César Leonardo Canales Rivera A00345026 Andres Aristizabal

Carlos Andrés Restrepo Marín A00355028 Andrés Aristizabal

Luis Felipe Sánchez Vega A00355727 Juan Manuel Reyes

**Engineering design**

**Phase 1. Identification of the problem**

Minecraft is a popular video game that has resurged recently. The game has an inventory system that consists of a 4x9 matrix which stores blocks that the player picks up. The maximum amount of blocks in each one of the matrix’s spaces is 64. If a player picks up a block, it goes to the space which contains the same block or to an empty space if the space is already full. However, the verification process to see where the block should be placed is way too slow and we’re required to design a solution that does this efficiently. Also, we’re to implement a functionality that allows the player to have multiple quick access bars that can be accessed in order to make building more efficient.

Identification of necessities and symptoms

* The addition of elements to the inventory has to be done effectively.
* The current addition of elements to the inventory is too slow.
* A functionality that allows the building to be more practical.
* An intuitive way to visualize the functionalities mentioned above.

Definition of the problem

Mojang requires the development of a software that allows them to visualize the new inventory features.

The software will have the following functionalities:

1. A graphical interface that shows the player’s inventory.
2. A functionality that allows the user to add blocks to the inventory.
3. A functionality that allows the user to drag blocks and create new quick access inventory.
4. A functionality that allows the user to travel through the different quick access bars.

**Phase 2. Recompilation of information**

**Block:** Blocks are the basic units of structure in Minecraft that can be directly placed in the game world or destroyed. They can be picked up and stored by the player.

**Inventory:** The inventory is a pop-up menu that the players use to manage the items (blocks and others) they carry. The inventory consists of a matrix of four rows and nine columns, each cell is called a slot.

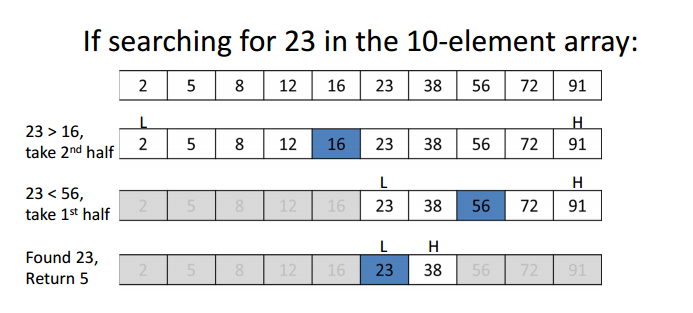
**Slot:** Slots are the place in which each type of item is placed. Each slot can store up to 64 items (blocks), but it depends on the type of item to store, some types of items can only be grouped into groups of 16, others can only contain one.

**Phase 3. Looking for creative solutions**

In order to find the most efficient solution for the problem of adding elements to the solution, various ideas are considered. Considering that the current adding time is linear, we require a solution that is faster than this. The solutions are given below:

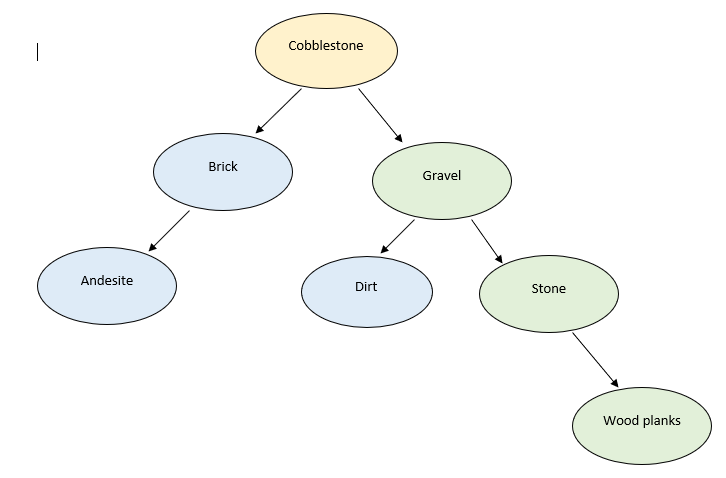
1. Binary search: The algorithm searches a sorted array by repeatedly dividing the search interval in half. It begins with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, it narrows the interval to the lower half. Otherwise it narrows it to the upper half. Binary search repeatedly checks until the value is found or the interval is empty.

To understand this idea better, the following graphic is presented:



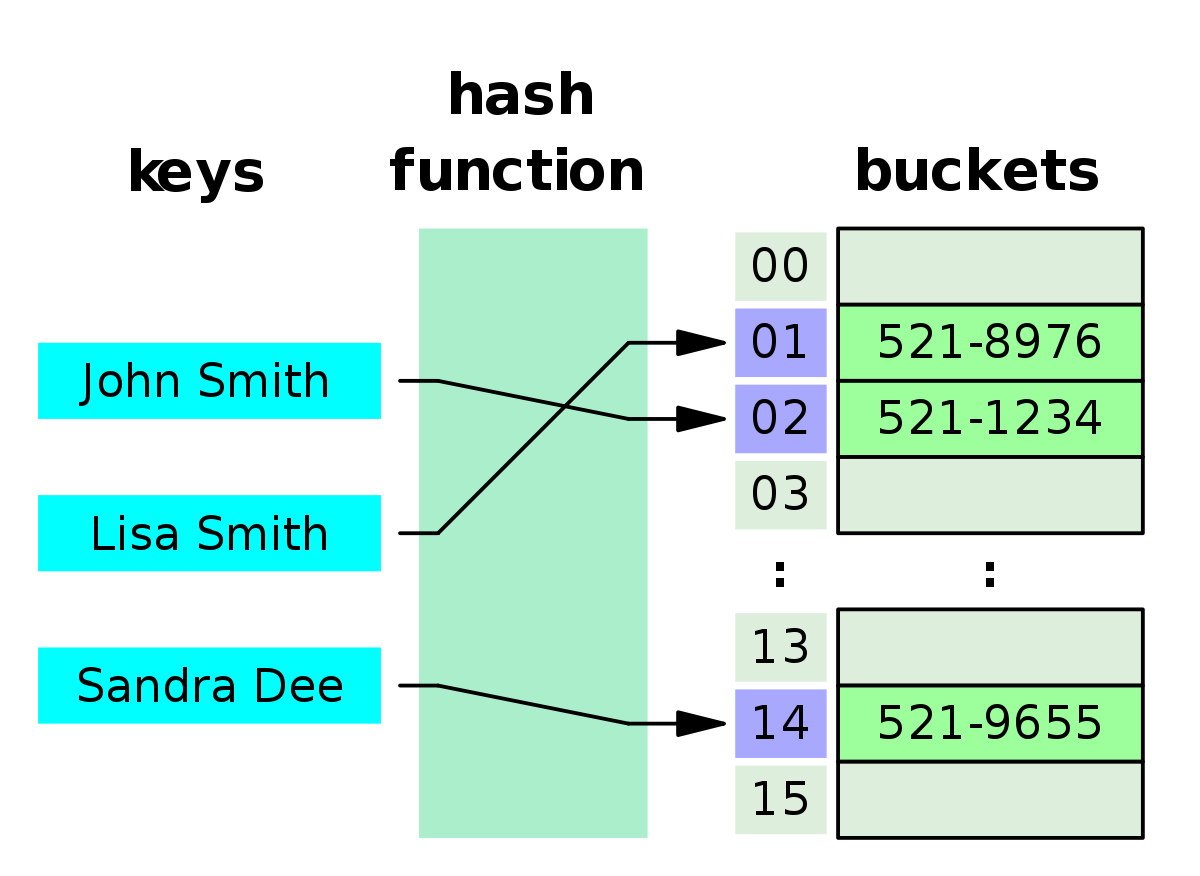
Note that the array of elements has to be sorted.

1. Binary search tree: This idea consists of storing the elements in a non linear data structure, which sorts them by an id. For this problem we can assume the id to be the name of the element. So once the data structure is built the elements can be sorted lexicographically. The binary search tree sorts the elements by putting the ones that are smaller than the root to the left and the ones that are bigger to the right.



This is an illustration of how the data structure would look.

1. Hash table: This approach consists of using a dictionary data structure which allows us to store and retrieve data quickly. The data structure is represented by a list of pairs, each pair consists of a key and an associated value. For this case the key would be the name of the element and the value would be the list of slots that contain that same element. Hash tables can have collisions which happen when two different keys have the same position in the table, this can be reduced by using different techniques. The most popular ones are chaining and open addressing.

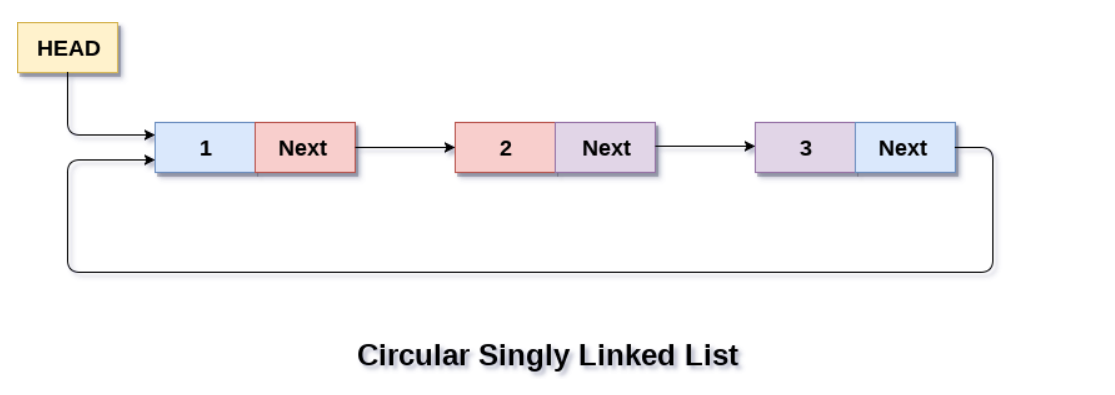


We also have to implement a new functionality that allows the player to have multiple access bars, each one having one unique material that gets transported from the inventory to the access bar when the player selects it, this has the objective of making building way more practical. We suggest the following solutions:

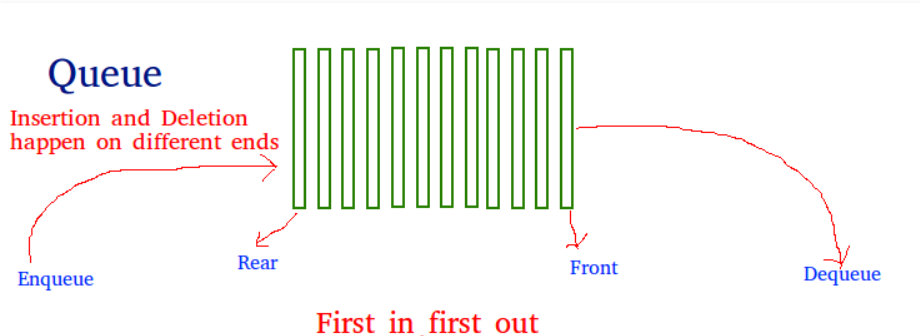
For the problem that consists of putting the blocks in the quick access bars we consider the same solutions as the last problem mentioned above because these two are similar in the way that we have to look for the blocks that are the same in the inventory. However, for the representation of the blocks in each one of the quick bars we propose using a Stack, this data structure allows us to get rid of the blocks as they are used and taking into account the fact that the blocks will be sorted in non increasing order, this structure is the most suitable.

Finally, for the problem of having multiple quick access bars and giving the player the ability to navigate through them, we suggest:

1. Circular linked list: A circular linked list is a continuous sequence of elements that are connected by “links” that are unidirectional. This means that each element is considered to be a node that has a connection to the left node in the list. The last element in the list has a connection to the first element hence why the list is called circular. For this problem, the nodes would be the different access bars and they would have links or connections between them in order for the player to be able to navigate through them.



1. Queue: A queue is a continuous sequence of elements, the particularity of this data structure comes at the moment of removing an element, for the queue only the first element can be removed or obtained. This means that if we have a queue on integers {3, 4, 5, 6}, only the 3 can be removed or obtained. Also, when inserting an element it goes to the back of the queue. In other words, the queue follows the First In First Out(FIFO) principle. For this problem, each element can be the quick access bar, when the player wants to move to the next quick access bar, the first element can be removed from the queue and placed in the back, simulating the desired effect.



**Phase 4. Transition from ideas to preliminary designs**

After reviewing the solutions proposed above, the following analysis was made:

The second proposed solution (binary search tree) is discarded, because at some point the binary search tree can have only one branch, that is, all the elements would be in a line. When this happens the worst case time of insertion and search of an element can be equal to the time that is currently in the application O (n). Therefore, it is necessary to balance the tree and this creates greater difficulty in the implementation.

Thus, there are two alternatives to consider

Binary search

* The elements must be ordered to apply binary search.
* An efficient solution must be sought to sort the elements previously.
* The time complexity can increase to more than O (log n) which is the complexity of the binary search due to the precondition that the elements must be sorted. A total intuitive complexity would be O(n log n).

Hash Table:

* It allows to have elements of the same type together.
* The temporal complexity for insertion and search operations is constant time O(1)
* A collision solution method that best represents the problem should be sought. A collision occurs when two different keys point to the same position.
* Several hash functions can be used to obtain the key value.

For the problem of having multiple quick access bars and giving the player the ability to navigate through them.

The two proposed solutions are good, so both will be reviewed:

Circular linked list:

* It allows you to navigate through it, without the need for additional operations
* Priority order must be given to show the bars.
* To add a new bar, the temporal complexity is constant time O (1)

Queue:

* Additional operations must be performed when the end of the bars is reached, show again from the beginning.
* The queue brings by default an order of priority, which is, first to enter first to leave.
* To add a new bar, the temporal complexity is constant time O (1)

**Phase 5. Evaluation and selection of solutions**

**Criteria**

Considering the options that remain after going through phase three and four, the following criteria has been proposed in order to choose the best solution. The greater the total score, the higher rated the algorithm will be.

* Criteria A. Efficiency, a fundamental requirement is finding specific types of blocks in the inventory in a fast way. The complexity of the solution is:
* [4] O(1) complexity
* [3] O(1+α) complexity
* [2] logarithmic complexity
* [1] n \*log(n) complexity
* Criteria B. Requirements of the solutions.
* [2] The solution does not need to do extra processes to give the right answer.
* [1] The solution needs to execute extra processes in order to give the correct output.
* Criteria C. Ease of implementation
* [2] the solution can be easily implemented in the programming language
* [1] the implementation of the solution may require more advanced techniques

**Evaluation**

Taking into account the criteria established above, we will carry out the evaluation of the solutions

*for looking for an item in the inventory.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Solution | Criteria A | Criteria B | Criteria C | Total |
| Solution 1. Hash tables | 3 | 2 | 1 | 6 |
| Solution 2. Binary search | 2 | 1 | 2 | 5 |

*for having multiple quick access bars.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Solution | Criteria A | Criteria B | Criteria C | Total |
| Solution 1. Circular linked list | 4 | 2 | 2 | 8 |
| Solution 2. Queue | 4 | 2 | 2 | 8 |

**Selection**

The requirements that were assigned indicate that the best solution for this problem is the one that implements hash tables since they allow us to search, insert and delete elements without looking for them in the whole inventory, instead, hash tables go directly to the position needed.

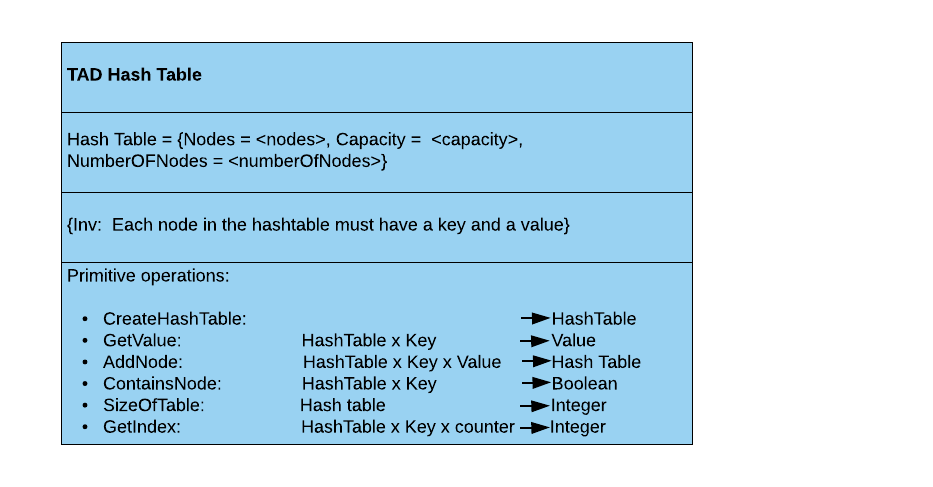
For the problem of having multiple quick access bars, the solution chosen is the one that uses queues. Since both solutions have the same rate, it was decided that, for the purposes of this project the second alternative is going to be used.

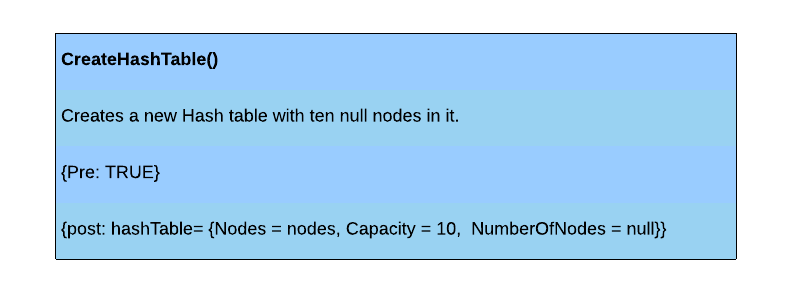
**Phase 6. Preparation of information and specifications**

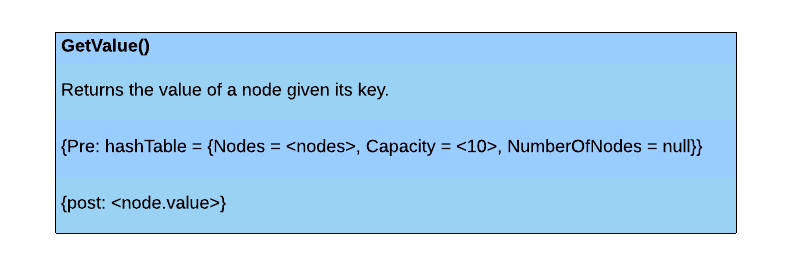
[***class diagram***](https://drive.google.com/file/d/1LOjDFo9atr6mwHwm9jizqgWns5eo4-jG/view?usp=sharing)

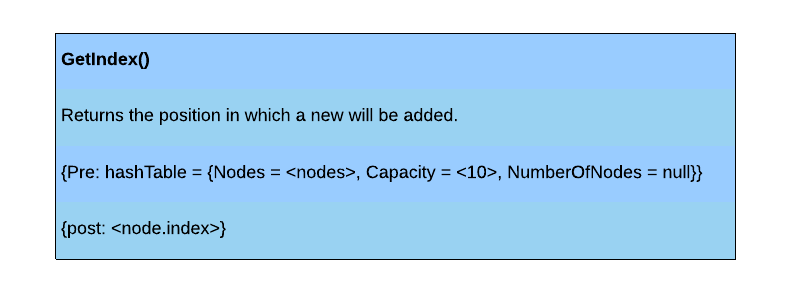
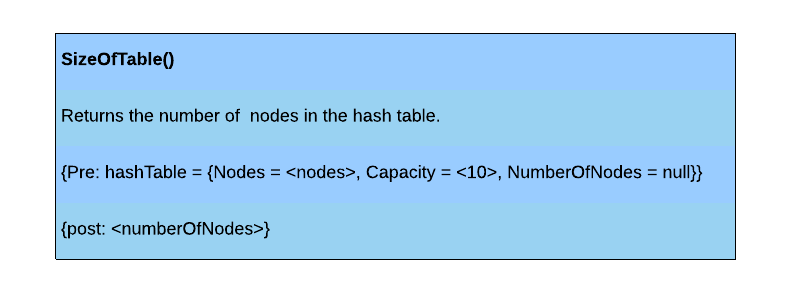
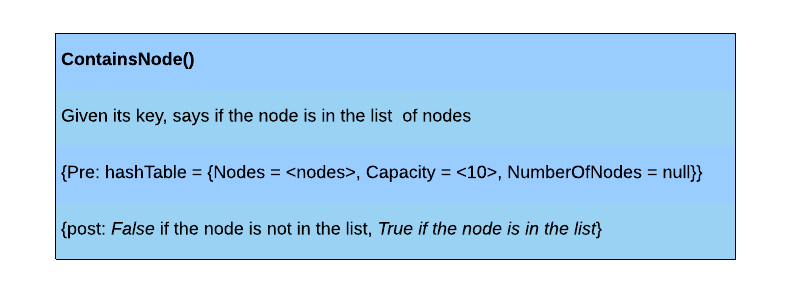
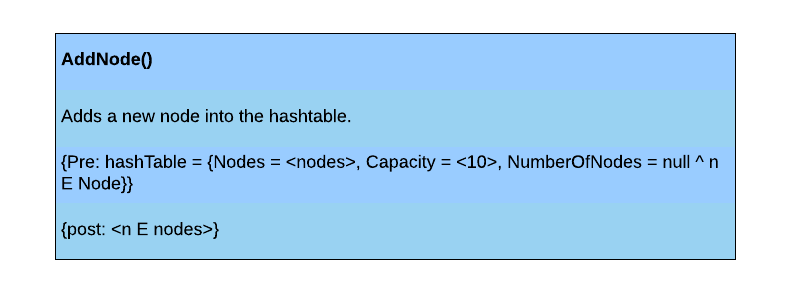
***TAD specification:***

**Hash Table:**

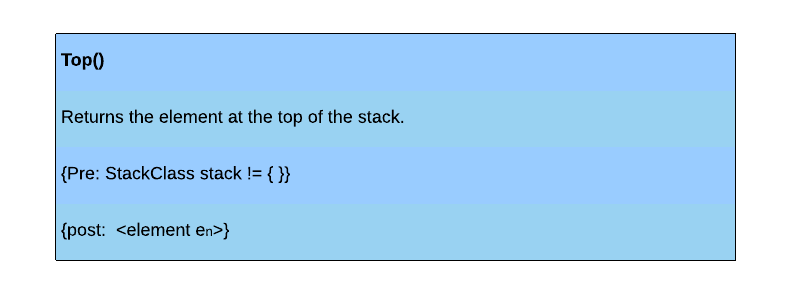
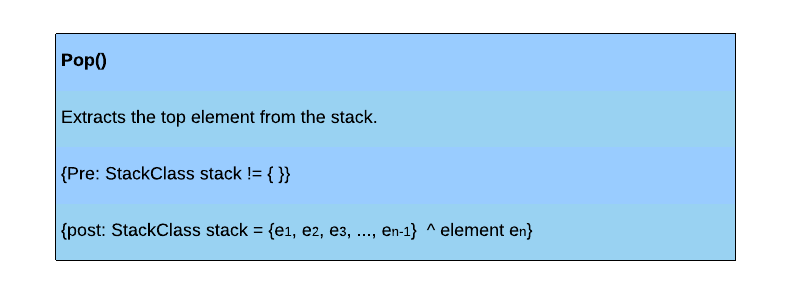
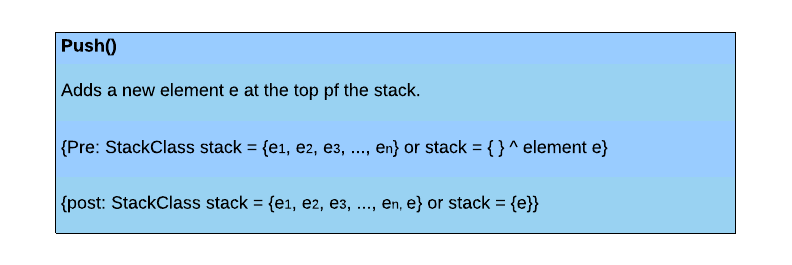
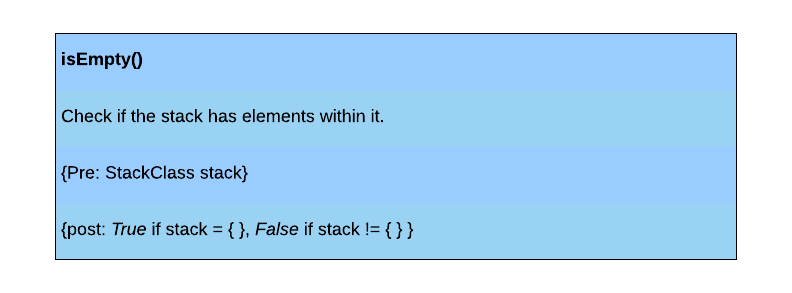
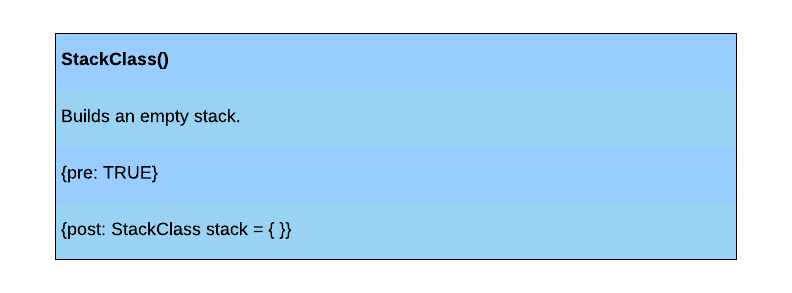
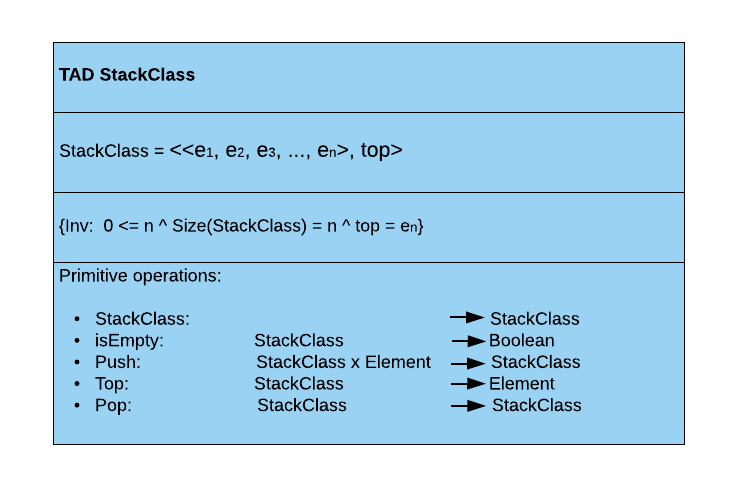
****

****

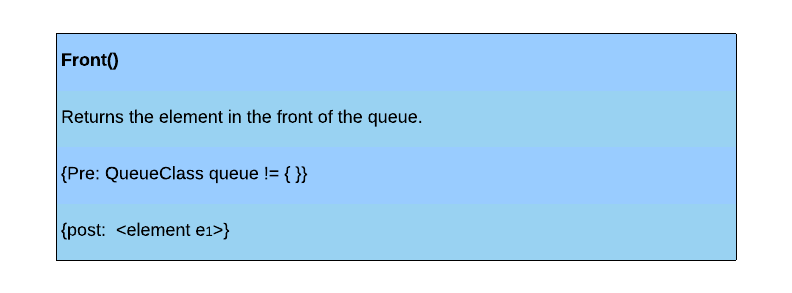
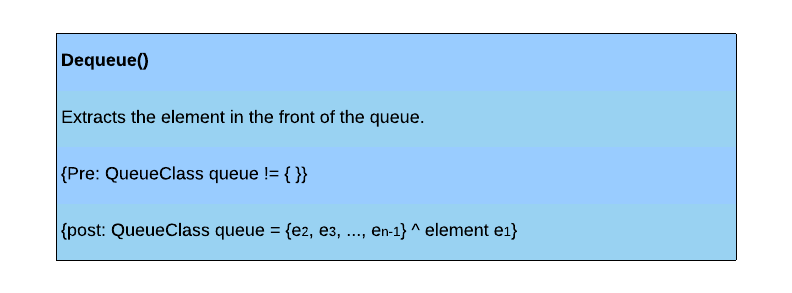
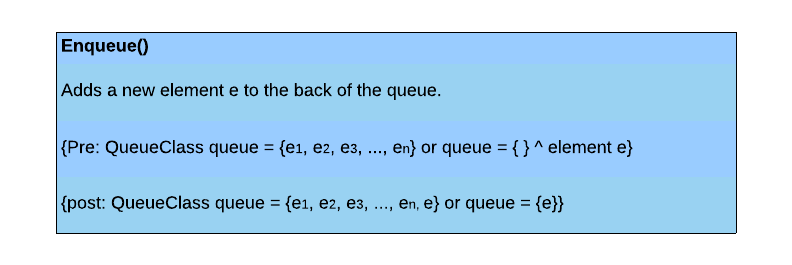
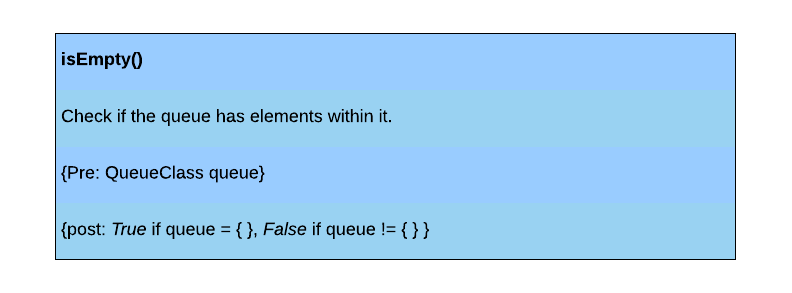
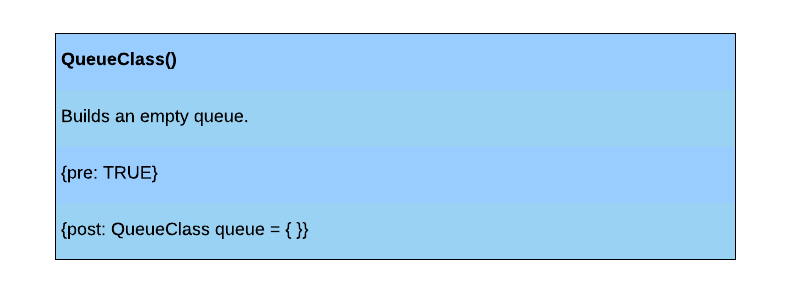
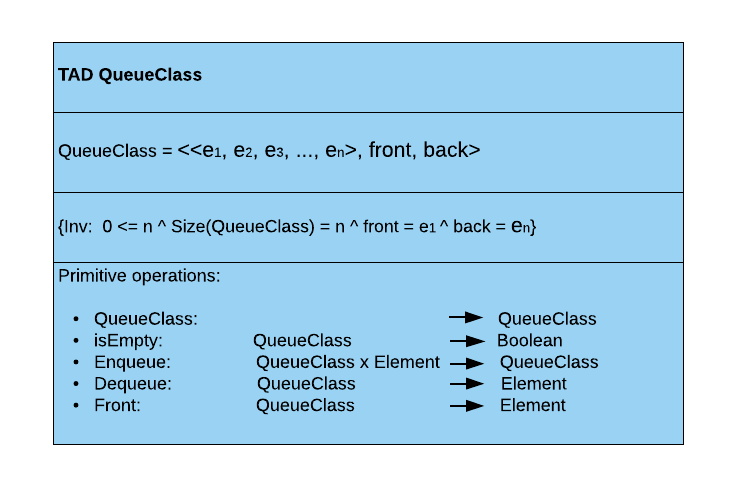
****

****

**Stack:**

****

**Queue:**

****

Problem specification (Input/Output)

* Problem: Adding blocks to the inventory.

Input: A number N, the quantity of the block and the name of the block to be added.

Output: The inventory matrix with the new block added.

Constraints:

* N must not be greater than 64, considering that each slot can only hold this much.
* The name of the block must be a valid minecraft block.

UNIT TEST DESIGN

Stages configuration.

|  |  |  |
| --- | --- | --- |
| **NAME** | **CLASS** | **STAGE** |
| stage1 | HashTableTest | Empty |
| stage2 | HashTableTest |  |
| stage3 | HashTableTest |  |

|  |  |  |
| --- | --- | --- |
| **NAME** | **CLASS** | **STAGE** |
| stage1 | StackClassTest | Empty |
| stage2 | StackClassTest |  |
| stage3 | StackClassTest |  |

|  |  |  |
| --- | --- | --- |
| **NAME** | **CLASS** | **STAGE** |
| stage1 | QueueClassTest | Empty |
| stage2 | QueueClassTest |  |
| stage3 | QueueClassTest |  |

Test cases design.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** check the correct creation of a hash table. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +HashTable () | Stage1 | None. | An empty hash table has been created. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** check if a node is correctly added to the hash table (standard case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +add() | Stage2 | integer k = 1  String v = "test add method" | The method adds a new node with a key and a value into the hash table. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if 10000 nodes are correctly added to the hash table (limit case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +add () | Stage2 | An integer k for each node.  A string v for each node. | The 10000 nodes were added to the hash table. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if given the key of node the method returns its value (standard case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +get () | Stage3 | Integer key = 19 | The method returns the string v = "stage3, test value: 19" which is the value of the node with key = 19 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if given the key of several nodes the method returns their values (limit case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +get () | Stage3 | An integer i that represents the key for each node | The method returns the value of each node for which its key was given. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method can find an element with the given key. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| HashTable | +containsKey () | Stage3 | An integer that represents the key of the element. | A boolean that says if an element within the table has the given key |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct creation of an empty stack. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +StackClass () | Stage1 | None. | The method creates an empty stack. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if an element is correctly added into the stack (standard case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +push () | Stage2 | An integer that is the new element to add in the stack. | The stack now has one element within it. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if 100000 elements are correctly added into the stack (limit case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +push () | Stage2 | 100000 integers that will be the elements to save. | The stack now has 100000 elements within it. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the last element of the stack is correctly deleted (standard case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +pop () | Stage2 | None. | Now, the size of the stack is size - 1 elements. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method can remove 99999 elements from the stack (limit case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +pop () | Stage2 | None. | The stack now has only one element within it.. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the last element added to the stack is the top element (standard case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +top () | Stage3 | None. | The method returns the element at the top of the stack. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check, in several times (99999 times), if the last element in the stack is the top element (limit case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +top () | Stage3 | None. | The method correctly returns the top element after each pop operation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check verifies is a stack is empty or not. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| StackClass | +isEmpty () | Stage2 | None. | The method returns *True* since the stack is empty. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct creation of an empty queue. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +QueueClass () | Stage1 | None. | The method creates an empty queue. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method adds elements to the queue in the correct order (standar case) | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +enqueue () | Stage2 | integer i = 10  integer j = 150 | The method adds two integers in the correct order. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method correctly adds 100000 elements to the queue (limit case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +enqueue () | Stage2 | 100000 different integers | Each element is added to the queue one after the other. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method correctly removes the first element in the queue (standard case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +dequeue () | Stage3 | None. | Now the queue has size-1 elements within it since the first element was deleted. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method correctly removes the first 99999 elements of a queue of 100000 elements (limit case). | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +dequeue () | Stage3 | None. | Now the queue has only one element within it, that is, the last element added. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Verifies is a queue is empty or not. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +isEmpty () | Stage2 | None. | The method returns *True* since the queue is an empty queue. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check if the method correctly returns the first element added to the list. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| QueueClass | +Front () | Stage3 | None. | The method returns the first element added to the queue. |

**Phase 7. Implementation of the design**

Implementation in a programming language, we decided to use Java in this case.

1. add block to inventory

|  |  |
| --- | --- |
| Name: | addBlock |
| Description: | add a block amount found to inventory |
| Input: | * type : String, type of material found * quantity : int, amount of material found |
| Output: | boolean, true if the item could be added to the inventory, otherwise false |

In the inventory class

public boolean addBlock(String type, int quantity){

…..

}

In the HashTable class

public void add ( K key, V value){

boolean found = false;

int counter = 0;

while( !found && counter<capacity) {

int index = getIndex(key, counter);

if(nodes.get(index)==null) {

found = true;

nodes.remove(index);

nodes.add(index, new HashNode<K, V>(key, value));

numNodes++;

}

counter++;

if(counter==capacity && !found) {

capacity\*=2;

for (int i = counter; i < capacity; i++) {

nodes.add(null);

}

}

}

}

public V get ( K key){

V value = null;

int counter = 0;

boolean found = false;

while( !found && counter<capacity) {

int index = getIndex(key, counter);

if(nodes.get(index)!= null && nodes.get(index).getKey().equals(key)) {

found = true;

value = nodes.get(index).getValue();

}

counter++;

}

return value;

}

1. new quick access bar

|  |  |
| --- | --- |
| Name: | newQuickAccessBar |
| Description: | create a new quick access bar |
| Input: | click on the item you want to create a quick access bar |
| Output: | show the new quick access bar |

**In the inventoryController class**

public void newQuickAccessBar(ActionEvent event) {

……

}

**In the Queue class**

public void enqueue( V value){

QueueNode<V> newNode = new QueueNode<V>(value);

if (isEmpty()) { //insertion into empty queue

first = newNode; // new node is referenced by first

}

else {

last.setNext(newNode); //insertion into non-empty queue

}

last = newNode;

}

public V front(){

V value = null;

if(!isEmpty()){

value = first.getValue();

}

return value;

}

**In the Stack class**

public void push(V value) {

top = new StackNode<V>(value, top);

}

public V top() {

V value = null;

if(!isEmpty()) {

value = top.getValue();

}

return value;

}