CS 367 - Introduction to Data Structures
Thursday, Dec 1, 2016

Program 5 due 10 pm Thursday, Dec 15th

Homework 9 due 10 pm tomorrow, Dec 2nd
Program 3 grade reports available – If you have any questions, contact your p3 grader by next week Tuesday 12/6

Last Time
Graphs
- DFS/BFS
- topological ordering
- examples

Today
Graphs
- more terminology
- Dijkstra’s algorithm

Hashing
- terminology
- designing a good hash function
- choosing table size

Next Time
Read: finish Hashing, start Sorting
Hashing
- expanding a hash table
- handling collisions
Java Support for Hashing
Tree Map vs. Hash Map
Sorting Intro
Dijkstra's Algorithm

Psuedo Code

for each vertex V
    initialize V’s visited mark to false
    initialize V's total weight to “infinity”
    initialize V's predecessor to null

set start vertex’s total weight to 0

create new priority queue pq
pq.insert( [start vertex total weight,start vertex] )

while !pq.isEmpty()
    [C’s total weight,C] = pq.removeMin()
    set C’s visited mark to true

    for each unvisited successor S adjacent to C
        if S's total weight can be reduced
            S's total weight = C's total weight + edge weight from C to S
            update S's predecessor to C
            pq.insert( [S’s total weight,S] )
        (if S already in pq we’ll just update S's total weight)
### Dijkstra’s Practice

**Reconstruct shortest path from A to F**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Priority Queue (just list smallest to largest)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited</th>
<th>Total Weight</th>
<th>Predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hashing

Goal:

Concept:

- hash table
- table size (TS)
- load factor (LF)
- key
- hash function
Ideal Hashing

Assume

- store 150 students records
- table is an array of student records
- null is sentinel value meaning element is unused
- key is the student id number, a 5 digit integer

11000, 11001, 11002, … 11048, 11049, … 11148, 11149

→ What would be a good hash function to use on the ID number?

```c
int hash(K key) {
}
```

Trivial Hash Function:

Perfect Hash Function:

```c
void insert(K key, D data) {

D lookup(K key) {

void delete(K key) {
```

The UW uses 10 digit ID numbers: 9012345789 9012345432 9023456789

→ Is a perfect hash function possible for these id numbers?

→ Would the last 3 digits of the ID work as above?

Collision:

Key Issues:

- 
- 
-
Designing a Hash Function

Good Hash Functions:

1.

2.

3.

4.

Java Hash Function Steps:

1.

2.
Techniques for Generating Hash Codes

Integer Key 90123456789

\[
123 \times 11 \quad + \quad 456 \times 121 \quad + \quad 789 \times 1
\]

Extraction

Weighting

Folding
Handling String Keys
Handling Double Keys
Choosing the Table Size

Table Size and Collisions

Assume 100 items with random keys in the range 0 – 9999 are being stored in a hashtable. Also assume the hash function is simply %\text{tablesize}.

→ How likely would a collision occur if the table had 10000 elements? 1000? 100?

Table Size and Distribution

Assume 50 items are stored in a hashtable. Also assume the hashCode function returns multiples of some value \( x \). For example, if \( x = 20 \) then hashCode returns 20, 40, 60, 80, 100, ...

→ How likely would a collision occur if the table had 60 elements? 50? 37?