CSCI319
ASSIGNMENT 2
Due 11:59pm 25/05/2015

The Chord system is a structured peer-to-peer network architecture realizing a scalable distributed hash table. In Chord, the peers are organized as a ring. Positions within the Chord for which there is a computing node are called *index nodes*. Each index node has a finger table, and, in addition, holds any data items associated with its own key value and all data items associated with keys larger than the previous index node in the Chord ring.
To succeed with this programming task, you will need to understand the Chord system very well. Hence, **do not commence the programming task until you have developed a good understanding of the Chord system**. You may need to study the lecture notes, the relevant paragraphs in the text book, and relevant internet sources in order to acquire the level of understanding of CHORD required for this assignment. Commence the work on assignment once you are confident to have understood how it works.
Your task is to simulate the Chord system on a single computer as a single (non-parallelized, not multi-threaded) process. We will keep this task as simple as possible. The main objectives are:

- to understand how the nodes in Chord communicate.
- to understand how nodes are managed (added/removed) from the system.
- to understand the effects of adding and removal of index nodes.
- to understand how the finger table is computed and maintained.
- to understand how data distribution takes place in Chord.
- to understand how searching is done efficiently in Chord.
- to understand the concept of scalable algorithms and fast code.

To keep the task as simple as possible, you are to write the simulation without the creation of additional threads during runtime. Follow the following guidelines:

- Develop data structures which are suitable to hold required information for each peer in a CHORD system. Note that index nodes can hold local information only (i.e, an index nodes never knows of all the other index nodes).
- Remember that your implementation is to simulate a distributed system. Do not make use of any global variables!

On the basis of the data structures developed by you, write a single threaded (single process) simulation in either C or C++ of the Chord system with the following functions (this is the minimum requirement). You may introduce any number of functions in addition to those stated here (Note: Functions which have three dots in their parameter list may accept any number of parameters. For example, the function `Init(n, …)`, this function will accept the parameter n as well as any other parameter that you may wish to pass to this function) :

`Init(n,...)`:
Create and initialize a new Chord system. The parameter n indicates the size of the new CHORD (the size of the hash ring). You can assume that n is a positive power-of-two value not exceeding 2^{31} (thus, n can be any one of the following values: 1,2,4,8,16,32,64,...,2^{31}). Note that in general the size of a Chord is much larger than the number of index nodes. Therefore, n can be a very large value whereas the number of index nodes can remain small. Hence, it is best to ensure memory is only allocated when needed (when adding an index node, when inserting a data item).
This function will also print your name and student number to the screen (print to standard output). Normally, in a real world CHORD, when a new CHORD is created than there will be at least one

index node (the node which created the CHORD will become the first index node in the system), and the new node would assume a random location within CHORD. We will simulate this by creating a new index node at location 0 (zero) within the ring as part of the `Init(.)` function. The size of the finger table (for each of the peers in this new system) is computed based on the parameter n such that $k=(\log_2(n))$, where k is the size of the finger table. Note that this equation is derived from the lecture notes which state that the CHORD is of size $n=2^k$. Thus, for example, if n=32, then k=5, or if n=4096 then k=12.
If `Init(.)` is called when there was a previously existing Chord (i.e. another Chord was generated by an earlier call to `Init(.)`. Then this function will completely remove any pre-existing CHORD from memory then initialize a new Chord system. The removing of a pre-existing CHORD requires the removal of all its associated peers from the system and the release of all allocated memory.
The value of n is obtained from a script file (see example below).

`AddPeer(ID, ...)`:
Adds one new peer (an index node) to your Chord system. The `ID` (an integer value) of the new peer is provided, and can be assumed to be unique. You will need to create a finger table of appropriate size (note that $n=2^k$, where n is the size of the Chord, and k is the size of the finger table) for this new index node. You may also decide to associate a key-list which are in-between the current index and the previous index node as described in the textbook. However, this is optional as it is possible to implement the CHORD without this key-list depending on how smart your implementation is. You may choose either of the two approaches. The value for `ID` is obtained from a script file (see example below). If the addition of the peer was successful then this function prints to the screen (standard output) the following: “PEER <ID> inserted”, where <ID> is the ID of the peer just inserted. Note: The addition of a new peer can require an update of the finger table of some of the other index nodes. This is a very important aspect!

`RemovePeer(ID, ...)`:
Removes a given peer from Chord, associates any data items that this peer held with the appropriate peer in the remaining CHORD, and appropriately updates the finger tables of the remaining peers. This function removes the peer completely, not just removing its finger table. You will need to think about how the removal of an index node affects (the finger table, data items, etc) the other, remaining index nodes. If the removal of a peer was successful then this function prints to the screen “PEER <ID> removed”. If this function causes the removal of the last remaining peer in the Chord, then all data items are lost (memory freed), and the program is to print “This was the last peer. The CHORD is now empty.”

`Find (ID, key, ...)`:
Starting from a peer identified by `ID`, this function searches for the peer responsible for the given hash key. You can assume that `ID` refers to the ID of an existing peer. The search is to be performed by using information provided by the finger tables of the peers visited during the search (as described in the lectures and the text book). Follow the algorithm as described in the book (and illustrated in Figure 5-4) to locate the peer responsible for the key quickly. The function prints to standard output the IDs of the peers visited separated by the '>' symbol and a newline character at the end. For example, if the IDs of the visited peers was 1,7,12 then this function will print to stdout 1>7>12

`Hash (string)`:
Computes a hash key for a given data item. You can assume that the data item is a string. The function returns a key value which is an integer within [0;n). The algorithm used by this function is shown at the end of this assignment.

`Insert (ID, string, ...)`:
Inserts a data item into the distributed hash table (the CHORD) from the node `ID`. The data item is given as a `string` parameter. The node will use the string to compute a hash-key. The string is then to be stored at the index node that is responsible for the computed key value. Note that this means that a peer may have to store more than one data value. Note also that the `Find(.,.)` may need to be called in order to locate the node responsible to hold the data item. This function will print to the screen the string “INSERTING <string> AT: “ follow by the value of the hash key computed.

`Delete (ID, string, ...):`
This function states that the index node `ID` requests the removal of a data item identified by the parameter `string` from the Chord. You can assume that the data item is given as a string parameter. This function uses the hash function to compute the associated key value, then `Find(.)` the peer responsible for the data item, then removes the string from the memory of that peer. If successful, this function will print to the screen: “REMOVED <string> FROM: “ follow by the ID value of the peer from which the string was removed.

`Print(ID, ...)`:
If `ID` refers to an index node in the CHORD, then this function will print "DATA AT INDEX NODE <ID>:", where <ID> is the `ID` value of the node, follow by a newline, then follow by the list of data items stored at this node (each list entry is separated by a newline), followed by the string "FINGER TABLE OF NODE <ID>:", a newline, and the content of the index table of the node. If there is no index node at position <ID> then this will print "NO PEER AT <ID>", followed by a newline.

`Read(filename)`:
Reads a set of instructions from a given file. The instructions are named analogous to the functions which need to be called. For example, a file may contain the following list of instructions:

```
Init 32
AddPeer 7
AddPeer 3
AddPeer 12
AddPeer 20
Insert 0 THIS IS A TEST
Insert 7 SOME TEXT
Insert 7 CSCI319
Print 12
Delete 0 THIS IS A TEST
RemovePeer 12
Print 0
Print 7
Insert 3 HELLO THERE
AddPeer 25
Insert 3 GOOD DAY
```

when compiled, the program should be able to be executed at a command line and accept the file name of a file containing the instructions. Assuming that the compiled code is named CHORD then the program should be able to run by issuing the following command line parameter

```
./CHORD myfile.dat
```

this would execute all instructions within the `myfile.dat`. There is one instruction per line in `myfile.dat`. Instructions not recognized by your program should quietly be ignored (i.e. no error message). Do not assume a maximum file size! In other words, the file may contain arbitrary number of instructions.

Note1: Any of the functions mentioned above may return a value of your choosing. You are free to define these function so that they can return a value if you wish to do so.

Note2: the task is not to implement a full Chord system. There is no need to produce a multi-threaded implementation. The aim is to:

1. Simulate basic concepts of Chord. In particular, a deeper understanding to searching within Chord, and the maintenance of index nodes is to be obtained.
2. Develop an appreciation of mechanisms required for maintaining index tables.
3. Understand the impact of the size of an index table, and the number of index nodes on the efficiency of the chord system.

Note 3: Your code will be compiled and marked using gcc or g++ as available within the Ubuntu virtual machine. Ensure that your code compiles and runs within the Unix/Linux environment. The correctness, functionality, and efficiency of your program will be marked. The code itself is being verified to contain your own work, and that it complies with the given tasks, but is otherwise not marked. We will use our own scripts to test your program. Do not submit your own script file(s).

Note 4: While the code will not be marked, the lecturer will look at the code in order to identify whether the task was addressed correctly. Remember that the task is to simulate Chord. This also means that no single peer has complete information. For example, this means that a search needs to be carried out when inserting or removing a peer from Chord. We will also look at the efficiency of your code.

Note 5: Your code is not to produce outputs other than those specified above. This means that you do not print to `stdout`, `stderr`, or any other output stream any debugging information, or any other information not required by this assignment. A violation to this will attract penalty marks!

Note 6: The correctness and speed (the efficiency) of your implementation will influence the marks that you can earn for this task.

Note 7: We will test your code on the ability to efficiently handle possibly large amounts of peers and possibly large amount of data items. We will test the correctness of the finger tables and the correct storage location of any data item stored in the Chord.

Attachment 1: Algorithm of a hash function for character strings:
Given a character string of arbitrary content and length, then the hash algorithm is as follows:

```
BEGIN Hash (string)
     UNSIGNED INTEGER key = 0;

     FOR_EACH character IN string
          key = ((key << 5) + key) ^ character;
     END FOR_EACH
     RETURN key;
END Hash
```

Note that '<<' is a bit shift operation which shifts bits to the left. Hence, `key<<5` shifts the bits in `key` by 5 positions to the left. Note also that ^ corresponds to the XOR operation, and that "character" corresponds to the ASC-II value of the character, and that the terminating NULL character is not considered part of a string.

Submission:

Use appropriate comments in your source code. You are required to upload the source code to *moodle* on or before the due date.

Assessment:

Marks allocation

Assessment Criteria		Marks
1.	Correctly implement functions : Init (10) AddPeer (10) RemovePeer (10) Find (10) Hash (5) Insert (10) Delete (10) Print (10) Read (5)	80
2.	Scalability & Efficiency	20