Date

PRIORITY QUEUE

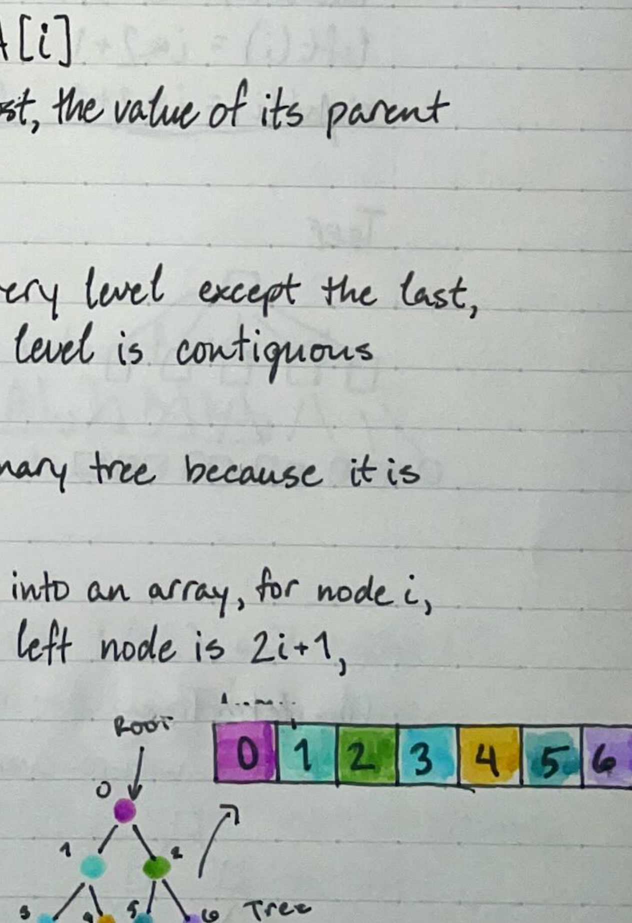
- In a queue, elements are added to one end and elements are removed from the other
- In a priority queue, elements are organized by priority
- Elements can be promoted to higher priority, but never demoted
- A linked list or array are no longer efficient for this. Instead we will use a heap (which is built on an array).
- Elements can be added anywhere, but are only removed from the front

HEAP

- A heap is a nearly complete binary tree that respects the heap property
  - ↳ Heap property => $A[\text{parent}(i)] \geq A[i]$
    - The value of a node is, at most, the value of its parent

Binary Tree

- Complete => Every level is full
- Nearly Complete => Complete on every level except the last, where the last level is contiguous
- Turn a binary tree into an array
- We can represent the heap as a binary tree because it is a nearly complete binary tree
- In a binary tree which is collapsed into an array, for node i, the parent of i is at $(i-1)/2$, the left node is $2i+1$, and the right node is $2i+2$
- Height of a node in a tree is the number of edges in the longest path from a leaf to the root
  - ↳ Node with no children



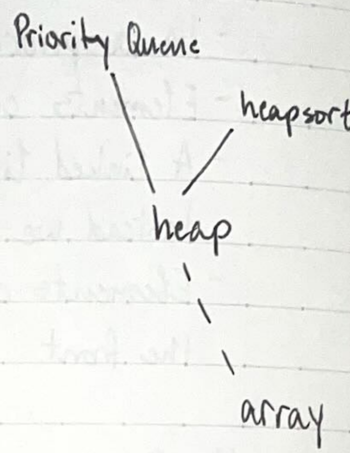
Date

HEAP

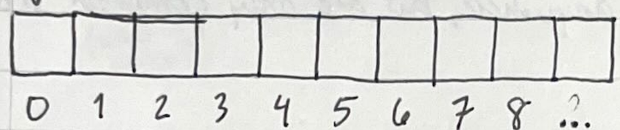
- A nearly complete binary tree that repeats the heap property.

- Max heap: A[Parent(i)] ≥ A(i)

  Imagine a heap of integers (or objects with key values

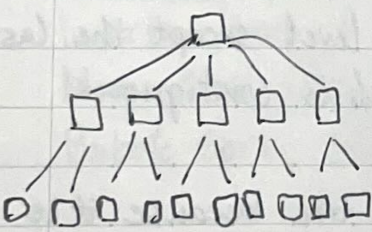


root



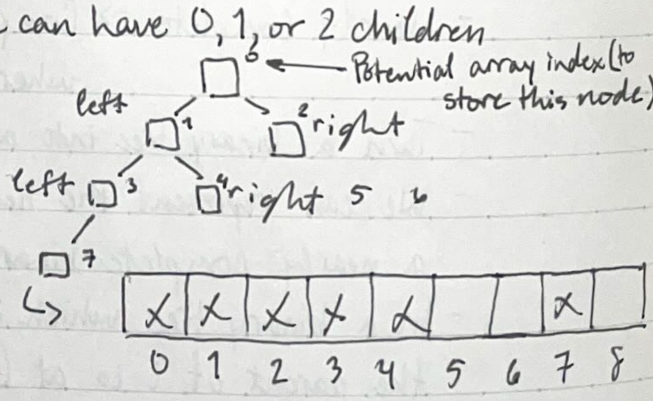
"zero-based" heap	1 based heap
root is A[0]	Parent (i) [i/2]
Parent(i) = (i-1)/2	left(i) 2i
left(i) = i*2+1	right(i) 2i+1
right(i) = i*2+2	

TREE



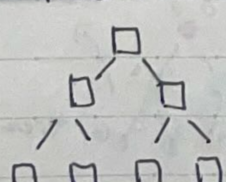
BINARY TREE

Node can have 0, 1, or 2 children



Complete Tree

- A tree where every level is filled



Date

Height of a node

- Defined as the number of edges on the longest simple downward path
- height is $O(\log n)$

Basic Operations

- Heapify (i) $O(\log n)$ maintain heap
- Buildheap $O(n)$ array to heap
- Heap sort $O(n \log n)$ heap to sorted array
- Heap Insert $O(\log n)$ add new value
- Extract Max $O(\log n)$ Remove top value
- Increase key $O(\log n)$ promote an elt., helper to insert

Support Operations

- swap(i,j) swaps A[i] & A[j]
- parent(i)
- left (i)
- right (i)

  } inline

- index_of_max (i, j, k) Returns index of max value $\underbrace{A[i], A[j], A[k]}$

  i: always valid

  j, k: maybe invalid

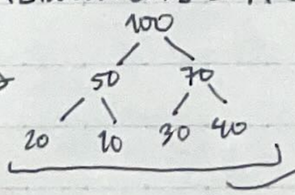
  ↳ WITH BOUNDS CHECK

Heapify (i)

- Assuming that binary heaps coded at left(i) and right (i) are valid heap, but that the node at i might violate the heap property.

$A[\text{Parent}(i)] \geq A(i)$

A valid heap →

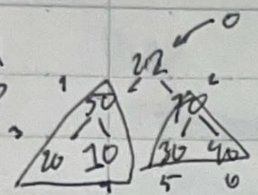


100	50	70	20	10	30	40

← Not completely sorted

Date

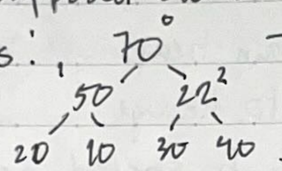
Δ - Valid heap



→ heapify(0)

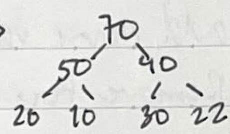
- The value at i "floats downward" so that the heap rooted at i becomes a heap, check for new violation at supposed element

Becomes:



heapify is called again at 2

heapify(2) →



→ heapify called again heapify(6)

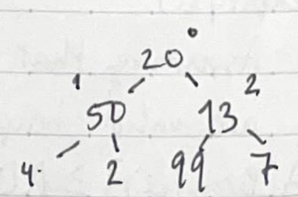
<u>-heap needs size variable</u>

Heapify Code ← Goes down to proper location

```
private: heapify(i) {
            n = index_of_max(i, left(i), right(i));
            if (n != i) {
                swap(A[i], A[n]);   ← swaps elements   (written as swap(i,n))
                heapify(n);         ← calls recursively
            }
         }
```

O(n) Build heap

- Convert an array A[0 ... n-1] (unordered) into a heap
- elements A[⌊n/2⌋ ... n-1] are already 1 element heaps



doesn't need to call heapify bc they are already heaps

Class is PQ

O(n)
```
buildheap() {
    for(int i = size/2 - 1; i >= 0, i--) {
        heapify(i);
```
O(logn)

Will be coding this in assignment

start w/ empty array & continue to add values then use heapify on it

O(n log n) Heap Sort ← Returns in ASCENDING order

→ assumes/requires a heap

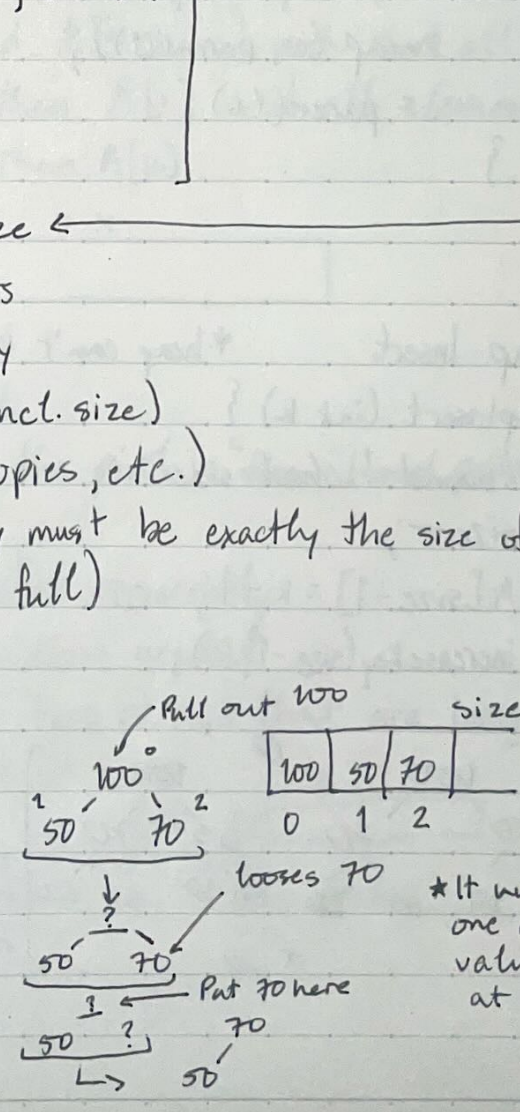
→ buildheap O(n)

* remember size ← Make a copy for you then put it back
* another array (copy values into this array)

```
heapsort() {
    for (int i = size-1; i > 0; i--) {
        swap(0, i);
        size--;
        heapify(0);
    }
```
• restore size

Implementation Requirements
- returns a sorted array
- heap is preserved (incl. size)
- do minimum work (copies, etc.)
- returned sorted array must be exactly the size of its elements (must be full)

O(log n) Extract Max
```
extractMax() {
    // check for valid size
    int max = A[0];
    A[0] = A[size-1];
    size--;
    heapify(0);
    returns max;
}
```



*It will always put one of the smallest values to the top at index 0

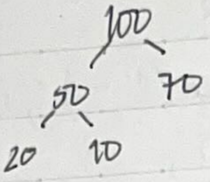
In assignment we will be coding decrease key

A[i] <= key

left(i) funct & other funcs will have just arith & not in the logic funcs (gunks them up)

Simulation:

100	50	70	20	10
0	1	2	3	4



↓

10	50	70	20	100

Opposite of heapify (works upward instead of downward)

Increase Key

public

index

key/value

```
increaseKey (i, k) {
  //bounds check, check index, check k is an increase, don't forget
  if (A[i] >= k) return;
  A[i] = k;
  while (i > 0 && A[parent(i)] < A[i]) {
     swap (i, parent(i));
     i = parent(i);
  }
}
```

will trigger when value increases to original val.

Ex: heapInsert(100)
99 → +1 = 100

Heap Insert   *heap can't be full (don't go out of bounds)

public

```
heapInsert (int k) {
  //bounds! check size
  size++;
  A[size-1] = k - 1;
  increaseKey(size - 1, k);
}
```

heap relies on the size int var.

has to be here to trigger insertKey

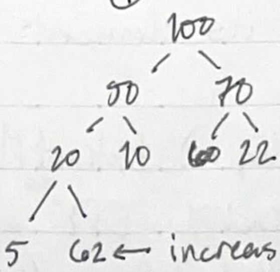
heapInsert(100);

100 - 1

size: 1

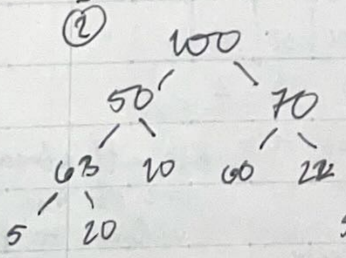
99		
0	1	

①

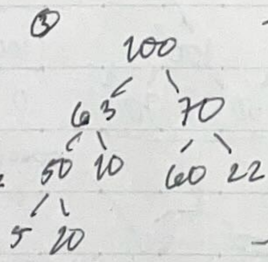


← increase to 63 then it checks 20 then 50

②



③



Ends here for now