CS280 – Data Structures

Hashing

A simple example…

- Suppose we have an array of 10 integers (with values from 0 to 99):

Unsorted 10-element array (full)

Sorted 10-element array (full)

- Complexity of searching in each?
  - Pros/Cons?

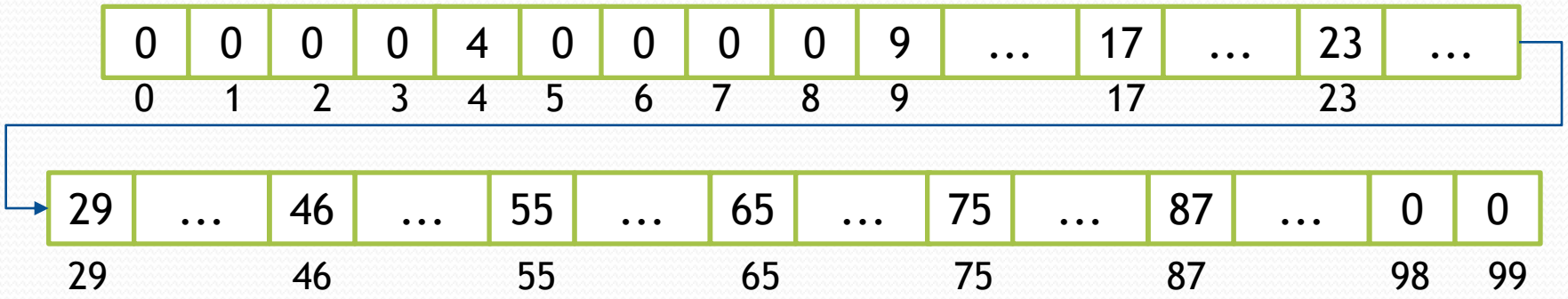
First attempt

Our Goal:

Given an item, we'd like to:

- *Perform some constant-time function, O(K).*
- *Locate the item in one step, O(1).*

- We stated that the values will be in the range of 0…99
- Simply Create an array of that size (100) and store the items in the array using their value as the index

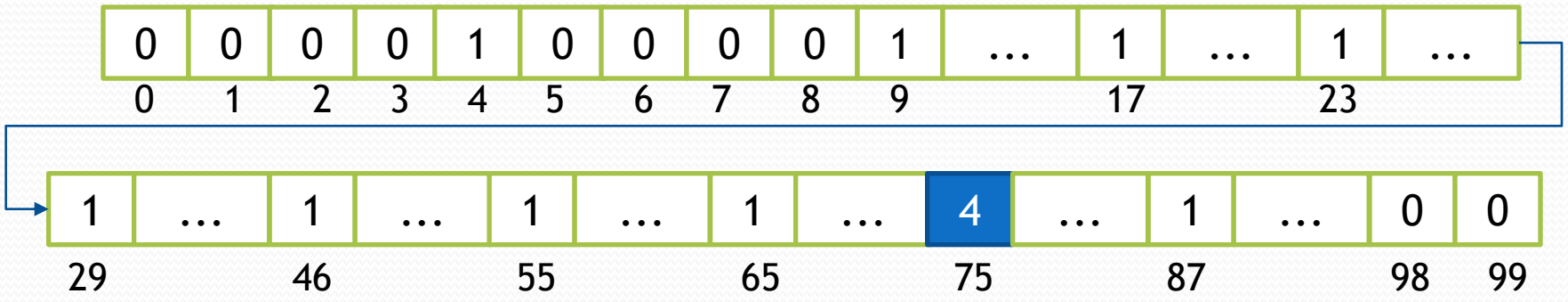


Towards Hash Table

- This is known as *direct addressing*
  - The value of elements is called the key. (Value is the index).
- We call this array a *hash table*.
- We call each element or cell a *slot*.
- Storing/locating an item in this way is a *key-based* algorithm
  - Given a key we can quickly find an item.

Dealing with duplicates

- Since we have unique data (0-99) we don't really have to store the values in the array .
- We simply mark that slot with 1, indicating the item's presence.
- Or, we can keep a count of how many instances of each value we have.



Going one step further

Given this *Student* structure:

```
struct Student
{
  string Name; // Maybe last and/or first
  long Year;   // 1 - 4
  float GPA;   // 0.00 - 4.00
  long ID;     // social security number, 000-00-0000 .. 999-99-9999
};
```

- Using the above techniques with this data we need a way to uniquely identify each Student.
- This means we need a ***key***.
- Typically, we use all or part of the data itself as a key.
- If some portion of the data is unique, it is a good candidate for a key

Picking the best candidate

- The *ID* appears to be the best candidate:
  - We could create an array that can contain all possible IDs (Social Security number for example).
  - This array would have to have at least 999,999,999 elements!!
  - Each element would be the size of a `Student` Struct
    - 20 bytes with a 8-character name.
  - $20\ bytes\ *\ 999{,}999{,}999 = 20{,}000\ GB$
  - Clearly, this won't work.

Hash Functions

- Hash functions are functions in the normal (mathematical) sense of the word.
  - I.e. given some input, produce one output.
- Essentially, a *hash function* is an operations that maps one (large) set of values into another (usually smaller) set of values.
- Therefore, a hash function is a *many-to-one* mapping. (Two different inputs can have the same output).

Hash functions

- More specifically, this function maps a value (key of *any* type) into an index(and integer) by performing arithmetic operations on the key.

- The resulting *hashed* integer indexes into a (*hash*) table. In this way, it is used as a way to lookup array elements quickly
  - *Arrays provide random access, aka constant time access*.

Back to our `Student` Example

- We decided to use the Social Security Number (ID) of each student because it is unique
- However we saw that the key was too large to use directly as an index
- We need a *hash function* to map that unique key into an index in the *hash table* (array).

$$H(K) = index$$

First attempt: SSHashFirst4

- *SSHashFirst4* is a hash function that simply extracts the first 4 digits of the ID:

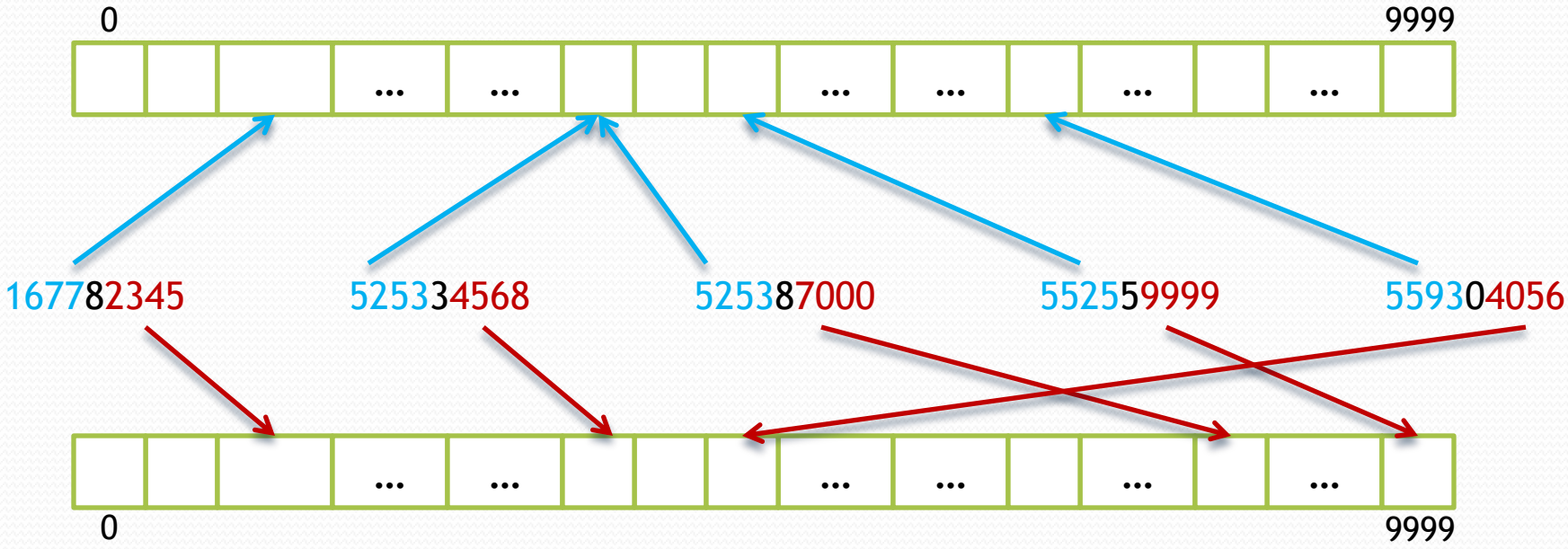
```
SSHashFirst4( 525334568 ) = 5253 !!!
SSHashFirst4( 559304056 ) = 5593
SSHashFirst4( 167782645 ) = 1677
SSHashFirst4( 525387000 ) = 5253 !!!
SSHashFirst4( 552559999 ) = 5525
```

Second attempt: SSLastFirst4

- *SSHashLast4* is a hash function that simply extracts the last 4 digits of the ID:

```
SSHashLast4( 525334568 ) = 4568
SSHashLast4( 559304056 ) = 4056
SSHashLast4( 167782645 ) = 2645
SSHashLast4( 525387000 ) = 7000
SSHashLast4( 552559999 ) = 9999
```

Graphically



Implementation Example

- These trivial hash functions could be implemented like this:

```
unsigned SSHashFirst4(unsigned SSNumber)
{
  return SSNumber / 100000;  // return first 4 digits
}

unsigned SSHashLast4(unsigned SSNumber)
{
  return SSNumber % 10000;  // return last 4 digits
}
```

Observations

- There are at least two (related) potential problems with `SSHashFirst4` function. What are they?
- The `SSHashLast4` suffers from the same problems but to a lesser degree.
- Advantage:
  - We no longer need an array that must be large enough to hold the entire range of values.
  - Our hash function is simple and fast (which is rare).

Collision

- Whenever two keys *hash* to the same index, then we have what we call a *collision*.
- We saw above that the `SSHashFirst4` leads to more collisions than `SSHashLast4`.
- However there is no guarantee that `SSHashLast4` won't result in collisions.
- There are techniques to deal with collisions. This is called *collision resolution*.

Collision Resolution

Collision Resolution

- However good our hash functions. Collisions are inevitable.
  - Simply because of the nature of the problem (*many-to-one mapping*).
- We will see different resolution *policies*:
  - *Linear probing*
  - *Chaining*

Linear Probing

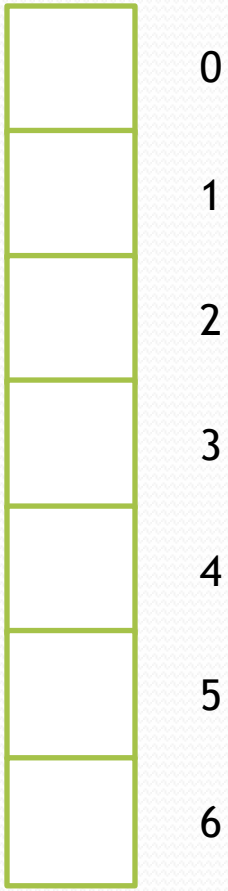
- The basic idea is that if a slot is already occupied, we simply move to the next unoccupied slot.
- The process of looking for unoccupied slots is called ***probing***. (anytime you search for a slot using the hashed value, it is considered ***probe***).
- This method of collision resolution is called *open-addressing*.
- Open addressing methods are used when the data itself is stored in the hash table.

Simple Example

- Using letters from A-Z as keys, a hash table of size 7.
- Each letter will be it's position in the alphabet
  - A = 1 (A_1), B = 2 (B_2), etc…
- Our hash function will be simple:

  $H(K) = K \,\%\, 7$ `(the key modulo 7)`

- So A_1 maps to 1, G_7 maps to 0, H_8 maps to 1, etc..



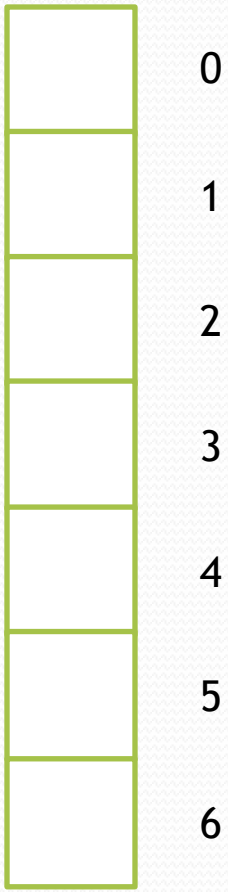
Example 1

- Inserting the following keys into our hash table:

$$S_{19}, P_{16}, I_9, N_{14}, A_1, L_{12}$$

- When hashed the keys will be mapped to:

```
H(S) = 5 (19 % 7)
H(P) = 2 (16 % 7)
H(I) = 2 ( 9 % 7)
H(N) = 0 (14 % 7)
H(A) = 1 ( 1 % 7)
H(L) = 5 (12 % 7)
```



Results – Example 1

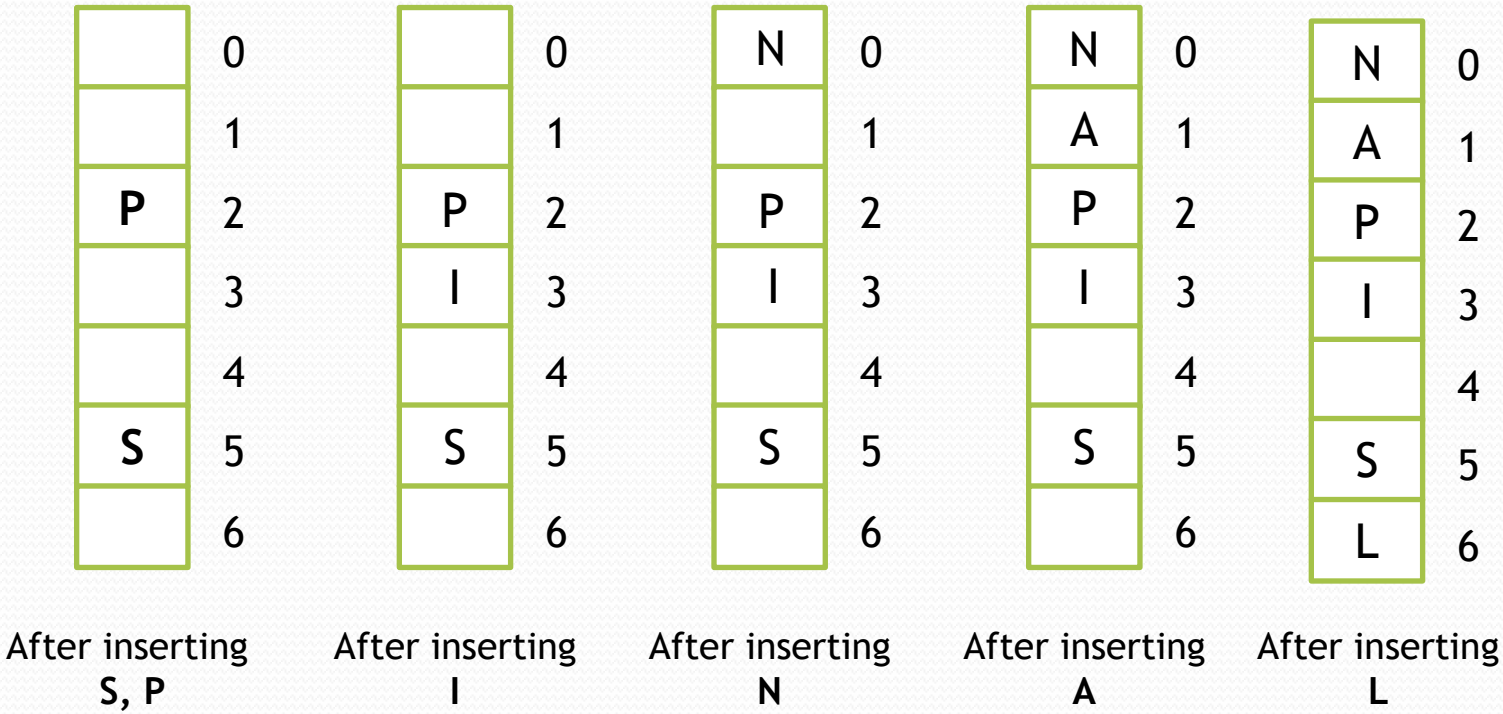
- Inserting the following keys into our hash table:

$$S_{19}, P_{16}, I_9, N_{14}, A_1, L_{12}$$

- Which Hash to the following values respectively:

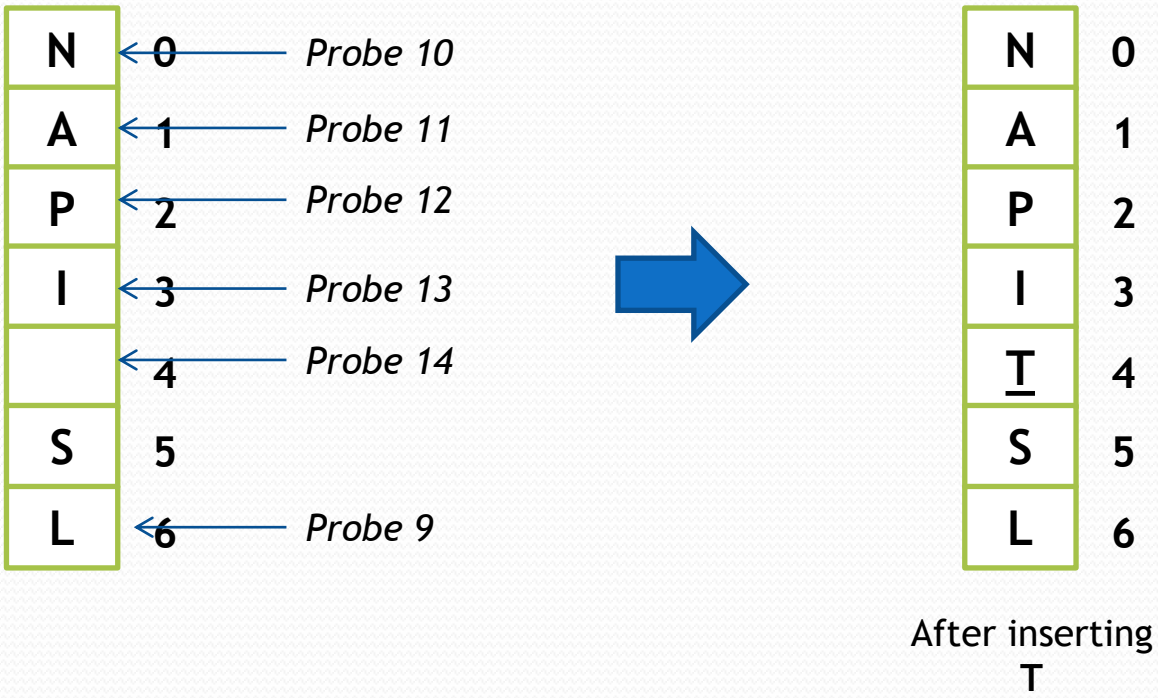
5, 2, 2, 0, 1, 5

- The total number of probes is 8



Results – Example 1

- Notice that inserting any letter *must* end up in slot 4.
- Inserting T_{20} H(T) = 6:



Results – Example 2

- Inserting the following keys into our hash table:

$$F_6, L_{12}, E_5, T_{20}, C_3, H_8$$

- Which Hash to the following values respectively:

6, 5, 5, 6, 3, 1

- The total number of probes is 11

E	0
T	1
H	2
C	3
	4
L	5
F	6

Considerations

- The direction of the linear probe is not important
  - Forward, or backward, as long as we are consistent
- We can't have duplicate keys, although two keys can hash to the same index.
- All the data is stored directly in the hash table (the array of slots).

Considerations (2)

- 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. **O(N)** 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)

Performance Characteristics

- We have seen that the performance of *linear probing* depends on the number of clusters in the table.
  - What are the chances of forming a cluster if there is one item in the table?
  - What are the chances of forming a cluster if there are two items in the table? Three? Four?

Load factor

- The number of items in the table divided by the size of the table is the *load factor*.

$$\frac{Items\ in\ table}{Size\ of\ the\ table} = Load\ Factor$$

- The load factor affects performance significantly.
  - Let's define a *hit* as finding an item.
  - Let's define a *miss* as discovering an item doesn't exist.

Knuth's formulas

- Knuth derived formulas show how probing is directly related to the load factor, where *x* is the load factor for a non-full table:

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

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

Load Factor (%)	Probe hits	Probe miss
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 insure 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

- P(K) is the Primary Hash function and is calculated once for searches.
- S(K) is the Secondary Hash and is calculated once only if there was a collision with P(K).
- First probe is just for the primary hash: P(K)
- Second probe: P(K) + S(K)
- Third probe: P(K) + 2S(K)
- Fourth probe: P(K) + 3S(K), etc…

Performance of double hashing

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

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

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

Expanding the Hash Table

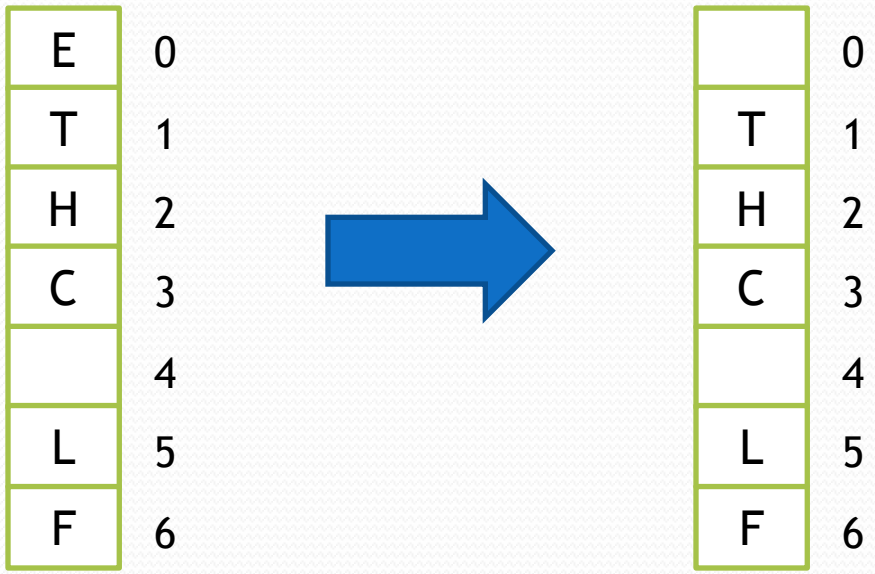
- The performance of the hash table algorithms depend on the *load factor* of the table
- Tables must not get full (or near full) or performance degrades
- If we cannot determine the amount of data we expect, we may need to grow it at runtime.
- This essentially means creating a new table and re-inserting all of the items.
- Expanding the table is costly, but is done infrequently.
- The cost is *amortized* over the run time of the algorithm
- This is the difference between average case and worst case .

Deleting items from a Linear Probing Hash Table

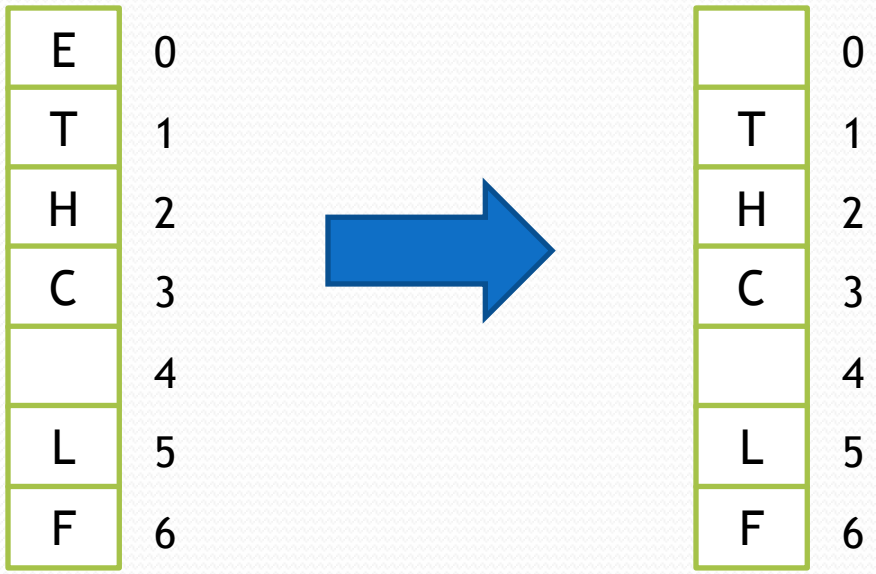
- Given the table containing the keys

$$F_6, L_{12}, E_5, T_{20}, C_3, H_8$$

- What happens to our search algorithm if we delete an item?



Deleting items from a Linear Probing Hash Table



- After deleting **E**, we search for the key **T**.
- Performing a linear probe from its hash value we arrive at an empty slot and conclude it isn't in the table.
- 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.
  - Our insertion/search algorithm would have to understand this as well
    - 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.
  - The clusters are very small (maybe 2 or 3 elements).

Collision Resolution Chaining

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

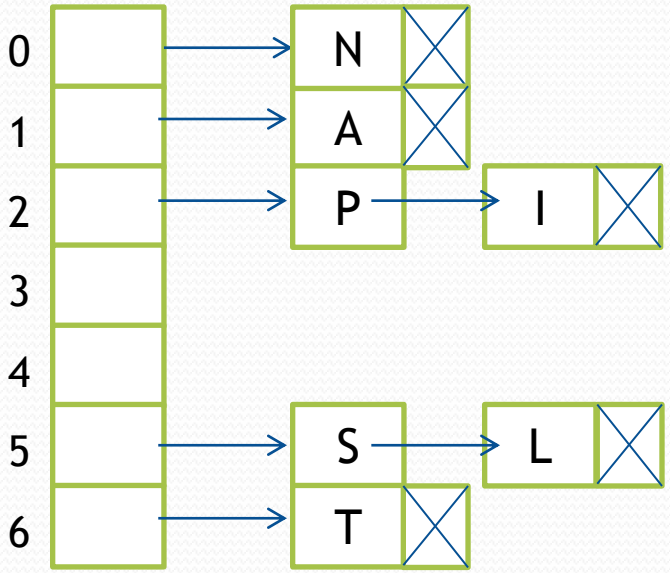
Collision Resolution Chaining

- Inserting the following keys into our hash table:

$$S_{19}, P_{16}, I_9, N_{14}, A_1, L_{12}$$

- When hashed the keys will be mapped to:

```
H(S) = 5 (19 % 7)
H(P) = 2 (16 % 7)
H(I) = 2 ( 9 % 7)
H(N) = 0 (14 % 7)
H(A) = 1 ( 1 % 7)
H(L) = 5 (12 % 7)
```



Collision Resolution Chaining

- The hash table now contains pointers (to nodes) instead of data.
- Our data structure has been somewhat reduced to a singly linked list
- Where do we insert into the list?
  - Front?
  - Back?
  - Middle?
- Is the list sorted?
- Splay (caching) hash tables?

Considerations on Chaining

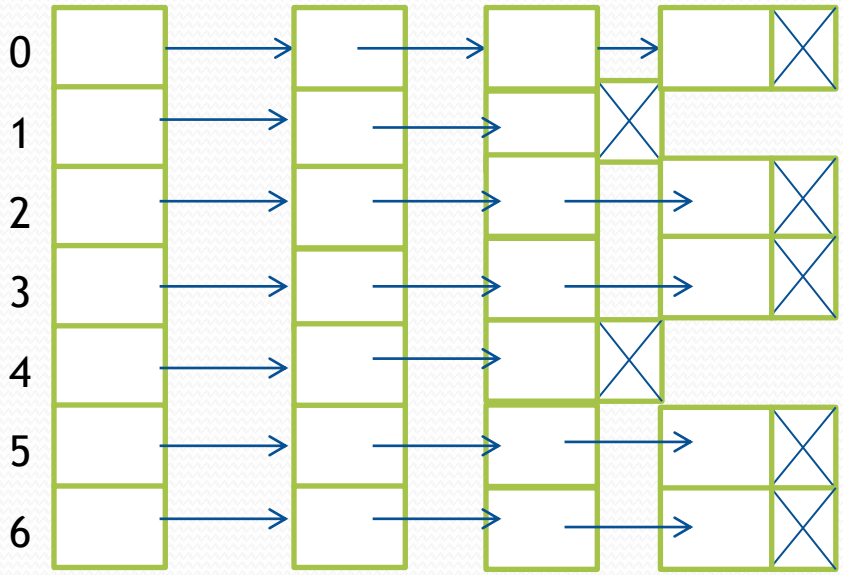
- We never run out of space
- More allocations required for nodes
- Implementing insert and delete is trivial compared to *open-addressing* above.
- Since we must insure there are no duplicates, we must always look for an item before adding (inserting it).
- Most time is spent searching through the linked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

Complexity of Chaining

- Recall the performance of linear probing:
  - With linear probing we minimize probing by keeping the has table around 2/3 full.
  - This gave us an average of two probes for a successful search and five 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

- With chaining:
  - There is no concept of “2/3” full.
  - Load factor is still calculated 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 (given a good hash function of course)



This example has a load Factor of 2.86 (N/M, 20/7)

Choosing the “Right” Hash Table

- Things to consider:
  - The data size is known (min, max)
  - The operations are known(insert, remove, search)
  - Performance matters more than memory(time/space trade-off)

Probing

- If the table is sparse (and memory is available), linear probing is very fast
- 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 has (stride).
- 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.
- Algorithms for inserting/deleting are peculiar.

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 (Not very useful in practice).

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).
- Strings are widely used as keys (sometimes the key is the data)

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;
}
```

bat, 100
cat, 101
dat, 102
pam, 107
amp, 107
map, 107
tab, 100
tac, 101
tad, 102
DigiPen, 39
digipen, 103
DIGIPEN, 90

Simple naïve hash function

- Simple, efficient, converts strings into integers
- Doesn't produce a good distribution

bat, 100
cat, 101
dat, 102
pam, 107
amp, 107
map, 107
tab, 100
tac, 101
tad, 102
DigiPen, 39
digipen, 103
DIGIPEN, 90

Program 14.1 in the Book

```
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;
}
```

bat, 27
cat, 120
dat, 2
pam, 56
amp, 188
map, 202
tab, 206
tac, 207
tad, 208
DigiPen, 162
digipen, 140
DIGIPEN, 198

Program 14.1 in the Book

- Produces a good distribution
- More complex, but not too bad
- Multiplier compensates for non-prime TableSize if table size if multiple/power of 2.

bat, 27
cat, 120
dat, 2
pam, 56
amp, 188
map, 202
tab, 206
tac, 207
tad, 208
DigiPen, 162
digipen, 140
DIGIPEN, 198

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;
}
```

bat, 170
cat, 4
dat, 49
pam, 160
amp, 102
map, 28
tab, 118
tac, 119
tad, 120
DigiPen, 12
digipen, 154
DIGIPEN, 91

A more complex hash function

- Produces a good distribution
- More complex, but not too bad

bat, 170
cat, 4
dat, 49
pam, 160
amp, 102
map, 28
tab, 118
tac, 119
tad, 120
DigiPen, 12
digipen, 154
DIGIPEN, 91

“Pseudo Universal“ Program 14.2 in the Book

```
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;
}
```

bat, 163
cat, 162
dat, 161
pam, 142
amp, 67
map, 148
tab, 127
tac, 128
tad, 129
DigiPen, 142
digipen, 64
DIGIPEN, 96

“Pseudo Universal“ Program 14.2 in the Book

- Produces a good distribution
- More complex, but not too bad
- Multiplier compensates for non-prime TableSize if table size if multiple/power of 2.

bat, 163
cat, 162
dat, 161
pam, 142
amp, 67
map, 148
tab, 127
tac, 128
tad, 129
DigiPen, 142
digipen, 64
DIGIPEN, 96

Comparisons (Table Size is 211)

Simple Hash	RSHash	PJWHash	UHash
bat, 100	bat, 27	bat, 170	bat, 163
cat, 101	cat, 120	cat, 4	cat, 162
dat, 102	dat, 2	dat, 49	dat, 161
pam, 107	pam, 56	pam, 160	pam, 142
amp, 107	amp, 188	amp, 102	amp, 67
map, 107	map, 202	map, 28	map, 148
tab, 100	tab, 206	tab, 118	tab, 127
tac, 101	tac, 207	tac, 119	tac, 128
tad, 102	tad, 208	tad, 120	tad, 129
DigiPen, 39	DigiPen, 162	DigiPen, 12	DigiPen, 142
digipen, 103	digipen, 140	digipen, 154	digipen, 64
DIGIPEN, 90	DIGIPEN, 198	DIGIPEN, 91	DIGIPEN, 96

Summary

- There are two parts to hash-based algorithms that implementations must deal with:
  - Computing the has 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.
- Hash tables whose size is a prime number help to insure good distribution. Also table sizes that are power of 2 work well if the stride is an odd number.
- Since the data is not sorted in any meaningful way, operations such as displaying all items in a sorted fashion are expensive. (Use another data structure for that operation).
- Hashing algorithms are used in other areas as well (e.g. cryptography)