CSE 326 Data Structures

Dave Bacon

Hashing / Disjoint Sets
Logistics

- Homework 5 on web due Friday
- Project 3 out soon
- Reading: Weiss Chapter 8
Double Hashing

\[ f(i) = i \times g(k) \]

where \( g \) is a second hash function

- Probe sequence:
  
  0\(^{th}\) probe = \( h(k) \mod \text{TableSize} \)
  
  1\(^{th}\) probe = \( (h(k) + g(k)) \mod \text{TableSize} \)
  
  2\(^{th}\) probe = \( (h(k) + 2 \times g(k)) \mod \text{TableSize} \)
  
  3\(^{th}\) probe = \( (h(k) + 3 \times g(k)) \mod \text{TableSize} \)
  
  ... 
  
  \( i\(^{th}\) \) probe = \( (h(k) + i \times g(k)) \mod \text{TableSize} \)
Double Hashing Example

\[ h(k) = k \mod 7 \quad \text{and} \quad g(k) = 5 - (k \mod 5) \]

<table>
<thead>
<tr>
<th>Probe 1</th>
<th>Probe 2</th>
<th>Probe 3</th>
<th>Probe 4</th>
<th>Probe 5</th>
<th>Probe 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>93</td>
<td>40</td>
<td>47</td>
<td>10</td>
<td>55</td>
</tr>
</tbody>
</table>
Resolving Collisions with Double Hashing

Hash Functions:
\[ H(K) = K \mod M \]
\[ H_2(K) = 1 + ((K/M) \mod (M-1)) \]

\[ M = \]

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13
28
33
147
43
Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

• When to rehash?
  – half full ($\lambda = 0.5$)
  – when an insertion fails
  – some other threshold

• Cost of rehashing?
Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity. (cost of doubling table and rehashing is amortized over many inserts)
Disjoint Sets

Chapter 8
Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
- Each set has a unique name, one of its members.
  - \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
Union

- Union(x, y) – take the union of two sets named x and y
  - \{3, 5, 7\}, \{4, 2, 8\}, \{9\}, \{1, 6\}
  - Union(5, 1)
    - \{3, 5, 7, 1, 6\}, \{4, 2, 8\}, \{9\},
**Find**

- **Find(x)** – return the name of the set containing x.
  - \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
  - \text{Find}(1) = 5
  - \text{Find}(4) = 8
Building Mazes

- Build a random maze by erasing edges.
Building Mazes (2)

- Pick Start and End
Building Mazes (3)

- Repeatedly pick random edges to delete.
Desired Properties

- None of the boundary is deleted.
- Every cell is reachable from every other cell.
- Only one path from any one cell to another. (There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.)
A Cycle
A Good Solution
A Hidden Tree
We have disjoint sets $S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots, \{36\}\}$ 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></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

End
Basic Algorithm

- $S = \text{set of sets of connected cells}$
- $E = \text{set of edges}$
- $\text{Maze} = \text{set of maze edges (initially empty)}$

While there is more than one set in $S$

- pick a random edge $(x,y)$ and remove from $E$
- $u := \text{Find}(x)$
- $v := \text{Find}(y)$
- if $u \neq v$ then
  // removing edge $(x,y)$ connects previously non-connected cells $x$ and $y$ - leave this edge removed!
  $\text{Union}(u,v)$
- else
  // cells $x$ and $y$ were already connected, add this
  // edge to set of edges that will make up final maze.
  add $(x,y)$ to $\text{Maze}$

All remaining members of $E$ together with $\text{Maze}$ form the maze
Example Step

Pick (8, 14)

Start

1  2  3  4  5  6
10 8  9 10 11 12
15 14 15 16 17 18
21 20 21 22 23 24
27 26 27 28 29 30
33 32 33 34 35 36

End

S

\{1, 2, 7, 8, 9, 13, 19\}
\{3\}
\{4\}
\{5\}
\{6\}
\{10\}
\{11, 17\}
\{12\}
\{14, 20, 26, 27\}
\{15, 16, 21\}
\{22, 23, 24, 29, 30, 32, 33, 34, 35, 36\}
Example

\begin{itemize}
\item S = \{1, 2, 7, 8, 9, 13, 19\}
\item 3
\item 4
\item 5
\item 6
\item 10
\item 11, 17
\item 12
\item 14, 20, 26, 27
\item 15, 16, 21
\item \ldots
\item 22, 23, 24, 29, 39, 32
\item 33, 34, 35, 36
\end{itemize}

\begin{itemize}
\item Find(8) = 7
\item Find(14) = 20
\item Union(7, 20)
\item S = \{1, 2, 7, 8, 9, 13, 19, 14, 20, 26, 27\}
\item 3
\item 4
\item 5
\item 6
\item 10
\item 11, 17
\item 12
\item 15, 16, 21
\item \ldots
\item 22, 23, 24, 29, 39, 32
\item 33, 34, 35, 36
\end{itemize}
Example

Pick (19, 20)

Start

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

End

S

\{1, 2, 7, 8, 9, 13, 19, 14, 20, 26, 27\}

\{3\}
\{4\}
\{5\}
\{6\}
\{10\}
\{11, 17\}
\{12\}
\{15, 16, 21\}
\{22, 23, 24, 29, 39, 32, 33, 34, 35, 36\}
Example at the End

S
{1, 2, 3, 4, 5, 6, 7, ..., 36}

E
Maze
Implementing the DS ADT

- $n$ elements,
  Total Cost of: $m$ finds, $\leq n-1$ unions

- Target complexity: $O(m+n)$
  i.e. $O(1)$ amortized

- $O(1)$ worst-case for find as well as union would be great, but...

Known result: both find and union cannot be done in worst-case $O(1)$ time
Attempt #1

• Hash elements to a hashtable
• Store set identifier for each element as data

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:
Attempt #2

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together

*runtime for find:*

*runtime for union:*

*runtime for m finds, n-1 unions:*
Attempt #3

• Hash elements to a hashtable
• Store set identifier for each element as data
• Link all elements in the same set together
• Always update identifiers of smaller set

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

[Read section 8.2]
Up-Tree for Disjoint Union/Find

Initial state: 1 2 3 4 5 6 7

After several Unions:

Roots are the names of each set.
Find Operation

Find(x) - follow x to the root and return the root

Find(6) = 7
Union Operation

Union(x, y) - assuming x and y are roots, point y to x.
Simple Implementation

• Array of indices

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Up[x] = 0 means x is a root.
Implementation

int Find(int x) {
    while (up[x] != 0) {
        x = up[x];
    }
    return x;
}

void Union(int x, int y) {
    up[y] = x;
}

runtime for Union():

runtime for Find():

runtime for m Finds and n-1 Unions:
Find Solutions

Recursive
Find(up[] : integer array, x : integer) : integer {
    //precondition: x is in the range 1 to size/
    if up[x] = 0 then return x
    else return Find(up,up[x]);
}

Iterative
Find(up[] : integer array, x : integer) : integer {
    //precondition: x is in the range 1 to size/
    while up[x] ≠ 0 do
        x := up[x];
    return x;
}
Now this doesn’t look good 😞

Can we do better? Yes!

1. Improve **union** so that **find** only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$

2. Improve **find** so that it becomes even better!
   - Path compression
   - Reduces complexity to almost $\Theta(m + n)$