Linked List Notes

Introduction

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions.
Linked Lists can addresses some of the limitations of arrays. A linked list is a linear data structure where each element is a separate object. Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list.

Advantages of Linked Lists

1. They are a dynamic in nature which allocates the memory when required.
2. Insertion and deletion operations can be easily implemented.
3. Stacks and queues can be easily executed.
4. Linked List reduces the access time.

Disadvantages of Linked Lists

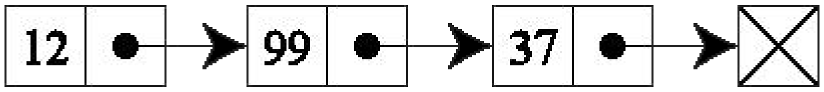
1. The memory is wasted as pointers require extra memory for storage.
2. No element can be accessed randomly; it has to access each node sequentially.
3. Reverse Traversing is difficult in linked list.

Applications of Linked Lists

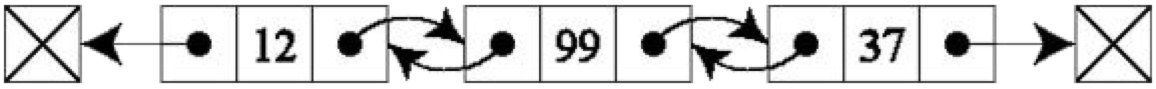
1. Linked lists are used to implement stacks, queues, graphs, etc.
2. Linked lists let you insert elements at the beginning and end of the list.
3. In Linked Lists we don’t need to know the size in advance.

Types of Linked Lists

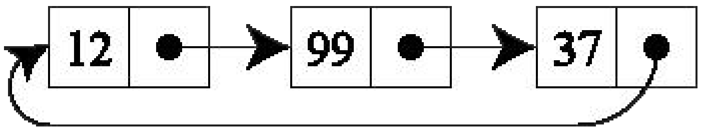
1. Singly - Each node contains only one link which points to the subsequent node in the list.



2. Doubly - It is a two-way list because one can move in either direction, either from left to right or from right to left.



3. Circular - Link field of the last node is not null, last node points to the header node.



The Node class

```
public class LinkedListNode<T> {
    T data;
    LinkedListNode<T> next;
    LinkedListNode(T data){
        this.data = data;
    }
}
```

Linked List Operations

addFirst

The method creates a node and prepends it at the beginning of the list. Steps to insert a Node at beginning :
1) The first Node is the Head for any Linked List.
2) When a new Linked List is instantiated, it just has the Head, which is Null.
3) Else, the Head holds the pointer to the first Node of the List.
4) When we want to add any Node at the front, we must make the head point to it.
5) And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6) The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
public LinkedListNode<T> addFirst(LinkedListNode<T> head, T item) {
        LinkedListNode<T> node = new LinkedListNode<>(item);
        node.next = head;
       return node;
  }
```

Traversing

Start with the head and access each node until you reach null.

```
void traverse(LinkedListNode<T> head) {
   while(head != NULL) {
        System.out.print(head.data+" ");
        head = head.next;
      }
    }
```

addLast

The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node. Steps to insert a Node at the end :

1) If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.

2) If the Linked List is not empty then we find the last node, and make it's next to the new Node, hence making the new node the last Node.

```
    public LinkedListNode<T> addLast(LinkedListNode<T> head, T item)
    {
      if(head == null){
        LinkedListNode<T> node = new LinkedListNode<>(item);
           return node;
         } else {
        LinkedListNode<T> tmp = head;
        while(tmp.next != null){
           tmp = tmp.next;
        }
        tmp.next = new LinkedListNode<T>(item);
            return head;
      }
    }
```

Inserting "after"

Find a node containing "key" and insert a new node after it.

```
    public void insertAfter(LinkedListNode<T> head, T key, T toInsert)
    {
      LinkedListNode<T> tmp = head;
      while(tmp != null && !tmp.data.equals(key))
      {
        tmp = tmp.next;
      }

      if(tmp != null){
        LinkedListNode<T> newNode  = new LinkedListNode<T> (toInsert);
        newNode.next = temp.next;
        temp.next = newNode;
       }
    }
```

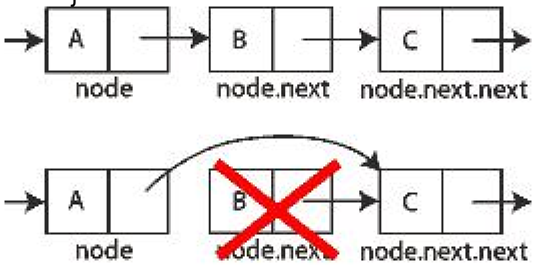
Deletion

Deleting a node can be done in many ways, like we first search the Node with data which we want to delete and then we delete it. In our approach, we will define a method which will take the data to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

1) If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
2) If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it. Find a node containing "key" and delete it.

```
  public static<T> LinkedListNode<T> deleteNode(LinkedListNode<T> head , T n){
        if(head.data == n){
           head = head.next;
           return head;
        }
        node<T> temp = head;
        node<T> tempPre = head;
        while(temp!=null && temp.data != n){
           tempPre = temp;
           temp = temp.next;
        }
        tempPre.next = temp.next;
        return head;
     }
```


A
B
C
node
node.next
node.next.next
A
B
C
node
node.next
node.next.next