Theme Algorithm Analysis

Dinu Ion-Irinel

Polytechnic University of Bucharest Faculty of Automation and Computers, 325CA
ion.irinel.dinu@stud.acs.upb.ro

Abstract. Comparison between two data structures that can be used to implement a priority queue (AVL and Heap).

Keywords: AVL · Heap · Data structures

1 Introduction

1.1 Description of the problem solved

A priority queue is a collection in which each item is associated with a priority that determines the order in which the item is to be inserted. The priority element will be placed at the top of the queue and this will also be the first extract from the queue. In the chosen issue, it is necessary to implement operations specific to such a priority queue such as adding a new item, deleting and obtaining the item with maximum value.

1.2 Practical applications for the chosen problem

A practical application that is based on a priority queue is CPU Scheduling, an algorithm that is used in operating systems. It handles process scheduling according to priority. If two processes with the same priority are ready for launch, the algorithm is used to design *FIFO* (First in First Out) and executes it on first come. In this type of scheduling algorithm, if it arrives, a newer process that has a higher priority than the running process , ie, the current process is running, ie is prohibited. Also, priority queues are used in the development of algorithms such as:

- Graph-specific algorithms (Dijkstra's Shortest Path)
- Data compression (Huffman Codes)
- Artificial Intelligence (A * Search Algorithm)
- Bandwidth management

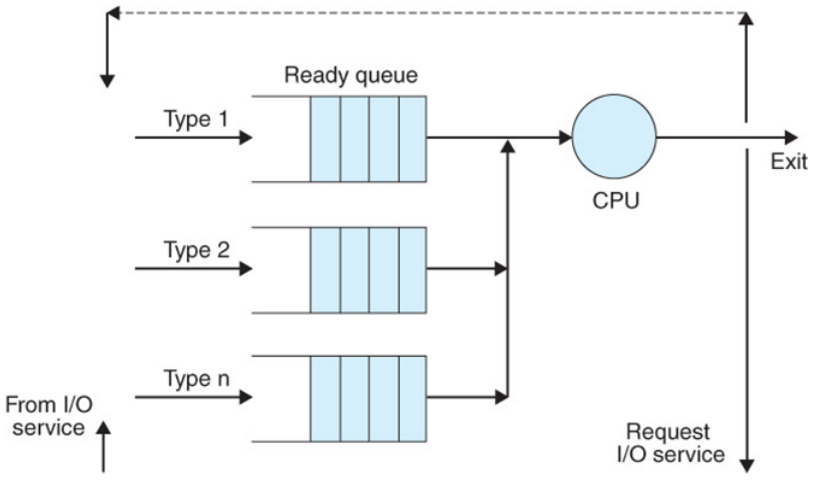

FIG. 1.An example of Multi-level Priority Queue CPU Scheduling [2]

1.3 Specifying the chosen solutions

In solving the problem, we decided to analyze two data structures through which a priority queue can be implemented: the AVL and a Max Heap.
The two data structures have the following properties:

- Both are types of binary trees
- Insert and delete operations have logarithmic complexity
- Both can perform an effective search for the maximum value item. In a Max Heap the maximum element will be in the root of the heap, and in an AVL the element with maximum value will be in the last node in the right subtree.

Because the time for a search is a very important factor in the problem, the probability of finding the desired item with the maximum value will increase with the execution time of the program.

1.4 Evaluation criteria for the proposed solution

The evaluation criteria for the proposed solution will be represented both by the efficiency of each algorithm in terms of execution time and in terms of specific complexity. In order to evaluate the correctness of the implemented algorithms, I set out to perform a series of tests, as follows:

- Tests [1-7] - will check a small number of basic operations:
  - insert / obtain maximum element
  - insert / delete / get maximum item
- Tests [8-15] - will introduce a larger set of operations and will focus more on the operation of obtaining the maximum element:
  - insert / obtain maximum element
  - insert / delete / get maximum item
  - between 0 - 10000 items will be inserted in the queue
  - the insert elements will have values between 0 - 10000
- Tests [16-23] - will be generated automatically and will test in a complex way the implemented operations:
  - 0 - 10000000 items will be queued
  - inserted elements will have values between 0 - 1000000

The first basic tests will be done manually, and for more complex tests with a large number of operations I will make a program that will generate the tests automatically. Also to compare the results obtained with those that are correct I will use a checker that checks if the two outputs are identical.

To test the efficiency of each algorithm I will analyze for each test the number of operations involved and the execution time in which the desired result will be returned.

2 Presentation of the chosen solutions

2.1 Description of how the chosen algorithms work

Heap

A Heap is a data structure, a binary tree in which the value present in the root is greater than the value of all children. This property is respected recursively for each subtree in the heap, practically each node has a value less than or equal to the current value of the parent.

From the point of view of implementation, a heap can be made both using chains of structures, connected by left and right pointers, this method having the advantage of the flexibility of resizing the shaft, and by using a vector and accessing nodes via an index. Implementing a heap using such a vector is more popular and efficient, having the advantage of removing pointer information, using a simple formula to determine each node in the heap.

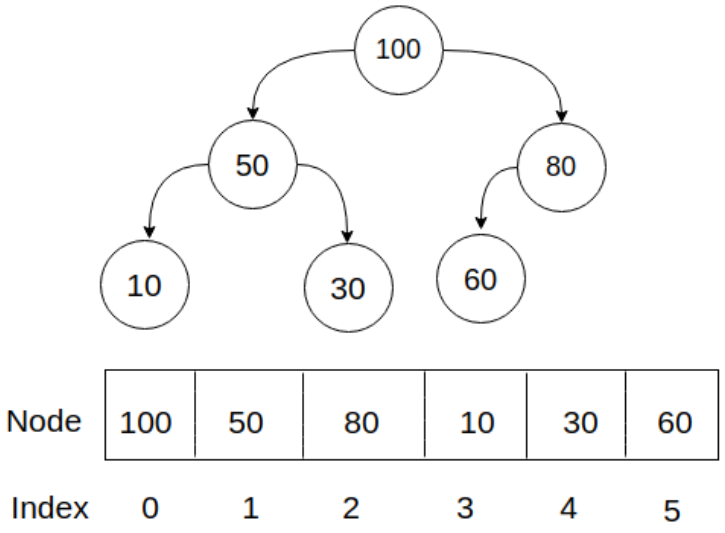

FIG. 2.An example of implementing a Max Heap using vector

The linear implementation of the heap using a vector becomes:

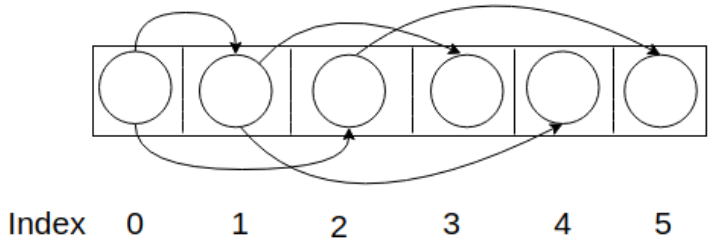

FIG. 3.Determination of nodes based on the index calculation formula

The indexing within the vector is done starting with 0, which represents the position of the root and then for each node the position of the descendants and the parent is calculated according to the following formula:

- Left node = 2 * i + 1, where i is the index of the current node
- Right Node = 2 * i + 2, where i is the index of the current node
- Parent = (i - 1) / 2, where i is the index of the current node

Max Heap specific operations

During the implementation of the operations of inserting, deleting, accessing an element from Max Heap, two operations are necessary through which to maintain the ownership of Max Heap:

Heapify Up

The operation exchanges the value added at the end of the vector or modified, with the higher values, going to the root, in order to maintain the desired heap property.

```
heapify_up
   determining the parent of the node
   parent_node = parent(index)
   if array[parent_node] < array[index]
   exchange the array[parent_node], array[index]
   call heapify_up with parent_node
```

Heapify Down

The operation involves the exchange of elements following the modification of a node in the heap, the new value being less than the value of the descendants, the exchange being made to the lower levels of the heap to maintain the desired heap property.

```
heapify_down
   access the left and right children of the current node
   index of great_node is index
   if right < heap_size and array[right] > array[index]
       great_node is right

   if left < heap_size and array[left] > array[great_node]
       great_node is left

   exchange array[great_node] with array[index]
   call heapify_down with great_node
```

Add

Through this operation, a new element is added to the heap.

```
add(new value)
   add the new item to the last position
   array.push_back(value);
   index_size is array.size - 1
   call heapify_up with index_size
```

Remove Max Element

This operation removes the item with the maximum value from the heap.

```
remove_max()
   if heap is not empty
       exchange array[0] with array[size - 1]
       remove last element of the vector
       call heapify_down with index 0 for first element
```

Max Element
By means of this operation the element with the maximum value is returned.

```
max_element()
   if the heap is not empty
   return array[0]
   return -1
```

AVL

The AVL is a data structure, a binary search tree, which is balanced after each operation of deleting or inserting a node. It works by the property that the difference between two subtrees of a node always has a maximum value of 1, and the rebalancing is done using double rotations.

From the implementation point of view, for an AVL are used structures or classes of pointers that contain: a pointer to the left node, a pointer to the right node, the actual value of the node and its height.

AVL-specific operations
In order to perform operations of deleting, inserting or extracting an element, a series of rotation operations are required, through which the shaft is rebalanced.

Rotate Right - if the equilibrium factor is positive and if the height of the left subtree is higher than that of the right subtree, a clockwise rotation is performed.
Left Right Rotate - if the equilibrium factor is positive and if the height of the left subtree is less than that of the right subtree, a left rotation is performed and then a right rotation.
Rotate Left - if the equilibrium factor is negative and if the height of the left subtree is less than that of the right subtree, a left rotation is performed.

Right Left Rotate - if the equilibrium factor is negative and if the height of the left subtree is higher than that of the right subtree, a right rotation is performed and then a left rotation.

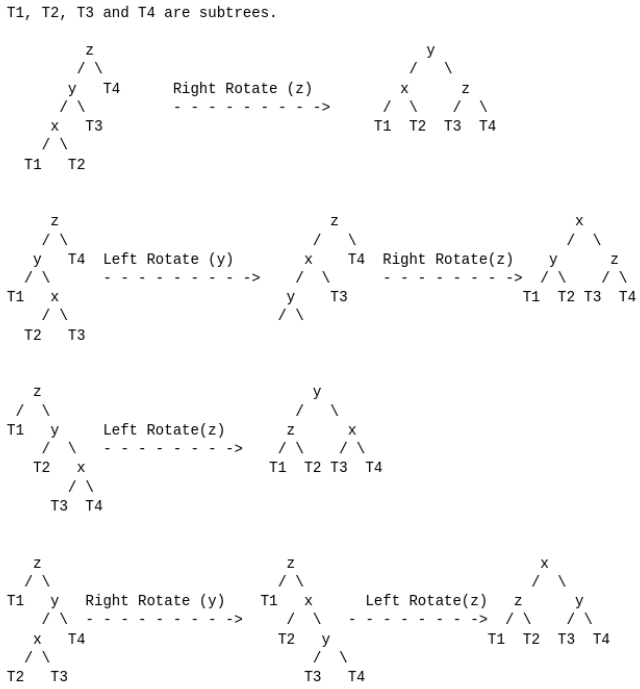

FIG. 4.Example of rotation types used for AVL rebalancing [8]

Add

Inserting a node into the AVL is similar to inserting it into a binary search tree. Search for the insertion position by comparing the key of the current node with the key of the node to be inserted, then choose the left or right subtree. After adding the element, the avl is balanced by updating the heights and applying the rotations.

```
add(nod , key)

   if node == NULL
       node = create_node(key)

   else if node.key > key
       insert (node.left, key)

   else
       insert (node.right, key)

   nod.height = 1 + maximum(height(node.left), height(node.right))
   apply rotations if necessary
```

Delete
The operation of deleting an element is similar to deleting an element from a binary search tree. We are looking for the node with the key we want to delete. If the node does not have two successors, if it is a leaf, it is replaced with NULL and if it is not a leaf, it is replaced with one of its non-null descendants. Otherwise, if the node has two successors, we will replace it with the smallest node in the right subtree. Finally, the AVL is rebalanced.

```
remove (node, key)
  if node == NULL
      return NULL

  if node.key < key
       remode (node.right, key)
  else if node.key > key
      remove (node.left, key)
  else
      if node.right == NULL
          node = node.left
      if node.left == NULL
          node = node.right
      else
          min_node_right = min_node(node->right)
          node.key = min_node_right.key;
          remove (node.right, min_node_right.key)

  nod.height = 1 + maximum(height(node.left), height(node.right))
  apply rotations if necessary
```

Delete Max Element
During this operation, the element with maximum value is deleted. It is located in the last node of the AVL's right subtree.

```
remove_max()
  obtaining and deleting the maximum element
  node = max_node(root);
  remove(root, node.data);
```

Max Element
In this operation, the node containing the maximum element is returned, this being the last node in the right subtree.

```
max_node(root)
  current = root;
  while current.right != NULL
      current = current.right;
  return current;
```

2.2 Analysis of the complexity of the solutions

Heap complexity analysis
1. Max Heap Insertion Complexity Analysis
- the complexity of adding a new node is: O (1)
- complexity of exchanges (heapify up operation): O (h)
- total complexity: O (1) + O (h) = O (h)
- in a complete binary tree, the height is equal to log N so the general complexity of inserting a node into a Max Heap is O (log N).

2. Delete Complexity Analysis in Max Heap
- complexity of leaf node exchange with parent node: O (1)
- complexity of deleting the end element: O (1)
- complexity of exchanges (heapify down operation): O (h)
- the general complexity of deleting a node from Max Heap is O (log N).

3. Analysis of the accessibility of the element with maximum value
- in order to obtain maximum value, the element from the heap root is returned, and this operation has general complexity O (1).

AVL complexity analysis
1. Analysis of the insertion complexity in AVL
- inserting a new element in the shaft requires rotations, calculating the balance factor and updating the height
- Complexity of rotation operations: O (1)
- complexity of height update: O (1)
- complexity of traversing the shaft height: O (log N)
- the general complexity of adding a node is O (log N).

2. Analysis of the wiping complexity in AVL
- in a similar way the operation of deleting a node requires rotations, calculating the equilibrium factor and updating the height after deletion
- both in the most unfavorable case and in the most favorable one, the general complexity of deleting a node is O (log N).

3. Analysis of the accessibility of the element with maximum value
- in order to determine the element with maximum value, the right subtree of the AVL had to be traversed
- in the most unfavorable case, when the node with the maximum element is the leaf, the access complexity of the maximum element is O (log N).
- in a favorable case, in which the element with the maximum value is right in the root, the complexity is O (1).
- on average, the general complexity of accessing the element with maximum value is O (log N).
In the worst case scenario, the height of an AVL is 1.44 log N.

This formula is deduced as follows:
Let Fh be a tall AVL tree, ime h, having the minimum number of nodes. Let Fl s , i Without AVL trees which are the left subtree s, and, respectively, the subtree of right of Fh. Then Fl or Fr must be high because Fl is , imea h-2. To suppose high , imea h-1, so that Fr is high, imea h-2. Ret , we know that Fr must be an AVL tree with the minimum number of nodes in all , and AVL trees with înalt, imea h-1. Similarly, Fr will have the minimum number of nodes in all , and tall AVL trees, ime h – 2. These trees are called Fibonacci trees.
/ *F* /= / *Fh* - 1 / + / *Fh* - 2 / +1 /
*F*0 /= 1 / *F*1 /= 2.
/ *Fh* / + 1 = (/ *Fh* - 1 / + 1) + (/ *Fh* - 2 / + 1)
(??)*h*+3
/ *Fh* / +1✓ $\frac{1}{\sqrt{5}}\frac{1+\sqrt{5}}{2}$
h ?? 1.44log Fn
The rarest possible AVL tree with n nodes is tall, ime h 1.44log N The worst case of înalt, The size of an AVL tree with n nodes is 1.44log N

MAX HEAP	Worst Case	Average Case
Space	O(N)	O(N)
Search Max Element	O(1)	O(1)
Insert	O(log N)	O(log N)
Delete	O(log N)	O(log N)

AVL	Worst Case	Average Case
Space	O(N)	O(N)
Search Max Element	O(log N)	O(log N)
Insert	O(log N)	O(log N)
Delete	O(log N)	O(log N)

FIG. 5.Table with the complexities of the two algorithms

2.3 Presentation of the main advantages and disadvantages for the solutions considered

The advantages of the Heap
- Implements the operations of insertion and deletion in logarithmic complexity
- Can be used to efficiently find the smallest / largest element in the heap
- Heap is always a balanced tree (complete)
- Due to the speed of operations, Heap is the basis for the implementation of sorting algorithms such as HeapSort which has a complexity O (N * log N)

Disadvantages of the Heap
A disadvantage of the Heap is the operation of searching for an element, which has a linear complexity O (n).

The advantages of AVL
- the main advantage of AVL is that it produces the most balanced tree in the worst case
- being perfectly balanced, the AVL shaft will take a very fast time to search for an element

Disadvantages of AVL
- a major disadvantage of AVL is the large number of rotations it performs for rebalancing
- the implementation of operations and rotations specific to an AVL is complicated

In general, the two algorithms have the advantages and disadvantages mentioned above, but restricting the case study to the approached problem, the representation of the Heap in the form of Max Heap has the great advantage of implementing the operation of accessing the element with maximum value in constant complexity. 1) compared to AVL, for which the same operation has O complexity (log N). In terms of implementation, AVL has a disadvantage, because its implementation is more difficult than that of a Max Heap using an STL vector. As a consequence, PriorityQueue in Java is implemented using a Max Heap.

3 Evaluation

3.1 Description of how to build the test set used for validation.

For each test I created an input file that contains a number of operations specific to a priority queue to be tested and an output file with which I will test the correctness of the results.
The input tests are used for both algorithms and are 23 in number.

To generate the tests, we created a script in python, which generates a series of random tests based on criteria using the random module.

To perform a test, a series of data are taken from the keyboard:
- the number of the test we want to create
- the number of items in the number list
- maximum range of elements
- yes / no if we want a sorted list of numbers
- sorted list - ascending / descending
- upper limit for the number of push operations
- upper limit for the number of random operations

In the generator, we created a list of the three commands. Then I created a list of numbers based on the data entered from the keyboard, which can be sorted or not. I created a file for the desired test, and randomly generated a number of push operations that I displayed in the file. Then we randomly generated a number of operations out of three possible. Values that are added to the queue are randomly generated from the list of numbers that were originally created.

The structure of a Test
- in each test there are a series of commands
- if the name of the command is push, a value will be read and the value will be added in the queue (push operation)
- if the command name is pop, the item with the highest priority in the queue will be deleted (pop operation)
- if the command name is top, the item with the highest priority in the queue will be returned (top operation)

Folder in:
- contains 23 input tests that will be applied to each algorithm
- each test is called "testX.in", where X represents the test number

Tests [1-8]
- check a small number of operations
- tests the general correctness of the implementation
- elements have values between [0 - 1000]
- are inserted in the queue between [0 - 100]

elements Test [9]
- contains 1000 push operations
- items are added in ascending order
- elements have values between [0

-1000] Test [10]
- contains 1000 push operations
- the elements are added in descending order
- elements have values between [0 -

10000] Tests [11-15]
contain between 0-10000 operations
element values range from [0-10000] Tests
[16-18]

- contain between [10000-99999] operations
- element values are between [0-90000] Tests [19-20]
- contain over 100,000 operations
- the values of the elements are between [0-100000]
Tests [21-22]
- contain over 1,000,000 operations
- the values of the elements are between [0-1000000]
- tests in a complex way the implemented algorithms
Test [23]
- contains over 10000000 operations
- the values of the elements are between [0-1000000]

For tests [11-23], a random number of push operations is generated first and then a random number of different operations are generated.

Folder out:
- contains three folders: best, p1 and p2 because in the tests there are elements with the same priority (duplicates), and the generated output is different for the two algorithms
- each subfolder contains 23 output tests which represent the correct tests that should be obtained and which will be used to test the correctness of the algorithms
- each test is called "testX.out", where X represents the test number

3.2 Computer system specifications

- Intel (R) Core (TM) i7-10750H CPU @ 2.60GHz
- GeForce GTX 1650 TI
- 16 GB RAM
- Memory Available: 13.808488 GB
- Operating system: Linux / Ubuntu

3.3 Illustration of the results of the evaluation of the solutions on the set of tests

To determine the execution time of each algorithm, we created two programs using the C ++ language that calculates the specific duration for each test. Using the chrono library, I set a clock at the beginning of the algorithm run and a clock at the end, after which I displayed the difference between the two durations. The results obtained are illustrated using a table and a series of representative graphs.

Numar Test	Timp Heap (microsecunde)	Timp AVL (microsecunde)	Diferenta de timp	Numar de operatii	Limita maxima a valorilor
1	92	94	2	4	100
2	51	48	3	7	100
3	57	64	7	9	100
4	59	76	17	5	100
5	56	81	25	10	100
6	87	92	5	16	100
7	77	63	7	17	100
8	78	90	12	89	1000
9	360	411	51	1001	1000
10	151	521	370	1001	10000
11	155	287	132	450	10000
12	902	2041	1139	4165	10000
13	1159	3254	2095	6236	10000
14	1570	3471	1901	7024	10000
15	2744	4866	2122	9997	10000
16	14200	24123	9923	48827	100000
17	16006	36567	20561	62462	100000
18	15076	47413	32337	70975	100000
19	39087	72247	33160	112233	100000
20	156184	572894	416710	903391	100000
21	459848	1287224	827376	122040	1000000
22	978517	8190510	7211993	6541213	1000000
23	4311754	10223953	5912199	12438494	1000000

FIG. 6.Execution time analysis table

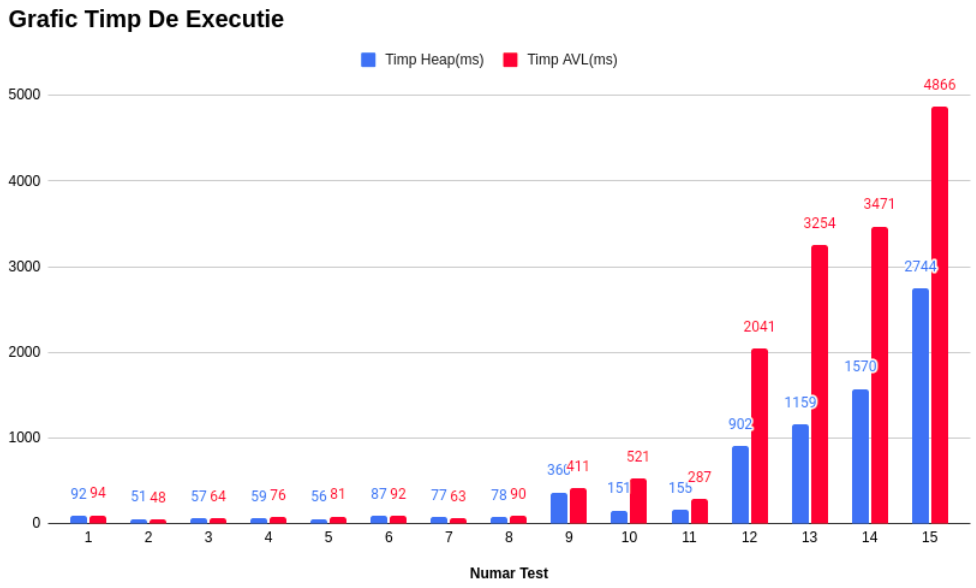

FIG. 7.Execution time graph for tests 1-15

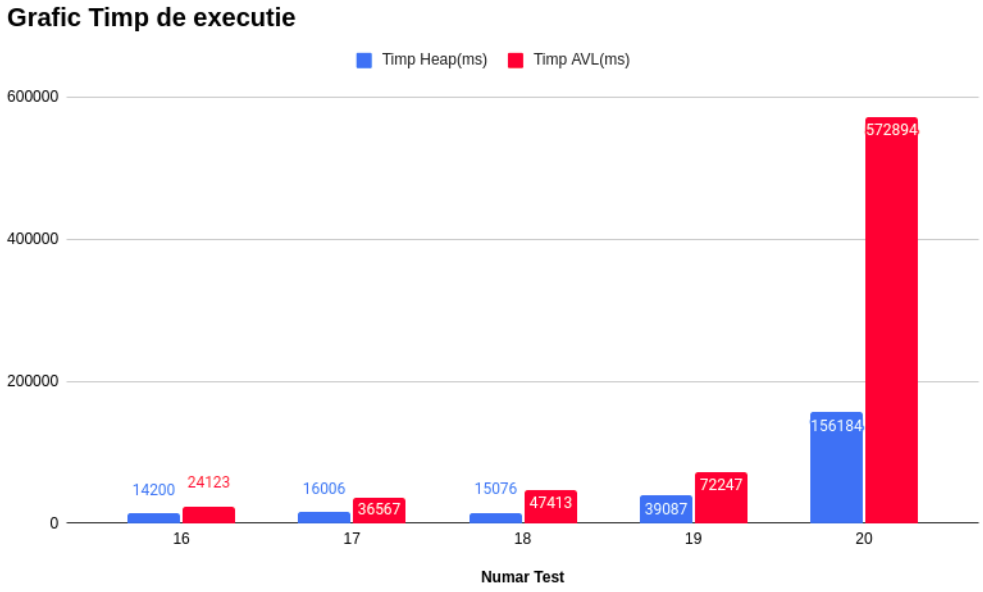

FIG. 8.Execution time graph for tests 16-20

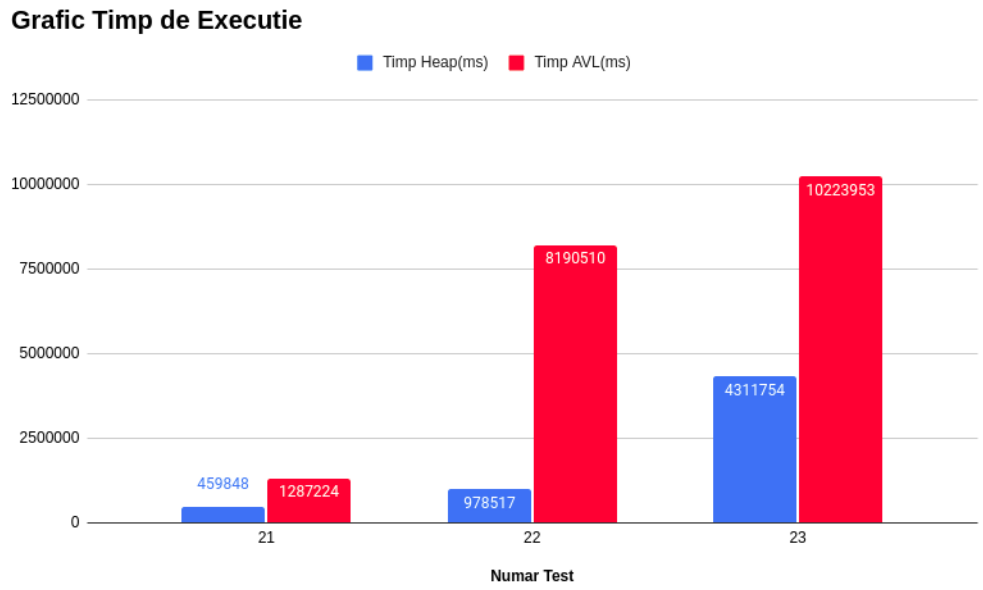

FIG. 9.Execution time graph for tests 21-23

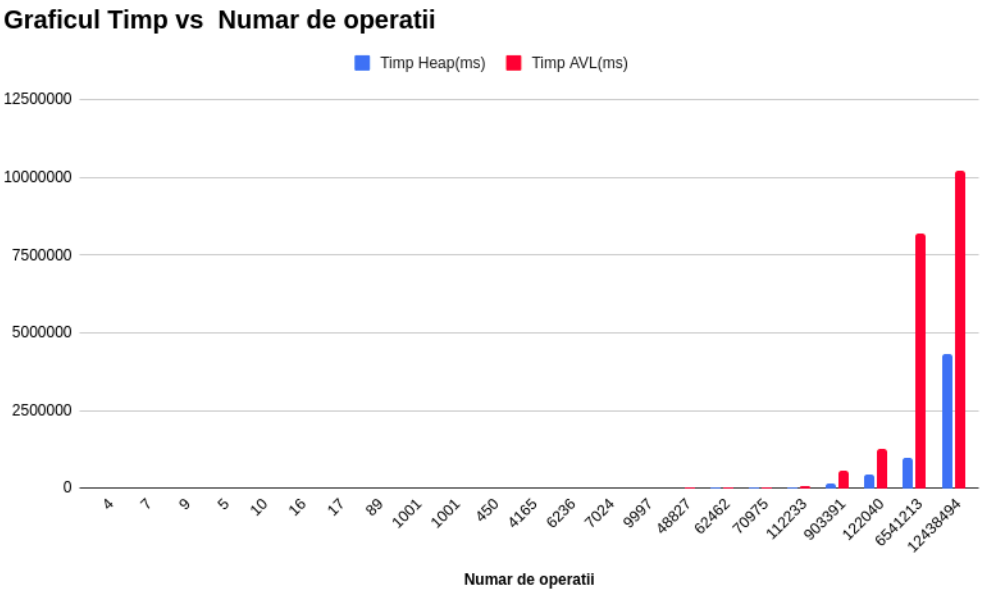

FIG. 10.Execution time graph relative to the number of operations

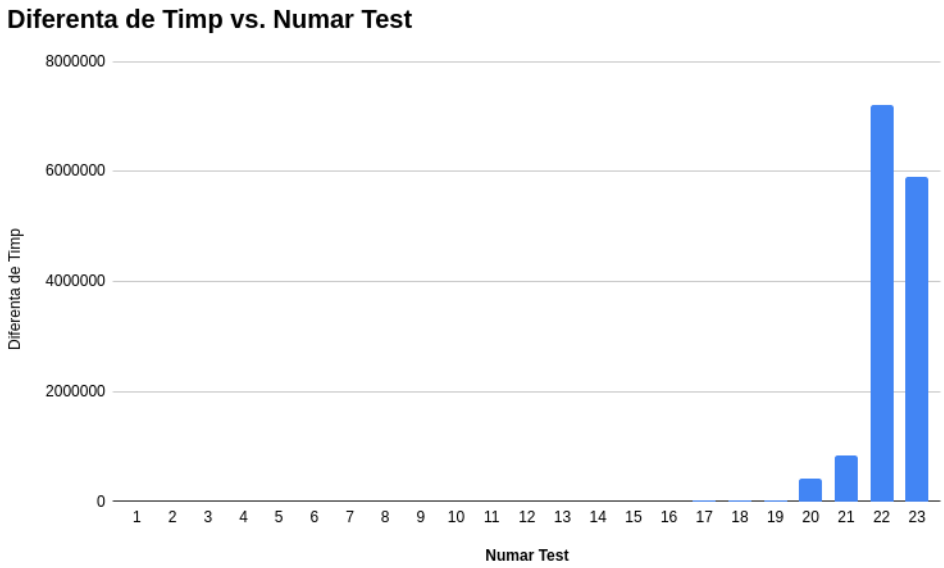

FIG. 11.Graph of the time difference between the two algorithms

3.4 Brief presentation of the values obtained on the tests

Through the tests performed, it is possible to observe both the correctness of the implementations of the two algorithms and the specific execution time of each one. Because the values differ from one run to another, we tried to average them. For the first tests, where the number of operations is not large, the execution time is approximately equal, which means that algorithms behave similarly for a small number of operations. But starting with test 7, there is a growing difference in execution time, in favor of the Heap algorithm. This is due to both the rotation operations required for the AVL to rebalance and the height upgrades. In tests 9 and 10 we performed the push operation using a series of sorted values ascending or descending. Analyzing results, this does not affect the execution time. In Test 23, where the number of operations is very complex, the Heap algorithm runs in 4311754 microseconds, while the AVL algorithm runs in 10223953 microseconds, the time difference being 5912199 microseconds.

4 Conclusions

Following the analysis performed on the two algorithms, it is observed that the Heap algorithm is superior to the AVL one, both from the point of view of the complexity of the implemented operations, and from the point of view of the execution time. AVL is also a complex Data Structure, more difficult to implement. In practice, because time for a search is a very important factor in a priority queue, its implementation is chosen using a Heap. So for the proposed problem, I would also approach an implementation of the priority queue based on a Max Heap algorithm, in order to obtain the element with the maximum priority in the shortest possible time.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms
2. http://faculty.salina.k-state.edu/tim/ossg/_images/priority_queues.png
3. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
4. https://www.geeksforgeeks.org/applications-priority-queue/
5. https://www.geeksforgeeks.org/heap-data-structure/
6. https://www.w3schools.in/data-structures-tutorial/avl-trees/
7. https://www.tutorialspoint.com/data_structures_algorithms/
8. https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
9. https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-09
10. https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/lab-11

The last time I accessed the references was December 17, 2021.