**Final Exam**

BBL 553E – Algorithms Engineering

*Valentin L’HOMEL – 922110009*

For the final exam of the Algorithm Engineering courses, we were asked to devise and implement a Skip List structure using all the knowledges we’ve acquired during the courses. In the following paragraphs, I will describe my own work and the results of it. In that end, I will first describe the settings of the hardware used to test the Skip List, then I will briefly describe the general behavior of a Skip List, as well as which tests did I ran in order to check its functionality and performance. I will also explain how to run these tests by yourself if you are interested. Then, I will present how have I design my Skip List, through which interrogations did I go, as well as some failure I have encountered. Lastly, I will compare my implementation of a Skip List with another one found on Github.

Considering the computer that I used both to create and test my algorithm, it is a ThinkPad T480, Type 20L6, produced by Lenovo, which is currently 3 years old. It Is equipped of a processor Intel Core i7 with 1.8 GHz and 8 GB of RAM. This computer is running under Windows 10 Pro N version 21H1. Programs have been compiled and tested under two different scenarios. First with Microsoft Visual C++ (MSVC) from the Visual Studio Build Tools 2019 version 16.8.2, and second, using GCC 9.3.0 under a Windows Subsystem for Linux (WSL) of Ubuntu 20.04.3.

Let’s now dive into what is a Skip List. A Skip List is a probabilistic data structure composed of several linked list. Each linked list holds more data than the previous ones, with the last one being a complete linked list of all the elements on it. Data on a Skip List are ordered while inserting, allowing for a quicker search of a given elements, following the strategy of a binary tree. Each element inside a Skip List will be part of a random number linked lists, usually defined as a probability of 1/2 for each linked list. A schematic picture of a typical Skip List is available at the end of the page.

Graphical user interface, application

Description automatically generatedAs far as the tests are concerned, I built two different test files, both focusing on different points. The first one, named testCorrectness.c, is here to test the efficacy of the algorithm. It will first try to create a Skip List, then insert some given numbers on it, then perform some search, before trying to remove some other elements, and finally it deletes the structure. The fact that I’m precising “some” numbers is because these can be freely changed by editing the testData.h file. At each step of insertion and deletion, the content of the structure is displayed to ensure the correct behavior of it, plus the whole structure and its different linked lists are displayed after all the insertions. An example of the expected output is shown on the next page.

Basic structure of a Skip List, with its different linked lists

<https://en.wikipedia.org/wiki/Skip_list#/media/File:Skip_list.svg>

Text

Description automatically generatedText

Description automatically generatedThe second test file, named testSpeed.c, will create a new Skip List and will insert a given number of elements, usually high, into the Skip List, then search and remove them all, one at a time. This operation is repeated three times, with three different sequences. For the first iteration, the elements used are already sorted in an ascending order, and in descending order on the second iteration. The last one is perform using elements in a random ordering. For each batch of operations, the time required to perform them all is displayed, as well as the list size after the insertions and deletions, and the number of removal operations which may have failed. Please, note that the time rendered is obtained using the clock function, which is dependent of your own CLOCKS\_PER\_SEC. Therefore, the results shown here may be different from those someone else may acquire.

Any tests which have been performed can be reiterate by anyone possessing the files I mentioned above, as well as the Skip List library or source code necessary. Codes have been written in plain C, and compiled using CMake. The CMakeLists.txt provides with the source code shall allow anyone to compile and run it. The source code provides not less than six somehow different implementation of the Skip List structure. They all share the same function and structure name, which are available on the SkipList.h file. The actual structure and underlying organization, however, change from one implementation to another. I will explain each of them later. In order to compile the source code, the following commands should be entered on a terminal:

cmake -B [build directory] -S [source directory]

cmake --build [build directory] --config Release

Where [build directory] and [source directory] should be replaced respectively by the desired directory where the metadata will be placed, and the directory containing the CMakeLists.txt. These two commands will generate the two executables describe previously, testCorrectness and testSpeed, on the source directory. Moreover, the six libraries implementing the Skip List should be available under the build directory on a Linux system, or under a /Release/ subdirectory of the build directory on a Windows system.

Additionally, the first command can accept the following arguments to change the build behavior:

-D SKIPLIST={ External, Basic, Cache, SIMD, Second }

-D RANDOM\_SEED=[seed]

-D NUMBER\_COUNT=[nb]

SKIPLIST will change the library used on the test. When left empty, the default, recommended version is used. RANDOM\_SEED will set the seed of the random generator to the given one. By default, the seed 20221202 is used. NUMBER\_COUNT will set the number of elements used by the testSpeed executable. The default value is one million.

## **External Library**

The external library ExternalSkipList is build using the ExternalSkipList.c code source. It is not my own work, but that of Troy Deck, available on Github (<https://github.com/tdeck/c-skiplist>). I slightly altered it to adhere with the definitions of my header file. These changes can be resumed at: removing a call to srand at the initialization of the list, changing the name of the functions, changing the key-value pair to a generic element and its comparison function, and adding the size and dump functions.

Its main particularities are a fixed number of linked lists, set at 8, and the uniqueness of each element.

## **Basic Library**

Text

Description automatically generatedThe basic library BasicSkipList is my first implementation of the Skip List, present at BasicSkipList.c. All other implementations are derived from it. The SkipList structure holds only two pointers, the head node and the comparison function. Each node consists of two pointers and an int, holding the stored value, a list of pointers to other nodes, creating the various linked lists, and the number of pointers in that list. Every node has at least one pointer on its list, and the linked list created using the first level is the full list. The alternative of holding a NULL pointer in the list instead of the size have been considered but deemed costlier in term of space and use. Each node is created using two calls to malloc, a first one to create the node, and a second one to create the list. The first node, i.e. the head, is created with a level of 1.

The insert function will, if required, enlarge the head node list by calling realloc, before looking for its own position. The number of linked lists the new node is part, later called the height, is obtained by looking at the parity of successive calls to rand. The new node will always be placed after other elements which may compared equal to itself, thus ignoring the uniqueness of elements.

The search function navigates through the different linked lists, from the highest one, the sparsest one, to the last one, the full list. It moves forward while the node is compared less than the desired one and go to the next linked list when it is no longer true. When on the last list, the next element’s value is return if this element is valid and compared as equal. It would be possible to stop sooner by looking if one element compared as equal in addition to the less comparison, however this would greatly increase the number of comparisons at each node. While our method may include up to the height number of comparison, this alternative would double the number of comparisons, which is an operation quite costly.

The remove function behaves exactly like the search function, except that it in additionally store the terminal node of each linked list.

## **Standard library**

The standard library SkipList is an improved version of the previous one, built with SkipList.c. Through minor algorithm and code optimization, as well as SIMD instructions, it manages to be 50% more efficient than the basic version.

First, the random function is replaced to a single call to rand, and the number of unset bits at the end of the random number gives us the height. Using the tzcnt SIMD instruction allow us to slightly reduce the time requires to get this number. Considering the number of bits on an int is counted, the maximum height possible is now 32. Therefore, the height variable has been replaced by a byte, which may reduce the space. However, due to the padding on the structure, the size does not in reality change.

A second optimization lie on the space of the structure. Instead of allocating two elements for a node, and therefore hazarding a discontinuity on the return memory, as well as a cache miss, a single block is allocated and shared between the node and its list. The head node is also initially created with the maximal size, that is 32 of height, thus avoiding any need to realloc, and another memory discontinuity.

Another minor code optimization lies on the reduction of arithmetic operations. While we initially iterated through the height until it reaches 0, and subtracting 1 to it to access the pointer on the list, we now iterate from height-1 until -1.

Another change made consisted of iterating through the nodes until a value greater than the desired one is found, instead of greater or equal. This may, on lists with few to no duplicate, allow us to more quickly stop the search. Because we initially stopped at the previous node, we actually traversed several lower nodes despite having found our desired node. Accepting the nodes which compared equal effectively disables this unnecessary traversal.

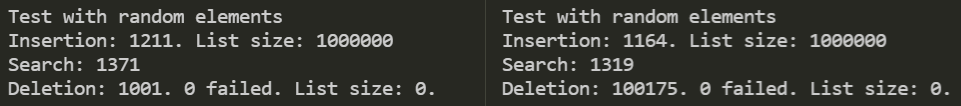
## **Second library**

The second library SecondSkipList, compiled from SkipList2.c, is actually very similar to the standard library. The only difference is on the internal linked list structure. Instead of maintaining one linked list per level, it creates several linked lists per level, ending whenever a node which compared greater is at a higher level. A schematic comparison is shown below for a better understanding. As a result, each node is guaranteed to be pointed by only one node, while a node may point to any number of nodes, or none. This effectively allow a few less comparisons, as every node will be compared only once.

Chart, box and whisker chart

Description automatically generated

Regular Skip List on the left, compared to the one from the second library

However, while this has been proven as efficient as the previous library on WSL, for some unknown reason, the removal method is slower with lesser nodes on Windows. For example, on a 1,000,000 list, it can very efficiently remove the first 700,000 nodes, but the last 300,000 are slower and slower as less nodes are present. Here is the comparison with the standard and second library.

## **Cache library**

The cache library SkipListCache, built with SkipListCache.c, was an attempt to reduce the cache miss while traversing through the list. The main idea was to keep relatively close on memory the nodes which will very likely be visited next, that is the next node with the same height. On that end, I review both the node structure and the way new nodes are created.

Text

Description automatically generatedWhen creating a new node, I actually allocate a block of memory large enough to holds ELEM\_PER\_NODE nodes at once, where ELEM\_PER\_NODE can be any desired value. Then I assign an index to each node on the block. Considering the node structure held some padding byte, I could add this index without altering the size of the node.

When inserting a new node into the list, I looked at the previous node on the same level, and then I looked whether the block where lied this node has a free space. If it does, I shift the other nodes of the block to the right on the memory and insert my new node on the created space. If there isn’t, then I allocate a new block, put the last node of the block on it, and once again shift and insert my node.

A screenshot of a computer

Description automatically generated with medium confidenceWhen removing, the opposite operation is performed. The nodes are shifted to the left, and the whole block is free when there is no more node on it.

In order to implement this, I used the same linked list structure as on the second library. As a matter of fact, I designed the cache library first, so it is the cache library that has been a reference to the second library. This is also why the second library does not holds a prev elements on its structure, contrarily to the cache. Thanks to that organization, it was easier to assemble the block: a new block is created whenever a node diverges in height from its predecessor or, as mentioned earlier, when the block is fully loaded.

## **SIMD library**

The final library is the SIMD library, SkipListSIMD, built with SkipListSIMD.c. The main idea behind this library was to reduce the time spent during comparison by doing several of them at once using SIMD instructions.

A picture containing shoji, crossword puzzle

Description automatically generatedThe comparison that I tried to parallelize were with the two next nodes with the same height than the current node, as well as the next node one level lower on the current node and the same on the next node. After calling the comparison function on these four nodes, I looked at which nodes first compared less than or equal to the desired result. That for, I used first the cmplt SIMD instructions, which gives 0 when the value is greater or equal, and -1, i.e. all bits set, otherwise. Then I used cmpestri to get the number of 0 on the result. Using that, I have been able to move to the desired node, or immediately decrease the height by up to two levels.

However, this happened to be very inefficient, so I stop its development early. For that reason, the current version is bugged, and I don’t plan to fix it.

# **Results**

The tests described earlier have been executed with each library, with NUMBER\_COUNT set at 100,000 to begin, then 1,000,000, 4,000,000 and 10,000,000. The results given in the below table have been obtained using the version compiled with MSVC. As mentioned, the value displayed is the return of the clock function, which is equal on this case to 1e-3 seconds. The same tests have been run 3 times, and only the best time in each case is displayed. Only the results for random numbers are displayed, with the first one being the insertion time, then the search time, and lastly the removal time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | External | Basic | Standard | Second | Cache | SIMD |
| 100,000 | 176  326  165 | 74  64  53 | 57  51  45 | 64  62  66 | 69  63  73 |  |
| 1,000,000 | 48,027  107,331  45,825 | 1,635  1,927  1,398 | 1,129  1,349  986 | 1,144  1,360  92,744 | 1,094  1,268  1,210 |  |
| 4,000,000 |  | 10,415  12,613  9,381 | 7,232  8,514  6,392 | 6,968  8,213 | 6,668  7,810  7,269 |  |
| 10,000,000 |  | 36,441  48,513  33,459 | 25,138  33,071  22,861 | 24,423  31,802 | 23,274  30,315  26,127 |  |

The following points should be considered while reading this table:

* the External library does not keep duplicate. As a matter of fact, the results given above concerned lists of size 99,943 and 994,749.
* the Cache library use blocks of 2 nodes.
* the value which are not included were not calculated due to their aberrant time, except for SIMD.
* the SIMD library does not hold any value because, as mentioned above, it is still incomplete and fail to do the operations.

From this table, we can see that the external library is less than efficient, on every operation. We can also see that the Standard, Second and Cache library are about 1.5 times more efficient than the Basic implementation.

I will also include the following table, which contains the allocated space, in bytes, used by each program during a run with 100,000 elements. Note that this is the output of the valgrind program, ran with the version compiled under WSL.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| External | Basic | Standard | Second | Cache | SIMD |
| 8,803,920 | 4,003,072 | 4,005,408 | 4,005,408 | 6,441,520 |  |

As we could expect, the Cache version use way more memory than the standard version, however it manages to be slightly faster than its compatriots.