EC-200 Data Structures

LAB MANUAL # 04

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

# Lab #04

# Linked List Implementation

# Lab Objective:

To implement Linked List ADT functions.

# Lab Description:

Arrays are used to save data in memory as long as program runs, our program can read/write data to arrays. Some pluses and minuses of arrays are

+ Fast element access.

-- Impossible to resize.

* Many applications require resizing!
* Required size not always immediately available.

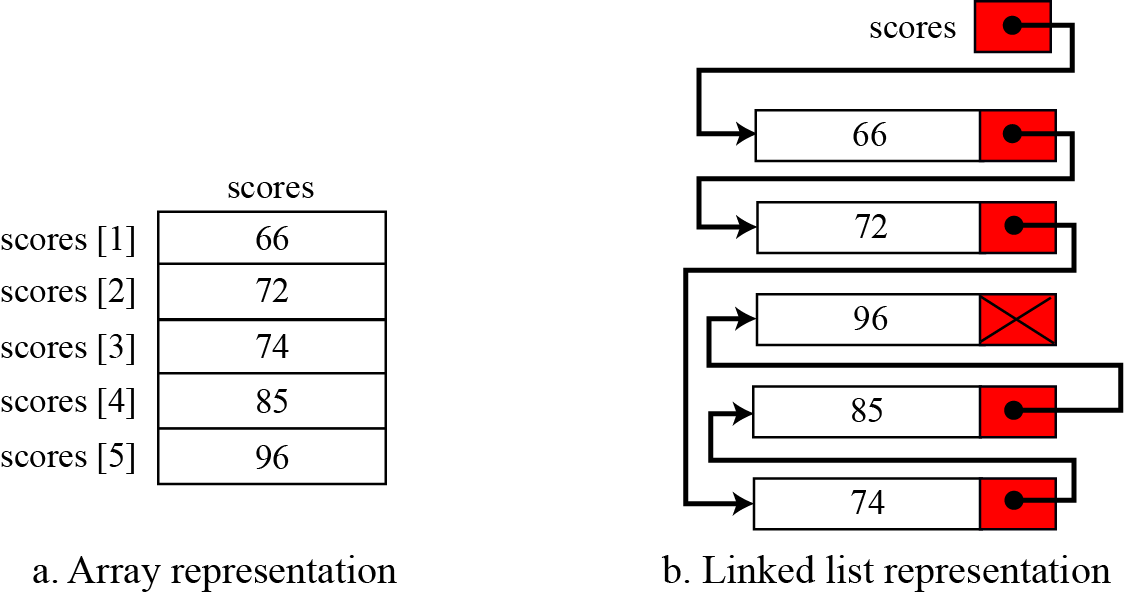
To overcome the Plus & minuses of arrays, Solution was proposed in form of Linked lists. Linked lists are type of data structure that provide Resize ability at run time with small computational complexity as compared to dynamic arrays.

## Linked List:

Linked List is a collection of data in which each element contains the location of the next element.

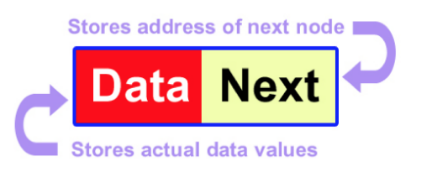
1. Each element contains two parts: data and link.
2. The link contains a pointer (an address) that identifies the next element in the list.

Both an array and a linked list are representations of a list of items in memory. The only difference is the way in which the items are linked together. Figure compares the two representations for a list of five integers



Nodes: Nodes are the basic building elements in a linked list. The nodes in a linked list are called self-referential records because each instance of the record contains a pointer to another instance of the same structural type.

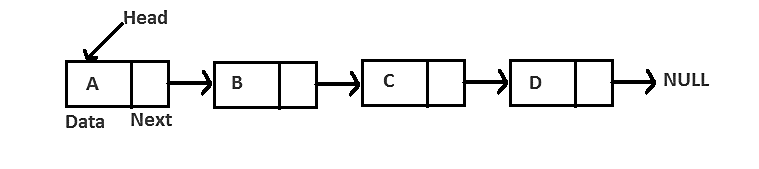
|  |
| --- |
| **Node structure** |
| Struct node  {  // data  node\* next;  } |



## Basic Operations in linked list ADT:

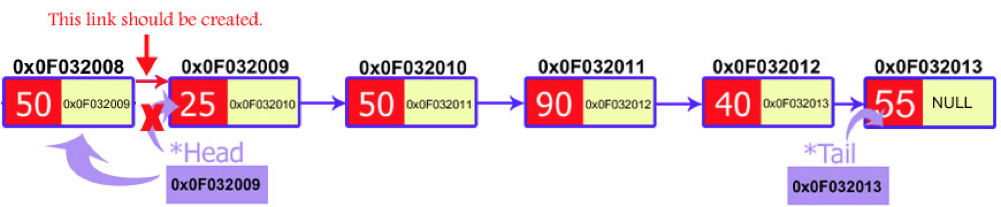
|  |
| --- |
| **Insert first node** |
| void insertFirstNode (int v)  {  node \*head=new node;  head->data=v;  head->next=NULL;  } |

1. Insertion (Start, End, Any position)
2. Deletion (Start, End, Any position)
3. Searching or Iterating through the list to display items.



## Inserting at the start

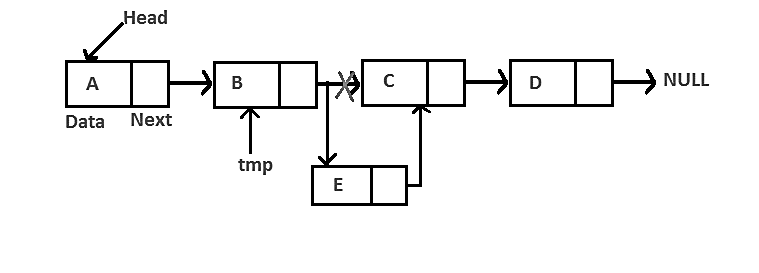
1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node



|  |
| --- |
| **Insert at start** |
| void insert(int v)  {  node \*p=new node;  p->data=v;  p->next=head;  head=p;  } |

## Inserting at any position

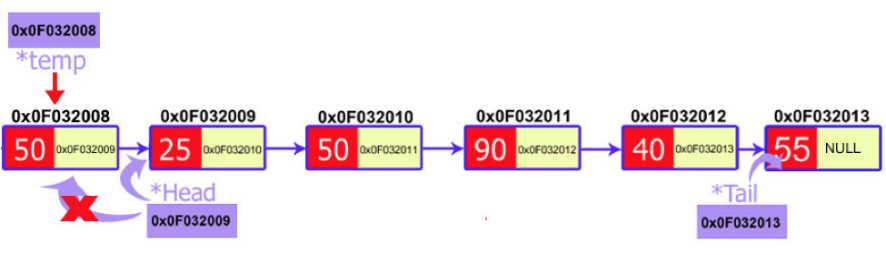
1. Allocate a new node
2. Insert new element.
3. Access address of previous node from the position you want to insert and address of current node (position node).
4. Pass the address of the new node in the next field of the previous node.
5. Pass the address of the current node in the next field of the new node.



We will access these nodes by asking the user at what position he wants to insert the new node. Now, we will start a loop to reach those specific nodes. We initialized our current node by the head and move through the linked list. At the end, we would find two consecutive nodes.

## Delete at the start

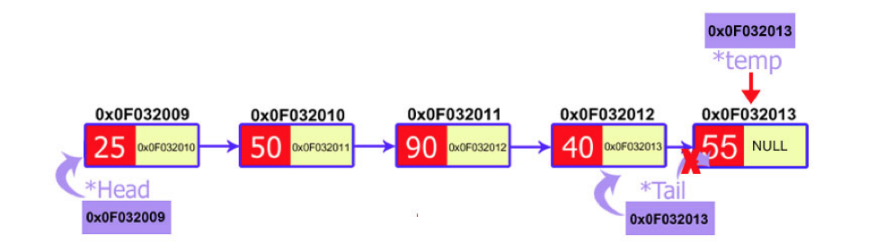
1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node
3. De allocate the first node



|  |
| --- |
| **Delete at start** |
| void delete\_first()  {  node \*temp=new node;  temp=head;  head=head->next;  delete temp; |

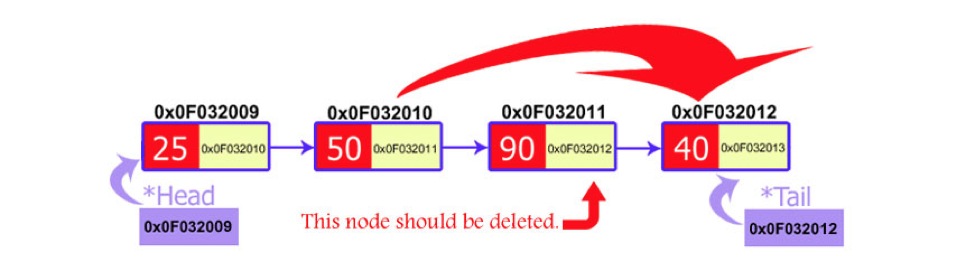
## Delete at the end

1. Make two temporary pointers (previous and current) and find nodes that comes before the last node.
2. Previous node will point to the second to the last node and the current node will point to the last node, i.e., node to be deleted.
3. Delete current node and make the previous node as the tail.



## Delete at any position

1. Make two temporary pointers (previous and current) and traverse until find specific node which needs to be deleted.
2. Delete respective node and pass the address of the node after it to the previous node.



**CAUTION!!**Always make sure that your linked list functions work correctly with an empty list.Empty Linked list is a single pointer having the value of NULL.

head = NULL;

# LAB TASK:

1. **Create template Linked List ADT and provide** 
   1. Constructor / copy constructor
   2. Destructor
   3. InsertAtStart(intval)
   4. InsertAtAnyposition(intposition,intval)
   5. insertAtEnd(intval)
   6. deleteAtstart()
   7. DeleteAtAnyposition(node)
   8. deleteAtEnd()
   9. Traverse()
   10. isEmpty()

**Note: Your program should have no memory leakage & dangling pointers. Your program should be validated properly and display warnings and errors to user as well. Reuse functions if needed to increase program efficiency.**

**TEST PLAN:**

1. **Execute your test plan. If you discover mistakes in your implementation of the List ADT, correct them and execute your test plan again.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Expected Results** | **Results/Status** |
| **1.** | Create an integer type list **myList** with default constructor | head=NULL |  |
| **2.** | Copy constructor | Head = NULL |  |
| **3.** | Check if the list isEmpty? | yes |  |
| **4.** | Insert values 1-5 in list |  |  |
| **5.** | Check if the list isEmpty? | No |  |
| **6.** | Traverse the list | 1,2,3,4,5 |  |
| **7.** | Delete from start |  |  |
| **8.** | Traverse the list | 2,3,4,5 |  |
| **9.** | Insert at Start (6) |  |  |
| **10.** | Traverse the list | 6,2,3,4,5 |  |
| **11.** | Insert at End (9) |  |  |
| **12.** | Traverse the list | 6,2,3,4,5,9 |  |
| **13.** | Delete from end |  |  |
| **14.** | Traverse the list | 6,2,3,4,5 |  |
| **15.** | Delete from position 3 |  |  |
| **16.** | Traverse the list | 6,2,4,5 |  |
| **17.** | Insert value 7 at position 4 |  |  |
| **18.** | Traverse the list | 6,2,4,7,5 |  |
| **19.** | Delete from start |  |  |
| **20.** | Traverse the list | 2,4,7,5 |  |
| **21.** | Delete from position 2 |  |  |
| **22.** | Traverse the list | 2,7,5 |  |
| **23.** | Delete from start |  |  |
| **24.** | Delete from start |  |  |
| **25.** | Delete from start |  |  |
| **26.** | Traverse the list | List is Empty |  |

**THINK:**

|  |
| --- |
| 1. What if we do not initialize head=NULL? |
| 1. Write 2 applications where linked lists are preferable than arrays. |
| 1. Can we access any node within the list directly using node’s index like arrays? |