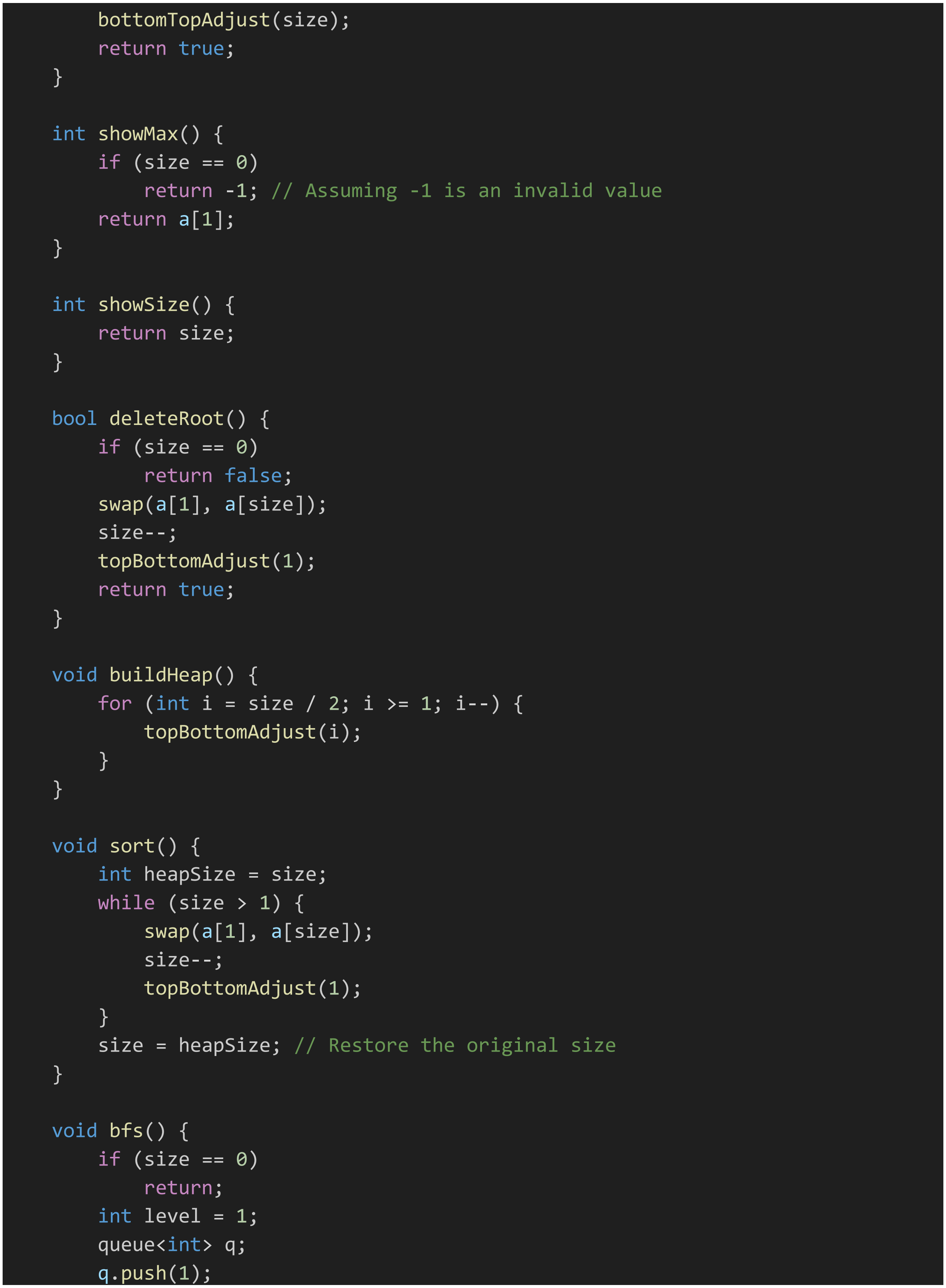
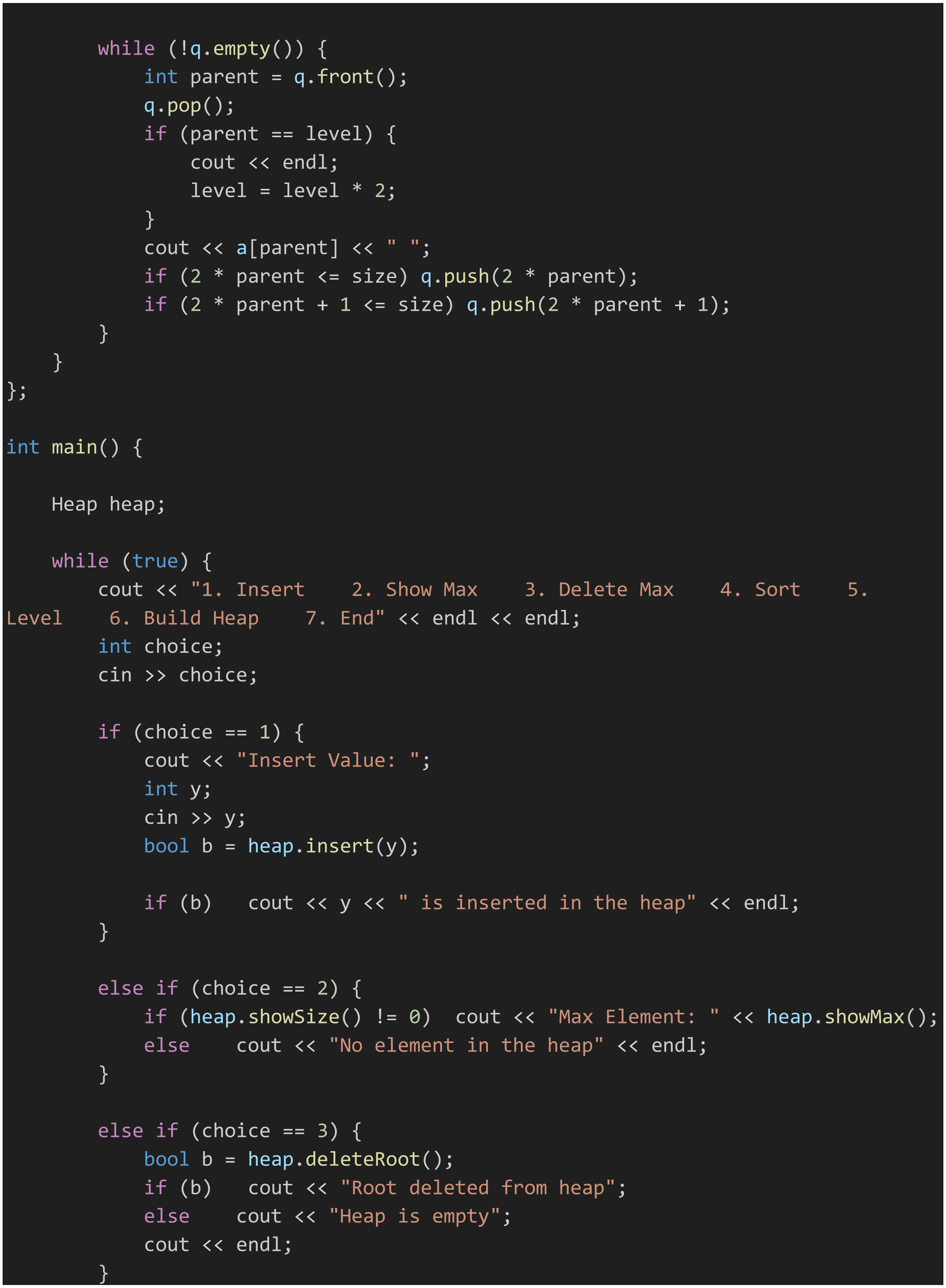
Codes for Finals

Heap

```
///Max Heap

#include<bits/stdc++.h>
using namespace std;

class Heap {
private:
    int a[101], size;

public:
    Heap() {
        size = 0;
    }

private:
    void bottomTopAdjust(int i) {
        while (i > 1 && a[i] > a[i / 2]) {
            swap(a[i], a[i / 2]);
            i = i / 2;
        }
    }

    void topBottomAdjust(int i) {
        int child;
        while (2 * i <= size) {
            child = 2 * i;
            if (child + 1 <= size && a[child + 1] > a[child])
                child++;
            if (a[i] >= a[child])
                break;
            swap(a[i], a[child]);
            i = child;
        }
    }

public:
    bool insert(int val) {
        if (size >= 100)
            return false;
        size++;
        a[size] = val;
```



```
        bottomTopAdjust(size);
        return true;
    }

    int showMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        return a[1];
    }

    int showSize() {
        return size;
    }

    bool deleteRoot() {
        if (size == 0)
            return false;
        swap(a[1], a[size]);
        size--;
        topBottomAdjust(1);
        return true;
    }

    void buildHeap() {
        for (int i = size / 2; i >= 1; i--) {
            topBottomAdjust(i);
        }
    }

    void sort() {
        int heapSize = size;
        while (size > 1) {
            swap(a[1], a[size]);
            size--;
            topBottomAdjust(1);
        }
        size = heapSize; // Restore the original size
    }

    void bfs() {
        if (size == 0)
            return;
        int level = 1;
        queue<int> q;
        q.push(1);
```

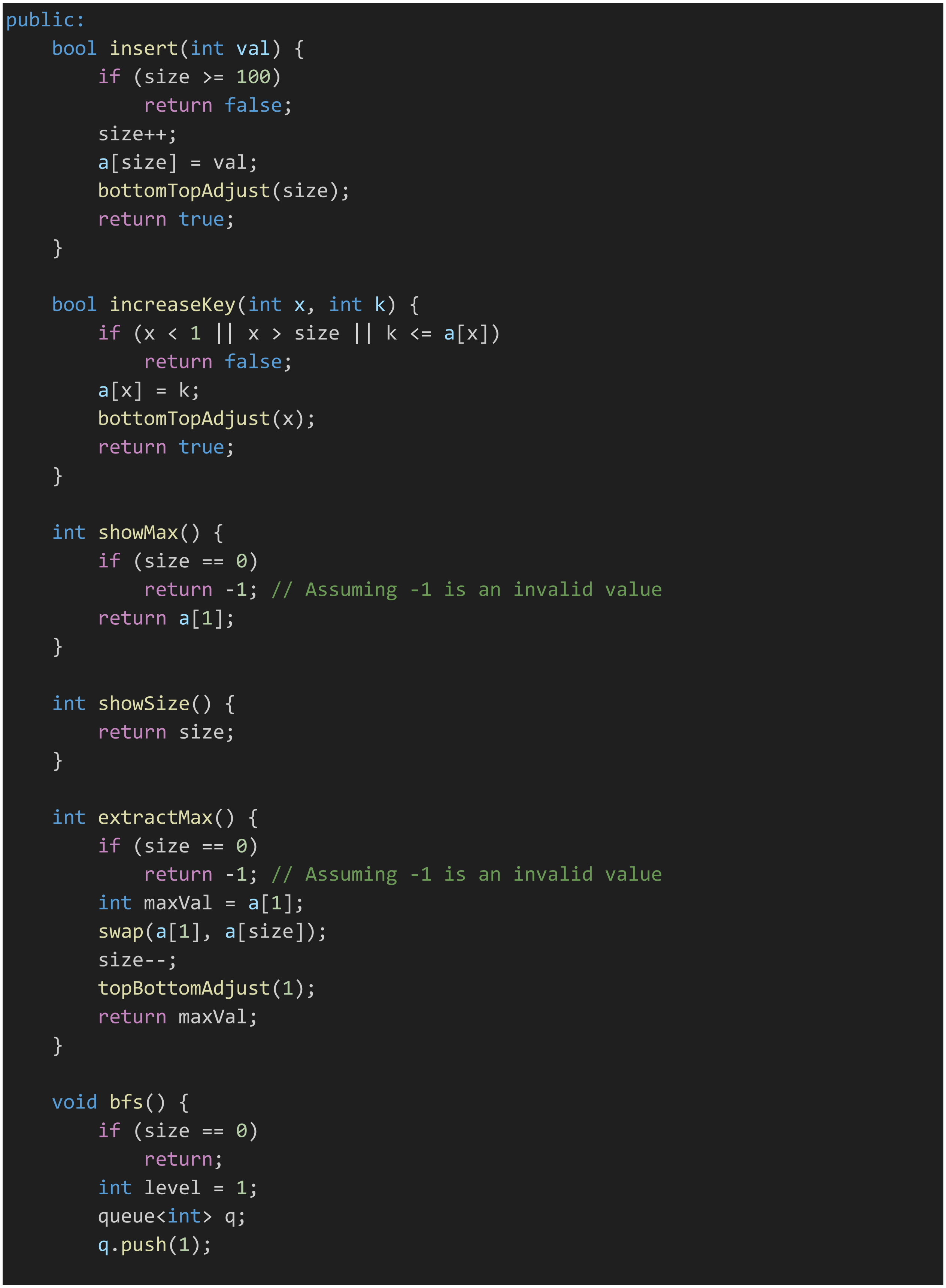


```
        while (!q.empty()) {
            int parent = q.front();
            q.pop();
            if (parent == level) {
                cout << endl;
                level = level * 2;
            }
            cout << a[parent] << " ";
            if (2 * parent <= size) q.push(2 * parent);
            if (2 * parent + 1 <= size) q.push(2 * parent + 1);
        }
    }
};

int main() {

    Heap heap;

    while (true) {
        cout << "1. Insert    2. Show Max    3. Delete Max    4. Sort    5.
Level    6. Build Heap    7. End" << endl << endl;
        int choice;
        cin >> choice;

        if (choice == 1) {
            cout << "Insert Value: ";
            int y;
            cin >> y;
            bool b = heap.insert(y);

            if (b)   cout << y << " is inserted in the heap" << endl;
        }

        else if (choice == 2) {
            if (heap.showSize() != 0)  cout << "Max Element: " << heap.showMax();
            else    cout << "No element in the heap" << endl;
        }

        else if (choice == 3) {
            bool b = heap.deleteRoot();
            if (b)   cout << "Root deleted from heap";
            else    cout << "Heap is empty";
            cout << endl;
        }
```

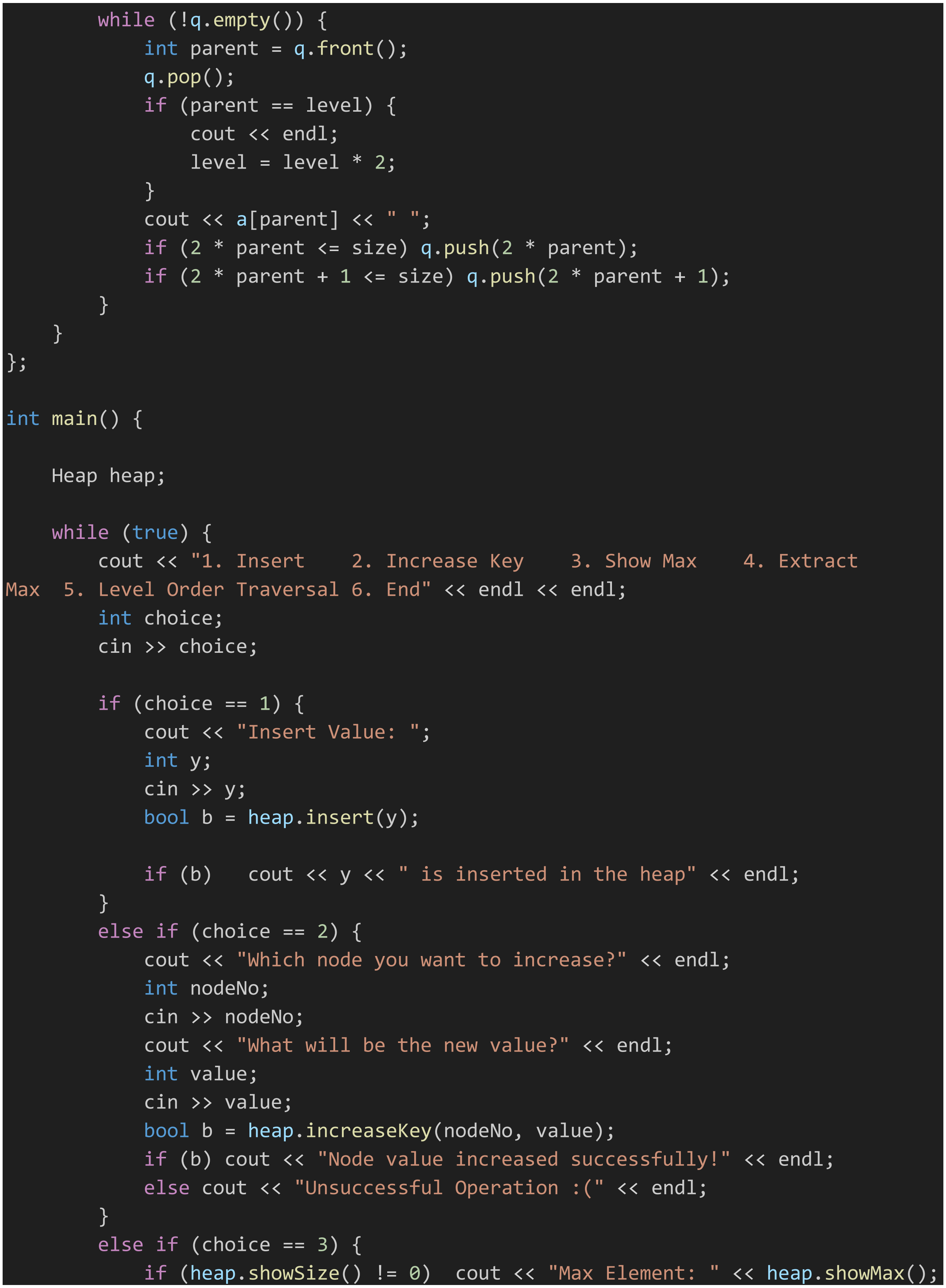
```cpp
        else if (choice == 4) {
            heap.sort();
        }

        else if (choice == 5) {
            cout << "Level Wise Traversal of the heap:" << endl;
            heap.bfs();
            cout << endl;
        }
        else if (choice == 6) {
            if (heap.showSize() == 0)
                cout << "Heap is Empty!" << endl;
            else
                heap.buildHeap();
        }
        else if (choice == 7) {
            break;
        }

        else {
            cout << "Invalid Choice" << endl;
        }
        cout << endl;
    }

    return 0;
}


/*
1 2
1 9
1 8
1 16
1 3
1 7
1 10
1 1
1 4
1 14
*/
```

PQ

```
///Priority Queue

#include<bits/stdc++.h>
using namespace std;

class Heap {
private:
    int a[101], size;

public:
    Heap() {
        size = 0;
    }

private:
    void bottomTopAdjust(int i) {
        while (i != 1) {
            if (a[i] > a[i / 2])
                swap(a[i], a[i / 2]);
            else
                break;
            i = i / 2;
        }
    }

    void topBottomAdjust(int i) {    /// HEAPIFY!
        int pseudoRoot = a[i];
        int pseudoIdx = i;
        while (i <= size / 2) {
            int leftVal = a[2 * i];
            int maxIdx = 2 * i;
            if ((2 * i + 1) <= size && a[2 * i + 1] > leftVal)
                maxIdx = 2 * i + 1;
            if (a[i] < a[maxIdx]) {
                swap(a[i], a[maxIdx]);
            }
            else {
                break;
            }
            i = maxIdx;
        }
    }
```



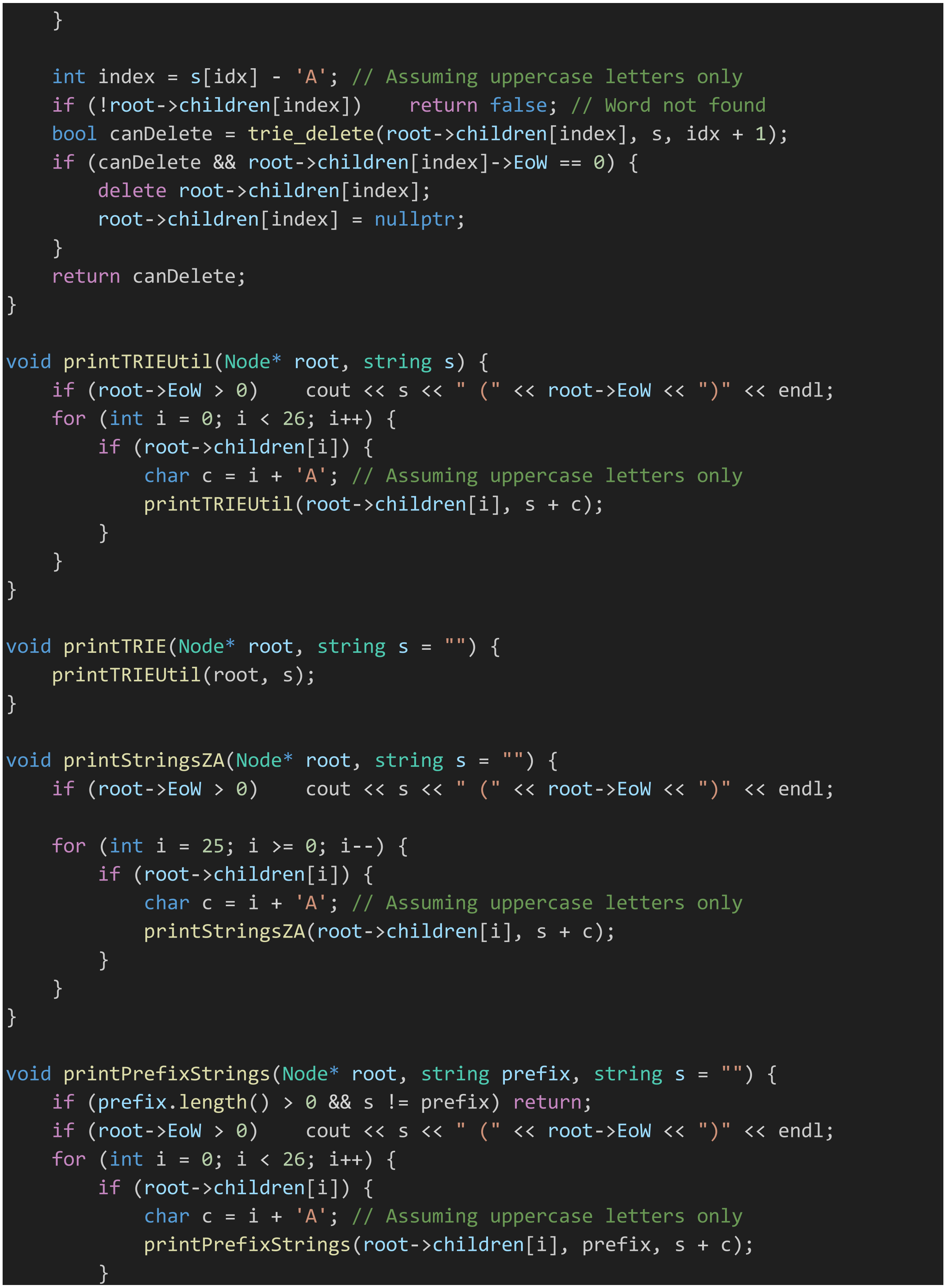
```cpp
public:
    bool insert(int val) {
        if (size >= 100)
            return false;
        size++;
        a[size] = val;
        bottomTopAdjust(size);
        return true;
    }

    bool increaseKey(int x, int k) {
        if (x < 1 || x > size || k <= a[x])
            return false;
        a[x] = k;
        bottomTopAdjust(x);
        return true;
    }

    int showMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        return a[1];
    }

    int showSize() {
        return size;
    }

    int extractMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        int maxVal = a[1];
        swap(a[1], a[size]);
        size--;
        topBottomAdjust(1);
        return maxVal;
    }

    void bfs() {
        if (size == 0)
            return;
        int level = 1;
        queue<int> q;
        q.push(1);
```



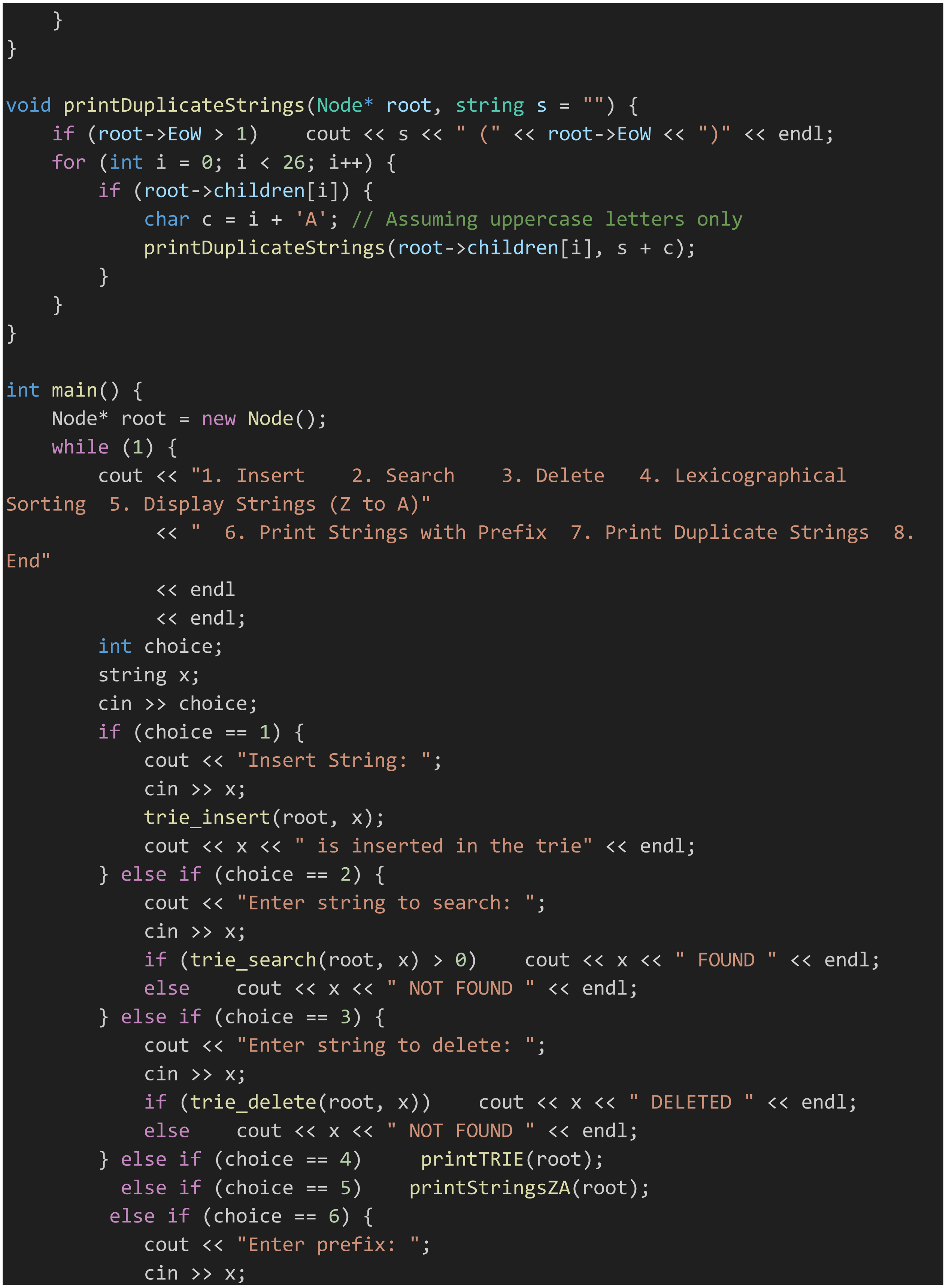
```
        while (!q.empty()) {
            int parent = q.front();
            q.pop();
            if (parent == level) {
                cout << endl;
                level = level * 2;
            }
            cout << a[parent] << " ";
            if (2 * parent <= size) q.push(2 * parent);
            if (2 * parent + 1 <= size) q.push(2 * parent + 1);
        }
    }
};

int main() {

    Heap heap;

    while (true) {
        cout << "1. Insert    2. Increase Key    3. Show Max    4. Extract
Max  5. Level Order Traversal 6. End" << endl << endl;
        int choice;
        cin >> choice;

        if (choice == 1) {
            cout << "Insert Value: ";
            int y;
            cin >> y;
            bool b = heap.insert(y);

            if (b)   cout << y << " is inserted in the heap" << endl;
        }
        else if (choice == 2) {
            cout << "Which node you want to increase?" << endl;
            int nodeNo;
            cin >> nodeNo;
            cout << "What will be the new value?" << endl;
            int value;
            cin >> value;
            bool b = heap.increaseKey(nodeNo, value);
            if (b) cout << "Node value increased successfully!" << endl;
            else cout << "Unsuccessful Operation :(" << endl;
        }
        else if (choice == 3) {
            if (heap.showSize() != 0)  cout << "Max Element: " << heap.showMax();
            else    cout << "No element in the heap" << endl;
        }
        else if (choice == 4) {
            if (heap.showSize() != 0)  cout << "Max element extracted: " << heap.extractMax();
            else    cout << "No element in the heap" << endl;
        }
        else if (choice == 5) {
            cout << "Level Wise Traversal of the heap:" << endl;
            heap.bfs();
            cout << endl;
        }
        else if (choice == 6)
            break;
        else {
            cout << "Invalid Choice" << endl;
        }
        cout << endl;
    }

    return 0;
}


/*
1 2
1 9
1 8
1 16
1 3
1 7
1 10
1 1
1 4
1 14
*/
```

TRIE

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int EoW;
    Node* children[26];
    Node() {
        EoW = 0;
        for (int i = 0; i < 26; i++) {
            this->children[i] = NULL;
        }
    }
};

void trie_insert(Node* root, string s) {
    Node* current = root;
    for (char c : s) {
        int index = c - 'A'; // Assuming uppercase letters only
        if (!current->children[index])     current->children[index] = new Node();
        current = current->children[index];
    }
    current->EoW++;
}

int trie_search(Node* root, string s, int k = 0) {
    Node* current = root;
    for (char c : s) {
        int index = c - 'A'; // Assuming uppercase letters only
        if (!current->children[index])    return 0; // Not found
        current = current->children[index];
    }
    return current->EoW;
}

bool trie_delete(Node* root, string s, int idx = 0) {
    if (!root) return false;
    if (idx == s.length()) {
        if (root->EoW > 0) {
            root->EoW--;
            return true;
        }
        return false;
```



```
    }

    int index = s[idx] - 'A'; // Assuming uppercase letters only
    if (!root->children[index])    return false; // Word not found
    bool canDelete = trie_delete(root->children[index], s, idx + 1);
    if (canDelete && root->children[index]->EoW == 0) {
        delete root->children[index];
        root->children[index] = nullptr;
    }
    return canDelete;
}

void printTRIEUtil(Node* root, string s) {
    if (root->EoW > 0)    cout << s << " (" << root->EoW << ")" << endl;
    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printTRIEUtil(root->children[i], s + c);
        }
    }
}

void printTRIE(Node* root, string s = "") {
    printTRIEUtil(root, s);
}

void printStringsZA(Node* root, string s = "") {
    if (root->EoW > 0)    cout << s << " (" << root->EoW << ")" << endl;

    for (int i = 25; i >= 0; i--) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printStringsZA(root->children[i], s + c);
        }
    }
}

void printPrefixStrings(Node* root, string prefix, string s = "") {
    if (prefix.length() > 0 && s != prefix) return;
    if (root->EoW > 0)    cout << s << " (" << root->EoW << ")" << endl;
    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printPrefixStrings(root->children[i], prefix, s + c);
        }
```

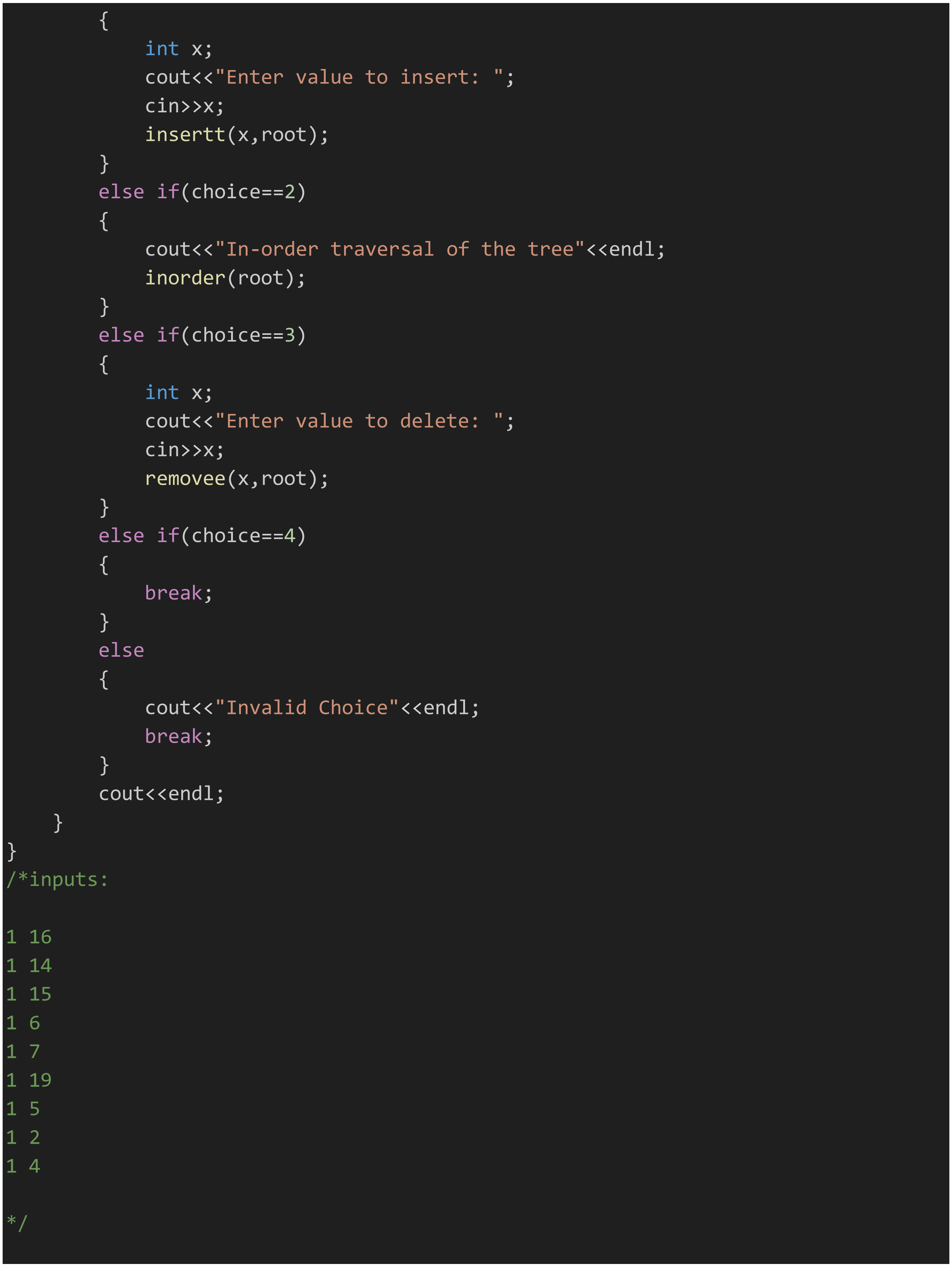


```
    }
}

void printDuplicateStrings(Node* root, string s = "") {
    if (root->EoW > 1)    cout << s << " (" << root->EoW << ")" << endl;
    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printDuplicateStrings(root->children[i], s + c);
        }
    }
}

int main() {
    Node* root = new Node();
    while (1) {
        cout << "1. Insert    2. Search    3. Delete   4. Lexicographical
Sorting  5. Display Strings (Z to A)"
             << "  6. Print Strings with Prefix  7. Print Duplicate Strings  8.
End"
             << endl
             << endl;
        int choice;
        string x;
        cin >> choice;
        if (choice == 1) {
            cout << "Insert String: ";
            cin >> x;
            trie_insert(root, x);
            cout << x << " is inserted in the trie" << endl;
        } else if (choice == 2) {
            cout << "Enter string to search: ";
            cin >> x;
            if (trie_search(root, x) > 0)    cout << x << " FOUND " << endl;
            else    cout << x << " NOT FOUND " << endl;
        } else if (choice == 3) {
            cout << "Enter string to delete: ";
            cin >> x;
            if (trie_delete(root, x))    cout << x << " DELETED " << endl;
            else    cout << x << " NOT FOUND " << endl;
        } else if (choice == 4)     printTRIE(root);
          else if (choice == 5)    printStringsZA(root);
         else if (choice == 6) {
            cout << "Enter prefix: ";
            cin >> x;
            printPrefixStrings(root, x);
        } else if (choice == 7)    printDuplicateStrings(root);
          else if (choice == 8)     break;
          else {
            cout << "Invalid Choice" << endl;
            break;
        }
        cout << endl;
    }
    return 0;
}
```

AVL Tree

```
#include<bits/stdc++.h>
using namespace std;
class Node{
public:
    Node *left;
    Node *right;
    int element;
    int height;
    Node(int x){
        element=x;
        this->left=NULL;
        this->right=NULL;
    }
};


int h(Node *u){
    return (u == NULL) ? -1 : u->height;
}


void LeftRotation(Node *&u){
    Node *v = u->right;
    u->right = v->left;
    v->left = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}
```

```
void RightRotation(Node *&u){
    Node *v = u->left;
    u->left = v->right;
    v->right = u;
    u->height = max(h(u->right), h(u->left))+1;
    v->height = max(h(v->right), h(v->left))+1;
    u=v;
}


void RightLeftRotation(Node *&u){
    RightRotation(u->right);
    LeftRotation(u);
}


void LeftRightRotation(Node *&u){
    LeftRotation(u->left);
    RightRotation(u);
}


void balance( Node *&u ){
    if(u == NULL)  return;
    if( h( u->left ) - h( u->right ) > 1 ){
        if( h( u->left->left ) >= h( u->left->right ) )    RightRotation( u );
        else    LeftRightRotation( u );
    }
    else if( h( u->right ) - h( u->left ) > 1 ){
        if( h( u->right->right ) >= h( u->right->left ) )    LeftRotation(u);
        else    RightLeftRotation(u);
    }
    else    u->height = max( h(u->left), h(u->right) ) + 1;
}


void insertt( int x, Node *&u){
    if( u == NULL )    u = new Node(x);
    else if( x < u->element )    insertt( x,u->left );
    else if( u->element <x )    insertt( x,u->right);
    balance(u);
}


Node* findMin(Node *u){
    if(u->left==NULL)    return u;
    u=u->left;
}


void removee( int x, Node *&u ){
    if( u == NULL )    return;
    if( x < u->element )
        removee( x, u->left );
    else if( u->element < x )
        removee( x, u->right );
    else if( u->left != NULL && u->right != NULL ) // Two children
    {
        u->element = findMin( u->right )->element;
        removee( u->element, u->right );
    }
    else //One child or no child
    {
        Node *oldNode = u;
        u = ( u->left != NULL ) ? u->left : u->right;
        delete oldNode;
    }
    balance( u );
}



void inorder(Node *u)
{
    if(u==NULL)
        return;
    inorder(u->left);
    cout<<u->element<<"  "<<h(u)<<endl;
    inorder(u->right);
}

int main()
{
    Node *root=NULL;
    while(1)
    {
        cout<<"1. Insert in AVL   2. In-order Traversal    3. Delete  4.
End"<<endl<<endl;
        int choice;
        cin>>choice;
        if(choice==1)
```



```
        {
            int x;
            cout<<"Enter value to insert: ";
            cin>>x;
            insertt(x,root);
        }
        else if(choice==2)
        {
            cout<<"In-order traversal of the tree"<<endl;
            inorder(root);
        }
        else if(choice==3)
        {
            int x;
            cout<<"Enter value to delete: ";
            cin>>x;
            removee(x,root);
        }
        else if(choice==4)
        {
            break;
        }
        else
        {
            cout<<"Invalid Choice"<<endl;
            break;
        }
        cout<<endl;
    }
}
/*inputs:

1 16
1 14
1 15
1 6
1 7
1 19
1 5
1 2
1 4

*/
```

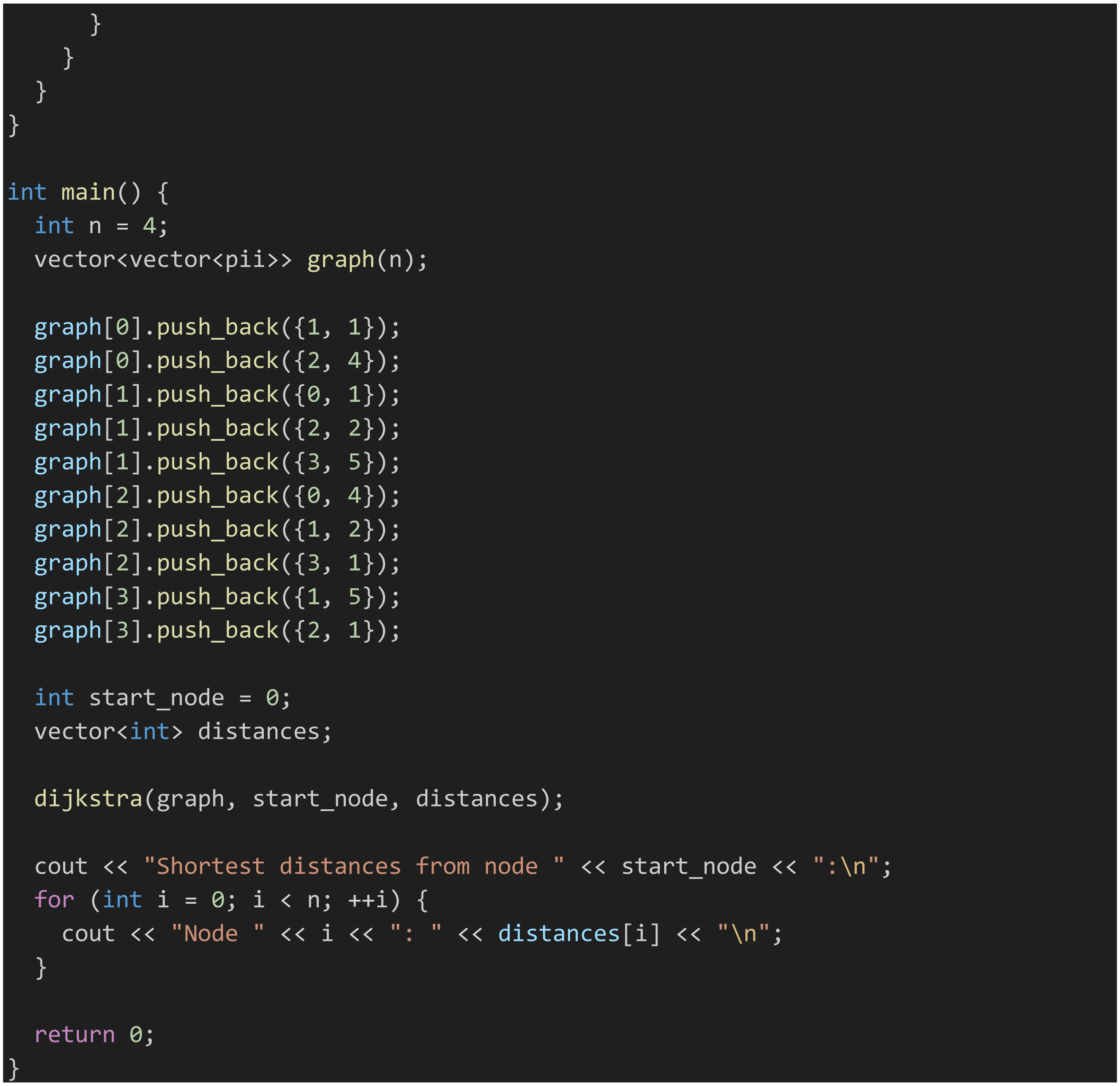
Single Source Shortest Path

Bellman Ford

```
/Bellman Ford
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

typedef pair<int, int> pii;

void bellmanFord(vector<vector<pii>>& graph, int start, vector<int>& distances) {
  int n = graph.size();
  distances.assign(n, INT_MAX);

  distances[start] = 0;

  for (int i = 0; i < n - 1; ++i) {
    for (int u = 0; u < n; ++u) {
      for (const auto& edge : graph[u]) {
        int v = edge.first;
        int weight = edge.second;

        if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
          distances[v] = distances[u] + weight;
        }
      }
    }
  }
}

int main() {
  int n = 4;
  vector<vector<pii>> graph(n);

  graph[0].push_back({1, 1});
  graph[0].push_back({2, 4});
  graph[1].push_back({0, 1});
  graph[1].push_back({2, 2});
  graph[1].push_back({3, 5});
  graph[2].push_back({0, 4});
  graph[2].push_back({1, 2});
  graph[2].push_back({3, 1});
  graph[3].push_back({1, 5});
  graph[3].push_back({2, 1});

  int start_node = 0;
  vector<int> distances;

  bellmanFord(graph, start_node, distances);

  cout << "Shortest distances from node " << start_node << ":\n";
  for (int i = 0; i < n; ++i) {
    cout << "Node " << i << ": " << distances[i] << "\n";
  }

  return 0;
}
```

Dijkstra

```
//Dijkstra
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

typedef pair<int, int> pii;

void dijkstra(vector<vector<pii>>& graph, int start, vector<int>& distances) {
  int n = graph.size();
  distances.assign(n, INT_MAX);
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  pq.push({start, 0});
  distances[start] = 0;

  while (!pq.empty()) {
    int u = pq.top().first;
    pq.pop();

    for (const auto& edge : graph[u]) {
      int v = edge.first;
      int weight = edge.second;

      if (distances[u] + weight < distances[v]) {
        distances[v] = distances[u] + weight;
        pq.push({v, distances[v]});
```



```
      }
    }
  }
}

int main() {
  int n = 4;
  vector<vector<pii>> graph(n);

  graph[0].push_back({1, 1});
  graph[0].push_back({2, 4});
  graph[1].push_back({0, 1});
  graph[1].push_back({2, 2});
  graph[1].push_back({3, 5});
  graph[2].push_back({0, 4});
  graph[2].push_back({1, 2});
  graph[2].push_back({3, 1});
  graph[3].push_back({1, 5});
  graph[3].push_back({2, 1});

  int start_node = 0;
  vector<int> distances;

  dijkstra(graph, start_node, distances);

  cout << "Shortest distances from node " << start_node << ":\n";
  for (int i = 0; i < n; ++i) {
    cout << "Node " << i << ": " << distances[i] << "\n";
  }

  return 0;
}
```

LCS

DP

```
//LCS
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

string findLCS(const string& X, const string& Y) {
  int m = X.length();
  int n = Y.length();

  int dp[m + 1][n + 1];

  for (int i = 0; i <= m; ++i) {
    for (int j = 0; j <= n; ++j) {
        if (i == 0 || j == 0)   dp[i][j] = 0;
        else if (X[i - 1] == Y[j - 1])    dp[i][j] = dp[i - 1][j - 1] + 1;
        else    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
    }
  }

  int i = m, j = n;
  string lcs;

  while (i > 0 && j > 0) {
    if (X[i - 1] == Y[j - 1]) {
      lcs = X[i - 1] + lcs;
      i--;
      j--;
    } else if (dp[i - 1][j] > dp[i][j - 1])  i--;
     else  j--;
  }

  return lcs;
}

int main() {
  string X = "AGGTAB";
  string Y = "GXTXAYB";

  string lcs = findLCS(X, Y);
  cout << "Longest Common Subsequence: " << lcs << endl;

  return 0;
}
```

Recursion

```
#include<bits/stdc++.h>
using namespace std;
int C[50][50];
void init(){
    for(int i=0;i<50;i++){
        for(int j=0;j<50;j++){
            C[i][j]=-1;
        }
    }
}
int LCS(string x,string y,int i,int j){
    if(i==0||j==0){
        C[i][j]=0;
        return C[i][j];
    }
    if(C[i][j]!=-1)    return C[i][j];
    if(x[i-1]==y[j-1])    return C[i][j]=1+LCS(x,y,i-1,j-1);
    return C[i][j]=max(LCS(x,y,i,j-1),LCS(x,y,i-1,j));
}
int main(){
    init();
    cout<<"Enter the text :";
    string x;
    cin>>x;
    cout<<"Enter the pattern :";
    string y;
    cin>>y;
    cout<<LCS(x,y,x.size(),y.size())<<"\n";
}
```

0-1 Knapsack

```
//0-1Knapsack Memoization
#include<bits/stdc++.h>
using namespace std;

int dp[2005][2005];
int c, n;
int p[2005],w[2005];

int knapsack(int i, int j)
{
    if(i<0 || j<=0) return 0;
    if(dp[i][j]!=-1)    return dp[i][j];
    int v1 = knapsack(i-1,j), v2=-1;
    if(w[i]<=j) v2 = p[i] + knapsack(i-1,j-w[i]);
    return dp[i][j] = max(v1, v2);
}

int main()
{
    cin>>c>>n;
    for(int i=0; i<n; i++)  cin>>w[i]>>p[i];
    for(int i=0; i<2005; i++)
        for(int j=0; j<2005; j++)
            dp[i][j] =  -1;

    cout<<knapsack(n-1,c)<<endl;
    for(int i=0; i<=n; i++){
        for(int j=0; j<=c; j++){
            cout<<dp[i][j]<<" ";
        }
        cout<<endl;
    }
}
/*
4 5
1 8
2 4
3 0
2 5
2 3

-1  8  8 -1  8
-1  8  8 -1 12
```

```
-1 -1  8 -1 12
-1 -1  8 -1 13
-1 -1 -1 -1 13
-1 -1 -1 -1 -1
*/
```

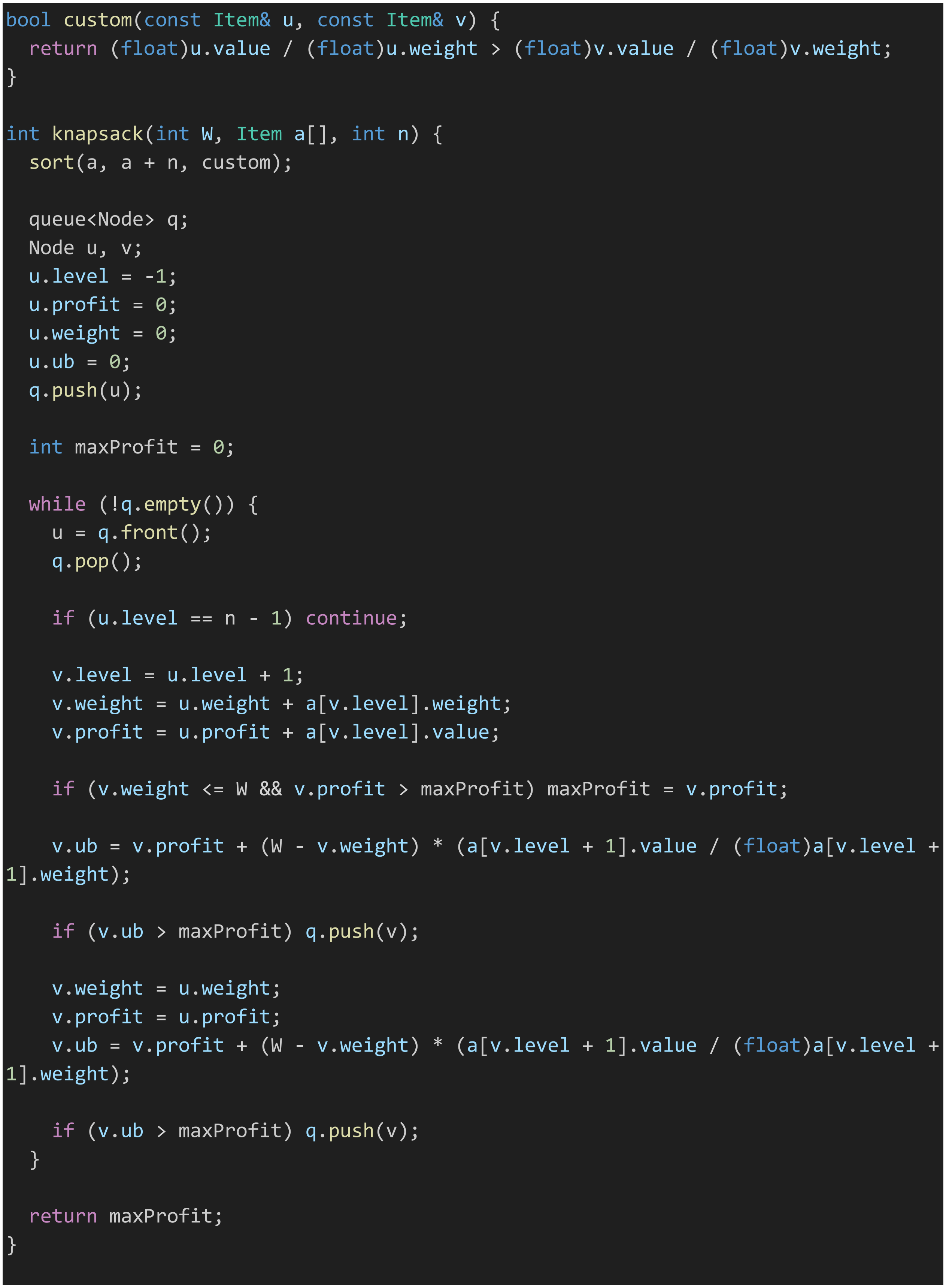
MCM

```
//MCM
#include <bits/stdc++.h>
using namespace std;

const int big = 99999999;

int m[100][100];
int s[100][100];
int d[100];

int MCM(int i, int j) {
    if (i == j) return 0;
    if (m[i][j] != 99999)   return m[i][j];
    int cost = 9999999;
    for (int k = i; k < j; k++) {
        cost = MCM(i, k) + MCM(k + 1, j) + d[i - 1] * d[k] * d[j];
        if (cost < m[i][j]) {
            m[i][j] = cost;
            s[i][j] = k;
        }
    }
    return m[i][j];
}

void printOptimalOrder(int i, int j) {
    if (i == j) cout << "A" << i;
    else {
        cout << "(";
        printOptimalOrder(i, s[i][j]);
        cout << " x ";
        printOptimalOrder(s[i][j] + 1, j);
        cout << ")";
    }
}

int main() {
    int n;
    int row[100], col[100];
    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> row[i] >> col[i];
        d[i] = row[i];
        d[i + 1] = col[i];
    }
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            m[i][j] = 99999;
            s[i][j] = -1;
        }
    }

    cout << "Minimum Cost: " << MCM(1, n) << endl;
    cout << "Optimal Order: ";
    printOptimalOrder(1, n);
    cout << endl;

    return 0;
}
```

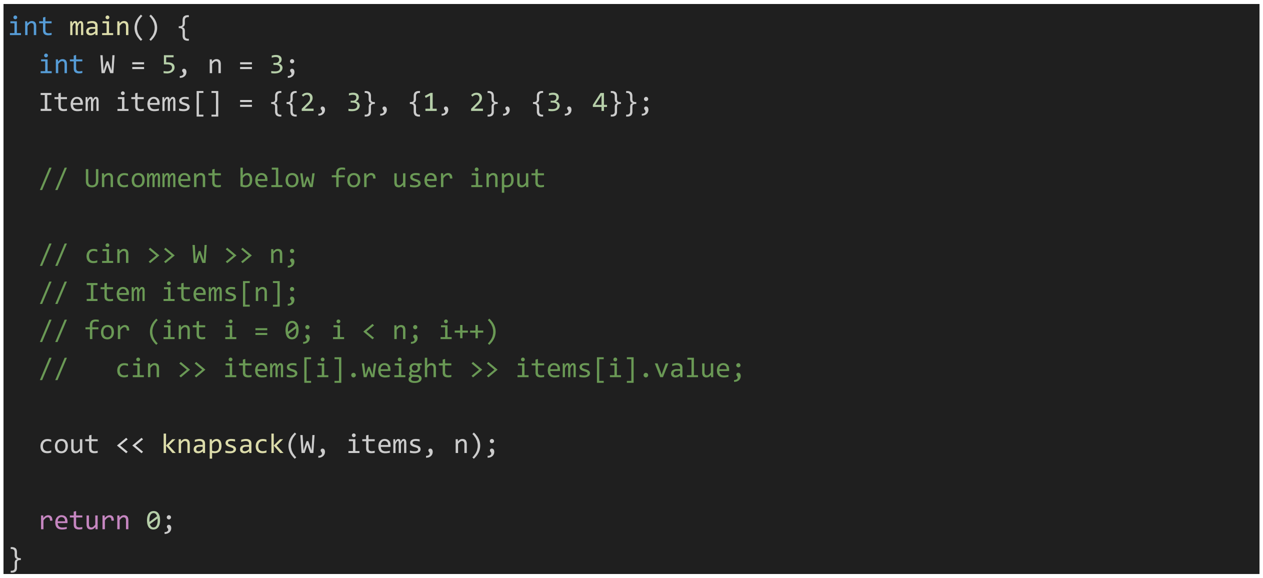
B&B

0-1 knapsack

```
//0-1 knapsack using B&B
#include <bits/stdc++.h>
using namespace std;

class Item {
public:
  int weight;
  int value;
};

class Node {
public:
  int level;
  int profit;
  float ub;
  int weight;
};
```



```cpp
bool custom(const Item& u, const Item& v) {
  return (float)u.value / (float)u.weight > (float)v.value / (float)v.weight;
}

int knapsack(int W, Item a[], int n) {
  sort(a, a + n, custom);

  queue<Node> q;
  Node u, v;
  u.level = -1;
  u.profit = 0;
  u.weight = 0;
  u.ub = 0;
  q.push(u);

  int maxProfit = 0;

  while (!q.empty()) {
    u = q.front();
    q.pop();

    if (u.level == n - 1) continue;

    v.level = u.level + 1;
    v.weight = u.weight + a[v.level].weight;
    v.profit = u.profit + a[v.level].value;

    if (v.weight <= W && v.profit > maxProfit) maxProfit = v.profit;

    v.ub = v.profit + (W - v.weight) * (a[v.level + 1].value / (float)a[v.level + 1].weight);

    if (v.ub > maxProfit) q.push(v);

    v.weight = u.weight;
    v.profit = u.profit;
    v.ub = v.profit + (W - v.weight) * (a[v.level + 1].value / (float)a[v.level + 1].weight);

    if (v.ub > maxProfit) q.push(v);
  }

  return maxProfit;
}
```



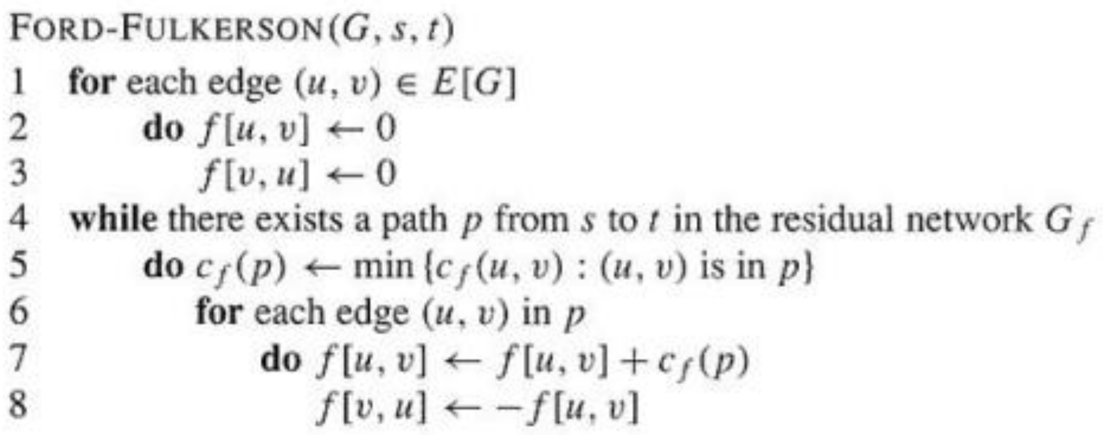
```
int main() {
  int W = 5, n = 3;
  Item items[] = {{2, 3}, {1, 2}, {3, 4}};

  // Uncomment below for user input

  // cin >> W >> n;
  // Item items[n];
  // for (int i = 0; i < n; i++)
  //   cin >> items[i].weight >> items[i].value;

  cout << knapsack(W, items, n);

  return 0;
}
```

TRavelling salesman

//Atto boro code chaibe na inshallah

Max Flow

//Simulation r algorithm



```
FORD-FULKERSON($G, s, t$)
for each edge $(u, v) \in E[G]$
        do $f[u, v] \leftarrow 0$
           $f[v, u] \leftarrow 0$
while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$
        do $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
           for each edge $(u, v)$ in $p$
                do $f[u, v] \leftarrow f[u, v] + c_f(p)$
                   $f[v, u] \leftarrow -f[u, v]$
```

Topo Sort

```cpp
//topo-sort
#include <bits/stdc++.h>
using namespace std;

void topo_sort(int vertices, int edges) {
    vector<char> ans;
    queue<char> q;
    map<char, vector<char>> graph;
    map<char, int> inDegree;

    vector<pair<char, char>> edgeList = {{'A', 'B'},{'A', 'C'},{'B', 'D'},{'B', 'E'},{'C', 'E'},{'D', 'F'},{'E', 'F'}};

    for (int i = 0; i < edges; i++) {
        char a = edgeList[i].first;
        char b = edgeList[i].second;
        graph[a].push_back(b);
        inDegree[b]++;
    }

    for (char c = 'A'; c <= 'F'; c++) if (inDegree[c] == 0) q.push(c);

    while (!q.empty()) {
        char v = q.front();
        q.pop();
        ans.push_back(v);
        for (int i = 0; i < graph[v].size(); i++) {
            char u = graph[v][i];
            inDegree[u]--;
            if (inDegree[u] == 0) {
                q.push(u);
            }
        }
    }

    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i];
        if (i < ans.size() - 1)  cout << "->";
    }
}

int main() {
    int vertices = 6;
    int edges = 7;
    topo_sort(vertices, edges);
    return 0;
}
```

Activity Selection

```
//Activity Selection
#include <stdio.h>
void ActivitySelection(int start[], int finish[], int n)
{
  printf("The following activities are selected:\n");
  int j = 0;
  printf("%d ", j);
  int i;
  for (i = 1; i < n; i++){
    if (start[i] >= finish[j]){
      printf("%d ", i);
      j = i;
    }
  }
}
int main()
{
  int start[] = {1, 3, 2, 0, 5, 8, 11};
  int finish[] = {3, 4, 5, 7, 9, 10, 12};
  int n = sizeof(start) / sizeof(start[0]);
  ActivitySelection(start, finish, n);
  return 0;
}
/* Output
The following activities are selected:
0 1 4 6
*/
```