Lecture 16: Priority Queues and Heapsort

- Priority Queue
- Heapsort
Priority Queue ADT

- Two operations:
  - Delete the maximum
  - Insert

- Applications: load balancing and interruption handling in OS, Huffman codes for compression, A* search for AI, Dijkstra’s and Prim's algorithm for graph search, etc.

- How can we implement a priority queue efficiently?
Option 1: Unordered array

- The *lazy* approach where we defer doing work (deleting the maximum) until necessary.
- Insert is $O(1)$ (will be implemented as push in stacks) and assume we have the space in the array.
- Delete maximum is $O(n)$ (have to traverse the entire array to find the maximum element and exchange it with the last element).
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq; // elements
    private int n; // number of elements

    // set initial size of heap to hold size elements
    public UnorderedArrayMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
        n = 0;
    }

    public boolean isEmpty() { return n == 0; }
    public int size() { return n; }
    public void insert(Key x) { pq[n++] = x; }

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < n; i++) {
            if (pq[max].compareTo(pq[i]) < 0) {
                max = i;
            }
        }
        Key temp = pq[max];
        pq[max] = pq[n-1];
        pq[n-1] = temp;

        return pq[--n];
    }
}
Practice Time

Given an empty array of capacity 10, perform the following operations in a priority queue based on an unordered array (lazy approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max
**Priorit Queue**

**Answer**

- insert P
- insert Q
- insert E
- `delete-max` → Q
- insert X
- insert A
- insert M
- `delete-max` → X
- insert P
- insert L
- insert E
- `delete-max` → P
Option 2: Ordered array

- The *eager* approach where we do the work (keeping the array sorted) up front to make later operations efficient.

- Insert is $O(n)$ (we have to find the index to insert and shift elements to perform insertion).

- Delete maximum is $O(1)$ (just take the last element which will be the maximum).
public class OrderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq; // elements
    private int n; // number of elements

    // set initial size of heap to hold size elements
    public OrderedArrayMaxPQ(int capacity) {
        pq = (Key[]) (new Comparable[capacity]);
        n = 0;
    }

    public boolean isEmpty() { return n == 0; }
    public int size() { return n; }
    public Key delMax() { return pq[--n]; }

    public void insert(Key key) {
        int i = n-1;
        while (i >= 0 && key.compareTo(pq[i]) < 0) {
            pq[i+1] = pq[i];
            i--;
        }
        pq[i+1] = key;
        n++;
    }
}
Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an ordered array (eager approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max
Answer
Option 3: Binary heap

- Will allow us to both insert and delete max in $O(\log n)$ running time.
- There is no way to implement a priority queue in such a way that insert and delete max can be achieved in $O(1)$ running time.
- Priority queues are synonyms to binary heaps.
Practice Time

- Given an empty binary heap that represents a priority queue, perform the following operations:

  1. Insert P
  2. Insert Q
  3. Insert E
  4. Delete max
  5. Insert X
  6. Insert A
  7. Insert M
  8. Delete max
  9. Insert P
  10. Insert L
  11. Insert E
  12. Delete max
Answer
Lecture 16: Priority Queues and Heapsort

- Priority Queue
- Heapsort
Basic plan for heap sort

- Use a priority queue to develop a sorting method that works in two steps:
  - 1) **Heap construction**: build a binary heap with all \( n \) keys that need to be sorted.
  - 2) **Sortdown**: repeatedly remove and return the maximum key.
**O(n \log n)** Heap construction

- Insert \( n \) elements, one by one, swim up to their appropriate position.
- We can do better!
- **Key insight:** After \( \text{sink}(a,k,n) \) completes, the subtree rooted at \( k \) is a heap.

```java
private static void sink(Comparable[] pq, int k, int n) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && pq[j-1].compareTo(pq[j]) < 0){
            j++
        }
        if (pq[k-1].compareTo(pq[j-1]) >= 0){
            break;
        }
        Comparable temp = pq[k-1];
        pq[k-1] = pq[j-1];
        pq[j-1] = temp;
        k = j;
    }
}
```
$O(n)$ Heap construction

- Insert all nodes as is in indices 1 to n. We will turn this binary tree into a heap.
- Ignore all leaves (indices $n/2+1,...,n$). Sink each internal node
- \textbf{for} (int $k = n/2; k >= 1; k--$)
  \hspace{1cm} \text{sink}(a, k, n);
Practice Time

- Run the first step of heapsort, heap construction, on the array [2, 9, 7, 6, 5, 8].
Answer: Heap construction

starting point (arbitrary order)

$k = \lfloor n/2 \rfloor = 6/2 = 3$
sink(3,6)

$k = 2$
sink(2,6)

$k = 1$
sink(1,6)
result (heap-ordered)
Sortdown

- Remove the maximum, one at a time, but leave in array instead of nulling out.

- **while** (n > 1) {
  - `exch(a, 1, n--);`
  - `sink(a, 1, n);`
}

- **Key insight**: After each iteration the array consists of a heap-ordered subarray followed by a sub-array in final order.
HEAPSORT

Sortdown

- while(n>1){
  exch(a, 1, n--);
  sink(a, 1, n);
}

Starting point (heap-ordered)

do while(n>1)
  exch(a, 1, n--);
  sink(a, 1, n);

test

Result (sorted)
Heapsort demo

**Sortdown.** Repeatedly delete the largest remaining item.
Practice Time

- Given the heap you constructed before, run the second step of heapsort, sortdown, to sort the array [2,9,7,6,5,8].
Answer: Sortdown
Heapsort analysis

- Heap construction (the fast version) makes $O(n)$ exchanges and $O(n)$ compares.
- Sortdown and therefore the entire heap sort $O(n \log n)$ exchanges and compares.
- In-place sorting algorithm with $O(n \log n)$ worst-case!
- Remember:
  - mergesort: not in place, requires linear extra space.
  - quicksort: quadratic time in worst case.
- Heapsort is optimal both for time and space in terms of Big-O, but:
  - Inner loop longer than quick sort.
  - Poor use of cache because it accesses memory in non-sequential manner, jumping around.
  - Not stable.
# Sorting: Everything you need to remember about it!

<table>
<thead>
<tr>
<th>Which Sort</th>
<th>In place</th>
<th>Stable</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>X</td>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>Insertion</td>
<td>X</td>
<td>X</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Use for small arrays or partially ordered</td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td>X</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; stable</td>
</tr>
<tr>
<td>Quick</td>
<td>X</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$n \log n$ probabilistic guarantee; fastest!</td>
</tr>
<tr>
<td>Heap</td>
<td>X</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; in place</td>
</tr>
</tbody>
</table>
Lecture 16: Priority Queues and Heapsort

- Priority Queue
- Heapsort
Readings:

- Recommended Textbook:
  - Chapter 2.4 (Pages 308-327), 2.5 (336-344)

- Website:
  - Priority Queues: [https://algs4.cs.princeton.edu/24pq/](https://algs4.cs.princeton.edu/24pq/)

- Visualization:
  - Create (compare the n and nlogn approaches) and heapsort: [https://visualgo.net/en/heap](https://visualgo.net/en/heap)

Practice Problems:

- In-class worksheet

- 2.4.1-2.4.11. Also try some creative problems.