COSC122 (2019) Lab 7.2
Heaps

Goals

This lab will give you practice with heaps and graphs; in this lab you will:

- complete the implementation for a heap class;
- implement a 3-heap

You should be familiar with Section 6.8 to 6.10 of the online textbook before doing this lab. [1]

Heaps

Data for a heap is typically stored in a list, rather than a set of connected node objects. This works well because heaps are always *complete* trees—that is, there are no gaps their structure. Instead of traversing *left* and *right* fields, you can calculate the list indices of child or parent items. If the root is stored in entry 1 (as in the textbook) then we have:

- left child = 2 × parent
- right child = 2 × parent + 1
- parent = child//2
  (Python's integer/floor division truncation works in our favour.)

heaps.py

The `heaps` module contains two classes—an abstract `Heap` class, and a partial implementation of a `MinHeap`. The doctests for the `MinHeap` class describe the basic operation of each method.

Inserting into a Heap

When an item is inserted into a heap, instead of searching for the correct location from the root (as you would with a binary search tree), the item is appended to the end of the heap and is *sifted-up* (you may also see the terms 'trickle-up', 'bubble-up', and 'percolate-up') to the correct *level* of the tree. As left and right branches don't have any special meaning, it doesn't matter which side of the heap the item ends up on.

Figure 1 shows the process for inserting a new item into a min-heap. It is appended to the end of the heap, and swapped with parents that are greater than it; this process repeats until its parent is either smaller than it, or the item reaches the root node.

Before starting on the implementation, complete the *Heap arrays* questions (quiz 7.2).

> ***Complete the* Heap arrays *questions quiz 7.2.***

In the `heaps` module, the `MinHeap` class is missing an implementation for the `insert` method. The `_sift_up` method has been completed for you, so you only need to implement the first step of the insert algorithm: append the item to the end of the `self._items` list, and call `_sift_up` with the index of the item.

Test your implementation with the provided doctests (the tests for `pop_min`, `peek_min`, and `validate` won't pass yet, but you shouldn't get any errors for `insert`).

[1] The textbook table of contents is here in case the direct link to 6.8 fails.

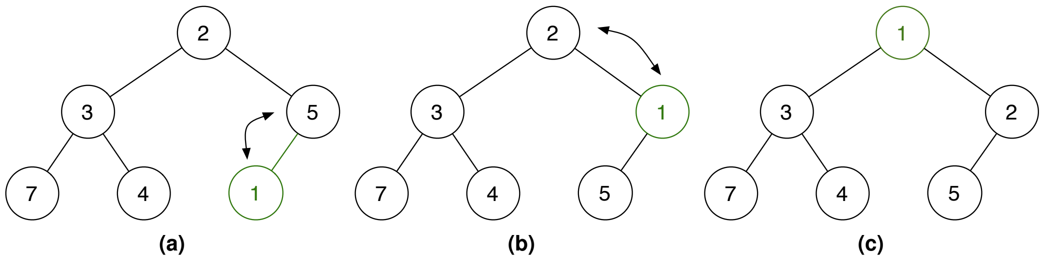

Figure 1: Inserting a new item into a min-heap.

Deleting/Popping from a Heap

Items can only be removed from the root of the heap—either the smallest (in a min-heap), or greatest (max-heap) item. Since a heap can't be arbitrarily traversed, deletions are a bit simpler than a binary search tree.

Figure 2 shows the process for popping/deleting the root item from a min-heap. The root item is replaced with the last item in the heap. The new root is then *sifted-down* to the correct level of the tree—that is, if any of its children are smaller than it, it is swapped with its smallest child.

> ***Complete the* Pop min *question in Lab quiz 7.2.***

The `MinHeap` class is missing an implementation for its `pop_min` method, which removes the smallest item from the heap (the root) and returns it. The `_sift_down` method has been provided for you, so you only need to implement the first step of the algorithm: replace the first item with the last and call `_sift_down` with the index of the root item. You'll also have to handle a couple of special cases—think about what should happen if there are one or two items in the heap; what if there are no items?

Test your implementation with the provided doctests.

Validating a Heap

Finally, you need to implement the `validate` method. It should return `True` if every node in the heap is smaller-than, or equal-to, its children; otherwise, it should return `False`. Although the heap is a kind of tree, the traversal algorithm that you should implement should be *iterative*—NOT *recursive*.

Complete the `validate` method, starting with the second element in the heap (the first element is the root, and has no parent node), and make sure that each node's parent is smaller-than, or equal-to, the current node.

Run the doctests to test your implementation.

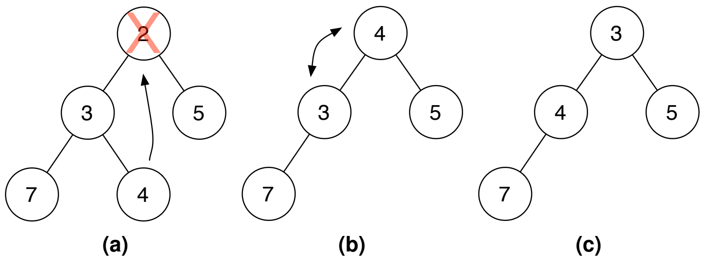

Figure 2: Deleting the root item in a min-heap.

Find and Fix the Bug in `MinHeap`

Although all of the doctests pass, there's a nasty bug in `MinHeap`. Try constructing the following heap:

```
>>> h = MinHeap()
>>> h.insert(1)
>>> h.insert(5)
>>> h.insert(2)
>>> h.insert(7)
>>> h.validate()
True
>>> h.pop_min()
1
>>> h.validate()
False  # Should be True!
```

Something seems to go wrong in `pop_min`. Trace out what the heap list should look like, and compare it to the actual list (using `h._items` or simply printing the heap using `print self` from within a heap method or `print my_heap` from outside the heap). Use your understanding of the heap algorithms and *Wing's* debugger to locate the bug and fix it. *Note*: the bug will be in one of the methods we gave you, since hopefully you have written `pop_min` correctly.

Add a case to the doctests for `pop_min` representing the case above (with `validate` always returning `True`) and run them to ensure that this case is always tested.

NOTE: Don't try the *Pop min revisited* question until you have `pop_min` working properly!!!

> ***Complete the* Pop min revisited *question(s) in Lab quiz 7.2.***

Max_3_Heap

Once you have got the binary min-heap working, you can move on to the dizzying heights of three-heaps. Skeleton code for the `Max_3_Heap` is provided in the `three_heap.py` module. Please note, we will be working on a maximum three-heap, ie, one where the maximum value is stored at the root of the heap and each child node must be smaller than or equal to its parent node. At this stage you need only:

1. implement `_sift_up` so that you can insert items in to the heap.
2. Try inserting values, and check that the structure remains correct.
3. Test it with the example 3-heap in Figure 3

Hint: After running the code below, `my_heap` should contain the heap shown in Figure 3 and `print(my_heap)` should show the correct heap. When answering the quiz you should include the None as the first item in the raw representation of the heap list.

```
my_heap = Max_3_Heap()
for item in [20,18,13,15,11,12,16,10,9,11,13,2,9,10,1]:
    my_heap.insert(item)
```

> ***Complete* 3-heap *questions in Lab quiz 7.2.***

NOTE: The deletion question is obviously based on the delete operation (see the extra exercise), but can be answered without implementing the code.

Slow heapify

As mentioned in the lectures heap sort has two steps, first make a heap by adding all the items to an empty heap and then repeatedly pop-min (or max) until the heap is empty. Optionally, as each item is popped it can be placed at the end of the heap list and the length of the heap reduced by 1 so that the value isn't considered to be part of the heap any more.

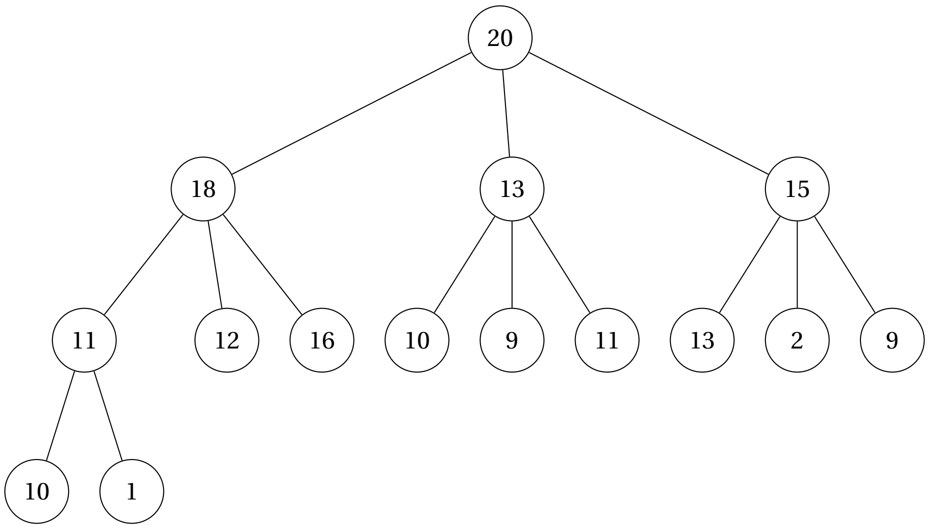

Figure 3: Three heap

Adding an item to a heap means that it will be appended to the heap list and then sifted up to get to its final resting place. The very first item that is added will created the root node of the whole heap (it initially goes at index 1 in the heap list, assuming the heap list starts at index 1). The second item added will go into index 2 of the heap list and then a sift-up is started from index 2. Third item will go at index 3 and then sifted up from there, etc... Rough pseudo code for this process follows:

```
for each item in the data:
    add/insert item into heap
```

If you think about it this boils down to the following:

```
set heap list to contain all the data in the original list
  (with a None inserted at the start if needed)
for i in range(start_index+1, len(heap_list)):
    sift up from index i
```

To add all of the values to the heap in this manner will use O(nlogn) comparisons in the average and worst case as many of the values will have to sift-up by up to $log_2 n$ levels. Remember that half of the items in a binary heap will added at the bottom level of the heap and therefore could have to sift-up $log_2 n$ levels. The next quarter of the values would be added at the level just above the bottom and therefore would sift-up by up to $log_2 n - 1$ levels, etc.

Note the same technique can obviously be used on heaps with bigger branch factors, ie, on 3-heaps, 4-heaps, etc... Such heaps will have fewer levels but will still end up taking O(nlogn) comparisons for the heapify.

> ***Complete the* Slow Heapify *question(s) in Lab quiz 7.2.***

Fast heapify

The slow heapify is an obvious way to make the heap that is needed for the second phase of the heap sort but it isn't the most efficient way to make the heap. It turns out that the heap can be built using a method that only needs O(n) comparisons. As this method is faster it is referred to as the *fast-heapify*. It works so well because it avoids the problem of having to do lots of sifting for lots of nodes. For example only one value will possibly need to sift through all the levels of the heap. Think about how you could turn a list into a heap using *sift-down* operations instead of sift up operations. See the lectures for more information on *fast-heapify*.

Note the same *fast-heapify* technique can obviously be used on heaps with bigger branch factors, ie, on 3-heaps, 4-heaps, etc... Such heaps will use more comparisons for sifting down a level but they will have a greater proportion of nodes in the bottom layer of the heap (which won't require any sifting down).

> ***Complete the* Fast Heapify *question(s) in Lab quiz 7.2.***

Extra Exercise

Once your have got the insertion to a 3-heap working, implement `_sift_down` for the 3-heap so that `pop_max` will work. Test that your implementation works with the 3-heap by checking to see that running `pop_max` on the heap given in Figure 3 returns the correct value and leaves the heap in the correct state.