Homework #2 Due 2/4/18 by 11:59pm*

Natalie Pueyo Svoboda, 997466498

February 4, 2018

Homeworks are to be submitted to Gradescope by the above due date. List your classmate collaborators on the front page. The quiz will occur in-class on Wednesday, 2/7/18. Problems come from:

Weiss "Data Structures and Algorithm Analysis in C++"

Sedgewick & Wayne "Algorithms"

- (Weiss) We are given an array that contains N numbers. We want to determine if there are two numbers whose sum equals a given number K. For instance, if the input is 8, 4, 1, 6, and K is 10, then the answer is yes (4 and 6). A number may be used twice. First, give an $O(n^2)$ algorithm to solve this problem. Then give the pseudocode of an $O(n \log n)$ algorithm to solve this problem (Hint: sort first).
- (Weiss) Give the pseudocode of a data structure that supports the stack `push` and `pop` operations, and a third operation `findMin`, which returns the smallest element in the data structure, all in $O(1)$ worst-case time.
- (Sedgewick & Wayne) Inserting the keys in the order $AXCSERH$ into an initially empty binary search tree gives a worst-case tree where every node has one null link, except one at the bottom, which has two null links. Give five other orderings of these keys that produce worst-case trees.
- (Sedgewick & Wayne) Suppose that a certain binary search tree has keys that are integers between 1 and 10, and we search for 5. Which sequence below cannot be the sequence of keys examined?
  a. 10, 9, 8, 7, 6, 5

*Last updated February 4, 2018

b. 4, 10, 8, 7, 5
c. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5
d. 2, 7, 3, 8, 4, 5
e. 1, 2, 10, 4, 8, 5

- (Sedgewick & Wayne) Give pseudocode for a binary search tree method `height()` that computes the height of the tree. Develop two implementations: the first, a recursive method (which takes linear time and space proportional to the height), and the second, a method that adds a field to each node in the tree (and takes linear space and constant time per query).

1 First Problem

Solution. a) A $\mathcal{O}(n^2)$ algorithm that could be used to solve this problem would involve two `for` loops with an `if` statement. The first `for` loop would be used to through the array one element at a time to get the first number in the sum. The second `for` loop would be nested in the first `for` loop and would also go through each element in the array to provide the second number in the sum. Lastly, there would be an `if` statement inside the second for loop that would check if each sum is equal to the given K value.

b) On the other hand, a $\mathcal{O}(n \cdot log(n))$ algorithm in pseudocode to do the same thing could be:

```
low := 0;
high := N - 1;
sort(array);
for(i := 0; j < N; j := j + 1)
   mid = floor((low + high) / 2);
   if(array[mid] = (K - array[i]))
      print("A match was found");
      break;
   else if(array[mid] < (K - array[i]))
      low := mid + 1;
   else
      high := mid - 1;
```

2 Second Problem

Solution. The way to implement a data structure that supports the stack `push()`,`pop()`, and `findMin()` is to use to stacks. The first is used as the 'normal' stack which saves each new value. The second stack, `minStack`, is used to save the history of the minimum values as they are pushed onto the stack.

```
class stackWithMin{
   Stack stack;
   Stack minStack;
   void listPrepend(Stack, item);
   void listRemoveAfter(Stack, int);
   void push(item);
   int pop();
   int findMin();
}swm;

push(newItem){
   swm.listPrepend(swm.stack, newItem);
   if newItem <= swm.minStack->head then
      swm.listPrepend(swm.minStack, newItem);
}

pop(){
   poppedItem := swm.stack->head;
   swm.listRemoveAfter(swm.stack, 0);
   if poppedItem = swm.minStack->head then
      swm.listRemoveAfter(swm.minStack, 0);
   return PoppedItem;
}

findMin(){
   minValue := swm.minStack->head;
   return minValue;
}
```

3 Third Problem

Solution. The following are five ways to order the keys $AXCSERH$ so as to give worst-case BST:

a) $ACEHRSX$

b) $ACEXSRH$

c) $ACXESHR$

d) $XSRHECA$

e) $XASCREH$

4 Fourth Problem

Solution. Out of the following sequences, only **(d)** would not be a possible sequence of examining a BST. This is because in case **(d)**, at the second level node 7 would be parent node to **both** nodes 3 and 8 according to BST rules. In this case, when searching for the value 5, by BST rules, the search would not traverse to another branch (see figure).

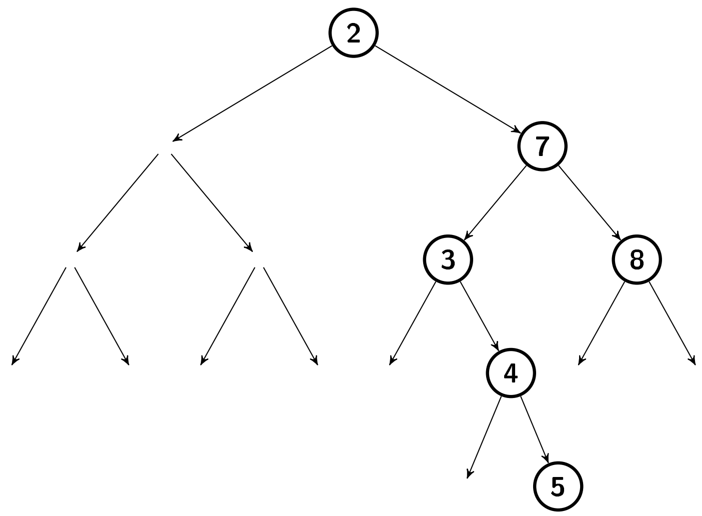
a) 10, 9, 8, 7, 6, 5

b) 4, 10, 8, 7, 5

c) 1, 10, 2, 9, 3, 8, 4, 7, 6, 5

d) 2, 7, 3, 8, 4, 5 (Does not work)

e) 1, 2, 10, 4, 8, 5



5 Fifth Problem

Solution. The first implementation with linear time and space proportional to the height:

```
int height(tree){
   left, right := 0;
   if tree = null then
      return -1;
   else
      if tree.leftbranch != null then
         left = height(tree.leftbranch);
    if tree.rightbranch != null then
```

```
        right = height(tree.rightbranch);
 return max(left, right) + 1;
}
```

The second implementation with linear space and constant time per query. This one requires that I also implement a way to add a field to a node. In this case, the `insert()` method recalculates each node height value using recursion after a new node is inserted. When the `height()` method is called, the method returns the value found at the pointer to the height of the root calculated in `insert()`.

```
node* insert(node* root, node* newNode){
   if root = null then
      newNode->height := 1;
      return newNode;
   if root->value < newNode->value then
      root->right := insert(root->right, newNode);
   else
      root->left := insert(root->left, newNode);

   // height at root starts at 1 so add that to calculation
   root->height := max(root->left->height, root->right->height) + 1;
   return root;
}

height(tree){
   heightOfTree->root;
   return heightOfTree;
}
```