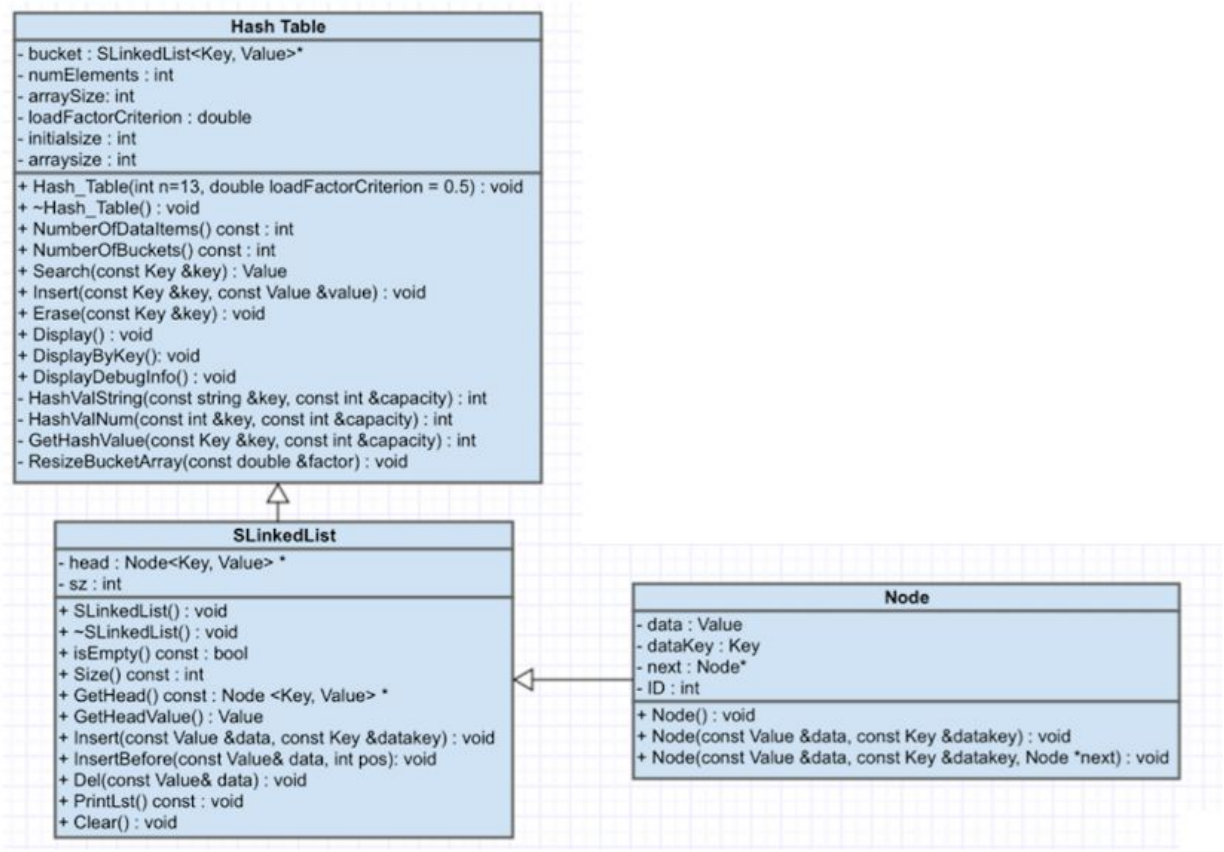
Hash Table



Overview

We designed our team's hash table using a dynamic array to hold the data. In order to support chaining, we implemented singly linked list and node classes, and each element of our dynamic array contains list objects rather than data elements. To prevent collisions from occurring due to insertion, we employed the following methods: (1) We utilized random hashing, with MULTIPLIER = 25173, ADDEND = 13849, and MODULUS = 65536; and (2) we ensured that one cannot insert a value at a key that already contains another value. Our hash table design is able to accept any data type for its keys and its values.

Data Members (Please note that all data members are private.)

int numElements: This integer holds the total number of data elements in all of the buckets of the hash table. It is private, but can be accessed through the NumberOfDataItems() method.

int arraySize: This integer holds the total number of buckets in the hash table. It is private, but can be accessed through the NumberOfBuckets() method.

double loadFactorCriterion: This double-precision floating point number stores the ratio of the number of elements to the array size at which, when inserting elements, the bucket array must be resized and its elements remapped. It is private and cannot be changed once initialized.

SLinkedList<Key, Value>* bucket: This array of singly-linked lists stores the data of all of the buckets in the hash table. It contains nodes that store both a value and the key of that value. The SLinkedList and Node classes contain Hash_Table as their friend. During runtime, the array is resized whenever the ratio of the number of elements to the array size reaches loadFactorCriterion.

Methods

Private:

int HashValString(const string& key, const int& capacity): This method gets the location in the bucket array that a key hashes to when it is a string by running its first character through a random hashing algorithm. The capacity is the number of elements in the array. O(n) is the runtime since strings can vary in length.

int HashValNum(const int& key, const int& capacity): This method gets the location in the bucket array that a key hashes to when it is an integer by running it through a random hashing algorithm. The capacity is the number of elements in the array. O(1) is the runtime since a constant number of operations are performed.

int GetHashValue(const Key& key, const int& capacity): This method calls either the HashValNum(...) or HashValString(...) methods, depending on which is necessary to call. The capacity is the number of buckets in the hash table. O(1) is the runtime when HashValNum(...) is called, since a constant number of operations are performed, and O(n) is the runtime when HashValString(...) is called, since n is the number of characters in a string.

void ResizeBucketArray(const double& factor): This method dynamically resizes the array of buckets and works as follows:

1. A new array of buckets is declared and initialized to be the current arraySize * factor.
2. A for-loop iterates through each bucket in the hash table.
    a. Within that loop, a while loop iterates through each node in the bucket's list object and remaps the items to those within the new array using the GetHashValue(...) method and the linked list's insert method.
3. To prevent memory leaks, the old array of buckets is deleted and then assigned to the address of the new array.
4. arraySize is then multiplied by factor.

The runtime of this function is O(n). The total number of operations for the double-looping structure is only arraySize + numElements, and these two values are related because of the resize operation. (The insertion operation for the list has a runtime of O(1) since all items are inserted at the beginning of the lists).

Public:

Hash_Table(int n = 13, double LoadFactorCriterion = 0.5): This method serves as a constructor for the hash table. It can be invoked either by using the default values or by assigning custom values. n is the number of buckets the hash table will contain in its initial state, and LoadFactorCriterion simply assigns a value to loadFactorCriterion (described above). The runtime is O(1) since a constant number of operations are performed.

~Hash_Table(): This method clears all of the buckets before deleting the hash table. The total runtime of this function is O(n) since n varies linearly with the number of elements in the table.

int NumberOfDataItems(): This method returns the total number of elements in all of the buckets of the hash table. Its runtime is O(1) since a constant number of operations are performed.

int NumberOfBuckets(): This method returns the total number of buckets in the hash table. Its runtime is O(1) since a constant number of operations are performed.

Value Search(const Key &key): This method searches the hash table for the value stored at a particular key using the GetHashValue(...) method. It works as follows:

1. The head node of the linked list at the particular hash value is obtained.

2. If the list is empty, an underflow error indicating that no value was returned is thrown. (O(1) since constant number of operations are employed.)
3. For non-empty lists, a while-loop iterates through each element of the list. If an node in the list has the same key as the one used in this method's argument, its value is returned.
4. If the node with the specified key is not found, an underflow error is thrown indicating that the bucket does not contain a data item with the specified key. This is the worst-case scenario. (O(n) since n could be the number of elements in the hash table.)

On average, the runtime of this method is O(1). However, in the worst case scenario where all of the items are in the same bucket, O(n) is this method's runtime.

void Insert(const Key &key, const Value &val): This method attempts to insert an item into the hash table at the desired key. It works as follows:

1. The index in the array of linked lists to assign the value to is determined using GetHashValue(...).
2. The head of the list is obtained, and if the list is empty, a value is simply inserted to the head node. (O(1) since constant number of operations is performed).
3. If the list is not empty, the entire list is traversed to ensure that the key at which the user is trying to insert a value is not already occupied. If it is, the insertion is unsuccessful. Otherwise, the element is inserted at the end of the list. (O(n) since number of operations varies directly with number of elements in that particular list).
4. If an insertion is successful and the number of elements / the array size is greater than loadFactorCriterion, the bucket array is resized using ResizeBucketArray(...) and all of the items are remapped.

In the worst case scenario, the bucket will contain all of the values in the hash table, and upon inserting another value, the hash table will need to be resized. The runtime of this scenario will be O(n) since the runtime of (3) for this function is O(n) and the runtime for ResizeBucketArray(...) is O(n) (please see description). On average, however, the insert method running time will be O(1) because the only operations that will be performed are getting the hash value and assigning the desired insertion value to the key.

void Erase(const Key &key): This function erases the value stored in the specified key. It works as follows:

1. The index in the array of linked lists to erase a value from is determined using GetHashValue(...).
2. The head of the list is obtained, and if the list is empty, a message is displayed indicating that the key does not contain a value. (O(1) since constant number of operations is performed).
3. If the list is not empty, it is traversed until the node containing the desired key is found. If it is, a deletion is performed and numElements is updated to reflect the change. If the entire list is searched, and an element of the specified key is not found (this is the worst case scenario), then a message stating that although the key hashed to the list being searched, no data item with the specified key was found. When this occurs, the runtime is O(n) because it varies directly with the number of elements in the list, and that number could be the number of elements in the hash table.

In summary, when the buckets are empty or contain only one element, the runtime will be O(1) (constant # of operations); otherwise, it will be O(n).

void Display() const: This method calls display for each of the buckets in the hash table. Because the total number of elements in all of the buckets of the hash table is proportional to the number of buckets, the total runtime of this method will be arraySize + numElements, or O(n). (Note that arraySize is included in the calculation because empty buckets print NULL).

void DisplayDebugInfo() const: This method shows relevant information about the hash table, such as the number of elements it contains, the maximum chain length, and the average chain length. Because it iterates through the array of buckets to determine the values that are outputted to the screen, the total runtime of this method is O(n).

Hashing Algorithm

In order to ensure the minimum number of collisions, our team employed random hashing and the ability to select a custom key. Our formula is as follows:

```
int randomInt = ((MULTIPLIER * (int)key) + ADDEND) % MODULUS;
return randomInt % capacity;
```

Where capacity is the number of elements currently in the array, MULTIPLIER = 25173, ADDEND = 13849, and MODULUS = 65536.

When the user enters a key, it is transformed into a large random integer and then reduced by modulo arithmetic to a hash value. Although this algorithm is sufficient for keys which are not strings, when a key is a string, a more complex algorithm must be employed to determine differences in key values. When the key is a string, the value to be transformed to a hash value is created by summing all of the characters in the string, and multiplying each character by a changing value. The following sequence of code is performed on strings:

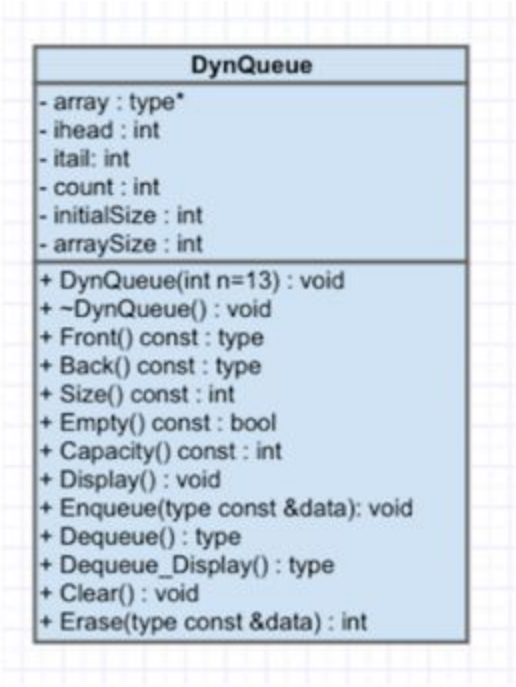
```
int sumCharacters = 0;
int keyLength = key.length();
double factor = 500000;

for (int i = 0; i < keyLength; i++)
{
        sumCharacters += (int)((double)key[i] * factor);
        factor /= 2.07;
}
```

The reason for the factor variable is to ensure that value of sumCharacters is not the same for two equal-length strings with the same characters. For instance, the value of the key for “Dog” and “God” would be the same if the variable sumCharacters wasn’t altered in any way (Dog = 68 + 111 + 103 = 282; God = 71 + 111 + 100 = 282). Unfortunately, this algorithm is stable only for keys whose sizes are less than or equal to 23 characters in length (of the normal ascii character set). Fortunately, though, it is able to prevent two keys with different strings that would sum to the same value from causing an error.

The maximum runtime of this algorithm is O(n), where $0 < n <= 23$ and n is the length of the string.

The advantage of using these two hashing algorithms is that if the user were to insert values using a sequence of keys in the format key, key + arraySize, key + 2 * arraySize, … , key + n * arraySize, all of the items would map to a different location on the hash table rather than ending up in the same linked list. Ultimately, this ensures that operations such as insert(), search(), and erase() run at O(1) rather than approaching O(n).

Dynamic Queue



DynQueue(int n = 13)

This is the constructor. It takes in a value to be the size of the array. The value defaults to 13. The minimum array size will be 1. It then creates an array of said value, and sets the other variables to appropriate values. Will be O(1).

~DynQueue()

Simply deletes the allocated array. Will be O(1).

type front() const

Returns the value at the front of queue, and will throw an exception if it is empty. Will be O(1).

type back() const

Returns the value at the front of queue, and will throw an exception if it is empty. Will be O(1).

int size() const

Returns the number of elements currently in the array. Will be O(1).

bool empty()

Checks count and sees if the array is empty. Will be O(1).

int capacity()

Returns the size of the array. Will be O(1).

void display()

The display utilizes a modified dequeue to display the list. It first creates a temporary array of the same size as the original. It dequeues the first value in the original array and sets it as the first value in the temporary array, and displays it. This happens until the original array is empty. Then it sets the temporary array as the main array, and resets all indexes to original values. This will be O(n).

void enqueue(type const &data)

Enqueue adds an element to the back of the queue. First it checks if the resulting array with the newly inserted value would be over the capacity. If so, it will create a new array of double the size and copy all elements over, and then proceeds with the enqueue. This would be the worst case, and would be O(n). It then sets the new value to the end of the list, with appropriate modifications to the indexes to loop around the queue and make it circular. This would be the average case with O(1).

type dequeue()

Dequeue modifies the head index to remove the first value, with modifications to loop around if need be. This would be O(1). It then checks if the array needs to be reduced in size. If so, it creates a new array with half the capacity and copies the elements over. This would be worst case scenario and be O(n). This then returns the value that was dequeued.

type dequeue_display()

This is the same as dequeue, however; it does not contain a resizing function. This is only used for the display, and would be O(1). This then returns the value that was dequeued.

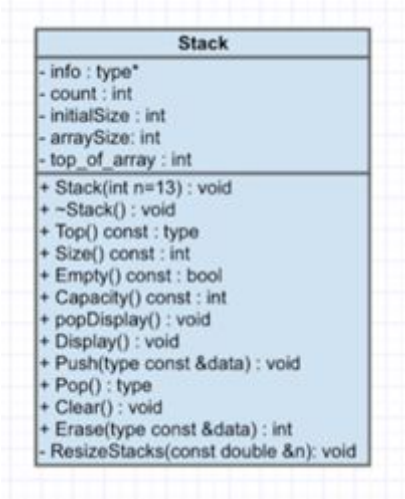
void clear()

Clear creates a new array of initial size, and sets it as the main array. This would be O(1).

int erase(type const &data)

Erase first creates a new array of the same size. It then takes the original array and checks the head value. If the value does not match the input value, it copies it over to the new array. Otherwise, it just skips the value. It keeps doing this until the original array is empty. The new array is size reduced as necessary. The running time would be O(n). Then it returns the number of elements erased.

Dynamic Stack



- Stack(int n=13) : void

The constructor for Stacks takes an argument to be the initial size of the array and allocates memory. If the argument is ever 0 or any negative number the array size will be set to one. The default array size is 13. The contractor also initializes the data members Info, count, initial size, array size, and top of array. This will be O(1).

- ~Stack() : void

The destructor deletes the memory allocated to the array initially saved in the constructor. This will be O(1).

- Top() const : type

The top function returns the element at the top of the stack by check the top of array data member and making sure the stack is not empty. The top of array data member is the index on the stack for that element being returned. This will be O(1).

- Size() const : int

The size function will return an integer with the currently array size. The array size is determined by the count data member which will keep incrementing every time an element is pushed to the stack. This will be O(1).

- Empty() const : bool

The empty function will return true or false whether the function is empty or not. This function uses the top of array data member to determine whether or not the the stack is empty. If the top of array function is -1 then the stack is empty, but if it is greater than the the stack is not empty. This will be O(1).

- Capacity() const : int

The capacity function returns an integer the will be the current allocated size of the array. This will simply return the value of array size. Array size will be updated any time there is a change in the size of the array determined by the number of elements in the stack. This will be O(1).

- popDisplay() : void

The pop display function is essentially a copy of the pop function but does not use the resizing function since the purpose of it is simply to display the elements. This function is only used when display is called. This function does throw. This will be O(1).

- Display() : void

The display function will display the elements in the stack by creating a temporary stack that will hold the values of the current stack while they are being pop from the current stack and displayed. After this, the elements will be pushed back to the stack in the same order. This will not require the stack to be resized as the stack will be the same length. This function will call the popDisplay method to pop the elements. This will be O(n).

- Push(type const &data) : void

The push method will take as an argument an element that will essentially be pushed into the stack. This method simply gets the element and inserts it at the correct array location in the stack. The method does call the resizing function if the array size is the same as the initial size since the stack will be out of space. This results in the array size being doubled. The array size and count are updated as needed. This will be O(1).

- Pop() : type

The pop function will eliminate the element at the top of the stack by using the data member top of array. This function will return the element that was eliminated once it is popped. The count and top of array data member are updated as needed. This function does throw an underflow error if the stack is empty. This will be O(1).

- Clear() : void

The clear function will delete the array that is currently allocated with the stack to eliminate all the variables. The data members count and top of array and array size are updated as required and a new array will be created with the initial size and will be empty. The function does check to see if the stack is already empty and will notify the user if that is the case. This will be O(1).

- Erase(type const &data) : int

The erase function will take as an argument as an element to be deleted. First the function will check if the stack is empty and will throw an underflow error if it is so. If not, the function will create a temporary array that will hold the elements in the stack that will need to be kept. The function check to see if the top of the stack element matches that if the augment element and will eliminate it if so and will update the count, top of array and number deleted members as needed. If it does not match then the element will be stored in the temp array while the function continues to run through the stack until it is empty. After, the function will push all the elements in the temporary array to the new array using the function push. The function will then check to see if after the elements were deleted the stack needs to be resized and finally will return the amount of elements that were deleted. This will be O(n).

- ResizeStacks(const double &n): void

The resize stack takes as an argument a double that is to determine where the function is being called from. This will determine if the function will use the doubling or the halved operations. The functions checks to see the argument passed and determines, if 1, the array will be halved. It will create a new temp stack where the current elements will be stored by using the pop and push functions. After the elements are stored in the new temporary stack, it will create a new stack with the new array size and initialize it. Then the temporary stack values will be moved back to the new stack using the pop and push functions. If the argument passed is anything else, then the stack size will be doubled. it will make sure that stack size is equal to the count and will then take similar steps to double the size of the stack. This function will be O(n).