

Data Structures & Algorithms

- DSA - Home
- DSA - Overview
- DSA - Environment Setup

Algorithm

- DSA - Algorithms Basics
- DSA - Asymptotic Analysis
- DSA - Greedy Algorithms
- DSA - Divide and Conquer
- DSA - Dynamic Programming

Data Structures

- DSA - Data Structure Basics
- DSA - Array Data Structure

Linked Lists

- DSA - Linked List Basics
- DSA - Doubly Linked List
- DSA - Circular Linked List

Stack & Queue

- DSA - Stack
- DSA - Expression Parsing
- DSA - Queue

Searching Techniques

- DSA - Linear Search
- DSA - Binary Search
- DSA - Interpolation Search
- DSA - Hash Table

Sorting Techniques

- DSA - Sorting Algorithms
- DSA - Bubble Sort
- DSA - Insertion Sort
- DSA - Selection Sort
- DSA - Merge Sort
- DSA - Shell Sort
- DSA - Quick Sort

Graph Data Structure

- DSA - Graph Data Structure
- DSA - Depth First Traversal
- DSA - Breadth First Traversal

Tree Data Structure	
DSA - Tree Data Structure	
DSA - Tree Traversal	
DSA - Binary Search Tree	
DSA - AVL Tree	
DSA - Spanning Tree	
DSA - Heap	
Recursion	
DSA - Recursion Basics	
DSA - Tower of Hanoi	
DSA - Fibonacci Series	
DSA Useful Resources	
DSA - Questions and Answers	
DSA - Useful Resources	
DSA - Discussion	

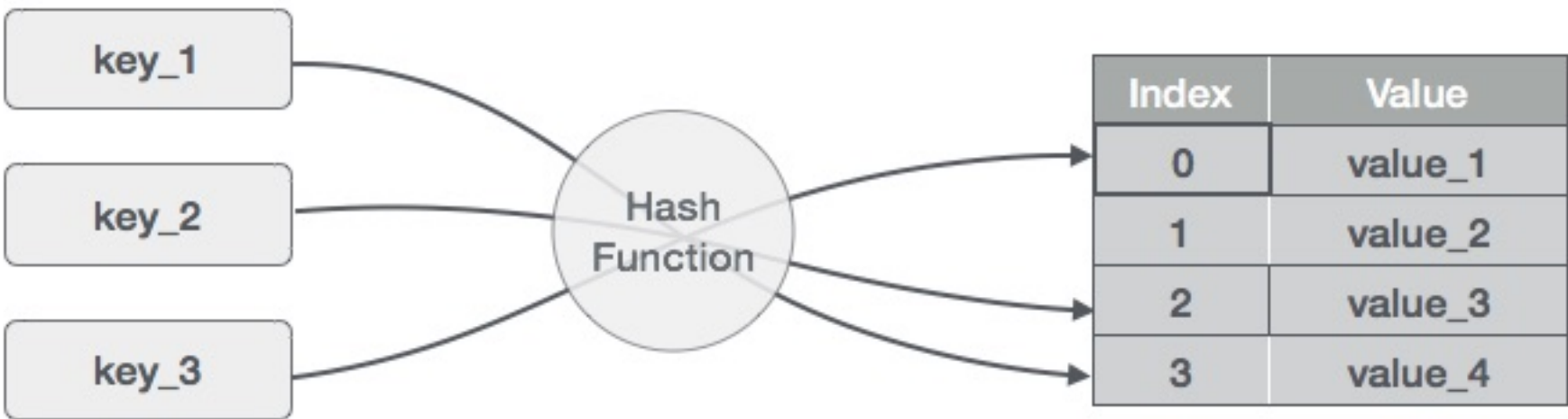
Data Structure - Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)

- (13,78)
- (37,98)

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** – search an element in a hashtable.
- **Insert** – insert an element in a hashtable.
- **delete** – delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
    int data;
    int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
    return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] != NULL){

        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
    item->data = data;
    item->key = key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty or deleted cell
    while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){
    int key = item->key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] !=NULL){

        if(hashArray[hashIndex]->key == key){
            struct DataItem* temp = hashArray[hashIndex];

            //assign a dummy item at deleted position
            hashArray[hashIndex] = dummyItem;
            return temp;
        }

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}
```

To see hash implementation in C programming language, please click here [↗](#) .



[Write for us](#) | [FAQ's](#) | [Helping](#) | [Contact](#)

© Copyright 2016. All Rights Reserved.

Enter email for newsletter

go