Lecture 8: Binary Heaps

Priority Queue Interface

- Keep track of many items, quickly access/remove the most important
  - Example: router with limited bandwidth, must prioritize certain kinds of messages
  - Example: process scheduling in operating system kernels
  - Example: discrete-event simulation (when is next occurring event?)
  - Example: graph algorithms (later in the course)
- Order items by key = priority so Set interface (not Sequence interface)
- Optimized for a particular subset of Set operations:
  - `build(X)`: build priority queue from iterable $X$
  - `insert(x)`: add item $x$ to data structure
  - `delete_max()`: remove and return stored item with largest key
  - `find_max()`: return stored item with largest key
- (Usually optimized for max or min, not both)
- Focus on `insert` and `delete_max` operations: `build` can repeatedly `insert`; `find_max()` can `insert(delete_min())`

Priority Queue Sort

- Any priority queue data structure translates into a sorting algorithm:
  - `build(A)`, e.g., insert items one by one in input order
  - Repeatedly `delete_min()` (or `delete_max()`) to determine (reverse) sorted order
- All the hard work happens inside the data structure
- Running time is $T_{build} + n \cdot T_{delete_max} \leq n \cdot T_{insert} + n \cdot T_{delete_max}$
- Many sorting algorithms we’ve seen can be viewed as priority queue sort:

<table>
<thead>
<tr>
<th>Priority Queue Data Structure</th>
<th>Operations $O(\cdot)$</th>
<th>Priority Queue Sort</th>
<th>Time</th>
<th>In-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Array</td>
<td>$n$</td>
<td>$1_{(a)}$</td>
<td>$n^2$</td>
<td>Y</td>
</tr>
<tr>
<td>Sorted Dynamic Array</td>
<td>$n \log n$</td>
<td>$n$</td>
<td>$1_{(a)}$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Set AVL Tree</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Goal</td>
<td>$n$</td>
<td>$\log n_{(a)}$</td>
<td>$\log n_{(a)}$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>

Selection Sort | Insertion Sort | AVL Sort | Heap Sort
Priority Queue: Set AVL Tree

- Set AVL trees support \( \text{insert}(x) \), \( \text{find\_min}() \), \( \text{find\_max}() \), \( \text{delete\_min}() \), and \( \text{delete\_max}() \) in \( O(\log n) \) time per operation.
- So priority queue sort runs in \( O(n \log n) \) time.
  - This is (essentially) AVL sort from Lecture 7.
- Can speed up \( \text{find\_min}() \) and \( \text{find\_max}() \) to \( O(1) \) time via subtree augmentation.
- But this data structure is complicated and resulting sort is not in-place.
- Is there a simpler data structure for just priority queue, and in-place \( O(n \log n) \) sort?
  - YES, binary heap and heap sort.
- Essentially implement a Set data structure on top of a Sequence data structure (array), using what we learned about binary trees.

Priority Queue: Array

- Store elements in an unordered dynamic array.
- \( \text{insert}(x) \): append \( x \) to end in amortized \( O(1) \) time.
- \( \text{delete\_max}() \): find max in \( O(n) \), swap max to the end and remove.
- \( \text{insert} \) is quick, but \( \text{delete\_max} \) is slow.
- Priority queue sort is selection sort! (plus some copying).

Priority Queue: Sorted Array

- Store elements in a sorted dynamic array.
- \( \text{insert}(x) \): append \( x \) to end, swap down to sorted position in \( O(n) \) time.
- \( \text{delete\_max}() \): delete from end in \( O(1) \) amortized.
- \( \text{delete\_max} \) is quick, but \( \text{insert} \) is slow.
- Priority queue sort is insertion sort! (plus some copying).
- Can we find a compromise between these two array priority queue extremes?
Array as a Complete Binary Tree

- **Idea:** interpret an array as a complete binary tree, with maximum $2^i$ nodes at depth $i$ except at the largest depth, where all nodes are left-aligned

1. d0 ______ 0____
2. d1 _______0____ _______0____
3. d2 ____0___ ___0___ 0 ___0______
4. d3 0 ___0___ 0

- Equivalently, complete tree is filled densely in reading order: root to leaves, left to right

- Perspective: bijection between arrays and complete binary trees

1. Q = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2. d0 0 -> _______0____
3. d1 1 2 -> _______1____ _______2____
4. d2 3 4 5 6 -> ___3___ ___4___ 5 6
5. d3 7 8 9 -> 7 8 9

- Height of complete tree perspective of array of $n$ item is $\lceil \lg n \rceil$, so balanced binary tree

Implicit Complete Tree

- Complete binary tree structure can be implicit instead of storing pointers
- Root is at index 0
- Compute neighbors by index arithmetic:

\[
\text{left}(i) = 2i + 1 \\
\text{right}(i) = 2i + 2 \\
\text{parent}(i) = \left\lfloor \frac{i - 1}{2} \right\rfloor
\]
Binary Heaps

- **Idea**: keep larger elements higher in tree, but only locally

- **Max-Heap Property** at node $i$: $Q[i] \geq Q[j]$ for $j \in \{\text{left}(i), \text{right}(i)\}$

- **Max-heap** is an array satisfying max-heap property at all nodes

- **Claim**: In a max-heap, every node $i$ satisfies $Q[i] \geq Q[j]$ for all nodes $j$ in subtree($i$)

  - **Proof**:
    - Induction on $d = \text{depth}(j) - \text{depth}(i)$
    - Base case: $d = 0$ implies $i = j$ implies $Q[i] \geq Q[j]$ (in fact, equal)
    - $\text{depth}(\text{parent}(j)) - \text{depth}(i) = d - 1 < d$, so $Q[i] \geq Q[\text{parent}(j)]$ by induction
    - $Q[\text{parent}(j)] \geq Q[j]$ by Max-Heap Property at parent($j$)

- In particular, max item is at root of max-heap

---

Heap Insert

- Append new item $x$ to end of array in $O(1)$ amortized, making it next leaf $i$ in reading order

  - **max_heapify_up($i$)**: swap with parent until Max-Heap Property
    - Check whether $Q[\text{parent}(i)] \geq Q[i]$ (part of Max-Heap Property at parent($i$))
    - If not, swap items $Q[i]$ and $Q[\text{parent}(i)]$, and recursively **max_heapify_up**(parent($i$))

- **Correctness**:
  - Max-Heap Property guarantees all nodes $\geq$ descendants, except $Q[i]$ might be $> \text{some of its ancestors}$ (unless $i$ is the root, so we're done)
  - If swap necessary, same guarantee is true with $Q[\text{parent}(i)]$ instead of $Q[i]$

- **Running time**: height of tree, so $\Theta(\log n)$!
Heap Delete Max

- Can only easily remove last element from dynamic array, but max key is in root of tree
- So swap item at root node \( i = 0 \) with last item at node \( n - 1 \) in heap array
- \( \text{max_heapify_down}(i) \): swap root with larger child until Max-Heap Property
  - Check whether \( Q[i] \geq Q[j] \) for \( j \in \{\text{left}(i), \text{right}(i)\} \) (Max-Heap Property at \( i \))
  - If not, swap \( Q[i] \) with \( Q[j] \) for child \( j \in \{\text{left}(i), \text{right}(i)\} \) with maximum key, and recursively \( \text{max_heapify_down}(j) \)
- Correctness:
  - Max-Heap Property guarantees all nodes \( \geq \) descendants, except \( Q[i] \) might be \(<\) some descendants (unless \( i \) is a leaf, so we’re done)
  - If swap is necessary, same guarantee is true with \( Q[j] \) instead of \( Q[i] \)
- Running time: height of tree, so \( \Theta(\log n) \)!

Heap Sort

- Plugging max-heap into priority queue sort gives us a new sorting algorithm
- Running time is \( O(n \log n) \) because each \text{insert} and \text{delete_max} takes \( O(\log n) \)
- But often include two improvements to this sorting algorithm:

In-place Priority Queue Sort

- Max-heap \( Q \) is a prefix of a larger array \( A \), remember how many items \(|Q|\) belong to heap
- \(|Q|\) is initially zero, eventually \(|A|\) (after inserts), then zero again (after deletes)
- \text{insert()} absorbs next item in array at index \(|Q|\) into heap
- \text{delete_max()} moves max item to end, then abandons it by decrementing \(|Q|\)
- In-place priority queue sort with Array is exactly Selection Sort
- In-place priority queue sort with Sorted Array is exactly Insertion Sort
- In-place priority queue sort with binary Max Heap is \textbf{Heap Sort}
Linear Build Heap

- Inserting $n$ items into heap calls `max_heapify_up(i)` for $i$ from 0 to $n - 1$ (root down):

  $$\text{worst-case swaps} \approx \sum_{i=0}^{n-1} \text{depth}(i) = \sum_{i=0}^{n-1} \lg i = \lg(n!) \geq (n/2) \lg(n/2) = \Omega(n \lg n)$$

- **Idea!** Treat full array as a complete binary tree from start, then `max_heapify_down(i)` for $i$ from $n - 1$ to 0 (leaves up):

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

- So can **build heap** in $O(n)$ time
- (Doesn’t speed up $O(n \lg n)$ performance of heap sort)

Sequence AVL Tree Priority Queue

- Where else have we seen linear build time for an otherwise logarithmic data structure? Sequence AVL Tree!
- Store items of priority queue in Sequence AVL Tree in **arbitrary order** (insertion order)
- Maintain max (and/or min) augmentation:
  ```
  \text{node}.\text{max} = \text{pointer to node in subtree of node with maximum key}
  ```
  – This is a subtree property, so constant factor overhead to maintain
- `find_min()` and `find_max()` in $O(1)$ time
- `delete_min()` and `delete_max()` in $O(\log n)$ time
- `build(A)` in $O(n)$ time
- Same bounds as binary heaps (and more)

Set vs. Multiset

- While our Set interface assumes no duplicate keys, we can use these Sets to implement Multisets that allow items with duplicate keys:
  - Each item in the Set is a Sequence (e.g., linked list) storing the Multiset items with the same key, which is the key of the Sequence
- In fact, without this reduction, binary heaps and AVL trees work directly for duplicate-key items (where e.g. `delete_max` deletes *some* item of maximum key), taking care to use $\leq$ constraints (instead of $<$ in Set AVL Trees)