Data Structures and Algorithms
CS-206

Linked Lists Part C

Instructor
Dr. Maria Anjum
Assistant Professor

Department of Computer Science
Lahore College for Women University

Linked List - Class

```
class linkedList                    // linked list class
{
private:
        typedef struct node{
        int data;
        node* next;
        }* nodePtr
        nodePtr head;
        nodePtr temp;
        nodePtr curr;
public:
        linkedList(){
        head = NULL;
        curr = NULL;
        temp = NULL;
                };
    void addNode(int addData); //function to add data
    void delNode(int delData); //function to delete data
    void printList();                       //print list
    ~linkedList();
}
```

Linked List – Class::void addNode(int addData);

```
void linkedList::addNode(int addData){
        nodePtr n= new node;
        n->next=NULL;
        n->data= addData;

        if (head!=NULL){
        curr = head;
                while (curr->next!= NULL)
                {
                        curr=curr->next;
                }
        curr->next= n;
        }

        else {
                head = n;   → What is the role of this statement?
        }

}
```

Linked List – Class::void delNode(int delData);

```
void linkedList::delNode(int delData){
        nodePtr delPtr= NULL;
        temp= head;
        curr= head;
        while (curr!= NULL && curr->data!=delData){
                temp = curr;
                curr = curr->next;
        }
        if (curr == NULL){
        cout<<delData <<" was not found in the list. "<<endl;
        delete delPtr; //delete creted pointer to avoid creating garbage
        }
        else{
                delPtr = curr;
                curr = curr->next;
                temp->next = curr;
                delete delPtr;

        cout<<"The value "<<delData<<" is deleted from the list. \n";
        }
        }
```

```
if(delPtr==head)
{
head=head->next;
temp=NULL;
}
```

Linked List – Class:: void printList();

```
void linkedList::printList(){
        curr=head;

        while (curr!=NULL)
        {
                cout<< curr->data<<endl;
                curr=curr->next;
        }
       //cout<< curr->data<<endl;
}
```

→ What will happen if we run this statement.

Linked List – Main Function

```
int main(int argc, char** argv) {
                    linkedList myList;
// add values to list
                    myList.addNode(15);
                    myList.addNode(11);
                    myList.addNode(1);
                    myList.addNode(8);
                    myList.printList(); // print list
// delete values from list
                    myList.delNode(11);
                    myList.delNode(1);
                    myList.printList(); // print list
                    myList.delNode(1);
                    }
```

Review Singly Linked List

- Linked lists are building blocks for many other data structures like stacks and queues.
- Linked lists are a sequence of nodes containing data fields and pointers to the next node (or) next node and previous nodes based on its type.
- Linked lists permit addition/ removal of node in constant time.
- Unlike arrays the order of linked list elements need not be contiguous in memory.
- Unlike arrays there is no upper limit on the amount of memory reserved.
- A singly linked list is the simplest of linked lists.
- Each node of a singly linked list has data elements and a single link (pointer) that points to the next node of the list (or) NULL if it is the last node of the list.
- Addition/ Deletion of a node to the singly linked list involves the creation/ deletion of the node and adjusting the node pointers accordingly.

Doubly Linked List

- Doubly linked lists are like singly linked lists, except each node has two pointers
- Traverse lists forward and backward.
- Insert anywhere in a list easily. This includes inserting **before a node**, **after a node**, **at the front of the list**, and **at the end of the list**.
- Deletion of nodes is very easily.

Doubly Linked List

```
typedef struct node {
 int data;
 node* next;
 node* prev;
} ;
Node* head;
Node* tail;
Node* n;
```

Circular Linked List

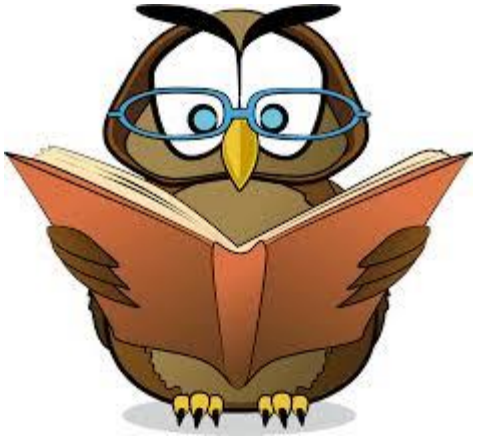
- A circular list is one in which the last node in the list points to the first node.

- **For example**, when multiple applications are running on a PC, operating system usually put the running applications on a list and cycles 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.

Circular Linked List

Circular linked lists also make implementation easier, because

it eliminates the boundary conditions associated with the **beginning and end of the list**, thus eliminating the special case code required to handle these boundary conditions.



Reading

- https://www.youtube.com/watch?v=YJRRpXYldVQ

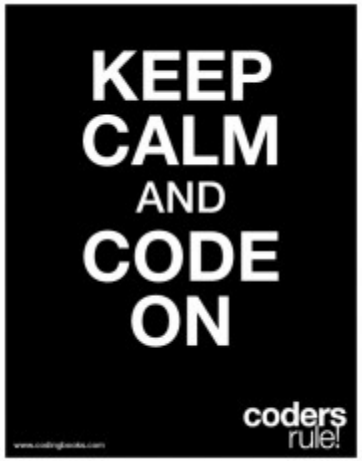
Reminder of Practical and Reading Assignment

- Write runable code for Singly linked List

- Create new nodes
- Insert a node at beginning
- Insert a node at the end of the list
- Insert a node somewhere in the list

- Delete a note at the beginning of the list
- Delete a node at the end of the list
- Delete a node somewhere in the list

- Display linked list
- Search a number in the list



Practical and Reading Assignment

- Write runable code for **Doubly linked List**

- Create new nodes
- Insert a node at beginning
- Insert a node at the end of the list
- Insert a node somewhere in the list

- Delete a note at the beginning of the list
- Delete a node at the end of the list
- Delete a node somewhere in the list

- Display linked list
- Search a number in the list