semester**: Fall 2024
**Instructor**: Sameer Faisal **T.A**: N/A

---

Note:

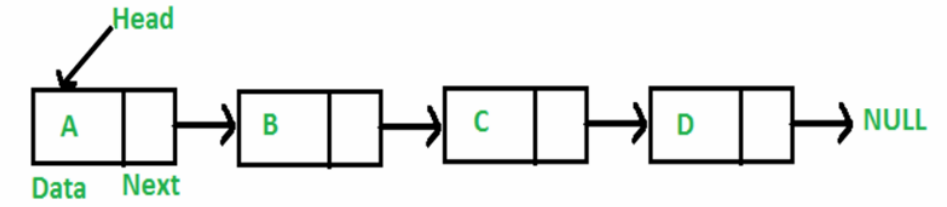
- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

---

# Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.
A singly linked list is indeed a type of linked list where each element (node) consists of two parts:

- **Data:** This part of the node holds the actual value or data that you want to store in the list. It can be any data type, such as integers, characters, or custom data structures.
- **Pointer to the Next Node:** This part of the node contains a reference or pointer to the next node in the list. This pointer helps maintain the structure of the linked list and allows you to traverse the list in a unidirectional manner, usually from the head (the first node) to the last node. The last node typically points to NULL to indicate the end of the list.




**Class Task # 1:** Implement a singly Linked List class.
Singly Linked List is an important bifurcation of Linked List data structure.
It is called Singly as it holds one data member and one link member associated to each node in the list.
In order to create a singly Linked List class, we first need a helper class to implement nodes. Each node object will have 3 public data members.

(a) Int type Key (unique to each node).
(b) Data.
(c) next pointer. Along that we will use a default constructor and a parameterized constructor of Node class to manipulate those data members.

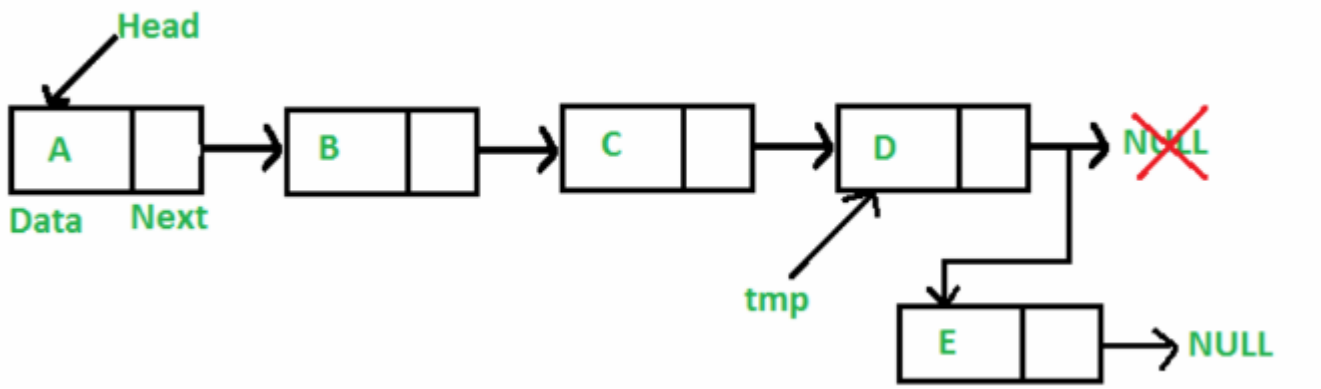
The singly Linked List class will only have a public Node type pointer variable “head” to point to the first node of the list. Together with the help of default and parameterized constructor the head’s value will be manipulated in the singly Linked List class.

```
//Node Object
Class Node
{
        //public members: key, data, next
         Node ()
        {
                //initialize both key and data with zero while next pointer with NULL.
        }
        Node (int k, int d)
        {
                //assign the data members initialized in default constructor to these arguments.
        }
};

//SinglyLinkedList Object
Class SinglyLinkedList
{
        // create head node as a public member
        //Default Constructor
        SinglyLinkedList()
        {
                //initialize the head pointer will NULL;
        }
        //Parameterized Constructor with node type 'n' pointer
        SinglyLinkedList (Node* n)
        {
                //Assign the head pointer to value of pointer n;
        }
};
```

**Class Task # 2:** Add a node at the end of a Singly Linked List (append).

**Add a Node at the End:**
In this task, the new node is always added after the last node of the given Linked List.
For example, if the given Linked List is 20->1->2->5->10 and we add an item 25 at the end, then the Linked List becomes 2->1->2->5->10->25.
Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.




To append a node at the end of the list, first check if there exists a node already with that key or not. For that you may need a helper function inside the singly Linked List class to perform the test. In case a node already exists with that key value, Intimate the programmer to use another key value to append a node. On the contrary, If the node doesn't exist, append a node at the end. Before that check if the list has some node or not, i.e., Check if the head pointer is null or not. If it is null, access the head and assign node n to it. Otherwise, traverse through the list to find the node whose next is Null, i.e., the last node in the list. Then, assign the node n to next pointer of the last node. Also, make sure the next pointer of node n (new last node) is null.

```
//Creating a Node type helper Function named node Exists (argument key)
Node nodeExists (int k)
{
//temporary pointer var to hold Null value;
//node type pointer to hold head pointers value;
// loop condition (Traverse through the node until the ptr variable points to null)
{
If the ptr variable's key = passed key argument
{
//assign the pointer ptr to temp variable;
}
Else
{
//assign the pointer ptr to point to the next pointer's value.
}
return the temp variable;
}
//Append Function
Void appendNode (Node* n, int data)
{
//create new node and assign data to it.
//check if the head pointer points to Null or not with a condition
```

```
if (head == NULL)
{
//If it does, assign the head pointer the passed pointer ‘n’. This will put the address of node n in
the head pointer.
}
// Else traverse through the list to find the next pointer holding Null as address (last node)
Else
{
if (nodeExists(n->key) != NULL ) {
// Print an intimation that a node holding passed key already exists.
}
else
{
If (head != NULL)
{
// assign the head pointer to a new pointer of type Node (say, ptr).
//loop through the list using ptr until the next of any node contains null.
//when ptr->next = NULL. Then assign the n node to ptr->next to append that node after
the last found node.
}
}
}
}
```

**Class Task # 3:** Add a node at the front of a Singly Linked List (Prepend a new node).

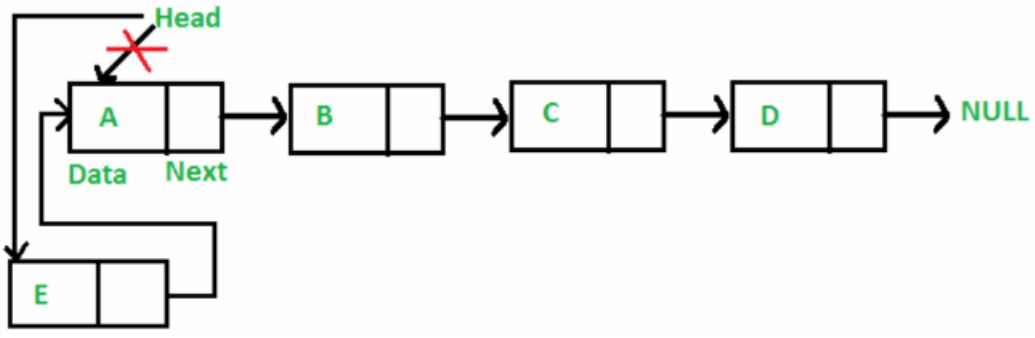
**Add a Node at the Front:**

The new node is always added before the head of the given Linked List. The newly added node becomes the new head of the Linked List.

For example, if the given Linked List is 20->10->15->17 and we add an item 5 at the front, then the Linked List becomes 5->20->10->15->17.

Let us call the function that adds at the front of the list is push ().

The push () must receive a pointer to the head pointer because push must change the head pointer to point to the new node.



To prepend a node, simply check if the key that is passed already exists or not, if yes, intimate the programmer to pass a new key value or else assign the new head node's value to the next pointer of new node. Then set the new head to be the new node's address.

```
//Prepending a Node.
void prependNode(Node* n)
{
// checking the value of node's key if the node with that key already exists.
if (nodeExists(n->key) != NULL ) {
// Print an intimation that a node holding passed key already exists.
}
else
{
// new node's next is pointing to the head i.e. address of first node.
// since we have changed the head pointer's value from first node to the new node, now the new
. // head will be pointing to address of new node.
}
}
```

**Class Task # 4:** Add a node after a given node in a Singly Linked List.

Consider a singly linked list:
4 -> 1 -> 5 - > 7 -> 2
You want to add the value 3 after node 1.
The List should be transformed as:
4 -> 1 -> 3-> 5-> 7 -> 2

To insert a new node after some node, create a void function named insertNodeAfter () carrying two arguments, one for the key of the node after which the insertion is to be done, the new node.

```
//Inserting a new node after some node.
Void insertAfterNode (key , new node)
{
//creating a node type pointer that calls nodeExists () and passes the key argument into it to check if
there exists a node with this key value
Node * ptr = nodeExists(k);
If(ptr == NULL)
{
//print a message saying no node exists with that key
}
Else
{
//check if any key with that already exists to avoid duplication
```

```
if (nodeExists(n->key) != NULL )
{
//Print an intimation that a node holding passed key already exists. Append a new node
with different key value.
}
Else
{
//now the next pointer of new node will be holding the address kept in next pointer of ptr.
//assign the address of node to the next pointer of the previous node (ptr).
//print a message that anode is inserted.
}
```

**Class Task # 5:**

Deleting a node from a Singly Linked List can have two cases:

1. Delete Last node.
2. Delete any other node.

Consider a singly linked list:

4 -> 1 -> 5 - > 7 -> 2

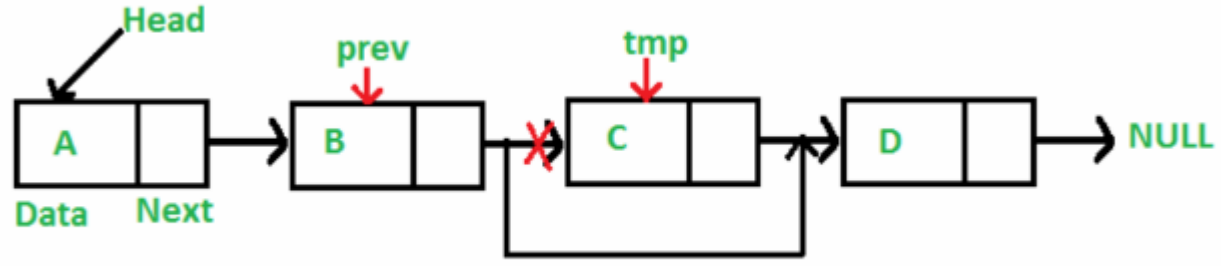
You want to delete the value 5.

The List should be transformed as:

4 -> 1 -> 7 -> 2

To delete a node from the linked list, we need to do the following steps.

1) Find the previous node of the node to be deleted.
2) Change the next of the previous node to hold the node next to the node to be deleted.
3) Free memory for the node to be deleted.



```
void deleteNode(key)
{
// create a Node type pointer to hold head (say, temp)
// create a Node type pointer to hold previous node (say, prev)
// check if head contains the key
if (temp!=NULL && temp->key==key){
// assign next of temp to temp, which will unlink the node pointed by head
// delete the temp node
// return
}
Else{
While(temp!=NULL && temp->key != key){ // traverse the list until temp is not NULL

//and temp's key is same as the key.

// set prev to temp
// set temp to temp's next pointer
}
If(temp == NULL){ // key not found.
// return
}
// unlink by setting temp's next to prev's next.
// free memory by deleting temp
}

}
```

**Class Task # 6:**
Update a node in a Singly Linked List:

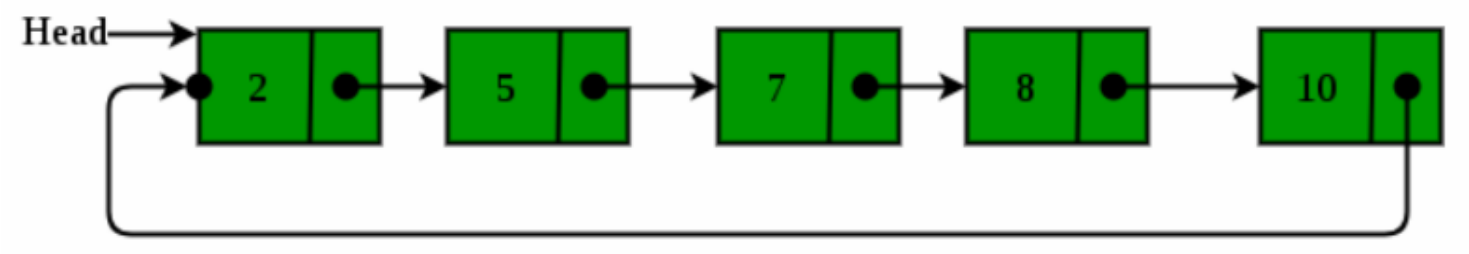
Consider a singly linked list:
4 -> 1 -> 5 - > 7 -> 2
You want to update the node 7 to 6.
The List should be transformed as:
4 -> 1 -> 5-> 6 -> 2

Updating Linked List or modifying Linked List means replacing the data of a particular node with the new data. Implement a function to modify a node's data when the key is given. First, check if the node exists using the helper function node Exists.

```
void updateNode(key, new_data)
{
// call the nodeExists function and hold the return value in a Node type pointer (say, ptr)
If(ptr!=NULL){
// set ptr's data as new data
}
Else{
// print a message sating node does not exist.
}
}
```

# Circular Linked List:

A Circular Linked List is a data structure in which elements, known as nodes, are connected in a circular fashion. Unlike a regular singly linked list, where the last node points to null, in a circular linked list, the last node points back to the first node, creating a loop.




Here's a list of common helper functions that are often implemented in a Circular Linked List:

**Append:** Add a new node to the end of the circular linked list.
**Insert:** Add a new node at a specific position in the circular linked list.
**Delete:** Remove a node with a given value from the circular linked list.
**Search:** Find a node with a specific value in the circular linked list.
**Display:** Print the elements of the circular linked list.
**Reverse:** Reverse the order of nodes in the circular linked list.

**Class Task # 7:**
Creation of Circular Linked List:

```
class Node{
    public:
    int data;
    Node * next;
    Node()
    {
        data =0;
        next=NULL;
    }


     Node(int val)
     {
         data=val;
         next = NULL;
     }

};
```

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

   void insertatend( int val)
   {
      Node* n= new Node(val);
      if(head == NULL)
      {
         head=n;
         tail=n;
         tail->next=head;

      }
      tail->next=n;
      tail=tail->next;
      tail->next=head;

   }
   void insertATFront( int val)
   {
      Node * n = new Node(val);
      tail->next= n;
      n->next=head;
      head=n;
```

# Lab Exercises

1. Solve the following problem using a Singly Linked List:

   Given a Linked List of integers (input by user), write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

   Examples:

```
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL
```

2. Solve the following problem using a Singly Linked List:

   Given a Linked List (input by user) of integers or string (as per your choice), write a function to check if the entirety of the linked list is a palindrome or not.

   Examples:

```
Input: 1->0->2->0->1->NULL
Output: Linked List is a Palindrome

Input: B->O->R->R->O->W->O->R->R->O->B->NULL
Output: Linked List is a Palindrome
```

3. Create a circular link list and perform the mentioned tasks:
    i. Insert a new node at the end of the list.
    ii. Insert a new node at the beginning of list.
    iii. Insert a new node at given position.
    iv. Delete any node.
    v. Print the complete circular link list.