PW SKILLS

Lesson Plan

Linked List-4

PW SKILLS

A doubly linked list in Java is a data structure where each element, called a node, contains three components: the data it holds, a reference to the previous node, and a reference to the next node. This structure allows traversal in both forward and backward directions.

Inserting a new node in a doubly linked list is very similar to inserting new node in linked list. There is a little extra work required to maintain the link of the previous node. A node can be inserted in a Doubly Linked List in four ways:

- At the front of the DLL.
- In between two nodes
  - After a given node.
  - Before a given node.
- At the end of the DLL.

Add a node at the front in a Doubly Linked List:

The new node is always added before the head of the given Linked List. The task can be performed by using the following 5 steps:

- Firstly, allocate a new node (say new_node).
- Now put the required data in the new node.
- Make the next of new_node point to the current head of the doubly linked list.
- Make the previous of the current head point to new_node.
- Lastly, point head to new_node.

Code:

```
// Adding a node at the front of the list
public void push(int new_data)
{
 // 1. allocate node
 // 2. put in the data */
 Node new_Node = new Node(new_data);

 // 3. Make next of new node as head and previous as NULL
 new_Node.next = head;
 new_Node.prev = null;

 // 4. change prev of head node to new node
 if (head =/= null)
  head.prev = new_Node;

 // 5. move the head to point to the new node
 head = new_Node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node in between two nodes:

It is further classified into the following two parts:

Add a node after a given node in a Doubly Linked List:

We are given a pointer to a node as prev_node, and the new node is inserted after the given node. This can be done using the following 6 steps:

- Firstly create a new node (say new_node).
- Now insert the data in the new node.
- Point the next of new_node to the next of prev_node.
- Point the next of prev_node to new_node.
- Point the previous of new_node to prev_node.
- Change the pointer of the new node's previous pointer to new_node.

Code:

```
// Given a node as prev_node, insert a new node
// after the given node
public void InsertAfter(Node prev_Node, int new_data)
{
 // Check if the given prev_node is NULL
 if (prev_Node == null) {
  System.out.println(
   "The given previous node cannot be NULL ");
  return;
 }

 // 1. allocate node
 // 2. put in the data
 Node new_node = new Node(new_data);

 // 3. Make next of new node as next of prev_node
 new_node.next = prev_Node.next;

 // 4. Make the next of prev_node as new_node
 prev_Node.next = new_node;

 // 5. Make prev_node as previous of new_node
 new_node.prev = prev_Node;

 // 6. Change previous of new_node's next node
 if (new_node.next ≠ null)
  new_node.next.prev = new_node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node before a given node in a Doubly Linked List:

Let the pointer to this given node be next_node. This can be done using the following 6 steps.

- Allocate memory for the new node, let it be called new_node.
- Put the data in new_node.
- Set the previous pointer of this new_node as the previous node of the next_node.
- Set the previous pointer of the next_node as the new_node.
- Set the next pointer of this new_node as the next_node.
- Now set the previous pointer of new_node.
- If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node.
- Else, if the prev of new_node is NULL, it will be the new head node.

Code:

```
// Given a node as prev_node, insert a new node
// after the given node
public void InsertBefore(Node next_Node, int new_data)
{
 // Check if the given next_node is NULL
 if (next_Node == null) {
  System.out.println(
   "The given next node cannot be NULL ");
  return;
 }

 // 1. Allocate node
 // 2. Put in the data
 Node new_node = new Node(new_data);

 // 3. Make previous of new node as previous of prev_node
 new_node.prev = next_Node.prev;

 // 4. Make the prev of next_node as new_node
 next_Node.prev = new_node;

 // 5. Make next_node as next of new_node
 new_node.next = next_Node;

 // 6. Change next of new_node's previous node
 if (new_node.prev != null)
  new_node.prev.next = new_node;
 else
  head = new_node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node at the end in a Doubly Linked List:

The new node is always added after the last node of the given Linked List. This can be done using the following 7 steps:

- Create a new node (say new_node).
- Put the value in the new node.
- Make the next pointer of new_node as null.
- If the list is empty, make new_node as the head.
- Otherwise, travel to the end of the linked list.
- Now make the next pointer of last node point to new_node.
- Change the previous pointer of new_node to the last node of the list.

Code:

```
// Add a node at the end of the list
void append(int new_data)
{
 // 1. allocate node
 // 2. put in the data
 Node new_node = new Node(new_data);

 Node last = head; /* used in step 5*/

 // 3. This new node is going to be the last node, so
 // make next of it as NULL
 new_node.next = null;

 // 4. If the Linked List is empty, then make the new
 // node as head
 if (head == null) {
  new_node.prev = null;
  head = new_node;
  return;
 }

 // 5. Else traverse till the last node
 while (last.next != null)
  last = last.next;

 // 6. Change the next of last node
 last.next = new_node;

 // 7. Make last node as previous of new node
 new_node.prev = last;
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

The deletion of a node in a doubly-linked list can be divided into three main categories:

- After the deletion of the head node.
- After the deletion of the middle node.
- After the deletion of the last node.

All three mentioned cases can be handled in two steps if the pointer of the node to be deleted and the head pointer is known.

If the node to be deleted is the head node then make the next node as head.
If a node is deleted, connect the next and previous node of the deleted node.

Algorithm:

Let the node to be deleted be del.
If node to be deleted is head node, then change the head pointer to next current head.
if headnode == del then
    headnode = del.nextNode
Set prev of next to del, if next to del exists.
if del.nextNode != none
    del.nextNode.previousNode = del.previousNode
Set next of previous to del, if previous to del exists.
if del.previousNode != none
    del.previousNode.nextNode = del.next
Below is the implementation of the above approach:

Code:

```
// Java program to delete a node from
// Doubly Linked List

// Class for Doubly Linked List
public class DLL {
 Node head; // head of list

 /* Doubly Linked list Node*/
 class Node {
  int data;
  Node prev;
  Node next;

  // Constructor to create a new node
  // next and prev is by default initialized
  // as null
  Node(int d) { data = d; }
 }

 // Adding a node at the front of the list
 public void push(int new_data)
 {
```

```
 // 1. allocate node
 // 2. put in the data
 Node new_Node = new Node(new_data);

 // 3. Make next of new node as head
 // and previous as NULL
 new_Node.next = head;
 new_Node.prev = null;

 // 4. change prev of head node to new node
 if (head ≠ null)
  head.prev = new_Node;

 // 5. move the head to point to the new node
 head = new_Node;
}

// This function prints contents of linked list
// starting from the given node
public void printlist(Node node)
{
 Node last = null;

 while (node ≠ null) {
  System.out.print(node.data + " ");
  last = node;
  node = node.next;
 }

 System.out.println();
}

// Function to delete a node in a Doubly Linked List.
// head_ref ⟶ pointer to head node pointer.
// del ⟶ data of node to be deleted.
void deleteNode(Node del)
{

 // Base case
 if (head == null || del == null) {
  return;
 }

 // If node to be deleted is head node
 if (head == del) {
  head = del.next;
 }

 // Change next only if node to be deleted
 // is NOT the last node
 if (del.next ≠ null) {
```

```
    del.next.prev = del.prev;
   }

   // Change prev only if node to be deleted
   // is NOT the first node
   if (del.prev ≠ null) {
    del.prev.next = del.next;
   }

   // Finally, free the memory occupied by del
   return;
  }

  // Driver Code
  public static void main(String[] args)
  {
   // Start with the empty list
   DLL dll = new DLL();

   // Insert 2. So linked list becomes 2→NULL
   dll.push(2);

   // Insert 4. So linked list becomes 4→2→NULL
   dll.push(4);

   // Insert 8. So linked list becomes 8→4→2→NULL
   dll.push(8);

   // Insert 10. So linked list becomes
   // 10→8→4→2→NULL
   dll.push(10);

   System.out.print("Original Linked list ");
   dll.printlist(dll.head);


   dll.deleteNode(dll.head); /*delete first node*/
   dll.deleteNode(dll.head.next); /*delete middle node*/
   dll.deleteNode(dll.head.next); /*delete last node*/

   System.out.print(
    "\nModified Linked list ");
   dll.printlist(dll.head);
  }
 }
```

Output:

Original Linked list 10 8 4 2
Modified Linked list 8

Complexity Analysis:

Time Complexity: O(1).
Since traversal of the linked list is not required, the time complexity is constant.
Auxiliary Space: O(1).
As no extra space is required, the space complexity is constant.

Q1. Split linked list in parts [Leetcode 725]

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val;
this.next = next; }
 * }
 */
class Solution {
    public ListNode[] splitListToParts(ListNode head, int k) {
        ListNode cur = head;
        int N = 0;
        while (cur != null) {
            cur = cur.next;
            N++;
        }

        int width = N / k, rem = N % k;

        ListNode[] ans = new ListNode[k];
        cur = head;
        for (int i = 0; i < k; ++i) {
            ListNode head1 = new ListNode(0), write = head1;
            for (int j = 0; j < width + (i < rem ? 1 : 0); ++j) {
                write = write.next = new ListNode(cur.val);
                if (cur != null) cur = cur.next;
            }
            ans[i] = head1.next;
        }
        return ans;


    }
}
```

Q2. Leetcode 2058

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val;
this.next = next; }
 * }
 */
class Solution {
    public int[] nodesBetweenCriticalPoints(ListNode head) {
        int ans[] = new int[2];
        ans[0]= -1;
        ans[1] = -1;

        if(head==null || head.next==null || head.next.next==null)
return ans;

        ArrayList<Integer> arr = new ArrayList<>();
        ListNode t = head.next;

        ListNode prev = head;
        int idx = 1;

        while(t.next!=null){
            if(t.val>prev.val && t.val>t.next.val) arr.add(idx);
            if(t.val<prev.val && t.val<t.next.val) arr.add(idx);

            idx++;
            prev = t;
            t=t.next;
        }

        if(arr.size()<2) return ans;

        ans[1] = arr.get(arr.size()-1) - arr.get(0);
        int min = Integer.MAX_VALUE;

        for(int i=1; i<arr.size(); i++){
            min = Math.min(arr.get(i)-arr.get(i-1),min);
        }
        ans[0] = min;
        return ans;

    }
}
```

Q3. Leetcode 2074

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val;
this.next = next; }
 * }
 */
class Solution {

    static void reverse(ArrayList<Integer> arr, int i, int j){

        while(i<j){
            int a = arr.get(i);
            arr.set(i,arr.get(j));
            arr.set(j,a);
            i++;
            j--;
        }
    }

    public ListNode reverseEvenLengthGroups(ListNode head) {

        if(head.next==null) return head;

        ArrayList<Integer> arr = new ArrayList<>();

        ListNode t = head;

        while(t!=null){
            arr.add(t.val);
            t=t.next;
        }

        int prev = 2;
        int idx = -1;

        for(int i=1; i+prev-1<arr.size();){
            reverse(arr,i,i+prev-1);
            if(i+2*prev+1>=arr.size()) idx = i+prev;
            else{
                idx = i+2*prev+1;
            }
            i=i+2*prev+1;
            prev=prev+2;
        }
```

```
        if(arr.size()-idx>0 && (arr.size()-idx)%2==0){
            reverse(arr,idx,arr.size()-1);
        }



        ListNode ans = new ListNode(-1);
        t=ans;
        for(int i=0; i<arr.size(); i++){
            t.next = new ListNode(arr.get(i));
            t=t.next;
        }

        return ans.next;

    }
}
```

Q4. Leetcode 138

```
/*
// Definition for a Node.
class Node {
    int val;
    Node next;
    Node random;

    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
    }
}
*/



        return ch;
    }class Solution {
    public Node createNode(int val) {
        Node nn = new Node(val);
        nn.next = null;
        nn.random = null;
        return nn;
    }

    public Node copyRandomList(Node head) {
        if (head == null)
            return null;
```

```
        Node o = head;
        Node c = null;
        Node ch = null;
        Node t;

        while (o ≠ null) {
            t = createNode(o.val);
            t.next = o.next;
            o.next = t;
            o = t.next;
        }
        o = head;
        while (o ≠ null) {
            c = o.next;
            if (o.random ≠ null) {
                c.random = o.random.next;
            }
            o = c.next;
        }

        o = head;
        c = head.next;
        ch = c;

        while (o ≠ null && c ≠ null) {
            o.next = c.next;
            o = o.next;

            if (o ≠ null) {
                c.next = o.next;
                c = c.next;
            }
        }


}
```

Q5. Leetcode 430

```
class Solution {

    public Node flatten(Node head) {
        if (head == null) {
            return null;
        }

        Node current = head;
        Stack<Node> stack = new Stack<>();
```

```
        while (current != null) {
            if (current.child != null) {
                Node nextNode = current.next;

                // Connect current node to the child list
                current.next = current.child;
                current.child.prev = current;
                current.child = null;

                // Push the next node onto the stack for later
processing
                if (nextNode != null) {
                    stack.push(nextNode);
                }
            } else if (current.next == null && !stack.isEmpty())
{
                // If there are no more nodes in the current
level,
                // pop a node from the stack and connect it to
the current node
                Node nextNode = stack.pop();
                current.next = nextNode;
                nextNode.prev = current;
            }

            current = current.next;
        }

        return head;
    }
}
```

PW SKILLS

THANK YOU !

PW SKILLS