CS 61B Data Structures and Programming Methodology

July 17, 2008

David Sun
Deletion

Delete a node given a key, if a node exists.

1. Find a node with key $k$ using the same algorithm as `find()`.
2. Return `null` if $k$ is not in the tree;
3. Otherwise, let $n$ be the first node with key $k$. If $n$ has no children, detach it from its parent and throw it away.
Deletion

4. If \( n \) has one child, move \( n \)'s child up to take \( n \)'s place. \( n \)'s parent becomes the parent of \( n \)'s child, and \( n \)'s child becomes the child of \( n \)'s parent. Dispose of \( n \).
5. If $n$ has two children:
   - Let $x$ be the node in $n$'s right subtree with the smallest key. Remove $x$; since $x$ has the minimum key in the subtree, $x$ has no left child and is easily removed.
   - Replace $n$'s entry with $x$'s entry. $x$ has the closest key to $k$ that isn't smaller than $k$, so the binary search tree invariant still holds.
Running Times

- In a perfectly (full) balanced binary tree with height/depth $h$, the number of nodes $n = 2^{(h+1)} - 1$.
- Therefore, no node has depth greater than $\log_2 n$.
- The running times of $\text{find}()$, $\text{insert}()$, and $\text{remove}()$ are all proportional to the depth of the last node encountered, so they all run in $O(\log n)$ worst-case time on a perfectly balanced tree.
Running Times

- What’s the running time for this binary tree?
- The running times of find(), insert(), and remove() are all proportional to the depth of the last node encountered, but $d = n - 1$, so they all run in $O(n)$ worst-case time.
Running Times

• The Middle ground: reasonably well-balanced binary trees
  – Search tree operations will run in $O(\log n)$ time.
• You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate balanced trees or not.
Running Times

• Binary search trees offer $O(\log n)$ performance on insertions of randomly chosen or randomly ordered keys (with high probability).

• Technically, all operations on binary search trees have $\Theta(n)$ worst-case running time.

• Algorithms exists for keeping search trees balanced. e.g., 2-3-4 trees.
“Holy Grail”

• Given a set of objects and an object $x$, determine immediately (constant time) if $x$ is in the set.

• What’s a situation where you can determine set membership in constant time?
  – The set contains integers with bounded values, i.e. for every $x$ in the set, $L < x < R$, and $L$ and $R$ are known.
General Pattern

• What’ve seen in a variety of data structures is the following behavior:

• The search may be slow if you are looking at a linear data structure and faster in the case of a binary search tree, where each step rules out half of the remaining candidates.
Array-like Search

• If we know where the item should be located in an array, given its index, search can be implemented in constant time.

• Key is to figure out how to do the small amount of computation.
Dictionaries

Problem:

– You have a large set of <Key, Value> pairs, e.g., <word, definition> pair.
– You want to be able to look up the definition of any word very quickly.
– How can we do this efficiently?
Naïve Data Structure

• Consider a limited version of the previous problem:
  – You are building a dictionary for only the 2-letter words in the English language.
  – How many 2-letter combinations are there?
  – \(26 \times 26 = 676\) possible two-letter words.

• Now we can:
  – Create an array with 676 references, initially all null.
  – Define a function `hashCode()` that maps each 2-letter word to a unique integer between 0 and 675.
  – This unique integer is an index into the array and the element at the index contains the definition of the word.
  – We can retrieve a definition in constant time, if it exists.
public class WordDictionary {
    private Definition[] defTable = new Definition[Word.WORDS];
    public void insert(Word w, Definition d) {
        defTable[w.hashCode()] = d;
    }
    Definition find(Word w) {
        return defTable[w.hashCode()];
    }
}

public class Word {
    public static final int LETTERS = 26, WORDS = LETTERS * LETTERS;
    public String word;
    //this function maps a 2 letter word to a number between 0 and 267
    public int hashCode() {
        return LETTERS * (word.charAt(0) - 'a') + (word.charAt(1) - 'a');
    }
}

Note: Java converts char to int automatically you can use chars in arithmetic operations.
Dictionaries

• What if we want to store every English word, not just the two-letter words?
  – The table "defTable" must be long enough to accommodate *pneumonoultramicroscopicsilicovolcanoconiosis*, 45 letters long (*according to the Oxford Dictionary "a facticious word alleged to mean 'a lung disease caused by the inhalation of very fine silica dust causing inflammation in the lungs. Occurring chiefly as an instance of a very long word.")
  – Unfortunately, declaring an array of length $26^{45}$ is out of the question.
  – English has fewer than one million words, so we should be able to do better.
Hash Table

- Suppose $n$ is the number of keys (words) whose definitions we want to store, and suppose we use a table of $N$ buckets, where $N$ is a bit larger than $n$, but much smaller than the number of possible keys.

  - $<\text{WordA, DefA}>$
    - $\text{hashCode(WordA)} = 1000$
    - $h(\text{hashCode(WordA)}) = 1000 \mod 6 = 4$

- A **hash table** is an array of size $N$ that maps a huge set of possible keys into its $N$ elements, called buckets, by applying a **compression function** to each hash code.

- The obvious compression function is:
  $$h(\text{hashCode}) = \text{hashCode mod } N \text{ (everything is in } 0 \text{ to } N-1)$$
Another Example

- $N = 200$ <Key, Value> items.
- Keys are longs, evenly spread over the range $0..2^{63} - 1$.
- $hashCode(K) = K$
- $h(hashCode(K)) = hashCode(K) \mod N$
- 100232, 433, and 10002332482 go into different buckets,
- But 10, 400210, and 210 all go into the same bucket.
Collision

- Several keys are hashed to the same bucket in the table if: 
  \[ h(\text{hashCode}(K1)) = h(\text{hashCode}(K2)). \]

\[ <\text{WordB, DefB}> \]
\[ \text{hashCode(WordB)} = 742 \]
\[ h(\text{hashCode(WordB)}) = 742 \mod 6 = 4 \]

- How to deal with collisions?
- How to design hash code to reduce the likelihood of collisions?
Chaining

• Idea:
  – Each bucket stores a chain (or linked list) of entries with the same hashcode.
  – For a new item, find its bucket and append the item to the end of the list.

• For this to work well, the hash code must avoid hashing keys to the same bucket.

• Example: #buckets \( N = 100 \)
Hash Table Operations

• Hash tables usually support at least three operations.
  
  - public Entry insert(key, value)
    1. Compute the key's hash code and compress it to determine the entry's bucket.
    2. Insert the entry (key and value together) into that bucket's list.

  - public Entry find(key)
    1. Hash the key to determine its bucket.
    2. Search the list for an entry with the given key. If found, return the entry; otherwise, return null.

  - public Entry remove(key)
    1. Hash the key to determine its bucket.
    2. Search the list for an entry with the given key. Remove it from the list if found. Return the entry or null.
Open Addressing

• Idea:
  – Put one data item in each bucket.
  – When there is a collision, just use another.

• Various ways to do this:
  – Linear probes: If there is a collision at $h(K)$, try $h(K)+m$, $h(K)+2m$, etc. (wrap around at end).
  – Quadratic probes: $h(K) + m$, $h(K) + m^2$, . . .
  – Double hashing: $h(K) + h'(K)$, $h(K) + 2h'(K)$, etc.

• Example:
  – $hashCode(K) = K$, $h(hashCode(K)) = K \mod N$, with $N = 10$, linear probes with $m = 1$.
  – Add 1, 2, 11, 3, 102, 9, 18, 108, 309 to empty table.

| 108 | 1 | 2 | 11 | 3 | 102 | 309 | 18 | 9 |

  – Things can get slow, even when table is far from full.
Load Factors

• The load factor of a hash table is $n/N$,
  – where $n$ is the number of keys in the table and
  – $N$ is the number of buckets
  – $n/N$ is the length of the bucket’s list if all entries are truly uniformly distributed.

• The hash code and compression function are "good," if the load factor stays within a small constant ($< 1$) the linked lists are all short, and each operation takes $O(1)$ time.

• However, if the load factor grows too large, performance is dominated by linked list operations and degenerates to $O(n)$ time.
Hash Code and Compression Function

• How do we design a “good” hash code and compression function?
  – Unfortunately it’s a bit of a black art.
  – Ideally, hash code and compression function maps each key to a uniformly distributed random bucket from zero to $N-1$ for any input.
  – Note: random does not mean that the hash code gives a random value each time. Hash code on the same object should return the same value each time!
A Bad Compression Function

• Consider integers:
  – Try $hashCode(i) = i$.
  – Then $h(hashCode) = hashCode \mod N$ where $N$ is 10000.
  – What’s wrong with this?

• Consider an application that only generates integer divisible by 4:
  – Any integer divisible by 4 $\mod 10000$ is divisible by 4.
  – Three quarters of the buckets are wasted!
Reading

- Objects, Abstraction, Data Structures and Design using Java 5.0
  - Chapter 8 pp472-476 pp479-480