UNIT III

Linked Lists: Singly Linked Lists, Doubly Linked Lists, Circular Lists

Hashing: Static Hashing, Hash Tables, Hash Functions, Overflow Handling.

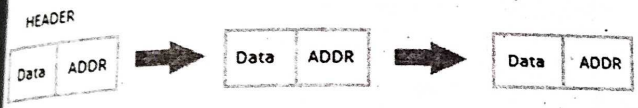
3.1 Linked Lists

3.1.1 Singly Linked Lists

Q1. What are linked lists? Write about it. **(Imp.)**

Ans :

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.



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 and queues can be easily executed.
- 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.

Applications of Linked Lists

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

Q2. Write the ADT of linked list.

Ans :

Linked List Abstrct Data Type

Although a linked list can be implemented in a variety of ways, the most fl exible implementation is by using pointers.

- Create() – it creates a linked list
- Traverse()- it traverse the entire list
- Insert(int data, position) - it insert the new element to the location at a given position.
- append(int data)- appends the new data element at the end of the lis
- Delete(int position) – deletes the list from the given position
- Reverse() – it reverses the list.

Q3. Explain the operations that can be performed on the linked lists. **(Imp.)**

Ans :

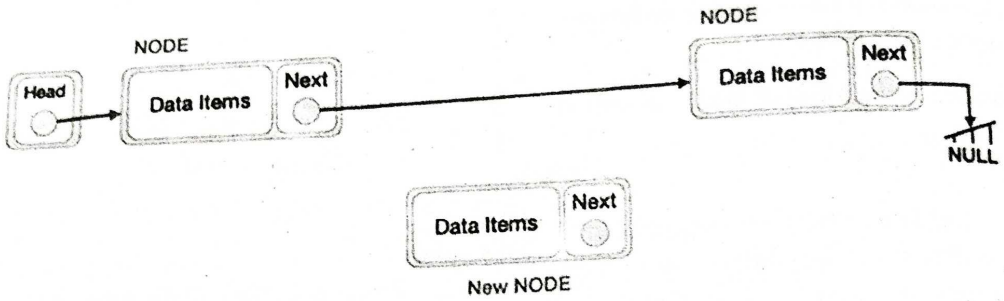
Basic Operations of Linked list

Following are the basic operations supported by a list.

- **Insertion**: Adds an element at the beginning of the list.
- **Deletion**: Deletes an element at the beginning of the list.
- **Display**: Displays the complete list.
- **Search**: Searches an element using the given key.
- **Delete**: Deletes an element using the given key.

Insertion Operation

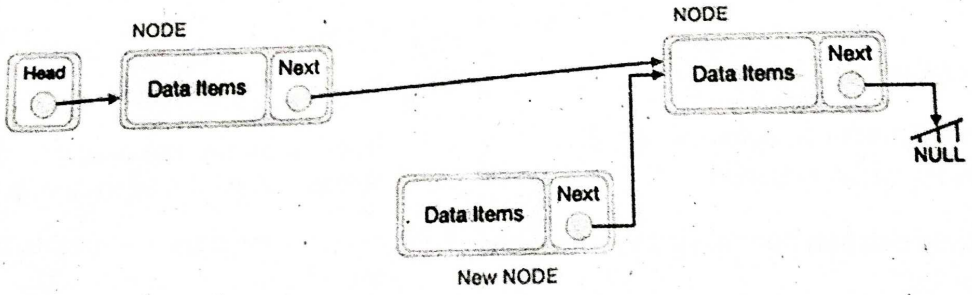
Adding a new node in linked list is a more than one step activity. We shall learn this with diagram here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C "

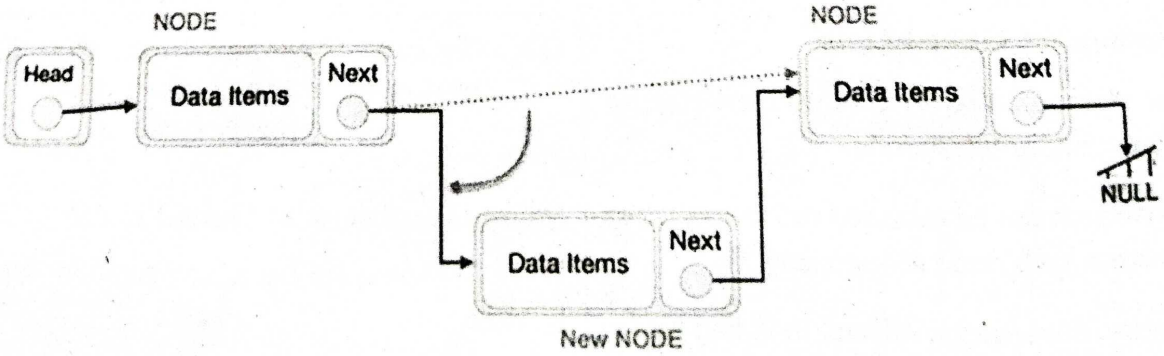
NewNode.next "> RightNode;

It should look like this "

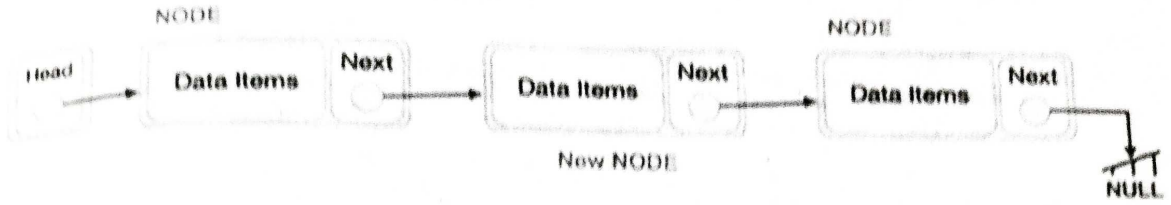


Now, the next node at the left should point to the new node.

LeftNode.next "> NewNode;



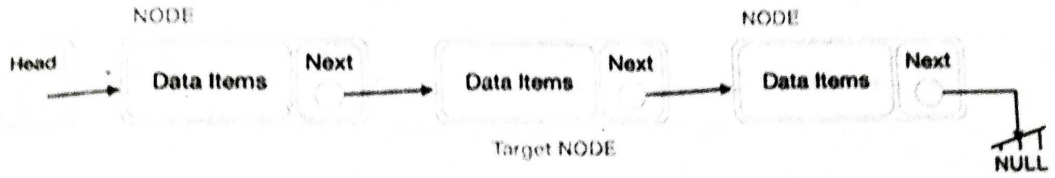
This will put the new node in the middle of the two. The new list should look like this "



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

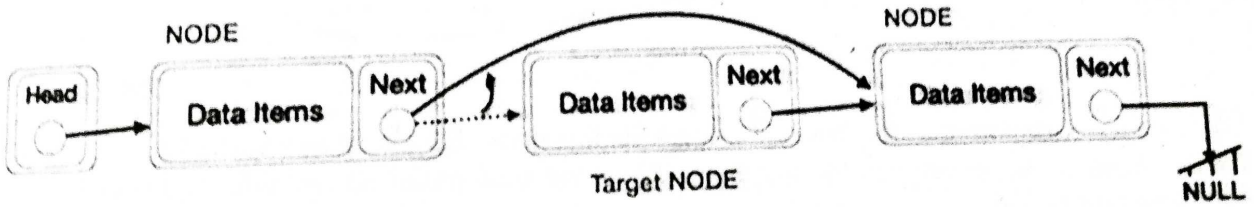
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



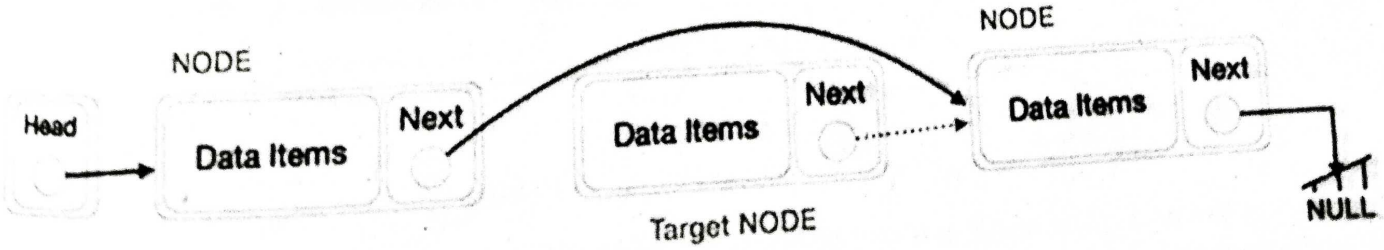
The left (previous) node of the target node now should point to the next node of the target node

LeftNode.next –> TargetNode.next;

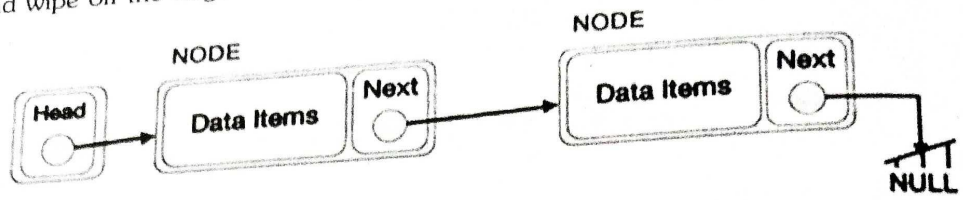


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next –> NULL;

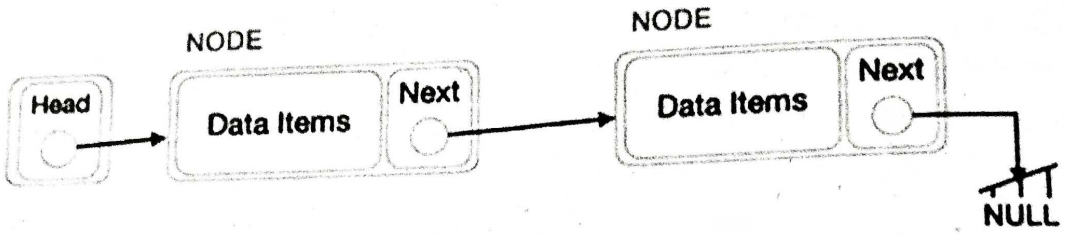


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

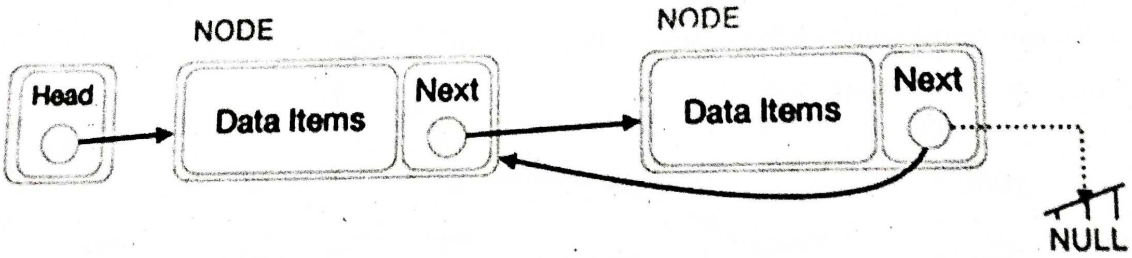


Reverse Operation

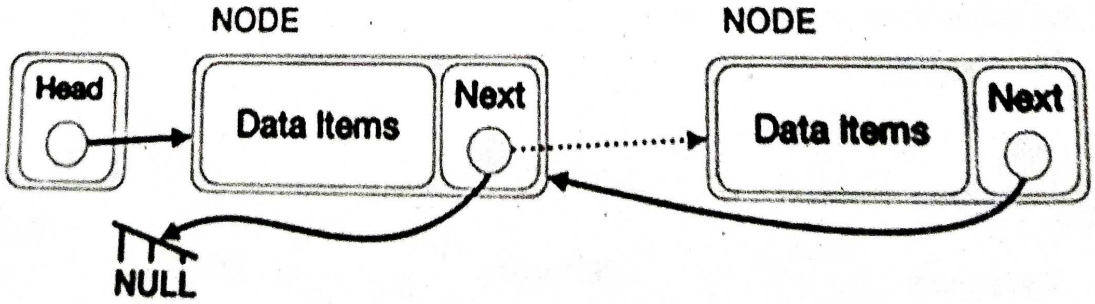
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



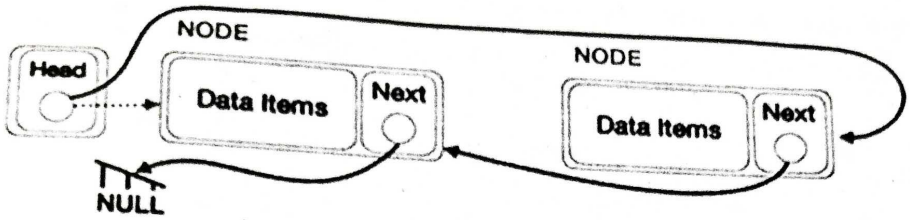
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node "



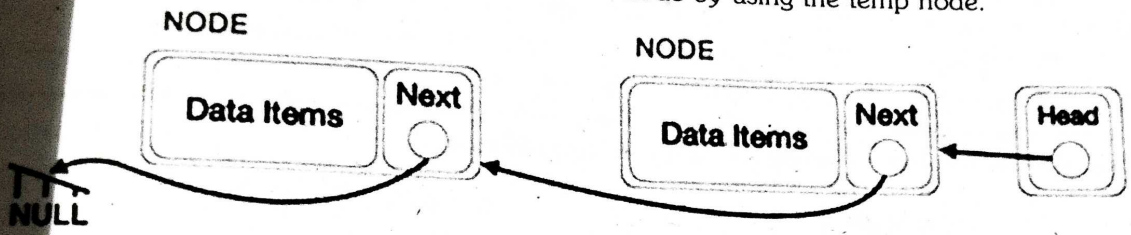
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



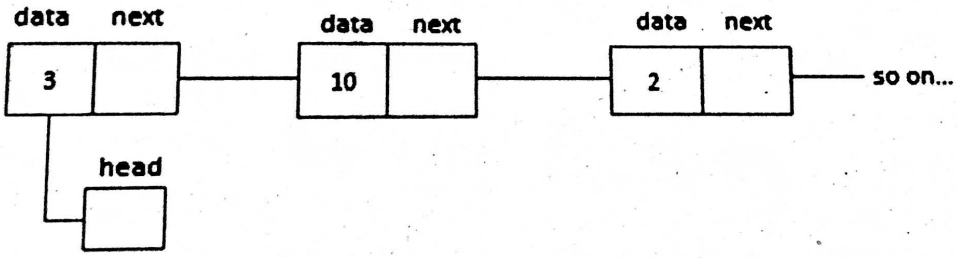
Q4. What are various types of linked lists.

Ans :

Types of Linked Lists

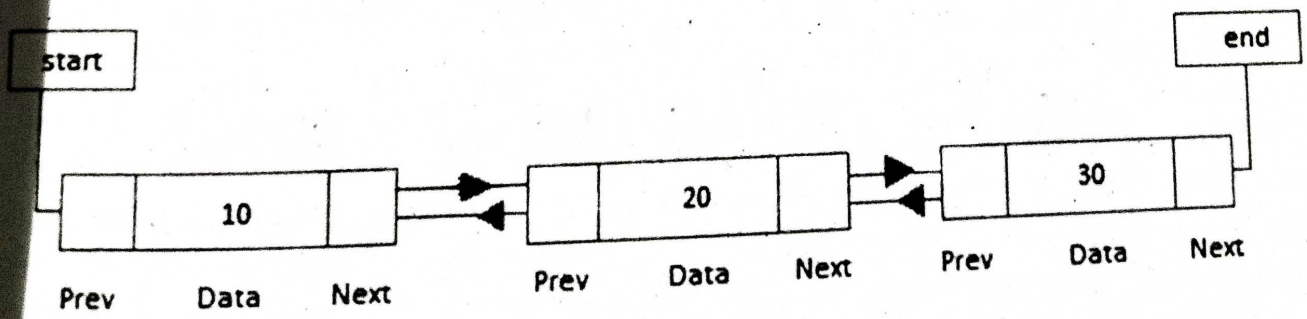
Singly Linked List

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



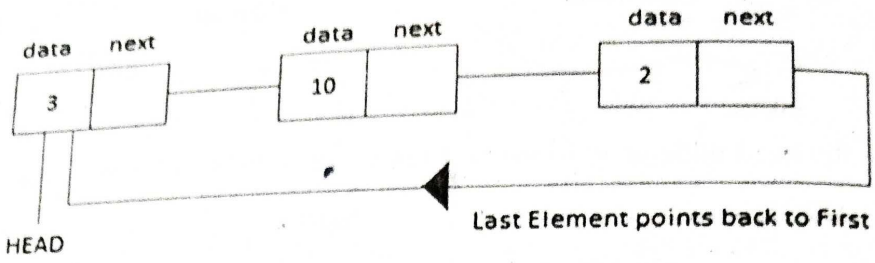
Doubly Linked List

In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



Circular Linked List

In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



Q5. Explain about singly linked list with its operations.

Ans :-

Singly Linked List

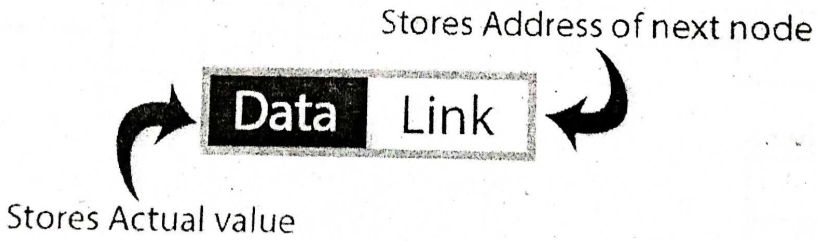
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other.

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.

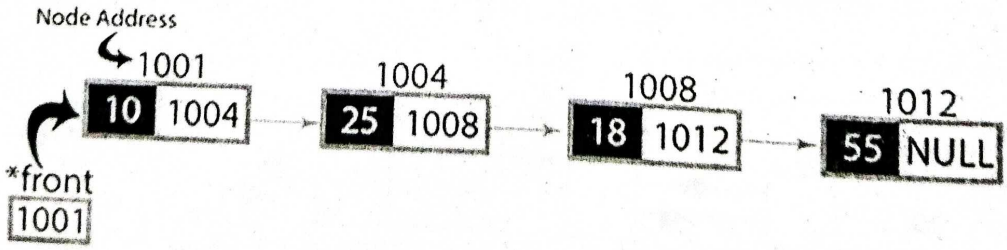
In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows.



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

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:** Include all the header files which are used in the program.
- **Step 2:** Declare all the user defined functions.
- **Step 3:** Define a Node structure with two members data and next
- **Step 4:** Define a Node pointer 'head' and set it to NULL.
- **Step 5:** 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 new Node'!next = NULL and head = newNode.
- **Step 4:** If it is Not Empty then, set new Node → 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).

Q6. Write a C++ program to implement singly linked list. **(Imp.)**

Ans :

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
  int info;
  struct node *next;
}*start;
/*
 * Class Declaration
 */
class single_llist
{
  public:
    node* create_node(int);
    void insert_begin();
    void insert_pos();
    void insert_last();
    void delete_pos();
    void sort();
    void search();
    void update();
    void reverse();
    void display();
    single_llist()
    {
      start = NULL;
    }
};

/*
 * Main :contains menu
 */
main()
{
  int choice, nodes, element, position, i;
  single_llist sl;
  start = NULL;
  while (1)
  {
    cout<<endl<<"......."<<endl;
    cout<<endl<<"Operations on singly linked
    list"<<endl;
    cout<<endl<<"......."<<endl;
```

```
    cout<<"1.Insert Node at beginning"<<endl;
    cout<<"2.Insert node at last"<<endl;
    cout<<"3.Insert node at position"<<endl;
    cout<<"4.Sort Link List"<<endl;
    cout<<"5.Delete a Particular Node"<<endl;
    cout<<"6.Update Node Value"<<endl;
    cout<<"7.Search Element"<<endl;
    cout<<"8.Display Linked List"<<endl;
    cout<<"9.Reverse Linked List "<<endl;
    cout<<"10.Exit "<<endl;
    cout<<"Enter your choice : ";
    cin>>choice;
    switch(choice)
    {
    case 1:
    cout<<"Inserting Node at Beginning:
    "<<endl;
      sl.insert_begin();
      cout<<endl;
      break;
    case 2:
      cout<<"Inserting Node at Last: "<<endl;
      sl.insert_last();
      cout<<endl;
      break;
    case 3:
    cout<<"Inserting Node at a given position:"
    <<endl;
      sl.insert_pos();
      cout<<endl;
      break;
    case 4:
      cout<<"Sort Link List: "<<endl;
      sl.sort();
      cout<<endl;
      break;
    case 5:
    cout<<"Delete a particular node: "<<endl;
      sl.delete_pos();
      break;
    case 6:
      cout<<"Update Node Value:"<<endl;
      sl.update();
      cout<<endl;
      break;
    case 7:
      cout<<"Search element in Link List:
      "<<endl;
      sl.search();
      cout<<endl;
      break;
    case 8:
      cout<<"Display elements of link list"
          <<endl;
      sl.display();
      cout<<endl;
      break;
    case 9:
      cout<<"Reverse elements of Link List"
          <<endl;
      sl.reverse();
      cout<<endl;
      break;
    case 10:
      cout<<"Exiting..."<<endl;
      exit(1);
      break;
    default:
      cout<<"Wrong choice"<<endl;
    }
  }
}
```



```
/*
 * Creating Node
 */
node *single_llist::creat
{
  struct node *temp,
  temp = new(struct
  if (temp == NULL
  {
    cout<<"Memo
    return 0;
  }
  else
  {
    temp->info =
    temp->next
    return temp;
  }
}

/*
 * Inserting elem
 */
void single_llist::
{
  int value;
  cout<<"En
  cin>>value;
  struct node
  temp = cre
  if (start ==
  {
    start = te
    start->n
  }
```

```
/*
 * Creating Node
 */
node *single_llist::create_node(int value)
{
  struct node *temp, *s;
  temp = new(struct node);
  if (temp == NULL)
  {
    cout<<"Memory not allocated "<<endl;
    return 0;
  }
  else
  {
    temp->info = value;
    temp->next = NULL;
    return temp;
  }
}

/*
 * Inserting element in beginning
 */
void single_llist::insert_begin()
{
  int value;
  cout<<"Enter the value to be inserted: ";
  cin>>value;
  struct node *temp, *p;
  temp = create_node(value);
  if (start == NULL)
  {
    start = temp;
    start->next = NULL;
  }
  else
  {
    p = start;
    start = temp;
    start->next = p;
  }
  cout<<"Element Inserted at beginning"<<endl;
}

/*
 * Inserting Node at last
 */
void single_llist::insert_last()
{
  int value;
  cout<<"Enter the value to be inserted: ";
  cin>>value;
  struct node *temp, *s;
  temp = create_node(value);
  s = start;
  while (s->next != NULL)
  {
    s = s->next;
  }
  temp->next = NULL;
  s->next = temp;
  cout<<"Element Inserted at last"<<endl;
}

/*
 * Insertion of node at a given position
 */
void single_llist::insert_pos()
{
```

```
        int value, pos, counter = 0;
        cout<<"Enter the value to be inserted: ";
        cin>>value;
        struct node *temp, *s, *ptr;
        temp = create_node(value);
        cout<<"Enter the postion at which node to be
inserted: ";
        cin>>pos;
        int i;
        s = start;
        while (s != NULL)
        {
            s = s->next;
            counter++;
        }
        if (pos == 1)
        {
            if (start == NULL)
            {
                start = temp;
                start->next = NULL;
            }
            else
            {
                ptr = start;
                start = temp;
                start->next = ptr;
            }
        }
        else if (pos > 1  && pos <= counter)
        {
            s = start;
            for (i = 1; i < pos; i++)
            {
                ptr = s;
                s = s->next;
            }
            ptr->next = temp;
            temp->next = s;
        }
        else
        {
            cout<<"Positon out of range"<<endl;
        }
    }

/*
 * Sorting Link List
 */
void single_llist::sort()
{
    struct node *ptr, *s;
    int value;
    if (start == NULL)
    {
        cout<<"The List is empty"<<endl;
        return;
    }
    ptr = start;
    while (ptr != NULL)
    {
        for (s = ptr->next;s !=NULL;s = s->next
        {
            if (ptr->info > s->info)
            {
                value = ptr->info;
                ptr->info = s->info;
                s->info = value;
            }
```

```
    }
    ptr = ptr->next;
  }
}

/*
 * Delete element at a given position
 */
void single_llist::delete_pos()
{
  int pos, i, counter = 0;
  if (start == NULL)
  {
    cout<<"List is empty"<<endl;
    return;
  }
  cout<<"Enter the position of value to be deleted: ";
  cin>>pos;
  struct node *s, *ptr;
  s = start;
  if (pos == 1)
  {
    start = s->next;
  }
  else
  {
    while (s != NULL)
    {
      s = s->next;
      counter++;
    }
    if (pos > 0 && pos <= counter)
    {
      s = start;
      for (i = 1;i < pos;i++)
      {
        ptr = s;
        s = s->next;
      }
      ptr->next = s->next;
    }
    else
    {
      cout<<"Position out of range"<<endl;
    }
    free(s);
    cout<<"Element Deleted"<<endl;
  }
}

/*
 * Update a given Node
 */
void single_llist::update()
{
  int value, pos, i;
  if (start == NULL)
  {
    cout<<"List is empty"<<endl;
    return;
  }
  cout<<"Enter the node postion to be updated: ";
  cin>>pos;
  cout<<"Enter the new value: ";
  cin>>value;
  struct node *s, *ptr;
```

```
        s = start;
        if (pos == 1)
        {
            start->info = value;
        }
        else
        {
            for (i = 0;i < pos - 1;i++)
            {
                if (s == NULL)
                {
                    cout<<"There are less than
"<<pos<<" elements";
                    return;
                }
                s = s->next;
            }
            s->info = value;
        }
        cout<<"Node Updated"<<endl;
    }

/*
 * Searching an element
 */
void single_llist::search()
{
    int value, pos = 0;
    bool flag = false;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    cout<<"Enter the value to be searched: ";
    cin>>value;
    struct node *s;
    s = start;
    while (s != NULL)
    {
        pos++;
        if (s->info == value)
        {
            flag = true;
            cout<<"Element "<<value<<" is fou
at position "<<pos<<endl;
        }
        s = s->next;
    }
    if (!flag)
        cout<<"Element "<<value<<" not fou
in the list"<<endl;
}

/*
 * Reverse Link List
 */
void single_llist::reverse()
{
    struct node *ptr1, *ptr2, *ptr3;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    if (start->next == NULL)
    {
        return;
    }
```

```
    ptr1 = start;
    ptr2 = ptr1->next;
    ptr3 = ptr2->next;
    ptr1->next = NULL;
    ptr2->next = ptr1;
    while (ptr3 != NULL)
    {
      ptr1 = ptr2;
      ptr2 = ptr3;
      ptr3 = ptr3->next;
      ptr2->next = ptr1;
    }
    start = ptr2;
}

/*
 * Display Elements of a link list
 */
void single_llist::display()
{
  struct node *temp;
  if (start == NULL)
  {
    cout<<"The List is Empty"<<endl;
    return;
  }
  temp = start;
  cout<<"Elements of list are: "<<endl;
  while (temp != NULL)
  {
    cout<<temp->info<<"->";
    temp = temp->next;
  }
  cout<<"NULL"<<endl;
}
```

OUTPUT

```
Operations on singly linked list
--------------------------------
1.  Insert Node at beginning
2.  Insert node at last
3.  Insert node at position
4.  Sort Link List
5.  Delete a Particular Node
6.  Update Node Value
7.  Search Element
8.  Display Linked List
9.  Reverse Linked List
10. Exit
Enter your choice : 8
Display elements of link list
The List is Empty.
--------------------------------
Operations on singly linked list
--------------------------------
1.  Insert Node at beginning
2.  Insert node at last
3.  Insert node at position
4.  Sort Link List
5.  Delete a Particular Node
6.  Update Node Value
7.  Search Element
8.  Display Linked List
9.  Reverse Linked List
10. Exit
Enter your choice : 5
Delete a particular node:
List is empty
--------------------------------
Operations on singly linked list
--------------------------------
1.  Insert Node at beginning
2.  Insert node at last
3.  Insert node at position
4.  Sort Link List
```

5. Delete a Particular Node
6. Update Node Value
7. Search Element
8. Display Linked List
9. Reverse Linked List
10. Exit

Enter your choice : 6
Update Node Value:
List is empty

Operations on singly linked list

---

1. Insert Node at beginning
2. Insert node at **last**
3. Insert node at position
4. Sort Link List
5. Delete a Particular Node
6. Update Node Value
7. Search Element
8. Display Linked List
9. Reverse Linked List
10. Exit

Enter your choice : 3
Inserting Node at a given position:
Enter the value to be inserted: 1010
Enter the postion at **which** node to be inserted: 5
Positon out of range

Operations on singly linked list

---

1. Insert Node at beginning
2. Insert node at **last**
3. Insert node at position
4. Sort Link List
5. Delete a Particular Node
6. Update Node Value
7. Search Element
8. Display Linked List
9. Reverse Linked List
10. Exit

Enter your choice : 1
Inserting Node at Beginning:
Enter the value to be inserted: 100
Element Inserted at beginning

Operations on singly linked list

---

1. Insert Node at beginning
2. Insert node at **last**
3. Insert node at position
4. Sort Link List
5. Delete a Particular Node
6. Update Node Value
7. Search Element
8. Display Linked List
9. Reverse Linked List
10. Exit

Enter your choice : 1
Inserting Node at Beginning:
Enter the value to be inserted: 200
Element Inserted at beginning

Operations on singly linked list

---

1. Insert Node at beginning
2. Insert node at **last**
3. Insert node at position
4. Sort Link List
5. Delete a Particular Node
6. Update Node Value
7. Search Element
8. Display Linked List
9. Reverse Linked List
10. Exit

Enter your choice : 8
Display elements of **link** list
Elements of list are:
200->100->NULL

3.1.2 Doubly Linked Lists

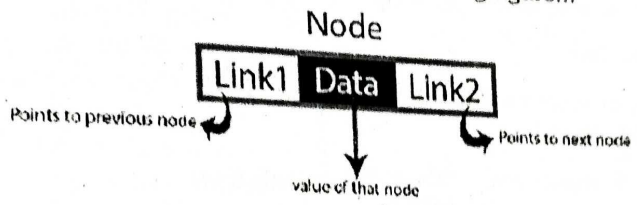
Q7. Explain doubly linked list operations. **(Imp.)**

Ans :

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.

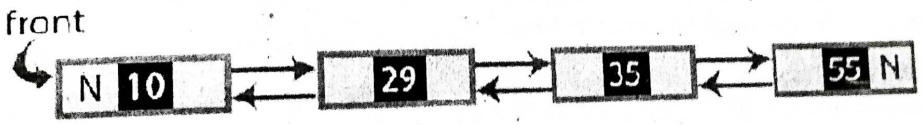
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

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



- 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

Insertion

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 node until it reaches to the node after we want to insert the newNode (until → data is equal to location, here loc is the node value after which we wa insert the newNode).
- **Step 6:** Every time check whether temp reached to the last node. If it is reache the last node then display 'Given node found in the list!!! Insertion not possib and terminate the function. Otherwise m the temp1 to next node.
- **Step 7:** Assign temp1 → next to tem newNode to temp1 '! next, temp1 newNode → previous, temp2 to newN → nextand newNode to temp2 → prev

Deletion

In a double linked list, the deletion operat 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 delet node from beginning of the double linked list.

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

Deleting from En

We can use node from end of

- **Step 1:** Ch == NULL)
- **Step 2:** If Empty!!! D terminate th
- **Step 3:** If i pointer 'ter
- **Step 4:** C Node (ter next both
- **Step 5:** If head and the functio
- **Step 6:** temp until list. (until
- **Step 7** previous

Deleting a Spe

We can u specific node fro

- **Step 1:** == NUL
- **Step 2:** Empty!!! terminate
- **Step 3:** pointer
- **Step 4:** reaches the last
- **Step 5:** display Deletion fuction.
- **Step 6:** which w list is ha

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).

Q8. Write a C++ program to implement doubly lined list.

Ans : **(Imp.)**

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
/*
 * Node Declaration
 */
using namespace std;
struct node
{
    int info;
    struct node *next;
    struct node *prev;
}*start;
/*
Class Declaration
*/
class double_llist
{
    public:
        void create_list(int value);
        void add_begin(int value);
        void add_after(int value, int
        position);
        void delete_element(int value);
        void search_element(int value);
        void display_dlist();
        void count();
        void reverse();
        double_llist()
{
start = NULL;
}
};
/*
 * Main: Conatins Menu
 */
int main()
{
    int choice, element, position;
    double_llist dl;
    while (1)
{
cout<<endl<<"————————————"<<endl;
cout<<endl<<"Operations on Doubly
linked list"<<endl;
cout<<endl<<"————————————"<<endl;
cout<<"1.Create Node"<<endl;
cout<<"2.Add at begining"<<endl;
cout<<"3.Add after position"<<endl;
cout<<"4.Delete"<<endl;
cout<<"5.Display"<<endl;
cout<<"6.Count"<<endl;
cout<<"7.Reverse"<<endl;
cout<<"8.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch ( choice )
{
case 1:
    cout<<"Enter the element: ";
    cin>>element;
    dl.create_list(element);
    cout<<endl;
    break;
case 2:
    cout<<"Enter the element: ";
    cin>>element;
    dl.add_begin(element);
    cout<<endl;
    break;
```

```
case 3:
    cout<<"Enter the element: ";
    cin>>element;
    cout<<"Insert Element after postion: ";
    cin>>position;
    dl.add_after(element, position);
    cout<<endl;
    break;
case 4:
    if (start == NULL)
    {
        cout<<"List empty,nothing to
        delete"<<endl;
    break;
    }
        cout<<"Enter the element for
        deletion: ";
    cin>>element;
    dl.delete_element(element);
    cout<<endl;
    break;
case 5:
    dl.display_dlist();
    cout<<endl;
    break;
case 6:
    dl.count();
    break;
case 7:
    if (start == NULL)
    {
        cout<<"List empty,nothing to
        reverse"<<endl;
    break;
    }
    dl.reverse();
    cout<<endl;
    break;
case 8:
    exit(1);
    default:
    cout<<"Wrong choice"<<endl;
    }
    }
        return 0;
    }
/*
 * Create Double Link List
 */
void double_llist::create_list(int value)
{
    struct node *s, *temp;
    temp = new(struct node);
    temp->info = value;
    temp->next = NULL;
  if (start == NULL)
  {
      temp->prev = NULL;
      start = temp;
  }
  else
  {
      s = start;
      while (s->next != NULL)
      s = s->next;
      s->next = temp;
    temp->prev = s;
  }
}
/*
 * Insertion at the beginning
 */
void double_llist::add_begin(int value)
```

```
{
        if (start == NULL)
        {
            cout<<"First Create the list."<<endl;
        return;
        }
        struct node *temp;
        temp = new(struct node);
        temp->prev = NULL;
        temp->info = value;
        temp->next = start;
        start->prev = temp;
        start = temp;
        cout<<"Element Inserted"<<endl;
}
/*
 * Insertion of element at a particular position
 */
void double_llist::add_after(int value, int pos)
{
    if (start == NULL)
    {
        cout<<"First Create the list."<<endl;
    return;
    }
    struct node *tmp, *q;
    int i;
    q = start;
    for (i = 0;i < pos - 1;i++)
    {
        q = q->next;
        if (q == NULL)
    {
        cout<<"There are less than ";
        cout<<pos<<" elements."<<endl;
        return;
    }
}
```

```
    tmp = new(struct node);
    tmp->info = value;
    if (q->next == NULL)
    {
        q->next = tmp;
        tmp->next = NULL;
        tmp->prev = q;
    }
    else
    {
        tmp->next = q->next;
        tmp->next->prev = tmp;
        q->next = tmp;
        tmp->prev = q;
    }
    cout<<"Element Inserted"<<endl;
    }
/*
 * Deletion of element from the list
 */
void double_llist::delete_element(int value)
{
    struct node *tmp, *q;
    /*first element deletion*/
    if (start->info == value)
    {
        tmp = start;
        start = start->next;
        start->prev = NULL;
        cout<<"Element Deleted"<<endl;
        free(tmp);
        return;
    }
    q = start;
    while (q->next->next != NULL)
    {
```

```
/*Element deleted in between*/
    if (q->next->info == value)
    {
        tmp = q->next;
        q->next = tmp->next;
        tmp->next->prev = q;
        cout<<"Element Deleted"<<endl;
        free(tmp);
        return;
        }
      q = q->next;
    }
/*last element deleted*/
    if (q->next->info == value)
    {
        tmp = q->next;
        free(tmp);
        q->next = NULL;
        cout<<"Element Deleted"<<endl;
        return;
    }
    cout<<"Element "<<value<<" not
    found"<<endl;
    }

/*
 * Display elements of Doubly Link List
 */
void double_llist::display_dlist()
{
    struct node *q;
    if (start == NULL)
    {
    cout<<"List empty,nothing to display"
    <<endl;
    return;
    }
    q = start;
    cout<<"The Doubly Link List is :"<<endl;
    while (q != NULL)
    {
        cout<<q->info<<" <-> ";
        q = q->next;
    }
    cout<<"NULL"<<endl;
    }
/*
 * Number of elements in Doubly Link List
 */
    void double_llist::count()
    {
        struct node *q = start;
        int cnt = 0;
        while (q != NULL)
    {
        q = q->next;
        cnt++;
    }
    cout<<"Number of elements are: "<<cnt
    <<endl;
    }
/*
 * Reverse Doubly Link List
 */
void double_llist::reverse()
{
    struct node *p1, *p2;
    p1 = start;
    p2 = p1->next;
    p1->next = NULL;
    p1->prev = p2;
```

```
    while (p2 != NULL)
    {
        p2->prev = p2->next;
        p2->next = p1;
        p1 = p2;
        p2 = p2->prev;
    }
    start = p1;
    cout<<"List Reversed"<<endl;
}
```

OUTPUT

```
Operations on Doubly linked list

1.  Create Node
2.  Add at begining
3.  Add after
4.  Delete
5.  Display
6.  Count
7   Reverse
8.  Quit
Enter your choice : 2
Enter the element: 100
First Create the list.

Operations on Doubly linked list

1.  Create Node
2.  Add at begining
3.  Add after
4.  Delete
5.  Display
6.  Count
7.  Reverse
8.  Quit
Enter your choice : 3
Enter the element: 200
Insert Element after postion: 1
First Create the list.

Operations on Doubly linked list

1.  Create Node
2.  Add at begining
3.  Add after
4.  Delete
5.  Display
6.  Count
7.  Reverse
8.  Quit
Enter your choice : 7
List empty,nothing to reverse
```

3.1.3 Circular Lists

Q9. What are circular lists? Explain the operations of circular lists.

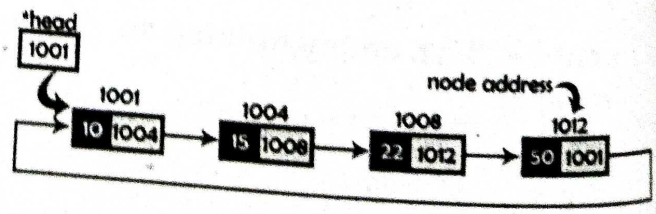
Ans : **(Imp.)**

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

Example



Operations

In a circular 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:** Include all the header files which are used in the program.
- **Step 2:** Declare all the user defined functions.
- **Step 3:** Define a Node structure with two members data and next
- **Step 4:** 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 circular 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 circular 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 head = newNode and newNode'!next = 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 (until 'temp → next == head').
- **Step 6:** Set 'newNode '! next =head', 'head = newNode' and 'temp '! next = head'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular 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 head = newNode and newNode → next = 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 == head).
- **Step 6:** Set temp → next = newNode and newNode '! next = head.

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 circular 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 head = newNode and newNode → next = 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 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 the 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:** If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8:** If temp is last node then set temp → next = newNode and newNode → next = head.
- **Step 8:** If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those 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 node from beginning of the circular 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 both 'temp1' and 'temp2' with head.
- **Step 4:** Check whether list is having only one node (temp1 → next == head)
- **Step 5:** If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)
- **Step 6:** If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)
- **Step 7:** Then set head = temp2 → next, temp1 → next = head and delete temp2.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular 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 == head)
- **Step 5:** If it is TRUE. Then, set head = NULL and delete temp1. And terminate from 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 temp1 reaches to the last node in the list. (until temp1 → next = = head)
- **Step 7:** Set temp2 → next = head and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular 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 (temp1 → next == head)
- **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 set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.
- **Step 10:** If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).
- **Step 11:** If temp1 is last node then set temp2 → next = head 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 circular Linked List

We can use the following steps to display the elements of a circular 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 head → data.

Q10. Write a C++ program to implement circular lists.

Ans : **(Imp.)**

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
    int info;
    struct node *next;
}*last;
/*
 * Class Declaration
 */
class circular_llist
{
    public:
        void create_node(int value);
        void add_begin(int value);
        void add_after(int value, int position);
```

```
        void delete_element(int value);
        void search_element(int value);
        void display_list();
        void update();
        void sort();
        circular_llist()
        {
            last = NULL;
        }
    };
    /*
     * Main :contains menu
     */
    int main()
    {
        int choice, element, position;
        circular_llist cl;
        while (1)
    {
        cout<<endl<<"————————————"<<endl;
        cout<<endl<<"Circular singly linked
                        list"<<endl;
        cout<<endl<<"————————————"<<endl;
        cout<<"1.Create Node"<<endl;
        cout<<"2.Add at beginning"<<endl;
        cout<<"3.Add after"<<endl;
        cout<<"4.Delete"<<endl;
        cout<<"5.Search"<<endl;
        cout<<"6.Display"<<endl;
        cout<<"7.Update"<<endl;
        cout<<"8.Sort"<<endl;
        cout<<"9.Quit"<<endl;
        cout<<"Enter your choice : ";
        cin>>choice;
        switch(choice)
        {
            case 1:
                cout<<"Enter the element: ";
                cin>>element;
                cl.create_node(element);
                cout<<endl;
                break;
            case 2:
                cout<<"Enter the element: ";
                cin>>element;
                cl.add_begin(element);
                cout<<endl;
                break;
            case 3:
                cout<<"Enter the element: ";
                cin>>element;
                cout<<"Insert element after position: ";
                cin>>position;
                cl.add_after(element, position);
                cout<<endl;
                break;
            case 4:
                if (last == NULL)
                {
                cout<<"List is empty, nothing
                delete"<<endl;
                break;
                }
                cout<<"Enter the element for deletion:
                cin>>element;
                cl.delete_element(element);
                cout<<endl;
                break;
            case 5:
                if (last == NULL)
                {
                cout<<"List Empty!! Can't search"<<en
                break;
                }
                cout<<"Enter the element to be searched
```



```
    cin>>elem
    cl.search_e
    cout<<en
    break;
case 6:
    cl.display_
    break;
case 7:
    cl.update(
    break;
case 8:
    cl.sort();
    break;
case 9:
    exit(1);
    break;
    default:
        co
    }
}
return 0;

/*
 * Create Circ
 */
void circular_
{
    struct node
    temp = n
    temp->inf
    if (last ==
    {
        last =
        temp->
```

```
    cin>>element;
    cl.search_element(element);
    cout<<endl;
    break;
case 6:
    cl.display_list();
    break;
case 7:
    cl.update();
    break;
case 8:
    cl.sort();
    break;
case 9:
    exit(1);
    break;
default:
        cout<<"Wrong choice"<<endl;
    }
  }
return 0;
}
/*
 * Create Circular Link List
 */
void circular_llist::create_node(int value)
{
struct node *temp;
temp = new(struct node);
temp->info = value;
if (last == NULL)
{
  last = temp;
  temp->next = last;
}
else
{
    temp->next = last->next;
    last->next = temp;
    last = temp;
}
}
/*
 * Insertion of element at beginning
 */
void circular_llist::add_begin(int value)
{
  if (last == NULL)
  {
      cout<<"First Create the list."<<endl;
      return;
  }
  struct node *temp;
  temp = new(struct node);
  temp->info = value;
  temp->next = last->next;
  last->next = temp;
}
/*
 * Insertion of element at a particular place
 */
void circular_llist::add_after(int value, int pos)
{
    if (last == NULL)
    {
        cout<<"First Create the list."<<endl;
        return;
    }
    struct node *temp, *s;
    s = last->next;
    for (int i = 0;i < pos-1;i++)
    {
```

```
            s = s->next;
            if (s == last->next)
            {
                cout<<"There are less than ";
                cout<<pos<<" in the list"<<endl;
                return;
            }
        }
        temp = new(struct node);
        temp->next = s->next;
        temp->info = value;
        s->next = temp;
    /*Element inserted at the end*/
        if (s == last)
        {
            last=temp;
        }
    }
    /*
     * Deletion of element from the list
     */
    void circular_llist::delete_element(int value)
    {
        struct node *temp, *s;
        s = last->next;
    /* If List has only one element*/
        if (last->next == last && last->info == value)
        {
            temp = last;
            last = NULL;
            free(temp);
            return;
        }
        if (s->info == value) /*First Element Deletion*/
        {
            temp = s;
            last->next = s->next;
            free(temp);
            return;
        }
        while (s->next != last)
        {
    /*Deletion of Element in between*/
        if (s->next->info == value)
        {
            temp = s->next;
            s->next = temp->next;
            free(temp);
            cout<<"Element "<<value;
            cout<<" deleted from the list"<<endl;
            return;
        }
            s = s->next;
        }
    /*Deletion of last element*/
        if (s->next->info == value)
        {
            temp = s->next;
            s->next = last->next;
            free(temp);
            last = s;
            return;
        }
            cout<<"Element "<<value<<" not fo
            in the list"<<endl;
        }
    /*
     * Search element in the list
     */
    void circular_llist::search_element(int value)
    {
```

```
struct node *s;
int counter = 0;
s = last->next;
while (s != last)
{
  counter++;
  if (s->info == value)
  {
    cout<<"Element "<<value;
    cout<<" found at position "<< counter
    <<endl;
    return;
  }
  s = s->next;
}
if (s->info == value)
{
  counter++;
  cout<<"Element "<<value;
  cout<<" found at position
        "<<counter<<endl;
  return;
}
cout<<"Element "<<value<<" not found in
the list"<<endl;


Display Circular Link List
*/
void circular_llist::display_list()
{
struct node *s;
if (last == NULL)
{
  cout<<"List is empty, nothing to display"
  <<endl;
  return;
}
s = last->next;
cout<<"Circular Link List: "<<endl;
while (s != last)
{
  cout<<s->info<<"->";
  s = s->next;
}
cout<<s->info<<endl;
}
/*
 * Update Circular Link List
 */
void circular_llist::update()
{
  int value, pos, i;
  if (last == NULL)
  {
    cout<<"List is empty, nothing to
    update"<<endl;
  return;
  }
    cout<<"Enter the node position to be
    updated: ";
  cin>>pos;
  cout<<"Enter the new value: ";
  cin>>value;
  struct node *s;
  s = last->next;
  for (i = 0;i < pos - 1;i++)
  {
    if (s == last)
    {
    cout<<"There are less than "<<pos<<"
    elements.";
    cout<<endl;
    return;
```

```
        }
            s = s->next;
        }
            s->info = value;
            cout<<"Node Updated"<<endl;
    }
    /*
     * Sort Circular Link List
     */
void circular_llist::sort()
{
    struct node *s, *ptr;
    int temp;
    if (last == NULL)
    {
        cout<<"List is empty, nothing to sort"<<endl;
        return;
    }
    s = last->next;
    while (s != last)
    {
        ptr = s->next;
        while (ptr != last->next)
        {
            if (ptr != last->next)
            {
                if (s->info > ptr->info)
                {
                    temp = s->info;
                    s->info = ptr->info;
                    ptr->info = temp;
                }
            }
            else
            {
                break;
            }
            ptr = ptr->next;
        }
        s = s->next;
    }
}
```

OUTPUT

```
Operations on Circular singly linked list
1. Create Node
2. Add at beginning
3. Add after
4. Delete
5. Search
6. Display
7. Update
8. Sort
9. Quit
Enter your choice : 1
Enter the element: 100

Operations on Circular singly linked list
1. Create Node
2. Add at beginning
3. Add after
4. Delete
5. Search
6. Display
7. Update
8. Sort
9. Quit
Enter your choice : 2
Enter the element: 200
```

Operations on Circular singly linked list

1. Create Node
2. Add at beginning
3. Add after
4. Delete
5. Search
6. Display
7. Update
8. Sort
9. Quit

Enter your choice : 6

Circular Link List:

200->100

3.2 HASHING

3.2.1 Static Hashing

Q11. What is Hashing? **(Imp.)**

Ans :

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

Q12. What is static hashing? Explain it.

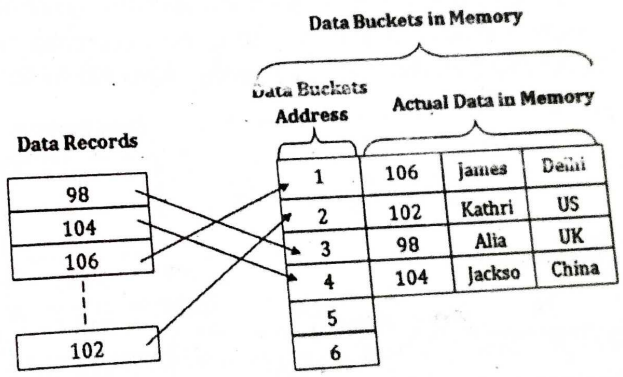
Ans :

Static hashing is a method of ' ing, or shortening a string of characters in computer programming, in which the set of shortened characters remains the same length to improve the ease with which data can be accessed. All objects listed in an object dictionary are static and may not change when static hashing is applied.

The process of static hashing creates a smaller, adaptable string of characters, making it faster and easier for users to find objects in a dictionary or groups of objects stored in a containing data structure. Static hashing helps to better utilize the available space in a database for objects, tables and other data within a system.

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



Operations of Static Hashing

i) Searching a record

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

ii) Insert a Record

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

iii) Delete a Record

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

iv) Update a Record

To update a record, we will first search it using a hash function, and then the data record is updated.

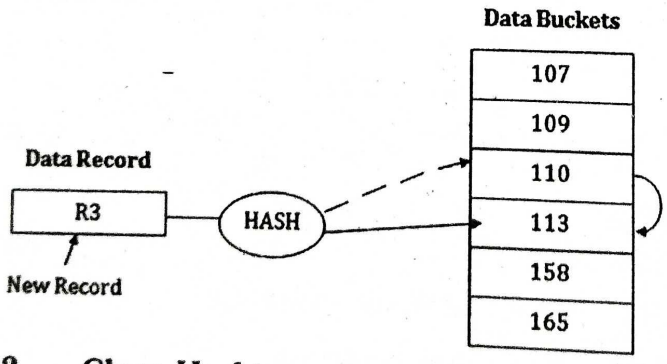
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as bucket overflow. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as Linear Probing.

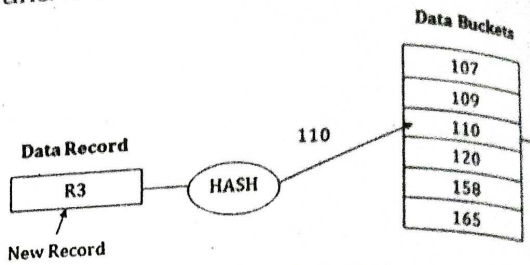
For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as Overflow chaining.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 bu and is linked to it.

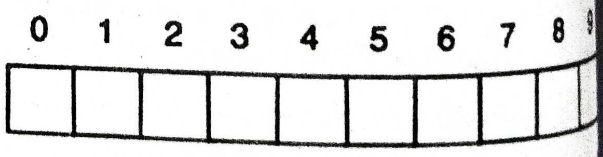


3.2.2 Hash Tables

Q13. What is Hash Table? Write about it

Ans :

- Hash table or hash map is a data struc used to store key-value pairs.
- It is a collection of items stored to mak easy to find them later.
- It uses a hash function to compute an in into an array of buckets or slots from wh the desired value can be found.
- It is an array of list where each list is know bucket.
- It contains value based on the key.
- Hash table is used to implement the m interface and extends Dictionary class.
- Hash table is synchronized and contains unique elements.



- The above figure shows the hash table w the size of n = 10. Each position of the ha table is called as **Slot**. In the above ha table, there are n slots in the table, name {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot slot 2 and so on. Hash table contains items, so every slot is empty.
- As we know the mapping between an it and the slot where item belongs in the ha table is called the hash function. The ha function takes any item in the collection a returns an integer in the range of slot nam between 0 to n-1.

- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example, $\lambda = 6/10$.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

3.2.3 Hash Functions

Q14. What is Hash Function? **(Imp.)**

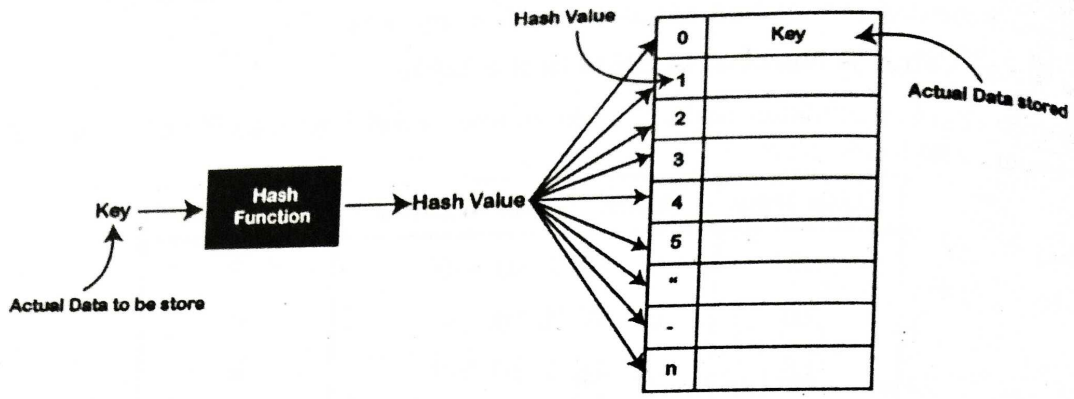
Ans :

- A fixed process converts a key to a hash key is known as a Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

dex = hash(key)



Q15. Explain various ways to calculate has function.

Ans :

There are three ways of calculating the hash function:

- Division method
- Folding method
- Mid square method

In the division method, the hash function can be defined as:

$$\mathbf{h(k_i) = k_i \ \% \ m;}$$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the has function to key 6 then the index would be:

$$h(6) = 6\%10 = 6$$

The index is 6 at which the value is stored.

1. Division Method

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen t be a prime number or a number without small divisors, since this frequently a minimum number collisions).

The hash function is:

$$\boxed{h(k) = k \bmod m} \qquad \boxed{h(k) = k \bmod m + 1}$$

For Example: if the hash table has size m = 12 and the key is k = 100, then h (k) = 4. Since requires only a single division operation, hashing by division is quite fast.

2. Mid Square Method

The key k is squared. Then function H is defined by

$$H(k) = L$$

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position k^2 must be used for all of the keys.

3. **Folding Method**

The key k is partitioned into a number of parts $k_1, k_2 \ldots k_n$ where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$$H(k) = k^1 + k^2 + \ldots + k^n$$

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2- digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

(a) **Division Method:** Choose a Prime number m close to 99, such as m =97, Then

H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) **Mid-Square Method:**

k =	3205	7148	2345
k^2 =	10272025	51093904	5499025
h (k) =	72	93	99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) **Folding Method:** Divide the key k into 2 parts and adding yields the following hash address:

H (3205) = 32 + 50 = 82 H (7148) = 71 + 84 = 55

H (2345) = 23 + 45 = 68

3.2.4 Overflow Handling

Q16. How can we handle overflow in hashing? Explain

(OR)

Explain various collision handling techniques in hashing.

(Imp.)

Ans :

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

$$h(26) = 26\%10 = 6$$

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

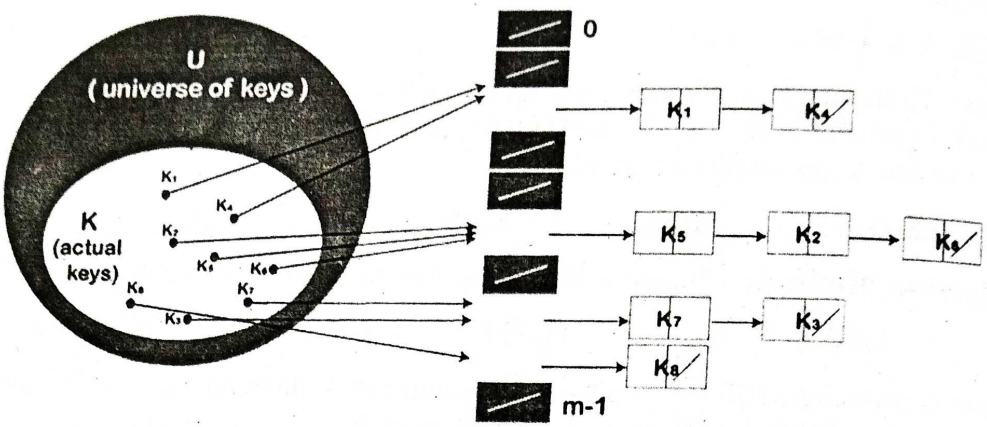
The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.

Collision Resolution by Chaining



Let's first understand the chaining to resolve the collision.

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

In this case, we cannot directly use h(k) = k/m as h(k) = 2k+3

- The index of key value 3 is:

  index = h(3) = (2(3)+3)%10 = 9

  The value 3 would be stored at the index 9.

- The index of key value 2 is:

  index = h(2) = (2(2)+3)%10 = 7

  The value 2 would be stored at the index 7.

- The index of key value 9 is:

  index = h(9) = (2(9)+3)%10 = 1

  The value 9 would be stored at the index 1.

- The index of key value 6 is:

  index = h(6) = (2(6)+3)%10 = 5

  The value 6 would be stored at the index 5.

- The index of key value 11 is:

  index = h(11) = (2(11)+3)%10 = 5

The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

The index of key value 13 is:

index = h(13) = (2(13)+3)%10 = 9

The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

The index of key value 7 is:

index = h(7) = (2(7)+3)%10 = 7

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

The index of key value 12 is:

index = h(12) = (2(12)+3)%10 = 7

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	((2*3)+3)%10 = 9
2	((2*2)+3)%10 = 7
9	((2*9)+3)%10 = 1
6	((2*6)+3)%10 = 5
11	((2*11)+3)%10 = 5
13	((2*13)+3)%10 = 9
7	((2*7)+3)%10 = 7
12	((2*12)+3)%10 = 7

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

1. Linear Probing

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

Quadratic Probing

In case of linear probing, searching is performed linearly. In contrast, quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion ki at first free location from $(u+i^2)\%m$ where i=0 to m-1.

Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where

m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

Index= $(5+0^2)\%10 = 5$

When i=1

Index = $(5+1^2)\%10 = 6$

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(9+0^2)\%10 = 9$

When i=1

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When i=0

Index= $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

When i=2

Index = $(7+2^2)\%10 = 1$

When i=3

Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

The final hash table would be:

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_1 at first free place from ***(u+v*i)%m*** where i=(0 to m-1)". In this case, u is the location computed using the hash function and v is equal to ($h_2(k)$%m).

Consider the same example that we use in quadratic probing.

A= 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and

$h_1(k) = 2k+3$

$h_2(k) = 3k+1$

key	Location (u)	v	probe
3	((2*3)+3)%10 = 9	-	1
2	((2*2)+3)%10 = 7	-	1
9	((2*9)+3)%10 = 1	-	1
6	((2*6)+3)%10 = 5	-	1
11	((2*11)+3)%10 = 5	(3(11)+1)%10 =4	3
13	((2*13)+3)%10 = 9	(3(13)+1)%10 = 0	
7	((2*7)+3)%10 = 7	(3(7)+1)%10 = 2	
12	((2*12)+3)%10 = 7	(3(12)+1)%10 = 7	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.

On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (5+4*0)%10 =5

When i=1

Index = (5+4*1)%10 = 9

When i=2

Index = (5+4*2)%10 = 3

Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.

The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (9+0*0)%10 = 9

We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.

The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (7 + 2*0)%10 = 7

When i=1

Index = (7+2*1)%10 = 9

When i=2

Index = (7+2*2)%10 = 1

When i=3

Index = (7+2*3)%10 = 3

When i=4

Index = (7+2*4)%10 = 5

When i=5

Index = (7+2*5)%10 = 7

When i=6

Index = (7+2*6)%10 = 9

When i=7

Index = (7+2*7)%10 = 1

When i=8

Index = (7+2*8)%10 = 3

When i=9

Index = (7+2*9)%10 = 5

Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert. Therefore, key 7 cannot be inserted in a hash table.

The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (7+7*0)%10 = 7

When i=1

Index = (7+7*1)%10 = 4

Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.

The final hash table would be:

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

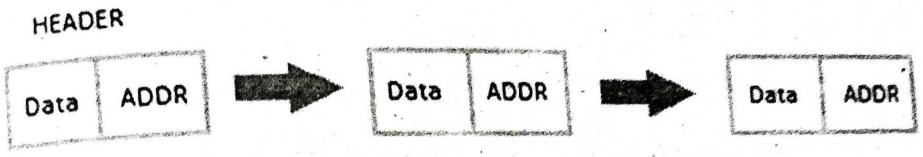
The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

Short Question and Answers

1. What are linked lists? Write about it.

Ans :

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.



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 and queues can be easily executed.
- 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.

2. Linked List Abstrct Data Type

Ans :

Although a linked list can be implemented in a variety of ways, the most fl exible implementation is by using pointers.

- **Create():** It creates a linked list.
- **Traverse():** It traverse the entire list.
- **Insert(int data, position):** It insert the new element to the location at a given position.
- **Append(int data):** Appends the new data element at the end of the list.
- **Delete(int position):** Deletes the list from the given position.
- **Reverse():** It reverses the list.

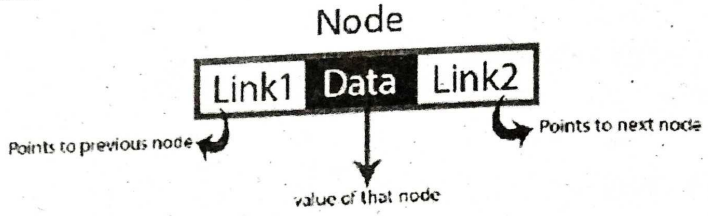
3. Doubly Linked Lists

Ans :

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

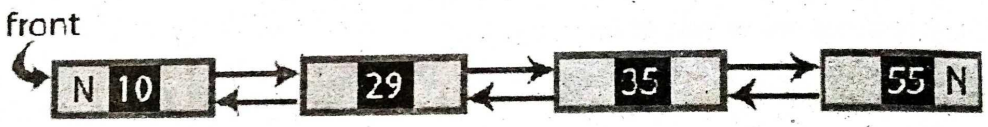
Double linked list is a sequence of elements in which every element has links to its previous and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can forward by using next field and can traverse backward by using previous field. Every node in a 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'** is used to store the address of the next node in the sequence and **'data'** field is used to store the value of that node.

Example



- 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**.

4. What are circular lists?

Ans :

In single linked list, every node points to its next node in the sequence and the last node NULL. But in circular linked list, every node points to its next node in the sequence but the points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node first node in the list.

5. What is Hashing?

Ans :

- Hashing is the process of mapping large amount of data item to smaller table with the hashing function.

- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

6. What is static hashing?

Ans :

Static hashing is a method of hashing, or shortening a string of characters in computer programming, in which the set of shortened characters remains the same length to improve the ease with which data can be accessed. All objects listed in an object dictionary are static and may not change when static hashing is applied.

The process of static hashing creates a smaller, adaptable string of characters, making it faster and easier for users to find objects in a dictionary or groups of objects stored in a containing data structure. Static hashing helps to better utilize the available space in a database for objects, tables and other data within a system.

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

7. Hash Table

Ans :

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

0	1	2	3	4	5	6	7	8	9

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.

8. What is Hash Function?

Ans :

- A fixed process converts a key to a hash key is known as a Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

9. Overflow Handling.

Ans :

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

$$h(26) = 26\%10 = 6$$

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- **Open Hashing:** It is also known as closed addressing.
- **Closed Hashing:** It is also known as open addressing.

10. Linear Probing

Ans :

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

Choose the Correct Answers

1. Which of the following information is stored in a doubly-linked list's nodes? [d]
   a) Value of node
   b) Address of next node
   c) Address of the previous node
   d) All of the above

2. Which type of linked list stores the address of the head node in the next pointer of the last node? [d]
   a) Singly Linked List
   b) Doubly Linked List
   c) Hashed List
   d) Circular Linked List

3. In which type of linked lists traversals can be performed in both directions? [b]
   a) Singly Linked Lists
   b) Doubly Linked Lists
   c) Circular Linked Lists
   d) None of the above

4. The type of pointer used to point to the address of the next element in a linked list? [c]
   a) Pointer to character
   b) Pointer to integer
   c) Pointer to node
   d) None of the above

5. Which of the following sorting algorithms can be applied to linked lists? [d]
   a) Merge Sort
   b) Insertion Sort
   c) Quick Sort
   d) All of the above

6. Insertion of an element at the middle of a linked list requires the modification of how many pointers? [a]
   a) 2
   b) 1
   c) 3
   d) 4

7. If several elements are competing for the same bucket in the hash table, what is it called? [7]
   a) Diffusion
   b) Replication
   c) Collision
   d) Duplication

8. The best technique for handling collision is: [d]
   a) Separate chaining
   b) Double hashing
   c) Linear probing
   d) Quadratic probing

9. Which one is the most desirable out of these traits of a hash function? [c]
   a) It must cause more collisions
   b) It must be easy to implement
   c) It must cause less collisions
   d) It must occupy less space

10. Which of the following is optimal to find an element at kth position at the linked list? [d]
   a) Singly Linked List
   b) Doubly Linked List
   c) Circular Linked List
   d) Array implementation of linked list.

Fill in the Blanks

1. _____ is the time complexity to search an element in the linked list?
2. Polynomial Addition can be implemented using _____
3. In a _____ linked list none of the nodes contains a NULL pointer
4. In a circular linked list insertion of a record requires the modification of _____ pointers.
5. _____ linked lists we use to search a sorted list in better than amortized O(n) complexity?
6. _____ is possible only when we can afford to allocate an array that has one position for e possible key.
7. _____ involves the process of formulating alternative indices for a key.
8. _____ is not a theoretical problem but actually occurs in real implementations of probing.
9. _____ comes under open addressing scheme to resolve collisions in hash tables.
10. _____ hash function used in the division method?

Answers

1. O(n)
2. Linked list
3. Circular list
4. 2
5. Skip lists
6. Direct addressing
7. Collision handling
8. Clustering
9. Quadratic probing
10. $h(k) = k \bmod m$

UNIT IV

Tre
Tre
tre
Gr
op

Q1. What is tree da

Ans :

A tree data struc

Tree data structu
s a recursive definitio

In tree data str
tores the actual data

In a tree data
1 number of links.

Example

D

Terminology

In a tree data

1. Root

In a tree data
can say that
node. We n