DESIGN DOCUMENT: Project 2

NAME: Naveela Sookoo
STUDENT ID: nnsookoo

The assigned task was to create a C++ implementation for hash table data structure using two different techniques: (i) open addressing using linear probing and (ii) separate chaining where the chains are unordered.

Open Addressing using Linear Probing

Two classes were created – ***class HashNode*** which stores the properties of each entry, that is the key, occupied and used value and ***class HashTable*** which stores the properties of a hash table and provides services such as hash index calculation, testing capacity of hash table, defining size of hash table, inserting a key, searching, returning the index of a key (where the key is located) and removing a key.

class HashNode:

This class contained only public members as these parameters is needed to perform operations within the ***class HashTable***. Its members are ***int key*** (to store the number defined by the user), ***int occupied*** (to denote if slot in the hash table has a key; can only have values 1 and -1) and ***int used*** (to store if slot in the hash table has been used before; can only have values 1 and -1). Initially, the decision was made to have only ***int key*** and ***int occupied***; however, an ***int used*** was added as it helped the search function in achieving a time complexity of almost constant.

The member functions include:

1. Constructors:
   a. To create an empty node: ***HashNode();***
      The key is set to 0, occupied status is set to -1 (meaning that the slot is not occupied) and used status is set to -1 (meaning that the slot has not been used before)
   b. To create a defined node: ***HashNode(const int& key, const int& occupied);***
      The key is set to the key defined and occupied status is set to the defined occupied status. The used status is set to 1 (meaning that the slot is now in use).
2. Destructor: ***~HashNode();***
   The key is set to 0 and occupied status is set to -1. The used status is still set to 1 here.

class HashTable:

This class contained both member objects (private) and member functions (public). Its private members included a ***vector <HashNode*> hash_table*** which is used as the primary hash table structure (its individual 'slots' represent the 'cells' of the hash table), an ***int max_size*** which is the maximum capacity of the hash table, and an ***int cur_size*** – the current size of the hash table which is incremented when a key is added and decremented when a key is removed.

The member functions include:

1. Constructor: ***HashTable();*** - This creates an "empty" hash table (current size set to 0).
2. Destructor: ***~HashTable();*** - This calls on destroy() which is explained next.
3. ***void destroy();*** - This function deletes the elements of the hash table. Firstly, this function checks if the hash_table vector is empty. If it is not empty, the procedure is carried out to delete and clear all elements from the vector as well as set the current size to 0. This function is extremely important in the ***init*** function. If the ***init*** is called twice within a function, it is necessary that the first hash table be deleted in order for the second, new hash table to be created. This function does not have a return type.

4. ***int HashFunction(const int val);*** - This function calculates the index value in which the key should be placed using the hash function "key mod max_size". This function has a time complexity of constant and it returns an integer.
5. ***bool isFull();*** - This function returns a true if the hash table is full. It checks if the maximum capacity of the hash table is equal to its current capacity. Its time complexity is constant.
6. ***bool init(const int m);*** - This function returns bool depending on if it was successful defining the size of the hash table. It uses the push_back() function found in the vector STL library to allocate empty hash table nodes to the hash table. The average runtime for this is linear.
7. ***bool insert(const int k)*** – This function returns bool depending on if it was successful in inserting a key. Firstly, it checks if the hash table is full; if so, it returns false (constant average runtime). Then it checks if the key is already found in the hash table using the search function (constant time complexity). If these errors do not occur, the hash index is processed using the hash function and an empty slot is located. When an empty slot is located, the key, along with an occupied status and used status of 1 is inserted into that slot of the hash table. The expected average runtime is constant as it is expected that the key will be entered at the index calculated by the hash function.
8. ***bool search(const int k);*** - This function returns bool depending on if the key was found in the hash table. It calculates the hash index using the hash function. The function then proceeds to search the hash table for the key. If the occupied flag is 1 and the key at the index equals the key given, it returns true. However, if an unused slot (status flag of -1) is come across, it returns false. This check is extremely helpful if the key happens to be found after a previously deleted key. The expected average runtime is, therefore, constant.
9. ***int returnIndex(const int k);*** - This function returns the index (int) in which the key is found. This function is primarily used in outputting the index in the driver file.
10. ***bool remove(const int k);*** - This functions returns bool depending on if the function was successful in removing the key in the hash table. It firstly checks if the key is not found in the hash table. If so, it returns false. The hash index is calculated using the hash function. The function then proceeds to search for the key and remove when found. The expected average runtime for this function is constant as it is expected that the key will be found at the index calculated by the hash function.

Separate Chaining (Unordered Chains)

Two classes were created – ***class HashNode*** which stores the properties of each entry, that is the key and a pointer to the next HashNode and ***class HashTable*** which stores the properties of a hash table and provides services such as hash index calculation, testing capacity of hash table, defining size of hash table, inserting a key, searching, returning the index of a key (where the key is located) and removing a key.

class HashNode:

This class contained only public members as these parameters is needed to perform operations within the ***class HashTable***. Its members are ***int key*** (to store the number defined by the user) and ***HashNode* next***.

The member functions include:

Constructors:

a. To create an empty node: ***HashNode();***
   The key is set to 0 and the next pointer is set to NULL.
b. To create a defined node: ***HashNode(const int& key);***

The key is set to the key defined and the next pointer is set to NULL.

Destructor: ***~HashNode();***
The key is set to 0 and the next pointer is set to NULL.

class HashTable:

This class contained both member objects (private) and member functions (public). Its private members included a ***HashNode **hash_table*** which is used as the primary hash table structure (its individual 'slots' represent the 'cells' of the hash table), an ***int size*** which is the maximum capacity of the hash table.

The member functions include:

1. Constructor: ***HashTable();*** - This creates an "empty" hash table (size set to 0).
2. Destructor: ***~HashTable();*** - This calls on destroy() which is explained next.
3. ***void destroy();*** - This function deletes the elements of the hash table. Firstly, this function checks if the hash_table vector is empty. If it is not empty, the procedure is carried out to delete and clear all elements from the linked list as well as set the size to 0. After the linked list at each index is cleared, the array is deleted. This function is extremely important in the ***init*** function. If the ***init*** is called twice within a function, it is necessary that the first hash table be deleted in order for the second, new hash table to be created. This function does not have a return type.
4. ***int HashFunction(const int val);*** - This function calculates the index value in which the key should be placed using the hash function "key mod size". This function has a time complexity of constant and it returns an integer.
5. ***bool isEmpty();*** - This function returns a true if the hash table is empty. It is used in the destroy function.
6. ***bool init(const int m);*** - This function returns bool depending on if it was successful defining the size of the hash table. It allocated a NULL to each index in the array The expected average runtime for this is linear.
7. ***bool insert(const int k)*** – This function returns bool depending on if it was successful in inserting a key. Firstly, it checks if the slot is empty. If so, the key becomes the first element of the linked list at that index. Secondly, it checks if they key is already found in the hash table using the search function (constant time complexity). Else it finds the next free space along the linked list and enters the key at the key. The expected average runtime is constant as it is expected that the key will be entered at the index calculated by the hash function.
8. ***bool search(const int k);*** - This function returns bool depending on if the key was found in the hash table. It calculates the hash index using the hash function. The function then goes to that index in the table. It checks to see if the first element of its respective linked list is NULL. If so, returns false. Else, it searches the linked list for the element. The expected average runtime is, therefore, constant.
9. ***int returnIndex(const int k);*** - This function returns the index (int) in which the key is found. This function is primarily used in outputting the index in the driver file.
10. ***bool remove(const int k);*** - This functions returns bool depending on if the function was successful in removing the key in the hash table. This function primarily keeps track of the previous node, finds the node to delete, removes it and links the next pointer of the previous node to the next node of the deleted key. The expected average runtime is constant.