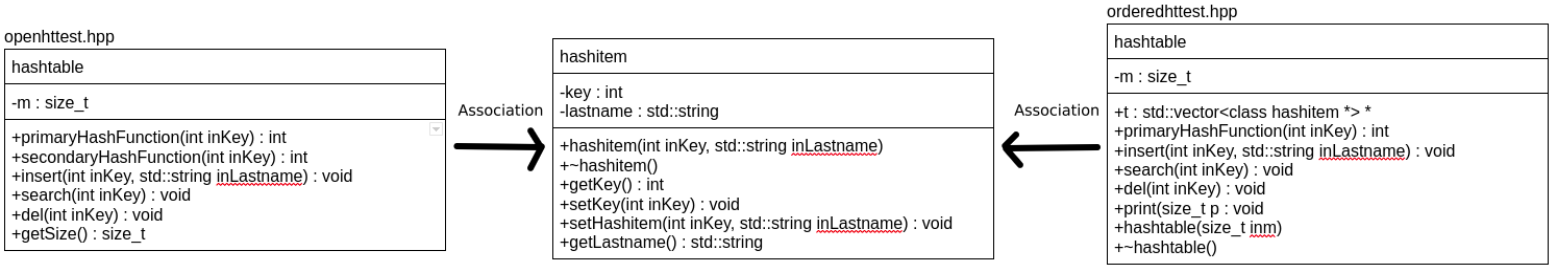
1. **Overview of Classes**

- What classes did you design: hashitem, hashtable
- The role of each class: the hashitem class stores the **key** and **lastname** while the hashtable class holds a hashtable and its functions (e.g. insert, delete, search, etc…).
- How do your classes relate: The hashitem class is a dependency of the hashtable class

2. **UML Class Diagram**




3. **Details on Class Decisions**

- hashitem class
  - The hashitem class includes private data members **key** and **lastname** as well as getters and setters to access them
  - I made the data members private as there is no reason for other things to access the variables directly except for member functions
  - It also includes a constructor and a destructor
- hashtable class
  - The hashtable class includes private data members **m** which is the size of the hashtable and has a getter as **m** does not change after its initial value is set in the constructor. Note that the following member functions are all public
- For open addressing
  - It also includes a public data member **t** which is an array of hashitems.
  - **int primaryHashFunction(int inKey)** : returns inKey mod m
  - **int secondaryHashFunction(int inKey)** : returns inKey divided by m mod m. If the result is even, we add one to it.
  - **bool insert(int inKey, std::string inLastname)** : we first search the hashtable for the key to determine if we've already inserted the key. Then we try to insert the key at the index of the primaryHashFunction, if we have a collision, we update the index using the secondaryHashFunction and try again until we've found an empty spot or we cannot insert as the table is full.
  - **void search(int inKey)** : check if we found the hashitem at the index of the primaryHashFunction. Then add the secondaryHashItem to the index and check again. Repeat until you have found the hashitem at the location. If we reach an empty index or finish iterating through the hashtable first, we did not find the key.

- **void del(int inKey)** : we search for the key and then set the key to -1. This is because we do not rehash the hashtable after a delete so we need a flag to tell other functions that we deleted a value.
- **size_t getSize()** : returns m

- For chaining
  - It also includes a public data member **t** which is an array of vectors of hashitem pointers.
  - **int primaryHashFunction(int inKey)** : returns inKey mod m
  - **int secondaryHashFunction(int inKey)** : returns inKey divided by m mod m. If the result is even, we add one to it.
  - **void insert(int inKey, std::string inLastname)** : Inserts a hashitem at the index from the primaryHashFunction. Goes through the chain and inserts itself once it finds an index that it is larger than.
  - **void search(int inKey)** : Goes through the vector at the index of the primaryHashFunction looking for a key which matches.
  - **void del(int inKey)** : Searches for the key using the above method, then when it finds the key it erases it from the vector and deletes it
  - **void print(size_t p)** : Loops through the chain at index p and prints every element
  - The reason why **t** is public for both open addressing and chaining is that it is accessed extremely frequently by non-member functions and accessing it through getters and setters would increase the number of function calls drastically

**4. Test Cases**

- I tested the following cases for open-addressing
  - Capacity test: I set the size of the array to **m** and inserted **m+1** items. Then I print out the hashtable to make sure that the last item failed to insert and was properly deleted from the memory
  - Double hash test: I inserted hashitems with keys such that I knew they would continuously collide. I used a hashtable of size 10 and inserted hashitems with keys that were a multiple of 10 (e.g. 10, 100, 1000)
  - Search test: I searched a full hashtable for keys and then deleted that specific key and repeated until the hashtable was empty and I confirmed this by searching for a random key to see what the search function would return. This would test that search and delete are working as expected.
  - Insert test: I tested a case where I had an item at **p** and I inserted another item that was rehashed to **p+1**. I then deleted the item at **p** and tried to insert an item with the same key as the item in **p+1**. This is to check that when inserting, we do not insert the same key twice (this covers a corner case where we would have to search through deleted items as I do not rehash the hashtable)
  - Delete test: to test that hashitems were properly being deleted, I inserted a hashitem and deleted it then attempted to search for it. I also printed out the hashtable to confirm that the deleted flag was properly being set

- I tested the following cases for chaining
    - Insert + search + print test: I inserted a large number of hashitems (more hashitems than the size of the hashtable) and searched for them to make sure they were all inserted. Then I printed it out to determine that they were sorted properly.
    - Insert test: I inserted five hashitems with the same primaryHashFunction result and checked that they all appeared at the same index and sorted in the hashtable using print
    - Deletion test: I tested to see how my hashtable would work if I had inserted two hashitems at one location and deleted the one with the largest key. What happened was that the larger hashitem would be deleted and the smaller hashitem would become the only item in the hashtable
- I used valgrind with the test file to determine that there was no memory leaks

**5. Performance Decisions**

- Assuming uniform hashing and average case: Note that with size **m** and **n** input nodes, each index has **n/m** nodes on average, so the probability of a successful insertion is (1 - n/m). Note that 1/(1-n/m) is the expected number of insertions. Since $n \propto m$, n/m and 1/(1-n/m) is always constant. Thus, the time it takes for collisions is always constant as the secondaryHashFunction is O(1).
- Open Addressing
    - Search: primaryHashFunction is O(1), then comparing the item's key is O(1). Then adding O(1) because of collisions that occur. Thus, searching is O(1).
    - Delete: search is O(1), then deleting the item is O(1). Thus, delete is O(1)
    - Insert: primaryHashFunction is O(1), then comparing the item's key and inserting is O(1), then adding O(1) because of collisions that occur. Thus, insert is O(1)
    - Thus, search, delete and insert are all O(1)
- Chaining
    - Search: First we call the hash function which is O(1) as it calculates the mod. To search we increment through the entire vector (which has n/m objects) at the index checking for an equal key which makes it O(n/m). Since n = O(m), we have O(O(m)/m) = O(1). Thus, it is (1)
    - Delete: We first search, then delete the hashitem, both are O(1). Thus, delete is also O(1).
    - Insert: First we call the hash function which is O(1) as it calculates the mod. To insert, we increment through the vector at the index until we either find our key or find a location to insert the key. Thus, it is O(n/m) = O(1) as explained above.
    - Print: This prints each object at an index which is O(n/m). Thus, it is O(n/m) = O(1) as explained above.
    - Thus, search, delete, insert and print are all O(1)