CS 106X
Lecture 21: Hashing
Wednesday, March 1, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 15
• Logistics
• Regrade requests due Friday
• Meeting sign-up with Chris:
  • http://stanford.edu/~cgregg/cgi-bin/inperson/index.cgi

• Binary Search Trees: using references to pointers
• Hashing
To insert into a binary search tree, we must update the left or right pointer of a node when we find the position where the new node must go.

In principle, this means that we could either

1. Perform arms-length recursion to determine if the child in the direction we will insert is NULL, or
2. Pass a reference to a pointer to the parent as we recurse.

The second choice above is the cleaner solution.

```python
set.insert(5)
```

```
       6
      /   \\
     2     8
    /     /   \
   1     4     3
  /     /     /       insert here
3
```
```cpp
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```

Using References to Pointers
Using References to Pointers

```cpp
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```

node (reference)

root

insert here
Using References to Pointers

```cpp
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```
Using References to Pointers

```cpp
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}

Using References to Pointers

node (reference)
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
Hashing!

First, a cool program, written by Chris Piech
First, a cool program, written by Chris Piech

We'll see how this was written by the end of lecture!
What we want is a way to implement find(), insert(), and remove() in O(1) time (the "holy grail").

There is a completely different method than what we have discussed before for storing key/value pairs that can actually do this! The method is called hashing, and to perform hashing, you use a hash function.

The values returned by a hash function are called hash values, hash codes, or (simply), hashes.
Suppose you have:
2-letter words and their definitions

A word is the key that addresses the value (definition)
We want to store these in an efficient data structure.

You could do this in a set, but a set is $O(\log n)$ -- can we do better? Yes!
We want to insert a definition into the dictionary, which is going to be comprised of an array (or "buckets"):

function `hashCode()` maps each two-letter word (key) to 0..675 buckets

Index into array
Possible definition for a Hash Function: Any algorithm that maps data to a number, and that is deterministic.

Example:
ox —> take each character, and treat it as a base-26 number. E.g., each character is assigned a number from 0-26:

Then, to get the hash for “ox”: $26 \times 14 + 1 \times 23 = 387$
“at” : $26 \times 0 + 1 \times 19 = 19$
Our current definition for a Hash Function: Any algorithm that maps data to a number, and that is deterministic.

Why does this help us with our goal for $O(1)$ access?

If we can map two-letter words deterministically, then we can put their definitions into an array at that location, and have $O(1)$ access!

e.g.,

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>...</th>
<th>19</th>
<th>20</th>
<th>...</th>
<th>386</th>
<th>387</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Definition of “at”**

**Definition of “ox”**
Our current definition for a Hash Function: Any algorithm that maps data to a number, and that is deterministic.

The idea is that we can now hash a key, put the value into an array (O(1)), find the value with the hash (O(1)), and delete the value from the array (O(1)).

If the hash function is fast, then all the operations we want are also fast.

Let's write some code!
const int LETTERS = 26;
const int WORDS = LETTERS * LETTERS;

class Word {
public:
   // limited to 2-character words
   int hashCode() {
       return LETTERS * (word[0] - 'a') + (word[1] - 'a');
   }

   Word(string w) { // constructor
       word = w;
   }

private:
   string word;
};

class WordDictionary {
public:
   WordDictionary() { // constructor
      defTable = new string[WORDS];
   }

   ~WordDictionary() { // destructor
      delete [] defTable;
   }

   void insert(Word w, string d) {
      defTable[w.hashCode()] = d;
   }

   string find(Word w){
      return defTable[w.hashCode()];
   }

private:
   string *defTable;
};
We've succeeded!
We have figured out a way to store key-value pairs with perfect O(1) access.

But wait…
What if we want to store all English words?

<table>
<thead>
<tr>
<th>Word</th>
<th>Letters</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methionylthreonylthreonylglutaminylarginyl...isoleucine</td>
<td>189,819</td>
<td>Chemical name, disputed...</td>
</tr>
<tr>
<td>Methionylglutaminylarginyltyrosylglutamyl...serine</td>
<td>1,909</td>
<td>Longest published word</td>
</tr>
<tr>
<td>Lopadotemachoselachogaleokranioleipsano...pterygon</td>
<td>183</td>
<td>Longest word coined by a major author</td>
</tr>
<tr>
<td>Pneumonoultramicroscopicsilicovolcanoconiosis</td>
<td>45</td>
<td>Longest word in a major dictionary</td>
</tr>
<tr>
<td>Supercalifragilisticexpialidocious</td>
<td>34</td>
<td>Famous for being created for Mary Poppins</td>
</tr>
<tr>
<td>Pseudopseudohypoparathyroidism</td>
<td>30</td>
<td>Longest non-coined word in a major dictionary</td>
</tr>
<tr>
<td>Floccinaucinihilipilification</td>
<td>29</td>
<td>Longest unchallenged nontechnical word</td>
</tr>
<tr>
<td>Antidisestablishmentarianism</td>
<td>28</td>
<td>Longest non-coined and nontechnical word</td>
</tr>
<tr>
<td>Honorificabilitudinitatibus</td>
<td>27</td>
<td>Longest word in Shakespeare's works; longest word in the English language featuring alternating consonants and vowels.</td>
</tr>
</tbody>
</table>
What if we want to store all English words?

"The summit where Tamatea, the man with the big knees, the climber of mountains, the land-swallow who travelled about, played his nose flute to his loved one"
Supercalifragilisticexpialidocious

Would need an array with $26^{34}$ buckets...too big!

English has \(~700,000\) words

p.s. $26^{34} \approx 10^{48}$, which is about the number of iron atoms in the Earth.
We need to conserve space. For a 700,000 word dictionary, we might only want to use a 800,000 element array (or much smaller, if we wanted only a subset of values)

**Better** definition for a Hash Function: Any algorithm that maps data to a number, that is *deterministic*, and that maps to a **fixed number of locations**.

But, remember, we need to store arbitrary words (i.e., we could add any word, of any length with the characters in our dictionary!)
Better Hash Function definition: Any algorithm that maps data to a number, that is *deterministic*, and that maps to a **fixed number of locations**.

A good method for mapping to a fixed number of locations is to use the modulus operator:

\[ h(\text{hashCode}) = \text{hashCode} \mod N \]

Where \( N \) is the length of the array we want to use. We say we have “compressed” the hash.
Hash Tables

Using the compression function to place keys into a fixed array, we have created a hash table. A hash table maps a huge set of possible keys into N buckets by applying a compression function to each hash code.

\[ h(\text{hashCode}) = \text{hashCode} \mod N \]

\[ 0..N-1 \]
Hash Codes and Compression Functions

Hash codes must be deterministic. Hash codes should be fast and distributed.

\[
\text{key} \quad \longrightarrow \quad \text{hash code} \quad \longrightarrow \quad [0, N-1]
\]

Birthday Hashing / Compressing: Hashing you!

a. decade of your birth year: \((\text{year} / 10) \% 10\)

b. last digit of your birth year: \(\text{year} \% 10\)

c. last digit of your birth month: \(\text{month} \% 10\)

d. last digit of your birth day: \(\text{day} \% 10\)
Keys into Buckets

There were some problems with our birthday hashes. What was the biggest problem?

We need to consider how big our hash table array is, relative to the number of keys we want to store.

\[ n : \text{number of keys (words) stored} \]
\[ N : \text{number of buckets in a table} \]

* a bit bigger than \( n \)
  * but much smaller than the number of possible keys
As we saw in the birthday hashes, it is the case that we can map two keys to the same bucket. E.g.,

$$h(\text{hashCode1}) = h(\text{hashCode2})$$
Example:

- key space: integers
- table size: 10
- `hashCode(K) = K mod 10`
- Insert: 7, 18, 41, 34
- How do we **find** them?
  - Can we perform `findMax()`
  - or `findMin()`?
- What if we now try to add 54?
Handling Collisions

Chaining: each bucket references a linked list of entries, called a *chain*.

**Q:** How do we know which definition corresponds to which word?

**A:** Store each key in the table with its definition
Chaining

entry = (key, value)

defTable

cat
mammal

swirl
spiral

novel
book

song

ox
beast of burden

hint
word invented by Shakespeare

supercalifragilisticexpialidocious
The Load Factor of a hash table:

\[
\frac{n \text{ : number of keys (words) stored}}{N \text{ : number of buckets in a table}} = \frac{n}{N}
\]

IF:

• The load factor stays low, and

• The hash code and compression function are “good,” and

• No duplicate keys, THEN

Each operation takes O(1) time!
The Load Factor of a hash table:

\[ \frac{n}{N} = \frac{\text{number of keys (words) stored}}{\text{number of buckets in a table}} \]

However, IF:

• The load factor gets big \( (n >> N) \), THEN

Each operation takes \( \Theta(n) \) time. 😞

If your load factor gets to big, move your hash table to a bigger array (the penalty is worth it).
Hash codes must be deterministic. Hash codes should be fast and distributed.

Ideal Hash: Map each key to a random bucket. Is an ideal hash collision-free? What does your intuition say about 2500 keys in 1,000,000 buckets?

Even an ideal hash will not remove all collisions:

"if 2,500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is a 95% chance of at least two of the keys being hashed to the same slot." [Wikipedia, Hash Table]
Hash Codes and Compression Functions

What makes a good compression function?

**Bad compression function:**
- Suppose keys are ints
- `hashCode(i) = i` (hashCode is itself)
- **Compression function** `h(hashCode) = hashCode mod N`
- `N = 10,000` buckets

Suppose keys are divisible by 4.

- `h()` is divisible by 4 too!
- **Bad news:** 3/4 of the buckets are never used!

The fix: make `N` prime. Now, once you take mod `N`, the numbers are not divisible by any number in particular.
A better compression function:

\[ h(\text{hashCode}) = ((a \times \text{hashCode} + b) \mod p) \mod N \]

\(a, b, p\): positive integers
\(p\) is a large prime
\(p >> N\)

Now, \(N\) (buckets) doesn’t need to be prime.
static int P = 16908799;
int hashCode(String key) {
    int hashVal = 0;
    for (int i=0; i<key.length(); i++) {
        hashVal = (127 * hashVal + key.charAt(i)) % P;
    }
    return hashVal;
}
Bad Hashcodes for Strings

1) Sum ASCII values of characters.
   - rarely exceeds 500 for most words
   - bunched up into 500 buckets
   - anagrams always collide!

2) Choose first three letters in a word, with $26^3$ buckets.
   - lots of words that begin with the same three letters (e.g., “pre”) but many that don’t (e.g., “xgs”)

3) Suppose we change $p$ in our previous hashCode() to 127.
   - bad because: $(127 \times \text{hashVal}) \mod 127 = 0$. 
Back to Hasham!
Hasham requires a large search space

Does anyone recognize this song?

Wang, A. An Industrial-Strength Audio Search Algorithm
Hasham requires a large search space

Does anyone recognize this song?

Wang, A. An Industrial-Strength Audio Search Algorithm
Hasham requires a large search space

Wang, A. An Industrial-Strength Audio Search Algorithm
Hasham requires a large search space

Should we hash the whole thing?
   No.
Find Pairs of Notes

Wang, A. An Industrial-Strength Audio Search Algorithm
Note Pairs

Wang, A. An Industrial-Strength Audio Search Algorithm
Note Pairs

![Graph showing frequency vs. time for different note pairs](image)

Votes:
- Dirty Paws (945)
- All Right (185)
- Nought (185)
- You Can Call (75)
- Me At (435)
- You Can Call (435)
- Riptide (35)
Note Pairs
Note Pairs

![Graph showing frequency vs. time](image)

**Votes**
- Dirty Paws (94s)
- All Right (18s)
- Night (7s)
- You Can Call (7s)
- Me At (43s)
- You Can Call (43s)
- Riptide (3s)
Note Pairs
int hash(int f1, int f2, int timeDelta) {
    int p = 31;
    int pre = f1 + (p * f2) + (p * p * timeDelta);
    return pre % NUM_BUCKETS;
}

You Can Call Me Al – Paul Simon. 23s,
You Can Call Me Al – Paul Simon. 54s,
Message in a Bottle – Police. 92s
Extra Slides
int main()
{
    WordDictionary wd;
    Word w1("ox");
    Word w2("at");

    // insert definitions
    wd.insert(w1, "bovine work animal");
    wd.insert(w2, "a place where something is");

    // find definitions for a word
    cout << wd.find(w1) << endl;
    cout << wd.find(w2) << endl;
}

Main for 2-character Dictionary
• **References:**
  - Wikipedia Hash Table: http://en.wikipedia.org/wiki/Hash_table (very good)
  - Powerpoint: http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/hashing.pdf

• **Advanced Reading:**
  - Youtube video: https://www.youtube.com/watch?v=MfhjkfocRR0 (good, 7min)