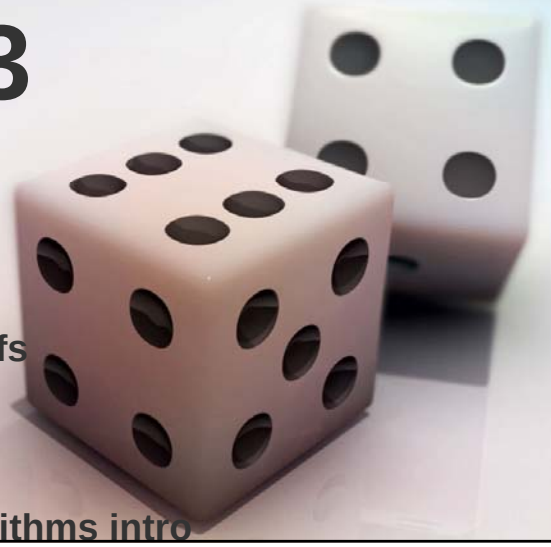
# MA/CSSE 473 Day 23



**Student questions**

**Space-time tradeoffs**

**Hash tables review**

**String search algorithms intro**

Sometimes using a little more space saves a lot of time

## SPACE-TIME TRADEOFFS

## Space vs time tradeoffs

- Often we can find a faster algorithm if we are willing to use additional space.
- Give some examples
- Examples:

## Space vs time tradeoffs

- Often we can find a faster algorithm if we are willing to use additional space.
- Give some examples (quiz question)
- Examples:
  - Binary heap *vs* simple sorted array. Uses one extra array position
  - Merge sort
  - Radix sort and Bucket Sort
  - Anagram finder
  - Binary Search Tree (extra space for the pointers)
  - AVL Tree (extra space for the balance code)

A Quick Review

# HASH TABLE IMPLEMENTATION

## Hash Table Review

- Section 7.4 of Levitin
- Excellent detailed reference: Weiss Chapter 20.
- Covered in 230
  - Both versions of the course
  - A link to one version: http://www.rose-hulman.edu/class/csse/csse230/201230/Slides/17-Graphs-HashTables.pdf
- Three questions on today's handout guide you through a quick review; the above link may be helpful. Do it with two other students. 20 minutes.
- Then we will prove a property of quadratic probing that is described in 230 but seldom proved there.

If you don't understand the effects of clustering, you might find the animation that is linked from this page to be especially helpful.
: http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

# Hashing Review

**Discuss the following questions in a group of three students**

- What problem do we try to solve by hashing?
- What is the general idea of how hashing works?
- Why does it fit into Chapter 7 (space-time tradeoffs)?
- What are the main issues to be addressed when discussing hashing implementation?
- How to choose between a hash table and a binary search tree?

# Terminology and analysis

If any of this terminology is unfamiliar, you should look it up

- collision
- load factor ($\lambda$)
- perfect hash function
- open addressing
  - linear probing
  - cluster
  - quadratic probing
  - rehashing
- separate chaining

## Some Hashing Details …

- Can be found on this page:
- http://www.rose-hulman.edu/class/csse/csse230/201230/Slides/17-Graphs-HashTables.pdf
- Similar to Weiss's presentation
- They are linked from here in case you didn't "get it" the first time in CSSE230.
- We will not go over all of them in detail in class.

## Collision Resolution: Quadratic probing

- With linear probing, if there is a collision at H, we try H, H+1, H+2, H+3, ... (all modulo the table size) 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.
  - Is it possible that it misses some available array positions?
  - I.e it repeats the same positions over and over, while never probing some other positions?

# Hints for quadratic probing

- **Choose a prime number for the array size, then …**
  - If the array is not more than half full, finding a place to do an insertion is guaranteed , and no cell is probed twice before finding it
  - 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
    $H + i^2 \pmod P \not\equiv H + j^2 \pmod P$.
- **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 a bit-shift for the multiplication].
    - nextProbe = nextProbe + (2i+1);
      if (nextProbe >= P) nextProbe -= P;

# 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

Search for a string within another string

## STRING SEARCH

## Brute Force String Search Example

The problem: Search for the first occurrence of a **pattern** of length m in a **text** of length n.
Usually, m is much smaller than n.

- What makes brute force so slow?
- When we find a mismatch, we can shift the *pattern* by only one character position in the *text*.

```
Text:    abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
Pattern: abracadabra
          abracadabra
           abracadabra
            abracadabra
             abracadabra
              abracadabra
```

# Faster String Searching

- Brute force: **worst case m(n-m+1)** **Was a HW problem**
- A little better: **but still Θ(mn) on average**
  - **Short-circuit the inner loop**

```
def search(pattern, text):
    n, m = len(text),len(pattern)
    for i in range(n-m+1):
        j = 0
        while j < m and text[i+j] == pattern[j]:
            j +=1
        if j==m:
            return i
    return False
```

# Horspool's Algorithm Intro

- A simplified version of the Boyer-Moore algorithm
- A good bridge to understanding Boyer-Moore
- Published in 1980
- What makes brute force so slow?
  - When we find a mismatch, we can only shift the pattern to the right by one character position in the text.
  - **Text:** abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
    **Pattern:** abracadabra
    abracadabra
    abracadabra
    abracadabra
- Can we shift farther?
  Like Boyer-Moore, Horspool does the comparisons in a counter-intuitive order (moves right-to-left through the pattern)

## Horspool's Main Question

- If there is a character mismatch, how far can we shift the pattern, with no possibility of missing a match within the text?
- What if the last character in the pattern is compared with a character in the text that does not occur in the pattern at all?
- `Text:    ... ABCDEFG ...`
  `Pattern:      CSSE473`

## How Far to Shift?

- Look at first (rightmost) character in the part of the text that is compared to the pattern:
- The character is not in the pattern

  `.....C.....` {*C* not in pattern)

  **BAOBAB**
- The character is in the pattern (but not the rightmost)

  `.....O.....`(O occurs once in pattern)

  **BAOBAB**

  `.....A.....`(A occurs twice in pattern)

  **BAOBAB**
- The rightmost characters do match

  `.....B.....`

  **BAOBAB**