Priority Queue ADT

- A priority queue stores a collection of elements which have a total order.
- Each element has a key value, key(x).
- Main methods of the Priority Queue ADT
  - `insert(x)` inserts an entry with key k and value x
  - `removeMin()` removes and returns the element with smallest key.
- This is the min-queue. Replace “min” by “max” we obtain the max-queue.

- Additional methods
  - `min()` returns, but does not remove, an entry with smallest key
  - `size()`
  - `isEmpty()`

- Applications:
  - Standby flyers
  - Auctions
  - Stock market
Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Every pair of such keys must be comparable according to a total order.

Definition of total order relation ≤

- **Comparability** property: either \( x \leq y \) or \( y \leq x \)
- **Reflexive** property: \( x \leq x \)
- **Antisymmetric** property: \( x \leq y \) and \( y \leq x \Rightarrow x = y \)
- **Transitive** property: \( x \leq y \) and \( y \leq z \Rightarrow x \leq z \)

Example

- A sequence of priority queue methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Priority Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5,A)</td>
<td>(3,B)</td>
<td>{ (5,A) }</td>
</tr>
<tr>
<td>insert(9,C)</td>
<td>(3,B)</td>
<td>{ (5,A), (9,C) }</td>
</tr>
<tr>
<td>insert(3,B)</td>
<td>(3,B)</td>
<td>{ (3,B), (5,A), (9,C) }</td>
</tr>
<tr>
<td>min( )</td>
<td>(5,A)</td>
<td>{ (5,A), (9,C) }</td>
</tr>
<tr>
<td>removeMin( )</td>
<td>(7,D)</td>
<td>{ (5,A), (7,D), (9,C) }</td>
</tr>
<tr>
<td>insert(7,D)</td>
<td>(5,A)</td>
<td>{ (5,A), (7,D), (9,C) }</td>
</tr>
<tr>
<td>removeMin( )</td>
<td>(9,C)</td>
<td>{ (7,D), (9,C) }</td>
</tr>
<tr>
<td>removeMin( )</td>
<td>null</td>
<td>{ (9,C) }</td>
</tr>
<tr>
<td>isEmpty( )</td>
<td>true</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Priority Queue Sorting

- We can use a priority max-queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMax operations
- The running time of this sorting method depends on the priority queue implementation.

Algorithm PQ-Sort$(S, C)$

Input sequence $S$, comparator $C$ for the elements of $S$
Output sequence $S$ sorted in increasing order according to $C$

$P \leftarrow$ priority queue with comparator $C$

while $\neg S$ is Empty()
    $e \leftarrow S$.removeFirst()
    $P$.insert($e$)

while $\neg P$ is Empty()
    $e \leftarrow P$.removeMax()
    $S$.insertFirst($e$)

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer’s main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is $o(n)$, where $n$ is the input size.
Sequence-based Priority Queue

- Implementation with an unsorted list
  - Performance:
    - \textbf{insert} takes $O(1)$ time since we can insert the item at the beginning or end of the sequence.
    - \textbf{removeMax} takes $O(n)$ time since we have to traverse the entire sequence to find the smallest key.

- Implementation with a sorted list
  - Performance:
    - \textbf{insert} takes $O(n)$ time since we have to find the place where to insert the item.
    - \textbf{removeMax} takes $O(1)$ time, since the smallest key is at the beginning.

How does the Priority Queue Sorting behave?

Selection-Sort, Insertion-Sort

- Selection-sort is a variation of PQ-sort where the priority queue is implemented with an unsorted sequence.
  - If an array is used, it can be implemented as in-place selection sort.
- Insertion-sort is a variation of PQ-sort where the priority queue is implemented with a sorted sequence.
  - If an array is used, it can be implemented as in-place insertion sort.
### Selection-Sort Example

<table>
<thead>
<tr>
<th>Input:</th>
<th>Priority Queue P</th>
<th>Sorted Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7, 4, 8, 2, 5, 3, 9)</td>
<td>(7, 4, 8, 2, 5, 3)</td>
<td>(9)</td>
</tr>
<tr>
<td></td>
<td>(7, 4, 8, 2, 5)</td>
<td>(8, 9)</td>
</tr>
<tr>
<td></td>
<td>(7, 4, 2, 5)</td>
<td>(7, 8, 9)</td>
</tr>
<tr>
<td></td>
<td>(4, 2, 5, 3)</td>
<td>(4, 5, 7, 8, 9)</td>
</tr>
<tr>
<td></td>
<td>(4, 2, 3)</td>
<td>(4, 5, 7, 8, 9)</td>
</tr>
<tr>
<td></td>
<td>(2, 3)</td>
<td>(2, 3, 4, 5, 7, 8, 9)</td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>(2, 3, 4, 5, 7, 8, 9)</td>
</tr>
</tbody>
</table>

### Insertion-Sort Example

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sequence S</th>
<th>Priority queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7, 4, 8, 2, 5, 3, 9)</td>
<td>(4, 8, 2, 5, 3, 9)</td>
<td>(7)</td>
</tr>
<tr>
<td></td>
<td>(8, 2, 5, 3, 9)</td>
<td>(4, 7)</td>
</tr>
<tr>
<td></td>
<td>(2, 5, 3, 9)</td>
<td>(4, 7, 8)</td>
</tr>
<tr>
<td></td>
<td>(5, 3, 9)</td>
<td>(2, 4, 7, 8)</td>
</tr>
<tr>
<td></td>
<td>(3, 9)</td>
<td>(2, 4, 5, 7, 8)</td>
</tr>
<tr>
<td></td>
<td>(9)</td>
<td>(2, 3, 4, 5, 7, 8)</td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>(2, 3, 4, 5, 7, 8, 9)</td>
</tr>
</tbody>
</table>
Balanced Search Tree Based Priority Queue

- Both insert and removeMax can be implemented using $O(\log n)$ time.
- Thus, PQ-sort can run in $O(n \log n)$.
- Can we have an in-place PQ-sort whose complexity is in $O(n \log n)$?
  - Yes, use heaps for PQ.

What is a heap?

- A (max) heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - (Max) Heap-Order: for every node $v$ other than the root, $key(v) \leq key(parent(v))$
  - Complete Binary Tree: let $h$ be the height of the heap
    - for $i = 0, \ldots, h - 2$, there are $2^i$ nodes of depth $i$
    - at depth $h-1$, the nodes are listed from left to right without gaps.
  - The last node of a heap is the rightmost node of depth $h - 1$. 

Height of a Heap

- **Theorem:** A heap storing \( n \) keys has height \( O(\log n) \)

  **Proof:** (we apply the complete binary tree property)
  - Let \( h \) be the height of a heap storing \( n \) keys
  - Since there are \( 2^i \) keys at depth \( i = 0, \ldots, h - 1 \) and at least one key at depth \( h \), we have \( n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 \)
  - Thus, \( n \geq 2^h \), i.e., \( h \leq \log n \).

<table>
<thead>
<tr>
<th>depth</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( h-1 )</td>
<td>( 2^{h-1} )</td>
</tr>
<tr>
<td>( h )</td>
<td>1</td>
</tr>
</tbody>
</table>

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store an item (key, element) at each node
- We keep track of the position of the last node
- For simplicity, we will show only the keys in the pictures

A min-heap:
**Insert into a Heap**

- Method \texttt{insert} of the priority queue ADT corresponds to the insertion of a key \( k \) to the heap.
- The insertion algorithm consists of three steps:
  - Find the position for a new node and create a new node \( z \).
  - Store \( k \) at \( z \).
  - Restore the heap-order property by up-heap bubble (discussed next).

**Up-Heap Bubbling**

- After the insertion of a new key \( k \), the heap-order property may be violated.
- Algorithm \texttt{up-heap-bubble} restores the heap-order property by swapping \( k \) along an upward path from the insertion node.
- \texttt{Up-heap-bubble} terminates when the key \( k \) reaches the root or a node whose key is greater than or equal to \( k \).
- Since a heap has height \( O(\log n) \), \texttt{up-heap-bubble} runs in \( O(\log n) \) time.
**removeMax from a Heap**

- Method `removeMax` of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps:
  - Replace the root key with the key of the last node `w`.
  - Release node `w`.
  - Restore the heap-order property by **down-heap-bubble** (discussed next).

**Down-heap bubbling (Heapify)**

- After replacing the root key with the key `k` of the last node, the heap-order property may be violated.
- Algorithm **down-heap-bubble** (or heapify) restores the heap-order property by swapping key `k` along a downward path from the root.
- **Down-heap-bubble** terminates when key `k` reaches a leaf or a node whose key is less than or equal to `k`.
- Since a heap has height $O(\log n)$, **down-heap-bubble** runs in $O(\log n)$ time.
Heap-Sort

- Consider a priority queue with \( n \) items implemented by means of a max-heap
  - the additional space used is \( O(n) \)
  - methods insert and removeMax take \( O(\log n) \) time.
- Using a heap-based priority queue, we can sort a sequence of \( n \) elements in \( O(n \log n) \) time
- It can be implemented in-place (\( O(1) \) additional space).
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort, when \( n \) is very large.

Array-based Heap Implementation

- We can represent a heap with \( n \) keys by means of an array of length \( n \).
- For the node at index \( i \)
  - the left child is at index \( 2i + 1 \)
  - the right child is at index \( 2i + 2 \)
- Links between nodes are not explicitly stored
- The (first portion of) input array \( A \) is used as heap.
- In-place (no additional array is needed) heap-sort:
  - For \( k = 1 \) to \( n-1 \)
    - \( A \).insert(\( A[k] \));
  - For \( k = n-1 \) downto \( 1 \)
    - \( A[k] = A \).removeMax();
- Time Complexity: \( O(n \log n) \)
We can construct a heap storing \( n \) given keys using a bottom-up construction with \( \log n \) phases.

In phase \( i \), pairs of heaps with \( 2^i - 1 \) keys plus one item are merged into heaps with \( 2^{i+1} - 1 \) keys.

### Merging Two (Min) Heaps

- We are given two two heaps and a key \( k \).
- We create a new heap with the root node storing \( k \) and with the two heaps as subtrees.
- We perform down-heap-bubble to restore the heap-order property.
Example of Max Heap

\[ A = [10, 7, 8, 25, 5, 11, 27, 16, 15, 4, 12, 6, 7, 23, 20] \]

Example (contd.)
Example (contd.)

Example (end)
Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[\lceil n/2 \rceil + 1 \ldots n]$ are leaves
- Apply MaxHeapify on elements between 1 and $\lfloor n/2 \rfloor$

**Alg:** BuildMaxHeap($A$)

1. $n = \text{length}[A]$
2. for $i \leftarrow \lfloor n/2 \rfloor$ downto 0
3. do MaxHeapify($A, i, n$)

---

Maintaining the Heap Property

- **Assumptions:**
  - Left and Right subtrees of $i$ are max-heaps
  - $A[i]$ may be smaller than its children

**Alg:** MaxHeapify($A, i, n$) {
1. $l \leftarrow \text{Left}(i)$;  // Left($i$) = $2i+1$
2. $r \leftarrow \text{Right}(i)$;  // Right($i$) = $2i+2$
3. max $\leftarrow i$;
4. if ($l < n$ && $A[l] > A[\text{max}]$) max $\leftarrow l$;
5. if ($r < n$ && $A[r] > A[\text{max}]$) max $\leftarrow r$;
6. if (max $\neq i$) {
7. exchange $A[i] \leftrightarrow A[\text{max}]$;
8. MaxHeapify($A, \text{max}, n$);
9. }
Running Time of BUILD MAX HEAP

**Alg:** BuildMaxHeap(A)

1. \( n = \text{length}[A] \)
2. \( \text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1 \)
3. \( \text{do MaxHeapify}(A, i, n) \quad O(\log n) \) \( \{ O(n) \) 

\( \Rightarrow \) Running time: \( O(n \log n) \)

- This is not an asymptotically tight upper bound

Analysis

- We visualize the worst-case time of a heapify (or sift-down) with a given path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual heapify path).
- Since each edge is traversed by at most once by these paths, the total length of these paths is \( O(n) \).
- Thus, bottom-up heap construction runs in \( O(n) \) time.
- Bottom-up heap construction is faster than \( n \) successive insertions and speeds up the first phase of heap-sort.
Running Time of BUILD MAX HEAP

MaxHeapify takes \( O(h) \) ⇒ the cost of MaxHeapify on a node \( i \) is proportional to the height of the node \( i \) in the tree

\[
T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h - i) = O(n)
\]

<table>
<thead>
<tr>
<th>Height</th>
<th>Level</th>
<th>No. of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_0 = 3 (\lfloor \log n \rfloor) )</td>
<td>( i = 0 )</td>
<td>( 2^0 )</td>
</tr>
<tr>
<td>( h_1 = 2 )</td>
<td>( i = 1 )</td>
<td>( 2^1 )</td>
</tr>
<tr>
<td>( h_2 = 1 )</td>
<td>( i = 2 )</td>
<td>( 2^2 )</td>
</tr>
<tr>
<td>( h_3 = 0 )</td>
<td>( i = 3 (\lfloor \log n \rfloor) )</td>
<td>( 2^3 )</td>
</tr>
</tbody>
</table>

- \( h_i = h - i \) height of the heap rooted at level \( i \)
- \( n_i = 2^i \) number of nodes at level \( i \)

\[
T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h - i) \leq n \sum_{k=0}^{\infty} \frac{k}{2^k}
\]

- The sum above is smaller than the sum of all elements to \( \infty \)
- The sum above is smaller than 2

Running time of BuildMaxHeap: \( T(n) = O(n) \)
HeapSort(A)

- Convert an array $A[0 \ldots n-1]$ into a max-heap
  - The elements in the subarray $A[\left\lfloor n/2 \right\rfloor \ldots n-1]$ are leaves.
  - Apply MaxHeapify on elements between 0 and $\left\lfloor n/2 \right\rfloor - 1$
- Repeatedly swap the max heap element with the last unsorted element and call MaxHeapify to maintain the heap property.

```
Alg: HeapSort(A) {
    1. $n = A$.length;
    2. for $i \leftarrow \left\lfloor n/2 \right\rfloor$ downto 0
       MaxHeapify(A, i, n);
    3. for $i \leftarrow n - 1$ downto 1 {       // $A[0..i]$ is a max heap
       MaxHeapify(A, 0, i);       // $A[i..n-1]$ is sorted with max $(n - i)$
    4. }} // elements of the original array.
```

Example: $A=[7, 4, 3, 1, 2]$

MaxHeapify(A, 1, 4) MaxHeapify(A, 1, 3) MaxHeapify(A, 1, 2)

MaxHeapify(A, 1, 1)
Stability

- A **STABLE sort** preserves relative order of records with equal keys.

<table>
<thead>
<tr>
<th>Name</th>
<th>Key</th>
<th>Phone</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaron</td>
<td>A</td>
<td>604-469-0023</td>
<td>Little</td>
</tr>
<tr>
<td>Andrew</td>
<td>A</td>
<td>874-069-1212</td>
<td>Miozma</td>
</tr>
<tr>
<td>Battle</td>
<td>O</td>
<td>212-878-4944</td>
<td>Blair</td>
</tr>
<tr>
<td>Chen</td>
<td>A</td>
<td>656-232-5141</td>
<td>Zhang</td>
</tr>
<tr>
<td>Fox</td>
<td>A</td>
<td>243-465-9091</td>
<td>Brown</td>
</tr>
<tr>
<td>Furia</td>
<td>A</td>
<td>766-091-9873</td>
<td>Brown</td>
</tr>
<tr>
<td>Gatal</td>
<td>B</td>
<td>645-303-0266</td>
<td>Walker</td>
</tr>
<tr>
<td>Hasha</td>
<td>B</td>
<td>898-123-5464</td>
<td>Farlow</td>
</tr>
<tr>
<td>Smith</td>
<td>A</td>
<td>232-441-5555</td>
<td>Miller</td>
</tr>
<tr>
<td>Zillini</td>
<td>C</td>
<td>343-587-6542</td>
<td>McVick</td>
</tr>
</tbody>
</table>

Sorted on first key:

Sort file on second key:

Records with key value 3 are not in order on first key!!

Summary

- A priority queue stores a collection of items.
- Each item has a key value.
- Main methods of the Priority Queue ADT:
  - `insert(x)`
    - inserts an item `x`
  - `removeMin()` (or `removeMax()`)
    - removes and returns the item with smallest (or max) key
- Using an array-based priority queue, each insert and `removeMin` can be implemented in O(log n).
- For Heap Sort, we create an array-based max heap in O(n) and each `removeMax` takes O(log n), so the total time is O(n log n).
- Heap Sort is a non-stable, in-place, optimal sorting method.