NOTES ON HASHING

Author: Jayakanth Srinivasan jksrini@mit.edu

Introduction

Any large information source (data base) can be thought of as a table (with multiple fields), containing information.

For example:

A telephone book has fields *name*, *address* and *phone number*. When you want to find somebody's phone number, you search the book based on the *name* field.

A user account on AA-Design, has the fields *user_id*, *password* and *home folder*. You log on using your *user_id* and *password* and it takes you to your *home folder*.

To find an entry (field of information) in the table, you only have to use the contents of **one** of the fields (say name in the case of the telephone book). You **don't** have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.

Ideally, the key should uniquely identify the entry, i.e. if the key is the *name* then no two entries in the telephone book have the same name.

We can treat the Table formulation as an abstract data type. As with all ADT's we can define a set of operations on a table

Operation	Description
Initialize	Initialize internal structure; create an empty table
IsEmpty	True iff the table has no elements
Insert	Given a key and an entry, insert it into the table
Find	Given a key, find the entry associated with the key
Remove	Given a key, find the entry associated with the key and remove it from the table

Table 1. Table ADT Operations

Implementation

Given an ADT, the implementation is subject to the following questions

- What is the frequency of insertion and deletion into the table?
- How many key values will be used?
- What is the pattern for searching for the keys? i.e. will most of the accesses use only one or two key values?
- Is the table small enough to fit into memory?
- How long should the table exist in memory?

We use the word node to represent an entry into the table. For searching, the key is typically stored separately from the table entry (even if the key is present in the entry as well). Can you think of why?

Unsorted Sequential Array

	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on*	

Figure 1. Unsorted Sequential Array Implementation

An array implementation stores the nodes consequtively in any order (not necessarily in ascending or descending order).

Operation	Description
Initialize	O(1)
IsEmpty	O(1) as you will only check if the first element is empty
Insert	O(1) as you will add to the end of the array
Find	O(n) as you have to sequentially search the array, in the worst case through the entire array
Remove	O(n) as you have to sequentially search the array, delete the element and copy all elements one place up

Table 2. Unsorted Sequential Array Table Operations

Unsorted Sequential Array

	key	entry
0	15	<data>
1	17	<data>
2	22	<data>
3	45	<data>
4	67	<data>
⋮	*and so on*	

Figure 2. Sorted Sequential Array Implementation

A sorted array implementation stores the nodes consequtively in either ascending or descending order.

Operation	Description
Initialize	O(1)
IsEmpty	O(1) as you will only check if the first element is empty
Insert	O(1) as you will add to the end of the array
Find	Olog(n) as you can perform a binary search operation Can you think of why?
Remove	O(n) as you have to perform a binary search and shuffle elements one place up

Table 3. Sorted Sequential Array Table Operations

Linked List (Sorted or Unsorted)

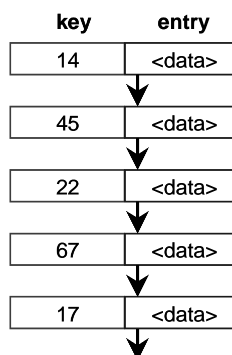

Figure 3. Linked List Implementation

An linked list implementation stores the nodes consequtively (can be sorted or unsorted).

Operation	Description
IsEmpty	O(1) as you will only check if head pointer is null
Insert	O(n) for a sorted list O(1) for an unsorted list, insert at the begining
Find	O(n) as you have to traverse the entire list in the worst case
Remove	O(n) as you have to traverse the list to find the node, removal is carried out using pointer operations

Table 4. Linked List Table Operations

Ordered Binary Tree

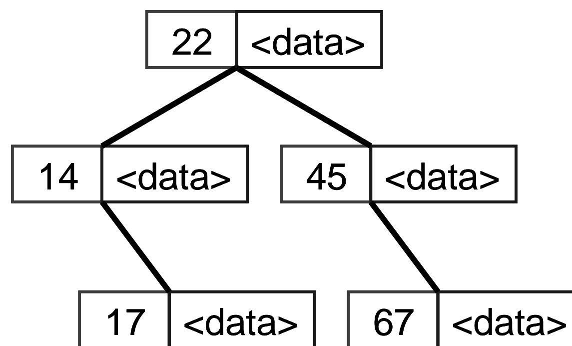

Fig 4. Ordered Binary Tree Implementation

An ordered binary tree is a rooted tree with the property left sub-tree < root < right sub-tree, and the left and right sub-trees are ordered binary trees.

Operation	Description
IsEmpty	O(1) as you will only check if the root is null
Insert	O(logn)) tree is an ordered binary tree
Find	O(log(n)) as the tree is an ordered binary tree
Remove	O(log(n)) as finding takes O(log(n)) and removal takes constant time as it is carried out using pointer operations

Table 5. Ordered Binary Tree Table Operations

Hashing

Having an insertion, find and removal of O(log(N)) is good but as the size of the table becomes larger, even this value becomes significant. We would like to be able to use an algorithm for finding of O(1). This is when hashing comes into play!

Hashing using Arrays

When implementing a hash table using arrays, the nodes are not stored consecutively, instead the location of storage is computed using the key and a ***hash*** function. The computation of the array index can be visualized as shown below:

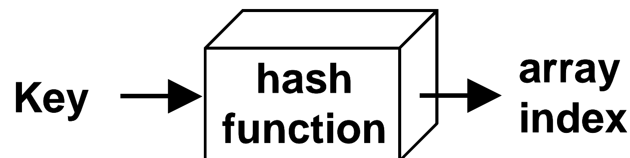

Figure 5. Array Index Computation

The value computed by applying the hash function to the key is often referred to as the hashed key. The entries into the array, are scattered (not necessarily sequential) as can be seen in figure below.

	key	entry
4	<key>	<data>
10	<key>	<data>
123	<key>	<data>

Figure 6. Hashed Array

The cost of the insert, find and delete operations is now only O(1). Can you think of why?

Hash tables are very good if you need to perform a lot of search operations on a relatively stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).

One the other hand, if traversals (covering the entire table), insertions, deletions are a lot more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

Hashing Performance

There are three factors the influence the performance of hashing:

- Hash function
  - should distribute the keys and entries evenly throughout the entire table
  - should minimize collisions

- Collision resolution strategy
  - Open Addressing: store the key/entry in a different position
  - Separate Chaining: chain several keys/entries in the same position

- Table size
  - Too large a table, will cause a wastage of memory
  - Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
  - The size should be appropriate to the hash function used and should typically be a prime number. Why? (We discussed this in class).

Selecting Hash Functions

The hash function converts the key into the table position. It can be carried out using:

- Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques.

  For Example: index := key MOD table_size

- 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. => the table size has to be atleast 999. Why?

- Folding: Partition the key into several pieces and then combine it in some convenient way.

  For Example:
  - For an 8 bit integer, compute the index as follows:
    Index := (Key/10000 + Key MOD 10000) MOD Table_Size.

  - For character strings, compute the index as follows:

```
Index :=0
For I in 1.. length(string)
Index := Index + ascii_value(String(I))
```

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.

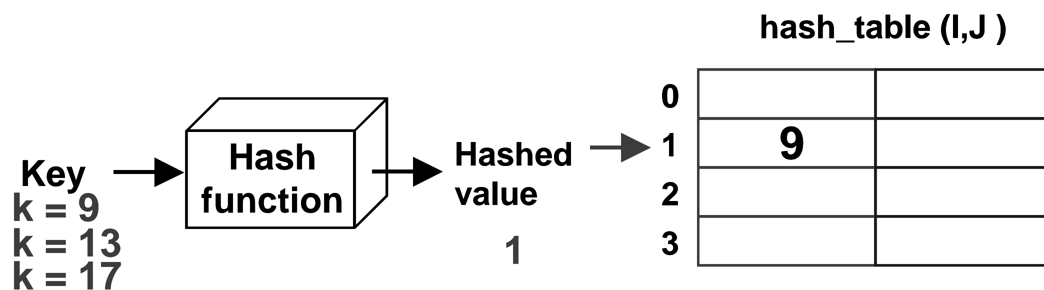

Figure 7. Collision Using a Modulus Hash Function

Collision Resolution

The hash table can be implemented either using

- Buckets: An array is used for implementing the hash table. The array has size m*p where m is the number of hash values and p (≥ 1) is the number of slots (a slot can hold one entry) as shown in figure below. The *bucket* is said to have p slots.

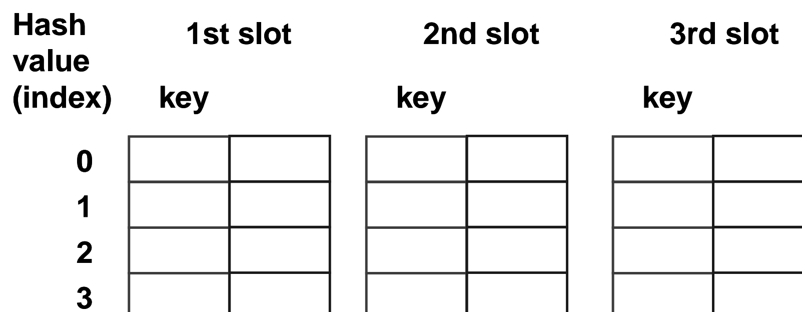

Figure 8. Hash Table with Buckets

- Chaining: An array is used to hold the key and a pointer to a liked list (either singly or doubly linked) or a tree. Here the number of nodes is not restricted (unlike with buckets). Each node in the chain is large enough to hold one entry as shown in figure below.

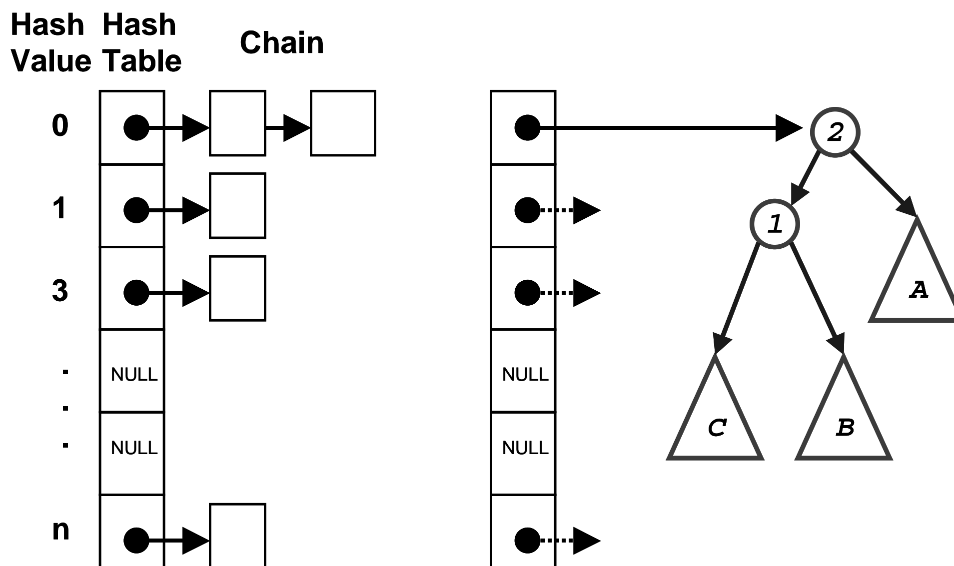

Figure 9. Chaining using Linked Lists / Trees

Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.

There are three schemes commonly used for probing:

- Linear Probing: The linear probing algorithm is detailed below:

```
Index := hash(key)
While Table(Index) Is Full do
        index := (index + 1) MOD Table_Size
if (index = hash(key))
        return table_full
else
        Table(Index) := Entry
```

- Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16, ….

- Double Hash: compute the index as a function of two different hash functions.

Chaining

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

The advantages of using chaining are

- Insertion can be carried out at the head of the list at the index
- The array size is not a limiting factor on the size of the table

The prime disadvantage is the memory overhead incurred if the table size is small.