ADS -LAB

1.STACK

```
#include<string>
#include<vector>
#include<iostream>

using namespace std;

class STACK{
   private:
    vector<string> stack;
    public:
     void push(string item)
     {
        cout<<item<<"is pushed into stack";
        stack.push_back(item);

     }
     void pop()
     {
       if(stack.empty())
       {
        cout<<"stack underflow";
        return;
       }
       cout<<stack.back()<<"is popped";
       stack.pop_back();
     }
```

```
    void display()
    {
       if(stack.empty())
      {
        cout<<"stack is empty";
        return;
      }
      for(auto i:stack)
        cout<<i<<" ";

    }

};

int main()
{
  STACK s;
  int n;
  string item;
  cout<<"\n1.PUSH 2.POP 3.DISPLAY ";
  while(1)
  {
  cout<<"\nenter your choice:";
  cin>>n;
  switch(n)
  {
     case 1:cout<<"\nenter any string to push:";
             cin>>item;
            s.push(item);
            break;
    case 2:s.pop();
```

```
        break;
        case 3:s.display();
        break;
    }
    }

}
```

2.QUEUE

```
#include<string>
#include<list>
#include<iostream>

using namespace std;

class QUEUE{
   private:
    list<string> queue;
    public:
      void insert(string item)
      {
          cout<<item<<" is pushed into stack";
          queue.push_back(item);

      }
      void deletee()
      {
        if(queue.empty())
        {
          cout<<"queue underflow";
          return;
```

```
        }
        cout<<queue.front()<<" is popped";
        queue.pop_front();
      }
      void display()
      {
         if(queue.empty())
        {
          cout<<"queue is empty";
          return;
        }
        for(auto i:queue)
          cout<<i<<" ";

      }

};

int main()
{
   QUEUE q;
   int n;
   string item;
   cout<<"\n1 .INSERT 2.DELETE 3.DISPLAY ";
   while(1)
   {
   cout<<"\n\nenter your choice:";
   cin>>n;
   switch(n)
   {
      case 1:cout<<"\nenter any string to push:";
```

```
            cin>>item;
            q.insert(item);
            break;
      case 2:q.deletee();
        break;
       case 3:q.display();
        break;
   }
   }
 return 0;
}
```

3.POLYNOMIAL

```
#include <iostream>
#include <list>
#include <cmath>

using namespace std;

struct Term {
   int coeff;
   int exp;
};

int evalPoly(list<Term> poly, int x) {
   int sum = 0;
   for (auto it : poly) {
      sum += it.coeff * pow(x, it.exp);
```

```
    }
    return sum;
}

list<Term> readPoly() {
    list<Term> poly;
    int n;
    cout << "Enter the number of terms: ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        Term term;
        cout << "Enter the coefficient and degree of term " << i + 1 << ": ";
        cin >> term.coeff >> term.exp;
        poly.push_back(term);
    }
    return poly;
}

void displayPoly(list<Term> poly) {
    for (auto it = poly.begin(); it != poly.end(); it++) {
        cout << it->coeff << "x^" << it->exp;
        if (next(it) != poly.end()) {
            cout << " + ";
        }
    }
}
```

```
int main() {
   int x;
   cout << "Enter the polynomial: \n";
   list<Term> poly = readPoly();
   cout << "Enter the value of x: ";
   cin >> x;
   cout << "The polynomial is: ";
   displayPoly(poly);
   cout << "\nEvaluated value of polynomial at x = " << x << " is: " <<
evalPoly(poly, x);
   return 0;
}
```

4 A.MERGING VECTOR

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> readVector(int n) {
   vector<int> v(n);
   for (int i = 0; i < n; i++) {
      cin >> v[i];
   }
   return v;
}
```

```
void displayVector(vector<int> &v) {
    for (int i : v) {
        cout << i << " ";
    }
    cout << endl;
}

void merge(vector<int> &v, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) {
        L[i] = v[l + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = v[m + 1 + i];
    }

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            v[k++] = L[i++];
        } else {
            v[k++] = R[j++];
```

```
        }
    }

    while (i < n1) {
        v[k++] = L[i++];
    }

    while (j < n2) {
        v[k++] = R[j++];
    }
}

void mergeSort(vector<int> &v, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(v, l, m);
        mergeSort(v, m + 1, r);
        merge(v, l, m, r);
    }
}

int main() {
    int n1, n2;
    cout << "Enter the size of the first vector: ";
    cin >> n1;
    cout << "Enter the elements of the first vector: ";
```

```
    vector<int> v1 = readVector(n1);

    cout << "Enter the size of the second vector: ";
    cin >> n2;
    cout << "Enter the elements of the second vector: ";
    vector<int> v2 = readVector(n2);

    vector<int> v;
    v.insert(v.end(), v1.begin(), v1.end());
    v.insert(v.end(), v2.begin(), v2.end());

    cout << "Vector 1: ";
    displayVector(v1);
    cout << "Vector 2: ";
    displayVector(v2);

    mergeSort(v, 0, v.size() - 1);
    cout << "Sorted merged vector: ";
    displayVector(v);

    return 0;
}
```

4 B. MERGING LIST

```
#include <iostream>
#include <list>
```

```
using namespace std;

list<int> readList(int n) {
   list<int> l;
   for (int i = 0; i < n; i++) {
      int x;
      cin >> x;
      l.push_back(x);
   }
   return l;
}

void displayList(list<int> l) {
   for (int x : l) {
      cout << x << " ";
   }
   cout << endl;
}

void merge(list<int> &lst, int l, int m, int r) {
   int n1 = m - l + 1;
   int n2 = r - m;
   list<int> L, R;

   auto it = lst.begin();
   advance(it, l);
```

```
    for (int i = 0; i < n1; i++) {
        L.push_back(*it++);
    }
    for (int i = 0; i < n2; i++) {
        R.push_back(*it++);
    }

    auto i = L.begin(), j = R.begin(), k = lst.begin();
    advance(k, l);
    while (i != L.end() && j != R.end()) {
        if (*i <= *j) {
            *k++ = *i++;
        } else {
            *k++ = *j++;
        }
    }

    while (i != L.end()) {
        *k++ = *i++;
    }

    while (j != R.end()) {
        *k++ = *j++;
    }
}
```

```
void mergeSort(list<int> &lst, int l, int r) {
   if (l < r) {
      int m = (r + l) / 2;
      mergeSort(lst, l, m);
      mergeSort(lst, m + 1, r);
      merge(lst, l, m, r);
   }
}

int main() {
   int n1, n2;
   cout << "Enter the size of first list: ";
   cin >> n1;
   cout << "Enter the elements of first list: ";
   list<int> l1 = readList(n1);

   cout << "Enter the size of second list: ";
   cin >> n2;
   cout << "Enter the elements of second list: ";
   list<int> l2 = readList(n2);

   list<int> l;
   l.merge(l1);
   l.merge(l2);

   cout << "List 1: ";
```

```
    displayList(l1);
    cout << "List 2: ";
    displayList(l2);

    mergeSort(l, 0, l.size() - 1);
    cout << "Sorted merged list: ";
    displayList(l);

    return 0;
}
```

5.HASHING

```
#include <iostream>
#include <vector>
#include <list>

using namespace std;

class Hashtable {
private:
    vector<list<int>> table;
    int table_size;

public:
    Hashtable(int n) {
        table_size = n;
        table.resize(n);
```

```
}

bool isPrime(int x) {
    if (x <= 1) return false;
    for (int i = 2; i <= x / 2; i++) {
        if (x % i == 0) return false;
    }
    return true;
}

double load_factor(int m) {
    return (double)m / (double)table_size;
}

void rehash() {
    int new_table_size = 2 * table_size;
    while (!isPrime(new_table_size)) {
        new_table_size++;
    }
    vector<list<int>> new_table(new_table_size);
    for (auto &it : table) {
        for (auto &key : it) {
            int index = key % new_table_size;
            new_table[index].push_back(key);
        }
    }
```

```
    table = move(new_table);
    table_size = new_table_size;
}

void insert(int val, int m) {
    int index = val % table_size;
    table[index].push_back(val);
    if (load_factor(m) > 0.75) {
        rehash();
    }
}

void assign() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    cout << "Enter the " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        insert(x, i + 1);
    }
}

void display() {
    int i = 0;
```

```
        for (auto &it : table) {
            cout << "[" << i++ << "]";
            for (auto &val : it) {
                cout << "->" << val;
            }
            cout << endl;
        }
    }
};

int main() {
    Hashtable h1(5);
    h1.assign();
    h1.display();
    return 0;
}
```

6A.LINEAR PROBING

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

class Hashtable {
private:
```

```
    vector<int> table;
    int table_size;

public:
    Hashtable(int n) {
        table_size = n;
        table.resize(n, -1);
    }

    bool isPrime(int x) {
        if (x <= 1) return false;
        for (int i = 2; i <= sqrt(x); i++) {
            if (x % i == 0) return false;
        }
        return true;
    }

    double load_factor(int m) {
        return (double)m / (double)table_size;
    }

    void rehash() {
        int new_table_size = 2 * table_size;
        while (!isPrime(new_table_size)) {
            new_table_size++;
        }
```

```
    vector<int> new_table(new_table_size, -1);
    for (auto val : table) {
        if (val != -1) {
            int index = val % new_table_size;
            while (new_table[index] != -1) {
                index = (index + 1) % new_table_size;
            }
            new_table[index] = val;
        }
    }
    table = move(new_table);
    table_size = new_table_size;
}

void linear_probing(int val, int m) {
    if (load_factor(m) > 0.75) {
        rehash();
    }
    int i = 0;
    int index = val % table_size;
    while (table[index] != -1) {
        index = (index + 1) % table_size;
        i++;
        if (i == table_size) {
            cout << "Table is full" << endl;
            return;
```

```
            }
        }
        table[index] = val;
    }

    void assign() {
        int n;
        cout << "Enter the number of elements: ";
        cin >> n;
        cout << "Enter the " << n << " elements: ";
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            linear_probing(x, i + 1);
        }
    }

    void display() {
        for (int i = 0; i < table_size; i++) {
            cout << "[" << i << "] -> " << table[i] << endl;
        }
        cout << endl;
    }
};

int main() {
```

```
    Hashtable h1(5);
    h1.assign();
    h1.display();
    return 0;
}
```

6.B QUADRATIC PROBING

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

class Hashtable {
private:
    vector<int> table;
    int table_size;

public:
    Hashtable(int n) {
        table_size = n;
        table.resize(n, -1);
    }

    bool isPrime(int x) {
```

```
    if (x <= 1) return false;
    for (int i = 2; i <= sqrt(x); i++) {
        if (x % i == 0) return false;
    }
    return true;
}

double load_factor(int m) {
    return (double)m / (double)table_size;
}

void rehash() {
    int new_table_size = 2 * table_size;
    while (!isPrime(new_table_size)) {
        new_table_size++;
    }
    vector<int> new_table(new_table_size, -1);
    for (auto val : table) {
        if (val != -1) {
            int index = val % new_table_size;
            while (new_table[index] != -1) {
                index = (index + 1) % new_table_size;
            }
            new_table[index] = val;
        }
    }
```

```
    table = move(new_table);
    table_size = new_table_size;
}

void quadratic_probing(int val, int m) {
    if (load_factor(m) > 0.75) {
        rehash();
    }
    int i = 0;
    int index = val % table_size;
    while (table[index] != -1) {
        index = (index + i * i) % table_size;
        i++;
        if (i == table_size) {
            cout << "Table is full" << endl;
            return;
        }
    }
    table[index] = val;
}

void assign() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    cout << "Enter the " << n << " elements: ";
```

```
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            quadratic_probing(x, i + 1);
        }
    }

    void display() {
        for (int i = 0; i < table_size; i++) {
            cout << "[" << i << "] -> " << table[i] << endl;
        }
        cout << endl;
    }
};

int main() {
    Hashtable h1(5);
    h1.assign();
    h1.display();
    return 0;
}
```

7.DOUBLE HASH

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

```
class Hashtable {
private:
vector<int> table;
vector<bool> isOccupied; // To keep track of occupied slots
int table_size;
int prime;
public:
Hashtable(int n) {
table_size = n;
table.resize(n, -1); // -1 indicates an empty slot
isOccupied.resize(n, false);
prime = getPrime(table_size);
}

bool isPrime(int x) {
if (x <= 1) return false;
for (int i = 2; i * i <= x; i++) {
if (x % i == 0) return false;
}
return true;
}

int getPrime(int x) {
while (!isPrime(x)) {
x--;
}
```

```
return x;
}

double load_factor(int m) {
return (double)m / (double)table_size;
}

void rehash() {
int old_size = table_size;
table_size = 2 * table_size;
while (!isPrime(table_size)) {
table_size++;
}

vector<int> old_table = table;
vector<bool> old_isOccupied = isOccupied;
table.clear();
table.resize(table_size, -1);
isOccupied.clear();
isOccupied.resize(table_size, false);
prime = getPrime(table_size);
for (int i = 0; i < old_size; i++) {
if (old_isOccupied[i]) {
insert(old_table[i]);
}
}
```

```
}

int hash1(int val) {

return val % table_size;
}

int hash2(int val) {
return 7 - (val % 7);
}

void insert(int val) {
int index = hash1(val);
int stepSize = hash2(val);
int i = 0;
while (isOccupied[index]) {
index = (index + stepSize) % table_size;
i++;
if (i > table_size) return; // This condiϴon prevents infinite loops
}
table[index] = val;
isOccupied[index] = true;
if (load_factor(i + 1) > 0.75) {
rehash();
}
}
```

```
void assign() {
int n;
cout << "Enter the number of elements: ";
cin >> n;
cout << "Enter the " << n << " elements: ";
for (int i = 0; i < n; i++) {
int x;
cin >> x;
insert(x);
}
}
void display() {
for (int i = 0; i < table_size; i++) {
cout << "[" << i << "]->" << table[i] << endl;

}
}
};

int main() {
Hashtable h1(11);
h1.assign();
h1.display();
return 0;
}
```

8.HEAP SORT

```
#include <iostream>
#include <vector>
using namespace std;
class PriorityQueue {
vector<string> heap;
public:
void heapify(int i, int n) {
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
if (left < n && heap[left] > heap[largest]) {
largest = left;
}
if (right < n && heap[right] > heap[largest]) {
largest = right;
}
if (largest != i) {
swap(heap[i], heap[largest]);
heapify(largest, n);
}
}
void heapsort() {
for (int i = heap.size() / 2 - 1; i >= 0; i--) {
heapify(i, heap.size());
}
```

```
for (int i = heap.size() - 1; i > 0; i--) {
swap(heap[0], heap[i]);
heapify(0, i);
}
}
void insert_item(string item) {
heap.push_back(item);
cout << "Inserted " << item << " into the heap" << endl;
heapsort();
}
void delete_item() {
if (!heap.empty()) {
string item = heap.back(); // Store the last item
heap.pop_back(); // Remove the last element
cout << "Deleted " << item << " from the heap" << endl;
} else {
cout << "Heap is empty" << endl;
}
}
void display() {
cout << "Heap: ";
for (auto item : heap) {
cout << item << " ";
}
cout << endl;
}
```

```
};
int main() {
PriorityQueue pq;
int choice;
string item;
while (true) {
cout << "\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ";
cin >> choice;
switch (choice) {
case 1:
cout << "Enter the item to be inserted: ";
cin >> item;
pq.insert_item(item);
break;
case 2:
pq.delete_item();
break;
case 3:
pq.display();
break;
case 4:
exit(0);
default:
cout << "Invalid choice" << endl;
break;
}
```

```
}
return 0;
}
```

9.BINOMIAL TREE

```
#include <iostream>
#include <vector>
using namespace std;
// Function to determine which Binomial Trees are present in the Binomial Heap
void identifyBinomialTrees(int n) {
vector<int> treeDegrees;
int degree = 0;
while (n > 0) {
if (n & 1) {
treeDegrees.push_back(degree);
}
n >>= 1; // Right shift to check the next bit
degree++;
}
// Output the degrees of the Binomial Trees present
cout << "Binomial Trees present are:";
for (int i = 0; i < treeDegrees.size(); ++i) {
cout << " b" << treeDegrees[i];
}
cout << endl;
}
```

```
int main() {

int n;

// Prompt user for input

cout << "Enter an integer n to identify Binomial Trees present in the Binomial Heap: ";

cin >> n;

identifyBinomialTrees(n);

return 0;

}
```

10.SHELL SORT

```
#include <iostream>

#include <string>

using namespace std;


void shell(string arr[], int n) {
   int gap, j, k;
   for (gap = n / 2; gap > 0; gap /= 2) {
      for (j = gap; j < n; j++) {
         for (k = j - gap; k >= 0; k -= gap) {
            if (arr[k + gap] >= arr[k])
               break;
            else {
               string temp = arr[k + gap];
               arr[k + gap] = arr[k];
               arr[k] = temp;
            }
```

```
            }
        }
    }
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    string arr[n];
    cout << "Enter " << n << " strings:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    cout << "Array before sorting: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    shell(arr, n);
    cout << "Array after sorting: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

11.QUICK SORT

```
#include<iostream>
```

```
#include<vector>

using namespace std;
void swap(vector<int>&arr,int left,int right)
{
   int temp=arr[left];
   arr[left]=arr[right];
   arr[right]=temp;

}
int med(vector<int>&arr,int left,int right)
{
   int mid=(left+right)/2;

   if(arr[right]<arr[left])
     swap(arr,left,right);

   if(arr[mid]<arr[left])
     swap(arr,left,mid);

   if(arr[right]<arr[mid])
     swap(arr,left,mid);


   return mid;
}
```

```
int partition(vector<int>&arr,int left,int right)
{
   int pindex=med(arr,left,right);
   int pval=arr[pindex];

   swap(arr,pindex,right);
   int store=left;

   for(int i=left;i<right;i++)
   {
      if(arr[i]<pval)
      {
         swap(arr,i,store);
         ++store;
      }
   }
   swap(arr,store,right);
   return store;

}

void quick(vector<int>&arr,int left,int right)
{
   if(left<right)
   {
```

```
        int p=partition(arr,left,right);
        quick(arr,left,p-1);
        quick(arr,p+1,right);
    }

}

int main()
{
    int n;
    cout<<"enter number of elements:";
    cin>>n;

    vector<int> arr(n);

    cout<<"enter "<<n<<" elements:";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    cout<<endl;

    cout<<"array before sorting:";
    for(int i=0;i<n;i++)
    {
        cout<<arr[i]<<" ";
```

```
    }
    cout<<endl;
    quick(arr,0,n-1);

     cout<<"array after sorting:";
    for(int i=0;i<n;i++)
    {
       cout<<arr[i]<<" ";
    }
    cout<<endl;

    return 0;
}
```

12.B TREE

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

const int max_keys = 5; // Maximum number of keys a node can hold

// Structure to hold a key and a pointer to the next node
struct Pair {
    int key;
    struct Node* next;
};
```

```
// Node class represents a node in the B-tree
class Node {
public:
   int no_of_keys; // Number of keys currently in the node
   Pair data[max_keys + 1]; // Array to hold keys and pointers, one extra for temporary
   Node* father; // Pointer to the parent node
   Node* first; // Pointer to the first child

   Node(); // Constructor to initialize a node
   bool leaf_node(); // Function to check if the node is a leaf
   void insert_in_a_node(Pair x); // Function to insert a key into the node
   Pair split_a_node(Pair x); // Function to split the node
   Node* next_index(int x); // Function to find the next child based on the key
   void display(); // Function to display the keys in the node
};

Node::Node() : no_of_keys(0), father(nullptr), first(nullptr) {
   for (int i = 0; i <= max_keys; i++) data[i].next = nullptr;
}

bool Node::leaf_node() {
   return (data[0].next == nullptr);
}

void Node::insert_in_a_node(Pair x) {
```

```
    int i = no_of_keys - 1;
    while (i >= 0 && data[i].key > x.key) {
        data[i + 1] = data[i];
        i--;
    }
    data[i + 1] = x;
    no_of_keys++;
}

Pair Node::split_a_node(Pair x) {
    Node* t = new Node;
    Pair my_pair;
    Pair temp[max_keys + 1];
    for (int i = 0; i < no_of_keys; i++) temp[i] = data[i];
    int i = no_of_keys - 1;
    while (i >= 0 && temp[i].key > x.key) {
        temp[i + 1] = temp[i];
        i--;
    }
    temp[i + 1] = x;

    int centre = (no_of_keys + 1) / 2;
    t->first = temp[centre].next;
    for (i = centre + 1; i <= no_of_keys; i++) {
        t->data[i - centre - 1] = temp[i];
    }
```

```
    t->no_of_keys = no_of_keys - centre;
    no_of_keys = centre;

    for (i = 0; i < no_of_keys; i++) {
        data[i] = temp[i];
    }
    t->father = father;
    my_pair.key = temp[centre].key;
    my_pair.next = t;
    return my_pair;
}

Node* Node::next_index(int x) {
    if (x < data[0].key) return first;
    for (int i = 0; i < no_of_keys; i++) {
        if (x <= data[i].key) return data[i - 1].next;
    }
    return data[no_of_keys - 1].next;
}

void Node::display() {
    cout << "(";
    for (int i = 0; i < no_of_keys; i++) cout << data[i].key << " ";
    cout << ")";
}
```

```
// BTree class represents the B-tree
class BTree {
   int m_keys; // Maximum number of keys per node
   Node* root; // Pointer to the root node
public:
   BTree(int n) : m_keys(n), root(nullptr) {}
   void insert(int x); // Function to insert a key into the B-tree
   void display_tree(); // Function to display the B-tree
};

void BTree::insert(int x) {
   Pair my_pair = {x, nullptr};
   if (root == nullptr) {
      root = new Node;
      root->insert_in_a_node(my_pair);
   } else {
      Node* p = root;
      while (!p->leaf_node()) p = p->next_index(x);
      if (p->no_of_keys < m_keys) {
         p->insert_in_a_node(my_pair);
      } else {
         my_pair = p->split_a_node(my_pair);
         while (true) {
            if (p == root) {
               Node* q = new Node;
               q->data[0] = my_pair;
```

```
                q->first = root;
                q->no_of_keys = 1;
                root = q;
                q->first->father = q;
                q->data[0].next->father = q;
                return;
            } else {
                p = p->father;
                if (p->no_of_keys < m_keys) {
                    p->insert_in_a_node(my_pair);
                    return;
                } else {
                    my_pair = p->split_a_node(my_pair);
                }
            }
        }
    }
}

void BTree::display_tree() {
    if (!root) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        int size = q.size();
```

```
        while (size--) {
            Node* p = q.front();
            q.pop();
            p->display();
            cout << " ";
            if (!p->leaf_node()) {
                q.push(p->first);
                for (int i = 0; i < p->no_of_keys; i++) q.push(p->data[i].next);
            }
        }
        cout << endl;
    }
}

// Main function to provide a menu-driven interface
int main() {
    int n, x, op;
    cout << "Enter the number of keys in a node: ";
    cin >> n;
    BTree b(n);
    do {
        cout << "\n**MENU";
        cout << "\n1.Insert \n2.Display \n3.Quit";
        cout << "\nEnter your choice: ";
        cin >> op;
        switch (op) {
```

```
        case 1:
            cout << "\nEnter the key to be inserted: ";
            cin >> x;
            b.insert(x);
            cout << "\nTree after insertion:";
            b.display_tree();
            break;
        case 2:
            b.display_tree();
            break;
        }
    } while (op != 3);
    return 0;
}
```