CMPS 2200 Recitation 06

As we've seen, greedy algorithms often have to find the next best decision. In this lab, we'll look at the heap, or priority queue, data structure, which is useful for greedy algorithms. In particular we will implement a binary "min-heap", which allows for efficient insertion and deletion operations.

In class we will discuss *meldable* heaps, but for this lab we will implement *binary heaps* using Python lists. A binary heap is a balanced binary tree that has the *heap property*: the value at any node is less than or equal to its children.

We will implement a data structure with the following operations

- *deleteMin*: remove the minimum element from the heap
- *insert(x)*: insert element x into the heap

1 Before we implement a heap, let's consider if we just maintain an unordered list/sequence. What is the work/span of *deleteMin* and *insert(x)*?

enter answer in answers.md

.
.
.

2 Our approach to implementing a heap will be to use a list that implicitly represents an almost-complete binary tree. An almost-complete binary tree has the property that every level of the tree has all possible nodes except the last level. The last level is filled left to right but may be missing nodes on the right if the number of elements in the heap is not a power of two.

.
.
.

2a) For a heap with n elements, what does this imply about the depth of the associated almost-complete binary tree?

enter answer in answers.md

.
.
.

2b) By maintaining the heap property, where is the minimum element of the heap? Can we say anything about the maximum element?

enter answer in answers.md

.
.
.

3 How do we represent an almost-complete binary tree with a list? Suppose we have a list of n elements, indexed from 0 to $n-1$. We can construct a tree from these elements by assigning the root to index 0, the left child of the root to index 1 and the right child of the root to index 2. Following this pattern, how can the left and right child of any node could be accessed? How can we access the parent of an element in the heap? For reference, our text provides some guidance.

Figure 1 shows an example of a binary heap represented by list $[10, 12, 15, 25, 30, 36]$:

Implement the stubs `lchild`, `rchild` and `parent` in `main.py`

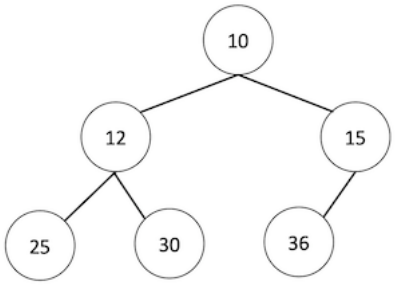

Figure 1: Binary heap represented by list $[10, 12, 15, 25, 30, 36]$

4 Now that we have defined a binary tree on a list, we'll look at two useful operations that will help us implement our binary heap. Suppose that we are at some node index x in the heap and we find the the heap property is violated. Then we will need to restore the heap property by traversing up or down the heap. So we will define:

- *reheapUp*(x): starting at element x, swap upward as long as the heap property is violated. By "swap upward," we mean swap a node with its parent.
- *reheapDown*(x): starting at element x, swap downward with the *smaller* child as long as the heap property is violated.

Implement these operations in `main.py`. What is the work/span of these two operations?

enter answer in answers.md

.
.
.

5 We can finally implement `deleteMin` and `insert` in `main.py`.

5a) For *deleteMin* we remove the root and replace it with the rightmost leaf in the heap. Then we must restore the heap property downward. Implement `deleteMin` and check its correctness.

.
.
.

5b) For *insert*, we add the new element as the rightmost leaf in the tree and restore the heap property upward. Implement `insert` and check its correctness.

.
.
.

6 Let's now use our heap for something useful. Recall that selection sort repeatedly locates the minimum element in order to sort. Implement `heapsort` by using a heap to efficiently retrieve minima in a list. To do so, first add all the elements to the list, then repeatedly delete the minimum to construct the sorted list. Test your implementation with `test_heapsort`. What is the work and span of `heapsort`?

enter answer in answers.md