# Binary Heaps & Priority Queues

- Priority Queue ADT
- Binary Heaps

# How do you read your e-mails?

Oldest first?
Newest first?
Most important first?

# What kinds of data structures model these methods?

# What kinds of data structures model these methods?

Oldest first: FIFO (Queue)
Newest first: LIFO (Stack)
Most important first: **Priority Queue**

# Priority Queue ADT

- Enforces some kind of "priority" on the items
- For general purposes, call the highest priority the "maximum" (or the minimum)
- Operations:
  - insert (**insert**)
  - remove maximum (**delMax**) xor remove minimum (**delMin**)
    - Note: You would not have both delMax and delMin in a single PQ–you would have one or the other depending on whether it is a Max PQ or a Min PQ
- Often, we store (key, value) pairs where the key is the "priority"
- **Key**: used to "rank" items—must be comparable

# Recall Project 1.

Basically, you got the “top M” integers by using a min PQ.

# Priority Queue API

`public class MaxPQ<Key extends Comparable<Key>>`

| | | |
|---|---|---|
| | `MaxPQ()` | *create a priority queue* |
| | `MaxPQ(int max)` | *create a priority queue of initial capacity* `max` |
| | `MaxPQ(Key[] a)` | *create a priority queue from the keys in* `a[]` |
| `void` | `insert(Key v)` | *insert a key into the priority queue* |
| `Key` | `max()` | *return the largest key* |
| `Key` | `delMax()` | *return and remove the largest key* |
| `boolean` | `isEmpty()` | *is the priority queue empty?* |
| `int` | `size()` | *number of keys in the priority queue* |

**API for a generic priority queue**

NOTE: a **MinPQ** would be basically the same except you would have **min**() to get the smallest key and **delMin**() to return and remove the smallest key.

Picture from [1]

# How would you implement a Priority Queue?

# Unordered Array

insert : $O(1)$

delMin : $O(N)$

insert 23, 10, 17, 28, 34, 89, 22, 10
remove min

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 23 | 10 | 17 | 28 | 34 | 89 | 22 | 10 |

# Ordered Array

insert : $O(N)$
delMin : $O(1)$

insert 23, 10, 17, 28, 34, 89, 22, 10
remove min

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 89 | 34 | 28 | 23 | 22 | 17 | 10 | ~~10~~ |

# Key Operations: **insert & delMax (or delMin)**

**Option 1**

Unordered array

Worst-case runtime analysis:

- **insert**
- **delMax**

**Option 2**

Ordered array

Worst-case runtime analysis:

- **insert**
- **delMax**

# Key Operations: **insert & delMax (or delMin)**

**Option 1**

Unordered array

Worst-case runtime analysis:

- **insert: O(1)**
- **delMax: O(n)**

**Option 2**

Ordered array

Worst-case runtime analysis:

- **insert: O(n)**
- **delMax: O(1)**

# Key Operations: **insert & delMax (or delMin)**

**Option 1**

Unordered array

Worst-case runtime analysis:

- **insert: O(1)**
- **delMax: O(n)**

LAZY!

**Option 2**

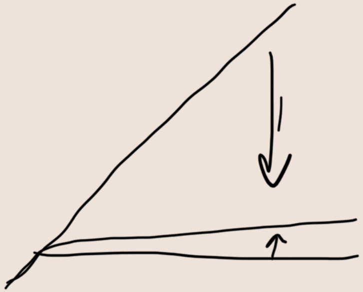
Ordered array

Worst-case runtime analysis:

- **insert: O(n)**
- **delMax: O(1)**

EAGER!

# SPOILER ALERT!!! There's a better way...



| data structure | insert | remove maximum |
|---|---|---|
| *ordered array* | $N$ | $1$ |
| *unordered array* | $1$ | $N$ |
| *heap* | $\log N$ | $\log N$ |
| *impossible* | $1$ | $1$ |

**Order of growth of worst-case running time for priority-queue implementations**

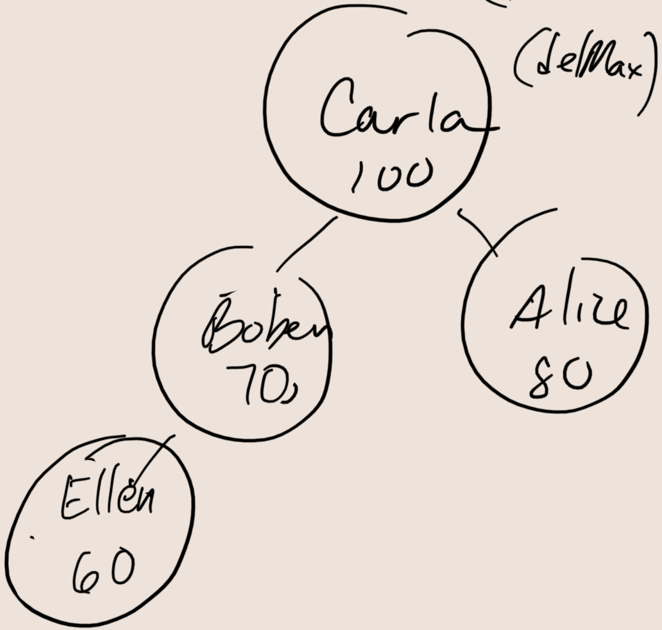
Picture from [1]

# Imagine a company...

- New people are always hired at the lowest level
- Only the CEO ever quits
- Final ranking is determined by an IQ test where any given employee can only be managed by another employee with a higher IQ.
- Any given person can only manage two people.

(insert) swim = bubble up

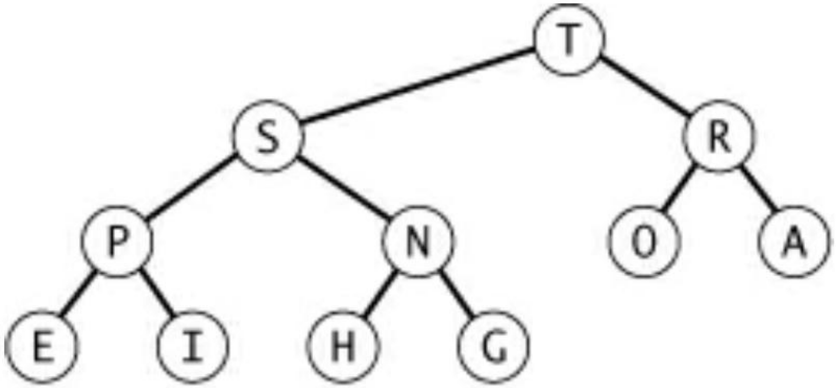
(delMax) Sink → bubble down



# So...what's a (max binary) heap?

**Definition.** A binary tree is *heap-ordered* if the key in each node is larger than or equal to the keys in that node's two children (if any).

NOTE: This is max-ordered binary heap. How would it be different if it was a min-ordered binary heap?



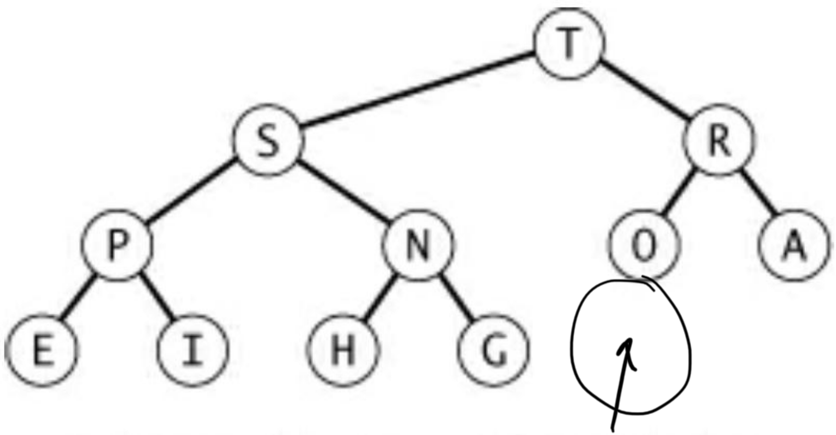

A heap-ordered complete binary tree

Picture from [1]

# So...what's a (binary) heap?

**Definition.** A binary tree is *heap-ordered* if the key in each node is larger than or equal to the keys in that node's two children (if any).

max

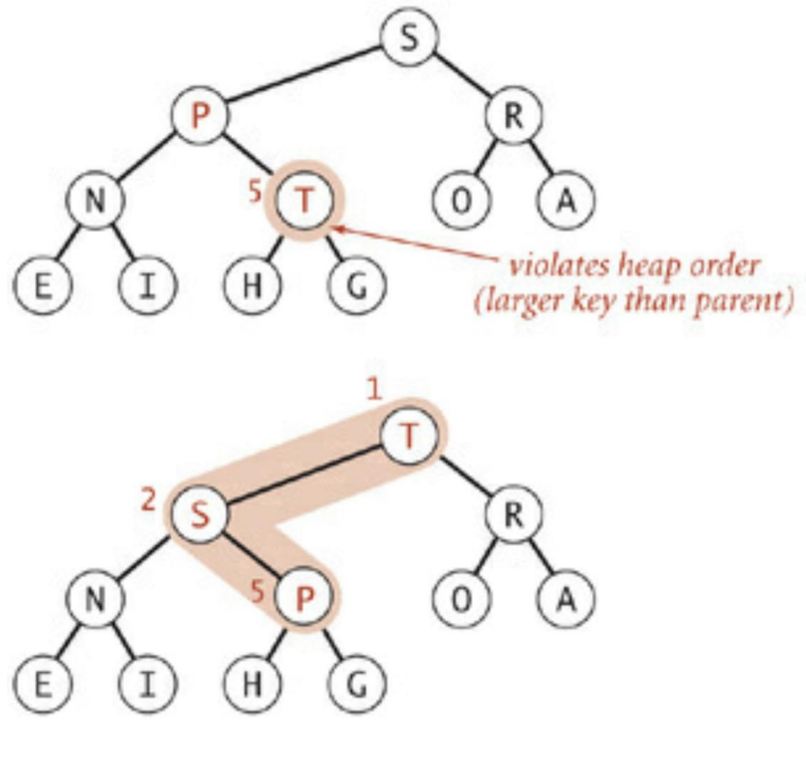



A heap-ordered complete binary tree

**Where is the maximum?** ROOT
**Where would we insert a new node?**
**So...which locations should be easy to access?**

Picture from [1]

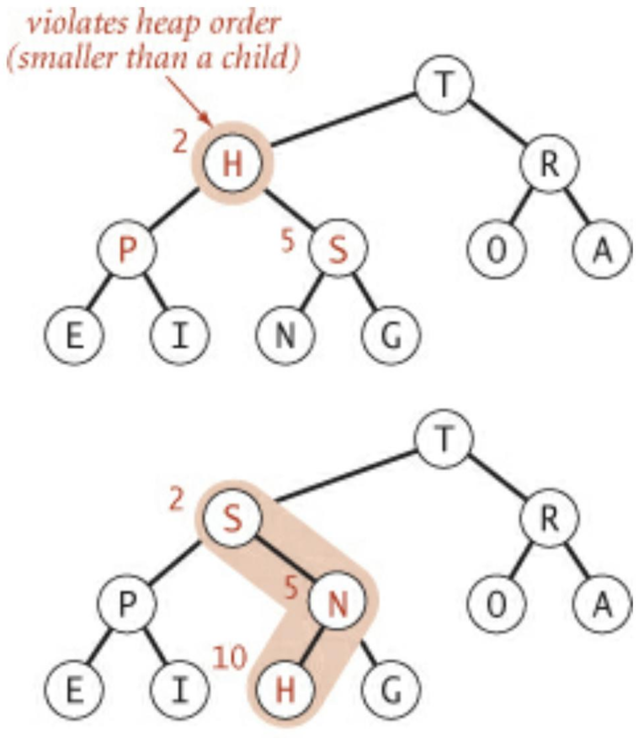
# Swimming up...



Bottom-up reheapify (swim)

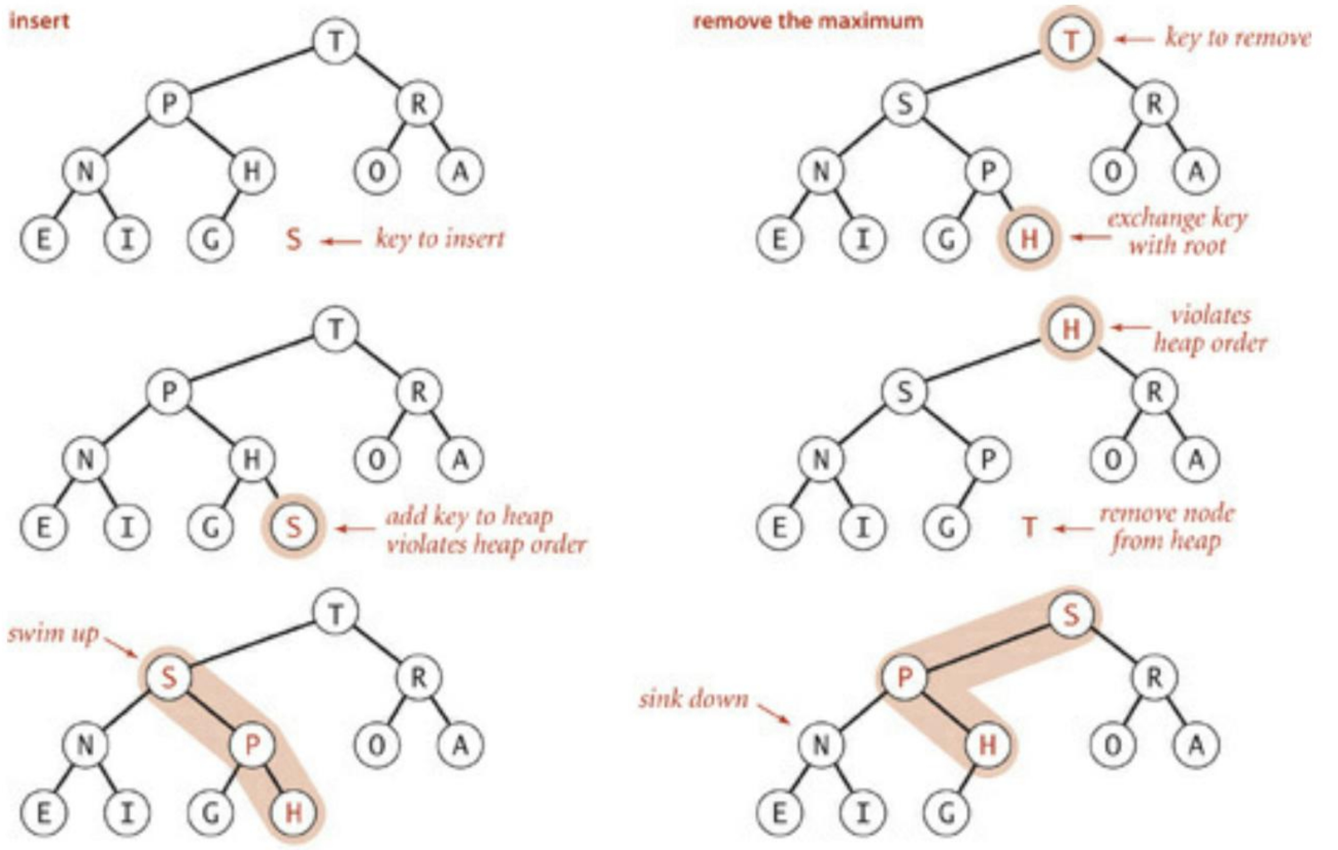
Picture from [1]

# Sinking down...



Top-down reheapify (sink)

Picture from [1]




Heap operations

Picture from [1]

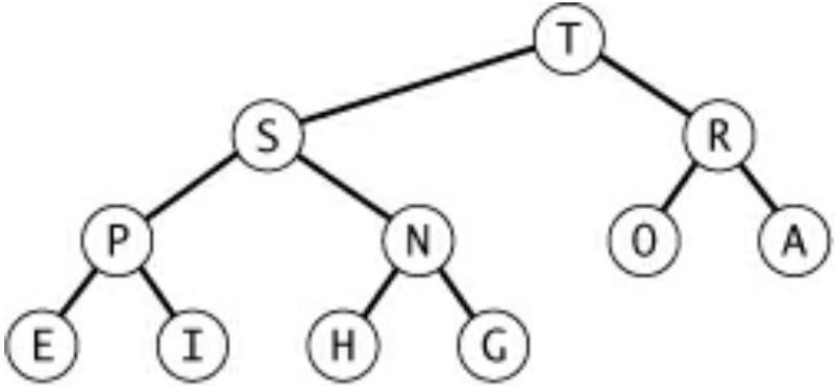
# So why is it O(logN)?

height : $O(\log N)$

complete tree
$\Rightarrow$ balanced tree

| data structure | insert | remove maximum |
|---|---|---|
| *ordered array* | $N$ | 1 |
| *unordered array* | 1 | $N$ |
| *heap* | $\log N$ | $\log N$ |
| *impossible* | 1 | 1 |

Order of growth of worst-case running time for priority-queue implementations




A heap-ordered complete binary tree

Pictures from [1]

# So why is it O(logN)?

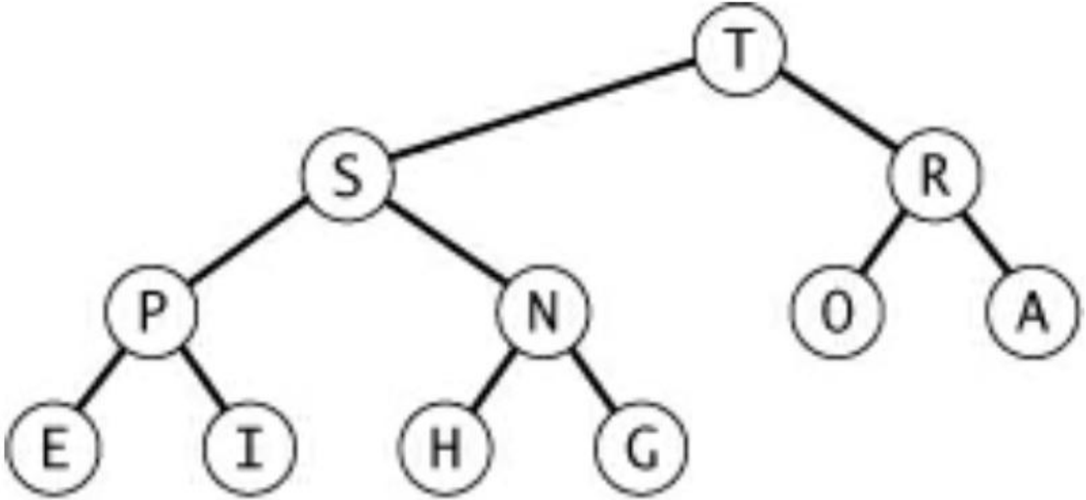
| data structure | insert | remove maximum |
|---|---|---|
| *ordered array* | $N$ | 1 |
| *unordered array* | 1 | $N$ |
| *heap* | $\log N$ | $\log N$ |
| *impossible* | 1 | 1 |

**Order of growth of worst-case running time for priority-queue implementations**

- A binary heap is always a complete binary tree,
- which means it is always balanced,
- which means the height is always *O(logN)*,
- which means that in the worst case, you have to *swim* or *sink* an item on a path of length *O(logN)*.

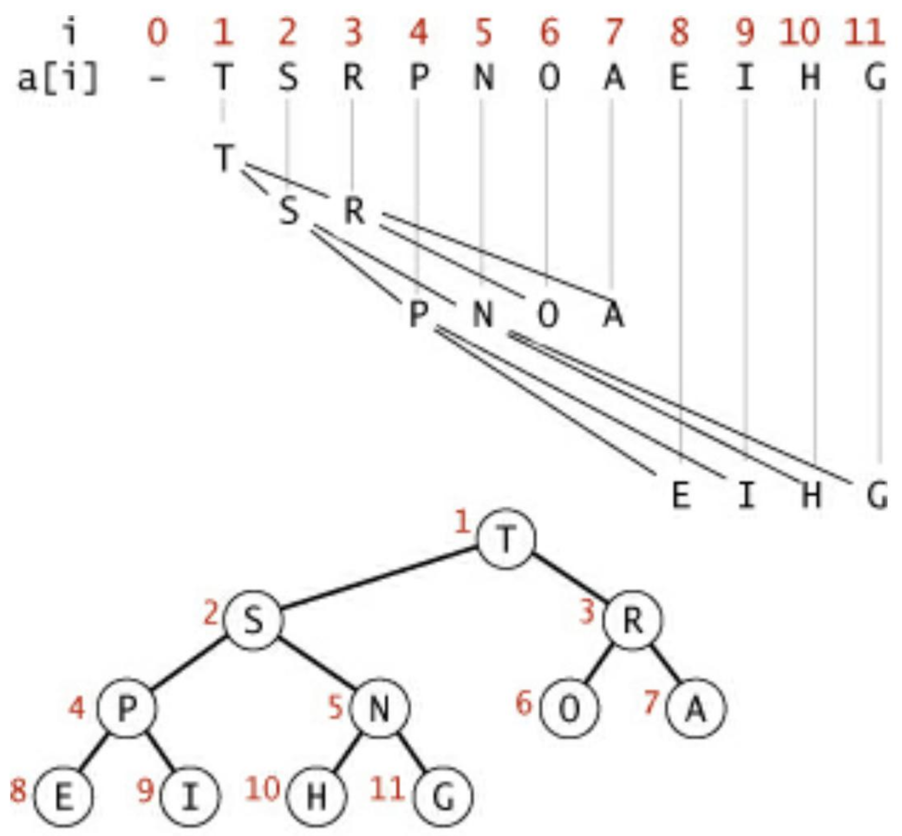
Pictures from [1]

# So how would you implement a binary heap?




A heap-ordered complete binary tree

Picture from [1]

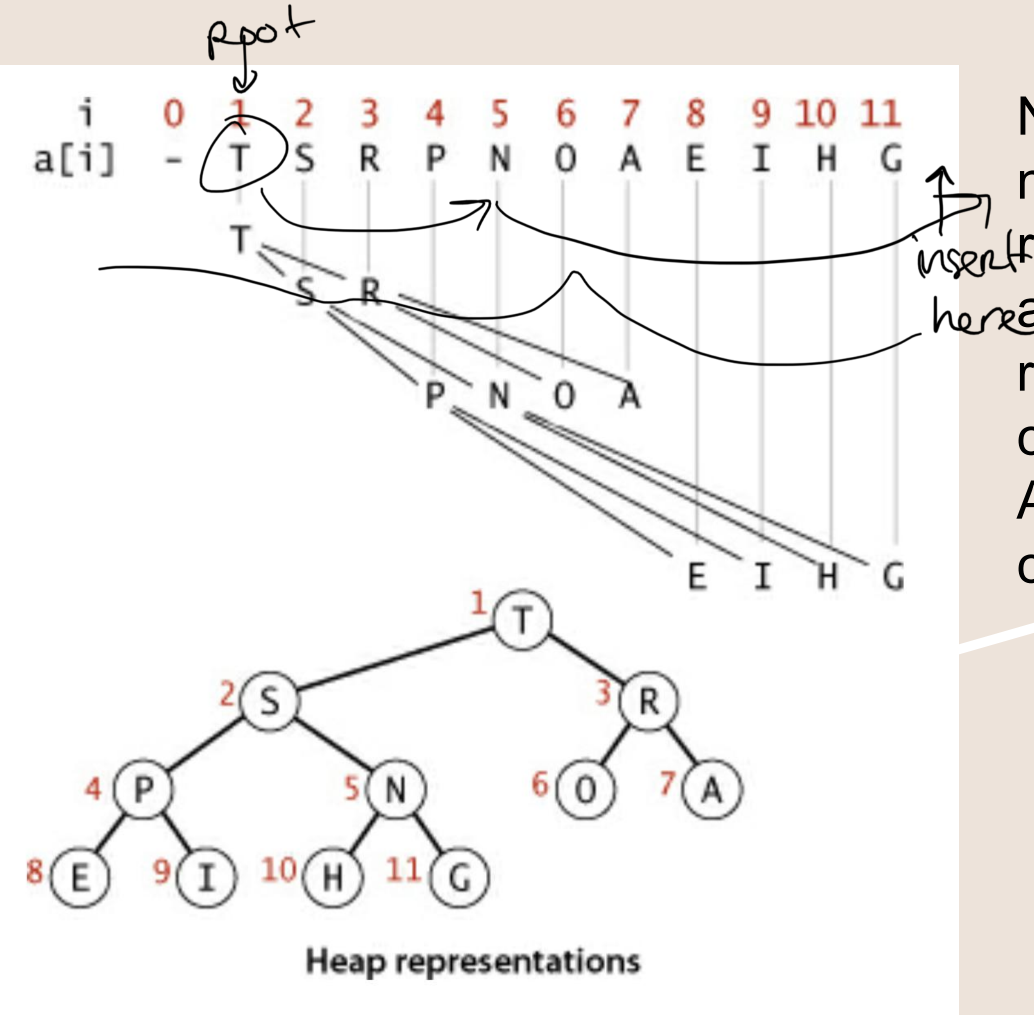


Heap representations

Picture from [1]

- If a node is at index **k**, it’s children are at **2k** and **2k + 1**
- If a node is at index **k**, its parent is at **floor(k/2)**
- Keep track of an insert index.
- Root (maximum) should always be at index 1
- Swim/Sink as appropriate

**In the worst case, any path from root to node is O(logn).**




Note that in this picture, index 0 is not used. But that is not a requirement. You could use index 0 as the root. It would just change the relationship between parents and children.

A node at index *k* would have children at $2k+1$ and $2k+2$.

Picture from [1]

# Array-based Binary Heap

parent: $k$ $\longrightarrow$ children: $2k+1$, $2k+2$

$\left\lfloor \frac{c-1}{2} \right\rfloor \longleftarrow c$

$k=0 \rightarrow 1, 2$

insert 23, 10, 17, 28, 34, 89, 22, 10
remove min

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 23 | 17 | 28 | 34 | 89 | 22 | |

# Common Misconceptions

- A binary heap is not the same thing as a priority queue. A priority queue is an ADT that stores and processes data according to some kind of priority ranking. A binary heap (which can be max-ordered or min-ordered) is a common way of implementing a PQ because it is a structure that does *insert* and *delMax* (or *delMin*) efficiently.
- A binary heap, in turn, can be implemented either as a tree or as an array.
- You can store anything in a PQ as long as you can assign some kind of ranking to it. Often, the “priority” is treated as the key in a key-value pair. e.g. (IQ, Employee), but the “key” and the “value” could also be the same.

# What are some applications that could use a PQ?

# References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)
[2] Book slides from Goodrich and Tamassia