Hashing

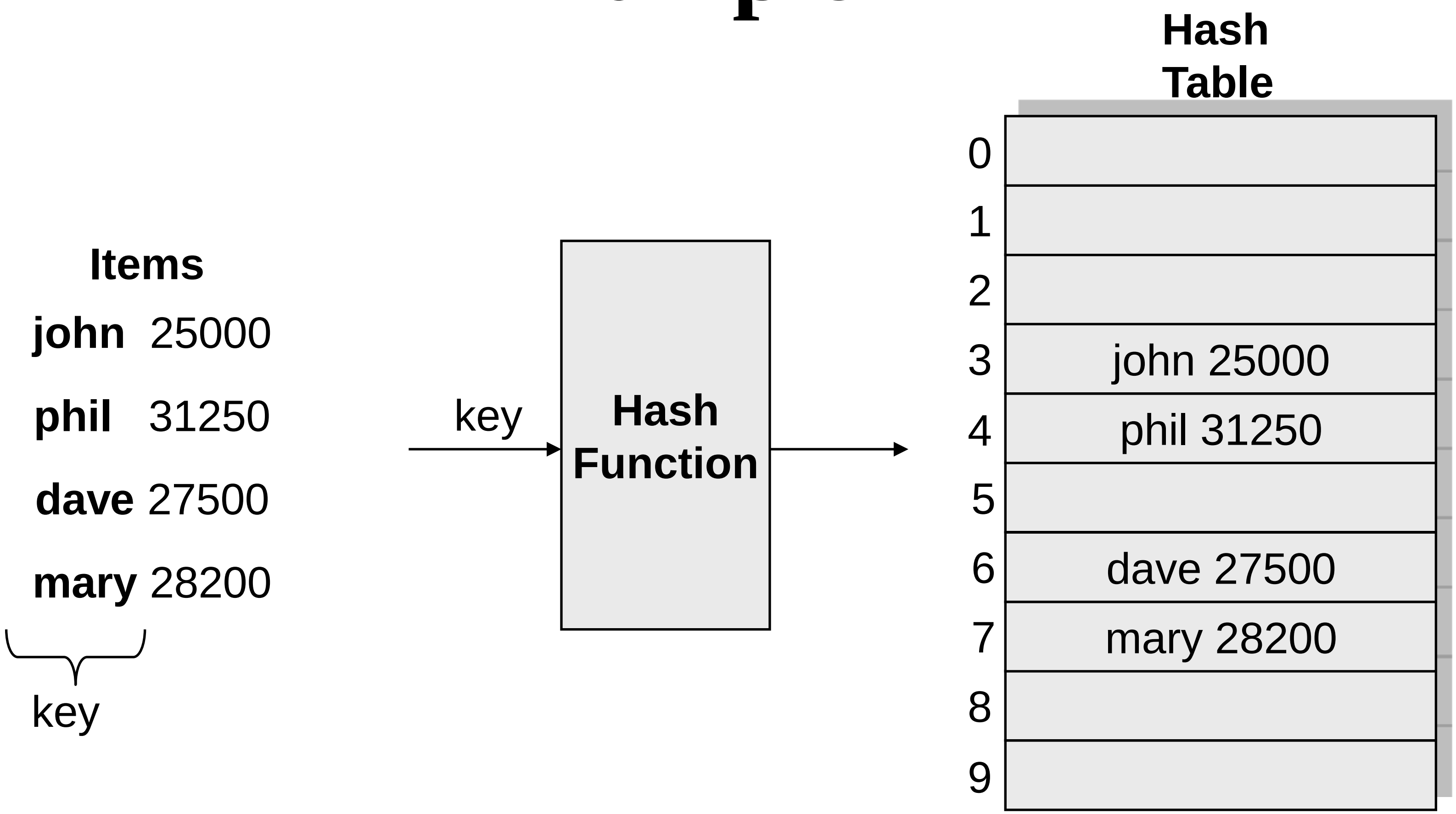
Hash Tables

- We’ll discuss the *hash table* ADT which supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called **hashing**.
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1*.
- The mapping is called a *hash function*.

Example


Hash
Table
Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash
Function
0
1
2
3 john 25000
4 phil 31250
5
6 dave 27500
7 mary 28200
8
9

Hash Function

- The hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but we don't.

Hash function

Problems:

- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Some methods

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

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;

}
```

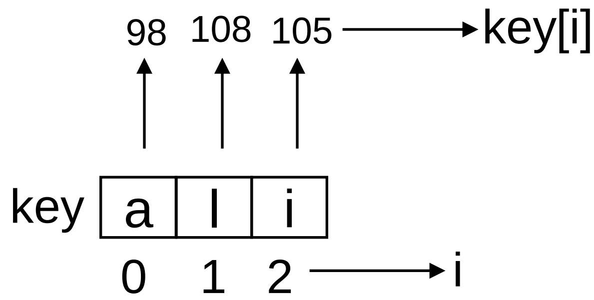
- In theory, **26 * 26 * 26** = **17576** different words can be generated. However, English is not random, only **2851** different combinations are possible.
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

Hash Function 3

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

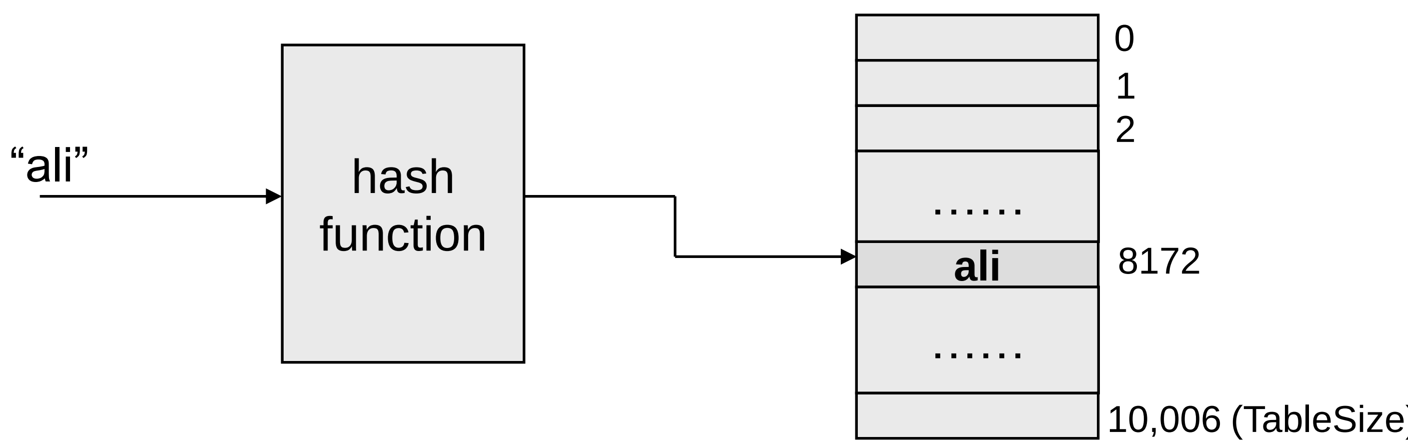
```
int hash (const string &key, int tableSize)
{
   int hashVal = 0;

   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;
};
```

Hash function for strings:



KeySize = 3;

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



Collision Resolution

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

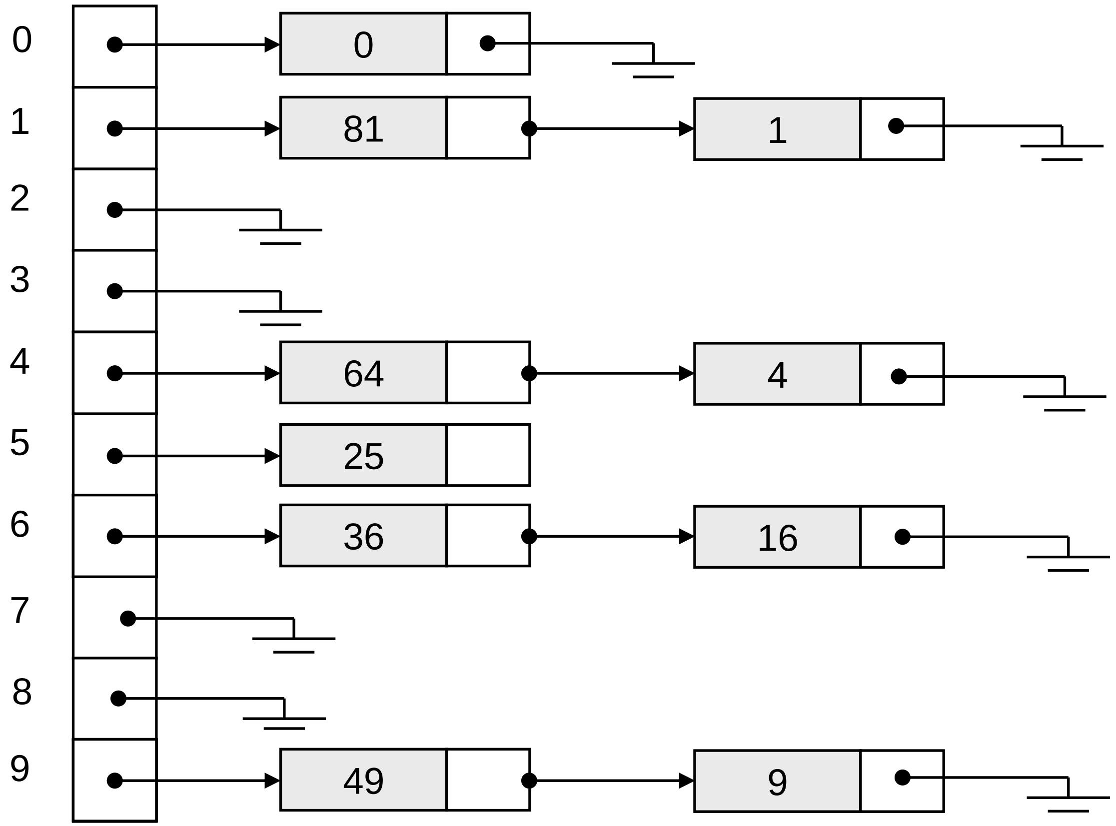
Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.



Operations

- **Initialization**: all entries are set to NULL
- **Find**:
  - locate the cell using hash function.
  - sequential search on the linked list in that cell.
- **Insertion**:
  - Locate the cell using hash function.
  - (If the item does not exist) insert it as the first item in the list.
- **Deletion**:
  - Locate the cell using hash function.
  - Delete the item from the linked list.

Analysis of Separate Chaining

- Collisions are very likely.
  - How likely and what is the average length of lists?
- Load factor λ definition:
  - Ratio of number of elements (N) in a hash table to the hash *TableSize*.
    - i.e. $\lambda = N/TableSize$
  - The average length of a list is also λ.
  - For chaining λ is not bound by 1; it can be > 1.

Cost of searching

- **Cost** = Constant time to evaluate the hash function + time to traverse the list.
- **Unsuccessful search**:
  - We have to traverse the entire list, so we need to compare λ nodes on the average.
- **Successful search**:
  - List contains the one node that stores the searched item + 0 or more other nodes.
  - Expected # of other nodes = x = (N-1)/M which is essentially λ, since M is presumed large.
  - On the average, we need to check *half* of the *other nodes* while searching for a certain element
  - Thus average search cost = 1 + $\lambda/2$

Summary

- The analysis shows us that the table size is not really important, but the load factor is.
- TableSize should be as *large* as the number of expected elements in the hash table.
  - To keep load factor around 1.
- TableSize should be *prime* for even distribution of keys to hash table cells.

Hashing: Open Addressing

Collision Resolution with Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
    - Generally the load factor should be below 0.5.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

Open Addressing

- More formally:
  - Cells $h_0(x), h_1(x), h_2(x), \ldots$ are tried in succession where $h_i(x) = (hash(x) + f(i))$ mod $TableSize$, with $f(0) = 0$.
  - The function f is the collision resolution strategy.
- There are three common collision resolution strategies:
  - Linear Probing
  - Quadratic probing
  - Double hashing

Linear Probing

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
  - i.e. f is a linear function of i, typically $f(i)= i$.
- Example:
  - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
  - Hash function is hash(x) = x mod 10.
    - $f(i) = i$;

Figure 20.4
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Find and Delete

- The find algorithm follows the same probe sequence as the insert algorithm.
  - A find for 58 would involve 4 probes.
  - A find for 19 would involve 5 probes.
- We must use *lazy deletion* (i.e. marking items as deleted)
  - Standard deletion (i.e. physically removing the item) cannot be performed.
  - e.g. remove 89 from hash table.

Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Analysis of insertion

- The average number of cells that are examined in an insertion using linear probing is roughly

$$(1 + 1/(1 - \lambda)^2) / 2$$

  - Proof is beyond the scope of text book.
- For a half full table we obtain 2.5 as the average number of cells examined during an insertion.
- Primary clustering is a problem at high load factors. For half empty tables the effect is not disastrous.

Analysis of Find

- An unsuccessful search costs the same as insertion.
- The cost of a successful search of X is equal to the cost of inserting X at the time X was inserted.
- For $\lambda = 0.5$ the average cost of insertion is 2.5. The average cost of finding the newly inserted item will be 2.5 no matter how many insertions follow.
- Thus the average cost of a successful search is an average of the insertion costs over all smaller load factors.

Average cost of `find`

- The average number of cells that are examined in an unsuccessful search using linear probing is roughly $(1 + 1/(1 - \lambda)^2) / 2$.
- The average number of cells that are examined in a successful search is approximately $(1 + 1/(1 - \lambda)) / 2$.
  - Derived from:

$$\frac{1}{\lambda}\int_{x=0}^{\lambda}\frac{1}{2}\left(1+\frac{1}{(1-x)^2}\right)dx$$

Linear Probing – Analysis -- Example

- What is the average number of probes for a successful search and an unsuccessful search for this hash table?
  - Hash Function: h(x) = x mod 11

<u>Successful Search:</u>

- 20: 9 -- 30: 8 -- 2 : 2 -- 13: 2, 3 -- 25: 3,4
- 24: 2,3,4,5 -- 10: 10 -- 9: 9,10, 0

Avg. Probe for SS = (1+1+1+2+2+4+1+3)/8=15/8

<u>Unsuccessful Search:</u>

- We assume that the hash function uniformly distributes the keys.
- 0: 0,1 -- 1: 1 -- 2: 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
- 9: 9,10,0,1 -- 10: 10,0,1

Avg. Probe for US =

(2+1+5+4+3+2+1+1+5+4+3)/11=31/11

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

Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- Collision function is quadratic.
  - The popular choice is $f(i) = i^2$.
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2$, $h+2^2$, … $h + i^2$.
  - i.e. It examines cells 1,4,9 and so on away from the original probe.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

Figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic Probing

- Problem:
  - We may not be sure that we will probe all locations in the table (i.e. there is no guarantee to find an empty cell if table is more than half full.)
  - If the hash table size is not prime this problem will be much severe.
- However, there is a theorem stating that:
  - If the table size is *prime* and load factor is not larger than 0.5, all probes will be to different locations and an item can always be inserted.

Theorem

- If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Some considerations

- How efficient is calculating the quadratic probes?
  - Linear probing is easily implemented. Quadratic probing appears to require * and % operations.
  - However by the use of the following trick, this is overcome:
    - $H_i = H_{i-1} + 2i - 1 \pmod{M}$

Some Considerations

- What happens if load factor gets too high?
  - Dynamically expand the table as soon as the load factor reaches 0.5, which is called *rehashing*.
  - Always double to a prime number.
  - When expanding the hash table, reinsert the new table by using the new hash function.

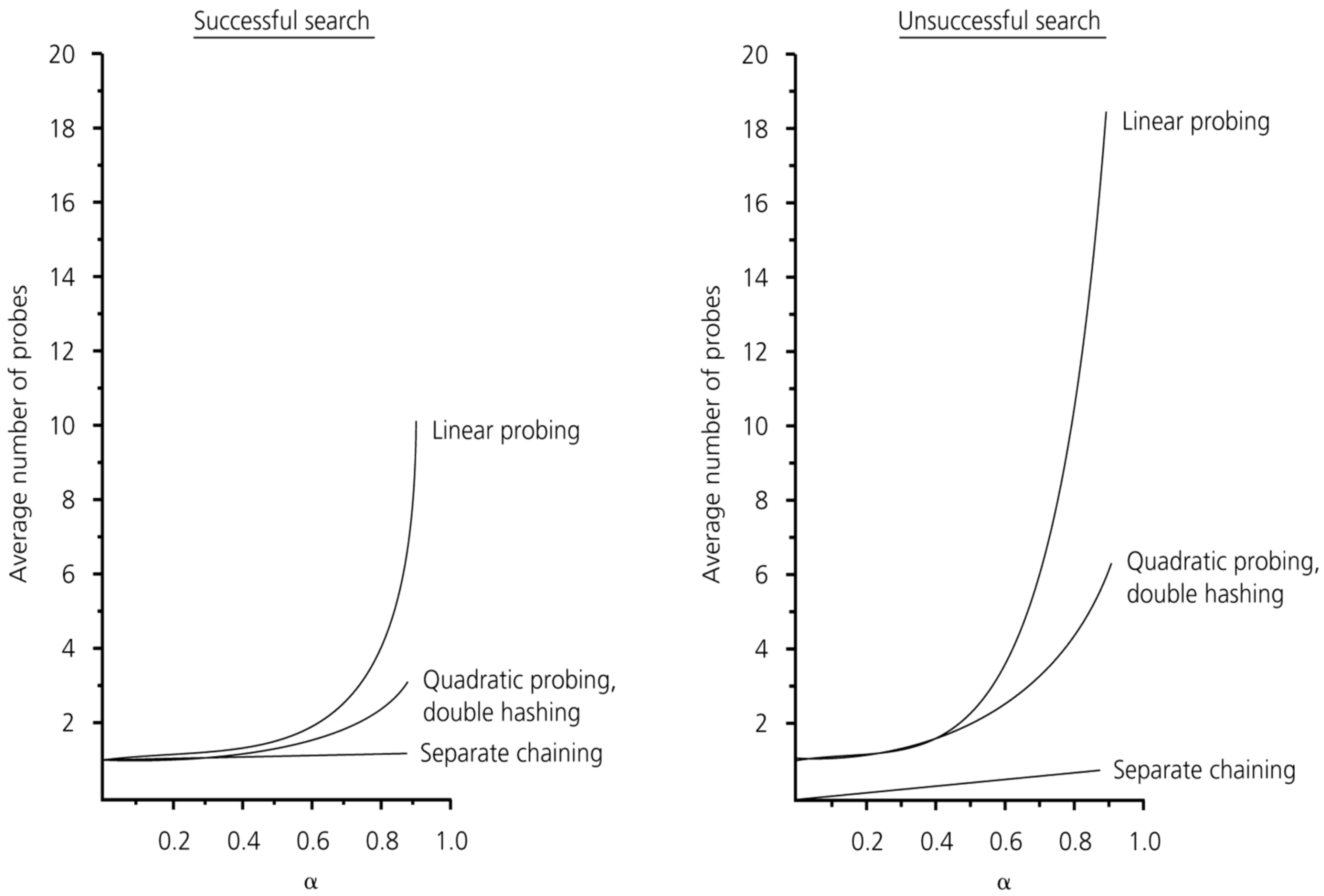
Analysis of Quadratic Probing

- Quadratic probing has not yet been mathematically analyzed.
- Although quadratic probing eliminates primary clustering, elements that hash to the same location will probe the same alternative cells. This is know as *secondary clustering*.
- Techniques that eliminate secondary clustering are available.
  - the most popular is *double hashing*.

Double Hashing

- A second hash function is used to drive the collision resolution.
  - $f(i) = i * hash_2(x)$
- We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.
- The function $hash_2(x)$ must never evaluate to zero.
  - e.g. Let $hash_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R - (x \bmod R)$ with R a prime smaller than TableSize will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)

The relative efficiency of four collision-resolution methods



Hashing Applications

- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Summary

- Hash tables can be used to implement the insert and find operations in constant average time.
  - it depends on the load factor not on the number of items in the table.
- It is important to have a prime TableSize and a correct choice of load factor and hash function.
- For separate chaining the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.