ECE 250 – Data Structures and Algorithms, Fall 2020

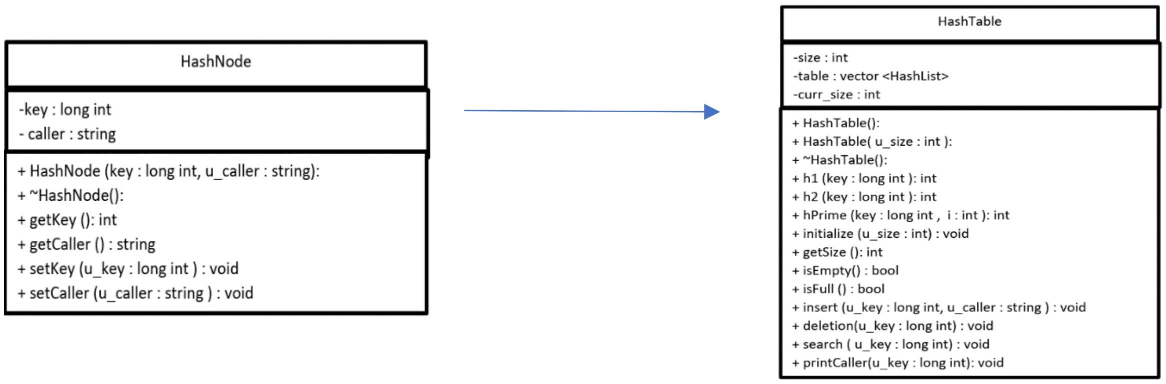
Design Document for Project 1 – Solving collisions through double hashing and chaining methods

A) Solving collisions through Double Hashing (Files – openhttest.cpp, hashA.cpp, hashA.h)

Overview of Classes

- Two classes used- class *HashNode* and class *HashTable*, declared in hashA.h and defined in hashA.cpp.
- HashNode class has data members, like caller and key which describe the characteristics of the data stored by the Hash Table. It has member functions which are used for setting and getting the values of the private data members.
- HashTable class stores a vector of HashNode elements. It has member functions like *insert, delete and search* defined to perform these operations. The objects of this class are used to set up new Hash Tables.

UML Diagram


HashNode
-key : long int
- caller : string
+ HashNode (key : long int, u_caller : string):
+ ~HashNode():
+ getKey (): int
+ getCaller () : string
+ setKey (u_key : long int) : void
+ setCaller (u_caller : string) : void
HashTable
-size : int
-table : vector <HashList>
-curr_size : int
+ HashTable():
+ HashTable(u_size : int):
+ ~HashTable():
+ h1 (key : long int): int
+ h2 (key : long int): int
+ hPrime (key : long int , i : int): int
+ initialize (u_size : int) : void
+ getSize (): int
+ isEmpty() : bool
+ isFull () : bool
+ insert (u_key : long int, u_caller : string) : void
+ deletion(u_key : long int) : void
+ search (u_key : long int) : void
+ printCaller(u_key : long int): void

Design Decisions and Performance Evaluations

- For both the classes simple default constructors were made. There is an extra parameterised constructor in HashNode class which is used in a function called initialise(_) of HashTable class.
- The destructor for HashTable class clears vectors using clear() function, thereby preventing memory leaks. HashNode has a simple destructor.
- HashNode class has simple accessor functions like *getCaller() and getKey(),* which return private data member values. It has setter functions like *setCaller(string)* and *setKey(int)* which each accept one parameter of appropriate data type. These are used to manipulate the value stored by the corresponding private member.
  All these operations are performed in constant time.

  <u>HashTable class has following functions –</u>

a) **h1(_), h2(_) and hPrime(_ , _)** - used to describe primary, secondary hash functions and return index values. Each take one long int parameter to calculate from key. hPrime(_ , _) also takes an integer which informs about collisions. These operations are performed in linear time of Θ(1).

b) **bool Isfull(), bool isEmpty()** – returns status of table by comparing current size with table size in constant time, Θ(1).

c) **void initialise (int)** - behaves like a constructor and helps clearing and resizing the table each time 'n' command is called. It uses the integer value to resize. While resizing, it inserts elements with a *key value of (-1), indicating empty state*. This operation is performed in constant time of n*Θ(1) = Θ(1) , once the size of the table is known.

d) int Search(long int) and void printCaller(long int)

- Both the functions take one long int parameter to represent key. Search(_) returns the position of the element for a given key, on the hash Table and printCaller() prints the name of the caller for that key, based on the value returned by search (key not found if return value = -1).

- Search is performed using a while loop, where the loop breaks if it encounters an empty spot (indicated by holding a key value of -1 at that position). For best case, Search runs for a constant average time Θ(c). Worst case would be linear time Θ(n), if the last element was inserted at the last empty spot during insertion.

e) **void Insert (long int, string)** – takes 2 parameters that store key value and caller name and inserts them into appropriate places while it resolves collisions. It first searches for the key in the table to prevent duplicates and checks if the table is full. This is performed in constant average times.
   - Best case would be when no collisions are met and insertion is straight forward, done in constant time, Θ(1).
   - Worst case would be when the element encounters maximum collisions and was inserted at the last empty spot. This would be done in linear time Θ(n).

f) **void Delete (long int**) – takes one long int parameter for key. First, checks if the key exists using search. It then deletes the matching element at the position (returned by search) by setting null value for caller name and *state 0 for key value to represent deleted state*. The best and worst case is based on search() function.

Test Cases

For each class, unit testing was done to ensure each member function was performing as per expectations. Some special test cases for insertion and deletion, were designed to address concerns like -

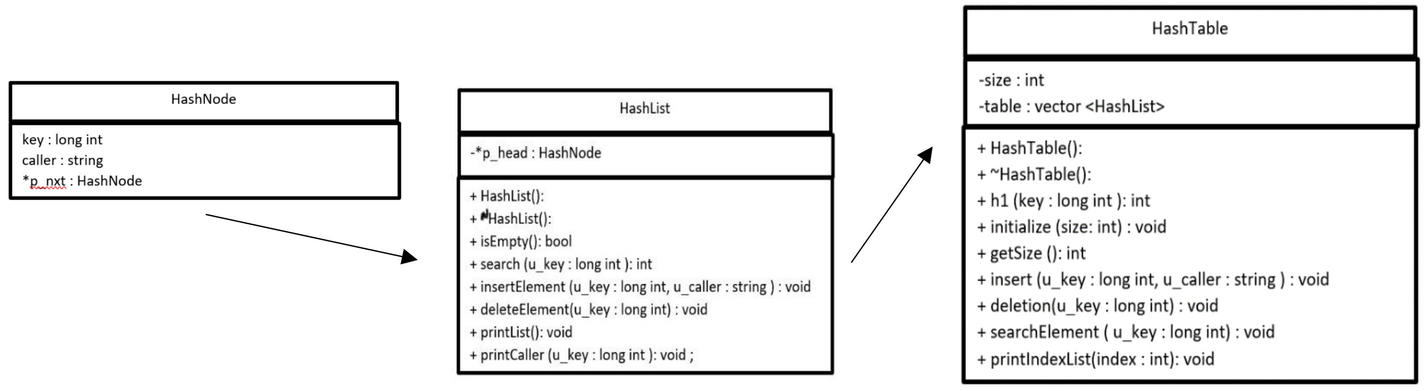
- full table or duplicate key then no insertion
- empty table or inexistent key then no deletion
- table resized then old keys should be wiped and if same keys inserted then, should be according to new size

B) Solving collisions through Chaining (Files – orderredhttest.cpp, hashB.cpp, hashB.h)

Overview of Classes and Structures

- One structure for linked list node (HashNode) with members representing key, caller, and pointer to next node
- One class for Linked lists(HashList), one class for Hash Tables (HashTable).
- Hash List class has one head pointer of the type HashNode. It has member functions that perform search insert delete operations on the linked list based on a key value taken as input.
- HashTable class has a vector of linked lists (Hash List) and it has member functions that wrap Hash List class functions to perform those operations at the right index.

UML Diagram


HashNode
key : long int
caller : string
*p_nxt : HashNode
HashList
-*p_head : HashNode
+ HashList():
+ ~HashList():
+ isEmpty(): bool
+ search (u_key : long int): int
+ insertElement (u_key : long int, u_caller : string) : void
+ deleteElement(u_key : long int) : void
+ printList(): void
+ printCaller (u_key : long int): void ;
HashTable
-size : int
-table : vector <HashList>
+ HashTable():
+ ~HashTable():
+ h1 (key : long int): int
+ initialize (size: int) : void
+ getSize (): int
+ insert (u_key : long int, u_caller : string) : void
+ deletion(u_key : long int) : void
+ searchElement (u_key : long int) : void
+ printIndexList(index : int): void

Design Decisions and Performance Evaluations

- The constructors for both classes are simple default constructors. The Destructor for HashList() takes constant time (n* Θ(1) = Θ(1)) once size is known. Destructor for HashTable is a simple clear command to empty the vector.

HashList class has following functions –

a) **HashNode* gethead()** - returns head of the list in constant time, Θ(1).

b) **bool isEmpty()** - checks if the list if empty in constant time, Θ(1) by comparing head pointer.

c) **void initialize(int) –** works exactly like the function for double hashing, with a constant tight bound of Θ(1).

d) **int search(long int) –** it accepts key value and searches the list till the key is greater than the key values of the elements that are already present. This comparison is possible since the list is arranged in ascending order. Best case would be if the very first element matches our criteria for sorting. The worst case would be searching till the end of the list. However, since we know the number of elements in the list will be a constant number, it will take average time to search the entire list. Hence, search is constant time on average, with a tight bound of Θ(1).

e) **void printCaller(long int) -** it accepts one input as key, performs a search on this value, to get the position and then, it prints the caller name at that position. The function uses a *while loop* to iterate till the number of iterations matches with the position of the element. Since the position is calculated by search, this function also performs with a constant time on average, having a tight bound of Θ(1).

f) **void insertion(long int, string) -** it accepts two parameters, for key and caller. It does the insertion of the element to in a sorted manner. Insertion is based on two cases. These cases are –
   i. **beginning of list –** this is the best case and would take constant time, Θ(1).
   ii. **later in the list –** has two pointers comparing current value and the next value with the key to be inserted. Worst case would be inserting at the end of the list but since the size of the list would be a constant, it performs in constant time on average, Θ(1).

   Insertion also makes use of the search function to ensure no duplicates, and search runs on constant time for average case.

g) **void deletion(long int) –** Deletions runs on the similar lines as search and insertion. The element is first searched in the list. Based on the return value of the search function, a while loop is run until number of iterations matches with position on the list. Best case would be if the deletion is at the very first node. Worst case would be if the deletion is at the end. Then we would be taking average case since the position of an element is fixed in a sorted list of constant input size. Therefore, the function performs in constant time on average, Θ(1).

h) **void printList() –** This function prints the key value of all the nodes present in a list. It takes constant time to run, which again depends on the input size. Since the input size is constant, the function performs in constant time for an average case, Θ(1).

- HashTable class has member functions which wrap the main functions of the linked list class. Since all the functions of the linked list class run on a constant time on average, the performance of these functions is on similar lines.

Test Cases

The testing strategy adapted was like the one for double hashing, i.e., unit testing and special cases. Extra precaution was taken care to ensure no memory leaks were encountered during rehashing, by using valgrind check.