Hash Tables - Motivation

- Consider the problem of storing several (key, value) pairs in a data structure that would support the following operations efficiently
  - Insert(key, value)
  - Delete(key, value)
  - Find(key)
- Data Structures we have looked at so far
  - Search Trees (BST, AVL, Splay) – all ops O(logN)
  - Btree – all ops O(height)

Can we make Find/Insert/Delete all O(1)?

Hash Tables

A **hash table** is a data structure that implements an array abstract data type, a structure that can map keys to values.

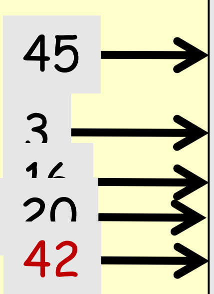
A **hash function** is used to compute an index (hash code) into an array of slots.

During lookup, the key is hashed and the resulting value is used as an index in the table.

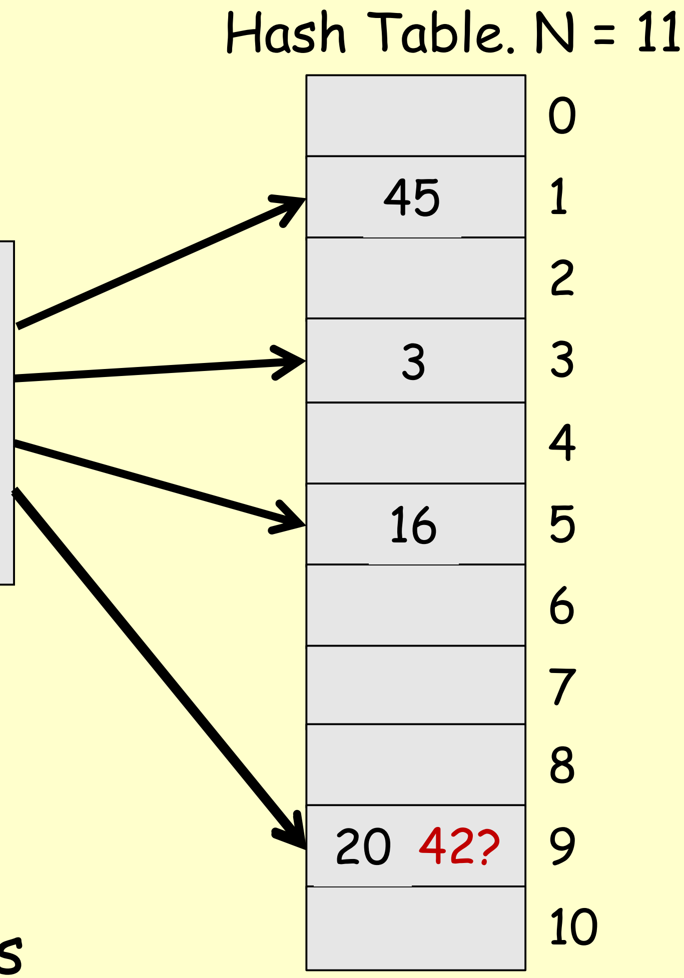
Access of data becomes very fast if index of the desired data is known.

Hash Tables – Main Idea

- Main idea: Use the key (string or number) to index directly into an array – O(1) time to access records



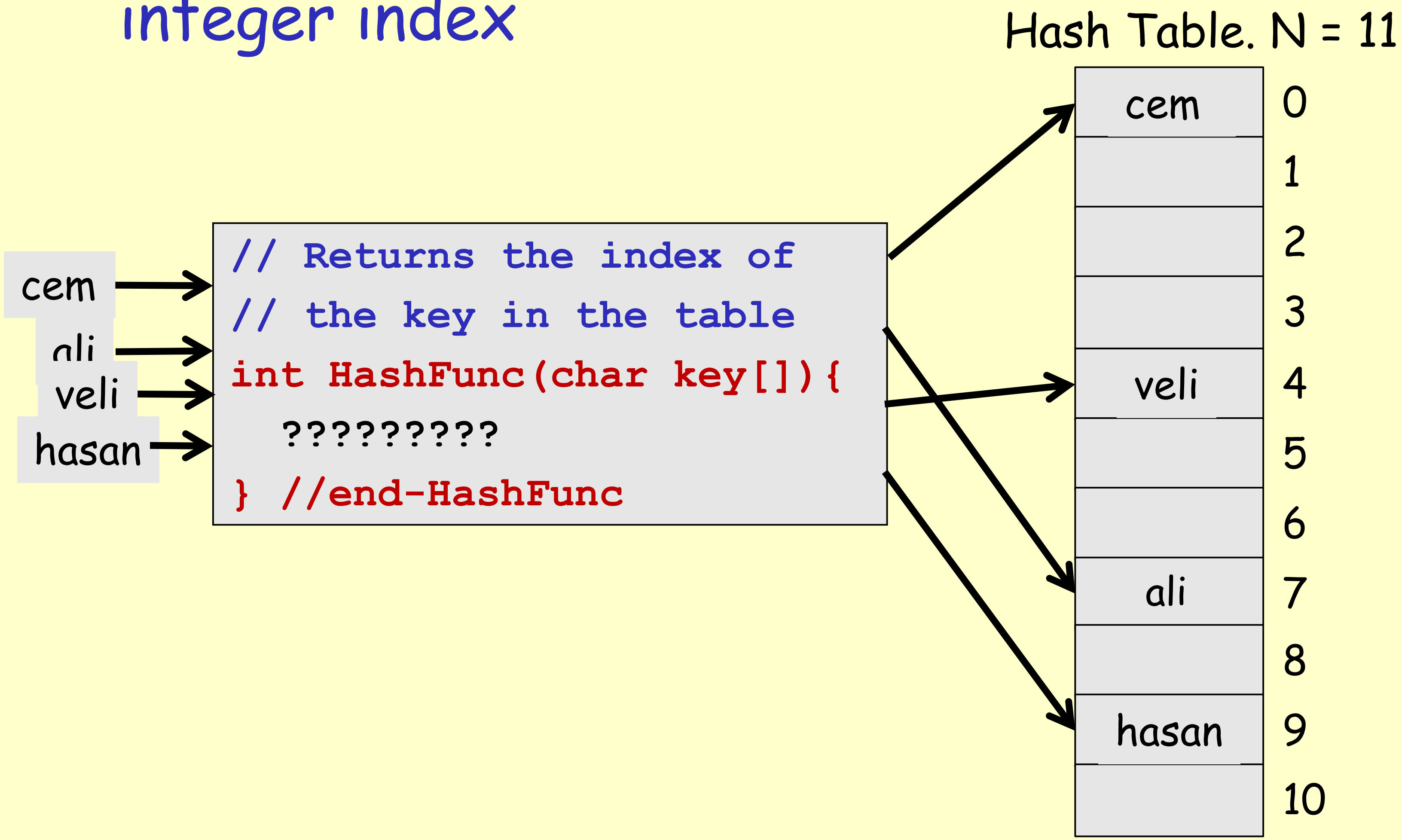

```
// Returns the index of
// the key in the table
int HashFunction(int key){
  return key%N;
} //end-HashFunction
```



- Keys 20 & 42 both map to the same slot – 9
  - Called a collision!
  - Need to handle collisions

Hash Tables - String Keys

- If the keys are strings, then convert them to an integer index



Hash Functions for String Keys

- If keys are strings, can get an integer by adding up ASCII values of characters in *key*

```
// Returns the index of
// the key in the table
int HashFunction(String key){
  int hashCode = 0;
  int len = key.length();

  for (int i=0; i<len; i++){
    hashCode += key.charAt(i);
  } //end-for

  return hashCode % N;
} //end-HashFunction
```

Problems?

1. Will map "abc" and "bac" to the same slot!

2. If all keys are 8 or less characters long, then will map only to positions 0 through 8*127 = 1016
   - Need to evenly distribute keys

Hash Functions for String Keys

- Problems with adding up char values for string keys
  1. If string keys are short, will not hash to all of the hash table
  2. Different character combinations hash to same value
     - "abc", "bca", and "cab" all add up to 6

- Suppose keys can use any of 29 characters plus blank
- A good hash function for strings: treat characters as digits in base 30 (using "a" = 1, "b" = 2, "c" = 3, ……… "z" = 29, " " (space) = 30)
  - "abc" = 1*30^2 + 2*30^1 + 3 = 900+60+3=963
  - "bca" = 2*30^2 + 3*30^1 + 1 = 1800+270+1=2071
  - "cab" = 3*30^2 + 1*30^1 + 2 = 2700+30+2=2732

- Can use 32 instead of 30 and shift left by 5 bits for faster multiplication

Hash Functions for String Keys

- A good hash function for strings: treat characters as digits in base 30
  - Can use 32 instead of 30 and shift left by 5 bits for faster multiplication

```
// Returns the index of the key in the table
int HashFunction(String key){
  int hashCode = 0;
  int len = key.length();

  for (i=0; i<len; i++){
    hashCode = (hashCode << 5) + key.charAt(i)-'a';
  } //end-for

  return hashCode % N;
} //end-HashFunction
```

Hash Table Size

- We need to make sure that Hash Table is big enough for all keys and that it facilitates the Hash Function's job to evenly distribute the keys
  - What if TableSize is 10 and all keys end in 0?
    - All keys would map to the same slot!
  - Need to pick TableSize carefully
    - typically, a prime number is chosen

Properties of Good Hash Functions

- Should be efficiently computable – O(1) time
- Should hash evenly throughout hash table
- Should utilize all slots in the table
- Should minimize collisions

Collisions and their Resolution

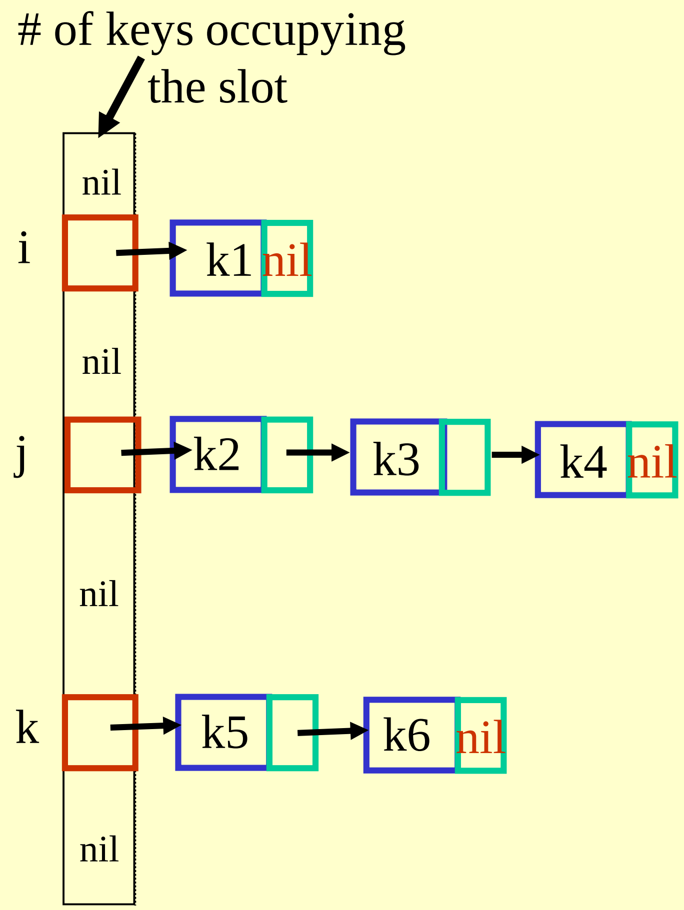
- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1

- Cannot store both data records in the same slot in array!

- Two different methods for collision resolution:
  - **Separate Chaining**: Use data structure (such as a linked list) to store multiple items that hash to the same slot
  - **Open addressing (or probing)**: search for empty slots using a second function and store item in first empty slot that is found

Separate Chaining

- Each hash table cell holds a pointer to a linked list of records with same hash value (i, j, k in figure)
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Can use a linked-list for Find/Insert/Delete in linked list
- Can also use BSTs: O(log N) time instead of O(N). But lists are usually small – not worth the overhead of BSTs



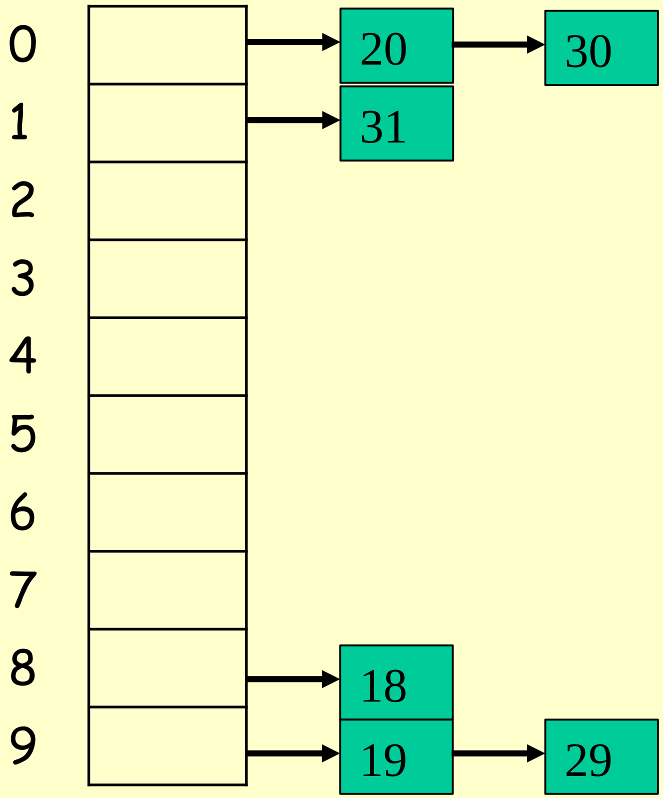
* Hash(k1) = i

* Hash(k2)=Hash(k3)=Hash(k4) = j

* Hash(k5)=Hash(k6)=k

Example

- **In-Class Example**: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 using separate chaining



- **18%10=8**
- **19%10=9**
- **20%10=0**
- **29%10=9**
- **30%10=0**
- **31%10=1**

Load Factor of a Hash Table

- Let N = number of items to be stored
- **Load factor LF = N/*TableSize***
- Suppose *TableSize* = 2 and number of items N = 10
  - LF = 5
- Suppose *TableSize* = 10 and number of items N = 2
  - LF = 0.2
- Average length of chained list = LF
- Average time for accessing an item = O(1) + O(LF)
  - Want LF to be close to 1 (i.e. *TableSize* ~N)
  - But chaining continues to work for LF > 1

Collision Resolution by Open Addressing

- Linked lists can take up a lot of space…
- Open addressing (or probing): When collision occurs, try alternative cells in the array until an empty cell is found
- Given an item X, try cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$
- **$h_i(X)$ = (Hash(X) + F(i)) mod *TableSize***
- Define F(0) = 0
- F is the collision resolution function. **Three possibilities**:
  - Linear: **F(i) = i**
  - Quadratic: **F(i) = i^2**
  - Double Hashing: **F(i) = i*$Hash_2$(X)**

Open Addressing I: Linear Probing

- **Main Idea**: When collision occurs, scan down the array one cell at a time looking for an empty cell
- $h_i(X) = (Hash(X) + i) \bmod TableSize$ ($i = 0, 1, 2, \ldots$)
- Compute hash value and increment until free cell is found

Example

- **In-Class Example**: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 using linear probing:
- hi(X) = (Hash(X) + i) mod *TableSize* (i = 0, 1, 2, …)

0	20
1	29
2	30
3	31
4	28
5	
6	
7	
8	18
9	19

- **(29+1)%10=0 i=1**
- **(29+2)%10=1 i=2**
- **(30+0)%10=0 i=0**
- **(30+1)%10=1 i=1**
- **(30+2)%10=2 i=2**
- **(31+0)%10=1 i=0**
- **(31+1)%10=2 i=1**
- **(31+2)%10=3 i=2**

- **(28+0)%10=8 i=0**
- **(28+1)%10=9 i=1**
- **(28+2)%10=0 i=2**
- **(28+3)%10=1 i=3**
- **(28+4)%10=2 i=4**
- **(28+5)%10=3 i=5**
- **(28+6)%10=4 i=6**

Load Factor Analysis of Linear Probing

- Recall: **Load factor LF = N/*TableSize***
- Fraction of empty cells = 1 – LF
- Fraction cells we expect to probe = 1/(1- LF)
- Can show that expected number of probes for:
  - Successful searches = O(1+1/(1- LF))
  - Insertions and unsuccessful searches = O(1+1/(1- LF)^2)
- Keep LF <= 0.5 to keep number of probes small (between 1 and 5). (E.g. What happens when LF = 0.99)

Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* (LF ~ 1), access time approaches O(N)
- Very prone to cluster formation (as in our example)
  - If key hashes into a cluster, finding free cell involves going through the entire cluster
  - Inserting this key at the end of cluster *causes the cluster to grow*: future Inserts will be even more time consuming!
  - This type of clustering is called *Primary Clustering*
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of distributing keys uniformly

Open Addressing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot
  Increment by i^2 instead of i
- **hi(X) = (Hash(X) + i^2) mod *TableSize*** (i = 0, 1, 2, …)
  - No primary clustering but secondary clustering possible

- Example 1: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10
- Example 2: Insert {1, 2, 5, 10, 17} with *TableSize* = 16

- Theorem: If *TableSize* is prime and LF < 0.5, quadratic probing will always find an empty slot

Example

- **In-Class Example**: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 using quadratic probing:
- $hi(X) = (Hash(X) + i^2) \bmod TableSize$ ($i = 0, 1, 2, \ldots$)

0	20
1	30
2	31
3	29
4	
5	
6	
7	
8	18
9	19

- **(29+0)%10=9 i=0**
- **(29+1)%10=0 i=1**
- **(29+4)%10=3 i=2**
- **(30+0)%10=0 i=0**
- **(30+1)%10=1 i=1**
- **(31+0)%10=1 i=0**
- **(31+1)%10=2 i=1**

Example

- **In-Class Example**: Insert {1, 2, 5, 10, 17} with *TableSize* = 16 using quadratic probing:
- $h_i(X) = (Hash(X) + i^2) \bmod TableSize$ $(i = 0, 1, 2, \ldots)$

0	
1	1
2	2
3	
4	
5	5
…	
10	10
…	
15	

- **(1+0)%16=1 i=0**
- **(2+0)%16=2 i=0**
- **(5+0)%16=5 i=0**
- **(10+0)%16=10 i=0**
- **(17+0)%16=1 i=0**
- **(17+1)%16=2 i=1**
- **(17+4)%16=5 i=2**
- **(17+9)%16=10 i=3**
- **(17+16)%16=1 i=4**
- **(17+25)%16=10 i=5**

Theorem: If *TableSize* is prime and LF < 0.5, quadratic probing will always find an empty slot

Open Addressing III: Double Hashing

- Idea: Spread out the search for an empty slot by using a second hash function
  - No primary or secondary clustering

- **$h_i(X)$ = (Hash(X) + i*$Hash_2$(X)) mod *TableSize*** for i = 0, 1, 2, …
- **E.g. $Hash_2$(X) = R – (X mod R)**
  - R is a prime smaller than *TableSize*

- Try this example: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 and R = 7

- No clustering but slower than quadratic probing due to Hash2

Example

- **In-Class Example**: Insert {18, 19, 20, 29, 31, 28} into empty hash table with *TableSize* = 10 and R=7 using double hasing:
- **hi(X) = (Hash(X) + i*Hash2(X)) mod *TableSize*** for i = 0, 1, 2, …
- **E.g. Hash2(X) = R – (X mod R)**

0	20
1	31
2	28
3	
4	
5	29
6	
7	
8	18
9	19

- **(29+0*hash2)%10=9 i=0**
- **(29+1*hash2(29))%10=5 i=1**
- **(31+0*hash2)%10=1 i=0**
- **(28+0*hash2)%10=8 i=0**
- **(28+1*hash2(28))%10=5 i=1**
- **(28+2*hash2(28))%10=2 i=2**

- Hash2(29)=7-(29%7)
- Hash2(29)=7-1=6

Lazy Deletion with Probing

- Need to use *lazy deletion* if we use probing (why?)
  - Think about how Find(X) would work…
- Mark array slots as "Active/Not Active"
- If table gets too full (LF ~ 1) or if many deletions have occurred:
  - Running time for Find etc. gets too long, and
  - Inserts may fail!
  - What do we do?

Rehashing

- **Rehashing** – Allocate a larger hash table (of size 2**TableSize*) whenever LF exceeds a particular value

- **How does it work?**
  - Cannot just copy data from old table: Bigger table has a new hash function
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table

- Running time = O(N)
  - but happens very infrequently

Applications of Hash Tables

- **In Compilers**: Used to keep track of declared variables in source code – this hash table is known as the "Symbol Table."

- **In storing information associated with strings**
  - Example: Counting word frequencies in a text

- **In on-line spell checkers**
  - Entire dictionary stored in a hash table
  - Each word is text hashed – if not found, word is misspelled.

Hash Tables vs Search Trees

- Hash Tables are good if you would like to perform ONLY Insert/Delete/Find

- Hash Tables are not good if you would also like to get a sorted ordering of the keys
  - Keys are stored arbitrarily in the Hash Table
- Hash Tables use more space than search trees
  - Our rule of thumb: Time versus space tradeoff ☺

- Search Trees support sort/predecessor/successor/min/max operations, which cannot be supported by a Hash Table