CH9.
PRIORITY QUEUES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
PRIORITY QUEUES

• Stores a collection of elements each with an associated “key” value
  • Can insert as many elements in any order
  • Only can inspect and remove a single element – the minimum (or maximum depending) element

• Applications
  • Standby Flyers
  • Auctions
  • Stock market
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
  • Entry \texttt{insert}(k, v)
    inserts an entry with key \( k \) and value \( v \)
  • Entry \texttt{removeMin()}
    removes and returns the entry with smallest key, or null if the the priority queue is empty

• Additional methods
  • Entry \texttt{min()}
    returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  • size(), isEmpty()
TOTAL ORDER RELATION

• Keys in a priority queue can be arbitrary objects on which an order is defined, e.g., integers

• Two distinct items in a priority queue can have the same key

• Mathematical concept of total order relation $\leq$

  • **Reflexive property:**
    
    $k \leq k$

  • **Antisymmetric property:**
    
    if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$

  • **Transitive property:**
    
    if $k_1 \leq k_2$ and $k_2 \leq k_3$ then $k_1 \leq k_3$
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 `getKey()`: returns the key for this entry
  • Value `getValue()`: returns the value associated with this entry
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, i.e., it is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT

  \textbf{Integer } \texttt{compare}(x, y): \texttt{return an integer } i \texttt{ such that}
  
  - $i < 0$ if $x < y$,
  - $i = 0$ if $x = y$
  - $i > 0$ if $x > y$
  - An error occurs if $a$ and $b$ cannot be compared.
PRIORITYQUEUESORT()  
SORTING WITH A PRIORITY QUEUE

- We can use a priority queue to sort a set of comparable elements
- Insert the elements one by one with a series of insert(e) operations
- Remove the elements in sorted order with a series of removeMin() operations
- Running time depends on the PQ implementation

Algorithm PriorityQueueSort()
Input: List L storing n elements and a Comparator C
Output: Sorted List L
1. Priority Queue P using comparator C
2. while ¬L.isEmpty() do
3. P.insert(L.first())
4. L.removeFirst()
5. while ¬P.isEmpty() do
6. L.insertLast(P.min())
7. P.removeMin()
8. return L
LIST-BASED PRIORITY QUEUE

Unsorted list implementation

• Store the items of the priority queue in a list, in arbitrary order

Unsorted list implementation

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

Sorted list implementation

• Store the items of the priority queue in a list, sorted by key

Sorted list implementation

• Performance:
  • `insert(e)` 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 of the list
SELECTION-SORT

• Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted list

• Running time of Selection-sort:
  • Inserting the elements into the priority queue with \( n \) insert(e) operations takes \( O(n) \) time
  • Removing the elements in sorted order from the priority queue with \( n \) removeMin() operations takes time proportional to
    \[
    \sum_{i=0}^{n} n - i = n + (n - 1) + \cdots + 2 + 1 = O(n^2)
    \]

• Selection-sort runs in \( O(n^2) \) time
Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted list (do $n$ `insert(e)` and then $n$ `removeMin()`)
**INSERTION-SORT**

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

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with $n \text{ insert}(e)$ operations takes time proportional to
    \[\sum_{i=0}^{n} i = 1 + 2 + \cdots + n = O(n^2)\]
  - Removing the elements in sorted order from the priority queue with a series of $n \text{ removeMin}()$ operations takes $O(n)$ time

- Insertion-sort runs in $O(n^2)$ time
EXERCISE
INSERTION-SORT

• Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted list (do $n$ insert(e) and then $n$ removeMin())

• Illustrate the performance of insertion-sort on the following input sequence:
  • (22, 15, 36, 44, 10, 3, 9, 13, 29, 25)
IN-PLACE INSERTION-SORT

• Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place (only $O(1)$ extra storage)

• A portion of the input list itself serves as the priority queue

• For in-place insertion-sort
  • We keep sorted the initial portion of the list
  • We can use $\text{swap}(i, j)$ instead of modifying the list

\[
\begin{array}{c}
5 & 4 & 2 & 3 & 1 \\
5 & 4 & 2 & 3 & 1 \\
4 & 5 & 2 & 3 & 1 \\
2 & 4 & 5 & 3 & 1 \\
2 & 3 & 4 & 5 & 1 \\
1 & 2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4 & 5
\end{array}
\]
HEAPS
A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:

- **Heap-Order**: for every node $v$ other than the root, $\text{key}(v) \geq \text{key}(v.\text{parent}())$
- **Complete Binary Tree**: let $h$ be the height of the heap
  - for $i = 0 \ldots h - 1$, there are $2^i$ nodes on level $i$
  - at level $h - 1$, nodes are filled from left to right

*Can be used to store a priority queue efficiently*
**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 $h$ keys
  - Since there are $2^i$ keys at level $i = 0 \ldots h - 1$ and at least one key on level $h$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = (2^h - 1) + 1 = 2^h$
  - Level $h$ has at most $2^h$ nodes: $n \leq 2^{h+1} - 1$
  - Thus, $\log(n + 1) - 1 \leq h \leq \log n \blacksquare$

<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>
EXERCISE
HEAPS

• Let $H$ be a heap with 7 distinct elements ($1, 2, 3, 4, 5, 6,$ and $7$). Is it possible that a preorder traversal visits the elements in sorted order? What about an inorder traversal or a postorder traversal? In each case, either show such a heap or prove that none exists.
INSERTION INTO A HEAP

• `insert(e)` consists of three steps
  • Find the insertion node $z$ (the new last node)
  • Store $e$ at $z$ and expand $z$ into an internal node
  • Restore the heap-order property (discussed next)
UPHEAP

• After the insertion of a new element $e$, the heap-order property may be violated

• Up-heap bubbling restores the heap-order property by swapping $e$ along an upward path from the insertion node

• Upheap terminates when $e$ reaches the root or a node whose parent has a key smaller than or equal to $\text{key}(e)$

• Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
REMOVAL FROM A HEAP

- `removeMin()` corresponds to the removal of the root from the heap.

- The removal algorithm consists of three steps:
  - Replace the root with the element of the last node $w$.
  - Compress $w$ and its children into a leaf.
  - Restore the heap-order property (discussed next).
• After replacing the root element of the last node, the heap-order property may be violated
• Down-heap bubbling restores the heap-order property by swapping element \( e \) along a downward path from the root
• Downheap terminates when \( e \) reaches a leaf or a node whose children have keys greater than or equal to \( \text{key}(e) \)
• 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
HEAP-SORT

• Consider a priority queue with $n$ items implemented by means of a heap
  • the space used is $O(n)$
  • `insert(e)` and `removeMin()` take $O(\log n)$ time
  • `min()`, `size()`, and `empty()` take $O(1)$ 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
EXERCISE
HEAP-SORT

• Heap-sort is the variation of PQ-sort where the priority queue is implemented with a heap (do \texttt{n insert(e)} and then \texttt{n removeMin()})

• Illustrate the performance of heap-sort on the following input sequence (draw the heap at each step):
  • \((22, 15, 36, 44, 10, 3, 9, 13, 29, 25)\)
ARRAY-BASED HEAP IMPLEMENTATION

• We can represent a heap with \( n \) elements by means of a vector of length \( n \)
  • Links between nodes are not explicitly stored
  • The leaves are not represented
  • The cell at index 0 is the root

• For the node at index \( i \)
  • the left child is at index \( 2i + 1 \)
  • the right child is at index \( 2i + 2 \)

• \texttt{insert(e)} corresponds to inserting at index \( n + 1 \)

• \texttt{removeMin()} corresponds to removing element at index \( n \)

• Yields in-place heap-sort
<table>
<thead>
<tr>
<th></th>
<th>Insert (e)</th>
<th>RemoveMin()</th>
<th>PQ-Sort total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordered List</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>(Insertion Sort)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unordered List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>(Selection Sort)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Heap,</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Vector-based Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Heap Sort)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MERGING TWO HEAPS

• We are given two heaps and a new element \( e \).
• We create a new heap with a root node storing \( e \) and with the two heaps as subtrees.
• We perform downheap to restore the heap-order property.
We can construct a heap storing $n$ given elements in using a bottom-up construction with $\log n$ phases.

In phase $i$, pairs of heaps with $2^i - 1$ elements are merged into heaps with $2^{i+1} - 1$ elements.
EXAMPLE
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

• One weakness of the priority queues so far is that we do not have an ability to update individual entries, like in a changing price market or bidding service.

• Recall that \texttt{insert(e)} returns an entry. We need to save these values to be able to adapt them.

• Additional ADT support (also includes standard priority queue functionality):
  • Entry \texttt{remove(e)} --- remove a specific entry \(e\)
  • Key \texttt{replaceKey(e, k)} --- replace the key of entry \(e\) with \(k\), and return the old key.
  • Value \texttt{replaceValue(e, k)} --- replace the value of entry \(e\) with \(k\), and return the old value.
**LOCATION-AWARE ENTRY**

- **Locators** decouple positions and entries in order to support efficient adaptable priority queue implementations (i.e., in a heap)
- Each position has an associated locator
- Each locator stores a pointer to its position and memory for the entry
POSIIONS VS. LOCATORS

• Position
  • represents a “place” in a data structure
  • related to other positions in the data structure (e.g., previous/next or parent/child)
  • often implemented as a pointer to a node or the index of an array cell
• Position-based ADTs (e.g., sequence and tree) are fundamental data storage schemes

• Locator
  • identifies and tracks a (key, element) item
  • unrelated to other locators in the data structure
  • often implemented as an object storing the item and its position in the underlying structure
• Key-based ADTs (e.g., priority queue) can be augmented with locator-based methods