Trees, Graphs - Assignment

Task 1: Balanced Binary Tree Check

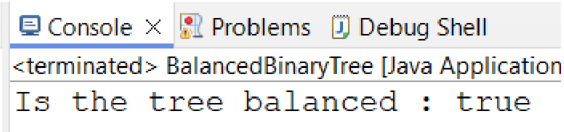
Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

Code:

```
package balancebinarytree;

class TreeNode {
int val;
TreeNode left;
TreeNode right;
TreeNode(int x){
val = x;
}
}
public class BalancedBinaryTree {

private int height(TreeNode root) {
if(root == null)
return 0;
return Math.max(height(root.left), height(root.right)) + 1;
}
public boolean isBalanced(TreeNode root) {
if(root == null)
return true;
if(Math.abs(height(root.left) - height(root.right)) > 1)
return false;
return isBalanced(root.left) && isBalanced(root.right);
}

public static void main(String[] args) {
```

```
BalancedBinaryTree tree = new BalancedBinaryTree();
TreeNode root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
root.left.right = new TreeNode(5);
System.out.println("Is the tree balanced : " +tree.isBalanced(root));
}
}
```

Output:



```
<terminated> BalancedBinaryTree [Java Application
Is the tree balanced : true
```

Task 2: Trie for Prefix Checking

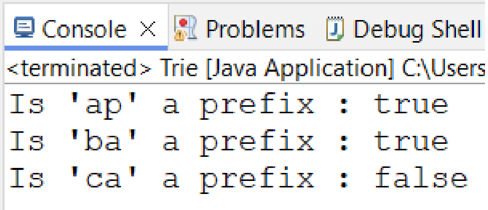
Implement a trie data structure in C# that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

Code:

```
package trieprefixcheking;

class TrieNode {
TrieNode[] children;
boolean isEndofWord;
public TrieNode() {
children = new TrieNode[26];
}
}

public class Trie {
private TrieNode root;
public Trie() {
root = new TrieNode();
```

```java
}
public void insert(String word) {
TrieNode node = root;
for(char ch : word.toCharArray()) {
int index = ch - 'a';
if(node.children[index] == null) {
node.children[index] = new TrieNode();
}
node = node.children[index];
}
node.isEndofWord = true;
}
public boolean isPrefix(String prefix) {
TrieNode node = root;
for(char ch : prefix.toCharArray()) {
int index = ch - 'a';
if(node.children[index] == null) {
return false;
}
node = node.children[index];
}
return true;
}
public static void main(String[] args) {
Trie trie = new Trie();
trie.insert("apple");
trie.insert("app");
trie.insert("bat");
trie.insert("ball");
System.out.println("Is 'ap' a prefix : "+trie.isPrefix("ap"));
System.out.println("Is 'ba' a prefix : "+trie.isPrefix("ba"));
System.out.println("Is 'ca' a prefix : "+trie.isPrefix("ca"));
}
}
```

Output:



```
<terminated> Trie [Java Application] C:\Users
Is 'ap' a prefix : true
Is 'ba' a prefix : true
Is 'ca' a prefix : false
```

Task 3: Implementing Heap Operations

Code a min-heap in Java with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation.

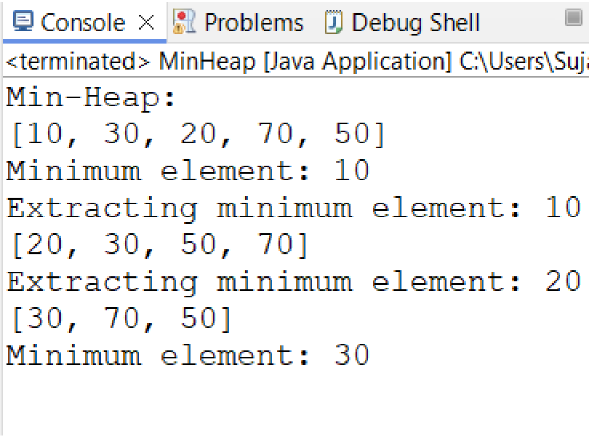
Code:

```
package minheap;
import java.util.ArrayList;
import java.util.List;

class MinHeap {
private List<Integer> heap;
public MinHeap() {
heap = new ArrayList<>();
}
private int parent(int i) { return (i - 1) / 2; }
private int leftChild(int i) { return 2 * i + 1; }
private int rightChild(int i) { return 2 * i + 2; }

public void insert(int key) {
heap.add(key);
int i = heap.size() - 1;
while (i != 0 && heap.get(parent(i)) > heap.get(i)) {
swap(i, parent(i));
i = parent(i);
}
```

```
}
public int extractMin() {
if (heap.size() == 0) {
throw new IllegalStateException("Heap is empty");
}
if (heap.size() == 1) {
return heap.remove(0);
}
int root = heap.get(0);
heap.set(0, heap.remove(heap.size() - 1));
heapifyDown(0);
return root;
}
public int getMin() {
if (heap.size() == 0) {
throw new IllegalStateException("Heap is empty");
}
return heap.get(0);
}
private void swap(int i, int j) {
int temp = heap.get(i);
heap.set(i, heap.get(j));
heap.set(j, temp);
}
private void heapifyUp(int i) {
while (i != 0 && heap.get(parent(i)) > heap.get(i)) {
swap(i, parent(i));
i = parent(i);
}
}
private void heapifyDown(int i) {
int smallest = i;
int left = leftChild(i);
int right = rightChild(i);
```

```
if (left < heap.size() && heap.get(left) < heap.get(smallest)) {
smallest = left;
}
if (right < heap.size() && heap.get(right) < heap.get(smallest)) {
smallest = right;
}
if (smallest != i) {
swap(i, smallest);
heapifyDown(smallest);
}
}
public void printHeap() {
System.out.println(heap);
}

public static void main(String[] args) {
MinHeap minHeap = new MinHeap();
minHeap.insert(30);
minHeap.insert(20);
minHeap.insert(10);
minHeap.insert(70);
minHeap.insert(50);

System.out.println("Min-Heap:");
minHeap.printHeap();
System.out.println("Minimum element: " + minHeap.getMin());
System.out.println("Extracting minimum element: " + minHeap.extractMin());
minHeap.printHeap();
System.out.println("Extracting minimum element: " + minHeap.extractMin());
minHeap.printHeap();
System.out.println("Minimum element: " + minHeap.getMin());
}
}
```

Output:



```
<terminated> MinHeap [Java Application] C:\Users\Suj
Min-Heap:
[10, 30, 20, 70, 50]
Minimum element: 10
Extracting minimum element: 10
[20, 30, 50, 70]
Extracting minimum element: 20
[30, 70, 50]
Minimum element: 30
```

Task 4: Graph Edge Addition Validation

Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

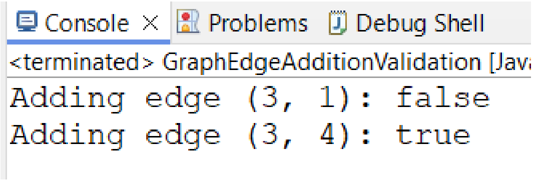
Code:

```
package graphedgeaddition;
import java.util.*;

class GraphEdgeAdditionValidation {
private Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

public void addEdge(int v, int w) {
adjacencyList.computeIfAbsent(v, k -> new ArrayList<>()).add(w);
}
public boolean addEdgeWithCycleCheck(int v, int w) {
adjacencyList.computeIfAbsent(v, k -> new ArrayList<>()).add(w);
if (hasCycle()) {
adjacencyList.get(v).remove((Integer) w);
return false;
}
return true;
```

```
}
private boolean hasCycle() {
Set<Integer> visited = new HashSet<>();
Set<Integer> recStack = new HashSet<>();
for (int vertex : adjacencyList.keySet()) {
if (detectCycle(vertex, visited, recStack)) {
return true;
}
}
return false;
}
private boolean detectCycle(int vertex, Set<Integer> visited, Set<Integer> recStack)
{
if (recStack.contains(vertex)) {
return true;
}
if (visited.contains(vertex)) {
return false;
}
visited.add(vertex);
recStack.add(vertex);
for (int neighbor : adjacencyList.getOrDefault(vertex, new ArrayList<>())) {
if (detectCycle(neighbor, visited, recStack)) {
return true;
}
}
recStack.remove(vertex);
return false;
}
public static void main(String[] args) {
GraphEdgeAdditionValidation graph = new GraphEdgeAdditionValidation();
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
```

```
System.out.println("Adding edge (3, 1): " + graph.addEdgeWithCycleCheck(3, 1));
System.out.println("Adding edge (3, 4): " + graph.addEdgeWithCycleCheck(3, 4));
}
}
```

Output:



```
<terminated> GraphEdgeAdditionValidation [Java
Adding edge (3, 1): false
Adding edge (3, 4): true
```

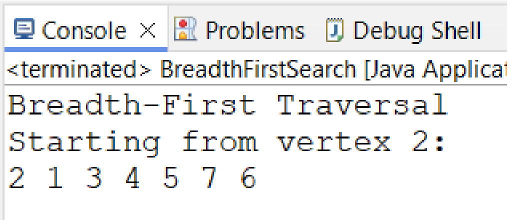
Task 5: Breadth-First Search (BFS) Implementation

For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

Code:

```
package breadthfirstsearch;
import java.util.*;

class BreadthFirstSearch {
private Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

public void addEdge(int v, int w) {
adjacencyList.computeIfAbsent(v, k -> new ArrayList<>()).add(w);
adjacencyList.computeIfAbsent(w, k -> new ArrayList<>()).add(v);
}
public void BFS(int startVertex) {
Set<Integer> visited = new HashSet<>();
Queue<Integer> queue = new LinkedList<>();

visited.add(startVertex);
queue.add(startVertex);
while (!queue.isEmpty()) {
int currentVertex = queue.poll();
System.out.print(currentVertex + " ");

for (int neighbor : adjacencyList.get(currentVertex)) {
if (!visited.contains(neighbor)) {
visited.add(neighbor);
queue.add(neighbor);
}
}
}
}

public static void main(String[] args) {
BreadthFirstSearch graph = new BreadthFirstSearch();
graph.addEdge(1, 2);
graph.addEdge(1, 3);
graph.addEdge(2, 3);
graph.addEdge(3, 4);
graph.addEdge(4, 2);
graph.addEdge(3, 5);
graph.addEdge(5, 6);
graph.addEdge(4, 7);
graph.addEdge(6, 7);

System.out.println("Breadth-First Traversal");
System.out.println("Starting from vertex 2:");
graph.BFS(2);
}
}
```

Output:



```
<terminated> BreadthFirstSearch [Java Applicat
Breadth-First Traversal
Starting from vertex 2:
2 1 3 4 5 7 6
```

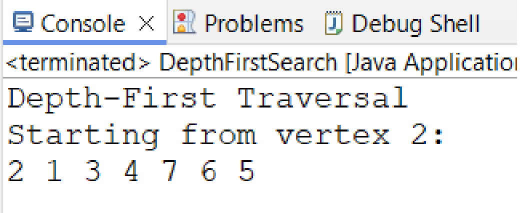
Task 6: Depth-First Search (DFS) Recursive

Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.

Code:

```
package depthfirstsearch;
import java.util.*;

class DepthFirstSearch {
private Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

public void addEdge(int v, int w) {
adjacencyList.computeIfAbsent(v, k -> new ArrayList<>()).add(w);
adjacencyList.computeIfAbsent(w, k -> new ArrayList<>()).add(v);
}
private void DFSUtil(int v, Set<Integer> visited) {
visited.add(v);
System.out.print(v + " ");
for (int neighbor : adjacencyList.get(v)) {
if (!visited.contains(neighbor)) {
DFSUtil(neighbor, visited);
}
}
}
public void DFS(int startVertex) {
Set<Integer> visited = new HashSet<>();
DFSUtil(startVertex, visited);
}
public static void main(String[] args) {
DepthFirstSearch graph = new DepthFirstSearch();
graph.addEdge(1, 2);
graph.addEdge(1, 3);
graph.addEdge(2, 3);
graph.addEdge(3, 4);
graph.addEdge(4, 2);
graph.addEdge(3, 5);
graph.addEdge(5, 6);
graph.addEdge(4, 7);
graph.addEdge(6, 7);

System.out.println("Depth-First Traversal");
System.out.println("Starting from vertex 2:");
graph.DFS(2);
}
}
```

Output:



```
<terminated> DepthFirstSearch [Java Applicatio
Depth-First Traversal
Starting from vertex 2:
2 1 3 4 7 6 5
```