Hashing

So Far We've Learned

Data Structure	add	search	remove
Unsorted array	O(1)	O(N)	O(N)
Unsorted LL	O(1)	O(N)	O(N)
Sorted array	O(N)	O(log N)	O(N)
Balanced BST (AVL)	O(log N)	O(log N)	O(N)
Heap	O(log N)	-	O(log N)
???	O(1)	O(1)	O(1)

Hash Table

- **Surprisingly Fast**
  - Hashing is a technique used to perform insertions, deletions and finds in constant average time (i.e. O(1))
- **Surprisingly Simple**
  - Easy to implement
- **Not perfect. What is the catch?**
  - Not efficient in operations that require any ordering information among the elements, such as *findMin*, *findMax* and printing the entire table in sorted order.
  - Need to have a good idea about the number of elements. Difficult to re-size dynamically
  - Performance degrades when it is close to full

A First Idea

- Here is a trivial way to achieve O(1): say we want to store student records based on student ID.
- Since each student ID is 7-digit long, let's have an array with capacity 10,000,000 so there is one entry for each possible ID (0 to 9,999,999)
- Add/Search/Remove all cost just O(1)
- What is the problem with this approach?

- **Not all 7-digit integers are valid student IDs.**
- **Many entries are empty. This is a huge waste of storage!**

Hashing

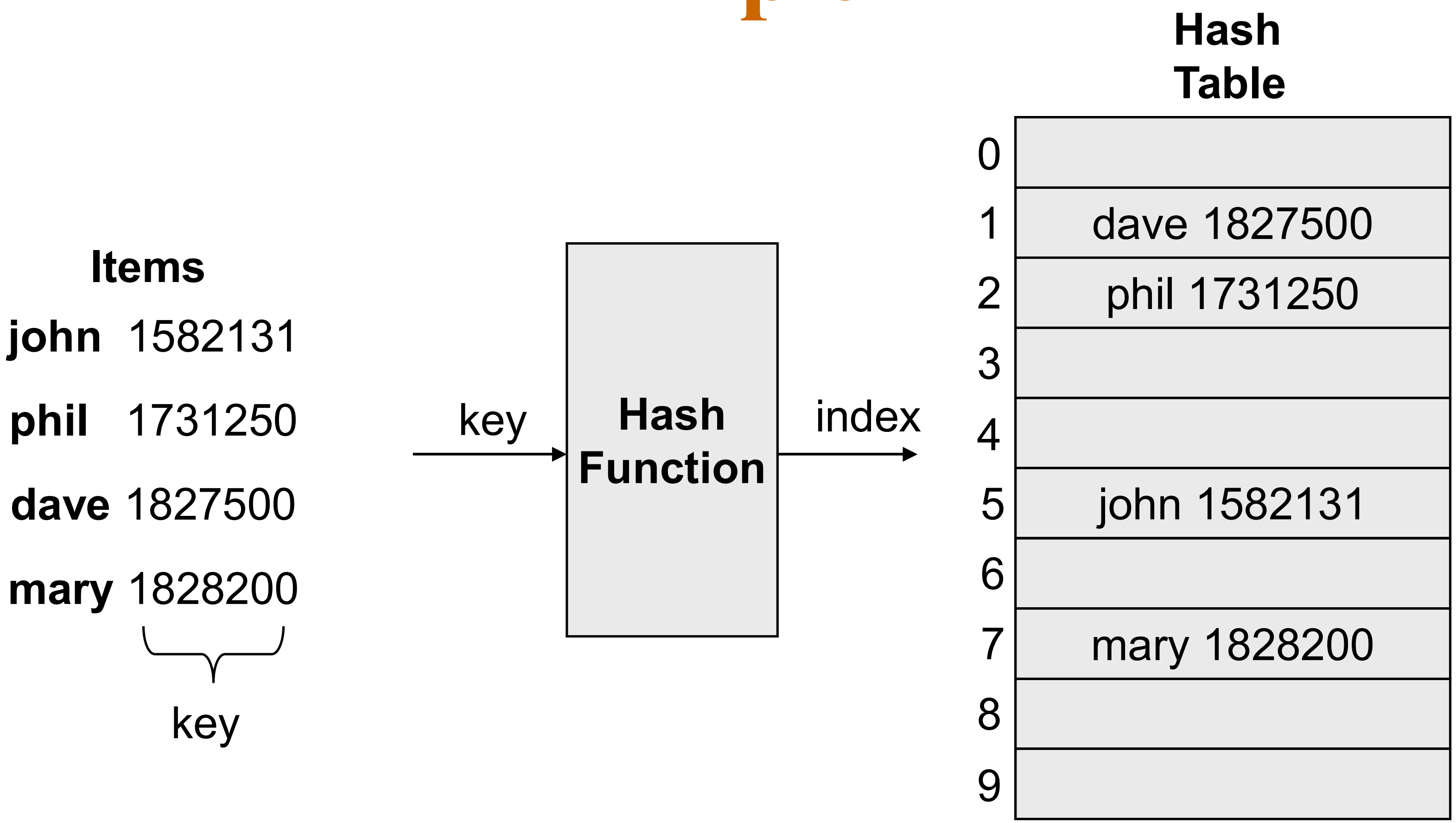
- **Hashing**: To map a key value (which can span a wide range) to an index (which has a much smaller range)
  - Sparsity is characteristic of such data, such as credit card numbers, dictionary words.
  - *Idea:* Store any given element value in a particular predictable index.
    - That way, adding / removing / looking for it are constant-time (O(1)).
- **Hash table**: An array that stores elements via hashing.

Hashing

- How do we map the huge range of possible key values to a smaller range so that they can be stored in a reasonably sized array?
  - E.g. Map 7-digit ID (key) to an array of size 20000. There can be many different ways, what's the simplest?
- Use modulo (%)

  `index = key % array_size`
- This is called a **hash function**.
- Note that the `array_size` must be at least the number of elements (e.g. # of students, but often a few times bigger.

Example



Hash
Table
Items
john 1582131
phil 1731250
dave 1827500
mary 1828200
key
key
Hash
Function
index
0
1
dave 1827500
2
phil 1731250
3
4
5
john 1582131
6
7
mary 1828200
8
9

Collisions

- One obvious problem is that multiple keys can map to the same index
- Assume array_size = 50000
  - 23245467 % 50000 = 45467
  - 43345467 % 50000 = 45467
- In fact (45467 + 50000 x k) % 50000 = 45467 for any positive integer k!
- This is called **collision**. It can be reduced by using a better hash function (e.g. array size should always be a prime number) But it cannot be avoided.

Hash function

- The hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.

Problems:

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

Hash Function 1

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

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

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

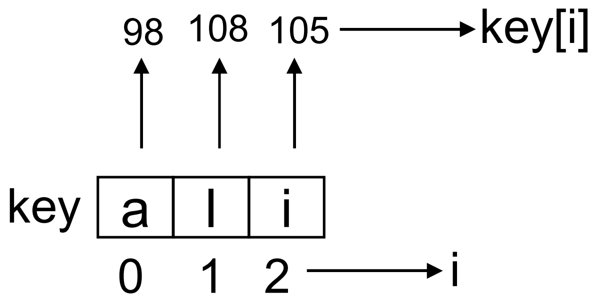
- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.
  - e.g. Table size =10000, key length <= 8, the hash function can assume values only between 0 and 1016

Hash Function 2

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

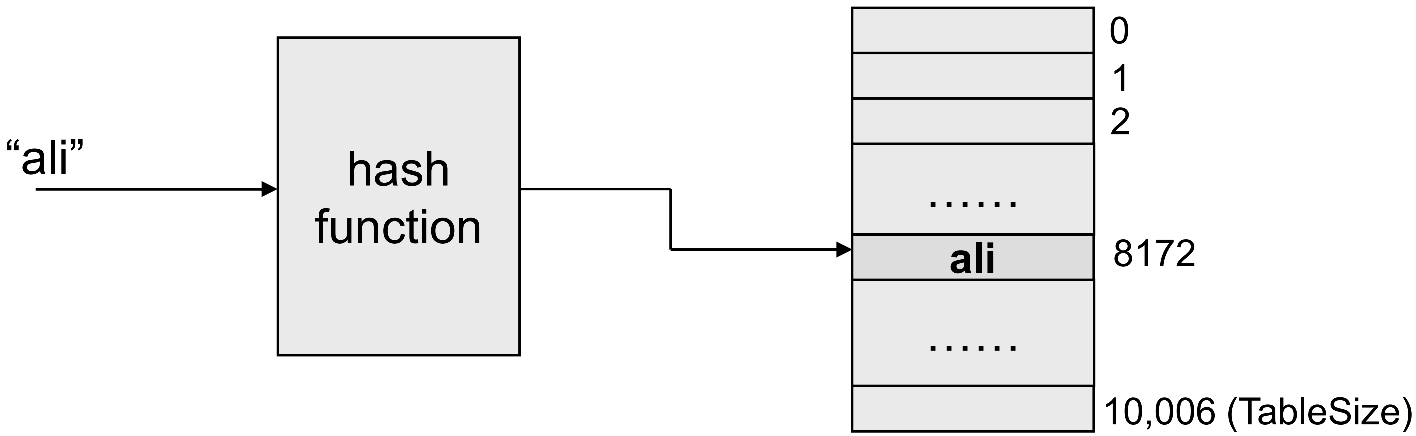
```
int hash(const string &key, int tableSize)
{
   int hashVal = 0;

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

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

   return hashVal;
};
```

Hash function for strings:



KeySize = 3;

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



Some other 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.

Collision Resolution

- There are empty slots in the hash table to store collided elements, because array size is at least the number of elements.
- We need a systematic way to search for such empty slots when collision happens. There are two general approaches:
  - **Open addressing**
  - **Separate chaining**

Open addressing

- In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.
- There are three common collision resolution strategies:
  1. **Linear Probing**: Moves to the next available index (wraps if needed).
  2. **Quadratic probing**: moves increasingly far away: +1, +4, +9, ...
  3. **Double hashing**: moving step size is determined by another hash function

Linear Probing

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.

  index, index +1, index +2, … (the array is circular, so back to 0 when reaching the end of the array)
- Example:
  - Insert items : 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
  - Hash function is hash(x) = x % 10.

Example

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

Searching

- When searching for an element, probe linearly until either the element is found, or an empty slot is encountered.
- The search 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.

Deletion

- We cannot delete an element by simply setting it to null. This will affect your search. Why?

- We must use *lazy deletion* (i.e. marking items as deleted)
  - We don't set the deleted element to null, instead we set a special flag to indicate it's gone. Insertion can overwrite a flagged slot; but search must continue across it.

Load Factor and Rehashing

- Load factor λ : number of elements (N) divided by the table size (capacity).
  - i.e. $\lambda = \dfrac{N}{TableSize}$
- **Example:** if a table has a capacity of 73, currently storing 40 elements, the load factor is: 40/73 = 56%
- When load factor becomes too large (close to 1, i.e. table getting full), it is necessary to increase hash table capacity. This is called **re-hashing.**
- You need to rehash every element to its new location in the new hash table. You can't simply copy elements over. Why?

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 % 11

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

<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

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.
- Thus insertion is O(1).

Analysis of Search

- 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 searching

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

- Thus search is O(1). So is deletion.

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.

Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- When collision happens, check the succeeding slots in **quadratic** steps:

  index, index+1, index+4, index+9, index+16, index+25 …

- Using increasingly larger steps reduce the possibility of forming clusters.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

Example

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.

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:
    - $H_i = (H_{i-1} + 2i - 1) \% M$

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*.

Double Hashing

- The problem with linear and quadratic probing is that once keys collide, they all follow exactly the same probing path, completely predictable.

- We need a way to generate probe steps that vary and not pre-determined.

- The solution is to pick a **probe size that varies depending on the key value**. This can be achieved using a second hashing function, thus the name "double hashing"

stepsize = $hash_2$(key)
index, index+stepsize, index+2*stepsize, index+3* stepsize, …

Double Hashing

- 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 good choice for secondary hashing:

  hash$_2$(key) = constant – (key % constant)

with constant being a prime smaller than TableSize.

e.g. hash$_2$(key) = 5 – (key % 5)   // stepsize

Collision Resolution

Two general approaches:

1. Open addressing
    - Linear probing
    - Quadratic probing
    - Double Hashing
2. **Separate chaining**

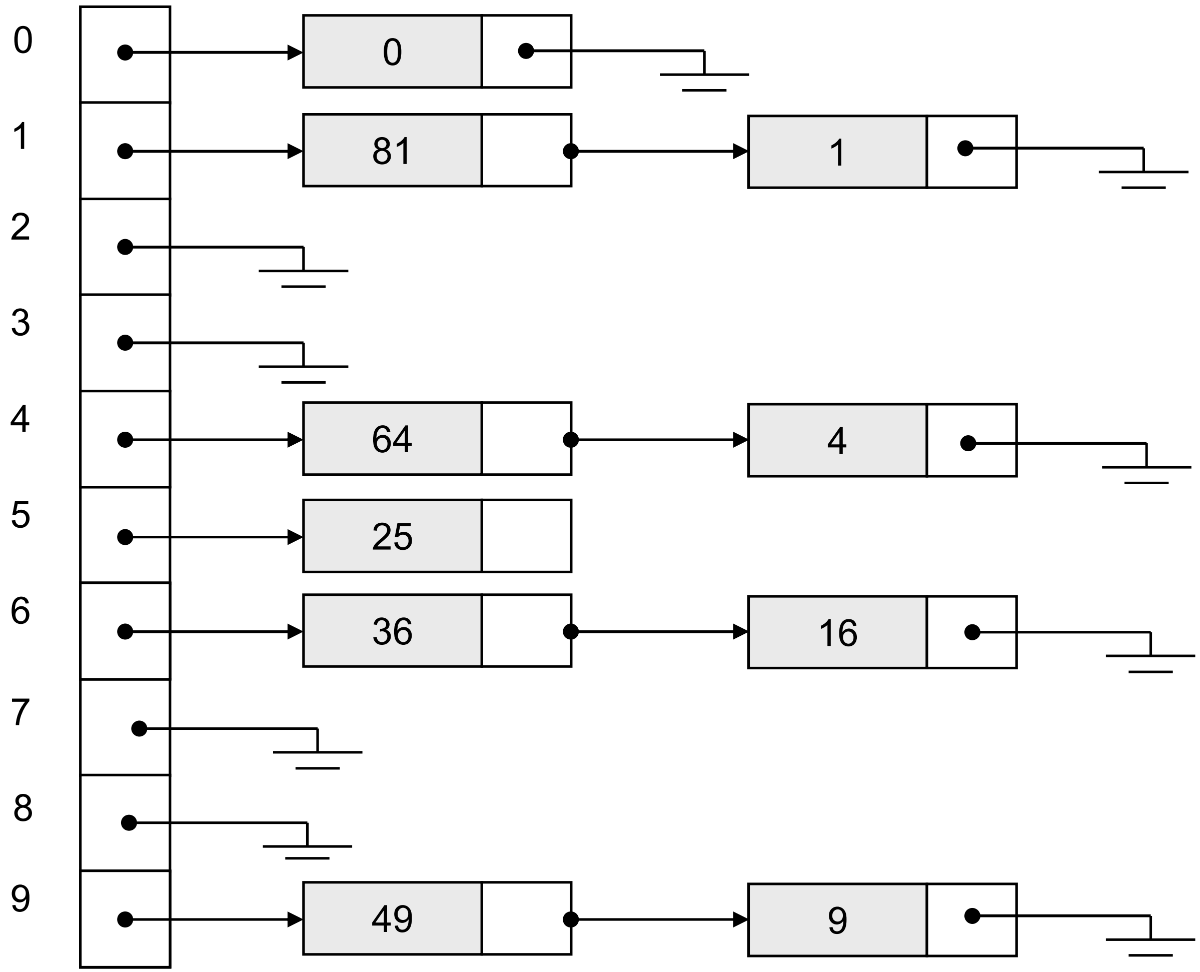
Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - Each index in array is a "bucket" which is commonly implemented as a Linked List.
  - 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

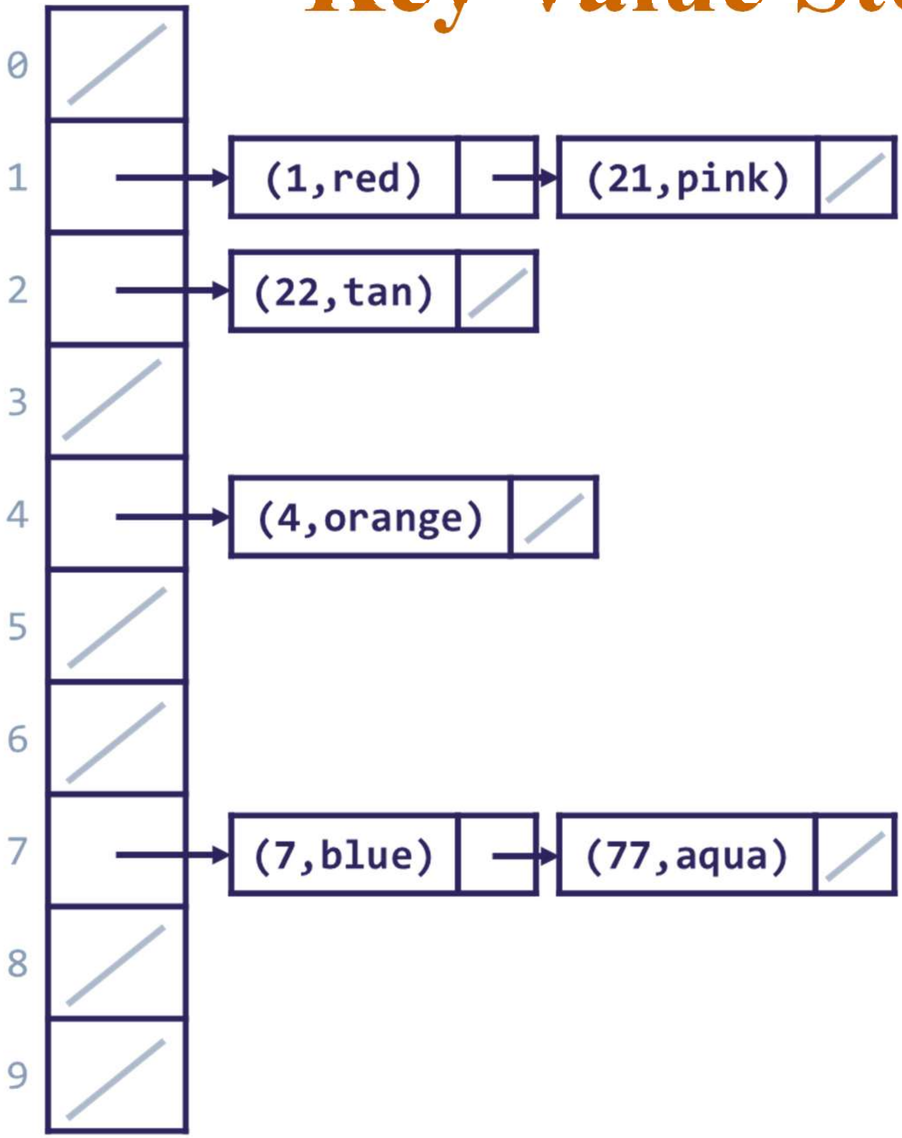
hash(key) = key % 10.



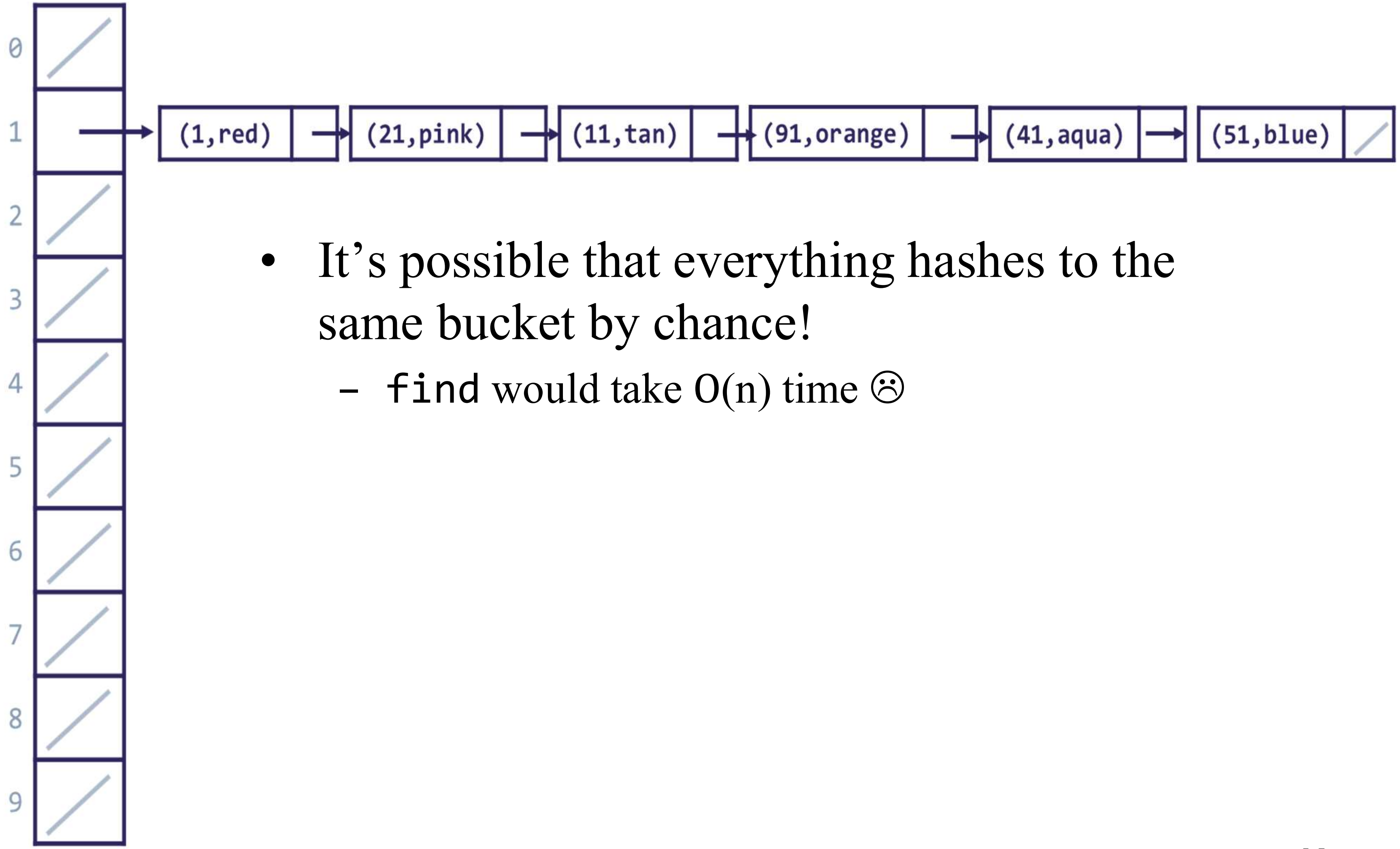
Operations

- **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.

Key-value Store

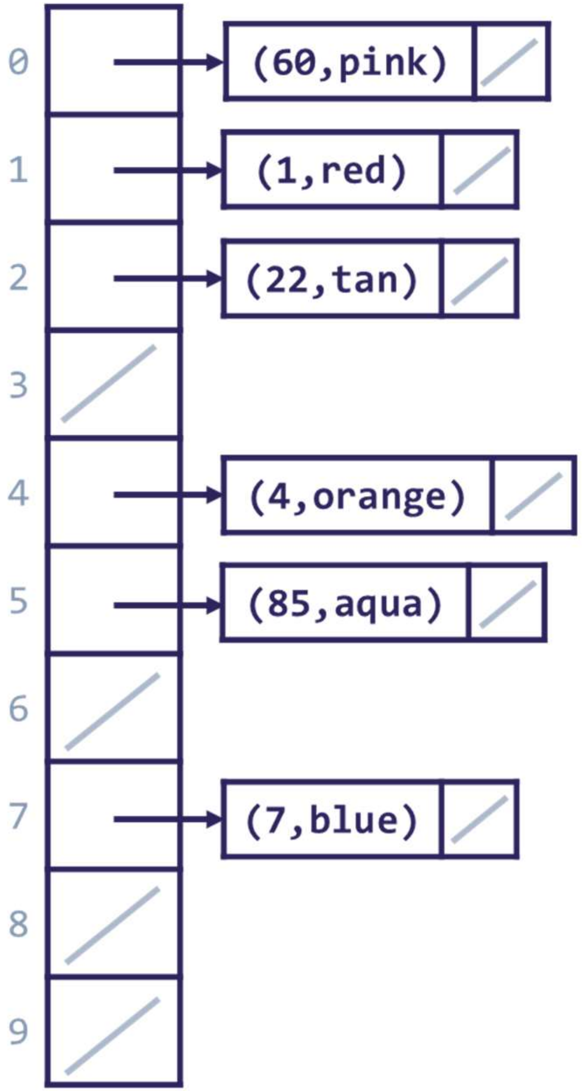


Separate Chaining Worst Case



- It's possible that everything hashes to the same bucket by chance!
  - `find` would take O(n) time ☹

Separate Chaining Best Case



- However, if everything is spread evenly across the buckets, find takes O(1)

Separate Chaining Average Case

- What is the average length of lists?
- Load factor λ : Ratio of number of elements (N) in a hash table to the hash *TableSize*.
  - i.e. $\lambda = \frac{N}{TableSize}$

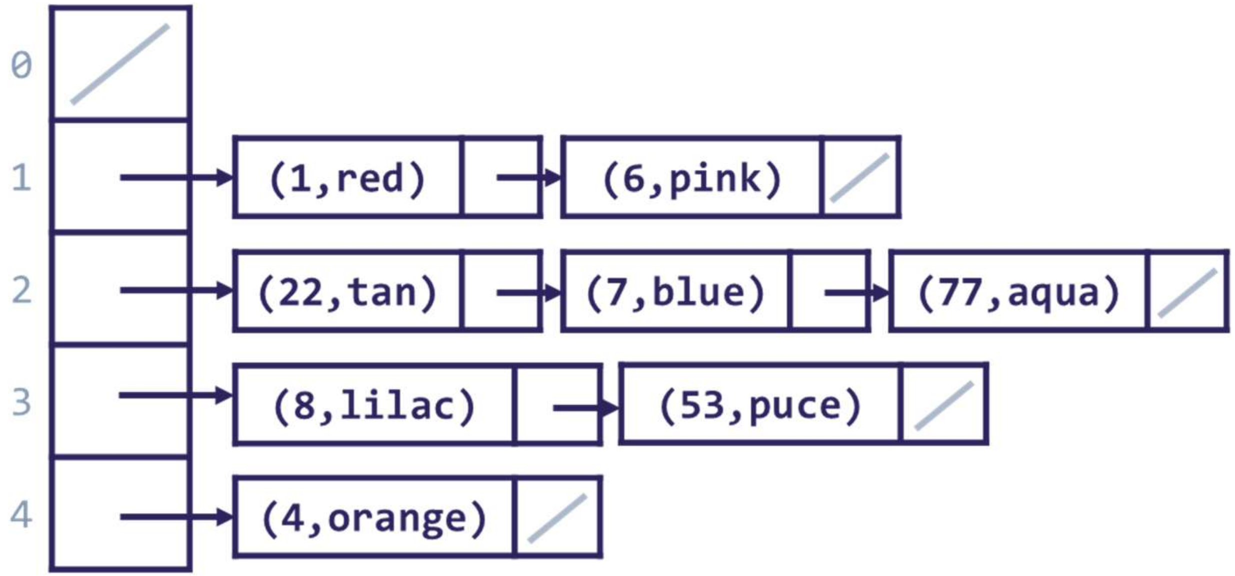
  – The average length of a list is also λ.
  – For chaining λ is not bound by 1; it can be > 1.

Separate Chaining in Practice

- A well-implemented separate chaining hash table will stay very close to the best case
  - Most of the time, operations are fast. Rarely, do an expensive operation that restores the table close to best case.
- How to stay close to best case?
  - Good distribution & Resizing!
- We can describe the "in-practice" case as what *almost always* happens:
  - (1) items are fairly evenly distributed
  - (2) assume resizing doesn't occur

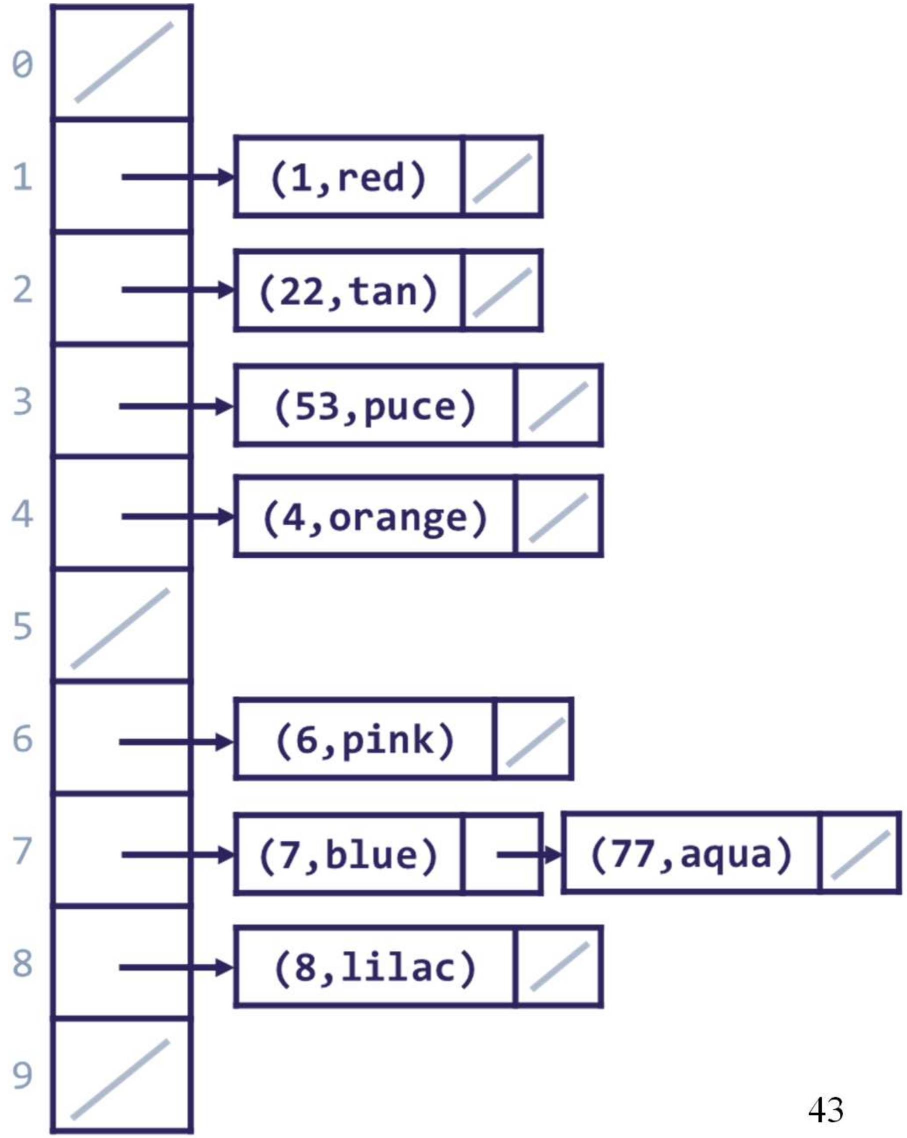
Resizing

- The runtime to scan each bucket is slowly growing up.
  - If we don't intervene, our in-practice runtime is going to hit O(n)
    - number of buckets is a constant, so N / TableSize is O(n)



How to Resize

1. Expand the HashTable (array)
2. **For every element in the old hash table, re-distribute**! Recompute its position by taking the mod with the new length



When to Resize?

- We want to make sure the buckets don't get "too full" for good runtime
- How do we quantify "too full"?
  - Look at the load factor λ(=average list length)
- If we resize when λ hits some *constant* value like 1:
  - We expect to see 1 element per bucket: **constant runtime**!
  - If we double the capacity each time, the expensive resize operation becomes less and less frequent

Hash Table Class for separate chaining

```
template <class HashedObj>
class HashTable
{
  public:
    HashTable(const HashedObj & notFound, int size=101 );
    HashTable( const HashTable & rhs )
      :ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ),
       theLists( rhs.theLists ) { }

    const HashedObj & find( const HashedObj & x ) const;

    void makeEmpty( );
    void insert( const HashedObj & x );
    void remove( const HashedObj & x );
    const HashTable & operator=( const HashTable & rhs );

  private:
    vector<List<HashedObj> > theLists;  // The array of Lists
    const HashedObj ITEM_NOT_FOUND;
};
int hash( const string & key, int tableSize );
int hash( int key, int tableSize );
```

Insert routine

```
/**
 * Insert item x into the hash table. If the item is
 * already present, then do nothing.
 */
template <class HashedObj>
void HashTable<HashedObj>::insert(const HashedObj & x )
{
   List<HashedObj> & whichList =
             theLists[ hash(x, theLists.size()) ];
  HashedObj* p = whichList.find( x );

   if( p == NULL )
     whichList.insert( x, whichList.zeroth( ) );
}
```

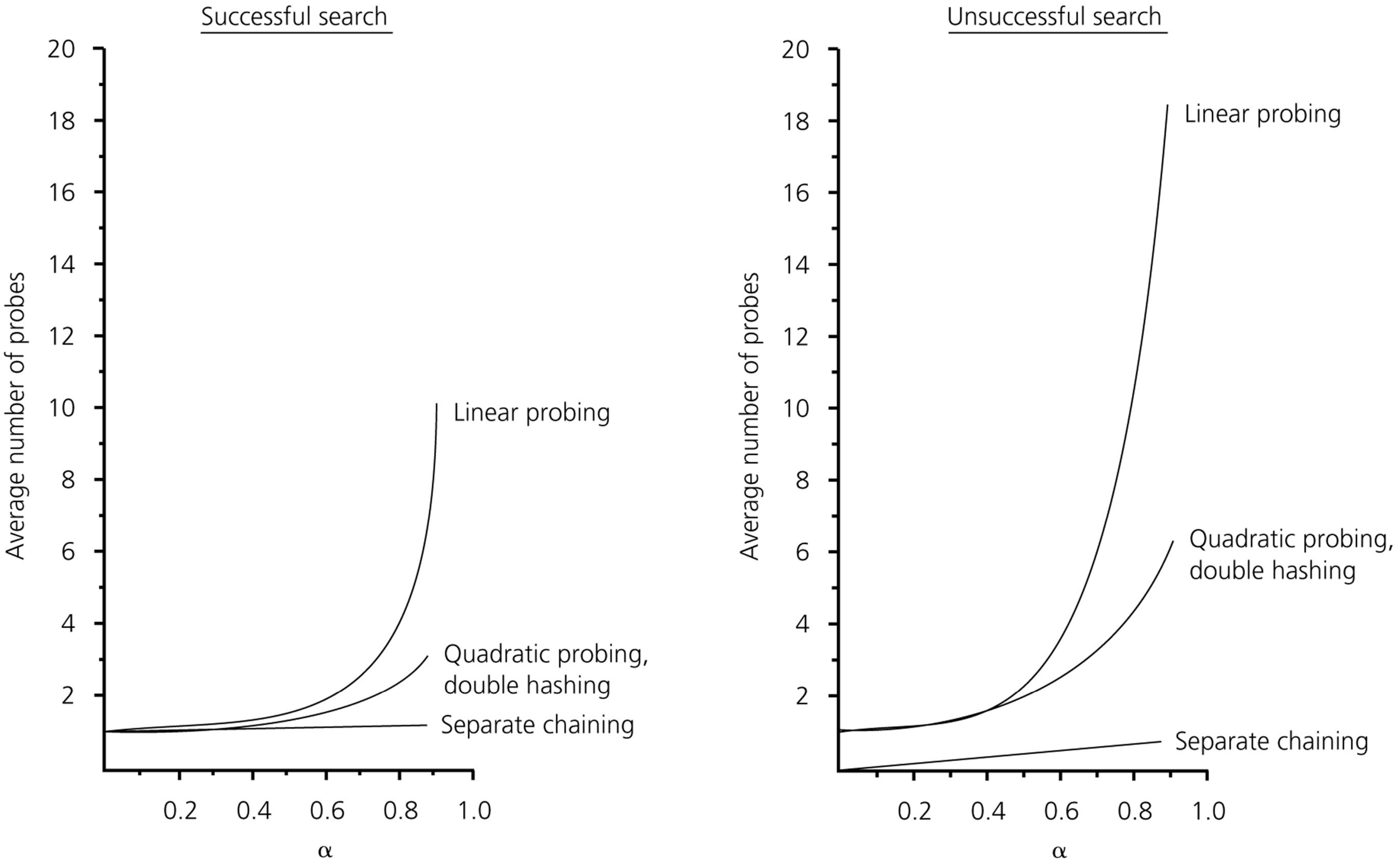
Remove routine

```
/**
 * Remove item x from the hash table.
 */
template <class HashedObj>
void HashTable<HashedObj>::remove( const HashedObj & x )
{
   theLists[hash(x, theLists.size())].remove( x );
}
```

Find routine

```
/**
 * Find item x in the hash table.
 * Return the matching item or ITEM_NOT_FOUND if not found
 */
template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const
   HashedObj & x ) const
{
    HashedObj * itr;
   itr = theLists[ hash(x, theLists.size()) ].find( x );
   if(itr==NULL)
     return ITEM_NOT_FOUND;
   else
     return *itr;
}
```

The relative efficiency of four collision-resolution methods



Examples of 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.

Example Problem

- Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

Possible solutions:

1. Check each pair of integers using a nested for loop : $O(N^2)$
2. Sort nums and scan the array from both end using two pointers: O(N log N)
3. Use a hash table : O(N) (but you need extra O(N) space)

Algorithm

```
HashTable<int> s
for(i=0 to end)
  if(!s.find(target - nums[i]))
    s.insert(nums[i])
  else
    print nums[i], target-nums[i]
```

std::unordered_set

```
unordered_set<int> s
for(i=0 to end)
  if(s.find(target - nums[i]) == s.end)
    s.insert(nums[i])
  else
    print nums[i], target-nums[i]
```