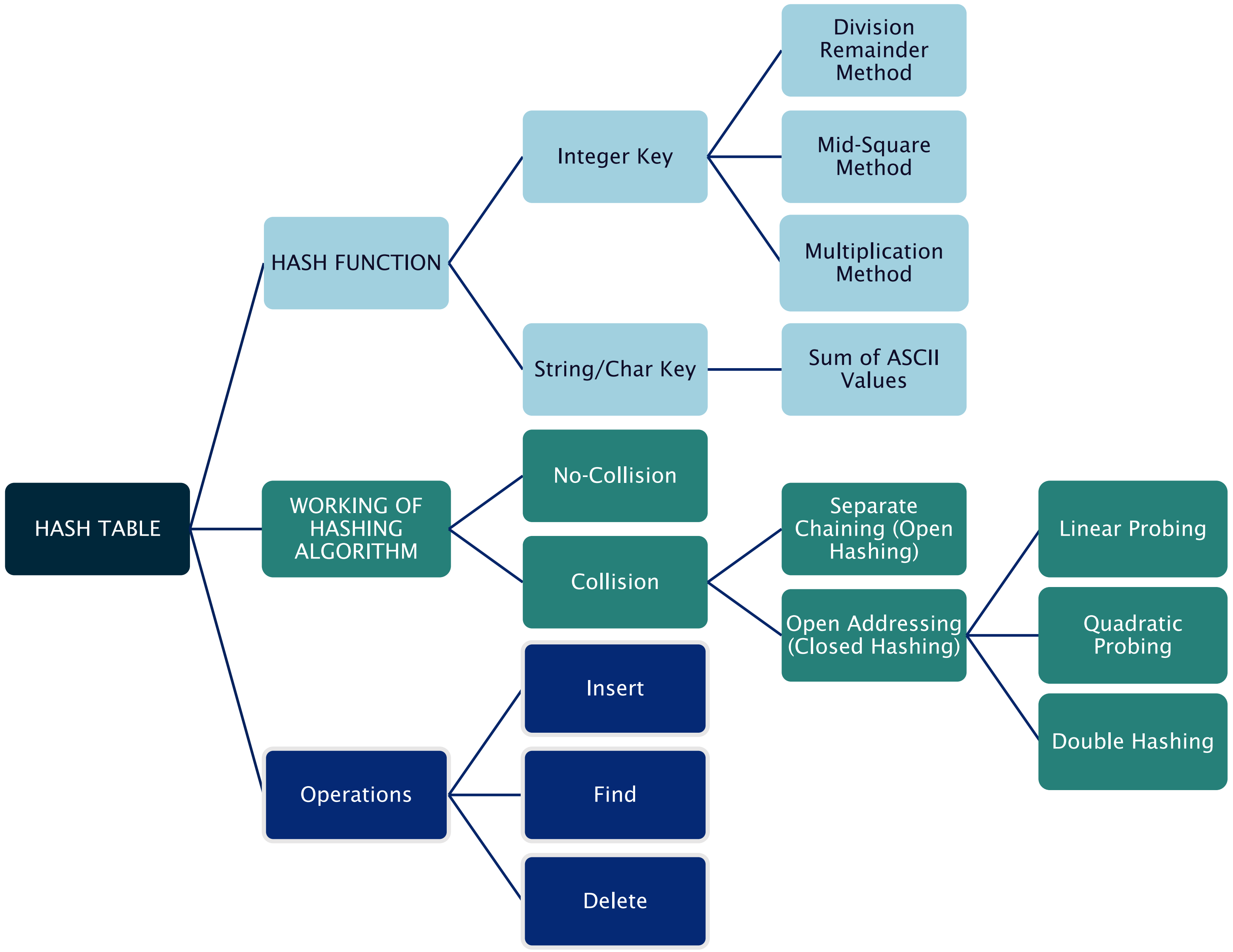
HASH TABLES

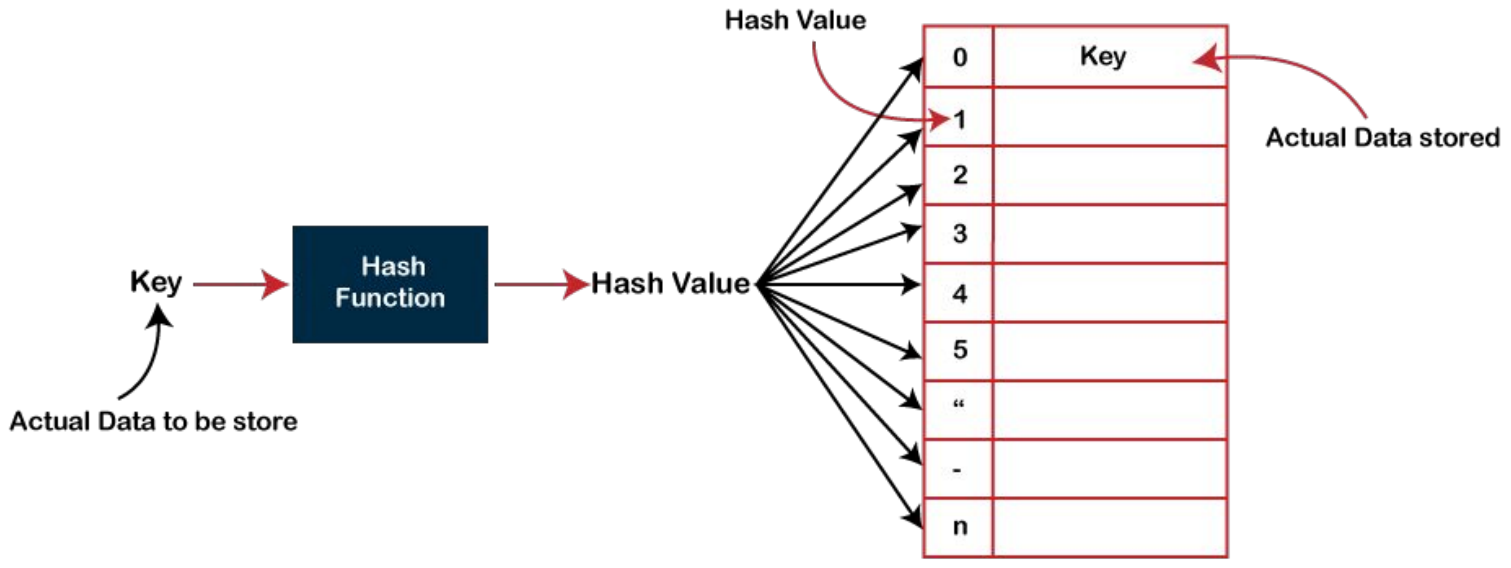
AN IMPLEMENTATION OF HASHING ALGORITHM

Contents

In this concept we will deeply explain this data structure, the algorithm behind it, different collision techniques, and some use case examples including different scenarios thus explaining hashing technique for each of them.



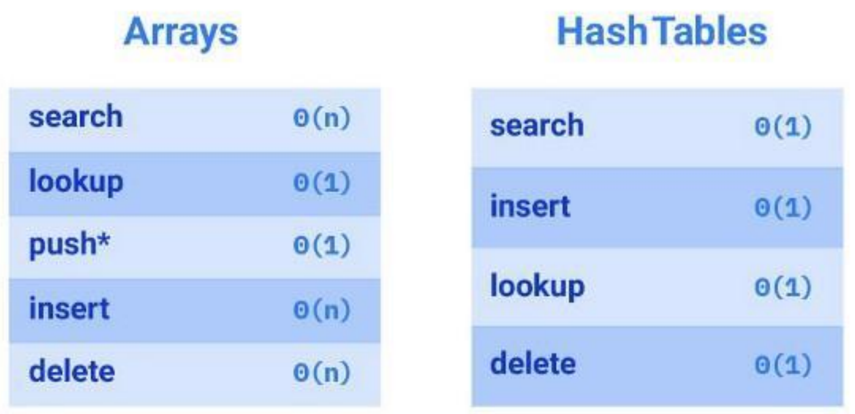
What is Hash Table?



- Hash Table is a data structure which stores data in an associative manner.
- In a hash table, data is stored in an array format, where each data value has its own unique index value.
- The Hash table data structure stores elements in key-value pairs where
  - **Key**- unique integer that is used for indexing the values
  - **Value** - data that are associated with keys.
- A hash table uses a hash function to compute an index, also called a hash value, into an array of buckets or slots, from which the desired value can be found.

Why
Hash Table?

- Hashing gives a more secure and adjustable method of retrieving data compared to any other data structure. It is quicker than searching for lists and arrays as most of these operate in O(n) run time complexity.
- Hashing allows O(1) queries/inserts/deletes to the table.
- The idea of a hash table is to provide a direct access to its items. So that is why the it calculates the "hash code" of the key and uses it to store the item, instead of the key itself.



Arrays

search	O(n)
lookup	O(1)
push*	O(1)
insert	O(n)
delete	O(n)

HashTables

search	O(1)
insert	O(1)
lookup	O(1)
delete	O(1)

Figure 1. Time Complexity Difference

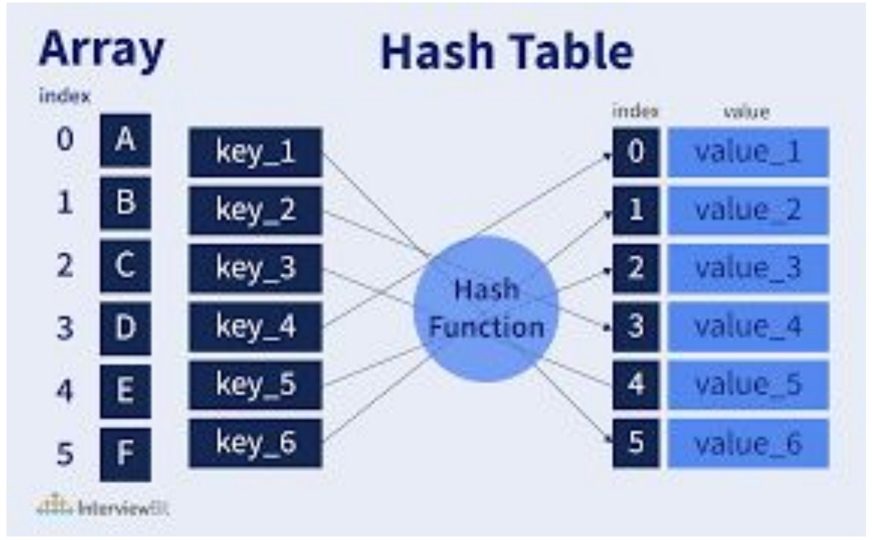
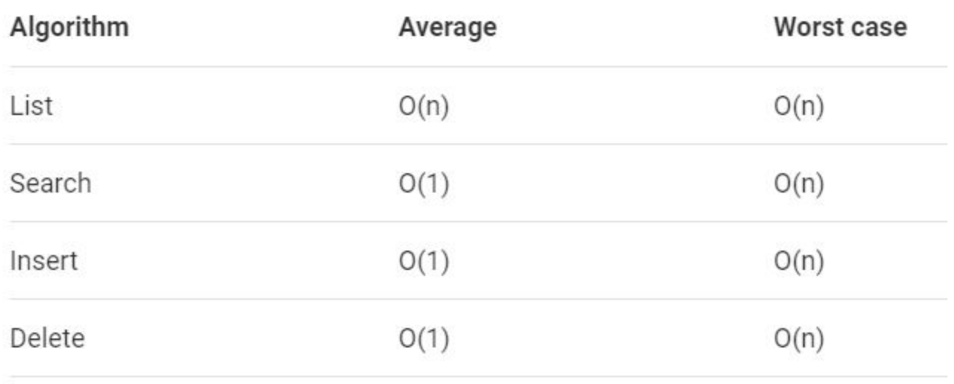


Figure 2. Array vs Hash Table

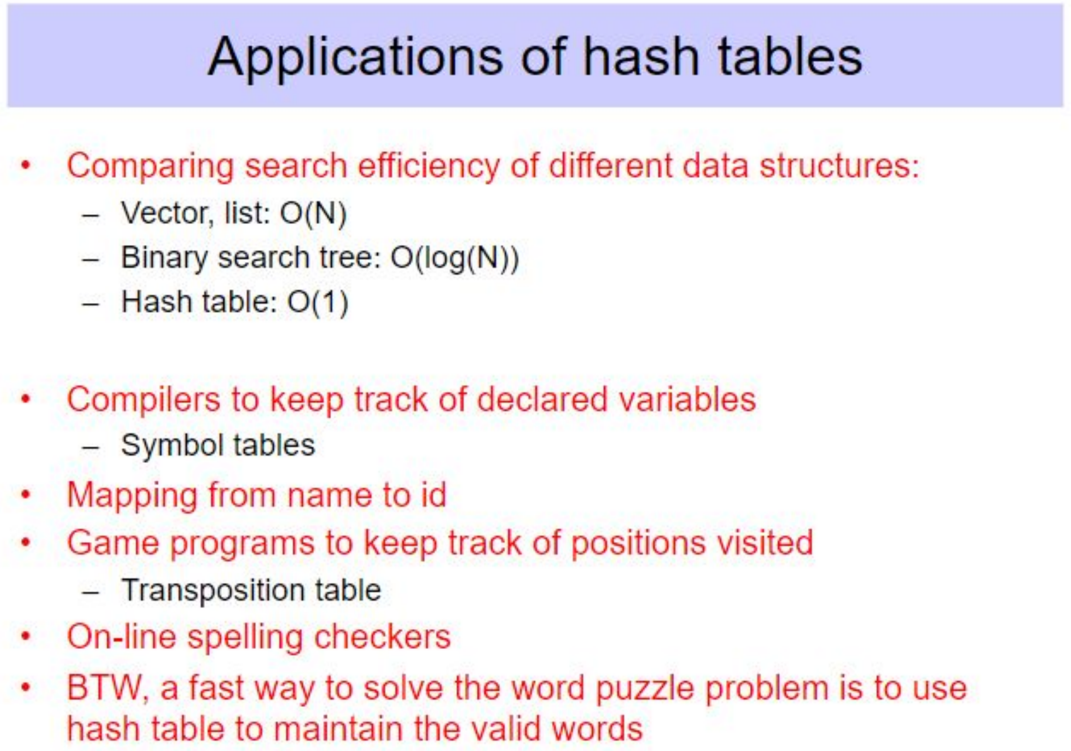
Uses of Hashing and Hash Tables:

- They are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches and sets.
- Compilers use Hash Tables to keep track of declared variables.
- Used for checking circularity of array.
- In most of the programming languages, there are built-in data types or data structures in the standard library that are based on hash tables e.g., dictionary in Python, or HashMap in Java



Algorithm	Average	Worst case
List	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Hash tables can perform nearly all methods (except list) very fast in O(1) time.

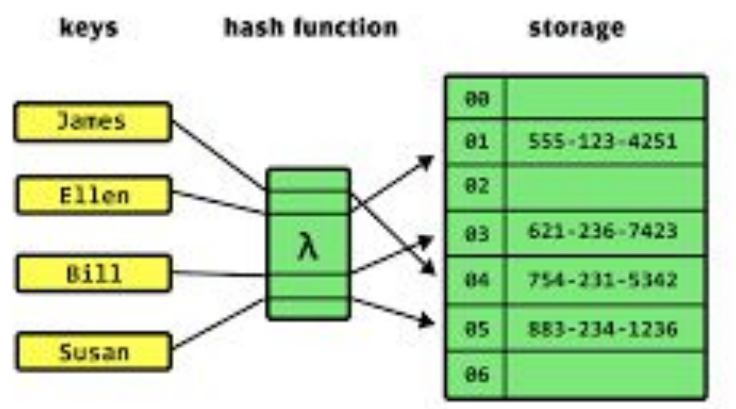


Applications of hash tables

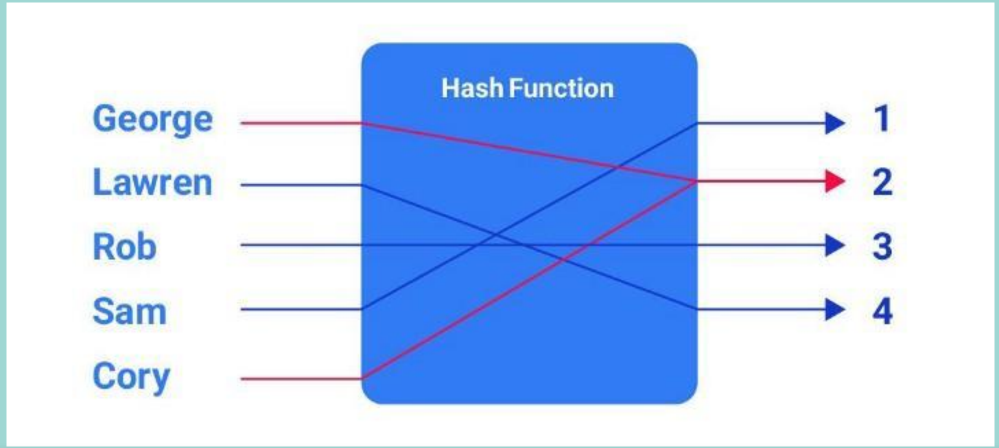
- Comparing search efficiency of different data structures:
  - Vector, list: O(N)
  - Binary search tree: O(log(N))
  - Hash table: O(1)
- Compilers to keep track of declared variables
  - Symbol tables
- Mapping from name to id
- Game programs to keep track of positions visited
  - Transposition table
- On-line spelling checkers
- BTW, a fast way to solve the word puzzle problem is to use hash table to maintain the valid words

Hash Function/ Hashing Algorithm:

- Hashing algorithms are functions that generate a fixed result (the hash value) from a given input.
- Hash function is used to map data of arbitrary size to fixed-size values.
- The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.
- The values are used to index a fixed-size table called a hash table.



What is Collision?
A hash collision occurs when a hash algorithm produces the same hash value for two different input values



Mathematically, $h(k1) = h(k2)$ when $k1 \neq k2$

Hash Function/ Hashing Algorithm:

- A Hash function can be categorized as:
  - **Good Hash Function**: (Minimum or No Collisions)

    Some of the properties of Good Hash Function are:
    - Very fast to compute (nearly constant)
    - One way; can not be reversed
    - Output does not reveal information on input
    - It should minimize duplication of output values

      (Hard to find Collisions)
  - **Bad Hash Function:**

    It has high possibility of duplication of output values. (Producing same Indices resulting in Collisions)

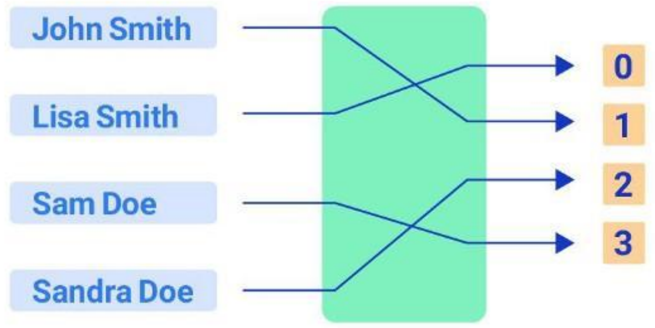


Figure 1. Example of a Good Hash Function

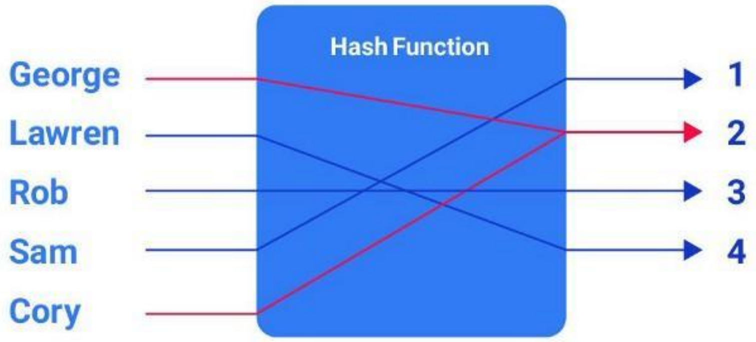
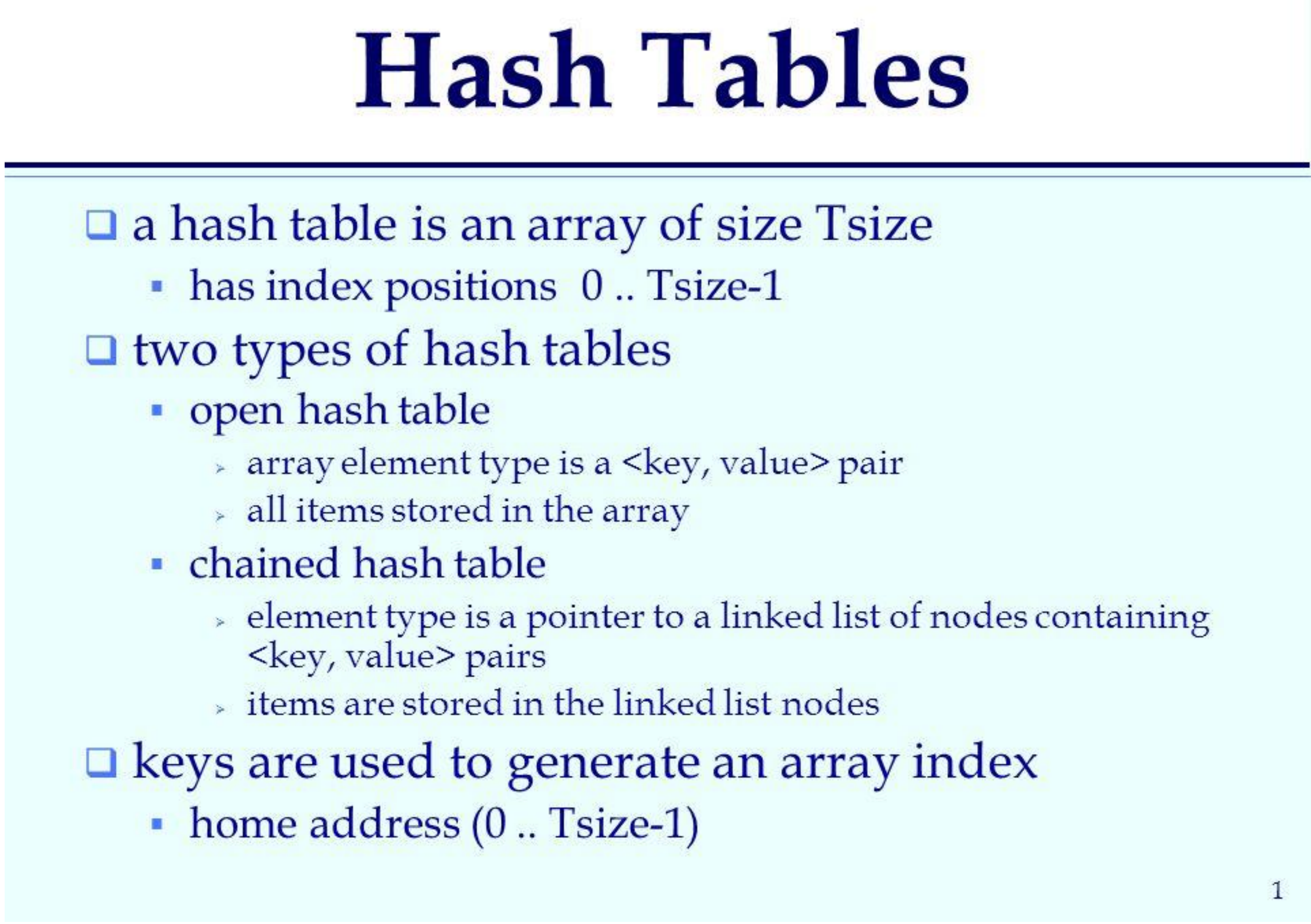


Figure 2. Example of a bad Hash Function

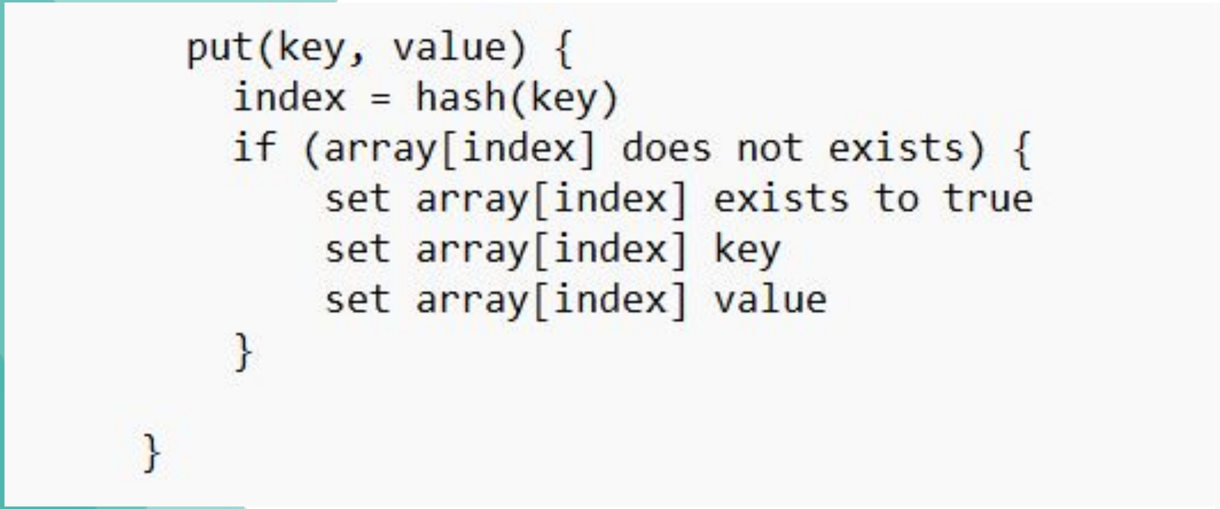
Hash Table & its Operations:



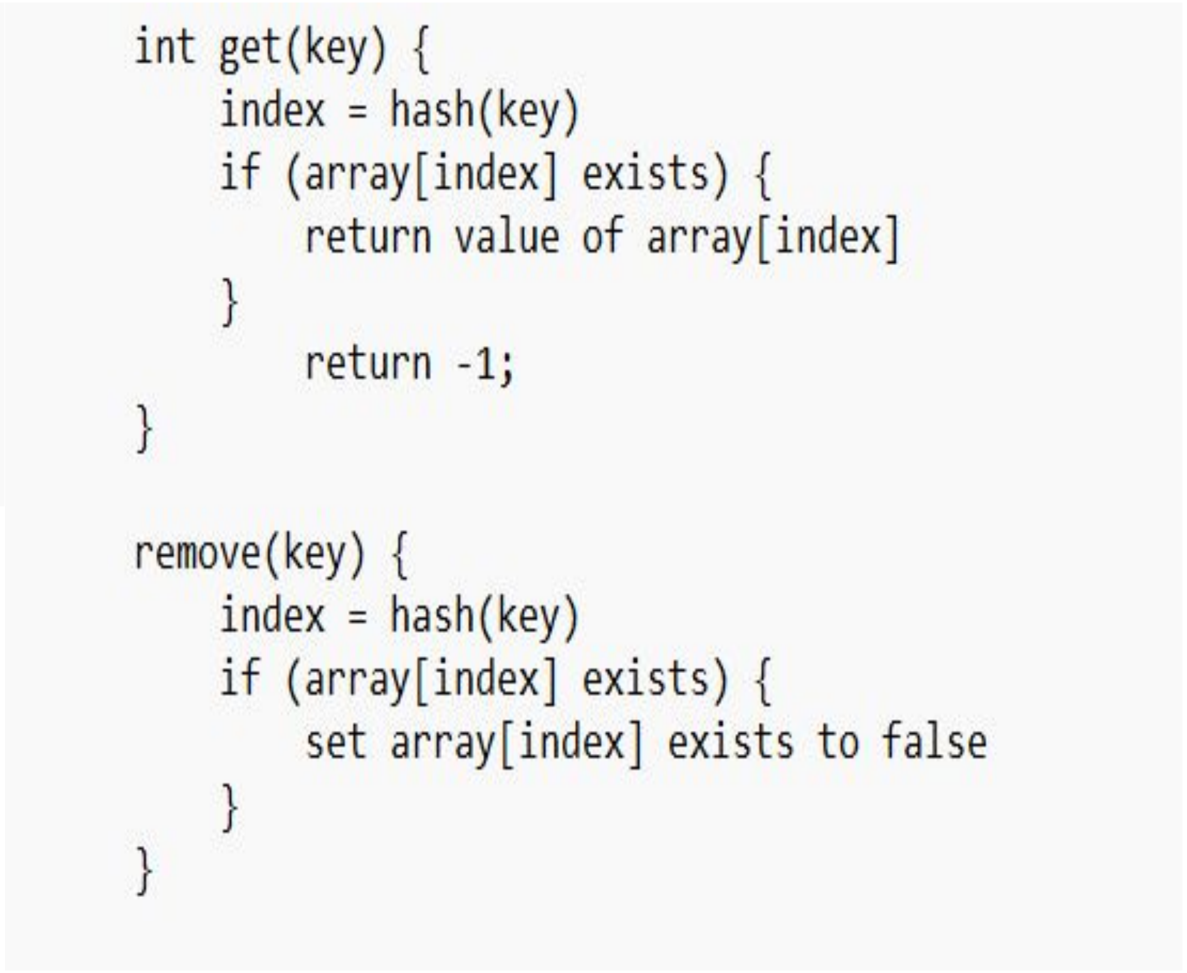
Hash Tables

- a hash table is an array of size Tsize
  - has index positions 0 .. Tsize-1
- two types of hash tables
  - open hash table
    - array element type is a <key, value> pair
    - all items stored in the array
  - chained hash table
    - element type is a pointer to a linked list of nodes containing <key, value> pairs
    - items are stored in the linked list nodes
- keys are used to generate an array index
  - home address (0 .. Tsize-1)

Sudo-Code



```
    put(key, value) {
      index = hash(key)
      if (array[index] does not exists) {
          set array[index] exists to true
          set array[index] key
          set array[index] value
      }

  }
```



```
int get(key) {
    index = hash(key)
    if (array[index] exists) {
        return value of array[index]
    }
        return -1;
}

remove(key) {
    index = hash(key)
    if (array[index] exists) {
        set array[index] exists to false
    }
}
```

Hash Function/ Hashing Algorithm:

Hash Algorithm vary depending upon the Input Key, the size of key, the size of table and the type of keys. There are many hash algorithms for other purposes such as Data Encryption etc. But here we will discuss some of Hashing Algorithms used in hash tables depending upon the input key:

- String/Character Key:

  Different logics can be used and implemented in a hash function to output an index for input string or character key based on the mathematical operations on the ASCII values of the characters.

- Integer Key:

  When the key of input value is numeric, the hash function can be implemented using various mathematical operational logics to calculate a viable and unique index to place the value at in the array.

Integer key Hash Functions:

- Division/Mod Method:

$h(K) = k \bmod M$

Here,

***k** is the key value, and*

***M** is the size of the hash table.*

This method is relatively fast but has high chances of collision.

- Mid-Square Method:

$h(K) = h(k \times k)$

Here,

***k** is the key value.*

This method is not good for large values of k and there is also a good chance of collision in this.

- Multiplication Method:

$h(K) = \text{floor}\,(M\,(kA \bmod 1))$

Here,

***M** is the size of the hash table.*

***k** is the key value.*

***A** is a constant value.* $0 < A < 1$

This method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

Integer key Hash Function Examples:

- Division/Mod Method:

$$k = 12345$$
$$M = 95$$
$$\begin{aligned} h(12345) &= 12345 \bmod 95 \\ &= 90 \end{aligned}$$

$$k = 1276$$
$$M = 11$$
$$\begin{aligned} h(1276) &= 1276 \bmod 11 \\ &= 0 \end{aligned}$$

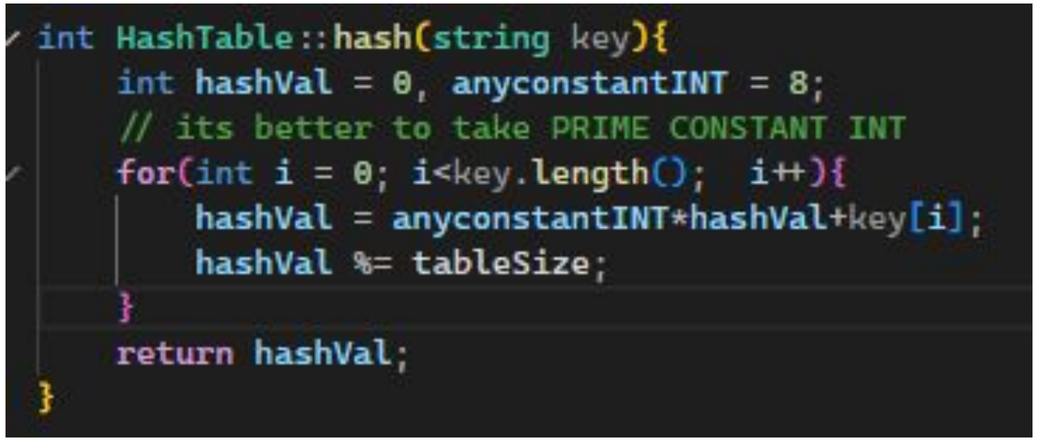
- Multiplication Method:

$$k = 12345$$
$$A = 0.357840$$
$$M = 100$$

$$\begin{aligned} h(12345) &= floor[\,100\,(12345 * 0.357840 \bmod 1)] \\ &= floor[\,100\,(4417.5348 \bmod 1)\,] \\ &= floor[\,100\,(0.5348)\,] \\ &= floor[\,53.48\,] \\ &= 53 \end{aligned}$$

String key Hash Functions:

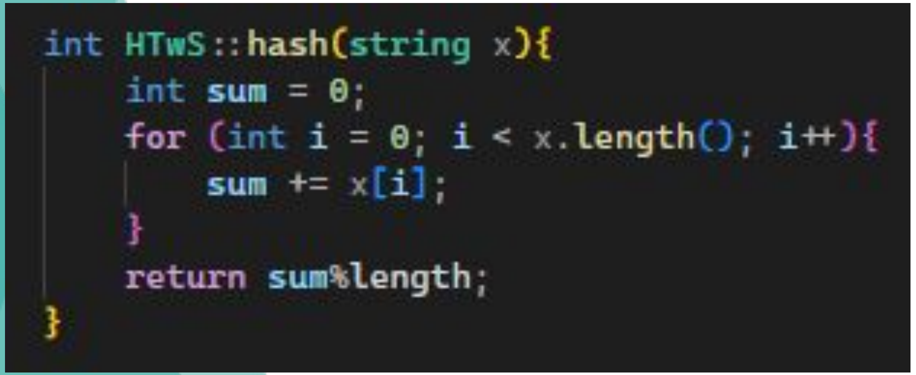
- Another Hash Function:



```
int HashTable::hash(string key){
    int hashVal = 0, anyconstantINT = 8;
    // its better to take PRIME CONSTANT INT
    for(int i = 0; i<key.length();  i++){
        hashVal = anyconstantINT*hashVal+key[i];
        hashVal %= tableSize;
    }
    return hashVal;
}
```

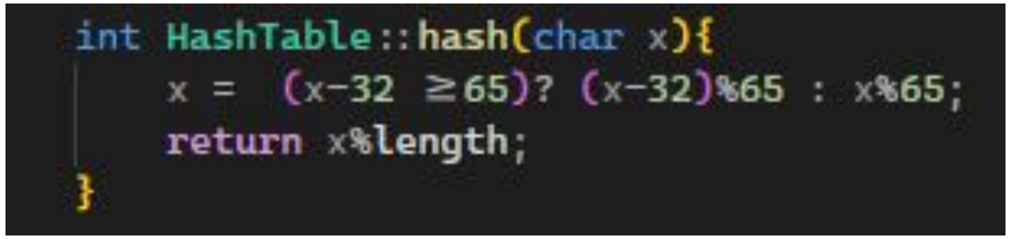
Primes are used because you have the best chance of obtaining a unique value when multiplying values by the prime number chosen and adding them all up

- ASCII Value SUM:



```
int HTwS::hash(string x){
    int sum = 0;
    for (int i = 0; i < x.length(); i++){
        sum += x[i];
    }
    return sum%length;
}
```

- Hash Function for Char Key:



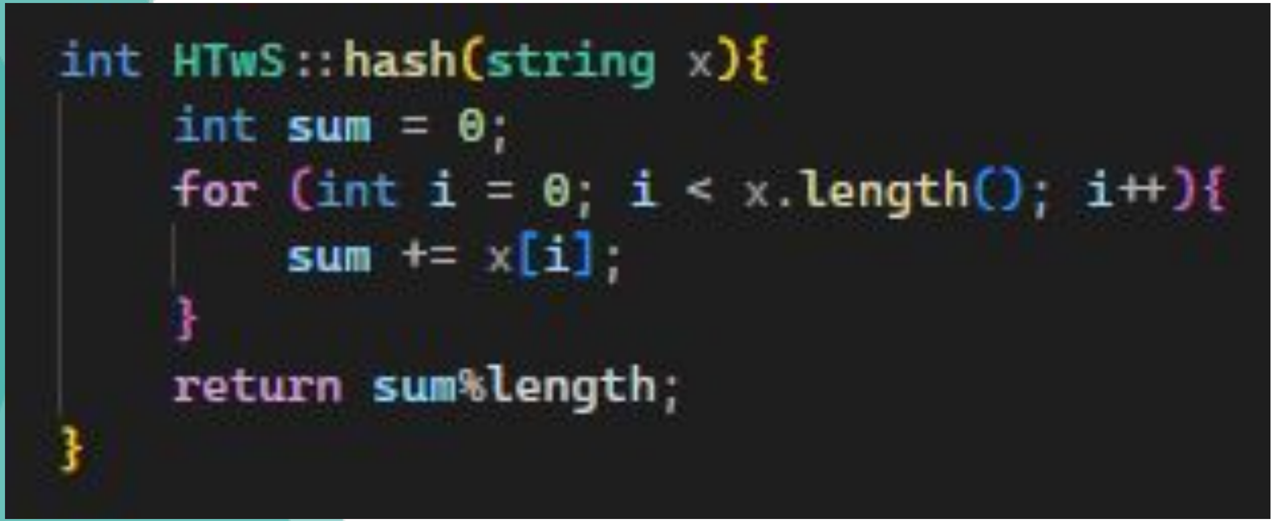
```
int HashTable::hash(char x){
    x =  (x-32 ≥65)? (x-32)%65 : x%65;
    return x%length;
}
```

It returns same hash value for the same character either upper or lower case

Note that if two strings are equal, their hash values should also be equal. But the inverse need not be true.

Collisions Example:

- String key Hash Function:



```
int HTwS::hash(string x){
    int sum = 0;
    for (int i = 0; i < x.length(); i++){
        sum += x[i];
    }
    return sum%length;
}
```

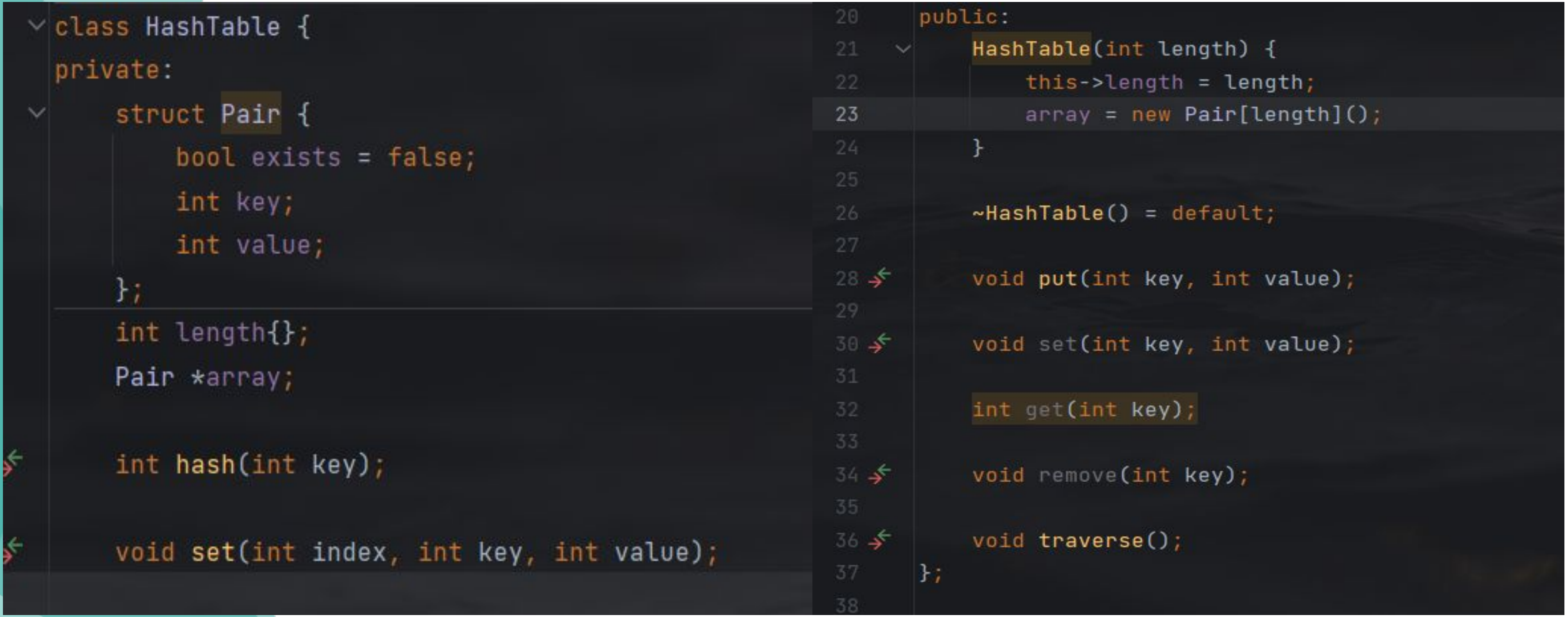
This is an example of a bad hash function resulting in collisions.

Let's give it an input of a student whose number we want to store against it. (Student Name -> KEY, Marks -> Value)

Let name be MAHAD it produces sum 475 whose modulus can be taken to produce and index.

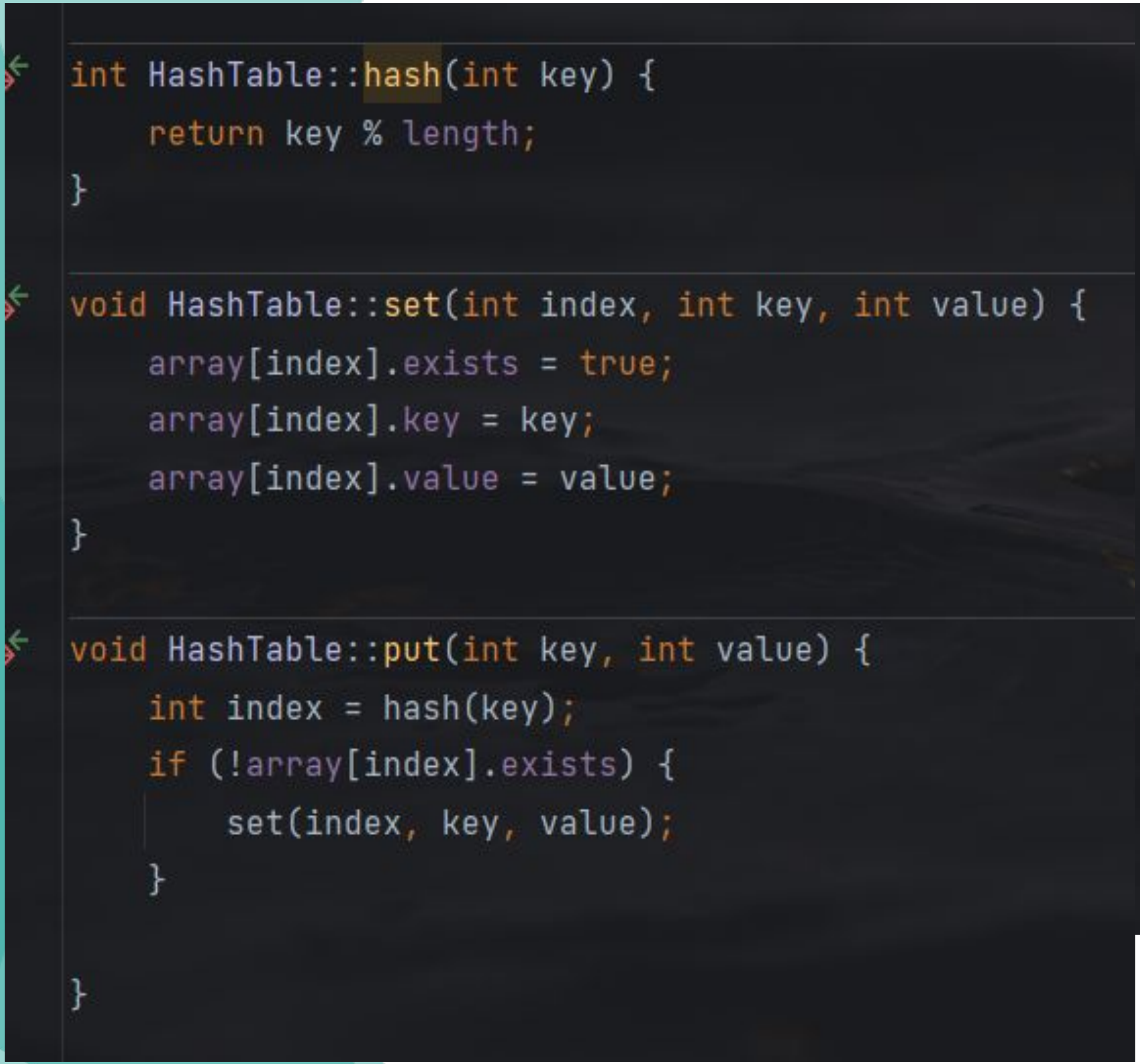
Let another name be AHMAD it will also produce its sum 475 and value of both keys will be placed at same index which will cause collision.

Implementing a simple Hash Table:
(Considering no collisions)

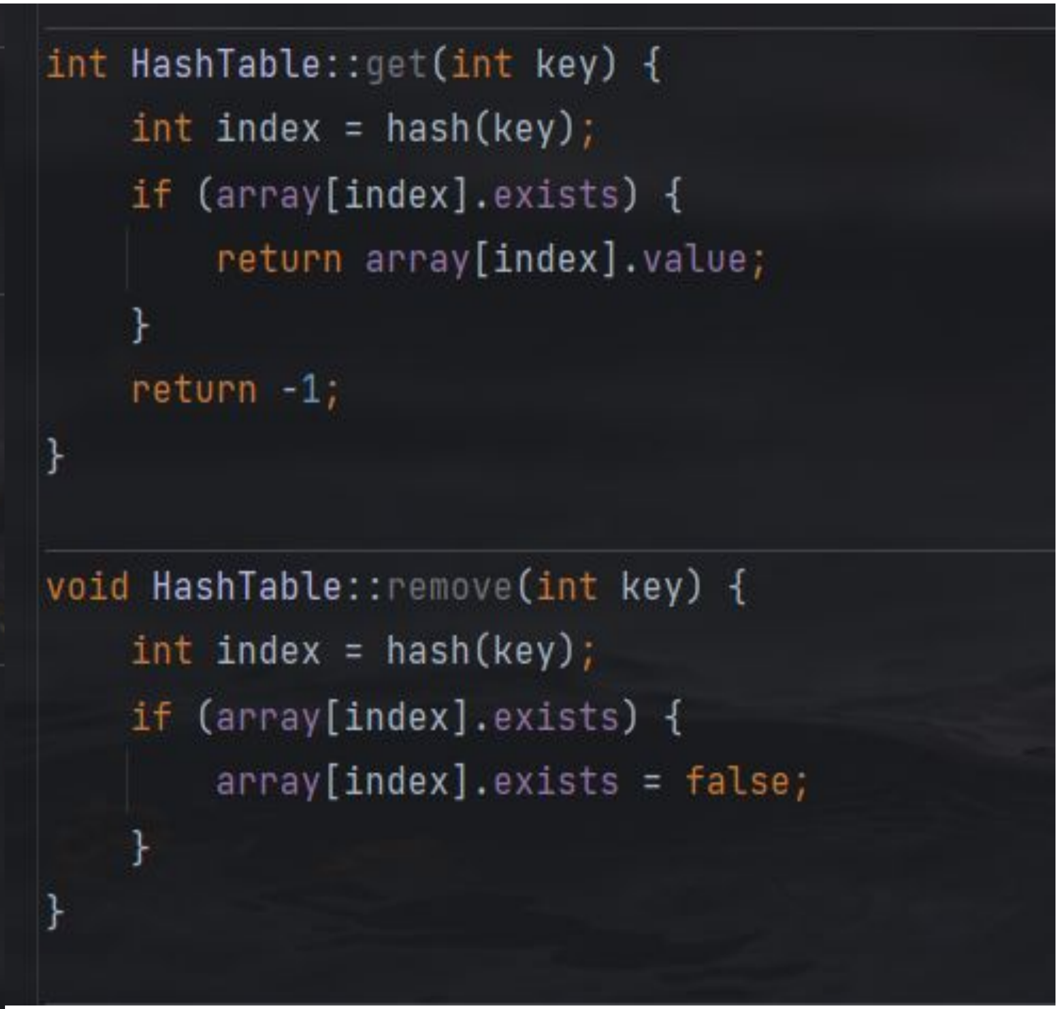


```
class HashTable {
private:
    struct Pair {
        bool exists = false;
        int key;
        int value;
    };
    int length{};
    Pair *array;

    int hash(int key);

    void set(int index, int key, int value);
```

```
public:
    HashTable(int length) {
        this->length = length;
        array = new Pair[length]();
    }

    ~HashTable() = default;

    void put(int key, int value);

    void set(int key, int value);

    int get(int key);

    void remove(int key);

    void traverse();
};
```



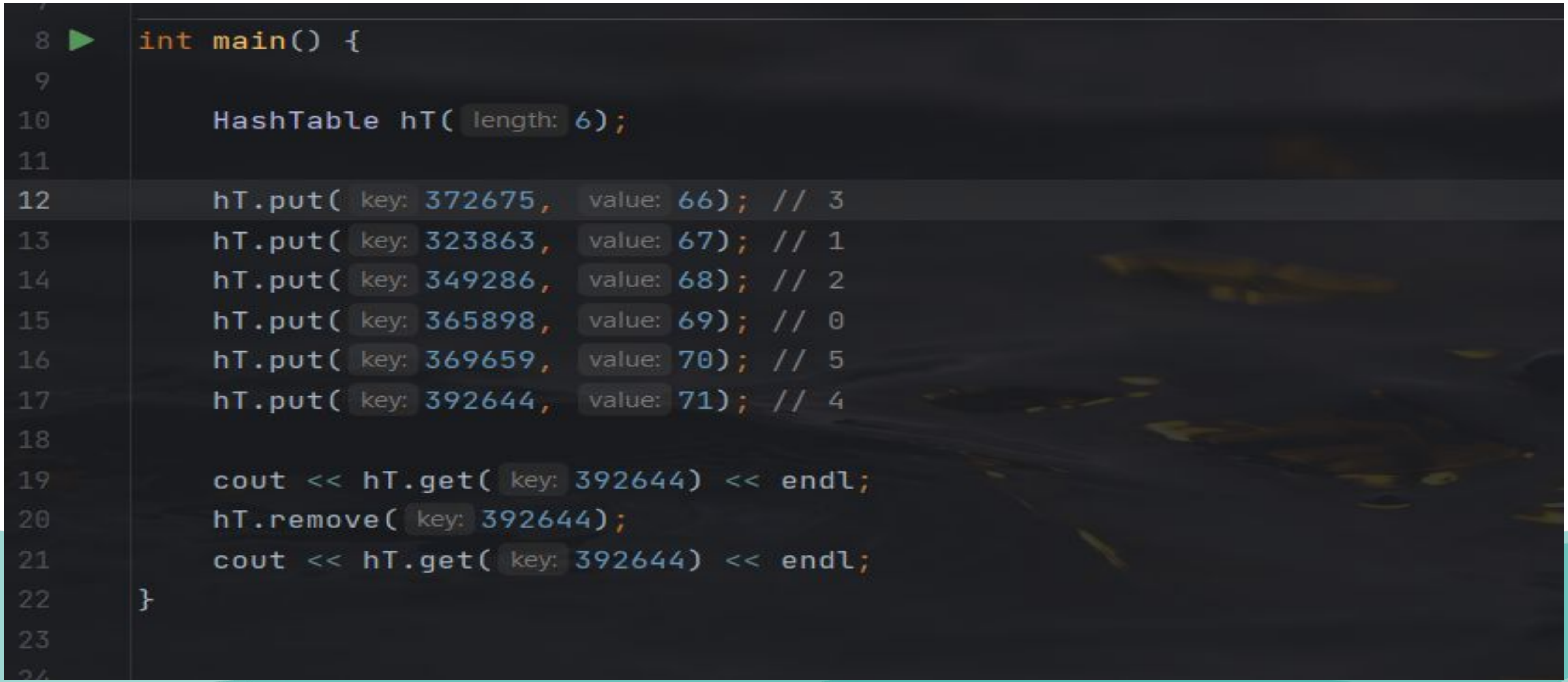
```
int HashTable::hash(int key) {
    return key % length;
}

void HashTable::set(int index, int key, int value) {
    array[index].exists = true;
    array[index].key = key;
    array[index].value = value;
}

void HashTable::put(int key, int value) {
    int index = hash(key);
    if (!array[index].exists) {
        set(index, key, value);
    }

}
```



```
int HashTable::get(int key) {
    int index = hash(key);
    if (array[index].exists) {
        return array[index].value;
    }
    return -1;
}

void HashTable::remove(int key) {
    int index = hash(key);
    if (array[index].exists) {
        array[index].exists = false;
    }
}
```

Implementing a simple Hash Table: (Considering no collisions)

Main Method for Dry Run



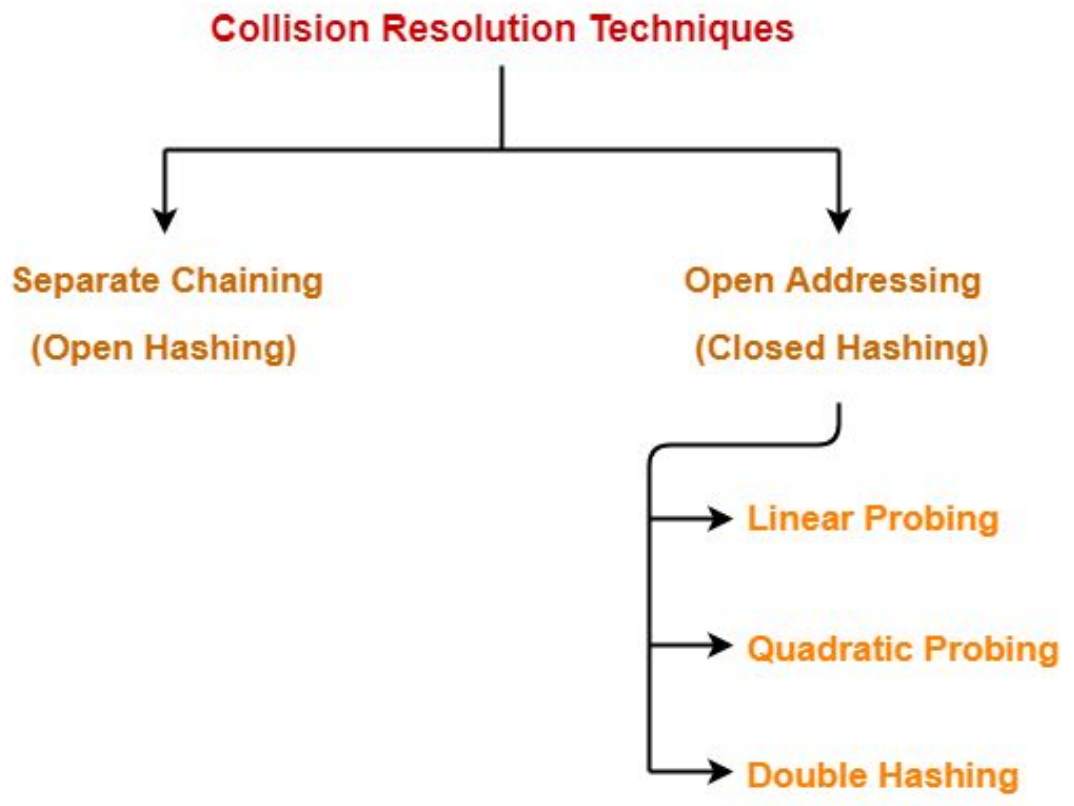
```
int main() {

    HashTable hT( length: 6);

    hT.put( key: 372675,  value: 66); // 3
    hT.put( key: 323863,  value: 67); // 1
    hT.put( key: 349286,  value: 68); // 2
    hT.put( key: 365898,  value: 69); // 0
    hT.put( key: 369659,  value: 70); // 5
    hT.put( key: 392644,  value: 71); // 4

    cout << hT.get( key: 392644) << endl;
    hT.remove( key: 392644);
    cout << hT.get( key: 392644) << endl;
}
```

Strategies to handle Hash Collisions:

In case no collision is possible (either our algorithm is ideal, or our keys are too different), we can simply use an algorithm to calculate index and place the values in an array.

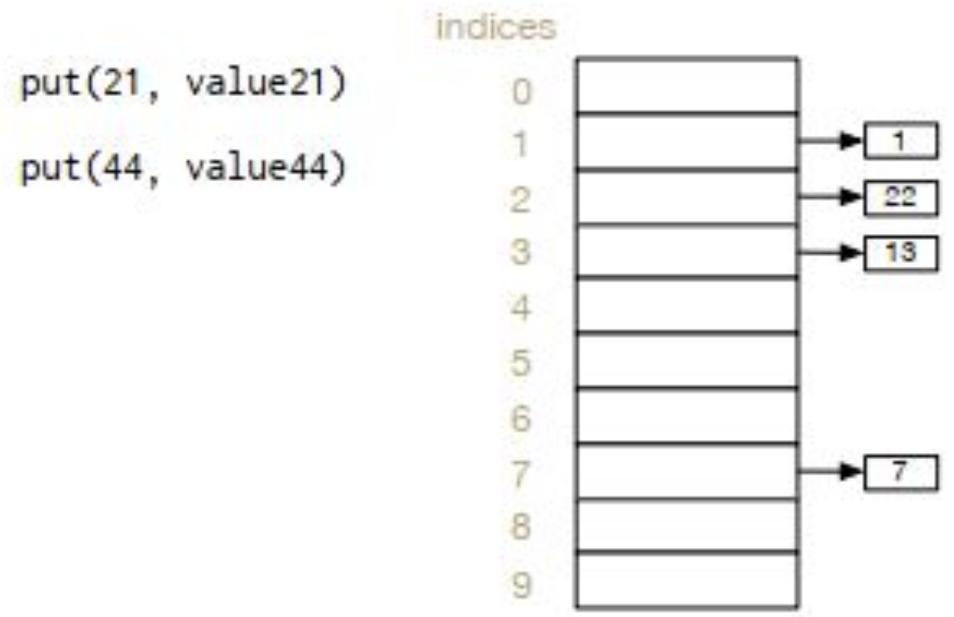
But if there is a possibility of Collision, we use the following techniques to tackle:

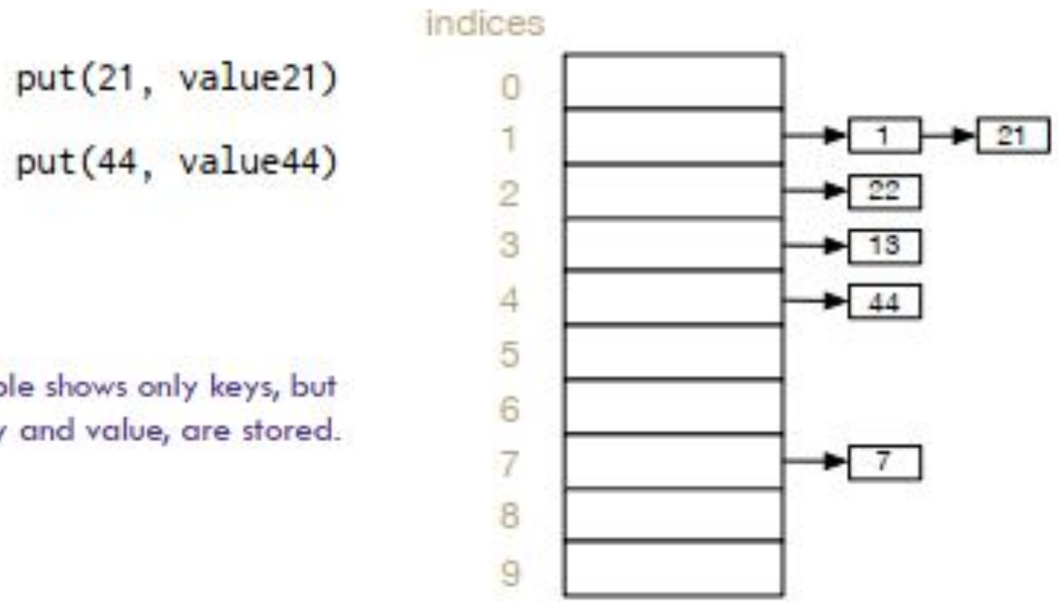
- Open Hashing or Separate Chaining:
- Closed Hashing or Open Addressing:
  - Linear Probing:
  - Quadratic Probing:
  - Double Hashing:



Separate Chaining:

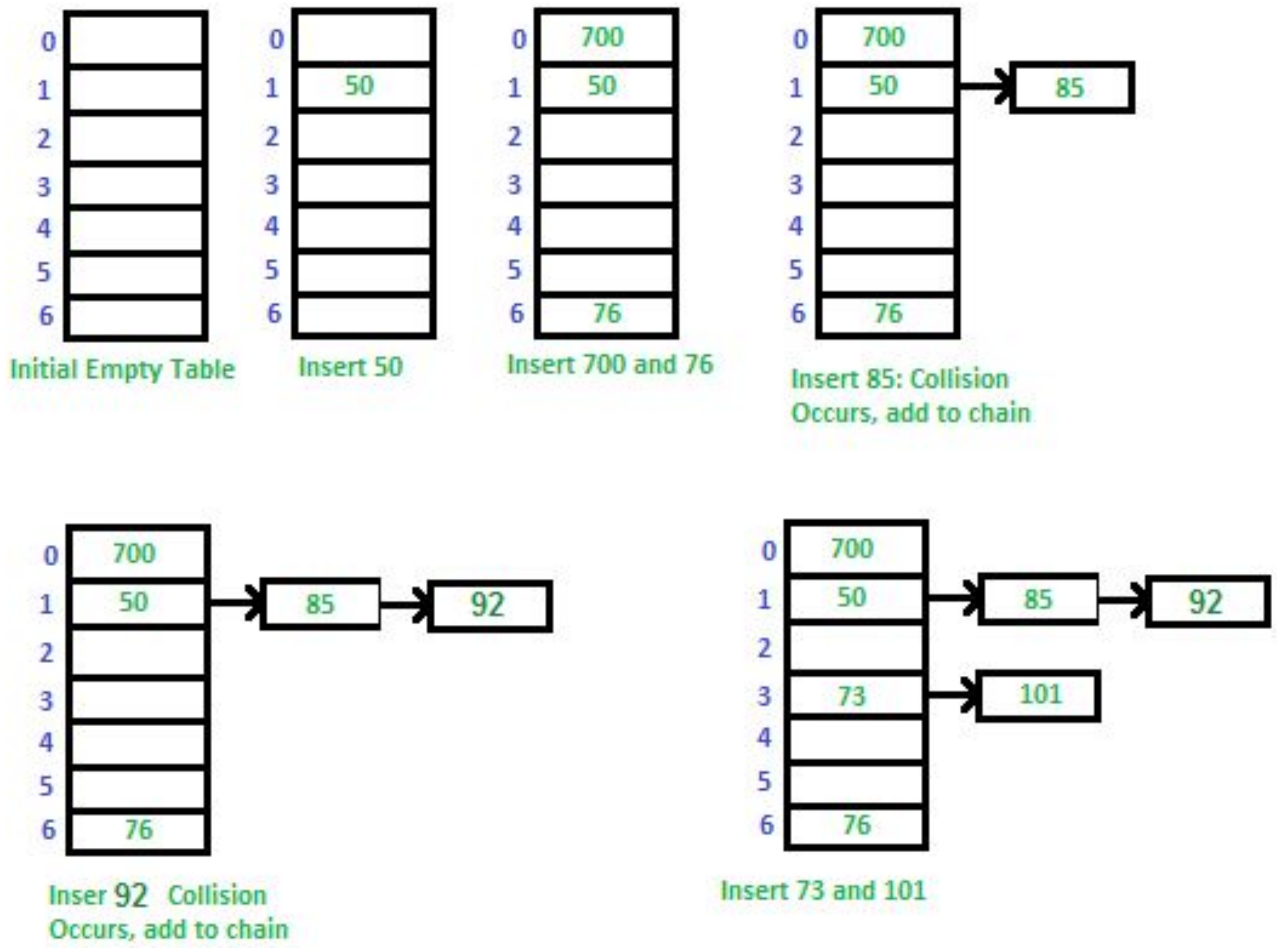
- Separate chaining is a collision resolution strategy where collisions are resolved by storing all colliding keys in the same slot (using linked list or some other data structure).
- Each slot stores a pointer to another data structure (usually a linked list or an AVL tree.





Note: For simplicity, the table shows only keys, but in each slot/node both, key and value, are stored.

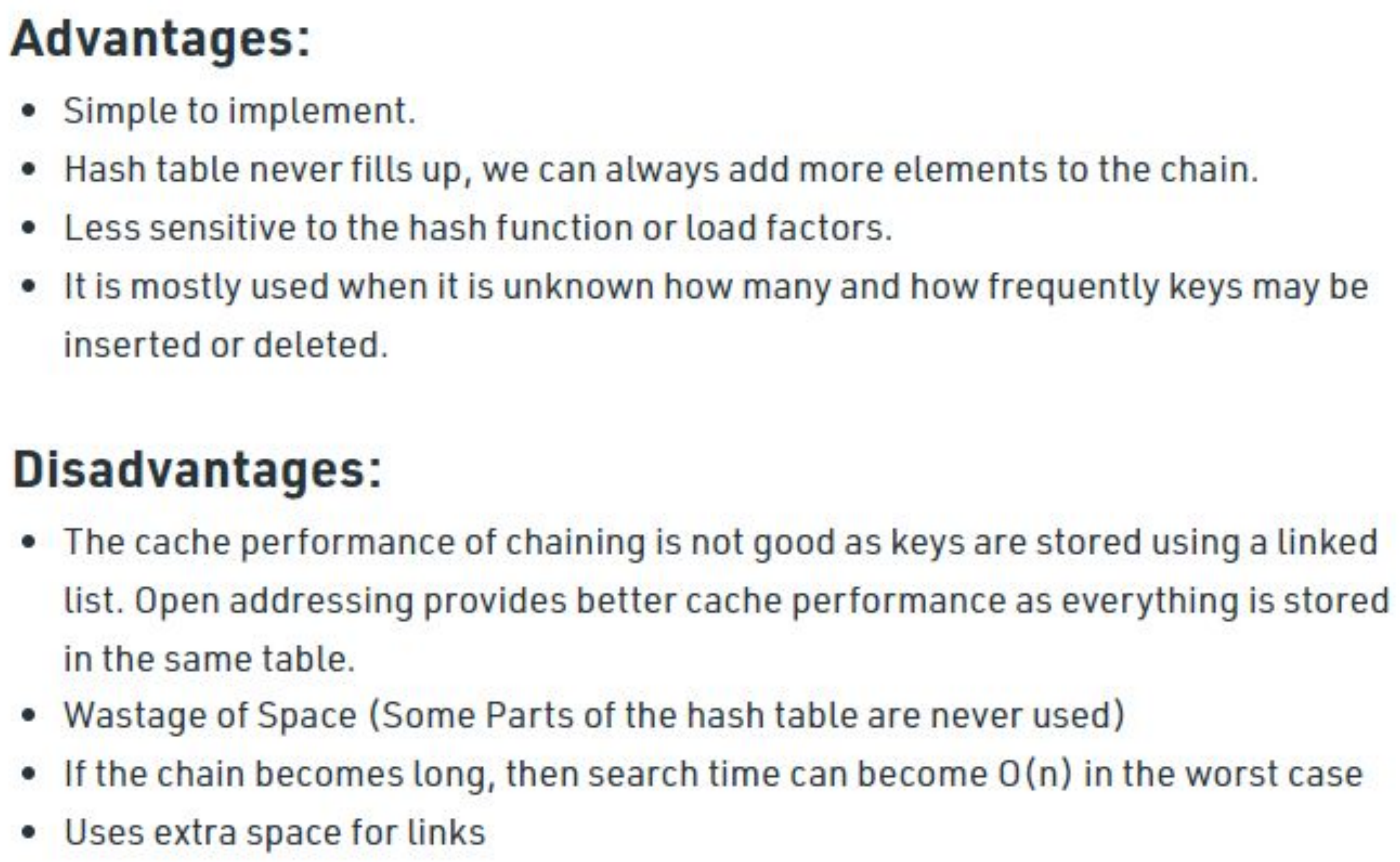
Separate Chaining:


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, add to chain
Inser 92 Collision Occurs, add to chain
Insert 73 and 101

Separate Chaining:

Running Time		
insert	BEST:	O(1)
	WORST:	O(n)
find	BEST:	O(1)
	WORST:	O(n)
delete	BEST:	O(1)
	WORST:	O(n)

(if insertions are always at the end of the linked list)



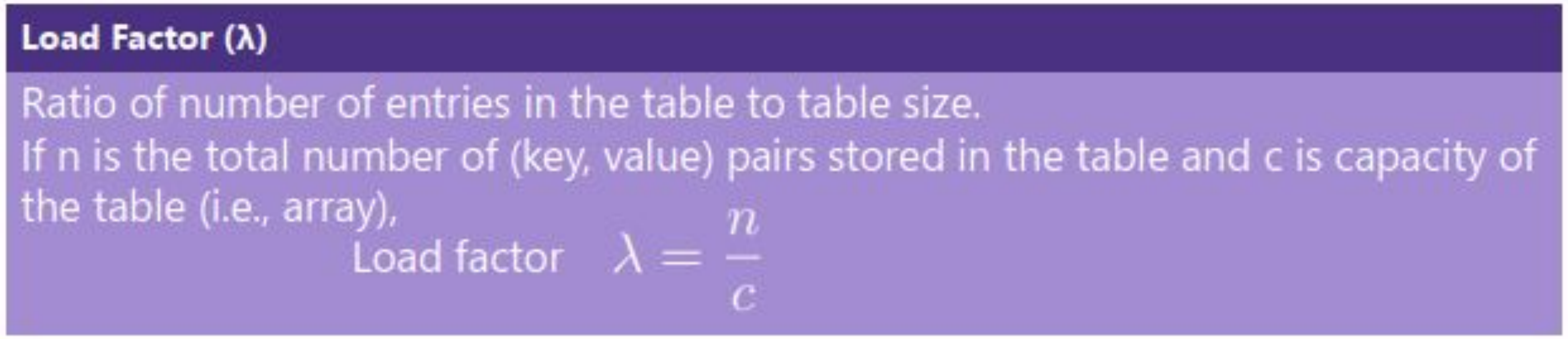
Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Performance of Separate Chaining:



Load Factor (λ)

Ratio of number of entries in the table to table size.
If n is the total number of (key, value) pairs stored in the table and c is capacity of the table (i.e., array),

Load factor $\lambda = \frac{n}{c}$

n = Number of keys to be inserted in hash table
c = Number of slots in hash table

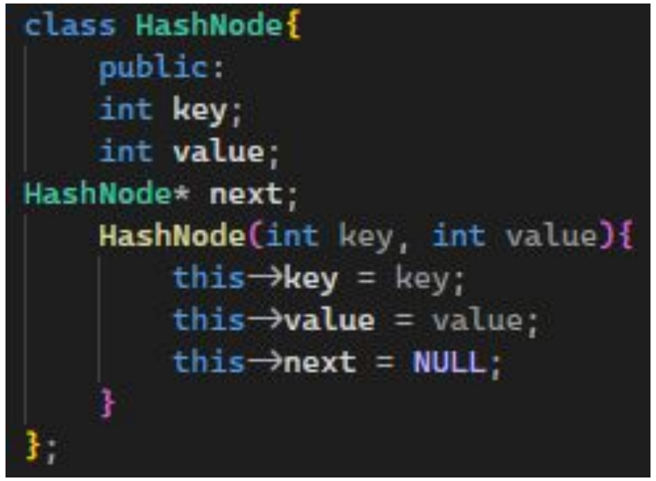
Load factor λ = n/c

Average Cases of Run Time Complexity:

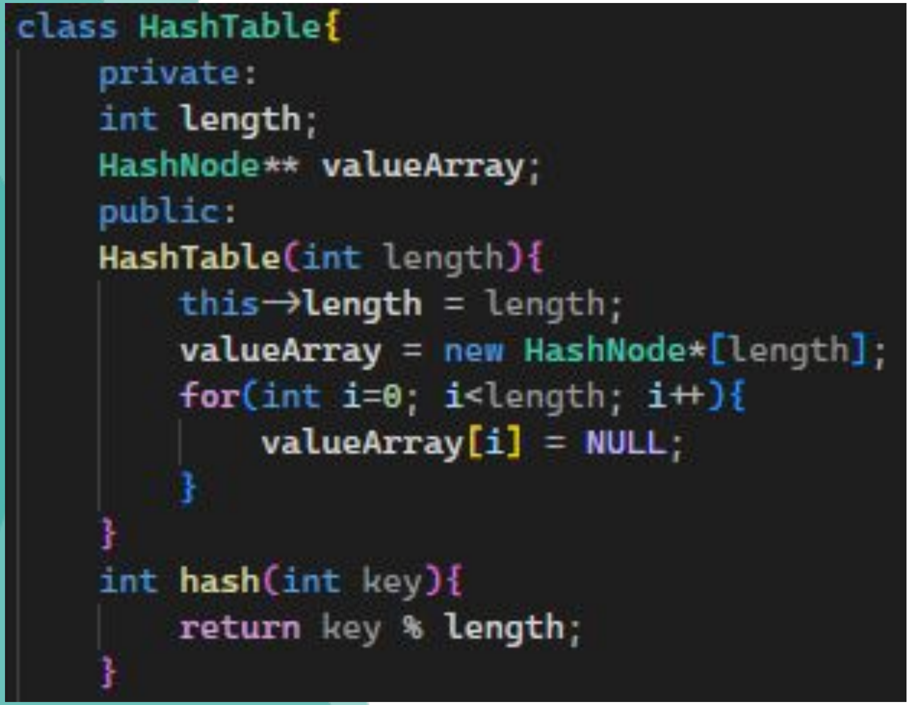
Expected time to search = O(1 + λ)
Expected time to delete = O(1 + λ)

Time to insert = O(1)
Time complexity of search insert and delete is O(1) if λ is 1

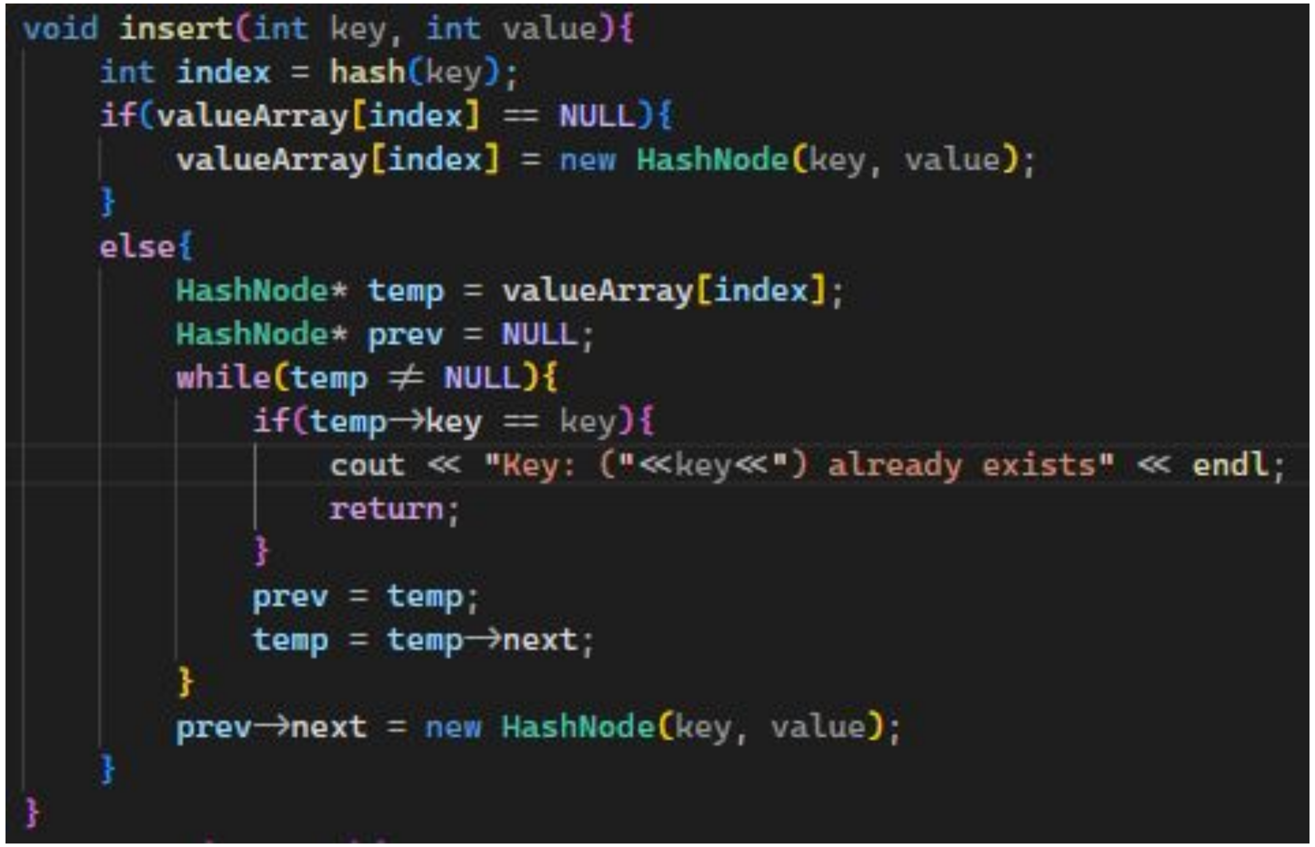
Implementing a Hash Table:
(Collision handling by Separate Chaining)



```
class HashNode{
    public:
    int key;
    int value;
HashNode* next;
    HashNode(int key, int value){
        this->key = key;
        this->value = value;
        this->next = NULL;
    }
};
```

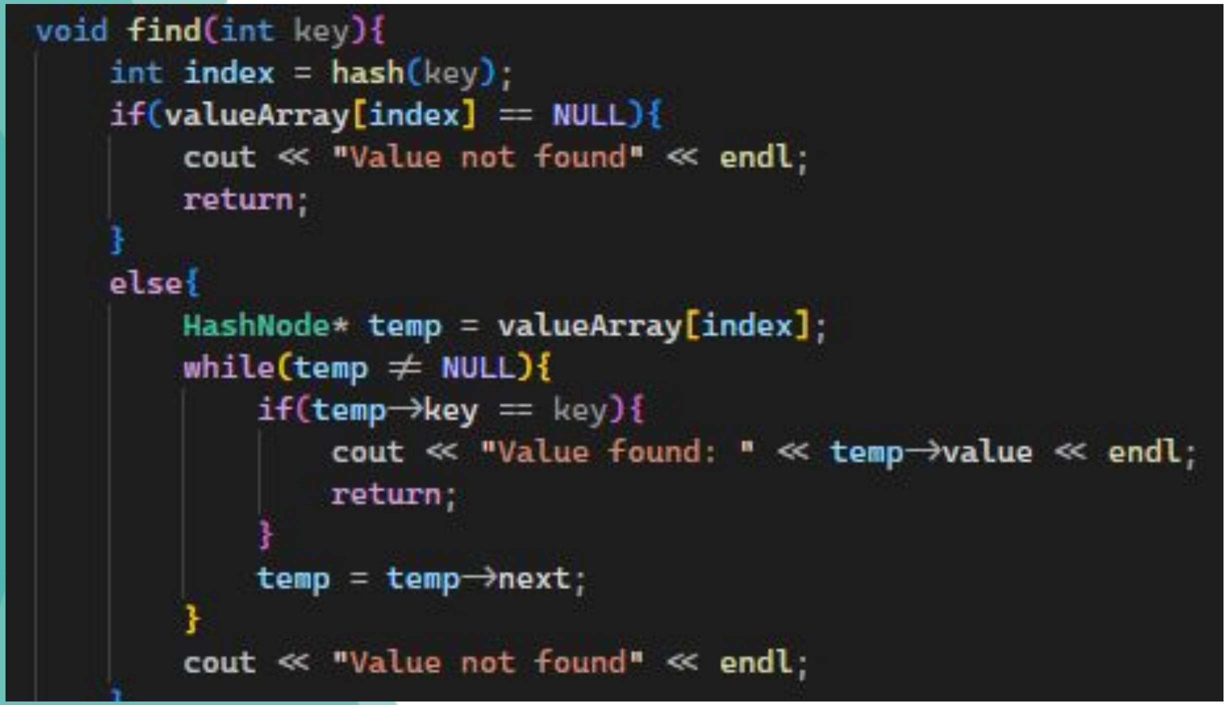


```
class HashTable{
    private:
    int length;
    HashNode** valueArray;
    public:
    HashTable(int length){
        this->length = length;
        valueArray = new HashNode*[length];
        for(int i=0; i<length; i++){
            valueArray[i] = NULL;
        }
    }
    int hash(int key){
        return key % length;
    }
```

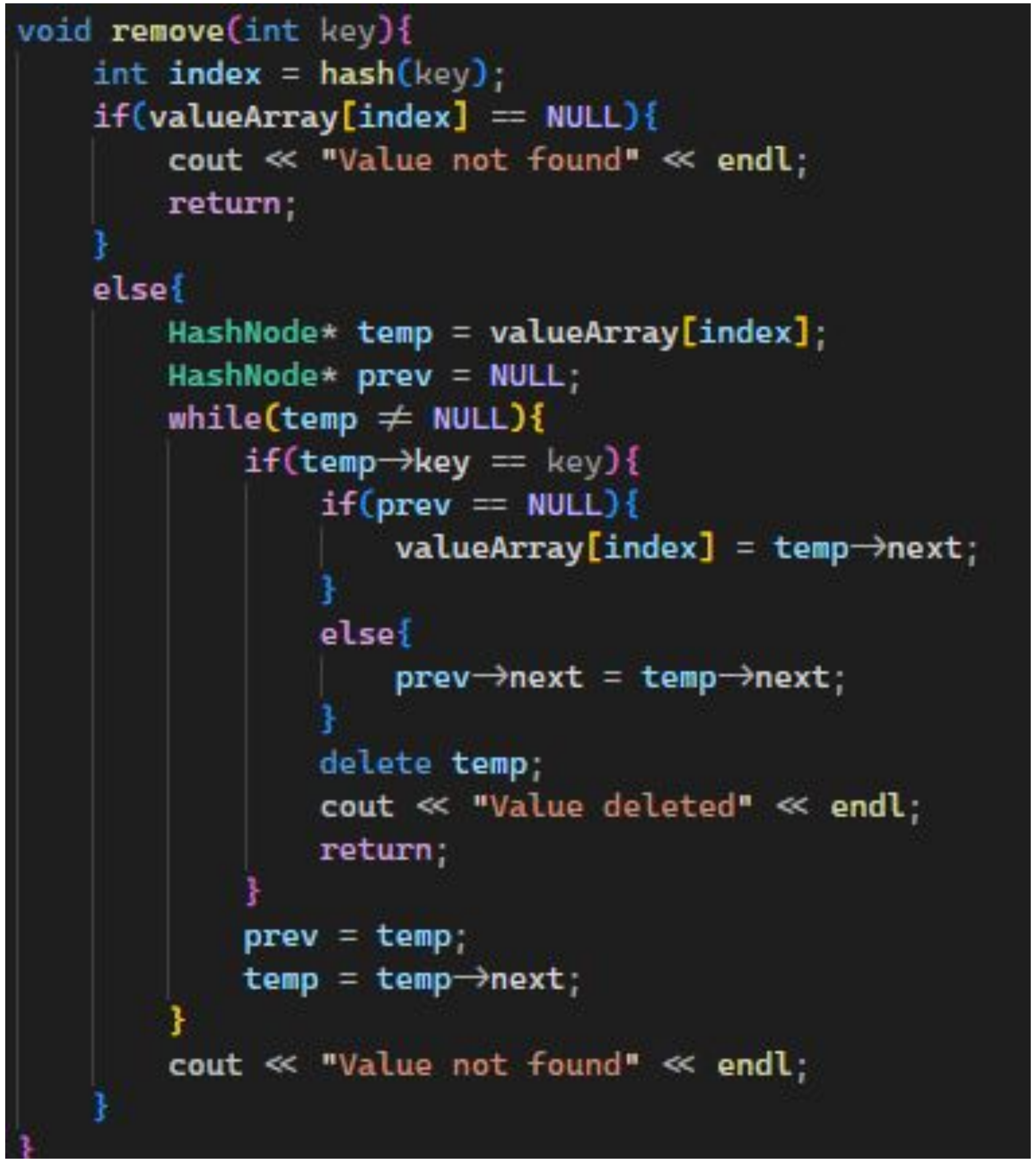


```
void insert(int key, int value){
    int index = hash(key);
    if(valueArray[index] == NULL){
        valueArray[index] = new HashNode(key, value);
    }
    else{
        HashNode* temp = valueArray[index];
        HashNode* prev = NULL;
        while(temp != NULL){
            if(temp->key == key){
                cout << "Key: ("<<key<<") already exists" << endl;
                return;
            }
            prev = temp;
            temp = temp->next;
        }
        prev->next = new HashNode(key, value);
    }
}
```

Implementing a Hash Table:
(Collision handling by Separate Chaining)

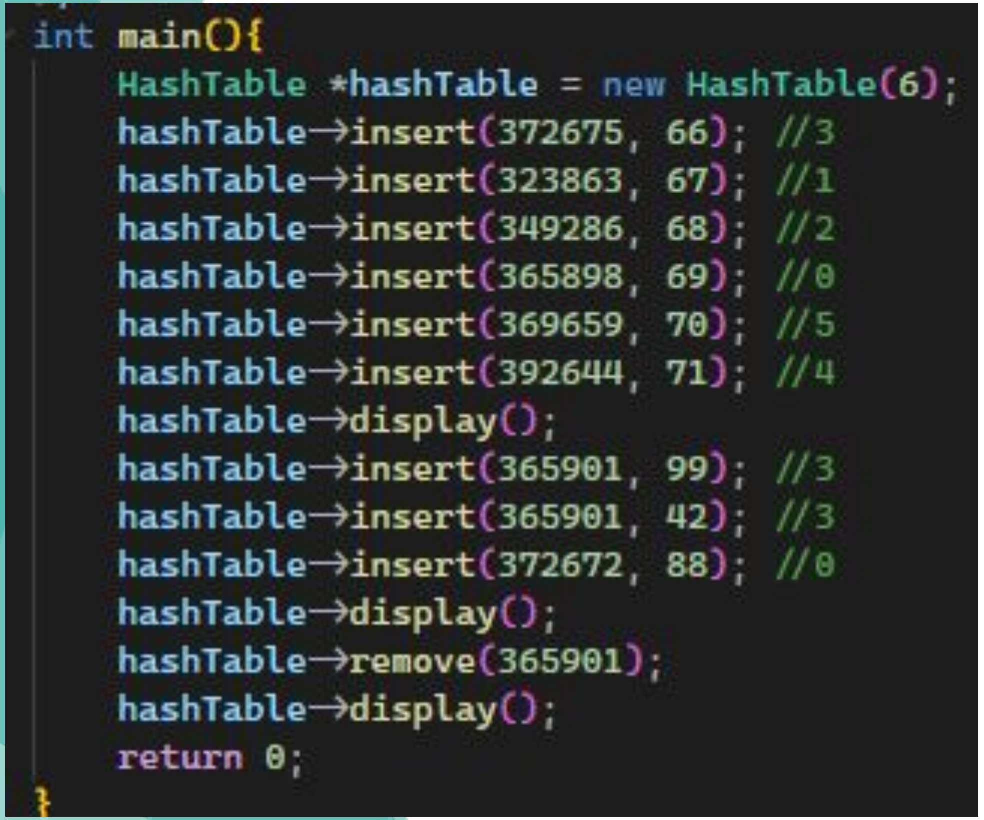


```cpp
void find(int key){
    int index = hash(key);
    if(valueArray[index] == NULL){
        cout << "Value not found" << endl;
        return;
    }
    else{
        HashNode* temp = valueArray[index];
        while(temp != NULL){
            if(temp->key == key){
                cout << "Value found: " << temp->value << endl;
                return;
            }
            temp = temp->next;
        }
        cout << "Value not found" << endl;
```

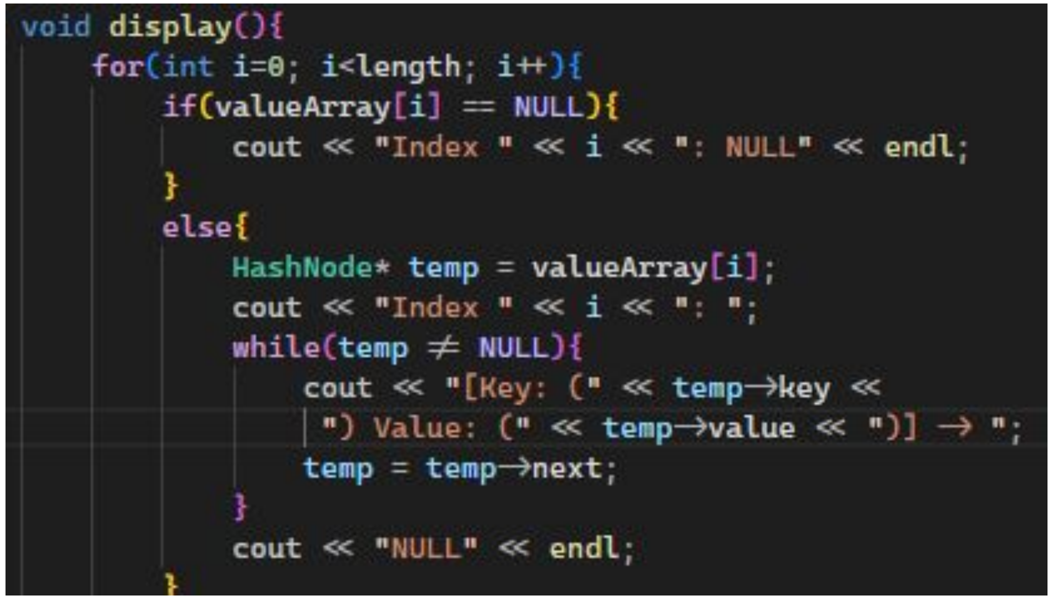


```cpp
void remove(int key){
    int index = hash(key);
    if(valueArray[index] == NULL){
        cout << "Value not found" << endl;
        return;
    }
    else{
        HashNode* temp = valueArray[index];
        HashNode* prev = NULL;
        while(temp != NULL){
            if(temp->key == key){
                if(prev == NULL){
                    valueArray[index] = temp->next;
                }
                else{
                    prev->next = temp->next;
                }
                delete temp;
                cout << "Value deleted" << endl;
                return;
            }
            prev = temp;
            temp = temp->next;
        }
        cout << "Value not found" << endl;
    }
}
```

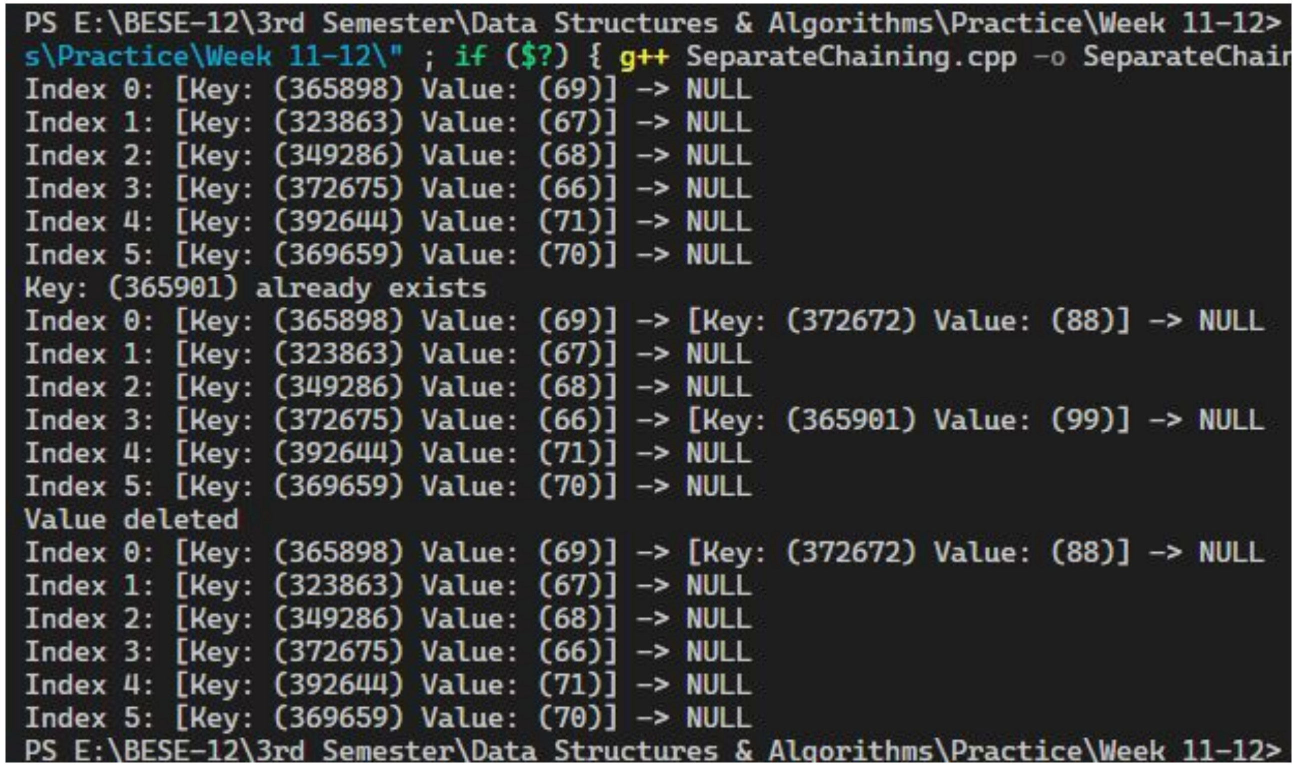
Implementing a Hash Table: (Collision handling by Separate Chaining)



```cpp
int main(){
    HashTable *hashTable = new HashTable(6);
    hashTable->insert(372675, 66); //3
    hashTable->insert(323863, 67); //1
    hashTable->insert(349286, 68); //2
    hashTable->insert(365898, 69); //0
    hashTable->insert(369659, 70); //5
    hashTable->insert(392644, 71); //4
    hashTable->display();
    hashTable->insert(365901, 99); //3
    hashTable->insert(365901, 42); //3
    hashTable->insert(372672, 88); //0
    hashTable->display();
    hashTable->remove(365901);
    hashTable->display();
    return 0;
}
```



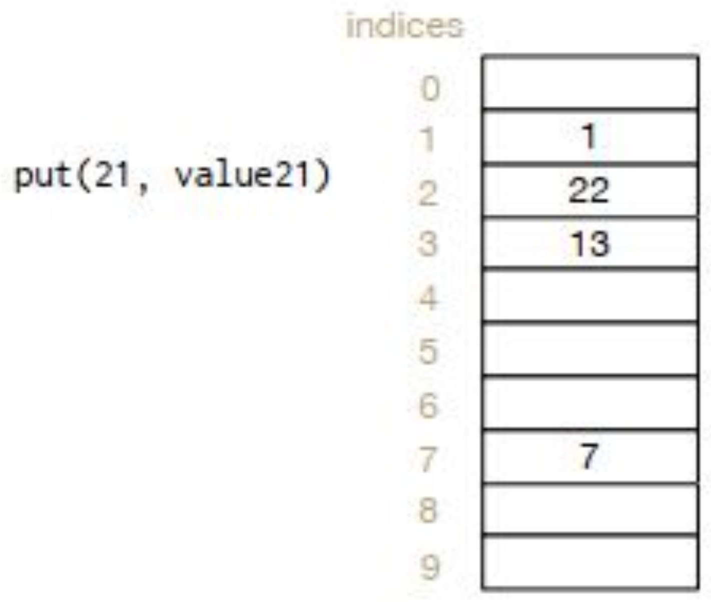
```cpp
void display(){
    for(int i=0; i<length; i++){
        if(valueArray[i] == NULL){
            cout << "Index " << i << ": NULL" << endl;
        }
        else{
            HashNode* temp = valueArray[i];
            cout << "Index " << i << ": ";
            while(temp != NULL){
                cout << "[Key: (" << temp->key <<
                ") Value: (" << temp->value << ")] -> ";
                temp = temp->next;
            }
            cout << "NULL" << endl;
        }
```

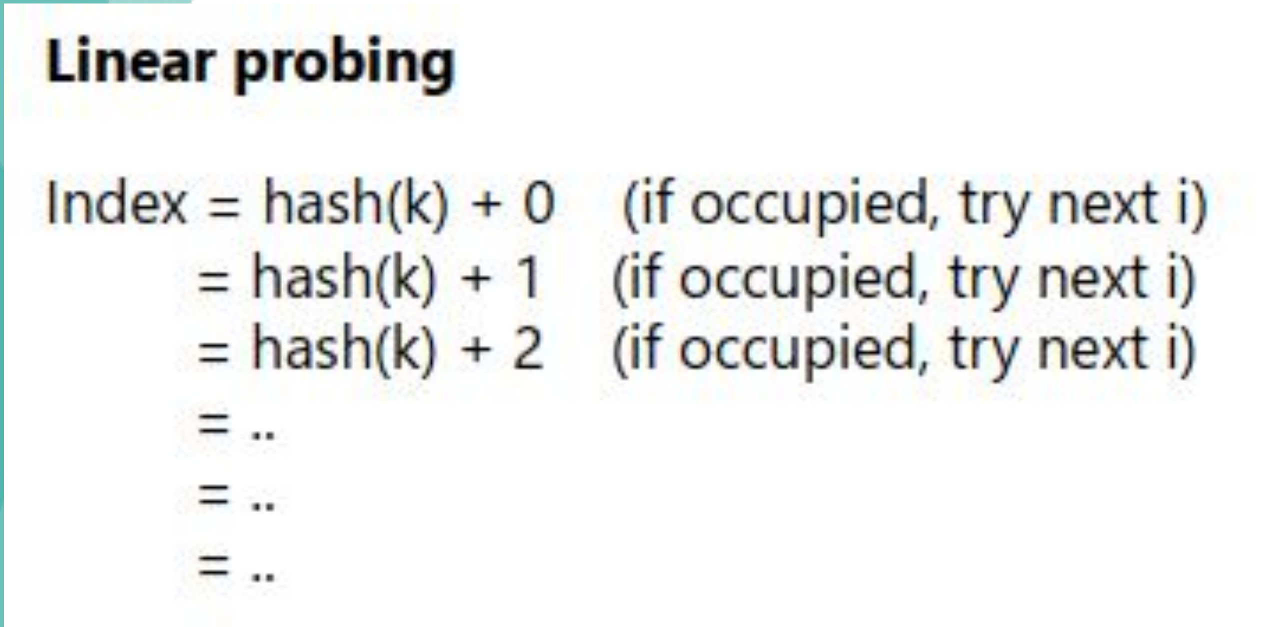


```
PS E:\BESE-12\3rd Semester\Data Structures & Algorithms\Practice\Week 11-12>
s\Practice\Week 11-12\" ; if ($?) { g++ SeparateChaining.cpp -o SeparateChai
Index 0: [Key: (365898) Value: (69)] -> NULL
Index 1: [Key: (323863) Value: (67)] -> NULL
Index 2: [Key: (349286) Value: (68)] -> NULL
Index 3: [Key: (372675) Value: (66)] -> NULL
Index 4: [Key: (392644) Value: (71)] -> NULL
Index 5: [Key: (369659) Value: (70)] -> NULL
Key: (365901) already exists
Index 0: [Key: (365898) Value: (69)] -> [Key: (372672) Value: (88)] -> NULL
Index 1: [Key: (323863) Value: (67)] -> NULL
Index 2: [Key: (349286) Value: (68)] -> NULL
Index 3: [Key: (372675) Value: (66)] -> [Key: (365901) Value: (99)] -> NULL
Index 4: [Key: (392644) Value: (71)] -> NULL
Index 5: [Key: (369659) Value: (70)] -> NULL
Value deleted
Index 0: [Key: (365898) Value: (69)] -> [Key: (372672) Value: (88)] -> NULL
Index 1: [Key: (323863) Value: (67)] -> NULL
Index 2: [Key: (349286) Value: (68)] -> NULL
Index 3: [Key: (372675) Value: (66)] -> NULL
Index 4: [Key: (392644) Value: (71)] -> NULL
Index 5: [Key: (369659) Value: (70)] -> NULL
PS E:\BESE-12\3rd Semester\Data Structures & Algorithms\Practice\Week 11-12>
```

(Open Addressing Methods) (1) Linear Probing:

Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.




Linear probing

```
Index = hash(k) + 0   (if occupied, try next i)
      = hash(k) + 1   (if occupied, try next i)
      = hash(k) + 2   (if occupied, try next i)
      = ..
      = ..
      = ..
```

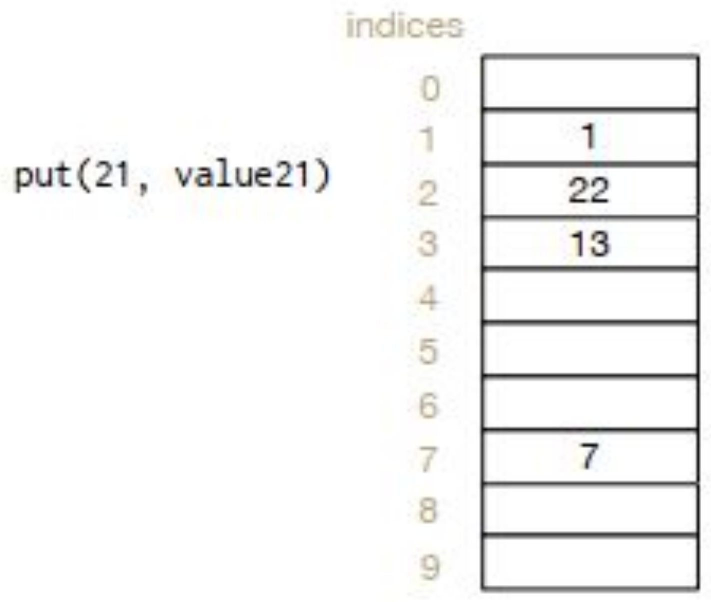
Simple Hash Function:
H(k) = k % length
Linear Probing Hash Function:
H'(k,i) = [H(k) + i] % length

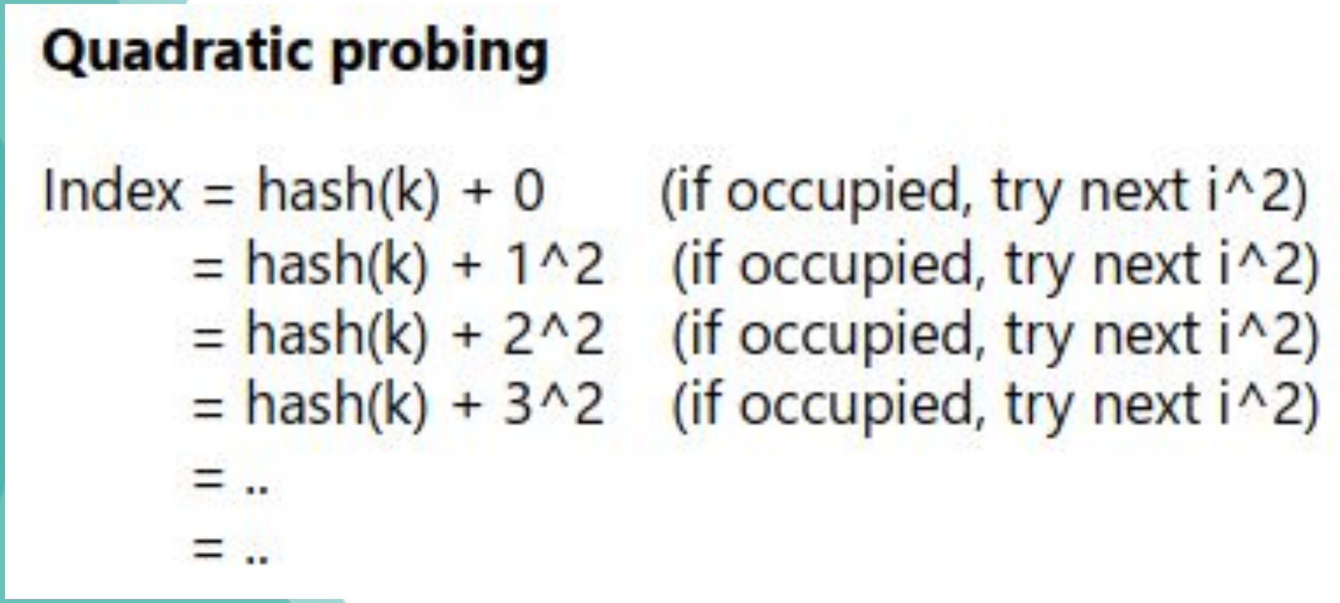
Example:
length = 10
keys/values are
43, 135, 72, 23, 99, 19, 82

Advantage: No Extra Space
Disadvantage: Search Time O(n), Deletion Difficult, Clustering (primary)

(Open Addressing Methods) (2) Quadratic Probing:

Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.





Quadratic probing

```
Index = hash(k) + 0       (if occupied, try next i^2)
      = hash(k) + 1^2   (if occupied, try next i^2)
      = hash(k) + 2^2   (if occupied, try next i^2)
      = hash(k) + 3^2   (if occupied, try next i^2)
      = ..
      = ..
```

Simple Hash Function:
H(k) = k % length
Quadratic Probing Hash Function:
H'(k,i) = [H(k) + i^2] % length

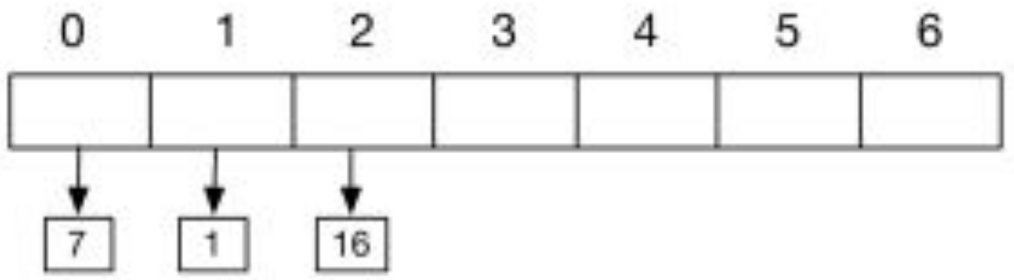
Example:
length = 10
keys/values are
42, 16, 91, 33, 18, 27, 36, 62

Advantage: No Extra Space, Clustering Resolved
Disadvantage: Search Time O(n), No Guarantee of finding free slot.

Example Question:

(Q4) Each table uses the hash function h(x) = x % 7, but different collision handling strategies.
Show where key 8 will be inserted in the following hash tables.

(4a) The following hash tables uses separate chaining



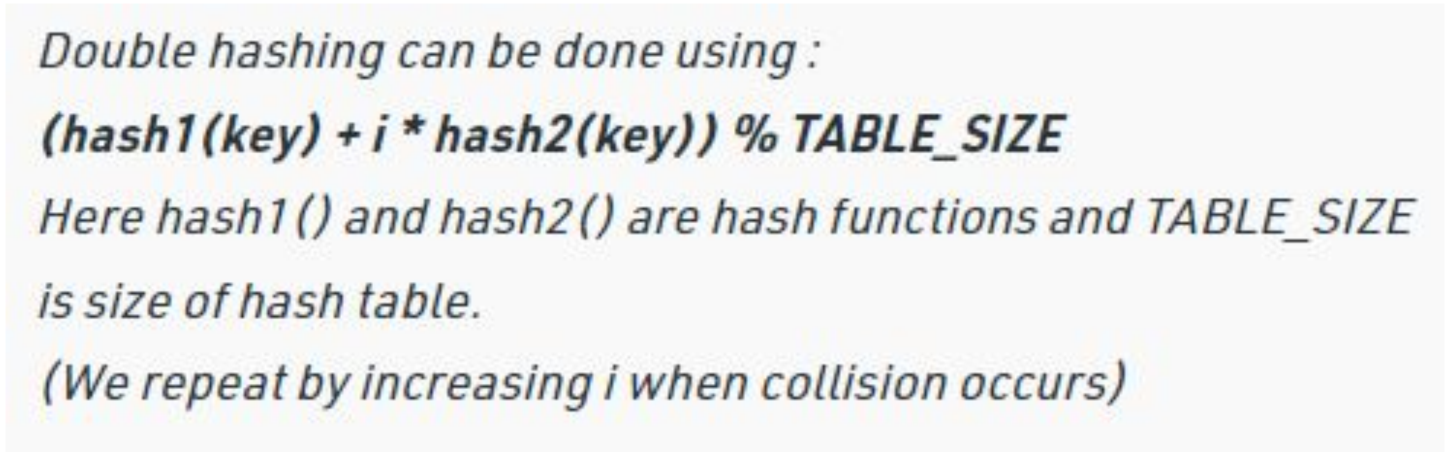
(4b) The following hash tables uses open addressing with linear probing

0	1	2	3	4	5	6
7	1	16				

(4c) The following hash tables uses open addressing with quadratic probing.

0	1	2	3	4	5	6
7	1	16				

(Open Addressing Methods) (3) Double Hashing:



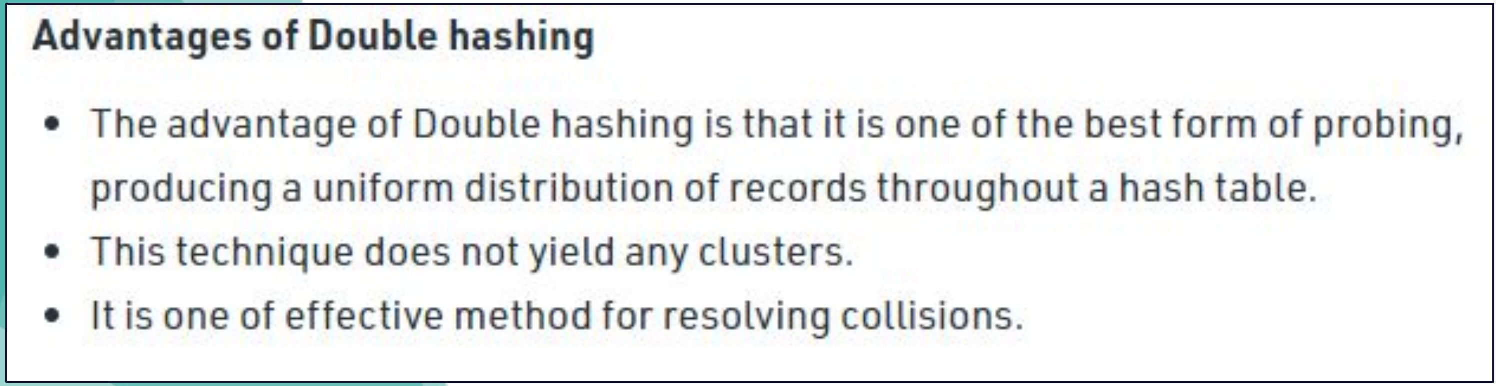
Double hashing can be done using :

(hash1(key) + i * hash2(key)) % TABLE_SIZE

Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table.

(We repeat by increasing i when collision occurs)

- **Double hashing** is a collision resolving technique in Open Addressed Hash tables.
- Double hashing uses the idea of applying a second hash function to key when a collision occurs



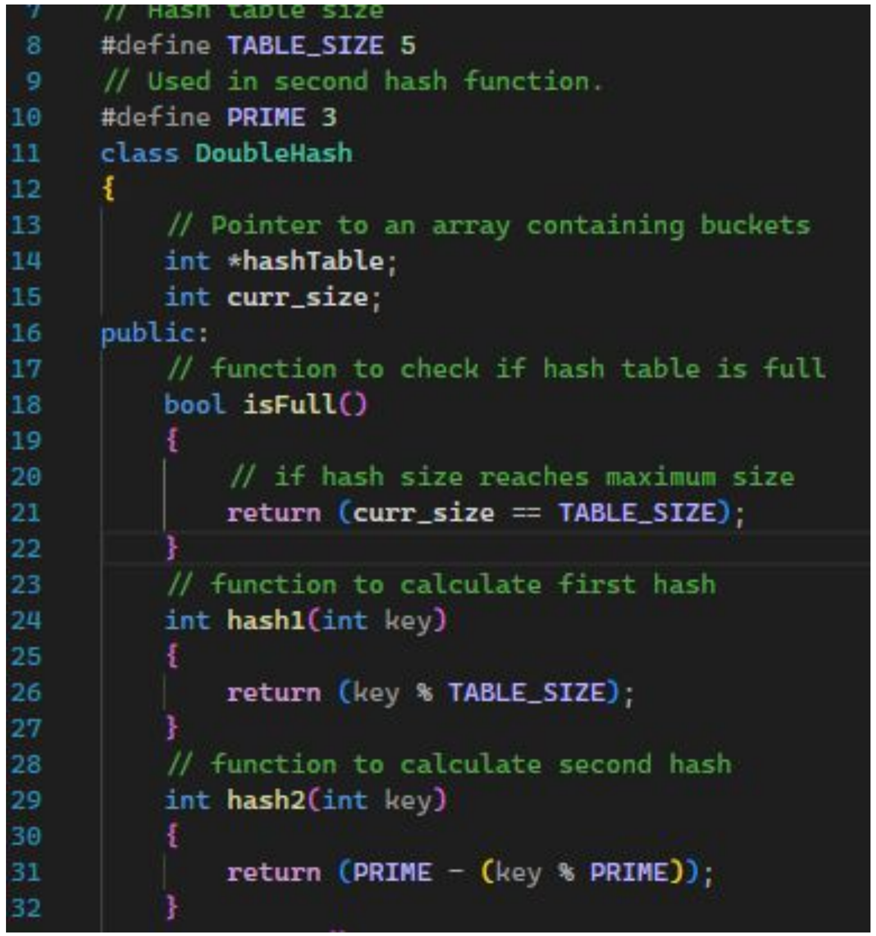
Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of effective method for resolving collisions.

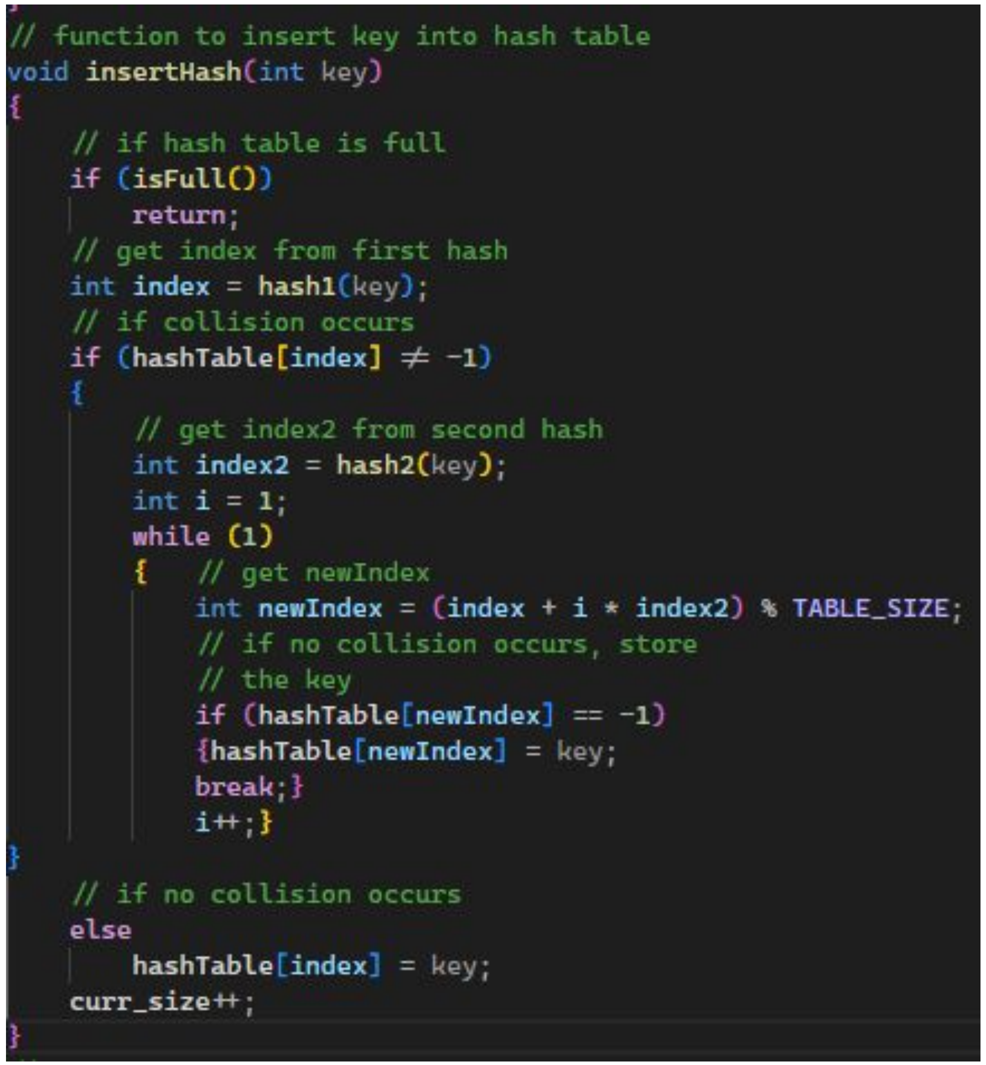
Simple Hash Function:
H1(k) = k mod length
`Default Hash function when there is no collision.`
Double Hashing Function:
H'(k,i) = [H1(k) + (i x H2(k)] % length
`This is only used when collision occurs`
Where H2(k) = prime – (k % prime)
prime can be any prime number integer or any other number relatively prime with length & prime<length

(Open Addressing Methods) Double Hashing Example:

Advantage: No Extra Space,
No Clustering
Disadvantage: Search Time O(n),

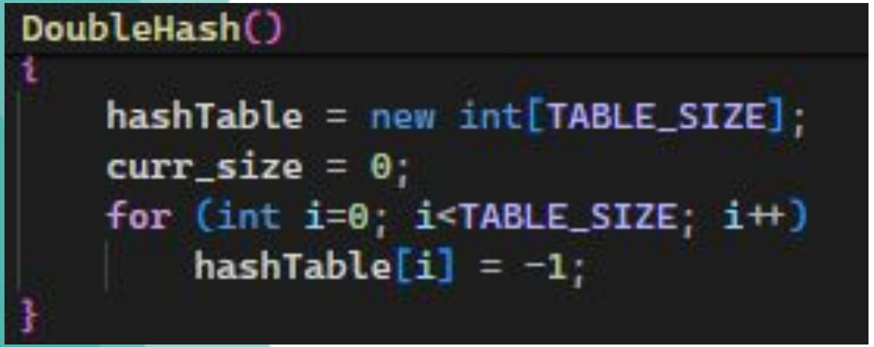


```
// Hash table size
#define TABLE_SIZE 5
// Used in second hash function.
#define PRIME 3
class DoubleHash
{
    // Pointer to an array containing buckets
    int *hashTable;
    int curr_size;
public:
    // function to check if hash table is full
    bool isFull()
    {
        // if hash size reaches maximum size
        return (curr_size == TABLE_SIZE);
    }
    // function to calculate first hash
    int hash1(int key)
    {
        return (key % TABLE_SIZE);
    }
    // function to calculate second hash
    int hash2(int key)
    {
        return (PRIME - (key % PRIME));
    }
```

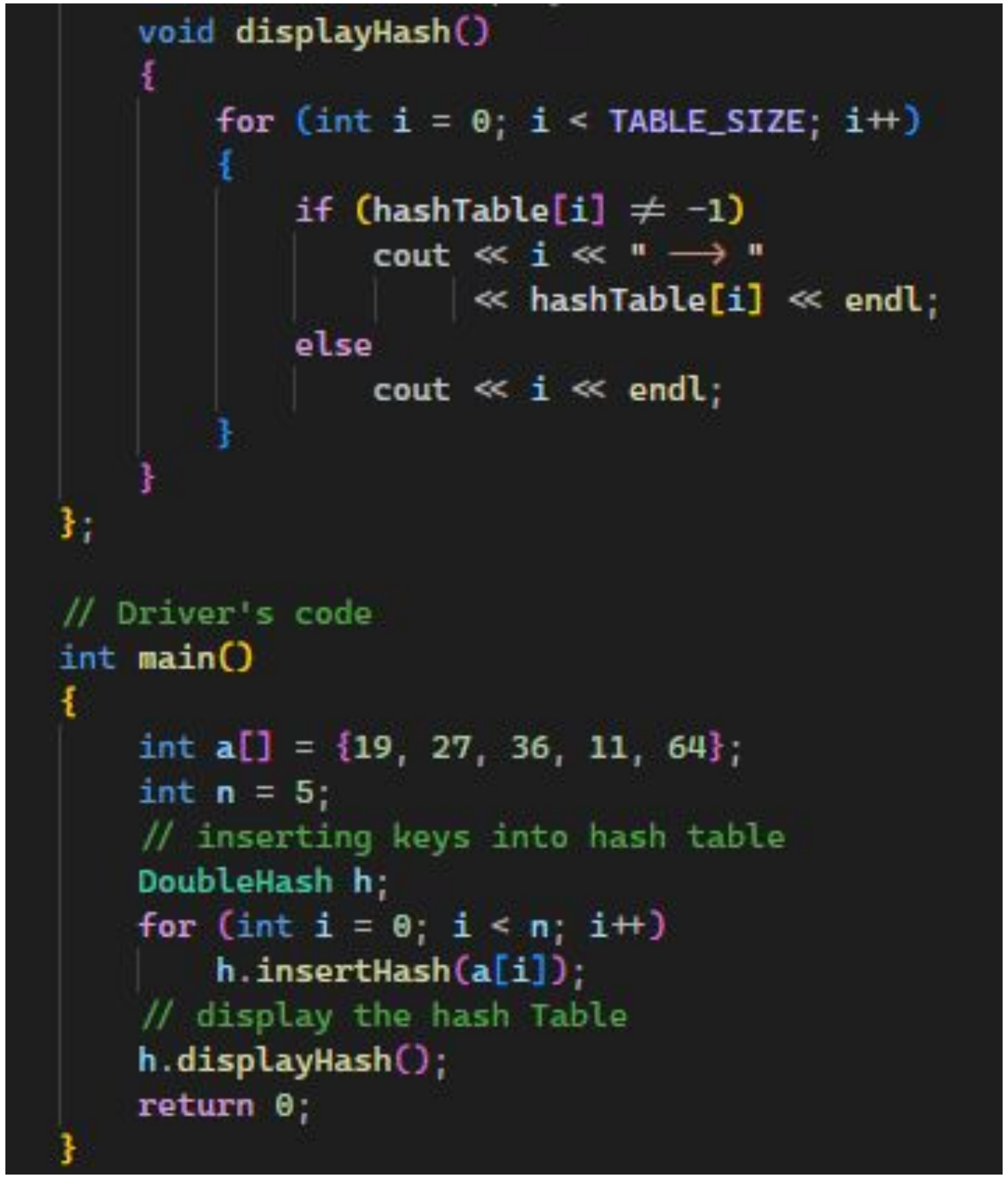


```
// function to insert key into hash table
void insertHash(int key)
{
    // if hash table is full
    if (isFull())
        return;
    // get index from first hash
    int index = hash1(key);
    // if collision occurs
    if (hashTable[index] ≠ -1)
    {
        // get index2 from second hash
        int index2 = hash2(key);
        int i = 1;
        while (1)
        {   // get newIndex
            int newIndex = (index + i * index2) % TABLE_SIZE;
            // if no collision occurs, store
            // the key
            if (hashTable[newIndex] == -1)
            {hashTable[newIndex] = key;
            break;}
            i++;}
}
    // if no collision occurs
    else
        hashTable[index] = key;
    curr_size++;
}
```

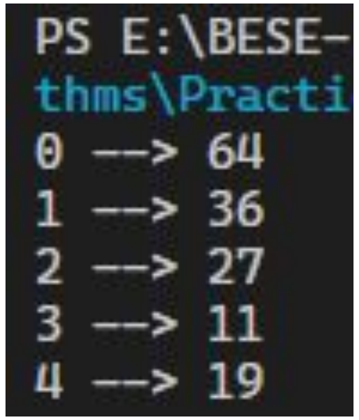
(Open Addressing Methods) Double Hashing Example:



```
DoubleHash()
{
    hashTable = new int[TABLE_SIZE];
    curr_size = 0;
    for (int i=0; i<TABLE_SIZE; i++)
        hashTable[i] = -1;
}
```



```
        void displayHash()
        {
            for (int i = 0; i < TABLE_SIZE; i++)
            {
                if (hashTable[i] != -1)
                    cout << i << " --> "
                             << hashTable[i] << endl;
                else
                    cout << i << endl;
            }
        }
};

// Driver's code
int main()
{
    int a[] = {19, 27, 36, 11, 64};
    int n = 5;
    // inserting keys into hash table
    DoubleHash h;
    for (int i = 0; i < n; i++)
        h.insertHash(a[i]);
    // display the hash Table
    h.displayHash();
    return 0;
}
```



```
PS E:\BESE-
thms\Practi
0 --> 64
1 --> 36
2 --> 27
3 --> 11
4 --> 19
```