1 Binary Heaps

Heaps are data structures that make it easy to find the element with the most extreme value in a collection of elements. A Min-Heap prioritizes the element with the smallest value, while a Max-Heap prioritizes the element with the largest value. Because of this property, heaps are often used to implement priority queues. As we saw in class, this is useful for Prim's algorithm (among many other things).

In this section, we will focus on *binary heaps*, specifically binary Max-Heaps. As the name suggests, a binary heap is a heap implemented as a complete binary tree, i.e. a binary tree in which all levels except possibly the bottom-most are completely filled, and the last level is filled from left to right. Its components are nodes of the tree, each with a key value, and it satisfies the following defining property:

Property (Max-Heap Property)**.** *For any node v in the heap, if u is any child of v, then $u \leq v$.*

You can find more about heaps by reading pages 151–169 in CLRS.

1.1 Heap Representation

While a heap can be represented as a complete binary tree, it is often more efficient to store a binary heap as an array. We call the first element in the heap element 1. Now, given an element i, we can find its left and right children with a little arithmetic:

Exercise.

- $\textsc{Parent}(i) =$
- $\textsc{Left}(i) =$
- $\textsc{Right}(i) =$

Solution.
The array representation is simplified by calling the first element in the heap 1:

- $\textsc{Parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\textsc{Left}(i) = 2i$
- $\textsc{Right}(i) = 2i + 1$

Completeness ensures that this representation of heaps is compact.

1.2 Heap operations

1.2.1 Max-Heapify

Max-Heapify(H, N): Given a complete binary tree H (represented as a list) and a node $N \in H$ for which the subtrees rooted at each of the children of N satisfy the Max-Heap property, this operation rearranges the tree rooted at N to also satisfy the Max-Heap property.

Max-Heapify(H, N):
Require: Left(N), Right(N) are each the root of a Max-Heap
  $(l, r) \leftarrow (\text{Left}(N), \text{Right}(N))$
  if Exists(l) and $H[l] > H[N]$ **then**
    $largest \leftarrow l$
  else
    $largest \leftarrow N$
  end if
  if Exists(r) and $H[r] > H[largest]$ **then**
    $largest \leftarrow r$
  end if
  if $largest \neq N$ **then**
    Swap($H[N], H[largest]$)
    Max-Heapify($H, largest$)
  end if
Ensure: N is the root of a Max-Heap

Exercise.

- *Run* Max-Heapify *with* $N = 1$ *on*

$$H = [14, 16, 10, 15, 7, 9, 6, 2, 4, 1]$$

- *What is* Max-Heapify*'s run-time?*

Solution.

- Max-Heapify(H, N) returns a Max-Heap.

$$[16, 14, 10, 15, 7, 9, 6, 2, 4, 1]$$

- Runs in $O(\log n)$ time because node N is moved down at most $\log n$ times.

1.2.2 Build-Heap

Build-Heap(A): Given an unordered array, makes it into a Max-Heap.

Build-Heap(A):
Require: A is an array.
  for $i = \lfloor$ length$(A)/2 \rfloor$ downto 1 **do**

MAX-HEAPIFY(A, i)
end for

Exercise.

- *Run* BUILD-HEAP *on* $A = [2, 1, 4, 3, 6, 5]$
- *Running time (loose upper bound):*
- *Running time (tight upper bound):*

Solution.

- $$[2, 1, 4, 3, 6, 5] \rightarrow$$
  $$[2, 1, 5, 3, 6, 4] \rightarrow$$
  $$[2, 6, 5, 3, 1, 4] \rightarrow$$
  $$[6, 3, 5, 2, 1, 4]$$
- $O(n \log n)$
- $O(n)$. Naively, BUILD-HEAP certainly takes at most $O(n \log n)$ time because MAX-HEAPIFY has time complexity $O(\log n)$ and is called $O(n)$ times, but intuitively, we should get a tighter bound because MAX-HEAPIFY is run more often at lower points on the tree, when subtrees are shallow. Making use of the fact that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h, the total number of comparisons that will have to be made is

  $$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h}\right)$$

  It is not hard to see that $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. So the runtime is $O(n)$.

1.2.3 Extract-Max

Extract-Max(H): Remove the element with the largest value from the heap.

EXTRACT-MAX(H):
Require: H is a non-empty MAX-HEAP
  $max \leftarrow H[root]$
  $H[root] \leftarrow H[\text{SIZE}(H)]$ {last element of the heap.}
  SIZE$(H) -= 1$
  MAX-HEAPIFY$(H, root)$
  return max

Exercise.

- *Run* EXTRACT-MAX *on* $H = [6, 3, 5, 2, 1, 4]$.

- *What is* Extract-Max*'s run time?*

Solution.

- Extract-Max first returns 6. It then moves the last element, 4, to the head (why the last element?) and Max-Heapify is used to maintain the heap structure:

$$[4, 3, 5, 2, 1] \to [5, 3, 4, 2, 1]$$

- $O(\log n)$

1.2.4 Insert

Insert(H, v): Add the value v to the heap H.

Insert(H, v):
Require: H is a Max-Heap, v is a new value.
  Size(H) $+= 1$
  $H[\text{Size}(H)] \leftarrow v$ {Set v to be in the next empty slot.}
  $N \leftarrow \text{Size}(H)$ {Keep track of the node currently containing v.}
  while N is not the root and $H[\text{Parent}(N)] < H[N]$ **do**
    Swap$(H[\text{Parent}(N)], H[N])$
    $N \leftarrow \text{Parent}(N)$
  end while

Exercise.

- *Run* Insert*(H, v) with $v = 8$ and*

$$H = [6, 3, 5, 2, 1, 4]$$

- *What is* Insert*'s runtime?*

Solution.

- Insert v into H as follows:

$$[6, 3, 5, 2, 1, 4, 8] \to$$
$$[6, 3, 8, 2, 1, 4, 5] \to$$
$$[8, 3, 6, 2, 1, 4, 5]$$

  Here, the while loop is halted by the condition that N is the root.

- The while loop takes time linear in the height of the tree, which is $O(\log n)$, so Insert's runtime is $O(\log n)$.

Exercise. *Suppose that we use a heap for sorting. What is the runtime? What's a disadvantage of this sorting method?*

Solution.
To sort using a heap, we can run Build-Heap once followed by Extract-Max until the heap is empty. The runtime of this is $O(n) + O(n \log n) = O(n \log n)$, the same as mergesort.

However, note that, unlike mergesort, heapsort is not stable.

Exercise. *Explain how to solve the following two problems using heaps.*

1. *Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.*
2. *Say that a list of numbers is k-*close *to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.*

Solution.
The solution of this problem is not included in this online copy of the section notes, but feel free to ask in section/office hours about how it works.

2 Minimum Spanning Trees

Exercise. *Prove that there is a unique minimum spanning tree on a connected undirected graph when the edge weights are distinct.*

Solution.
The solution of this problem is not included in this online copy of the section notes, but feel free to ask in section/office hours about how it works.

Exercise.

(CLRS 24.1-8)

Show that the list of edge weights for a minimal spanning tree is independent of the choice of such a tree.

Solution.
It suffices to show that we can walk from T to T' while preserving the MST property, since it is then clear that the lists of edge weights must match. For this it is enough to show that we can walk them both to an MST T'' while preserving the MST property. To do this, keep selecting the lightest edge e in $(T - T') \cup (T' - T)$. If $e \in T - T'$, make the exchange on T': Adding e to T' creates a cycle, and some edge e' of this cycle must be in $T' - T$; we remove e'. This makes T' lighter so it must again be an MST (which means e' and e had equal weight), and T, T' now have more edges in common than before. This process terminates when both T, T' reach the same tree T''.

Exercise. *Describe an efficient algorithm that, given an undirected graph G, determines a spanning tree of G whose largest edge weight ("bottleneck") is minimum over all spanning trees of G.*

Solution.
We are looking for a minimum bottleneck spanning tree, which we will abbreviate MBST. But the cut property holds for MBST's just as it does for MST's, so we can use the same greedy algorithms to construct the MBST. In fact, this shows that any MST is an MBST. However, the reverse inclusion is false.