The hash table abstract data type (ADT) is described below. For this assignment you will

- Implement the hash table ADT in the C programming language using an array of linked lists.
- Describe ADT features that are not supported by C.
- Implement the hash table ADT in the C++ programming language using an array.
- Change the C++ implementation, but not the interface, of the ADT to use an array of linked lists.
- Describe C++ features that support ADT’s

Your code will be submitted electronically. It will be tested by the instructor with drivers supplied separately using the gnu C and C++ compilers on one of the CS Department’s linux machines in the taz cluster. There is one driver for the C code and a second driver for the C++ code. Directions for submission and the drivers will be available on the instructor’s website.

The written portion of the project will be submitted on paper in the form of a 1–2 page paper.

The ADT is given below. You may need additional functions and data structures to implement the type.

Functions: type hashtable of elem
create: int → hashtable
insert hashtable × elem → hashtable
remove hashtable × key → hashtable
lookup hashtable × key → elem

Variables: hashtable T; elem e,f; key k;

Axioms:

lookup(create(m),k) = error
remove(create(m),k) = error
lookup(insert(T,e),k) = if e.key == k then e else lookup(T,k)
remove (insert(T,e),k) = if e.key== k then T else insert(remove(T,k),e)

For more information about hash tables, see Ch 11 of Introduction to Algorithms, second edition, by Cormen, Leiserson, Rivest and Stein, or a Data Structures book.

A hashtable is an effective data structure for implementing dictionaries with operations of insert, lookup, and remove. Under reasonable assumptions, the expected time to search for an element in a hashtable is O(1). Hash tables are particularly useful when the universe of keys (say all legal C++ identifiers) is much larger than the working set of keys (all C++ identifiers in a single program).

Let \( U \) be a universe of keys, \( K \) the set of ”working” keys, \( T \) a hashtable indexed from 0 to \( m-1 \), and \( h : U \rightarrow \{0, \ldots, m-1\} \). An element with key \( k \) hashes to slot \( h(k) \). \( h(k) \) is the hash value of key \( k \). For this project, elements contain an integer key and an integer type and the hash function is \( h(k)=k\%m \).

If function \( h \) is not one-to-one, then two keys may hash to the same slot. This is called a collision. Collisions may be resolved by chaining or by a linear probe.

In chaining, the hash table is represented by an array of linked lists. \( T[j] \) points to a linked list that contains the keys that has to location \( j \). Lookups and removes are done by hashing to the desired list, searching the list for the given key, and then returning or removing the element that is found.

In a linear probe, the hash table is represented by a single array. Each array element is either an element or NIL, or DELETED. The hash function determines a location in the array. If the location is NIL, the element is inserted into that array location. If the location is full, the array is searched circularly from that location and the element is inserted in the first NIL or DELETED location. An error occurs if the table is full. Lookups and removes are carried out the same way, except that DELETED locations are skipped. A search terminates when the key is found or an empty location is reached. A location is set to DELETED when its contents is removed.