Algorithms
(Priority Queues)

Pramod Ganapathsi
Department of Computer Science
State University of New York at Stony Brook

January 15, 2021
Contents

- Priority Queue ADT
- Implementations of a Priority Queue
  - Using an Unsorted List
  - Using a Sorted List
  - Using an Array-Based Heap
  - Using a Tree-Based Heap
- Sorting Based on Priority Queue
  - Selection Sort
  - Insertion Sort
  - Heap Sort
A priority queue is a tree-based data structure consisting of key-value pairs. A priority queue uses the whatever in, priority-out principle. A priority queue has two major operations: insert and delete-min.
Applications of priority queues

- Flight queue with customer priority
- Call center queue with customer priority
- Technical support queue with customer priority
- Vaccination queue with citizen priority
- All applications of sorting
## Priority queue ADT

<table>
<thead>
<tr>
<th>Method</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(k, v)</code></td>
<td>Creates an entry with key k and value v in the priority queue.</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the number of entries in the priority queue.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns a boolean indicating whether the priority queue is empty.</td>
</tr>
</tbody>
</table>
### Operations on a priority queue contents

<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></td>
<td>{ (5,A) }</td>
</tr>
<tr>
<td>insert(9,C)</td>
<td></td>
<td>{ (5,A), (9,C) }</td>
</tr>
<tr>
<td>insert(3,B)</td>
<td></td>
<td>{ (3,B), (5,A), (9,C) }</td>
</tr>
<tr>
<td>min()</td>
<td>(3,B)</td>
<td>{ (3,B), (5,A), (9,C) }</td>
</tr>
<tr>
<td>removeMin()</td>
<td>(3,B)</td>
<td>{ (5,A), (9,C) }</td>
</tr>
<tr>
<td>insert(7,D)</td>
<td></td>
<td>{ (5,A), (7,D), (9,C) }</td>
</tr>
<tr>
<td>removeMin()</td>
<td>(5,A)</td>
<td>{ (7,D), (9,C) }</td>
</tr>
<tr>
<td>removeMin()</td>
<td>(7,D)</td>
<td>{ (9,C) }</td>
</tr>
<tr>
<td>removeMin()</td>
<td>(9,C)</td>
<td>{ }</td>
</tr>
<tr>
<td>removeMin()</td>
<td>null</td>
<td>{ }</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Priority queue complexity

<table>
<thead>
<tr>
<th>Method</th>
<th>Unsorted list</th>
<th>Sorted list</th>
<th>Array heap</th>
<th>Linked-tree heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>min</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(\log n)^*$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(\log n)^*$</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>

* $=$ amortized complexity
Priority queue using an unsorted list

We use a PositionalList which in turn uses DLL.
- insert method inserts a key-value entry at the end of the list in $O(1)$ time.
- min or removeMin requires scanning the entire list in $O(n)$ time.
Priority queue using a sorted list

<table>
<thead>
<tr>
<th>Method</th>
<th>Sorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>min</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>

- We use a **PositionalList** which in turn uses **DLL** but sorted by nondecreasing keys.  
- min or removeMin requires the retrieval or removal of the first element in $\mathcal{O}(1)$ time.  
- insert method inserts a key-value entry at the appropriate position after scanning the list in $\mathcal{O}(n)$ time.
Heap
Heap

<table>
<thead>
<tr>
<th>Method</th>
<th>Unsorted list</th>
<th>Sorted list</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- Unsorted list has excellent insert time but worse removeMin time. Sorted list has excellent removeMin time but worse insert time.
- **Is it possible to get the best of both the worlds?**
  - Nope! It is impossible to get the best of both the worlds. *(Why not?)*
  - However, we can definitely get the better of both the worlds using the **heap** data structure.
Heap: Example

(4,C)

(5,A)

(15,K) (16,X) (14,E)
(25,J) (12,H) (11,S) (13,W)

(6,Z)

(20,B)
A heap is a binary tree $T$ that satisfies two properties:

1. **Relational property.** Deals with how keys are stored in $T$.
   - **Heap-order property.** Key stored at a node is less than or equal to the keys stored at its child nodes.

2. **Structural property.** Deals with the shape of $T$.
   - **Almost-complete binary tree property.** A heap $T$ with height $h$ is an almost-complete binary tree if levels $0, 1, 2, \ldots, h - 1$ of $T$ have the maximal number of nodes possible (namely, level $i$ has $2^i$ nodes, for $0 \leq i \leq h - 1$) and the remaining nodes at level $h$ reside in the leftmost possible positions at that level.

A heap $T$ storing $n$ entries has height $h = \lfloor \log n \rfloor$. 
Heap: Insert

\[ \text{T.INSERT}(k, v) \]

1. Insert pair \((k, v)\) at the last node
2. Up-heap bubble until heap-order property is not violated

• Complexity of insert is \( \mathcal{O}(\log n) \)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: Insert (2, T)
Heap: RemoveMin

T.RemoveMin\((k, v)\)

1. Remove the root node
2. Move the last node to the root node position
3. Down-heap bubble until heap-order property is not violated

- Complexity of removeMin is \(\mathcal{O}(\log n)\)
Heap: RemoveMin
Heap: RemoveMin
Heap: RemoveMin
Heap: RemoveMin

(5,A)
(13,W)
(15,K)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(6,Z)
(7,Q)
(20,B)
Heap: RemoveMin

(5,A)

(13,W)

(16,X) (25,J) (14,E) (12,H) (11,S)

(9,F)
Heap: RemoveMin

(5,A)

(9,F)

(15,K)

(16,X)

(14,E)

(12,H)

(11,S)

(25,J)

(16,X)

(25,J)

(14,E)

(12,H)

(11,S)

(6,Z)

(20,B)
Heap: RemoveMin
Heap: RemoveMin
Heap: Array-based representation
Heap: Array-based representation

\[
f(p) = \begin{cases} 
0 & \text{if } p \text{ is the root}, \\
2f(q) + 1 & \text{if } p \text{ is the left child of position } q, \\
2f(q) + 2 & \text{if } p \text{ is the right child of position } q. 
\end{cases}
\]
Heap: Array-based implementation

```java
class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();

  public HeapPriorityQueue() { super(); }
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }

  public int size() { }
  public Entry<K,V> min() {...}
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {...}
  public Entry<K,V> removeMin() {...}

  protected void upheap(int j) {...}
  protected void downheap(int j) {...}
  protected void heapify() {...}
  protected void swap(int i, int j) {...}

  protected int parent(int j) {...}
  protected int left(int j) {...}
  protected int right(int j) {...}
  protected boolean hasLeft(int j) {...}
  protected boolean hasRight(int j) {...}
}
```
Heap: Array-based implementation

```java
/** Inserts a key-value pair and return the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key); // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    heap.add(newest); // add to the end of the list
    upheap(heap.size() - 1); // upheap newly added entry
    return newest;
}
```
Heap: Array-based implementation

```java
/** Moves the entry at index j higher, to restore the heap property. */
protected void upheap(int j) {
    while (j > 0) { // continue until reaching root (or break statement)
        int p = parent(j);
        if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
        swap(j, p);
        j = p; // continue from the parent’s location
    }
}
```
/** Removes and returns an entry with minimal key. */

```java
public Entry<K,V> removeMin() {
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() - 1); // put minimum item at the end
    heap.remove(heap.size() - 1); // and remove it from the list;
    downheap(0); // then fix new root
    return answer;
}
```
Heap: Array-based implementation

```java
/** Moves the entry at index j lower, to restore the heap property. */
protected void downheap(int j) {
    while (hasLeft(j)) { // continue to bottom (or break statement)
        int leftIndex = left(j);
        int smallChildIndex = leftIndex; // although right may be smaller
        if (hasRight(j)) {
            int rightIndex = right(j);
            if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
                smallChildIndex = rightIndex; // right child is smaller
        }
        if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0) break;
        swap(j, smallChildIndex);
        j = smallChildIndex; // continue at position of the child
    }
}
```
Heap: Array-based implementation

1. protected int parent(int j) { return (j-1) / 2; }  // truncating division
2. protected int left(int j) { return 2*j + 1; }
3. protected int right(int j) { return 2*j + 2; }
4. protected boolean hasLeft(int j) { return left(j) < heap.size(); }
5. protected boolean hasRight(int j) { return right(j) < heap.size(); }
## Heap: Complexity

<table>
<thead>
<tr>
<th>Method</th>
<th>Array heap</th>
<th>Linked-tree heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty, min</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

* = amortized complexity

- Up-heap and down-heap bubbling take $O(\log n)$ time.
- Array runtimes are amortized due to array resizing.
### java.util.PriorityQueue class

<table>
<thead>
<tr>
<th>Our Priority Queue ADT</th>
<th>java.util.PriorityQueue class</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(k,v)</td>
<td>add(new SimpleEntry(k,v))</td>
</tr>
<tr>
<td>min()</td>
<td>peek()</td>
</tr>
<tr>
<td>removeMin()</td>
<td>remove()</td>
</tr>
<tr>
<td>size()</td>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty()</td>
</tr>
</tbody>
</table>

- User-defined priority can be given to the class by sending a comparator object when constructing the priority queue.
- Key-value pair can be considered by using java.util.AbstractMap.SimpleEntry class
Priority queue sort

**Priority-Queue-Sort** (sequence $S$, priority queue $P$)

1. Insert the $n$ elements of $S$ into $P$
2. RemoveMin the $n$ elements of $P$ into $S$
/** Sorts sequence S, using initially empty priority queue P. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
    int n = S.size();
    for (int j = 0; j < n; j++) {
        E element = S.remove(S.first());
        P.insert(element, null); // element is key; null value
    }
    for (int j = 0; j < n; j++) {
        E element = P.removeMin().getKey();
        S.addLast(element); // the smallest key in P is next placed in S
    }
}
### Selection Sort

**Priorities:**

- **P** = unsorted list: Selection sort

**Sequence S**

- Input: $(7, 4, 8, 2, 5, 3, 9)$

**Priority Queue P**

- Insert $(7)$: $(7)$
- Insert $(4, 8, 2, 5, 3, 9)$: $(7, 4)$
- Insert $(8, 2, 5, 3, 9)$: $(7, 4)$
- Insert $(7, 4, 8, 2, 5, 3, 9)$

- Remove Min $(2)$: $(7, 4, 8, 5, 3, 9)$
- Remove Min $(2, 3)$: $(7, 4, 8, 5, 9)$
- Remove Min $(2, 3, 4)$: $(7, 4, 8, 5, 9)$
- Remove Min $(2, 3, 4, 5)$: $(7, 4, 8, 5, 9)$
- Remove Min $(2, 3, 4, 5, 7)$: $(7, 4, 8, 5, 9)$
- Remove Min $(2, 3, 4, 5, 7, 8)$: $(7, 4, 8, 5, 9)$
- Remove Min $(2, 3, 4, 5, 7, 8, 9)$: $(7, 4, 8, 5, 9)$

---

**Table:**

<table>
<thead>
<tr>
<th>Input</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>$(4, 8, 2, 5, 3, 9)$</td>
<td>$(7)$</td>
</tr>
<tr>
<td>(b)</td>
<td>$(8, 2, 5, 3, 9)$</td>
<td>$(7, 4)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>$(7, 4, 8, 2, 5, 3, 9)$</td>
</tr>
<tr>
<td><strong>Remove Min</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>$(2)$</td>
<td>$(7, 4, 8, 5, 3, 9)$</td>
</tr>
<tr>
<td>(b)</td>
<td>$(2, 3)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
<tr>
<td>(c)</td>
<td>$(2, 3, 4)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
<tr>
<td>(d)</td>
<td>$(2, 3, 4, 5)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
<tr>
<td>(e)</td>
<td>$(2, 3, 4, 5, 7)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
<tr>
<td>(f)</td>
<td>$(2, 3, 4, 5, 7, 8)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
<tr>
<td>(g)</td>
<td>$(2, 3, 4, 5, 7, 8, 9)$</td>
<td>$(7, 4, 8, 5, 9)$</td>
</tr>
</tbody>
</table>
P = unsorted list: Selection sort

- Phase 1 time = \( \sum_{i=1}^{n} O(1) = O(n) \)
- Phase 2 time = \( \sum_{i=n}^{1} O(i) = O(n^2) \)
- Total time = \( O(n^2) \)
**P = sorted list: Insertion sort**

<table>
<thead>
<tr>
<th>Input</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(7, 4, 8, 2, 5, 3, 9)</td>
<td>()</td>
</tr>
<tr>
<td>insert</td>
<td>(a)</td>
<td>(4, 8, 2, 5, 3, 9)</td>
</tr>
<tr>
<td></td>
<td>(b)</td>
<td>(8, 2, 5, 3, 9)</td>
</tr>
<tr>
<td></td>
<td>(c)</td>
<td>(2, 5, 3, 9)</td>
</tr>
<tr>
<td></td>
<td>(d)</td>
<td>(5, 3, 9)</td>
</tr>
<tr>
<td></td>
<td>(e)</td>
<td>(3, 9)</td>
</tr>
<tr>
<td></td>
<td>(f)</td>
<td>(9)</td>
</tr>
<tr>
<td></td>
<td>(g)</td>
<td>()</td>
</tr>
<tr>
<td>removeMin</td>
<td>(a)</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>(b)</td>
<td>(2, 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(g)</td>
<td>(2, 3, 4, 5, 7, 8, 9)</td>
</tr>
</tbody>
</table>
$P = \text{sorted list: Insertion sort}$

- Phase 1 time $= \sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$
- Phase 2 time $= \sum_{i=n}^{1} \mathcal{O}(1) = \mathcal{O}(n)$
- Total time $= \mathcal{O}(n^2)$
### P = heap: Heap sort

<table>
<thead>
<tr>
<th>Input</th>
<th>Sequence $S$</th>
<th>Priority Queue $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$(7, 4, 8, 2, 5, 3, 9)$</td>
<td>()</td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>$(4, 8, 2, 5, 3, 9)$</td>
<td>$(7)$</td>
</tr>
<tr>
<td>(b)</td>
<td>$(8, 2, 5, 3, 9)$</td>
<td>$(4, 7)$</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>$(2, 4, 3, 7, 5, 8, 9)$</td>
</tr>
<tr>
<td>removeMin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>$(2)$</td>
<td>$(3, 4, 8, 7, 5, 9)$</td>
</tr>
<tr>
<td>(b)</td>
<td>$(2, 3)$</td>
<td>$(4, 5, 8, 7, 9)$</td>
</tr>
<tr>
<td>(c)</td>
<td>$(2, 3, 4)$</td>
<td>$(5, 7, 8, 9)$</td>
</tr>
<tr>
<td>(d)</td>
<td>$(2, 3, 4, 5)$</td>
<td>$(7, 9, 8)$</td>
</tr>
<tr>
<td>(e)</td>
<td>$(2, 3, 4, 5, 7)$</td>
<td>$(8, 9)$</td>
</tr>
<tr>
<td>(f)</td>
<td>$(2, 3, 4, 5, 7, 8)$</td>
<td>$(9)$</td>
</tr>
<tr>
<td>(g)</td>
<td>$(2, 3, 4, 5, 7, 8, 9)$</td>
<td>()</td>
</tr>
</tbody>
</table>
P = heap: Heap sort

- Phase 1 time = \( \sum_{i=1}^{n} O(\log n) = O(n \log n) \)
- Phase 2 time = \( \sum_{i=n}^{1} O(\log n) = O(n \log n) \)
- Total time = \( O(n \log n) \)
## Priority queue sort: Complexity

<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heap sort</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>