Hashing

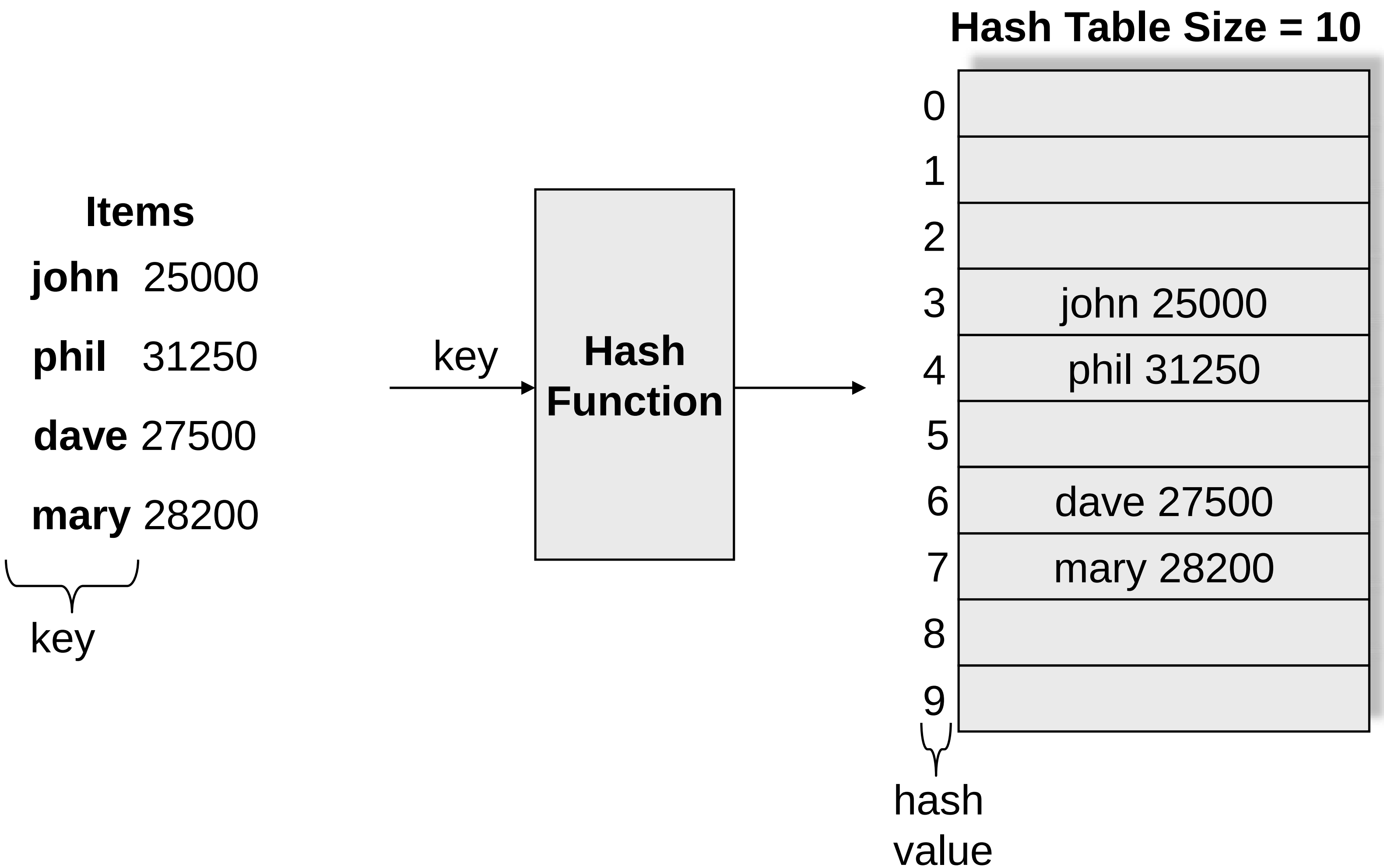
Hash Tables

- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
  - Worst-case times O(n)
- *Hash table* ADT supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called **hashing**.
- 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

- Hash table structure is merely an array of some fixed size.
- A stored item needs to have a data member, called ***key***,
- Key is used to compute the index value for the item.
  - Key could be an *integer*, a *string*, etc
- The size of the array is *TableSize*.
- The items in the hash table are indexed from *0* to *TableSize – 1*.
- Each key is mapped into some index - called *hash value*.
- The mapping of key to a hash value is called a *hash function*.

Example


Hash Table Size = 10
Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
0
1
2
3
4
5
6
7
8
9
john 25000
phil 31250
dave 27500
mary 28200
hash value

Hash Function

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

- Issues:
  - 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 Functions

- If the input keys are integers then simply *Key % 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:
  - First convert it into a numeric value.

Some methods

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking the last 3 digits of the key: H(IDNum) = IDNum % 1000 = hash value
- **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;
}
```

- Limitations;
  – 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

Collision Resolution

- Collision:
  - If an element hashes to the same value as an already inserted element, it is called collision.
- Collision Resolutions:
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

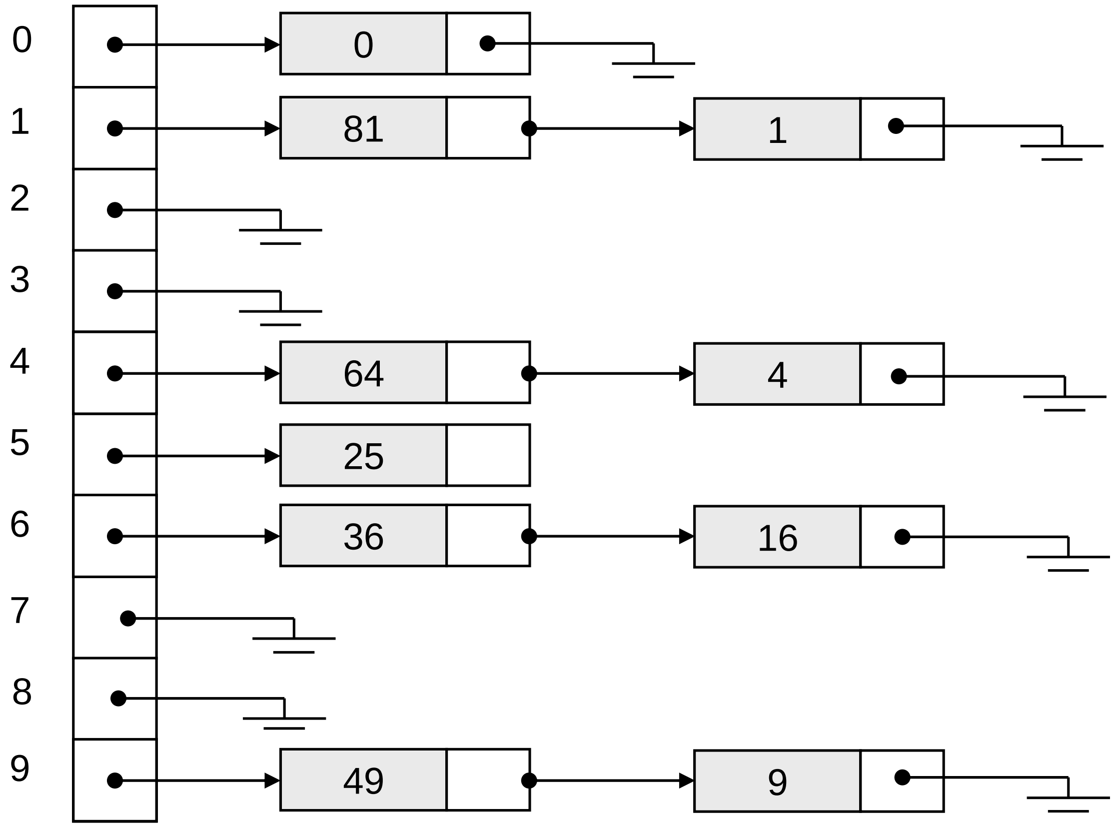
Separate Chaining

- All the elements that hash on to same index are put on a linked list.
  - 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.



Separate Chain – 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

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.