Lectures 22-23

Priority Queues and Binary Heaps

Lectures 22-23

Today:

- Priority Queue ADT
  - Array implementations
  - Recursive Implementations
- Binary Heaps

Priority Queue ADT (Review)

Data / Properties:

- a collection of obkects and their associated keys

Operations / Methods:

- insert (*x*, *key*) into the collection
- remove object with smallest key
- change the key of an object
- isEmpty

Priority Queue Implementation — Array (Review)

First attempt to implement using an array

- keys are stored in first *N* elements of the array
- two obvious approaches

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`arr:`	34	13	8	5	3	-	-	-
						`len`		

Approach 1:

- store keys in
    - `.removeMin()` costs O(1) - min ar the end of list is easy to remove
    - `.insert(x, k)` costs O(N) - similar to last iteration of insertion sort

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`arr:`	34	3	8	5	13	-	-	-
						`len`		

Approach 2:

- store keys in
    - `.insert(x, k)` costs O(1)
    - `.removeMin()` costs O(N) - find the min

Priority Queue Implementation — Array (cont'd)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`arr:`	34	3	8	5	13	-	-	-
						`len`		

Approach 2 b):

- store keys in *any* order
- also store
  - `.insert(x, k)` costs O(1) plave(x,k) at end of array; update minpos
  - `.removeMin()` costs Text

```
.insert(x, key) {

    arr[len++] = node(x, key);
    if (minpos== -1 || arr[minpos].key > key)
        minpos = len-1;

}
```

```
.removeMin() {

    swap(arr[—len], arr[minpos]);
    return arr[len];

}
```

Recursive Definition of Priority Queue

Strategy: Apply divide and conquer so the min of any priority queue is easy to find.

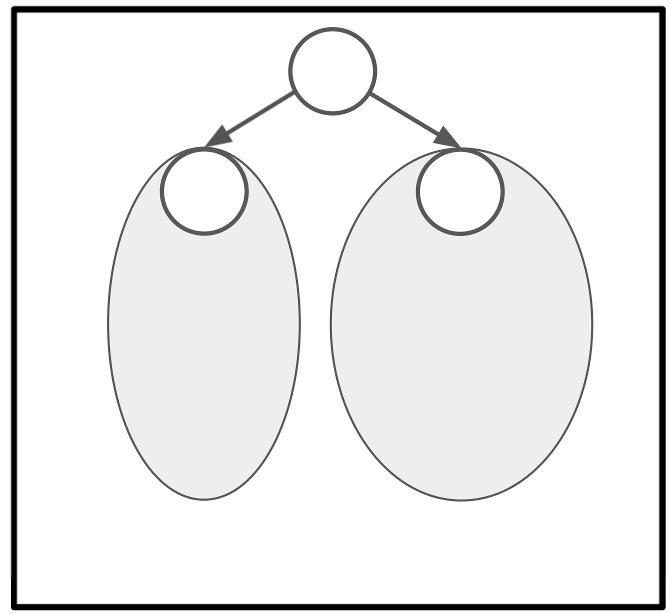
Organize the keys of a Priority Queue by:

- Its min element[at the top]
- 2x priority queues (of roughly equal size) to hold the rest of the elements

Because it's defined recursively, min of the left and min of the right are easy to find

Text

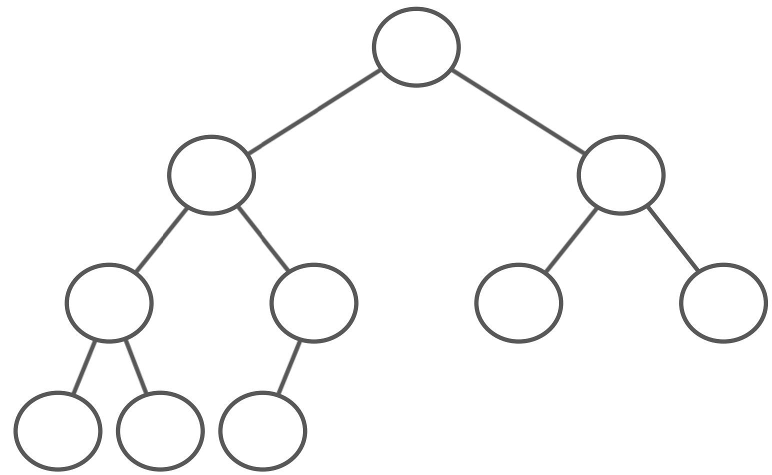
- `.removeMin()` — compare the two mins and promote it



Use a tree, but how to maintain balance?

- best is completed tree -> binary heap

Binary Heap



A *binary heap* is a balanced binary tree that follows the two invariants:

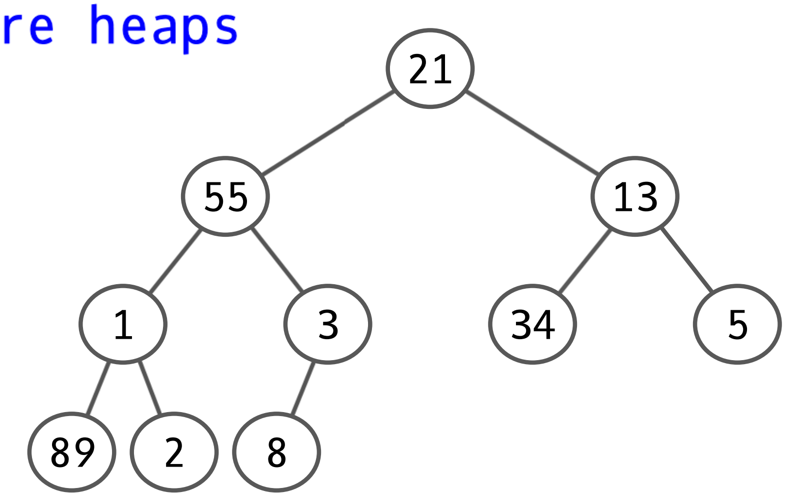
- every node's key is less than or equal to the keys of its children
- the tree is *complete*, which means:
  - every level full except maybe the last(no gasps)
  - leaves on last level as far to left as possible
- min is easy to find root
- second min is easy to find level 1
- heap repair:
  - "trickle down"
  - "trickle up"

Binary Heap

Build a binary heap   bottom up

- every leaf is   heap

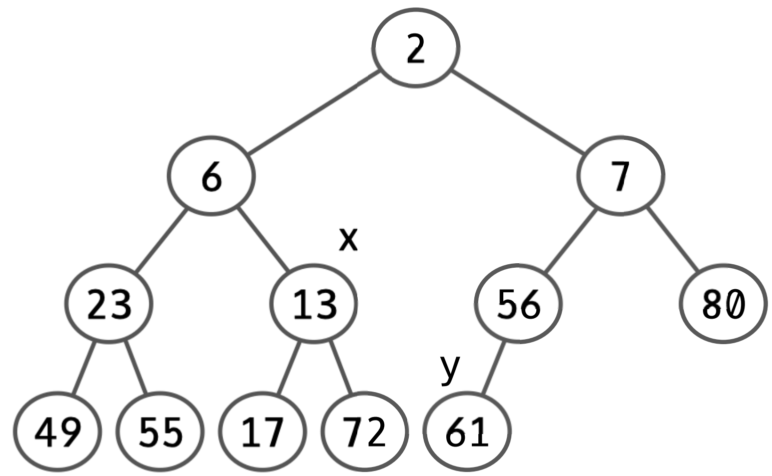
```
// Pre:  both left(x) and right(x) are heaps
void heapify(x) {

    if(key(x) <= min(key(left(x)), key(right(x))))
        return; // its a heap
    else if (key(left(x)) < key(right(x)))
        swap(x, left(x));
        heapify(left(x);
    else
        swap(x, right(x));
        heapify(right(x));

}
```



Binary Heap — `.decreaseKey(x,k)`

Useful for

- E.g., `.decreaseKey(x, 1)`, `.decreaseKey(y, 5)`
- Strategy: TRickle up




```
//  Pre: newkey <= key(x)
void decreaseKey(x, newkey) {

        key(x) = newkey;
        trickleUp(x);

}
```

```
void trickeUp(x)
if(parent(x) && key(x) < key(parent(x)))
swap(x, parent(x));
trickleUp(parent(x));
```

Binary Heap — `.extractMin()`

Easy to find the min - root

Easy to find the next min - compare roots of subtrees

Strategy 1: Text

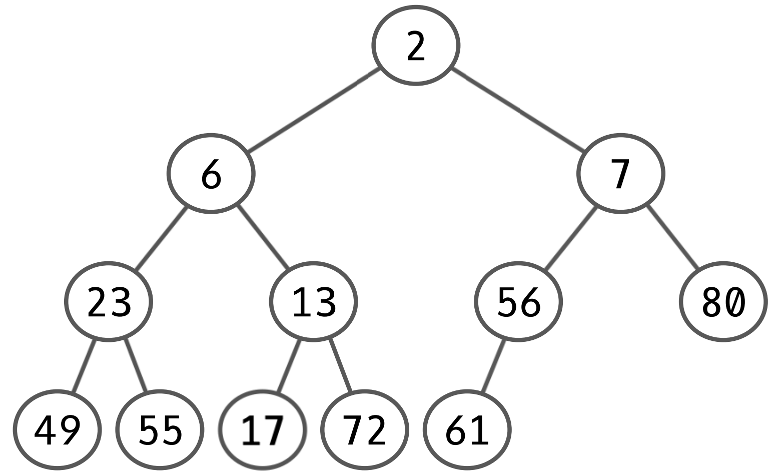
- promote next min & recurse!
- tree may no longer be complete

Strategy 2:

- promote last element to root & trickle down

`.enqueue(x,k):`

- main concern is to keep tree complete
- insert at bottom of tree & trickle up



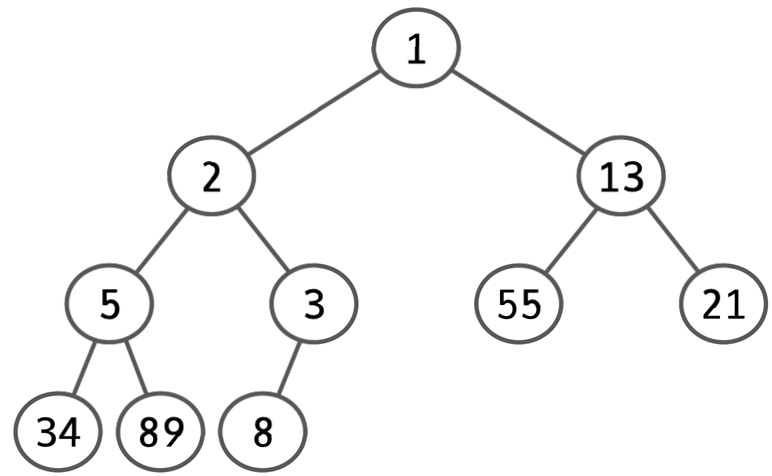
Binary Heap — Array Implementation

Why use a complete binary tree?

- tree is balanced h=logn
- for easy storage as an array

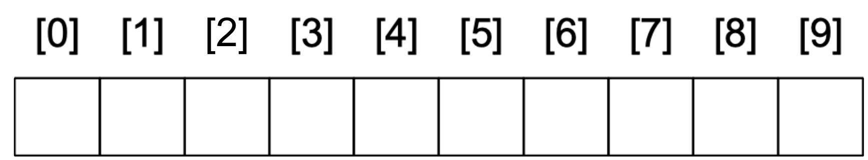
Array storage:

- follow level order traversal
- can transform array => heap w/np extra data strucutre cost



Relationship among elements?

- left(*i*) = 2i + 1
- right(*i*) = 2i + 2
- parent(*i*) = (i-1)/2



Heap Operations

Heap has `.min()`, and `.enqueue(x,k)`, but not:

- .search()
- .succesor()
- .predecessor()
- .max()

Q. How much would .max() cost?

- search all leaves. How many? = (N+1)/2 = O(N)

Running times of:

- `.decreaseKey(x,k)?`
- `.extractMin()?` — one call to trickleDown() or trickleUp() O(h)
- `.enqueue(x,k)?`

Convert Array ↦ Heap?

Strategy 1: `.enqueue()` × *N* items

Sum of all depths

Strategy 2: Build the tree bottom-up

Sum of all heights

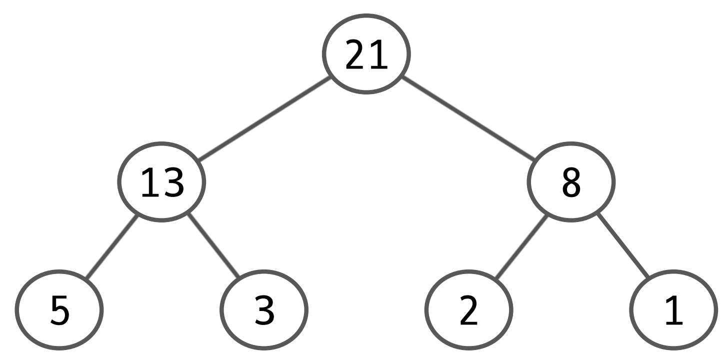
```
for (i = N-1; i >= 0; i--)
        trickleDown(i);
```

… or …

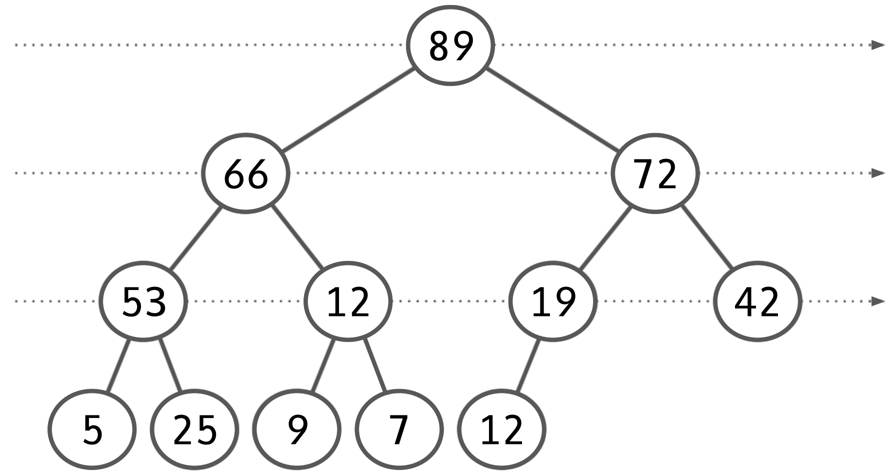
```
for(i=(N-2)/2; i>=0; i—)
    trickleDown(i);
```

Q. Why does this work?

Q. What’s the difference in running time?



Strategy 2: Analysis



What is the sum of all heights?

In a tree of height h there are:

- at most nodes of height = 1
- at most nodes of height = 2
- …
- at most of height = $h - 1$
- at most of height = h

Dijkstra’s Algorithm — Running Time Analysis

Problem: Find shortest distance to all reachable locations.

Algorithm:

```
initialize all distances ← ∞ (unreachable), except distance(start) ← 0
create an empty queue Q; enqueue all nodes → Q
while Q not empty {
        remove min node from Q → current
        if next is neighbour of current {
                distance(next) = min { distance(next),
                                       distance(current) + weight }
        }
}
}
```