Lab 6: Separate Chaining

For this lab, you need to implement a **hash table** that resolves collisions by using **separate chaining**.

You will be working on the project already given, and you will need to complete the implementation where indicated. The project contains **two (2) classes**:

- **HTArray** – This class creates objects that point to dynamic arrays of pointers. This is similar to the **DArray** class, with the difference that each index of the array contains pointers to objects of the class **LinkedList**.

  Syntax:

  LinkedList **ht; ← This is a **pointer** (**ht**) that will point to an **array of pointers** that point to **objects** of the class **LinkedList**.

- **Linked List –** This is the usual singly-linked list class that also contains the class **Node**.

The file **lab_06_graph** gives a visual example of how all the classes and data types are connected.

The class **HTArray** already has **member variables**:

- A **pointer ht** that points to a **dynamic array of pointers**, which in turn point to **objects** of the **LinkedList class** (**and DO take a minute to think about this one!**)
- An **int** that stores the **capacity** of the array (the **default capacity** is a **constant** set to **10**)
- An **int** that stores the **number of elements** in the whole hash table.

The **default constructor** of the **HTArray class** is already implemented, so that you can see how an array of pointers is declared and initialized. Below you can find the functions you need to implement to complete the class **HTArray**:

- **Overloaded insertion operator**
  - Prints all the elements in the hash table in the format shown below.
  - Calls the function **printList** from the **LinkedList** class to print each list.

- **Overloaded constructor**
  - **Parameter:** an **int** storing a given capacity for the array.
  - Initializes all member variables. For each index in the array, you need to initialize an object of the class **LinkedList**.

- Function **addElement**
  - **Parameter:** an int that stores the new element to add.
  - Calls the function **hashValue** to get the index where the element should be inserted.
    - The hash value gives you the index, but you need to go into the linked list associated with that index to store the element.

- Function **getNumOfElements**
  - Returns the number of elements in the whole hash table.

- Function **isEmpty**
  - Returns true if the hash table is empty and false otherwise.

- Function **searchElement**
  - **Parameter:** an **int** that stores an element to search.
  - Calls the function **hashValue** to determine the index associated with the linked list that might have the given element and returns true if the element is found, or false otherwise.

- Function **hashValue**
  - **Parameter:** an **int** that stores the key that determines the hash value.
  - This function is **private**.
  - Returns the hash value associated with the key given. Function to use:

h(key) = key mod n;

where **n** is the **capacity** of the array (**NOT** the number of elements)

- **Destructor**
  - You have several items stored in the heap; therefore, you need to carefully think how you will implement the destructor. I will **NOT** give any hints or help; you have enough knowledge to figure out how this function should be implemented.

The **Main.cpp** file contains testing cases to test your program. You should trace them by hand to understand where everything is going AND you should use the debugger to see how everything is organized.

Your **FORMAT** of your **output** should be **EXACTLY** as shown below (note that this is only the first test case.)

```
IDX 0:
IDX 1:
IDX 2:
IDX 3: 3 23 43
IDX 4:
IDX 5: 5 15
IDX 6:
IDX 7:
IDX 8:
IDX 9:

Key 23 found.
Key 15 found.
Key 3 found.
Key 5 found.
Key 43 found.
Key 100 not found.
```