CS 124 Section 1

Binary Min/Max Heaps

Yash Nair

February 2021

Table of Contents

- 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

Data Structure	Add Element	Find Max	Find and Remove Max	Build from Unsorted Array
Sorted Array	$O(n)$	$O(1)$	$O(1)$	$O(n \log n)$
Unsorted Array	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n)$

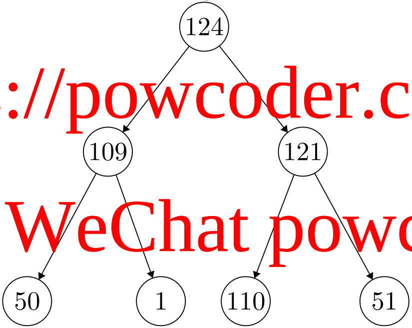
Heap Operations Overview

- 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.

Binary Trees as Arrays Example

$[124, 109, 121, 50, 1, 110, 51]$



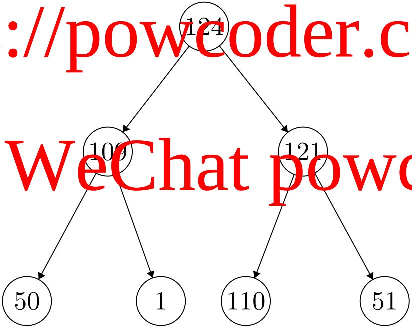
Binary Trees as Arrays

- 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:
    - $\texttt{PARENT}(i) =$
    - $\texttt{LEFT}(i) =$
    - $\texttt{RIGHT}(i) =$

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)

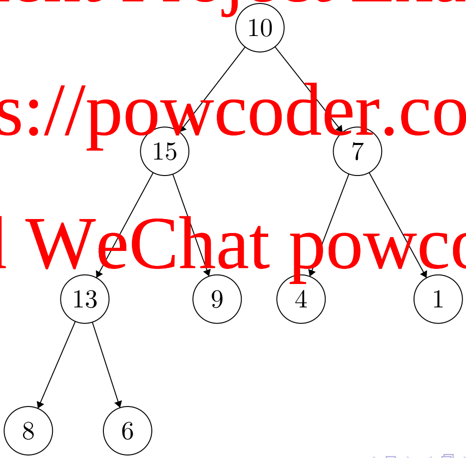
```
(l, r) ← (Left(N), Right(N))
if Exists(l) and H[l] > H[N] then
    largest ← l
else
    largest ← N
end if
if Exists(r) and H[r] > H[largest] then
    largest ← r
end if
if largest ≠ N then
    Swap(H[N], H[largest])
    Max-Heapify(H, largest)
end if
```

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: $O(\log n)$.

Building a Heap

Algorithm 2 Build-Heap(A)

Require: A is an array.

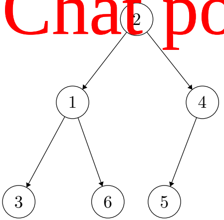
for $i = \lfloor \text{length}(A)/2 \rfloor$ down to 1 **do**

  Max-Heapify(A, i)

end for

Idea: Start from last node with child and work up

Run Build-Heap on:



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 $h = 0, \ldots, \lfloor \log n \rfloor$
  - 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

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

  - $$\sum_{h=0}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{1.5^h}{2^h} = \sum_{h=0}^{\infty} \left(\frac{3}{4} \right)^h = 4,$$

    so runtime is $O(n)$

Peek

- 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

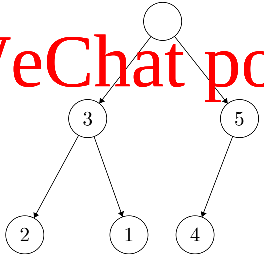
$max \leftarrow H[root]$

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

$\text{SIZE}(H) -= 1$

MAX-HEAPIFY(H, $root$)

return max



Runtime of Extract-Max

Algorithm 3 *

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.}

$\text{SIZE}(H) -= 1$

MAX-HEAPIFY($H, root$)

return max

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

Insert

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

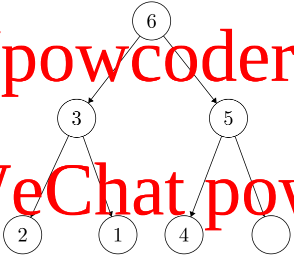
Algorithm 4 INSERT(H, v)

Require: H is a MAX-HEAP, v is a new value.

  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

Example

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



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.

```
Size(H) += 1
H[Size(H)] ← v {Set v to be in the next empty slot.}
N ← 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 ← Parent(N)
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

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)


```
(l, r) ← (LEFT(N), RIGHT(N))
if EXISTS(l) and H[l] > H[N] then
    largest ← l
else
    largest ← N
end if
if EXISTS(r) and H[r] > H[largest] then
    largest ← r
end if
if largest ≠ N then
    SWAP(H[N], H[largest])
    MAX-HEAPIFY(H, largest)
end if
```

Ensure: N is the root of a MAX-HEAP 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.