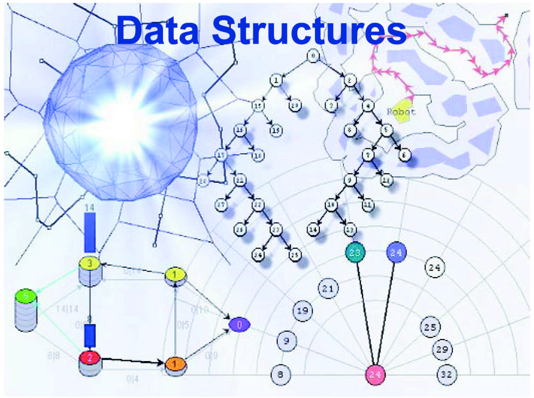
BBM 201
DATA STRUCTURES

Lecture 10:
Hashing & Hash Tables



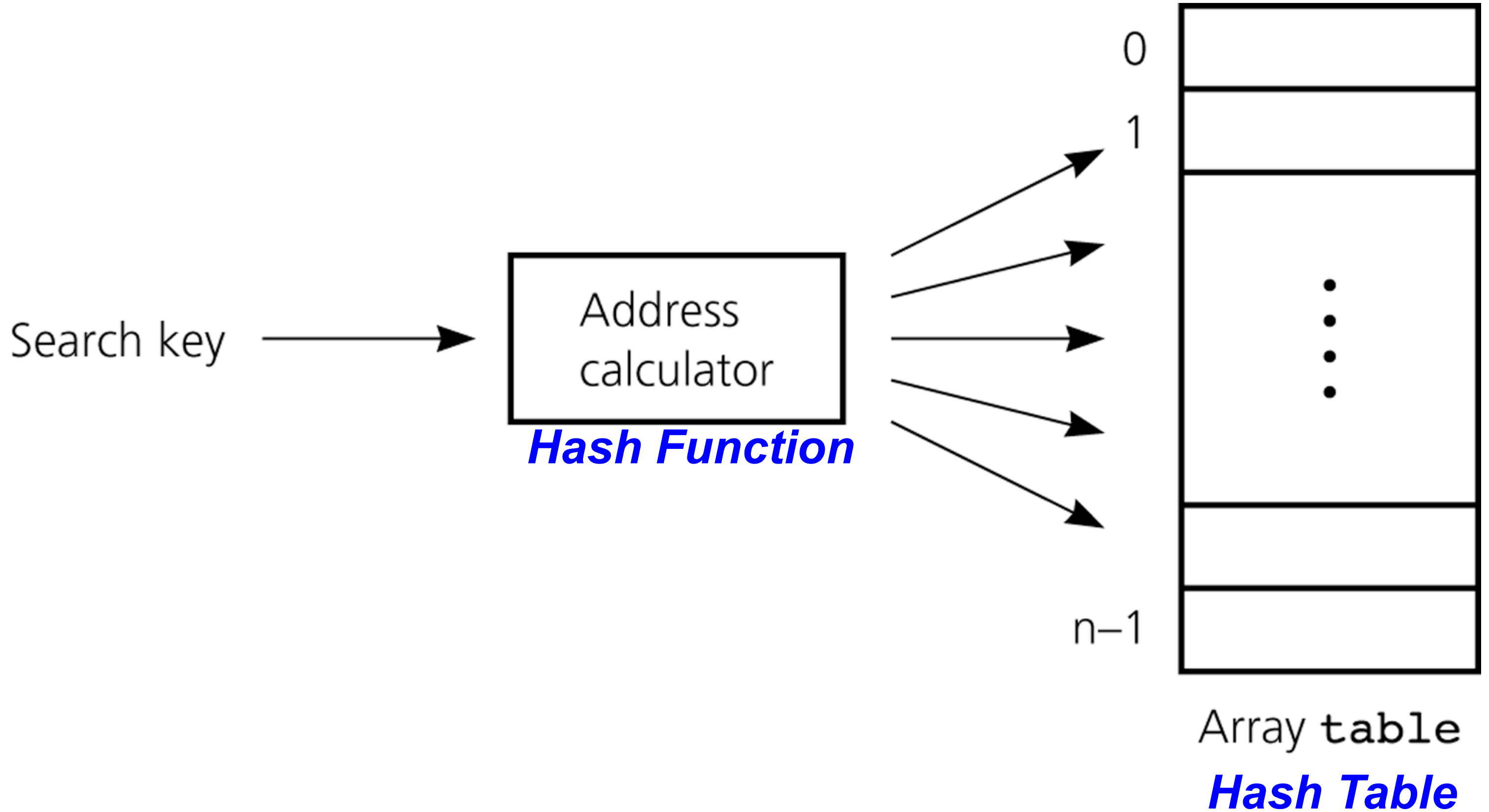
Hashing

- Using balanced search trees (2-3, 2-3-4, red-black, and AVL trees), we implement **table operations in O (logN) time**
  - retrieval, insertion, and deletion
- Can we find a data structure so that we can perform these table operations **even faster** (**e.g., in O(1) time**)?
  - **Hash Tables**

Hash Tables

- In hash tables, we have
  - **An array** (index ranges 0 … n – 1) and
    - Each array location is called a ***bucket***
  - **An address calculator** (***hash function***), which maps a search key into an array index between 0 … n – 1

Hash Function -- Address Calculator



Hashing

- A **hash function** tells us where to place an item in array called a **hash table**.
  - This method is known as **hashing**.
- Hash function **maps a search key into an integer** between 0 and n – 1.
  - We can have different hash functions.
  - Hash function depends on key type (int, string, ...)
    - E.g., **h(x) = x mod n**, where x is an integer

Collisions

- A **perfect hash function** maps each search key into a unique location of the hash table.
  - A perfect hash function is possible if we know all search keys in advance.
  - In practice we do not know all search keys, and thus, a hash function can map more than one key into the same location.
- **Collisions** occur when a hash function maps more than one item into the same array location.
  - We have to resolve the collisions using a certain mechanism.

Hash Functions

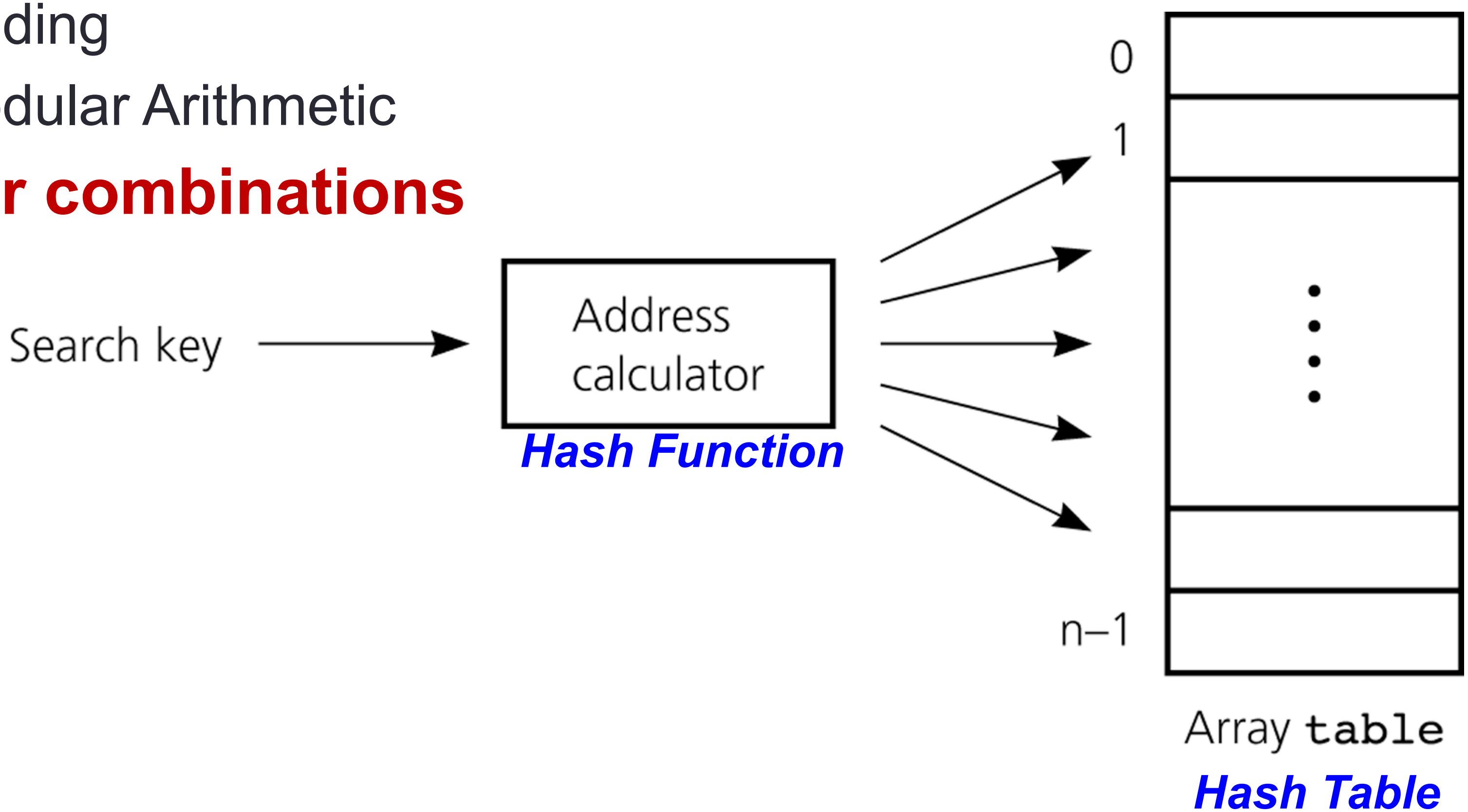
- We can design different hash functions.
- But a **good hash function** should
  - be easy and fast to compute
  - place items uniformly (evenly) throughout the hash table.
- We will consider only **integer hash functions**
  - On a computer, everything is represented with bits.
  - They can be converted into integers.
    - 1001010101001010000110…. remember?

Everything is an Integer

- If search keys are strings, think of them as integers, and apply a hash function for integers.
- For example, strings can be encoded using ASCII codes of characters.
- Consider the string "NOTE"
  - ASCII code of **N** is 4Eh (01001110), **O** is 4Fh (01001111), **T** is 54h(01010100), **E** is 45h (01000101)
  - Concatenate four binary numbers to get a new binary number 01001110010011110101010001000101= 4E4F5445h = **1313821765**

How to Design a Hash Function?

- **Three possibilities**
  1. Selecting digits
  2. Folding
  3. Modular Arithmetic
- **Or, their combinations**



Hash Functions -- Selecting Digits

- **Select certain digits** and combine to create the address.
- For example, suppose that we have 11-digit Turkish nationality ID's
  - Define a hash function that selects the 2nd and 5th most significant digits

    h(033475678) = **37**

    h(023455678) = **25**

  - Define the table size as 100
- Is this a good hash function?
  - No, since it does not place items uniformly.

Hash Functions -- Folding

- **Folding –** selecting all digits and adding them.

- For example, suppose previous nine-digit numbers
  - Define a hash function that selects all digits and adds them

    h(033475678) = **0 + 3 + 3 + 4 + 7 + 5 + 6 + 7 + 8** = 43

    h(023455678) = **0 + 2 + 3 + 4 + 5 + 5 + 6 + 7 + 8** = 40
  - Define the table size as 82

- We can select a group of digits and add the digits in this group as well.

Hash Functions -- Modular Arithmetic

- **Modular arithmetic –** provides a simple and effective hash function.

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

- The table size should be a prime number.
  - ***Why? Think about it.***

- We will use modular arithmetic as our hash function in the rest of our discussions.

Why Primes?

- Assume you hash the following with x mod 8:
  - 64, 100, 128, 200, 300, 400, 500

0	64 128 200 400
1	
2	
3	
4	100 300 500
5	
6	
7	

Why Primes?

- Now try it with x mod 7
  - 64, 100, 128, 200, 300, 400, 500

0	
1	64 128 400
2	100
3	500
4	200
5	
6	300

Rationale

- If we are adding numbers $a_1, a_2, a_3 \ldots a_4$ to a table of size m
  - All values will be hashed into multiples of

    $gcd(a_1, a_2, a_3 \ldots a_4, m)$

  - For example, if we are adding 64, 100, 128, 200, 300, 400, 500 to a table of size 8, all values will be hashed to 0 or 4

    gcd(64,100,128,200,300,400,500, **8**) = 4

  - When m is a prime $gcd(a_1, a_2, a_3 \ldots a_4, m) = 1$, all values will be hashed to anywhere

    gcd(64,100,128,200,300,400,500,**7**) = 1

    unless $gcd(a_1, a_2, a_3 \ldots a_4) = m$, which is rare.

Every integer that shares a common factor with the tableSize will be hashed into an index that is a multiple of this factor.

Hashing a Sequence of Keys

- $K = \{K_1, K_2, \ldots, K_n\}$
- E.g., `Hash("test") = 98157`
- Design Principles
  - Use the entire key
  - Use the ordering information

Use the Entire Key

```
unsigned int Hash(const string& Key) {
  unsigned int hash = 0;
  for (string::size_type j = 0; j != Key.size(); ++j)
  {
        hash = hash ^ Key[j] // exclusive or
  }
  return hash;
}
```

- Problem: Hash(“ab”) == Hash(“ba”)

Use the Ordering Information

```
unsigned int Hash(const string &Key) {
  unsigned int hash = 0;
  for (string::size_type j = 0; j != Key.size(); ++j)
  {
        hash = hash ^ Key[j];
        hash = hash * (j%32);
  }
  return hash;
}
```

Better Hash Function

```
unsigned int Hash(const string& S)
{
  string::size_type i;
  unsigned long int bigval = S[0];

  for (i = 1; i < S.size(); ++i)
         bigval = ((bigval & 65535) * 18000) // low16 * magic_number
         + (bigval >> 16) // high16
         + S[i];

  bigval = ((bigval & 65535) * 18000) + (bigval >> 16);
  // bigval = low16 * magic_number + high16
  return bigval & 65535; // return low16
}
```

```
/* some values:
f(a) = 42064
f(b) = 60064
f(abcd) = 41195
f(bacd) = 39909
f(dcba) = 29480
f(x) = 62848
f(xx) = 44448
f(xxx) = 15118
f(xxxx) = 28081
f(xxxxx) = 45865
*/
```

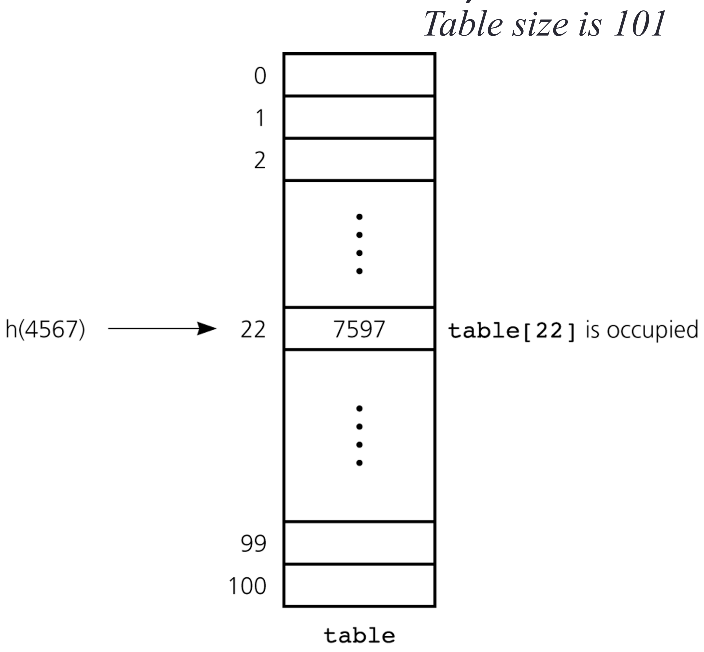
Collision Resolution

- **Collision resolution –** two general approaches
  - **Open Addressing**

    Each entry holds one item
  - **Chaining**

    Each entry can hold more than one item

    (Buckets – hold certain number of items)



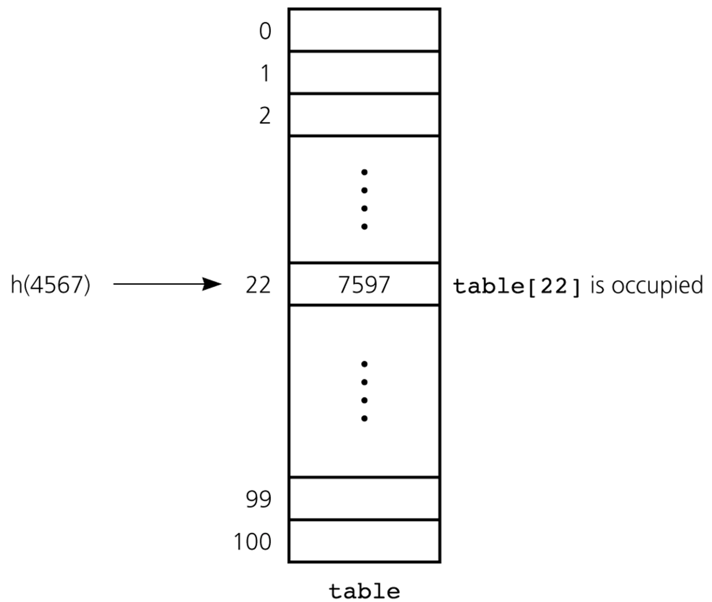
Open Addressing

- **Open addressing** – probes for some other empty location when a collision occurs.

- **Probe sequence:** sequence of examined locations. Different open-addressing schemes:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Open Addressing -- Linear Probing

- **linear probing**: search table sequentially starting from the original hash location.
  - Check next location, if location is occupied.
  - Wrap around from last to first table location



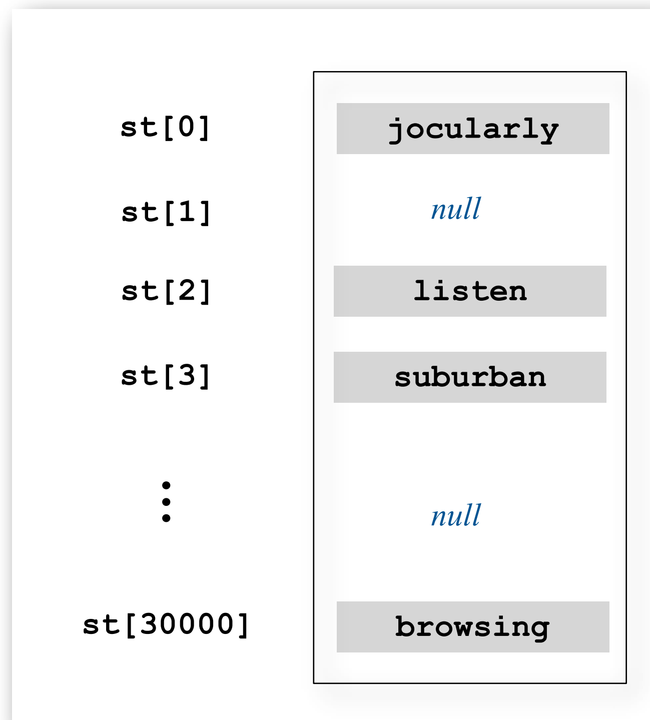
Linear Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 → 2+1=3
    - 25 mod 11 = 3 → 3+1=4
    - 24 mod 11 = 2 → 2+1, 2+2, 2+3=5
    - 10 mod 11 = 10
    - 9 mod 11 = 9 → 9+1, 9+2 mod 11 =0

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Collision resolution: open addressing

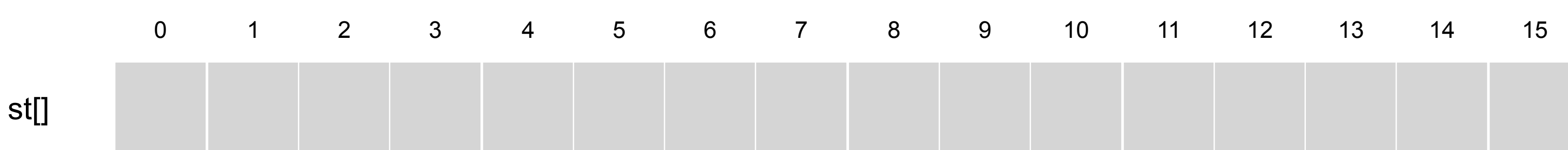
- Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953] When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.



M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

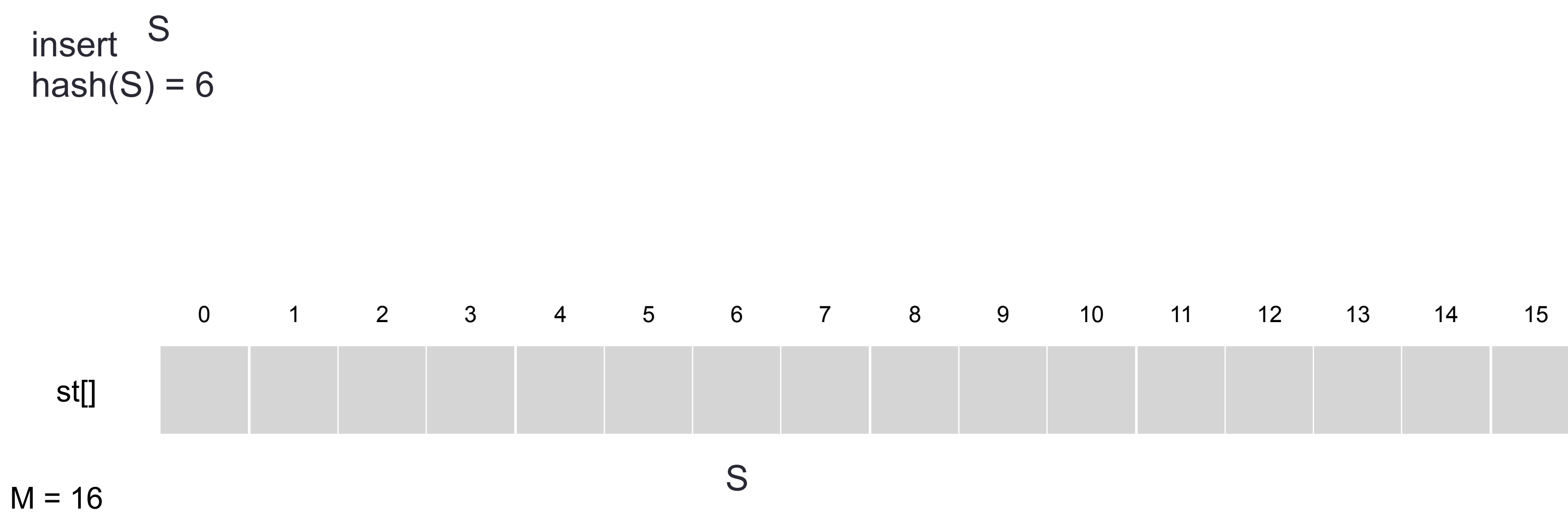
insert S
hash(S) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.



Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert S
hash(S) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert E
hash(E) = 10

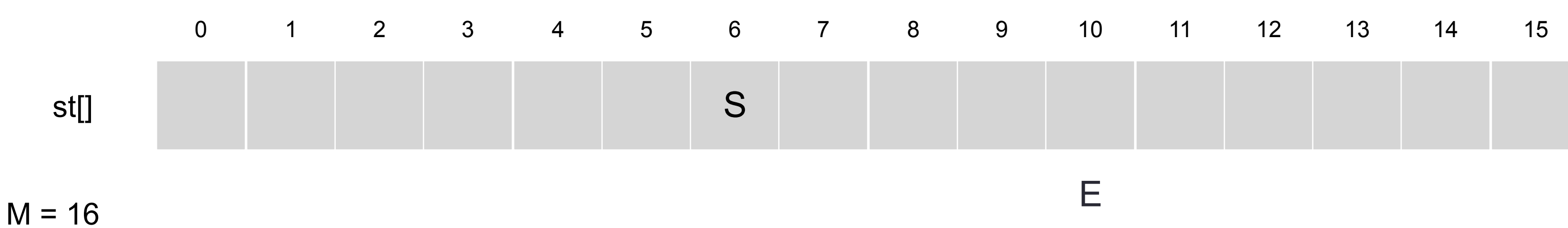
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert E
hash(E) = 10



Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert E
hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert A
hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert A
hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]							S				E					
					A											

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert A
hash(A) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert R
hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert R
hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					
															R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert R
hash(R) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert C
hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert C
hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E				R	
						C										

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert C
hash(C) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
					H											

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
						H										

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
							H									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
								H								

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert X
hash(X) = 15

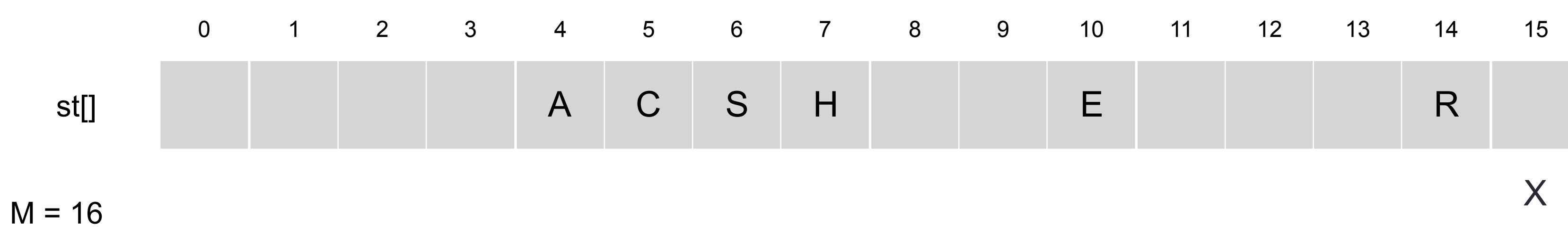
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert X
hash(X) = 15



Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert X
hash(X) = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert M
hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert M
hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	X
		M														

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert M
hash(M) = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert P
hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert P
hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X
															P	

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert P
hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E				R	X
	P															P

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert P
hash(P) = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
							L									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
								L								

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
									L							

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search
hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search E
hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
											E					

M = 16

search hit
(return corresponding value)

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
							L									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

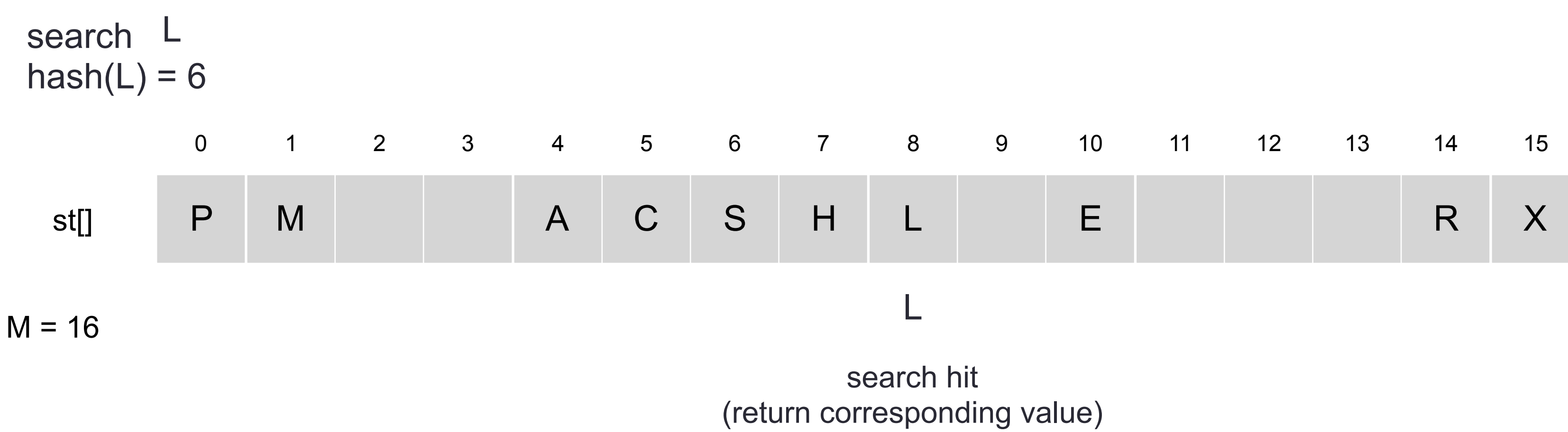
search L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
								L								

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.



Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search K
hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search K
hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
						K										

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

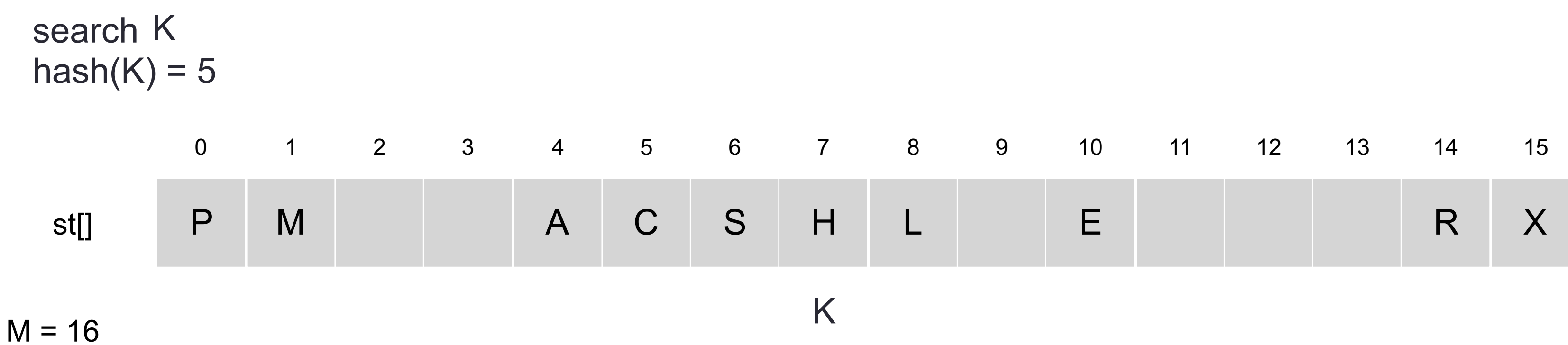
search K
hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
							K									

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.



Linear probing hash table

- Hash. Map key to integer i between 0 and $M-1$.
- Insert. Put at table index i if free; if not try $i+1$, $i+2$, etc.
- Search. Search table index i; if occupied but no match, try $i+1$, $i+2$, etc.

search K
hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
									K							

M = 16

Linear probing hash table

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

search K
hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
										K						

M = 16

search miss
(return null)

Linear probing - Summary

- Hash. Map key to integer i between 0 and $M - 1$.
- Insert. Put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search. Search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

- Note. Array size M must be greater than number of key-value pairs N.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Linear Probing -- Clustering Problem

- One of the problems with linear probing is that table items tend to **cluster** together in the hash table.
  - i.e. table contains groups of consecutively occupied locations.
- This phenomenon is called **primary clustering**.
  - Clusters can get close to one another, and merge into a larger cluster.
  - Thus, the one part of the table might be quite dense, even though another part has relatively few items.
  - Primary clustering causes long probe searches, and therefore, decreases the overall efficiency.

Open Addressing -- Quadratic Probing

- **Quadratic probing:** almost eliminates clustering problem

- Approach:
  - Start from the original hash location `i`
  - If location is occupied, check locations $i+1^2$, $i+2^2$, $i+3^2$, $i+4^2$ ...
  - Wrap around table, if necessary.

Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ $2+1^2=3$
    - 25 mod 11 = 3 ➔ $3+1^2=4$
    - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11,
      $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

A simple implementation

```
#define TABLESIZE ...
typedef KEYTYPE ...
typedef RECTYPE ...
struct record{
    KEYTYPE k;
    RECTYPE r;
} table[TABLESIZE];
```

Assume:
-we have a hash function *h(key)*
-we have a probing function *f(i)** *to resolve collisions*.
-we have a special value *nullkey*
    -used to indicate an *empty* record.

*(*This function is referred to as rehash function in Langsam et al.'s Book)*

A simple implementation

```
int search(KEYTYPE key, RECTYPE rec)
{
    int i;
    i = h(key);           // hash the key
    while (table[i].k != key && table[i].k != nullkey)
        i = p(i);        // probe for available position
    if (table[i].k == nullkey)
    {
        // insert the record into the empty position
        table[i].k = key;
        table[i].r = rec;
    }
    return i;
}
```

Here:

-*h(key)* *is the function key % TABLESIZE*

-*p(i) is the function that accepts one array index and produces another.*

(This function is referred to as rehash function in Langsam et al.'s Book)

Example: *(i+1) % TABLESIZE in linear probing.*

A simple implementation

```
int search(KEYTYPE key, RECTYPE rec)
{
    int i;
    i = h(key);              // hash the key
    while (table[i].k != key && table[i].k != nullkey)
        i = p(i);          // probe for available position
    if (table[i].k == null
    {
        // insert the reco
        table[i].k = key;
        table[i].r = rec;
    }
    return i;
}
```

The number of iterations in this loop determines the efficiency of the search.
-It's over if the key is found in the table or
we identified an empty location (then we can insert the new record)

Here:
-*h(key) is the function key % TABLESIZE*
-*p(i) is the function that accepts one array index and produces another:*
***Example:** (i+1) % TABLESIZE in linear probing.*

```
int search(KEYTYPE key, RECTYPE rec)
{
    int i;
    i = h(key);           // hash the key
    while (table[i].k != key && table[i].k != nullkey)
        i = p(i);      // probe for available position
    if (table[i].k == null
    {
        // insert the reco
        table[i].k = key;
        table[i].r = rec;
    }
    return i;
}
```

Note that, this loop may execute forever!
Why?

Here:
-*h(key)* *is the function* *key % TABLESIZE*
-*p(i)* *is the function:*
(i+1) % TABLESIZE in linear probing.
(i^2) *% TABLESIZE in quadratic probing*

```c
int search(KEYTYPE key, RECTYPE rec)
{
    int i;
    i = h(key);           // hash the key
    while (table[i].k != key && table[i].k != nullkey)
        i = p(i);       // probe for available position
    if (table[i].k == nullkey)
    {
        // insert the record into the empty position
        table[i].k = key;
        table[i].r = rec;
    }
    return i;
}
```

Note that, this loop may execute forever!

1- The table can be full

- we can detect this by counting the records in the table

2- Depending on the definition of the function *p*, (e.g. p(i) = (i+2)%TABLESIZE) it is possible to loop indefinitely even if there are some (even many empty positions) in the table.

Note that, this loop may execute forever!

Assume that the function *p* is defined as follows:
p(i) = (i+200) % 1000 for a table size of 1000.

- Each key can be place in only one of the five positions.
- It is possible that these positions are full while much of the table is empty.

Note that, this loop may execute forever!

Assume that the function *p* is defined as follows:
p(i) = (i+200) % 1000 for a table size of 1000.

- Each key can be place in only one of the five positions.
- It is possible that these positions are full while much of the table is empty.

Although quadratic probing eliminate the *primary clustering*, they do not eliminate another phenomenon, known as *secondary clustering*.

secondary clustering: Different keys that hash to the same value follow the same index path.

Open Addressing -- Double Hashing

- **Double hashing** also reduces clustering.
- **Idea**: increment using a **second hash function h_2**. Should satisfy:

$$h_2(key) \neq 0$$

$$h_2 \neq h_1$$

- Probes following locations until it finds an unoccupied place

Probing function in double hashing depends on i and key values:

$f(i, key) = (i + h_2(key))$ % TABLESIZE

remember that

$i = h_1(key)$

$h_1(key)$
$h_1(key) + h_2(key)$
$h_1(key) + 2*h_2(key)$,
...

Double Hashing -- Example 1

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

    $$h_1(x) = x \bmod 11$$

    $$h_2(x) = 7 - (x \bmod 7)$$

  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
    - 14 mod 11 = 3 ➔   3+7=10
    - 91 mod 11 = 3 ➔   3+7, 3+2*7 mod 11=6

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Double Hashing -- Example 2

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

$h_1(x) = x \bmod 11$

$h_2(x) = 1 + (x \bmod t)$

, where t = *tablesize-1* *(t = 10 here)*

  - Insert keys: 58, 14, 91, 69, 80, 102, 25, 113, 124
    - 58 mod 11 = 3
    - 14 mod 11 = 3 ➔ 3+(1+4)=8 mod 11=8
    - 91 mod 11 = 3 ➔ 3+(1+1)= 5 mod 11=5
    - 69 mod 11 = 3 ➔ 3+(1+9)= 13 mod 11=2
    - 80 mod 11 = 3 ➔ 3+(1+0)= 4 mod 11=4
    - 102 mod 11 = 3 ➔ 3+(1+2)= 6 mod 11=6
    - 25 mod 11 = 3 ➔ 3+(1+5)= 9 mod 11=9
    - 113 mod 11 = 3 ➔ 3+(1+3)= 7 mod 11=7
    - 124 mod 11 = 3 ➔ 3+(1+4)= 8 (full) + (1+4) mod 11=2 (full)+ (1+4) mod 11 = 7 (full) + (1+4) mod 11 = 1 —> $3+4*h_2(124) \bmod 11$

0	
1	124
2	69
3	58
4	80
5	91
6	102
7	113
8	14
9	25
10	

Rehashing

- Hash Table may get full
  - No more insertions possible
- Hash table may get *too* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table
- Cost of Rehashing = O(N)
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter

Rehashing Example

Original Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

0	6
1	15
2	23
3	24
4	
5	
6	13

After Rehashing

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Open Addressing -- Retrieval & Deletion

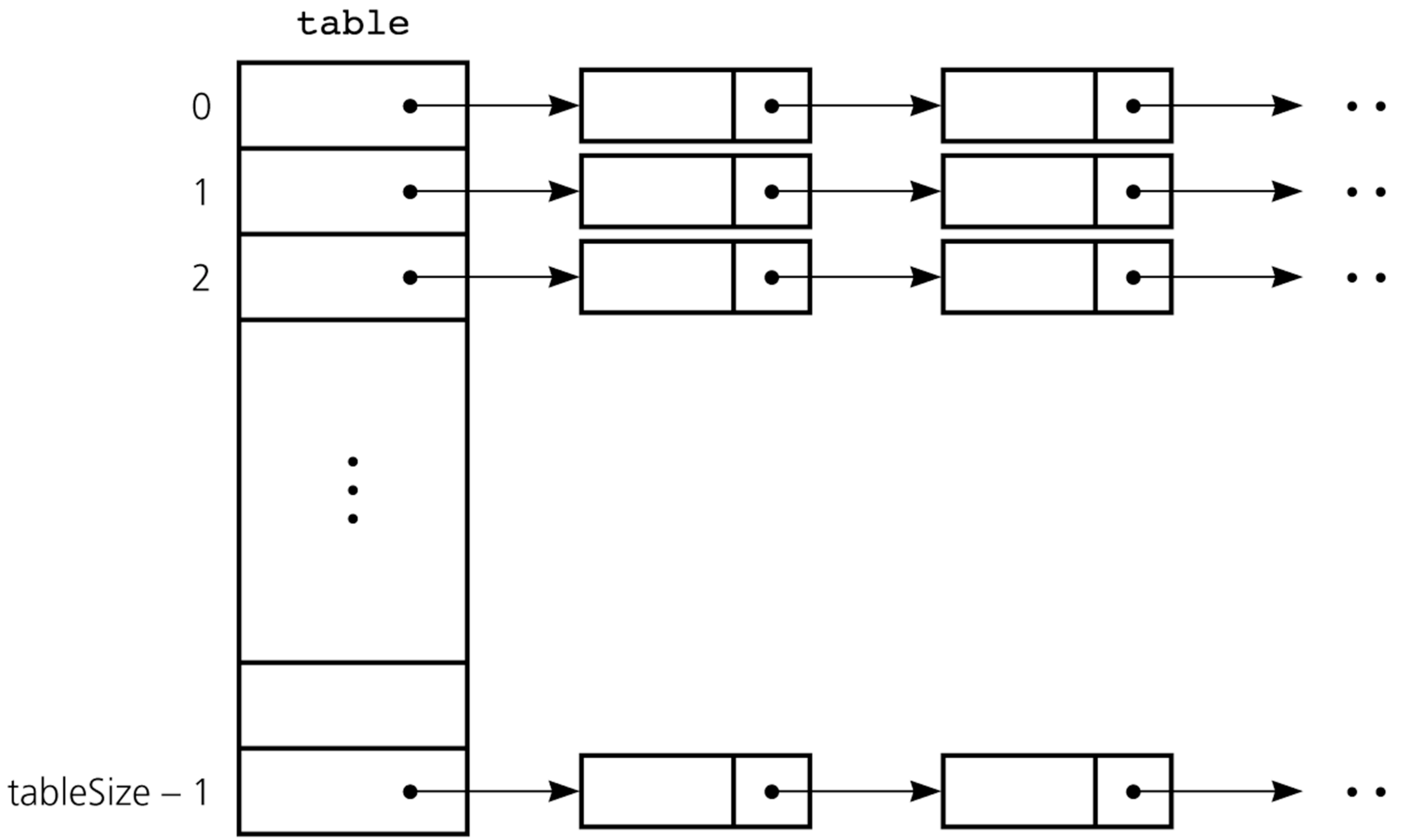
- **Retrieving an item** with a given key:
  - (same as insertion): probe the locations until we find the desired item or we reach to an empty location.

- **Deletions** in open addressing cause complications
  - We CANNOT simply delete an item from the hash table because this new empty (a deleted) location causes to stop prematurely (incorrectly) indicating a failure during a retrieval.
  - Solution: We have to have three kinds of locations in a hash table: ***Occupied, Empty, Deleted***.
  - A deleted location will be treated as an occupied location during retrieval.

Separate Chaining

- Another way to resolve collisions is to change the structure of the hash table.
  - In open-addressing, each location holds only one item.
- **Idea 1**: each location is itself an array called bucket
  - Store items that are hashed into same location in this array.
  - Problem: What will be the size of the bucket?
- **Idea 2:** each location is itself a linked list. Known as **separate-chaining**.
  - Each entry (of the hash table) is a pointer to a linked list (the chain) of the items that the hash function has mapped into that location.

Separate Chaining



Each location of the hash table contains a pointer to a linked list

Hash Table in C++ Library

- Supported in C++11 STL
- Both set and map
  - unordered_set (multiset)
  - unordered_map (multimap)
- To use, include the corresponding header files
  - <unordered_set>
  - <unordered_map>
- Examples
  - unordered_set<string> ht_str;
  - unordered_map<int, bool> ht_int;

ANALYSIS

Hashing -- Analysis

- An analysis of the average-case efficiency of hashing involves the **load factor** α*:*

  $$\alpha = \textit{(current number of items) / tableSize}$$

- α measures how full a hash table is.
  - Hash table should not be too loaded if we want to get better performance from hashing.

- In average case analyses, we assume that the hash function uniformly distributes keys in the hash table.

- Unsuccessful searches generally require more time than successful searches.

Separate Chaining -- Analysis

- **Separate Chaining –** approximate average number of comparisons (probes) that a search requires :

$$1+\frac{\alpha}{2}$$ *for a successful search*

$$\alpha$$ *for an unsuccessful search*

Note that α is the average length of a linked list in the table

- It is the most efficient collision resolution scheme.
- But it requires more storage (needs storage for pointers).
- It easily performs the deletion operation. Deletion is more difficult in open-addressing.

Linear Probing -- Analysis

- **Linear Probing –** approximate average number of comparisons (probes) that a search requires :

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

for a successful search

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

for an unsuccessful search

- Insert and search cost depend on length of cluster.
  - Average length of cluster = α = N / M.
  - Worst case: all keys hash to the same cluster.

Linear Probing -- Analysis

- **Linear Probing –** approximate average number of comparisons (probes) that a search requires :

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$ *for a successful search*

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$ *for an unsuccessful search*

- As load factor increases, number of collisions increases, causing increased search times.
- To maintain efficiency, it is important to prevent the hash table from filling up.

Linear Probing -- Analysis

Example: Find the average number of probes for a successful search and an unsuccessful search for this hash table? Use the following hash function: **h(x) = x mod 11**

Successful Search: ***Try 20, 30, 2, 13, 25, 24, 10, 9***

20: 9 **30:** 8 **2:** 2 **13:** 2,3

25: 3, 4 **24:** 2, 3, 4, 5 **10:** 10 **9:** 9, 10, 0

Avg. no of probes = (1+1+1+2+2+4+1+3)/8 = 1.9

Unsuccessful Search: ***Try 0, 1, 35, 3, 4, 5, 6, 7, 8, 31, 32***

0: 0, 1 1: 1 35: 2, 3, 4, 5, 6

3: 3, 4, 5, 6 4: 4, 5, 6 5: 5, 6

6: 6 7: 7 8: 8, 9, 10, 0, 1

31: 9, 10, 0, 1 32: 10, 0, 1

Avg. no of probes =(2+1+5+4+3+2+1+1+5+4+3)/11 = 2.8

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Quadratic Probing & Double Hashing -- Analysis

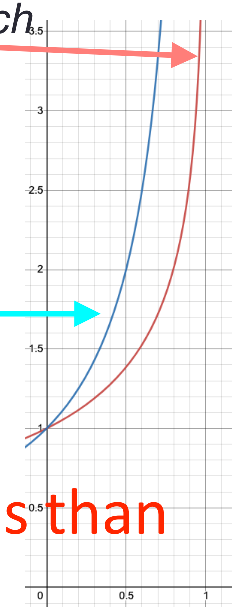
- The approximate average number of comparisons (probes) that a search requires is given as follows:

$$\left[\frac{1}{\alpha}\left(\log_e \frac{1}{1-\alpha}\right)\right] = \frac{-\log_e(1-\alpha)}{\alpha}$$

for a successful search

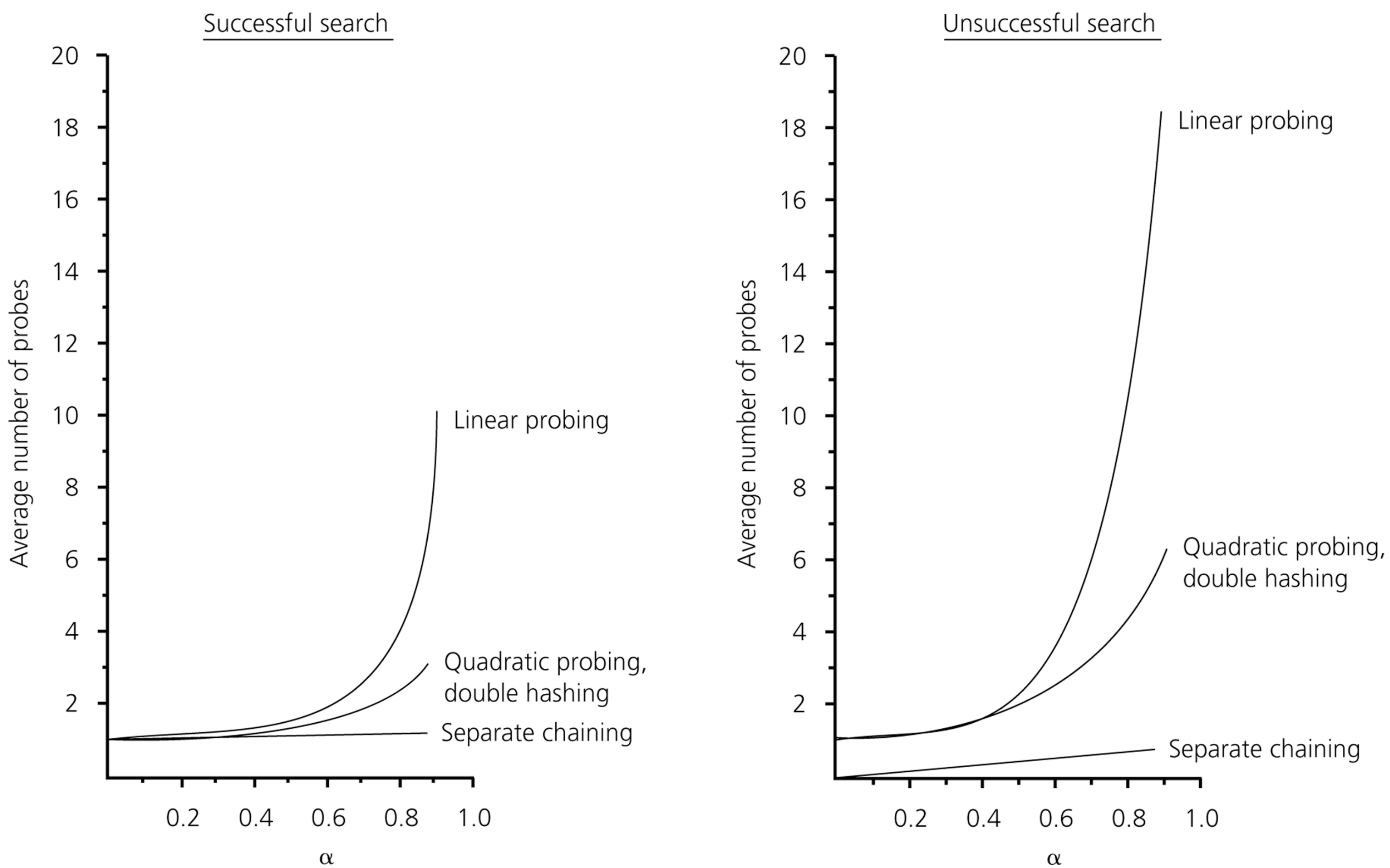
$$\frac{1}{1-\alpha}$$

for an unsuccessful search



- On average, both methods require fewer comparisons than linear probing.

The relative efficiency of four collision-resolution methods



What Constitutes a Good Hash Function

- A hash function should be **easy** and **fast** to compute.
- A hash function should **scatter the data evenly** throughout the hash table.
  - How well does the hash function scatter random data?
  - How well does the hash function scatter non-random data?
- Two general principles :
  1. The hash function should use entire key in the calculation.
  2. If a hash function uses modulo arithmetic, the table size should be prime.

Example: Hash Functions for Strings

Hash Function 1

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
    int hasVal = 0;

    for (int i = 0; i < key.length(); i++)
        hashVal += key[i];
    return hashVal % tableSize;
}
```

- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.
  - e.g. Table size =10000, key length <= 8, the hash function can assume values only between 0 and 1016

Hash Function 2

- Examine only the first 3 characters of the key.

```
int hash (const string &key, int tableSize)
{
    return (key[0]+27 * key[1] + 729*key[2]) % tableSize;
}
```

- In theory, **26** * **26** * **26** = **17576** different words can be generated. However, English is not random, only **2851** different combinations are possible.
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

This is simply a polynomial in *a* that takes the components (x_0, x_1, ..., x_{k-1}) of an object *x (key[i])* as its coefficients

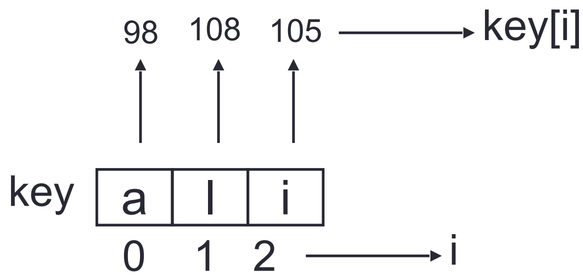
Hash Function 3

$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

```
int hash (const string &key, int tableSize)
{
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++)
     hashVal = 37 * hashVal + key[i];

    hashVal %=tableSize;
    if (hashVal < 0)    /* in case overflows occurs */
     hashVal += tableSize;

    return hashVal;
};
```

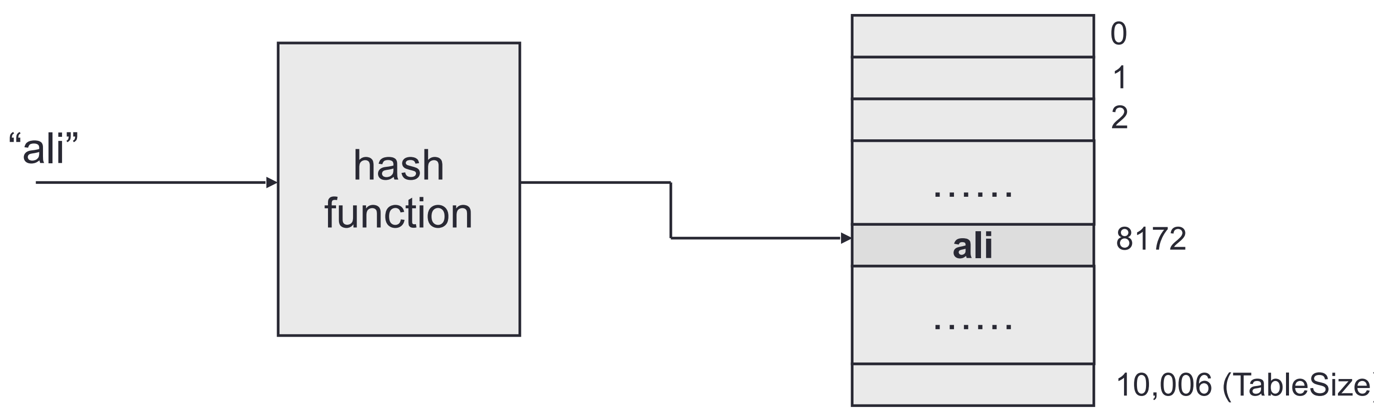
Using prime numbers in multiplications generate better hash values:
As stated in https://www.cpp.edu/~ftang/courses/CS240/lectures/hashing.htm
Experiments have shown that 33, 37, 39, or 41 are particularly good choices when working with character strings that are English words. In a list of over 50,000 English words, taking *a* to be 33, 37, 39, or 41 produced less than 7 collisions in each case.

Hash function for strings:



KeySize = 3;

hash("ali") = $(105 * 1 + 108*37 + 98*37^2) \% 10{,}007 = 8172$



Hash Table versus Search Trees

- In most of the operations, the hash table performs better than search trees.
- However, traversing the data in the hash table in a sorted order is very difficult.
  - For similar operations, the hash table will not be good choice (e.g., finding all the items in a certain range).