Data Structure

First Assignment

Lists,

Queues

and stacks.

Alejandro Martin Sánchez – 03203201Z

Carlos Garrido Junco – 02570033J

UAH 2021-2022|Computer Science|

INDEX

ANALYSIS

In this section, we are going to explain the requirements for this assignment. It is practically based on the management and implementation of stacks, queues and list. The user will enter two numbers to calculate a set of numbers >100.000 that will be introduce in each data structure. Then the program is going to display five options to the user to choose.

1. ADTs specifications.

In this section, the abstract data structures will be explained, as well as its methods in a reduced way.

As we said, we need three types of ADTs for making the program. These are stacks, queues and lists (stacks and queues are derived from lists).

Stack will have a LIFO(Last in-First out) behavior, queue will have a FIFO (First in-First out) behavior and finally list can insert an element in any position of the list and also take an element in any position of the list.

This is a small outline of the three ADTs specifications:

Spec *STACK* [INTERGER]

Genre stack, integer

Spec *QUEUE* [INTERGER]

Genre queue, integer

Spec *LIST* [INTERGER]

Genre list, integer

2. Definition of operations of the ADT.

To fit the requirements we add extra operations in some of the ADTs that will be explained in this section as well as the basic operations of the used ADTs.

Stack methods:

- Top: returns the top element of the stack that is, the las element introduced in it. Does not erase from the stack. No arguments are passed.

- Push: insert a new item into the stack. The item is passed as an argument and placing it at the top position. No return value.

- Pop: retrieves the top element of the stack (las element introduce in it) and delete it from the stack. Return the delete item.

- Empty: no arguments passed, returns true if the stack has no element, otherwise false.

- Makenull: deletes all the elements of the stack in order to make an empty stack. No arguments are passed. No return value.

- SortStack: Sort the elements of the stack. Has a pointer to a stack like arguments. No return value.

Queues methods:

- Enqueue: inserts a new element into the rear of the queue. The element is passes as an argument. No return value.

- Dequeue: returns the front elements of the queue and delete it, this element was the first inserted. No arguments.

- Front: returns the first element of the queue without remove it. No arguments.

- Makenull: deletes all the elements of the queue in order to make an empty queue. No arguments. No return value.

- Empty: returns true if the queue has no element, otherwise false. No arguments.

- SortQueue: sort the elements of the queue. Has a pointer to a stack like arguments. No return value.

Lists methods:

- Insert: insert a new element at the front of the list. The elements is passed as an argument. No return value.

- ListDelete: finds and delete an element. The element is passes as an argument. No return value.

- ListLenght: return an integer with the length of the list. No argument.

- Positioninsert: insert a new element in a position. Two arguments the new element and the position. No return value.

- SortList: sort the elements of the list. No return value.
- Empty: return true if the list has no elements, otherwise return false. No arguments.
- MakeNull: delete all the elements of the list. No return value. No arguments.

Now to finish the specifications these diagrams are shown:

```
Spec STACK [INTERGER]
genre stack, integer
operations
        pop:     stack -> item
        push:    stack item -> stack
        top:     stack -> item
        makenull: stack -> stack
        empty: stack -> Boolean
        SortStack: stack -> stack
```

```
Spec QUEUE [INTERGER]
genre queue, integer
operations
        enqueue: queue item -> queue
        dequeuer: queue -> item
        front: queue -> item
        makenull: queue -> queue
        empty: queue -> Boolean
        SortQueue: queue -> queue
```

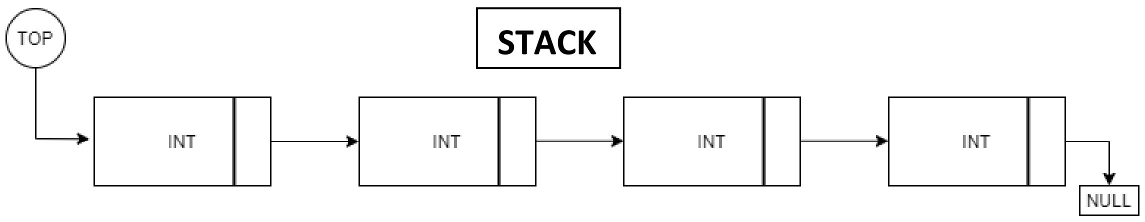
```
Spec LIST [INTERGER]
genre queue, integer
operations
        Insert: list item -> list
        listlenght: list -> integer
        lisdelete: list item -> list
        positioninsert: item position -> list
        SortList: list -> list
        empty: list->Boolean
        makenull: list -> list
```

DESIGN

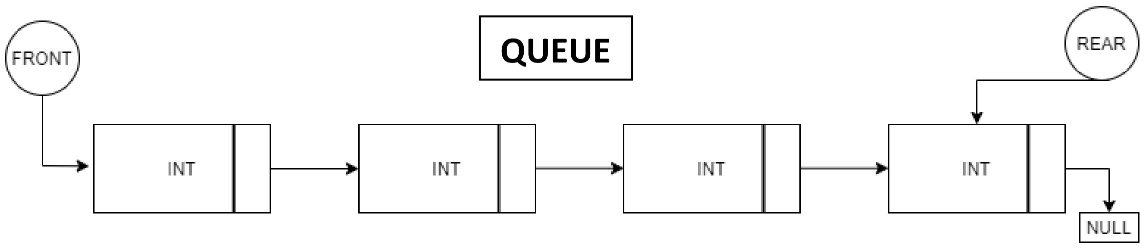
Once the specifications are shown, we go on to show the design of the application. As well as the diagrams and a representation of how the ADT is stored in memory. Also in this section the classes are explained.

1. *Diagram of the representation of the ADT in the memory of the computer.*

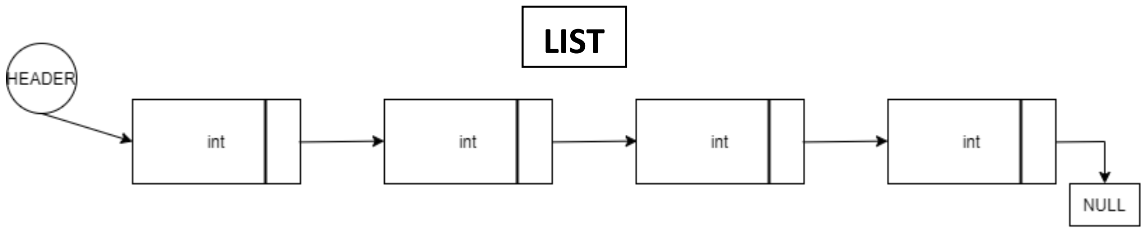
For this, we are going to use box diagrams, which are a simple way to show how the ADTs are stored in the memory of the computer. The arrows represents pointers and the boxes represent an element made by an integer and a pointer.



This box diagram represents the stack ADT. It has a top pointer where deletions and insertions are done.



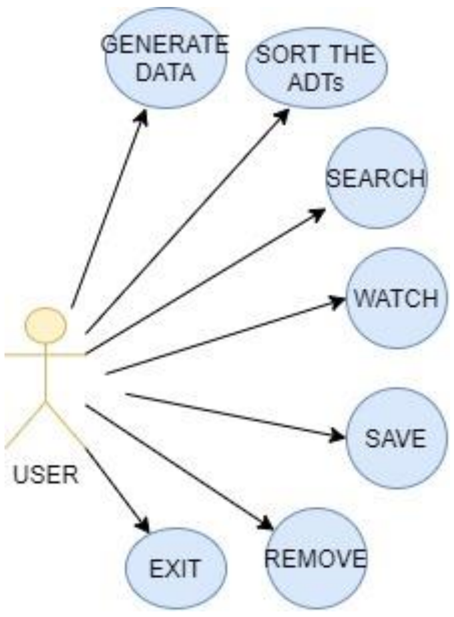
This box diagram represents the queue ADT. It has two pointers, one pointing to de front of the queue and the other pointing to the rear. Insertions are made in the rear and deletions are made in the front.



This last box diagram represent a list. It has a similar structure with the stack it has also one pointer, but this time call header. The difference is that deletions and insertions can be done in any position.

2. *UML diagram and class diagram.*

Through the following figures, we are going to see in a simple way the structure of the program and its modes of use.

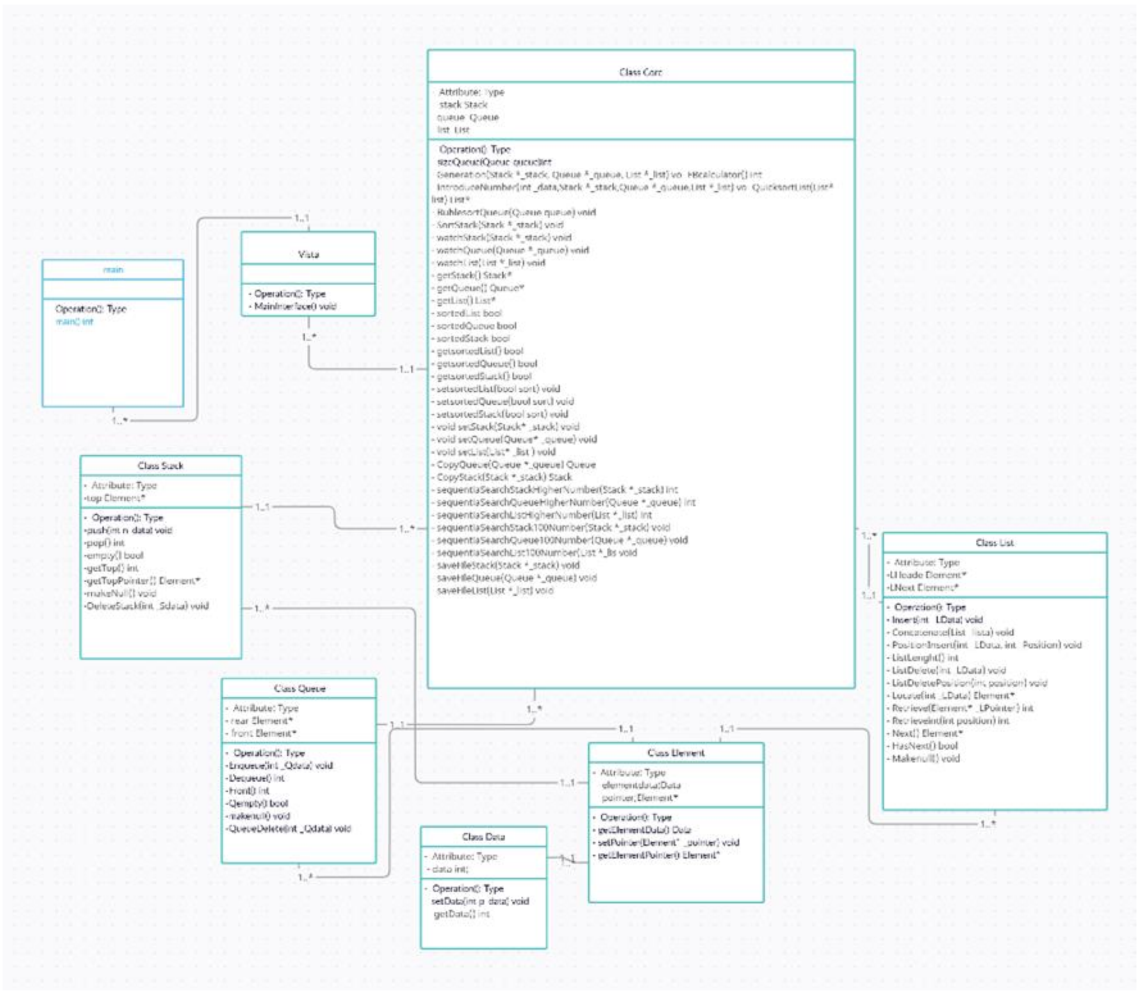


This diagram shows the actions a user can do when the program is running.

First, the program will generate data base in two natural numbers between one and one hundred entered by the user.

Once the data is generate, will appear a menu with a user interface with five options (Sort the ADTs, add data to each ADTs, search, remove data from the ADTs, watch the ADTs, save file and exit). All these actions will be explained later in more detail.

These figure represents the class diagram.


Class Core
Attribute: Type
stack Stack
queue Queue
list List
Operation(): Type
sizeQueue(Queue queue)int
Generation(Stack *_stack, Queue *_queue, List *_list) vo HBcalculator() int
IntroduceNumber(int _data,Stack *_stack,Queue *_queue,List *_list) vo QuicksortList(List* list) List*
- BublesortQueue(Queue queue) void
- SortStack(Stack *_stack) void
- watchStack(Stack *_stack) void
- watchQueue(Queue *_queue) void
- watchList(List *_list) void
- getStack() Stack*
- getQueue() Queue*
- getList() List*
- sortedList bool
- sortedQueue bool
- sortedStack bool
- getsortedList() bool
- getsortedQueue() bool
- getsortedStack() bool
- setsortedList(bool sort) void
- setsortedQueue(bool sort) void
- setsortedStack(bool sort) void
- void setStack(Stack* _stack) void
- void setQueue(Queue* _queue) void
- void setList(List* _list) void
- CopyQueue(Queue *_queue) Queue
- CopyStack(Stack *_stack) Stack
- sequentiaSearchStackHigherNumber(Stack *_stack) int
- sequentiaSearchQueueHigherNumber(Queue *_queue) int
- sequentiaSearchListHigherNumber(List *_list) int
- sequentiaSearchStack100Number(Stack *_stack) void
- sequentiaSearchQueue100Number(Queue *_queue) void
- sequentiaSearchList100Number(List *_lis void
saveFileStack(Stack *_stack) void
saveFileQueue(Queue *_queue) void
saveFileList(List *_list) void
main
Operation(): Type
main() int
Vista
- Operation(): Type
- MainInterface() void
Class Stack
- Attribute: Type
-top Element*
- Operation(): Type
-push(int n_data) void
-pop() int
-empty() bool
-getTop() int
-getTopPointer() Element*
-makeNull() void
-DeleteStack(int _Sdata) void
Class Queue
- Attribute: Type
- rear Element*
- front Element*
- Operation(): Type
-Enqueue(int _Qdata) void
-Dequeue() int
-Front() int
-Qempty() bool
-makenull() void
-QueueDelete(int _Qdata) void
Class Data
- Attribute: Type
- data int;
- Operation(): Type
setData(int p_data) void
getData() int
Class Element
- Attribute: Type
elementdata:Data
pointer;Element*
- Operation(): Type
- getElementData() Data
- setPointer(Element* _pointer) void
- getElementPointer() Element*
Class List
- Attribute: Type
-LHeade Element*
-LNext Element*
- Operation(): Type
- Insert(int _LData) void
- Concatenate(List _lista) void
- PositionInsert(int _LData, int _Position) void
- ListLenght() int
- ListDelete(int _LData) void
- ListDeletePosition(int position) void
- Locate(int _LData) Element*
- Retrieve(Element* _LPointer) int
- Retrieveint(int position) int
- Next() Element*
- HasNext() bool
- Makenull() void
1..1
1..*
1..1
1..*
1..1
1..*
1..*
1..*
1..1
1..*
1..1
1..1
1..1
1..1
1..1
1..*
1..1
1..*

3. Explanation of the ADTs clases.

In this section we will explain the methods for each ADT. In some methods you will find more boxes diagrams to show what happens when that method is executed.

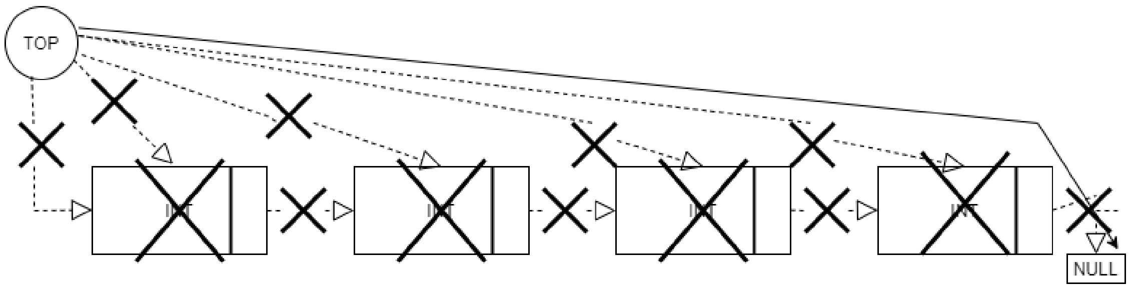
All of the three linear ADTs have a very similar structure that's is because stacks and queues are special cases of lists:

- Stacks: presents a LIFO behavior that is the last element to enter in the stack is the first element that goes out. Like a list it has one pointer pointing to the top call 'Top'.
- Queues: presents a FIFO behavior that is the first element to enter is the first element to go out. It has two pointer one pointing at the first element called 'Front' and one pointing at the last element called 'Rear'.

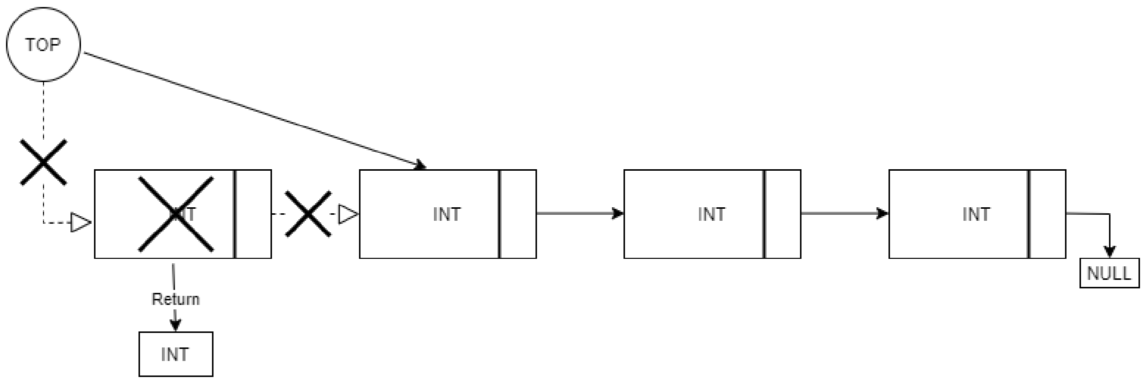
4. Explanation of the ADTs methods.

Stacks methods:

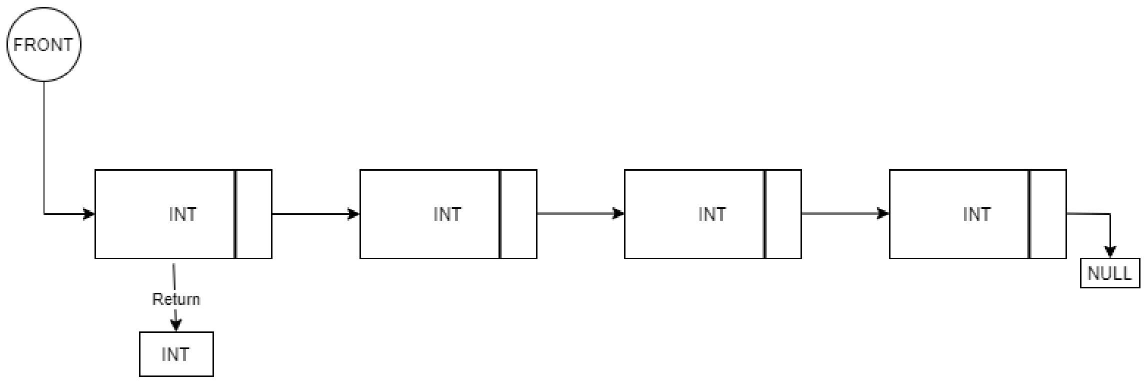
- Makenull(): Deletes all the elements and makes top point to null. This is done with a loop until de stack is enmpty.



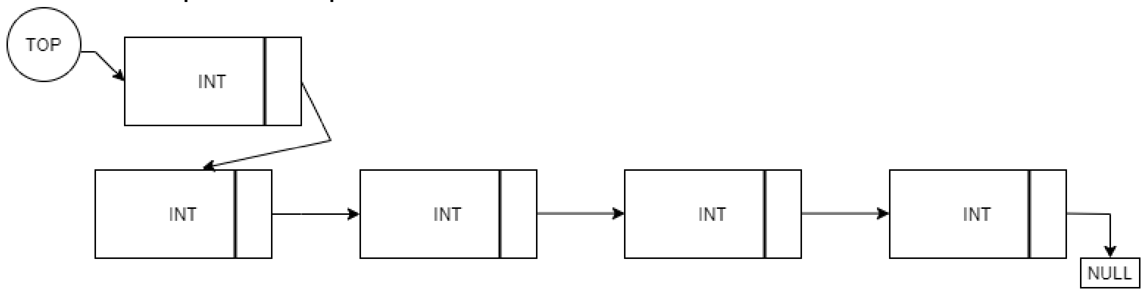
- Pop():Returns the value of the element pointed by top and erase that element. Then, top will point to the next element.



- Top():returns the value of the element pointed by top without delete it.



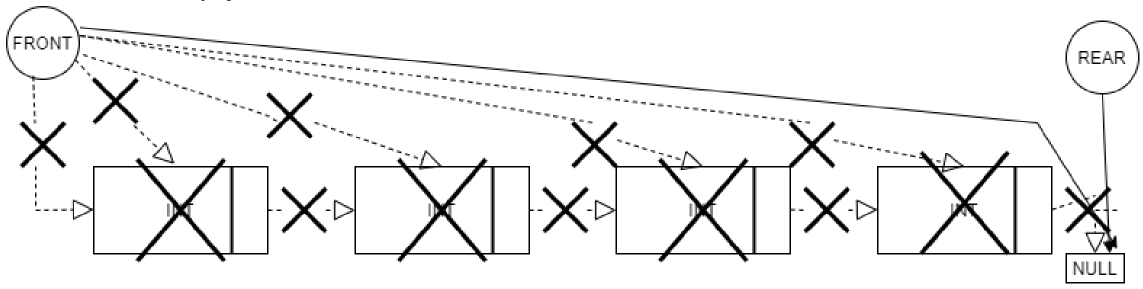
- Push(): create a new element with the value given as an argument to the method. The pointer of this new element will have the memory address of the top and then top passes to point the this new element.



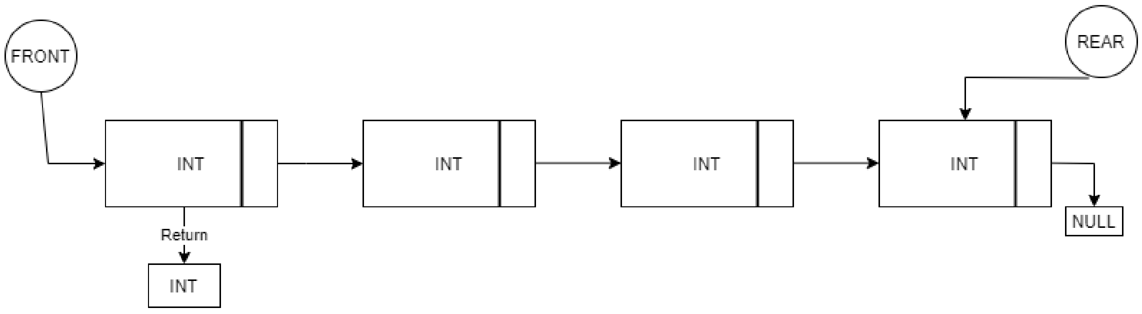
- Empty(): compare it top is equal to null and returns true if it is and false if it is not.

Queues methods:

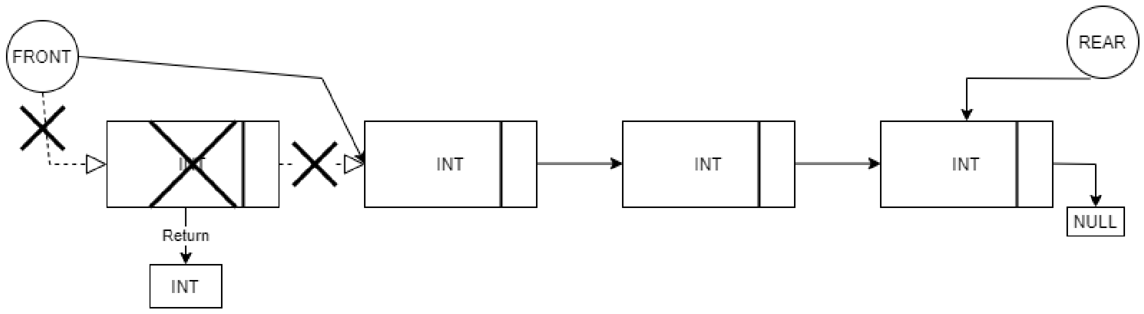
- Makenull(): Deletes all the cell of the queue and points front and rear point to null. The deletion is done until de queue is empty.



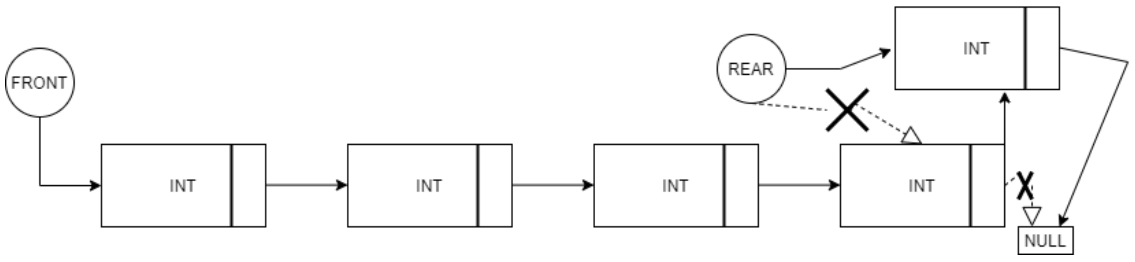
- Front(): this method returns the value of the element pointin by front without delete it.



- Dequeue(): returns the value of the element pointing by front. Then front point to the next element and the first element is deleted.



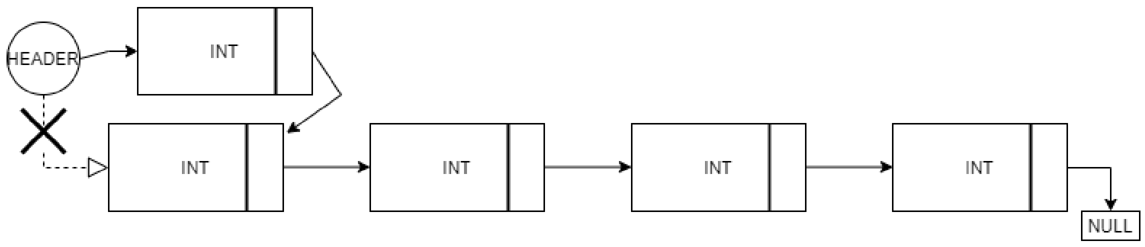
- Enqueue(): Create a new element with a value given as an argument to the method. The pointer of this new element will point to null and the pointer of the cell pointing by rear will have the direction of the new element then rear will point to the new element.



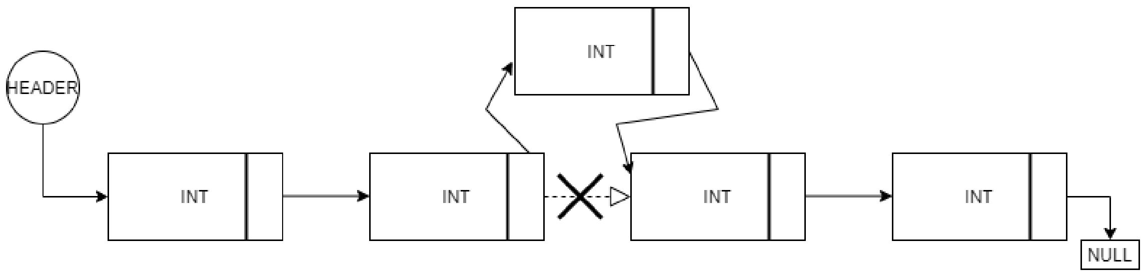
- Empty():compare if the front and the rear are pointing to null it they are it will return true otherwise will return false.

Lists methods:

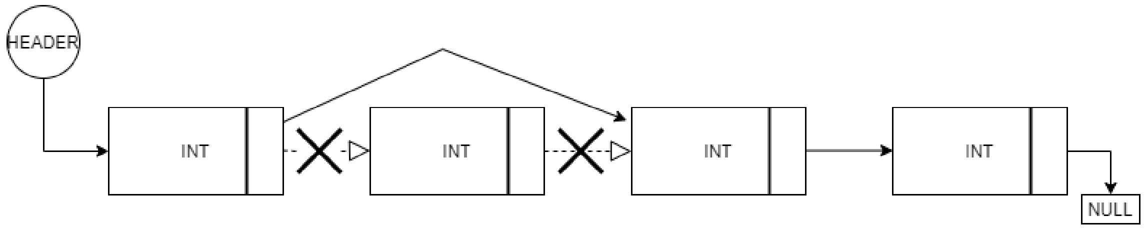
- Insert(): create a new element with a value given as an argument to the method. The pointer of this new element will point at the same position as the header an then the header will point to new element.



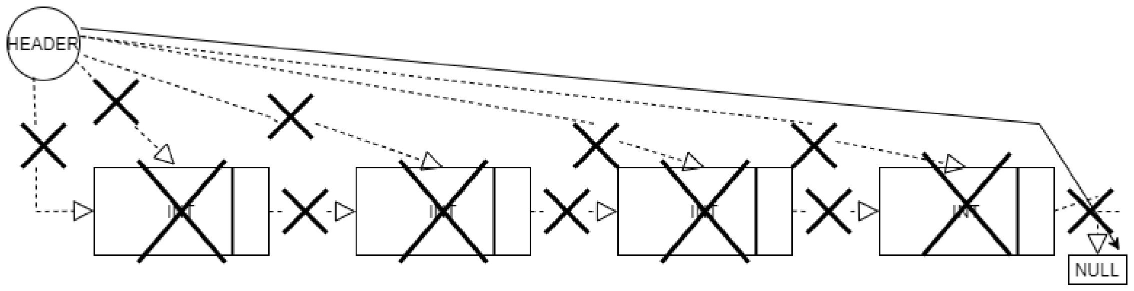
- PositionInsert(): create a new element with a value given as an argument to the method. The position to insert also will be given as an argument. The pointer of the new element will be the pointer of the previous element and the pointer of the previous element will point to the new element.



- Delete(): it will return the value of the element in the position given as an argument. Then this element is deleted.



- Makenull(): this method will delete all the elements until the header is pointing to null.



- Empty(): return true if the header is equals to null else return false.

5. *Explanation of the behavior of the program.*

This section will explain how the behavior of the program is. When the program it initialize create an empty stack, queue and list. Then the user is asked to introduce two numbers to generate an amount of numbers > 100.000 then this numbers are load into the queue the list and the stack and generate a file with the numbers.

After the number generation process, the program will display a menu with the following options:

1. Sort the ADTs: When this option is selected, will call to three methods (one for each ADT) and sort the numbers. Also generate a file but this time with all the numbers sorted.

2. Search option: this option will show in the display the maximum value of all the numbers generated as well as the 100 lower values of this numbers.

3. Watch data: This option will display the content of one of the ADTs selected

4. Remove data number: this option will ask the user to introduce a number and then will find it in the ADTs and delete it.

5. Add data number: this option will ask the user to introduce a number and then this number will be inserted in each ADT.

6. Save in file: this option will save the ADT in a text file.

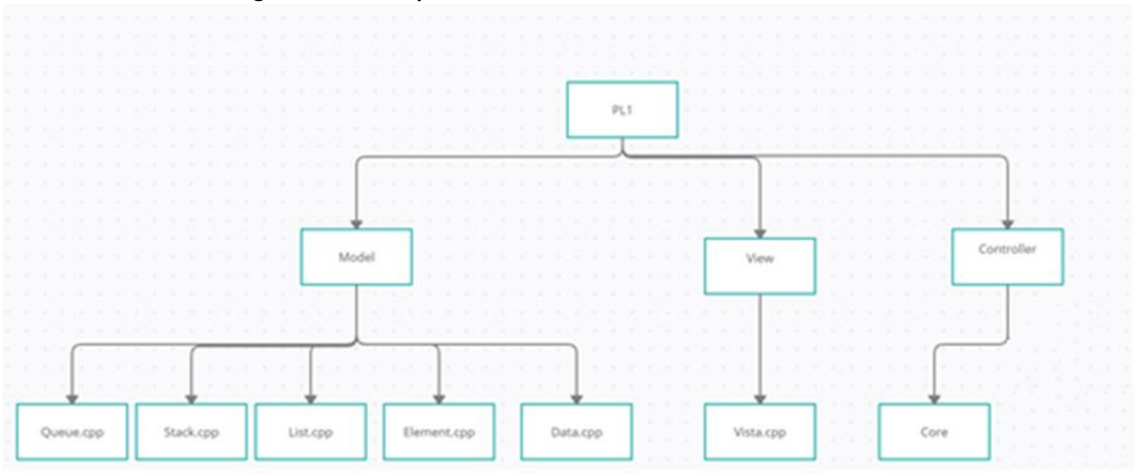
7. Exit: when the user make this action it will automatically exit the program.

All these options will be more developed in this document.

IMPLEMENTATION

At this point, the project will be discussed in general terms as well as its architecture as well as the problems in general that have been encountered.

The project follows an MVC architecture pattern (model view controller), we have chosen such a pattern since it allows us to separate the logic and view layer from the model. Which we will explain later. We have also selected c ++ as the language to program the application, because it is an object-oriented language, which will allow us to model our stacks, queues and lists in classes, as well as allowing us to use pointers.



Our model will be the classes List.cpp, Stack.cpp, Queue.cpp, Element.cpp, Data.cpp. In them come the general methods of a list, a stack and a queue, except in the Element class and Data, which come with their getters and setters, so that they can be accessed through them. In addition, in the Element class has an element of type Data and it has a pointer of type Element which will allow us to access the next element. In the model part, the problems that we have encountered have been how to connect the different elements in addition to doing the general methods of the aforementioned data structures.

In our view, made up of the view.cpp class, it is the one that will be in charge of the visual part of the interaction between the user and the control or logic part, as well as including the required menu, plus some actions that have been added because They were considered necessary to improve the user experience, as an option to be able to exit from each menu, an option to be able to choose which data structure to sort, and search in it. In addition to the option of being able to save each data structure in a file. The problems that we have found in this section have been rather, in how to approach the application so that it is the easiest to use for a user.

Finally, there would be the control layer composed of the core.cpp class, it is where the entire logic section is located, which will function as an intermediary between the model and the visual layer. In it you will find the algorithms for searching, saving, ordering, delete duplicates and accessing the different data structures of the project, which we will explain later. In this section, the biggest problems we have had, have been the memory access while the different elements are ordered and saved, as well as the design, development and implementation of the ordering algorithms such as search algorithms.

For the search algorithms we have chosen sequential search, since, although the binary search is worse, the binary search is worse since it has a complexity of O (n) vs a complexity O (logn). In the requirements it was stipulated that the search had unordered structures, in addition to the fact that said structures both the stack and the queue only allow sequential access, therefore the binary search in these two would not have served us.

As for the search algorithms we have used Quicksort and Bublesort, For the queue and the stack we have used Bublesort since it can be implemented in a simple way since we are going to compare two by two in inserted, in the stack using an auxiliary stack and in the queue at the end of it. As for the list we have selected Quicksort, since being a list It has indexed access, so it can be erased more easily and recursively divided the list, so its complexity will be O (nlogn) vs Bublesort, which has a complexity O (n2);

Now we are going to expand the explanation of the menu’s options.

1. Sort the ADTs:

   This method as said, sorts the ADTs from the lowest value to the max value. We have created one method for each ADT all implemented in the Core.

   -QuickSortList(): This method in resum is “divide and conquer”. We choose one element of the list and call it pivot, then we divide the original into 3 different list. The first one contain all the elements that are <= pivot. The second one contains the pivot. The third one contains de elements >=v. The we make recursion over the first and the third list. Finally when all the elements are sorted we concatenate the three lists.

   -BubbleSortQueue(): This method compare one element of the queue with the next one. If the first element is lowest the next one we dequeuer the first element and enqueuer it in an auxiliary queue. This is done until all the elements are sorted.

-SortStack(): this method to sort a stack need an auxiliary stack. If the top element of the auxiliary stack is lower than the top element of the stack, we push the element into the stack. Once we finish ordering we obtain the auxiliary stack but its sorted from higher to lowest, but like stack reverse the order we pop() the auxiliary stack and push() into the original stack to reverse the order and obtain the sorted stack.

2. Search option:

   For this method, we have also implemented one function for each ADT. However, all of them have the same methodology. With the specifics method of each ADT the compare all the numbers and save the bigger one. It push, dequeu, or return the element and compare with the next one and save the bigger one. Then all this elements are push, enqueuer or insert into an auxiliary stack, queue or list.

3. Watch Data:

   This method goes through the list get the element display it and insert() the element. With the queue, dequeue() the element display it and enqueuer() it. With the stack, pop() the element display it and push() it.

4. Remove Data:

   For the list, when the value to delete is found it delete the element and the pointer of the previous element point now to the next element we use the ListDelete() method.

   For the Queue, we need an auxiliary queue. First we dequeue() the elements of the original compare with the element to delete and enqueue() into the auxiliary queue, if we found the element we stop dequeue() and we dequeue() the elements of the auxiliary queue and enqueue() into the original queue. We called this method QueueDelete().

   For the stack, we need an auxiliary stack. First we pop() the elements of the original stack and compare it with the element to delete and push() it to the auxiliary stack, if we found the element we stop pop() and we pop() all the elements of the auxiliary stack and push() it the original stack. We call this method DeleteStack().

5. Add data:

   We introduce the value given by the user with the default methods of each ADT. Push() for stacks, enqueuer() for queues and insert() for list.

6. Save file:

   For this methods we need to include the <fstream> library to work with files.

   We create three methods saveFileStack(), SaveFileQueue() and saveFileList().

   saveFileStack(): pop() all the elements, save them into the file, push() them into the satck, this is made with a for loop until the length of the satck.

   saveFileQueue(): dequeue() all the elements, save them into the file, enqueue() them into the queue, this is made with a for loop until the length of the queue.

   saveFileList(): return() all the elements, save them into the file, insert() them into the list, this is made with a for loop until the length of the list.

REVIEW

To finish in this section we are going to talk about the running time of the operations and methods used in each class.

1. ***Running time of operations.***

Stack.cpp:

- Push() -> O(1)
- Pop() -> O(1)
- Empty() -> O(1)
- getTop() -> O(1)
- makeNull() -> O(n)
- DeleteStack() -> O(n)

Queue.cpp:

- Enqueue() -> O(1)
- Dequeue() -> O(1)
- Front() -> O(1)
- Qempty -> O(1)
- Makenull() -> O(n)
- QueueDelete() -> O(n)

List.cpp:

- Insert() -> O(1)
- Concatenate -> O(n)
- ListLenght -> O(n)
- ListDelete -> O(n)
- Makenull() -> O(n)
- Lempty() -> O(1)
- PositionInsert() -> O(n)
- ListDeletetePosition -> O(n)

Core.cpp

- Generation() -> O(n)
- SortStack() -> O(n^2)
- SortQueue() -> O(n^2)
- QuickSortList() -> O(n log n)
- sequentiaSearchStackHigherNumber -> O(n)
- sequentiaSearchQueueHigherNumber -> O(n)
- sequentiaSearchQueueHigherNumber -> O(n)
- sequentiaSearchStack100Number -> O(n^2)
- sequentiaSearchQueue100Number -> O(n^2)
- sequentiaSearchList100Number -> O(n^2)

BIBLIOGRAPHY

-Subject documents

-Book: Data Structures and Algorithms.

-Webs:

www.stackoverflow.com

www.cplusplus.com

www.docs.microsoft.com