Data Structures and Algorithms

Priority Queues
Outline

• Priority Queues
• Heaps
• Adaptable Priority Queues
Priority Queues
Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - `insert(k, x)` inserts an entry with key k and value x
  - `removeMin()` removes and returns the entry with smallest key
- 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
- Two distinct entries in a priority queue can have the same key

Mathematical concept of total order relation $\le$
- Reflexive property: $x \le x$
- Antisymmetric property: $x \le y \land y \le x \Rightarrow x = y$
- Transitive property: $x \le y \land y \le z \Rightarrow x \le z$
Entry ADT

• An entry in a priority queue is simply a key-value pair
• Priority queues store entries to allow for efficient insertion and removal based on keys
• Methods:
  – key(): returns the key for this entry
  – value(): returns the value associated with this entry

• As a Java interface:

```java
/**
 * Interface for a key-value pair
 *
 **/

public interface Entry {
    public Object key();
    public Object value();
}
```
Comparator ADT

• A comparator encapsulates the action of comparing two objects according to a given total order relation
• A generic priority queue uses an auxiliary comparator
• The comparator is external to the keys being compared
• When the priority queue needs to compare two keys, it uses its comparator

• The primary method of the Comparator ADT:
  – compare(x, y): Returns an integer i such that i < 0 if \( a < b \), i = 0 if \( a = b \), and i > 0 if \( a > b \); an error occurs if \( a \) and \( b \) cannot be compared.
Example Comparator

- Lexicographic comparison of 2-D points:

```java
/** Comparator for 2D points under the standard lexicographic order. */
public class Lexicographic implements Comparator {
    int xa, ya, xb, yb;
    public int compare(Object a, Object b) throws ClassCastException {
        xa = ((Point2D) a).getX();
        ya = ((Point2D) a).getY();
        xb = ((Point2D) b).getX();
        yb = ((Point2D) b).getY();
        if (xa != xb)
            return (xb - xa);
        else
            return (yb - ya);
    }
}
```

- Point objects:

```java
/** Class representing a point in the plane with integer coordinates */
public class Point2D {
    protected int xc, yc; // coordinates
    public Point2D(int x, int y) {
        xc = x;
        yc = y;
    }
    public int getX() {
        return xc;
    }
    public int getY() {
        return yc;
    }
}
```
Priority Queue Sorting

- We can use a priority 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 removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm \textit{PQ-Sort}(S, C)
\begin{itemize}
  \item[Input] sequence \( S \), comparator \( C \) for the elements of \( S \)
  \item[Output] sequence \( S \) sorted in increasing order according to \( C \)
\end{itemize}

\( P \leftarrow \) priority queue with comparator \( C \)
\begin{align*}
\text{while } & \neg S.\text{isEmpty}() \\
  e & \leftarrow S.\text{removeFirst}() \\
  P.\text{insert}(e, 0) \\
\text{while } & \neg P.\text{isEmpty}() \\
  e & \leftarrow P.\text{removeMin}().\text{key}() \\
  S.\text{insertLast}(e)
\end{align*}
Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Phạm Bảo Sơn - DSA
Selection-Sort

• Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence.
• Running time of Selection-sort:
  1. Inserting the elements into the priority queue with $n$ insert operations takes $O(n)$ time.
  2. Removing the elements in sorted order from the priority queue with $n$ removeMin operations takes time proportional to $1 + 2 + \ldots + n$.
• Selection-sort runs in $O(n^2)$ time.
## Selection-Sort Example

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<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>(g)</td>
<td>(7)</td>
<td>(7, 4, 2, 5, 3, 9)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase 2</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<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, 8, 5, 9)</td>
</tr>
<tr>
<td>(d)</td>
<td>(2, 3, 4, 5)</td>
<td>(7, 8, 9)</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>
Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence.

- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with $n$ insert operations takes time proportional to $1 + 2 + \ldots + n$
  2. Removing the elements in sorted order from the priority queue with a series of $n$ removeMin operations takes $O(n)$ time

- Insertion-sort runs in $O(n^2)$ time
**Insertion-Sort Example**

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

**Phase 1**

(a) $(4,8,2,5,3,9)$ | $(7)$
(b) $(8,2,5,3,9)$ | $(4,7)$
(c) $(2,5,3,9)$ | $(4,7,8)$
(d) $(5,3,9)$ | $(2,4,7,8)$
(e) $(3,9)$ | $(2,4,5,7,8)$
(f) $(9)$ | $(2,3,4,5,7,8)$
(g) () | $(2,3,4,5,7,8,9)$

**Phase 2**

(a) $(2)$ | $(3,4,5,7,8,9)$
(b) $(2,3)$ | $(4,5,7,8,9)$
.. | ..
.. | ..
(g) $(2,3,4,5,7,8,9)$ | ()
In-place Insertion-sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place.
- A portion of the input sequence itself serves as the priority queue.
- For in-place insertion-sort:
  - We keep sorted the initial portion of the sequence.
  - We can use swaps instead of modifying the sequence.

Phạm Bảo Sơn - DSA
Heaps
Recall Priority Queue ADT

• A priority queue stores a collection of entries
• Each entry is a pair (key, value)
• Main methods of the Priority Queue ADT
  – insert(k, x) inserts an entry with key k and value x
  – removeMin() removes and returns the entry with smallest key
• Additional methods
  – min() returns, but does not remove, an entry with smallest key
  – size(), isEmpty()
• Applications:
  – Standby flyers
  – Auctions
  – Stock market
Recall Priority Queue Sorting

• We can use a priority queue to sort a set of comparable elements
  – Insert the elements with a series of insert operations
  – Remove the elements in sorted order with a series of removeMin operations
• The running time depends on the priority queue implementation:
  – Unsorted sequence gives selection-sort: $O(n^2)$ time
  – Sorted sequence gives insertion-sort: $O(n^2)$ time
• Can we do better?

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 $\neg S.isEmpty()$
  $e \leftarrow S.remove(S.first())$
  $P.insertItem(e, e)$

while $\neg P.isEmpty()$
  $e \leftarrow P.removeMin()$
  $S.insertLast(e)$
Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every internal node $v$ other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let $h$ be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are $2^i$ nodes of depth $i$
    - at depth $h$, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost node of depth $h$
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 \( n \geq 1 + 2 + 4 + \cdots + 2^{h-1} + 1 \)
- 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 a (key, element) item at each internal node
• We keep track of the position of the last node
• For simplicity, we show only the keys in the pictures

(2, Sue)
(5, Pat)
(9, Jeff)
(7, Anna)
(6, Mark)
Insertion into a Heap

• Method insertItem 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 insertion node $z$ (the new last node)
  - Store $k$ at $z$
  - Restore the heap-order property (discussed next)
Upheap

- After the insertion of a new key $k$, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping $k$ along an upward path from the insertion node
- Upheap terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
Removal from a Heap

- Method removeMin 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 \( w \).
  - Remove \( w \).
  - Restore the heap-order property (discussed next).
Downheap

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key $k$ along a downward path from the root.
- Downheap terminates when key $k$ reaches a leaf or a node whose children have keys greater than or equal to $k$.
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time.
Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal
• Consider a priority queue with \( n \) items implemented by means of a heap
  – the space used is \( O(n) \)
  – methods insert and removeMin take \( O(\log n) \) time
  – methods size, isEmpty, and min take time \( O(1) \) time

• Using a heap-based priority queue, we can sort a sequence of \( n \) elements in \( O(n \log n) \) time

• The resulting algorithm is called heap-sort

• Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort
Vector-based Heap Implementation

- We can represent a heap with $n$ keys by means of a vector of length $n + 1$
- For the node at rank $i$
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank 1
  
Yields in-place heap-sort
Merging Two Heaps

- We are given 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 downheap to restore the heap-order property
Bottom-up Heap Construction

- We can construct a heap storing \( n \) given keys in using a bottom-up construction with \( \log n \) phases.

- In phase \( i \), pairs of heaps with \( 2^i - 1 \) keys are merged into heaps with \( 2^{i+1} - 1 \) keys.
Example
Example (contd.)
Example (contd.)
Example (end)
Analysis

• We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path).
• Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy 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.
Adaptable Priority Queues
Recall the Entry and Priority Queue ADTs

• An **entry** stores a (key, value) pair within a data structure

• Methods of the entry ADT:
  – key(): returns the key associated with this entry
  – value(): returns the value paired with the key associated with this entry

• Priority Queue ADT:
  – insert(k, x) inserts an entry with key k and value x
  – removeMin() removes and returns the entry with smallest key
  – min() returns, but does not remove, an entry with smallest key
  – size(), isEmpty()
Motivating Example

• Suppose we have an online trading system where orders to purchase and sell a given stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  – The key, p, of an order is the price
  – The value, s, for an entry is the number of shares
  – A buy order (p,s) is executed when a sell order (p’ ,s’) with price p’ ≤p is added (the execution is complete if s’ ≥s)
  – A sell order (p,s) is executed when a buy order (p’ ,s’) with price p’ ≥p is added (the execution is complete if s’ ≥s)
• What if someone wishes to cancel their order before it executes?
• What if someone wishes to update the price or number of shares for their order?
Methods of the Adaptable Priority Queue ADT

- `remove(e)`: Remove from `P` and return entry `e`.
- `replaceKey(e,k)`: Replace with `k` and return the key of entry `e` of `P`; an error condition occurs if `k` is invalid (that is, `k` cannot be compared with other keys).
- `replaceValue(e,x)`: Replace with `x` and return the value of entry `e` of `P`.
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5, A)</td>
<td>$e_1$</td>
<td>(5, A)</td>
</tr>
<tr>
<td>insert(3, B)</td>
<td>$e_2$</td>
<td>(3, B), (5, A)</td>
</tr>
<tr>
<td>insert(7, C)</td>
<td>$e_3$</td>
<td>(3, B), (5, A), (7, C)</td>
</tr>
<tr>
<td>min()</td>
<td>$e_2$</td>
<td>(3, B), (5, A), (7, C)</td>
</tr>
<tr>
<td>key($e_2$)</td>
<td>3</td>
<td>(3, B), (5, A), (7, C)</td>
</tr>
<tr>
<td>remove($e_1$)</td>
<td>$e_1$</td>
<td>(3, B), (7, C)</td>
</tr>
<tr>
<td>replaceKey($e_2$, 9)</td>
<td>3</td>
<td>(7, C), (9, B)</td>
</tr>
<tr>
<td>replaceValue($e_3$, D)</td>
<td>C</td>
<td>(7, D), (9, B)</td>
</tr>
<tr>
<td>remove($e_2$)</td>
<td>$e_2$</td>
<td>(7, D)</td>
</tr>
</tbody>
</table>
Locating Entries

• In order to implement the operations remove(k), replaceKey(e), and replaceValue(k), we need fast ways of locating an entry e in a priority queue.
• We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.
Location-Aware Entries

• A locator-aware entry identifies and tracks the location of its (key, value) object within a data structure

• Intuitive notion:
  – Coat claim check
  – Valet claim ticket
  – Reservation number

• Main idea:
  – Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

Phạm Bảo Sơn - DSA
List Implementation

• A location-aware list entry is an object storing
  – key
  – value
  – position (or rank) of the item in the list
• In turn, the position (or array cell) stores the entry
• Back pointers (or ranks) are updated during swaps
Heap Implementation

• A location-aware heap entry is an object storing
  – key
  – value
  – position of the entry in the underlying heap
• In turn, each heap position stores an entry
• Back pointers are updated during entry swaps
Using location-aware entries we can achieve the following running times (times better than those achievable without location-aware entries are highlighted in red):

<table>
<thead>
<tr>
<th>Method</th>
<th>Unsorted List</th>
<th>Sorted List</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>min</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceKey</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>replaceValue</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>