CS 180 - Hashing

Announcements

- Checkpoint on Monday
- Program due Thursday
- HW will be up next week, due last day of class
- Review session: Friday of finals week at 10:30
Recap of trees

BSTs - insert, find & remove in $O(n)$ time

AVL trees:
- insert, find, & remove in $O(\log n)$ time

Key idea: height-balance property
Other trees

- Splay trees: After every insert/delete, performs a move-to-root operation called splaying, which gives an amortized $O(\log n)$ behavior.

- Red-Black trees: more complex than AVL trees and give only $O(1)$ "rotations" after each insert or delete.
New problem: Data Storage

<table>
<thead>
<tr>
<th>Locker #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Dan</td>
</tr>
<tr>
<td>355</td>
<td>Kevin</td>
</tr>
<tr>
<td>101</td>
<td>Tracy</td>
</tr>
<tr>
<td>53</td>
<td>Nitish</td>
</tr>
<tr>
<td>201</td>
<td>David</td>
</tr>
</tbody>
</table>

We want to be able to retrieve a name quickly when given a locker number.

\[
(\text{Let } n = \# \text{ of people, } m = \# \text{ of lockers})
\]
How could we store this?

1. Vectors
   - Size: $O(m)$
   - Find: $O(1)$

2. List
   - Size: $O(n)$
   - Find: $O(n)$

3. AVL Tree
   - Size: $O(n)$
   - Find: $O(\log n)$
Other examples

- Course # and schedule info
- Flight # and arrival info
- URL and html page
- Color and BMP file

Not always easy to figure out how to store and look up.
Dictionaries

A data structure which supports the following:

```c
void insert (keyType &k, dataType &d)
dataType find (keyType &k)
void remove (keyType &k)
```

Note: Everything is based on keys!

Don't know keyType - might not correspond to an int.
Data Structures

First thing to note: 
An array is a dictionary.

key: index
data: value at a position

Other alternatives:
(go back 3 slides) 
But these aren’t good enough.
Hashing

Assuming $m > n$, an array is not very space efficient.

We would like to use $O(n)$ space, not $O(m)$.

But then the key needs to get smaller.
Definition: A hash function \( h \) maps each key in our dictionary to an integer in the range \([0, N-1]\). \( N \neq n \)

(N should be much smaller than \( m = \# \) of keys.)

Then given \((k, e)\), we store \((k, e)\) in array spot \( A[h(k)] \).

Diagram:
- Keys: \( M \)
- Spots: \( h(k) \) in \( A \)
- \( h(k) \) \( \rightarrow \) \((k, e)\)
Good hash functions:

- Are fast: $O(1)$
- Don’t have collisions: $\exists$ are unavoidable:
  Putting 100 things into 10 spots means some will collide.

Key space (size $m$) $\rightarrow (k, e) \rightarrow h(k) \rightarrow \begin{array}{c} 0 \\ 1 \\ 2 \\ \vdots \\ N-2 \\ N-1 \end{array}$

Minimize collisions & deal with them if they happen.

$(k_1, e_1) \neq (k_2, e_2)$ but $h(k_1) = h(k_2)$
So we have a few steps.

1. Take k and make it a number. (goal 32-bit key)
   (Remember, keys can be anything!)
   Ex: char, int, or short (all 32-bits)

   ASCII already there
   already there
Ex: long or float - 64 bits
(K needs to be 32 bits)

32 bits
64 bits

3

\[ a + b \] is 32-bit
```c
int hashCode (long x) {
    return int((unsigned long)(x >> 32)) + int(x) * 3;
}
```

- Shifts over 32 bits
- Cuts off those 32 bits

O(1)
What about strings?

(Think ASCII.)

69 + 114 + 105 + 110 = 32-bit

Goal: a single int.
But, in some cases, a strategy like this can backfire:

temp01 and temp10 and pm0te1

call hash to same #

We want to avoid collisions between “similar” strings (or other types).
A Better Idea: Polynomial Hash Codes

Pick \( a \neq 1 \) and split data into \( k \) 32-bit parts: \( x = (x_0, x_1, x_2, x_3, \ldots, x_k) \)

Let \( h(x) = x_0a^{k-1} + x_1a^{k-2} + \cdots + x_{k-2}a + x_{k-1} \)

Ex: Erin with \( a = 37 \)

\[ h(\text{Erin}) = \frac{69}{37^3} + \frac{114}{37^2} + \frac{105}{37} + \frac{110}{1} \]
Side Note: How long does this take?
(In terms of \( k = \# \) of parts)

\[ h(x) = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1} \]

\( k \) additions

\( 2k \) multiplications \( \rightarrow k^2 \) additions

\( (k^2 \text{ mult.}) \)

HORNER'S RULE:

\[ x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots )) \]

\( 6k \) additions + \( k \) multiplications
Polynomial Hashing

This strategy makes it less likely that similar keys will collide. (Works for floats, strings, etc.)

What about overflow? Take modulo all these exponents mean a bit #
Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by a shift each 32-bit piece by some # of bits.

Also works well in practice.

modulo or cyclic shifts are commonly used
Key space
letters, #s, whatever

1

Int (32 bits)

h

\[ \{0, \ldots, N-1\} \]

#3 collisions
Step 2: Compression maps

Now we can assume every key $k$ is an integer.
Need to make it fit between $0$ and $O \cdot N$ (not $0$ and $2^{32}$).

Idea:
- map everything to $0$

Why is that bad?

one giant collision
Modular compression maps

Take \( h(k) = k \mod N \)

What does \( \mod \) mean again?

(like we used in leaky stacks)

\[ \text{remainder} \]
While good, strange behaviors can happen.