CS280 – Data Structures

Hashing

Overview

- Introduction
- Time vs. space trade-off
- Hash functions
- Collision resolution
  - Linear probing
  - Chaining
- More hashing
- Summary

Introduction

- Most of our algorithms deal with
  - Inserting
  - Deleting
  - Searching

- The most fundamental operation is searching because most other operations depend on it as well.
  - Complexity in Searching in Arrays:
  - Complexity in Searching in Trees:
  - Complexity in Searching in Lists:

Introduction

- Most of our algorithms deal with
  - Inserting
  - Deleting
  - Searching

- The most fundamental operation is searching because most other operations depend on it as well.
  - Complexity in Searching in Arrays: O(N) and O(log N).
  - Complexity in Searching in Trees: O(N) and O(log N).
  - Complexity in Searching in Lists: O(N).

Introduction

- Is this the best we can do?
  - YES, if we have to use a comparison function to identify an item

- What we want is to find an item **without** having to compare it to other items.

- Given an item, we'd like to:
  - Perform some constant-time function, **O(K).**
  - Locate the item in one step, **O(1).**

- **That is an overall complexity of O(k), which is VERY GOOD.**

A Simple Example…

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

23	4	17	46	29	87	75	9	65	55

Unsorted 10-element array (full)

4	9	17	23	29	46	55	65	75	87

Sorted 10-element array (full)

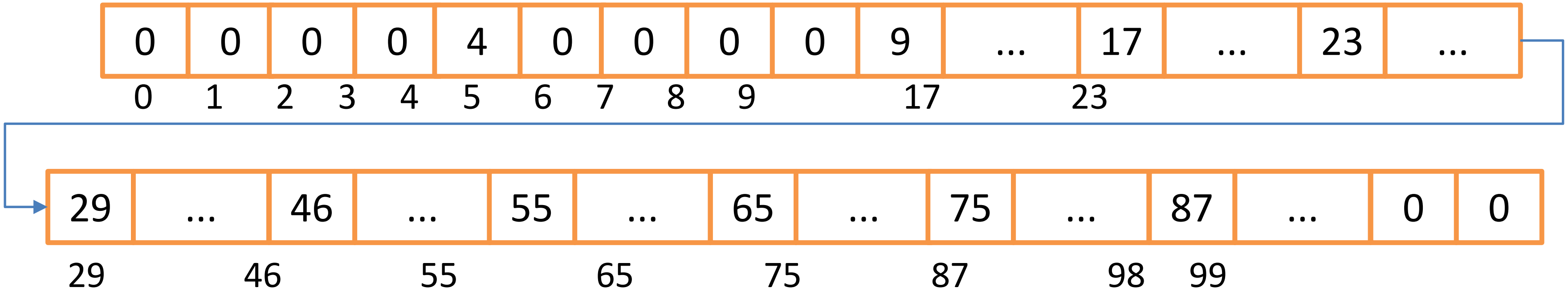
- Complexity of searching in each?

  Pros/Cons?

First Attempt

- Our objective: given an item, we’d like to:
  - Perform some constant-time function, O(K).
  - Locate the item in one step, O(1).

- Since we stated that the values will be in the range of 0…99, we can simply create an array of that size (100) and store the items in the array using their value as the index



0 0 0 0 4 0 0 0 0 9 … 17 … 23 …

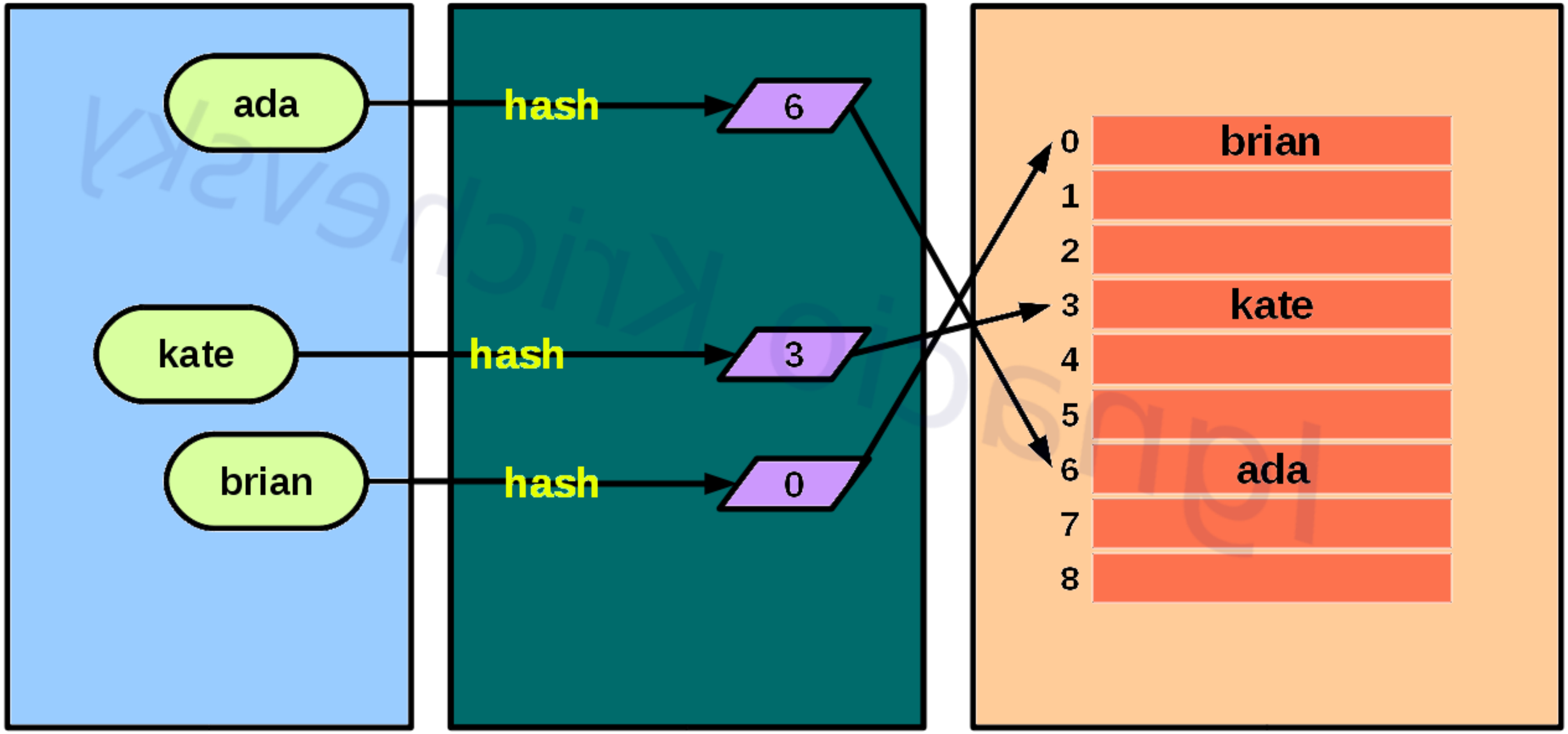
0 1 2 3 4 5 6 7 8 9 17 23

29 … 46 … 55 … 65 … 75 … 87 … 0 0

29 46 55 65 75 87 98 99

Towards Hash Table

- This is known as direct addressing.
  - Using the value as the index.
  - The value of elements is called the key.
- 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.



ada → hash → 6

kate → hash → 3

brian → hash → 0

0	brian
1	
2	
3	kate
4	
5	
6	ada
7	
8	

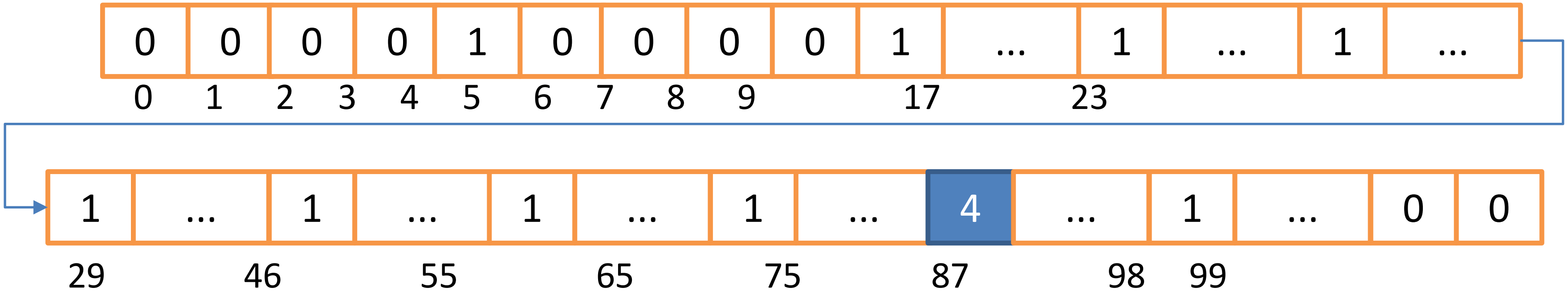
elements

hash function

hash table

Towards Hash Table

- In this example, since we have unique data (0-99) we don’t really have to store the values in the array.
  - It is implied by the index.
- We can keep a count of how many instances of each value we have.



Considerations

- What is the complexity of this approach?
- What are the pros?
- What are the cons?
- What are the limitations?
  - What if we were storing social security numbers?

The Time vs. Space Trade-off

- The time vs. Space trade-off remains the major dilemma when dealing with algorithm and data structures.

- Ultimately, we need to strike a balance between:
  - How much memory are we allowed to use?
  - How much time does each operation take?

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

Going One Step Further

- Let U denote the universal set, the range of values.
- Which one of the data members will be a good candidate?

```
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
};
```

- Name – Character array uniqueness depends on size (**U** can be quite large).
- Year – Small integer, unlikely to be unique (**U** is very small).
- GPA – Floating point, unlikely to be unique (**U** is small).
- ID – Relatively large integer, guaranteed to be unique (**U** is very large)

Picking the Best Candidate

- The ID appears to be the best candidate, why?
  - We could create an array that can contain all possible IDs (social security numbers).

- This array would have to have at least 999,999,999 elements!!
  - Each 9-digit number would be an index into this (very) large array.

- Each element would be the size of a `Student` struct.
  - 20 bytes with 8-character name
  - 20 bytes $\times 1{,}000{,}000{,}000 = 20\ \mathrm{GB}$

Using a Portion of the ID

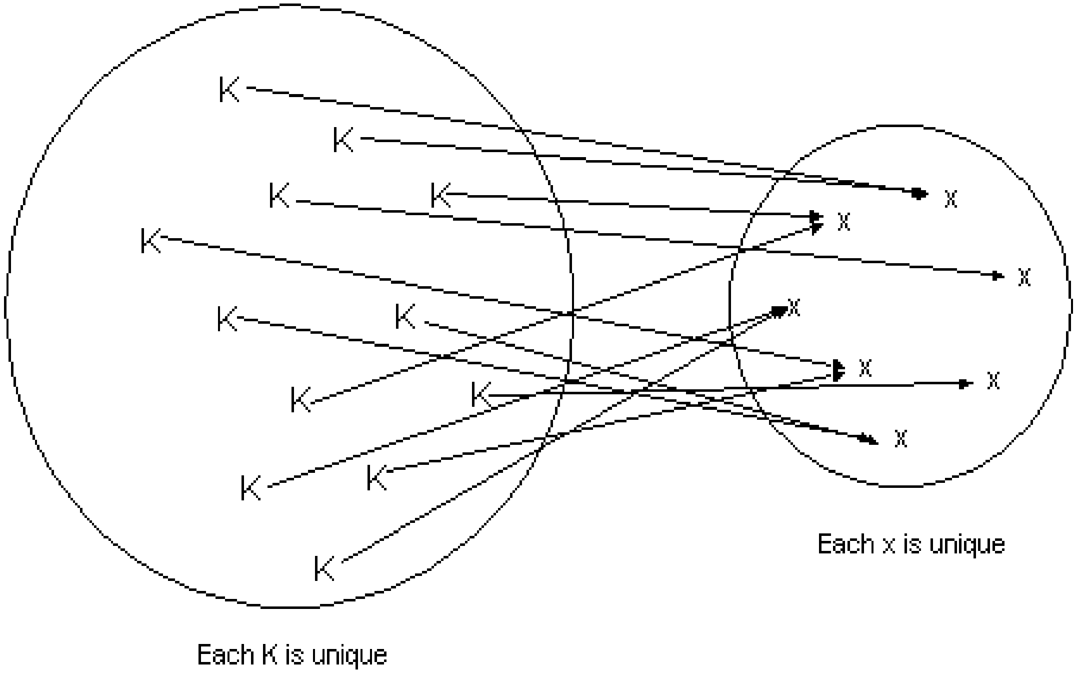
- We may choose to use just a “few” of the digits.
- Which digits? (e.g. first 3, first 4, last 4, middle 5)
- What are the limitations of these choices?
- Can we overcome these limitations?

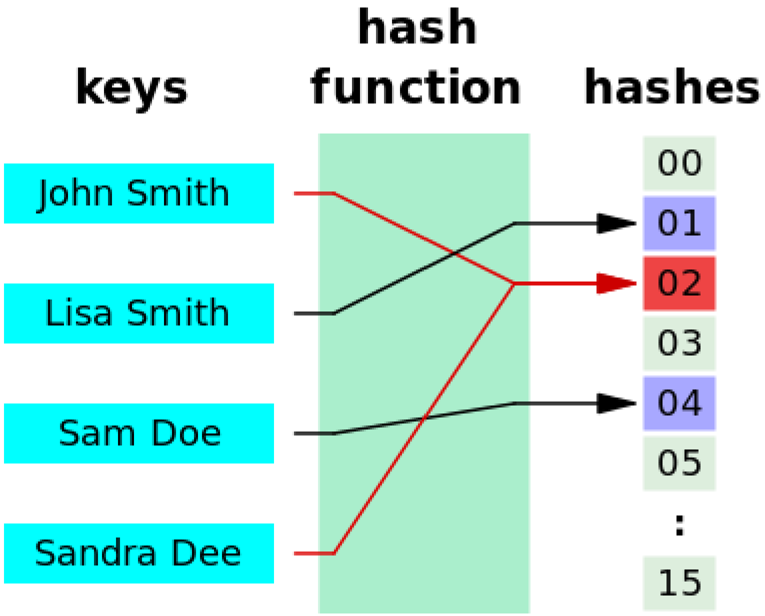
Hash Functions

- **Hash functions** are functions in the normal (mathematical) sense of the word.
  - i.e. given some input, produce one output.
  - A hash function is an operation that maps one (large) set of values into another (usually smaller) set of values.

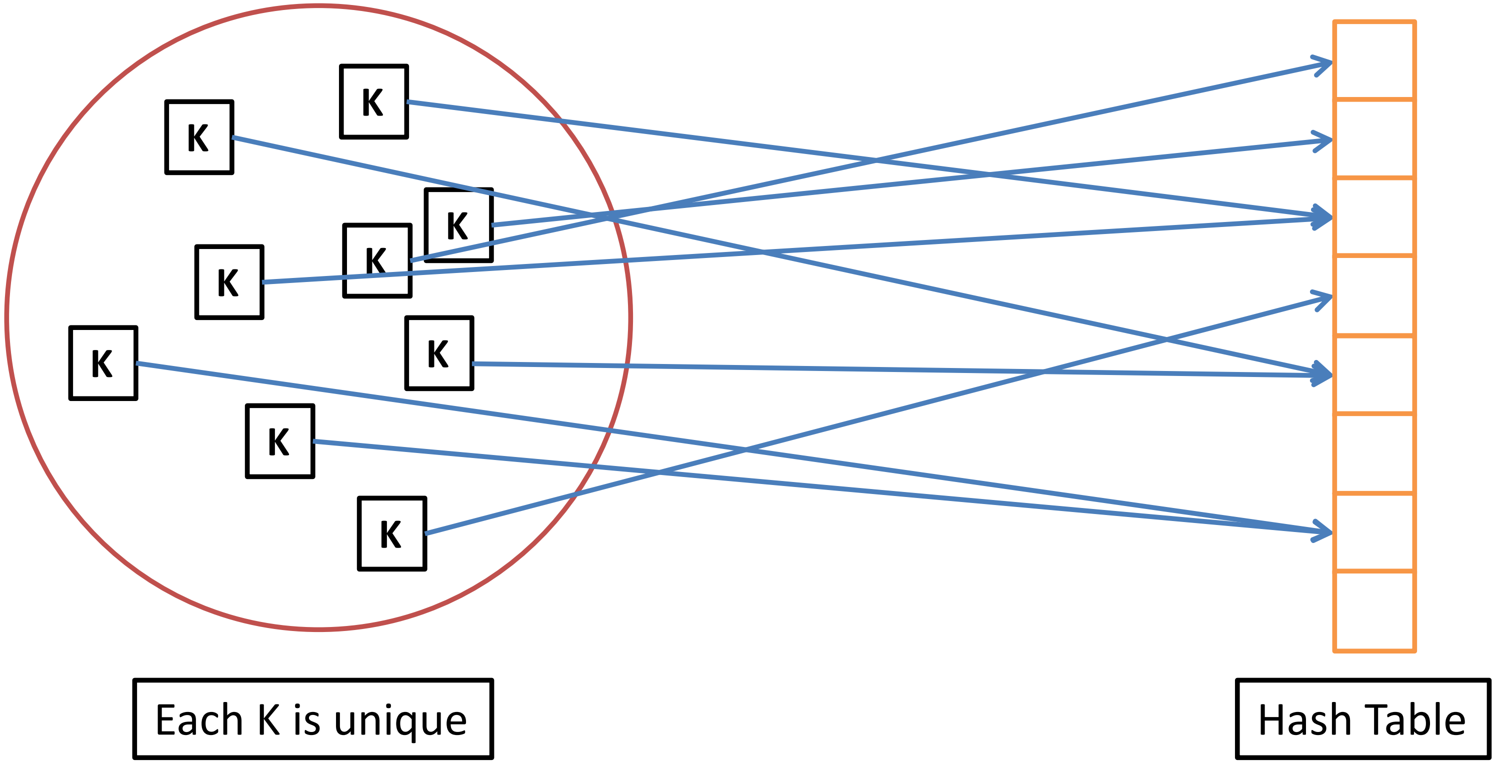
- Essentially, a hash function is an operations that maps one (large) set of values into another (usually smaller) set of values.
  - Therefore, a hash function is a many-to-one mapping. (Two different inputs can have the same output).

Hash Functions





Mapping into a Hash Table



Hash Functions

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

- The resulting hashed integer indexes into a (hash) table.
  - Arrays provide random access, aka constant time access.

Back to Our `Student` Example

- We decided to use the ID of each student as the key since it is unique.

- However, we saw that the key was too large to use directly as in index.

- We need a hash function to map that unique key into an index in the hash table (array) of manageable size. $\mathrm{H(K) = index}$

First Attempt: SSHashFirst4

- `SSHashFirst4` is a hash function that simply extracts the first 4 digits of the ID.

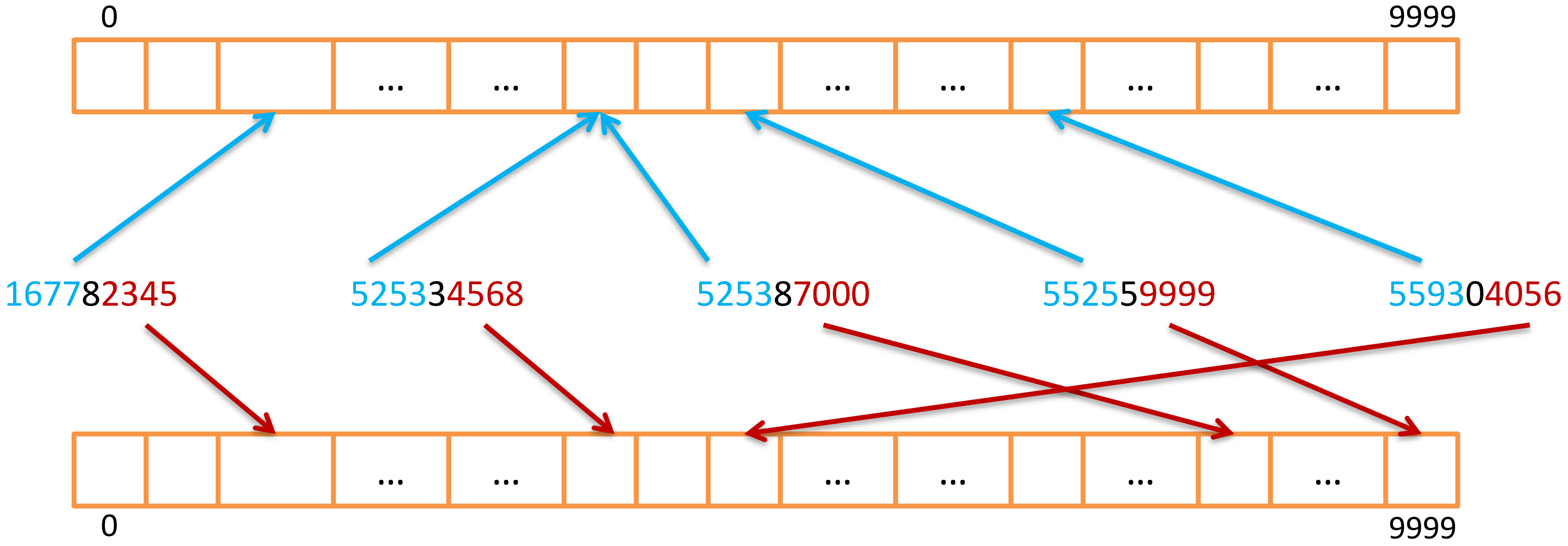
- `SSHashFirst4( 525334568 ) =` **5253**
- `SSHashFirst4( 559304056 ) = 5593`
- `SSHashFirst4( 167782645 ) = 1677`
- `SSHashFirst4( 525387000 ) =` **5253**
- `SSHashFirst4( 552559999 ) = 5525`

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



167782345

525334568

525387000

552559999

559304056

Implementation

- These trivial hash functions could be implemented like this:

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

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

Implementation

- 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 is a potential problem with `SSHashFirst4` function.
- The `SSHashLast4` suffers from the same problems but to a lesser degree.

- Advantages:
  - 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.

Observations

- There is a potential problem with `SSHashFirst4` function.
  - Different SSN (input key) may result in the same index.
- The `SSHashLast4` suffers from the same problems but to a lesser degree.

- Advantages:
  - We no longer need an array that must be large enough to hold <u>the entire range of values</u>.
  - Our hash function is simple and fast.

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

Hash-based Algorithms

- Note: The keys (input) used in a hash table must be unique. However, the resulting hashed values (output) do not have to be unique (and often are not).

- There are two parts to hash-based algorithms
  - Compute the hash function to produce an index from a key
  - Deals with the inevitable collisions

Collision Resolution by Probing

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

- Hash table of size 7
- Hash function: $\mathrm{H(k)=k\%7}$
- Let’s insert the following keys into the hash table: $19, 16, 9, 14, 1, 12$

Some Questions

- How many probes would be performed?
- What happens when we try to lookup (find) 9? How about 2?
- What if we inserted 20 into the table?
- What happens when we now try to lookup 2?
- **Another example:** Using linear probing to resolve collisions, what would the hash table look like during and after the insertions of the keys: 6,12,5,20,3,8
  - How many probes would be performed?

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).
- Load Factor