Introduction to Hashing & Hashing Techniques

- **Review of Searching Techniques**
- **Introduction to Hashing**
- **Hash Tables**
- **Types of Hashing**
- **Hash Functions**
- **Applications of Hash Tables**
- **Problems for which Hash Tables are not suitable**

Review of Searching Techniques

- Recall the efficiency of searching techniques covered earlier.
- The sequential search algorithm takes time proportional to the data size, i.e, **`O(n)`**.
- Binary search improves on liner search reducing the search time to **`O(log n)`**.
- With a BST, an **`O(log n)`** search efficiency can be obtained; but the worst-case complexity is **`O(n)`**.
- To guarantee the **`O(log n)`** search time, BST height balancing is required (i.e., AVL trees).

Review of searching Techniques (cont'd)

- The efficiency of these search strategies depends on the number of items in the container being searched.

- Search methods with efficiency independent on data size would be better.

- Consider the following Java class that describes a student record:

```
class StudentRecord {
   String name;    // Student name
   double height;  // Student height
   long id;        // Unique id
}
```

- The `id` field in this class can be used as a *search key* for records in the container.

Introduction to Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.

  · A linked list implementation would take **`O(n)`** time.

  · A height balanced tree would give **`O(log n)`** access time.

  · Using an array of size 100,000 would give **`O(1)`** access time but will lead to a lot of space wastage.

- Is there some way that we could get **`O(1)`** access without wasting a lot of space?

- The answer is hashing.

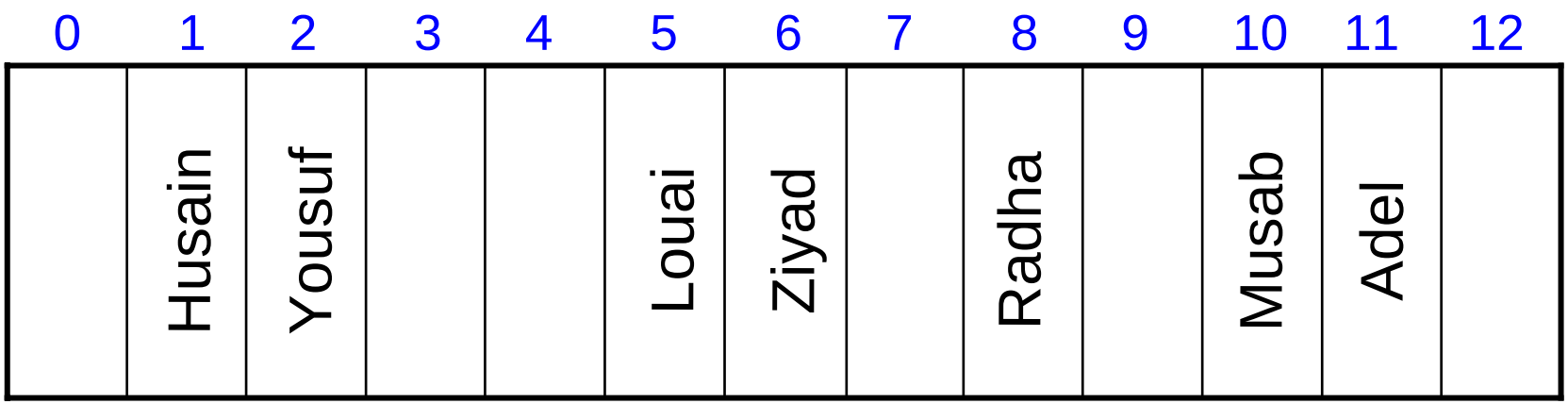
Example 1: Illustrating Hashing

- **Use the function `f(r) = r.id % 13` to load the following records into an array of size 13.**

Al-Otaibi, Ziyad	1.73	985926
Al-Turki, Musab Ahmad Bakeer	1.60	970876
Al-Saegh, Radha Mahdi	1.58	980962
Al-Shahrani, Adel Saad	1.80	986074
Al-Awami, Louai Adnan Muhammad	1.73	970728
Al-Amer, Yousuf Jauwad	1.66	994593
Al-Helal, Husain Ali AbdulMohsen	1.70	996321

Example 1: Introduction to Hashing (cont'd)

Name	ID	h(r) = id % 13
Al-Otaibi, Ziyad	985926	6
Al-Turki, Musab Ahmad Bakeer	970876	10
Al-Saegh, Radha Mahdi	980962	8
Al-Shahrani, Adel Saad	986074	11
Al-Awami, Louai Adnan Muhammad	970728	5
Al-Amer, Yousuf Jauwad	994593	2
Al-Helal, Husain Ali AbdulMohsen	996321	1



Hash Tables

- There are two types of Hash Tables: **Open-addressed Hash Tables** and **Separate-Chained Hash Tables**.

- **An Open-addressed *Hash Table*** is a one-dimensional array indexed by integer values that are computed by an index function called a ***hash function***.

- **A Separate-Chained *Hash Table*** is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a ***hash function***.

- Hash tables are sometimes referred to as ***scatter tables***.

- Typical hash table operations are:

  - ***Initialization.***
  - ***Insertion.***
  - ***Searching***
  - ***Deletion.***

Types of Hashing

- There are two types of hashing :

  1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.

  2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

- The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table.

- Note: The higher the load factor, the slower the retrieval.

- With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.

Hash Functions

- A ***hash function***, `h`, is a function which transforms a key from a set, `K`, into an index in a table of size n:

  `h: K -> {0, 1, ..., n-2, n-1}`

- A key can be a number, a string, a record etc.

- The size of the set of keys, `|K|`, to be relatively very large.

- It is possible for different keys to hash to the same array location.

- This situation is called *collision* and the colliding keys are called *synonyms*.

Hash Functions (cont'd)

- A good hash function should:
  - *Minimize* collisions.
  - Be *easy* and *quick* to compute.
  - Distribute key values *evenly* in the hash table.
  - Use *all the information* provided in the key.

Common Hashing Functions

1. Division Remainder (using the table size as the divisor)

- Computes hash value from key using the % operator.
- Table size that is a power of 2 like 32 and 1024 should be avoided, for it leads to more collisions.
- Also, powers of 10 are not good for table sizes when the keys rely on decimal integers.
- Prime numbers not close to powers of 2 are better table size values.

Common Hashing Functions (cont'd)

2. Truncation or Digit/Character Extraction

- Works based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
- Very fast but digits/characters distribution in keys may not be very even.

Common Hashing Functions (cont'd)

3. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.

- To map the key 25936715 to a range between 0 and 9999, we can:
  - split the number into two as 2593 and 6715 and
  - add these two to obtain 9308 as the hash value.

- Very useful if we have keys that are very large.

- Fast and simple especially with bit patterns.

- A great advantage is ability to transform non-integer keys into integer values.

Common Hashing Functions (cont'd)

4. Radix Conversion

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

$$55354_{10} = 38652_{11}$$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

Common Hashing Functions (cont'd)

5. Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key **3121** into a hash table of size **1000**, we square it **3121^2 = 9740641** and extract **406** as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

Common Hashing Functions (cont'd)

6. Use of a Random-Number Generator

- Given a seed as parameter, the method generates a random number.
- The algorithm must ensure that:
    - It always generates the same random value for a given key.
    - It is unlikely for two keys to yield the same random value.
- The random number produced can be transformed to produce a valid hash value.

Some Applications of Hash Tables

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

- **Browser Cashes**: Hash tables are used to implement browser cashes.

Problems for Which Hash Tables are not Suitable

1. **Problems for which data ordering is required.**
   Because a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, proximity queries, selection, and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementations that keep the keys in order, but they are far from efficient.

2. Problems having **multidimensional data**.

3. **Prefix searching** especially if the keys are long and of variable-lengths.

4. **Problems that have dynamic data**:
   Open-addressed hash tables are based on 1D-arrays, which are difficult to resize once they have been allocated. Unless you want to implement the table as a dynamic array and rehash all of the keys whenever the size changes. This is an incredibly expensive operation. An alternative is use a separate-chained hash tables or dynamic hashing.

5. **Problems in which the data does not have unique keys.**
   Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate-chained hash tables.

Exercises

1. What in your opinion is the single most important motivation for the development of hashing schemes while there already are other techniques that can be used to realize the same functionality provided by hashing methods?

2. How many storage cells will be wasted in an array implementation with O(1) access for records of 10,000 students each with a 7-digit ID number?

3. Must a hash table be implemented using an array? Will an alternative data structure achieve the same efficiency? If yes, why? If no, what condition must the data structure satisfy to ensure the same efficiency as provided by arrays?

4. Which of the techniques for creating hash functions is most general? Why?

5. Why do prime numbers generally make a good selection for hash table sizes?