Hash Tables

- Also, called associative arrays or hash maps or maps or lookup table or dictionary.

- Hash tables associate keys with values. Data records are stored in a hash table. The position of a data record in the hash table is determined by its key.

- Has a similar concept to arrays but is not indexed by integers but other forms of data such as string.

- There are many situations when we want to index elements differently than just by integers. Common examples are strings (for dictionaries, phone books, menus, database records).

A table is a sequence of pairs. The first component of the pair is called the key. It serves as the index into the table, generalizing the subscript integer used in arrays. The second component is called the value of its key component. It contains the information being looked up. In the dictionary example, the key is the word being looked up, and the value is that word’s definition (and everything else listed for that word).

- One application of hashing is cryptography

- The keys are mapped to hash values (also called hash codes, digests, hashes, hash keys) in the hash table using a hash function.

![Diagram of key, hash function, and array index]
0, 1, 872,….,999 are hash values/indexes to the hash table entries (key, value) – name and telephone numbers in this example.

- Complexity of insert, find and delete operations is O(1) (without collisions)

**Selecting hash functions**

1. **Modular Arithmetic**: Compute the index by dividing the key with some value and use the remainder as the index.

For Example: \( \text{index} := \text{key MOD table\_size} \)

\( H(k) = k \mod 10. \)

Insert the keys 83, 14, 29, 70, 45, 22
83 will go to index 3
14 will go to index 4
29 will go to index 9, etc

2. Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table. For example: If student id’s are the key 928324312 then select just the last three digits as the index i.e. 312 as the index

- Equivalent objects MUST hash to the same location
- A good hash function is essential for good hash table performance.
  - A good hash function is fast to compute.
  - A good hash function minimize collision as much as possible

A poor choice of hash function is likely to lead to clustering behavior, in which the probability of keys mapping to the same hash bucket (i.e. a collision) is significantly greater than would be expected from a random function.

- Collision

Let us consider the case when we have a single array with four records, each with two fields, one for the key and one to hold data (we call this a single slot bucket). Let the hashing function be a simple modulus operator i.e. array index is computed by finding the remainder of dividing the key by 4.

Array Index := key MOD 4

Then key values 9, 13, 17 will all hash to the same index. When two (or more) keys hash to the same value, a collision is said to occur.

When two keys map to the same location in the hash table We try to avoid it, but number---of---keys exceeds table size

Hashing Objects in Java
All Java objects contain the following method:

```java
public int hashCode() //Returns an integer hash code for this object.
```

We can call `hashCode` on any object to find its preferred index.

**Example**

```java
import java.io.*;
public class Test {
    public static void main(String args[]) {
        String Str = new String("Data Structures and Algorithms");
        System.out.println("Hashcode for Str :" + Str.hashCode());
    }
}
```

How is `hashCode` implemented?
- A String's `hashCode` adds the ASCII values of its letters.

- **Collision Resolution**

  1. **Separate Chaining (also known as open hashing or closed addressing)**

     In the simplest chained hash table technique, each slot in the array references (is a pointer to) a bucket/linked list of inserted records that collide to the same slot.

     Insertion requires finding the correct slot, and appending to either end of the list in that slot; deletion requires searching the list and removal.

     For separate chaining, the worst-case scenario is when all entries are inserted into the same bucket (thus clustering occurs), in which case the hash table is ineffective and the cost is that of searching the bucket data structure.

     If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number $n$ of entries in the table.

     - Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant.
     - An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache ineffective.
2. **Open addressing/closed hashing**

In general, open addressing means resolving collisions by trying a sequence of other positions in the table.

Open addressing hash tables can store the records directly within the array. A hash collision is resolved by **probing** (searching through alternate locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table)

- **linear probing** - in which the interval between probes is fixed — often at 1

**Example**

![Diagram of hash table with linear probing](image)

When 568 is inserted, it hashes to position 8, which is occupied, so the next position is used. When 208 is inserted, positions 8, 9, 0, and 1 must be tried before the empty position 2 is found.

**Exercise: insert 38, 19, 8, 109, 10 using linear probing**

Hash function \( h(\text{key}) = \text{key} \mod \text{tableSize} \)
There are three problems with linear probing:

1. The table can fill up. With chaining, if we underestimate the number of elements in the set, the lists get longer and search is slower. With linear probing, the hash table fails catastrophically: **when there's no more room, we simply can't insert any more elements.**

   We can solve this problem by rehashing when the table gets too full. Rehashing is copying all of the elements into a fresh table. If we make the new table larger, as we did with our ArrayList class, the new table is not full.

2. **We can't simply remove an item to delete it.** Suppose we removed 480 and then searched for 208. We would hit an unoccupied position and incorrectly conclude that 208 is not in the table.

   We get around this problem by replacing a deleted item with a special value deleted. This is neither null nor is it equals() to any target, so searches continue past it. This in turn creates another problem: the table may become full of deleted items, with very few actual data elements. This, too, can be solved by occasionally rehashing.

3. **Clusters of contiguous occupied positions tend to occur and grow.** Once a cluster appears, an element which hashes to a position in the cluster may result in a linear search to the end of the cluster. Worse, an insertion into any position in a cluster expands the cluster.

   - **quadratic probing - in which the interval between probes increases linearly (hence, the indices are described by a quadratic function)**

     In quadratic probing the offset from x is the square of the step number, so the probe goes to x, x+1, x+4, x+9, x+16, and so on.

     - 0th probe: h(key) % TableSize
     - 1st probe: (h(key) + 1) % TableSize
     - 2nd probe: (h(key) + 4) % TableSize
     - 3rd probe: (h(key) + 9) % TableSize
     - ... 
     - ith probe: (h(key) + i^2) % TableSize

     - Example in handout

   - **double hashing - in which the interval between probes is fixed for each record but is computed by another hash function.**

     A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so.
For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs

- **Java Map interface**

In Java, there is an interface Map defined in the java.util package. Both the keys and the information are Objects.

Java's Map interface has the following public methods (and a few more not listed here):

```java
public interface Map {
    public boolean isEmpty()
    public int size()
    public Object put(Object key, Object value)
    public Object get(Object key)
    public Object remove(Object key)
    public void clear()
    .... }
```

There are several classes provided for you that implement this interface.

```
Object
   __ AbstractMap -------- Map
       |   EnumMap
       |   HashMap
       |   LinkedHashMap
       |   TreeMap
       |   SortedMap
       |   NavigableMap
```