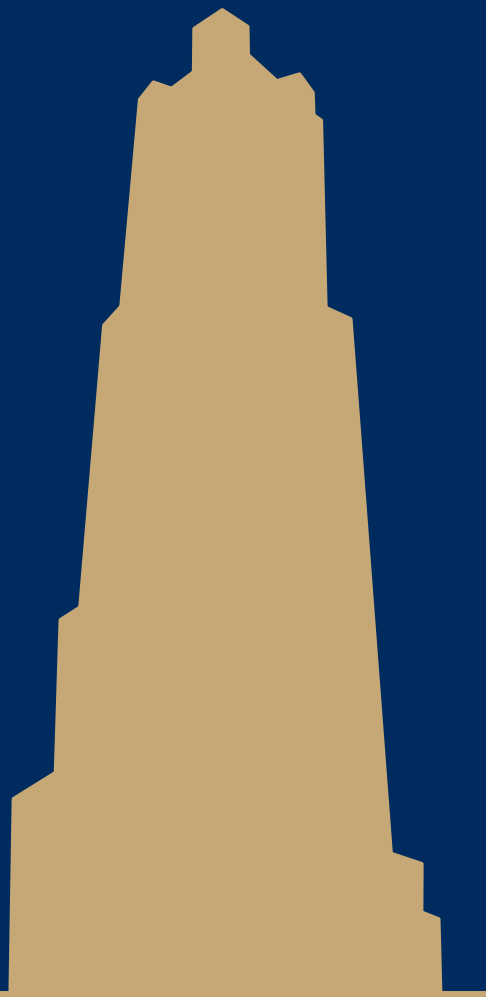
CS 1501

Priority Queues



The Priority Searching Problem

Given a collection of items, and the ability to determine the relative priority of any two items, return an item with the highest priority

We mentioned priority queues in building Huffman tries

- Primary operations they needed:
  - Insert
  - Find item with highest priority
    - E.g., `findMin()` or `findMax()`
  - Remove an item with highest priority
    - E.g., `removeMin()` or `removeMax()`
- How do we implement these operations?
  - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - O(1)
- Find:
  - Search for the highest priority item (e.g., min or max)
  - O(n)
- Remove:
  - Search for the highest priority item and delete
  - O(n)

- Runtime for use in Huffman tree generation?

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - O(n)
- Find:
  - Return the item at the end of the array
  - O(1)
- Remove:
  - Return and delete the item at the end of the array
  - O(1)

- Runtime for use in Huffman tree generation?

So what other options do we have?

- What about a binary search tree?
  - Insert
    - Average case of O(lg n), but worst case of O(n)
  - Find
    - Average case of O(lg n), but worst case of O(n)
  - Remove
    - Average case of O(lg n), but worst case of O(n)
- OK, so in the average case, all operations are O(lg n)
  - No constant time operations
  - Worst case is O(n) for all operations

What about a red-black BST?

- Seems like overkill...
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
  - Can we take advantage of this to get efficient performance with a simpler implementation?
    - Yes!

The heap

- A heap is complete binary tree such that:
  - For each node T in the tree:
    - T.item is of a higher priority than T.right_child.item
    - T.item is of a higher priority than T.left_child.item
- It does not matter how T.left_child.item relates to T.right_child.item

The *heap property*

Heap PQ runtimes

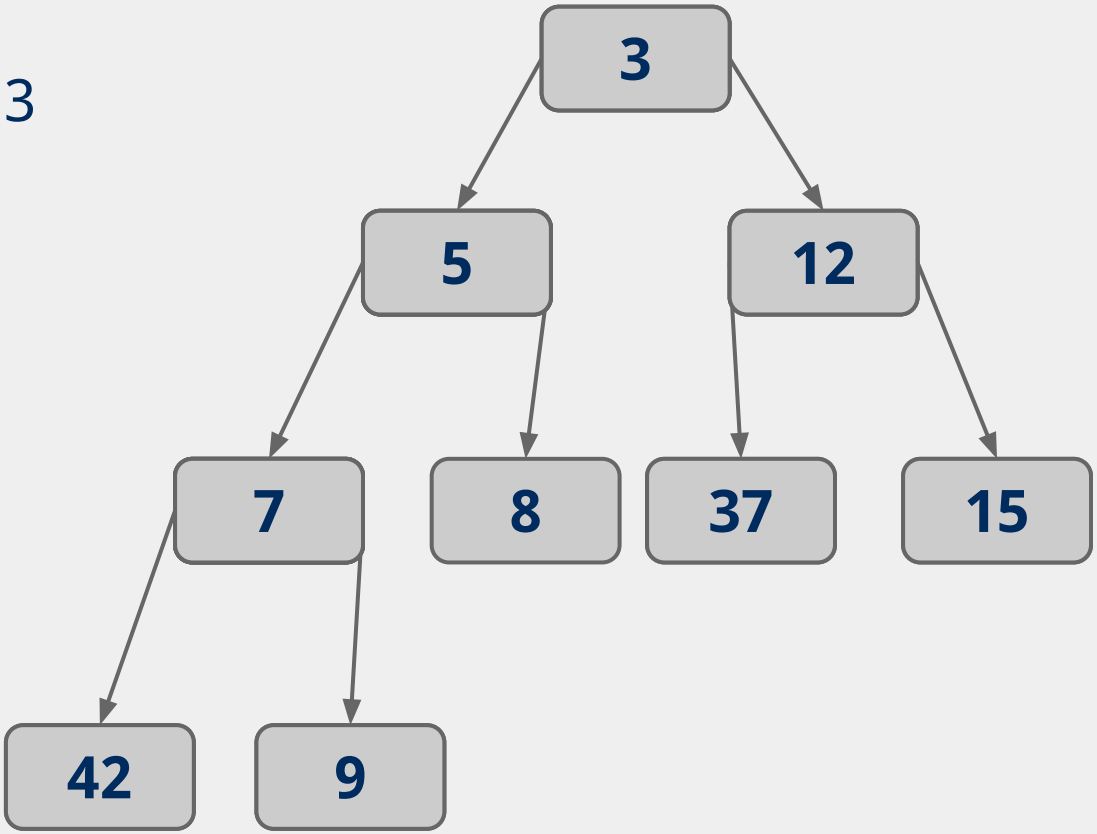
- Find is easy
  - Simply the root of the tree
    - O(1)
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

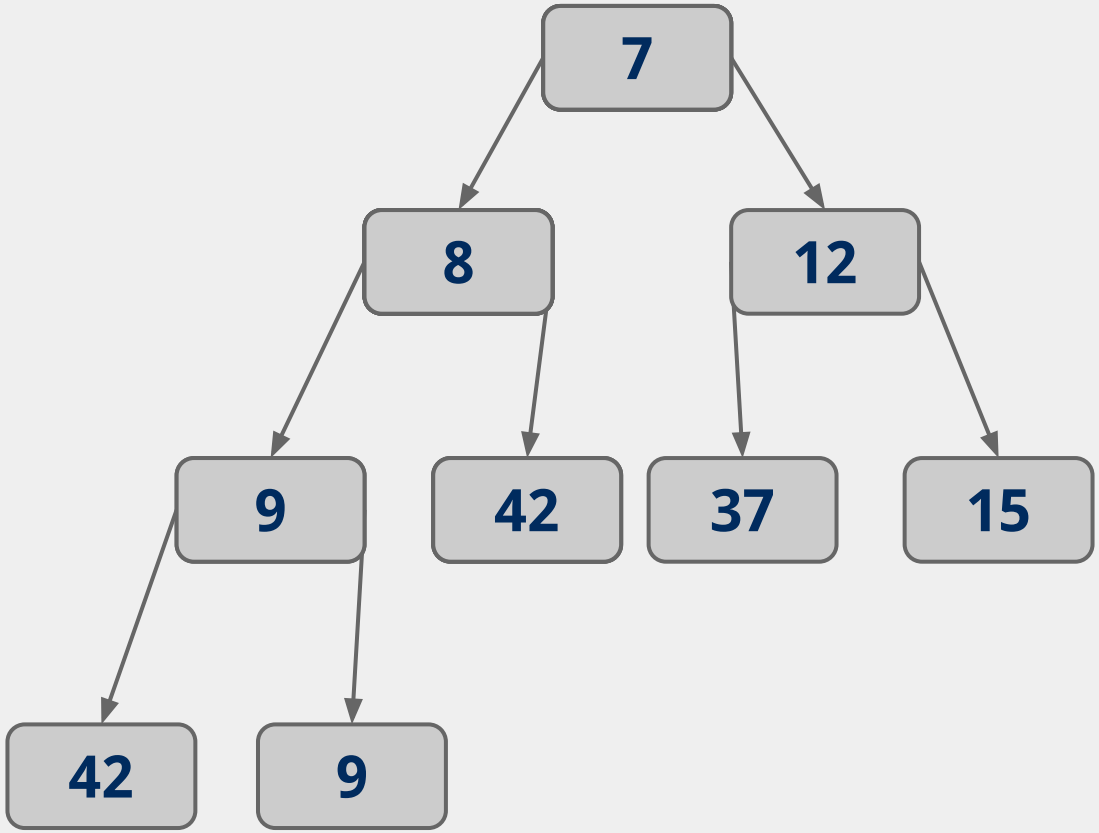


Heap remove

- Tricky to delete root…
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property…
      - So we push the root down the tree until it is supporting the heap property

Min heap removal

NO!



Heap runtimes

- Find
  - O(1)
- Insert and remove
  - Height of a complete binary tree is lg(n)
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are O(lg n)

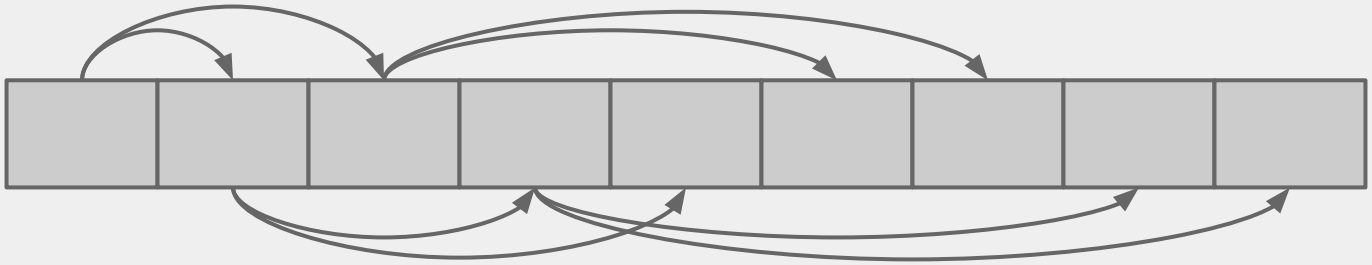
Heap implementation

- Simply implement tree nodes like for BST
  - This requires overhead for dynamic node allocation
  - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
  - We can easily represent a complete binary tree using an array

Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = ⌊(i - 1) / 2⌋
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

For arrays indexed from 0



The Sorting Problem

Given a collection of items, produce a collection with items arranged in a sorted order

Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - O(n lg n)
- In-place?
  - Yes
- Stable?
  - No

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice:
    def __init__(self, store, price):
        self.store = store
        self.price = price
    def compareTo(other):
        if self.price < other.price:
            return -1
        elif self.price > other.price:
            return 1
        else:
            return 0
```

The Priority Searching Problem

Given a collection of items, and the ability to determine the relative priority of any two items, return an item with the highest priority

Storing Objects in PQ

- What if we want to update an Object?
  - What is the runtime to find an arbitrary item in a heap?
    - O(n)
    - Hence, updating an item in the heap is O(n)
  - Can we improve on this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

Indirection example

- `n = CardPrice("NE", 333.98);`
- `a = CardPrice("AMZN", 339.99);`
- `g = CardPrice("GME", 338.00);`
- `b = CardPrice("BB", 349.99);`

- Update price for NE: 340.00
- Update price for GME: 345.00
- Update price for BB: 200.00

Indirection

"NE":2
"AMZN":1
"GME":3
"BB":0