Hash Table Assignment Interview Notes

Hash Table Overview

General Overview

A hash table is a data structure that organizes and stores data in a way that allows for efficient retrieval and storage operations. It uses a technique called hashing to map keys to corresponding values in an array. The purpose of a hash table is to provide fast access to data elements, making it particularly useful for tasks such as searching, inserting, and deleting items in constant time complexity on average. By using a hash function to calculate the index where data is stored, hash tables can achieve rapid data retrieval, making them invaluable for various applications, such as database management, caching, and implementing associative arrays or key-value pairs.

Real World Applications

Hash tables have a variety of uses in the real world. Here are some examples of what a hash table could be used for:

- Caching
- Compiler Optimization
- Compiler Symbol Tables
- Counting & Frequency Analysis
- Cryptographic Algorithms
- Databases
- DNS Resolution
- Hash-Based Password Storage
- Image Processing
- Language Processing

Hash tables are generally used in applications where data retrieval is very important. Searching through large data sets can be very time and resource intensive, because of the nature of hash tables they tend to be an optimal choice for tackling such problems.

Chaining

Chaining in hash tables is a collision resolution technique where multiple elements with different keys hash to the same index or bucket. Instead of overwriting the existing value, chaining allows these elements to be stored together in a linked list or another data structure at the same bucket. Each bucket acts as a small container, holding multiple key-value pairs. When a collision occurs, the new element is appended to the existing chain in the bucket. During retrieval or deletion, the hash table traverses the chain, locating the specific key-value pair based on the key. Chaining is a simple and effective method to handle collisions, ensuring that hash tables can efficiently store and retrieve data, even when multiple keys map to the same location in the array.

Linear Probing

Linear probing in hash tables is a collision resolution technique where elements with different keys that hash to the same index are placed in the next available (unoccupied) position in the table, effectively 'probing' forward until an empty slot is found. When a collision occurs, the algorithm checks the next position in the table and repeats the process until it finds an empty location. This method is also known as open addressing because it involves exploring alternative slots within the array to resolve collisions. Linear probing ensures that all elements are eventually stored in the primary array, without using additional data structures like linked lists. However, it may suffer from clustering, where consecutive collisions lead to more collisions, potentially degrading performance. Despite this limitation, linear probing remains a widely used technique due to its simplicity and cache-friendliness, enabling hash tables to maintain $\mathcal{O}(1)$ average case complexity for basic operations when load factors are kept low.

Quadratic Probing

Quadratic probing in hash tables is a collision resolution technique that addresses collisions by systematically probing the table using quadratic increments until an empty slot is found. When a collision occurs and the initial hashed position is occupied, the algorithm checks positions at increasingly distant intervals, following a quadratic sequence (i.e., 1, 4, 9, 16, and so on), to determine the next potential position. This method aims to disperse the elements more evenly across the table, reducing the clustering effect observed in linear probing. Quadratic probing avoids the primary clustering issue but may still encounter secondary clustering, which can affect its performance in high-load scenarios. Nevertheless, it remains a valuable alternative to linear probing for collision resolution, as it maintains $\mathcal{O}(1)$ average case complexity for basic operations when the load factor is kept relatively low, while being more cache-efficient compared to some other open addressing methods.

Double Hashing

Double hashing is a collision resolution technique used in hash tables to address key collisions by probing alternative positions based on a secondary hash function. When a collision occurs and the primary hash function places an element at an occupied index, double hashing uses a secondary hash function to calculate an increment value that determines the next probing position. The secondary hash function ensures that the increment value is non-zero and relatively prime to the table size, enabling the algorithm to explore different positions and avoid clustering. Double hashing provides good distribution of elements in the table, reducing the likelihood of collisions and achieving efficient data retrieval. It is a popular choice for collision resolution, as it maintains the $\mathcal{O}(1)$ average case complexity for basic operations, such as insertion, deletion, and retrieval, even under high load factors, making it a robust and reliable approach for hash table implementations.

Efficiency

The runtime complexity of a hash table is generally considered to be $\mathcal{O}(1)$ on average for basic operations such as insertion, deletion, and retrieval. This constant time complexity arises from the efficient mapping of keys to their corresponding values using a hash function. In the ideal scenario, each key hashes to a unique index, ensuring direct access to the desired element. However, in certain cases, hash collisions may occur when multiple keys hash to the same index, leading to a slight increase in access time. To address collisions, hash tables use techniques like chaining or open addressing, which may, in rare instances, result in worst-case scenarios with a time complexity of $\mathcal{O}(n)$. Overall, though, the average case remains constant time, making hash tables an invaluable data structure for a wide range of applications where rapid data retrieval and storage are crucial.

Operation	**Best Case** $\Omega(n)$	**Average Case** $\Theta(n)$	**Worst Case** $\mathcal{O}(n)$
Inserting	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$
Inserting (Collisions)	$\Omega(n)$	$\Theta(n)$	$\mathcal{O}(n)$
Removal	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$
Searching	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$

In general, the runtime complexity of a hash table tends to be $\mathcal{O}(1)$. This is what makes it so efficient in the operations that are used with them. In the case where a collision occurs (insertion where a bucket is already occupied) the runtime complexity of the algorithm then tends to become $\mathcal{O}(n)$.

Data Structure

Overview

To implement a hash table, several key functions, data structures, and classes are required. The fundamental components include a hash function, an array (or a dynamic array like a list or vector) to store the data, and a mechanism to handle collisions. The hash function is responsible for converting the keys into array indices, ensuring uniform distribution and efficient retrieval. The array serves as the main storage container for the key-value pairs, and its size typically depends on the expected number of elements and the desired load factor. To handle collisions, a collision resolution technique is necessary, such as chaining (using linked lists or other data structures to store multiple elements at the same index) or open addressing (finding alternative positions in the array for collided keys). Additionally, it is beneficial to encapsulate the hash table functionalities into a class, providing a clean interface for insertion, deletion, and retrieval operations, as well as methods for resizing the array when needed to maintain efficiency. By combining these components, developers can create a robust and efficient hash table implementation suitable for various real-world applications.

Structures

In order for a hash table to function correctly it requires a couple of custom structures and classes. First, we examine the custom structures that are built for use in this hash table. The two custom structures that were created for this assignment are **Hash Node** and **Hash Table**.

- **Hash Node** - This custom structure represents the contents of an individual bucket in a hash table. This structure has the following data members:
  - **deleted** - Boolean value that represents if a node has been removed from a hash table.
  - **hashcode** - Unsigned integer value that represents the hash code that is computed with a hash function.
  - **key** - String value that represents a key associated with a value.
  - **value** - String value that represents the value that is associated with a key.
- **Hash Table** - This custom structure holds the hashed data. This structure has the following data members:
  - **bucket_func** - A function that returns an unsigned integer value that represents where a node should be placed.
  - **capacity** - Unsigned integer value that represents the number of addressable buckets in a hash table.
  - **hash_func** - A function that returns an unsigned integer value that represents a hash code.
  - **size** - Unsigned integer value that represents the number of actual entries in a hash table.
  - **table** - A dynamic array of of hash node pointers.

Classes & Functions

Along with the custom data structures that have been defined previously, there are two functions (**DJB2()** and **ModuloBucketFunc()**) and the class **Hash** that are used to implement the custom structures. First, we examine the two custom functions before we examine the **Hash** class:

- **DJB2()** - This function hashes a string value and assigns an unsigned integer value in the 32-bit integer value range.
- **ModuloBucketFunc()** - This function puts keys into a specified bucket index.

The above functions provide a base for how we calculate hash codes and where we determine where the codes will be stored in the hash table. The **Hash** class provides an implementation of all these individual structures and functions to create a functioning hash table. This class' member functions are now examined:

- **Contains** - Determines if a non-deleted node is present in a hash table.
- **GetVal()** - Returns the value associated with a key in a hash table.
- **Hash()** - A constructor.
- **~Hash()** - A de-constructor.
- **InitNode()** - Creates and initializes a 'hash_node' structure that will occupy a bucket in a hash table.
- **InitTable()** - Creates and initializes a 'hash_table' structure and returns a pointer to it.
- **Load()** - Returns a load factor that describes how 'full' a hash table may be at a given time.
- **PrintTable()** - Prints the contents that are present in a given hash table.

- **Remove()** - Marks a node as deleted and removes a node from a hash table if the given node is present in a hash table.
- **Resize()** - Resizes a hash table to have a new specified capacity.
- **SetKVP()** - Creates a mapping between a given key and a value pair in a hash table.

These member functions are responsible for encapsulating the implementation of the structures that were created previously. We now will examine each of these algorithms one by one to obtain a better understanding of how these functions operate.

Algorithms

Overview

The algorithms in a hash table mainly revolve around the fundamental operations of insertion, deletion, and retrieval of key-value pairs. When inserting a new element, the hash function calculates the index for the key, and if there are no collisions, the element is directly stored at that index. In the case of a collision, the chosen collision resolution technique (such as chaining or open addressing) is used to find an alternative position for the new element. For retrieval, the hash function identifies the index for the given key, and the associated value is returned. Deletion involves locating the element in the hash table and removing it while handling any potential collisions. Additionally, some hash table implementations may incorporate techniques like dynamic resizing to maintain a low load factor and prevent performance degradation as the number of elements increases. The efficiency of these algorithms is crucial to achieve the expected $\mathcal{O}(1)$ average time complexity for basic operations, making hash tables an indispensable data structure for many practical applications.

DJB2

The fist algorithm that we will examine is the 'DJB2()' algorithm. This algorithm is responsible for creating a hash code that is to be used in a hash table.

DJB2 Algorithm

```
unsigned int DJB2(std::string key){
    unsigned int hash = 5381;
    for (size_t i=0; i < key.length(); i++) {
        char c = key[i];
        hash = ((hash << 5) + hash) + c;
    }
    return hash;
}
```

The DJB2 algorithm is a simple and widely used hash function designed to efficiently convert a given input string (key) into a corresponding unsigned integer hash value. It initializes the hash to 5381 and iterates through each character of the input string. For each character, it performs bitwise left shift (<<) on the current hash by 5 positions, then adds the original hash value to the result, and finally adds the ASCII value of the current character to the hash. This process continues for all characters in the input string, effectively combining their contributions to create the final hash value. The algorithm's simplicity and effectiveness in distributing hash values make it popular for various applications where a fast and reasonably distributed hash function is required, such as in hash tables, caching, and hashing-based data structures.

As mentioned previously, the **DJB2** algorithm is responsible for calculating the hash code of a given input string. This algorithm is widely used in a variety of contexts for multiple different data structures. It can be thought of the heart of all the algorithms that are required to create a hash table.

ModuloBucketFunc

The next algorithm that we will examine is the 'ModuloBucketFunc()' algorithm. This algorithm is responsible for calculating the index of the bucket for the hash node that is being inserted into a hash table.

ModuloBucketFunc Algorithm

```
unsigned int ModuloBucketFunc(unsigned int hashcode, unsigned int cap){
    unsigned int b = hashcode % cap;
    return b;
}
```

The ModuloBucketFunc algorithm is responsible for calculating the index of a bucket in a hash table. Whether this algorithm is used for inserting or searching, it determines the index of a specific hash code. This bucket index is found by taking the specific hash code of a key and calculating the modulus of the hash code with the capacity of the table.

The **ModuloBucketFunc** algorithm is responsible for calculating the bucket index of a current hash code. This algorithm is critical for how we perform operations with the hash table.

Contains

The next algorithm that we will examine is the 'Contains()' algorithm. This algorithm searches through a hash table and determines if a specific value is present in a hash table.

Contains Algorithm

```
bool Hash::Contains(shared_ptr<hash_table> tbl, std::string key){
    // Calculate the hash code and bucket index of the given key
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    // Create a dummy variable to indicate the buckets that have been probed in the table
    unsigned int buckets_probed = 0;
    // Traverse the table until buckets probed reaches the size of the table
    while (buckets_probed < tbl->size) {
        // If the current nodes key data member matches that of the input parameter "key"
        // return true
        if (tbl->table->at(bucket_idx)->key == key) {
            return true;
        }
        // Increment the bucket index and buckets probed values
        bucket_idx = (bucket_idx + 1) % tbl->size;
        buckets_probed++;
    }
    // Return false if nothing was found
    return false;
}
```

The function above begins by calculating the hash code and bucket index of the given key that is being searched for in the hash table. We then create an integer value that is designated to keep track of the number of buckets that have been 'probed'. We then begin traversing the hash table until the number of buckets that have been probed is equal to that of the tables size. In each iteration, we first check to see if they is present in the current bucket that we are examining, and it it is we return true. If the key is not present in the current bucket we Increment the bucket index and buckets probed variable and continue on to the next bucket. This process will repeat until either we find the bucket that contains the key we are searching for or until the number of buckets that we have probed is equal to that of the hash tables size. If the key is not found in the hash table then we return false.

Similar to other data structures that we have worked with in this course, hash tables are traversed with the use of a while loop and a break condition inside the while loop. In the case of the **Contains** function we will continue searching in the hash table until we either find the key in the hash table or until we determine that we have looked in all buckets.

GetVal

The next algorithm that we will examine is the 'GetVal()' algorithm. This algorithm returns the value of a key if that key is present in a hash table. If the key is not present in a hash table then the algorithm will return an empty string.

GetVal Algorithm

```
std::string Hash::GetVal(shared_ptr<hash_table> tbl, std::string key){
    // Calculate the hash code and bucket index of the given key
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    // Create a dummy variable to indicate the buckets that have been probed in the table
    unsigned int buckets_probed = 0;
    // Traverse the table until buckets probed reaches the size of the table
    while (buckets_probed < tbl->size) {
        // If the current nodes key data member matches that of the input parameter "key",
        // return its value
        if (tbl->table->at(bucket_idx)->key == key) {
            return tbl->table->at(bucket_idx)->value;
        }
        // Increment the bucket index and buckets probed values
        bucket_idx = (bucket_idx + 1) % tbl->size;
```

```
        buckets_probed++;
    }
    // Return null if nothing was found
    return "";
}
```

The GetVal algorithm functions in a similar way as the Contains algorithm. Similar to that of the Contains algorithm, we begin by calculating the hash code and bucket index of the key. We then create a dummy variable that will keep track of the number of buckets that we have probed while traversing the hash table. Once we begin traversing the hash table, we check if the current bucket that we are examining contains the key that we are searching for. If the current bucket does contain the key that we are looking for, then we return that keys value and stop traversing the hash table. If the current bucket does not contain the key that we are searching for, then we increment both the bucket index and the buckets probed value to move on to the next bucket. If the key is not found in the hash table then we return an empty string.

Much like the **Contains** algorithm, **GetVal** traverses the hash table to determine if a key is present in the hash table. The only difference between the two algorithms is that if **GetVal** determines that the key is present in the hash table then it returns the value of the key instead of just a boolean value indicating if a key is present in the hash table. **GetVal** will also return an empty string value if the key does not exist in the hash table.

Hash & ∼Hash

The constructor and de-constructor of this class was not defined. Because we are using smart pointers for our objects that are dynamic, we do not need to manually delete our pointers. We could initialize some of our data members of both the node and table but it was found that it wasn't necessary.

InitNode

The next algorithm that we will examine is the 'InitNode' algorithm. This algorithm initializes a hash node object to default values. After the object has been initialized to its default values, we return this node as the algorithms output.

InitNode Algorithm

```
shared_ptr<hash_node> Hash::InitNode(std::string key, unsigned int hashcode, std::string val){
    // Node to be returned
    shared_ptr<hash_node> ret(new hash_node);
    // Assign input parameters to hash_node data members
    ret->deleted = false;
    ret->key = key;
    ret->hashcode = hashcode;
    ret->value = val;
    // Return node
    return ret;
}
```

The InitNode algorithm is responsible for creating a hash node object that is assigned to default values. We first create a new hash node object called 'ret' and then we assign the data members of the object to the input parameters that are fed in the algorithm. We also flag the boolean value 'deleted' to be false and once these values have been assigned we return the node as the algorithms output.

Similar to other data structures that we have worked with, **InitNode** is responsible for accurately assigning data members to that of input parameters. Once these data members have been assigned we return the object so that it can be used in other contexts.

InitTable

The next algorithm that we will examine is the 'InitTable' algorithm. This algorithm is similar to that of 'InitNode' algorithm. But instead of initializing a node, like in 'InitNode', we are initializing a hash table with 'InitTable'. After we initialize the table to its default values, we return the table as the algorithms output.

InitTable Algorithm

```
shared_ptr<hash_table> Hash::InitTable(unsigned int cap){
    // Table to be returned
    shared_ptr<hash_table> ret(new hash_table);
    // Assign input parameters to the data members of the the hash table "ret"
    ret->capacity = cap;
    ret->size = 0;
    ret->occupied = 0;
    ret->table = shared_ptr<htable>(new htable(cap));
    for (int i = 0; i < ret->table->size(); i++) {
        ret->table->at(i) = nullptr;
    }
    ret->hash_func = &DJB2;
    ret->bucket_func = &ModuloBucketFunc;
    // Return the modified hash table
    return ret;
}
```

The InitTable algorithm is doing the same as the InitNode algorithm but instead of working a hash node we are working with a hash table. In this algorithm, we assign the default values of a hash table to the input parameters that are fed to this algorithm. We also initialize the functions 'hash_func' and 'bucket_func' to that of **DJB2** and **ModuloBucketFunc** respectively. All of the pointers inside the 'table' data member are initially set to null and the 'size' and 'occupied' members are initially set to zero. Once these data members have been assigned we return the table as the output of the algorithm.

Both **InitNode** and **InitTable** are very similar in how they operate, the distinctions between the two are what the algorithms return upon finishing and what the data members of each object are being assigned to.

Load

The next algorithm that we will examine is the 'Load' algorithm. This algorithm is responsible for calculating how 'full' a hash table is.

Load Algorithm

```
float Hash::Load(shared_ptr<hash_table> tbl){
    // Cast the integer values to floats
    float size = static_cast<float>(tbl->size);
    float cap = static_cast<float>(tbl->capacity);
    // Calculate the load factor
    float load = size / cap;
    // Return the load factor
    return load;
}
```

The Load algorithm is calculating how full a hash table is. This is done by calculating the ratio of the size to the capacity of the hash table in question. In order to calculate this, we have to statically cast the 'size' and 'capacity' data members to floats so that the ratio of the two can be calculated. Once these variables have been statically casted the ratio, the algorithm returns the ratio.

The **Load** algorithm calculates how 'full' a hash table is by calculating the ratio of data members for the hash table.

PrintTable

The next algorithm that we will examine is the 'PrintTable' algorithm. This algorithm prints the contents of a hash table.

PrintTable Algorithm

```
void Hash::PrintTable(shared_ptr<hash_table> tbl){
    cout << "Hashtable:" << endl;
```

```
    cout << "  capacity: " << tbl->capacity << endl;
    cout << "  size:     " << tbl->size << endl;
    cout << "  occupied: " << tbl->occupied << endl;
    cout << "  load:     " << Load(tbl) << endl;
    if (tbl->capacity < 130) {
        for (unsigned int i=0; i < tbl->capacity; i++) {
            cout << "[" << i << "]    ";
            if (!tbl->table->at(i)) {
                cout << "<empty>" << endl;
            } else if (tbl->table->at(i)->deleted) {
                cout << "<deleted>" << endl;
            } else {
                cout << "\"" << tbl->table->at(i)->key << "\" = \""
                << tbl->table->at(i)->value << "\"" << endl;
            }
        }
    } else {
        cout << "<hashtable too big to print out>" << endl;
    }
}
```

The PrintTable algorithm is a function designed to display the contents and statistics of a given hash table. It takes a shared pointer to the hash table as input and outputs details such as the capacity, current size, number of occupied slots, and the load factor (ratio of occupied slots to capacity). The function then checks if the hash table's capacity is less than 130; if so, it proceeds to print each element in the table. For each index, it displays either "<empty>" if the slot is empty, "<deleted>" if the slot was marked as deleted, or the key-value pair stored at that index. If the capacity is greater than or equal to 130, the function prints a message indicating that the hash table is too big to be printed out. This algorithm is useful for developers to inspect and understand the state of the hash table, aiding in debugging and performance analysis.

Similar to other data structures, **PrintTable** prints the contents of the hash table to provide a debugging technique for other algorithms implementation.

Remove

The next algorithm that we will examine is the 'Remove' algorithm. This algorithm is responsible for removing a node from a hash table.

Remove Algorithm

```
bool Hash::Remove(shared_ptr<hash_table> tbl, std::string key){
    // Calculate the hash code and bucket index of the given key
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    // Create a dummy variable to indicate the buckets that have been probed in the table
    unsigned int buckets_probed = 0;
    // Traverse the table until buckets probed reaches the size of the table
    while (buckets_probed < tbl->size) {
        // If the current nodes key data member matches that of the input parameter "key",
        // remove the node from the table and return true
        if (tbl->table->at(bucket_idx) != nullptr && tbl->table->at(bucket_idx)->key == key) {
            tbl->table->at(bucket_idx) == nullptr;
            tbl->table->at(bucket_idx)->deleted = true;
            tbl->size--;
            return true;
        }
        // Increment the bucket index and buckets probed values
        bucket_idx = (bucket_idx + 1) % tbl->size;
        buckets_probed++;
    }
    // Return false if nothing was found
    return false;
```

```
}
```

The Remove algorithm is a function designed to remove a key-value pair from the given hash table. It calculates the hash code and the bucket index for the input 'key' using the DJB2 hash function and the provided bucket function. The function then traverses the table, probing the buckets to find the key to be removed. If the key is found in a non-null bucket, it marks the bucket as deleted and removes the node from the table, decreasing the table's size by one, before returning true to indicate a successful removal. If the key is not found after probing all buckets, the function returns false to signify that the key was not present in the hash table. This algorithm efficiently handles collisions and ensures the removal operation has a time complexity of $\mathcal{O}(1)$ on average, making it suitable for hash table implementations requiring a delete operation.

Similar to other algorithms previously mentioned, **Remove** first traverses the table to determine if the key is present in the table.

Resize

The next algorithm that we will examine is the 'Resize' algorithm. This algorithm is responsible for resizing a hash table

Resize Algorithm

```
void Hash::Resize(shared_ptr<hash_table>& tbl, unsigned int new_capacity){
    // Create a new table that is going to be used to update the old table
    shared_ptr<hash_table> new_table = InitTable(new_capacity);
    // Copy the size and occupied data members of the old table to the new one
    new_table->size = tbl->size;
    new_table->occupied = tbl->occupied;
    // Copy the hash nodes of the old table to the new table
    for (int i = 0; i < tbl->table->size(); i++) {
        if (tbl->table->at(i) != nullptr) {
            // Calculate the hash code and bucket index of the current node for the new table
            unsigned int hash = DJB2(tbl->table->at(i)->key);
            unsigned int bucket_idx = new_table->bucket_func(hash, new_table->capacity);
            new_table->table->at(bucket_idx) = tbl->table->at(i);
        }
    }
    // Update the old table to be the new table
    tbl = new_table;
}
```

The Resize algorithm is a function designed to resize the given hash table to a new specified capacity. It first creates a new hash table with the desired capacity using the "InitTable" function. The size and occupied data members of the old table are then copied to the new one to maintain consistency. Next, it iterates through the nodes of the old table and rehashes them to calculate their new bucket index in the resized table. The nodes are then copied to the corresponding positions in the new table. Finally, the old table is updated to point to the new table, effectively resizing and rehashing the hash table. This algorithm efficiently adjusts the capacity of the hash table, enabling it to handle changes in the number of elements while maintaining an average time complexity of $\mathcal{O}(n)$ for resizing, where n is the number of elements in the hash table.

The **Resize** algorithm is responsible for resizing a hash table to a different capacity. This is done by creating a copy of the original table and then creating a new table and copying the contents of that table to the new table and then returning it after the algorithms execution.

SetKVP

The next algorithm that we will examine is the 'SetKVP' algorithm. This algorithm is responsible for assigning key value pairs of a specific hash node and then adding the node to a hash table.

SetKVP Algorithm

```
bool Hash::SetKVP(shared_ptr<hash_table> tbl, std::string key, std::string value){
    // Determine the hash value and bucket index for node
```

```
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    // Create a value that indicates if a bucket has been probed to be inserted
    unsigned int buckets_probed = 0;
    // Create a hash node that is to be inserted into the hash table
    shared_ptr<hash_node> node = InitNode(key, hash, value);
    // Repeat process until all possible buckets have been probed
    while (buckets_probed < tbl->table->size()) {
        // Check if the current bucket is empty, if it is, insert the node at that index
        // and update data members
        if (tbl->table->at(bucket_idx) == nullptr) {
            tbl->table->at(bucket_idx) = node;
            tbl->occupied++;
            tbl->size++;
            return true;
        }
        // Check if the current bucket's key is equal to that of the input parameters key,
        // if so, update value
            else if (tbl->table->at(bucket_idx)->key == key) {
            tbl->table->at(bucket_idx)->value = value;
            return true;
        }
        // Increment the bucket index and buckets probed values
        bucket_idx = (bucket_idx + 1) % tbl->table->size();
        buckets_probed++;
    }
    return false;
}
```

The SetKVP algorithm is a function designed to insert or update a key-value pair (KVP) in the given hash table. It first calculates the hash code and bucket index for the input key using the DJB2 hash function and the provided bucket function. Then, it creates a new hash node containing the key-value pair to be inserted or updated. The algorithm proceeds to probe the buckets in the table until all possible buckets have been checked. If an empty bucket is found, the new node is inserted at that index, and the table's occupied and size data members are updated accordingly. If a non-empty bucket with a matching key is found, the value is updated, and the function returns true to indicate a successful update. If all buckets are probed, and no suitable empty bucket or matching key is found, the function returns false, indicating that the insertion or update operation was not successful. This algorithm ensures efficient and collision-handling insertions and updates in the hash table, with an average time complexity of $\mathcal{O}(1)$ and a worst-case time complexity of $\mathcal{O}(n)$, where n is the number of elements in the hash table.

SetKVP adequately assigns a key value pair for a hash node that is to be inserted into a hash table. This algorithm handles collisions by updating the key's value instead of creating something like a linked list where multiple elements can be stored at a single index. This algorithm is rather efficient as it searches through a hash table for the correct index to insert a hash node and usually does so rather quickly.