CS 124 Section #2 **Heaps and Graphs** 2/6/17

1 Heaps

Heaps are data structures that make it easy to find the element with the most extreme value in a collection of elements. They are also known a priority queues because elements with higher priority are popped off first. 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. In section today, we will focus on the **binary max heap**.

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

1.1 What is a heap?

Max heaps are a very use data structure which maintains some structure on a list of numbers such that you can do the following operations. The goal is to be able to very quickly find the maximum element in a list, while supporting insertions and deletions to the list.

- PEEK(): what is the largest element in the list?
- EXTRACTMAX(): remove the largest element from the list
- INSERT(x): insert the element x into the list.

The goal is to have come up with a way to implement heaps such that all three of the above operations are fast. Heaps will be useful when we talk about Dijkstra's algorithm for shortest paths.

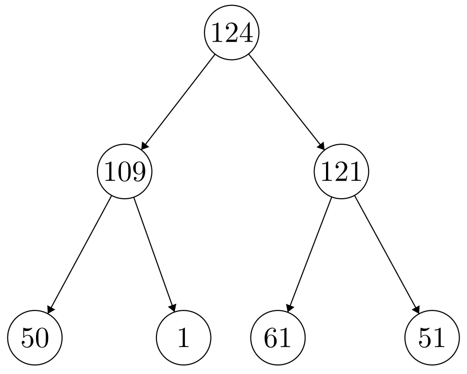
An efficient way to do this is to use a binary tree to store the list of numbers. This won't be an ordinary binary tree, but one that satisfies the **heap property**, which is:

If x is a parent of y in the binary tree, then $x > y$

1.2 Representing a Heap

One way to represent binary trees more easily is to use an 1-indexed array. For example, the following array will be used to represent the following max-heap

$$[124, 109, 121, 50, 1, 61, 51]$$



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 1. How would you find the index of the parent, left child and right child of the ith element in the heap?

- Parent$(i) =$
- Left$(i) =$
- Right$(i) =$

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

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

1.3 Heap operations

Before we look at how peek, insertion and deletion work, let's first implement some helper functions that will be useful for maintaining the heap structure and building a heap from scratch.

1.3.1 Max-Heapify

Max-Heapify(H, N): Given that the children of the node N in the Max-Heap H are each the root of a Max-Heap, rearranges the tree rooted at N to be a Max-Heap. We will call this when we break the heap property in order to fix it.

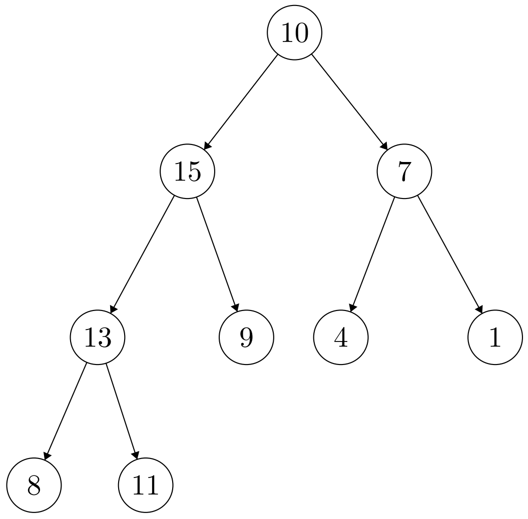
```
Max-Heapify$(H, N)$
Require: Left$(N)$, Right$(N)$ are each the root of a Max-Heap
 1: $(l, r) \leftarrow$ (Left$(N)$, Right$(N)$)
 2: if Exists$(l)$ and $H[l] > H[N]$ then
 3:     $largest \leftarrow l$
 4: else
 5:     $largest \leftarrow N$
 6: end if
 7: if Exists$(r)$ and $H[r] > H[largest]$ then
 8:     $largest \leftarrow r$
 9: end if
10: if $largest \neq N$ then
11:     Swap$(H[N], H[largest])$
12:     Max-Heapify$(H, largest)$
13: end if
Ensure: $N$ is the root of a Max-Heap
```

Description: First, we set *larger* to be the bigger of N and ℓ, the value of the node N or its left child. Then, we find the larger of that and the value of N's right child. We swap $H[$*larger*$]$ with the root so that now the root is bigger in value than the two children. We recursively call MAX-HEAPIFY on the side that we swapped with. If no swap occurred, we do not recurse.

Exercise 2.

- Run MAX-HEAPIFY with $N = 1$ on

$$H = [10, 15, 7, 13, 9, 4, 1, 8, 11]$$



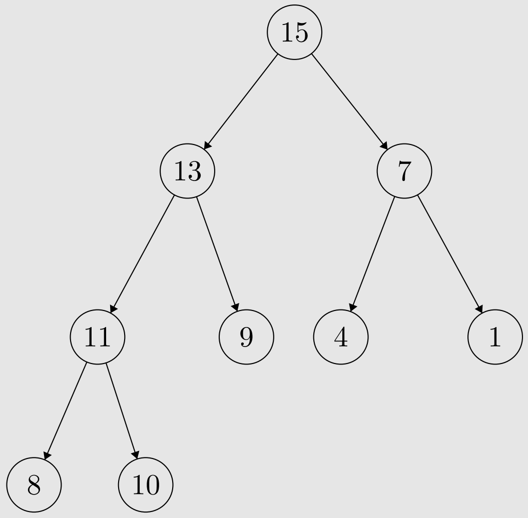
  Note that every level besides the one with $N = 1$ does satisfy the max-heap property. That is an important assumption when calling max-heapify.

- What is MAX-HEAPIFY's run-time?

Solution

- MAX-HEAPIFY(H, N) returns a max-heap

$$[15, 13, 7, 11, 9, 4, 1, 8, 10]$$



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

1.3.2 Build-Heap

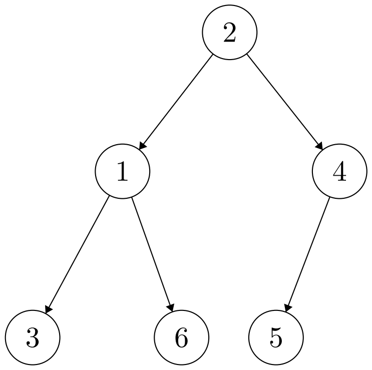
Build-Heap(A): Given an unordered array, makes it into a max-heap.

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

Description: Begin at the leaf nodes and call MAX-HEAPIFY so that gradually, the all the elements, starting at the bottom, will follow the heap property.

Exercise 3.

- Run Build-Heap on $A = [2, 1, 4, 3, 6, 5]$



- Running time (loose upper bound):
- Running time (tight upper bound):

Solution

-
$$[2, 1, 4, 3, 6, 5] \rightarrow$$
$$[2, 1, 5, 3, 6, 4] \rightarrow$$
$$[2, 6, 5, 3, 1, 4] \rightarrow$$
$$[6, 3, 5, 2, 1, 4]$$

- $O(n \log n)$ because we perform n operations of MAX-HEAPIFY and those all take $O(\log n)$ time each.

- $O(n)$. Intuitively, we might be able to 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

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

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

1.3.3 Peek

Exercise 4. If someone gave you a max heap H and wanted you to tell me the maximum element in H (without removing it), how would you do so? What is the run time?

Solution
You would simply return $H[0]$ because the root of the max heap is guaranteed to be the largest element. This runs in $O(1)$ time.

1.3.4 Extract-Max

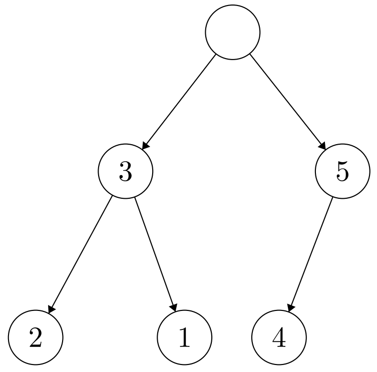
Extract-Max(H): Remove the element with the largest value from the heap and fix everything so that the heap structure is maintained.

```
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.}
  $\text{SIZE}(H) -= 1$
  $\text{MAX-HEAPIFY}(H, root)$
  return $max$
```

Description: Once we remove the max (the root), we need to replace it with something that's already in the tree. We choose to take one of the leaves and move it to the root and then max-heapify the heap.

Exercise 5.

- 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? it guaranteed to be a leaf. It would be much harder to pluck out a node with children.) and MAX-HEAPIFY is used to maintain the heap structure:

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

- $O(\log n)$ because we're performing just one max heapify operation.

1.3.5 Insert

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

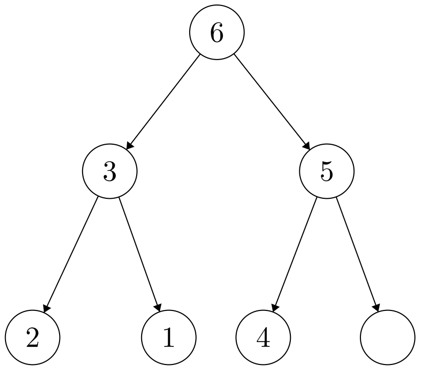
```
INSERT($H, v$)
Require: $H$ is a MAX-HEAP, $v$ is a new value.
  SIZE($H$) $+= 1$
  $H$[SIZE($H$)] $\leftarrow v$ {Set $v$ to be in the next empty slot.}
  $N \leftarrow$ SIZE($H$) {Keep track of the node currently containing $v$.}
  while $N$ is not the root and $H$[PARENT($N$)] $< H[N]$ do
    SWAP($H$[PARENT($N$)], $H[N]$)
    $N \leftarrow$ PARENT($N$)
  end while
```

Description: Add the node to the leaf and then promote it upwards until it is smaller than its parent.

Exercise 6.

- 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:

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

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

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

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

- INSERT's runtime is $O(\log n)$ because you do the while loop at most $\log n$ times.

Exercise 7. Why might you use a heap over a binary search tree with a pointer to the smallest element?

Solution
Even though the run-times are the same, the binary heap is *self-balancing*, meaning that the last level of the tree is always the only one to not be completely filled. Of course there are balanced binary search trees as well, but can any be implemented as easily as the above?

1.4 Relation to Dijkstra's Algorithm

In section today, we talked about max heaps, but in Dijkstra's algorithm, we use a min heap. Everything works in the same way except the heap invariant get flipped so that every parent is smaller than all its children. In Dijkstra's algorithm, the heap we use stores pairs (v, n) with $v \in V$ and $n \in \mathbb{N}$, where we build the heap based on the n's, but keep the v's as identification as to which vertex this heap element represents. Dijkstra's algorithm heaps require the following operations where H is the heap:

Operation	Run-time	**Description**	**Use in Dijkstra's**
DELETEMIN(H)	$O(\log n)$	pop the minimum element u off the heap	finish processing u, the closest vertex in the heap to the source, and want to insert its unvisited neighbors
INSERT(H, v, n)	$O(\log n)$	insert a new vertex into H with priority n	after popping u off the heap, we add all of u's unvisited neighbors v
DECREASE-KEY (H, v, n)	$O(\log n)$	decrease the key of v from whatever it is to n	we have found a shorter path to v

Remember that the run-time of Dijkstra's algorithm is $\boxed{O(|V| \cdot \text{DELETEMIN} + |E| \cdot \text{INSERT})}$

2 Depth First Search

In lecture, we talked how to represent graphs on a computer, as well as talked about DFS, BFS and Dijkstra's algorithm and their various applications. We will go more in depth into the shortest path algorithms in the next section.

2.1 Important Properties

- DFS is a systematic way to traverse a graph, touching all of the vertices of the graph once.
- The run-time is $O(|V| + |E|)$.
- DFS produces tree edges, forward edges, back edges and cross edges. **Important:** The classification of an edge is entirely based on how the DFS is run. An edge can be a tree edge in one iteration of DFS and a back edge in another.
- DFS also produces pre and post-order numbers that tell you when a node was first touched and when it has been done searching all that node's neighbors.
- DFS can be used for detecting cycles, finding topological sorts and determining the strongly connected components of a graph.

2.2 Exercises

Exercise 8. George goes to the ice cream shop which has n different ice cream flavors. He goes in with a list with m statements of the form "I prefer flavor X to flavor Y". Your job is to determine if this list is consistent with itself. This means that if George prefers X to Y and Y to Z, then he must prefer X to Z as well. This extends to having more than 3 flavors as well. Design an efficient algorithm for determining if George's list is consistent with itself and state its runtime. (Hint: create a graph)

Solution
We can create a graph where each of the ice cream flavors are a vertex and each of the m pairs (X, Y) is an edge from flavor X to flavor Y. George's list is inconsistent if this graph contains any cycles. Therefore, we can do a simple DFS of the graph in $O(m + n)$ time to determine if such a cycle exists.

Exercise 9. As in the exercise above, George goes back to the ice cream store with a list of m statements of the form "I prefer flavor X to flavor Y". This time, we're guaranteed that his list is consistent, which means that George has some internal ranking of the n flavors. Suppose that m is large enough that only 1 ranking of the n flavors satisfy the m statements. Design an efficient algorithm for finding George's internal ranking of the n flavors.

Solution
If we were to draw out the graph again like in the previous problem, we should find a directed acyclic graph. To find George's preferences, we perform a topological sort of these nodes (which is possible because of the directed acyclic graph). Once this is done, all we have to do is read off the nodes in the topological sorted order and those will be George's preferences (in either ascending or descending order).

Exercise 10. Answer T/F for the following problems:

(a) We know that the node with the highest post-order belongs to a source SCC. Then the node with the lowest post-order always belongs to a sink SCC.

(b) Suppose two vertices u and v in a directed graph satisfy $pre(u) < post(u) < pre(v) < post(v)$, then there can be no edge in either direction between u and v.

(c) In a DFS of a directed graph G, the set of vertices reachable from the vertex with lowest post-order is a strongly-connected component of G.

(d) In a DFS of a directed graph G, the set of vertices reachable from the vertex with highest post-order is a strongly-connected component of G.

(e) If a DFS has a cross edge, the graph is not strongly connected.

Solution

(a) **False**. Consider G with vertices a, b, c and edges $(a, b), (b, a), (a, c)$. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. Although b has the lowest post-order, it's not part of a sink SCC.

(b) **False**. Consider G with vertices a, b and edge (b, a). DFS with $a : [1, 2], b : [3, 4]$.

(c) **False**. Consider G with vertices a, b, c and edges $(a, b), (b, a), (a, c)$. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. a and c are reachable from b, but $\{a, b, c\}$ are not an SCC.

(d) **False**. Consider G with vertices a, b, c and edges $(a, b), (c, a), (c, b)$. DFS with $a : [1, 4], b : [2, 3], c : [5, 6]$. c has the highest postorder and both a, b are reachable from c. But G is not strongly connected.

(e) **False**. Consider G with vertices a, b, c and edges $(a, b), (b, a), (a, c), (c, b)$. G is strongly connected. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. Then (c, b) is a cross-edge.