Hashing

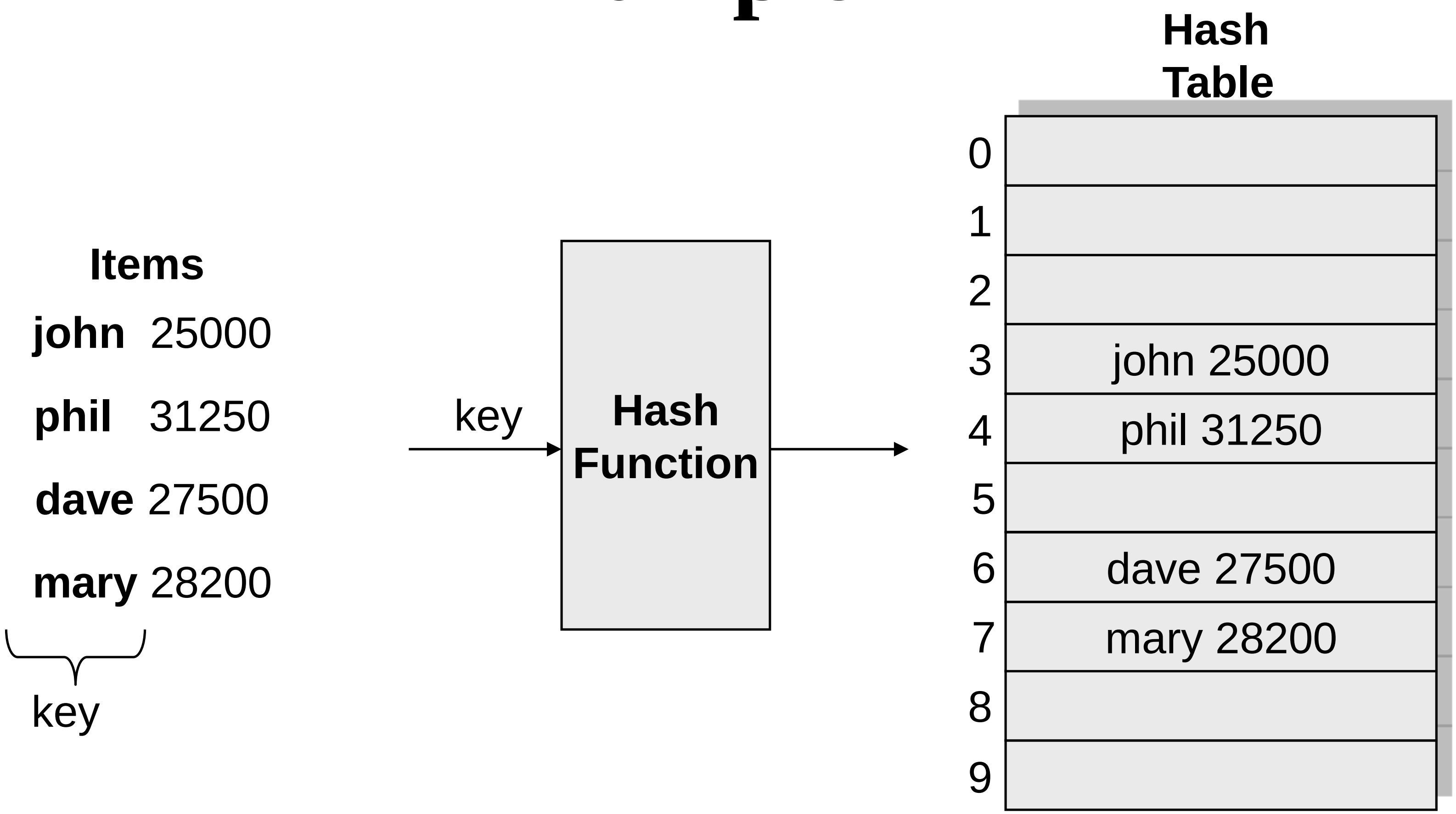
Hash Tables

- 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

Problems:

- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- How to decide table size, hash func, hash map code
- 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

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;
}
```

- Many words have the same sum
- 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 (127*8) where 127 is largest integer value for a char.

Hash Function 2

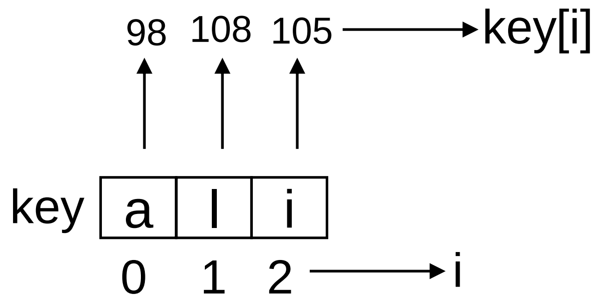
- Examine only the first 3 characters of the key.

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

}
```

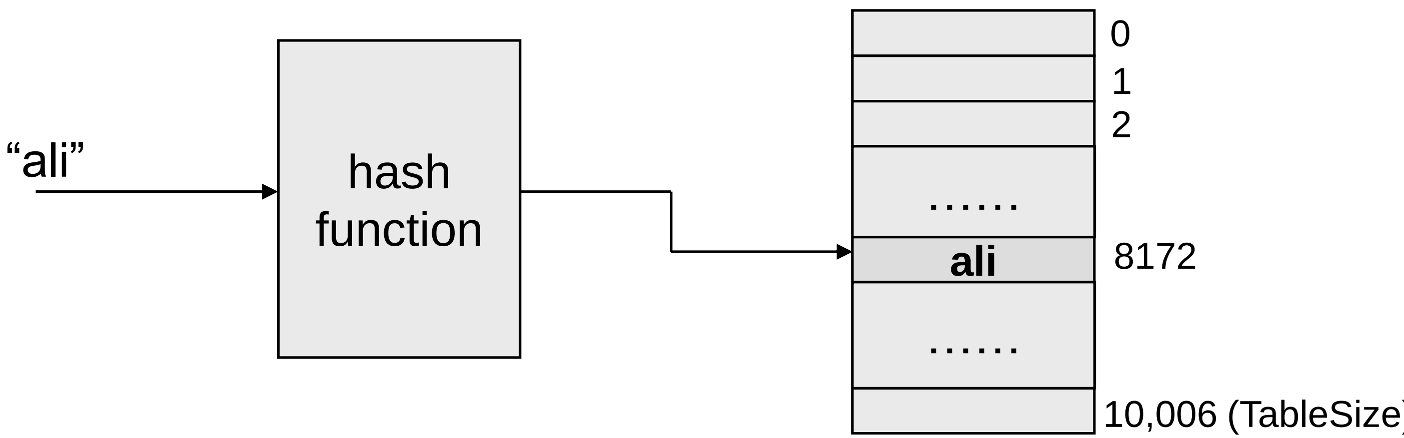
- In theory, **26 * 26 * 26** = **17576** different words can be generated. However, English is not random, only **2851** different combinations are possible.
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

Hash function for strings:



KeySize = 3;

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



Hash Code map

Horners Rule with x=33,37,39,41 experimentally found to give 6 collisions among words in the English dictionary

Compression Code

Examples

$H(k)=k \bmod m$

Pick m to be prime to avoid dependency on last few characters

Based on how much load you want to give each cell (k)
in the chain you can nearest prime at n/k

$H(k)=m(k\,A \bmod 1)$ for $0<A<1$

$H(k)= (ak + b) \bmod m$ where a and m should be co-prime

Universal Hashing: Pick out of a bunch of hashes at random for one round of hash table filling.

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

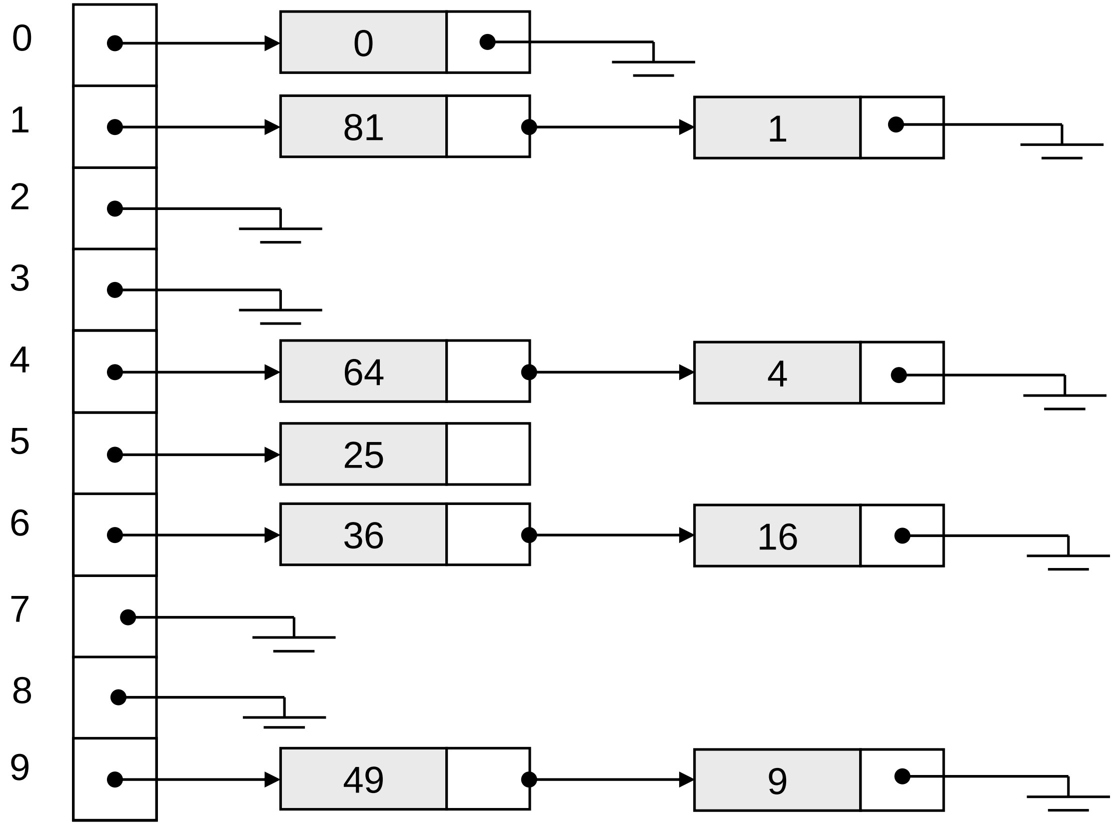
Separate Chaining

- 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.

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 $f(i) = i$.
  - In other words: Linear probing is when the interval between two successive probes is fixed. Usually at 1. Not necessarily
- 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.

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.