CptS 223 – Advanced Data Structures in C++

Written Homework Assignment 4: More Algorithm Analysis and Red-black Trees

Assigned: Wednesday, October 7, 2020
Due: Wednesday, October 14, 2020

I. Problem Set:

1. **(7 pts)** Given the following two functions:

```
int g (int n)
{
  if(n <= 0)
  {
    return 0;
  }
  return 1 + g(n - 1);
}
```

```
int f (int n)
{
  int sum = 0;
  for(int i = 0; i < n; i++)
  {
    sum += 1;
  }
  return sum;
}
```

a. (2 pts) State the runtime complexity of both f() and g().
The runtime complexity of g() is O(n).
The runtime complexity of f() is O(n).

b. (2 pts) State the memory (space) complexity for both f() and g().
The memory space complexity for g() is O(1).
The memory space complexity for f() is O(1).

c. (3 pts) Write another function called "int h(int n)" that does the same thing, but is significantly faster.

```
Int h(int n)
{
        Return n;
}
```

2. **(5 pts)** State g(n)'s runtime complexity:

```
int f (int n){
  if(n <= 1){
    return 1;
  }
  return 1 + f(n/2);
}

int g(int n){
  for(int i = 1; i < n; i *= 2){
    f(i);
  }
}
```

The runtime complexity of g(n) is O(n^2).

3. **(18 pts)** Provide the algorithmic efficiency using Big-O for the following tasks. Justify your answer, often with a small piece of pseudocode or a drawing to help with your analysis.

   a. (3 pts) Determining whether a provided number is odd or even.

```
Int g(int n)
{
    If(n % 2 == 0)
    {
            Std::cout << "n is even\n";
            Return 0;
    }
    Else
    {
            Std::cout << "n is odd\n";
            Return 1;
    }
}
```

The algorithmic efficiency of using Big-O notation for this task is O(1). The algorithm will always use the same number of steps no matter what n is inputted as.

b. (3 pts) Determining whether or not a number exists in a list.

```
Bool exists(std::list<int> l, int n)
{
    For(std::list<int>::iterator i = l.begin(); i != l.end(); ++i)
    {
            If(*i == n)
            {
                    Return true;
            }
    }
    Else
    {
            Return false;
    }
}
```

The algorithmic efficiency of using Big-O notation for this task is O(n). The number of operations that this algorithm requires is directly proportional to the size of the list.

c. (3 pts) Finding the smallest number in a list.

```
int findMin(std::list<int> l)
{
        Int min = l.front();

        For(std::list<int>::iterator i = l.begin(); i != l.end(); ++i)
        {
                If(*i < min)
                {
                        min = *i;
                }
        }
        Return min;
}
```

The algorithmic efficiency of using Big-O notation for this task is O(n). The number of operations that this algorithm requires is directly proportional to the size of the list.

d. (3 pts) Determining whether or not two **unsorted** lists of the same length contain all of the same values (assume no duplicate values).

```
Bool duplicate(std::list<int> l, std::list<int> l2)
{
    Bool check = false;
    for(std::list<int>::iterator i = l.begin(); i != l.end(); ++i)
    {
            for(std::list<int>::iterator j = l2.begin(); j != l2.end(); ++j)
            {
                    If(*l == *l2)
                    {
                            Check = true;
                            Break;
                    }
                    Else
                    {
                            Check = false;
                    }
            }
    }

    Return check;
}
```

The algorithmic efficiency of using Big-O notation for this task is O(n^2). The number of operations that this algorithm requires is based on the size of both of the list * the size of the list, in addition this algorithm includes a nested for loop within a for loop.

e. (3 pts) Determining whether or not two **sorted** lists contain all of the same values (assume no duplicate values).

```
Bool duplicate(std::list<int> l, std::list<int> l2)
{
    For(std::list<int>::iterator i = l.begin() j = l2.begin(); i != l.end() j !=
    l2.end; ++i ++j)
    {
        if(*l != *l2)
        {
            Return false;
        }
    }
    Return true;

}
```

The algorithmic efficiency of using Big-O notation for this task is O(n). The number of operations that this algorithm requires is based on the size of one of the lists and only requires one for loop based on the length of one list.

f. (3 pts) Determining whether a number is in a BST.

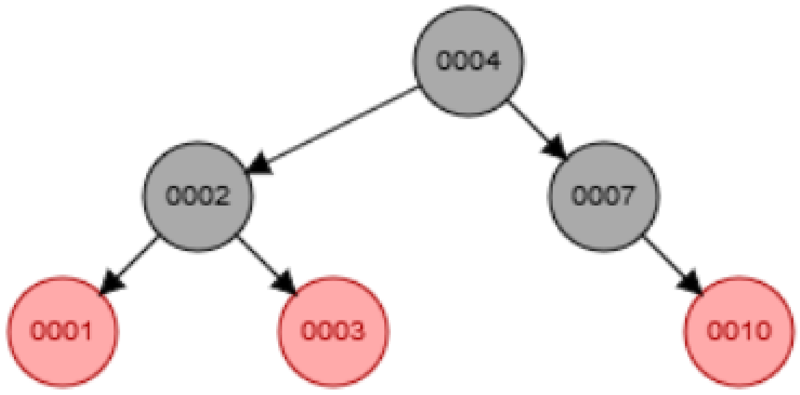
```
Bool search(Node<int> *root, int n)
{
    Return searchHelper(*root, n);
}
Bool searchHelper(Node<int> *root, int n)
{
    If(root == nullptr)
        Return false;
    If(root == n)
        Return true;
    If(n < root->left)
        Root = root->left;
    Else
        Root = root->right;
}
```

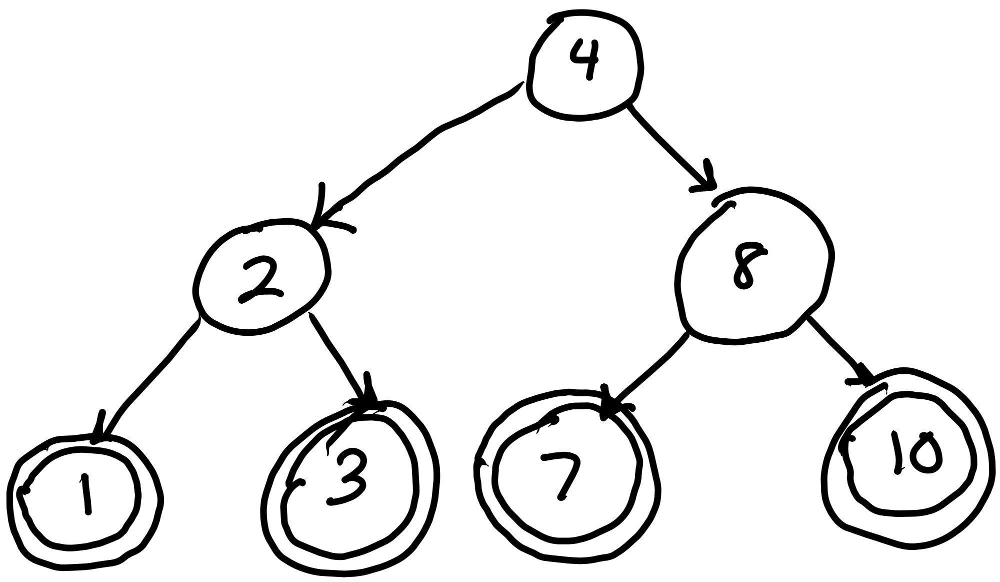
The algorithmic efficiency of using Big-O notation for this task is O(logn). The number of operations that this algorithm requires is based on the height of the BST tree which increases at a rate of log(n) (n being # of nodes).

4. **(5 pts)** Compare and contrast the worst-case time complexities for inserts, deletes, and searches in an AVL tree and red-black tree. Provide some insights to how the complexities are determined.

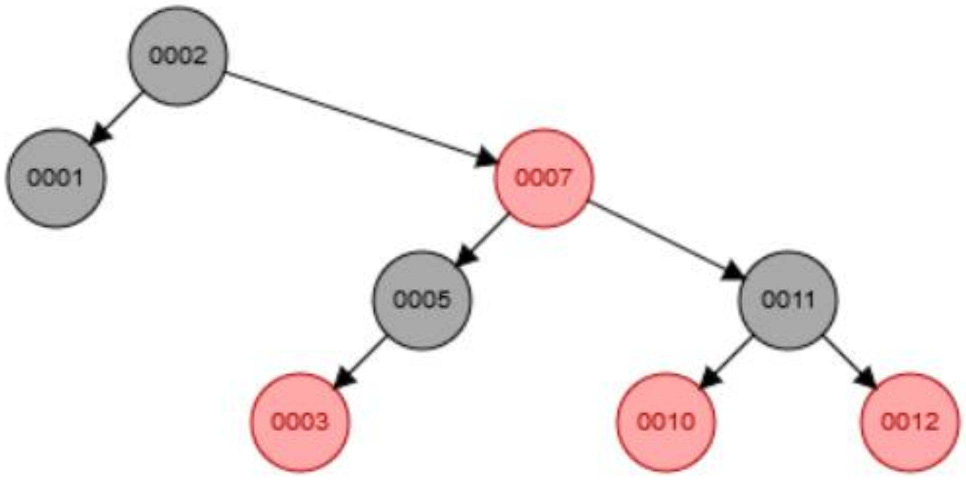
   The worst-case time complexities for inserts, deletes, and searches in an AVL tree and red-black tree are all O(logn). Because both trees are self-balancing and because their height determines the time-complexity of each of their traversals. The worst-case time complexities will be O(logn) because the heights of both of the trees are increasing at a rate of (log2n) or (logn).

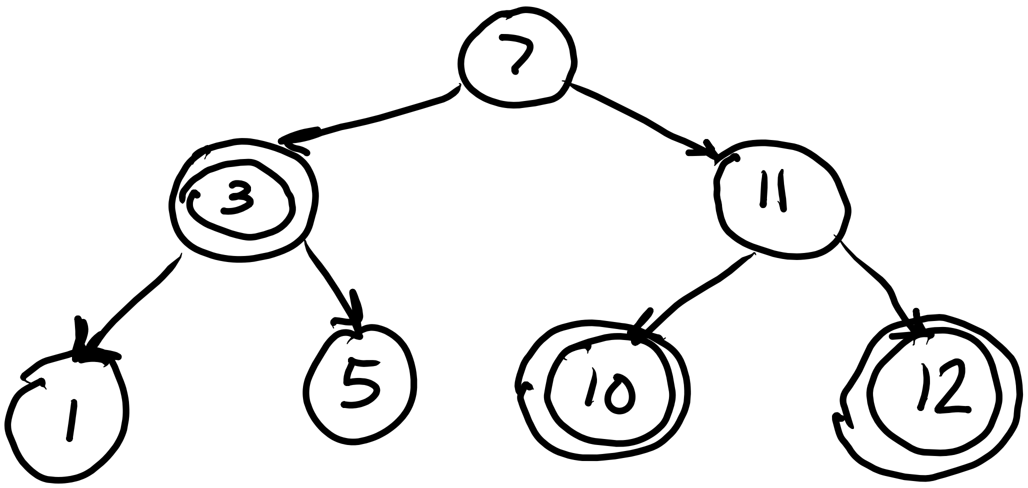
5. **(5 pts)** Insert the value "8" into the following Red-Black tree; draw the result. Use Double-circle to denote red nodes and single circle to denote black nodes.





6. **(5 pts)** Delete the value "2" from the following Red-Black tree; draw the result. Use Double-circle to denote red nodes and single circle to denote black nodes.





7. **(10 pts)** Write an algorithm using pseudocode or C++ for inserting integers into a red-black tree.

Insert the new node red

If the new node is the root, change the new node black and end insertion.

Traverse through the tree (left if the integer is less or right if the integer is greater than the respective parent node) then insert the new node to its respective parent node

While all the conditions (root == black, red nodes left and right do not point to red child nodes, black height of node difference in subtrees is no greater than 1)

- If the new node is inserted and connected to a parent red node, then change the parent and uncle node to black.

- If the violated node has a red parent and a black uncle, then rotate
  - Perform a right rotation if the node was inserted to the left sub tree
  - Perform a left rotation if the node was inserted to the right sub tree
- Change the root node color to black.

II. Submitting Written Homework Assignments:

1. On your local file system, create a new directory called HW4. Move your HW4.pdf file in to the directory. In your local Git repo, create a new branch called HW4. Add your HW4 directory to the branch, commit, and push to the remote origin.

III. Grading Guidelines:

This assignment is worth 55 points. We will grade according to the following criteria:

- See above problems for individual point totals.