You are to implement an in-memory extensible hash table.

# Requirements

You are to implement an in-memory extensible hash table class in C++. The hash table is to store **non-negative** integers. This class is obviously (I hope) not incredibly useful but is intended for you to learn and demonstrate your knowledge of the index structure discussed in class.

In practice, if the structure was used as an index the table would store key-address pairs where the address would be the disk page that the record with the matching key was contained. Since we are not going to connect our hash table to data we will just store the key value (an integer).

The hash table should be implemented as described in class. This means that the hash table class must have a directory that doubles in size at the appropriate times and buckets of a fixed size. Global depth and local depth must be recorded. Read the next two sections carefully for further details. *Your hash table should allow duplicate values to be inserted*.

# Class Structure

You should implement two classes, an *ExtensibleHashTable* class and a *Bucket* class. Each class should be separated into *.h* and *.cpp* files. The *ExtensibleHashTable* class must contain a directory structure that is an array of pointers to *Bucket*s (not an array of *Bucket*s). The *Bucket* class must contain an array of integers.

Both classes will require other attributes some of which are referenced in the descriptions of the methods and the implementation (the next section).

You must implement the following public methods:

- *ExtensibleHashTable*() – this constructor creates an empty hash table; the number of keys that can be stored in a bucket size should be set to 4; the directory should initially

consist of two entries and the hash function should use only the last bit of the hash value

- *ExtensibleHashTable*(*int*) – this constructor creates an empty hash table; the parameter sets the number of keys that can be stored in each bucket; the directory should initially consist of two entries and the hash function should use only the last bit of the hash value
- *bool find*(*int*) – searches the hash table for the key; if found returns *true*, otherwise returns *false*
- *void insert*(*int*) – inserts the key into the hash table in the appropriate bucket
- *bool remove*(*int*) – searches the hash table for the key; if found, removes all matching values (as there may be duplicate keys) from the hash table; returns *true* if the key was found and removed, *false* otherwise
- *void print*() – prints the contents of the directory and buckets; see below for exactly what should be printed

I expect you to implement additional helper methods which should be made private. Given that this is a container class written in C++ you should also implement a (public) destructor, overloaded assignment operator and copy constructor.

You must use the names exactly as I've set out above and failure do to so will result in a significant mark penalty.

# Implementation Details

## Hash Function

Keys should be assigned to buckets based on the key's bit value. The hash table should use a family of hash functions as described in class. Initially, only the right most bit of the key value should be used to determine which directory entry to look up and which bucket to insert the key in. As the directory size (and global depth) increases more bits should be used.

For example, if the key value was 299 (100101011) and the global depth was 1 the bucket pointer in index *one* of the directory

should be followed. Conversely if the global depth was 5 then the bucket pointer in index *eleven* of the directory should be followed – using the last five digits of the bit value of the key (01011). Note that you can find the bit value of the key by using C++ bit operations – but there is no need to do this. There is a much simpler way using powers of 2 and the modulo operation.

## Structure

The hash table must have a directory which should be an array of size $2^{\text{global depth}}$ of bucket pointers. The bucket objects should contain the local depth and an array of keys. The keys in buckets do not have to be stored in order and may be searched using linear search. Note that this is an inefficient process for an in-memory hash table but recall that we are simulating a table where buckets represent disk pages and we are less concerned with in-memory search costs within a bucket.

## Method Implementation Notes

**Find**

Use the hash function to find the directory index. Follow the pointer in the directory to a bucket and search the bucket for the key. Returns true if key is found, false otherwise.

**Insertion**

Use the hash function to find the directory index. Follow the pointer in the directory to a bucket. Insert the key if the bucket has room.

Otherwise, if the bucket is full, compare the local depth of the bucket to the global depth. *If the global depth is greater* then make a new bucket and assign a pointer to it in the appropriate index of the directory and distribute the values in the original bucket between the original and the new bucket as appropriate.

*If global depth and local depth are the same*, the directory size must be doubled before creating the new bucket. This also entails setting the pointers in the new half of the directory to existing buckets. Note that these processes may need to be repeated in

the situation where a bucket splits, but all the values end up in one of the buckets.
For more details see the class presentation.
If it is not possible to insert a value into the hash table, the insertion method should throw a *runtime_error* exception. This can occur where a bucket is filled with identical search key values and an attempt is made to insert another such value. I would suggest testing for this circumstance when the bucket splits.

**Removal**

Use the hash function to find the directory index. Follow the pointer in the directory to a bucket. Remove any incidences of the key from the bucket. You will probably need to reorganize the contents of the bucket to keep all the empty space in the "top" half of the underlying array. *If the bucket is empty after the removal do not do anything else*. That is, leave the bucket empty rather than deleting the bucket and possibly decreasing the size of the directory.

**Print**

Print the hash table. The directory contents and the buckets it points to should be printed. Each line of the output should show the directory index and pointer value, the latter displayed in hex, followed by the contents of the bucket pointed to and its local depth. Each bucket (its keys and local depth) should only be printed once which means that higher index directory entries that point to the same bucket as a lower index should not show the bucket again; see below for an example.

```
0: 000456D8 --> [64,200,-] (2)
1: 0004AFE0 --> [153,-,-] (3)
2: 000453D0 --> [66,218,-] (2)
3: 00045410 --> [67,-,-] (2)
4: 000456D8 -->
5: 0004B020 --> [13,253,109] (3)
6: 000453D0 -->
7: 00045410 -->
```

This example uses the same keys as the class example from the presentation. The addresses are expected to be different for different executions of the same program.

# Other Project Requirements

There are some specific requirements that you must follow in completing the project.

## Classes

You should only submit four files: *ExtensibleHashTable.h*, *ExtensibleHashTable.cpp*, *Bucket.h* and *Bucket.cpp*. If you implement other classes for your project then define them in those four files – all class definitions should go in the *.h* files and all method implementations in the *.cpp* files. This isn't necessarily the best way to do this but it's the simplest and easiest for marking purposes when I don't know exactly what additional classes you are going to implement.

## Libraries

You should only use standard C++ libraries in your project and should not use any OS or GUI specific libraries. You may use the STL vector class. If you want to use any other STL collection class, you must first email me for permission. Use of exception classes is allowed (and essential in at least one case). If you implement your hash table in Visual Studio I would strongly recommend testing it in Linux using g++.

## Compilation

We will test your projects in Linux and use the g++ compiler with the *-std=C++17* flag to compile them. While you are welcome to develop your solution in whatever environment you prefer you should make sure it can be compiled and run as described above. **Solutions that cannot be compiled will not be marked**. If you are unable to write a method, **write a stub function for it**. If you are unable to complete a method, comment out the non-functional sections of the method so that you receive marks for any test cases the method is able to pass.

## Main Function and Testing

You should not submit a main function. You will obviously need to implement a main function for your testing, *which should be in a separate file which you should not submit*. If you write a main function in one of the files that you do submit then delete it or comment it out before submission.

We will be writing our own main function that tests your hash table's methods.