CS302 – Data Structures using C++

Final Exam

Instructor: Dr. Kostas Alexis | Teaching Assistants: Shehryar Khattak, Mustafa Solmaz
Semester: Fall 2018
Date: December 14 2018

Student First Name ____________________ **Student Last Name** ____________________
Student NSHE ID ____________________ **Student E-mail** ____________________

Grade [XYZ/100]:

- Q1: _____ (/ 15%)
- Q2: _____ (/ 35%)
- Q3: _____ (/ 25%)
- Q4: _____ (/ 25%)

Final Grade:

Note: Maximum effective grade for this exam is 100. If you had "bonus grade" from your midterm we will add it.

Question 1 [Topic: Heaps] [Percentage: 15%]: A heap data structure can be efficiently implemented in a range using the C++ Standard Template Library (STL). In this exercise you are asked to utilize STL in order to build and manipulate a heap efficiently. In particular, you are requested to:

1. Make a heap consisting of 10 different integers of your own choosing. Include a statement to show the maximum element of the heap.
2. Add a new value that is the mean of the random values you creates in the previous step. Floor the value if needed (truncate decimal part).
3. Delete the maximum element of the heap and
4. Sort the heap.

Write down the code to achieve all the operations mentioned above and clarify which section of the code does what.

Question 2 [Topic: Binary Trees] [Percentage: 35%]: Consider the following subsets of code:

File: BinaryNodeTree.h

```
#ifndef BINARY_NODE_TREE_
#define BINARY_NODE_TREE_

#include "BinaryTreeInterface.h"
#include "BinaryNode.h"
#include "PrecondViolatedExcept.h"
#include "NotFoundException.h"
#include <memory>

template<class ItemType>
class BinaryNodeTree : public BinaryTreeInterface<ItemType>
{
private:
   std::shared_ptr<BinaryNode<ItemType>> rootPtr;

protected:
   //------------------------------------------------------------
   // Protected Utility Methods Section:
   // Recursive helper methods for the public methods.
   //------------------------------------------------------------
   int getHeightHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const;
   int getNumberOfNodesHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr)
const;

   // Recursively adds a new node to the tree in a left/right fashion to keep
tree balanced.
   auto balancedAdd(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                    std::shared_ptr<BinaryNode<ItemType>> newNodePtr);

   // Removes the target value from the tree.
   virtual auto removeValue(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                            const ItemType target, bool& isSuccessful);

   // Copies values up the tree to overwrite value in current node until
   // a leaf is reached; the leaf is then removed, since its value is stored in
the parent.
   auto moveValuesUpTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);

   // Recursively searches for target value.
   virtual auto findNode(std::shared_ptr<BinaryNode<ItemType>> treePtr,
```

```
                         const ItemType& target, bool& isSuccessful) const;

   // Copies the tree rooted at treePtr and returns a pointer to the root of
the copy.
   auto copyTree(const std::shared_ptr<BinaryNode<ItemType>> oldTreeRootPtr)
const;

   // Recursively deletes all nodes from the tree.
   void destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);

   // Recursive traversal helper methods:
   void preorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>>
treePtr) const;
   void inorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>>
treePtr) const;
   void postorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>>
treePtr) const;

public:
   //------------------------------------------------------------
   // Constructor and Destructor Section.
   //------------------------------------------------------------
   BinaryNodeTree();
   BinaryNodeTree(const ItemType& rootItem);
   BinaryNodeTree(const ItemType& rootItem,
                  const std::shared_ptr<BinaryNodeTree<ItemType>> leftTreePtr,
                  const std::shared_ptr<BinaryNodeTree<ItemType>>
rightTreePtr);
   BinaryNodeTree(const std::shared_ptr<BinaryNodeTree<ItemType>>& tree);
   virtual ~BinaryNodeTree();

   //------------------------------------------------------------
   // Public BinaryTreeInterface Methods Section.
   //------------------------------------------------------------
   bool isEmpty() const;
   int getHeight() const;
   int getNumberOfNodes() const;

   ItemType getRootData() const throw(PrecondViolatedExcept);
   void setRootData(const ItemType& newData);

   bool add(const ItemType& newData); // Adds an item to the tree
   bool remove(const ItemType& data); // Removes specified item from the tree
   void clear();
```

```cpp
   ItemType getEntry(const ItemType& anEntry) const throw(NotFoundException);
   bool contains(const ItemType& anEntry) const;

   //-------------------------------------------------------------
   // Public Traversals Section.
   //-------------------------------------------------------------
   void preorderTraverse(void visit(ItemType&)) const;
   void inorderTraverse(void visit(ItemType&)) const;
   void postorderTraverse(void visit(ItemType&)) const;

   //-------------------------------------------------------------
   // Overloaded Operator Section.
   //-------------------------------------------------------------
   BinaryNodeTree& operator=(const BinaryNodeTree& rightHandSide);
}; // end BinaryNodeTree

#include "BinaryNodeTree.cpp"
#endif
```

File: BinaryNode.h

```cpp
portion
   std::shared_ptr<BinaryNode<ItemType>> leftChildPtr;  // Pointer to left
child
   std::shared_ptr<BinaryNode<ItemType>> rightChildPtr; // Pointer to right
child

public:
   BinaryNode();
   BinaryNode(const ItemType& anItem);
   BinaryNode(const ItemType& anItem,
              std::shared_ptr<BinaryNode<ItemType>> leftPtr,
              std::shared_ptr<BinaryNode<ItemType>> rightPtr);

   void setItem(const ItemType& anItem);
   ItemType getItem() const;

   bool isLeaf() const;

   auto getLeftChildPtr() const;
   auto getRightChildPtr() const;

   void setLeftChildPtr(std::shared_ptr<BinaryNode<ItemType>> leftPtr);
   void setRightChildPtr(std::shared_ptr<BinaryNode<ItemType>> rightPtr);
}; // end BinaryNode
```

```
#include "BinaryNode.cpp"
#endif
```

Provide implementations for the protected methods:

- `void inorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;` **[10/35]**
- `void postorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;` **[10/35]**
- `void preorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;` **[10/35]**
- `void destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);` **[5/35]**

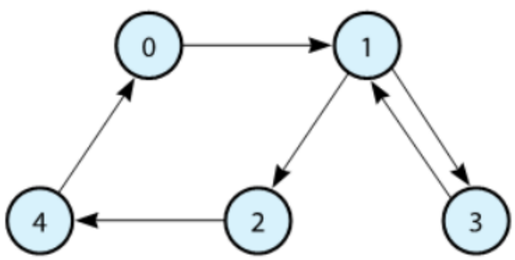
In your implementation of destroyTree utilize postorder traversal.

Question 3 [Topic: Dijkstra's Shortest Paths] [Percentage: 25%]: dijkstra

Q3.1 [18%/25%]: For a graph represented by its adjacency matrix `int graph[V][V]` and a starting vertex `int src` find the shortest paths for all vertices using the Dijkstra's algorithm. Function declaration follows:

```
void dijkstra(int graph[V][V], int src)
```

Q3.2 [2%/25%]: Write the adjacency matrix for the directed graph shown below



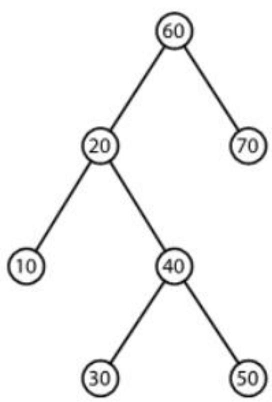
Q3.3 [5%/25%]: Use the depth-first strategy to traverse the graph in the Figure above, beginning with vertex 0. List the vertices in the order visited.

Question 4 [Topic: Comprehensive] [Percentage: 25%]: Answer to all the questions set below.

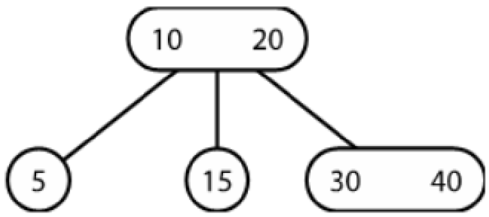
Q4.1 [10%/25%]: Trees-related questions

- Beginning with an empty binary search tree, what binary search tree is formed when you add the following letters in the order given? J, N, B, A, W, E, T

- Represent the following binary tree with an array



- What is the result of adding 3 and 4 to the 2-3 tree shown below?



- Why does a node in a red-black tree require less memory than a node in a 2-3-4 tree?

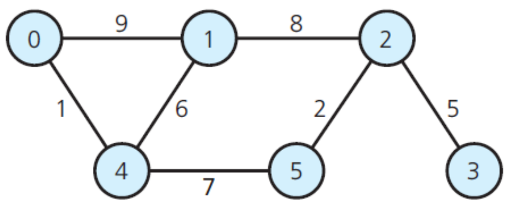
- Why can't a Red-Black Tree have a black child node with exactly one black child and no red child?

- What is the maximum height of a Red-Black Tree with 14 nodes?

Q4.2 [5%/25%]: Graphs-related questions

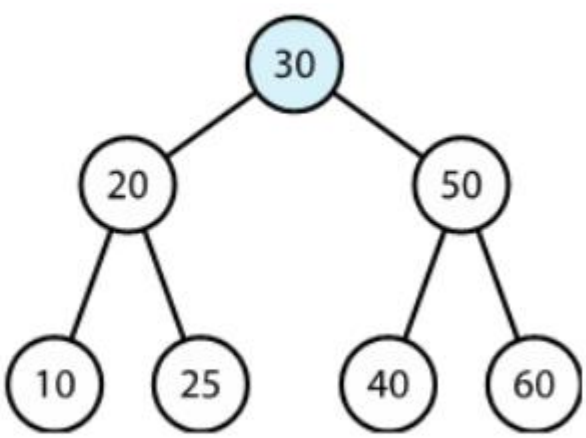
- Is it possible for a connected undirected graph with five vertices and four edges to contain a simple cycle? Explain your answer.

- Draw the BFS spanning tree whose root is vertex 0 for the graph shown below.

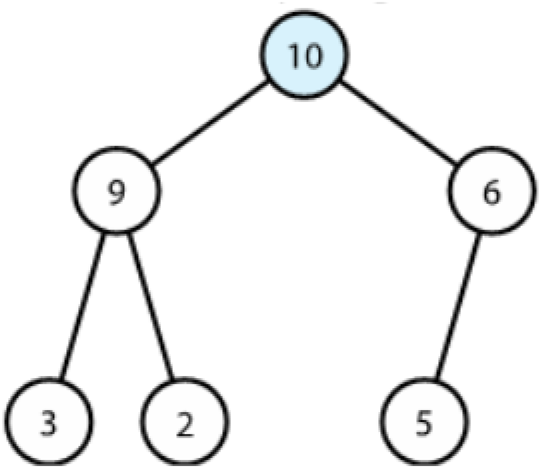


Q4.3 [5%/25%]: Heaps-related questions

- Is the full binary tree in the figure below, a semiheap?



- Consider the maxheap in the figure below, draw the heap after you add 12 and remove 12.



- Visualize the initially empty myHeap after the following sequence of operations
    - myHeap.add(2)
    - myHeap.add(3)
    - myHeap.add(4)
    - myHeap.add(1)
    - myHeap.add(9)
    - myHeap.remove()
    - myHeap.add(7)
    - myHeap.add(6)
    - myHeap.remove()
    - myHeap.add(5)

Q4.4 [5%/25%]: Dictionaries-related questions

- What is the Big-O function for addition, removal, retrieval and traversal of
  - unsorted array-based dictionary:
  - unsorted link-based dictionary:
  - sorted array-based dictionary:
  - sorted link-based dictionary:
  - BST-based dictionary: