**HashMap**

High Level Design

October 2018

Author

Keshava Murthy V

RecoverPoint - DellEMC

Contents

# Introduction………………………………………………………………….3

## Problem Statement……………………………………………………….3

## General Description………………………………………………………3

## Definitions and Acronym………………………………………………..3

## Limitations………………………………………………………………....3

# HashMap Essentials……………………………………………………………3

## Iterator………………………………………………………………………….4

**2.1.2 Iterator Design…………………………………………………….………..4**

## Hash Function………………………………………………………………...5

# Design Details…………………………………………………………………..5

# Class Diagram………………………………………………………………….6

## Data Structures………………………………………………………………6

# HashMap API List……………………………………………………………..7

# System Flow…………………………………………………………………..7

## Unit Test………………………………………………………………………8

# Statistics and Debuggability……………………………………………....8

## Time Measure………………………………………………………..…….8

# Scale and Performance…………………………………………………...8

# Future Implementation…………………………………………………….8

**Introduction**

* 1. **Problem Statement**

Design and implement a hash map which can be used with any data type (user defined data types like class/struct etc. or primitive type like int/float/string/char etc.) in C++ programming language.

**1.2 General Description**

HashMap is a way of storing the data (Key-Value Pair) efficiently onto the memory in such a way that the data can be accessed a lot faster compared to any other storage structure in memory. This is done using the unique value that each data gets out of hash function which is called as hash. This hash upon modifications acts as an index where the data is stored. The key acts as an input to the hash function for generating the hash.

**1.3 Definitions and Acronyms**

1. Binary Search Tree (BST):

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

**1.4 Limitations**

* This library can be used with only integer keys.
* Collision: It is a situation in which two keys will have the same hash value.

1. **HashMap Essentials**

The below are the essentials of the HashMap..

**2.1 Iterator:**

* It provides an abstraction for accessing many different types of Containers (Collection of Various Objects implemented as a Class Template).
* They're a flexible way to access the data in containers that don't have obvious means of accessing all of the data.
* Iterators are more generous.
* Iterators provide the convenience to programmers by taking care of growth in the size of the container. [This is done with the help of the end()].
* Code Reusability is another feature of the Iterators.

**2.1.2 Iterator Design:**

Here's the basic declaration for an iterator is to do in-order traversals.

begin(): It works by finding the first valid Node present in the hashmap starting from the first index in the Array of Pointers.

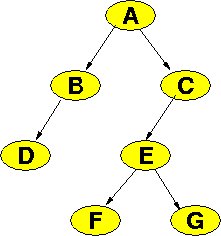
end(): It uses the right most leaf node of the valid Node which is traversed from the last index of the Array of Pointers.

Operator \*(): The dereference operator is overloaded to return the key of the various entries while iterating.

Operator !=(): This operator is overloaded to return a boolean value telling whether the begin() has reached the end().

Operator ++(): Increment operator iterates through the subsequent pointers starting from the begin() all the way to the end().

**Here’s the principle behind overloading ++:**

****

* If the current node has a non-null right child,
  + Take a step down to the right
  + Then run down to the left as far as possible
* If the current node has a null right child,
  + move up the tree until we have moved over a left child link.

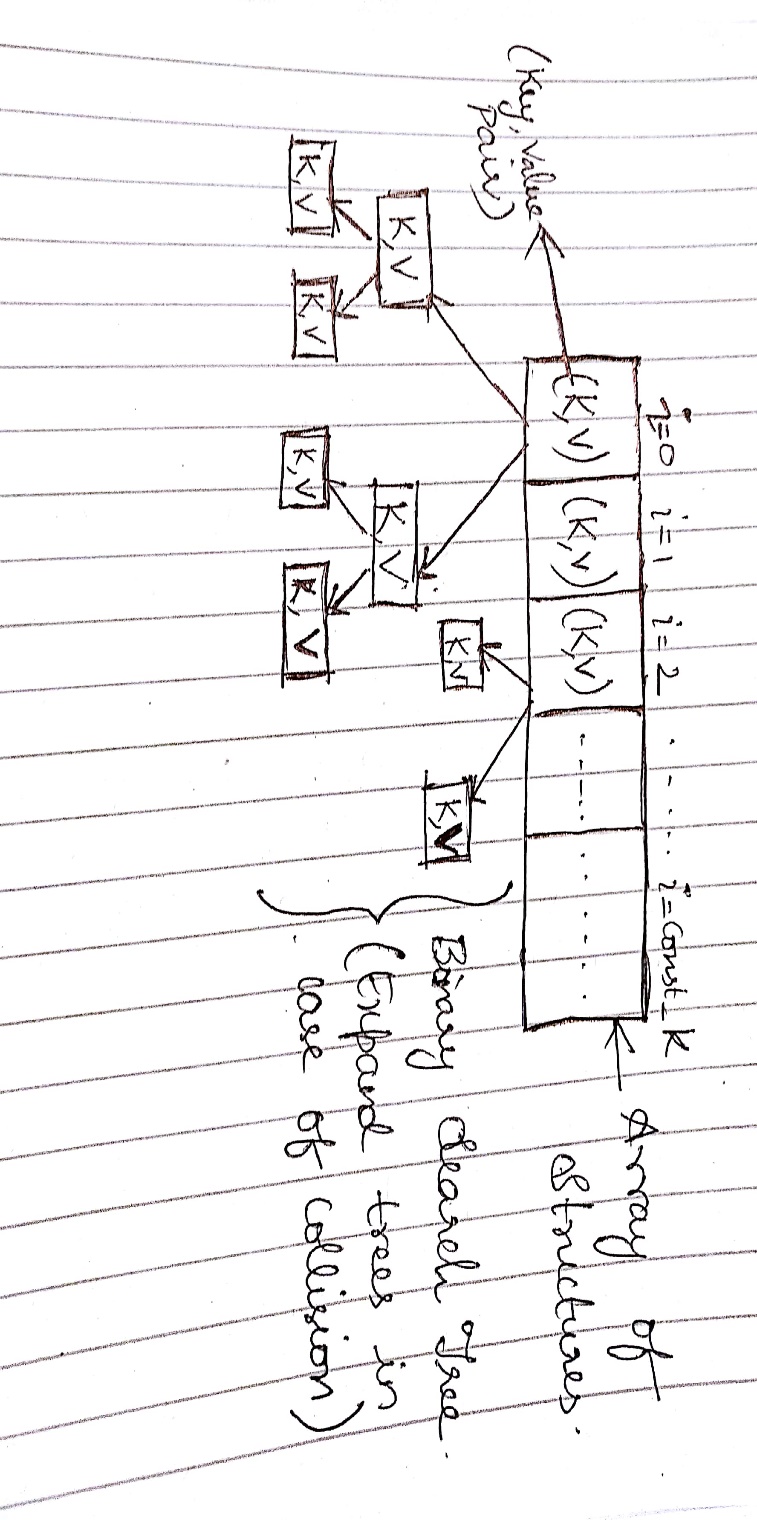
Since our Design has Multiple Root Concept of BST, after traversing one tree we check for the next valid pointer in the array to explore further into the Hashmap until the breaking condition is met.

**2.2 Hash Function**

**Jenkins Hash**

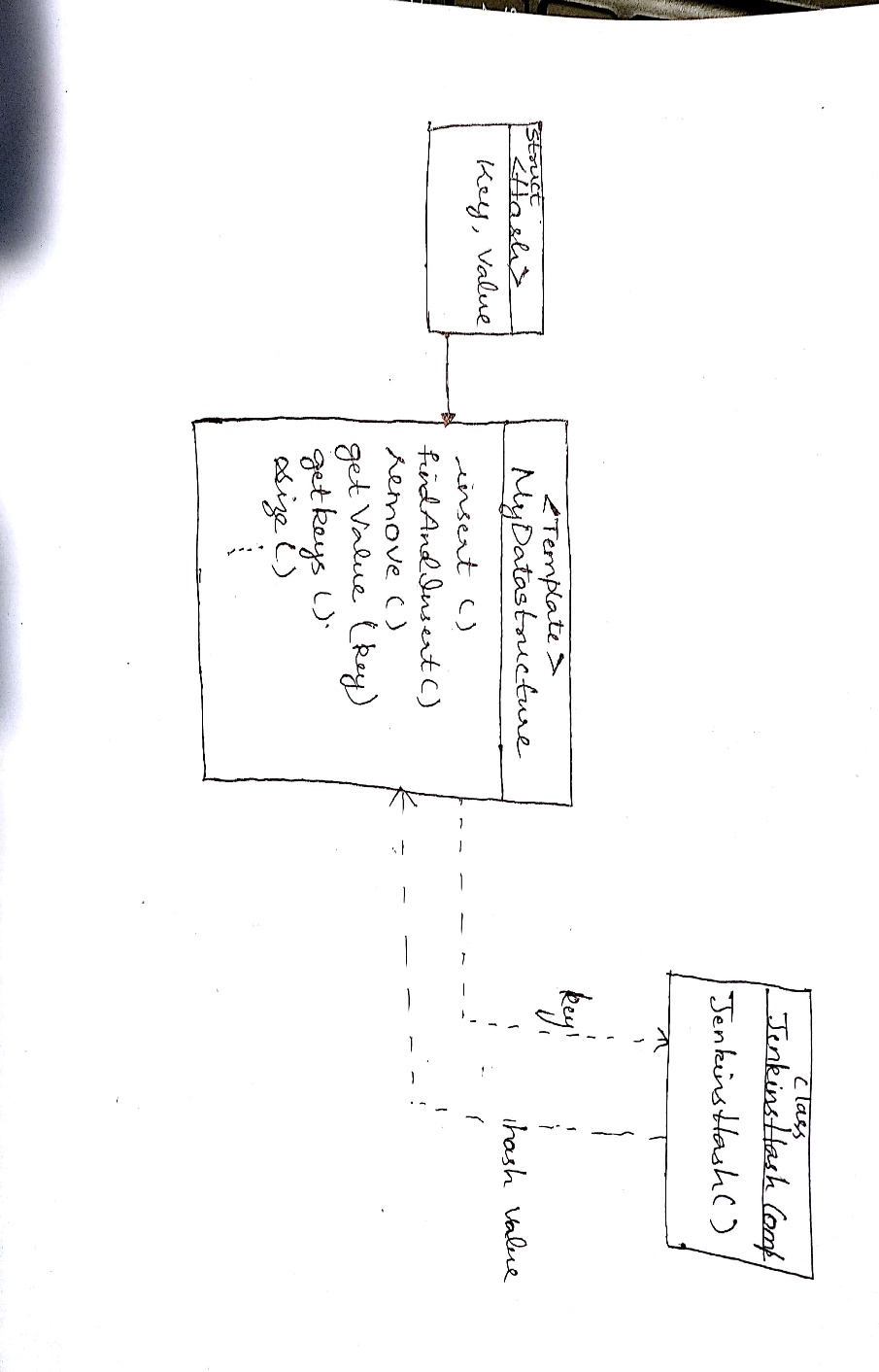
* This is used to compute the hash value for the key-value pair. It provides minimal collision.
* And through various mechanisms we deduce the hash value into index where the key-value pair can be stored.
* There are many Hash Functions and this hash is chosen out of the discussion.

1. **Design Details:**



Basically, we create a template class of data structure which is suitable to fit in any type of key-Value pair (primitive type/non primitive).

1. **Class Diagram**

****

* 1. **Data Structure**

1. **Array of pointers:** This is the primary place of storing the Key-value pair. The index of this array is determined by the hash(Jenkins Hash) which is discussed above.

This will save memory space because of pointers acquiring only 4 bytes (32 bit OS)/8 bytes(64 bit OS) in function operation.

For e.g. If there is 1MB of a structure, then the whole array can be called/passed using pointers utilizing the above-mentioned space.

**Alternatives and reasons for not opting:**

* Lists (Singly/Doubly Linked Lists) : Direct Indexing is not possible and have to traverse through the list right from the starting to search for a pair.
* Stacks and Queues: LIFO and FIFO behavior respectively is not preferred.

1. **Binary Search Tree(BST) :** In case of the index collision that occurs during hashing, we expand that element of the Array(Root Element) into a BST.

**Alternatives and reasons for not opting:**

* Lists: It has poor access and search time complexity of O(n) compared to BST (O(logn)).
* AVL TREE/ RED BLACK TREE: It has the same performance as that of BST (O(logn)) in average case.

# 5 Hashmap API list:

* bool insert(const Key&, Value&) : insert function will add key/value without checking whether key is already present or not. (Always insert key/value untill there is some exception like memory not availabe etc)
* bool findAndInsert(const Key&, Value&) : findAndInsert function will check whether key already exists or not. If key is duplicate then it should return false else true
* bool remove(const Key& key): remove function should remove the entry from the hash map
* Value& operator[](const Key &key) : is overloading [] operator
* Value& get(const Key& key) : to get value based on input key
* void getKeys(std::vector<Key>& keyVector): Returns a vector of all of the keys in the table. we can iterate over the keys to get the values
* uint32\_t size() : returns the total number of entries present in hash map
* uint32\_t getTotalNumberOfCollision(): will return total number of collisions happened in the hash map
* uint32\_t getNumberOfCollisionPerSlot(uint32\_t slotNumber): will return number of collisions happened for the input slot number
* void printCollisionStatistics(): will print all the collision for all the slots sequentially in format like (number of collision at index %u are %u\n)

**6 System Flow**

* Using the insert API to insert a key-value pair without checking if the key is already present in the table.
* Using findAndInsert() validate the key, if duplicate found send a warning message.
* Use the above two API’s successively to create a Hash Table.
* For any kind of searching or removal operations use the appropriate APIs to provide the same.

**6.1 Unit Test**

* We create test Functions through main to simulate the functionality.

# Statistics and Debuggability

* The API getTotalNumberOfCollision() will let us know the total no. of collisions that have occurred in the Hash Table.
* printCollisionStatistics() will help in analyzing the collisions occurred in each index of the Table.

For. E.g. **Table Index No.of Collision**

1. **5**
2. **0**
3. **3**
4. **2**

* Logs: They are used in each module of the program to identify where the error lies or what is the source of the error (whether it is from the neighboring modules ). This is done through simple Print Statements.

## Time Measure

* The overall running time for the creation and operation of the hash table is measured for the comparison of us with other HashMap libraries.

# Scale and Performance

**TBD**

# Future Implementation

* We could support every possible Key datatype