Course Notes for

CS 1501
Algorithm Implementation

By

John C. Ramirez

Department of Computer Science

University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- So far what data structures do we have that allow for good searching?
  - Sorted arrays (lgN using Binary Search)
  - BSTs (if balanced, search is lgN)
  - Using Tries (Theta(K) where we have K characters in our string)
- Note that using Tries gives us a search time that is independent of N
  - However, Tries use a lot of memory, especially if strings do not have common prefixes

- Can we come up with another Theta(1) search that uses less memory for arbitrary keys?
- Let's try the following:
  - Assume we have an array (table), T of size M
  - Assume we have a function h(x) that maps from our key space into indexes {0,1,…,M-1}
    - Also assume that h(x) can be done in time proportional to the length of the key
  - Now how can we do an Insert and a Find of a key x?

- Insert

```
i = h(x);
T[i] = x;
```

- Find

```
i = h(x);
if (T[i] == x)
    return true;
else
    return false;
```

0	
1	
2	
3	
…	
i	x
…	
M-1	

- Simplistic idea of hashing
  - Why simplistic?
  - What are we ignoring here?
    - Discuss

Collisions

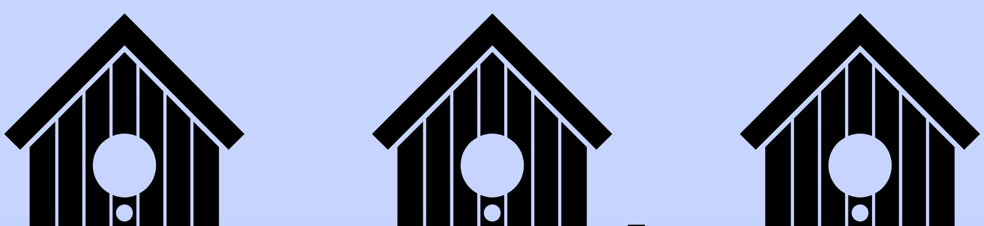
- Simple hashing fails in the case of a collision:

  $$h(x_1) == h(x_2), \text{ where } x_1 \mathrel{!=} x_2$$

  - Two distinct keys hash to the same location!
- Can we **avoid collisions** (i.e. guarantee that they do not occur)?
  - Yes, but only when size of the key space, $\mathcal{K}$, is less than or equal to the table size, M
    - When |K| <= M there is a technique called ***perfect hashing*** that can ensure no collisions
    - It also works if N <= M, but the keys are known in advance, which in effect reduces the key space to N
      - > Ex: Hashing the keywords of a programming language during compilation of a program

Collisions

- When |K| > M, by the pigeonhole principle, collisions cannot be eliminated
  - We have more pigeons (potential keys) than we have pigeonholes (table locations), so at least 2 pigeons must share a pigeonhole
  - Unfortunately, this is usually the case
  - For example, an employer using SSNs as the key
    - Let M = 1000 and N = 500
    - It seems like we should be able to avoid collisions, since our table will not be full
    - However, |K| = 10^9 since we do not know what the 500 keys will be in advance (employees are hired and fired, so in fact the keys change)



- So we must redesign our hashing operations to work despite collisions
  - We call this collision resolution
- Two common approaches:

1) Open addressing
   - If a collision occurs at index i in the table, try alternative index values until the collision is resolved
     - Thus a key may not necessarily end up in the location that its hash function indicates
     - We must choose alternative locations in a consistent, predictable way so that items can be located correctly
     - Our table can store at most M keys

2) Closed addressing

- Each index i in the table represents a collection of keys
  - Thus a collision at location i simply means that more than one key will be in or searched for within the collection at that location
  - The number of keys that can be stored in the table depends upon the maximum size allowed for the collections

- We will look at examples from both of these approaches

- Before discussing resolution in detail
  - Can we at least keep the number of collisions in check?
  - Yes, with a good hash function
    - The goal is to make collisions a (pseudo) "random" occurrence
      - Collisions will occur, but due to chance, not due to similarities or patterns in the keys
  - What is a good hash function?
    - It should utilize the entire key (if possible) and exploit any differences between keys
    - It should also utilize the full address space of the hash table

Reducing the number of collisions

- Let's look at some examples
  - Consider hash function for on-campus Pitt students based on phone numbers, where M = 1000
    - Attempt 1: First 3 digits of number
      - > H(412-XXX-XXXX) = 412
      - > Good or bad?
      - > BAD! First 3 digits are area code and most people in this area live within a few area codes { 412, 724, etc }
    - Better?
      - > Think about this – what can we do?
      - > Take phone number as an integer % M
        - > In effect this is getting the last 3 digits
      - > Why better? Still only 3 digits!
      - > For arbitrary 10-digit numbers the last 3 digits don't have any special designation and tend to be pseudo-random

Reducing the number of collisions

- Consider hash function for words into a table of size M
  - Attempt 1: Add ASCII values
    - > Ex: H("STOP") → 83 + 84 + 79 + 80 = 326
    - > Is this good / bad? Let's think about it…
    - > Problem 1: Does not fully exploit differences in the keys
    - > Ex: H("STOP") = H("POTS") = H("POST") = H("SPOT")
    - > Even though we use the entire key, we don't take into account the positions of the characters
    - > Problem 2: Does not use the full address space
    - > Even small words will have H(X) values in the 100s
    - > Even larger words will have H(X) values well below 1000
    - > Thus for ex. M = 1000 there will likely be collisions in the middle of the table and many empty locations at the beginning and the end of the table

Reducing the number of collisions

- Better?
  - > Utilize all of the characters and the positions and all of the table
    - > How?
  - > Consider **integers** and how they differ from each other
  - > 1234 != 4321 != 2341 != 3412 … etc
  - > Why are they different?
  - > Each digit has a different power of 10
  - > $1234 = 1*10^3+2*10^2+3*10^1+4*10^0$
  - > $4321 = 4*10^3+3*10^2+2*10^1+1*10^0$
- Can we do something similar for hash values of arbitrary strings?
  - > YES!
- Let's first consider this *ideally*, then we will get more practical

Reducing the number of collisions

- Integers with given digits in given positions are different because we have 10 digits and each location is a different power of 10
- We can apply the same idea to ASCII characters
  - > We have 256 ASCII characters, so let's multiply each digit by a different power of 256
- Ex: H("STOP") = $83 * 256^3 + 84 * 256^2 + 79 * 256^1 + 80 * 256^0$
- Ex: H("POTS") = $80 * 256^3 + 79 * 256^2 + 84 * 256^1 + 83 * 256^0$
  - > This will definitely distinguish the hash values of all strings
- Ok this will utilize all of the characters and positions, but what about utilizing all of the table?
  - > Recall that our table is size M
  - > Note that these values will get very large very quickly
  - > So we can take the raw value % M
  - > This will likely "wrap" around the table many times, and should utilize all of the locations

Reducing the number of collisions

- Let's now think about how we will do this *in practice*
  - Note how big the numbers will get – very quickly larger than even a long can store
    - If we use an int or even long the values will wrap and thus no longer be unique for each String
      - This is ok – it will just be a collision
  - Calculating the values should be done in an efficient way so that H(X) can be done quickly
    - There is an approach called Horner's method that can be applied to calculate the H(X) values efficiently
- See handout hashCode.java
  - We will also look at this during our interactive lecture

- One good approach to hashing:
  - Choose M to be a prime number
  - Calculate our hash function as

    h(x) = f(x) mod M

    - where f(x) is some function that converts x into a large "random" integer in an intelligent way
      - It is not actually random, but the idea is that if keys are converted into very large integers (much bigger than the number of actual keys) collisions will occur because of the pigeonhole principle, but they will be less frequent
  - There are other good approaches as well

Linear Probing

- Back to Collision Resolution
  - Open Addressing
    - The simplest open addressing scheme is Linear Probing
      - Idea: If a collision occurs at location i, try (in sequence) locations i+1, i+2, … (**mod M**) until the collision **is resolved**
      - For Insert:
        - Collision is resolved when an empty location is found
      - For Find:
        - Collision is resolved (**found**) when the item is found
        - Collision is resolved (**not found**) when an empty location is found, or when index circles back to i
      - Look at an example

Linear Probing Example

Index	Value	Probes
0		
1		1
2		
3		1
4		2
5		2
6		1
7		3
8		*5*
9		
10		

14	$h(x) = 3$
17	$h(x) = 6$
25	$h(x) = 3$
37	$h(x) = 4$
34	$h(x) = 1$
16	$h(x) = 5$
26	$h(x) = 4$

$$h(x) = x \bmod 11$$

- Performance
  - **Theta(1)** for Insert, Search for normal use, subject to the issues discussed below
    - > In normal use at most a few probes will be required before a collision is resolved
- Linear probing issues
  - What happens as table fills with keys?
  - Define LOAD FACTOR, $\alpha = N/M$
  - How does α affect linear probing performance?
  - Consider a hash table of size M that is empty, using a good hash function
    - > Given a random key, x, what is the probability that x will be inserted into any location i in the table?

$1/M$

Linear Probing

– Consider now a hash table of size M that has a cluster of C consecutive locations that are filled

> Now given a random key, x, what is the probability that x **will be inserted** into the location immediately following the cluster?

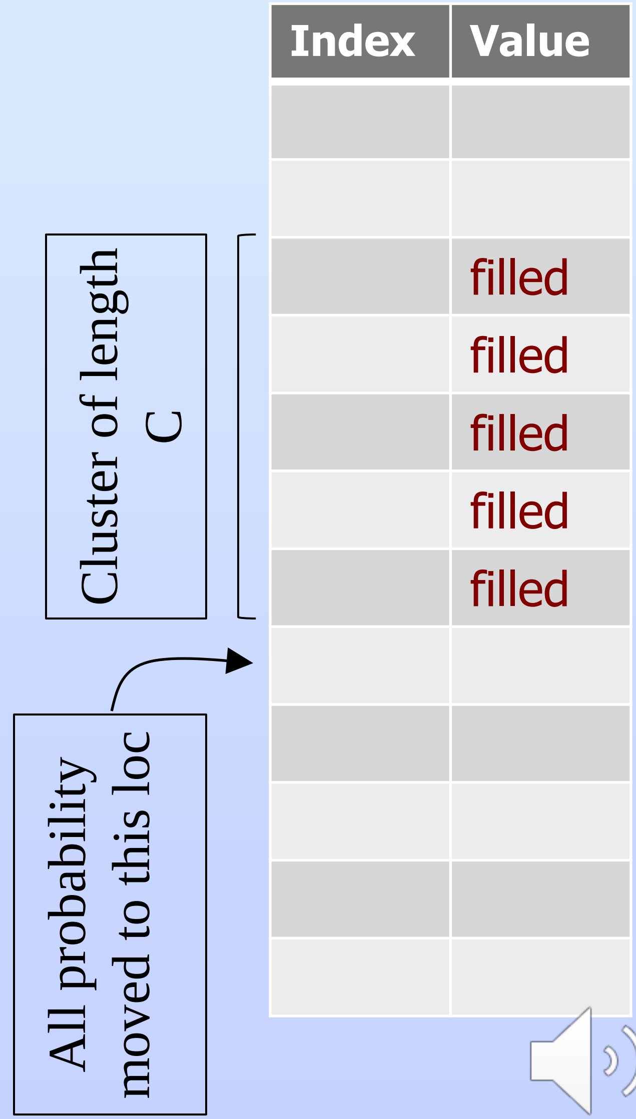
$$(C+1)/M$$

– Why?

> The probability of mapping x to a given location is still 1/M

> But for any i in the cluster C, x will end up after the cluster

> Thus we have C locations in the cluster plus 1 directly after it



Linear Probing

- Why is this bad?
  - Recall how collisions in LP are resolved
    - > Collision is resolved found when key is found
      - > Somewhere within the cluster
    - > Collision is resolved not found when empty location is found
      - > Must traverse the entire cluster
  - Clearly as the clusters get longer we need more probes in both situations, but especially for not found
  - As α increases cluster sizes begin to approach M
  - Search times will now degrade from Theta(1) to Theta(M) == Theta(N)
    - > Note here that Theta(M) == Theta(N) in this case since our table can hold at most M keys

- How can we "fix" this problem?
  - Even AFTER a collision, we need to make all of the locations available to a key
  - This way, the probability from filled locations will be redistributed throughout the remaining empty locations in the table, rather than just being pushed down to the first empty location after the cluster
  - Ideal idea:
    > I have C locations filled and (M-C) locations empty

    > Instead of the insert probability of the C locations falling on just a few locations, we make it so that

    > P(insert at a filled location) == 0

    > P(insert at any empty location) == 1/M + (C/M)/(M-C)

    > > There is a probability of C/M that the hash value will be to a filled location

Improving Over Linear Probing

> We'd like that probability to be divided evenly amongst the (M-C) remaining open locations

> Now after a collision at index i, a key would still be equally likely to be inserted at any remaining open location

> We won't achieve this ideal but we can come close

– How to do this?

> How about making the increment 5 instead of 1?

> No help! Why?

> > We are still making clusters, they are just physically separated from each other

> > Any constant increment will have the same result