**THE DESIGN METHOD IN ENGINEERING**\*

**PHASE 1**

**Problem identification:**

A ground transportation company has a problem when the passengers are going to enter to the bus, because it’s hard to decide who enters first, because that could and delay the travel. In the other hand, there are passengers that could have other qualities for which they should be prioritized, such as age, or having some sort of disability. The rest of the customers could be confused by not knowing in which order to enter, so a determining factor must be established among them, which dictates their order. The same will happen when the exit process initiates. There could be some that consider that if they entered, they should leave in the same way, or others, could think it’s better if the ones that entered last exits first.

|  |  |
| --- | --- |
| Customer | Ground transportation company |
| User | workers |
| Functional requirements | R.F 1 = Load database.  R.F 2 = Organize boarding.  R.F 3 = Assign priority.  R.F 4 = Organize disembarcation. |
| Context of the problem | Defining the order in wich the people should enter the bus might generate some complications. To avoid any kind of trouble, there must be defined a way to enter that acomplish the previous conditions, in the first case, for the entry, the priority of the passenger, his location on the bus and his order of arrival, for departure, take into account its proximity to the door, its proximity to the corridor and ultimately the order of arrival of passengers. |
| Non-functional requirements |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | Load database | | |
| Summary | The system should give the information of all the passengers. | | |
| Inputs | **input name** | **Data type** | **Selection or repetition condition** |
| none |  |  |
| General activities needed to obtain the results | The system will be able to read and interpret a plain text file that will contain passenger information. | | |
| Result or postcondición | The file was successfully read and interpreted | | |
| Outputs | **Output name** | **Data type** | **Selection or repetition condition** |
| errorMessage | String | In case the database is non-existent, or it is not possible to read it. |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | Organize boarding | | |
| Summary | Implement an efficient search algorithm that allows users to find information in the shortest possible time to organize their approach. | | |
| Inputs | **Input name** | **Data type** | **Selection or repetition condition** |
| passenger | Passenger | The passenger must be a passenger of that bus. |
| General activities needed to obtain the results | Filter passengers and their information.  Display the list of passengers. | | |
| Result or postcondición | passenger list | | |
| Outputs | **Output Name** | **Data Type** | **Selection or repetition condition** |
| passengerList | Passenger |  |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | Assign priority | | |
| Summary | Based on the list of passengers, they will be assigned a priority. | | |
| Inputs | **Input name** | **Data Type** | **Selection or repetition condition** |
| passengerList | Array |  |
| General activities needed to obtain the results | * Receive the list with passenger information. * Determine what type of passenger you are. * Conditions such as a passenger require special attention, age or any movement discapasity, each of these will add an additional point. * Ordinate passengers based on their entry priority (with a priority queue). | | |
| Result or postcondición | Leaked passenger list | | |
| Outputs | **Output** | **Data type** | **Selection or repetition condition** |
| pritorityList | Array |  |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | Exit Order | | |
| Summary | Find the best exit order | | |
| Inputs | **Nombre entrada** | **Tipo de dato** | **Condición de selección o repetición** |
| passengerList | Array |  |
| General activities needed to obtain the results | * The exit will be made by the proximity to the door and the corridor. * To avoid disorder, the exit must proceed in order, however, with the same proximity to the corridor, there could be two passengers, so, the one to leave first will be the one that arrived earlier. | | |
| Result or postcondition | Ordered list by exit priority. | | |
| Outputs | **Output name** | **Data type** | **Selection or repetition condition** |
| exitList | Array |  |
|  |

**PHASE 2**

As we know, it is necessary to establish an order of entry and exit.

On a bus, a seat identifier is usually used to say in wich position it is located. the rows are ordered from 1 to N depending on the bus size, from the front to the back. the maximum size of a bus its an articulated bus with 160 seats aprox.

**IMPORTANT CLARIFICATIONS ON THE STATEMENT:**

The order to board the bus will be divided in sections and, in all sections, the order will be the arriving order of the customers.

there are other factors that will add a higher priority, such as: if they require special attention or if they belong to the elderly. in case of presenting two clients with the same priority, the one who arrived before will enter first.

For the departure, the established order will be by its proximity in relation to the door and corridor.

To search for passenger information, the most efficient way possible.

(KeepCoding, 2023)

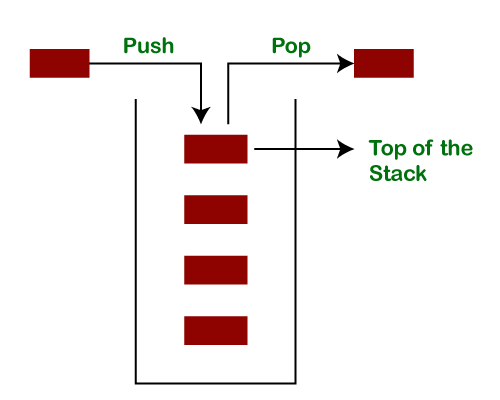
(IONOS, 2020)

Stack

The temporal complexity of the Stack data structure is O(1) for all operations, including inserting and deleting 1 items. This is because Stack is a linear data structure that follows the LIFO (Last In First Out) principle, which means that the last item inserted is the first to come out and vice versa 12.

In terms of implementation, the Stack data structure can be implemented using a matrix or a linked list 1. Implementation based on The linked list does not have a fixed and limited capacity, but it does require more memory due to the additional pointers required to store the 1 items.

[Stack Data Structure - GeeksforGeeks](https://www.geeksforgeeks.org/stack-data-structure/)



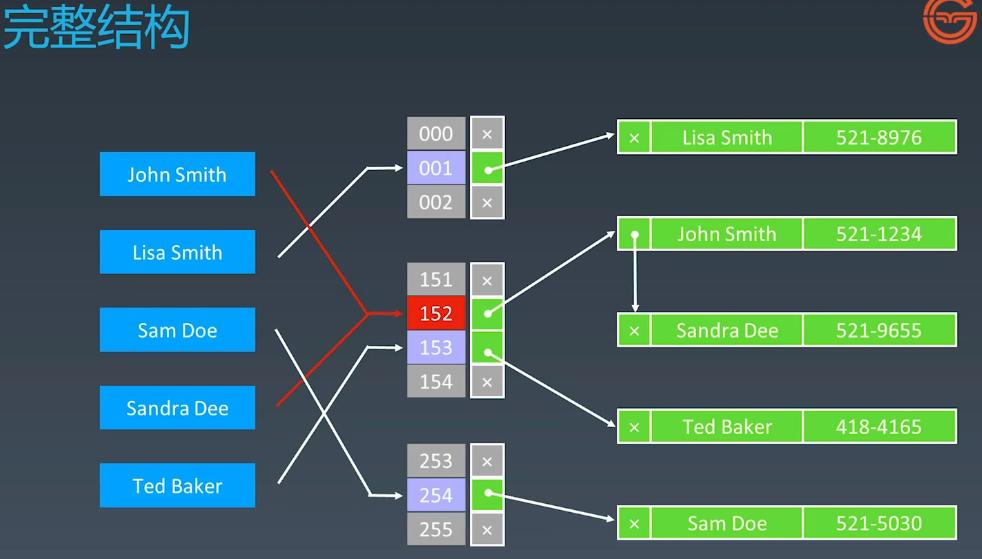
Hash:

On average, if the hash function is well designed, there are very few conflicts and the temporal complexity of the operation is usually O(1) 12. However, if the hash function is not well designed and there are many conflicts, in the worst case, the temporal complexity will degenerate into O(n) 2.

In the case of querying a hash table, four main steps can be identified:

1. Determine the key and calculate the index based on the key.
2. Get the linked list of key-values corresponding to the index position according to the index.
3. Cycle through the linked list of key-values and find the corresponding input key-value pair according to the key.
4. Get the value.

Each of these steps must be O(1) to ensure that the temporal complexity of the HashMap is O(1) 1. However, it appears that the third step has the greatest impact on the temporal complexity of the linked list cycle and the temporal complexity of the linked list 1 search. Therefore, it is necessary to minimize conflicts in the hashing algorithm so that the length of the linked list is as short as possible and its temporal complexity can be guaranteed to be O(1) 1.



**PHASE 3**

**Brainstorming:**

***Spontaneous generation of ideas designed to solve a specific problem.***

**What structures can we use to solve the problem?**

We established that the best data structures we could use were stacks, queues, heaps and hash tables, which are also in line with what we’ve been taught in this semester.

**Passenger departure:**

The departure will be done in relation to their proximity to the door, their proximity to the corridor and ultimately the order of arrival will also be considered. But we have two types of passengers, so, in matters of code, it might be helpful to implement each one in a different stack to establish its differentiation and meet the established criteria.

**PHASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS**

As we can see, if you purchase a premium ticket, elements such as accumulated miles are immediately considered.

However, if you access the normal ticket, no matter how many miles you have accumulated, they simply won't be considered.

Although the ticket would not have attributes beyond the seat, so we could save ourselves the creation of this class and simply perform an enumeration.

**Priority Types:** Each priority equals one point. The passenger will have boolean attributes that will be "unlocked" or accessed only if the passenger buys a VIP ticket. In this case, if those attributes are equal to 'true', this will add points to that passenger.

**Departure:**

It is considered better to separate VIP passengers and regular passengers for departure and place them in separate structures.

Using the same structure for both is discarded.

**Arrival:**

Similarly, to departure, it is considered better to separate VIP ticket passengers and regular ticket passengers for arrival, and the structures to be used will be queues andpriority queues.

Using the same structure for both is discarded.

**FASE 5: Evaluation and selection of the best solution**

**TADs:**

|  |
| --- |
| **TAD STACK** |
|  |
|  |
| **Constructor:**  Stack to Stack  **Modifiers:**  Push to Stack  Popto Stack  **Analyzing:**  isEmptyto boolean  top to element |

|  |
| --- |
| **Stack ()**  Builds an empty stack.  {pre:}  {post: Stack s = Ø} |

|  |
| --- |
| **Push**  Adds the new element *e* to stack *s.*  {pre: Stack s = }  {pos: Stack s = } |

|  |
| --- |
| **Pop**  Extracts from the stack *s*, the most recently inserted element.  {pre: Stack s ≠ Ø}  {post: Stack s = } |

|  |
| --- |
| **Top**  Recovers the value of the element on the top of the stack.  {pre: Stack s ≠ Ø}  {post: } |

|  |
| --- |
| **isEmpty**  Determines if the stack s is empty or not.  {pre: Stack s}  {post: true if *s* = Ø or false if *s* ≠ Ø} |

|  |
| --- |
| **TAD HASH TABLES** |
|  |
|  |
| **Constructor operations:**  HashTable to HashTable  **Modifiers operations:**  Hash to Position  Insert to HashTable, boolean.  Delete to HashTable, boolean  **Analyzing operations:**  Search to boolean |

|  |
| --- |
| **HashTable**  Builds an empty hashTable.  {pre: none}  {post: Hash Table h = Ø} |

|  |
| --- |
| **Hash**  It takes a key as input and returns an index position in the hash table. It allows quick access to hash table elements by obtaining their storage position from their key.  {pre: key ∈ K, K = {k | k is a valid value to be taken as a key} table\_size ∈ Z+ }  {post: Hash Table h = h (key, table\_size) ∈ Z 0 ≤ h (key, table\_size) < table\_size} |

|  |
| --- |
| **Delete**  Is used to delete a key-value pair to the table.  {pre: hash\_table ∈ HashTables ∧ key ∈ K ∧ value ∈ V}  {post: true = if key , if key } |

|  |
| --- |
| **Insert**  Is used to add a new key-value pair to the table.  {pre: HashTable h = key ∈ K: the key is an element of the set K of valid keys.  value ∈ V: The value is an element of the set V of valid values for the hash table.}  {post: ∀ (k', v') ∈ H’: if k' = k, entonces v' = v, if k' ≠ k, entonces (k', v') ∈ H, | H'| = |H| + 1} |

|  |
| --- |
| **Search**  Is used to delete a key-value pair to the table.  {pre: ∀ k ∈ K, tabla\_hash [h (k, table\_size)] = (k, v) → k is a valid key for the hash table and hash\_table is a valid hash table.}  {post: if k ∈ keys(h) -> k, if keys(h) -> null} |

**PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS**

**Test Stack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la Prueba:** Store the database with stack | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado esperado** |
| StackTest | testPush() | data containing keys are created and this one will be the one that will be organized on the stack | key1 = 10;  key2 = 20;  key3 =30;  .push(key1);  .push(key2);  .push(key3); | keys stored on a stack in ascendant order  key1,  key2,  key3 |
| StackTest | testTop() | Shows the key on top | int key = 10;  int key2 = 20;  .push(key) | key2 |
| StackTest | testEmpty()  testException | Validates if the stack is empty and throws an exception | int key = 10;  .push(key)  .isEmpty()    .pop(key) | returns a boolean that shows a positive result, implying that the empty and exception work as intended. |

**PHASE 7: DESIGN IMPLEMENTATION**

**The following link contains the implementation in code of the analysis done in this document: https://github.com/Tudibyte/estructurasT1**