Data Structures Lab 5

Course: Data Structures (CL2001) **Semester:** Spring 2023
Instructor: Bushra Sattar **T.A:** Muhammad Anas

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.

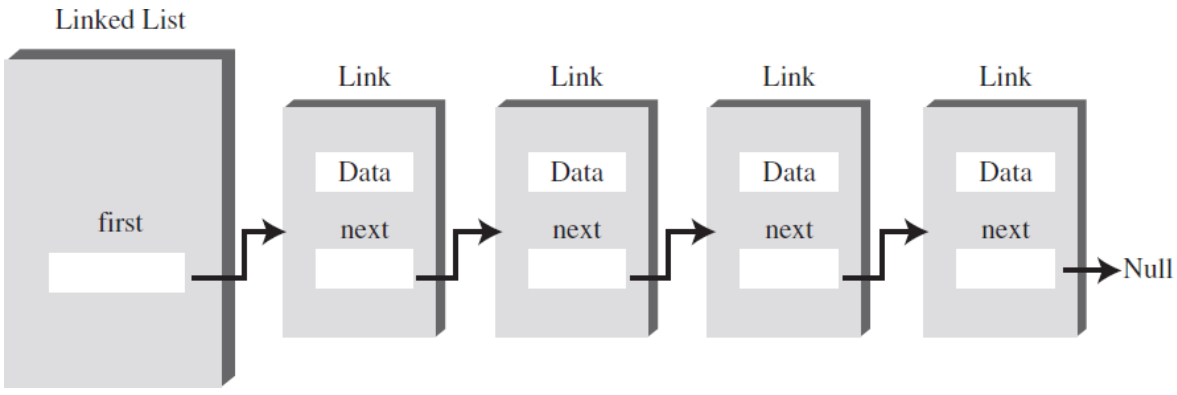
Linked List:

We saw that arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas, in an ordered array, insertion is slow. In both kinds of arrays, deletion is slow. Also, the size of an array can't be changed after it's created.
Linked lists are probably the second most commonly used general purpose storage structures after arrays.

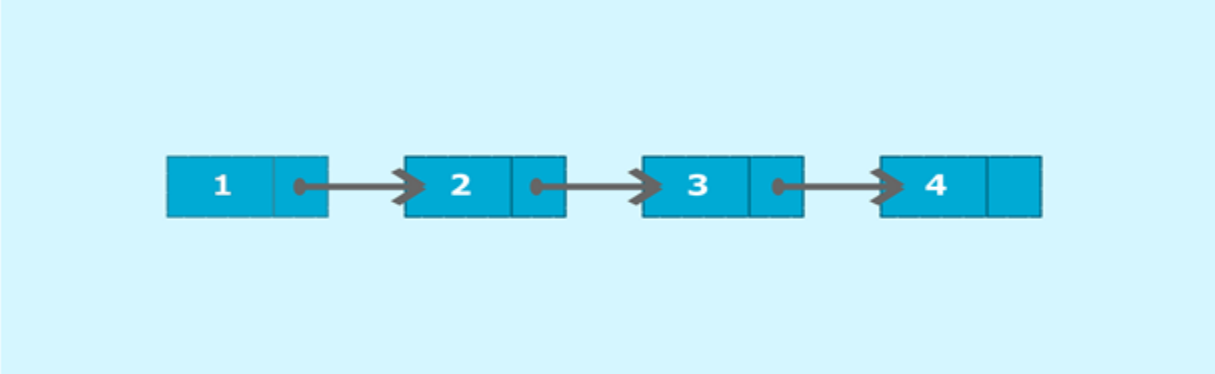
Linked list has the following properties

- Successive elements are connected by pointers/references
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)

The linked list is a versatile mechanism suitable for use in many kinds of general-purpose databases. It can also replace an array as the basis for other storage structures such as stacks and queues. In fact, you can use a linked list in many cases in which you use an array unless you need frequent random access to individual items using an index. A simple linked list is shown below:



The linked list shown above is called as **Singly** or **Single-Ended Linked List** because we have the reference of only one end called the **first** (which refers/points to the front end of the list) and can move only **forward** because each node has the reference of next node only.



Consider the above example; node 1 is the head of the list and node 4 is the tail of the list. Each node is connected in such a way that node 1 is pointing to node 2 which in turn pointing to node 3. Node 3 is again pointing to node 4. Node 4 is pointing to null as it is the last node of the list.

Algorithm

- Create a class Node which has two attributes: data and next. Next is a pointer to the next node.
- Create another class which has two attributes: head and tail.
- addNode() will add a new node to the list:
- Create a new node.
- It first checks, whether the head is equal to null which means the list is empty.
- If the list is empty, both head and tail will point to the newly added node.
- If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

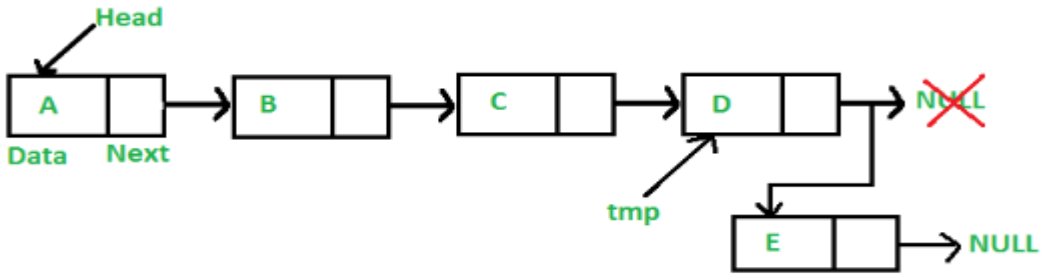
a. display() will display the nodes present in the list:

- Define a node current which initially points to the head of the list.
- Traverse through the list till current points to null.
- Display each node by making current to point to node next to it in each iteration.

Task # 1: Implement a singly Linked List class

Task # 2: Add a node at the end of a Singly Linked List.

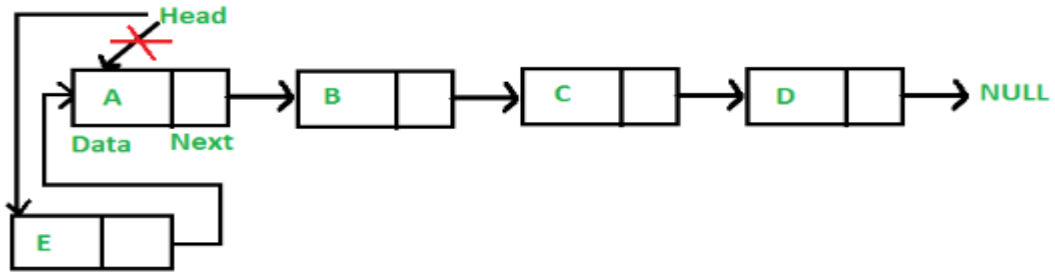
Add a Node at the End:
In this task, the new node is always added after the last node of the given Linked List.
For example, if the given Linked List is 20->1->2->5->10 and we add an item 25 at the end, then the Linked List becomes 2->1->2->5->10->25.
Since a Linked List is typically represented by the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

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.



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

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

Task # 5

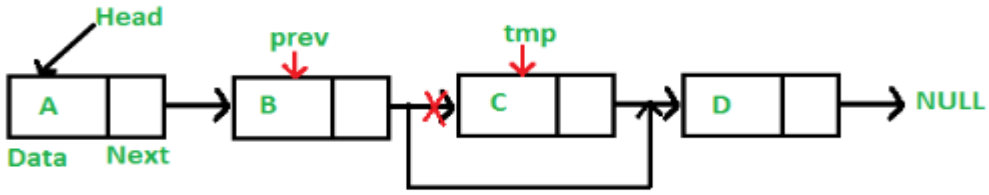
Delete a node from a Singly Linked List

- Delete Last node
- Delete any other node

- Consider a singly linked list 4 -> 1 -> 5 - > 7 -> 2 and 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.



Task # 6

Update a node in a Singly Linked List

Consider a singly linked list 4 -> 1 -> 5 - > 7 -> 2 and 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.

Task # 7

Solve the following problem using a Singly Linked List.

Given a Linked List of integers, 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
```

Task # 8

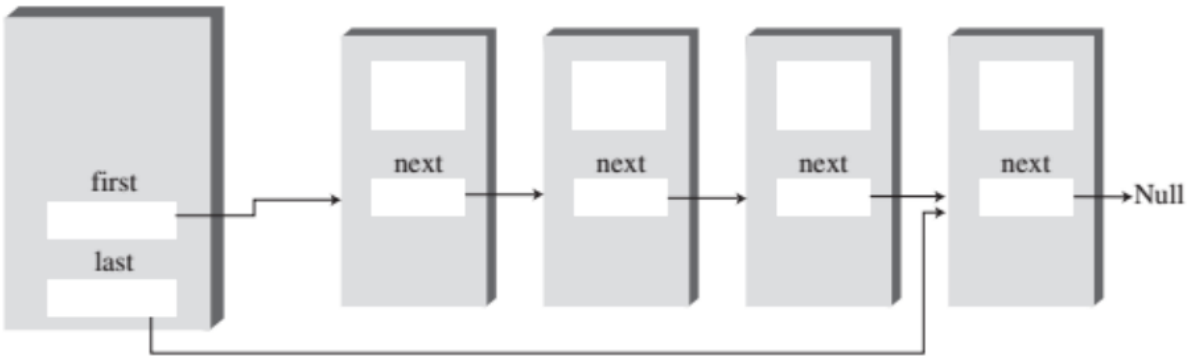
Solve the following problem using a Singly Linked List.

Given a Linked List of integers or string, write a function to check if the entirety of the linked list is a palindrome or not
Examples:

```
Nodes of singly linked list:
1 2 3 2 1
Given singly linked list is a palindrome
```

Double-Ended Linked List

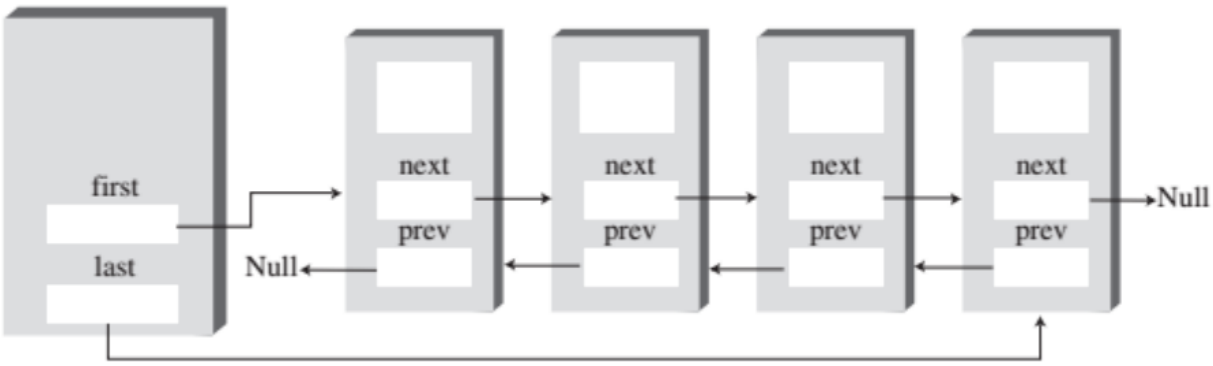
A double-ended list is similar to an ordinary linked list, but it has one additional feature: a reference to the last link as well as to the first link.



The reference to the last link permits you to insert a new link directly at the end of the list as well as at the beginning. Of course, you can insert a new link at the end of an ordinary single- ended list by iterating through the entire list until you reach the end, but this approach is inefficient. Access to the end of the list as well as the beginning makes the double-ended listsuitable for certain situations that a single-ended list can't handle efficiently. One such situation is implementing a queue.

Doubly Linked List

There is a third type of list called the doubly linked list (not to be confused with the double-ended list). What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. This type of list is shown in the following figure:

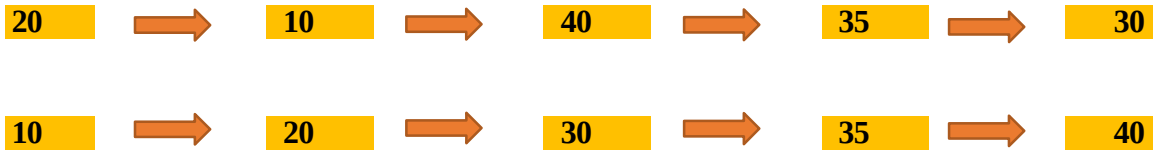


Practice Task 9: Sorted Linked List Example

```
class Link
{
       public long dData;                    // data item
       public Link next;                     // next link in list
       // -------------------------------------------------------------
       public Link(long dd)                  // constructor
       { dData = dd; }
       // -------------------------------------------------------------
       public void displayLink()             // display this link
       { System.out.print(dData + " "); }
}  // end class Link
////////////////////////////////////////////////////////////////
class SortedList   {
       private Link first;                   // ref to first item
       // -------------------------------------------------------------
       public SortedList()                   // constructor
       { first = null; }
       // -------------------------------------------------------------
       public boolean isEmpty()              // true if no links
       { return (first==null); }
       // -------------------------------------------------------------
       public void insert(long key)          // insert, in order
       {
              Link newLink = new Link(key);     // make new link
              Link previous = null;             // start at first
              Link current = first;
```

```
        // until end of list,
        while(current != null && key > current.dData)
        {                               // or key > current,
               previous = current;
               current = current.next;       // go to next item
        }
        if(previous==null)                  // at beginning of list
               first = newLink;                 // first --> newLink
        else                                // not at beginning
               previous.next = newLink;         // old prev --> newLink
        newLink.next = current;             // newLink --> old currnt
 }  // end insert()
 public Link remove()            // return & delete first link
 {                              // (assumes non-empty list)
        Link temp = first;                  // save first
        first = first.next;                 // delete first
        return temp;                        // return value
 }
 //-------------------------------------------------------------
       public void displayList()
       {
              System.out.print("List (first-->last): ");
              Link current = first;       // start at beginning of list
              while(current != null)      // until end of list,
              {
                     current.displayLink();   // print data
                     current = current.next;  // move to next link
              }
              System.out.println("");
       }
} // end class SortedList
class SortedListApp
{
       public static void main(String[] args)
       {                               // create new list
              SortedList theSortedList = new SortedList();
              theSortedList.insert(20);    // insert 2 items
              theSortedList.insert(40);

              theSortedList.displayList(); // display list

               theSortedList.insert(10);     // insert 3 more items
               theSortedList.insert(30);
               theSortedList.insert(50);

               theSortedList.displayList(); // display list

               theSortedList.remove();       // remove an item

               theSortedList.displayList(); // display list
        }  // end main()
}  // end class SortedListApp
```

Practice Task 9: Use Sorted Linked List as a sorting mechanism to sort the following unordered array into an ordered array, as shown below.



Task-10:

Create a doubly link list and perform the mentioned tasks.

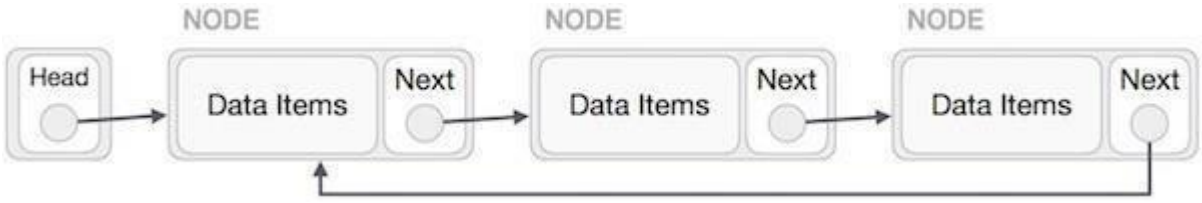
i. Insert a new node at the end of the list.
ii. Insert a new node at the beginning of list.
iii. Insert a new node at given position.
iv. Delete any node.
v. Print the complete doubly link list.

Circular Linked List

Circular Linked List is a variation of the Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

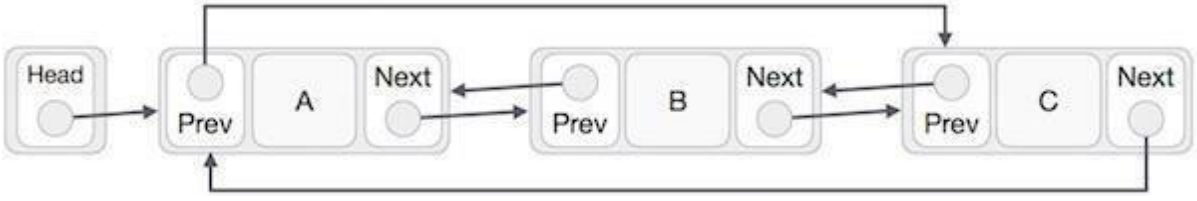
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node



Doubly Linked List as Circular

In the doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, the following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as the doubly linked list.
- The first link's previous points to the last of the list in case of a doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.
- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list

Task-11:

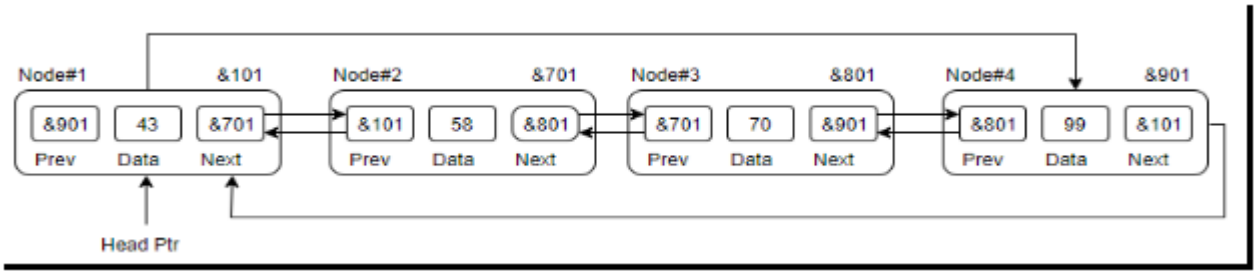
Create a circular link list and perform the mentioned tasks.

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.

Task-12:

Create a circular Double 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 double link list.



Task-13:

Give an efficient algorithm for concatenating two doubly linked lists **L** and **M**, with head and tail preserved nodes, into a single list that contains all the nodes of **L** followed by all the nodes of **M**