**Daniel Alejandro Fernández Robles**
**A00354694**

## Step 1. Identification of the problem.

The specific needs of the problematic situation, symptoms and conditions for resolution are listed below.

Identification of needs and symptoms:

There is currently a very slow algorithm to take the internal inventory blocks to the quick access bar. This consists of verifying the type of block requested to then go one by one to the slots in the system, and in this way detect that if a stone block is obtained, it must be grouped with the stone blocks that are present in the Inventory.

The RAM usage is excessive.

The speed at which the quick access bar allows building does not meet the expectations of the users.

Definition of the problem:

Blocks inventory management is very inefficient and the quick access mode doesn't accomplish the users expectations.

Requirements:

| **Name** | **R.#1. Add blocks to the inventory** |
|---|---|
| **Summarize** | The program will allow you to add a set of blocks in a slot with available space and the rest(if exist) in another slot of the inventory, keeping each slot with blocks of the same type. |
| **Input** | ● A string representing the type of blocks to be inserted.<br>● An integer representing the total number of blocks to be inserted. |
| **Output** | An integer giving the total number of blocks that was not possible to insert. |

| **Name** | **R.#2. Remove blocks from the inventory** |
|---|---|
| **Summarize** | The program will allow you to take a set of blocks out of the inventory. |
| **Input** | ● A string representing the type of blocks to be removed.<br>● An integer representing the total number of blocks to be removed. |
| **Output** | The blocks were taken out of the inventory |

| | |
|---|---|
| **Name** | **R.#3. Add a quick access bar** |
| **Summarize** | The program will allow you to have **n** quick access bars, where **n** is the total number of different types of blocks in the inventory. They will be autofilled, this means that whenever a slot gets empty, if there are more blocks of the same type in the inventory, they will be taken automatically to the quick access bar. So every time you add blocks of a type that is not already contained in the inventory, a new quick access bar is created to allocate them. |
| **Input** | ● The type of blocks this quick access bar will contain. |
| **Output** | A boolean representing whether or not it was possible to add the requested quick access bar. It is true if there was not a quick access bar already containing the same type of blocks and false otherwise. |

| | |
|---|---|
| **Name** | **R.#4. Remove a quick access bar** |
| **Summarize** | Each time a quick access bar is emptied (that is, there are no more blocks of that type), the quick access bar is deleted to free memory and lets the list of quick access bars contain only useful information (avoiding showing an empty quick access bar for the user). |
| **Input** | ● The quick access bar to be removed. |
| **Output** | The quick access bar has been removed. |

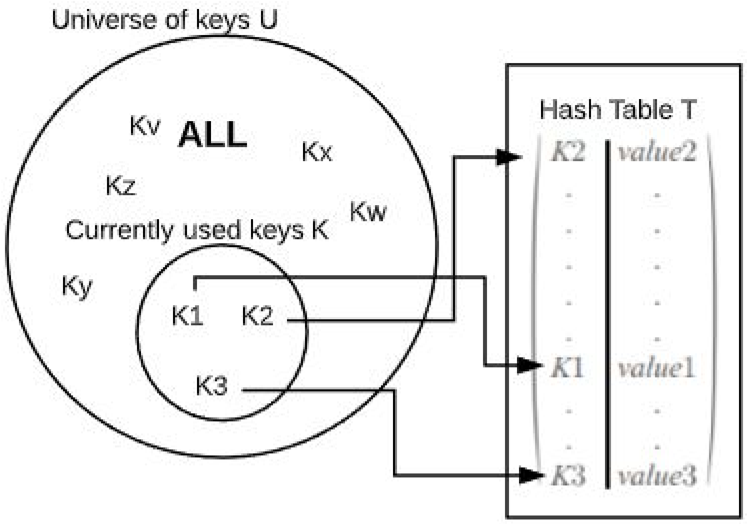
## Step 2. Gathering needed information.

Dictionary:

A dictionary is a general data structure that stores elements by pairs(keys and values) and maps them according to its keys. The dictionary can’t hold two elements with the same key. According to the implementation, when you try to add an element with a key that is already stored in the dictionary either you get an error or the element data is overwritten.

Hash table:

A hash table is a sort of dictionary so It has all the characteristics and behaviors mentioned above. It is implemented as an array in which slots contains elements stored according to a hash function.



Hash function:

A hash function is a mathematical relation that finds the slot were an element must be allocated, using its key as entry. In other words, it takes values from U and returns values in T, that is:

$$h: U \rightarrow T = \{0, 1, \dots, m-1\}$$
$$m = |T|$$

## Step 3. Searching for creative solutions.

First option: Keep the elements ordered

This approach can be useful since binary search can be implemented to improve search time. Also, it's an array based implementation and is easy to understand, implement and test. Suppose you fill in the spaces in the order in which the elements arrive and group them into the first available spaces in the table. It requires moving a part of the elements each time they are grouped or removed from behind other slots containing elements to keep the table sorted. In the worst case this takes O(n) time to insert or delete as, although not necessarily all elements in the table will be moved, the verification is performed to know if the element should be moved.
The quick access bars can be seen as another matrix where each row represents the submenu containing elements of the same type and the number of columns would be 9 as this is the maximum slots per quick access bar. Each row (quick access bar) will be treated as an array implementation of a stack with maximum 9 elements in it. In order to the stack to be autofilled, whenever the top element is consumed a new element from the hash table is passed by reference into it if available.

Second option: A key registry hash table, an inventory hash table and a queue of stacks of blocks

The main idea of the game is to allocate different groups of blocks in different slots, but they have a limit of 64 blocks of the same type per slot. So, if a slot is full of the element **X** and the user wants to add more elements of type **X**, another slot is allocated to store another group of 64 blocks of **X** type. For making the inventory system

efficient, a hash table is the first option that comes to mind, but if we let it have the behavior just mentioned, various elements with key **X** would be in the hash table(so it wouldn't be a hash table). The solution to this problem is to use two hash tables, one of them is the main hash table that represents the inventory and the other one is a registry hash table in which if there are various elements of type **X**, they are stored in a list in their corresponding slot(so the type **X** would be the key and the lists of elements is the value). When new elements arrive to the registry hash table and are mapped to the slot with the list of elements of the same type, you check if there is available space in the element in the top and fill it, if there are remaining blocks an aleatory key is generated by appending a random string to **X** (e.g. X01a~A). You must verify this random key is not already taken, if it is, you must generate it again until it is not.
Once a new aleatory key is generated, a new element is created with that key and the value is the remaining number of blocks. We put it in the top of the list and perform a secondary hashing to map this new element to the inventory hash table.
The number of blocks at the top of set of blocks of the type that is being used in the registry hash table decrease as the user uses blocks in it to build. As this element is the same that is contained in the inventory hash table, it's not necessary to do anything else in it.
When all 64 blocks of a set are consumed (the counter reaches 0) it is removed from the inventory hash table and subsequently from the key registry hash table. When all elements of the same type are consumed, the stack that contains elements of that type in the key registry hash table is going to be empty. As both hash tables are going to be mapped by open addressing, the previously described procedure has to let the flag **DELETED** (only in the key registry hash table as it is not necessary in the inventory hash table) in that slot, that means the traveling of the table is not going to stop because of this slot pointing to null and it's simultaneously going to let us know whether new elements would be inserted there or the travel must continue until finding the corresponding slot or a free one. Also, this doesn't affect the search of available elements of other types.
In order to pass the key registry elements to the quick access submenus, the top of the stack will be saved in a temporary variable since this will be the one with the least number of blocks and therefore as each menu is a stack, you have to add it last so that it is on top and ends up spending first.
When the user goes from a submenu to the next, elements will be taken back to the key registry hashtable by the same procedure they were pulled out of it. Then, the next submenu is prepared by passing the appropriate elements to the quick access bar.
Submenus (each quick access bar) are stacks in a queue.
While you move through the submenus you must dequeue them from the actual queue and store the ones that were already seen in another queue in order to preserve their current sequence. This allows you to go through the submenus over and over again and see them in the same order as the last time you took a round.

## Step 4. Stepping from ideation to preliminary designs (including modeling).

The characteristics of the only two alternatives that were yield in the previous step are listed below:

Keep the elements ordered:

**--** Search can be performed by binary search method, improving time complexity.
**--** It’s not just easy to understand but easy to implement.
**--** It is a mess to insert elements as they are sorted and must remain sorted. This implies to move various elements to other slots in order to add the new elements in a position that maintains the table sorted.
**--** It’s also a mess to delete elements because of the same reason insertion is a mess.
**--** It allows retrieval of elements by just requesting the specific type of blocks you need as binary search would retrieve the first set of blocks of the requested type that finds.

A key registry hash table, an inventory hash table and a queue of stacks of blocks:

**--** Retrieval of elements is very quick as it takes them out of stacks and this procedure takes constant time.
**--** Insertion of elements is very quick as they are stored in a stack and insertion in a stack takes constant time. Also, the secondary insertion in the main inventory is quick.
**--** Deletion of elements is very quick because of the same reason retrieval is quick.
**--** Deletion of all the elements of a specific type doesn’t affect the behavior of the key registry hash table as it uses the special flag **DELETED**.
**--** It allows retrieval of elements by just passing the type of the element required as there’s no difference between blocks in the same slot of the key registry hash table (e.g “Dirt” is requested by the user and no matter if you retrieve “Dirt~Ap01”, “Dirt~aD50” or any variant of “Dirt” + aleatory string).

## Step 5. Evaluation and selection of preferred solution.

Criterion A: Ease of interpretation.

[3] Easy peasy
[2] I had to think for a while but I get the idea
[1] What?

Criterion B: Efficiency.

[3] Constant
[2] Logarithmic
[1] Linear

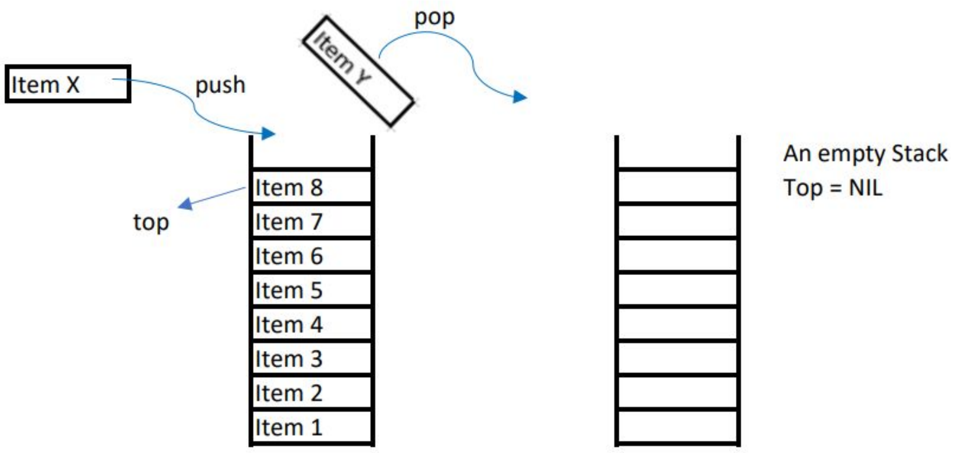
| | Criterion A | Criterion B | Total |
|---|---|---|---|
| First Option | 3 | 1 | 4 |
| Second Option | 2 | 3 | 5 |

Selection:

The second option is the one that is going to be used.

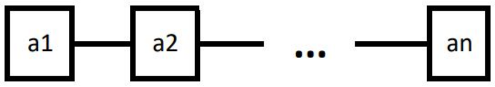
## Step 6. Preparation of reports, plans, and specifications.

Statement of Abstract Data Types

| ADT Stack | |
|---|---|
| Definition | The ADT Stack is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion, deletion and retrieval of elements only at one end of the sequence (the "top"). It's because of this that is called a last-in-first-out (or LIFO) list.<br>A stack is either empty, or it consists of a sequence of items. |
| Object Representation |  Item X<br>push<br>Item Y<br>pop<br>top<br>Item 8<br>Item 7<br>Item 6<br>Item 5<br>Item 4<br>Item 3<br>Item 2<br>Item 1<br>An empty Stack<br>Top = NIL |
| Invariant | (s.isEmpty() == true $\wedge$ top == NIL) $\oplus$ (s = $\{a_n \rightarrow a_{n-1} \rightarrow \ldots \rightarrow a_1\}$ $\wedge$ top == $a_n$) |
| Procedures | ● **Constructor operations**<br>○ *createStack()*<br>//Creates an empty stack<br>//**post:** A stack with no elements inside has been created<br>● **Analyzer operations**<br>○ *top()*<br>//Returns the element at the top of the stack but does not<br>//modify it.<br>//**post:** Returns the top element of the stack. If the stack is<br>//empty then its top is NIL and this is the retrieval.<br>○ *isEmpty()*<br>//Determines whether a stack is empty.<br>//**post:** Returns TRUE if the top of the stack is NIL and FALSE<br>//otherwise.<br>○ *size()*<br>//Returns the size of the stack.<br>//**post:** Returns the total number of elements in the stack.<br>● **Modifier operations**<br>○ *push(newItem)* |

| | |
|---|---|
| | //Adds newItem to the stack.<br>//**post:** newItem is at the top of the stack and the previous top<br>//is under it in the same order as before.<br>○ *pop()*<br>//Removes and returns the top element of the stack.<br>//**pre:** The stack is not empty.<br>//**post:** The top is taken off the stack and the new top is the<br>//element that was under the previously returned element. If<br>//the stack is empty, it results in an error. |

| Axioms of the ADT Stack |
|---|
| 1. (aStack.createStack()).size() = 0<br>2. (aStack.push(item)).size() = aStack().size() +1<br>3. (aStack.pop()).size() = aStack.size() - 1<br>4. (aStack.createStack()).isEmpty() = TRUE<br>5. (aStack.push(item)).isEmpty() = FALSE<br>6. (aStack.createStack()).pop() = ERROR<br>7. (aStack.push(item)).top() = item<br>8. (aStack.push(item)).pop() = item<br>9. (((aStack.push(item1)).push(item2)).pop()).top() = item1 |

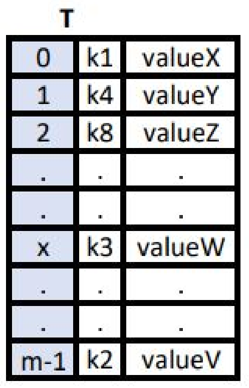
| **ADT Queue** | |
|---|---|
| Definition | The ADT Queue is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion of elements only at the back of the sequence. Deletion and retrieval of elements are only performed at the front of the sequence. It's because of this that is called a first-in-first-out (or FIFO) list.<br>A queue is either empty, or it consists of a sequence of items. |
| Object Representation |  <br>front = $a_1 \wedge$ back = $a_n$ |
| Invariant | $(q.isEmpty() == true \wedge front == back == NIL) \oplus (q = \{a_1 \to a_2 \to \dots \to a_n\} \wedge front == a_1 \wedge back == a_n)$ |
| Procedures | ● **Constructor operations**<br>○ *createQueue()*<br>//Creates an empty queue.<br>//**post:** A queue with no elements inside has been created.<br>● **Analyzer operations** |

- ○ *front()*
  //Returns the element at the front of the queue but does not modify it.
  //**post:** Returns the front element of the queue. If the queue is
  //empty then its front is NIL and this is the retrieval.
- ○ *isEmpty()*
  //Determines whether a queue is empty.
  //**post:** Returns TRUE if the front of the queue is NIL and
  //FALSE otherwise.
- ○ *size()*
  //Returns the size of the queue.
  //**post:** Returns the total number of elements in the queue.

- **Modifier operations**
  - *enqueue(newItem)*
    //Adds newItem to the queue.
    //**post:** newItem is at the back of the queue.
  - *dequeue()*
    //Removes and returns the front element of the queue.
    //**pre:** The queue is not empty.
    //**post:** The front is taken off the queue and the new front is the
    //element that was behind the previously returned element. If
    //the queue is empty, it results in an error.

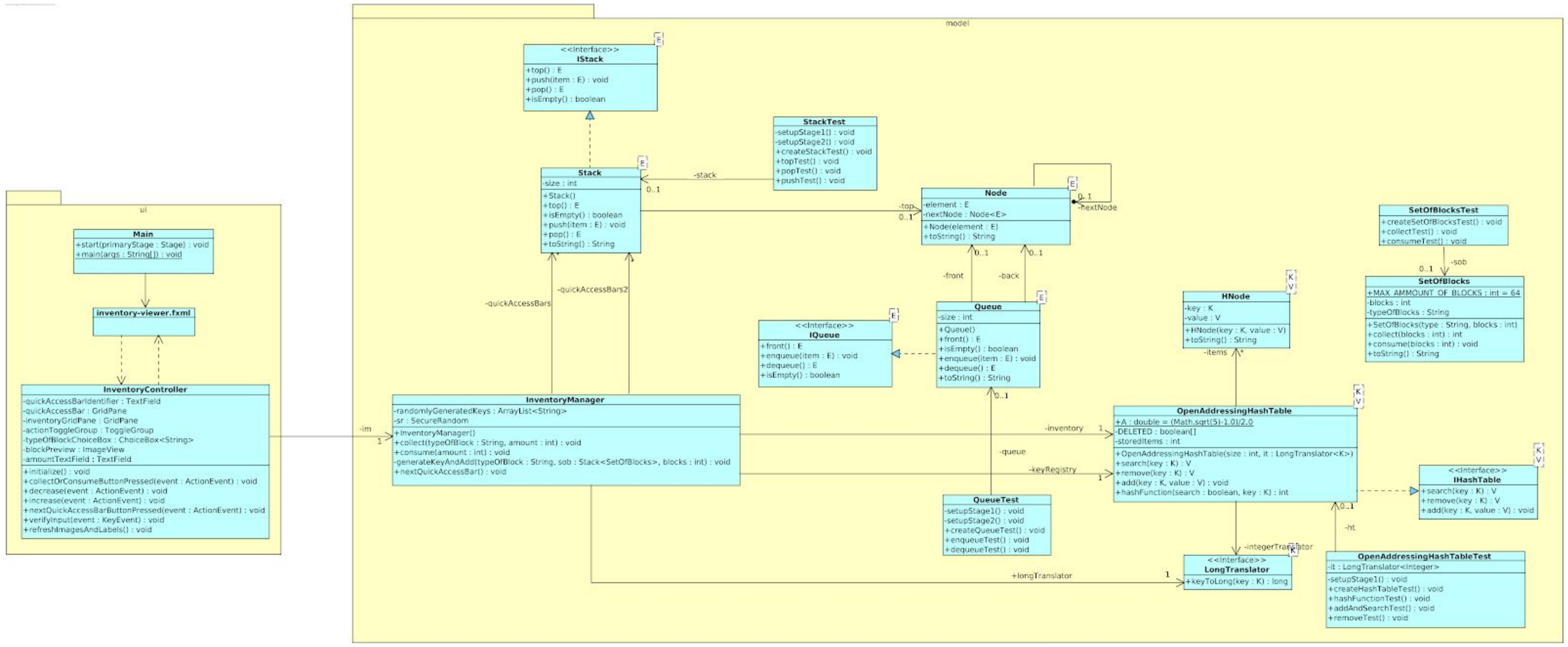
| Axioms of the ADT Queue |
|---|
| 1. (aQueue.createQueue()).size() = 0 |
| 2. (aQueue.enqueue(item)).size() = aQueue().size() +1 |
| 3. (aQueue.dequeue()).size() = aQueue.size() - 1 |
| 4. (aQueue.createQueue()).isEmpty() = TRUE |
| 5. (aQueue.enqueue(item)).isEmpty() = FALSE |
| 6. (aQueue.createQueue()).dequeue() = ERROR |

| **ADT Hash Table** | |
|---|---|
| Definition | The ADT hash table is a sort of dictionary, which means it stores elements in pairs, a search key and a value. Insertion, deletion and retrieval of items is made by search key and not by value. As in dictionaries, there are no two elements with the same search key. A hash table can be empty. The ADT Hash Table is an array of elements together with a hash function and access procedures. The access procedures permit insertion, deletion, and retrieval of an item by means of the hash function. The hash function determines the location in the table for any item, using its search key. The hash function takes a search key and maps it into an |

| | |
|---|---|
| | integer array index. |
| Object Representation |  T<br>0 k1 valueX<br>1 k4 valueY<br>2 k8 valueZ<br>. . .<br>. . .<br>x k3 valueW<br>. . .<br>. . .<br>m-1 k2 valueV |
| Invariant | h: U→T={0,1,2,**...**,m-1} ∧ m = \|T\| = \|U\| |
| Procedures | ● **Constructor operations**<br>○ *createHashTable(size)*<br>//Creates an empty hash table of the specified size.<br>//**post:** An empty hash table has been created.<br>● **Analyzer operations**<br>○ *tableRetrieve(searchKey)*<br>//Retrieves the element corresponding to this searchKey.<br>//**post:** Retrieves the element that matches this searchKey or<br>//NIL otherwise.<br>○ *isEmpty()*<br>//Determines whether the hash table is empty.<br>//**post:** Returns TRUE if it is empty or FALSE otherwise.<br>○ *tableLength()*<br>//Determines the number of items in the table<br>//**post:** It returns the total number of items in the hash table.<br><br>● **Modifier operations**<br>○ *tableDelete(searchKey)*<br>//Deletes the item that has a matching searchKey.<br>//**post:** Deletes the element which search key matches<br>//searchKey and returns it, otherwise returns NIL.<br>○ *tableInsert(newItem)*<br>//Adds newItem to the hash table<br>//**post:** newItem is stored in the position specified by the hash<br>//function. |

| Axioms of the ADT Hash Table |
|---|
| 1. (aHT.createHashTable(m)).tableLength() = 0<br>2. (aHT.tableInsert(item)).tableLength() = aHT.tableLength() +1<br>3. (aHT.tableInsert(item)).tableRetrieve(item.searchKey) = item |

```
4. (aHT.tableInsert(item)).tableDelete(item.searchKey) = TRUE
```


ui
Main
+start(primaryStage : Stage) : void
+main(args : String[]) : void
inventory-viewer.fxml
InventoryController
-quickAccessBarIdentifier : TextField
-quickAccessBar : GridPane
-inventoryGridPane : GridPane
-actionToggleGroup : ToggleGroup
-typeOfBlockChoiceBox : ChoiceBox<String>
-blockPreview : ImageView
-amountTextField : TextField
+initialize() : void
+collectOrConsumeButtonPressed(event : ActionEvent) : void
+decrease(event : ActionEvent) : void
+increase(event : ActionEvent) : void
+nextQuickAccessBarButtonPressed(event : ActionEvent) : void
+verifyInput(event : KeyEvent) : void
+refreshImagesAndLabels() : void
-im
1
model
<<Interface>>
IStack
+top() : E
+push(item : E) : void
+pop() : E
+isEmpty() : boolean
E
Stack
-size : int
+Stack()
+top() : E
+isEmpty() : boolean
+push(item : E) : void
+pop() : E
+toString() : String
StackTest
-setupStage1() : void
-setupStage2() : void
+createStackTest() : void
+topTest() : void
+popTest() : void
+pushTest() : void
-stack
0..1
-top
0..1
Node
-element : E
-nextNode : Node<E>
+Node(element : E)
+toString() : String
0..1
-nextNode
-front
-back
0..1
0..1
-quickAccessBars
-quickAccessBars2
InventoryManager
-randomlyGeneratedKeys : ArrayList<String>
-sr : SecureRandom
+InventoryManager()
+collect(typeOfBlock : String, amount : int) : void
+consume(amount : int) : void
-generateKeyAndAdd(typeOfBlock : String, sob : Stack<SetOfBlocks>, blocks : int) : void
+nextQuickAccessBar() : void
<<Interface>>
IQueue
+front() : E
+enqueue(item : E) : void
+dequeue() : E
+isEmpty() : boolean
Queue
-size : int
+Queue()
+front() : E
+isEmpty() : boolean
+enqueue(item : E) : void
+dequeue() : E
+toString() : String
0..1
-queue
QueueTest
-setupStage1() : void
-setupStage2() : void
+createQueueTest() : void
+enqueueTest() : void
+dequeueTest() : void
-inventory
1
-keyRegistry
1
HNode
-key : K
-value : V
+HNode(key : K, value : V)
+toString() : String
-items
*
K
V
OpenAddressingHashTable
+A : double = (Math.sqrt(5)-1.0)/2.0
-DELETED : boolean[]
-storedItems : int
+OpenAddressingHashTable(size : int, it : LongTranslator<K>)
+search(key : K) : V
+remove(key : K) : V
+add(key : K, value : V) : void
+hashFunction(search : boolean, key : K) : int
<<Interface>>
IHashTable
+search(key : K) : V
+remove(key : K) : V
+add(key : K, value : V) : void
-integerTranslator
<<Interface>>
LongTranslator
+keyToLong(key : K) : long
+longTranslator
1
0..1
-ht
OpenAddressingHashTableTest
-it : LongTranslator<Integer>
-setupStage1() : void
+createHashTableTest() : void
+hashFunctionTest() : void
+addAndSearchTest() : void
+removeTest() : void
SetOfBlocksTest
+createSetOfBlocksTest() : void
+collectTest() : void
+consumeTest() : void
0..1
-sob
SetOfBlocks
+MAX_AMMOUNT_OF_BLOCKS : int = 64
-blocks : int
-typeOfBlocks : String
+SetOfBlocks(type : String, blocks : int)
+collect(blocks : int) : int
+consume(blocks : int) : void
+toString() : String

## Sources.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. Chapters 10 and 11.