Lecture 4: Heaps and Priority Queues

Menu
- Priority Queues
- Heaps
- Heapsort (different from Insertion or Merge Sort seen in prior lectures)

Priority Queue
A data structure that implements a set of elements, each associated with a key. We can define it as an abstract data type (for those who have taken CS005). The object is defined via the set of allowable histories (sequences of operations), maintaining an invariant.

N.B. →

*Insert*(S, x): insert element x into set S
*max*(S): return element of S with largest key
*extract-max*(S): return element of S with largest key and remove it from S.

Increase-key(S, x, k): increase the value of x's key to new value k. (Assumed to be at least as large as current value)

Heap
(Implements a priority queue)
An array, visualized as a binary tree (nearly complete)

```
1 2 3 4 5 6 7 8 9 10
```

16 14 10 8 7 9 3 2 4 1

So we can do arbitrary arrays
Heap representation of an array.

N.B. →

Heap as a tree

- Root: first element in the array (i=1)
- Parent(i) = \( \lfloor i/2 \rfloor \) (index of parent) (should be \( \lfloor i \rfloor \) to be accurate)
- Left(i) = 2i (left child)
- Right(i) = 2i + 1 (right child)

So no pointers needed...

Max-Heap properties:
- The key of a node is \( \geq \) the keys of its children

N.B. →

- Is the heap above a max-heap?
  - Max-heap similar
- What is the trivial one heap \( \rightarrow \) max
- Is extract-max trivial? No \( \rightarrow \) need to maintain invariant
- If we do extract-max \( n \) times we get a sorted sequence

Big question: how?

Example of array that is not max heap or min heap

How do you take an array and turn it into a max heap?
Heap Operations

build-max-heap: produce a max-heap from an unordered array

max-heapify: correct a single violation of the heap property at root of subtree.

N.B.

Max-heapify(A, i)

- Assume that trees rooted at left(i) and right(i) are max-heaps.
- Then if A[i] violates max-heap property, correct it by "trickling" A[i] down the tree, making subtree rooted at i a max heap.

Max-heapify(A, 2)

work bottom up

Max-heapify(A, 3)
Done

What is the complexity of max-heapify?
1. It's a balanced binary tree! \( O(\log n) \) depth
2. Single violation (subtrees & violation one max-heaps)
   So time complexity is \( O(\log n) \).

N.B. →

**Build-max-heap** \( (A) \)

Converts \( A[1..n] \) into a max-heap

**Build-max-heap** \( (A) \):

For \( i = \lfloor n/2 \rfloor \) down to 1
   do **Max-Heapify** \( (A, i) \)

Why start at \( n/2 \)?
Because elements \( A[\lfloor n/2 + 1 \rfloor, \ldots, n] \) are all leaves.

**Time?** \( O(n \log n) \) via simple analysis

- About \( O(\log n) \) at worst per any of the top \( \lfloor n/2 \rfloor \) nodes
  - So \( \lfloor n/2 \rfloor \cdot O(\log n) = O(n \log n) \)

Can we do a better analysis of this algorithm?  

\( O(n) \) → first level \( O(1) \), second \( O(2) \), ..., \( \log n \) level \( O(\log n) \)

\( n/4 \) \( n/8 \) \( n/2 \log n \) 1 node

Observe: **Max-Heapify** takes \( O(1) \) for nodes one level above leaves and in general \( O(c) \) time for nodes that are \( c \)-levels above leaves.

\( n/4 \) with level 1, \( n/8 \) with level 2, 1 node with level \( \log n \)
Total amount of work in the for-loop can be summed as:

\[ \frac{n}{4}(c_1) + \frac{n}{8}(c_2) + \frac{n}{16}(3c) + \ldots + \frac{n}{2^n}(\log c) \]

Setting \( \frac{n}{4} = 2^k \)

\[ c \cdot 2^k \left( \frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \ldots + \frac{k+1}{2^k} \right) \]

arithmetic expression bounded by a constant \( c_3 \)

So build-max-heap is \( \Theta(n) \)!

**Heap-sort:**

1. Build-max-heap from unordered array
2. Find minimum element \( A[1] \)
4. Discard node \( n \) from heap (by decreasing heap-size variable).
5. New root may violate max heap property, but its children are max heaps. Run max-heapify to fix this.
6. Go to step 2 unless heap is empty.
So heap-sort takes $O(n \log n)$

Sorts in place...