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 as 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**; although it is most commonly visualized as a binary tree, we will show how to implement it using index manipulation on an array.

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

1.1 What is a heap?

Max heaps are a very useful 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?
- **DELETEMAX()**: remove the largest element from the list
- **INSERT(v)**: insert the element v into the list.
- **INCREASEKEY(N, v)**: increase the value of node N to have value v.

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 $i$th 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 MaxHeapify

**MaxHeapify**($H$, $N$): 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.

**Description:** First, we set $larger$ to be the bigger of $N$ and $\ell$, 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 MaxHeapify on the side that we swapped with. If no swap occurred, we do not recurse.

**Exercise 2.**

- Run MaxHeapify with $N = 1$ on $H = [10, 15, 7, 13, 9, 4, 1, 8, 11]$
MaxHeapify($H, N$)

**Require:** LEFT($N$), RIGHT($N$) are each the root of a Max-Heap

1: $(l, r) \leftarrow (\text{LEFT}(N), \text{RIGHT}(N))$
2: if EXITS($l$) and $H[l] > H[N]$ then
3: $\text{largest} \leftarrow l$
4: else
5: $\text{largest} \leftarrow N$
6: end if
7: if EXITS($r$) and $H[r] > H[\text{largest}]$ then
8: $\text{largest} \leftarrow r$
9: end if
10: if $\text{largest} \neq N$ then
11: $\text{SWAP}(H[N], H[\text{largest}])$
12: $\text{MAXHEAPIFY}(H, \text{largest})$
13: end if

**Ensure:** $N$ is the root of a Max-Heap

---

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

- What is MAXHEAPIFY’s run-time?

**Solution**

- MAXHEAPIFY($H, N$) 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 BuildHeap

**BuildHeap**($A$): Given an unordered array, makes it into a max-heap.

```plaintext
BuildHeap(A)

Require: $A$ is an array.
for $i = \lfloor \text{length}(A)/2 \rfloor$ downto 1 do
  MAXHEAPIFY($A, i$)
end for
```

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

1.3.3 Peek

**Exercise 3.** 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 DeleteMax

**DeleteMax**($H$): Remove the element with the largest value from the heap and fix everything so that the heap structure is maintained.

**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 MaxHeapify the heap.
DeleteMax($H$)

Require: $H$ is a non-empty Max-Heap

$\text{max} \leftarrow H[\text{root}]$

$H[\text{root}] \leftarrow H[\text{SIZE}(H)]$ \{last element of the heap.\}

\text{SIZE}(H) = 1

\text{MAXHEAPIFY}(H, \text{root})

\text{return} \ max

Exercise 4.

- Run \text{DELETEMAX} on $H = [6, 3, 5, 2, 1, 4]$.
- What is \text{DELETEMAX}'s run time?

\begin{tabular}{c}
\begin{tikzpicture}
  \node (root) {5} [grow=up, text=white]
  \node (left) {3} [grow=south, text=white, below=of root]
  \node (right) {4} [grow=south, text=white, right=of left]
  \node (bottom-left) {2} [grow=east, text=white, below=of left]
  \node (bottom-right) {1} [grow=east, text=white, below=of right]

  \draw (root) -- (left);
  \draw (root) -- (right);
  \draw (left) -- (bottom-left);
  \draw (right) -- (bottom-right);
\end{tikzpicture}
\end{tabular}

Solution

- \text{DELETEMAX} 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 \text{MAXHEAPIFY} 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

\text{Insert}($H, v$): Add the value $v$ to the heap $H$.

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

Exercise 5.

- Run \text{INSERT}($H, v$) with $v = 8$ and $H = [6, 3, 5, 2, 1, 4]$
**Insert**($H, v$)

**Require:** $H$ is a Max-Heap, $v$ is a new value.

* $	ext{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**
* $\text{Swap}(H[\text{Parent}(N)], H[N])$
* $N \leftarrow \text{Parent}(N)$
**end while**

- 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 6.** 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?

**1.3.6 IncreaseKey**

IncreaseKey($H, N, v$): Increase the value of node $N$ to a new, higher value $v$ in the heap $H$.

**Exercise 7.**
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. The ice cream store George frequents wants to use a list of words to put up in a banner. However, they want to order the words such that they overlap first and last letters, so if word \( s \) comes before string \( t \), the last letter of \( s \) must be the same as the first letter of \( t \). Otherwise, the ordering does not matter, but the words must wrap into a loop. For example, \{EPIC, CHOCOLATE\} is a set of words that can be formed into a loop, but \{VANILLA, AWESOME\} is not. How can George algorithmically tell whether such an ordering is possible?

Solution
We construct a directed graph where the nodes are letters of the alphabet, and an edge is added for every word, from the first letter of the word to the last letter of the word.

After deleting all nodes with no edges, we need to only check that the remaining graph is strongly connected, and that each remaining node has the same in-degree as out-degree. We can run Kosaraju’s Algorithm (think DFS) to check for one strongly connected component, and can iterate through all of the edges and keep a hash for the in and out-degree for each node.

Exercise 11. 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 \( \text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \), 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.