INDEX

Sr. No.	Title	Date	Remark & Sign
1	Write a program that performs basic operations on an array, including: a. Adding an element at a given index within the array. b. Removing an element from a specified index in the array.		
2	Write programs to implement: a. Linear search b. Binary search (on a sorted array)		
3	Develop a program that performs the following tasks: a. Implement a stack data structure using an array. b. Evaluate a postfix expression using Stack.		
4	Write a program to: a. Implement a queue data structure using an array. b. Simulate simple Queue system for managing orders.		
5	Implement a program to manage a singly linked list with the following functionalities: a. Initialize an empty singly linked list. b. Add a new node at the start, at the end in the list. c. Delete a node from a given position in a linked list		
6	Write programs to implement and compare the following sorting algorithms: a. Selection Sort b. Bubble Sort c. Insertion Sort		

7	Design a program to manage to a. Create a Binary Search Tree b. To perform different types of traversals on BST		
8	Develop a program to a. Store and retrieve data from the hash table. b. Write a program to implement the collision technique.		

Data Structure Practical

Practical 1: Array Basic Operations

Title:

Write a program that performs basic operations on an array, including:

a. Adding an element at a given index within the array.

b. Removing an element from a specified index in the array.

Objective:

To understand how to manipulate arrays by adding and removing elements at specific positions.

Theory:

Arrays are a fundamental data structure that store elements of the same type in contiguous memory locations. In most programming languages, arrays have fixed sizes, so insertion and deletion involve shifting elements to maintain array order.

Syntax:

Single Dimensional Array

type arrayName [arraySize];

Example:

int num[10];

Insertion at index i requires shifting elements from i onwards to the right.

Deletion at index i requires shifting elements from i+1 onwards to the left.

Code:

```
#include<iostream>
int size= 5,ub;
void trav(int[]);
void insert(int[],int,int);
void del(int[],int);
using namespace std;
int main()
{
	int data[size]={1,5,6,8},pos,num,loc;
	ub=3;
	trav(data);
	cout<<"\nEnter position to insert and data\n";
	cin>>pos>>num;
	insert(data,pos,num);
	trav(data);
	cout<<"\nEnter index location to delete data";
	cin>>loc;
	del(data,loc);
	trav(data);
	return 0;
}
void trav(int data[])
{
	cout<<"\nArray elements are\n";
	for(int i=0;i<=ub;i++)
	{
		cout<<data[i]<<"\t";
	}
}
void insert(int data[],int pos,int num)
{
	int j=ub;
	while(j>=pos)
	{
```

```
                data[j+1]=data[j];
                j=j-1;
        }
        data[pos]=num;
        ub++;
}
void del(int data[],int loc)
{
        for(int i=loc;i<=ub-1;i++)
        {
                data[i]=data[i+1];
        }
        ub--;
}
```

Output:

```
Array elements are
1       5       6       8
Enter position to insert and data
3 100
Array elements are
1       5       6       100     8
Enter index location to delete data
2 6
Array elements are
1       5       100     8
```

Practical 2: Implement Linear and Binary Search

Title:

Write programs to implement and compare:

a. Linear Search

b. Binary Search (on a sorted array)

Objective:

• To implement basic searching algorithms.
• To compare the performance of linear and binary search techniques.

Theory:

1. Linear Search
• Sequentially checks each element in the array.
• Time Complexity:
o Best Case: O(1)
o Worst/Average Case: O(n)
• Works on unsorted or sorted arrays.
2. Binary Search
• Repeatedly divides the sorted array in half to locate the element.
• Time Complexity:
o Best Case: O(1)
o Worst/Average Case: O(log n)
• Requires the array to be sorted.

Code1:

```
#include<iostream>
int SIZE=5;
int lsearch (int[],int);
using namespace std;
int main()
{
	int num[SIZE]={4,67,89,90,500},item,val1;
	cout<<"\nEnter element to search:";
	cin>>item;
	cout<<"\nLinear Search:\n";
	val1=lsearch(num,item);
	if(val1==-1)
```

```
            cout<<"Element Not Found";
        else
        cout<<"Element Found at index position"<<val1;
        return 0;
}
int lsearch(int num[],int item)
{
        for(int i=0;i<SIZE;i++)
        {
                if(item == num[i])
                {
                        return(i);
                }
        }
        return -1;
}
```

Output:

Enter element to search:90
Linear Search:
Element Found at index position3

OR

Enter element to search:3
Linear Search:
Element Not Found

Code2:

```
#include<iostream>
int SIZE=5;
int bsearch (int[],int);
using namespace std;
int main()
{
        int num[SIZE]={4,67,89,90,500},item,val1;
        cout<<"\nEnter element to search:";
        cin>>item;
```

```
	cout<<"\nBinary Search:\n";
	val1=bsearch(num,item);
	if(val1==-1)
		cout<<"Element Not Found";
	else
	cout<<"Element Found at index position"<<val1;
	return 0;
}
int bsearch(int num[],int item)
{
	int beg=0,end=SIZE-1;
	int mid=(beg+end)/2;
	while(beg<=end && num[mid]!=item)
	{
		if(item<num[mid])
			end=mid-1;
		else
			beg=mid+1;
			mid=(beg+end)/2;
	}
	if(num[mid]==item)
		return mid;
	else
		return -1;
}
```

Output:

Enter element to search:67
Binary Search:
Element Found at index position1

OR

Enter element to search:3
Binary Search:
Element Not Found

Practical 3: Stack Implementation and Postfix Evaluation

Title:

Develop a program that performs the following tasks:

a. Implement a stack data structure using an array.

b. Evaluate a postfix expression using the stack.

Objective:

• To understand stack operations (push, pop, peek) using arrays.

• To evaluate a postfix (Reverse Polish Notation) expression using a stack.

Theory:

Stack:

A stack is a Last-In-First-Out (LIFO) data structure, where the last element added is the first to be removed. Basic operations:

• Push: Add element to the top.

• Pop: Remove element from the top.

• Peek: View the top element without removing it.

Postfix Expression:

A postfix expression is a mathematical notation where operators follow operands (e.g., 5 6 + = 5 + 6). It eliminates the need for parentheses.

Evaluation Algorithm:

1. Traverse the expression from left to right.
2. If operand: push to stack.

3. If operator: pop two elements, apply the operator, push result back.
4. Final result is at the top of the stack.

Code1:

```
#include<iostream>
using namespace std;
const int Max=5;
int top=-1;
int stack[Max];
void push(int item)
{
	if(top==Max-1)
		cout<<"\nStack is full";
	else
	{
		top++;
		stack[top]=item;
		cout<<"\n"<<"is added to stack\n";
	}
}
void pop()
{
	if(top=-1)
	{
		cout<<"\nStack is Empty";
	}
	else
	{
		int item=stack[top];
		cout<<"\nDeleted data is "<<item;
		top--;
	}
}
void show()
```

```
{
        int i;
        if(top==-1)
                cout<<"\nStack is Empty";
        else
        {
                cout<<"\nStack element:\n";
                for(i=0;i<=top;i++)
                        cout<<stack[i]<<"\t";
        }
}
int main()
{
        int ch,data;
        while(1)
        {
                cout<<"\n1->Push";
                cout<<"\n2->Pop";
                cout<<"\n3->List stack";
                cout<<"\n4->Exit";
                cout<<"\nEnter your choice:";
                cin>>ch;
                switch(ch)
                {
                        case 1:cout<<"Enter Data:";
                                cin>>data;
                                push(data);break;
                        case 2:pop();break;
                        case 3:show();break;
                        case 4:exit(0);
                }
        }
}
```

Output:

```
1->Push
2->Pop
3->List stack
4->Exit
Enter your choice:1
Enter Data:103
is added to stack
1->Push
2->Pop
3->List stack
4->Exit
Enter your choice:3
Stack element:
101     102     103
1->Push
2->Pop
3->List stack
4->Exit
Enter your choice:2
Deleted data is 103
```

Code2:

```
#include<iostream>
using namespace std;
int stack[20];
int top = -1;
void push(int x)
{
        top++;
        stack[top]=x;
}
int pop()
{
        return stack[top--];
}
```

```
 int main()
{
    char e [50];
        int n1,n2,n3,num;
        cout<<"Enter postfix expression :";
        cin>>e;
        char *ptr = e;
        while(*ptr != '\0')
        {
                if(isdigit(*ptr))
                {
                        num = *ptr - 48;
                        push(num);
           }
           else
           {
                n1 = pop();
                n2 = pop();
                switch(*ptr)
                {
                        case '+':n3=n2+n1;break;
                        case '-':n3=n2-n1;break;
                        case '*':n3=n2*n1;break;
                        case '/':n3=n2/n1;break;
              }
              push(n3);

          }
          ptr++;
   }
    cout<<"\nThe result of expression=\n\n"<<pop();
   return 0;
}
```

Output:

Enter postfix expression :35+59
The result of expression=

Practical 4: Queue Implementation and Simulation

Title:

Write a program to:

a. Implement a queue data structure using an array.

b. Simulate a basic queue system for managing orders..

Objective:

• To implement a queue using arrays.

• To understand queue behavior (FIFO – First In, First Out) through real-life simulation.

Theory:

Queue:

A queue is a linear data structure that follows the FIFO principle:

• Enqueue: Add element to the rear.

• Dequeue: Remove element from the front.

Applications:

• Print queues

• CPU scheduling

• Customer service systems

Code1:

```
#include<iostream>
using namespace std;
const int MAX=3;
int front,rear;
int q[MAX];
void enqueue(int n)
{
	if(rear==MAX-1)
```

```
                cout<<"\nQueue is Full.";
        else
        {
                if(front==-1)
                front=0;
                rear++;
                q[rear]=n;
        }
}
void dequeue()
{
        int n;
        if(front==-1)
                cout<<"\nQueue is Empty";
        else
        {
                n=q[front];
                if(front==rear)
                        front=rear=-1;
                else
                        front=front+1;
                        cout<<"\nDeleted item is="<<n;
        }
}
void display()
{
        int i;
        if(front==-1)
                cout<<"\nQueue is Empty,";
        else
        {
                cout<<"\nQueue is Contains.";
                for(i=front;i<=rear;i++)
                        cout<<"\n\t"<<q[i];
        }
}
int main()
```

```
{
        int n,i,ch;
        front=rear=-1;
        while(1)
        {
                cout<<"\n1.Enqueue \n2.Dequeue \n3.Display \n4.Exit";
                cout<<"Enter your Choice:";
                cin>>ch;
                switch(ch)
                {
                        case 1:cout<<"\nEnter an Element to be inserted:";
                                        cin>>n;
                                        enqueue(n);break;
                        case 2:dequeue();break;
                        case 3:display();break;
                        case 4:exit(0);break;
                        default:cout<<"\nWrong Choice.....";
                }
        }
}
```

Output:

```
1.Enqueue
2.Dequeue
3.Display
4.ExitEnter your Choice:1
Enter an Element to be inserted:20
1.Enqueue
2.Dequeue
3.Display
4.ExitEnter your Choice:3
Queue is Contains.
       10
       20
1.Enqueue
2.Dequeue
3.Display
```

```
4.ExitEnter your Choice:2
Deleted item is=10
1.Enqueue
2.Dequeue
3.Display
4.ExitEnter your Choice:3
Queue is Contains.
        20
```

Code2:

```
#include<iostream>
using namespace std;
#define MAX 3
int orders[MAX];
int front = -1, rear = -1;
void addorder(int ono)
{
        if (rear == MAX-1)
        {
                cout << "Order queue is full! Please wait...";
        }
        Else
        {
                if(front==-1)
                front=0;
                rear++;
                orders[rear] = ono;
                cout << "Order added: " <<ono << endl;
        }
}
void process()
{
```

```
        if (front == -1 && rear == -1)
        {
                cout << "No orders to process.\n";
        }
        else
        {
                cout << "Processing order: " << orders[front] << endl;
                if (front == rear)
                front = rear = -1;
                else
                front++;
        }
}
void showorder()
{
        if (front == -1 && rear == -1)
        {
                cout << "No pending orders.\n";
                return;
        }
                cout << "Pending orders:\n";
                for (int i = front; i <= rear; i++) {
                cout << "- " << orders[i] << endl;
        }
}
int main()
{
        int ch, ordnum;
        while(1)
        {
                cout << "\n--- Order Queue System ---\n";
                cout << "1. Add Order\n";
```

```
            cout << "2. Process Order\n";
            cout << "3. Show pending Order\n";
            cout << "4. Exit\n";
            cout << "Choose an option: ";
            cin >> ch;
            switch (ch)
            {
                  case 1:
                  cout <<"Enter order name: ";
                  cin>>ordnum;
                  addorder(ordnum);break;
                  case 2:process();break;
                  case 3:showorder();break;
                  case 4:exit(0);break;
            }
      }
      return 0;
}
```

Output:

```
----Order Queue System----
1.Add Order
2.Process Order
3.Show Pending Order
4.Exit
Choose an option:1
Enter Order number:103
Order added103
----Order Queue System----
1.Add Order
2.Process Order
3.Show Pending Order
4.Exit
Choose an option:3
```

Pending orders:
-101
-102
-103
----Order Queue System----
1.Add Order
2.Process Order
3.Show Pending Order
4.Exit
Choose an option:2
Processing order:101

Practical 5: Singly Linked List Operations

Title:

Implement a program to manage a singly linked list with the following functionalities:

a. Initialize an empty singly linked list.
b. Add a new node at the start, at the end in the list.
c. Delete a node from a given position in a linked
d. Display Linked List

Objective:

To understand the creation and manipulation of a singly linked list and implement basic insertion and deletion operations.

Theory:

A singly linked list is a dynamic data structure where each node contains data and a pointer to the next node. Unlike arrays, linked lists do not require contiguous memory, allowing dynamic memory allocation.

Every Node consists of:

Data/Info − Info part stores the value of a Node.

Next − Next part stores the address of the next node to which it is connected.

Types of Linked List:

Singly Linked List

Doubly Linked List

Circular Linked List

Singly Linked List Representation of Node:

```
struct node
{
int data;
struct node *next;
};
```

Dynamic Memory Management Functions in C++ new Operator

The new operator requests for the allocation of the block of memory of the given size of type on the Free Store. If sufficient memory is available, a new operator initializes the memory to the default value according to its type and returns the address to this newly allocated memory.

Syntax

```
new data_type;
```

Eg. Node *newnode=new Node;

delete Operator

In C++, delete operator is used to release dynamically allocated memory. It deallocates memory that was previously allocated with new.

Syntax

delete ptr; where, ptr is the pointer to the dynamically allocated memory

Code:

```
#include<iostream>
using namespace std;
```

```
struct node
{
	int data;
	node *next;
};
node *head;
void atbegin(int n)
{
	node *newnode=new node;
	newnode->data=n;
	newnode->next=head;
	head=newnode;
	cout<<"\nNode Inserted at Beginning";
}
void append (int n)
{
	node *newnode,*temp;
	newnode=new node;
	newnode->data=n;
	newnode->next=NULL;
	if(head==NULL)
	{
		head=newnode;
	}
	else
	{
		temp=head;
		while(temp->next!=NULL)
		{
			temp=temp->next;
		}
		temp->next=newnode;
	}
	cout<< "\nNode Appended at End";
}
bool deldata(int num)
{
```

```
        node *temp, *prev;
        temp=head;
        while(temp!=NULL)
        {
                if(temp->data==num)
                {
                        if(temp==head)
                        {
                                head=temp->next;
                   }
                   else
                   {
                        prev->next=temp->next;
                        }
                        delete temp;
                        return temp;
                        }
                        else
                        {
                                prev=temp;
                                temp= temp->next;
                        }
                }
       return false;
}
void display()
{
        node *temp;
        if(head==NULL)
        {
                cout<<"\nList is empty";
        }
        else
        {
                temp=head;
                cout<<"\n List contains:\n";
                while(temp!=NULL)
```

```
        {
   cout<<"\t"<<temp->data;
                temp=temp->next;
        }
}
}
int main()
{
        int n,ch,loc;
        while(1)
        {
                cout<<"\n1.Insert At beginning\n2.Append(At the
end)\n3.Delete\n4.Display";
                cout<<"\nEnter your choice:";
                cin>>ch;
                switch(ch)
                {
                case 1: cout<<"\nEnter Data";
                         cin>>n;
                         atbegin(n);break;
                case 2: cout<<"\nEnter Data";
                        cin>>n;
                         append(n);break;
                case 3: if(head==NULL)
                          cout<<"List is Empty\n";
                                else
                                {
                                        cout<<"\nEnter the number to delete : ";
                                        cin>>n;
                                        if(deldata(n))
                                           cout<<n<<"\ndeleted successfully\n";
                                        else
                                        cout<<n<<"\nNot found in the list\n";
                                }
                                break;
                case 4: display();break;
                case 5: exit(0);break;
```

```
                default:cout<<"\nWrong Choice";
                }

        }
        return 0;
}
```

Output:

```
1.Insert At beginning
2.Append(At the end)
3.Delete
4.Display
Enter your choice:1
Enter Data100
Node Inserted at Beginning
1.Insert At beginning
2.Append(At the end)
3.Delete
4.Display
Enter your choice:2
Enter Data200
Node Appended at End
1.Insert At beginning
2.Append(At the end)
3.Delete
4.Display
Enter your choice:4
 List contains:
        100     200
1.Insert At beginning
2.Append(At the end)
3.Delete
4.Display
Enter your choice:3
Enter the number to delete : 100
100
deleted successfully
```

Practical 6: Implement and Compare Sorting Algorithms

Title:

Write programs to implement and compare the following sorting algorithms:

a. Selection Sort

b. Bubble Sort

c. Insertion Sort

Objective:

- To understand and implement basic comparison-based sorting algorithms.
- To compare their performance based on time complexity and number of comparisons/swaps.

Theory:

1. Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them if they are not in the intended order. The “bubble” sort is called so because the list elements with greater value than their surrounding elements “bubble” towards the end of the list.

For example, after the first pass, the largest element is bubbled towards the right most position.

2. Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the

unsorted part are picked and placed at the correct position in the sorted part.

3.Selection Sort:

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Code1:

```
#include <iostream>
using namespace std;
void sel_sort(int a[],int n)
{
	int i,j,tmp;
	for (i=0;i<n-1;i++)
	{
		for (j=i+1;j<n;j++)
		{
			if(a[i] > a[j])
			{
				tmp =a[i];
				a[i]=a[j];
				a[j]=tmp;
			}
		}
	}
	cout<<"Printing the sorted array:\n";
	for(i=0;i<n;i++)
		cout<<a[i]<<"\t";
}
int main()
{
	int la[100], size, i;
	cout<<"Enter size of array:\n";
```

```
        cin>>size;
        cout<<"Enter"<<size<<"integers:\n";
        for (i=0;i<size;i++)
                cin>>la[i];
        sel_sort(la, size);
        return 0;
}
```

Output:

```
Enter size of array:
6
Enter6integers:
7 8 3 9 5 6
Printing the sorted array:
3       5       6       7       8       9
```

Code2:

```
#include<stdlib.h>
#include<time.h>
#include<iostream>
using namespace std;
void bubble_sort(int a[], int n)
{
        int i=0,j=0,tmp,cnt;
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(a[j] > a[j+1])
                        {
                                tmp = a[j];
                                a[j]=a[j+1];
                                a[j+1] = tmp;
                        }
                }
        }cout<<"\nPrinting th sorted array:\n";
        for(i=0;i<n;i++)
```

```
                cout<<"\t"<<a[i];
}
int main()
{
        int la[1000], n, i;
        cout<<"Enter size of the array:";
        cin>>n;
        cout<<"Numbers are:\n";
        for(i=0;i<n;i++)
        {
                la[i]=rand();
                cout<<"\t"<<la[i];
        }
        time_t begin=time(NULL);
        bubble_sort(la,n);
        time_t end=time(NULL);
        cout<<"\nTime Required in Seconds"<<(end-begin);
        return 0;
}
```

Output:

```
Enter size of the array:5
Numbers are:
      41      18467   6334    26500   19169
Printing th sorted array:
      41      6334    18467   19169   26500
Time Required in Seconds0
```

Code3:

```
#include<iostream>
using namespace std;
void inssort(int num[], int size)
{
        int i, j, temp;
        for(i=0;i<=size-1;i++)
        {
                temp=num[i];
```

```
            j=i-1;
            while(j>=0 && temp<=num[j])
            {
                num[j+1]=num[j];
                j--;
            }
            num[j+1]=temp;
        }
        cout<<"\nSorted Elements are:\n";
        for(i=0;i<size;i++)
        {
            cout<<num[i]<<"\t";
        }
}
int main()
{
        int la[100], size, i;
        cout<<"Enter size of array:\n";
        cin>>size;
        cout<<"Enter"<<size<<"integers:\n";
        for (i=0;i<size;i++)
            cin>>la[i];
        inssort(la, size);
        return 0;
}
```

Output:

```
Enter size of array:
6
Enter6integers:
3 5 2 8 0 1
Sorted Elements are:
0       1       2       3       5       8
```

Practical 7: Tree Traversals on a Binary Tree

Title:

Design a program to manage to
a. Create a Binary Search Tree
b. To perform preorder, post-order and in-order traversals on BST

Objective:

• To understand and implement three basic types of depth-first traversals on a binary tree..

Theory:

A binary tree is a hierarchical structure in which each node has at most two children: left and right.
Traversal refers to visiting each node in a systematic order. The three main depth-first traversal techniques are:

1. Inorder (LNR)
    - o Visit left subtree
    - o Visit node
    - o Visit right subtree
2. Preorder (NLR)
    - o Visit node
    - o Visit left subtree
    - o Visit right subtree
3. Postorder (LRN)
    - o Visit left subtree
    - o Visit right subtree
    - o Visit node

Code:

```
#include<iostream>
const int MAX=7;
using namespace std;
struct node
{
	int data;
	node *left, *right;
};
node *root=NULL;
```

```
void insert(int data)
{
	node *newnode=new node;
	newnode->data=data;
	newnode->left=NULL;
	newnode->right=NULL;
	node *temp;
	if(root==NULL)
	{
		root=newnode;
		return;
	}
	else
	{
		temp=root;
	}
	while(1)
	{
		if(data<temp->data)
		{
			if(temp->left==NULL)
			{

				temp->left=newnode;
				return;
			}
			else
			{
				temp=temp->left;
			}
		}
		else
		{
			if(temp->right==NULL)
			{
				temp->right=newnode;
				return;
```

```
                        }
                        else
                        {
                                temp=temp->right;
                        }
                }
        }
}
void preorder(struct node *root)
{
        if(root!=NULL)
        {
                cout<<root->data<<"\t";
                preorder(root->left);
                preorder(root->right);
        }
}
void inorder(struct node *root)
{
        if(root!=NULL)
        {
                inorder(root->left);
                cout<<root->data<<"\t";
                inorder(root->right);
        }
}
void postorder(struct node *root)
{
        if(root!=NULL)
        {
                postorder(root->left);
                postorder(root->right);
                cout<<root->data<<"\t";
        }
}
int main()
{
```

```
        int i;
        int ar[]={27,14,35,9,11,16,18};
        for(i=0;i<MAX;i++)
        {
                insert(ar[i]);
        }
        cout<<"\nPreorder Traversal\n";
        preorder(root);
        cout<<"\nInorder Traversal\n";
        inorder(root);
        cout<<"\nPostorder Traversal\n";
        postorder(root);
        return 0;
}
```

Output:

```
Preorder Traversal
27      14      9       11      16      18      35
Inorder Traversal
9       11      14      16      18      27      35
Postorder Traversal
11      9       18      16      14      35      27
```

Practical 8: Hash Table Implementation with Collision Handling

Title:

Develop a program to:

a. Store and retrieve data from the hash table.

b. Write a program to implement the collision technique

Objective:

- To understand the concept of hashing for efficient data access.
- To implement basic hashing and collision resolution techniques.

Theory:

Hash Table:
A hash table is a data structure that maps keys to values using a hash function to compute an index where the data is stored.
Key Concepts:
• Hash Function: Converts a key into a valid table index.
• Collision: Occurs when multiple keys map to the same index. Collision Resolution Techniques:
• Linear Probing: If a position is occupied, check the next available slot (sequentially).
• Chaining: Each slot contains a linked list of entries.

In this practical, we use Chaining for collision handling.

Code:

```
#include<iostream>
#define size 10
using namespace std;
struct node
{
	int data;
	struct node *next;
};
node *chain[size];
void insert(int value)
{
	node *newnode = new node;
	newnode->data = value;
	newnode->next = NULL;
	int key = value % size;
	if(chain[key] == NULL)
		chain[key] = newnode;
	else
	{
		struct node *temp = chain[key];
		while(temp->next!=NULL)
		{
			temp = temp->next;
```

```
                }
                temp->next = newnode;
        }
}
void print()
{
        int i;
        for(i=0;i<size;i++)
        {
                struct node *temp = chain[i];
                cout<<"chain["<<i<<"]-->";
                while(temp!=NULL)
                {
                        cout<<temp->data<<"-->";
                        temp = temp->next;
                }
                cout<<"NULL\n";
        }
}
int main()
{
        insert(7);
        insert(0);
        insert(3);
        insert(10);
        insert(4);
        insert(5);
        print();
        return 0;
}
```

Output:

```
chain[0]-->0-->10-->NULL
chain[1]-->NULL
chain[2]-->NULL
chain[3]-->3-->NULL
chain[4]-->4-->NULL
chain[5]-->5-->NULL
```

chain[6]-->NULL
chain[7]-->7-->NULL
chain[8]-->NULL
chain[9]-->NULL