4-2 Code Reflection

Armon Wilson

SNHU

CS300 – Data Structures & Algorithms

**HashTable();**

**Purpose:**

The primary purpose of the HashTable() constructor is to initialize the hash table data structure, including size.

**Techniques Implemented:**

To achieve its purpose, the constructor utilizes a dynamic array to store bid data. Each element within this array represents a bucket in the hash table. The constructor employs a separate chaining mechanism for handling collisions, whereby each bucket can store multiple bids in a linked list fashion. The Node struct houses both the bid data and a pointer to the next node in the chain.

**Challenges & Solutions:**

No challenges during implementation. Tests went smoothly.

**Pseudo-Code:**

FUNCTION HashTable()

tableSize = DEFAULT\_SIZE

nodes.resize(tableSize)

ENDFUNCTION

FUNCTION HashTable(size)

this.tableSize = size

nodes.resize(tableSize)

ENDFUNCTION

**virtual ~HashTable();**

**Purpose:**

The virtual ~HashTable() destructor is designed to release the memory occupied by the hash table when it's no longer needed.

**Techniques Implemented:**

To achieve its purpose, the destructor iterates through all buckets within the hash table. For each bucket, it traverses the linked list of nodes, systematically deallocating the memory used by each node. This traversal and deallocation process ensures that all dynamically allocated memory associated with the hash table is freed.

**Challenges & Solutions**:

No challenges during implementation. Tests went smoothly.

**Pseudo-Code:**

FUNCTION ~HashTable()

FOR EACH bucketIndex IN range(0, tableSize)

current = address of nodes[bucketIndex]

WHILE current is not null

temp = current

current = current->next

delete temp

ENDWHILE

ENDFOR

ENDFUNCTION

**Hash;**

**Purpose:**

The core purpose of the hash function is to transform a given integer key into a corresponding index within the hash table's array of buckets. This index determines where the associated data will be stored or retrieved.

**Techniques Implemented:**

Hash function employs a straightforward modulo operation. The key is divided by the table size, and the remainder is used as the index.

**Challenges & Solutions:**

No challenges during implementation. Tests went smoothly.

**Pseudo-Code:**

FUNCTION hash(key)

RETURN key modulo tableSize

ENDFUNCTION

**void Insert(Bid bid);**

**Purpose:**

The purpose of the Insert function is to add a new bid to the hash table. It calculates a hash key for the bid, then inserts it either at the calculated position if it's empty or unused, or at the end of the linked list in that position if there's a collision.

**Techniques Implemented:**

* Hashing: The function uses a hash function to compute a numerical key based on the bid ID.
* Separate Chaining: Collisions are resolved using separate chaining, where each bucket can store a linked list of nodes.

**Challenges & Solutions:**

While working on the Insert function, several challenges arose. One major issue was ensuring correct node insertion in the case of collisions. Implementing a proper traversal mechanism through the linked list within a bucket was crucial to append the new node at the end. Additionally, handling empty buckets or those containing previously removed nodes (marked by UINT\_MAX) required special logic to ensure proper node placement and memory management.

**Pseudo-Code:**

FUNCTION Insert(bid)

CREATE new Node containing bid data

CALCULATE hash key based on bid ID

GET reference to the node at the calculated hash key

IF the retrieved node is empty

COPY new node data to the retrieved node and DELETE new node

ELSE IF the retrieved node is unused

SET the retrieved node's key to the new node's key

COPY new node data to the retrieved node

SET the retrieved node's next pointer to null

DELETE new node

ELSE

WHILE the next pointer of the retrieved node is not null

MOVE to the next node in the linked list

ENDWHILE

SET the next pointer of the retrieved node to the new node

ENDIF

ENDFUNCTION

**void PrintAll();**

**Purpose:**

The PrintAll function is designed to display all the bids stored within the hash table to the console.

**Techniques Implemented:**

The function employs a nested loop structure. The outer loop traverses the hash table's underlying vector, which represents the buckets. Within each bucket, an inner loop iterates through the linked list (if any) associated with that bucket. A null check (current != nullptr) ensures the traversal halts when the end of a linked list is reached. The key != UINT\_MAX condition is used to filter out unused or deleted nodes.

**Challenges and Solutions:**

A challenge lies in the structure of the hash table, which uses separate chaining for collision resolution. This means multiple bids might hash to the same bucket, requiring traversal of a linked list within that bucket. The nested loop structure effectively addresses this challenge, ensuring that all bids are accessed and printed, regardless of collisions.

**Pseudo-Code:**

FUNCTION PrintAll()

FOR each index i from 0 to tableSize - 1 DO

current ← address of node at index i

IF current is not null AND current's key is not UINT\_MAX THEN

Output current's key, bid ID, title, and fund

current ← current's next node

WHILE current is not null DO

Output current's key, bid ID, title, amount, and fund

current ← current's next node

ENDWHILE

ENDIF

ENDFOR

ENDFUNCTION

**void Remove(string bidId);**

**Purpose:**

 The Remove function is designed to eliminate a bid with a specific bidId from the hash table.

**Techniques Implemented**:

To accomplish its task, the Remove function employs a combination of hashing and linked list manipulation techniques. It calculates the hash of the bidId to identify the relevant bucket in the hash table. Subsequently, it traverses the linked list within that bucket, comparing the bidId of each node against the target bidId. When a match is found, the function deftly adjusts the next pointers of the surrounding nodes to remove the target bid from the list while ensuring the integrity of the remaining linked structure.

**Challenges & Solutions:**

 A primary challenge in implementing the Remove function is correctly handling the scenario where the bid to be deleted is the head of the linked list in its bucket. If this case is not handled explicitly, it could lead to a dangling pointer, causing instability in the program. The solution to this challenge involves a conditional check within the traversal loop, which updates the head of the list appropriately when the target bid is the first node.

**Pseudo-Code:**

FUNCTION Remove(bidId)

key = hash(bidId) // Convert bidId to an integer key

current = &nodes[key] // Get reference to the head node in the bucket

previous = nullptr // Initialize previous pointer

WHILE current is not null DO

IF current->bid.bidId matches bidId THEN // Bid found

IF previous is not null THEN

previous->next ← current->next // Remove from middle/end

ELSE

nodes[key] = \*current->next // Remove head node

DELETE current // Free memory

RETURN // Exit function

ENDIF

previous = current // Move to the next node

current = current->next

ENDWHILE

ENDFUNCTION

**Bid Search(string bidId);**

**Purpose:**

 The Search function aims to locate a specific bid within the hash table using its unique identifier (bidId).

**Techniques Implemented**:

* Hashing: Employs the hash function to transform the input bidId into a numerical key, which determines the bucket's index within the hash table.
* Linked List Traversal: Systematically examines each node in the linked list associated with the calculated bucket index.
* Key-Value Matching: Compares the bidId of each node against the target bidId for potential matches.

**Challenges & Solutions:**

The main challenge was handling the potential for null pointer dereferences when a bid was not found or a bucket was empty. The solution involved carefully restructuring the code, removing unnecessary checks, and ensuring null checks were in place before accessing node data.

**Pseudo-Code:**

FUNCTION Search(bidId)

key = hash(bidId)

current = nodes[key]

// Check if the bidId exists in the first node

IF current is not null AND current->bid.bidId equals bidId THEN

RETURN current->bid

ENDIF

// Check if the key is not found or the node is unused

IF current is null OR current->key equals UINT\_MAX THEN

RETURN bid // default bid (not found)

ENDIF

WHILE current is not null DO

IF current->bid.bidId equals bidId THEN

RETURN current->bid

ENDIF

current = current->next

ENDWHILE

RETURN bid // default bid (not found)

ENDFUNCTION