Revision Lab 2

Task 1 About Binary Trees

Answer these questions about Binary Trees.

a) What is the search order for DFS Pre-order search, In-order search and Post-order search?

b) What is a full binary tree? What is a complete binary tree?

c) What is the formula used to calculate the balanced factor of a node?

d) How do you tell if a binary tree is balanced?

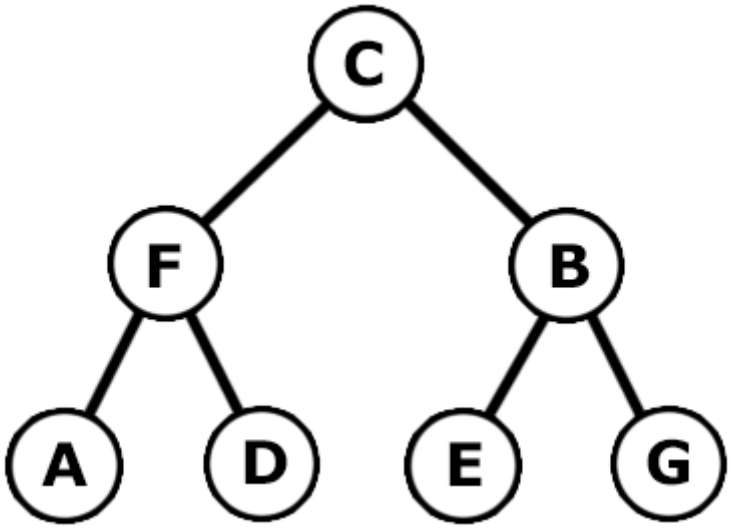
Task 2 Tree Exercises

Fill in the height, depth and balanced factor of each tree below.

Write the order of nodes for each search: BFS, Pre-order Search, In-order Search and Post-order Search.

Determine if the tree is full or complete.

1)



Node	Height	Depth	Balanced Factor
A			
B			
C			
D			
E			
F			
G			

BFS:

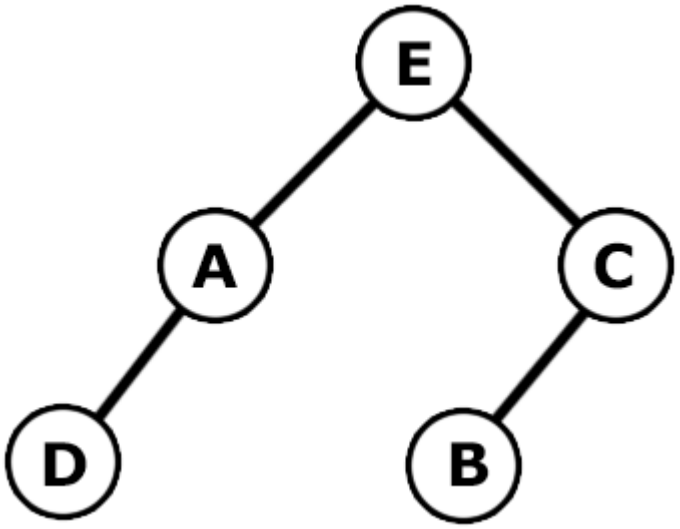
Pre-Order:

In-Order:

Post-Order:

Is the tree full or complete? If not, why?

2)



Node	Height	Depth	Balanced Factor
A			
B			
C			
D			
E			

BFS:

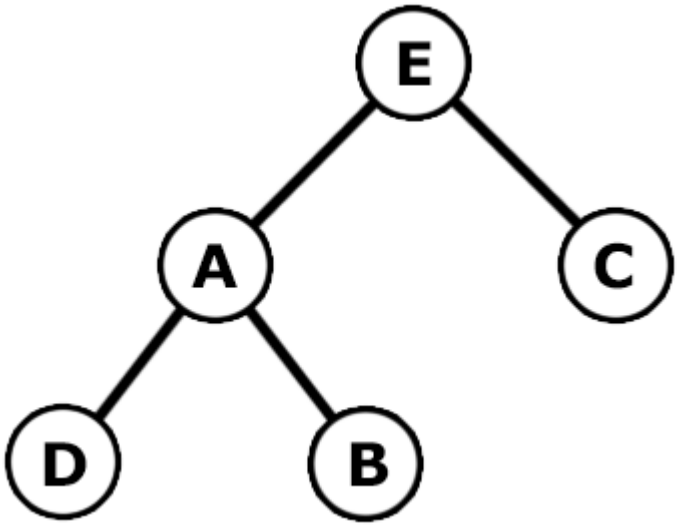
Pre-Order:

In-Order:

Post-Order:

Is the tree full or complete? If not, why?

3)



Node	Height	Depth	Balanced Factor
A			
B			
C			
D			
E			

BFS:

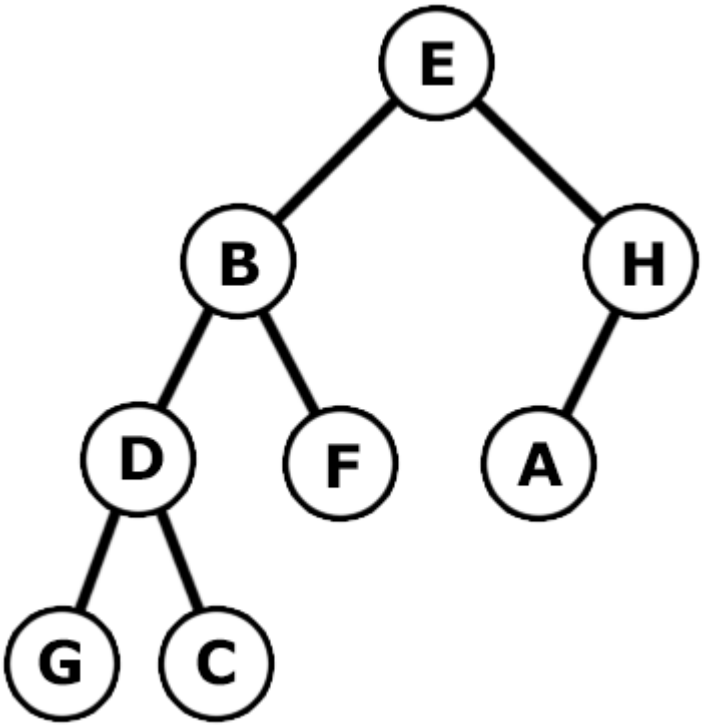
Pre-Order:

In-Order:

Post-Order:

Is the tree full or complete? If not, why?

4)



Node	Height	Depth	Balanced Factor
A			
B			
C			
D			
E			
F			
G			
H			

BFS:

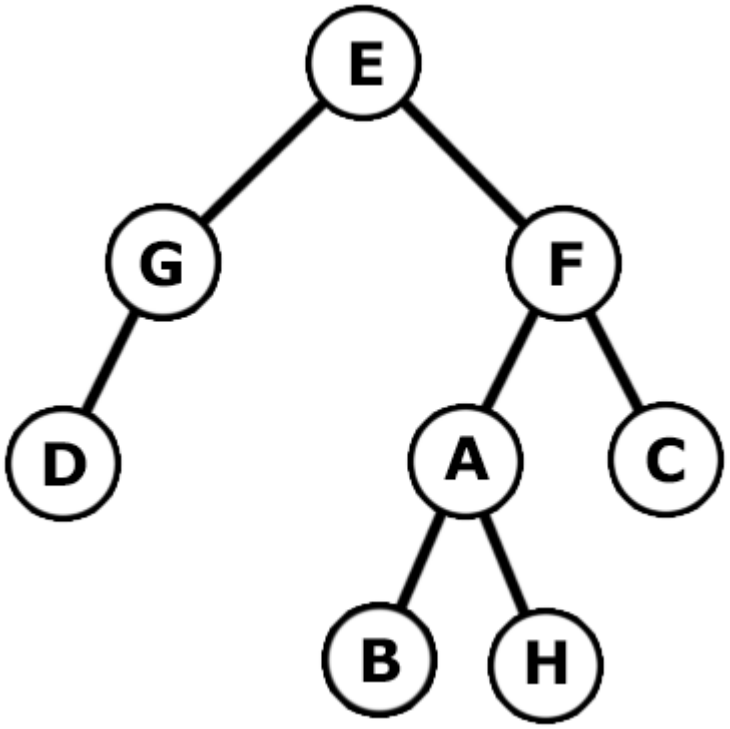
Pre-Order:

In-Order:

Post-Order:

Is the tree full or complete? If not, why?

5)



Node	Height	Depth	Balanced Factor
A			
B			
C			
D			
E			
F			
G			
H			

BFS:

Pre-Order:

In-Order:

Post-Order:

Is the tree full or complete? If not, why?

Task 3 BTree and TreeVisitor Class

Check out BTree and TreeVisitor class from Lab 8 and familiarize yourself with how to implement the functions. Fill in the missing parts of the TreeVisitor.h and BTree.h below from memory.

TreeVisitor.h

```
#pragma once
#include <iostream>

template <class T>
class TreeVisitor
{
public:
	virtual ~TreeVisitor() {} //virtual default destructor
	virtual void preVisit(const T& aKey) const {}
	virtual void postVisit(const T& aKey) const {}
	virtual void inVisit(const T& aKey) const {}

	virtual void visit(const T& aKey) const
	{
		std::cout << aKey << " ";
	}
};

template <class T>
class PostOrderVisitor : public TreeVisitor<T> {
public:
	virtual void postVisit(const T& aKey) const {
		visit(aKey);
	}
};

//////(Fill in missing template class PreOrderVisitor)

template <class T>
class InOrderVisitor : public TreeVisitor<T> {
public:
	virtual void inVisit(const T& aKey) const {
		visit(aKey);
	}
};
```

BTree.h

```
#pragma once
#include <stdexcept>
#include "TreeVisitor.h"
template<class T>
class BTree {
private:
	const T* fKey;
	BTree <T>* fLeft;
	BTree <T>* fRight;
	BTree() :fKey((T*)0) {
		fLeft = &NIL;
		fRight = &NIL;
	}
public:
	static BTree<T> NIL;

	BTree(const T& aKey) /////(Fill in what is missing)
	{
		/////(Fill in what is missing)
	}

	~BTree()
	{
		/////(Fill in what is missing)
	}

	bool isEmpty() const
	{
		/////(Fill in what is missing)
	}

	const T& key() const
	{
		/////(Fill in what is missing)
	}

	BTree& left() const
	{
		/////(Fill in what is missing)
	}

	BTree& right() const
	{
		if (isEmpty())
			throw std::domain_error("Empty BTree");
		return *fRight;
	}

	void attachLeft(BTree<T>* aBTree)
	{
		/////(Fill in what is missing)
	}
```

```
	void attachRight(BTree<T>* aBTree)
	{
		if (isEmpty())
			throw std::domain_error("Empty BTree");
		if (fRight != &NIL)
			throw std::domain_error("Non-empty sub tree");
		fRight = new BTree<T>(*aBTree);//makes allocation on heap
	}

	BTree* detachLeft()
	{
		//////(Fill in what is missing)
	}

	BTree* detachRight()
	{
		if (isEmpty())
			throw std::domain_error("Empty BTree");
		BTree<T>& Result = *fRight; //changed to pointer variable
		fRight = &NIL;
		return &Result;
	}

	void transverseDepthFirst(const TreeVisitor<T>& aVisitor) const
	{
		//////(Fill in what is missing)
	}
};

template<class T>

BTree<T> BTree<T>::NIL;
```