Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
  - maxHeapify()
  - buildMaxHeap()
  - removeMax() and insert()
- Heapsort
Why Heaps?

- Often, we just want to know the minimum or maximum in a dataset:
  - Prim's Algorithm
  - Dijkstra's Algorithm
  - Event dispatch (simulation or OS)
  - Priority queue (job scheduler)
Why Heaps?

• Often, we just want to know the minimum or maximum in a dataset:
  – Prim's Algorithm
  – Dijkstra's Algorithm
  – Event dispatch (simulation or OS)
  – Priority queue (job scheduler)

• A heap is a data structure which always keeps track of the max (or min) key
Heaps

• A heap is:
  - A binary tree (not a search tree!)
  - “Nearly complete”
    • Allows an array representation!
  - Each parent is greater or equal to than both of its children (or less, in a min-heap)

Don't confuse the heap data structure with the heap memory region!
“Nearly Complete” Binary Trees

- “Nearly complete” binary trees:
  - Fill in all levels but the bottom
  - Fill in the bottom from left-to-right
“Nearly Complete” Binary Trees

- If we number the nodes from the root down, a cool property develops...
“Nearly Complete” Binary Trees

- If we number the nodes from the root down, a cool property develops...
  - Compare each node's # to that of its two children
    - Parent: $k$
    - Left: $2k + 1$
    - Right: $2k + 2$
“Nearly Complete” Binary Trees

- If we number the nodes from the root down, a cool property develops...
  - Compare each node's # to that of its two children
    - Parent: $k$
    - Left: $2k + 1$
    - Right: $2k + 2$

**Induction Practice:**
Can you prove that this holds for the leftmost node on each level?
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- **Heaps stored in Arrays**
- The Heap Property
  - `maxHeapify()`
  - `buildMaxHeap()`
  - `removeMax()` and `insert()`
- Heapsort
Heaps and Arrays

- We can store a “nearly complete” binary tree in an array
  - No holes, no wasted space
Heaps and Arrays

• No need for pointers, just do index arithmetic
Heaps and Arrays

- No need for pointers, just do index arithmetic
Heaps and Arrays

- No need for pointers, just do index arithmetic

Something to Note:
Recursive algorithms always need to know the full array.

The tree rooted at node 7 does not have the same layout as the tree rooted at node 0!
Heaps and Arrays

NOTE

From here on, we'll use arrays OR trees to represent a heap.

Either is valid, we'll just use whatever makes the algorithm clearest.

WARNING

We've used numbers to represent node #s so far.

From here on, we'll use them to represent the keys stored at given nodes.
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
  - maxHeapify()
  - buildMaxHeap()
  - removeMax() and insert()
- Heapsort
The Heap Property

- In a **max-heap**, every node has a key which is \( \geq \) the keys in both children.
  - A **min-heap** simply reverses the comparison.
The Heap Property

- Duplicates are allowed (even parent-to-child)
The Heap Property

- There is **no defined order** between sibling nodes (unlike a BST!)

When I drew this picture, I naturally (and subconsciously) put larger keys to the left...except in one place.

The Heap doesn't care!
The Heap Property

- Therefore, an “in-order walk” of a heap really doesn't work.
The Heap Property

- We can find the max (or min, in a min-heap) in $O(1)$ time. But finding the other extreme requires a brute-force search...
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
- \texttt{maxHeapify()}
- \texttt{buildMaxHeap()}
- \texttt{removeMax()} and \texttt{insert()}
- Heapsort
Building Heaps

So how do we build a heap?

- First, we'll define a function to move a key down through the tree, to find its proper place: $O(\lg n)$
- Then we'll apply this function $n/2$ times
- And somehow end up with an $O(n)$ algorithm!
maxHeapify()

- The `maxHeapify()` function moves a given key down through the tree until it satisfies the heap property
  - Assumes that both children (if they exist) are the roots of max-heaps
  - If heap property doesn't hold at current node, swap parent with one child, then recurse into the child
  - Terminates when heap property holds OR we move the key into a leaf node
maxHeapify()

We run `maxHeapify()` at the root (key=10).

We assume that the heap property already holds in both subtrees.
We run `maxHeapify()` at the root (key=10).

The heap property does not hold at the root.
We find that 99 is the max of the two children.
maxHeapify()

We swap the parent and the child; this establishes the heap property at the parent node.
maxHeapify()

Discussion Q:

Why doesn't this run the risk of violating the heap property further up in the tree (that is, if we weren't at the root)?

Of course, this can violate the heap property lower down.
maxHeapify()
maxHeapify()

Always swapping with the max child...
maxHeapify()

Always swapping with the max child...
maxHeapify()

...until the heap property is re-established.

If we hit a leaf, then the heap property is trivially true.
maxHeapify()

- maxHeapify() runs in $O(h)$ time
  - 2 comparisons, 1 swap per step:
    - $O(1)$ time per step
  - $O(h)$ steps

- Since heaps are always “nearly complete” trees, $O(h) = O(\log n)$
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
- `maxHeapify()`
- `buildMaxHeap()`
- `removeMax()` and `insert()`
- Heapsort
The `buildMaxHeap()` function turns an unsorted array into a max-heap

- Treats the lower half of the array as leaves
  - Each leaf trivially fulfills the heap property
- Starting at the **rightmost** (that is, deepest) internal node, run `max_heapify()` on every internal node
  - Values filter down in the direction of leaves
  - When it returns, the internal node is root of a subtree which is a max-heap
BuildMaxHeap()

Start with an unsorted array of values.

This picture shows both the tree and the array, for clarity. Of course, there is only one copy in reality.
Each leaf is a (trivial) max-heap.
We run `maxHeapify()` at node 6.

It obviously will need to swap with its (only) child.
When `maxHeapify()` terminates, we have a max-heap at that node.
We work our way backwards through the array.

```
buildMaxHeap()
```

```
1  2  3  4  5  11  6  7  8  9  10
```
We work our way backwards through the array.
buildMaxHeap()

We work our way backwards through the array.
We work our way backwards through the array.
BuildMaxHeap()

We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
buildMaxHeap()

We work our way backwards through the array.
buildMaxHeap()

At the end, we have a max-heap.
A trivial analysis would assume that \texttt{buildMaxHeap()} \textbf{takes} \(O(n \ lg \ n)\).

In truth, it takes \(O(n)\)!

- We won't be going through the proof here.
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
- `maxHeapify()`
- `buildMaxHeap()`
- `removeMax()` and `insert()`
- Heapsort
The `removeMax()` function removes the root, and then rebuilds the heap:
- Move the last element (last leaf) up to the root
- Use `maxHeapify()` to push it to the correct location
To remove the maximum, we first extract the maximum key, then move the last key up to the root.
removeMax()
When we remove an element, we shorten the `heapSize` variable.

Typically, we don't actually change the length of the array – we just ignore the last slot.
removeMax()

Max: 12

Note that (except for the root), the entire tree is still a valid max-heap.
We use `maxHeapify()` to move the 3 down from the root to its proper location.
We use maxHeapify() to move the 3 down from the root to its proper location.
We use maxHeapify() to move the 3 down from the root to its proper location.
We use maxHeapify() to move the 3 down from the root to its proper location.
The `insert()` function adds a new leaf at the end, then percolates it up:

- Analogous to `maxHeapify()`, but just in reverse
- Only have to compare each node to its parent.
We insert the new value at the last position.
We compare its value to its parents, recursively, until we either hit the root or find a parent $\geq$ the value.
We swap as we recurse up.
We swap as we recurse up.
We swap as we recurse up.
Efficiency Warning

WARNING

Always use `maxHeapify()` - that is, push down – instead of `insert()` - that is, push up – when implementing `buildMaxHeap()`.

`maxHeapify()` will give you $O(n)$ performance.

`insert()` will give you $O(n \ lg \ n)$ performance.
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
  - maxHeapify()
  - buildMaxHeap()
  - removeMax() and insert()
- Heapsort
Heapsort

Could we use a heap to perform a sort?

It's Heapsort!

- Good:
  - $O(n \log n)$ worst case – as good as Merge Sort
  - Requires no auxiliary space – like Quicksort

- Bad:
  - Not stable
  - A little slower than Quicksort in practice (typically)
Heapsort

How does HeapSort work?

- Use $\text{buildMaxHeap()}$ to convert the array into a max-heap
- Remove max value, store it at the tail of the array
  - Loop, with the array holding both the (shrinking) heap and the (growing) array
Heapsort

Imagine that this is the state of the array right after a heap was constructed.

The heap fills the entire array.
Heapsort

We remove the maximum as before. We shift the last value up to the root, then run `maxHeapify()` to push it back down.
Heapsort

Max: 100

We remove the maximum as before. We shift the last value up to the root, then run `maxHeapify()` to push it back down.
But now, we **re-insert** 100 at the end of the array.

We don't think of this as part of the heap anymore – it is a sorted subarray.
Heapsort

Max: 11

We repeat this with every element in the heap – remove the max, then add to the array at the end.
Heapsort

Max: 11

We repeat this with every element in the heap – remove the max, then add to the array at the end.
We repeat this with every element in the heap – remove the max, then add to the array at the end.
Heapsort

We repeat this with every element in the heap – remove the max, then add to the array at the end.
We repeat this with every element in the heap – remove the max, then add to the array at the end.
Heapsort

This continues...until every element has been removed, and “added” to the sorted subarray.
Heapsort

• Overall:
  - $O(n)$ to run buildMaxHeap()
  - $O(n \ lg \ n)$ to remove the maximums and store them in the upper array

• Thus:
  - $O(n \ lg \ n)$ in the **worst case**
Topic 12: Heaps and Heapsort

- Heaps as Binary Trees
- Heaps stored in Arrays
- The Heap Property
- maxHeapify()
- buildMaxHeap()
- removeMax() and insert()
- Heapsort

Summary