Question Bank

Doubly Linked List

Objective Type Questions

1. In doubly linked list
   a) A pointer is maintained to store both next and previous nodes.
   b) Two pointers are maintained to store next and previous nodes.
   c) A pointer to self is maintained for each node.
   d) None of the above
2. In a doubly linked list, the number of pointers affected for an insertion operation will be
   a) 4
   b) 0
   c) 1
   d) Depends upon the nodes of the doubly linked list
3. Consider the following doubly linked list: head-1-2-3-4-5-tail. What will be the list after performing the given sequence of operations?

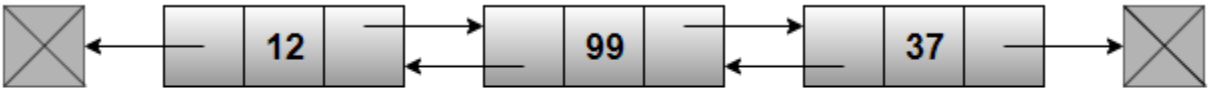
```
Node temp = new Node(6,head,head.getNext());
head.setNext(temp);
temp.getNext().setPrev(temp);
Node temp1 = tail.getPrev();
tail.setPrev(temp1.getPrev());
temp1.getPrev().setNext(tail);
```

   a) head-6-1-2-3-4-5-tail
   b) head-6-1-2-3-4-tail
   c) head-1-2-3-4-5-6-tail
   d) head-1-2-3-4-5-tail
4. Which of the following is optimal to find an element at the kth position at the linked list?
   a. Singly linked lists
   b. Doubly linked lists
   c. Circular linked lists
   d. Array implementation of linked list
5. What is the worst case time complexity of inserting a node in a doubly linked list?
   a) O(nlogn)
   b) O(logn)
   c) O(n)
   d) O(1)

Understanding question [Blooms level two]

1. What do you understand by Doubly Linked List?
   The doubly linked list includes a pointer (link) to the next node as well as to the previous node in the list. The two links between the nodes may be called "forward" and "backward, "or "next" and "prev (previous)." A doubly linked list is shown below whose

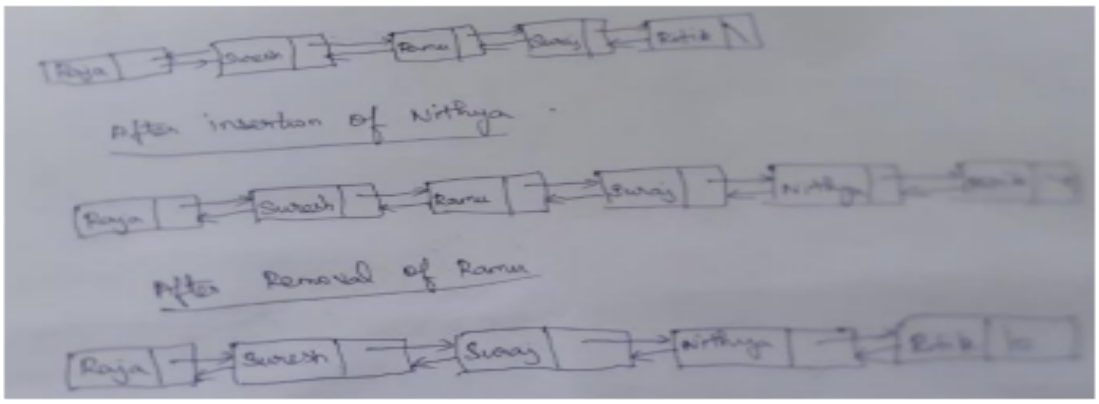
nodes consist of three fields: an integer value, a link that points to the next node, and a link that points to the previous node.



Doubly Linked List

A technique (known as **XOR-linking**) is used to allow a doubly-linked list to be implemented with the help of a single link field in each node. However, this technique needs more ability to perform some operations on addresses, and therefore may not be available for some high-level languages.
Most of the modern operating systems use doubly linked lists to maintain references to active threads, processes, and other dynamic objects.

2. Raja, Suresh, Ramu, Suraj, and Ritik are friends. They are playing a ball-passing game in the holidays, and they were allowed to pass a ball in bidirectional mode. After some time, Nithya enters after Suraj. After adding Nithya, Ramu was removed from the game. Illustrate the insertion and deletion operation for the given scenario with suitable pseudo code.



Insertion at specific position at doubly linked list

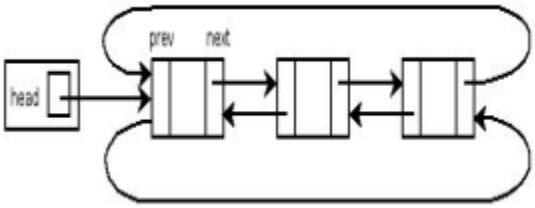
```
Struct node *insert At Postion (Struct node *head, int data, int item)
{
Struct node *temp, *P;
temp=(Struct node*)malloc(Sizeof(struct node));
temp->info=data;
P=head;
While(P!=NULL)
{
If(P->info==item)
{
temp->previous=P;
Temp->next=P->next;
If(P->next->previous=temp);
```

```
P->next=temp;
Return head;
}
P= P->next;
}
Print("%d not present in the list \n \n", item); return head;
}
```

Deletion of a node in the middle

```
temp=head->next;
While(temp->next!=NULL)
{
If(temp->info==data)
{
temp->previous->next=temp->next;
temp->next->previous=temp->previous;
free(temp);
return head;
}
temp=temp->next;
}
```

3. When doubly linked list can be represented as circular linked list?
   In a doubly linked list, all nodes are connected with forward and backward links to the next and previous nodes respectively. In order to implement circular linked lists from doubly linked lists, the first node's previous field is connected to the last node and the last node's next field is connected to the first node.



4. Explain the operations of doubly linked list.

Below are operations on the given DLL:

- **Add a node at the front of DLL:** The new node is always added before the head of the given Linked List. And the newly added node becomes the new head of DLL & maintaining a global variable for counting the total number of nodes at that time.

- Traversal of a Doubly linked list
- **Insertion of a node:** This can be done in three ways:

a. **At the beginning:** The new created node is insert in before the head node and head points to the new node.
b. **At the end:** The new created node is insert at the end of the list and tail points to the new node.
c. **At a given position:** Traverse the given DLL to that position(**let the node be X**) then do the following:
    i. Change the next pointer of new Node to the next pointer of Node X.
    ii. Change the prev pointer of next Node of Node X to the new Node.
    iii. Change the next pointer of node X to new Node.
    iv. Change the prev pointer of new Node to the Node X.

Scenario Based Questions

1. N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease-key operation, a pointer is provided to the record on which the operation is to be performed. An algorithm performs the following operations on the list in this order: O(N) delete, O(log N) insert, O(log N) find, and O(N) decrease-key. What is the time complexity of all these operations puts together?

   Here, it is given that pointer is provided to the record on which the decrease key operation is to be performed. So, time complexity of decrease key operation is $\Theta(1)$

   For insert – $\Theta(N)$, we need to insert the element at the end of the sorted list.

   For delete – $\Theta(1)$ time is needed, as pointer is directly given

   For decrease key – $\Theta(N)$, because after decreasing the key value, all the elements must be sorted again.

   Now combining all the operations together:

   For delete = $\Theta(N)* \Theta(1)$ [because $\Theta(N)$ delete operations]

   For insert = $\Theta(\log N)*\Theta(N)$ [because $\Theta(\log N$ insert operations are given)]

   For find = $\Theta(\log N)*\Theta(N)$

   For decrease-key = $\Theta(N)* \Theta(N)$

   So, overall complexity = $\Theta(N)* \Theta(1) + \Theta(\log N)* \Theta(N) + \Theta(\log N)* \Theta(N)+ \Theta(N)* \Theta(N)$

   Time complexity of all these operations put together = $\Theta(N^2)$

2. You are given a doubly-linked list of size ‘N’ consisting of positive integers. Now your task is to reverse it and display the reversed list.

```
#include <stdio.h>
```

```
#include <stdlib.h>
 // A Doubly Linked List Node
struct Node
{
   int data;
   struct Node *next, *prev;
};

// Utility function to push a node at the beginning of the doubly linked list
void push(struct Node** headRef, int key)
{
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = key;
   node->prev = NULL;
   node->next = *headRef;

   // change `prev` of the existing head node to point to the new node
   if (*headRef != NULL) {
      (*headRef)->prev = node;
   }

   // update head pointer
   *headRef = node;
}

// Helper function to print nodes of a doubly linked list
void printDDL(char* msg, struct Node* head)
{
   printf("%s: ", msg);
   while (head != NULL)
   {
      printf("%d —> ", head->data);
      head = head->next;
   }

   printf("NULL\n");
}

// Function to swap `next` and `prev` pointers of the given node
void swap(struct Node* node)
{
   struct Node* prev = node->prev;
   node->prev = node->next;
   node->next = prev;
}
```

```
// Function to reverse a doubly-linked list
void reverseDDL(struct Node** headRef)
{
    struct Node* prev = NULL;
    struct Node* curr = *headRef;

    // traverse the list
    while (curr != NULL)
    {
        // swap `next` and `prev` pointers for the current node
        swap(curr);

        // update the previous node before moving to the next node
        prev = curr;

        // move to the next node in the doubly linked list
        // (advance using `prev` pointer since `next` and `prev` pointers were swapped)
        curr = curr->prev;
    }

    // update head pointer to the last node
    if (prev != NULL) {
        *headRef = prev;
    }
}

int main(void)
{
    int keys[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(keys)/sizeof(keys[0]);

    struct Node* head = NULL;
    for (int i = 0; i < n; i++) {
        push(&head, keys[i]);
    }

    printDDL("Original list", head);
    reverseDDL(&head);
    printDDL("Reversed list", head);

    return 0;
        }
```

Output:

```
Original list: 5 -> 4 -> 3 -> 2 -> 1 -> NULL
Reversed list: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
```