Hashing

Portions © S. Hranilovic, S. Dumitrescu and R. Tharmarasa

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

- Chapter 20 - all except for section 20.4.1

Hashing

- Fast searching and insertion are typically conflicting operations.
  - → Unsorted list - Insertion/Deletion, $\Theta(1)$, Searching, $\Theta(n)$ (average and worst case)
  - → Sorted list - Insertion/Deletion, $\Theta(n)$, Searching, $\Theta(\log n)$ (average and worst case)
  - → Binary Search Tree - Insertion/Deletion, $\Theta(\log n)$, Searching, $O(\log n)$ (average case)

Hashing

- Fast searching and insertion are typically conflicting operations.
  - → Unsorted list - Insertion/Deletion, $\Theta(1)$, Searching, $\Theta(n)$ (average and worst case)
  - → Sorted list - Insertion/Deletion, $\Theta(n)$, Searching, $\Theta(\log n)$ (average and worst case)
  - → Binary Search Tree - Insertion/Deletion, $\Theta(\log n)$, Searching, $O(\log n)$ (average case)
- Insertions are done quickly when they are done at a fixed place.
- Searches can occur quickly when there is **structure** or **ordering** to the items.

Hashing

- Fast searching and insertion are typically conflicting operations.
  - → Unsorted list - Insertion/Deletion, $\Theta(1)$, Searching, $\Theta(n)$ (average and worst case)
  - → Sorted list - Insertion/Deletion, $\Theta(n)$, Searching, $\Theta(\log n)$ (average and worst case)
  - → Binary Search Tree - Insertion/Deletion, $\Theta(\log n)$, Searching, $O(\log n)$ (average case)
- Insertions are done quickly when they are done at a fixed place.
- Searches can occur quickly when there is **structure** or **ordering** to the items.

- Consider the **Set ADT** ...
  - Array approach
  - Linked-list approach

McMaster University

Problem

- Say we want to represent a set of integers with values in the range 0 to 65535 (E.g.: $\{7, 100, 201\}$)
- We need **insertions, deletions and searches** to be very fast (constant time).

Problem

- Say we want to represent a set of integers with values in the range 0 to 65535 (E.g.: $\{7, 100, 201\}$)
- We need **insertions, deletions and searches** to be very fast (constant time).

Solution

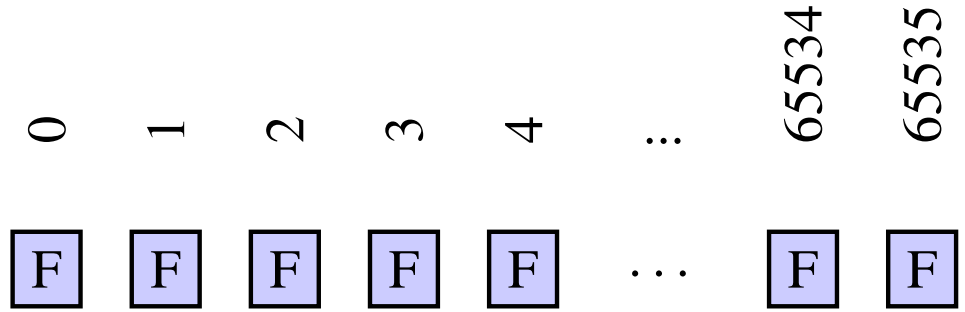
- Represent set using a **bit vector**
  - In an array with 2^{16} `boolean`'s, if i is in set put a `TRUE` at index i.
  - To represent the empty set, set all array elements to `FALSE`.
  - To insert some value i, set the array element at index i to `TRUE` .
- Array element access is very fast if we know the index.

Set ADT

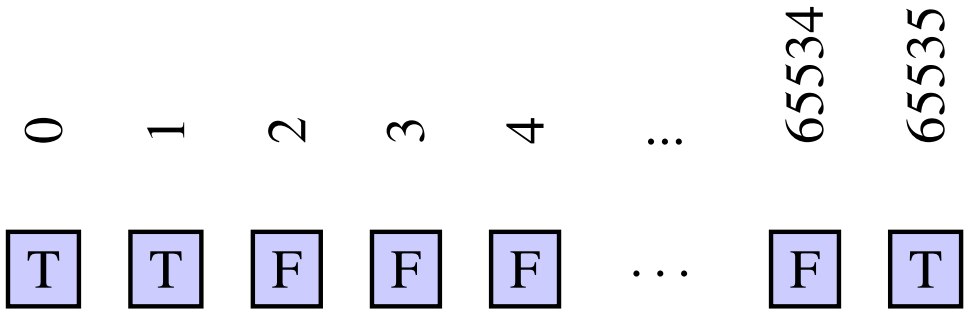
Say you want to represent a set of integers over the range 0 to 65535 using a **bit vector**.

→ In an array with 2^{16} `boolean`'s, if i is in set put a 'T' at index i.

Example: Empty Set



Example: $\{0, 1, 65535\}$

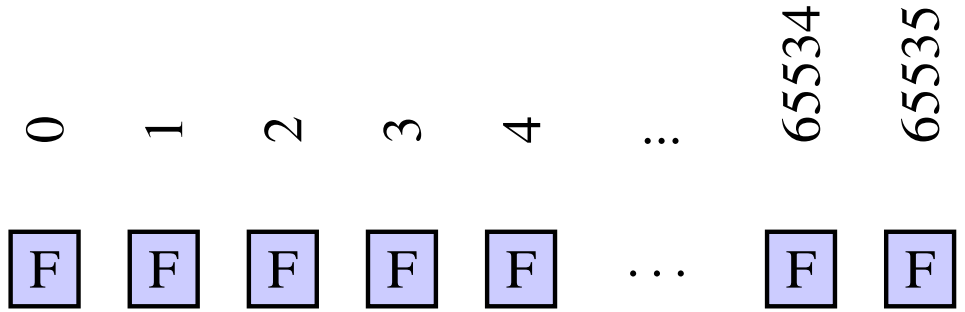


Set ADT

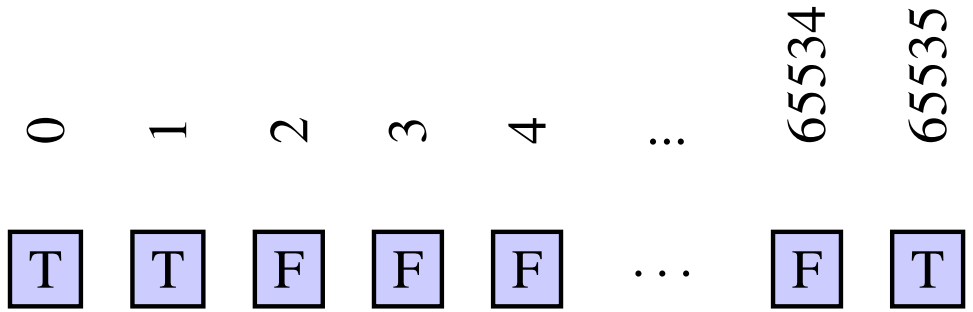
Say you want to represent a set of integers over the range 0 to 65535 using a **bit vector**.

→ In an array with 2^{16} `boolean`'s, if i is in set put a 'T' at index i.

Example: Empty Set



Example: $\{0, 1, 65535\}$



☞ This technique is very inefficient in terms of memory required, especially for small sets

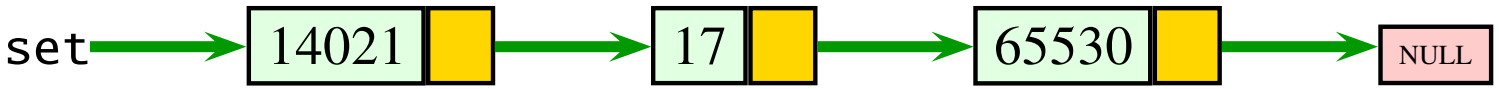
☞ However, searching, insert and delete can be done in $\Theta(1)$ time

Represent the same set using a linked list

→ For each element added to the list, verify that it is not already in the list then add it to the list.

Example: Empty Set

Example: $\{17, 14021, 65530\}$ (Notice order is not important)

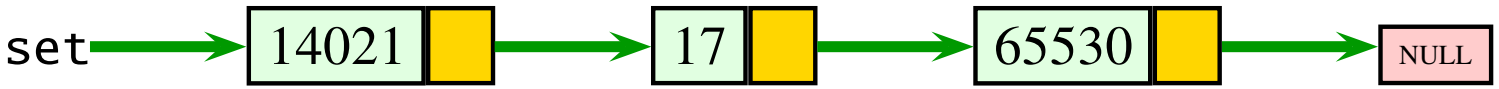


Represent the same set using a linked list

→ For each element added to the list, verify that it is not already in the list then add it to the list.

Example: Empty Set

Example: $\{17, 14021, 65530\}$ (Notice order is not important)



☞ This technique is more efficient in terms of memory required, especially for small sets

☞ However, searching takes $\Theta(n)$ time.

Set ADT Comparison

- The bit vector approach sacrifices memory efficiency for search speed
- The linked list approach is more efficient on space but slower

Set ADT Comparison

- The bit vector approach sacrifices memory efficiency for search speed
- The linked list approach is more efficient on space but slower

Tradeoff

Would like to devise a method to combine the two approaches more optimally:

- Waste slightly more time over array approach
- Waste slightly more space over linked list approach

Set ADT Comparison

- The bit vector approach sacrifices memory efficiency for search speed
- The linked list approach is more efficient on space but slower

Tradeoff

Would like to devise a method to combine the two approaches more optimally:

- Waste slightly more time over array approach
- Waste slightly more space over linked list approach

Hashing is a technique which allows for this tradeoff

Hashing – Definitions

- **Hashing** allows for searching by **direct** access based on the key value
  - → The **key** is the value that we are searching for in the data structure
- Define a **hash table** as an array of M elements labelled $0, 1, 2, \ldots, (M-1)$.
- A **hash function**, $h(x)$, *maps* every possible key, x, to an index into the hash table.
  - → Each key is mapped to the index given by the hash function
  - → The value of the hash function tells where to **start** searching for the value
  - → The hash function should be easy to compute and should try to distribute the keys as evenly as possible in the hash table.

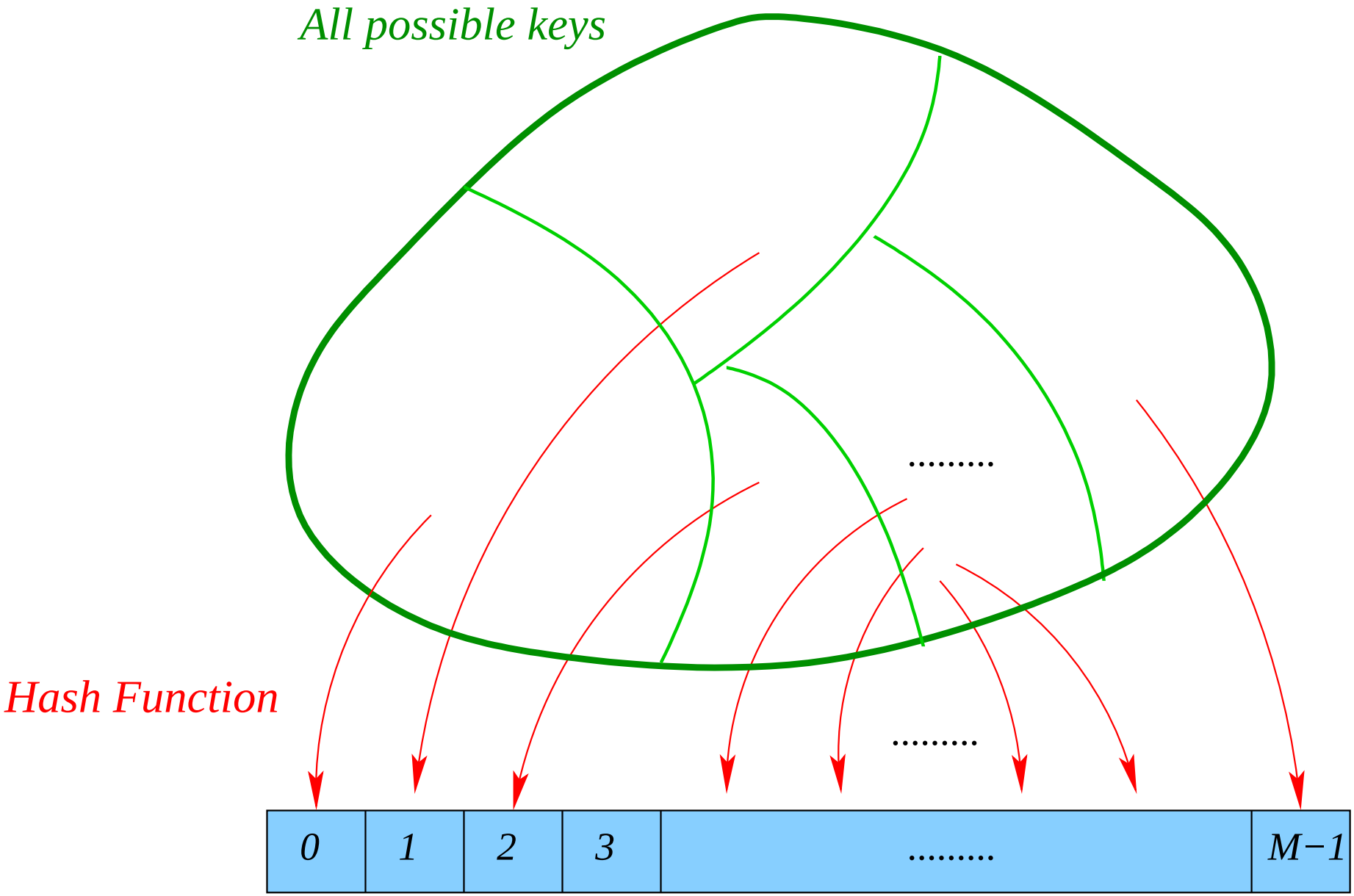
Hashing – Definitions

- **Hashing** allows for searching by **direct** access based on the key value
  - → The **key** is the value that we are searching for in the data structure
- Define a **hash table** as an array of M elements labelled $0, 1, 2, \ldots, (M-1)$.
- A **hash function**, $h(x)$, *maps* every possible key, x, to an index into the hash table.
  - → Each key is mapped to the index given by the hash function
  - → The value of the hash function tells where to **start** searching for the value
  - → The hash function should be easy to compute and should try to distribute the keys as evenly as possible in the hash table.

Example: The bit vector implementation of the Set ADT is really a trivial hash function in which

$$h(x) = x$$

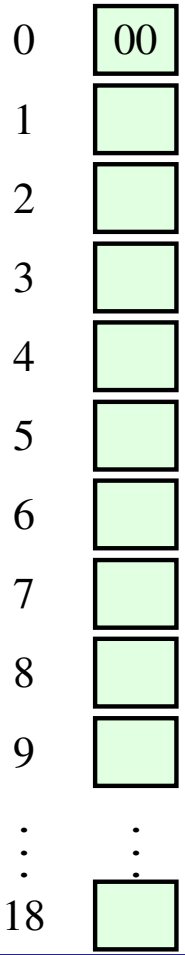
Hashing – Definitions



Hashing – Examples

- Consider the set of two digit integers, $s_1 s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.
- Let $h(s_1 s_2) = s_1 + s_2$. The 100 possible keys are mapped to a hash table of 18 elements.

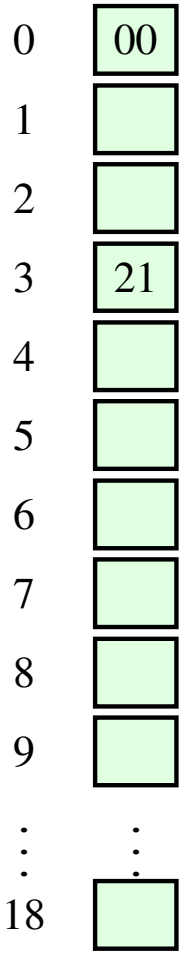
Insert: $00 \rightarrow h(00) = 0$



Hashing – Examples

- Consider the set of two digit integers, $s_1 s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.
- Let $h(s_1 s_2) = s_1 + s_2$. The 100 possible keys are mapped to a hash table of 18 elements.

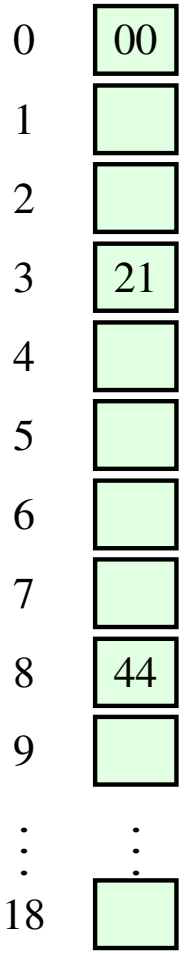
Insert: $21 \rightarrow h(21) = 3$



Hashing – Examples

- Consider the set of two digit integers, $s_1 s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.
- Let $h(s_1 s_2) = s_1 + s_2$. The 100 possible keys are mapped to a hash table of 18 elements.

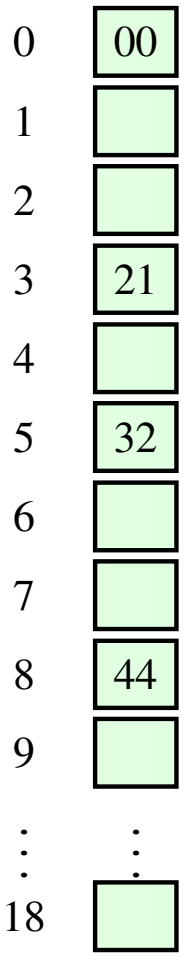
Insert: $44 \rightarrow h(44) = 8$



Hashing – Examples

- Consider the set of two digit integers, $s_1 s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.
- Let $h(s_1 s_2) = s_1 + s_2$. The 100 possible keys are mapped to a hash table of 18 elements.

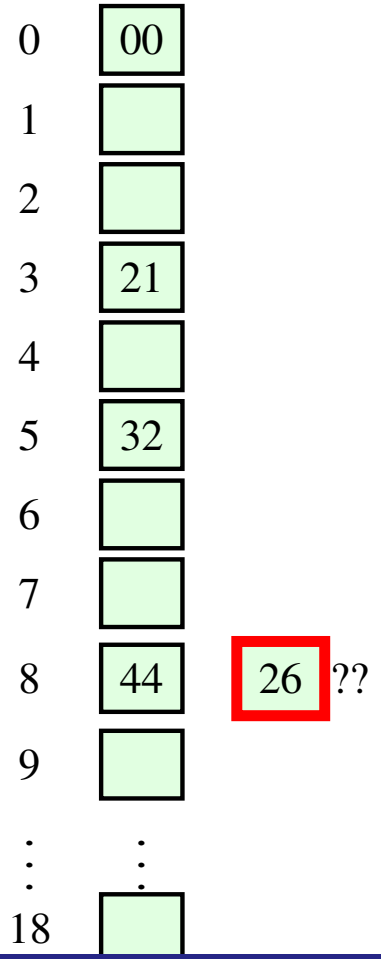
Insert: $32 \rightarrow h(32) = 5$



Hashing – Examples

- Consider the set of two digit integers, $s_1 s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.
- Let $h(s_1 s_2) = s_1 + s_2$. The 100 possible keys are mapped to a hash table of 18 elements.

Insert: $26 \rightarrow h(26) = 8$ – **Collision!** How do we handle this case?



Hashing – Collisions

- Collisions are **unavoidable** !
  - → For keys x_1, x_2, $h(x_1) = h(x_2)$ for $x_1 \neq x_2$.
- The range of the possible values of keys is typically much larger than the hash table.
  - → This is unlike the bit vector Set ADT implementation in which the number of possible keys **equals** the size of the hash table.
- Need a collision resolution strategy
  1. Compute the table location to access using $h(x)$
  2. If position $h(x)$ is occupied by another value, then apply the **collision resolution strategy**.

Hashing – Applications

- Hashing is useful when the number of possible keys is much larger than the number of keys used.
- **Examples**:
  - Spell Check Dictionary - huge number of letter combinations and many fewer words
  - Data Bases - e.g., 7 digit student numbers have 10^7 possible key values and only about 30,000 students.
  - Compiler Symbol Table - A compiler keeps track of all symbols which is much less then all possible variable and function names.

Hashing – Applications

- Hashing is useful when the number of possible keys is much larger than the number of keys used.
- **Examples**:
  - Spell Check Dictionary - huge number of letter combinations and many fewer words
  - Data Bases - e.g., 7 digit student numbers have 10^7 possible key values and only about 30,000 students.
  - Compiler Symbol Table - A compiler keeps track of all symbols which is much less then all possible variable and function names.
- Hashing, unlike trees, **cannot**:
  - Find the minimum or maximum elements, find elements in a range of values.
  - Print all elements in $\Theta(n)$ time
- But, hashing can answer the question of whether or not a key in the a set very quickly
  - $\Theta(1)$ on average if hash table is not very full

Hash Functions

Question: What features are important to have a good hash function ?
Answer:

- Need to design the hash function very carefully in order to have good performance.
- The hash function should be designed to **minimize the chance of a collision**.
  - → However, no matter how hard you try it is nearly impossible to avoid collisions altogether

Question: What features are important to have a good hash function ?
Answer:

- Need to design the hash function very carefully in order to have good performance.
- The hash function should be designed to **minimize the chance of a collision**.
  - → However, no matter how hard you try it is nearly impossible to avoid collisions altogether

Example: Consider this classroom. Say we wish to index every person, x in the class using the hash function

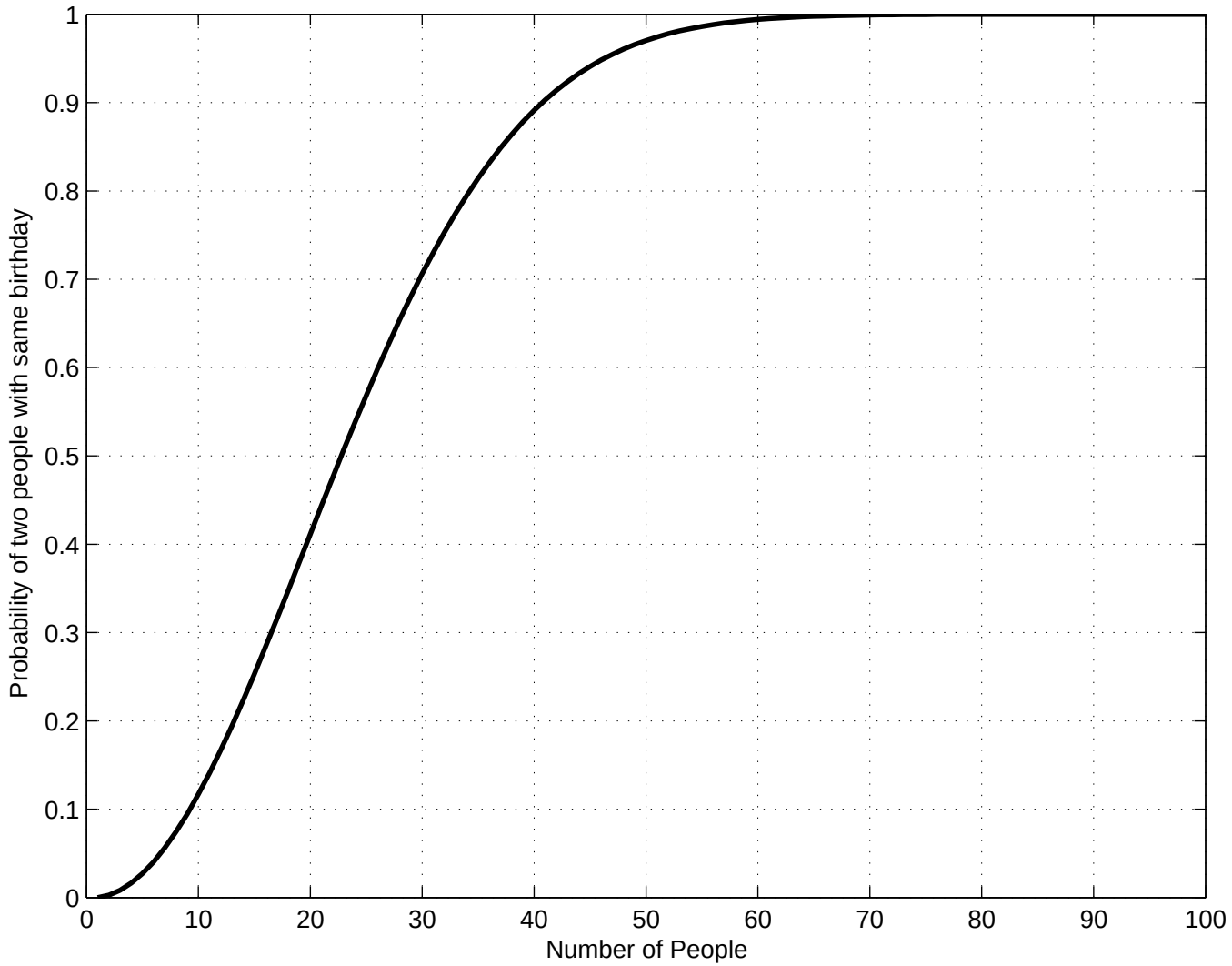
$$h(x) = \text{Birthday of person } x$$

(i.e, the day in the year). It is possible to show, and you will in EE 3TQ4, that

$$\text{Prob(at least two people have same birthday in a room of } n) = 1 - \frac{365 \times 364 \times \cdots (365 - n + 1)}{365^n}$$

This is the probability of having a collision.

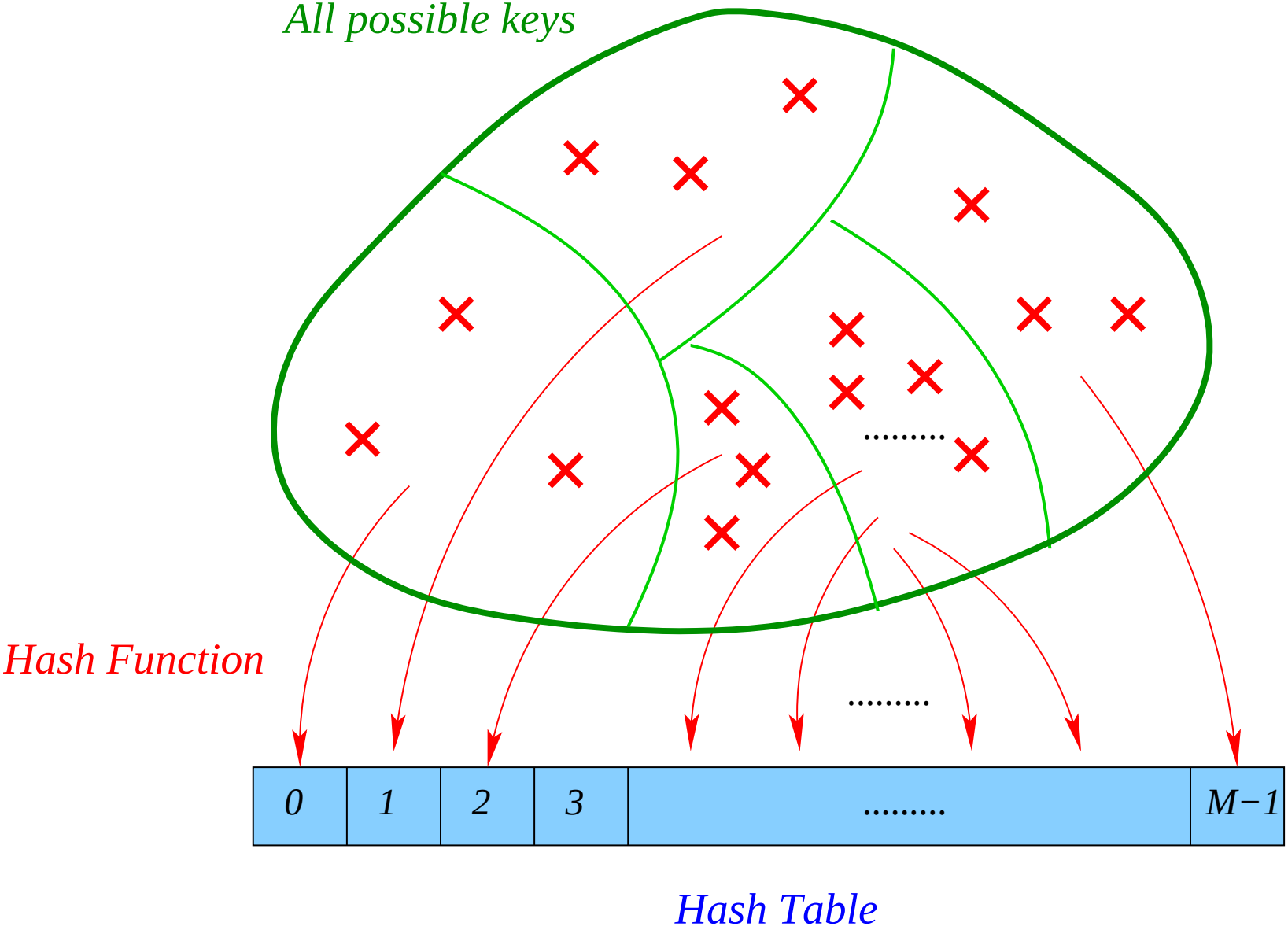
- If there are more than 23 students, the probability of at least one collision is greater than 50%, even if most spots in hash table are empty.


Probability of two people with same birthday
Number of People
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100

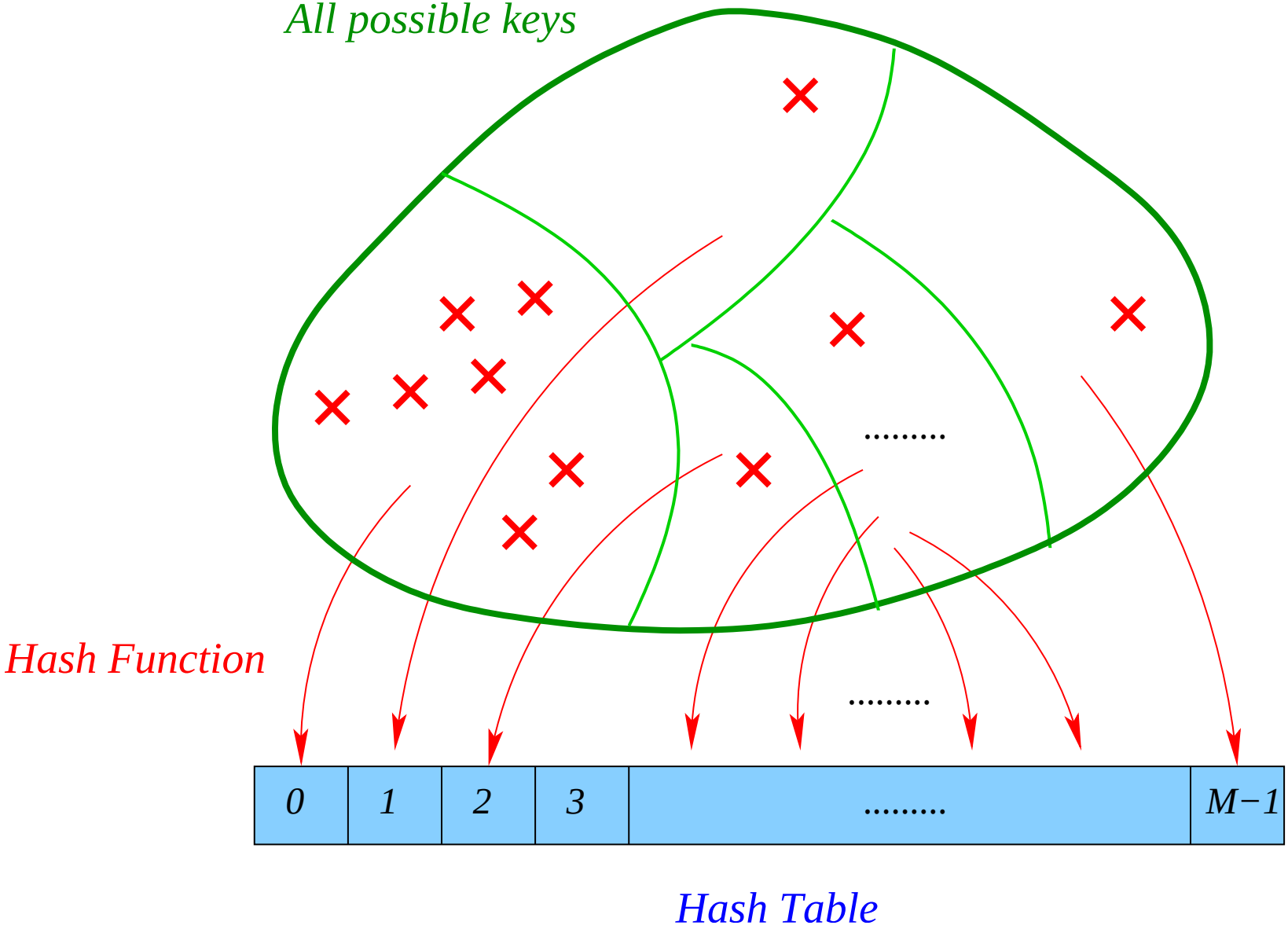
Hash Functions

- A **good** hash function is one which makes sure the hash table is filled evenly.
  - → Tries to make sure that each element of the hash table is equally likely to get value
- If we **do not** know anything about the distribution of the incoming keys
  - → Design a hash function assuming all key values are equally likely and ensure that all hash values are equally likely.
- If we **do** know the distribution of the incoming keys
  - → Design a hash function which depends on the distribution of the input to ensure an approximately uniform distribution over hash values.

A Good Hash Function - assume uniform distribution over the **x**'s



A Bad Hash Function - assume uniform distribution over the **x**'s



Hash Functions

Good hash functions should ...

- spread hash values in a "random" fashion over all table values
- not be too difficult to compute
- not be too dependent on the size of the input key
- don't assume anything special about the set of allowable keys

☞ Let's consider some hash functions for **strings**

Hash Functions – Example 1

A simple hash function for strings.

```
1 #define M 10007
2 unsigned int h(char *key){
3     unsigned int hashVal = 0;
4     while(*key != '\0')
5         hashVal += *key++;
6     return(hashVal % M);
7 }
```

→ The modulo operation wraps the resulting hash value to a valid index into the hash table.

→ **Good** Uses every character of in the key.

→ **Bad** If the sum is not large enough (relative to `M`), the `mod` operation gives a poor distribution of table indices.

Hash Functions – Example 1

A simple hash function for strings.

```
1 #define M 10007
2 unsigned int h(char *key){
3     unsigned int hashVal = 0;
4     while(*key != '\0')
5         hashVal += *key++;
6     return(hashVal % M);
7 }
```

→ The modulo operation wraps the resulting hash value to a valid index into the hash table.

→ **Good** Uses every character of in the key.

→ **Bad** If the sum is not large enough (relative to `M`), the `mod` operation gives a poor distribution of table indices.

- ▻▻ Example: if `M=10,000`, consider all 10 letter words.
- ▻▻ `'A'==65` and `'Z'==90`. The maximum possible sum is 900.
- ▻▻ The resulting hash values are clustered at the low end of the hash table.

Hash Functions – Example 2

Another hash function for strings (assumes at least two characters and '\0')

```
1 #define M 10007
2 unsigned int h(char *key){
3     unsigned int hashVal;
4     hashVal = key[0] + 27*key[1] + 27*27*key[2];
5     return(hashVal % M);
6 }
```

→ Adds the 27 possible characters (26 letters and a space) using correct "place holders"

Hash Functions – Example 2

Another hash function for strings (assumes at least two characters and '\0')

```
1 #define M 10007
2 unsigned int h(char *key){
3     unsigned int hashVal;
4     hashVal = key[0] + 27*key[1] + 27*27*key[2];
5     return(hashVal % M);
6 }
```

→ Adds the 27 possible characters (26 letters and a space) using correct "place holders"

→ **Problems**:

- ⇨ The hash function computes only $27^3 = 17,576$ possible output hash values.
- ⇨ Any string with the same first three characters hash to the same value. There are many more possible keys than hash values
- ⇨ Using the first three chars is not a good key since not all combinations are likely. At most 2851 three character combos are possible in English !
- ⇨ **At most** 28% of all table values will be hashed to.

Hash Functions – Example 3

A better hash function for strings.

```
1 #define M 10007
2 unsigned int h(char *key){
3     unsigned int hashVal = 0;
4     while(*key)
5         hashVal = (hashVal << 5) ^ *key++;
6     return(hashVal % M);
7 }
```

- → This hash function uses all the key elements. This is good to ensure the hash value is randomized, however, it can increase the time to compute the function.
- → It spreads out the hash value better.
- → The << operator is a left-shift operator. Shifting a character left by 5 places is the same as multiplying by $2^5 = 32$.
- → The '$\wedge$' operation is the bitwise XOR, bitwise half-adder.

Hash Functions – Example 4

A good hash function which is widely used.

```
1  unsigned int ELFhash(char *key){
2      unsigned long g,h = 0;
3      while(*key){
4          h = (h<<4) + *key++;
5          g = h & 0xF0000000L;
6          if (g)
7              h ^= g>>24;
8          h &= ~g;
9      }
10     return(h % M);
11 }
```

- This hash function is used in the "**E**xecutable and **L**inking **F**ormat" (**ELF**) used to keep track of object files and executables in *UNIX System V*, Release 4.
- It works on strings of any length, uses every char in `key` and combines values of elements to give a relatively even distribution over hash table.

Collision Resolution Strategies

McMaster University

- As discussed earlier, collisions are **unavoidable**.

Question: What **collision resolution strategy** can we use to handle these cases?

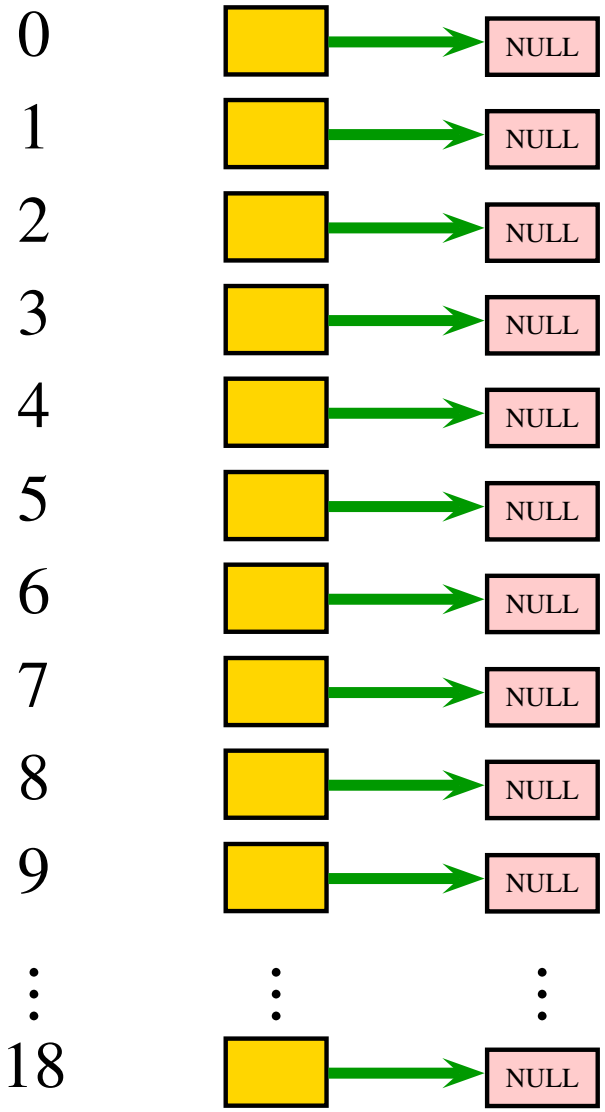
- **Open Hashing**
  - when a collision occurs, extra space is added to the hash table using dynamic memory allocation
- **Closed Hashing**
  - when a collision occurs, a secondary function is used to find an alternate location.

Open Hashing

- Also known as **separate chaining**
- The hash table is an array of pointers which point to a linked data structure
  - → Linked-list
  - → Tree
- To perform a `find`, use hash function to determine which list to traverse, then traverse list until you find the element.
- To `insert`, use hash function to find appropriate list and insert if it is new to list.

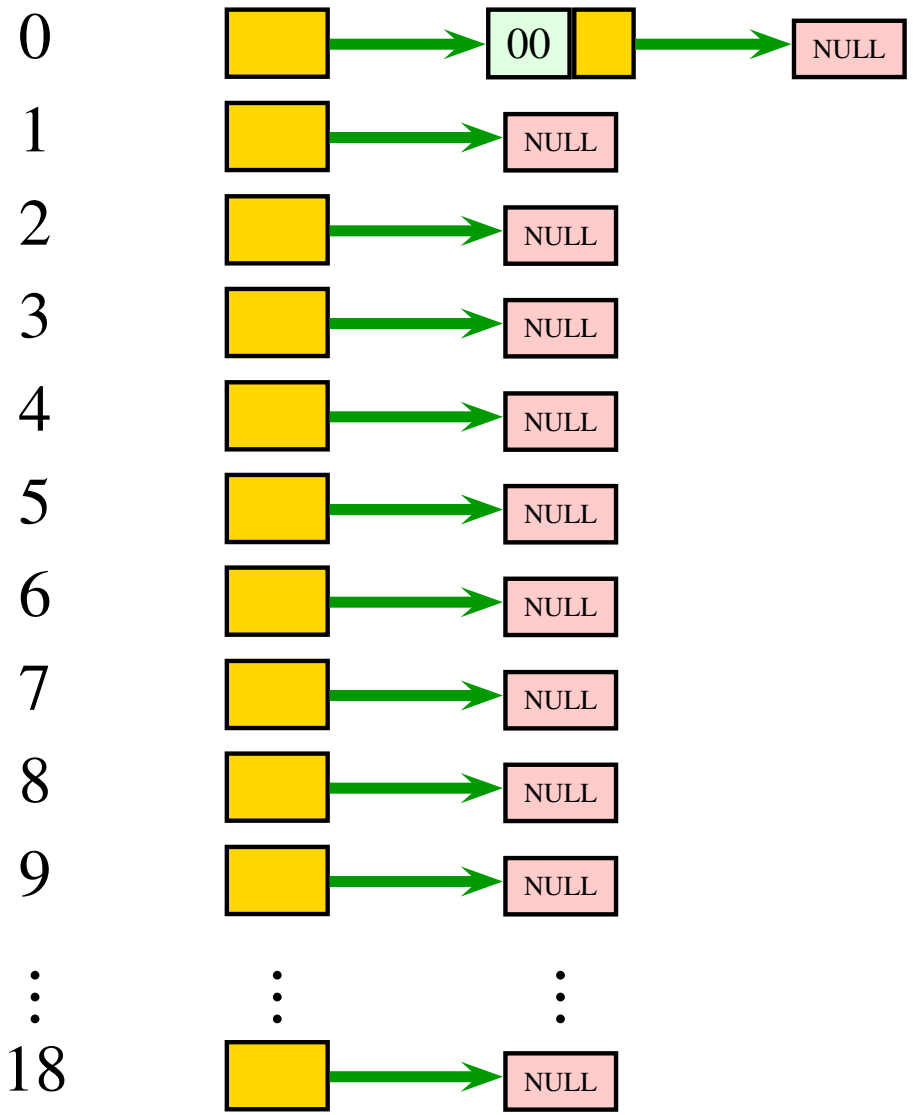
Example: Consider the previous example of hashing two digit integers $s_1 s_2$ using the function $h(s_1 s_2) = s_1 + s_2$

Create Hash Table



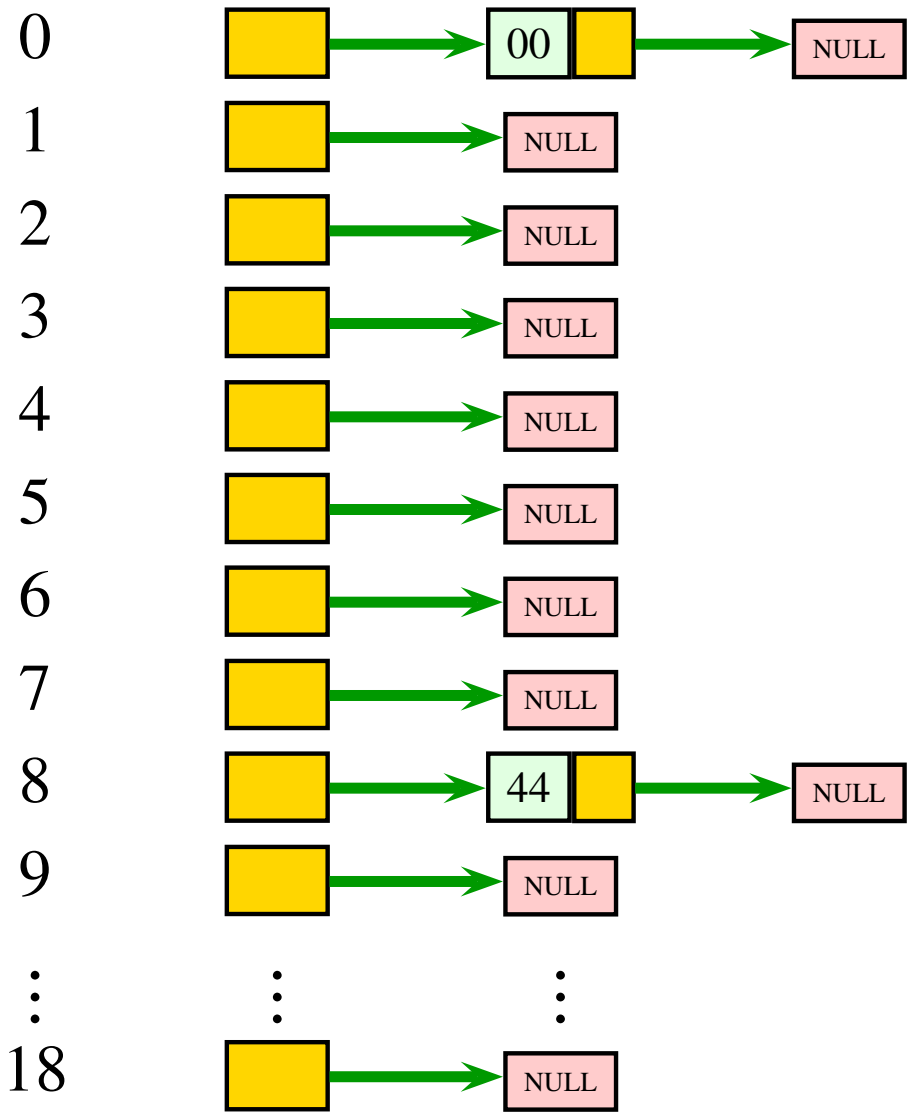
Open Hashing

Insert $00 \rightarrow h(00) = 0$



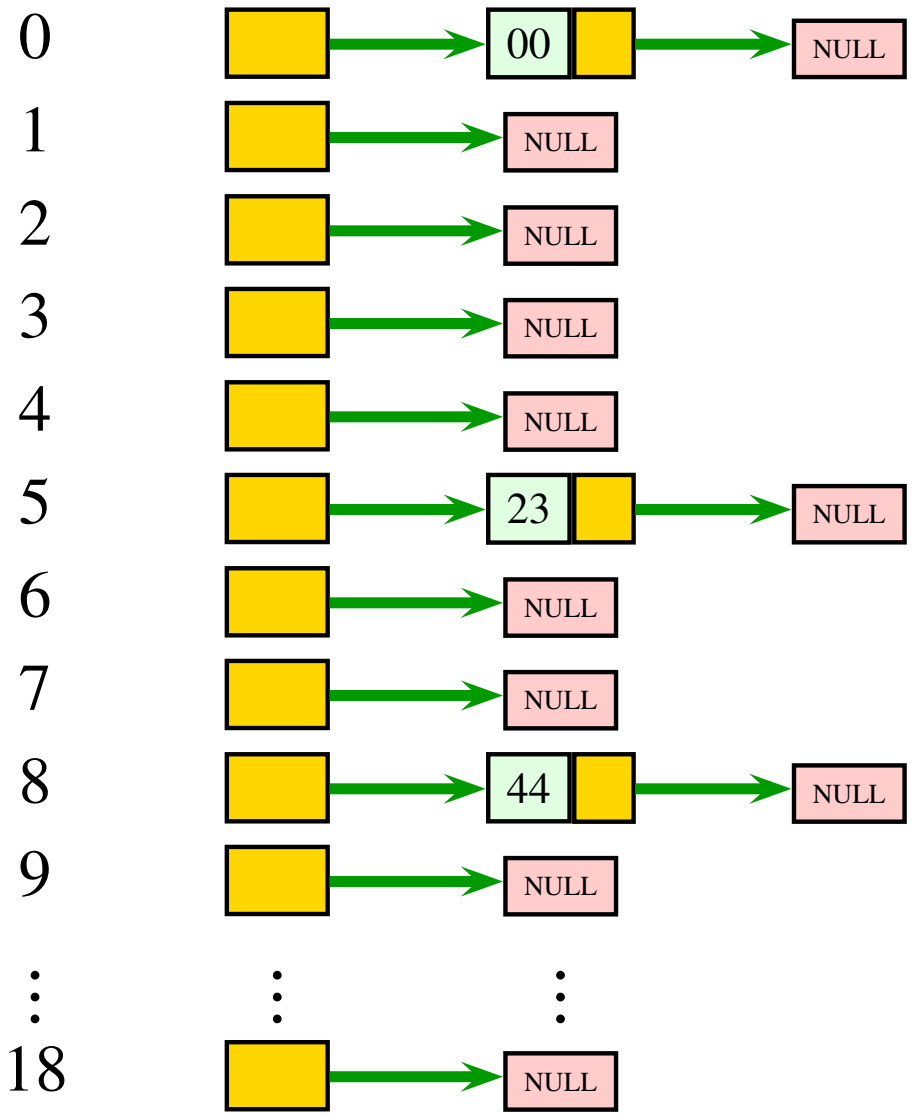
Open Hashing

Insert $44 \rightarrow h(44) = 8$



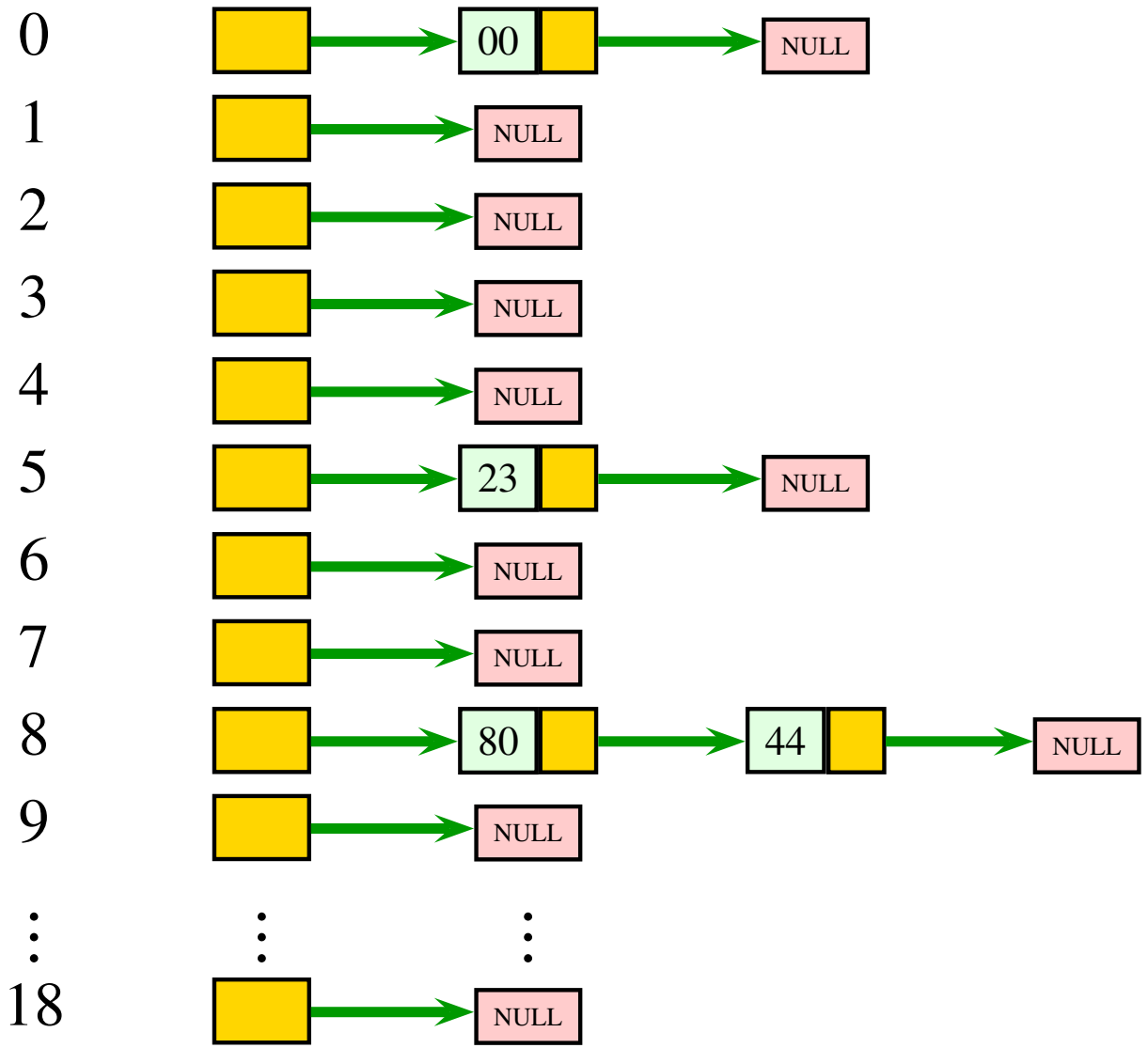
Open Hashing

Insert $23 \rightarrow h(23) = 5$



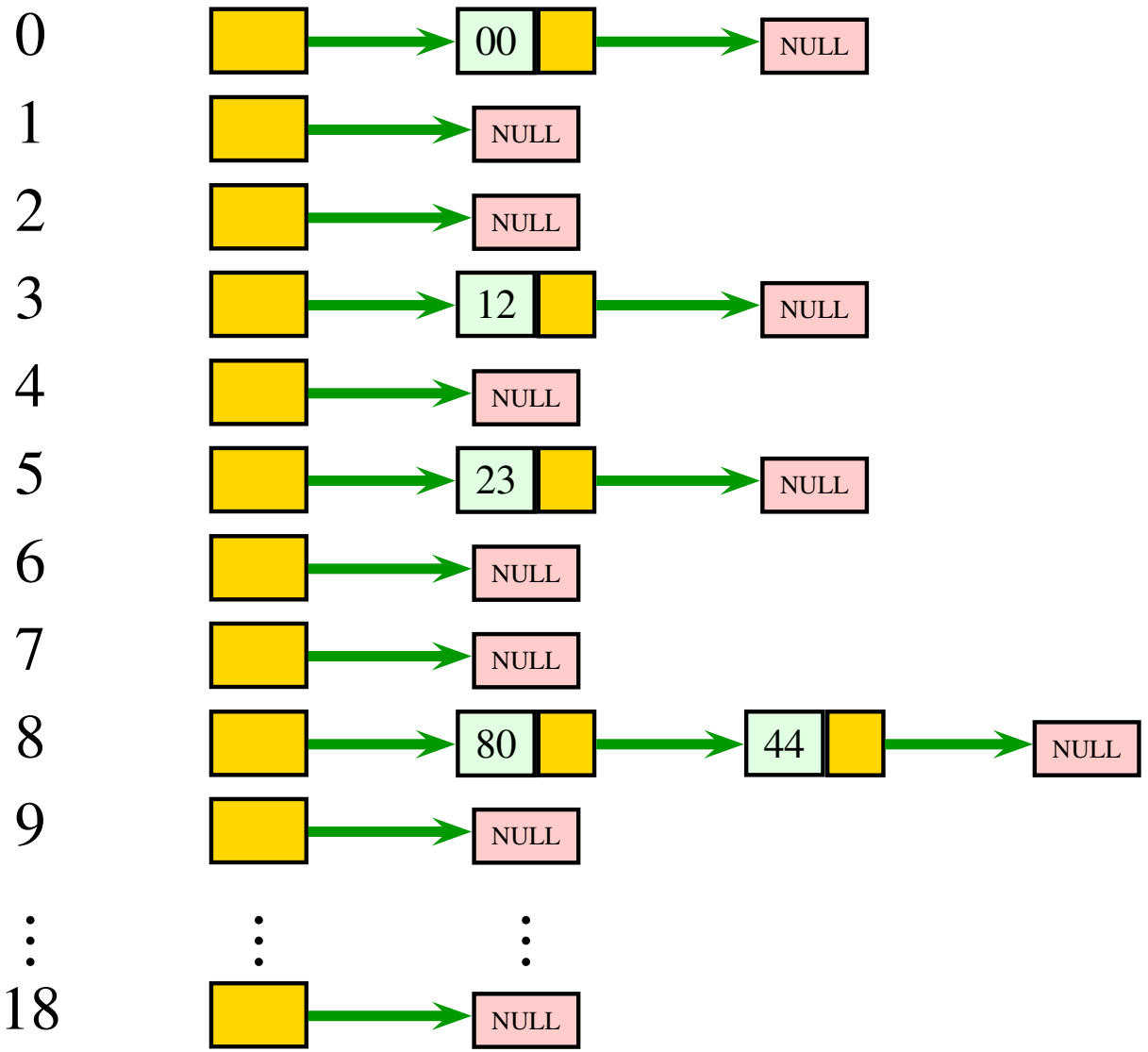
Open Hashing

Insert $80 \rightarrow h(80) = 5$



Open Hashing

Insert $12 \rightarrow h(12) = 3$



Open Hashing

- This technique is flexible and can handle any type of collision
- Assume that computing the hash value takes $\Theta(1)$ time. Then the run-time for insert, delete and find is **linear in the size of the list** corresponding to $h(x)$.

Open Hashing

- This technique is flexible and can handle any type of collision
- Assume that computing the hash value takes $\Theta(1)$ time. Then the run-time for insert, delete and find is **linear in the size of the list** corresponding to $h(x)$.
- If the hash function is poor, new values entered into the hash table **cluster** around a given value in the table
  - → In the example, the hash table at entry 8 has a cluster of two elements.
  - → A good hash function will minimize the length of these clusters
  - → Large clusters mean longer insert, delete and search time!
- A **good** hash function will **minimize** the length of the lists.

Open Hashing

- In general, we do not want the lists in each entry of the hash table to get very long
  - → The number of probes for a successful search is increased
  - → The number of probes for an **unsuccessful** search is also increased

Open Hashing

- In general, we do not want the lists in each entry of the hash table to get very long
  - → The number of probes for a successful search is increased
  - → The number of probes for an **unsuccessful** search is also increased
- It is not necessary to use linked-lists to handle collisions at each entry of the hash table
  - → You can use binary **trees** instead of linked-lists to speed up searching in each bin.
  - → Usually this complexity is not required since we will develop hash functions which limit the size of each list.

Define: The **load factor** as

$$\lambda = \frac{n}{M}$$

where n is the number of elements in the hash table.

- The particular size of the hash table is not important (on average), but rather the load factor
- If hash function is **good** – there are on **average** λ elements in each list.
  - The average case running time for insert, delete and find is $\Theta(\lambda)$.
- In general, we want to have a hash table large enough so that $\lambda \approx 1$.
- Average cost of **insert, delete, find** becomes $\Theta(1)$.
- If the hash function is not good and clusters elements to a given element in the hash table, performance will degrade and linked lists get long.

Open Hashing

- **Pros**
  - Simple and quick implementation – reuse list ADT routines
  - Easy to add an element to the hash table – just insert element to the appropriate linked list.

Open Hashing

- **Pros**
  - → Simple and quick implementation – reuse list ADT routines
  - → Easy to add an element to the hash table – just insert element to the appropriate linked list.
- **Cons**
  - → Both delete and insert routines require $\Theta(n)$ at worst since they may have to traverse all elements – Note for hash tables with small λ the impact of this is much smaller
  - → Need to handle pointers and dynamic memory allocation. This requires an overhead in time and also requires extra space.

Closed Hashing

- Also known as **open addressing**
- Open hashing can be slow due to the overhead in memory allocation and pointer manipulation.

Key Features:

- Use **static** memory allocation for the hash table, i.e., the hash table is an array of **fixed** size.
- When a collision occurs, the collision are resolved by hashing the key to a "nearby" value in the hash table.
- The load factor, $\lambda \leq 1$ since we are using static memory allocation.
  - → In general, $\lambda < 0.5$.

Closed Hashing – Collision Resolution Strategies

- When a collision occurs, **recompute** the hash value until an empty spot in the hash table is found.
- For the key x compute the keys $h_i(x)$, for $i = 0, 1, 2, \ldots$, until an empty spot is found, where

$$h_i(x) = (h(x) + h_c(i)) \text{ modulo } M$$

  for a hash table of M elements.
- $h_c(i)$ is termed the **collision resolution function** and $h_c(0) = 0$.

Closed Hashing – Collision Resolution Strategies

There are three common strategies to implement collision resolution strategies:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

Simple Idea:

- Try to insert the element into the hash table
- If the element doesn't fit into the hash table, try the next element in the hash table.

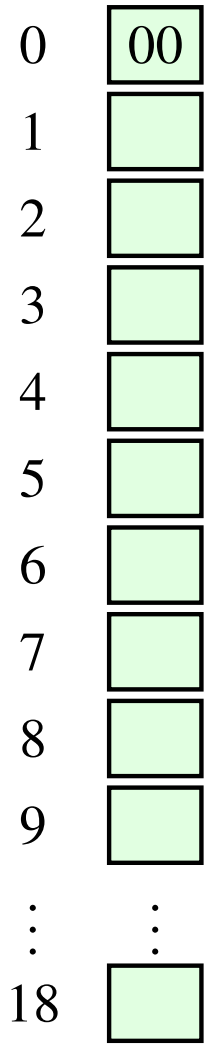
$$h_c(i) = i$$

→ Starting at $h(x)$, try the hash table entries sequentially (with wrap around) until an empty one is found.

Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

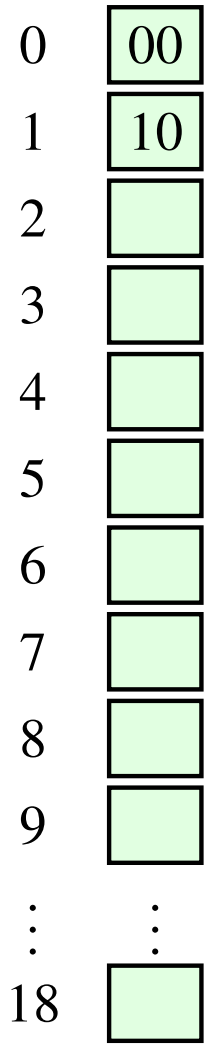
Insert: $00 \rightarrow h(00) = 0$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

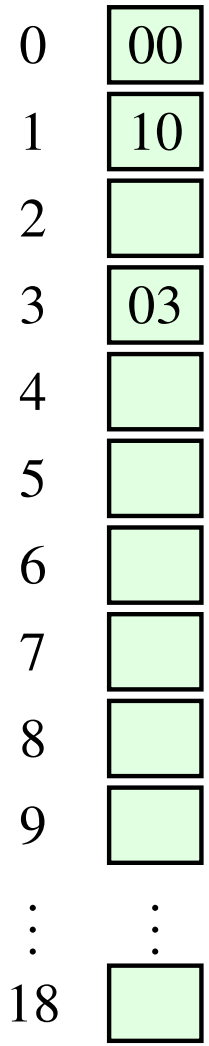
Insert: $10 \rightarrow h(10) = 1$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

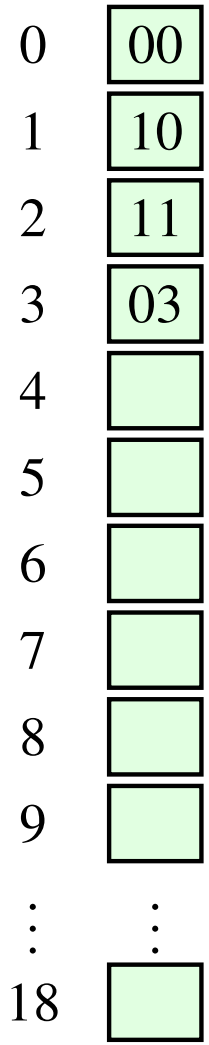
Insert: $03 \rightarrow h(03) = 3$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

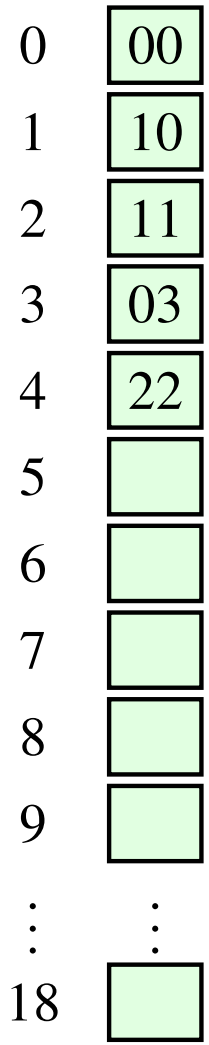
Insert: $11 \rightarrow h(11) = 2$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

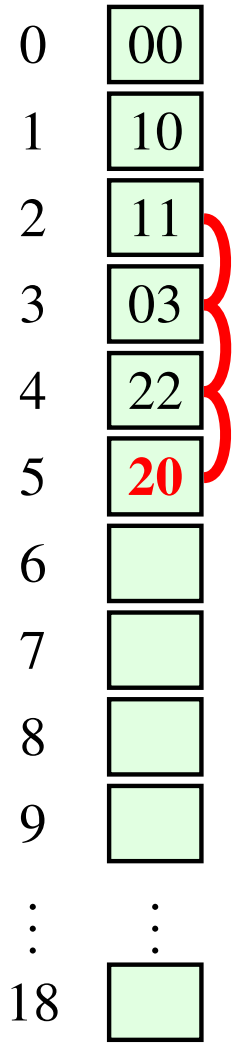
Insert: $22 \rightarrow h(22) = 4$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

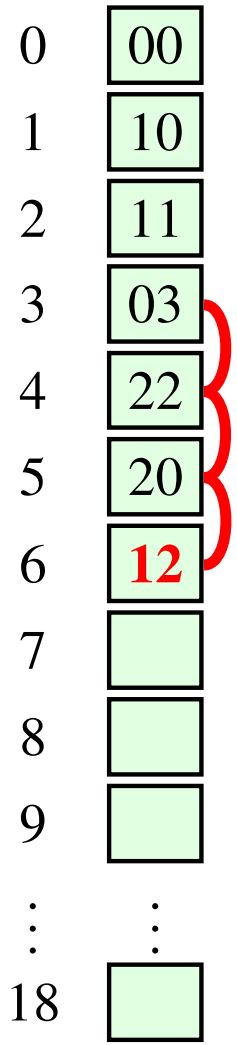
Insert: $20 \rightarrow h(20) = 2$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

Insert: $12 \rightarrow h(12) = 3$



Linear Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

Insert: $62 \rightarrow h(62) = 8$

0	00
1	10
2	11
3	03
4	22
5	20
6	12
7	
8	62
9	
⋮	⋮
18	

Linear Probing

Pros:

- The strategy is simple and works well enough if the table is sufficiently empty

Linear Probing

Pros:

- The strategy is simple and works well enough if the table is sufficiently empty

Cons:

- **Primary Clustering**: tend to have areas of the hash table which have many elements and others which are empty.
- Even if the table is not full, collisions are still likely

Quadratic Probing

- In order to try to deal with the clustering problem, check hash table locations progressively further away

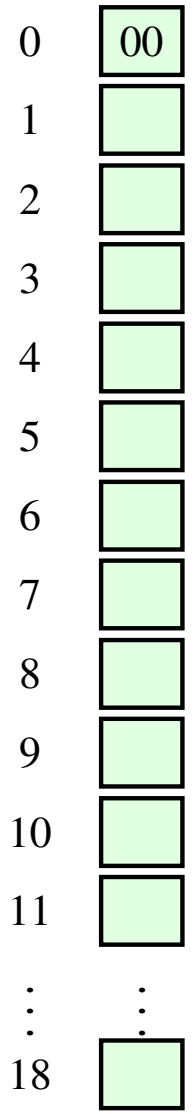
$$h_c(i) = i^2$$

- Less primary clustering, however, have **secondary clustering**.
- Not all locations are checked – no guarantee of finding an empty location
- If the table is less than half empty (i.e., $\lambda < 0.5$) and M is prime, it is always possible to insert a new element.

Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

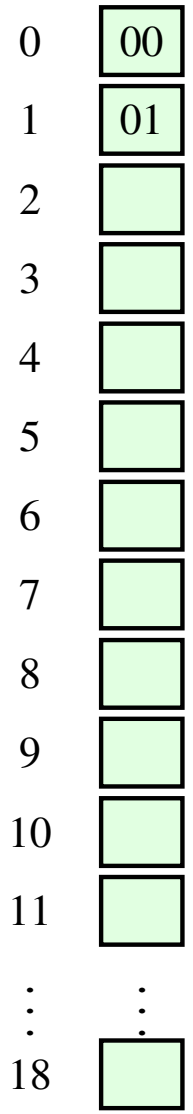
Insert: $00 \rightarrow h(00) = 0$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

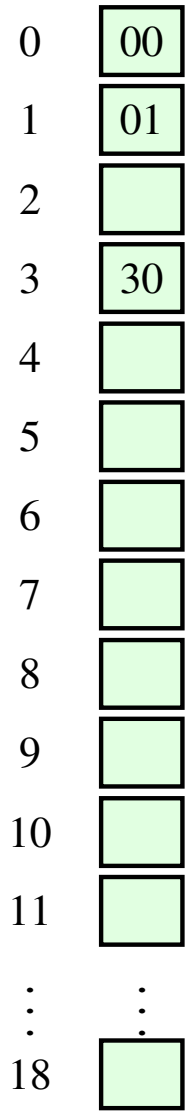
Insert: $01 \rightarrow h(01) = 1$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

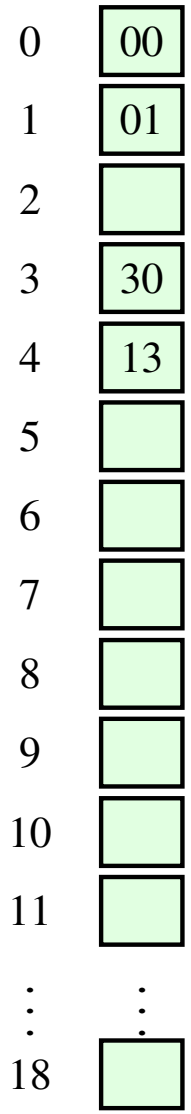
Insert: $30 \rightarrow h(30) = 3$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

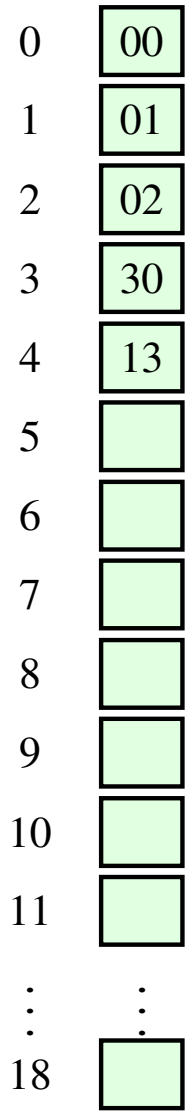
Insert: $13 \rightarrow h(13) = 4$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

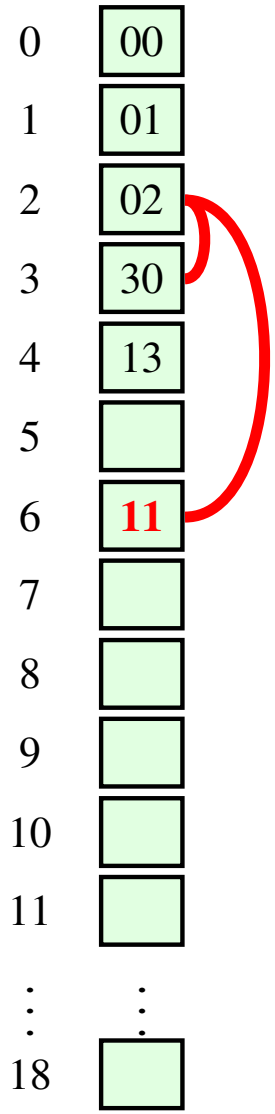
Insert: $02 \rightarrow h(02) = 2$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

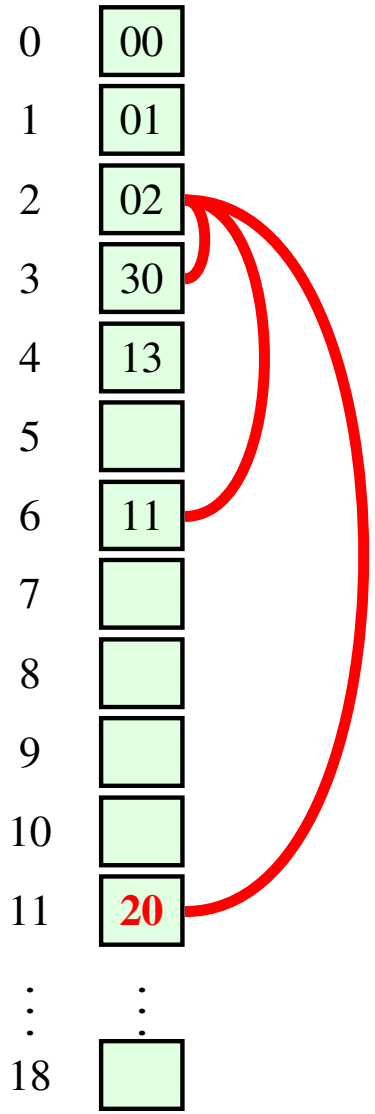
Insert: $11 \rightarrow h(11) = 2$



Quadratic Probing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

Insert: $20 \rightarrow h(20) = 2$



Quadratic Probing

Pros:

- The strategy is still simple and eliminates primary clustering.

Quadratic Probing

Pros:

- The strategy is still simple and eliminates primary clustering.

Cons:

- May cause secondary clustering
- To guarantee insertion is possible need to have a table of prime size and must be sufficiently empty.

Double Hashing

- When a collision occurs, use another hash function to resolve

$$h_c(i) = ih_2(x)$$

for a **secondary hash function**, $h_2(x)$.

- Check alternate hash table locations for $I = 0, 1, 2, \ldots$ until an empty slot is found.

$$h_i(x) = (h(x) + ih_2(x)) \text{ modulo } M$$

Double Hashing

- When a collision occurs, use another hash function to resolve

$$h_c(i) = ih_2(x)$$

for a **secondary hash function**, $h_2(x)$.

- Check alternate hash table locations for $I = 0, 1, 2, \ldots$ until an empty slot is found.

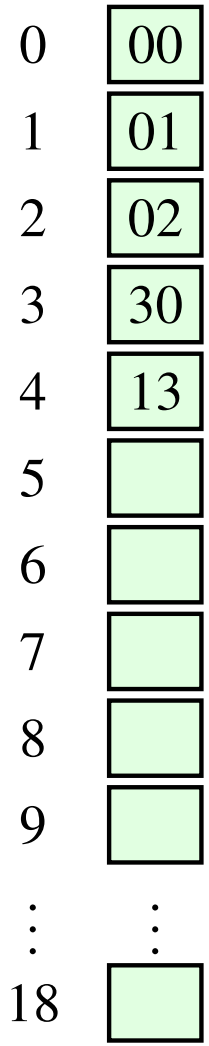
$$h_i(x) = (h(x) + ih_2(x)) \text{ modulo } M$$

- The function $h_2(x)$ can never be zero for any key value
- Performs nearly as well as random hashing
- May be slower since need to evaluate two hash functions

Double Hashing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ and $h_2(x) = 5 - (s_1 s_2) \bmod 5$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

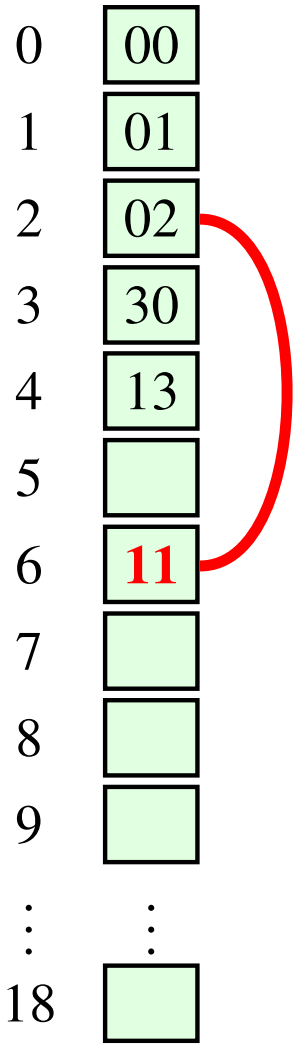
Insert: 00, 13, 02, 30, 01 (no collisions)



Double Hashing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ and $h_2(x) = 5 - (s_1 s_2) \bmod 5$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

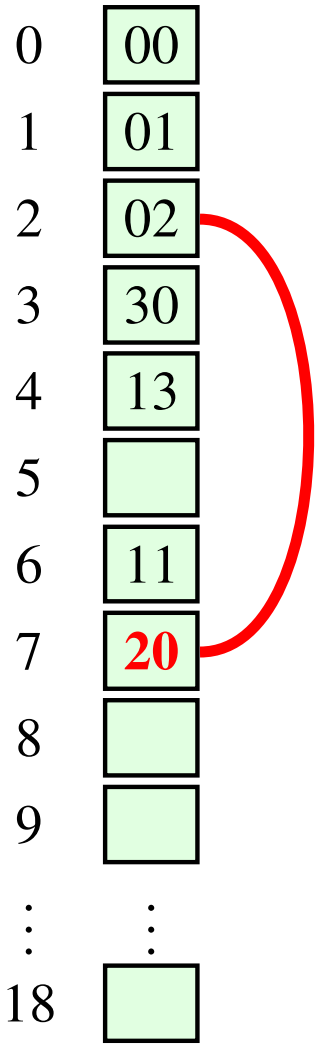
Insert: $11 \rightarrow h(11) = 2 \rightarrow 2 + (5 - 11 \bmod 5) = 6$



Double Hashing – Example

- Let $h(s_1 s_2) = s_1 + s_2$ and $h_2(x) = 5 - (s_1 s_2) \bmod 5$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$.

Insert: $20 \rightarrow h(20) = 2 \rightarrow 2 + (5 - 20 \bmod 5) = 7$

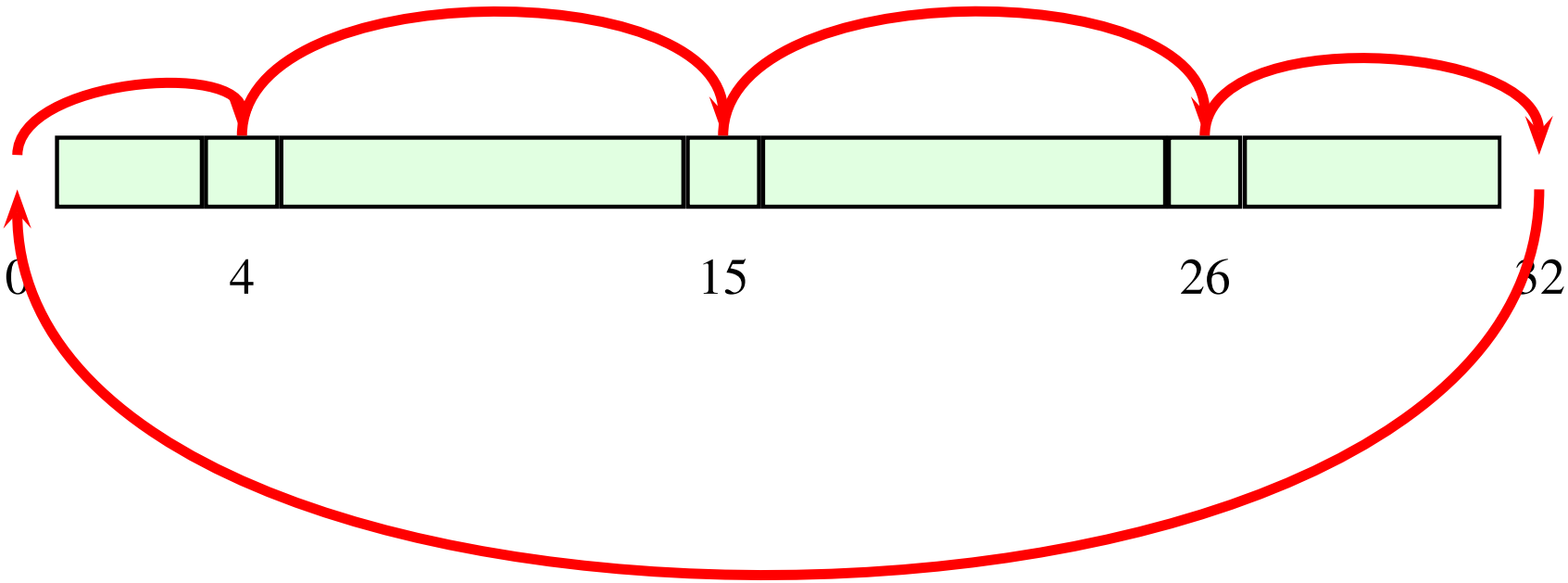


Double Hashing

Question: Why is it important that hash table length M is prime ?
Answer: If not, then it is possible that an insertion will fail even if the hash table is not full.

Double Hashing

Question: Why is it important that hash table length M is prime ?
Answer: If not, then it is possible that an insertion will fail even if the hash table is not full.
Example: Say $h(x) = 4$, $h_c(x) = 11$ and $M = 33$.



Closed Hashing – Find

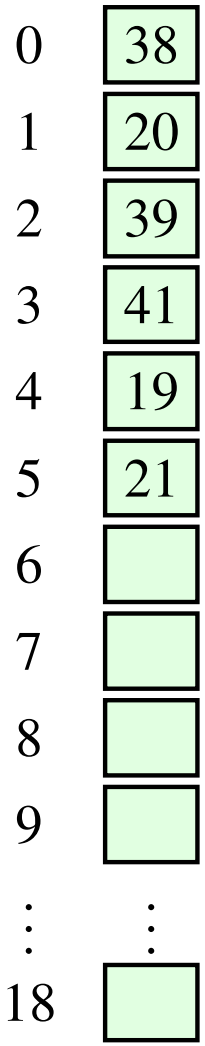
To find if an element x is in the hash table:

- Compute $h(x)$ and check the location computed by $h(x)$.
- While not found or index points to empty element
  - → Compute the next index using the collision resolution strategy

Closed Hashing – Find – Example

- Let $h(x) = x \bmod 19$ where x is taken from the range $0, 1, 2, \ldots, 99$ (linear probing).

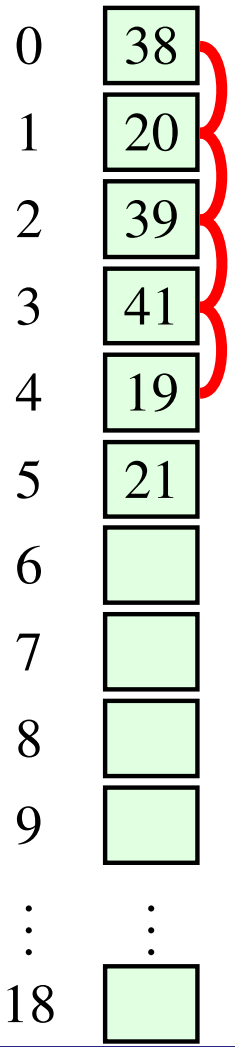
Find: $19 \rightarrow h(19) = 19 \bmod 19 = 0$.



Closed Hashing – Find – Example

- Let $h(x) = x \bmod 19$ where x is taken from the range $0, 1, 2, \ldots, 99$ (linear probing).

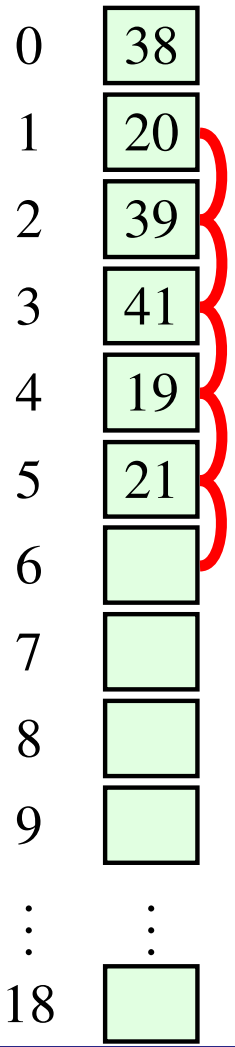
Find: $19 \rightarrow h(19) = 19 \bmod 19 = 0$. **Found** after 5 probes.



Closed Hashing – Find – Example

- Let $h(x) = x \bmod 19$ where x is taken from the range $0, 1, 2, \ldots, 99$ (linear probing).

Find: $58 \rightarrow h(58) = 58 \bmod 19 = 1$. **NOT FOUND** - concluded after 6 probes.



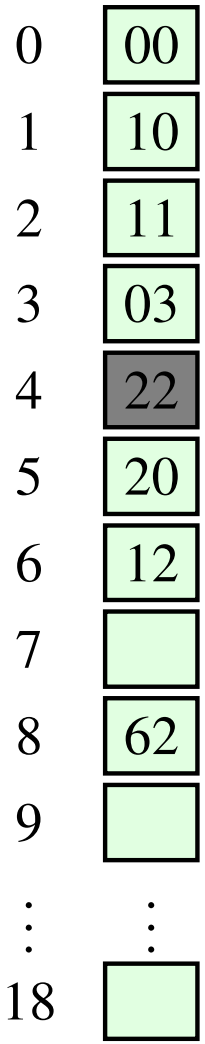
Closed Hashing – Deletion

- You cannot simply delete an element from the hash table, since it will break the "chain" to the remaining elements.
- Perform **lazy deletion** instead, i.e, "mark" the hash table entry as deleted
  - → When searching or inserting ignore elements which are marked as deleted and proceed to next element according to hash function and collision resolution strategies.
  - → Entries marked as deleted can be reused to speed up insertions.

Closed Hashing – Deletion

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$, (linear probing).

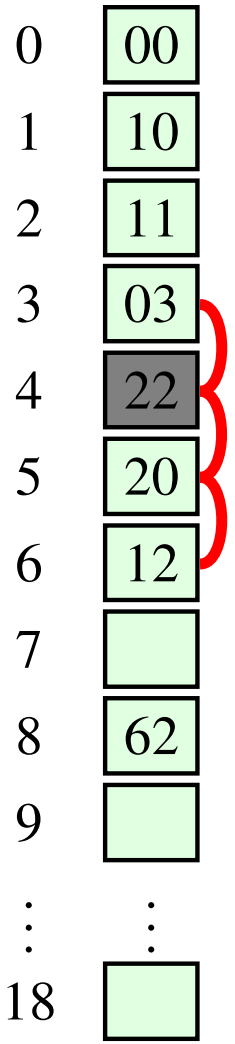
Delete: $22 \rightarrow h(22) = 4$ – mark deleted



Closed Hashing – Deletion

- Let $h(s_1 s_2) = s_1 + s_2$ where s_1, s_2 are taken from the range $0, 1, 2, \ldots, 9$, (linear probing).

Find: $12 \rightarrow h(12) = 3$

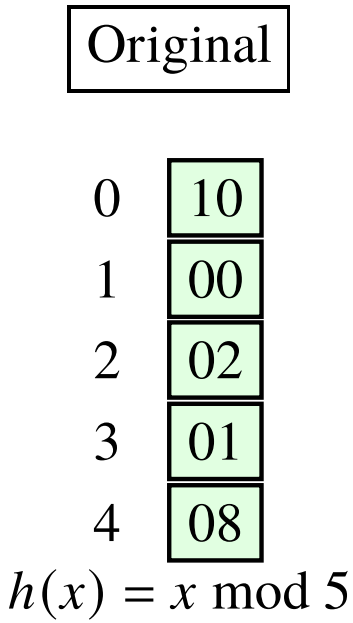


Rehashing

- When the hash table gets too full, all operations slow down due to the increased probability of collision.
- Good performance is achieved with closed hashing if $\lambda < 0.5$.
- When $\lambda > 0.5$ **rehash** into a bigger hash table
  - → Allocate a new hash table of size a prime number, at least double the original size.
  - → Then visit every entry in the original hash table and insert non-deleted elements into the larger hash table using the new hash function.

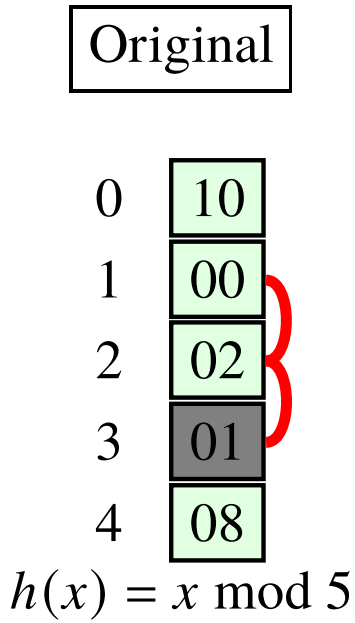
Rehashing – Example

Insert: 10,02,00,01,08,



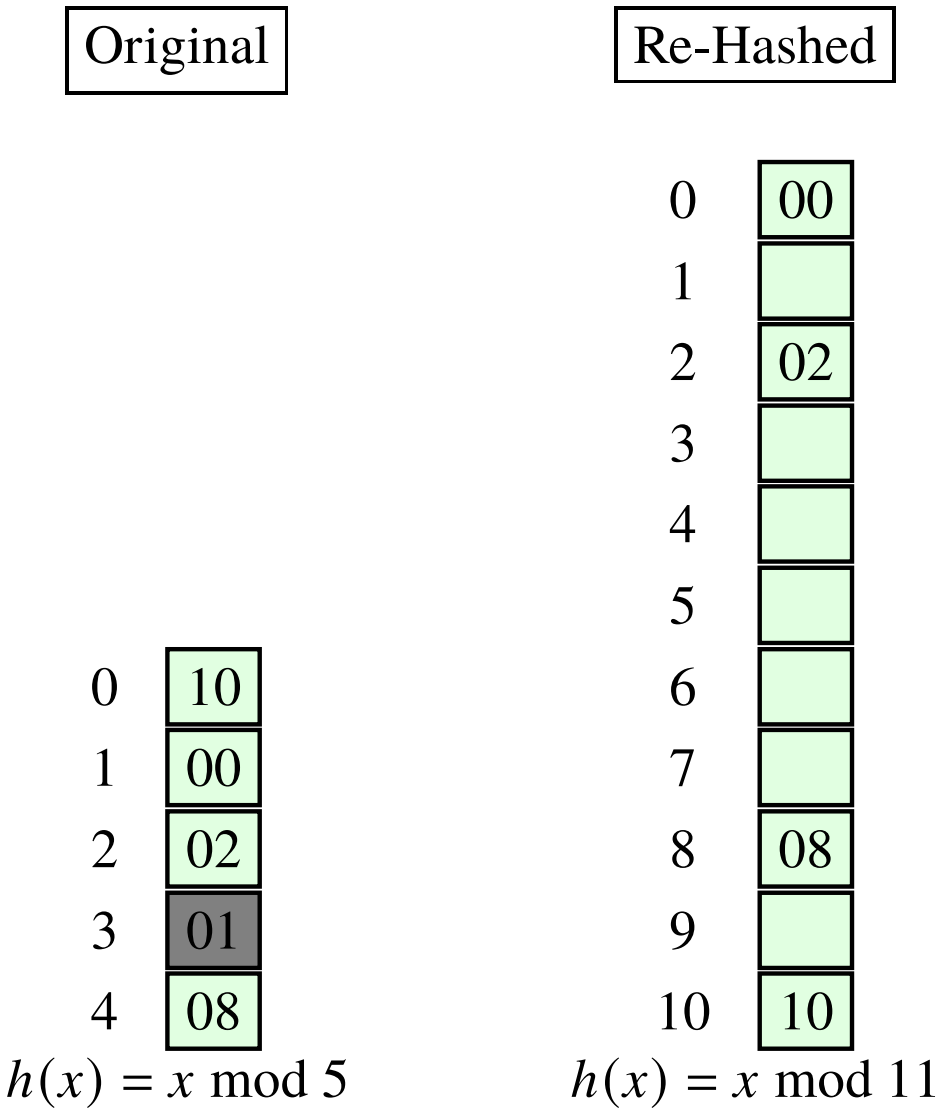
Rehashing – Example

Insert: 10,02,00,01,08, **Delete**: 01



Rehashing – Example

Insert: 10,02,00,01,08, **Delete**: 01, **Rehash**.



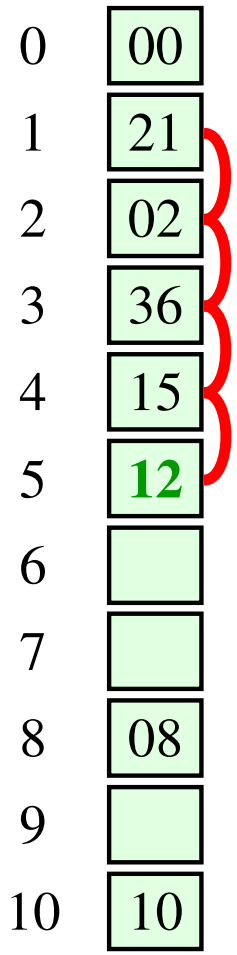
Rehashing

- Rehashing can also be used to reduce the size of the hash table if there are many deleted elements.
  - → Even **after** elements are deleted, they still slow down search since they must be probed.
- Rehashing is an expensive operation, $\Theta(n)$. Do not do it lightly.
- Heuristics for rehashing with quadratic probing:
  - → rehash when table is more than half full
  - → rehash when an insert operation fails
  - → rehash when $\lambda > \lambda_{\text{critical}}$ which is determined by the number of deleted and inserted elements.

Closed Hashing – Observation 1

- The number of probes required to insert a new element into a hash table is the same as the number for probes for an unsuccessful search for that element (assuming no deleted elements)

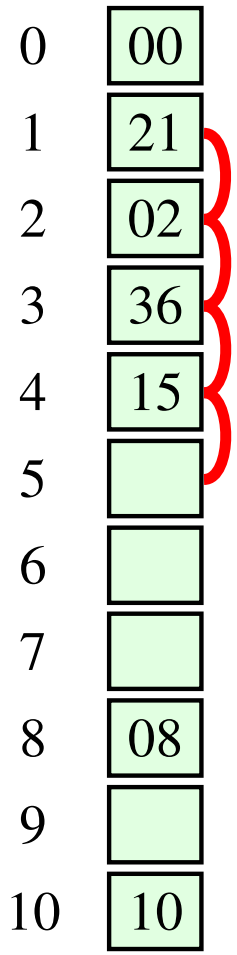
Example: Insert 12 – 5 probes required



Closed Hashing – Observation 1

- The number of probes required to insert a new element into a hash table is the same as the number for probes for an unsuccessful search for that element (assuming no deleted elements)

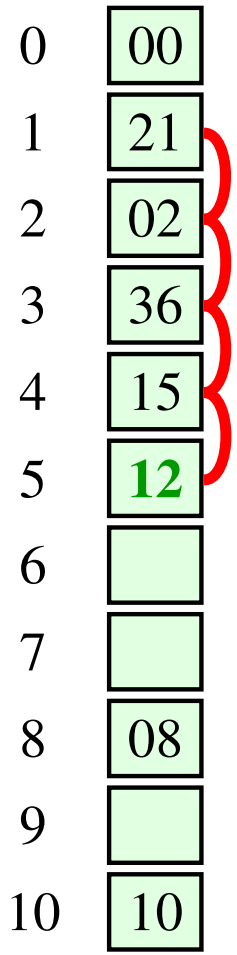
Example: Find 12 (not in hash table) – 5 probes required



Closed Hashing – Observation 2

- The number of probes required to find an element already in a hash table is the same as the number for probes required to insert it.

Example: Insert 12 – 5 probes required

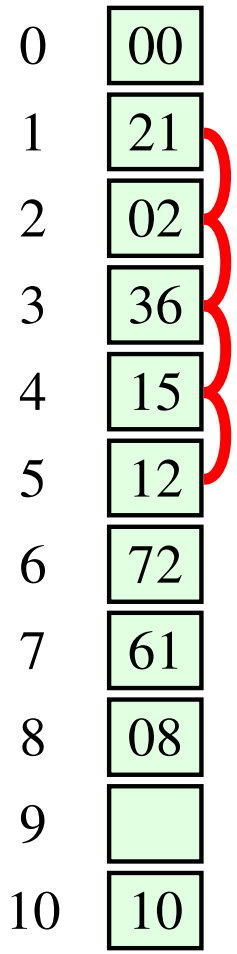


$h(x) = x \bmod 11$, linear probing

Closed Hashing – Observation 2

- The number of probes required to find an element already in a hash table is the same as the number for probes required to insert it.

Example: Find 12 – 5 probes required

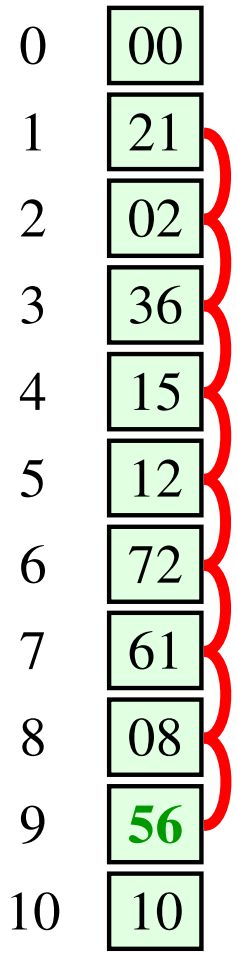


$h(x) = x \bmod 11$, linear probing

Closed Hashing – Observation 3

- Value inserted when the hash table is nearly full, i.e. λ large, take many probes to find even if many elements are deleted.

Example: Insert 56 – 9 probes required

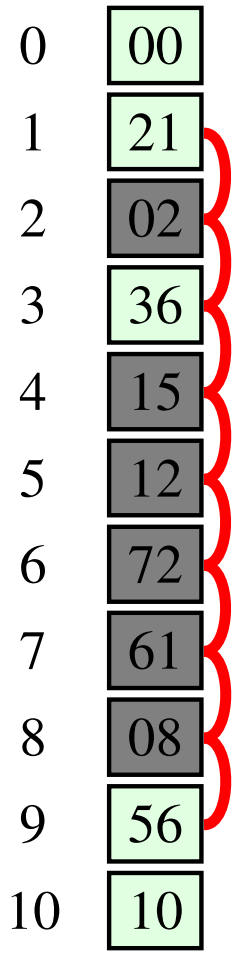


$h(x) = x \bmod 11$, linear probing

Closed Hashing – Observation 3

- Value inserted when the hash table is nearly full, i.e. λ large, take many probes to find even if many elements are deleted.

Example: Find 56 – 9 probes required



$h(x) = x \bmod 11$, linear probing

Closed Hashing – Observation 4

- Hash table entries marked for deletion can be reused, speeding up insertions.

Example: Insert 89

0	00
1	21
2	02
3	36
4	15
5	12
6	72
7	61
8	08
9	56
10	10

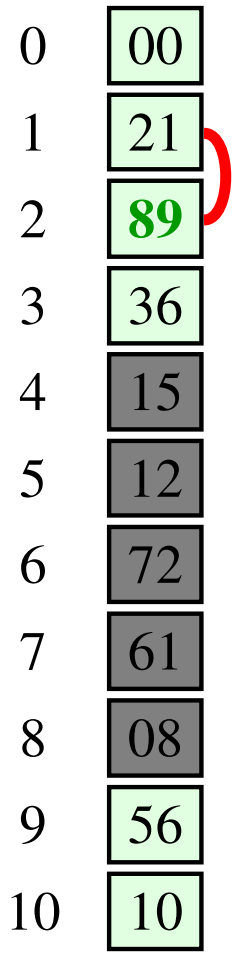
$h(x) = x \bmod 11$, linear probing

Closed Hashing – Observation 4

McMaster University

- Hash table entries marked for deletion can be reused, speeding up insertions.

Example: Insert 89



$h(x) = x \bmod 11$, linear probing

Performance of Closed Hashing

Definitions: Assume that there are n elements in the hash table of size M.

- $\lambda = n/M$ – the **load factor** is the ratio of the number of elements in the table to the size of the hash table.
- U_n – the average number of probes required to determine that an element is not in the hash table (or to insert).
- S_n – the average number of probes required to determine that an element is in the hash table.
- Assume there are no deleted elements.

Performance of Closed Hashing – Example

Linear Probing, keys s_1, s_2 from $0, 1, \ldots, 9$ and $h(s_1 s_2) = s_1 + s_2$.

Index	Element	# Fail	# Success	Index	Element	# Fail	# Success
0	**00**	8	1	10		1	-
1	**01**	7	1	11		1	-
2	**02**	6	1	12		1	-
3	**30**	5	1	13		1	-
4	**13**	4	1	14		1	-
5	**20**	3	4	15		1	-
6	**11**	2	5	16		1	-
7		1	-	17		1	-
8	**08**	2	1	18		1	-

- $\lambda = 8/19 \approx 0.42$
- $U_n = 221/92 \approx 2.4$ – average number of probes to be unsuccessful
- $S_n = 15/8 \approx 1.88$ – average number of probes when an element is found

Performance of Closed Hashing – Theory

In general, for linear probing, the average number of probes to sucessfully/unsucessfully find an element can be derived to be,

$$U_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

$$S_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

Performance of Closed Hashing – Theory

Consider a simplified analysis.

Simplifying Assumptions:

- → Assume hash function with is truly **random** and each hash table entry is equally likely to be probed.
- → Each probe is independent of the previous probes.
- → The hash table gets very large, thus for fixed λ, we have $n \to \infty$.
- ➠ The estimate derived under these assumptions is close to the practical performance of quadratic probing and double hashing.

Performance of Closed Hashing – Theory

→ Probability that a given hash table entry is full = λ

→ Probability that a given hash table entry is not full = $1 - \lambda$.

Therefore,

$$
\begin{aligned}
U_n &= \Pr\{\text{slot 1 empty}\} \times 1 + \Pr\{\text{slot 1 full,slot 2 empty}\} \times 2 + \Pr\{\text{slot 1,2 full,slot 3 empty}\} \times 3 + \ldots \\
&= (1-\lambda) + 2 \underbrace{\lambda}_{\text{prob. a given slot full}} \cdot \underbrace{(1-\lambda)}_{\text{prob. a given slot empty}} + 3\lambda^2(1-\lambda) + \ldots \\
&= (1-\lambda) \sum_{i=0}^{n-1} i\lambda^{i-1}
\end{aligned}
$$

What happens when the hash table gets very large, i.e., $n \to \infty$?

We know

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

if $0 < a < 1$. Differentiating both side with respect to a gives,

$$\sum_{i=0}^{\infty} i a^{i-1} = \frac{1}{(1-a)^2}$$

Substituting into previous expression and taking limit as $n \to \infty$,

$$\begin{aligned} U_n &= \lim_{n\to\infty} (1-\lambda) \sum_{i=0}^{n-1} i\lambda^{i-1} \\ &= (1-\lambda)\frac{1}{(1-\lambda)^2} \\ &= \frac{1}{1-\lambda} \end{aligned}$$

- ☞ **Note:** This satisfies our intuition since $1-\lambda$ fraction of the hash table cells are empty, and so we expect $1/(1-\lambda)$ to be the average number of cells to probe.
- ☞ However, in practice, independence assumption of probes is not valid due to **primary clustering**

Performance of Closed Hashing – Theory

For the average number of probes to a successful find, recall that

- → The number of probes required to find an element already in a hash table is the same as the number for probes at the time of its insertion. (See *Closed Hashing – Observation 2*)
- → The number of probes required to insert and element in a hash table is the same as the number for probes for unsuccessful search of element. (See *Closed Hashing – Observation 1*)

Performance of Closed Hashing – Theory

For the average number of probes to a successful find, recall that

- The number of probes required to find an element already in a hash table is the same as the number for probes at the time of its insertion. (See *Closed Hashing – Observation 2*)
- The number of probes required to insert and element in a hash table is the same as the number for probes for unsuccessful search of element. (See *Closed Hashing – Observation 1*)
- Therefore, number of probes to find the $(i + 1)$-th inserted key is U_i
  - The hash table had only i elements when the key was inserted
  - The number of probes for insertion equals the number of probes for unsuccessful find.
- Notice that earlier insertions are "cheaper" than later ones as the load factor goes from 0 to λ.

So, we can estimate the average S_n (using assumption of independent probes) as,

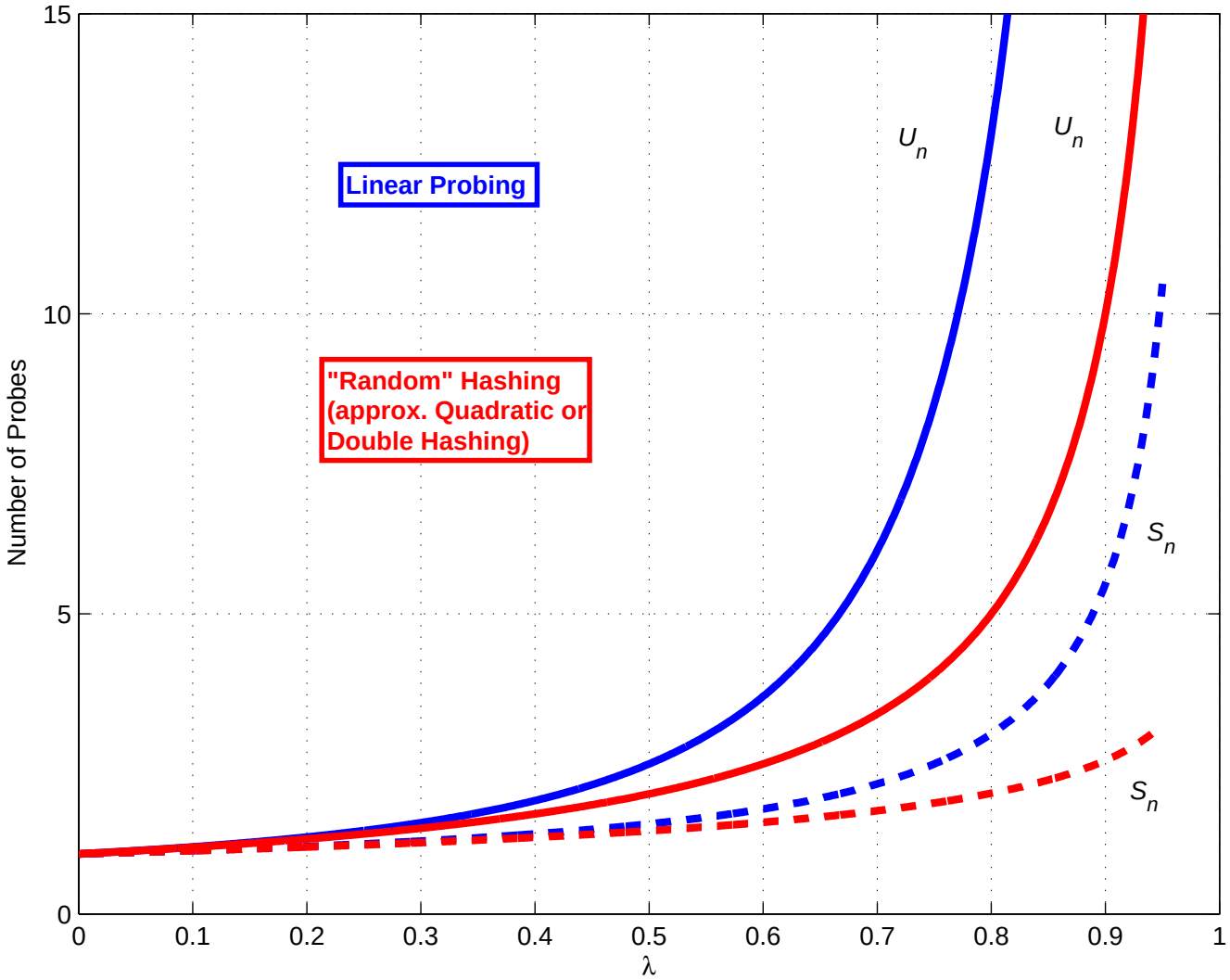
$$\begin{aligned} S_n &= \frac{1}{n}(U_0 + U_1 + \cdots + U_{n-1}) \\ &\approx \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/M} \end{aligned}$$

For constant λ and $n \to \infty$,

$$\begin{aligned} \lim_{n\to\infty} S_n &= \lim_{n\to\infty}\frac{1}{\lambda}\sum_{i=0}^{n-1}\frac{1}{1-i\frac{\lambda}{n}}\cdot\frac{\lambda}{n} \\ &= \frac{1}{\lambda}\int_0^{\lambda}\frac{1}{1-x}dx \qquad (*) \\ &= \frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right) \end{aligned}$$

☞ (*) Since limit of Riemann sum approaches the definite integral.

Performance of Closed Hashing – Theory



Hashing – Overview

- Average-case running time of `insert`, `find` and `delete` is $\Theta(1)$
- Worst-case running time of `insert`, `find` and `delete` is $\Theta(n)$
  - → Can be made unlikely with careful implementation
- Carefully design the hash function!
  - → It should be **easily computable** and should **distribute keys well**
- The load factor has to be sufficiently small!
  - → In open hashing λ is typically close to 1
  - → In closed hashing $\lambda < 0.5$ (when no deletions)
  - → Deleted items slow down operations in closed hashing, even if λ is very small - when the application requires deletions, open hashing is more commonly used.
- Used in compiler symbol tables, online spelling checkers, in game programs - to store transposition tables.

Hash Tables versus Binary Search Trees (BST)

- `insert`, `find` and `delete` are faster on average for hash tables ($\Theta(1)$ vs. $\Theta(\log n)$)
- Worst-case running time of `insert`, `find` is the same
  - In hash tables good average-case performance for any input can be guaranteed - randomly choose the hash function at different executions for the same input
  - Consistent worst-case performance on some inputs cannot be avoided with binary search trees - example: sorted input
- BST's support routines that require order, while hash tables cannot.
  - Example BST:
    - `findMin`, `findMax` - $\Theta(\log n)$ on average
    - output elements in order or output all items in a certain range - $\Theta(n)$
  - Hash tables cannot implement this functionality operations.

Next Topic ...

In this topic we have considered the problem of searching in a data set. In the next topic we will look at an equally fundamental operation ...

☞ **Sorting**