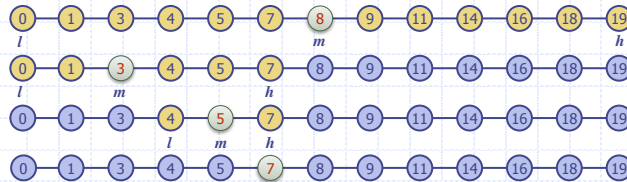


Dictionaries



Dictionary ADT

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
 - word-definition pairs
 - credit card authorizations
 - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)
- Dictionary ADT methods:
 - **get(k)**: if the dictionary has an entry with key k, returns it, else, returns null
 - **getAll(k)**: returns an iterable collection of all entries with key k
 - **put(k, o)**: inserts and returns the entry (k, o)
 - **remove(e)**: remove the entry e from the dictionary
 - **entrySet()**: returns an iterable collection of the entries in the dictionary
 - **size()**, **isEmpty()**

Example

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	null	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
remove(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(5)	null	(7,B),(2,C),(8,D),(2,E)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
 - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
 - **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - **get** and **remove** take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

The getAll and put Algorithms

Algorithm `getAll(k)`

Create an initially-empty list `L`

for `e: D` **do**

if `e.getKey() = k` **then**

`L.addLast(e)`

return `L`

Algorithm `put(k,v)`

Create a new entry `e = (k,v)`

`S.addLast(e)` {`S` is unordered}

return `e`

The remove Algorithm

Algorithm `remove(e)`:

{ We don't assume here that `e` stores its position in `S` }

`B = S.positions()`

while `B.hasNext()` **do**

`p = B.next()`

if `p.element() = e` **then**

`S.remove(p)`

return `e`

return `null` {there is no entry `e` in `D`}

Hash Table Implementation

- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Search Table

- A search table is a dictionary implemented by means of a sorted array
 - We store the items of the dictionary in an array-based sequence, sorted by key
 - We use an external comparator for the keys
- Performance:
 - `get` takes $O(\log n)$ time, using binary search
 - `put` takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
 - `remove` takes $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search

- Binary search performs operation `get(k)` on a dictionary implemented by means of an array-based sequence, sorted by key
 - similar to the high-low game
 - at each step, the number of candidate items is halved
 - terminates after a logarithmic number of steps
- Example: `get(7)`

