CS 124 Section 1
Binary Min/Max Heaps

Yash Nair

February 2021
What is a Heap and Why do we Use Them?
  - Heap Basics
  - Representing a Heap

Implementing Heap Operations

Problems
Purpose of Heaps

- Data structure to find (and also remove) most extreme (i.e., maximal or minimal) element in a set we can add to quickly
- **MAX-HEAP** finds maximal element, **MIN-HEAP** finds minimal
- We will consider **MAX-HEAP**, but note that a **MIN-HEAP** is just a **MAX-HEAP** if we multiply the elements by $-1$
Using a Sorted Array is Slower than a Heap

- Do the following operations using a sorted array: insert 5, insert 3, insert 2, find and remove max, find and remove max, insert 1
## Comparing Operation Times

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Add Element</th>
<th>Find Max</th>
<th>Find and Remove Max</th>
<th>Build from Unsorted Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Unsorted Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Heap Operations Overview

- **PEEK()**: what is the largest element in the heap?
- **EXTRACTMAX()**: remove the largest element from the heap
- **INSERT(x)**: insert the element $x$ into the heap
- **BUILD-HEAP(A)**: given an unsorted array, $A$, build a binary heap

Want to be able to do these operations quickly by using a binary tree structure.
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. 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 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?
- **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.

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, 110, 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:

```
i - 1 \quad \text{left child}
i + 1 \quad \text{right child}
```

```
109
121
```

```
50
1
110
51
```

[Diagram of the binary max heap with the array `[124, 109, 121, 50, 1, 110, 51]` and corresponding tree structure]
Can order elements of left-aligned binary tree and view them in array.

Letting the first element (i.e., root) of tree have index 1, what are the following:

- PARENT($i$) =
- LEFT($i$) =
- RIGHT($i$) =
Heap Representation

- Heap property: If \( x \) is a parent of \( y \) in the binary tree, then \( x > y \)
- A heap is a binary tree that satisfies the heap property

\[
[124, 109, 121, 50, 1, 110, 51]
\]
Max-Heapify($H, N$)

- **Input:** $N$ is node in the left-aligned binary tree $H$ such that the $N$ is the root of a MAX-HEAP, except that $N$ may be smaller than its children (i.e., but all lower layers satisfy the heap property)
- **Goal:** Rearrange nodes so that tree rooted at $N$ is a MAX-HEAP

Max-Heapify($H, N$)

**Require:** $N$ is the root of a MAX-HEAP except, possibly, at $N$ (i.e., $N$ could be smaller than its children)

```
1:  (l, r) ← (LEFT($N$), RIGHT($N$))
2:  if EXISTS(l) and $H[l] > H[N]$ then
3:      largest ← l
4:  else
5:      largest ← $N$
6:  end if
7:  if EXISTS(r) and $H[r] > H[largest]$ then
8:      largest ← r
9:  end if
10: if largest ≠ $N$ then
11:    SWAP($H[N], H[largest]$)
12:    MAX-HEAPIFY($H, largest$)
13: end if

**Ensure:** $N$ is the root of a MAX-HEAP
Example

Run Max-Heapify with $N = 1$ on the below graph. What is the runtime?

$$H = [10, 15, 7, 13, 9, 4, 1, 8, 6]$$
Runtime of Max-Heapify

- $O(\log n)$ comparisons are made
- $O(\log n)$ swaps are made
- Total runtime: $O(\log n)$. 
Algorithm 2 Build-Heap(A)

Require: A is an array.
for i = ⌈length(A)/2⌉ down to 1 do
    Max-Heapify(A, i)
end for

Idea: Start from last node with child and work up

Run Build-Heap on:

```
2
   1 4
  3 6 5
```
Runtime of Build-Heap

- Naive runtime analysis: $O(n)$ calls to Max-Heapify, each takes $O(\log n)$ time, so $O(n \log n)$
- Tighter analysis:
  - When we run Max-Heapify from node of height $h$, runtime is $O(h)$
  - Build-Heap does this for heights $h = 0, \ldots, \lceil \log n \rceil$
  - At most $\lceil n/2^{h+1} \rceil$ nodes at height $h$ (Proof: induction on $h$. BC: $h = 0$; there are at most $\lceil n/2 \rceil$ nodes at height 0. IS: remove last layer, then new tree has at most $n - \lceil n/2 \rceil$ nodes and height $h$ in original tree is now height $h - 1$, so, by IH, at most $(n - \lceil n/2 \rceil)/2^h \leq \lceil n/2^{h+1} \rceil$)
  - Gives runtime bound of
    \[
    \sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\log n} \frac{h}{2^h} \right)
    \]
    
    \[
    \sum_{h=0}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{1.5^h}{2^h} = \sum_{h=0}^{\infty} \left( \frac{3}{4} \right)^h = 4,
    \]
    so runtime is $O(n)$
Q: How to return the maximum element of MAX-HEAP, \( H \), quickly?

A: Return \( H[0] \) (i.e., the root)! Guaranteed to be maximal by Build-Heap.
Extract-Max

Remove largest element of heap, and ensure that heap structure is maintained:

**Algorithm 3 **

**Extract-Max**($H$)

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

1. $max \leftarrow H[root]$
2. $H[root] \leftarrow H[\text{SIZE}(H)]$ \{last element of the heap,\}
3. $\text{SIZE}(H) -= 1$
4. $\text{Max-Heapify}(H, root)$
5. **return** $max$

---

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 is guaranteed to be a leaf. It would be much harder to pluck out a node with children), and then **Max-Heapify** is used to maintain the heap structure:

  
  ![Heap Diagram]

  

  

- **O**($\log n$) because we’re performing just one **Max-Heapify** operation.

---

Exercise (6).

- Run **Insert**($H, v$) with $v=8$ and $H=[6, 3, 5, 2, 1, 4]$.

- 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 is guaranteed to be a leaf. It would be much harder to pluck out a node with children), and then **Max-Heapify** is used to maintain the heap structure:

  
  ![Heap Diagram]

  

  

- **O**($\log n$) because we’re performing just one **Max-Heapify** operation.
Runtime of Extract-Max

that an \( n \)-element heap has height \( \log n \) and at most \( \frac{d}{2}n^{h+1} \) nodes of any height \( h \), the total number of comparisons that will have to be made is

\[
\sum_{h=0}^{\log n} \frac{d}{2}n^{h+1}
\]

It is not hard to see that \( P \) converges 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(\( H \)):

\[
\begin{align*}
&{\text{Require: } H \text{ is a non-empty MAX-HEAP}} \\
&\text{max} \leftarrow H[root] \\
&H[root] \leftarrow H[\text{Size}(H)] \quad \{\text{last element of the heap.}\} \\
&\text{Size}(H) -= 1 \\
&\text{MAX-HEAPIFY}(H, root) \\
&\text{return } max
\end{align*}
\]

Since we only perform one Max-Heapify: \( O(\log n) \)
Insert

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

**Algorithm 4** **Insert**\((H, v)\)

**Require:** \(H\) is a Max-Heap, \(v\) is a new value.

\[
\text{SIZE}(H) += 1 \\
H[\text{SIZE}(H)] \leftarrow v \quad \{\text{Set } v \text{ to be in the next empty slot.}\} \\
N \leftarrow \text{SIZE}(H) \quad \{\text{Keep track of the node currently containing } v.\} \\
\text{while } N \text{ is not the root and } H[\text{PARENT}(N)] < H[N] \text{ do} \\
\quad \text{SWAP}(H[\text{PARENT}(N)], H[N]) \\
\quad N \leftarrow \text{PARENT}(N) \\
\text{end while}
\]

Solution. • Extract-Max first returns 6. It then moves the last element, 4, to the head (Why the last element? It is guaranteed to be a leaf. It would be much harder to pluck out a node with children), and then **Max-Heapify** is used to maintain the heap structure:

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

• \(O(\log n)\) because we’re performing just one **Max-Heapify** operation.

Exercise (6).

• Run **Insert**\((H, v)\) with \(v = 8\) and \(H = [6, 3, 5, 2, 1, 4]\).
Example

Run $\text{Insert}(H, v)$ with $v = 8$ on the heap below:

Solution. Insert $v$ into $H$ as follows:

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

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.
### Runtime of Insert

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

**Algorithm 4** INSERT($H, v$)

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

1. $\text{SIZE}(H) += 1$
2. $H[\text{SIZE}(H)] \leftarrow v$ \{Set $v$ to be in the next empty slot.\}
3. $N \leftarrow \text{SIZE}(H)$ \{Keep track of the node currently containing $v$.\}
4. **while** $N$ is not the root and $H[\text{PARENT}(N)] < H[N]$ **do**
   1. Swap($H[\text{PARENT}(N)], H[N]$)
   2. $N \leftarrow \text{PARENT}(N)$
5. **end while**

- While loop occurs $O(\log n)$ times
- Both steps on while loop take constant time
- Total runtime: $O(\log n)$
Recap

- **MAX-HEAP** allows us to find the maximum of a collection of elements quickly.
- Max-Heapify, which ensures heap property, takes $O(\log n)$ time.
- Build-Heap, which builds a MAX-HEAP give collection of numbers using Max-Heapify, takes $O(n)$ time.
- Peek, which returns maximal element in heap, takes $O(1)$ time by returning the root.
- Extract-Max, which removes the maximal element and then Max-Heapifies to maintain the heap structure, takes $O(\log n)$ time.
- Insert, which promotes the added node up until it is smaller than its parent to ensure heap structure, takes $O(\log n)$ time.
0. Hand-Running Heap Operations

Start with the empty heap and perform the following operations: Insert(3), Insert(1), Insert(4), Insert(1), Extract-Max, Insert(5), Insert(9), Extract-Max, Insert(2), Extract-Max, Extract-Max, Insert(6). What does the resulting heap look like?
1. Iterative Max-Heapify (CLRS 6.2-5)

Recall the pseudocode for the recursive implementation of Max-Heapify below. Write an iterative version, taking the same amount of time. In particular, your iterative version may not call the original Max-Heapify algorithm nor itself.

Max-Heapify($H, N$)

Require: $N$ is the root of a MAX-HEAP except, possibly, at $N$ (i.e., $N$ could be smaller than its children)
1: $(l, r) \leftarrow (\text{LEFT}(N), \text{RIGHT}(N))$
2: if EXISTS($l$) and $H[l] > H[N]$ then
3:   $\text{largest} \leftarrow l$
4: else
5:   $\text{largest} \leftarrow N$
6: end if
7: if EXISTS($r$) and $H[r] > H[\text{largest}]$ then
8:   $\text{largest} \leftarrow r$
9: end if
10: if $\text{largest} \neq N$ then
11:   SWAP($H[N], H[\text{largest}]$)
12: end if
Ensure: $N$ is the root of a MAX-HEAP
2. Tightness of Runtime Analyses

a.) Show that the $O(\log n)$ upper-bound on Max-Heapify’s runtime is tight. In other words, construct an infinite family of arrays, $\{A_n\}_{n=1}^{\infty}$, with $A_n$ a MAX-HEAP except, possibly, at the root (i.e., the root may be smaller than at least one of its children), and length($A_n$) = $n$ for all $n$, so that the running time, $T(n)$, of Max-Heapify($A_n$, 1) is $\Omega(\log n)$.

b.) Let $\{A_n\}_{n=1}^{\infty}$ be any infinite family of arrays with length($A_n$) = $n$. Prove that the running time, $T(n)$, of Build-Heap($A_n$) is $\Theta(n)$. 
3. Sorting with a MAX-HEAP

Consider the following sorting algorithm: run Build-Heap, then repeatedly Extract-Max and append to a sorted list. How efficient is this algorithm? Give a *tight* bound on the running time.