MA/CSSE 473 Day 28

Hashing review

B-tree overview

Dynamic Programming

MA/CSSE 473 Day 28

Assignment	Old due date	New due date
10	Tuesday, Oct 19	Wednesday, Oct 20
Convex Hull	Thursday, Oct 21	Friday, Oct 22
11	Saturday, Oct 23	Tuesday, Oct 26
12	Tuesday, Oct 26	Thursday, Oct 28
13	Thursday, Oct 28	Wednesday, Nov 3

Convex Hull Late Day until Saturday at 8 AM

HW 11 is a good one to try to earn an extra late day. It is shorter than most assignments.

- **Take-home exam** available by Oct 29 (Friday) at 9:55 AM, due Nov 1 (Monday) at 8 AM.
- **Student Questions**
- Hashing summary
- B-Trees – a quick look
- Dynamic Programming

Convo schedule
Monday:
Section 1: 9:35 AM
Section 2: 10:20 AM

Take-Home Exam

- Available no later than 9:55 AM on Friday October 29. Due 8 AM Monday, Nov 2
- Two parts, each with a time limit of 3-4 hours
  - exact limit will be set after I finish writing questions.
  - Measured from time of ANGEL view of problems to submission back to ANGEL drop box.
- Covers through HW 12 and Section 8.2.
- A small number of problems (3-5 in each part)
- For most problems, partial credit for good ideas, even if you don't entirely get it.
- A blackout on communicating with other students about this course during the entire period from exam availability to exam due time.

Some Hashing Details

- The next slides are from CSSE 230.
- They are here in case you didn't "get it" the first time.
- We will not go over all of them in detail in class.
- If you don't understand the effect of clustering, you might find the animation that is linked from the slides especially helpful.

Collision Resolution: Linear Probing

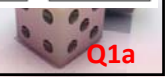
- When an item hashes to a table location occupied by a non-equal item, simply use the next available space.
- Try H+1, H+2, H+3, …
  - With wraparound at the end of the array
- Problem: Clustering (picture on next slide)

Weiss Figure 20.4
Linear probing hash table after each insertion

Animation:
http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89



Q1a

Analysis of linear probing

- Dependent on the **load factor**, λ, which is the ratio of the number of items in the table to the size of the table. Thus $0 \leq \lambda \leq 1$.
- For a given λ, what is the expected number of probes before an empty location is found?
- For simplicity, assume that all locations are equally likely to be occupied, and equally likely to be the next one we look at. Then the probability that a given cell is empty is $1 - \lambda$, and thus the expected number of probes before finding an empty cell is (write it as a summation).

```
> simplify(sum(i*(1-lambda)*lambda^(i-1), i=1..infinity));
```

$$-\frac{1}{\lambda - 1}$$

Analysis of linear probing (continued)

- The "equally likely" probability is not realistic, because of **clustering**
- Large blocks of consecutive occupied cells are formed. Any attempt to place a new item in any of those cells results in extending the cluster by at least one item
- Thus items collide not only because of identical hash values, but also because of hash values that happen to put them into the cluster
- Average number of probes when λ is large:
  - $0.5\ [\ 1 + 1/(1- \lambda)^2\ ]$.
    - For a proof, see Knuth, *The Art of Computer Programming*, Vol 3: Searching Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998.
  - What are the values for $\lambda = 0, 0.5, 0.75, 0.9$?
  - **When λ approaches 1, this gets bad!**
  - But if λ is close to zero, then the average is near 1.0

So why consider linear probing?

- Easy to implement
- Simple code has fast run time per probe
- Works well when load factor is low
  - It could be more efficient just to rehash using a bigger table once it starts to fill.
  - What is often done in practice: rehash to an array that is double in size once the load factor reaches 0.5
- What about other fast, easy-to-implement strategies?

Quadratic probing

- With linear probing, if there is a collision at H, we try H, H+1, H+2, H+3,... until we find an empty spot.
  - Causes (primary) clustering
- With quadratic probing, we try H, $H+1^2$. $H+2^2$, $H+3^2$,...
  - Eliminates primary clustering, but can cause secondary clustering.

Q1b

Hints for quadratic probing

- **Choose a prime number for the array size**
  - If the array used for the table is not more than half full, finding a place to do the insertion is guaranteed , and no cell is probed twice
  - Suppose the array size is p, a prime number greater than 3
  - Show by contradiction that if i and j are ≤ ⌊p/2⌋, and i≠j, then $H + i^2 \not\equiv H + j^2 \pmod{p}$.
- **Use an algebraic trick to calculate next index**
  - Replaces mod and general multiplication with subtraction and a bit shift
  - Difference between successive probes:
    - $H + (i+1)^2 = H + i^2 + (2i+1)$ [can use bit-shift for the multiplication]
    - nextProbe = nextProbe + (2i+1);
      if (nextProbe >= P) nextProbe -= P;

Q2

Quadratic probing analysis

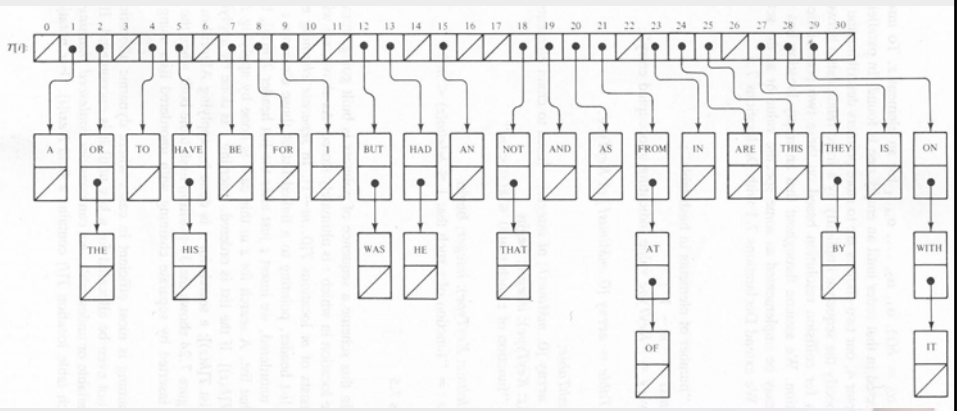
- No one has been able to analyze it
- Experimental data shows that it works well
  - Provided that the array size is prime, and is the table is less than half full

Other approaches to collision resolution

- Double hashing
  - A second hash function is used to calculate an offset **d** to use in probing. Try locations h+d, h+2d, h+3d, etc
- Separate chaining
  - Rather than an array of items, we use an array of linked lists. When multiple items hash to the same location, we add them to the list for that location
  - Picture on next slide
  - No clustering effect
    - But we use space for the links(that space could have been used to make the array larger).
    - If many items have the same hash code, the chains can become long.

Q1c

Hashing with Chaining



Analysis: Hashing with Chaining

- With chaining, the load factor may be > 1.
- Assume a hash function that distributes keys evenly in the table. If there are n keys in the table (backed by an array of size m), the average chain should be λ elements long
- So it takes constant time to compute the hash function plus λ /2 to search within the chain.
- If $\lambda \approx 1$, this is VERY fast
- But there is the extra space for the pointers, which could have been used to make the table larger if open addressing was used