Final Exercises

CSC 122 B
Due: 5/14 at 5:00 PM via Moodle

Submission format:
Please zip up your Visual Studio project and upload to Moodle, as you would for any assignment.

If you need to email your project to me for some reason, please recall the following from hw0:

1. With DePauw mail servers, .exe files are not allowed as attachments – even when they're inside of a .zip. Unfortunately, Visual Studio projects contain an .exe file. So to get around this problem, we'll just delete it.
    a. It's ok to do this – the .exe file will be automatically regenerated every time you compile, so it's ok to delete it right before making a .zip and emailing the .zip.
2. The .exe. file is in the Debug folder. Confusingly, there are actually multiple Debug folders, but the one you want is probably the first one you'll see. So find the Debug folder that has the `.exe` file.
    a. The .exe file will have the extension .exe, and will be of type "Application" according to Windows.
3. Delete the .exe file.
4. Then create the .zip as usual. This is the file you can email to me.

Policies:
Please make sure you have read and agree to the policies specified in the Final Procedures document. Failure to follow these policies can result in severe penalties, including failure of the exam, the course, and academic integrity charges.

Sorted Stack ADT

In this exercise we consider a new ADT: a *sorted stack*. A sorted stack has the following operations:

- push – put a new item on the stack
- pop – remove and return the *lowest item* from the stack
- peek – return the *lowest item* from the stack, but don't remove it
- getSize – return the number of elements on the stack.
- isEmpty – return true if the stack is empty, false otherwise.

Note in particular the pop and peek operations. When I say "lowest item", I'm referring to the *value* of that element (according to some sorting order), not the *position* in the stack. For example, suppose I'm talking about a sorted stack of ints, and the stack contains the numbers 6, 4, 8, 9, 3, 12, 15, 7, 5 in some order. The pop would return the 3, since it's the lowest item on the stack. If I then call pop again, it would return 4, and so on.

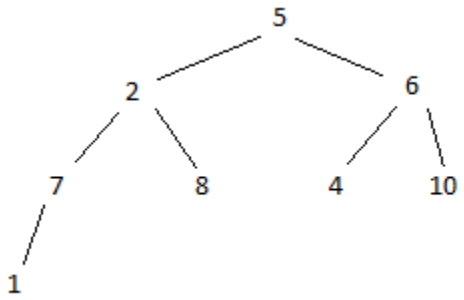
Representing a Tree with an Array

We're going to implement the sorted stack ADT using a special kind of binary tree. So far we've implemented binary trees using pointers, but here we'll instead use the other key building block of all data structures, an array.

To represent a binary tree as an array, we'll do the following:

- Don't use index 0 at all.
- For the element at index i, its left child is at index i*2, and its right child is at index i*2+1.

Suppose, for example, we have the array [?, 5, 2, 6, 7, 8, 4, 10, 1]. The "?" indicates that it doesn't matter what is at index 0, since that position isn't used. This array represents the following tree:



Observe, for example:

- 6 has left child 4 and right child 10
- 6 is at index 3
- 4 is at index 6, which is 3*2
- 10 is at index 7, which is 3*2+1

We can also find the parent by using /2. For example:

- 4 is at index 6, and 4's parent is at index 3. 6/2 = 3.
- 10 is at index 7, and 10's parent is at index 3. 7/2 = 3. (using "/" as "integer division")

Verify yourself with a couple other examples that when considering the provided array and tree, an element at index i has left child at i*2, right child at i*2+1, and parent at i/2.

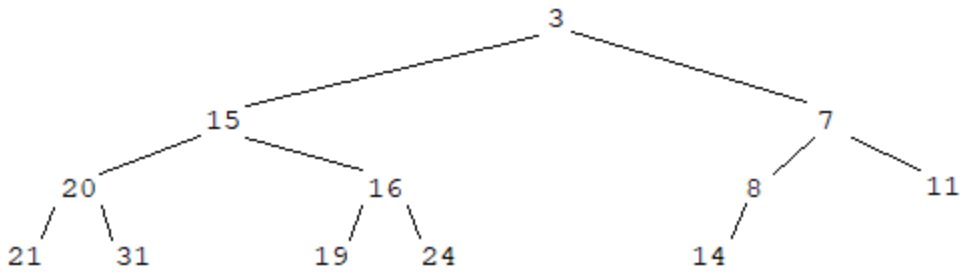
It is possible to create a binary tree for which the corresponding array would have gaps in the middle. But when starting with an array, it is always possible to create a corresponding binary tree when following the rules above.

A Tree with Particular Properties

We need to consider binary trees with particular properties. These will not be binary *search* trees, but they will have a different kind of ordering requirement:

Property #1: Every node must be smaller than its children.

So, for example, the tree **above** does ***not*** meet this requirement. However, the tree **below** does meet this requirement:



Note that the above requirement does not necessarily mean that each level has nodes less than all nodes at the next level. For example, 15 is not less than its "nephews/nieces" at the next level, 8 and 11.

Also note that by the above requirement, the root is guaranteed to be the smallest element in the entire tree.

We have one more requirement for these special trees:

Property #2: We must be able to represent the tree as an array without any "gaps" in the array.

As you'll see, we won't have any trouble representing a tree in this way in our code. In fact, it will be convenient – this array approach will simplify our code when compared to a possible pointer-based implementation. It will also enable us to easily find the parent of node i by simply computing i/2.

Implementing the Sorted Stack ADT

So, back to the sorted stack ADT. We will implement this ADT using a tree with the properties described above, represented as an array. Please do this in files called TreeSortedStack.cpp and TreeSortedStack.h. We'll assume that the values we're working with are ints.

I recommend you start by studying very carefully the hw3 code for IntArrayLS – the array-based implementation of the linear structure ADT. If you don't understand that code at this time, you should not move forward in this problem until you do. For your convenience, the .h and .cpp file contents are included in IntArrayLS-given.txt. It turns out that you will want to copy some of this code exactly as it is into your TreeSortedStack implementation.

In your class, include public methods for the five sorted stack operations listed above (push, pop, peek, getSize, isEmpty). In addition, please include a constructor that takes an int for initialCapacity, a destructor, and a print. Any other methods you write should be private.

If the user tries to peek or pop from an empty structure, show an error message and return -1. (See sample output farther below.)

peek, getSize, and isEmpty should be rather straightforward once you understand how this structure works. I have some additional comments for print, push and pop.

For print:

- Take the print method and helper methods I provided in hw6 on BinarySearchTrees. This gives a tree-like visual representation of the structure. You'll need to adjust the code a little bit, since that code uses a pointer-based representation of a tree rather than an array like we're using here.

What should pop return? Recall that with this special tree structure, the smallest element is at the root. So that element should be returned. The tricky part is to figure out what the tree should look like after a pop. (What should the new root be once the old root is removed?). Similarly for push: a new element must be added while still maintaining the required tree properties.

For push:

- You will sometimes need to expand the array, because the structure is growing.
- Once you're sure you have room in the array, put the new element at the first open spot at the end of the array. But this won't necessarily satisfy property #1. So we may need to move it up the tree in a systematic way in order to maintain property #1:
    - Let childID be the index at which you just put the new element
    - Let parentID be the index of the parent of the new element
    - Loop as long as you haven't gotten to the root and the parent is bigger than the child
        - Swap the parent and child in the array
        - Let childID = parentID
        - Compute a new parentID for the new value of childID

For pop:

- You'll need to return the element at the root. The question is, what element should next become the root?
- Start by moving the last element in the array to the root position. But this won't necessarily satisfy property #1. So we may need to move it down the tree in a systematic way in order to maintain property #1:
    - Let parentID start at the root (referring to the element we just moved there)
    - Let smallestID be the index of the smaller of the two children
    - Loop as long as parentID doesn't refer to a leaf and the parent is bigger than the child
        - Swap the parent and child in the array
        - Let parentID = smallestID
        - Compute a new smallestID for the new value of parentID

Notice the similarities in the pseudocode for the key parts of both push and pop. I recommend you try these algorithms on paper on a specific example, to see how they work.

Sample Output

Consider the provided Tester.cpp file. You may add another test function if you want to, and call it from the main. Please do not modify the officialTest function at all, though. If you think you need to modify it for your code to work, then there is something wrong with your code.

When the officialTest function is run, you should obtain the output below. I recommend you do the work shown below by hand first, to practice, before you try to implement anything.

```
isEmpty: 1
push 10
```

```
------------------------------
10
------------------------------
push 12
------------------------------
10
    L: 12
------------------------------
push 5
------------------------------
5
    L: 12
    R: 10
------------------------------
push 7
------------------------------
5
    L: 7
        L: 12
    R: 10
------------------------------
push 9
------------------------------
5
    L: 7
        L: 12
        R: 9
    R: 10
------------------------------
pop: 5
peek: 7
------------------------------
7
    L: 9
        L: 12
    R: 10
------------------------------
push 16
------------------------------
7
    L: 9
        L: 12
        R: 16
    R: 10
------------------------------
push 13
------------------------------
7
    L: 9
        L: 12
        R: 16
    R: 10
```

```
        L: 13
------------------------------
push 4
------------------------------
4
    L: 9
        L: 12
        R: 16
    R: 7
        L: 13
        R: 10
------------------------------
push 11
------------------------------
4
    L: 9
        L: 11
            L: 12
        R: 16
    R: 7
        L: 13
        R: 10
------------------------------
==========================================
getSize: 8
peek: 4
pop: 4
------------------------------
7
    L: 9
        L: 11
        R: 16
    R: 10
        L: 13
        R: 12
------------------------------
pop: 7
------------------------------
9
    L: 11
        L: 12
        R: 16
    R: 10
        L: 13
------------------------------
pop: 9
------------------------------
10
    L: 11
        L: 12
        R: 16
    R: 13
```

```
------------------------------
pop: 10
------------------------------
11
    L: 12
        L: 16
    R: 13
------------------------------
pop: 11
------------------------------
12
    L: 16
    R: 13
------------------------------
pop: 12
------------------------------
13
    L: 16
------------------------------
pop: 13
------------------------------
16
------------------------------
pop: 16
------------------------------
------------------------------
==========================================
getSize: 0
Structure is empty. pop returning -1 as placeholder.
peek: -1
Structure is empty. pop returning -1 as placeholder.
pop: -1
```