Project 2 Design Document

Bill Yao (b29yao)

1. Overview of classes

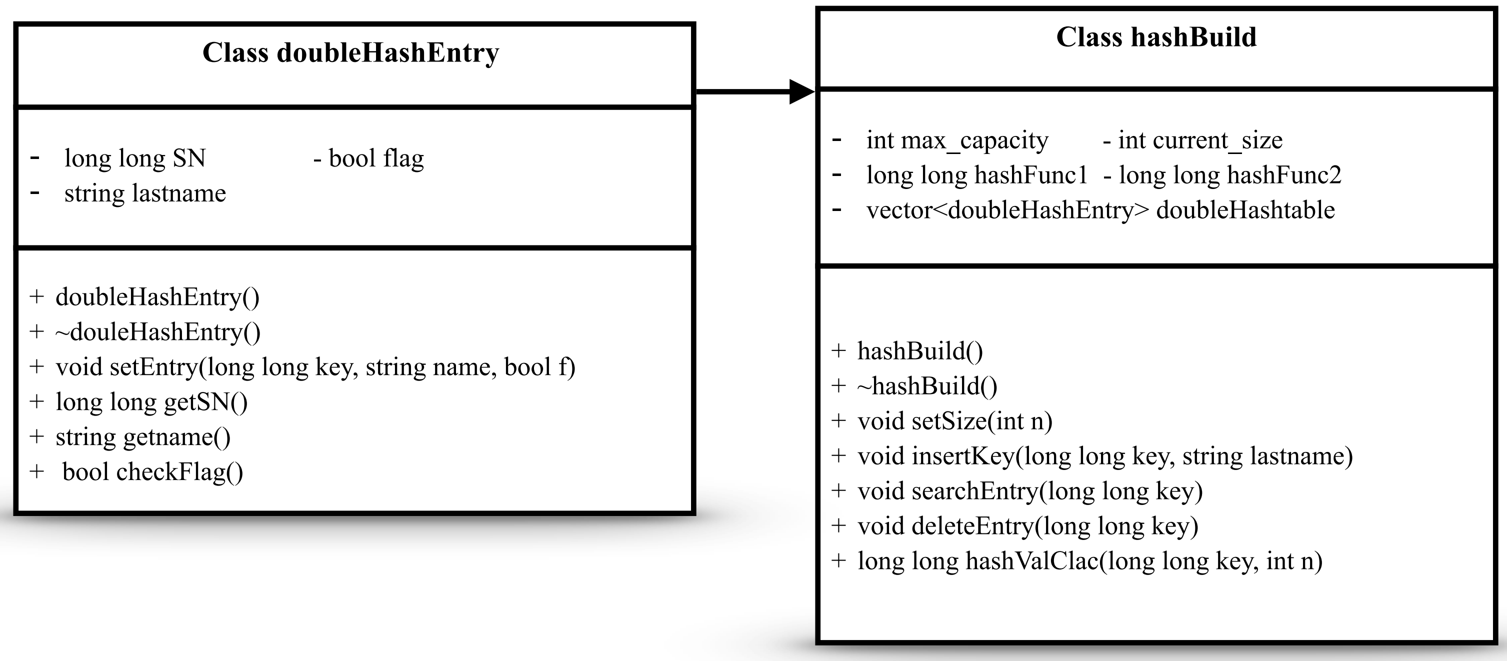
In this project, I have implemented two classes for the double-hashing hashtable and three classes for the chaining hashtable.

For the double-hashing hashtable, we have a “doubleHashEntry” class that stores the student number, the student’s last name, and a boolean flag that detects if the node has been used. Also, we have setter and getter member functions to store inputs into the node and get the variable out of the node. Then we have a “hashBuild” class that constructs a vector with node class. It has variables such as “max_capacity” and “current_size” to check the size of the hashtable, and also two variables that store the value of two hash functions.

For the chaining hashtable, we have a “node" class that stores the student number and students’ last names, with several getters and setters functions that manipulate the inputs for each node. Then we have a “chain” class that uses a vector to store every node. Both of these two classes are friends with a final implementation class “hashtable”. The tastable class also uses a vector to store every chain and it has several member functions to do setting, insertion, searching, deletions, etc.

2. UML class diagram

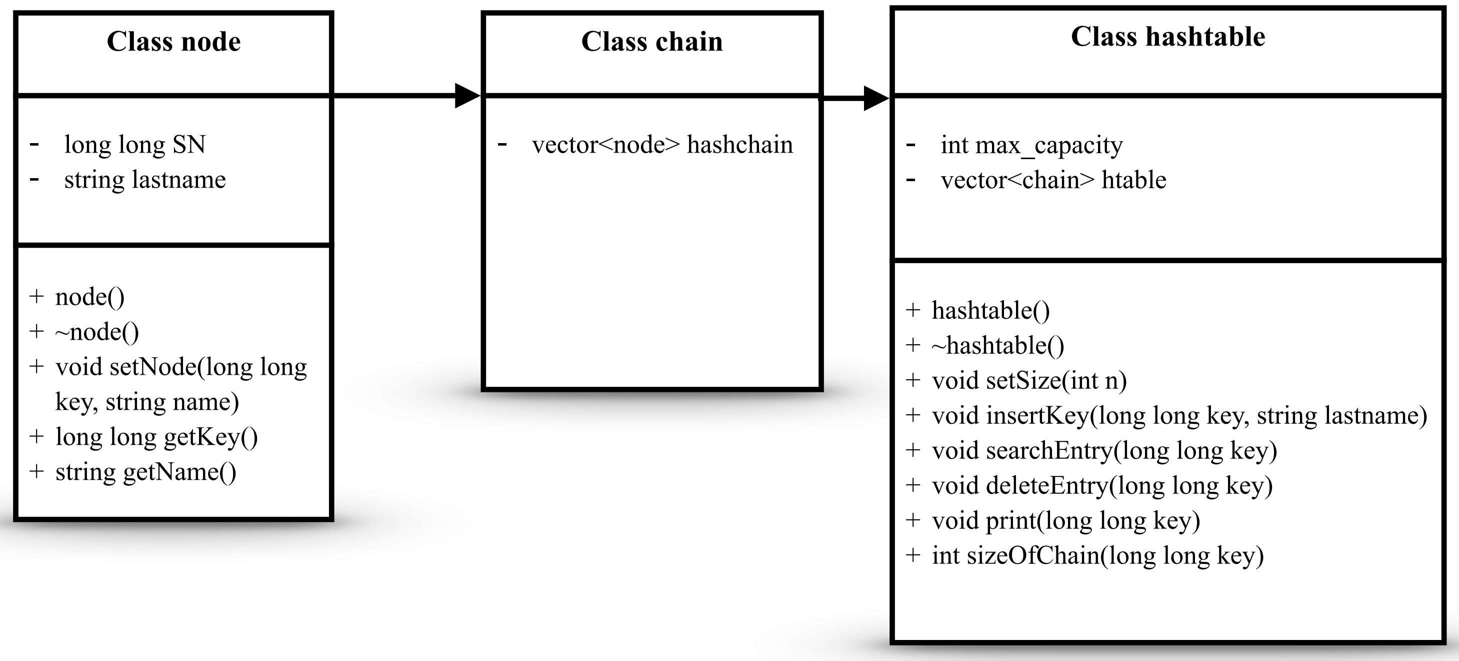
Double hashing:



```
Class doubleHashEntry

-   long long SN               - bool flag
-   string lastname

+ doubleHashEntry()
+ ~douleHashEntry()
+ void setEntry(long long key, string name, bool f)
+ long long getSN()
+ string getname()
+  bool checkFlag()
```

```
Class hashBuild

-   int max_capacity        - int current_size
-   long long hashFunc1  - long long hashFunc2
-   vector<doubleHashEntry> doubleHashtable

+ hashBuild()
+ ~hashBuild()
+ void setSize(int n)
+ void insertKey(long long key, string lastname)
+ void searchEntry(long long key)
+ void deleteEntry(long long key)
+ long long hashValClac(long long key, int n)
```

Chaining:



3. Design Decisions

For class doubleHashEntry and class node, these are the two basic classes that hold the information of a node after each insertion. We need getters and setters in order to store and grab the member variables value in the node. And we initialize values in the constructor for each variable. There are no operators that we used to override, nor did we use any pass by reference on any parameters since they are constantly taking in new data as the input is being pushed in and popped out.

For class chain, this only contains a vector that is consist of node. We would use it to consist hashtable in the chaining section.

For hashBuild and hashtable, we have identical member functions such as insertion, deletion and searching. The logics are like the following:

void setSize(int n): if the n is a integer -> then set maximum capacity and size to be n;

void insertKey(long long key, string lastname): 1.loop through the entire hashtable -> 2.if the student number at position [hashVal] is equal to input key, insertion fails -> 3.if no student number and student name stored at position [hashVal], insertion success

void searchEntry(long long key): 1.loop through the entire hashtable->2.if the passed-by node has never been used, deletion fails ->3.if the student number at last [hashVal] is not equal to the key, deletion fails->4.if the passed-by node has been used but the key still not equal to student number, increase iterator

4. Testing Strategy

For open-addressing hashtable, we have four different test cases for each member function:

Capacity Test: We set the hashtable size to be, for example, 4. (m 4) Then we do the insertion of four random nodes, to see if the vector will still accept the fifth input.

Insertion Test: We set the hashtable size to be 4, and separately insert 5, 9, 13, to test if the node is placed in the right index when the collision happens. Then we use the search function to determine where those keys are placed in the hashtable;

Deletion Test: When we delete a node from the hashtable, we need to check if the node inserted afterwards is still accessible. For example, if the vector contains 5, 9, 13. After deleting 9, we wanna s 13 and s 5 to check their position. Also, we can search 9 to see if it was correctly deleted.

Searching Test: Very similar to the deletion test, we would delete a middle node and see if the following node is still accessible

For chaining hashtable, we also implemented four different test cases for member functions:

Insertion Test: We insert a series of nodes that would be placed in the same chain, and then we print the selected chain to see if the nodes are placed in descending order.

Deletion Test: After deleting a node from a chain, we would print the chain to check if it was deleted. Or we can use searching function to see if we can still access the deleted node.

Searching Test: After inserting a new node, we try searching it and printing out its location. Then we delete the node, and see if the searching still works on this node.

Print Test: We need to check when a chain, after several deletions, is completely empty, if the print function can still print the chain, or if it prints out the empty message.

5. Performance considerations

For the open-addressing hashtable, the **average running time is constant** for each insertion, deletion and searching. Since the worst running time case is O(n), when it has to iterate through the hashtable in order to find a suitable position after calculating the hash functions, whereas the best running time is O(1) when the insertion, deletion and searching have no collusion and directly access the node in the hashtable.

For the chaining hashtable, the **average running time is also constant** for insertion, deletion and searching. The best running time comes when the first node in the selected is our desirable position, whereas the worst running time comes when we have to traverse through the entire chain in order to locate the node/position.