Sorting and Searching
Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - `insert(e)`
    - inserts an entry e
  - `removeMin()`
    - removes an entry with smallest key (the one that `min` would return).

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

- Applications:
  - Standby flyers
  - Auctions
  - Stock market
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 PQ-Sort(S)**

**Input** sequence $S$

**Output** sequence $S$ sorted in increasing order

$P \leftarrow$ priority queue

while $\neg S$.empty ()
  
  $e \leftarrow S$.front ()
  
  $S$.eraseFront ()
  
  $P$.insert ($e$)

while $\neg P$.empty ()
  
  $e \leftarrow P$.min ()
  
  $P$.removeMin ()
  
  $S$.insertBack ($e$)
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
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 \( n \) \text{insert} operations takes \( O(n) \) time.
2. Removing the elements in sorted order from the priority queue with \( n \) \text{removeMin} operations takes time proportional to
   \[
   n + (n - 1) + \ldots + 2 + 1
   \]

Selection-sort runs in \( O(n^2) \) time.
### Selection-Sort Example

**Input:**

Sequence S: (7,4,8,2,5,3,9)

Priority Queue P: ()

#### Phase 1

(a) (4,8,2,5,3,9) (7)
(b) (8,2,5,3,9) (7,4)

.. .. ..

(g) () ()

#### Phase 2

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

**Input:** (7,4,8,2,5,3,9)  
**Priority queue P:** ()

### Phase 1

<table>
<thead>
<tr>
<th></th>
<th><strong>Sequence S</strong></th>
<th><strong>Priority queue P</strong></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>(4,7)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,5,3,9)</td>
<td>(4,7,8)</td>
</tr>
<tr>
<td>(d)</td>
<td>(5,3,9)</td>
<td>(2,4,7,8)</td>
</tr>
<tr>
<td>(e)</td>
<td>(3,9)</td>
<td>(2,4,5,7,8)</td>
</tr>
<tr>
<td>(f)</td>
<td>(9)</td>
<td>(2,3,4,5,7,8)</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(2,3,4,5,7,8,9)</td>
</tr>
</tbody>
</table>

### Phase 2

<table>
<thead>
<tr>
<th></th>
<th><strong>Sequence S</strong></th>
<th><strong>Priority queue P</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(3,4,5,7,8,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(4,5,7,8,9)</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
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 to modify the sequence.
Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - **Heap-Order Property**: for every internal node \( v \) other than the root, \( key(v) \geq key(parent(v)) \) (this is a min-heap; there are also max-heaps.).
  - **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 leaves are as far to the left as possible

- The last node of a heap is the rightmost node of maximum depth.
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 going upwards.
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 going downwards.
Heap-Sort

- 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`, `empty`, 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
Merge-Sort

- Merge-sort on an input sequence \( S \) with \( n \) elements consists of three steps:
  - **Divide**: partition \( S \) into two sequences \( S_1 \) and \( S_2 \) of about \( n/2 \) elements each
  - **Recur**: recursively sort \( S_1 \) and \( S_2 \)
  - **Conquer**: merge \( S_1 \) and \( S_2 \) into a unique sorted sequence

Algorithm \( \text{mergeSort}(S, C) \)

**Input** sequence \( S \) with \( n \) elements

**Output** sequence \( S \) sorted

\[
\text{if } S.\text{size}() > 1 \\
\quad (S_1, S_2) \leftarrow \text{partition}(S, n/2) \\
\quad \text{mergeSort}(S_1) \\
\quad \text{mergeSort}(S_2) \\
\quad S \leftarrow \text{merge}(S_1, S_2)
\]
Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$.

- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.

Algorithm $\text{merge}(A, B)$

**Input** sequences $A$ and $B$ with $n/2$ elements each

**Output** sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.\text{empty()} \land \neg B.\text{empty()}$

if $A.\text{front()} < B.\text{front()}$

\[ S.\text{addBack}(A.\text{front()}); A.\text{eraseFront();} \]

else

\[ S.\text{addBack}(B.\text{front()}); B.\text{eraseFront();} \]

while $\neg A.\text{empty()}$

\[ S.\text{addBack}(A.\text{front()}); A.\text{eraseFront();} \]

while $\neg B.\text{empty()}$

\[ S.\text{addBack}(B.\text{front()}); B.\text{eraseFront();} \]

return $S$
Merge-Sort Analysis

- Merge-sort on an input sequence $S$ with $n$ elements consists of:
  - **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - **Recur**: recursively sort $S_1$ and $S_2$
  - **Conquer**: merge $S_1$ and $S_2$ into a unique sorted sequence

$T(n) : \text{time for Merge-sort on a sequence of } n \text{ elements}$

- $O(n)$
- $2T(n/2)$
- $O(n)$

$T(n) = 2T(n/2) + O(n)$
$T(n) \in O(n \log n)$
Quick-Sort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide:** pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal \( x \)
  - \( G \) elements greater than \( x \)
- **Recur:** sort \( L \) and \( G \)
- **Conquer:** join \( L \), \( E \) and \( G \)
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$. 

<table>
<thead>
<tr>
<th>depth</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n - 1$</td>
<td>1</td>
</tr>
</tbody>
</table>
Expected Running Time

Consider a recursive call of quick-sort on a sequence of size $s$

- **Good call**: the sizes of $L$ and $G$ are each less than $3s/4$
- **Bad call**: one of $L$ and $G$ has size greater than $3s/4$

A call is **good** with probability $1/2$

- $1/2$ of the possible pivots cause good calls:
Expected Running Time, Part 2

- **Probabilistic Fact:** The expected number of coin tosses required in order to get \( k \) heads is \( 2^k \)
- For a node of depth \( i \), we expect
  - \( i/2 \) ancestors are good calls
  - The size of the input sequence for the current call is at most \((3/4)^{i/2} n\)

Therefore, we have
- For a node of depth \( 2\log_{4/3} n \), the expected input size is one
- The expected height of the quick-sort tree is \( O(\log n) \)
- The amount or work done at the nodes of the same depth is \( O(n) \)
- Thus, the expected running time of quick-sort is \( O(n \log n) \)

![Diagram showing expected height and time per level]

Total expected time: \( O(n \log n) \)
Sorting Decision Tree Height

- The height of the decision tree is a lower bound on the running time.
- Each leaf specifies how to “unscramble” an input permutation.
- Every input permutation must lead to a separate leaf.
- There are \( n! = 1 \cdot 2 \cdot \ldots \cdot n \) leaves.
- So the height is at least \( \log(n!) \).
The Lower Bound

Any comparison-based sorting algorithm takes at least 
\( \log (n!) \) time.

Therefore, any such algorithm takes time at least

\[
\log (n!) \geq \log \left( \frac{n^2}{2} \right) = (n/2) \log (n/2).
\]

That is, any comparison-based sorting algorithm must run in \( \Omega(n \log n) \) time.
The Lower Bound

The preceding argument uses the fact that

\[ n! \geq \left( \frac{n}{2} \right)^\frac{n}{2} \]

which we can easily see by writing out what \( n! \) means:

\[ n! = n \cdot (n - 1) \cdot \ldots \cdot \left( \frac{n}{2} + 1 \right) \cdot \frac{n}{2} \cdot \left( \frac{n}{2} - 1 \right) \cdot \ldots \cdot 2 \cdot 1 \]

\[ \geq n \cdot (n - 1) \cdot \ldots \cdot \left( \frac{n}{2} + 1 \right) \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}}_{\frac{n}{2} \text{ factors}} = \left( \frac{n}{2} \right)^\frac{n}{2} \]
Bucket-Sort

- Let be $S$ be a sequence of $n$ (key, element) entries with keys in the range $[0, N - 1]$

- Bucket-sort uses the keys as indices into an auxiliary array $B$ of sequences (buckets)
  - **Phase 1**: Empty sequence $S$ by moving each entry $(k, o)$ into its bucket $B[k]$
  - **Phase 2**: For $i = 0, ..., N - 1$, move the entries of bucket $B[i]$ to the end of sequence $S$

- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time
  - Bucket-sort takes $O(n + N)$ time

---

**Algorithm** `bucketSort(S, N)`

- **Input** sequence $S$ of (key, element) items with keys in the range $[0, N - 1]$
- **Output** sequence $S$ sorted by increasing keys

```
B ← array of $N$ empty sequences
while ¬S.empty()
    $(k, o) ← S.front()$
    S.eraseFront()
    $B[k].insertBack((k, o))$
for $i ← 0$ to $N - 1$
    while ¬B[i].empty()
        $(k, o) ← B[i].front()$
        $B[i].eraseFront()$
        $S.insertBack((k, o))$
```
Radix-Sort

- Radix-sort uses bucket-sort as the stable sorting algorithm in each dimension (digit)
- Radix-sort is applicable to tuples where the keys in each dimension \( i \) are integers in the range \([0, N - 1]\)
- Radix-sort runs in time \( O(d(n + N)) \)

Algorithm `radixSort(S, N)`

**Input** sequence \( S \) of \( d \)-tuples such that \((0, \ldots, 0) \leq (x_1, \ldots, x_d)\) and \((x_1, \ldots, x_d) \leq (N - 1, \ldots, N - 1)\) for each tuple \((x_1, \ldots, x_d)\) in \( S \)

**Output** sequence \( S \) sorted in lexicographic order

**for** \( i \leftarrow d \) **downto** 1

`bucketSort(S, N)`
Binary Search

- Binary search performs operation `find(k)` on a dictionary or sorted sequence.
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example: `find(7)`
Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$
- External nodes do not store items

- An inorder traversal of a binary search tree visits the keys in increasing order
Search

- To search for a key $k$, we trace a downward path starting at the root.
- The next node visited depends on the comparison of $k$ with the key of the current node.
- If we reach a leaf, the key is not found.
- Example: get(4):
  - Call TreeSearch(4, root)

Algorithm $\text{TreeSearch}(k, v)$

```java
if v.isExternal ()
    return v
if k < v.key()
    return $\text{TreeSearch}(k, v.left())$
else if k = v.key()
    return v
else { k > v.key() }
    return $\text{TreeSearch}(k, v.right())$
```

© 2010 Goodrich, Tamassia