Outline

• Priority Queue
• Binary Heaps
• Implementation and demo
• HeapSort
Example 1: Scheduling

• **EDF (Earliest Deadline First) Scheduling**
  • Tasks wait in the queue
  • A task with a shorter deadline has a higher priority
  • Executes a job with the earliest deadline

\[ T_1 \ldots T_3 \quad T_2 \quad \ldots \quad T_n \quad \ldots \]
• Task $T_1$ is dispatched and removed from the Task waiting queue.

• Before $T_1$ is completed, Task $T_{n+1}$ arrives. It has the earliest deadline. $T_{n+1}$ will be dispatched next.
Priority Queue

• EDF scheduler processes Tasks in order. But not necessarily in full sorted order and not necessarily all at once.

• An appropriate data type for Task Waiting Queue supports two operations: remove the maximum priority task and insert new tasks. Such a data type is called a priority queue.

• Priority queues are characterized by the remove the maximum and insert operations.
public interface PriorityQueue <T extends Comparable<T> >
{
    void insert(T t);
    void remove() throws EmptyQueueException;
    T top() throws EmptyQueueException;
    boolean empty();
}
Example 2: Statistics

- Find the largest $M$ items in a stream of $N$ items (N huge, M large)
  - $N$ is huge, cannot sort in memory
  - $M$ is large, insert, remove must be fast.

Order of growth of finding the largest $M$ in a stream of $N$ items

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>$N \log N$</td>
<td>$M$</td>
</tr>
<tr>
<td>Array</td>
<td>$N M$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
Elementary Implementations

- Unordered Array:
  - Insert: $1$
  - Remove Max: $N$
  - Max: $N$

- Ordered Array:
  - Insert: $N$
  - Remove Max: $1$
  - Max: $1$

- Linked List (unsorted):
  - Insert: $1$
  - Remove Max: $N$
  - Max: $N$

- Binary Tree

Order-of-growth of running time for priority queue with $N$ items

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered Array</td>
<td>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Ordered Array</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Linked List (unsorted)</td>
<td>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Goal</td>
<td>Log $N$</td>
<td>Log $N$</td>
<td>$1$</td>
</tr>
</tbody>
</table>
Binary Heap

- Complete Binary Tree
- Each node is larger than (or equal to) its two children (if any).
Complete Binary Tree in Nature
Binary Heap Properties

• The largest is found at the root.
• Height of complete tree with \(N\) nodes is \([\lg N]\)
• Height only increases when \(N\) is a power of 2
Binary Heap Representations

- Array representation of a complete binary tree
  - Take nodes in level order
  - No explicit links needed
Binary Heap Representations

- Largest key is \(a[1]\), which is root of binary tree.
- Can use array indices to move through tree.
- Parent of node at \(k\) is at \(k/2\).
- Two children of the node at \(k\) are in positions \(2k\) and \(2k + 1\).
**Promotion**: Child's key becomes larger key than its parent's key.

To eliminate the violation:
- Exchange key in child with key in parent.
- Repeat until heap order restored.

```java
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        swap(k, k/2);
        k = k/2;
    }
}
```
Insertion in a heap:
- Insert. Add node at end, then swim it up.
- Cost. At most $\lg N$ compares.

```
public void insert(T t) {
    pqArray.add(t);
    Size++;
    swim(Size);
}
```
**Demotion:** Parent's key becomes smaller than one (or both) of its children's keys.

To eliminate the violation:
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```java
private void sink(int k) {
    while (2 * k <= Size) {
        int j = 2 * k;
        if (j < Size && less(j, j + 1)) j++;
        if (!less(k, j)) break;
        swap(k, j);
        k = j;
    }
}
```
Remove the maximum in a heap:
- Delete max: Replace root with node at end, then sink it down.
- Cost: At most 2 lg N compares.

```java
public void remove(){
    if(Size == 0){
        throw new EmptyQueueException("Queue is empty.");
    }
    pqArray.set(1,pqArray.get(Size));
pqArray.remove(Size);
    Size--;
sink(1);
}
```
Insertion

Binary Heap Demo

Priority Queue

Insert 34

Violation.

swim
Binary Heap Demo

Insertion

Violation.
swim
Binary Heap Demo

Insertion

Violation.

swim
Binary Heap Demo

Insertion

35

26

15

12

1

24

23

21

2

33

4

34

5

Done!

Violation.

swim

Prioriry Queue
Binary Heap Demo

Remove max:

Delete the last leaf
Move the last leaf to root
Binary Heap Demo

Remove

Violation. sink

Violation. sink

7/12/20
Binary Heap Demo

Remove

Violation. sink

Remove

Priority Queue

7/12/20
Binary Heap Demo

Remove

Prioriry Queue
## Binary Heap Java Code Demo

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue.java</td>
<td>Interface</td>
</tr>
<tr>
<td>MaxPQ.java</td>
<td>PQ implementation</td>
</tr>
<tr>
<td>GraphVizWrite.java</td>
<td>Visualize the heap</td>
</tr>
<tr>
<td>EmptyQueueException.java</td>
<td>Exception</td>
</tr>
<tr>
<td>MaxPQTest.java</td>
<td>main method</td>
</tr>
<tr>
<td>InputHelper.java</td>
<td>input utility</td>
</tr>
</tbody>
</table>
Cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered Array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Ordered Array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Linked List (unsorted)</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td><strong>Binary Heap</strong></td>
<td>Log N</td>
<td>Log N</td>
<td>1</td>
</tr>
</tbody>
</table>
Immutability of keys

- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Immutability: implementing in Java
- Immutable data type. Can't change the data type value once created.
- Immutable. `String`, `Integer`, `Double`, `Color`, `Vector`, `Transaction`, `Point2D`.
- Mutable. `StringBuilder`, `Stack`, `Counter`, Java array.
Heap Sort

- Sort an array using heap representations
- worst case running time $O(n \log n)$
- an **in-place** sorting algorithm: only a constant number of array elements are stored outside the input array at any time. thus, require at most $O(1)$ additional memory
**Heap Sort**

- **Idea:**
  1. Create max-heap with all N keys.
  2. Repeatedly remove the maximum key.

**Original Array**

```
Original Array
```

```
Build a Max Heap
```

```
Sorted Array
```

```
Prioriry Queue
```
Step 1: Build max-heap

Build heap using bottom-up method
for (int k = N/2; k >= 1; k--)
sink(k, N);

Arbitrary Array
Step 1: Build max-heap

Build heap using bottom-up method

for (int k = N/2; k >= 1; k--)
sink(k, N);

Arbitrary Array
Step 1: Build max-heap

Build heap using bottom-up method

for (int k = N/2; k >= 1; k--)
    sink(k, N);

Arbitrary Array

Max-heap
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

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

Heap ordered array
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

```
while (N > 1) {
    exch(1, N--);
    sink(1, N);
}
```

Heap ordered array
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

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

Heap ordered array
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

```java
while (N > 1) {
    exch(1, N--);
    sink(1, N);
}
```

Heap ordered array

Priority Queue
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

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

Heap ordered array
Step 2: Sortdown

Remove the maximum, one at a time
Leave in array, instead of nulling out.

```java
while (N > 1) {
    exch(1, N--);
    sink(1, N);
}
```

Heap ordered array

Sorted Result

Prioriry Queue
Implementation and Demo
• Heap construction uses fewer than 2 N compares and exchanges.
• Heapsort uses at most 2 N lg N compares and exchanges.

**Significance:**
• In-place sorting algorithm with N log N worst-case.
• Mergesort: no, linear extra space.
• Quicksort: no, quadratic time in worst case.
• Heapsort: yes
• Heapsort is optimal for both time and space,

**Disadvantages:**
• Makes poor use of cache memory.
• Not stable.
## Sorting Algorithms Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>In place</th>
<th>Stable</th>
<th>Worst</th>
<th>Average</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td>$n^2$</td>
<td>2NlogN</td>
<td>NlogN</td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td>NlogN</td>
<td>NlogN</td>
<td>NlogN</td>
<td>NlogN</td>
</tr>
<tr>
<td>Heap Sort</td>
<td></td>
<td></td>
<td>2NlogN</td>
<td>2NlogN</td>
<td>NlogN</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td>NlogN</td>
<td>NlogN</td>
<td>NlogN</td>
<td>NlogN</td>
</tr>
</tbody>
</table>