Lecture 8: Binary Heaps

Priority Queues

- New interface which implements a limited Set (intrinsic keyed order)
- Efficiently remove the most important item (highest key), optimize for **build** and **space**
  - Example: router with limited bandwidth, must prioritize certain kinds of messages; process scheduling in operating system kernels; graph algorithms (later in the course)
- Priority queue operations:

`build(A)`	build priority queue from iterable `A`
`insert(x)`	add item x to priority queue
`delete_max()`	remove and return the stored item with largest key

- (Usually max, can make a minimum priority queue via negation)
- Delete max only new operation: can be simulated via `find_max` followed by a `delete`
- (If know $n < m$ can get worst-case: array of size m, store length n of priority queue prefix)

Priority Queue Sort

- **Additional Goal:** in-place $O(n \log n)$ sorting algorithm
- Given a priority queue, we can use it to sort
- Build the priority queue on `A` (or insert items one at a time)
- Repeatedly remove and return the maximum element
- If `build`, `insert`, `delete_max` have running times B, I, D, runs in $\min(B, n \cdot I) + n \cdot D$

Priority Queue Data Structure	Operations $O(\cdot)$			Priority Queue Sort	
	`build(A)`	`insert(x)`	`delete_max()`	Time	In-place?
Dynamic Array	n	$1_{(a)}$	n	n^2	Y
Sorted Dynamic Array	$n \log n$	n	$1_{(a)}$	n^2	Y
Balanced Binary Tree	$n \log n$	$\log n$	$\log n$	$n \log n$	N
Goal	n	$\log n_{(a)}$	$\log n_{(a)}$	$n \log n$	Y

Priority Queue Sort: Array

- Maintain the first k items as a priority queue implemented with an **unsorted array**
- `insert` take no time, just increase k by 1 to incorporate next item
- To `delete_max`, find max via linear search, swap to end and decrease k by 1
- `insert` is quick $O(1)$, but `delete_max` is slow $O(n)$, runs in $O(n^2)$
- This is exactly selection sort!

Priority Queue: Sorted Array

- Maintain the first k items as a priority queue implemented with a **sorted array**
- `insert` takes linear time to put next item $k + 1$ in correct place in sorted order
- `delete_max` takes no time (max is already at end) so just decrease k by 1
- `insert` is slow $O(n)$, but `delete_max` is slow $O(\log n)$, runs in $O(n^2)$
- This is exactly insertion sort!

Balance Insert/Delete In-place

- Can we balance the cost of insertions and deletions?
- Yes! Use balanced binary tree: insert/delete in $O(\log n)$ time, sort in $O(n \log n)$ time
- Not in place (build a linked tree)
- Can we get $O(n \log n)$ sorting **in-place**, i.e., using at most $O(1)$ additional space?

Array as a Complete Binary Tree

- **Idea:** interpret array (or array prefix) as a left-aligned complete binary tree
- A binary tree is **complete** if it is fully balanced: every level except last is full
- A binary tree is **left-aligned** if leaves in last level are packed to the left
- A binary tree is **packed** if it is complete and left-aligned
- There is exactly **one** left-aligned complete binary tree on n nodes for any n

```
A = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
    |                   |
    |    heap prefix    | k = 9
    |                   |
d0   0                     ->        _______0_____
d1     1 2                 ->    _____1_____  __2__
d2         3 4 5 6         ->   __3__   __4   5   6
d3                 7 8 9   ->   7   8   9
```

- So there is a **bijection** between arrays and packed binary trees
- Height of packed tree perspective of array of n item is $\Theta(\log n)$
- Implicit tree: compute parent/left/right by index arithmetic (no need to store pointers!)
  - Root is at index 0
  - Parent of index i is at index $\left\lfloor \frac{i-1}{2} \right\rfloor$
  - Left child of index i is at index $2i + 1$
  - Right child of index i is at index $2i + 2$

Binary Heaps

- **Idea:** keep larger elements higher in tree, but only locally
  - Node max-heap property (MHP) at i: $Q[i] \geq Q[\text{left}(i)], Q[\text{right}(i)]$
  - Tree max-heap property (Tree MHP) at i: $Q[i] \geq Q[j]$ for every index $j \in S(i)$
- A **max-heap** is an array where every node satisfies the **node** MHP
- **Claim:** Every node in max-heap satisfies the **tree** MHP
- Proof:
  - Claim: If j is in subtree $S(i)$ and $d = \text{depth}(j) - \text{depth}(i)$, then $Q[i] \geq Q[j]$
  - Induction on d:
  - Base case: $d = 0$ implies $i = j$ implies $Q[i] \geq Q[i]$
  - $\text{depth}(\text{parent}(j)) - \text{depth}(i) < d$, so $Q[i] \geq Q[\text{parent}(j)]$ by induction
  - $Q[\text{parent}(j)] \geq Q[j]$ by node MHP at $\text{parent}(j)$ $\square$
- In particular, if MHP everywhere, max item is at root

Binary Heap Insert

- Given array satisfying MHP, how to insert an element?
  - Append to end or expand prefix (next leaf)
  - Swap with parent until MHP satisfied!
- Correctness:
  - MHP assumes all nodes $\geq$ descendants, except $Q[c]$ might be $>$ some ancestors
  - If swap necessary, same assumption is true with $Q[c]$ swapped with $Q[p]$
- Running time: height of tree, so $O(\log n)$!

Binary Heap Delete Max

- Given array satisfying MHP, how to delete item with maximum key?
  - Swap root to end and remove it, or reduce prefix
  - Swap new root with its larger child until MHP satisfied!
- Correctness:
  - MHP assumes all nodes $\geq$ descendants, except $Q[p]$ might be $<$ some descendants
  - if swap is necessary, same property holds with $Q[p]$ swapped with $Q[c]$

Heap Sort

- Running time: height of tree, so $O(\log n)$!
- Can insert and delete max, each in-place and in $O(\log n)$ time
- Yields **heap sort**, an in-place $O(n \log n)$ comparison sorting algorithm

Linear Build Heap

- To insert n items, each item is heapified down from root, worst-case is $\Omega(n \log n)$ swaps:

$$\text{worst-case swaps is } \approx \sum_{i=0}^{n-1} \lg i = \lg(n!) = \Theta\left(\lg\left(\sqrt{n}(n/e)^n\right)\right) = \Omega(n \log n)$$

- **Idea!** Treat full array as a packed binary tree from start, then fix MHP from leaves to root

$$\text{worst-case swaps is } \approx \sum_{i=0}^{n-1} (\lg n - \lg i) = \lg\left(\frac{n^n}{n!}\right) = \Theta\left(\log\left(\frac{n^n}{\sqrt{n}(n/e)^n}\right)\right) = O(n)$$

- Uses at most $O(n)$ swaps, so the heap can be built in linear time