Module 7 Heaps

Heaps are a special type of binary tree (complete binary tree), in which the value of any given node is larger than the value of its children (in a max heap), or smaller than the value of its children (in a min heap).

- Associated wiwth the heap is the priority queue ADT
- Time complexity of heap creation: nlog(n)/ heapify: n

Full and Complete Binary Trees

- Full Binary Trees
  - 每个node 都有两个/ 没有 子node
- Complete Binary Trees
  - 是一个Full Tree，但同时最底层所有子node都必须在tree的左边
  - 也就是说子node的左边空了，右边满了，这就不是一个complete tree
  - height will always be log(n) because we won't go down a level until all the nodes are full

Priority Queues and Heaps

The element with the **highest priority** is the first one dequeued. Typically, this corresponds to the element with the **lowest priority value**.

Heaps and Priority Queue Basics

The priority queue ADT has an interface that looks like this:

- `insert()` – insert an element with a specified priority value
- `first()` – return the element with the lowest priority value (the "first" element in the priority queue)

- `remove_first()` – remove (and return) the element with the lowest priority value

---

Heap Implementation

The left and right children of a node at index i are stored at indices `2 * i + 1` and `2 * i + 2`, respectively.

The parent of a node at index i is stored at `(i − 1) / 2` (using the floor that results from integer division).

Inserting Into A Heap

Inserting an element into the array representation of a heap follows this procedure:

1. Put the new element at the end of the array.
2. Compute the inserted element's parent index, `(i − 1) / 2`.
3. Compare the value of the inserted element with the value of its parent.
4. If the value of the parent is greater than the value of the inserted element, swap the elements in the array and repeat from step 2.
    - Do not repeat if the element has reached the beginning of the array.

Removing From A Heap

Accessing the minimum element in an array-based heap is easy; just return the value of the first element in the array.

Removing the minimum element is slightly more involved. It follows this procedure:

1. Remember the value of the first element in the array (to be returned later).
2. Replace the value of the first element in the array with the value of the last element, and remove the last element.
3. If the array is not empty (i.e., it started with more than one element), compute the indices of the children of the replacement element (`2 * i + 1` and `2 * i + 2`).

    - If both of these elements fall beyond the bounds of the array, we can stop here.
4. Compare the value of the replacement element with the minimum value of its two children (or possibly one child).
5. If the replacement element's value is greater than its minimum child's value, swap those two elements in the array, and repeat from step 3.

Heapsort

Given the heap and its operations described above, we can implement an efficient (O(n log n)), in-place sorting algorithm called heapsort.

The first thing heapsort does is build a heap out of the array using the procedure described above.

- Remember, as mentioned in the video, that we only have to "heapify" the non-leaf nodes
- Despite how it might look in code, building a heap is actually O(n)Links to an external site.

Then, to complete the sort, we use a procedure similar to our heap removal operation above, with a few small tweaks:

- Keep a running counter k that is initialized to one less than the size of the array (i.e., the last element).
- Instead of replacing the first element in the array (the min) with the last element (the kth element), we swap those two elements in the array.
    - The array itself remains the same size, and we decrement k.
- When percolating the replacement value down to its correct place in the array, we do not go beyond the end of the heap portion (see below).
    - Thus, the heap portion is effectively shrinking by 1 with each iteration.

We repeat this procedure until k reaches the beginning of the array.