CSE 326: Data Structures
Hashing
Neva Cherniavsky
Summer 2006

Announcements
• Midterms
  › Gary will hand out tomorrow
• Project Phase C due tomorrow
  › Brief overview of Kruskal’s method today

Cute Application
• Build a random maze by erasing edges.

Cute Application
• Pick Start and End

Cute Application
• Repeatedly pick random edges to delete.

Desired Properties
• None of the boundary is deleted
• Every cell is reachable from every other cell.
• There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.
A Cycle

Start

End

A Good Solution

Start

End

Number the Cells

We have disjoint sets $S = \{(1), (2), (3), \ldots, (36)\}$, each cell is unto itself. We have all possible edges $E = \{(1, 2), (1, 7), (2, 8), (2, 3), \ldots\}$ 60 edges total.

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>End</td>
</tr>
</tbody>
</table>

Basic Algorithm

- $S$ = set of sets of connected cells
- $E$ = set of edges
- Maze = set of maze edges initially empty

While there is more than one set in $S$
pick a random edge $$(x,y)$$ and remove from $E$
u := Find(x);
v := Find(y);
if $u = v$ then
Union(u,v);
else
add $(x,y)$ to Maze
All remaining members of $E$ together with Maze form the maze

Example Step

Pick (8,14)

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>End</td>
</tr>
</tbody>
</table>

Example

We have disjoint sets $S = \{(1,2,7,8,9,13,19), (3), (4), (5), (6), (10), (11), (12), (14), (20), (26,27), (15), (16,21)\}$

Find(8) = 7
Find(14) = 20
Union(7,26)

We have disjoint sets $S = \{(1,2,7,8,9,13,19,14,20,26,27)\}$

Find(8) = 7
Find(14) = 20
Union(7,26)
Hash Tables

• Constant time accesses!

• A hash table is an array of some fixed size, usually a prime number.

• General idea:

```plaintext
hash function: h(K)
```

key space (e.g., integers, strings)  TableSize – 1

Simple Hash Table

T

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Hash function:

\[ h : U \rightarrow \{ 0, 1, ..., \text{TableSize} - 1 \} \]

U is the universe of keys

\[ h(\text{name}) \] is the hash value of “name”

\[ h(\text{Judy Jones}) = 4 \]

\[ h(\text{Jerry Lee}) = 7 \]

Find(“name”) = T[h(“name”)]

Example

• key space = integers

• TableSize = 10

• \( h(K) = K \mod 10 \)

• Insert: 7, 18, 41, 94

Another Example

• key space = integers

• TableSize = 6

• \( h(K) = K \mod 6 \)

• Insert: 7, 18, 41, 34
General Idea

- Key space of size M, but we only want to store subset of size N, where N<<M.
  - Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
  - Keys are student names. We want to look up student records quickly by name.
  - Keys are chess configurations in a chess playing program.
  - Keys are URLs in a database of web pages.

Hash Functions

1. simple/fast to compute,
2. Avoid collisions
3. have keys distributed evenly among cells.

Time for insert/delete/find?

 Downsides?

Sample Hash Functions:

- key space = strings
- $s = s_0 s_1 s_2 \ldots s_{k-1}$

1. $h(s) = s_0 \mod \text{TableSize}$
2. $h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \text{TableSize}$
3. $h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 3^i \right) \mod \text{TableSize}$

Designing a Hash Function for web URLs

$s = s_0 s_1 s_2 \ldots s_{k-1}$

Issues to take into account:

$h(s) =$

Good Hash Functions

- Integers: Division method
  - Choose Hsize to be a prime (Why?)
  - $h(n) = n \mod \text{Hsize}$
  - Example. Hsize = 23, $h(50) = 4$, $h(1257) = 15$
  - When might this fail?

Character Strings

- $x = a_0a_1a_2\ldots a_m$ is a character string. Define
  - $\text{int}(x) = a_0 + a_1 \cdot 128 + a_2 \cdot 128^2 + \ldots + a_m \cdot 128^{m-1}$
  - $h(x) = \text{int}(x) \mod \text{Hsize}$

- Compute $h(x)$ using Horner’s Rule
  - $h := 0$
  - for $i = m$ to 0 by -1 do $h := (a_i + 128h) \mod \text{Hsize}$
  - return $h$
### tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - `tableSize = 10`
    - data hashes to 0, 3, 0, 5, 1, 0, 0
  - `tableSize = 11`
    - data hashes to 10, 9, 5, 0, 2, 9, 7

### A Bad Hash Function

- Keys able1, able2, able3, able4
  - `Hsize = 128`
  - `int(ablex) mod 128 = int(a) = 97`
  - Thus, `h(ablex) = h(abley)` for all `x` and `y`

  What is the central problem we're trying to avoid?
  - How can we fix it?