multiple choice, true false, fill in the blanket (brute force quiz tuesday)

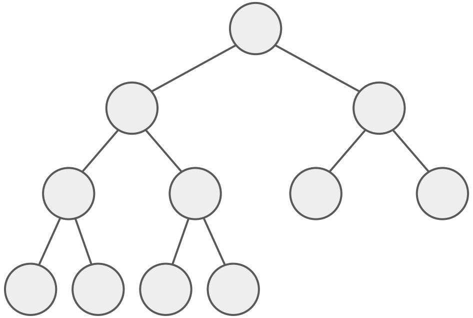
Heaps

With Applications in Sorting and Priority Queues

Heaps

shape of heap exclusively depends on the number of elements stored it. every heap with 11 elements for example has this shape

A heap is a data structure with many applications in computer science. We will be imagining heaps as tree based structures, and implementing them using arrays.



Two important things about heaps:

very strict about shape

- They are a nearly complete tree.
  (meaning except for the bottom level, always complete otherwise)
- They have the "heap property," either min or max, which indicates how the heap is partially ordered.

In a nearly complete tree every level except the lowest on are always filled. The lowest level is always filled from left-to-right. New nodes are always added in the lowest level, in the left-most "open" spot.

Heaps

The heap property describes how a heap is partially ordered, through the use of node keys. Each node is the root of a heap (i.e., the heap property applies recursively).

In a **max-heap**, the key of a node is always greater than, or equal to the key of its children. The max is at the root.

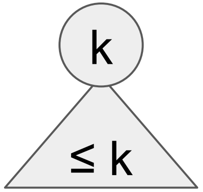
says nothing about relative order of children

Analogously, the key of a node in a **min**-heap is less than or equal to the key of it’s children, and its min is at the root.

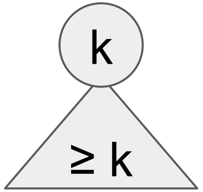
strict shapes -> consequences in efficiency

Heaps bear some similarities to binary search trees (BST). Heaps allow duplicate keys. Heaps are always balanced, which is not guaranteed in a BST. The heap property is a relaxation of the BST property.

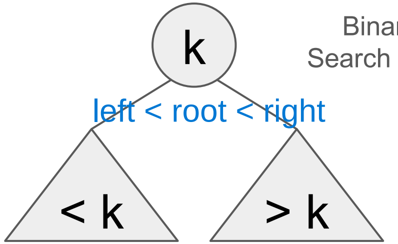
The root is "a max", not the max b/c duplicate values are allowed



Max-Heap



Min-Heap



Binary Search Tree

heap is opposite of binary tree (whose shape is very flexible, order of operations matters)

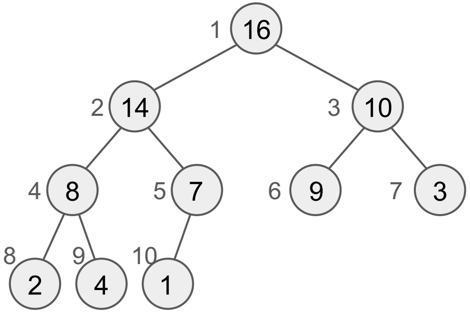
Heaps

In a heap, there is no implicit relationship between the children of a node.

Here are two max-heaps, each happens to have 10 elements:

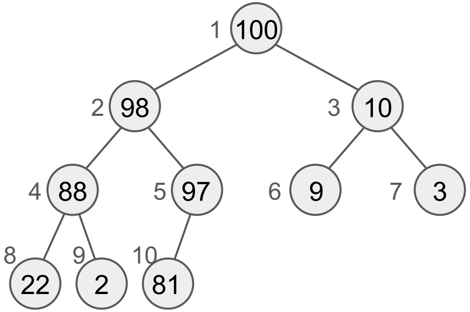
notice they're the same shape, since the shape depends on the number of elements it stores

n/2 leaves



how to confirm that these are max heaps? begin at the bottom (position 10). is 1 a heap? 1 is bigger (has no nodes so trivial). is the heap rooted at 4 a max heap? yes. same for 2, 3, 9. how about 7? yes because it's bigger than 1.

root > left && root > right
if this isn't the case, it's a problem



problem also if root < left OR root < right by deMorgan

by the time we get to the root, the left and right sub heaps are legit max heaps, the only check at the root is to compare the value at the root with the root of its left and right subheaps b/c you know now that the subheaps are legit

Heaps

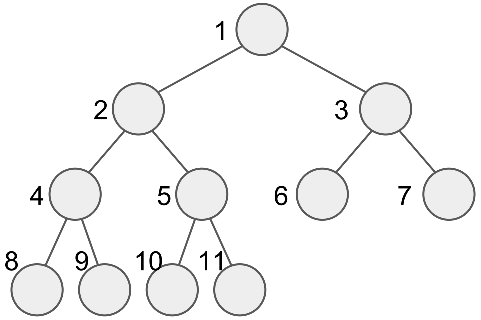
A max-heap, H, supports the following operations:

- max (H) value at the root, accessor — Return the element with the largest key without removing the element from H.
- extractMax (H) mutator (modify the heap) — Return the element with the largest key and remove it from H.
- insert (H, x) — Add the new element, x, to H.
- changePriority (H, x, k) — when dealing with priority queue (since priorities sometimes dynamic) Change the key of x to the new value k.

A min-heap supports min and extractMin, instead of the "max" operations.

Heaps

position 0: skip
position 1: root (max value of everyone)
position x: last node of the tree



We implement a heap efficiently using an array such that for the node in the "tree" at index i:

extremely efficient in navigating thru the array

The index of its parent is i / 2 (integer division)

left shift:
5/2 = 2

The index of its left child is 2i

right shift: 2(1) = 2

The index of its right child is 2i + 1

right shift + increment:
2(1) + 1 = 3

Notice that at least half the nodes in the tree are leaves, this means that they are subtrees of size one, which are trivially heaps.

To find the relations for a node, use the formulas given.

Example: Let's look at the node in position 5.

The parent of node 5 is in position 5 / 2 = 2.

Its left child is in position 2 * 5 = 10

Its right child is in position (2 * 5) + 1 = 11

precond: subheaps are legit
if violation, only occurs at root
- to identify violation, compare node with immediate children (constant work, 2) (leaves could never cuz subheaps are good)

Fixing a heap violation

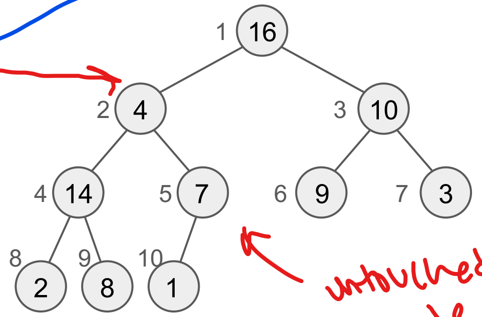
Here is an almost correct max-heap with a **single heap violation** at the node with index 2.

To fix this we call the Max_Heapify method on node 2 which swaps node 2 with the max of its children (in this example, node 4).

Notice that node 5, the “other child” of node 2, is a fine heap because its tree is unchanged and all its values < previous parent < new parent.

But swapping a smaller value into node 4 causes a heap violation. This will be fixed by calling Max_Heapify on node 4, which swaps node 4 with node 9.

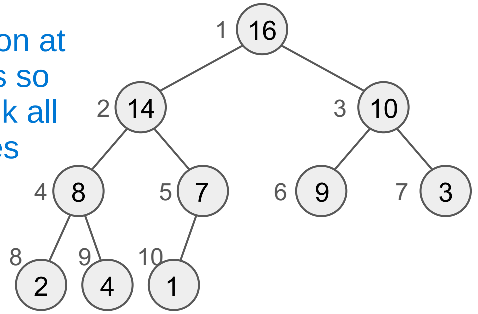
Then check node 9: Not a violation so done!



Initial Heap

untouched the whole time

worst case is violation at root and the value is so tiny that it has to sink all the way to the leaves



Final Heap

subheap that we didn't touch is fine (7) -> help runtime
4's subheaps are legit too because of the first, so can recursively call
replace w/ whichever is largest (4 -> 14)
max_heapify(4) to get (4 -> 8)

Q: Is this a legitimate max heap?

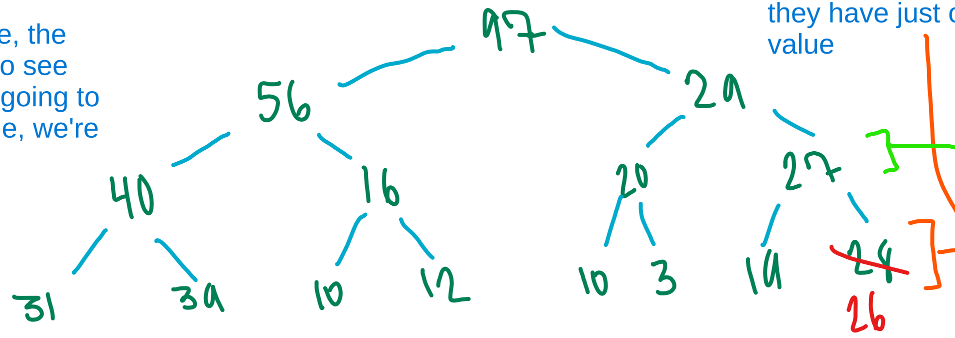
start checking from here since the 2nd half are trivial heaps

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	15	97	56	29	40	16	20	27	31	39	10	12	10	3	19	28

size of heap

even though we imagine the heap as a nearly complete binary tree, the heap is actually stored in an array. one could do index arithmetic to see who is the children of who and who is the parent of who, so we're going to draw the tree. we know it's going to be stored by levels. for this one, we're going to say that the tree starts from index 1.

leaves are trivial max/min heaps b/c they have just one value

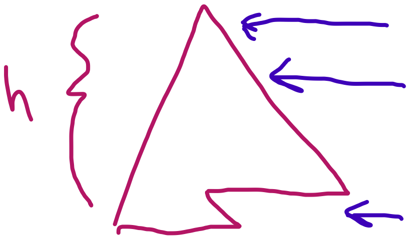


let's start from the rightmost, bottommost group, which is the tree of the 27, 19, and 28 (index 7). this isn't a max heap b/c the root is smaller than one of its leaves (27 < 28). let's change this to 26 for example's sake (which would be a max heap). next let's do the tree at index 6. this is a max heap. so is index 5, 4, 3, 2, and 1. notice to verify that the tree at index 3 is a max heap, the subtrees of 20 and 27 must also be max heaps.

if we added 1 to the value we got, we would get 100000000... (2^h+1), which is how we came to this conclusion

these are the parents of leaves, the work is constant b/c it compares to 2 children and worst case 1 swap

n? h



level	#nodes	sum(binary)
0	1 (2^0)	0000001
1	2 (2^1)	0000010
...	...	...
h	$+\ 2^h$	+ 1000000...
	n	1111111...

$2^{h+1} - 1$

when $n = 2^{h+1}$

$\log n \sim \log 2^{h+1} = h+1 \sim h$

the height of the heap is logn

constant work

there are (n/2)/2 of leaves parents

Creating a Max-Heap

constant amt of work (2 comparisons, 1 swap)

O(1) work/level
* O(logn) levels (proof by induction slides 4)

```
//fixes one violation of the heap property
//precondition: left(i) and right(i) must be max heaps
Max_Heapify(array A, index i){
    if(A[i] < A[2i] OR A[i] < A[2i + 1]){
        //swap the parent and biggest child
        swap(A[i], max(A[2i], A[2i + 1]));
        //fix any new violations caused by the swap
        Max_Heapify(A, index of swapped child);
    }
}
```

left

right

if violation found (i out of order w/ left or right)

Runtime: $O(\log(n))$

halfway point

in our example on the blank pg, work on 4-7, 2-3, then 1

```
//turns an input array into a max-heap
Build_Max_Heap(array A){
    for(i = n / 2; i ≥ 1; i--)
        Max_Heapify(A, i);
}
```

root

a

logn

n because of additive increment (-1)

multiply these

work backwards

leaves are in the second half of the array, where all the leaves are trivial heaps (don't have to check them)

the node that has children starts at the value before the halfway point, so working with the array R to L

Runtime: at first glance, appears to be $O(n\log(n))$ but, by careful analysis, we will show that a tighter bound is $O(n)$.

for more than half of the nodes on the tree, it's constant work, so we're overcounting

Runtime of Build_Max_Heap

total work done by Build_Max_Heap $= \sum_{l=0}^{\log n} n_l \cdot h_l \cdot c$

We stated before that `Build_Max_Heap` is in *O* (n * log(n)) since we are processing n/2 nodes and the worst case for the height of the heap is log(n).

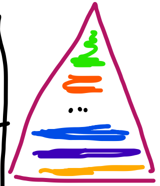
Notice that we can prove by induction that, written in next slide

in a complete binary tree (with the root at level 0),

the number of nodes in level k is 2^k, for all k >= 0.

Also,

the number of nodes in complete binary tree with k levels is $(2^{k+1} – 1)$, for all k >= 0.



nl = # nodes @ level l

1 node @ root

2 children of root

...

n/8 grandparents of leaves

n/4 parents of leaves

n/2 leaf nodes

÷2

hl = height of (sub)heaps rooted @ level l

log n

log n − 1

...

2

1

0 (1 max heaps)

+1

Runtime of Build_Max_Heap

We stated before that we would like to show the Build_Max_Heap function to be
lvl upper bounded by *O*(n). To do this, we notice that:

0 The **one** node at the root of the heap has to perform up to **c * log(n)** operations.

1 The **two** nodes at the next level have to perform up to **c * [log(n) – 1]** operations.

2 The **four** nodes at the next level have to perform up to **c * [log(n) – 2]** operations.

…

The **n / 4 (n/4=(n/2) / 2)** nodes at the level above the leaves have to perform up to **c * 1** operations.

[COULDN'T FIT ON NEXT SLIDE] we only do logn once, at the root. this is because when we're max heapifying, we're only max heapifying recursively with one of the children rather than both; so linearly from parent to leaf, so we concern ourselves with 1 child per level if n isn't a power of 2, we'll round up to the next power of 2 and compute with that

we start at the parents of leaves

Creating a Max-Heap

as we travel up the heap

c · num → height of subheaps @ that level

2^h ÷ denom → # of nodes @ level

$n = 2^x$, $x = \log n$

$h = x - 2$

For convenience, let's assume the tree is complete and thus n / 4 = 2^k for some k, and rewrite the number of steps in build_max_heap as:

↳ also power of 2

$$c * 2^k \left[\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots + \frac{k+1}{2^k} \right]$$

constant amount of work

$2^h * \frac{2}{2^1}$ gives us grandparents of leaves

① · ② = amount of work parents of leaves

+1

÷2

# of leaf parents

great grandparents

height of heap / # of heaps

root

$\frac{2^h}{2^h} = 1$ node (root)

c · h+1 = height @ root (# of parents of leaves, +1 is the leaves)

The series can be rewritten in summation notation which allows us to further manipulate it.

$$\left[\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots + \frac{k+1}{2^k} \right] = \sum_{i=0}^{k} \frac{i+1}{2^i} < \sum_{i=0}^{\infty} \frac{i+1}{2^i} = \boxed{c_2}$$

finite sum bounded by infinite sum

only one is log n @ root

The finite summation is less than its infinite counterpart, which is itself convergent and thus, bounded by a constant we call c_2. So, **the number of steps in build_max_heap is bounded by**

→ n/4

$$(c * 2^k) * c_2 = (c * c_2) * 2^k = c_3 * 2^k = c_3 * (n / 4) = c_4 * n \in O(n)$$

↳ bracketed part

↳ we get the constant

Other Heap Operations

max(H): returns the element in position 1. $O(1)$

extractMax(H): Put the last leaf at the root position and call max_heapify on root. $O(\log(n))$

more careful b/c this is a mutator; not only returning the max, we change the heap, so we remove the root and rearrange the values

insert(H, x): Place x at the last leaf and shift up until it not longer violates the heap property. $O(\log(n))$

we add x at the end. now we need it to bubble up to where it needs to be bubbled. the worst is to the root because it's very large, which is logn [levels].

changePriority(H, x, k): Find the element, x, change its key to k. If k is less than the old key, call max_heapify on x. If k is greater than the old key, shift up as needed. $O(\log(n))$

let's say there's some value at the root, m. we save m b/c we need to return it when we're done. we need to rearrange the values now. we also need to shrink the size of the heap by 1, which changes the shape. there's a last element x. since you have a copy of m, you have a blank space, so we can move that up to where the m was. notice that we only messed with the root here. the subheaps are in good shape now although a violation is introduced at the root. to fix it, we call maxHeapify (logn) and everything else is constant.

feed in an array of values, they're organized

Applications of Heaps: Heapsort

We can use a max heap to sort the elements in the array in non-decreasing order, as follows:

doesn't completely sort the elements, it creates a partial ordering, but not complete

1. Call build_max_heap on an unsorted array. positionWhereUnsortedPortionEnds = n.
2. Extract_max. This calls max_heapify and leaves a “free spot” at positionWhereUnsortedPortionEnds.
3. Swap max into A[positionWhereUnsortedPortionEnds].
4. positionWhereUnsortedPortionEnds--.

Repeat steps 2-4 until the heap is empty.

The time complexity of this in-place sort is:

n calls to extract_max (see previous slide)

(the cost of step 1) + *O*(n) * (the cost of steps 2-4) = *O*(n) + *O*(n * [log(n) + 1 + 1]) = ***O*(nlog(n))**.

Applications of Heaps: Priority Queue

A priority queue implements a set, S, of elements, where each element is associated with a key. As the elements queue up, those with the highest/lowest priority, as determined by their keys, are placed at the front of the queue.

To remove the element with the highest priority, swap the root item, A[1], with the item at the end of the heap, A[n]. Then remove and return the old root (now in A[n]), decrement the heap size, and call max_heapify on the new root.

To add an element to the queue, add it at the end of the heap and check if it breaks the heap property. If it does, swap it and its parent and check again. Continue until the new element no longer breaks the heap property.

Practice Quiz: Heaps

1. Beginning with the min heap [10 18 70 60 45 75 175 85 84], perform the following operations **in sequence**. Show the heap at the end of each operation.

a. ExtractMin () // show the heap after the operation [18 45 70 60 84 75 175 85]

b. (On the same heap, after doing the operation above) Insert (25)
   // show the heap after the operation [18 25 70 45 84 75 175 85 60]

2. The amount of work performed by an Insert operation on a heap depends on the value being inserted.

a. Give an example of a value that would have caused even more work that the Insert operation given in problem 1.b. 5 anything less than 18

b. What is the worst-case runtime of the Insert operation? Explain.

it would be logn b/c that's the number of levels it would have
to bubble up to (worst case being the root)

Graph Representations

A Graph G = (V, E) with n nodes/vertices can be stored in memory using:

- Adjacency matrix: Nodes are numbered 1 through n and edges are stored in a 2-D array with nxn elements. A[i][j] indicates the existence/weight of an edge from node i to node j. An undirected graph can be stored in a triangular matrix.

- Adjacency list: Nodes that have an edge between them are called “adjacent” “neighbors.” For each node, maintain a list of its adjacent neighbors (including edge costs, if any).

Prim’s MST algorithm can be implemented with an Adjacency list representation of the input graph and a Priority Queue.

A MinHeap-Based Priority Queue for Prim's MST

```
theInCrowd = {};
set the currentDistance to all nodes to INFINITY;
set the currentDistance to some initial node to 0; // this is where we'll start

while(there are nodes outside theInCrowd){ // there are more nodes to be spanned
     v = the outside node that is closest to TheInCrowd;  // n nodes * O(log n)
     add v to theInCrowd;
     for each outgoing edge of v, let's call it (v,w)
        if (w is not in theInCrowd and EdgeWeight(v,w) < currentDistance(w))
           currentDistance(w) = EdgeWeight(v,w);  // O(n²) edges * O(log n)
}
```

ϵ $O(n^2\log(n))$, same as Kruskal's MST

1a. [10 18 70 60 45 75 145 85 84] size 9

m x(last)

↓ swap size 8

[84 ———————— | 10]

↓ min-heapify ↳ not in heap

[18 84 70 60 45 75 145 85 | 10]

[18 45 _ 60 84 _ _ 85]

1b. [18 45 70 60 84 75 145 85 $\boxed{25}$]

+1 size=9

compare 25 to pos 4

25 < 60 → swap

[18 45 70 $\boxed{25}$ ——— 60]

unbubbled now since leaf swapped & other half

[18 25 70 45 ———]

25 > 18 so we good