New York University Abu Dhabi
CS-UH 3010 - Spring 2018
Programming Assignment 1
Due: February 17th, 2018

Preamble:

The objective of this assignment is to become familiar with the use and programming in the `Linux` *Operating System* environment. In `C/C++`, you will develop an application termed `myapp`, that lets you access information of interest fast. Your application will be based on the *skip-list* that helps rapidly access data in main-memory.

Your program:

- deals with student records. Assume for simplicity that we only handle information about the current semester. A student can take a number of courses depending on her academic progress towards degree requirements.
- creates a *skip-list* main-memory resident structure that will allow a user to access record information based on the *student ID (key).* The *skip-list* structure is *dynamic* as it can either *arbitrarily expand* or *shrink* on–demand. The main-memory space that the application occupies is gracefully released before your program terminates.
- loads information for students either individually (one record at a time) or in-bulk (with the use of a file).
- looks-up information about a specific student and her pertinent information. Look-ups could be done for designated student *groups* as well.
- generates statistics when it comes to the number of comparisons carried out for questions to be realized.
- may delete a record from the structure that has been maintained for an an individual student so far.

Operations including look-ups, insertions/deletions of records derivation of statistics are all launched through a *prompt* that your application should provide.

Procedural Matters:

◇ Your program is to be written in `C++` or `C` and must run on the `NYUAD Linux (CentOS)` Server.
◇ You will have to first submit your project electronically and subsequently, demonstrate your work.
◇ Nabil Rahiman (`nr83-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment.

Project Description:

Figure 1 depicts the key features of the structure to be used in your program. In the upper part of the figure, we present the *skip-list* structure that offers quick look-ups; the structure consists of nodes that are created on the fly as we input records of information with each such record being uniquely identified by the *student-ID.*

In the lower part of Figure 1, we depict the records for all students who have become part of the overall structure for the current semester. Each record consists of a student ID, last name, first name, age, year-of-matriculation, GPA, and number of courses in which the student is enrolled at this time. It is worth pointing out that student records are "hung" from the *skip-list.* For example, Mehmet Iqbal has ID 1497 was initially matriculated in school in 2013, is 22 years old, his GPA is 3.85, and is taking 5 courses. Evidently, dots (that appear in Figure 1 indicate records for specific individuals enrolled during this semester.

How we create and dynamically maintain the *skip-list* structure is discussed in the original research paper by William Pugh [1] that you can also find on the *NYUclasses* portal. We also provide a short description of the *skip-list* operations in *Appendix I.*

[1] `https://dl.acm.org/citation.cfm?doid=78973.78977`

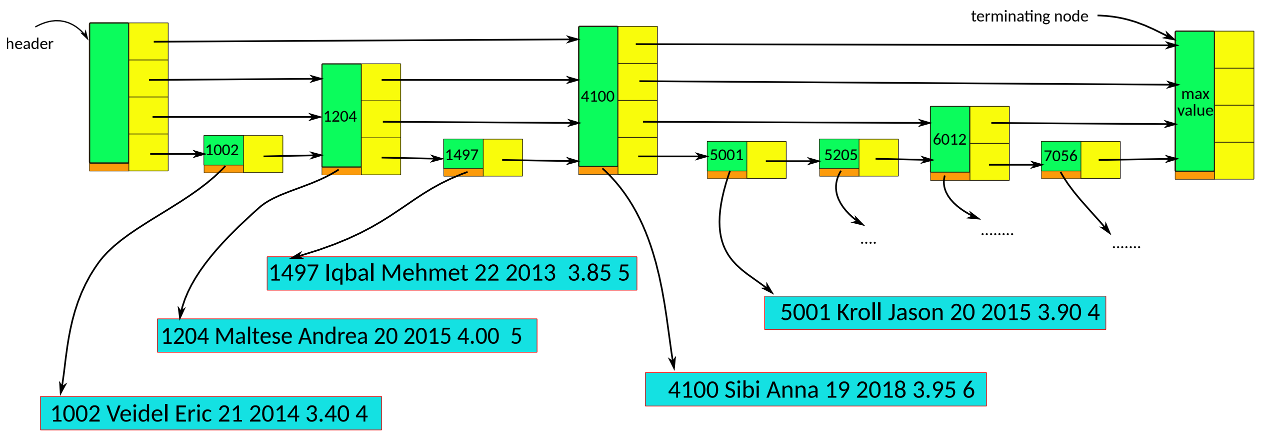
header
terminating node
1002
1204
1497
4100
5001
5205
6012
7056
max value
....
....
....
1497 Iqbal Mehmet 22 2013 3.85 5
1204 Maltese Andrea 20 2015 4.00 5
1002 Veidel Eric 21 2014 3.40 4
5001 Kroll Jason 20 2015 3.90 4
4100 Sibi Anna 19 2018 3.95 6

Figure 1: A sample of the structure that your program should use

How Your Application Should Be Invoked:
Your myapp application can use one or more flag (in-line) options that you may deem required. A possible invocation of your application would be:

```
mymachine-promt >>  ./myapp -m MaxNumOfPointers -v MaxValue
```

where flag `-m` indicates the *maximum number of forwarding pointers* a skip-list node may feature, and flag `-v` provides the *maximum* value the *key* value of the student IDs may take. Evidently, you may opt for using additional flags of your choice in the invocation line of your program. Provided that you either know or can anticipate that your structure will handle N records, the `MaxNumOfPointers` can be set to $\lceil(\log(N))\rceil$.

Once your application gets initialized, it then presents the user with a prompt. Every time, the user types in a command, myapp should reciprocate by taking the required action. Your program should be able to handle the following commands:

1. `ins <studID> <lname> <fname> <age> <myear> <gpa> <numofcourses>`
   inserts all pertinent pieces of information for a specific student whose ID is designated by the value of `<studID>`.

2. `find <studID>`
   lookup the record of whose ID is provided with `<studID>`.

3. `sfind <studID>`
   lookup the record of whose ID is provided with `<studID>` and at the end provide the number of comparisons this operation had to carry out while trying to locate the respective record.

4. `range <studIDa> <studIDb>`
   fetch and display all information for students whose IDs fall in the (numeric) interval `<studIDa>` to `<studIDb>`.

5. `gpa <studID>` or `gpa <studIDa> <studIDb>`
   compute the GPA of a specific student or the average GPA of students in the range `<studIDa>` to `<studIDb>`.

6. `del <studID>`
   delete the record of student with `<studID>`.

7. `print`
   print on the `tty` the entire structure is a format that can be readily understood.

8. `load <file>`
   carry out all `myapp` commands listed one-per-line in `<file>`; this can be used for bulk-loading records into your structure.
9. `exit`
   free all space allocated gracefully and exit the application.
10. Any other command that may facilitate the development of the program.

What you Need to Submit:

1. A directory that contains all your work including source, header, Makefile, a readme file, etc.
2. A short write-up about the design choices you have taken in order to design your program(s); 1-2 pages in ASCII-text would be more than enough.
3. All the above should be submitted in the form of `tar` or `zip` file bearing your name (for instance `AlexDelis-Proj1.tar`).
4. Submit the above tar/zip-ball using *NYUclasses*.

Noteworthy Points:

1. You **have to use *separate compilation*** in the development of your program.
2. Although it is understood that you may exchange ideas on how to make things work and seek advice from fellow students, **sharing of code is *not allowed***.
3. If you use code that is not your own, you will have to provide ***appropriate citation*** (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what and how such pieces of code do.

***Appendix I*: Concise Description of the Skip-List**

The *skip-list* structure is similar to that of a simple linked-list. In contrast to the basic linked-list layout such as that of Figure 2, a skip-list node may have one or more *forwarding pointers.* These forwarding pointers may directly lead to nodes appearing further down the list.

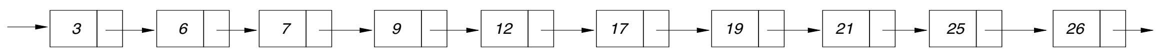

Figure 2: Ordered Linked List

Figure 3 represents a skip-list based on the `int` keys that are the same with those used in Figure 2. As this new structure involves additional lists that *bypass* intermediate nodes, it was named *skip-list.*

Every node has always a maximum number (`MaxLevel`) of forwarding pointers that may not all find themselves in use. The specific number of forwarding pointers each node actually features, is randomly determined by the insertion algorithm of the structure. The only nodes excepted from this last rule are the very first (`header`) and very last (`terminating`) nodes. The header always features `MaxLevel` forwarding pointers that point either to the `terminating` or some intermediate node. The forwarding pointers of the terminating node are all `NULL`.

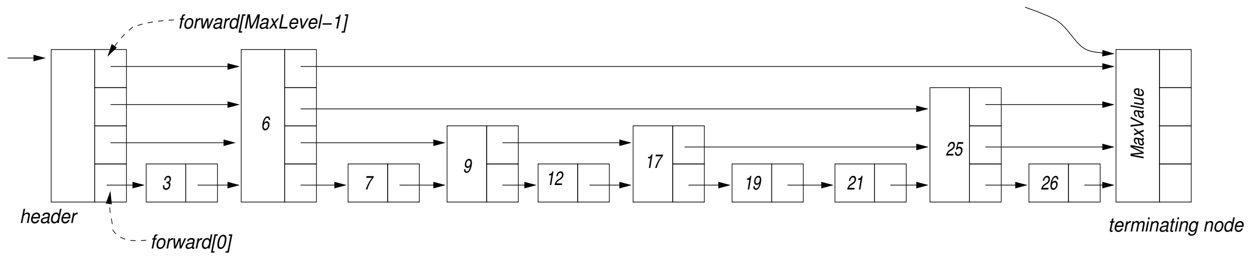

Figure 3: Skip-List

Basic Concepts

1. Every skip-list node must have a *key* field that uniquely identifies the node on the entire structure. Each node also features a pointer to the record of information we want to store as well as `MaxLevel` forwarding pointers. Fig. 3 does not show the stored records for brevity; here, we only depict the forwarding pointers used to transition to subsequent nodes.
2. The header node has `MaxLevel` of forwarding pointers that correspond to levels: 0..`MaxLevel-1`
3. When a new key (and record) is being inserted, we randomly select the level of forwarding pointers with which this key will become part of the structure. A node set to operate at level i has i forwarding pointers that direct towards subsequent nodes on the list. The remaining `MaxLevel-`i pointers either point into the last node or to `NULL` as they are never visited!

Initialization of the Skip-List

Every node can be create by using a structure of the following type:

```
struct node {
     int key;
     record *ptr;
     struct node* forwarding[MaxLevel];
     }
```

The skip-list of your program will be initialized as soon as `myapp` gets executed. The inline parameter `MaxNumOfPointers` designates the value that `MaxLevel` actually takes and `MaxValue` is respectively a very large value that no actual key from those used can have.

Initially, the skip-list features only two nodes: the header and the terminating node. The header is essentially used to provide access to forwarding pointers. The key value of the header has no practical significance as we are never concerned about this.

The terminating node maintains a key whose value is `MaxValue`. As mentioned earlier, `MaxValue` is a value that is greater than all the key values used by our records and gets designated at the structure initialization time. The forwarding pointers of the terminating node all point to `NULL`. Figure 4 depicts how the skip-list is initially set up.

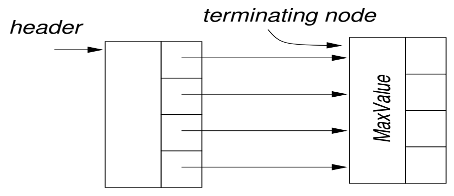

Figure 4: Empty Skip-list

The Search Algorithm

The search method returns the pointer to the node that has a specific *key* (or returns nothing at all if no such key is found on the structure). Algorithm 1 outlines the look-up process. The search commences from level `MaxLevel-1`. When the looked up value cannot be found at this level, then the search moves to level below. This process continues until we reach level 0. The x variable used by Algorithm 1 serves as a pointer to a `struct` that designates a node.

Algorithm 1 $Search(header, searchKey)$

```
1: $x := header$
2: for $i = MaxLevel - 1$ downto 0 do
3:    while $x \to forward[i] \to key < searchKey$ do
4:       $x := x \to forward[i]$
5:    end while
6: end for
7: $x := x \to forward[0]$
8: if $x \to key = searchKey$ then
9:    return $x \to value$
10: else
11:    return failure
12: end if
```

Figure 5 shows the search path we have to follow in order to locate the node with key=21.

Insertion Algorithm

The insertion algorithm first identifies the correct position in which a key has to be inserted[2] and inserts a new node with the key in question (if of course this key is not already part of the skip-list). Algorithm 2

[2]along with the pertinent record

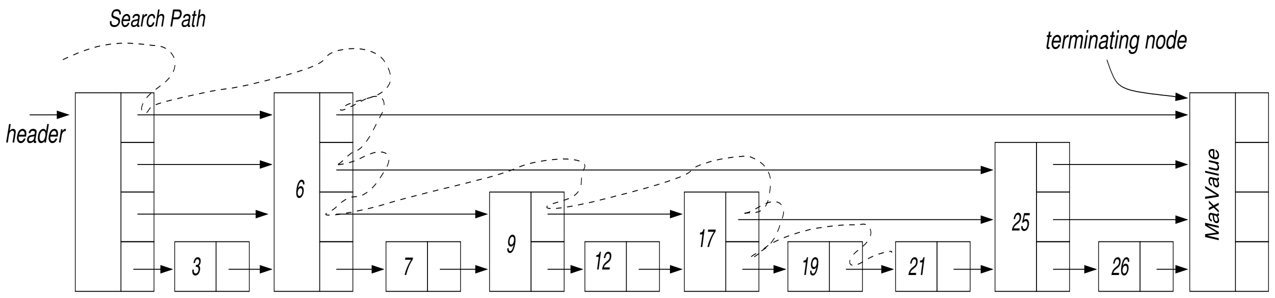

Figure 5: Query path followed while looking for key 21

outlines the insertion for a key while Figure 6 offers an example in which key 17 (and the corresponding node) is inserted.

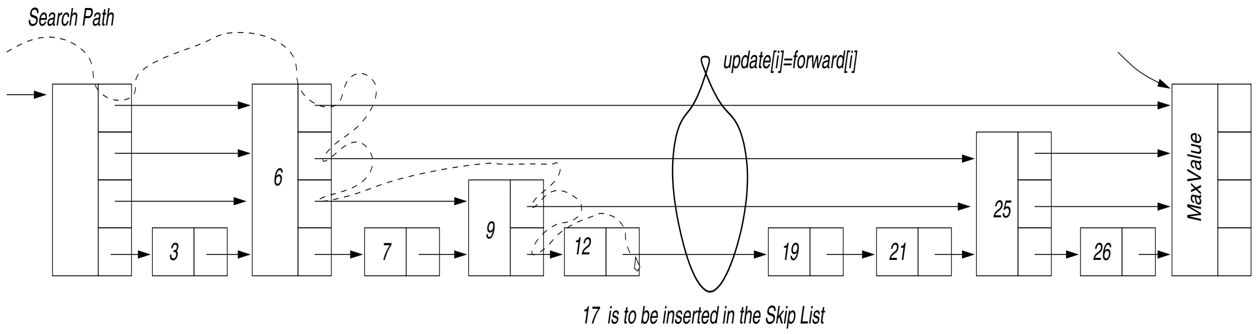

Figure 6: Inserting key 17

The level at which an insertion actually occurs is defined by the function `randomLevel()` that is a random number generator; you can implement this function using the library calls `srand()/rand()` [3]. Please note that this random number generator should create values in the range 0..`MaxLevel`. In Algorithm 2, the new node is created through the invocation of the function `makeNode(lvl, searchKey, value)` where `lvl` is the adopted level of the node, `searchKey` is the key and `value` is the record of information to be inserted. At this stage, all forwarding pointers of the newly created node are set to point to the terminating node.

In its last stage, Algorithm 2 executes a `for-loop` that is responsible for the "matching" of the new node in the list. The local variable `update` used by the algorithm is a 1-dimension table that has `MaxLevel` pointers to node structures.

Deletion Algorithm

Algorithm 3 deletes the record (data) and skip-list node containing the *search key*. Function `free()` sets pertinent memory unused by the node under deletion free.

[3]do `man rand` to find out more.

Algorithm 2 $Insert(header, searchKey, newValue)$

```
local update[0..MaxLevel-1]
$x := header$
for $i = MaxLevel - 1$ downto 0 do
    while $x \to forward[i] \to key < searchKey$ do
        $x := x \to forward[i]$
    end while
    $update[i] := x$
end for
$x := x \to forward[0]$
if $x \to key = searchKey$ then
    $x \to value := newValue$
else
    $lvl := randomLevel()$
    $x := makeNode(lvl, searchKey, value)$
    for $i = 0$ to $lvl$ do
        $x \to forward[i] := update[i] \to forward[i]$
        $update[i] \to forward[i] := x$
    end for
end if
```

Algorithm 3 $Delete(header, searchKey)$

```
local update[0..MaxLevel-1]
$x := header$
for $i = MaxLevel - 1$ downto 0 do
    while $x \to forward[i] \to key < searchKey$ do
        $x := x \to forward[i]$
    end while
    $update[i] := x$
end for
$x := x \to forward[0]$
if $x \to key = searchKey$ then
    for $i = 0$ to $MaxLevel - 1$ do
        if $update[i] \to forward[i] \neq x$ then
            $break$
        end if
        $update[i] \to forward[i] := x \to forward[i]$
    end for
    $free(x)$
end if
```