Lab_heaps: Precarious Priority Queues

Welcome to Lab Heaps!

Overview

Heaps are used to quickly find the highest priority items. The heap structure uses an array to represent a tree. The insert, and remove functions run in O(lg n) time. The building of a heap has a run time of O(n), and this is where it differs from previous structures such as an AVL tree.

Heap Properties

A heap is a complete binary tree, where all the descendants of the root have less of a priority than the root. For a Min (Max) Heap, the root will always be the smallest (largest) element. A Heap is not like a BST; an in-order traversal does not necessarily yield an ordered list.

Exercise 1: Suppose the current node is at index **i**, fill in the blank for the indices of certain locations in the tree. You will be implementing these as functions in your lab. Remember, these formulas depend on whether you are populating the 0th index of the array or not.

Current node =   **i**

Root Index   =   0   Left child   =

Right Child   =   Parent   =

Root Index   =   1   Left child   =

Right Child   =   Parent   =

Insert, Remove, and Heapify

Inserting into the heap:
- append the new element to the end of the array storing the heap.
- "bubble up" the new element until it is in the right position. **HeapifyUp** is the "bubble up" function.

Removing an element from a heap:
- pop out the root and replace it with the element at the end of the array
- "bubble down" from the root to preserve the heap property. **HeapifyDown** is the "bubble down" function.

Exercise 2.1: Suppose 7 is inserted into the heap below. What will 7's left and right children be?
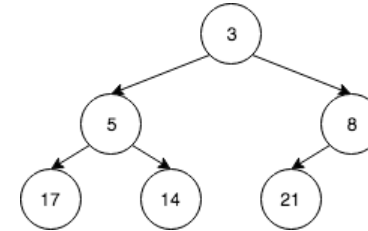
Left child   =   Right Child   =

Exercise 2.2: What is the array representation of the tree after **7** is inserted?

*							

Exercise 2.3: What is the array representation of the tree if **3** is removed, after **7** is inserted?

*							

Build a Heap:

When given an unsorted array, the benefit of using heaps is that you can build it in O(n) time. The right half of the array (All of the nodes that are leaves) is already in a heap structure. Recursively using heapifyDown() from the last element that is not a leaf (that is, from the second last "level" in the tree) to the beginning of the array will convert the unsorted array to a heap.

*	98	90	3	14	34	67	29

Exercise 3: After using the buildHeap algorithm on the array above, draw the corresponding tree.

In the programming part of this lab, you will complete the following functions:

- root(), leftChild(), rightChild(), parent()
  - o Exercise 1 of the worksheet will be helpful
- hasAChild(), maxPriority()
  - o For hasAChild(), think about how you can use the previous implemented functions to find if a node has a child.
- heapifyDown(), heapifyUp()
  - o Given in lecture
- heap(const std::vector<T>& elems)
  - o Read the "Build a Heap" section
- pop() and peek()
  - o pop() is similar to removeMin() from lecture
- push() and empty()
  - o push() is similar to insert() from lecture
- updateElem()
  - o Given a key and a new priority, update that key's priority and reformat the tree to satisfy heap properties in O(log n)

As your TA and CAs, we're here to help with your programming for the rest of this lab section! 😌