Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE

Map ADT : using lists

Abstract Data Type: Maps

- A map models a collection of key-value entries that is searchable `by the key'
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
  - (may be allowed in a 'multi-map')

Common Applications of Maps

- Address book:
  - key=name, value=address
- Student-record database
  - key=student-ID, value=name, marks, etc.
- Dictionaries:
  - key=word, value=‘meaning’
- Symbol table:
  - key=symbol, value=‘something internally useful’
  - standard component of compilers, and any program that has to read data from a file

Also known as “associative arrays” (perl, php) because they act like an array in which the index need not be an integer. Maps are very useful in many coding tasks

The Map ADT over pairs <K,V>

Map ADT methods:

- V get(K k): if the map M has an entry with key k, return its associated value; else, return null
- V put(K k, V v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- V remove(K k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- int size(), boolean isEmpty()
- {K} keys(): return an iterable collection of the keys in M
- {V} values(): return an iterable collection of the values in M
- {<K,V>} entries(): return an iterable collection of the entries in M

VERY IMPORTANT

Be careful not to confuse different usages of binary trees:

- **Priority Queue &**
  "Binary Tree implementations of heaps" (in other lectures)

with

- **MAP &**
  "Binary Search Trees" (in later lecture)

Both insert entries based on a key.
The crucial difference is the way entries are accessed:

- MAP: access **any** key
- PQ: access only the **minimum** key

A Simple List-Based Map

- It is straightforward to implement a map using a list – either singly or doubly-linked.
- We also store separate size counter, n, so that "getSize()" is O(1), and maintain it on adding or removing elements

The get(k) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value
3. Else return null (or false)

Worst case: O(n)

The put(k,v) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value, and overwrite with the new value v
3. Else insert <k,v> as a node at the end of the list

Note: we cannot just put the new entry at the start of the list, because we need to check that the key k is not already stored !

Worst case: O(n)

The remove(k) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value and remove the node.
3. Else return null/false

Worst case: O(n)

List-Based Map

- Overall: the operations are O(n) because of needing to traverse the list.
- The unsorted list implementation is suitable only for maps of small size
- We could use a sorted list, but still O(n)
  - It does not help as we need to be able to access any k, not just the minimum

Remark: Prototyping ADTs

- List version of a Map is
  "simple but inefficient"
- Quite common that ADTs can be implemented very easily using lists & arrays.
  - Easier to be correct
  - Even though slow, they can be useful
    - allow the rest of the project to proceed without waiting for the efficient version
    - can be used for a regression test of a "faster but trickier" later version
    - e.g. use in a "shadow mode" where the "slow-obviously-correct" version is used together with the "fast-possibly-buggy" version and the results checked against each other.