Lab Assignment #5: Binary Search Trees

1. Objectives

The objectives of this assignment are to provide you with more practice on the use of the various C++ concepts/constructs introduced in the lectures so far, including classes, dynamic memory allocation, pointers, I/O, and dynamic data structures. This will be done in the context of implementing a binary search tree class. You will also learn how to use a tool called `valgrind` to ensure that you code is free of memory allocation and de-allocation problems.

2. Problem Statement

In this assignment, you will implement a simple database to store and retrieve data. You are creating a simple "domain name server" – a database that can rapidly look up string names, such as `www.google.com`, and return a 32-bit integer that is the Internet Protocol (IP) address corresponding to that "domain name". This is the function performed by the domain name servers that allow you to type in easy to remember website names and have your web browser automatically go to the correct IP address (computer) to retrieve the web data.

You will implement three classes: `DBentry, TreeNode and TreeDB`. The `DBentry` class will be used to create objects that hold an entry of the database. Each entry has a key (a string, corresponding to the name of the internet domain) and some data (a 32-bit integer that gives the IP address and a boolean flag that indicates whether the computer at that IP address is active or not). The `TreeDB` and `TreeNode` classes will be used to create a binary search tree database of `DBentry` objects. `TreeDB` represents the entire binary tree, while each `TreeNode` represents a single node in the tree.

In a real domain names server, we would have to rapidly search billions of domain names (strings) as we looked for the IP address corresponding to a name. The fact that a binary search tree can find entries in large datasets very efficiently, and can also add new entries efficiently, makes it a very appropriate data structure for this application.

2. Command and Output Specification

Your database will be exercised by a simple parser. Whenever the parser expects the user to enter a command, it should prompt the user for input by outputting a greater than sign followed by a space:

> *user input would appear here*

The parser reads commands from `cin`, calls the appropriate `TreeDB` and `DBentry` functions, and sends the appropriate output to `cout`. Each command consists of an operation, followed by zero or more arguments. The command and the arguments are separated by white spaces, and will always appear on the same line. You parser should process commands until the end-of-file (`eof`) is encountered. *You may assume that all the input is syntactically correct* – your program will not be tested with invalid commands, the wrong number of arguments, or misspelt

arguments. *The only error messages you need to generate are those listed below*. The commands and their parameters are:

`insert` *`name IPaddress status`*. This command creates a new entry with *`name`* (a string) as the key and *`IPaddress`* (a non-negative integer) and *status* (**`active`** or **`inactive`**) as specified. You may assume that the name, IPaddress, and active arguments are separated by spaces and that there are no spaces within an argument. The new entry is then inserted into the database. If there already exists an entry with the same key, the error message "`Error: entry already exists`" is printed to `cout`. Otherwise "`Success`" is printed.

`find` *`name`*. This command finds the entry with the key *`name`* in the database, and prints its contents to `cout` as `name : IPaddress : status`. Name and IPaddress are printed as a string and an unsigned integer, respectively, while status is either `active` or `inactive`. Between values a space, colon and space should be printed.
If no such entry exists, the error message "`Error: entry does not exist`" is printed to `cout`.

`remove` *`name`*. This command deletes the entry with the key *`name`* from the database. If no such entry exists, the error message "`Error: entry does not exist`" is printed to `cout`. Otherwise "`Success`" is printed.

`printall`. This command prints all the entries in the database, sorted in ascending order of keys, one entry per line.

`printprobes` *`name`*. This command finds the entry with the key *`name`* in the database. If no such entry exists, the error message "`Error: entry does not exist`" is printed to `cout`. Otherwise, the number of probes (DBentries examined during the tree search) is printed to `cout`.

`removeall`. This command deletes all the entries in the database, returning it to the empty state. When done, "`Success`" is printed.

`countactive`. This command counts the number of entries in the database that are active and prints this count to cout.

`updatestatus` *`name status`*. This command updates the status of the entry with the given name; status must be either **`active`** or **`inactive`**. If no entry with *name* exists, the error message "`Error: entry does not exist`" is printed to `cout`. Otherwise "`Success`" is printed.

The following is an example of commands and their outputs. The example assumes an empty database at the beginning.

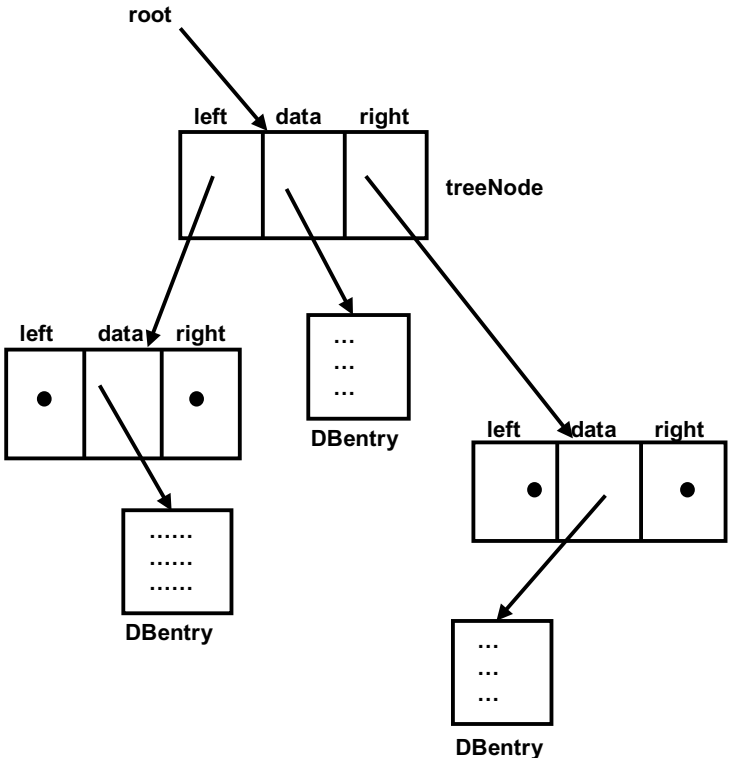
```
> insert www.google.com 283983 active
Success
> insert www.yahoo.com 191333 active
Success
> insert www.eecg.utoronto.ca 179333 inactive
Success
> insert www.altera.com 283299 active
Success
> find www.google.com
```

```
www.google.com : 283983 : active
> updatestatus www.google.com inactive
Success
> printall
www.altera.com : 283299 : active
www.eecg.utoronto.ca : 179333 : inactive
www.google.com : 283983 : inactive
www.yahoo.com : 191333 : active
> printprobes www.altera.com
3
> remove www.eecg.utoronto.ca
Success
> printprobes www.altera.com
2
> countactive
2
> printall
www.altera.com : 283299 : active
www.google.com : 283983 : inactive
www.yahoo.com : 191333 : active
> removeall
Success
> find www.google.com
Error: entry does not exist
> printall
> (Ctrl+D)
```

3. The Tree-based Database

The tree-based implementation assumes that the database is made of a BST, with data being pointers to `DBentry`s, as shown below.

The `left` and `right` pointers fields in each `TreeNode` are pointers to the left and right subtrees respectively. The BST is "ordered" by keys. That is, the `name` field of each `DBentry` is the key of the BST.



4 Code Structure and Development Procedure

Create a sub-directory called `lab5` in your `ece244` directory, using the `mkdir` command. Make it your working directory. You are given 3 .h files to get you started in the lab; download them from the portal.

4.1 The `DBentry` Class

The `DBentry` class has three fields: one to represent the key and two data fields. It also has the several methods to set and access this private data, as shown in the header file below.

```
class DBentry {
private:
        string name;
        unsigned int IPaddress;
        bool active;

public:
        // constructors
        DBentry();
        DBentry (string _name, unsigned int _IPaddress,
                 bool _active);

        // the destructor
        ~DBentry();

        // sets the domain name, which we will use as a key.
        void setName(string _name);

        // sets the IPaddress data.
        void setIPaddress(unsigned int _IPaddress);

        // sets whether or not this entry is active.
```

```
        void setActive (bool _active);

        // returns the name.
        string getName() const;

        // returns the IPaddress data.
        unsigned int getIPaddress() const;

        // returns whether or not this entry is active.
        bool getActive() const;

        // prints the entry in the format
        // name : IPaddress : active  followed by newline
        // active is printed as a string (active or inactive)
        friend ostream& operator<< (ostream& out,
                                    const DBentry& rhs);
};
```

Write the implementation of this class in a file called `DBentry.cpp`. We suggest you implement this class first, and then test it with a simple test harness – a `main ()` function that creates, modifies and prints DBentries. This is a good practice in developing larger programs: build components one at a time, and test each one before building a larger component that relies on it.

For example, if you write your test harness in a file called testDBentry.cpp, you could then test `DBentry` using:

```
g++ -g -Wall testDBentry.cpp DBentry.cpp -o testDBentry
./testDBentry
```

4.2 The `TreeNode` Class

This class stores individual nodes of the tree. Each node stores left and right pointers to children of this node, as well as a pointer to a `DBentry` that stores the key and data associated with a node. Its header file is given below.

```
#include "DBentry.h"

class TreeNode {
private:
    DBentry* entryPtr;
    TreeNode* left;
    TreeNode* right;

public:
    // A useful constructor
    TreeNode(DBentry* _entryPtr);

    // the destructor
    ~TreeNode();

    // sets the left child of the TreeNode.
    void setLeft(TreeNode* newLeft);

    // sets the right child of the TreeNode
    void setRight(TreeNode* newRight);

    // gets the left child of the TreeNode.
    TreeNode* getLeft() const;

    // gets the right child of the TreeNode
```

```
    TreeNode* getRight() const;

    // returns a pointer to the DBentry the TreeNode contains.
    DBentry* getEntry() const;
};
```

Write the implementation of this class in a file called `TreeNode.cpp`. Once again we recommend that you implement `TreeNode` completely and test it with a test harness (another .cpp file that implements a `main()` function and possibly helper subroutines that create, manipulate and print `TreeNodes`) before continuing on to the next part of the lab.

4.3 The `TreeDB` Class

This class represents the overall database, and has methods to perform various useful functions on the database. An initial header file is given below. You will have to add new private functions to this header file to complete the lab.

```
#include "TreeNode.h"
#include "DBentry.h"

class TreeDB {

private:
   TreeNode* root;
   int probesCount;
   // You will need to add additional private functions

public:
   // the default constructor, creates an empty database.
   TreeDB();

   // the destructor, deletes all the entries in the database.
   ~TreeDB();

   // inserts the entry pointed to by newEntry into the database.
   // If an entry with the same key as newEntry's exists
   // in the database, it returns false. Otherwise, it returns true.
   bool insert(DBentry* newEntry);

   // searches the database for an entry with a key equal to name.
   // If the entry is found, a pointer to it is returned.
   // If the entry is not found, the NULL pointer is returned.
   // Also sets probesCount
   DBentry* find(string name);

   // deletes the entry with the specified name (key) from the database.
   // If the entry was indeed in the database, it returns true.
   // Returns false otherwise.
   // See Section 6 for the *required* removal method so you match
   // exercise's output.
   bool remove(string name);

   // deletes all the entries in the database.
   void clear();

   // prints the number of probes stored in probesCount
   void printProbes() const;

   // computes and prints out the total number of active entries
   // in the database (i.e. entries with active==true).
   void countActive () const;

   // Prints the entire tree, in ascending order of key/name
   // The entries are printed one per line, using the
   // operator<< for DBentry.
   friend ostream& operator<< (ostream& out, const TreeDB& rhs);
};

// You *may* choose to implement the function below to help print the
// tree.  You do not have to implement this function if you do not wish to.
ostream& operator<< (ostream& out, TreeNode* rhs);
```

4.4 Main.cpp

Implement your parser in `Main.cpp`. You will call public functions in `TreeDB` and `DBentry` to implement the commands after you have parsed them.

4.5 Coding Requirements

1. No global variables are to be used.
2. All data members of `DBentry`, `TreeNode` and `TreeDB` must be of private type.
3. You should minimize the number of additional public functions you create. No additional public functions in `DBentry`, `TreeNode` and `TreeDB` are required to complete the lab.
4. Your program should free all its memory; we will check for memory leaks using valgrind, and deduct marks if your program leaks memory.
5. As the purpose of this lab is to implement a binary search tree "from scratch", you are not allowed use the standard template library.

5. Deliverables

You must submit the following files:

- Main.cpp
- DBentry.h
- DBentry.cpp
- TreeNode.h
- TreeNode.cpp
- TreeDB.h
- TreeDB.cpp

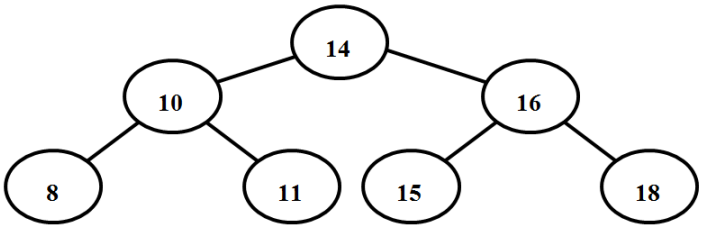
Submit your lab using

```
~ece244i/public/submit 5
```

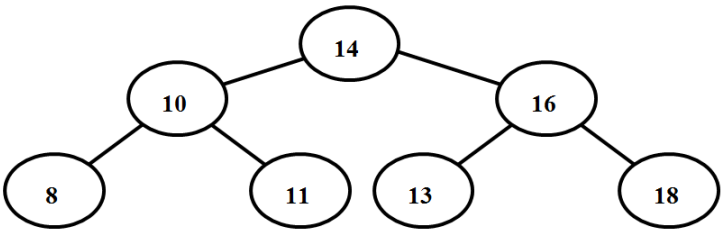
6. Background

A binary search tree (BST) is a binary tree for which: (1) each node has a key that is unique; (2) the key of each node is greater than the keys of all nodes in its left subtree; and (3) the key of each node is smaller than the keys of all nodes in its right subtree. Thus, a BST is an ordered binary tree in which left is smaller than node, which in turn is smaller than right. Since this order property holds for every node in the tree, the left and right subtrees of any node are also binary search trees. That is, the order property holds recursively.

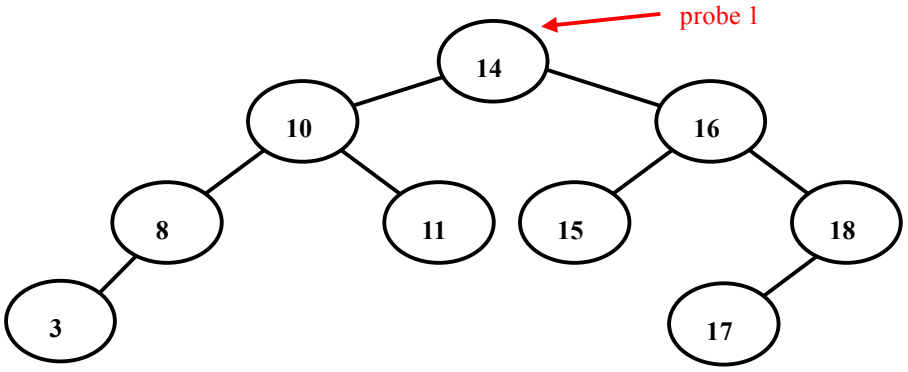
For example, the binary tree below is a BST, in which the keys of the nodes are integers. The key of the root is larger than any of the keys in its left subtree and is smaller than any of the keys in its right subtree. Each of the left and right subtrees is also BST. The left subtree is rooted at a node whose key 10 is larger than all the keys in its own left subtree (8), but smaller than all the keys in its own right subtree (11).



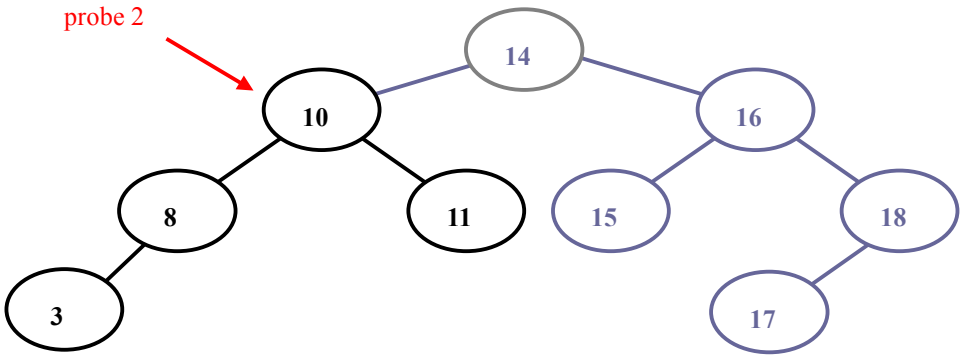
On the other hand, the binary tree below is not a BST (why? Hint: hotel elevators often skip this floor).



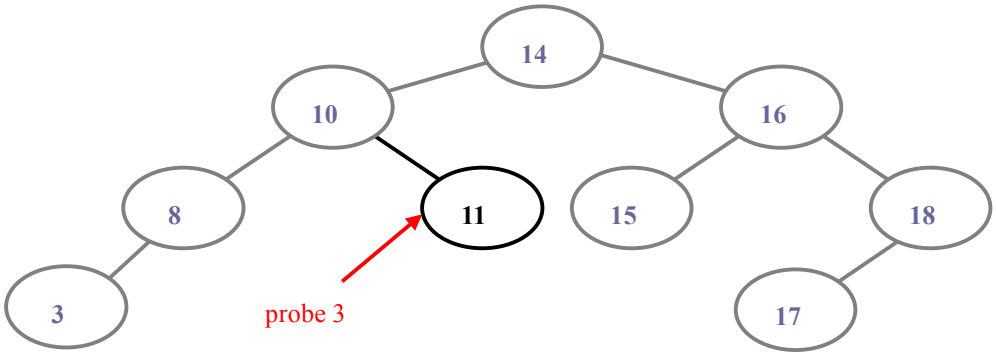
The order property of a BST makes it easy and efficient to search for a node in the tree, with a given search key. Given a BST and a search key ***k***, the goal is to determine if there exists a node in the BST that has a key ***k*** (remember, because the keys are unique, there can be at most one such node). One simply compares the search key ***k*** to the key of the root of the BST. If they are equal, then the root is the node searched for. If ***k*** is less than the key of the root, then if a node exists with a matching key, the node must be in the left subtree. One then recursively repeats the search in the left subtree. Otherwise, ***k*** is greater than the key of the root, and thus, if a node exists with a matching key, it must be in the right subtree. Thus, one recursively repeats the search in the right subtree. Each comparison of the search key with a key of a node in the tree is called a ***probe***. The process of searching for a node with the key 11 in an example BST is illustrated below.



Compare search key 11 to 14. Since 11 < 14, look in the left subtree.



Compare search key 11 to 10. Since 11 > 10, look in the right subtree.

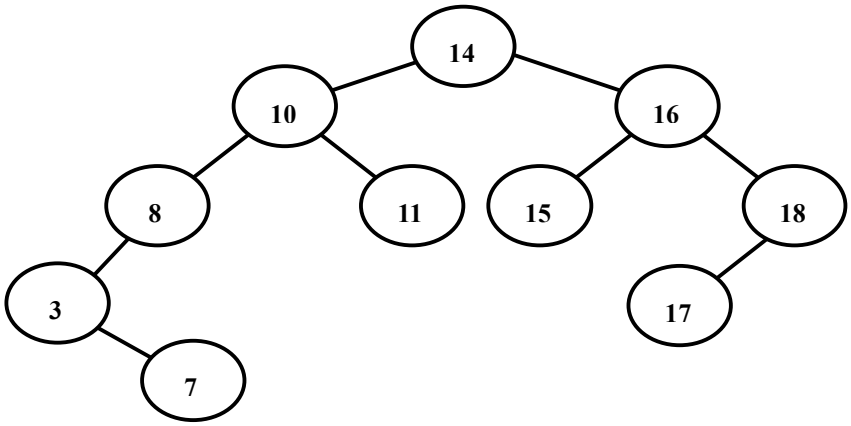


Compare search key 11 to 11. Since 11 = 11, the node with key equal to the search key is found.

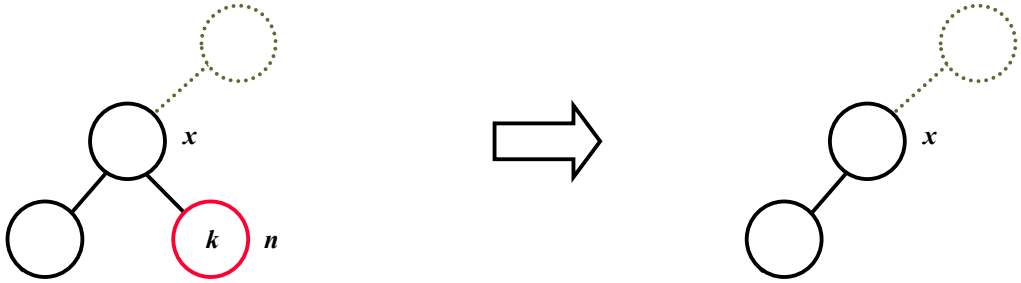
Had the search key been 7, then four probes would have been needed. The first compares 7 to 14, the second compares 7 to 10, the third compares 7 to 8, and the fourth compares 7 to 3. At this point, the search should examine the right subtree of the node with key 3. However, since such subtree does not exist, the search concludes that no node with key 7 exists in the BST.

Inserting new nodes on a BST must be performed in a way that maintains the order property of the BST. Given a BST and a new node ***n*** whose key is ***k***, the goal is to insert the node ***n*** in such a way that the tree remains a BST. It is assumed that no node exists in the tree with a key ***k***. The process is very similar way to searching. The key of the new node (i.e., ***k***) is compared to the key of the root. If ***k*** is less than the key of the root, the new node must be inserted in the left subtree. Otherwise, the new node must be inserted in the right subtree. This is repeated recursively until either no left subtree or no right subtree exists. The new node is inserted there.

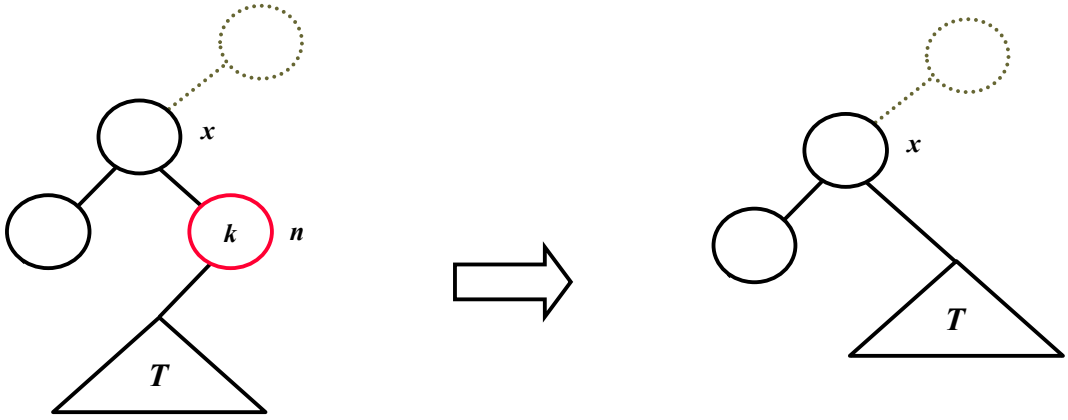
Thus, if a new node with key 7 is to be inserted on the same example BST used to illustrate searching above, the same steps of searching for a node with key 7 would be performed. When the search concludes with failure because of the lack of a right subtree of the node with key 3, the new node is inserted as the right child of this node, as shown below.



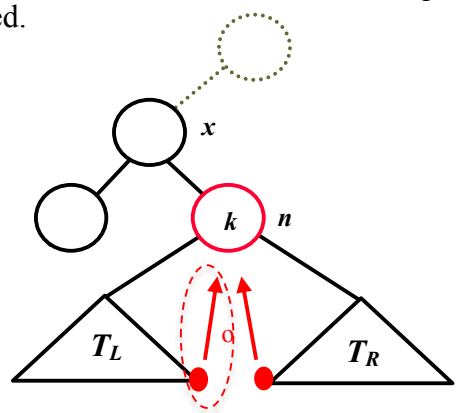
Deleting a node from a BST must also be done in a way that maintains the order property of the BST. Given a BST and a node ***n*** with key ***k*** in the BST, the goal is to delete the node ***n*** in such a way that the tree remains a BST. The actions taken after the node is deleted depend on the location of the node in the tree. If the node ***n*** is a leaf, then the node is just deleted, with no further actions necessary. This is shown in the figure below.



If the node n is not a leaf, but it has only one subtree ***T*** (either left or right), the node ***n*** is removed from the tree, and the its subtree ***T*** is then "connected" to ***n***'s parent (***x***), as shown in the figure below. In this example, ***n*** has only a left subtree ***T***. When ***n*** is deleted, ***T*** must be re-attached to the BST. Since all the nodes in ***T*** have keys that are less than the key of ***n*** (i.e., ***k***), and since ***x***, the parent of ***n***, has a key which is less than ***k***, making ***T*** the left subtree of ***x*** (i.e., making the root of ***T*** the right child of ***x***) maintains the order property of the BST (key(***x***) < key (***i***), where ***i*** is any node in ***T***).

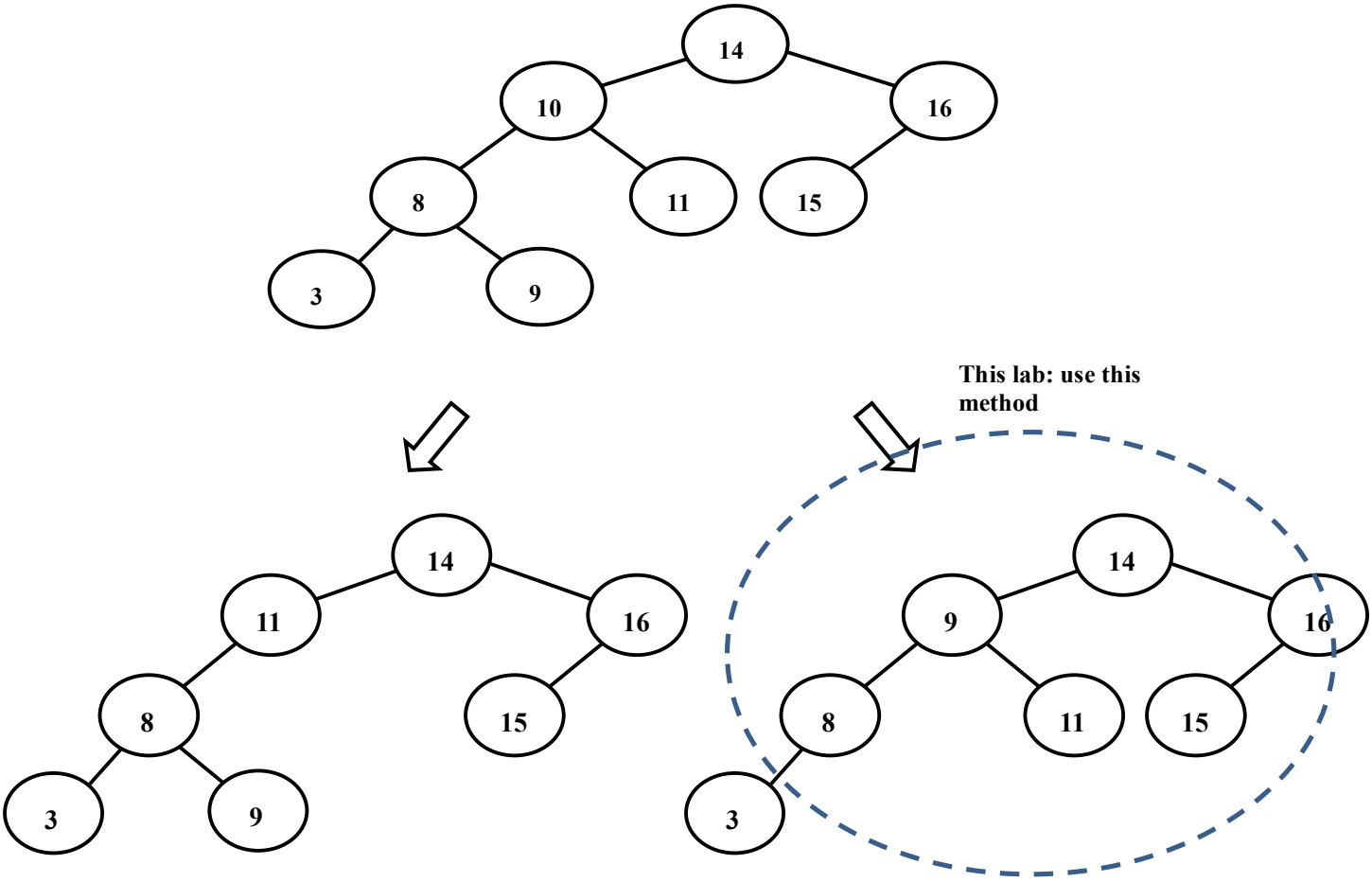


Finally, if the node to be deleted has two subtrees, then when the node is deleted, a node from one of the two subtrees replaces it. The replacement node is selected to maintain the order property of the BST. If the replacement node is selected from the left subtree of the deleted node ***n***, then it must be the node with the largest key in this subtree (i.e., right most as explained above); call it ***l***. This way when ***l*** replaces n, the key of ***l*** is greater than the key of any node its now left subtree, and at the same time smaller than the key of any node in its now right subtree. This is shown the in figure below. Similarly, if the replacement node is selected from the right subtree of the deleted node ***n***, then it must be the node with the smallest key in this subtree (i.e., the left most as explained above); call it ***s***. This way when ***s*** replaces ***n***, the key of ***s*** is smaller than any node in its now right subtree, and at the same time larger than the key of any node in its now left subtree. Thus, irrespective of which subtree to select a replacement node from, the order property of the BST is preserved.



This lab: Use circled method

Thus, in the example BST shown below, deleting the node with the key 10 can be done in the two ways shown; by replacing the deleted node with the node with the key 11, or the node with the key 9. **For this lab, you must use the "replace the deleted node with the maximum node in the left subtree method"; that is node 9 must be selected to take node 10's location in the tree. Otherwise, your tree will differ from that in the exercise program after a remove command, and your printprobes output will not match that of exercise after a deletion.**



In this assignment, you will be penalized for having memory leaks in your code. Use of the `valgrind` memory tester is strongly recommended to catch invalid reads, writes, and allocation/deallocation. It will tell you when and on what function and line the error occurs. For memory leaks, it will tell you where the leaking block was allocated. Read the tutorial on `valgrind` that is released on Quercus and use it to make sure that your program is free of memory leaks.