Department of Computing
Hong Kong Polytechnic University

COMP 2011 Data Structures

Assignment Three (Total Marks: 100)

Submission Deadline: 23:55, 29 November, 2014

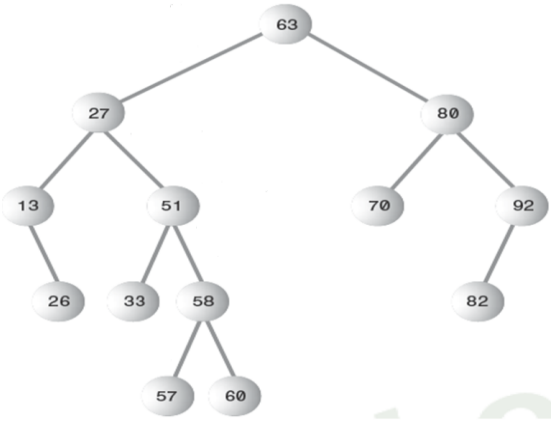
Note: Please submit your assignment by the deadline via Learn@PolyU.

Discussion with classmates is encouraged. But if the plagiarism is found, both you and the person who allows you to copy will get ZERO point.

1. Background

Breadth-first search **and** ***depth-first search*** **are two commonly-used strategies to traverse or search tree or graph data structures. We have learned various data structures including stacks, queues, and trees. In this assignment, you will combine them together to implement breadth-first search and depth-first search with the binary tree.**

In breadth-first search, given a tree, we start from the root at level 0, and then go down and traverse nodes level by level; at each level, we traverse nodes from left to right.



For example, given the following binary tree, with the breadth-first search, the traversal sequence is:

Level 0: 63
Level 1: 27, 80
Level 2: 13, 51, 70, 92
Level 3: 26, 33, 58, 82
Level 4: 57, 60

In depth-first search, given a tree, we start at the root and go as far as possible along each branch before backtracking. For example, given the above tree, with the depth-first search, the traversal sequence is:

13, 26, 27, 33, 51, 57, 58, 60, 63, 70, 80, 82, 90

This is the in-order traversal sequence we learned in the lecture.

In this assignment, we will practice how to implement breadth-first search and depth-first search with the binary tree we have learned. Breadth-first search can be implemented based on a queue (the basic idea: remove a node from the queue and traverse it, and then insert all of its children into the queue). So in Question 1, we will practice how to extend the linked-list-based queue so it can support tree nodes (each link can be connected to a tree node). Based on it, in Question 2, we will practice the breadth-first search by adding a new display

method to traverse all nodes accordingly.

Depth-first search can be implemented in a recursive manner (as we have learned in the lecture) or in an iterative manner based on the stack. In this assignment, to practice the knowledge we have learned about stacks, you are required to implement the depth-first search using an iterative manner based on the stack without recursion. So in Question 3, we will practice how to extend the linked-list-based stack so it can support tree nodes (each link can be connected to a tree node). In Question 4, we will practice the depth-first search by adding a new display method to traverse all nodes according. One implementation of the depth-first search is as follows:

Mark all nodes be "not traversed"
Push the root into the stack
While the stack is not empty
Pop up the node A from the stack;
If A has left child and the left child has not been traversed
Push A onto the stack
Push A's left child onto the stack
Continue
Else
Traverse A and mark A be "traversed"
If A has right child
Push A's right child onto the stack

Finally, in Question 5, you are required to implement delete() by replacing the deleted node with its predecessor.

2. Questions

1) (10 marks)

Revise the java program LinkQueue.java (attached) so it can support the tree node. The class for tree nodes (Class Node) and the application class have been given; you need to revise classes Link, FirstLastList, and LinkQueue as shown below:

```
class Node
{
    public int iData;              // data item (key)
    public double dData;           // data item
    public Node leftChild;         // this node's left child
    public Node rightChild;        // this node's right child
```

```
    public void displayNode()      // display ourself
    {
        System.out.print('{');
        System.out.print(iData);
        System.out.print(", ");
        System.out.print(dData);
        System.out.print("} ");
    }
 }  // end class Node

 class Link
{//Your revised code }
class FirstLastList
{//Your revised code}
class LinkQueue
{//Your revised code}

class LinkQueueTreeNodeApp
{
    public static void main(String[] args)
    {
        LinkQueue theQueue = new LinkQueue();

        for( int i=0; i<20; i++)
        {
            Node newNode = new Node();
            newNode.iData=i+1;
            newNode.dData=i+1.0;
            theQueue.insert(newNode);
        }

        theQueue.displayQueue();            // display queue
        theQueue.remove();                 // remove items
        theQueue.remove();
        theQueue.displayQueue();            // display queue
    }  // end main()
```

```
}  // end class LinkQueueApp
```

2) **(25 marks)**

Revise the binary search tree we have learned in the lecture by adding the following new method in class Tree:

public void breadthFirstDisplay()

In this method, display all nodes by using the breadth-first search strategy. To implement the breadth-first search, you can use the linked-list-based queue you implemented in Question 1. Revise class Node by adding

public int level;

so we can use "*level*" to denote the level of a node (the root node is at level 0), and revise *displayNode()* accordingly. In main() of class TreeApp, add the following code:

```
case 'b':
    theTree.breadthFirstDisplay();
    break;
```

so breadthFirstDisplay() can be called to display a tree based on the following format:

```
Level x: iData-xx, dData-yy
```

Here, x represents the level of a node in the tree; xx and yy are the iData and dData values of a node, respectively. (Attached please find the related java program – tree.java).

3) **(10 marks)**

Revise the java program LinkStack.java (attached) so it can support the tree node. The class for tree nodes (Class Node) and the application class have been given; you need to revise classes Link, LinkList, and LinkStack as shown below:

```
class Node
{
    public int iData;             // data item (key)
    public double dData;          // data item
    public Node leftChild;        // this node's left child
    public Node rightChild;       // this node's right child

    public void displayNode()     // display ourself
    {
```

```
        System.out.print('{');
        System.out.print(iData);
        System.out.print(", ");
        System.out.print(dData);
        System.out.print("} ");
    }
 }  // end class Node

 class Link
{//Your revised code }
class LinkList
{//Your revised code}
class LinkStack
{//Your revised code}

class LinkStackApp
{
   public static void main(String[] args)
   {
        LinkStack theStack = new LinkStack(); // make stack

        for( int i=0; i<20; i++)
        {
            Node newNode = new Node();
            newNode.iData=i+1;
            newNode.dData=i+1.0;
            theStack.push(newNode);
            theStack.displayStack();            // display stack
        }
        theStack.pop();                     // pop items
        theStack.pop();
        theStack.displayStack();            // display stack
    }  // end main()
 }  // end class LinkStackApp
```

4) (25 marks)

Revise the binary search tree we have learned in the lecture by adding the following new method in class Tree:

public void depthFirstStackDisplay()

In this method, display all nodes by using the depth-first search strategy with an iterative manner based on a stack (you can use the linked-list-based stack you implemented in Question 3). Revise class Node by adding

public boolean flag;

so we can use "*flag*" to denote whether or not this node has been traversed, which you may need in your implementation. In main() of class TreeApp, add the following code:

```
case 'p':
    System.out.print("Print the keys in ascending order: ");
    theTree.depthFirstStackDisplay();
    break;
```

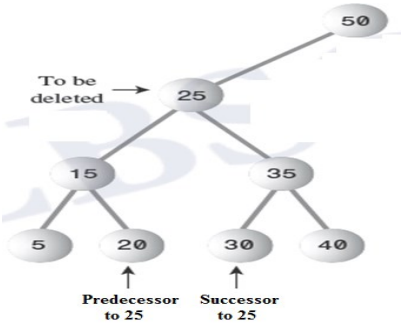
so depthFirstStackDisplay() can be called to display all key values in ascending order.

5) (30 marks)

Revise the binary search tree we have learned in the lecture by adding the following new method in class Tree:

public boolean deleteRPredecessor(int key)

In this method, a node whose *iData* equals *key* will be deleted from the tree, and different from what we have learned (using the inorder successor as the replacement node), the deleted node will be replaced with its inorder predecessor. If such a node is found and deleted, return true; otherwise, return false. The tree may be empty.



The predecessor of a node with a key k in a binary search tree is the node with the largest key that belongs to the tree and its key value is strictly less than k. As shown in the example, the predecessor of the node with the key 25 is the node with the key 20.

What to submit – A zip file includes:

(1) The source code; (2) A report (with screen snapshots) to show the output of each program.