EE 2331 Data Structures and Algorithms, Semester B, 2009/10

Tutorial 12: Binary Search Trees

Week 12 (15th April, 2010)

The tasks of tutorial exercises are divided into three levels. Level 1 is the basic tasks. You should have enough knowledge to complete them after attending the lecture. Level 2 is the advanced tasks. You should able to tackle them after revision. Level 3 is the challenge tasks which may be out of the syllabus and is optional to answer. I expect you to complete at least task A in the tutorial.

Outcomes of this tutorial

1. Able to insert a node into binary search trees using iteration
2. Able to delete a node from binary search trees using iteration
3. Able to delete a node from binary search trees using recursion
4. Able to test the completeness of user defined functions

Binary search tree (BST) is a special kind of binary tree. For an arbitrary node *p*, the values of all nodes in its left subtree are smaller than the value of node *p*, and at the same time the value of node *p* is smaller than the values of all nodes in its right subtree. The following figure is an example of BST.

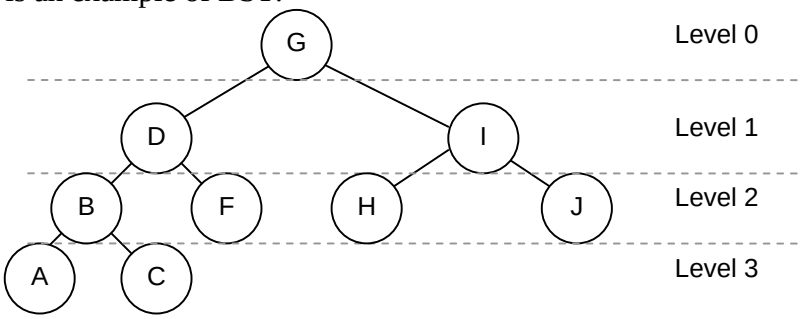

Figure 1. The letters in the circle represent the values of the tree node.

In this tutorial, the provided program will read the implicit array representation of a binary tree from a file and build the corresponding linked list tree. The nodes should be compared using their ASCII value (i.e. 'A' < 'B', 'Z' < 'a'). You are going to complete the insert and delete functions.

The implicit array representation is stored in a text file with the following format:

Row	Content	Remark
1st	$C_1\ C_2\ C_3 \ldots C_n$ <*new line*>	The implicit array representation

The structure of BST is defined as follows:

```
typedef char TreeElement;
typedef struct _treenode {
        TreeElement data;
        struct _treenode *left, *right;
} TreeNode;

typedef struct _tree {
        struct _treenode *root;
} Tree;
```

Task A (Level 2): Insertion of BST

Design the iterative function **int insert(Tree *t, TreeElement e)** to insert a node with value equals to *e* into the tree. The function accepts a pointer to tree (*t*), and returns 1 to indicate the completion of the function. **It will return 0 if the node already exists**.

Expected Output:

```
Enter the file name for testing: test1.txt

The array tree is: [GDIBFHJAC]
The linked list tree is:
    J
  I
    H
G
    F
  D
      C
    B
      A
Enter your action ( 1) Insert, 2) Delete, 3) Delete (recursion), 4) Quit ): 1
Enter the node value: Y
      Y
    J
  I
    H
G
    F
  D
      C
    B
      A
```

Discussion: Is it possible to insert a duplicated node into the BST? Why or why not? There are many special cases. Can you have to handle all of them? For example, test2.txt is an empty tree. Try to insert as many different nodes as possible to test your function.

Task B (Level 2): Deletion of BST

Design the iterative function **int delete(Tree *t, TreeElement e)** to delete the node with value equals to *e* from the tree. The function accepts a pointer to tree (*t*), and returns 1 to indicate the completion of the function. **It will return 0 if the node does not exist**.

Please refer to p135 to p145 of lecture notes 09 for the detail description of the deletion algorithm.

Expected Output:

```
Enter the file name for testing: test1.txt

The array tree is: [GDIBFHJAC]
The linked list tree is:
    J
  I
    H
G
    F
  D
      C
    B
      A
Enter your action ( 1) Insert, 2) Delete, 3) Delete (recursion), 4) Quit ): 2
Enter the node value: G
    J
  I
H
    F
  D
      C
    B
      A
```

Discussion: There are many special cases in this function. Can you handle all of them? (e.g. the node to be deleted may be an internal node or root node, there may be no element in the right subtree of the node to be deleted, etc). Try to delete as many different nodes as possible to test your function.

Task C (Level 3): Deletion of BST Using Recursion

Can you implement the recursive version of the delete function? Please note that **int delete_recur(TreeNode **p, TreeElement e)** need to use a call-by-reference argument *p* which is storing the address of a tree node. The code of recursive version is generally shorter.