ITT201 Data Structures

Module 5 : Hash Tables

Anoop S K M
(https://sites.google.com/site/skmanoop)
Department of Information Technology
Govt. Engg. College, Sreekrishnapuram, Palakkad

Acknowledgements

- All the pictures are taken from the Internet using Google search.
- Wikipedia also referred.

Lecture 37

Recap & Goals

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees &Traversals, BST, Expression Tree, Heap Tree
  - Graphs, Graph Terminologies, Graph Traversals, Dijkstra's Algorithm

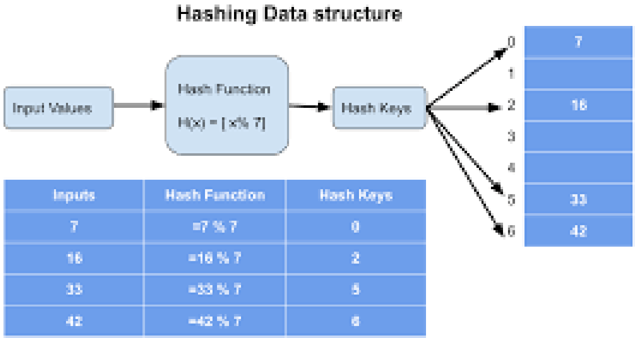
Today We Will See...

- Module 5 : Hash Tables

To Cover

- Hash Tables, Hash Functions, Features of hash function.
- Different Hash Functions
  - Division Method
  - Multiplication Method
  - Mid Square Method
  - Folding Method
- Collision Resolution Techniques: Closed Hashing
  - Linear probing, Drawbacks
  - Remedies Radom Probing
  - Double hashing/Re-hashing
  - Quadratic Probing
- Collision Resolution Technique : Open Hashing (Separate Chaining)

Introduction to Hashing Data Structure



Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Hash Table & Hash Function

- A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found

ID	NAME
352123	BRUCE LEE
12534	LEONARDO DICAPRIO
34654	TOM HANKS
12312	JACK NICHOLSON
54634	AI PANCINO

→ HASH FUNCTION →

INDEX	NAME
1	BRUCE LEE
2	
3	TOM HANKS
4	JACK NICHOLSON
5	
6	LEONARDO DICAPRO
7	AI PANCINO

Features of Hash Functions

- The hash value is fully determined by the data being hashed.
- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible hash values.
- The hash function generates very different hash values for similar strings.

Different Hash Functions

- Division Method
- Multiplication Method
- Mid Square Method
- Folding Method

Division Method

- Divide by the size of slots/ buckets (b) and take the remainder
- $h(k) = k \mod b$
- values ranges from $0, 1, \ldots b - 1$

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Multiplication Method

- $h(k) = \lfloor m(kA \mod 1) \rfloor$
  - m - the slot size
  - k - the key value
  - A - a constant value between 0 and 1, i.e $0 \leq A \leq 1$

Example: $k = 123, m = 100, A = 0.618033$

$$\begin{aligned} h(123) &= \lfloor 100 * (123 * 0.618033) \mod 1) \rfloor \\ &= \lfloor 100 * (76.018059 mod 1) \rfloor \\ &= \lfloor 100 * (0.018059) \rfloor \\ &= \lfloor 1.8055 \rfloor \\ &= 1 \end{aligned}$$

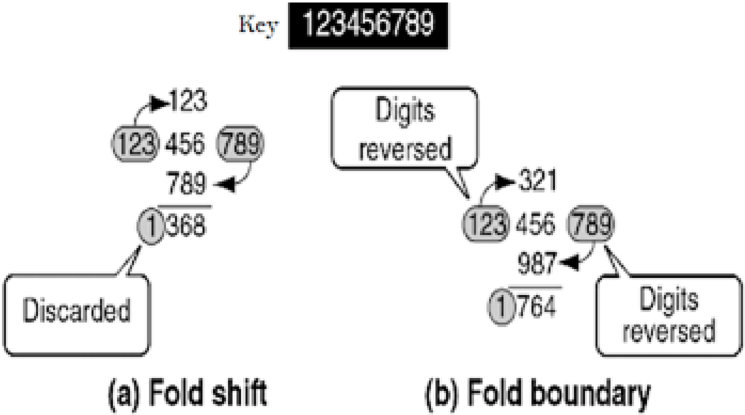
Mid Square Method

- key value is taken and it is squared.
- Then, some digits from the middle are extracted.
- For eg. if the bucket size is 100,

Mid-Square Method

K=	3205	7148	2345
K2=	102**72**025	510**93**904	54**99**025
H(K)=	72	93	99

Folding Method

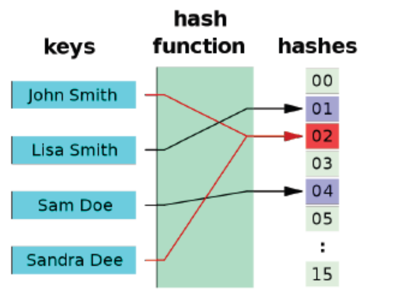


Hash Fold Examples

What is Collision ?

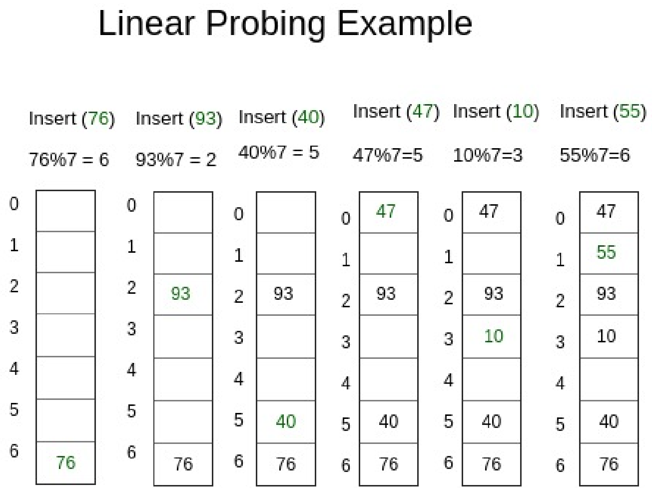
Collision in Hashing

A Collision occurs when more than one value to be hashed by a particular hash function, hash to the same slot/index in the table.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

Closed Hashing-Linear Probing



Double Hashing

Double Hashing

Double hashing is a computer programming technique used in conjunction with open addressing in hash tables to resolve hash collisions, by using a secondary hash of the key as an offset when a collision occurs

Double hashing/Re-hashing

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

Hash2(10) = 7 – (10%7) = 4

(Hash1(10) + 1*Hash2(10))%13= 1

(Hash1(10) + 2*Hash2(10))%13= 5

Collision

Quadratic Probing

Quadratic Probing

Quadratic probing is a scheme where we look for the i^2-th slot in the i-th iteration if the given hash value $h(x)$ collides..

Suppose $h(x) \mod m$ be the hash function and m be the slot/bucket size. Then,

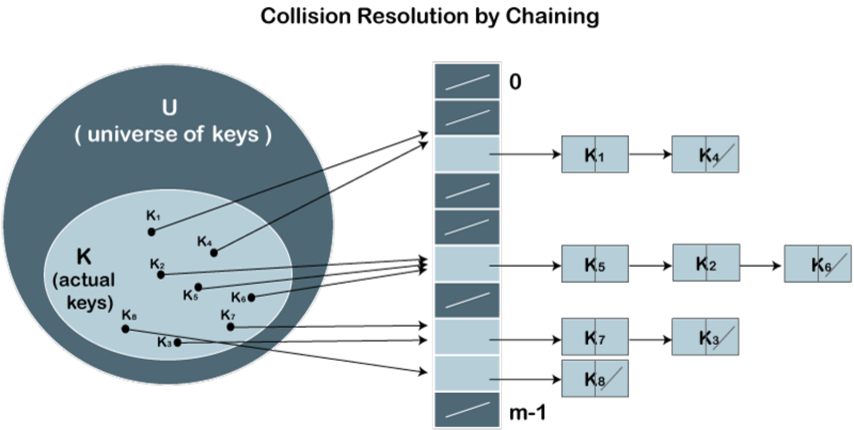
- If the slot $hash(x) \mod m$ cause collision, try $(hash(x) + 1^2) \mod m$
- If $(hash(x) + 1^2) \mod m$ also is collision, try $(hash(x) + 2^2) \mod m$
- If $(hash(x) + 2^2) \mod m$ also is collision, try $(hash(x) + 3^2) \mod m$
- continue the above till collision doesn't occur.

Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **$h(x) = x \bmod 11$**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ $2+1^2=3$
    - 25 mod 11 = 3 ➔ $3+1^2=4$
    - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11,
      $9+3^2$ mod 11 =7

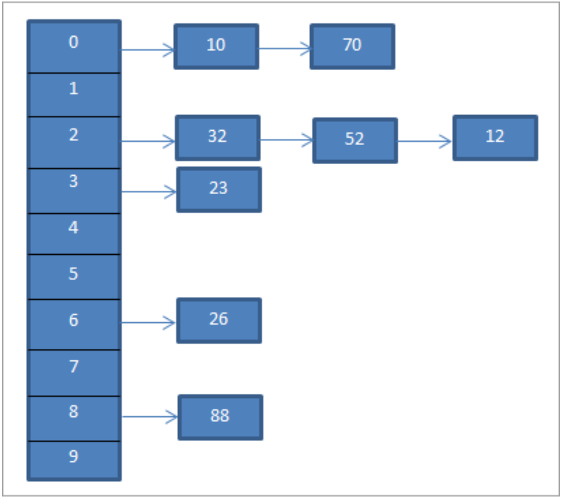
0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

Collision Resolution



Open Hashing

Separate Chaining



Lecture 38

Recap & Goals

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees &Traversals, BST, Expression Tree, Heap Tree
  - Graphs, Graph Terminologies, Graph Traversals, Dijkstra's Algorithm
- Module 5 : Hash Tables

Today We Will See...

- Unique Tree from Prefix/Postfix and Infix

Lecture 39

Recap & Goals

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Searching - Linear and Binary Searches
  - Sorting $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort
  - Sorting $O(n \log n)$: Merge Sort, Quick Sort
- Module 2 : Linked Lists
  - Singly, Doubly, Circular Linked Lists
  - Dynamic Memory Management
- Module 3 : Stacks and Queues
  - Stacks and its applications
  - Queues and Types of queues
- Module 4 : Trees and Graphs
  - Trees, Binary Trees &Traversals, BST, Expression Tree, Heap Tree
  - Graphs, Graph Terminologies, Graph Traversals, Dijkstra's Algorithm
- Module 5 : Hash Tables

Today We Will See...

- Growth of Functions

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
- Bubble Sort : $O(n^2)$, Merge Sort : $O(n \log n)$.

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
- Bubble Sort : $O(n^2)$, Merge Sort : $O(n \log n)$.
- Merge Sort is asymptotically more efficient that Bubble Sort

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
- Bubble Sort : $O(n^2)$, Merge Sort : $O(n \log n)$.
- Merge Sort is asymptotically more efficient that Bubble Sort
- Bubble Sort, Insertion Sort, Selection Sort are all asymptotically same!

Big O(O)

If the running time of an algorithm for an input n is given as

$$T(n) = c_0 + c_1 n + c_2 n^2 + \ldots + c_i n^i$$

then we say the running time of the algorithm is $O(n^i)$

Running Time Complexity : Example 1

```
void printFirstElementOfArray(int arr[n])
{
     printf("First element of array = %d",arr[0]);
}
```

Running Time Complexity : Example 1

```
void printFirstElementOfArray(int arr[n])
{
    printf("First element of array = %d",arr[0]);
}
```

Time Complexity is $O(n)$? No!

This is a constant time algorithm, So Time Complexity is $O(1)$

Running Time Complexity : Example 2

```
void printAllElementOfArray(int arr[], int n)
{
     int size=n;
     for (int i = 0; i < size; i++)
     {
         printf("%d\n", arr[i]);
     }
}
```

Running Time Complexity : Example 2

```
void printAllElementOfArray(int arr[], int n)
{
     int size=n;
     for (int i = 0; i < size; i++)
     {
         printf("%d\n", arr[i]);
     }
}
```

Time Complexity is $O(n)$, as the number of times printf executes is dependent on the size which is equal to n

Running Time Complexity : Example 3

```
void printAllPossibleOrderedPairs(int arr[], int n)
{
    int size = n;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Running Time Complexity : Example 3

```
void printAllPossibleOrderedPairs(int arr[], int n)
{
    int size = n;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Time Complexity is ? $O(n^2)$, as the number of times printf executes is dependent on the size $\times$size which is equal to n^2

Running Time Complexity : Example 4

```c
void printAllItemsTwice(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

Running Time Complexity : Example 4

```
void printAllItemsTwice(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

Time Complexity is ? $O(2n)$? It is OK, But we generally avoid the constant term associated and will write it as $O(n)$

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
int size = n;
    printf("First element of array = %d\n",arr[0]);
for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
int size = n;
    printf("First element of array = %d\n",arr[0]);
for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Time Complexity is ? $O(1 + \frac{n}{2} + 100)$? Like the previous example will write it as $O(n)$, by avoiding the constant terms.

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
    int size = n;
    printf("First element of array = %d\n",arr[0]);
    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
    int size = n;
    printf("First element of array = %d\n",arr[0]);
    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Time Complexity is ? $O(1 + \frac{n}{2} + 100)$? Like the previous example will write it as $O(n)$, by avoiding the constant terms.

Running Time Complexity : Example 6

$$O(n^3 + 50n^2 + 10000) is \ O(n3)$$
$$O((n + 30) * (n + 5)) is \ O(n^2)$$

Running Time Complexity : Example 7

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}  \pause
```

- Running Time dependent upon the element that is searched

Running Time Complexity : Example 7

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}  \pause
```

- Running Time dependent upon the element that is searched
- Best Case, Worst Case and Average Case

Running Time Complexity : Example 7

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}  \pause
```

- Running Time dependent upon the element that is searched
- Best Case, Worst Case and Average Case
- Best Case $O(1)$

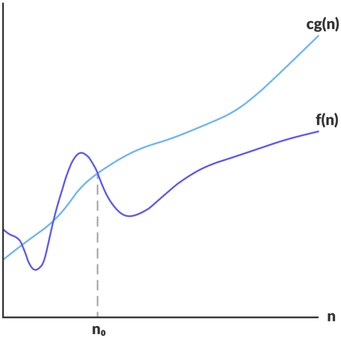
Running Time Complexity : Example 7

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}  \pause
```

- Running Time dependent upon the element that is searched
- Best Case, Worst Case and Average Case
- Best Case $O(1)$
- Average and Worst Case are both ? $O(n)$ Why?

Big O : Formal Definition

$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$



f(n) = O(g(n))

Big O Example

$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$

Let $f(n) = n^3 + 50n^2 + 100, g(n) = n^3$. Prove that $f(n) = O(n^3)$
Find c, n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$

Big O Example

$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$

Let $f(n) = n^3 + 50n^2 + 100, g(n) = n^3$. Prove that $f(n) = O(n^3)$
Find c, n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$
What about $c = 151, n_0 = 1$

Big O Example

$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$

Let $f(n) = n^3 + 50n^2 + 100, g(n) = n^3.$ Prove that $f(n) = O(n^3)$
Find c, n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$
What about $c = 151, n_0 = 1$
$n^3 + 50n^2 + 100 \leq 151n^3$, Hence $O(n^3 + 50n^2 + 100) is$ $O(n^3)$

Big O Example

$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \le f(n) \le c * g(n)$ for all $n \ge n_0\}$

Let $f(n) = n^3 + 50n^2 + 100, g(n) = n^3$. Prove that $f(n) = O(n^3)$
Find c, n_0 such that $f(n) \le c * g(n)$ for all $n \ge n_0$
What about $c = 151, n_0 = 1$
$n^3 + 50n^2 + 100 \le 151n^3$, Hence $O(n^3 + 50n^2 + 100) is$ $O(n^3)$