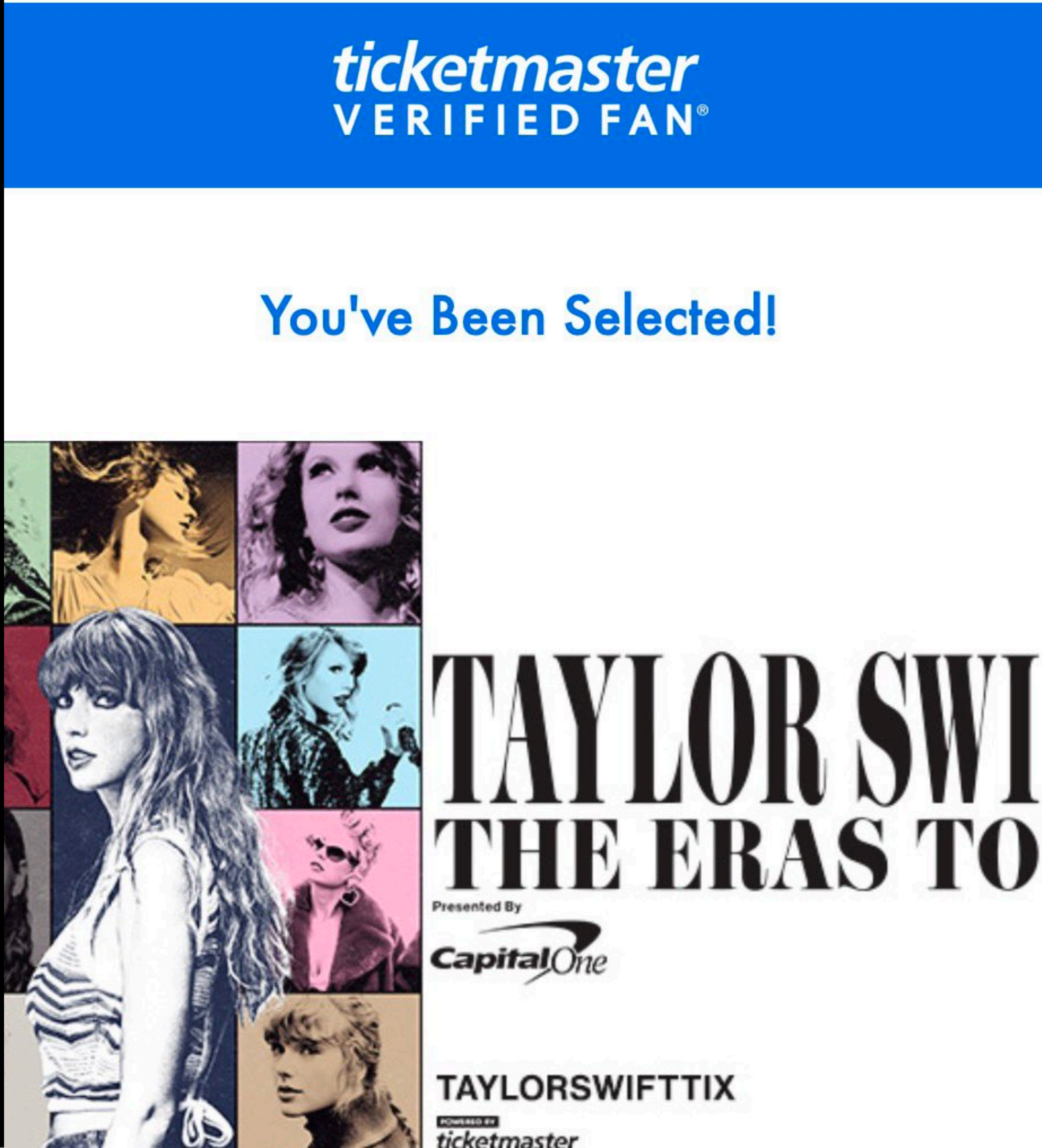
CIT 5940

Heaps & Priority Queues

The Problem: Selling Concert Tickets

We want to sell tickets to a group of fans. How do you prioritize who gets a chance to buy tickets?

- We want to be able to register every person's interest
- As long as tickets remain, we want to sell them to the person in line with the highest priority.



The Problem: Selling Concert Tickets

We could use a List or FIFO Queue to store the people who want to buy tickets.

- Hard to order based on priority other than FIFO
- Slow to both retrieve & remove members of a linear structure

The Problem: Selling Concert Tickets

We could use a `TreeSet` to store the people who want to buy tickets.

- Easier to store and order people by priority
- Still $O(n \log n)$ to register & retrieve all n interested fans.
- n can be very big $\Rightarrow$


HOME > MUSIC > NEWS

Nov 17, 2022 8:55am PT

Ticketmaster Explains Taylor Swift Ticket Chaos Amid Outrage; 'Eras Tour' Broke Record With Over 2 Million Tickets Sold in One Day

By Zack Sharf

The Problem: Selling Concert Tickets

Some further optimizations:

- We don't care about the relative ordering of the people who are **not** at the front of the line
- It's possible that the number of tickets t may be much smaller than the number of interested fans n, so if we can pay some costs *per ticket* rather than *per fan*, that would be helpful

Formalizing the Interface

Goal	Method	Notes
Add a new buyer to the queue	`boolean add(Buyer b)`	Duplicates?
Check if a buyer is already present	`boolean contains(Object o)`	Sometimes preferable to offload this to a different data structure
Get the highest priority buyer	`Buyer poll()`	Sometimes called `findMin()`/`findMax()`

Towards a Priority Queue

A **Priority Queue** is a data structure that supports the previous operations, plus other related ones listed here.

- Implements the `Queue` interface, so it is ordered
- Instead of ordering by position, we order by "priority"
  - can be the natural ordering of the contents, or defined using custom comparator
- Implemented using a **Heap** data structure
  - Kind of a tree, kind of linear

Complete Binary Tree

Recall: a **Complete Binary Tree** is a binary tree where the nodes are filled in row by row, with the bottom row filled in left to right.

Theorem: There is only one complete binary tree of n nodes for any value of n.

Height of a Complete Binary Tree

Other Theorem: The height of a complete binary tree of n nodes is $O(\log n)$

Idea:

- n is between two consecutive powers of two: $2^{h-1} \leq n < 2^h$
- If a tree has a height of $h-1$, we can store $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements
- $n \leq 2^h - 1$, so height of $h-1$ is all that's necessary
- $2^{h-1} \leq n$, so $h-1 \leq \log n$ and therefore $h \in O(\log n)$

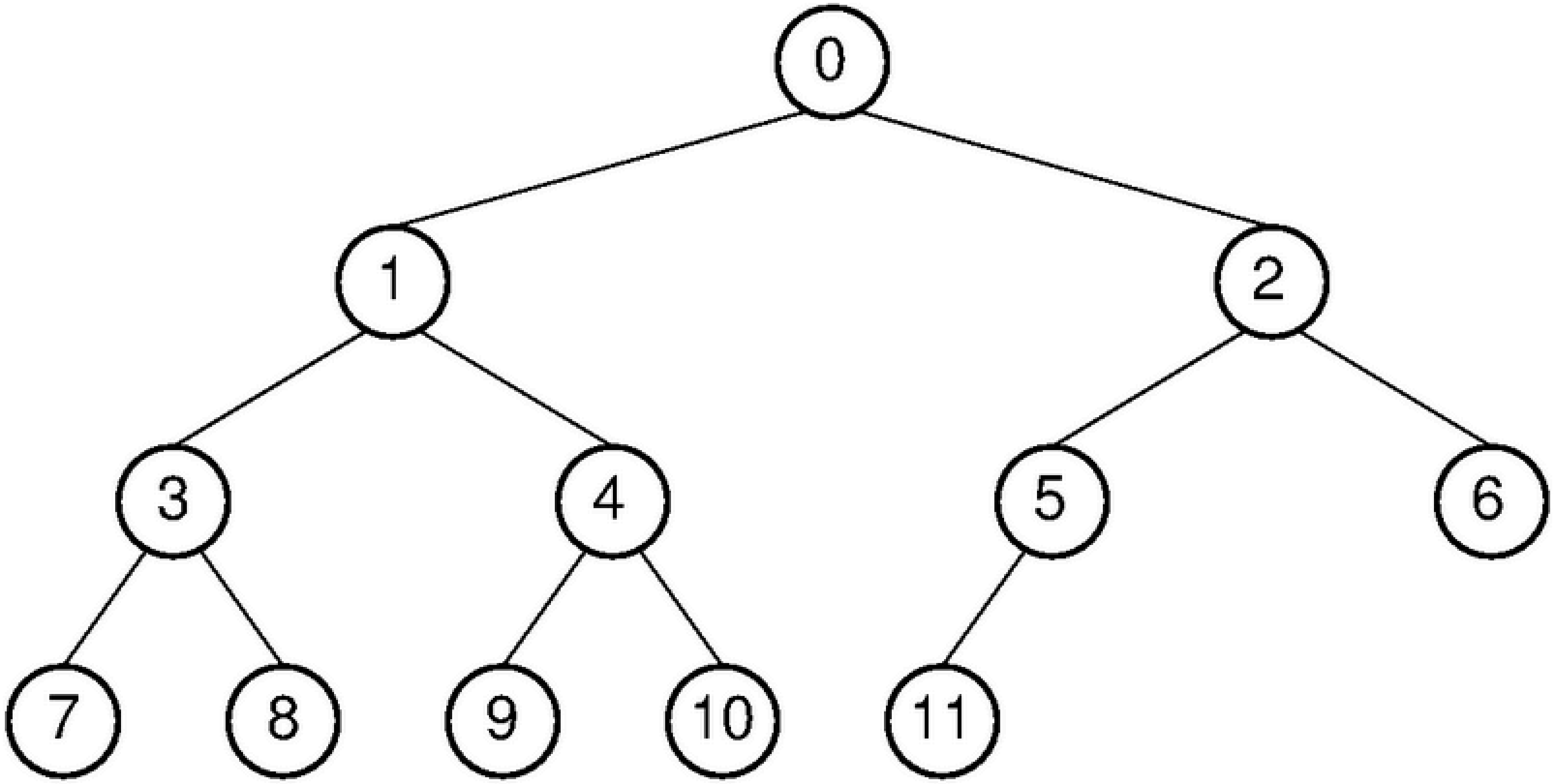
Complete Binary Tree as an Array

Since the shape of a complete binary tree is totally determined by its size, we don't have to actually store a tree with nodes.

Records are stored in an array, using indices counted from top to bottom, left to right.

- The root is at index `0`
- The left child of the root is at index 1
- The right child of the root is at index 2
- The left child of the left child of the root is at index 3
- ...

Array Representation

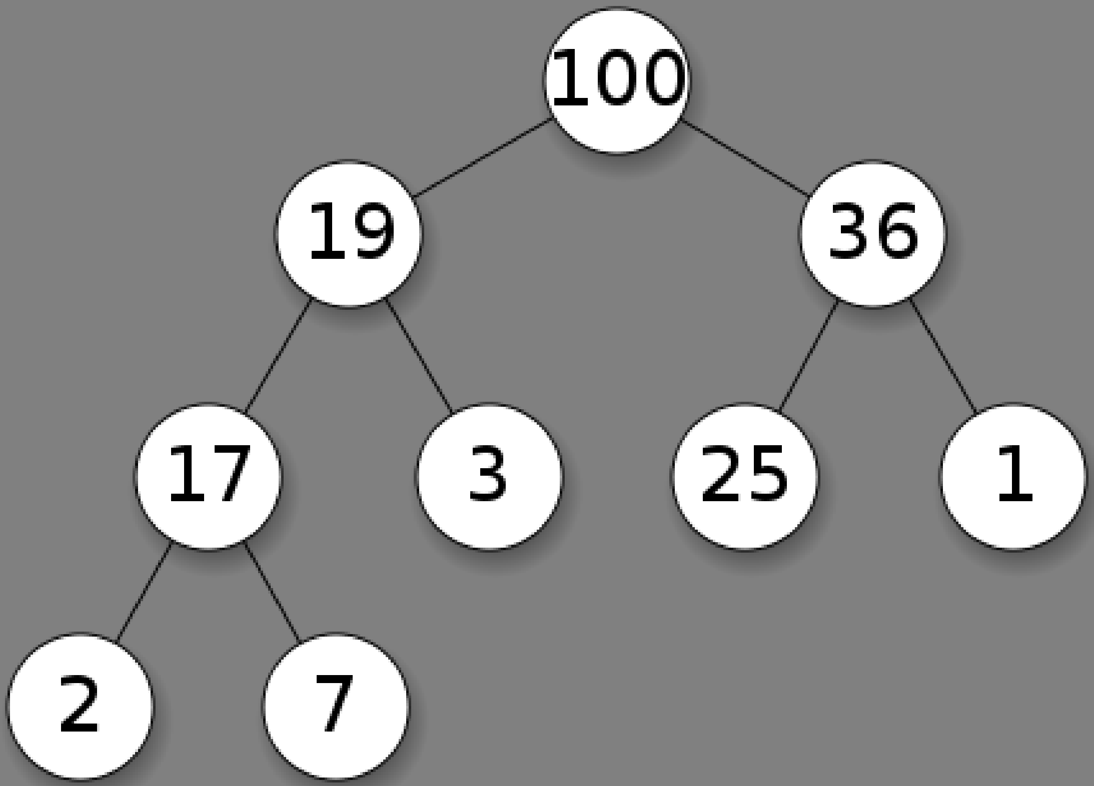


(a)

0	1	2	3	4	5	6	7	8	9	10	11

Exercises

1. Draw the tree-based representation of the complete binary tree that is represented by the array `[6 4 5 3 1 2]`
2. Come up with the array representation of the complete binary tree on the right.



Ancestors, Descendants, and Siblings

Given a node at position i in the array representation of a complete binary tree of n nodes,

- its parent is at position $\lfloor \frac{i-1}{2} \rfloor$
- its left child is at position $2i + 1$
- its right child is at position $2i + 2$
- its left sibling is at position $i - 1$ if i is even and $i \neq 0$
- its right sibling is at position $i + 1$ if i is odd and $i + 1 < n$

Takeaways

- We can represent a complete binary tree as an array
  - no pointer overhead
  - fast traversal using simple arithmetic
- We can resize the underlying array as needed
  - keep unused array space as low as possible
  - expand when needed, but amortization keeps time cost low

Binary Heaps

- Invented by J. W. J. Williams in 1964.
- Complete Binary Trees with additional rules about ordering.
- Implemented using the complete binary trees array representation.
- Values in a heap are *partially ordered*
- The ordering relationship is between the value stored at any node and the values of its children

Binary Heaps

Two types of heaps

- Max Heap: every node stores a value that is **greater than or equal** to the value of either of its children.
    - The root stores the maximum of all values in the tree.
- Min Heap: every node stores a value that is **less than or equal** to that of its children
    - The root stores the minimum of all values in the tree.

No guarantees about ordering relative to siblings!

Binary Heap & Priority Queue

- Priority Queues only require us to know the highest priority item—good for a partially sorted Data Structure like a Heap.
- Min/Max Heap Operations:
    - `add`
    - `poll`
        - Min Heap: the smallest value
        - Max Heap: the largest value
    - `heapify`: generate a min/max heap from an array of values

Conclusion: use a heap to implement a priority queue.

Friday's Recitation Activity

Rules of engagement:

- 30 minutes at end of recitation
- No leaving early! Sit still, please.
- Don't sit next to each other. Don't use your phone.

Topics:

1. Encoding/decoding a message using a Huffman Tree
2. Tree Traversal
3. Array—Tree Duality of a Heap
4. Executing Heap Operations

How to Maintain the Heap Property?

Two crucial operations, `siftUp` and `siftDown`, are used to maintain the heap property.

- Efficient operations that take **local** violations of the heap property and fix them.
- They are used in `add` and `poll` operations.
- They also allow us to modify the priority of an existing item and find a new home for it.

Binary Heap: `siftUp`

As long as a node is greater than its parent…

- swap it with its parent
- `siftUp` on the same node in the new location

```java
private void siftUp(int index) {
    while (index > 0) {
        int parent = parent(index);
        Buyer key = heapArray[index];
        Buyer parentKey = heapArray[parent];

        // Check for max heap property violation
        if (key.compareTo(parentKey) <= 0) {
            return; // if this is <= parent, then no more sifting to do
        } else { // otherwise, swap this with parent and keep moving up
            heapArray[index] = parentKey;
            heapArray[parent] = key;
            index = parent;
        }
    }
}
```

`siftUp` Runtime Analysis:

- Each swap is a constant time operation
- only as many swaps as the depth of the heap in the worst case
- depth of a complete binary tree is $O(\log n)$

$\Rightarrow$ `siftUp` is $O(\log n)$

Binary Heap: `siftDown`

As long as a node is less than one of its children

- swap it with its greater child
  - this way, each local swap maintains the heap property
- `siftDown` on the same node in the new location
  - restore the global heap property

Runtime analysis:

- Each swap is a constant time operation
- only as many swaps as the depth of the heap in the worst case

$\Rightarrow$ `siftDown` is $O(\log n)$

Heap: add

- Put the new value at the end of the array
- Increment the size of the heap
- call siftUp on that new value
- recurse upwards as necessary

Runtime analysis: $O(\log n)$ in the worst case.

Heap: add

```
public boolean add(E e) {
   // resize if needed
   if (size == heapArray.length) {
       grow();
   }
   // Add the new value to the end of the array
   heapArray[size++] = e;
   // sift up from the index we added to
   siftUp(size - 1);
   return true;
}
```

Heap: remove

- Swap the value at the front of the array with the value at the end of the array
- Decrement heap size
- Call siftDown on that root
- Recurse downwards as necessary

Runtime analysis: $O(\log n)$ in the worst case.

Heap: poll

```java
public E poll() {
    // Save the max value from the root of the heap
    E maxValue = (E) heapArray[0];
    // Move the last item in the array into index 0
    E replaceValue = (E) heapArray[--size];
    if (size > 0) {
        heapArray[0] = replaceValue;
        // Sift down to restore max heap property
        siftDown(0);
    }
    return maxValue;
}
```

Activity

1. Draw the MinHeap that results when you insert items with keys 3, 4, 1, 6, 9, 5, 7, 2
2. Rebuild the above tree after removing the (smallest) value at the top of the heap
3. Using the new heap, remove the value at position 1

Array Heapification

Goal: build a heap from any array of Comparable records

Implementation: For each record in the first half of the array, `siftDown` the record

- We can avoid calling `siftDown` on the second half of the array.
  - Everything in the second half is a leaf, which is already a valid heap.
- This builds a heap.
  - `siftDown` creates a valid heap rooted at the location we call it on *provided that the left and right subtrees are also heaps*
  - If we `siftDown` starting on the last non-leaf node and work our way up, the heap property is maintained at each step.

Array Heapification Runtime Analysis

(Assume a *full and complete* binary tree for ease of analysis)

- `siftDown()` requires $h - 1 \in O(h)$ many swaps, where h is the height of the heap
- At any height h, the number of nodes is at most $\frac{n}{2^h}$
    - There is one node at the root, two nodes at the next level, four nodes at the next, etc.

Overall, the number of swaps to `siftDown` all non-leaves is:

$$O(\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i}) = O(\frac{n}{2}\sum_{i=1}^{\log n}\frac{i-1}{2^{i-1}}) = O(n)$$

Corrolaries

- `heapify` is $O(n)$
    - Better to gather all records in an array and then `heapify` the array once than to `insert` records one by one (Why?)
- `heapSort` is a valid $O(n \log n)$ sorting algorithm
    - `poll` the root of the heap and `siftDown` the last record to the root
    - Repeat until the heap is empty
    - The records will be in ascending order
- We can build our ticketing priority queue!

Designing a Ticketing Queue

- Allow n people to enter a ticket queue for t tickets
- Create a heap of size n and `heapify` it.
- `poll` the heap t times to get the t highest priority people and offer them each a chance to buy a ticket
- repeat until you've sold all tickets or the heap is empty

$O(n + t\log n)$ runtime, which is better than the $O(n\log n)$ runtime of a TreeSet. 😄