CHAPTER 11
SETS, AND SELECTION

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO AND JORY DENNY
SETS
A set is an ordered data structure similar to an ordered map, except only elements are stored (and yes elements must be unique)

We represent a set by the sorted sequence of its elements

By specializing the auxiliary methods the generic merge algorithm can be used to perform basic set operations:

- **Union** - $A \cup B$ – Return all elements which appear in $A$ or $B$ (unique only)
- **Intersection** - $A \cap B$ – Return only elements which appear in both $A$ and $B$
- **Subtraction** - $A \setminus B$ – Return elements in $A$ which are not in $B$

The running time of an operation on sets $A$ and $B$ should be at most $O(n_A + n_B)$

**Set union:**
- if $a < b$ \( S.\text{insertFront}(a) \)
- if $b < a$ \( S.\text{insertFront}(b) \)
- else $a = b$ \( S.\text{insertFront}(a) \)

**Set intersection:**
- if $a < b$ \( \text{do nothing} \)
- if $b < a$ \( \text{do nothing} \)
- else $a = b$ \( S.\text{insertBack}(a) \)
GENERIC MERGING

- Generalized merge of two sorted sets $A$ and $B$
- Auxiliary methods (generic functions)
  - aIsLess($a, S$)
  - bIsLess($b, S$)
  - bothAreEqual($a, b, S$)
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

**Algorithm** genericMerge($A, B$)

**Input:** Sets $A, B$ (implemented as sequences)

**Output:** Set $S$

1. $S \leftarrow \emptyset$
2. while $\neg A$.empty() $\land \neg B$.empty() do
3.   $a \leftarrow A$.front(); $b \leftarrow B$.front()
4.   if $a < b$
5.     aIsLess($a, S$) //generic action
6.     $A$.eraseFront();
7.   else if $b < a$
8.     bIsLess($b, S$) //generic action
9.     $B$.eraseFront();
10. else //a = b
11.     bothAreEqual($a, b, S$) //generic action
12.     $A$.eraseFront(); $B$.eraseFront();
13. while $\neg A$.empty() do
14.   aIsLess($A$.front(), $S$); $A$.eraseFront();
15. while $\neg B$.empty() do
16.   bIsLess($B$.front(), $S$); $B$.eraseFront();
17. return $S$
Any of the set operations can be implemented using a generic merge.

For example:
- For intersection: only copy elements that are duplicated in both list.
- For union: copy every element from both lists except for the duplicates.
- All methods run in linear time.
Can use search trees such that the key is equivalent to the element to implement a set, allows fast ordering of data.
THE SELECTION PROBLEM

- Given an integer $k$ and $n$ elements $\{x_1, x_2, \ldots, x_n\}$, taken from a total order, find the $k$-th smallest element in this set.
  - Also called order statistics, the $i$th order statistic is the $i$th smallest element
  - Minimum - $k = 1$ - 1st order statistic
  - Maximum - $k = n$ - $n$th order statistic
  - Median - $k = \left\lfloor \frac{n}{2} \right\rfloor$
  - etc
The Selection Problem

- Naïve solution - SORT!
- We can sort the set in $O(n \log n)$ time and then index the $k$-th element. 

\[
\begin{array}{cccccc}
7 & 4 & 9 & 6 & 2 & \rightarrow \ 2 & 4 & 6 & 7 & 9
\end{array}
\]

- Can we solve the selection problem faster?
THE MINIMUM (OR MAXIMUM)

**Algorithm** minimum($A$)

**Input:** Array $A$

**Output:** minimum element in $A$

1. $m \leftarrow A[1]$
2. for $i \leftarrow 2 \ldots n$ do
3. $m \leftarrow \min(m, A[i])$
4. return $m$

• Running Time
  • $O(n)$

• Is this the best possible?
Quick-select is a randomized selection algorithm based on the \textit{prune-and-search} paradigm:

- **Prune**: pick a random element \(x\) (called pivot) and partition \(S\) into
  - \(L\) elements \(< x\)
  - \(E\) elements \(= x\)
  - \(G\) elements \(> x\)

- **Search**: depending on \(k\), either answer is in \(E\), or we need to recur on either \(L\) or \(G\)

- **Note**: Partition same as Quicksort
An execution of quick-select can be visualized by a recursion path.

- Each node represents a recursive call of quick-select, and stores \( k \) and the remaining sequence.

\[
\begin{align*}
k &= 5, \ S = (7, 4, 9, 3, 2, 6, 5, 1, 8) \\
k &= 2, \ S = (7, 4, 9, 6, 5, 8) \\
k &= 2, \ S = (7, 4, 6, 5) \\
k &= 1, \ S = (7, 6, 5) \\
&5
\end{align*}
\]
EXERCISE

- Best Case - even splits (n/2 and n/2)
- Worst Case - bad splits (1 and n-1)

Derive and solve the recurrence relation corresponding to the best case performance of randomized quick-select.

Derive and solve the recurrence relation corresponding to the worst case performance of randomized quick-select.
Consider a recursive call of quick-select on a sequence of size $s$:

- **Good call**: the size of $L$ and $G$ is at most $\frac{3s}{4}$.
- **Bad call**: the size of $L$ and $G$ is greater than $\frac{3s}{4}$.

A call is good with probability $\frac{1}{2}$.

1/2 of the possible pivots cause good calls:

- **Bad pivots**: 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Good pivots**: 10, 11, 12, 13, 14, 15, 16
- **Bad pivots**: 1, 2, 3, 4, 5, 6, 7, 8, 9
Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two.

Probabilistic Fact #2: Expectation is a linear function:
- $E(X + Y) = E(X) + E(Y)$
- $E(cX) = cE(X)$

Let $T(n)$ denote the expected running time of quick-select.

By Fact #2, $T(n) < T\left(\frac{3n}{4}\right) + bn \ast (expected \# \ of \ calls \ before \ a \ good \ call)$

By Fact #1, $T(n) < T\left(\frac{3n}{4}\right) + 2bn$

That is, $T(n)$ is a geometric series: $T(n) < 2bn + 2b\left(\frac{3}{4}\right)n + 2b\left(\frac{3}{4}\right)^2n + 2b\left(\frac{3}{4}\right)^3n + \cdots$

So $T(n)$ is $O(n)$.

We can solve the selection problem in $O(n)$ expected time.
We can do selection in $O(n)$ worst-case time.

Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:

- Divide $S$ into $\frac{n}{5}$ sets of 5 each
- Find a median in each set
- Recursively find the median of the “baby” medians.

See Exercise C-11.22 for details of analysis.