Data Structure and Algorithm

Homework #3

Due: 2:20pm, Tuesday, April 9, 2013

TA email: dsa1@csie.ntu.edu.tw

=== Homework submission instructions ===

- For Problem 1, submit your source code, a Makefile to compile the source, and a brief documentation to the SVN server (katrina.csie.ntu.edu.tw). You should create a new folder "`hw3`" and put these three files in it.
- The filenames of the source code, Makefile, and the documentation file should be "`main.c`", "`Makefile`", and "`report.txt`", respectively; you will get some penalties in your grade if your submission does not follow the naming rule. The documentation file should be in plain text format (`.txt` file). In the documentation file you should explain how your code works, and anything you would like to convey to the TAs.
- For Problem 2 through 5, submit the answers via the SVN server (electronic copy) or to the TA at the beginning of class on the due date (hard copy).
- Except the programming assignment, each student may only choose to submit the homework in only one way; either all in hard copies or all via SVN. If you submit your homework partially in one way and partially in the other way, you might only get the score of the part submitted as hard copies or the part submitted via SVN (the part that the grading TA chooses).
- If you choose to submit the answers of the writing problems through SVN, please combine the answers of all writing problems into only ONE file in the pdf format, with the file name in the format of "`hw3_[student ID].pdf`" (e.g. "`hw3_b01902888.pdf`"); otherwise, you might only get the score of one of the files (the one that the grading TA chooses).
- Discussions with others are encouraged. However, you should write down your solutions by your own words. In addition, for each problem you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution to that problem.
- **NO LATE SUBMISSION IS ALLOWED** for the homework submission in hard copies - no score will be given for the part that is submitted after the deadline. For submissions via SVN (including the programming assignment and electronic copies of the writing problems), up to one day of delay is allowed; however, the score of the part that is submitted after the deadline will get some penalties according to the following rule (the time will be in seconds):

$$\text{LATE SCORE} = \text{ORIGINAL SCORE} \times (1 - DelayTime/86400)$$

Problem 1. k-dimensional Tree (30%)

In computer science, a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are useful data structures for several applications. A common application of k-d tree is to find the nearest point to a specific point (See Figure 1 for an example). In addition, the data structure supports point insertion or deletion operations. In this homework, you are asked to implement k-d tree for the $k = 1$ case, and the data structure should support the operations of querying the nearest point, point insertion, and point deletion.

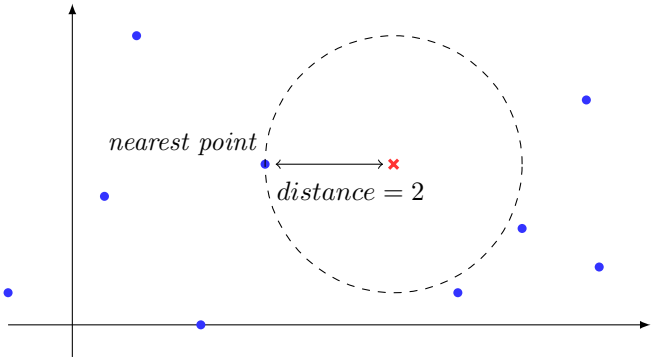

Figure 1: Nearest point searching in 2-dimensional space. The dots are existing points on the plane and the cross is the point to which we are trying to find the nearest point. Note that the point to which we are asked to find the nearest point does not need to be one of existing points on the plane.

A 1-d tree (k-d tree with $k = 1$) is just a simple binary search tree. A point in the 1-d space, i.e., on a number line, corresponds to a node in the binary search tree. The insertion (deletion) of points to (from) current set of points corresponds to inserting (deleting) nodes to (from) the current binary search tree. As an example, Figure 2 shows a few points on the number line and the corresponding 1-d tree.

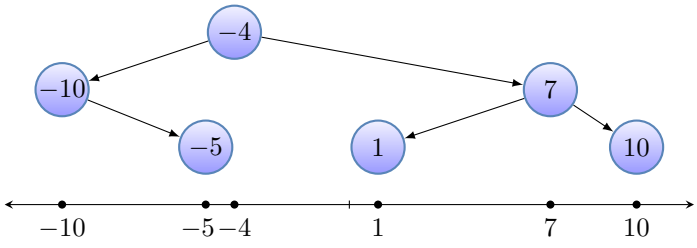

Figure 2: Points on the number line and the corresponding 1-d tree, which is also a binary search tree.

Finding the nearest point out of a set of existing points P to a specific point is similar to searching for a node, X, in the binary search tree. The only difference is that, in the process

of finding the nearest point, if X in the corresponding binary search tree cannot be found, the "closest node" found in the tree (corresponding to the closest point) should be reported. In other words, if X is not found, we are looking for a node Y such that the absolute difference between X and Y is the minimum, i.e., $Y = \arg\min_y |X - y|, \forall y \in P$. Since the tree is a binary search tree, we only have to check the nodes that we have gone through in the process of finding X. See Figure 3 for an example.

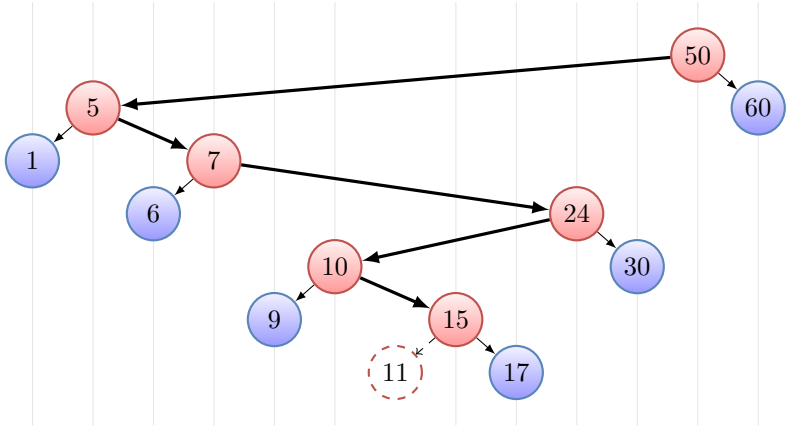

Figure 3: This figure illustrates how to search for the closest node in a 1-d tree. The nodes that we have visited when we search for node "11" are $50, 5, 7, 24, 10, 15$. The closest one to "11" is "10", and it is the nearest point since its absolute difference to 11 is $= |11 - 10| = 1$ and is the minimum.

Your program will be tested with the test data in the following format. The first line in the input is a positive integer N, which represents the number of operations in the input. Each of the following N lines specifies one of the 3 types of operations below. For convenience, we call a point with coordinate x on the number line as "point x". Assume that the set of points P is initially empty. All numbers in the input are integers.

- "`insert x`": Add point x to the current set P. It is guaranteed that point x will not be in P before the insertion.
- "`delete x`": Delete point x from current set P. It is guaranteed that point x is in P before the deletion.
- "`query x`": Find the nearest point in P to point x and output its coordinate. If there are multiple points in P with the same minimum distance to x, output all of them in ascending order. It is guaranteed that there will be at least one point in P before the query.

You may assume that the test data is generated randomly. That is, your binary search tree will not become too unbalanced even if you do all insertions from the test data sequentially.

Input example

```
7
insert 4
insert 10
insert 5
query 7
delete 5
query 7
query 10
```

Output example

```
5
4 10
10
```

When performing the first query, the set of points is $\{4, 5, 10\}$, in which 5 is the nearest one to 7. When performing the second query, the set of points is $\{4, 10\}$, in which both 4 and 10 are the nearest points to 7. When performing the third query, since 10 is in the current set, the nearest point is itself.

Please use "main" as the filename of your main program.

Problem 2. A Piece of Graph (20%)

Let $G = (V, E)$ be an undirected graph. Prove the following statements:

2.1. (5%) If we sum up the degrees of all vertices in G, the answer must be an even number.

2.2. (5%) If we can find two vertices $u, v \in V$ such that there are two different paths from u to v, then G must contain a cycle.

2.3. (5%) If for any two vertices $u, v \in V$ there is exactly one path from u to v, then G is a tree. (Hint: recall that a tree is a connected graph with no cycle)

2.4. (5%) If G is a connected graph with n vertices, then G must have at least $n - 1$ edges.

Problem 3. Tree Traversal Orders (15%)

3.1. (5%) A **max binary tree** is a binary tree in which the key value of each node is no smaller than the key values of its children (if it has any). Given the **in-order** traversal sequence of a **max binary tree**, please derive and draw the tree.

in-order: $7, 31, 32, 9, 15, 3, 38, 21, 23, 13, 37, 25, 18, 30, 34, 26$

3.2. (5%) Given the **pre-order** traversal sequence of a **binary search tree**, please derive and draw the tree.

pre-order: $16, 5, 1, 9, 8, 10, 11, 36, 29, 24, 23, 27, 25, 32, 38, 40$

3.3. (5%) Convert the **binary search tree** in the previous problem into a **in-order threaded binary tree** and draw the tree. Please specify the type of each link in the tree, i.e., whether it is a thread or an ordinary tree edge. You also have to draw the dummy node and its links in the tree.

Problem 4. Merge Binary Search Trees (20%)

After learning the properties and the operations of binary search tree (BST), let's try to work on a more advanced problem. You are given two complete BST, i.e., a BST that is also a complete binary tree, and are asked to merge them into one BST.

Assume that $Tree_1$ has N nodes and $Tree_2$ has M nodes, and all nodes in both trees have distinct key values. The structure of a node is shown in the following code.

```
struct node{
    int value;
    struct node* left;
    struct node* right;
};
typedef struct node Node;
```

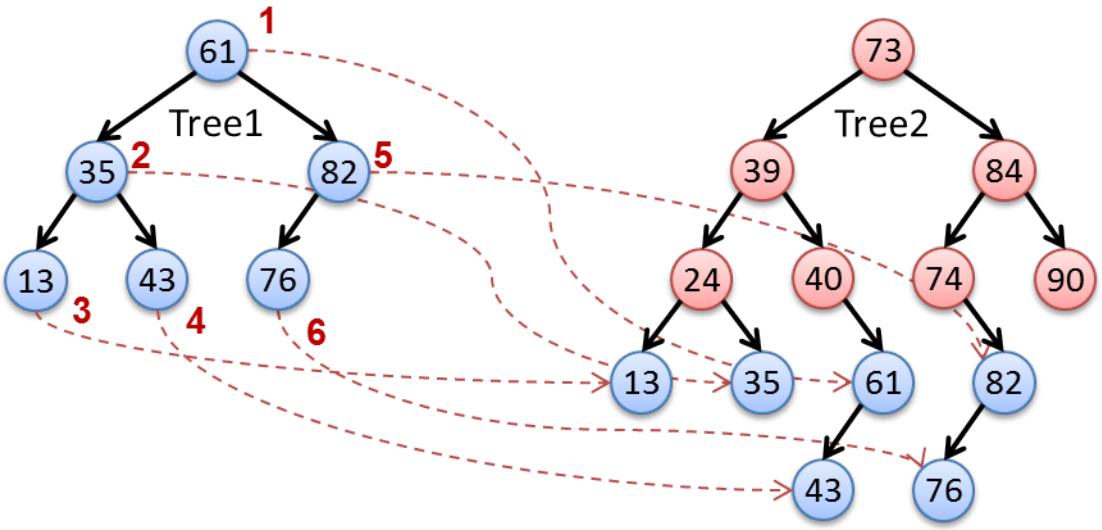

Figure 4: Insertion of nodes in $Tree_1$ in pre-order to $Tree_2$

4.1. (5%) A simple algorithm is given in the following, implementing the procedures shown in Figure 4. The algorithm performs **pre-order** traversal on $Tree_1$ and, in each time it visits a node, it inserts the node to $Tree_2$ using the ordinary BST insertion operation. Please give the time complexity of this algorithm in terms of N and M using the big-Oh notation and make sure this bound is as tight as possible.

```
Node* insert(Node* tree2, int value){
    if(tree2 == NULL){
        Node *newNode = (Node*)malloc(sizeof(Node));
        newNode->value = value;
        return newNode;
    }
    if(value < tree2->value)
        tree2->left = insert(tree2->left, value);
    else tree2->right = insert(tree2->right, value);
    return tree2;
}

Node* mergeTree1(Node* tree1, Node* tree2){
    if(tree1 == NULL) return tree2;
    tree2 = insert(tree2, tree1->value);
    mergeTree1(tree1->left, tree2);
    mergeTree1(tree1->right, tree2);
    return tree2;
}
```

4.2. (15%) We are not satisfied the previous algorithm, since it is not only inefficient in some cases but also prone to the unbalanced problem. Thus, we will propose an algorithm that includes the following steps:

(a) Convert the two BSTs into two sorted arrays with in-order traversals.

(b) Merge the two sorted array into one.

(c) Convert the merged sorted array into a BST which has the minimum height.

(Hint: One solution is to take the middle element in the sorted array as the root of the final BST, then treat the elements on the left as the nodes in the left subtree of the root and the elements on the right as the right subtree of the root. Perform the same operation to all subtrees recursively.)

The implementation of this algorithm in C is shown in the function *mergeTree*2 below. But all inner functions are unfinished. Please copy and complete the skeleton code of inner functions and make sure that *mergeTree*2 runs in $O(N + M)$ time overall. Show the running time of each of the functions you implemented and from there show that the overall running time is indeed $O(N + M)$. The skeleton of the inner functions is shown in the next page. You can treat N and M as constant integers and use them in your code directly.

```
1  Node* mergeTree2(Node* tree1, Node* tree2){
2      if(tree1 == NULL) return tree2;
3      if(tree2 == NULL) return tree1;
4
5      int *array1 = (int*)malloc(N*sizeof(int));
6      int *array2 = (int*)malloc(M*sizeof(int));
7      int *sortedArray = (int*)malloc((N+M)*sizeof(int));
8
9      array1 = treeToArray(tree1, array1);
10     array2 = treeToArray(tree2, array2);
11     sortedArray = mergeArray(array1, array2, sortedArray);
12     Node *tree = arrayToTree(sortedArray, N+M);
13
14     free(array1);
15     free(array2);
16     free(sortedArray);
17
18     return tree;
19 }
```

```c
/*
 * int* treeToArray(Node*, int*);
 * Give a root of a BST and an initialized array,
 * return the sorted array which is converted from BST.
 */
int* treeToArray(Node* tree, int* array){
    if(tree == NULL) return array;
    int *head = array;
    //TODO add some statements here

    return head;
}

/*
 * int* mergeArray(int*, int*, int*);
 * Give two sorted arrays and an initialized array,
 * return the sorted array which is merged from them.
 */
int* mergeArray(int* array1, int* array2, int* array){
    int *sortedArray = array;
    //TODO add some statements here

    return sortedArray;
}

/*
 * int* arrayToTree(int*, int);
 * Give a sorted array and its size,
 * return the root of BST which is converted from array.
 */
Node* arrayToTree(int* array, int size){
    if(size == 0) return NULL;
    Node* node = (Node*)malloc(sizeof(Node));
    //TODO add some statements here

    return node;
}
```

Problem 5. Linked-Lists and Trees (15%)

After over 20 hours of coding, Michael has just finished his DSA homework in time, in which he implemented a sorted descending circular doubly linked-list as the data structure to perform some complicated operations that are specified in the homework description by the Professor.

It is now one hour before the deadline. He decides to go ahead and submit the homework via SVN. However, when reading the submission guidelines again, he finds that there is a small line of words at the bottom:

Please use a heap or a BST to implement the operations in this homework.

He is shocked and has no idea how to finish this homework in time. There is no time to re-write the program from scratch. The only feasible solution he can come up with is to find the functions to convert linked-lists to heaps and BSTs. Luckily, the C structure defined for a doubly linked-list node and a tree node is exactly the same - the node of either data structure includes a data portion and two pointers. Nodes of doubly linked-lists have the previous and next pointers while the tree nodes have left-child and right-child pointers.

The prototype of function and the C structure of the node is shown as follows:

```
1 struct node {
2     int index;
3     struct node* prev;  //Should be the left child ptr in tree
4     struct node* next;  //Should be the right child ptr in tree
5 };
6 struct node* link_lst_to_heap(struct node* head);
```

5.1. (5%) Linked-lists to heaps (using the linked tree representation, not the array representation shown in the lecture)

Please design an algorithm to covert a circular doubly linked-list to a heap in $O(N)$ time. You are allowed to use a few extra pointers but not allowed to use extra nodes, i.e., no malloc operations.

5.2. (10%) Linked-lists to BSTs

(a) Please design an algorithm to convert a circular doubly linked-list to a binary search tree in $O(N)$ time. You are allowed to use a few extra pointers but not allowed to use extra nodes, i.e., no malloc operations. (Hint: the BST can be extremely unbalanced, but it is still a BST. :-P)

(b) In certain cases, the BST produced from the algorithm in the previous question may have very poor query performance. We would like to convert it to a balanced BST, where the depths of the two subtrees of every node differ by 1 or less. More definitions on wiki can be found:

http://en.wikipedia.org/wiki/Binary_tree

In this case, the depth of the tree would be minimized. You are asked to design an algorithm to recursively operate on the descending circular doubly linked-list and convert it to a balanced BST in $O(N \log N)$ time. Same as before, you are allowed to use a few extra pointers but not allowed to use extra nodes, i.e., no malloc operations.

5.3. (bonus 5%) Please take a look at the description of a data structure called skip-list,

http://en.wikipedia.org/wiki/Skip_list

and explained how it works and what is the time complexity of the operation of building a balanced binary search tree from a circular doubly linked-skip-list.