CS 61B Spring 2017

Hashing

Discussion 9: March 15, 2017

1 Data Structures

Give a tight asymptotic bound for each of the following problems. Provide your bound in $\Theta(\cdot)$ if it exists, otherwise provide both the $O(\cdot)$ and $\Omega(\cdot)$ bound.

Note that answers may differ depending on the assumptions made.

1.1 Insertion of one element into each of the following data structures where N is the number of elements already in the collection.

(a) `ArrayList`

$\Omega(1)$ if there is enough space, $O(N)$ if a resize is needed.

(b) `LinkedList`

$\Theta(1)$.

(c) `BSTMap` (binary search tree)

$\Omega(1)$, $O(N)$ if the tree is spindly.

(d) `TreeSet` (balanced search tree)

$\Theta(\log N)$ since the tree is always bushy.

(e) `HashSet`

$\Omega(1)$ in the ideal case, $O(N)$ if there are many collisions since we need to scan for possible duplicates.

Note that if the keys implement `Comparable`, the upper bound is reduced to $O(\log N)$ since Java's implementation converts long external chains to balanced search trees.

1.2 Containment check (`contains`) of one element in each of the following data structures where N is the number of elements already in the collection.

(a) `ArrayList`

$\Omega(1)$ if the element is found near the beginning, $O(N)$ if the element is not in the list at all.

(b) `LinkedList`

$\Omega(1)$, $O(N)$.

(c) `BSTMap` (binary search tree)

$\Omega(1)$ if the element is found near the root, $O(N)$ if the tree is spindly.

(d) `TreeSet` (balanced search tree)

$\Omega(1)$ if the element is found near the root, $O(\log N)$ otherwise.

(e) `HashSet`

$\Omega(1)$ in the ideal case, $O(N)$ if there are many collisions.

1.3 Suppose we're designing a hash table. Compare and contrast each of the following external chaining implementations. Why would you use one over the other?

Remember that the external chaining mechanism has a number of important constraints. First, in the ideal scenario, the size of the chain is some constant factor proportional to the load factor. Second, we need to traverse the chain on every insertion to see if there already exists an entry with the same key. These facts motivate our answers.

(a) Linked list

Since the size of the list is ideally some constant factor, traversing and adding elements to the linked list is relatively inexpensive and the same performance for all insertions.

(b) Resizing array

Array insertion can be expensive in the case where we need to resize and copy over each element in the array. Arrays have fast random access but the external chain can't take advantage of that fact.

(c) Balanced search tree

An additional constraint is that elements need to be `Comparable`. There is greater overhead in terms of memory cost than linked lists since there are more pointers to keep track. But we can limit the time spent searching through the external chain to $O(\log N)$ because the tree is balanced.

(d) Hash table

Without a new hash function, elements already destined for this external chain may be more likely to collide again. But if we have a second hash function, this strategy can be useful. See cuckoo hashing.

2 Hash Codes

There is a problem with each `hashCode()` method below (correctness, distribution, efficiency). Assume there are no problems with the correctness of `equals()`.

2.1
```
class PokeTime {
    int startTime;
    int duration;
    public int getCurrentTime() {
        // Gets the current system clock time
    }
    public int hashCode() {
        return 1021 * (startTime + 1021 * duration + getCurrentTime());
    }
    public boolean equals(Object o) {
        PokeTime p = (PokeTime) o;
        return p.startTime == startTime && p.duration == duration;
    }
}
```

Incorrect: The `hashCode()` is non-deterministic.

2.2
```
class Phonebook {
    List<Human> humans;
    public int hashCode() {
        int h = 0;
        for (Human human : humans) {
            // Assume Human::hashcode is correct
            h = (h + human.hashCode()) % 509;
        }
        return h;
    }
    public boolean equals(Object o) {
        Phonebook p = (Phonebook) o;
        return p.humans.equals(humans);
    }
}
```

Poor distribution: The `hashCode()` only distributes numbers between -508 and 508, which is an inefficient use of the full integer range and will cause more collisions than necessary.

2.3
```
class Person {
    Long id;
    String name;
    Integer age;
    public int hashCode() {
        return id.hashCode() + name.hashCode() + age.hashCode();
    }
    public boolean equals(Object o) {
        Person p = (Person) o;
        return p.id == id;
```

```
    }
}
```

Incorrect: Persons that are `equals()` do not necessarily have the same `hashCode()`.