Edwin Jones

CS-300

Hash Tables Code Reflection

The purpose of this code is to utilize hash tables to store data imported from a CSV file in a data structure that excels in faster searching, deletion, and insertions of data versus other data structures like vectors or linked lists. This solves the problem of potentially needing to iterate through every single element for similar functionality. This is because the data structure is just an array or vector with each element being composed of the same or different data structure like arrays, vectors, or linked lists as with this assignment for example. The benefit is that if the data used to generate the key that is assigned to the bucket (the index) is known, the index of the data structure can be found with O(1) time complexity and all that’s left to do is find the element with data matching the search query. The absolute worst case time complexity is O(n) since a hash table could potentially have 1 key containing all data if the hash function isn’t good or right for the application.

The biggest challenge while trying to use a hash table was implementing the Remove() function. The biggest I encountered was when trying to delete a node given a bidId. I initially tried:

nodes.erase(nodes.begin() + currentKey);

but this just erases every node assigned to the key generated by bidId regardless of if the node has a matching bidId and it also doesn’t properly deallocate memory allocated with “new”. I then tried just emptying all the node’s bid info with:

if (nodes.at(currentKey).bid.bidId == bidId) {

nodes.at(currentKey).bid.bidId.clear();

nodes.at(currentKey).bid.title.clear();

nodes.at(currentKey).bid.fund.clear();

nodes.at(currentKey).bid.amount = 0.0;

} else {...}

While this does strip that node and only that node of information, it still uses memory as it still has nodes pointing to it and it points to a node, and it is still technically storing data. Rather than implementing a condition to overwrite empty nodes by using the Insert() function, then relying on any new data to be stored to meet the condition which would make that allocated memory useful, I opted to just delete the node’s data and free the memory with the following lines:

if (nodes.at(currentKey).bid.bidId == bidId) {

Node \*tempNode = nodes.at(currentKey).next;

nodes.at(currentKey) = \*tempNode;

delete tempNode;

return;

}

*// first node does not have matching bidId, check the rest of the nodes*

else {

Node \*nextNode = nodes.at(currentKey).next;

Node \*prevNode = &nodes.at(currentKey);

while (nextNode != nullptr) {

if (nextNode->bid.bidId == bidId) {

prevNode->next = nextNode->next;

delete nextNode;

return;

}

prevNode = nextNode;

nextNode = nextNode->next;

}

}

For this application, memory leaks/bloat aren’t really a big concern since the amount of data being written/read is relatively low once the hash table is populated, I still wanted to practice proper memory management and avoid any issues if given more data that require more frequent creation and/or deletion.

// bid information to store

struct Bid {

string bidId

string title

string fund

double amount

Bid()

amount will default to 0.0

}

// blueprint for populating the hash table with chaining

class HashTable {

struct Node:

initialize bid of type Bid

unsigned int key

Node \*next

Node() {

Bid bid

Set key to max integer

Set next to nullptr

}

Node(Bid aBid) : Node() {

Set bid to aBid

}

Node(Bid aBid, unsigned int aKey) : Node(aBid) {

Set key to aKey

}

vector<Node> nodes

set unsigned int tableSize to an immutable value

}

HashTable::HashTable() {

resize nodes vector to tableSize

}

HashTable::HashTable(unsigned int size) {

set tableSize equal to size

resize nodes vector to tableSize

}

HashTable::~HashTable(){

for a value i=0 increment it by 1 until i<tableSize

initialize Node pointer current = next node

while current node != nullptr

initialize Node point toDelete = next node

delete toDelete node

}

unsigned int HashTable::hash(int key){

return key modulo tableSize

}

// determines where to put bid information

void HashTable::Insert(Bid bid) {

set currentKey = hashed bidId

if the node at currentKey has a key = max integer

set node at currentKey = a new node created using bid and currentKey params

else:

initialize nextNode pointer = the address of nodes at currentKey

while the next node != nullptr

set nextNode = next node

set the node after nextNode = a new node created using bid and currentKey params

}

// prints nodes in ascending key order and order added to bucket

void HashTable::PrintAll() {

for each node in nodes

print that nodes bid information

initialize next point = the next node

while next != nullptr

print that nodes bid information

set next = next node

}

// removes node with matching bidId

void HashTable::Remove(string bidId) {

set currentKey = hashed bidId

if the nodes bidId = given bidId

delete that node

else

traverse all nodes with currentKey

if the next nodes bidId = given bidId

delete that node

}

// returns node with matching bidId

void HashTable::Search(bidId) {

declare an empty bid

if the nodes bidId = given bidId

return that nodes bid information

else

traverse all nodes with currentKey

if the next nodes bidId = given bidId

return that nodes bid information

}