

## Data Structures Lab

### Session 4

**Course:** Data Structures (CS2001)

**Instructor:** Mafaza Mohi,

**Semester:** Fall 2022

**T.A:** N/A

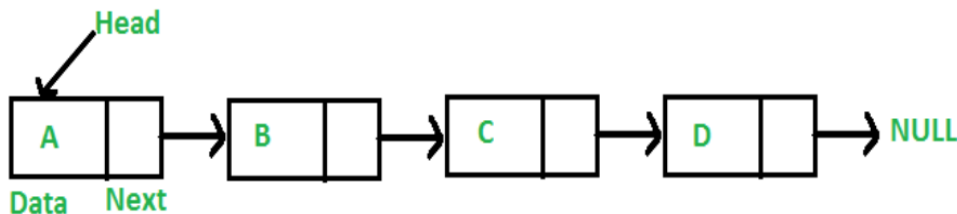
---

#### Note:

- Maintain discipline during the lab.
  - Listen and follow the instructions as they are given.
  - Just raise your 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.



**Task:** Implement a SinglyLinkedList 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 SinglyLinkedList 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 SinglyLinkedList 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 SinglyLinkedList 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;
    }
};

```

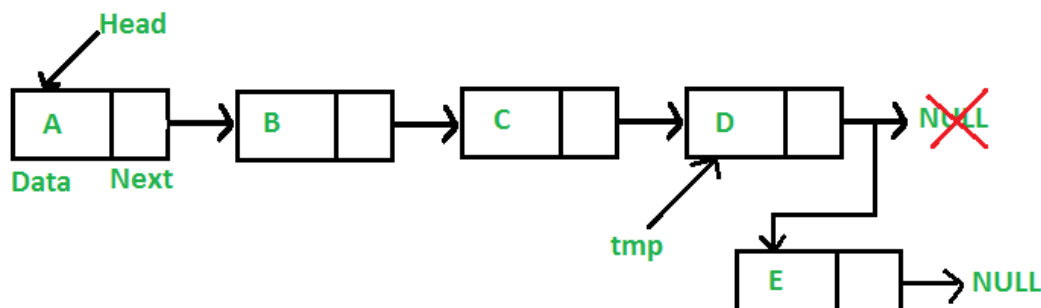
---

### 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 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

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 SinglyLinkedList 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 the 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 that 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 nodeExists (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.
    }
}

```

---

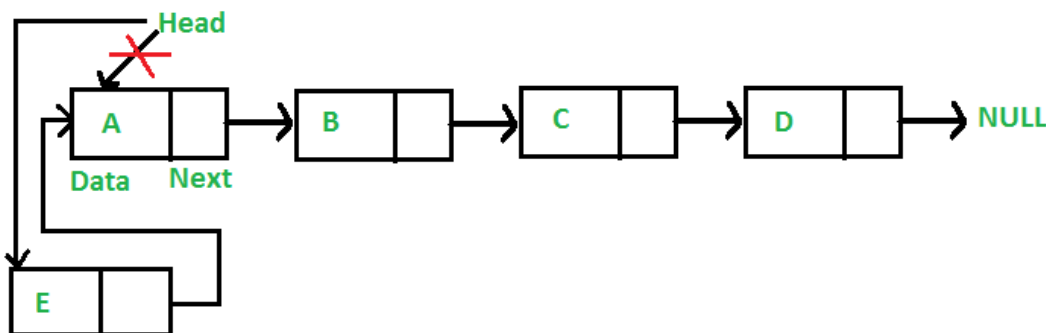
### Add a Node at the Front:

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

For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25.

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.
        }
    }

```

---

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.

```

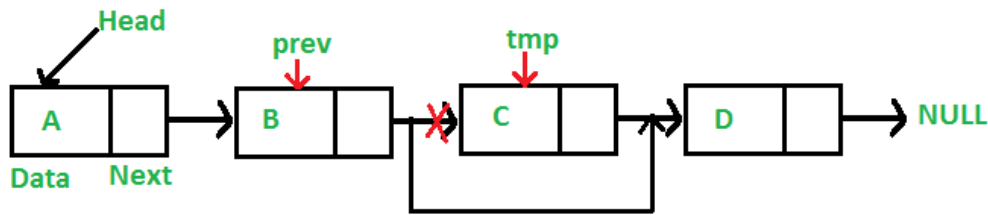
Void insertAfterNode (key , new node)
{
    //creating a node type pointer that calls nodeExists () and passes the key argument into it to check if
    there exists a node with this key value
    Node * ptr = nodeExists(k);
    If(ptr == NULL)
    {
        //print a message saying no node exists with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

```

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
    }
}
  
```

---

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 nodeExists.

```

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 saying node does not exist.
    }
}

```

---

### Task # 1:

Implement a singly linked list with the following functions:

1. Insert a node at head
2. Insert a node at tail/end/back
3. Insert a node at any position
4. Delete a node by value
5. Delete head
6. Delete tail
7. Delete a node at any position.

### Task # 2 :

Alter a node's data in a Singly Linked List

**Input:** S->A->T->A

**Output:** D->A->T->A

### Task # 3

Solve the following problem using a Singly Linked List.

Given a singly linked list of characters, write a function to make word out of given letters in the list

Example:

**Input:** C->S->A->R->B->B->E->L->NULL

**Output:** S->C->R->A->B->B->L->E->NULL

### Task # 4:

Use the class of SLL created by you during the lab task 1. Do the following:

1. Reverse the linked list
2. Sort the contents of linked list
3. Find the duplicates in the linked list

Lab4: Singly Linked List		
Std Name:		Std_ID:
Lab1-Tasks	Completed	Checked
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		
Task# 7		

Instructor signature: \_\_\_\_\_