Chapter 4
Priority Queues & Heapsort
CS3330 Algorithms

Priority Queue ADT
- A priority queue stores a collection of elements.
- 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.

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

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
    - removeMax takes \(O(n)\) time since we have to traverse the entire sequence to find the smallest key

Selection-Sort, Insertion-Sort
- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence.
- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence.

What is a heap?
- A (min) heap is a binary tree storing keys at its internal 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
    - \(h = 0\) to \(h\), there are \(2^h\) nodes of depth \(i\)
    - at depth \(k\), the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal 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 \( n \geq 1 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1 \)
- Thus, \( n \geq 2^h - 1 \), i.e., \( h \leq \log n \).

Heaps and Priority Queues

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

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 \( z \) and expand \( z \) into an internal node.
  - Restore the heap-order property by Sift-up (discussed next).

Sift-Up

- After the insertion of a new key \( k \), the heap-order property may be violated.
- Algorithm Sift-up restores the heap-order property by swapping \( k \) along an upward path from the insertion node.
- Sift-up 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 \( O(\log n) \), Sift-up 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 \( w \).
  - Compress \( w \) and its children into a leaf.
  - Restore the heap-order property by Heapify or Sift-down (discussed next).

Downheap

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

Consider a priority queue with $n$ items implemented by means of a max-heap:
- the 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.
- 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();
- Cost: $O(n \log n)$.

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 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.

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. $l \leftarrow \text{Left}(i)$; \hspace{1em} // Left($i$) = $2i+1$
  2. $r \leftarrow \text{Right}(i)$; \hspace{1em} // Right($i$) = $2i+2$
  3. $\max \leftarrow i$;
  4. if ($l < n$ \&\& $A[l] > A[\max]$) $\max \leftarrow l$;
  5. if ($r < n$ \&\& $A[r] > A[\max]$) $\max \leftarrow r$;
  6. if ($\max \neq i$) {
      7. exchange $A[i] \leftrightarrow A[\max]$;
      8. MaxHeapify($A$, $\max$, $n$);
    9. }
}

HeapSort($A$)

**Alg:** HeapSort($A$) {
  1. $n \leftarrow \text{length}$;
  2. for $i \leftarrow \lfloor n/2 \rfloor$ downto 0 {
    3. MaxHeapify($A$, $i$, $n$);
    4. for $i \leftarrow n-1$ downto 1 { \hspace{1em} // $A[0 \ldots i]$ is a max heap
      5. exchange $A[i] \leftrightarrow A[0]$;
      6. MaxHeapify($A$, 0, $i$); \hspace{1em} // $A[i \ldots n-1]$ is sorted with max $(n-i)$
    7. }
  }
}

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 constant with the input size.

Stability

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

Sorted on first key:

<table>
<thead>
<tr>
<th>Name</th>
<th>First Key</th>
<th>Second Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12</td>
<td>120</td>
</tr>
<tr>
<td>Brown</td>
<td>24</td>
<td>240</td>
</tr>
<tr>
<td>White</td>
<td>36</td>
<td>360</td>
</tr>
<tr>
<td>Davis</td>
<td>48</td>
<td>480</td>
</tr>
<tr>
<td>Johnson</td>
<td>60</td>
<td>600</td>
</tr>
</tbody>
</table>

Sort file on second key:

<table>
<thead>
<tr>
<th>Name</th>
<th>First Key</th>
<th>Second Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>36</td>
<td>360</td>
</tr>
<tr>
<td>Davis</td>
<td>48</td>
<td>480</td>
</tr>
<tr>
<td>Brown</td>
<td>24</td>
<td>240</td>
</tr>
<tr>
<td>Smith</td>
<td>12</td>
<td>120</td>
</tr>
<tr>
<td>Johnson</td>
<td>60</td>
<td>600</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>First Key</th>
<th>Second Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12</td>
<td>120</td>
</tr>
<tr>
<td>Brown</td>
<td>24</td>
<td>240</td>
</tr>
<tr>
<td>White</td>
<td>36</td>
<td>360</td>
</tr>
<tr>
<td>Davis</td>
<td>48</td>
<td>480</td>
</tr>
<tr>
<td>Johnson</td>
<td>60</td>
<td>600</td>
</tr>
</tbody>
</table>
## 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.