LECTURE 5 OF 12

# Priority Queues and Heapsort

**Data Structures & Algorithms**

Based on Robert Sedgewick & Kevin Wayne
*Algorithms, 4th Edition* (Addison-Wesley, 2011)
Chapter 2.4
Duration: 4 hours (with breaks)

---

# Learning Objectives

By the end of this lecture, students will be able to:

1. Define the priority queue abstract data type and articulate why it is a fundamental building block for dozens of important algorithms.
2. Implement a max-oriented priority queue using a binary heap stored in an array, including the swim and sink helper operations.
3. Derive and prove the time complexities of insert (O(log n)) and delete-max (O(log n)) for heap-based priority queues.
4. Implement heapsort as a two-phase algorithm (heap construction followed by sortdown), prove its O(n log n) worst-case bound, and explain why it sorts in place.
5. Describe the indexed priority queue abstraction and explain its role in graph algorithms such as Dijkstra's shortest paths and Prim's minimum spanning tree.
6. Apply priority queues to solve practical problems including multiway merge, median maintenance, top-K selection, and event-driven simulation.

---

# Section 1: The Priority Queue API

*Estimated time: 20 minutes*

## 1.1 Motivation: Why Priority Queues?

Consider the following scenario. You are writing an operating system scheduler. Thousands of processes are waiting to run. Each process has a priority level. At every scheduling quantum, you must answer the question: *which process should run next?* The answer is always the process with the highest priority. But processes arrive and depart continuously, so you cannot simply sort once and be done.

This is the essence of the **priority queue** problem: maintain a dynamic collection of items with associated priorities, supporting efficient insertion and efficient removal of the extremal (maximum or minimum) element.

Priority queues appear everywhere in computer science:

- **Event-driven simulation.** Events are inserted with timestamps; the next event to process is always the one with the smallest timestamp.
- **Job scheduling.** Operating systems, print queues, and network routers all schedule work by priority.
- **Graph algorithms.** Dijkstra's shortest-paths algorithm and Prim's minimum spanning tree algorithm both rely on a priority queue as their central data structure.
- **Data compression.** Huffman coding builds an optimal prefix-free code by repeatedly extracting the two lowest-frequency symbols from a priority queue.
- **Computational geometry.** Sweep-line algorithms process events in geometric order using a priority queue.
- **AI and search.** The A* search algorithm uses a priority queue ordered by estimated total cost.

In all these applications, the collection is too large or too dynamic to sort repeatedly. We need a data structure that supports insertion and extraction of the extremum in better than linear time.

## 1.2 The API

We define a **max-priority queue** as an abstract data type supporting the following operations:

**DEFINITION 1.1 --- MAX-PRIORITY QUEUE API**

| Operation | Description |
|---|---|
| `MaxPQ()` | Create an empty priority queue |
| `MaxPQ(int max)` | Create a priority queue with initial capacity |
| `insert(Key v)` | Insert a key into the priority queue |
| `Key delMax()` | Remove and return the largest key |
| `boolean isEmpty()` | Is the priority queue empty? |
| `int size()` | Number of keys in the priority queue |
| `Key max()` | Return the largest key (without removing it) |

A **min-priority queue** (`MinPQ`) is analogous, replacing `delMax` with `delMin` and `max` with `min`. The choice between max and min is a matter of convention; one can be trivially converted to the other by negating keys or reversing the comparator.

Throughout most of this lecture we work with a max-priority queue, following Sedgewick's convention for presenting heapsort.

## 1.3 Elementary Implementations

Before we develop the heap-based implementation, let us consider two naive approaches to understand the problem.

**Unordered array implementation.** Maintain the keys in an array in no particular order. Insertion appends to the end in O(1) time. To find and remove the maximum, we must scan the entire array --- O(n) time. This is just selection sort's inner loop repeated once.

```
class UnorderedArrayMaxPQ:
    keys[]   // array of keys
    n = 0    // number of keys

    insert(key):
        keys[n++] = key           // O(1)

    delMax():
        max_index = 0
        for i = 1 to n-1:         // O(n) scan
            if keys[i] > keys[max_index]:
                max_index = i
        swap keys[max_index] with keys[n-1]
        return keys[--n]
```

**Ordered array implementation.** Maintain the keys in sorted order. Insertion requires finding the correct position and shifting elements --- O(n) time (just like insertion sort's inner loop). The maximum is always at the end, so `delMax` returns it in O(1) time.

```
class OrderedArrayMaxPQ:
    keys[]   // sorted array of keys
    n = 0    // number of keys

    insert(key):
        i = n - 1
        while i >= 0 and keys[i] > key:   // O(n) shift
            keys[i+1] = keys[i]
            i--
        keys[i+1] = key
        n++

    delMax():
        return keys[--n]                   // O(1)
```

**Linked-list implementations** have similar tradeoffs: an unordered list gives O(1) insert but O(n) removal of the max; an ordered list gives O(n) insert but O(1) removal of the max.

## 1.4 Analysis of Elementary Implementations

| Implementation | insert | delMax | max | Space |
|---|---|---|---|---|
| Unordered array | O(1) | O(n) | O(n) | O(n) |
| Ordered array | O(n) | O(1) | O(1) | O(n) |
| Unordered list | O(1) | O(n) | O(n) | O(n) |
| Ordered list | O(n) | O(1) | O(1) | O(n) |
| **Binary heap** | **O(log n)** | **O(log n)** | **O(1)** | **O(n)** |

The binary heap achieves O(log n) for *both* insert and delMax, which is a dramatic improvement when the queue contains millions of items. For example, with n = 1,000,000, the elementary implementations require up to 1,000,000 operations for one of the two operations, while the heap requires at most about 20.

This is the power of the heap data structure, which we develop in the next section.

---

*[5-minute break]*

---

# Section 2: Binary Heaps

*Estimated time: 60 minutes*

## 2.1 Complete Binary Trees

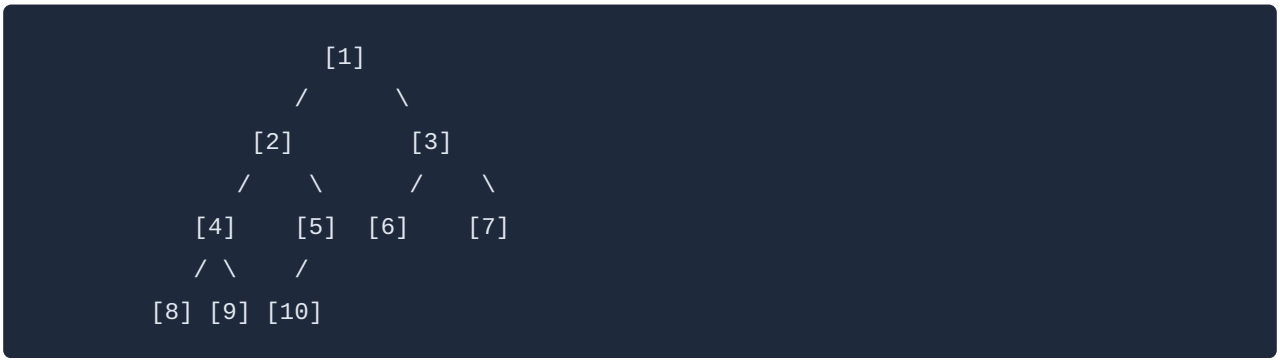
A **binary tree** is a rooted tree in which every node has at most two children, called the *left child* and the *right child*.

**DEFINITION 2.1 --- COMPLETE BINARY TREE**

A binary tree is **complete** if every level except possibly the last is completely filled, and all nodes in the last level are as far left as possible.

A complete binary tree of height h has between 2^h and 2^(h+1) - 1 nodes. Equivalently, a complete binary tree with n nodes has height floor(lg n).

Here is a complete binary tree with 10 nodes:




The numbers in brackets are the *positions* (indices) of the nodes when we number them level by level, left to right, starting from 1. This numbering is the key to the array representation.

## 2.2 The Heap-Ordered Property

**DEFINITION 2.2 --- HEAP ORDER (MAX-HEAP)**

A binary tree is **heap-ordered** (max-heap-ordered) if the key at each node is greater than or equal to the keys at both of its children (if they exist). Equivalently, no node's key exceeds its parent's key.

In a max-heap, the root contains the maximum key. In a min-heap, the definition is reversed: each node's key is less than or equal to its children's keys, and the root contains the minimum.

**DEFINITION 2.3 --- BINARY HEAP**

A **binary heap** is a complete binary tree that satisfies the heap-ordered property.

Example of a max-heap with 10 elements:


Verify: T >= S, T >= R, S >= P, S >= N, R >= O, R >= A, P >= E, P >= I, N >= G. Every parent is at least as large as its children.

## 2.3 Array Representation of a Binary Heap

The most elegant aspect of the binary heap is that it can be stored in a simple array *with no explicit pointers*. We use the position numbering from the complete binary tree:

**PROPOSITION 2.1.** In a complete binary tree stored in an array `a[1..n]` (1-indexed, with `a[0]` unused):

- The parent of node at position `k` is at position `k/2` (integer division).
- The children of node at position `k` are at positions `2k` and `2k+1`.
- The root is at position 1.
- The nodes at positions `n/2 + 1` through `n` are leaves.

*Proof.* This follows directly from the level-by-level numbering. At level L (starting from 0), there are 2^L nodes, numbered from 2^L to 2^(L+1) - 1. If a node is at position k, its parent at the level above is at position floor(k/2), and its children at the level below are at positions 2k and 2k+1.

For the heap shown above, the array representation is:

```
Index:  0   1   2   3   4   5   6   7   8   9  10
Value:  -   T   S   R   P   N   O   A   E   I   G
```

Position 0 is unused (this makes the arithmetic clean). The parent of G (position 10) is at position 10/2 = 5, which is N. The children of S (position 2) are at positions 4 (P) and 5 (N). Everything checks out.

**Why is this representation good?**

1. **No pointers.** We save the memory overhead of left/right child pointers and parent pointers.
2. **Cache-friendly traversal.** Array elements are stored contiguously in memory, so moving between parent and child involves simple arithmetic on indices. (Though, as we will discuss later, the access pattern of heapsort is still less cache-friendly than quicksort.)
3. **Compact.** The complete binary tree property guarantees there are no "gaps" in the array.

## 2.4 The Swim Operation (Bottom-Up Reheapify)

When we insert a new element at the end of the heap (the next available position), it may violate the heap order with respect to its parent. The **swim** operation restores heap order by repeatedly exchanging the node with its parent until the node reaches a position where it is no longer larger than its parent (or it reaches the root).

```
swim(k):
    while k > 1 and a[k/2] < a[k]:
        swap a[k] with a[k/2]
        k = k / 2
```

**Detailed trace.** Suppose we have the following heap and we insert the key "X":

```
Before insert:
Index:  1   2   3   4   5   6   7   8   9  10
Value:  T   S   R   P   N   O   A   E   I   G

Insert X at position 11:
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  T   S   R   P   N   O   A   E   I   G   X
```

Now swim(11):

• k=11, parent at 11/2=5, a[5]=N, a[11]=X. X > N, so swap:

```
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  T   S   R   P   X   O   A   E   I   G   N
```

• k=5, parent at 5/2=2, a[2]=S, a[5]=X. X > S, so swap:

```
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  T   X   R   P   S   O   A   E   I   G   N
```

• k=2, parent at 2/2=1, a[1]=T, a[2]=X. X > T? Let us say alphabetically yes (X > T). So swap:

```
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  X   T   R   P   S   O   A   E   I   G   N
```

• k=1, loop terminates (k is not > 1).

The tree now looks like:


Heap order is restored. The swim operation traced a path from a leaf to the root, doing at most floor(lg n) swaps.

## 2.5 The Sink Operation (Top-Down Reheapify)

When the root of the heap is removed and replaced with the last element, that element is likely too small for the root position. The **sink** operation restores heap order by repeatedly exchanging the node with the *larger* of its two children, until the node is larger than both children or it reaches a leaf.

```
sink(k, n):
    while 2*k <= n:
        j = 2*k                          // left child
        if j < n and a[j] < a[j+1]:      // right child is larger
            j++
        if a[k] >= a[j]:                  // heap order satisfied
            break
        swap a[k] with a[j]
        k = j
```

**Why exchange with the larger child?** If we exchanged with the smaller child, that child would become the parent of the larger child, violating heap order.

**Detailed trace.** Starting from the heap after our insertion of X:

```
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  X   T   R   P   S   O   A   E   I   G   N
```

Perform delMax: save X, move N (position 11) to position 1, decrement size to 10.

```
Index:  1   2   3   4   5   6   7   8   9  10
Value:  N   T   R   P   S   O   A   E   I   G
```

Now sink(1, 10):

- k=1, children at 2 (T) and 3 (R). Larger child is T at j=2. N < T, so swap:

```
Index:  1   2   3   4   5   6   7   8   9  10
Value:  T   N   R   P   S   O   A   E   I   G
```

- k=2, children at 4 (P) and 5 (S). Larger child is S at j=5. N < S, so swap:

```
Index:  1   2   3   4   5   6   7   8   9  10
Value:  T   S   R   P   N   O   A   E   I   G
```

- k=5, children at 10 (G). Only one child: j=10. N >= G, so stop.

Heap order is restored. The sink operation traced a path from root to an internal node near the leaves, doing at most floor(lg n) swaps.

## 2.6 Insert and Delete-Max

With swim and sink, the priority queue operations are straightforward.

**Insert:**

```
insert(key):
    a[++n] = key        // add key at end
    swim(n)             // restore heap order
```

**Delete maximum:**

```
delMax():
    max = a[1]          // root is the maximum
    swap a[1] with a[n]
    n--                  // remove last element
    sink(1, n)           // restore heap order
    a[n+1] = null        // prevent loitering
    return max
```

**PROPOSITION 2.2.** In an n-item binary heap, insert requires at most 1 + lg n compares, and delete-max requires at most 2 lg n compares.

*Proof.* Insert does one compare per level as it swims up from the bottom. The height of the tree is floor(lg n), so at most 1 + floor(lg n) compares. Delete-max does two compares per level as it sinks from the top (one to find the larger child, one to compare with the node), for at most 2 floor(lg n) compares.

## 2.7 Complete Trace: Sequence of Operations

Let us trace through a sequence of operations on an initially empty max-priority queue, inserting the letters H, E, A, P, S, O, R, T and then performing two delMax operations.

**Insertions:**

```
insert(H):  [H]                          size=1
            Heap: (H)

insert(E):  [H, E]                       size=2
            swim(2): E < H, stop.
            Heap:   (H)
                   /
                 (E)

insert(A):  [H, E, A]                    size=3
            swim(3): A < H, stop.
            Heap:   (H)
                   /   \
                 (E)   (A)

insert(P):  [H, E, A, P]                 size=4
            swim(4): P > E, swap positions 4 and 2.
                     [H, P, A, E]
                     P < H, stop.
            Heap:    (H)
                    /   \
                 (P)     (A)
                /
              (E)

insert(S):  [H, P, A, E, S]              size=5
            swim(5): S > P, swap positions 5 and 2.
                     [H, S, A, E, P]
                     S > H, swap positions 2 and 1.
                     [S, H, A, E, P]
            Heap:    (S)
                    /   \
                 (H)     (A)
                /   \
              (E)   (P)

insert(O):  [S, H, A, E, P, O]           size=6
            swim(6): O > A, swap positions 6 and 3.
                     [S, H, O, E, P, A]
```

```
                    O < S, stop.
            Heap:     (S)
                     /    \
                  (H)      (O)
                 /   \    /
               (E)  (P) (A)

insert(R):  [S, H, O, E, P, A, R]          size=7
            swim(7): R > O, swap positions 7 and 3.
                    [S, H, R, E, P, A, O]
                    R < S, stop.
            Heap:     (S)
                     /    \
                  (H)      (R)
                 /   \    /   \
               (E)  (P) (A)  (O)

insert(T):  [S, H, R, E, P, A, O, T]       size=8
            swim(8): T > E, swap positions 8 and 4.
                    [S, H, R, T, P, A, O, E]
                    T > H, swap positions 4 and 2.
                    [S, T, R, H, P, A, O, E]
                    T > S, swap positions 2 and 1.
                    [T, S, R, H, P, A, O, E]
            Heap:       (T)
                       /    \
                    (S)      (R)
                   /   \    /   \
                 (H)  (P) (A)  (O)
                 /
               (E)
```

**Deletions:**

```
delMax():   Return T.
            Swap a[1] with a[8]: [E, S, R, H, P, A, O, T]
            Decrement n to 7:    [E, S, R, H, P, A, O]
            sink(1, 7):
              k=1, children S(2), R(3). Larger: S at 2. E < S, swap.
              [S, E, R, H, P, A, O]
              k=2, children H(4), P(5). Larger: P at 5. E < P, swap.
              [S, P, R, H, E, A, O]
              k=5, no children (2*5=10 > 7). Stop.
            Result: [S, P, R, H, E, A, O]
            Heap:       (S)
                       /   \
                    (P)     (R)
                   /  \    /  \
                 (H) (E) (A) (O)

delMax():   Return S.
            Swap a[1] with a[7]: [O, P, R, H, E, A, S]
            Decrement n to 6:    [O, P, R, H, E, A]
            sink(1, 6):
              k=1, children P(2), R(3). Larger: R at 3. O < R, swap.
              [R, P, O, H, E, A]
              k=3, children A(6). Only one child. O >= A, stop.
            Result: [R, P, O, H, E, A]
            Heap:       (R)
                       /   \
                    (P)     (O)
                   /  \    /
                 (H) (E) (A)
```

## 2.8 Heap Properties: A Summary

| Property | Value |
|---|---|
| Structure | Complete binary tree stored in array `a[1..n]` |
| Height | floor(lg n) |
| Parent of node k | k/2 |
| Children of node k | 2k, 2k+1 |
| Insert time | O(log n) --- at most 1 + lg n compares |
| Delete-max time | O(log n) --- at most 2 lg n compares |
| Find-max time | O(1) --- return a[1] |
| Space | O(n) --- array of size n+1 |

---

*[15-minute break]*

---

# Section 3: Heapsort

*Estimated time: 45 minutes*

## 3.1 Overview

Heapsort is an elegant sorting algorithm that exploits the binary heap to sort an array in place in O(n log n) worst-case time. It was invented by J.W.J. Williams in 1964 and improved by Robert Floyd shortly thereafter.

The algorithm has two phases:

1. **Heap construction.** Rearrange the original array so it satisfies the heap-order property.
2. **Sortdown.** Repeatedly extract the maximum and place it at the end of the array.

## 3.2 Heap Construction: Bottom-Up Method

The naive approach to building a heap is to insert elements one by one, each taking O(log n) time, for a total of O(n log n). Floyd's bottom-up method is more efficient.

**Key insight:** In an array of n elements, the entries from position n/2 + 1 through n are already trivially heap-ordered because they are leaves (they have no children). We only need to sink the internal nodes, starting from position n/2 and working backward to position 1.

```
heapConstruct(a[], n):
    for k = n/2 downto 1:
        sink(a, k, n)
```

**Why does this work?** When we sink node k, both subtrees rooted at 2k and 2k+1 are already valid heaps (because we process nodes from bottom to top). So sinking node k makes the subtree rooted at k into a valid heap.

**THEOREM 3.1.** Bottom-up heap construction uses at most 2n compares and at most n exchanges.

*Proof.* The cost of sinking a node at height h is at most 2h compares (two per level: one to find the larger child, one to compare with the node). The number of nodes at height h in a complete binary tree with n nodes is at most ceil(n / 2^(h+1)).

Total compares <= sum over h=0 to floor(lg n) of (n / 2^(h+1)) * 2h
= n * sum over h=0 to floor(lg n) of h / 2^h
<= n * sum over h=0 to infinity of h / 2^h
= n * 2
= 2n

The infinite series sum h/2^h = 2 is a well-known result (obtainable by differentiating the geometric series sum x^h = 1/(1-x) and evaluating at x = 1/2).

Similarly, the number of exchanges is at most n. This is a remarkable result: **we can build a heap in linear time**, even though inserting n elements one by one would cost O(n log n).

## 3.3 The Sortdown Phase

After heap construction, a[1] is the maximum. We swap it with a[n], decrement the heap size, and sink a[1] to restore heap order among the first n-1 elements. We repeat until the heap is empty.

```
sortdown(a[], n):
    while n > 1:
        swap a[1] with a[n]
        n--
        sink(a, 1, n)
```

Each iteration places the next-largest element in its correct final position.

## 3.4 The Complete Heapsort Algorithm

```
heapsort(a[], n):
    // Phase 1: Heap construction (bottom-up)
    for k = n/2 downto 1:
        sink(a, k, n)

    // Phase 2: Sortdown
    while n > 1:
        swap a[1] with a[n]
        n--
        sink(a, 1, n)
```

## 3.5 Complete Trace

Let us sort the array: S O R T E X A M P L E

Using 1-indexed positions:

```
Initial array:
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  S   O   R   T   E   X   A   M   P   L   E
```

**Phase 1: Heap Construction**

n = 11, so n/2 = 5. We sink positions 5, 4, 3, 2, 1.

```
sink(5, 11): a[5]=E, children a[10]=L, a[11]=E. Larger child: L at 10.
             E < L, swap positions 5 and 10.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  S   O   R   T   L   X   A   M   P   E   E
             k=10, 2*10=20 > 11, stop.

sink(4, 11): a[4]=T, children a[8]=M, a[9]=P. Larger child: P at 9.
             T >= P, stop. No change.

sink(3, 11): a[3]=R, children a[6]=X, a[7]=A. Larger child: X at 6.
             R < X, swap positions 3 and 6.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  S   O   X   T   L   R   A   M   P   E   E
             k=6, 2*6=12 > 11, stop.

sink(2, 11): a[2]=O, children a[4]=T, a[5]=L. Larger child: T at 4.
             O < T, swap positions 2 and 4.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  S   T   X   O   L   R   A   M   P   E   E
             k=4, children a[8]=M, a[9]=P. Larger: P at 9.
             O < P, swap positions 4 and 9.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  S   T   X   P   L   R   A   M   O   E   E
             k=9, 2*9=18 > 11, stop.

sink(1, 11): a[1]=S, children a[2]=T, a[3]=X. Larger child: X at 3.
             S < X, swap positions 1 and 3.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  X   T   S   P   L   R   A   M   O   E   E
             k=3, children a[6]=R, a[7]=A. Larger: R at 6.
             S < R, swap positions 3 and 6.
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  X   T   R   P   L   S   A   M   O   E   E
             k=6, 2*6=12 > 11, stop.
```

Heap after construction:

```
                  (X)
                /     \
              (T)       (R)
             /   \     /   \
           (P)   (L) (S)   (A)
          / \    /
        (M) (O) (E)  (E is at index 11, but this
                      is a leaf at position 10's sibling)
```

Wait --- let me re-draw with the correct positions:

```
                    (X)
                  /     \
                (T)       (R)
               /   \     /   \
             (P)   (L) (S)   (A)
            / \    /
          (M) (O) (E)
                   \
                   (E)
```

More precisely (level order):

```
Level 0:                  X
Level 1:           T             R
Level 2:       P       L       S       A
Level 3:    M    O   E   E
```

**Phase 2: Sortdown**

```
Step 1: swap a[1]=X with a[11]=E. Heap size = 10.
Index:  1   2   3   4   5   6   7   8   9  10 | 11
Value:  E   T   R   P   L   S   A   M   O   E | X
        sink(1, 10): E sinks... T rises to root.
        After: [T, P, R, O, L, S, A, M, E, E | X]
        (Detailed: E<T swap(1,2), E<P swap(2,4), E<O swap(4,9), stop)

        Wait, let me redo this carefully:
        sink(1,10): a[1]=E, children T(2), R(3). Larger: T at 2. E<T, swap.
Index:  1   2   3   4   5   6   7   8   9  10 | 11
Value:  T   E   R   P   L   S   A   M   O   E | X
        k=2, children P(4), L(5). Larger: P at 4. E<P, swap.
Value:  T   P   R   E   L   S   A   M   O   E | X
        k=4, children M(8), O(9). Larger: O at 9. E<O, swap.
Value:  T   P   R   O   L   S   A   M   E   E | X
        k=9, 2*9=18 > 10, stop.

Step 2: swap a[1]=T with a[10]=E. Heap size = 9.
Index:  1   2   3   4   5   6   7   8   9 | 10  11
Value:  E   P   R   O   L   S   A   M   E | T   X
        sink(1,9): E<R swap(1,3). E<S swap(3,6). k=6, 2*6=12>9, stop.
Value:  R   P   S   O   L   E   A   M   E | T   X

Step 3: swap a[1]=R with a[9]=E. Heap size = 8.
Value:  E   P   S   O   L   E   A   M | R   T   X
        sink(1,8): E<S swap(1,3). E>=E and E>=A, stop.
Value:  S   P   E   O   L   E   A   M | R   T   X
        (Check: k=3, children E(6),A(7). Larger: E at 6. E>=E, stop.)

Step 4: swap a[1]=S with a[8]=M. Heap size = 7.
Value:  M   P   E   O   L   E   A | S   R   T   X
        sink(1,7): M<P swap(1,2). M<O swap(2,4). k=4, 2*4=8>7, stop.
Value:  P   O   E   M   L   E   A | S   R   T   X

Step 5: swap a[1]=P with a[7]=A. Heap size = 6.
Value:  A   O   E   M   L   E | P   S   R   T   X
        sink(1,6): A<O swap(1,2). A<M swap(2,4). k=4, 2*4=8>6, stop.
Value:  O   M   E   A   L   E | P   S   R   T   X
```

```
Step 6: swap a[1]=O with a[6]=E. Heap size = 5.
Value:  E   M   E   A   L | O   P   S   R   T   X
        sink(1,5): E<M swap(1,2). E>=A and... k=2, children A(4),L(5).
        Larger: L at 5. E<L, swap.
Value:  M   L   E   A   E | O   P   S   R   T   X

Step 7: swap a[1]=M with a[5]=E. Heap size = 4.
Value:  E   L   E   A | M   O   P   S   R   T   X
        sink(1,4): E<L swap(1,2). k=2, children A(4). E>=A, stop.
Value:  L   E   E   A | M   O   P   S   R   T   X

Step 8: swap a[1]=L with a[4]=A. Heap size = 3.
Value:  A   E   E | L   M   O   P   S   R   T   X
        sink(1,3): A<E swap(1,2). k=2, 2*2=4>3, stop.
Value:  E   A   E | L   M   O   P   S   R   T   X
        (Children of 1: E(2), E(3). Both equal. Pick left, j=2. A<E? No, a[1]=A but
         we already swapped. Let me redo.)
        a[1]=A, children a[2]=E, a[3]=E. Larger: E at 2 (tie, pick left). A<E, swap.
Value:  E   A   E | L   M   O   P   S   R   T   X

Step 9: swap a[1]=E with a[3]=E. Heap size = 2.
Value:  E   A | E   L   M   O   P   S   R   T   X
        sink(1,2): a[1]=E, child a[2]=A. E>=A, stop.

Step 10: swap a[1]=E with a[2]=A. Heap size = 1. Done.
Value:  A | E   E   L   M   O   P   S   R   T   X
```

**Final sorted array:**

```
Index:  1   2   3   4   5   6   7   8   9  10  11
Value:  A   E   E   L   M   O   P   S   R   T   X
```

The array is sorted in ascending order!

## 3.6 Analysis of Heapsort

**THEOREM 3.2.** Heapsort uses at most 2n lg n + 2n compares (and at most n lg n + n exchanges) to sort n items.

*Proof.*

- Heap construction: at most 2n compares (Theorem 3.1).
- Sortdown: each of the n-1 iterations calls sink on a heap of size at most n, using at most 2 lg n compares. Total: at most 2(n-1) lg n compares.
- Combined: 2n + 2(n-1) lg n <= 2n lg n + 2n.

**PROPOSITION 3.1.** Heapsort sorts in place, using only O(1) extra space beyond the input array.

The only extra storage needed is a constant number of local variables. This is a significant advantage over mergesort, which requires O(n) auxiliary space.

## 3.7 Comparison with Other O(n log n) Sorts

| Property | Heapsort | Mergesort | Quicksort |
|---|---|---|---|
| Worst-case time | O(n log n) | O(n log n) | O(n^2) |
| Average-case time | O(n log n) | O(n log n) | O(n log n) |
| In-place? | Yes | No (O(n) aux) | Yes |
| Stable? | No | Yes | No (typically) |
| Cache-friendly? | Poor | Moderate | Excellent |
| Practical speed | Slower | Moderate | Fastest |

**Why is heapsort slower in practice despite having the same asymptotic bound as quicksort?**

1. **Poor cache locality.** Heapsort accesses array elements that are far apart. When we sink an element, we jump from position k to 2k to 4k, and so on. These positions are far apart in memory, causing many cache misses. Quicksort, by contrast, accesses elements sequentially (or nearly so), which is highly cache-friendly.

2. **Inner loop overhead.** Heapsort's inner loop (in sink) involves a comparison to find the larger child and then a comparison with the current node. Quicksort's inner loop is a single comparison and an index increment.

3. **The constant factor in 2n lg n.** The leading constant for heapsort is about 2, compared to about 1.39 for quicksort (on average). That factor of roughly 1.44x matters in practice.

Despite being slower in practice, heapsort has a unique combination of properties: worst-case O(n log n) time and in-place sorting. No other comparison sort achieves both. This makes heapsort the algorithm of choice in situations where worst-case guarantees matter and memory is constrained.

**Historical note.** Heapsort was the first algorithm to achieve O(n log n) worst-case in-place sorting. It remains the only widely-known comparison sort with this property. (Introsort, used in many standard libraries, starts with quicksort and switches to heapsort if the recursion depth gets too large, giving O(n log n) worst-case.)

---

*[10-minute break]*

---

# Section 4: Indexed Priority Queues

*Estimated time: 30 minutes*

## 4.1 Motivation

In many graph algorithms, we associate priorities with *vertices* (or other items identified by integer indices), and we need to *change the priority* of an item that is already in the queue. A standard heap does not support this efficiently because there is no way to find a specific item without scanning the entire array.

**Example: Dijkstra's algorithm.** We maintain tentative shortest-path distances for all vertices. When we relax an edge (u, v) and find a shorter path to v, we need to *decrease* v's priority (its tentative distance). This requires locating v in the priority queue and adjusting its position.

The **indexed priority queue** solves this by associating each item with an integer index from 0 to maxN-1.

## 4.2 The Indexed Min-Priority Queue API

**DEFINITION 4.1 --- INDEXED MIN-PRIORITY QUEUE**

An indexed min-priority queue maintains a collection of key-value pairs where each item is identified by a unique integer index in {0, 1, ..., maxN-1}.

| Operation | Description |
|---|---|
| `IndexMinPQ(int maxN)` | Create an indexed PQ with indices 0 to maxN-1 |
| `insert(int i, Key key)` | Associate key with index i |
| `changeKey(int i, Key key)` | Change the key associated with index i |
| `decreaseKey(int i, Key key)` | Decrease the key associated with index i |
| `contains(int i)` | Is index i in the priority queue? |
| `delete(int i)` | Remove index i and its associated key |
| `int delMin()` | Remove and return the index of the minimum key |
| `Key minKey()` | Return the minimum key |
| `int minIndex()` | Return the index associated with the minimum key |
| `boolean isEmpty()` | Is the priority queue empty? |
| `int size()` | Number of items in the priority queue |

Note that we use a min-PQ here because the primary application (Dijkstra's algorithm) extracts the minimum.

## 4.3 Implementation with Three Parallel Arrays

The implementation uses three arrays:

- `keys[i]` stores the key (priority) associated with index i.
- `pq[k]` stores the index of the item at heap position k. That is, `pq[k] = i` means the item with index i is at position k in the heap.
- `qp[i]` stores the heap position of the item with index i. That is, `qp[i] = k` means item i is at heap position k. If item i is not in the queue, `qp[i] = -1`.

The invariant is: `pq[qp[i]] = i` and `qp[pq[k]] = k`.

```
class IndexMinPQ:
    maxN          // maximum number of elements
    n = 0         // current number of elements
    keys[maxN]    // keys[i] = priority of item i
    pq[maxN+1]    // pq[k] = index of item at heap position k
    qp[maxN]      // qp[i] = heap position of item i (-1 if absent)

    insert(i, key):
        n++
        qp[i] = n
        pq[n] = i
        keys[i] = key
        swim(n)

    changeKey(i, key):
        keys[i] = key
        swim(qp[i])
        sink(qp[i])

    decreaseKey(i, key):
        keys[i] = key
        swim(qp[i])           // key got smaller, might swim up (min-heap)

    delMin():
        min_index = pq[1]
        swap(1, n)
        n--
        sink(1)
        qp[min_index] = -1
        keys[min_index] = null
        pq[n+1] = -1
        return min_index

    contains(i):
        return qp[i] != -1

    // Helper: compare keys at heap positions
    greater(k1, k2):
        return keys[pq[k1]] > keys[pq[k2]]
```

```
// Helper: swap items at heap positions
swap(k1, k2):
    temp = pq[k1]
    pq[k1] = pq[k2]
    pq[k2] = temp
    qp[pq[k1]] = k1
    qp[pq[k2]] = k2

swim(k):
    while k > 1 and greater(k/2, k):
        swap(k, k/2)
        k = k/2

sink(k):
    while 2*k <= n:
        j = 2*k
        if j < n and greater(j, j+1):
            j++
        if not greater(k, j):
            break
        swap(k, j)
        k = j
```

## 4.4 Analysis

**PROPOSITION 4.1.** In an indexed priority queue with n items:

| Operation | Time |
|---|---|
| `insert` | O(log n) |
| `delMin` | O(log n) |
| `changeKey` | O(log n) |
| `decreaseKey` | O(log n) |
| `delete` | O(log n) |
| `contains` | O(1) |
| `minKey` | O(1) |
| `minIndex` | O(1) |

All logarithmic operations follow from the fact that swim and sink traverse at most the height of the heap, which is floor(lg n). The constant-time operations follow from direct array access.

**Space:** O(maxN) for the three parallel arrays, regardless of the current number of items n.

## 4.5 Applications in Graph Algorithms

**Dijkstra's shortest paths.** The algorithm maintains a set of vertices whose shortest-path distances are known and a priority queue of vertices whose distances are tentative. At each step, it extracts the vertex with minimum tentative distance and relaxes its outgoing edges, potentially decreasing the priorities of neighboring vertices.

```
dijkstra(G, source):
    dist[source] = 0
    pq.insert(source, 0.0)
    while not pq.isEmpty():
        v = pq.delMin()
        for each edge (v, w) with weight c:
            if dist[v] + c < dist[w]:
                dist[w] = dist[v] + c
                if pq.contains(w):
                    pq.decreaseKey(w, dist[w])
                else:
                    pq.insert(w, dist[w])
```

With an indexed min-PQ backed by a binary heap, Dijkstra's algorithm runs in O(E log V) time, where E is the number of edges and V is the number of vertices.

**Prim's minimum spanning tree.** Similarly to Dijkstra, Prim's algorithm grows an MST by repeatedly adding the minimum-weight edge crossing the cut between the tree and the remaining vertices. The indexed priority queue maintains, for each non-tree vertex, the weight of the lightest edge connecting it to the tree.

---

# Section 5: Applications of Priority Queues

*Estimated time: 20 minutes*

## 5.1 Event-Driven Simulation

In an event-driven simulation, the system state changes only when *events* occur. Events are stored in a priority queue ordered by their scheduled time. The simulation loop:

```
pq = MinPQ()
pq.insert(initial events)
while not pq.isEmpty():
    event = pq.delMin()              // next event in time order
    if event is still valid:
        update system state
        insert any new events into pq
```

This approach is far more efficient than a time-stepped simulation when events are sparse. Sedgewick illustrates this with a simulation of colliding particles, where the priority queue manages the times of predicted collisions.

## 5.2 Huffman Coding

Huffman coding constructs an optimal prefix-free binary code for data compression. The algorithm repeatedly extracts the two symbols with the lowest frequencies, combines them into a new node, and reinserts the combined node. This requires a min-priority queue.

```
huffman(frequencies[]):
    pq = MinPQ()
    for each symbol s with frequency f:
        pq.insert(new leaf node(s, f))
    while pq.size() > 1:
        left = pq.delMin()
        right = pq.delMin()
        parent = new internal node(left.freq + right.freq, left, right)
        pq.insert(parent)
    return pq.delMin()       // root of Huffman tree
```

The resulting tree assigns shorter codes to more frequent symbols, minimizing the expected code length. With a binary heap, the algorithm runs in O(n log n) time for n symbols.

## 5.3 Multiway Merge

Given K sorted input streams, merge them into a single sorted output stream. The priority queue holds the current element from each stream. At each step, extract the minimum and advance that stream.

```
multiwayMerge(streams[0..K-1]):
    pq = MinPQ()
    for i = 0 to K-1:
        if streams[i] is not empty:
            pq.insert((streams[i].next(), i))
    while not pq.isEmpty():
        (value, i) = pq.delMin()
        output value
        if streams[i] is not empty:
            pq.insert((streams[i].next(), i))
```

This runs in O(N log K) time, where N is the total number of elements across all streams. This is much better than O(N log N) when K is small.

**Application.** External sorting: sort a file that does not fit in memory by sorting chunks that do fit, writing them to disk, and then using multiway merge to combine them.

## 5.4 Median Maintenance (The Two-Heap Trick)

**Problem.** Given a stream of integers, maintain the running median efficiently.

**Solution.** Use two heaps:
- A **max-heap** for the lower half of the data.
- A **min-heap** for the upper half of the data.

Invariant: the max-heap has the same size as the min-heap, or one more element. The median is always the maximum of the max-heap (plus the minimum of the min-heap if both have equal size).

```
insert(x):
    if maxHeap is empty or x <= maxHeap.max():
        maxHeap.insert(x)
    else:
        minHeap.insert(x)

    // Rebalance: maxHeap should have same size or one more
    if maxHeap.size() > minHeap.size() + 1:
        minHeap.insert(maxHeap.delMax())
    else if minHeap.size() > maxHeap.size():
        maxHeap.insert(minHeap.delMin())

median():
    if maxHeap.size() > minHeap.size():
        return maxHeap.max()
    else:
        return (maxHeap.max() + minHeap.min()) / 2.0
```

Each insertion takes O(log n) time. Finding the median takes O(1) time.

## 5.5 Top-K Problems

**Problem.** Given a stream of N items, find the K largest.

**Solution.** Maintain a min-priority queue of size K. For each new item: if the queue has fewer than K items, insert it. Otherwise, if the new item exceeds the minimum in the queue, replace the minimum with the new item.

```
topK(stream, K):
    pq = MinPQ()
    for each item in stream:
        pq.insert(item)
        if pq.size() > K:
            pq.delMin()       // discard the smallest
    return pq                 // contains the K largest items
```

This runs in O(N log K) time and uses only O(K) space. When K is much smaller than N (e.g., finding the top 10 out of a billion), this is dramatically better than sorting the entire stream.

---

*[10-minute break]*

---

# Section 6: Worked Examples and Practice Problems

*Estimated time: 30 minutes*

## Problem 1: Trace Heap Insertion

**Problem.** Starting with an empty max-heap, insert the keys 5, 3, 8, 1, 9, 2, 7 one at a time. Show the array after each insertion.

**Solution.**

```
insert(5):  [5]
            No swim needed (root).

insert(3):  [5, 3]
            swim(2): 3 < 5, stop.

insert(8):  [5, 3, 8]
            swim(3): 8 > 5, swap positions 1 and 3.
            [8, 3, 5]

insert(1):  [8, 3, 5, 1]
            swim(4): 1 < 3, stop.

insert(9):  [8, 3, 5, 1, 9]
            swim(5): 9 > 3, swap positions 5 and 2.
            [8, 9, 5, 1, 3]
            swim(2): 9 > 8, swap positions 2 and 1.
            [9, 8, 5, 1, 3]

insert(2):  [9, 8, 5, 1, 3, 2]
            swim(6): 2 < 5, stop.

insert(7):  [9, 8, 5, 1, 3, 2, 7]
            swim(7): 7 > 5, swap positions 7 and 3.
            [9, 8, 7, 1, 3, 2, 5]
            swim(3): 7 < 9, stop.
```

**Final heap:**

```
            (9)
          /     \
       (8)       (7)
      /    \    /    \
    (1)   (3) (2)   (5)

Array: [9, 8, 7, 1, 3, 2, 5]
```

---

## Problem 2: Trace Delete-Max

**Problem.** Starting from the heap [9, 8, 7, 1, 3, 2, 5], perform two delMax operations. Show the array state after each.

**Solution.**

```
delMax #1:
    Return 9.
    Swap a[1]=9 with a[7]=5. Array: [5, 8, 7, 1, 3, 2] (n=6)
    sink(1, 6):
      k=1, children 8(2), 7(3). Larger: 8 at 2. 5 < 8, swap.
      [8, 5, 7, 1, 3, 2]
      k=2, children 1(4), 3(5). Larger: 3 at 5. 5 >= 3, stop.
    Result: [8, 5, 7, 1, 3, 2]

            (8)
          /     \
       (5)       (7)
      /    \    /
    (1)   (3) (2)

delMax #2:
    Return 8.
    Swap a[1]=8 with a[6]=2. Array: [2, 5, 7, 1, 3] (n=5)
    sink(1, 5):
      k=1, children 5(2), 7(3). Larger: 7 at 3. 2 < 7, swap.
      [7, 5, 2, 1, 3]
      k=3, no children (2*3=6 > 5). Stop.
    Result: [7, 5, 2, 1, 3]

            (7)
          /     \
       (5)       (2)
      /    \
    (1)   (3)
```

## Problem 3: Bottom-Up Heap Construction

**Problem.** Build a max-heap from the array [4, 7, 2, 9, 1, 6, 3, 8, 5] using bottom-up construction. Show the array after each sink operation.

**Solution.**

```
Initial:
Index:  1   2   3   4   5   6   7   8   9
Value:  4   7   2   9   1   6   3   8   5

n = 9, n/2 = 4. Sink positions 4, 3, 2, 1.

sink(4, 9): a[4]=9, children a[8]=8, a[9]=5. Larger: 8 at 8.
            9 >= 8, stop. No change.
Array:  4   7   2   9   1   6   3   8   5

sink(3, 9): a[3]=2, children a[6]=6, a[7]=3. Larger: 6 at 6.
            2 < 6, swap positions 3 and 6.
Array:  4   7   6   9   1   2   3   8   5
            k=6, 2*6=12 > 9, stop.

sink(2, 9): a[2]=7, children a[4]=9, a[5]=1. Larger: 9 at 4.
            7 < 9, swap positions 2 and 4.
Array:  4   9   6   7   1   2   3   8   5
            k=4, children a[8]=8, a[9]=5. Larger: 8 at 8.
            7 < 8, swap positions 4 and 8.
Array:  4   9   6   8   1   2   3   7   5
            k=8, 2*8=16 > 9, stop.

sink(1, 9): a[1]=4, children a[2]=9, a[3]=6. Larger: 9 at 2.
            4 < 9, swap positions 1 and 2.
Array:  9   4   6   8   1   2   3   7   5
            k=2, children a[4]=8, a[5]=1. Larger: 8 at 4.
            4 < 8, swap positions 2 and 4.
Array:  9   8   6   4   1   2   3   7   5
            k=4, children a[8]=7, a[9]=5. Larger: 7 at 8.
            4 < 7, swap positions 4 and 8.
Array:  9   8   6   7   1   2   3   4   5
            k=8, 2*8=16 > 9, stop.

Final heap:
Array:  9   8   6   7   1   2   3   4   5

              (9)
            /     \
```

```
          (8)       (6)
         /   \     /   \
       (7)   (1) (2)   (3)
      /   \
    (4)   (5)
```

---

## Problem 4: Heap Property Verification

**Problem.** Which of the following arrays represent valid max-heaps? For each, verify or identify the violation.

(a) [10, 9, 8, 7, 6, 5, 4]
(b) [10, 3, 8, 7, 6, 5, 4]
(c) [5, 10, 8, 7, 6, 3, 4]

**Solution.**

(a) [10, 9, 8, 7, 6, 5, 4]

```
Check parent >= children:
  a[1]=10 >= a[2]=9 and a[3]=8   YES
  a[2]=9  >= a[4]=7 and a[5]=6   YES
  a[3]=8  >= a[6]=5 and a[7]=4   YES
```

**Valid max-heap.** (In fact, this is a sorted array in descending order, which is always a valid max-heap.)

(b) [10, 3, 8, 7, 6, 5, 4]

```
Check:
  a[1]=10 >= a[2]=3 and a[3]=8   YES
  a[2]=3  >= a[4]=7?             NO! 3 < 7.
```

**Not a valid max-heap.** Violation at position 2: parent 3 is less than child 7.

(c) [5, 10, 8, 7, 6, 3, 4]

```
Check:
  a[1]=5 >= a[2]=10?            NO! 5 < 10.
```

**Not a valid max-heap.** Violation at the root: 5 < 10.

---

## Problem 5: Heapsort Comparison Count

**Problem.** When sorting an array of 15 elements using heapsort, what is the maximum number of compares during:
(a) Heap construction?
(b) Sortdown?

**Solution.**

(a) By Theorem 3.1, heap construction uses at most 2n = 30 compares.

More precisely, the bound is sum over heights: nodes at height h cost at most 2h compares. For n=15, the tree is a perfect binary tree of height 3:
- 8 nodes at height 0: 0 compares each = 0
- 4 nodes at height 1: 2 compares each = 8
- 2 nodes at height 2: 4 compares each = 8
- 1 node at height 3: 6 compares each = 6

Total: 0 + 8 + 8 + 6 = **22 compares** (tighter than the 2n=30 bound).

(b) During sortdown, we perform 14 delMax operations. Each calls sink on a heap of decreasing size. The maximum compares for each sink on a heap of size m is 2 floor(lg m).

Total: 2*floor(lg 14)* + *2*floor(lg 13) + ... + 2*floor(lg 2)* + *2*floor(lg 1)
= 2*(3+3+3+3+3+3+3+2+2+2+2+1+1+0)*
= *2*28
= **56 compares**

Combined maximum: 22 + 56 = **78 compares** (well within the 2n lg n + 2n ~ 117 bound).

---

## Problem 6: Indexed Priority Queue Trace

**Problem.** Trace the following operations on an indexed min-priority queue (maxN = 5):

```
insert(0, 7)
insert(1, 3)
insert(2, 9)
insert(3, 1)
delMin()
decreaseKey(2, 2)
delMin()
```

**Solution.**

We track `keys[]`, `pq[]` (heap position -> index), and `qp[]` (index -> heap position).

```
insert(0, 7):
  keys = [7, -, -, -, -]
  pq = [-, 0, -, -, -, -]   (position 1 holds index 0)
  qp = [1, -1, -1, -1, -1]
  n = 1. No swim needed.

insert(1, 3):
  keys = [7, 3, -, -, -]
  pq = [-, 0, 1, -, -, -]
  qp = [1, 2, -1, -1, -1]
  n = 2. swim(2): keys[pq[1]]=7 > keys[pq[2]]=3, so swap heap positions.
  pq = [-, 1, 0, -, -, -]
  qp = [2, 1, -1, -1, -1]

insert(2, 9):
  keys = [7, 3, 9, -, -]
  pq = [-, 1, 0, 2, -, -]
  qp = [2, 1, 3, -1, -1]
  n = 3. swim(3): keys[pq[1]]=3 < keys[pq[3]]=9, stop.

insert(3, 1):
  keys = [7, 3, 9, 1, -]
  pq = [-, 1, 0, 2, 3, -]
  qp = [2, 1, 3, 4, -1]
  n = 4. swim(4): keys[pq[2]]=7 > keys[pq[4]]=1, swap.
  pq = [-, 1, 3, 2, 0, -]
  qp = [4, 1, 3, 2, -1]
  swim(2): keys[pq[1]]=3 > keys[pq[2]]=1, swap.
  pq = [-, 3, 1, 2, 0, -]
  qp = [4, 2, 3, 1, -1]
  swim(1): stop (root).

  State: Heap order by keys: 1, 3, 9, 7
  pq: [-, 3, 1, 2, 0, -]  => position 1 has index 3 (key=1), position 2 has index 1 (key=3), etc.

delMin():
  min_index = pq[1] = 3 (key=1). Return 3.
  Swap heap positions 1 and 4.
  pq = [-, 0, 1, 2, 3, -]
```

```
  qp = [1, 2, 3, 4, -1]
  n = 3. qp[3] = -1, keys[3] = null, pq[4] = -1.
  sink(1, 3): keys[pq[1]]=7, children: keys[pq[2]]=3, keys[pq[3]]=9. Smaller: 3 at j=2.
              7 > 3, swap positions 1 and 2.
  pq = [-, 1, 0, 2, -, -]
  qp = [2, 1, 3, -1, -1]
  k=2, children at 4: 2*2=4 > 3, stop.

  State: keys = [7, 3, 9, -, -], heap: 3, 7, 9

decreaseKey(2, 2):
  keys[2] = 2.  qp[2] = 3.
  swim(3): keys[pq[1]]=3 > keys[pq[3]]=2? Yes, but position 3's parent is 3/2=1.
           keys[pq[1]]=3, keys[pq[3]]=2. 3 > 2, swap positions 1 and 3.
  pq = [-, 2, 0, 1, -, -]
  qp = [2, 3, 1, -1, -1]
  swim(1): stop.

  State: keys = [7, 3, 2, -, -], heap: 2, 7, 3

delMin():
  min_index = pq[1] = 2 (key=2). Return 2.
  Swap heap positions 1 and 3.
  pq = [-, 1, 0, 2, -, -]
  qp = [2, 1, 3, -1, -1]
  n = 2. qp[2] = -1, keys[2] = null, pq[3] = -1.
  sink(1, 2): keys[pq[1]]=3, child: keys[pq[2]]=7. 3 < 7, stop.

  Final state: keys = [7, 3, -, -, -], heap: 3, 7
  Remaining items: index 0 (key=7) and index 1 (key=3).
```

---

## Problem 7: Top-K with a Heap

**Problem.** Using a min-heap of size 3, find the 3 largest values from the stream: 4, 1, 8, 3, 9, 2, 7, 5.

**Solution.**

```
Process 4: heap = [4].                    Size < 3, insert.
Process 1: heap = [1, 4].                 Size < 3, insert.
Process 8: heap = [1, 4, 8].             Size = 3, insert.
Process 3: min = 1. 3 > 1, delMin & insert. heap = [3, 4, 8].
           (After delMin: [4, 8]. Insert 3: [3, 8, 4]. swim: 3<8, stop.)
           Let me just track the contents: {3, 4, 8}
Process 9: min = 3. 9 > 3, delMin & insert. heap = {4, 8, 9}.
Process 2: min = 4. 2 < 4, skip.          heap = {4, 8, 9}.
Process 7: min = 4. 7 > 4, delMin & insert. heap = {7, 8, 9}.
Process 5: min = 7. 5 < 7, skip.          heap = {7, 8, 9}.

Final answer: The 3 largest are {7, 8, 9}.
```

---

## Problem 8: Prove a Heap Property

**Problem.** Prove that the second-largest key in a max-heap of n >= 2 elements must be a child of the root.

**Proof.** Let M be the maximum key (at the root) and let S be the second-largest key, located at some position p in the heap. We will show that p must be 2 or 3 (i.e., a child of the root).

Suppose for contradiction that S is not a child of the root. Then S has a parent node at position p/2, and that parent is not the root. By the heap-order property, the parent's key >= S. But the parent is also not the root, so the parent's key < M (unless the parent's key equals M, which would mean a tie for the maximum, but then S is not the unique second-largest).

Wait, let me handle this more carefully. The parent of S at position p/2 has key >= S. If p/2 != 1 (i.e., the parent is not the root), then the parent's key is in the range [S, M]. If the parent's key = S, that is fine (ties). If the parent's key > S, then the parent's key is strictly between S and M, which contradicts S being the second-largest distinct key.

But the problem says "second-largest key," which could include duplicates. Let us interpret it as: there is no key strictly between S and M.

Case 1: If the parent's key = M, then the parent also holds a maximum key. But the parent is not the root, so the parent's own parent has key >= M, and by induction up to the root, which has key M. This is consistent

(multiple copies of M). But then S is not the parent, and we continue: S's grandparent has key >= parent's key = M >= S. Still consistent, but S could be anywhere.

Actually, the cleaner statement is: **the second-largest key in a max-heap must be at position 2 or 3 (a child of the root), assuming all keys are distinct.**

*Proof (distinct keys).* Let the root a[1] be the unique maximum. Consider any node at position p where p != 2 and p != 3. The path from p to the root passes through p's parent (at p/2), which is not the root. The parent's key a[p/2] > a[p] (by heap order, with distinct keys). Since a[p/2] > a[p] and a[p/2] < a[1], the node at position p cannot be the second-largest, because a[p/2] is strictly larger than a[p] and strictly smaller than a[1], so a[p/2] is a better candidate for second-largest. Therefore the second-largest must be at position 2 or 3. QED.

---

# Summary of Key Results

| Topic | Key Result |
|---|---|
| Binary heap insert | O(log n) --- at most 1 + lg n compares |
| Binary heap delete-max | O(log n) --- at most 2 lg n compares |
| Bottom-up heap construction | O(n) --- at most 2n compares |
| Heapsort | O(n log n) worst case, in place |
| Heapsort compares | At most 2n lg n + 2n |
| Indexed PQ operations | O(log n) for insert, delete, changeKey |
| Top-K selection | O(N log K) time, O(K) space |
| Multiway merge | O(N log K) for K streams |

Heapsort occupies a unique position in the sorting landscape: it is the only comparison-based sort that is simultaneously worst-case O(n log n) and in-place. While quicksort is faster in practice due to better cache behavior, heapsort provides a guaranteed performance bound that makes it valuable in real-time systems and as a fallback in hybrid sorting algorithms like introsort.

The priority queue abstraction, implemented via binary heaps, is one of the most versatile data structures in computer science. Its applications span operating systems, network routing, graph algorithms, data compression, simulation, and computational geometry. Mastering the heap is essential for any serious study of algorithms.

---

# References


1. **Sedgewick, R., & Wayne, K.** (2011). *Algorithms*, 4th Edition. Addison-Wesley. Chapter 2.4: Priority Queues.

2. **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapter 6: Heapsort.

3. **Williams, J. W. J.** (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7(6), 347-349.

4. **Floyd, R. W.** (1964). Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12), 701.

5. **Musser, D. R.** (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8), 983-993. (The introsort algorithm that uses heapsort as a fallback.)

6. **Sedgewick, R.** (2002). *Algorithms in Java*, 3rd Edition. Addison-Wesley. Parts 1-4.

7. **Knuth, D. E.** (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley. Section 5.2.3: Sorting by Selection (Heapsort).


---

*End of Lecture 5*