CS234 MODULE 12 - HEAPS

- Min Heap
- Max Heap
- Uses

Last Updated 07-31
Complete Trees

A Heap is a kind of complete binary tree.

We covered those briefly last module.

Reminder: It’s a binary tree where every layer has the maximum number of nodes, except (optionally) the last layer. If the last layer is not complete, all of its nodes are as far left as possible.
Complete Tree Examples (again)

Complete

```
  6
  / \   
 4   8
 / \ / \ 
2   5 7 9
```

Not Complete

```
  15
   /   
  4   23
     /    
  8   16
   /     
 16   42
```
Why Does Completeness Matter?

A complete tree can be stored in an array **efficiently**. It is stored in layer-order form.
The Math

If there are $n$ nodes in the complete tree, the array only needs length $n$. No waste.

The root is at index 0.

If a node is at index $i$, then its children are at $2i+1$ and $2i+2$.
The Min-heap

A Min-heap is a complete binary tree with one extra rule: The **min-heap property**

A node must be less than **both** of its children.
The Min-heap

Unlike a BST, “left” and “right” have no special meaning.

The above are both valid.
What’s good about a Min Heap?

The root is the smallest value. There’s also a Max Heap, where the largest value is the root.

That’s handy, if the data structure can be maintained when adding and removing values.

Let’s see how, then:
Inserting into a Min-heap

To insert into a min-heap: There’s only one place a new node can go. At the end of the bottom layer.

This might violate the **heap property** (and does in this example)
Fixing The Violation

If a node has a value less than its parent, but otherwise the tree is a valid min-heap:

Can we switch C and A? We don’t know!
C < D, and C < A, but that doesn’t mean A < D.
Starting from the Bottom

11 and 23 can switch because: A < B < C, so if the new node Less than its parent, it must be less than its other sibling!

Since it has no children, no Need to worry about them.
Done
Repeat?

If we swap the new value with its parent: It could also be less than its new parent!
Repeat

If we repeat the process: It’s still OK.
Why?

This time: If we swap a small child with a large parent, we moved a big thing down.

Can it be bigger than its new children?

NO! Because they used to be its children, before we started swapping. It was OK before, it is OK now.
Final Swap

Before Swap:

```
        4
       / \
      11  8
     /   / \
    15  42 23
   /           \
  16            16
```

After Swap:

```
        1
       / \ 
      11  8
     /   / \
    15  42 23
   /           \
  16            16
```
What’s That Called?

This process is usually called “Sifting Up” or “Bubbling Up” or “Heapifying Up”

The run-time is $O(\log n)$ –
It always starts at the bottom, and there are $\lg(n)$ layers
Removing From a Min Heap

Usually you use a min heap because you want to remove the smallest value.

To do this: Fill the root with the **last** thing (that’s the only node that can be removed).
Removing the Min Value

Before:

1
/   /
4   11
/   /
8   42
/   /
16  23

After:

1

1
/  
4   16
/   /
8   11
/   /
15  42
/   
23
Fixing the Violation

That 16 isn’t allowed there, it’s greater than both children.

Solution: Switch it with the smaller of its two children. Repeat until it’s in an allowed spot (this might be the bottom again).

This called “Sifting Down” (or “Bubbling Down”, “Heapifying Down”)
Sifting Down
Sifting Down – Runtime Complexity

Each layer we examine two child nodes.

\[ 2 \times \lg(n) = O(\log n) \]
Heapsort

We can use a Heap to sort an array:

Insert all $n$ values into a Min Heap: $O(n \log n)$

Extract the smallest value $n$ times and put back into the array: $O(n \log n)$

That’s good! However, the constant is higher than for merge sort.
Heapsort, In Place

A heap can be stored in an array. Just like with insertion sort, we can use part of the array for the heap, and part for the sorted values. This time, we’ll use a Max Heap.
Heapsort

Legend:  HEAP, UNSORTED, SORTED

[8,6,7,5,3,0,9]

Insert 6

[8,6,7,5,3,0,9]

Insert 7

[8,6,7,5,3,0,9]
Heapsort

Legend:  HEAP, UNSORTED, SORTED

\([8,6,7,5,3,0,9]\)

Insert 5, 3, and 0 (still no violations!)

\([8,6,7,5,3,0,9]\)
Heapsort

Legend: HEAP, UNSORTED, SORTED

[8,6,7,5,3,0,9]

Insert 9 (violation)

[8,6,7,5,3,0,9]

Sift Up (past 7 and 8)

[9,6,8,5,3,0,7]
Heapsort

Now we have heapified the input. We can now begin extracting the largest value.

Since this deletes the last node it frees up the last element of the array.

That’s where the largest value belongs! How convenient!
Remove Max

- Remove 9 (i.e. Swap 9 and 7)
  \[7,6,8,5,3,0,9\]

- Sift Down on 7
  \[8,6,7,5,3,0,9\]
Repeat

\[8,6,7,5,3,0,9\] Extract 8, Sift down 0 (past 7)

\[7,6,0,5,3,8,9\] Extract 7, Sift down 3 (past 6 and 5)

\[6,5,0,3,7,8,9\] Extract 6, Sift down 3 (past 5)

\[5,3,0,6,7,8,9\] Extract 5, Sift down 0 (past 3)

\[3,0,5,6,7,8,9\] Extract 3

\[0,3,5,6,7,8,9\] (0 is placed in the correct spot by this swap)
Bottom Up Heapify

Instead of repeatedly inserting values into an empty heap (a Top Down approach) you can pretend the array is already a heap, and then Sift Down to fix violations of the heap property.

```
[1, 0, 4, 5, 3, 2, 9]
```
Sift Down, Bottom Up

Diagram of a tree transformation showing the process of sift down and sift up operations in a search tree.
Is That Better?

What’s the runtime?

Cannot sift down leaves, so we only consider the bottom most internal nodes.

How many are there? About n/4. They sift down in 1 step

How many above that layer? n/8. They sift down in 2 step.

Above that? n/16. They sift in 3 steps.
Maximum Number of Sift Steps

\[ 1 \times \frac{n}{4} + 2 \times \frac{n}{8} + 3 \times \frac{n}{16} + \ldots + h \times \frac{1}{1} = \]

\[ = \frac{n}{4} \sum_{k=1}^{h} \left( \frac{k}{2^{k-1}} \right) < \frac{n}{4} \sum_{k=1}^{\infty} \left( \frac{k}{2^{k-1}} \right) = n \]

Calculus was involved, but I spared you. You’re welcome
Heapsort Video

- https://youtu.be/_bkow6lykGM
What Other Use?

A Min Heap can be used to implement a Priority Queue if: You use a compound key.

Have the Priority Queue keep track of the total number of events (ever, not just its current length!)

Compare keys based on priority first, then timestamp if there is a tie.
Priority Queue Example

• In Class