Recap

- Hashing
  - Hash function
  - Hash table
- Collision
  - Collision resolution
  - Linear probing

Considerations

- If the table is sparsely populated, searching is fast since we'd expect to perform one or two probes.
- If the table is nearly full, we will be spending most of our time resolving collisions. What is the worst case?
  - Probing for an open slot handles collisions, but won't help if we run out of slots.
- Collision tend to form groups of items
  - We call these groups **clusters**.
- Clusters tend to grow quickly. (Snowball effect)

Load Factor

- Load Factor = (Items in table)/(Size of the hash table)

- The current value of load factor affects the performance significantly
  - Let’s define a hit as finding an item.
  - Let’s define a miss as discovering that an item doesn’t exist.

Knuth's Formulas

- Show how probing is directly related to the load factor x, for a non-full table.

- Average number of probes for a hit: $\dfrac{1+\dfrac{1}{1-x}}{2}$

- Average number of probes for a miss: $\dfrac{1+\dfrac{1}{(1-x)^2}}{2}$

Knuth’s Formulas

Load Factor (%)	Probe hits	Probe misses
5	1.03	1.05
10	1.09	1.12
20	1.13	1.28
30	1.21	1.52
40	1.33	1.89
50	1.50	2.50
60	1.75	3.62
70	2.17	6.06
80	3.00	13.00
90	5.50	50.50
95	10.5	200.50

Other Probing Methods

- The fundamental problem with linear probing is that all of the probes trace the same sequence.

- Quadratic probing: 1, 4, 9, 16, 25, etc.
- Pseudo-random probing: Probe by a random value
  - Must use key as the **seed** to ensure repeatability
- Double hashing: Use another hash function to determine the probe sequence.
  - Hash function: P(K), primary hash gives starting point (index)
  - Probe function: S(K), second hash gives the stride (offset for subsequent probes)

Other Probing Methods

- The fundamental problem with linear probing is that all of the probes trace the same sequence.

- Quadratic probing: Probe $1, 4, 9, 16, 25$, etc.
- Pseudo-random probing: Probe by a random value
  - Must use key as the **seed** to ensure repeatability
- Double hashing: Use another hash function to determine the probe sequence.
  - Hash function: P(K), primary hash gives starting point (index)
  - Probe function: S(K), second hash gives the stride (offset for subsequent probes)

Other Probing Methods

- The fundamental problem with linear probing is that all of the probes trace the same sequence.

- Quadratic probing: Probe 1, 4, 9, 16, 25, etc.
- Pseudo-random probing: Probe by a random value
  - Must use key as the **seed** to ensure repeatability
- Double hashing: Use another hash function to determine the probe sequence.
  - Hash function: P(K), primary hash gives starting point (index)
  - Probe function: S(K), second hash gives the stride (offset for subsequent probes)

Other Probing Methods

- The fundamental problem with linear probing is that all of the probes trace the same sequence.

- Quadratic probing: Probe 1, 4, 9, 16, 25, etc.
- Pseudo-random probing: Probe by a random value
  - Must use key as the **seed** to ensure repeatability
- Double hashing: Use another hash function to determine the probe sequence.
  - Hash function: P(K), primary hash gives starting point (index)
  - Probe function: S(K), second hash gives the stride (offset for subsequent probes)

Double Hashing

- $\mathrm{P(k)}$ is the primary Hash function and is computed once for searches.

- $\mathrm{S(k)}$ is the Secondary Hash and is computed once only if there was a collision with $\mathrm{P(k)}$.

- First probe is just for the primary hash: $\mathrm{P(k)}$
- Second probe: $\mathrm{P(k) + S(k)}$
- Third probe: $\mathrm{P(k) + 2S(k)}$
- Fourth probe: $\mathrm{P(k) + 3S(k)}$, etc.

Examples

- Insert the following keys into a hash table of size 11, using $\mathrm{P(k)=k\%11}$
  - Linear probing
  - Quadratic probing
  - Double hashing $\mathrm{S(k)=k\%7+1}$

- 11,22,33,12,13,25,18

Performance of Double Hashing

- Average number of probes for a hit: $\frac{1}{x}\ln\left(\frac{1}{1-x}\right)$

- Average number of probes for a miss: $\frac{1}{1-x}$

- Recall the formulae in linear probing:

$$\frac{1+\frac{1}{1-x}}{2} \qquad \frac{1+\frac{1}{(1-x)^2}}{2}$$

Performance of Double Hashing

Load Factor (%)	Probe hits	Probe miss
5	1.03	1.05
10	1.05	1.11
20	1.12	1.25
30	1.19	1.43
40	1.28	1.67
50	1.39	2.00
60	1.53	2.50
70	1.72	3.33
80	2.01	5.00
90	2.56	10.00
95	3.15	20.00

Double Hashing

Probe hits	Probe misses
1.03	1.05
1.09	1.12
1.13	1.28
1.21	1.52
1.33	1.89
1.50	2.50
1.75	3.62
2.17	6.06
3.00	13.00
5.50	50.50
10.5	200.50

Linear Probing

[Full Table]

Linear v.s. Double Probing

- If the table is sparse (and memory is available), linear probing is very fast, however
  - Performance can degrade rapidly once clusters start forming.
- Double hashing uses memory more efficiently (smaller table or more full), costs a little more to compute secondary hash.
- For sparse tables, linear probing and double hashing require about the same number of probes, but double hashing will take more time since it must compute a second hash.
- For nearly full tables, double hashing is better than linear probing due to the less likelihood of collisions.

Expanding the Hash Table

- The performance of the hash table algorithms depend on the load factor of the table.

- Tables must not get full (or near full) or performance degrades.

- If we cannot determine the amount of data we expect, we may need to grow it at runtime.
  - This essentially means creating a new table and re-inserting all of the items.
  - Expanding the table is costly, but is done infrequently.
  - The cost is amortized over the run time of the algorithm

Deletion From Hash Table

Deleting items: Linear Probing Hash Table

- Insert(SPINAL)
  - $h(S_{19}) = 5$
  - $h(P_{16}) = 2$
  - $h(I_9) = 2$
  - $h(N_{14}) = 0$
  - $h(A_1) = 1$
  - $h(L_{12}) = 5$

N	A	P	I		S	L
0	1	2	3	4	5	6

delete(S)

N	A	P	I		S	L
0	1	2	3	4	5	6

N	A	P	I			L
0	1	2	3	4	5	6

find(L) = NOT FOUND???

- Deleting an item from a cluster presents a problem as the deleted item could be part of a linear probe sequence.

Handling Deletions: Solution #1

- Marking slots as deleted (MARK)
- Each slot can be in one of three states:
  - Occupied
  - Unoccupied
  - Deleted.
- Search until we find the item or encounter the first unoccupied slot.
  - Insert at first *deleted* or *unoccupied* slot
  - Need to remember where first deleted slot is when we insert an item
- Load factor is decreased when a slot is marked as deleted.

Handling Deletions: Solution #2

- Adjust the table (PACK) after a deletion.
- For each item after the deleted item that is in the cluster, mark its slot unoccupied and insert it back into the table.
- Works well for relatively sparse tables because the number of re-insertions is small.

Collision Resolution by Chaining

Collision Resolution by Chaining

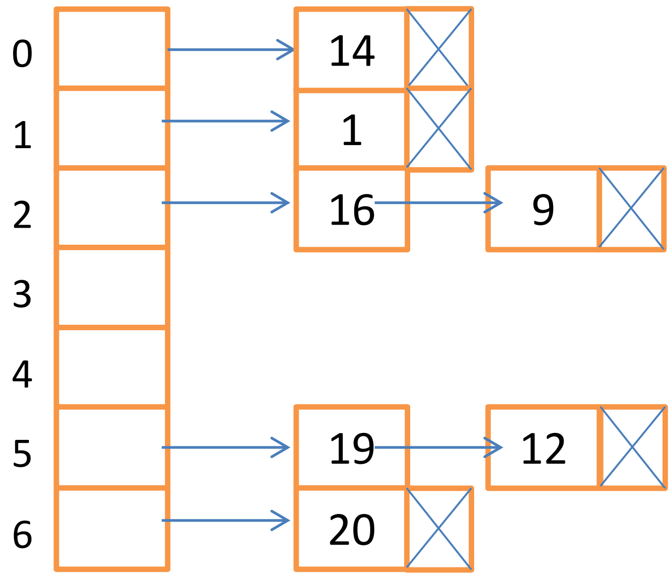
- With the open-addressing scheme, the data is stored in the hash table itself.

- In this scheme, the data is stored outside of the hash table.

- This method is called chaining (or separate chaining).
  - Instead of storing items in the hash table (in the slot indexed by the hashed key), we store them on a linked list.
  - The hash table simply contains pointers to the first item in each list.

Collision Resolution by Chaining

- Insert the following keys into the hash table: 19, 16, 9, 14, 1, 12, 20



Collision Resolution by Chaining

- Our data structure has been somewhat reduced to a singly linked list.

- Where do we insert into the list?
  - Front? Back? Middle?

- Should the list be sorted?

- Splay (caching) hash tables?

Considerations on Chaining

- We never run out of space (subject to the available memory).
- Implementing insert and delete is trivial compared to open-addressing above.
- Since we must ensure there are no duplicates, we must always look for an item before adding (inserting it).
  - Most time is spent searching through the linked lists.

Complexity of Chaining

- Recall the performance of linear probing:

$$\frac{1+\frac{1}{1-x}}{2} \qquad \frac{1+\frac{1}{(1-x)^2}}{2}$$

- With linear probing we minimize probing by keeping the hash table below 2/3 full.
  - An average of 2 probes for a successful search and 3 for an unsuccessful one (average cluster size of 3)

- Note that these are constants, not related to the number of elements in the table. $O(k)$

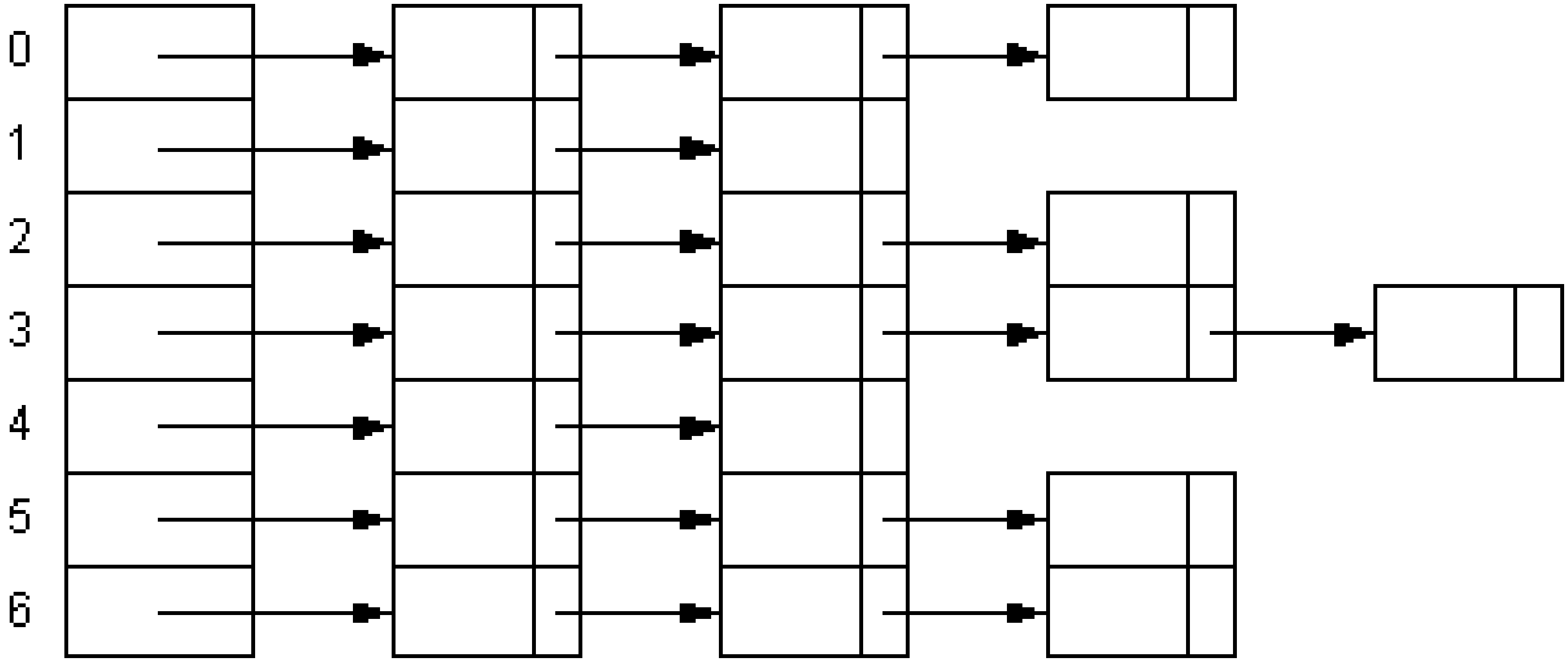
Complexity of Chaining

- There is no concept of “2/3” full.

- Load factor is still computed the same as before, but now it is likely to be greater than 1.

- Think of the load factor as being the average lengths of the lists

Complexity of Chaining



What is the load factor of this example?
What is the average number of nodes visited in a successful search?
What is the average number of nodes visited in an unsuccessful search?

Complexity of Chaining

- Complexity with a poor hash function?
  - O(N), Why/When?
- Complexity with a good hash function?
  - O(N/M), Depends on the load factor.
  - What makes a good hash function?

Advantages of Chaining

- Has the potential benefit that removing an item is trivial.
- Trivial to implement (linked list algorithms readily available).
- Node allocation can be expensive, but can be implemented efficiently with a memory manager(ObjectAllocator).
- Degrades gracefully as the average length of each lists grows. (No snowballing effect, i.e. clustering)
- Lists could be sorting using a BST or other data structure.

More Hashing

Hashing Strings

- Until now, all of our keys have been numeric. (integers)
- Often, we don't have a numeric key (or the key is a composite)
- Many algorithms exist for hashing non-numeric keys (transforming non-numeric data to numeric data).
  - Cyclic Redundancy Check (CRC) algorithms can hash entire files. (CRC Calculator) Run CRC on this data: 00110100111100101101110010100110
- Strings are widely used as keys (sometimes the key is the data itself)

Simple Naïve Hash Function

```
unsigned SimpleHash(const char *Key, unsigned TableSize){
  // Initial value of hash
  unsigned hash = 0;

  // Process each char in the string
  while (*Key){
    // Add in current char
    hash += *Key;
    // Next char
    ++Key;
  }

  // Modulo so hash is within the table
  return hash % TableSize;
}
```

Sample run with TableSize = 173

```
bat,138
cat,139
dat,140
pam,145
amp,145
map,145
tab,138
tac,139
tad,140
DigiPen,153
digipen,44
DIGIPEN,166
```

A Better Hash Function

```
int RSHash(const char *Key, int TableSize){
  int hash = 0;            // Initial value of hash
  int multiplier = 127; // Prevent anomalies

  // Process each char in the string
  while (*Key){
    // Adjust hash total
    hash = hash * multiplier;

    // Add in current char and mod result
    hash = (hash + *Key) % TableSize;

    // Next char
    ++Key;
  }
    // Hash is within 0 - (TableSize - 1)
  return hash;
}
```

Sample run with TableSize = 173

```
bat,93
cat,133
dat,0
pam,127
amp,16
map,10
tab,103
tac,104
tad,105
DigiPen,115
digipen,37
DIGIPEN,44
```

A more Complex Hash Function

```
int PJWHash(const char *Key, int TableSize){
    // Initial value of hash
  int hash = 0;
   // Process each char in the string
  while (*Key){
     // Shift hash left 4
    hash = (hash << 4);
     // Add in current char
    hash = hash + (*Key);
     // Get the four high-order bits
    int bits = hash & 0xF0000000;
    // If any of the four bits are non-zero,
    if (bits){
      // Shift the four bits right 24 positions
(...bbbb0000)
      // and XOR them back in to the hash
      hash = hash ^ (bits >> 24);
      // Now, XOR the four bits back in (sets them
all to 0)
      hash = hash ^ bits;
    }
    // Next char
    ++Key;
  }
    // Modulo so hash is within the table
  return hash % TableSize;
}
```

Invented by P.J. Weinberger

Sample run with
TableSize = 173

```
bat,114
cat,24
dat,107
pam,58
amp,46
map,158
tab,33
tac,34
tad,35
DigiPen,130
digipen,159
DIGIPEN,77
```

Pseudo Universal Hash Function

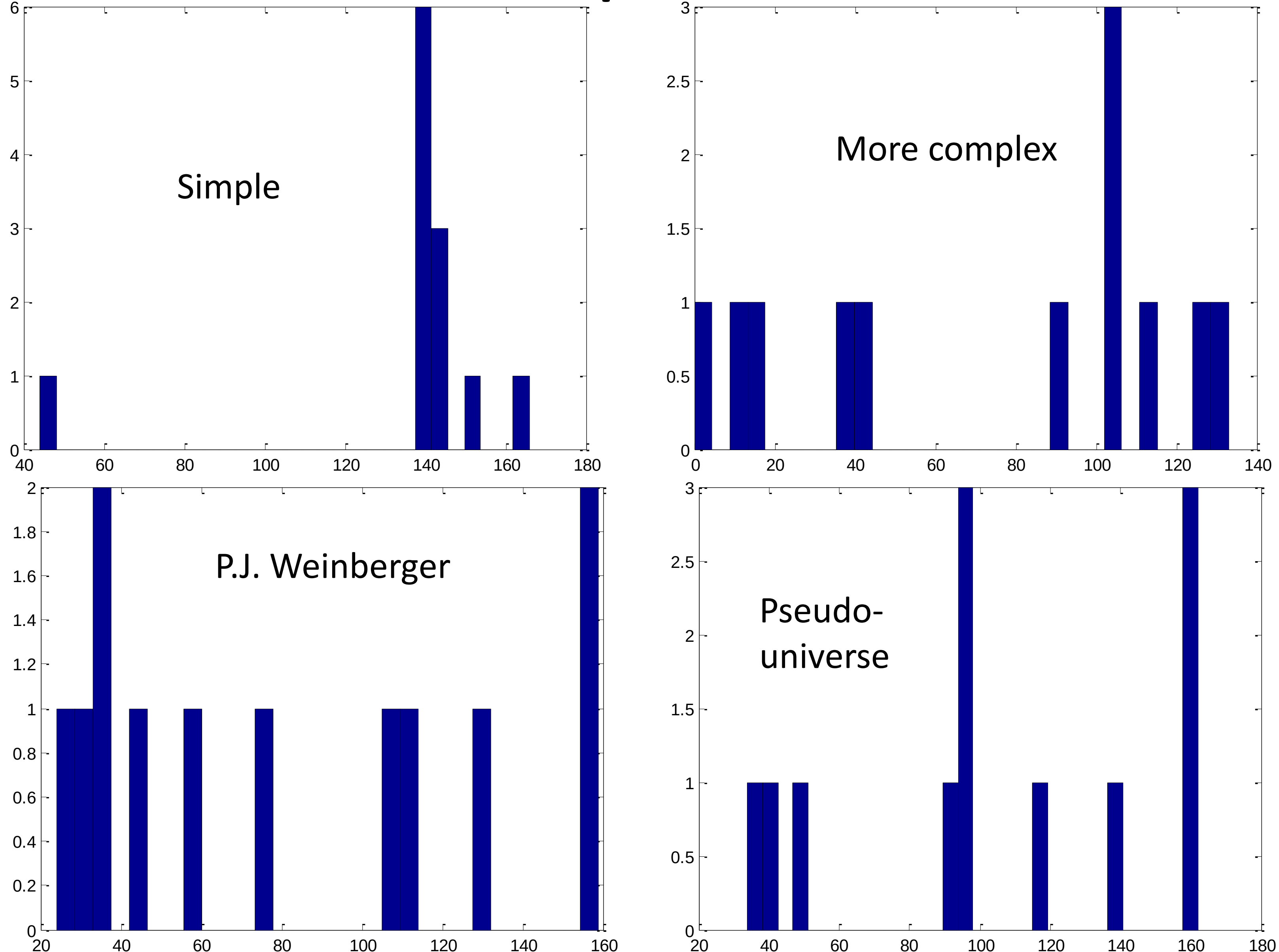
```
int UHash(const char *Key, int TableSize){
  int hash = 0;      // Initial value of hash
  int rand1 = 31415; // "Random" 1
  int rand2 = 27183; // "Random" 2
  // Process each char in string
  while (*Key){
    // Multiply hash by random
    hash = hash * rand1;
    // Add in current char, keep within TableSize
    hash = (hash + *Key) % TableSize;
    // Update rand1 for next "random" number
    rand1 = (rand1 * rand2) % (TableSize-1);
    // Next char
    ++Key;
  }
  // Account for possible negative values
  if (hash < 0)
    hash = hash + TableSize;

  // Hash value is within 0 - TableSize - 1
  return hash;
}
```

Sample run with TableSize = 173

```
bat,34
cat,42
dat,50
pam,139
amp,95
map,118
tab,160
tac,161
tad,162
DigiPen,92
digipen,97
DIGIPEN,96
```

Comparisons



Summary

- There are two parts to hash-based algorithms that implementations must deal with:
  - Computing the hash function to produce an index from a key.
  - Dealing with the inevitable collisions
- Hash tables rely on the fact that the data is uniformly and randomly distributed
  - Since we cannot control the data that is provided from the user, we must ensure that it is randomly distributed by hashing it.
- Hashing algorithms are used in other areas as well (e.g. cryptography)

Interesting Links

- Hash function performance and distribution
- Performance of various hash functions
- Various hash-related information
- More from Bob Jenkins
- Fowler / Noll / Vo (FNV) Hash
- GNU perfect hash function generator
- C Minimal Perfect Hashing Library