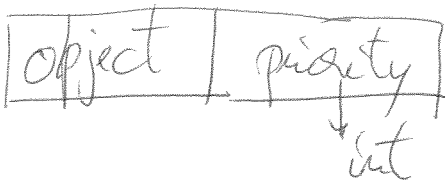
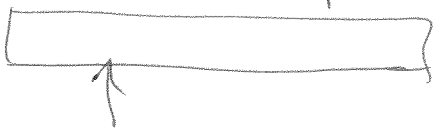
COMP 250 - Lecture 28: Heaps

Priority queue: deque objects in order of priority



Always dequeue object w highest priority as efficient as possible

- Sorted array (sorted by priority)
  - O(1) to access most priority element
  - Insert  O(n) (shift)
- Sorted linked list
  - O(1) access (same for removal)
  - Insert: list traversal O(n)

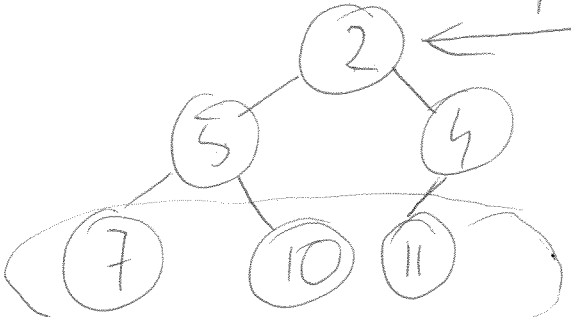
cheaper (pay a bit more for removal)

Heap (internally to suppot a priority queue)

↳ Binary tree

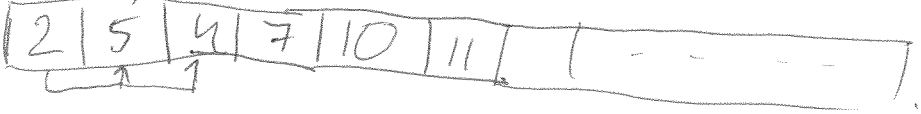
↳ At every node: parent has more priority than children

Low int means more priority

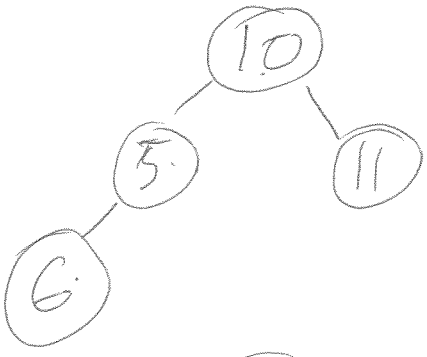


node (key) is lower than children (no ordering among children)

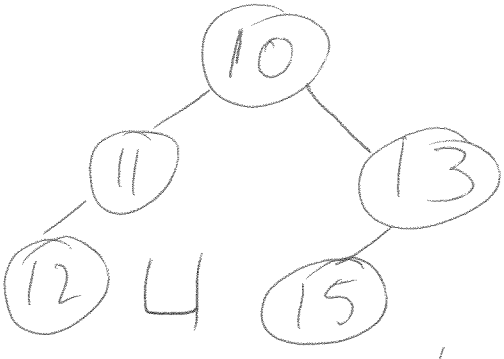
→ levels fill left → right: level i filled before i+1



No gaps in array

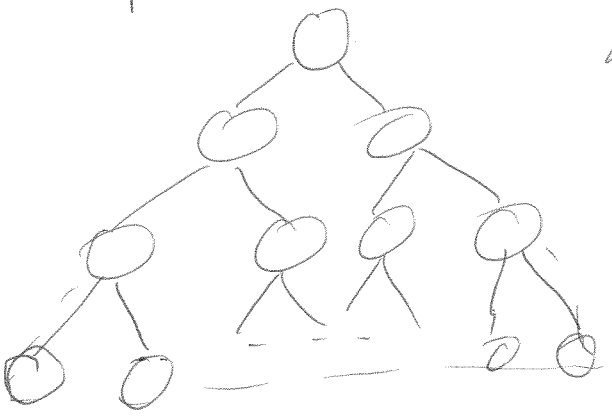


Not a heap: 10 larger than 5



Not a heap: last level fills left-to-right.

Heap of height h: how many nodes does it have?



$$\underbrace{1+2+2^2+\cdots+2^{h-1}}_{\frac{2^h-1}{2-1}+1} + \underbrace{1}_{\text{last level}}$$

$$= 2^h$$

$$1+2+\cdots+2^{h-1}+2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1$$

n is # of nodes, $h = \log_2 n$

Any alg that has to go down the tree should be $O(\log_2 n)$

Root: index 0

1st level nodes: indices 1, 2

2nd level: 3 4 5 6

Index i: children are at $2i+1$, $2i+2$

No need to keep extra references!

Child → parent $2i+1$, $2i+2 \rightarrow i$

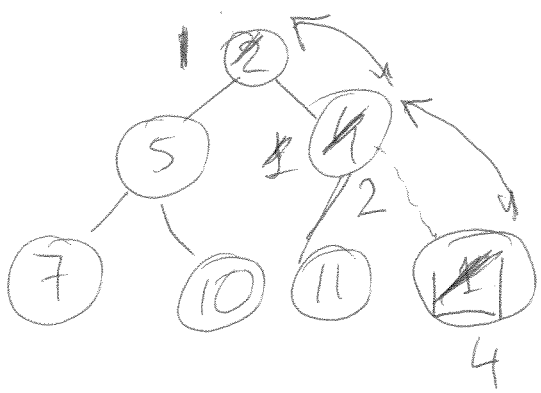
j: index of child $\left\lfloor \frac{j-1}{2} \right\rfloor$

Operations on heaps

- findMin() → find element with highest priority
- insert(Object o, int priority)
- removeMin() - returns elem with highest priority

```
return (root.content)   O(1)
```

insert / removeMin(): keep heap property



Insert(1)

"Bubble up" the 1 i.e. swap upwards in tree

insert (method inside Heap class)

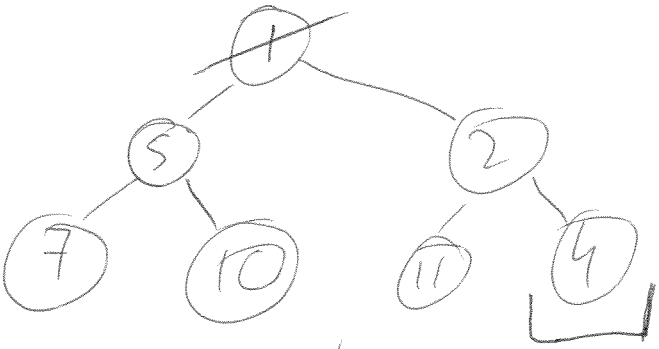
```
Algorithm insert (Object o, int priority)
// wrap this inside the heap nodes
lastNode ← getLast()  // index of 1st available spot
  // last position where to put stuff
lastNode.setKey(priority)
lastNode.content(o)
n ← lastNode; //n is an index
while (n.getParent() != null and n.getParent().getKey() >
        ↑                                          priority)
    not at root yet
    swap(n, n.parent) // swap in the array
```

→ O(1)

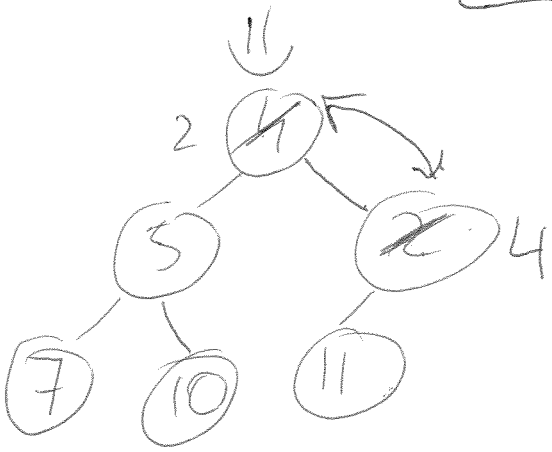
while: at most goes on longest path (aka n. swaps = height) ⇒ $O(\log_2 n)$

(instead of $O(n)$)

(may need to resize array)

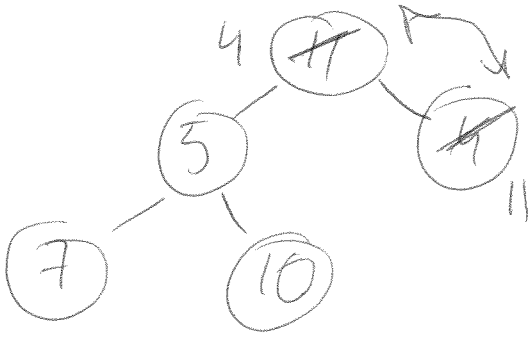


removeMin()
↳ return the element with key 1



"Bubble down" the 4 until heap property is satisfied
swap with smallest child

removeMin() → return 2




```
Object removeMin()
Object result = root.getContent()          ] O(1)
swap(root, lastNode)
n ← root
while (n has child && n.getKey() >
       min(n.left.getKey(), n.right.getKey()))
    swap(n, minIndexChild)
return result;
```

$O(\log_2 n)$

insert : $O(n)$ remove : $O(1)$

insert : $O(\log n)$ remove : $O(\log n)$

Use heaps for sorting

```
for (int i=0 , i<a.length ; i++) - n times
    h.insert(a[i]) -          ≈ O(log n)
for (int i=0; i<a.length ; i++) - n times
    a[i] ← h.removeMin()  ≈ O(log n)
```

$\Rightarrow O(n \log n)$

Heap sort