Leaning Objective:

In this Module you will be learning the following:

- Doubly Linked Lists and it operations

Introduction:

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

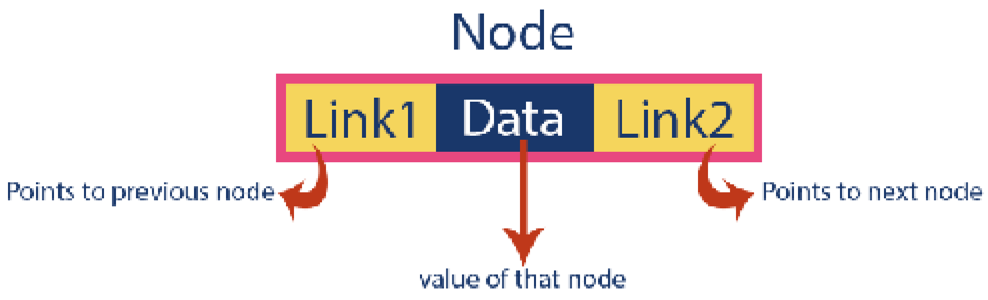
Material:

Double Linked List

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

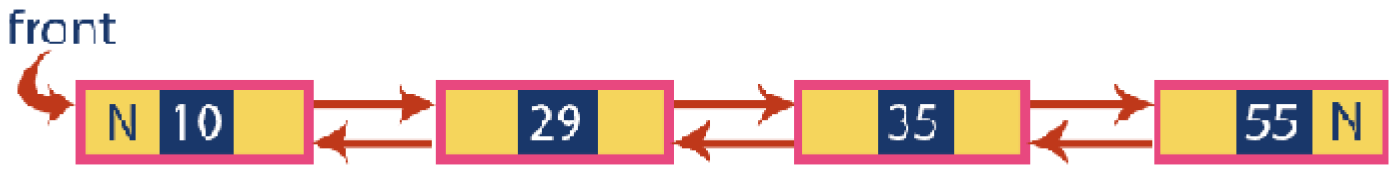
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



NOTE

⁕ In double linked list, the first node must be always pointed by **head**.

⁕ Always the previous field of the first node must be **NULL**.

⁕ Always the next field of the last node must be **NULL**.

Operations

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

<u>**Insertion**</u>

In a double linked list, the insertion operation can be performed in three ways 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 double linked list...

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

Inserting At End of the list

We can use the following steps to insert a new node at end of the double 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 assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **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:** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

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 double 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, assign **NULL** to **newNode → previous** & **newNode → next** and **newNode** to **head**.
- **Step 4:** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5:** Keep moving the **temp1** 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 **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7:** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

<u>**Deletion**</u>

In a double linked list, the deletion operation can be performed in three ways 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 double 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 → previous** is equal to **temp → next**)
- **Step 5:** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double 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 has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5:** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7:** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double 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:** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last 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 fuction.
- **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 which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9:** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10:** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).

- **Step 11:** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12:** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

We can use the following steps to display the elements of a double 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:** Display **'NULL <--- '**.
- **Step 5:** Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6:** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Problem set:

1. Implement DLL with following operations:

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns true if the list is empty, and false otherwise.
first( ):	Returns (but does not remove) the first element in the list.
last( ):	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
addAtPosition(e,pos):	Adds a new element at given position.
removeFirst( ):	Removes and returns the first element of the list.
removeLast( ):	Removes and returns the last element of the list.
removeAtPosition(pos):	Removes and returns the given position element of the list.

2. Implement the Following operation to the previous DLL?

   1. moveToFront

      moveToFront remove the given item node in the DLL and add that node at Front position.

   2. moveToBack

      moveToBack remove the given item node in the DLL and add that node at Back position.

   3. Splice

      We implement all basic list operations in terms of the single operation splice shown in following Fig. splice cuts out a sublist from one list and inserts to the given position. The sublist is specified by handles a and b to its first and its last element, respectively. In other words, b must be reachable from a by following zero or more next pointers but without going through the dummy item. splice does not change the number of items in the system.

      If the position is 0 means insert at Head, if the position is -1 means insert at Tail, if the position is 'T' means insert at Tth position

```
// Remove $\langle a,\dots,b\rangle$ from its current list and insert it after $t$
// $\dots,a',a,\dots,b,b',\dots + \dots,t,t',\dots \mapsto \dots,a',b',\dots + \dots,t,a,\dots,b,t',\dots$
Procedure splice(a,b,t : Handle)
    assert a and b belong to the same list, b is not before a, and $t \notin \langle a,\dots,b\rangle$

    // cut out $\langle a,\dots,b\rangle$
    $a' := a \to prev$
    $b' := b \to next$
    $a' \to next := b'$
    $b' \to prev := a'$

    // insert $\langle a,\dots,b\rangle$ after $t$
    $t' := t \to next$

    $b \to next := t'$
    $a \to prev := t$

    $t \to next := a$
    $t' \to prev := b$
```

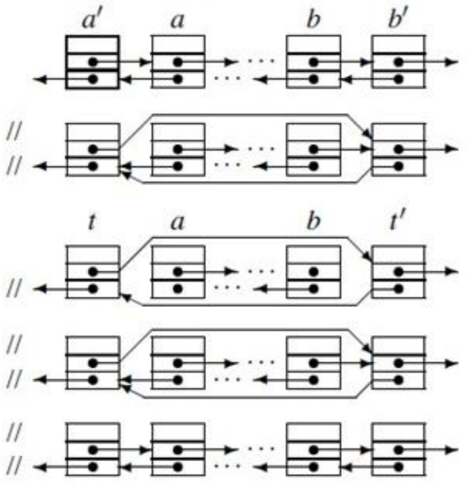


Fig. 3.3. Splicing lists

   4. Swap

      Swap operation swaps two sublists in constant time, i.e., sequences

$$(\langle\dots,a',a,\dots,b,b',\dots\rangle,\langle\dots,c',c,\dots,d,d',\dots\rangle)$$

are transformed into

$$(\langle\dots,a',c,\dots,d,b',\dots\rangle,\langle\dots,c',a,\dots,b,d',\dots\rangle)$$

5. Rotate

Rotate the DLL to the right $\langle a,\ldots,b,c\rangle \mapsto \langle c,a,\ldots,b\rangle$. Generalize your algorithm to rotate $\langle a,\ldots,b,c,\ldots,d\rangle$ to $\langle c,\ldots,d,a,\ldots,b\rangle$ in constant time.

6. Sort

Sort the DLL.