ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM JORY DENNY AND MUKULIKA GHOSH

CH 9: MAPS AND DICTIONARIES
OUTLINE

- Map ADT (Ch. 9.1)
- Dictionary ADT (Ch. 9.5)
- Ordered Map ADT (Ch. 9.3)
A map models a searchable collection of key-value pair (called entries)

- Multiple items with the same key are not allowed
- Often called “associative” containers

Applications:
- address book or student records
- mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)
MAP ADT

- Map ADT methods:
  - $\text{find}(k)$ – if $M$ has an entry $e = (k, v)$, return an iterator $p$ referring to this entry, else, return special end iterator.
  - $\text{put}(k, v)$ – if $M$ has no entry with key $k$, then add entry $(k, v)$ to $M$, otherwise replace the value of the entry with $v$; return iterator to the inserted/modified entry
  - $\text{erase}(k), \text{erase}(p)$ – remove from $M$ entry with key $k$ or iterator $p$; An error occurs if there is no such element.
  - $\text{size()}, \text{empty()}, \text{begin()}, \text{end}()$
We can imagine implementing the map with an unordered list

- `find(k)` – search the list of entries for key `k`
- `put(k, v)` – search the list for an existing entry, otherwise call `insertBack((k, v))`
- Similar idea for erase functions
- Complexities?
  - $O(n)$ on all

```
header <-> ↓ <-> ↓ <-> ↓ <-> ↓ <-> ↓ <-> ↓ <-> ↓ <-> ↓ <-> ↓<-> tail

9 | c | 6 | c | 5 | c | 8 | c
```
A direct address table is a map in which
- The keys are in the range \([0, N]\)
- Stored in an array \(T\) of size \(N\)
- Entry with key \(k\) stored in \(T[k]\)

Performance:
- \(\text{put}(k, v), \text{find}(k), \text{and erase}(k)\) all take \(O(1)\) time
- Space - requires space \(O(N)\), independent of \(n\), the number of entries stored in the map
- The direct address table is not space efficient unless the range of the keys is close to the number of elements to be stored in the map, i.e., unless \(n\) is close to \(N\).
The dictionary ADT models a searchable collection of key-value entries.

The main difference from a map is that multiple items with the same key are allowed.

Any data structure that supports a dictionary also supports a map.

Applications:
- Dictionary which has multiple definitions for the same word.
Reminder - Map ADT methods:

- **find(k)** – if $M$ has an entry $e = (k, v)$, return an iterator $p$ referring to this entry, else, return special end iterator.
- **put(k,v)** – if $M$ has no entry with key $k$, then add entry $(k, v)$ to $M$, otherwise replace the value of the entry with $v$; return iterator to the inserted/modified entry.
- **erase(k), erase(p)** – remove from $M$ entry with key $k$ or iterator $p$; An error occurs if there is no such element.
- **size(), empty(), begin(), end()**

The Dictionary ADT adds/modifies the Map ADT to support multiple entries with the same key. Describe how the operations of finding, inserting, and removing elements should be modified from the Map ADT to support the Dictionary ADT.

- Finding elements
- Inserting elements
- Removing elements
Reminder - Map ADT methods:

- **find**(k) — if M has an entry e = (k, v), return an iterator p referring to this entry, else, return special end iterator.
- **put**(k, v) — if M has no entry with key k, then add entry (k, v) to M, otherwise replace the value of the entry with v; return iterator to the inserted/modified entry.
- **erase**(k), **erase**(p) — remove from M entry with key k or iterator p; An error occurs if there is no such element.
- size(), empty(), begin(), end()

Dictionary ADT adds/modifies the following to the Map ADT:

- **find**(k) — Return an iterator p referring to an arbitrary entry with this key, else, return special end iterator.
- **findAll**(k) — Return iterators (b, e) s.t. that all entries with key k are between them, not including e.
- **insert**(k, v) — Insert an entry with key k and value v, returning an iterator to the newly created entry.
  - Note: This replaces “put(k, v)”
- **erase**(k) — remove an arbitrary entry with key k; An error occurs if there is no such element.
- **erase**(p) — remove entry with iterator p; An error occurs if there is no such element.
- size(), empty(), begin(), end()
An Ordered Map/Dictionary supports the usual map/dictionary operations, but also maintains an order relation for the keys.

Naturally supports

- **Ordered search tables** - store dictionary in a vector by non-decreasing order of the keys
- Utilizes binary search

Ordered Map/Dictionary ADT adds the following functionality to a map/dictionary

- **firstEntry(), lastEntry()** – return iterators to entries with the smallest and largest keys, respectively
- **ceilingEntry(k), floorEntry(k)** – return an iterator to the least/greatest key value greater than/less than or equal to k
- **lowerEntry(k), higherEntry(k)** – return an iterator to the greatest/least key value less than/greater than k
Binary search performs operation \( \text{find}(k) \) on an ordered search table
- similar to the high-low game
- at each step, the number of candidate items is halved
- terminates after a logarithmic number of steps

Example: \( \text{find}(7) \)
## MAP/DICTIONARY IMPLEMENTATIONS

<table>
<thead>
<tr>
<th></th>
<th>put((k, v))</th>
<th>find((k))</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Address Table (map only)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ordered Search Table (ordered map/dictionary)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>