Leaning Objective:

In this Module you will be learning the following:

- Linked Lists
- Types of Linked Lists
- Singly Linked Lists and it operations

Introduction:

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

Material:

Linked List

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as **"Node"**.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks, queues, graphs .. etc, can be easily implemented.
- Linked List reduces the access time.

Disadvantages of Linked Lists

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

Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Single Linked List?

Simply a list is a sequence of data, and linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

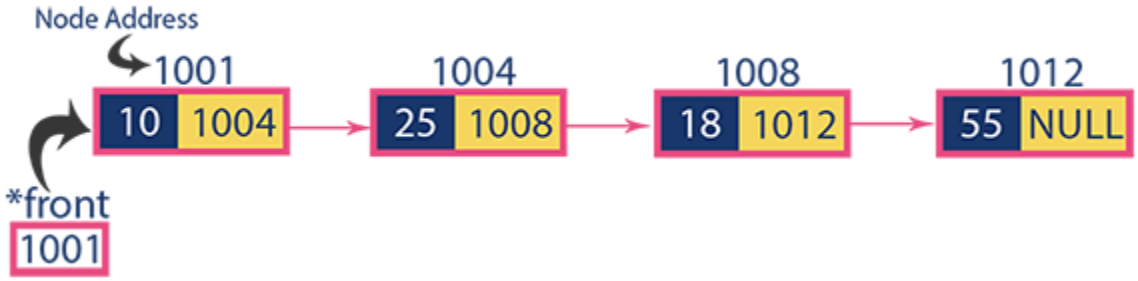
In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence. The graphical representation of a node in a single linked list is as follows...

NOTE

❋ In a single linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "head").

❋ Always next part (reference part) of the last node must be NULL.

Example



Operations

In a single linked list we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1:** Declare all the **user defined** functions.
- **Step 2:** Define a **Node** structure with two members **data** and **next**
- **Step 3:** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4:** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1:** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**).

- **Step 3:** If it is **Empty** then, set **head** = **newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6:** Set **temp** → **next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6:** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Check whether list is having only one node (**temp** → **next** == **NULL**)
- **Step 5:** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1** → **next** == **NULL**)

- **Step 5:** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
- **Step 7:** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5:** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7:** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head** → **next**) and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1** → **next == NULL**).
- **Step 11:** If **temp1** is last node then set **temp2** → **next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5:** Finally display **temp** → **data** with arrow pointing to **NULL** (**temp** → **data ---> NULL**).

Problem set:

1. **Implement singly linked list with following operations:**

   Insert Eelement

   Insert Element after a specified Element

   Insert Element before a specified Element

   Remove Element at index i

   Remove a specified Element

   PopFront (Remove element at index 0)

   PopBack (Remove the last element)

   PushFront (Add element at index 0)

   PushBack (Add element at last)

   Concat with anothe list

   Concat with another list at a specified index

   Make empty

2. Write a program to print elements of a linked list in reverse order by using same singly linked list.
3. Given two linked lists how do you find the elements which are common to both the lists. For example, if the lists are 10->5->23->33->3->11->1->20 and 1->55->243->32->3->911->21->2 then the result should be 1 and 3. Provide efficient way of solving this. Reduce the time and space complexity as much as possible.

   Hint: One way is to compare each element of first list with elements of second list. But this algorithm is not efficient. Other-way is sort to the data of both the lists and see the common elements. Yet other-way of solving this is: use hash table. Try implementing an efficient algorithm.

4. Given a linked list how do we check whether there is a loop in the list or not? [Submit the Java code and also a doc explaining why this algorithm works] Hint: Floyd Cycle Finding Algorithm
5. For the previous problem, why cannot we select a jump of 3 at a time? [Submit your analysis in DOC]
6. Given a linked list, write a Java program to swap the alternative nodes of it. You should not create a new list and copy the elements. For example, if the list is 10->5->23->33->3->11->1->20->40 then the result should be 5->10->33->23->11->3->20->1->40.