Spring assignment

Laboratory Assignments

Data Structures

Fall 2020

Universidad de Alcalá de Henares
Lucas Jorge Espejo – 01674654R

INDEX:

1. ANALYSIS

The program has highly detailed comments on each method of each class. All the operations that have had a minimum use of complexity have been commented both on the functionality of the function as well as on its sentences and calls that it carries by the function.

A) Data Flow

Once the program is run, the first part is the user interface where we would have three methods for printing in the different operations it can make, collect the option the user chose and run it.

The user will have six operations: simulate the user arriving 1h, search a phone number, list of 100 phone-IDs in the tree, efficiency comparison, 555 printing and exit the program.

The data input is made at UserInterface class, in addition at this class is where it would print and also where we call the Logic class. In the UserInterface class beside what we said earlier, we have five methods which are the ones that made the calls to different methods inside Logic class. Finally in this class will be three functions for printing the different structure types (printList, printStack, printUserData).

We must remember that if the user introduces not admitted or incorrect values, the program will notify the user of the mistake. That is why we need an exception method that works correctly.

Each class is managed by the Logic class, which also manages the program. That means that Logic class commands over every remaining classes, except UserInterface. Logic class follows the orders of UserInterface, which is the one that chose what to do. In addition Logic class will make most of the program complex operations.

The first thing we will do is create our first UserData class because to collect a phone, this phone will have an ID, a MMR and phone numbers. This class will have 3 variables that will keep each ID, MMR and phone numbers.

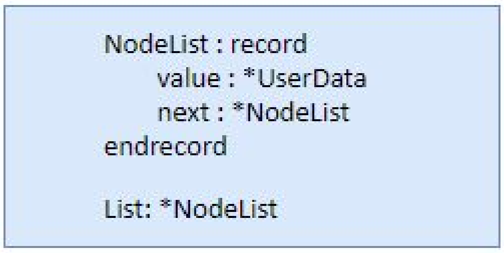
B) ADT Specifications

B.1) Explanation of every ADT and adaptations.

The adaptation of the original data structures to the statement has been carried out as follows:

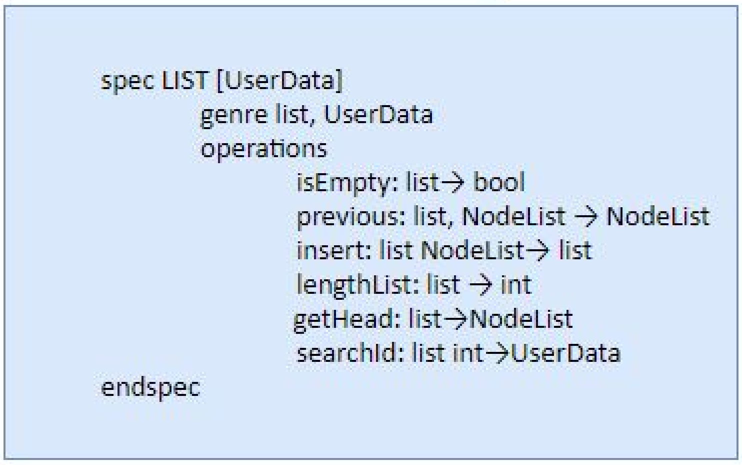
- The types of data that are stored in a List are UserDatas, in the images below we detail what each nodelist is made of and the functions or methods of the List
- The types of data that are stored in a Queue are int and char, we explain it in detail below.
- In the stack the types of data that are stored are UserDatas, in the images below we detail what each nodestack is made of and the functions or methods of the Stack
- In the Binary Tree the data types that are stored are UserDatas, in the images below we detail what each nodeBinaryTree and the functions or methods of the BinaryTree are made of.

Specifications and datatypes: (List, Stack, Queue, Binary Tree)



```
NodeList : record
      value : *UserData
      next : *NodeList
endrecord

List: *NodeList
```



```
spec LIST [UserData]
        genre list, UserData
        operations
                isEmpty: list→ bool
                previous: list, NodeList → NodeList
                insert: list NodeList→ list
                lengthList: list → int
                getHead: list→NodeList
                searchId: list int→UserData
endspec
```

```
NodeStack : record
      fo : *UserData
      next : *NodeStack
endrecord
Stack : *NodeStack
```

```
spec STACK [UserData]
genre stack, UserData
          Operations
                    isEmpty: stack → bool
                    push: stack, NodeStack→ stack
                    pop: stack → UserData
                    getTop: stack → UserData
                    getTopOfStack: stack→ NodeStack
                    getLenght: stack→ int
                    searchId: stack int,stack→UserData
endspec
```

```
NodeQueue : record
      value : *UserData
      next : *NodeQueue
endrecord
Queue : *NodeQueue
```

```
spec QUEUE [char,int]
genre queue, char, int
          Operations
                    frontChar: queue → char
                    frontInt: queue → int
                    enqueue: queue, char→ queue
                    enqueue: queue, int→ queue
                    dequeueInt: queue → int
                    dequeueChar: queue → char
                    lengthQueue: queue → int
                    isEmpty: queue → bool
endspec
```

```
NodeBinaryTree : record
        value : *UserData
        parent : *NodeBinaryTree
        leftchild: *NodeBinaryTree
        rightchild: *NodeBinaryTree
endrecord

BinaryTree: *NodeBinaryTree
```

```
spec BINARYTREE [UserData]
        genre BinaryTree, UserData
        operations
                searchId: binaryTree int, NodeBinaryTree, UserData→BinaryTree
                search: binaryTree int, NodeBinaryTree→ UserData→
                cutNumber: binaryTree int,int → int
                isLeaf: binaryTree NodeBinaryTree→bool
                preorder: binaryTree NodeBinaryTree, List→BinaryTree
                getRoot: binaryTree→ NodeBinaryTree
                setRoot: binaryTree NodeBinaryTree→ BinaryTree
                recover: binartyTree NodeBinaryTree, List, Stack → BinaryTree
                insert: binaryTree NodeBinaryTree, UserData→ NodeBinaryTree
                preorderList: binaryTree int, NodeBinaryTree→ BinaryTree
                getLenght: binaryTree→int

 endspec
```

B.2) Dynamic implementation vs Static implementation.

STACK

STATIC IMPLEMENTATION	DYNAMIC IMPLEMENTATION
pop() → O(1)	pop() → O(1)
push(element) → O(1)	push(element) → O(1)
getTop() → O(1)	getTop() → O(1)
isEmpty() → O(1)	isEmpty() → O(1)

QUEUE

STATIC IMPLEMENTATION	DYNAMIC IMPLEMENTATION
dequeueChar() → O(1)	dequeueChar() → O(1)
dequeueInt() → O(1)	dequeueInt() → O(1)
enqueue(element) → O(1)	enqueue(element) → O(1)
enqueue(element) → O(1)	enqueue(element) → O(1)
frontInt() → O(1)	frontInt() → O(1)
frontChar() → O(1)	frontChar() → O(1)
isEmpty() → O(1)	isEmpty() → O(1)
lengthQueue() → O(1)	lengthQueue() → O(1) * *1Note*

LIST

STATIC IMPLEMENTATION	DYNAMIC IMPLEMENTATION
insert(element) → O(1)	insert(element) → O(1)
previous() → O(n)	previous() → O(n)
empty() → O(1)	empty() → O(1)
lengthList → O(1)	lengthList → O(1) * *1Note*
getHead → O(1)	getHead → O(1)

BINARY TREE

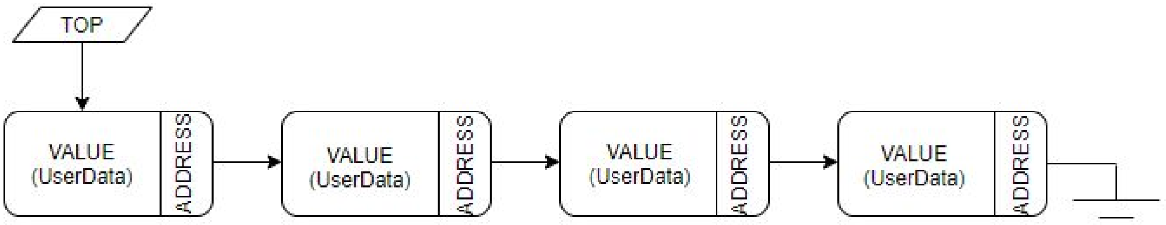
STATIC IMPLEMENTATION	DYNAMIC IMPLEMENTATION
insert(element) → O(n)	insert(element) → O(log(n))
search(element) → O(n)	search(element) → O(log(n))
searchId(element) →O(n)	searchId(element) → O(n) * *2Note*
getRoot → O(1)	getroot → O(1)
setRoot → O(1)	setRoot → O(1)
getLength → O(1)	getLength → O(1) * *1Note*

Note 1: getLength() have a running time of O(1) because take advantage of the insert() function, increasing by one each time that an element is inserted at the data structure.
Note 2: searchId have a execution time of O(n) because the BinaryTree use phoneNumber as a key so that we have to search the id using linear searc

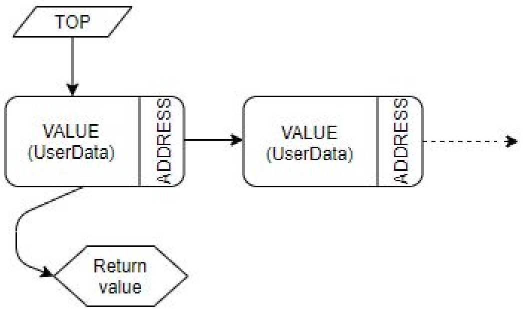
2. DESIGN

A)Box diagrams, explanation of operations.

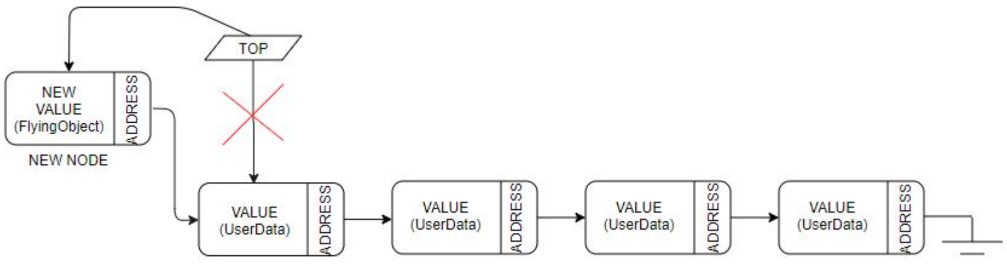
STACK



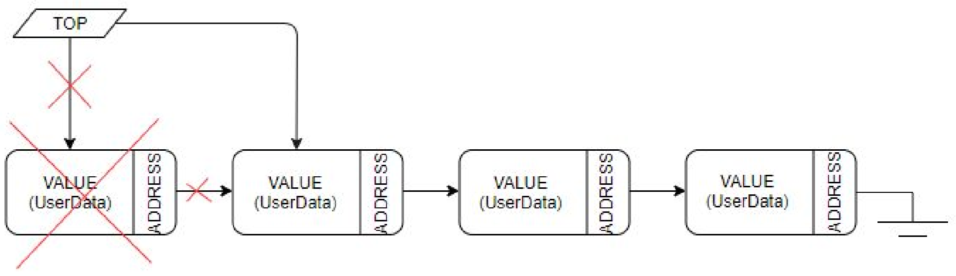
top()



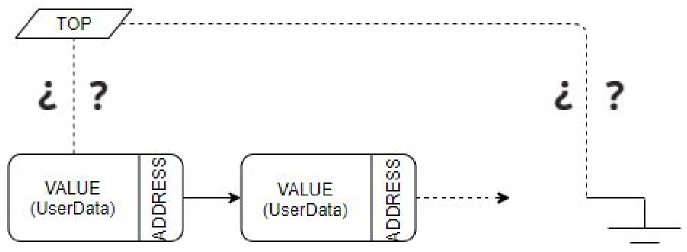
push()



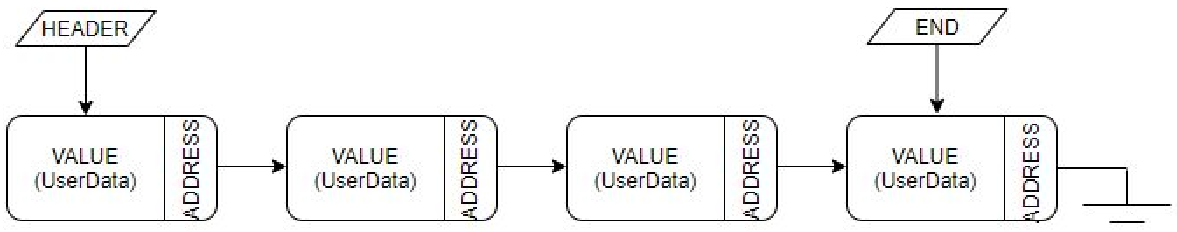
pop()



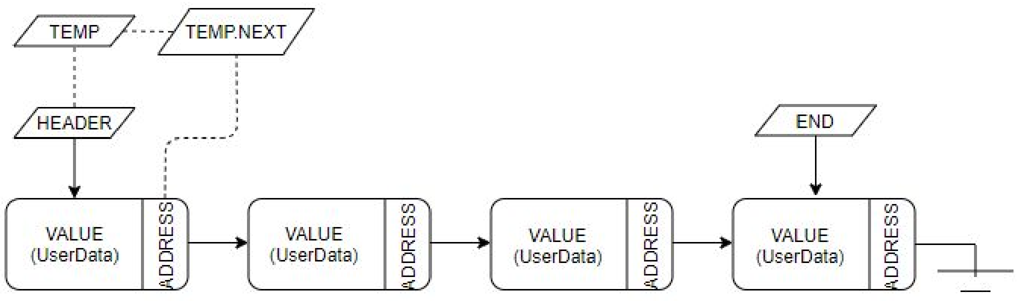
isEmpty()

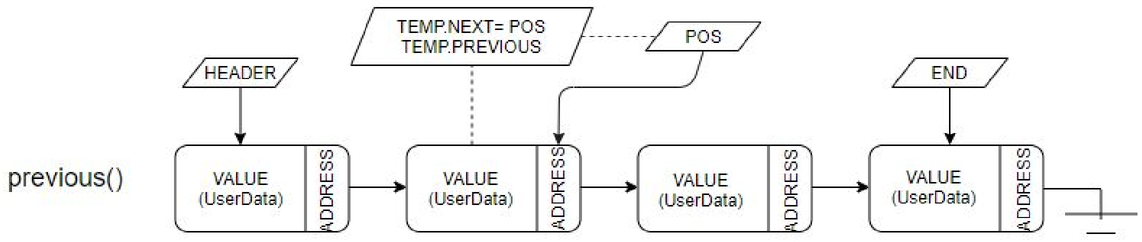


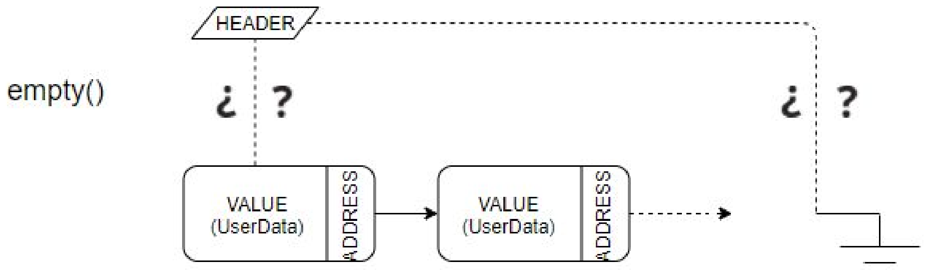
LIST

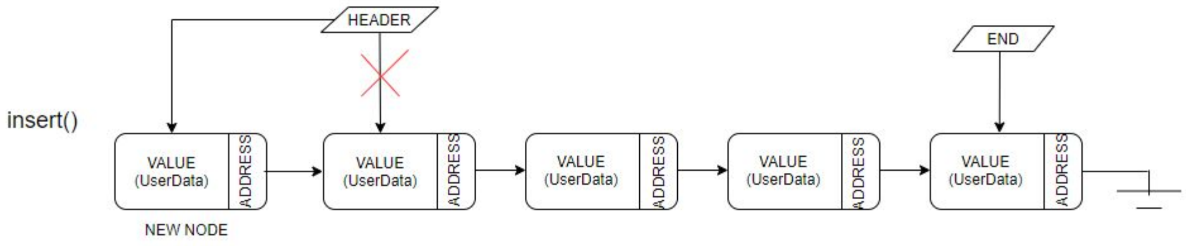


next()

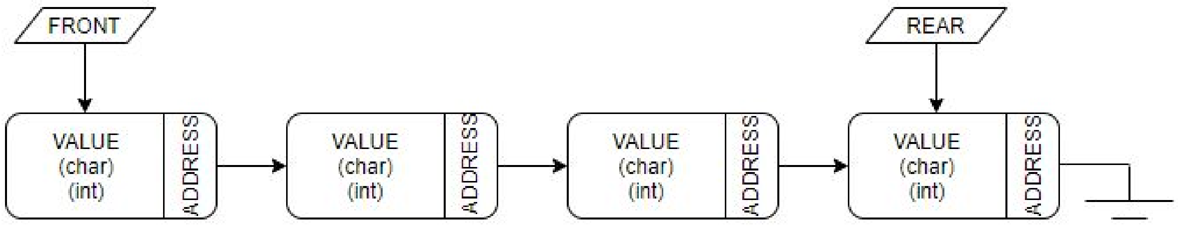



previous()
HEADER
TEMP.NEXT= POS
TEMP.PREVIOUS
POS
END
VALUE (UserData)
ADDRESS

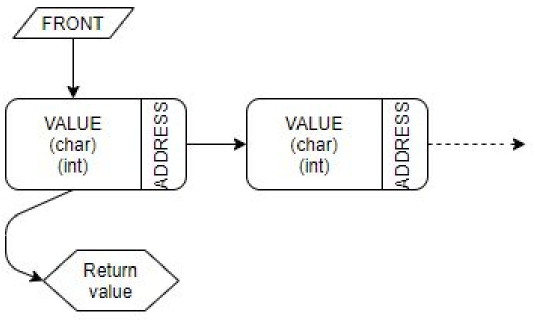

empty()
HEADER
¿ ?
¿ ?
VALUE (UserData)
ADDRESS


insert()
HEADER
END
VALUE (UserData)
ADDRESS
NEW NODE

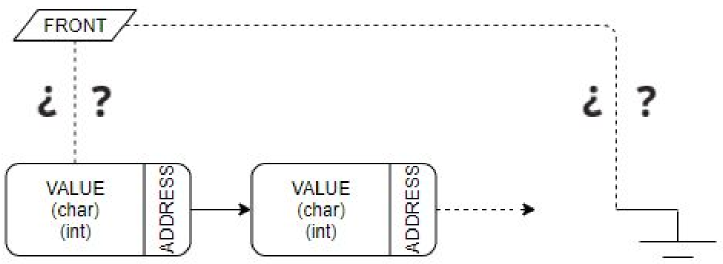
QUEUE


FRONT
REAR
VALUE (char) (int)
ADDRESS

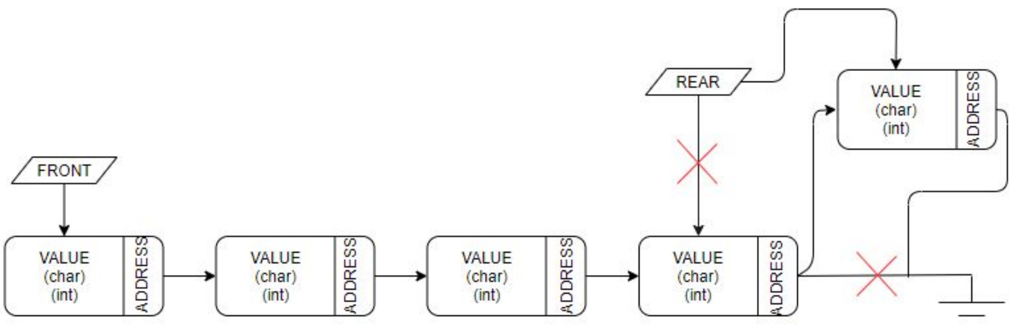
frontInt()
frontChar()



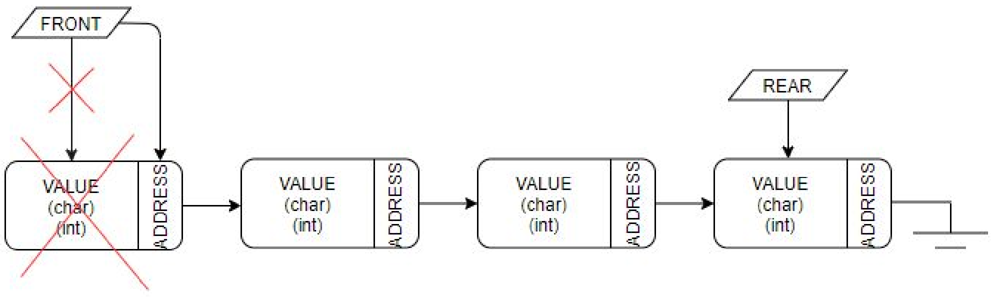
Empty()



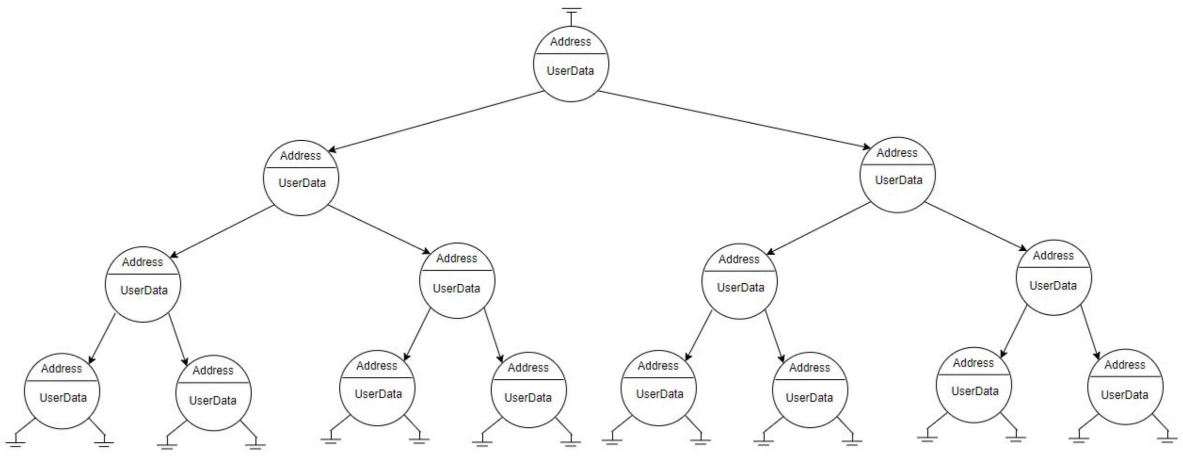
Enqueue()



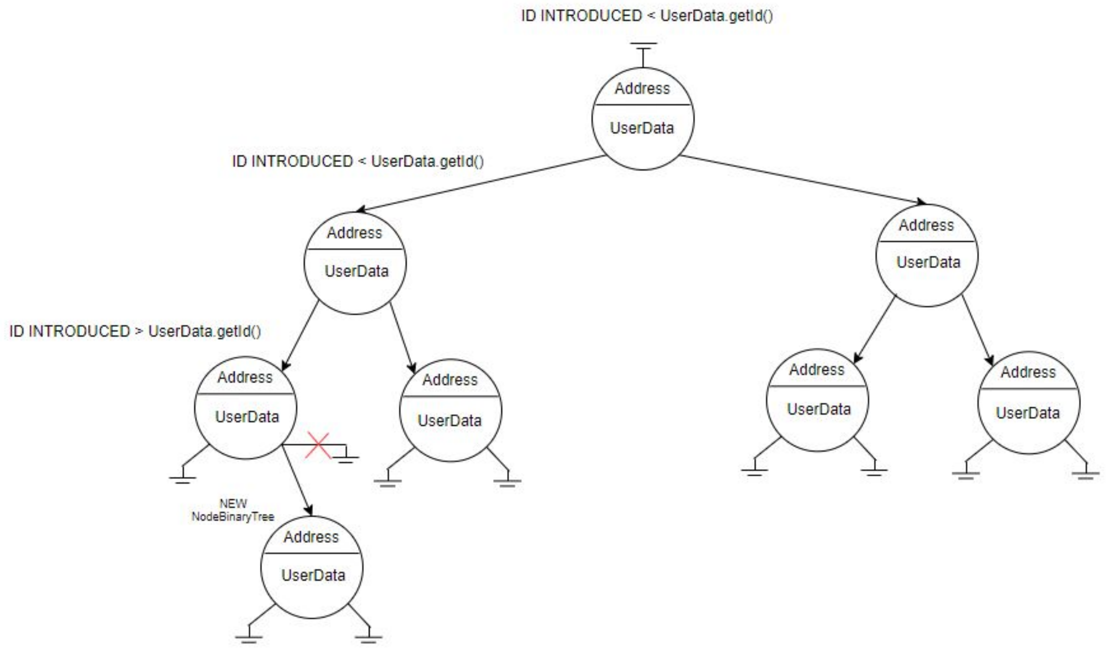
Dequeue()



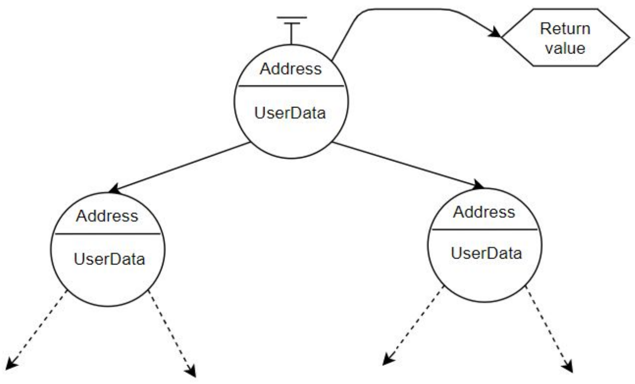
BINARY TREE



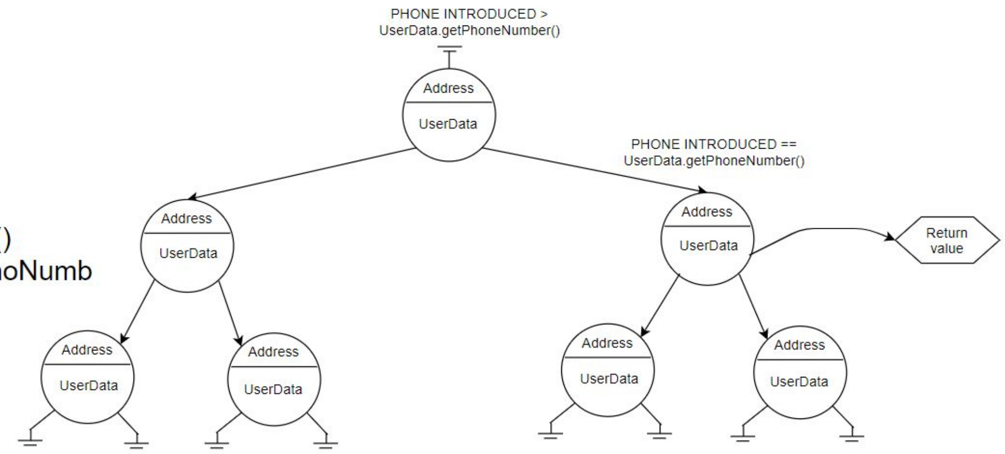
insert()


ID INTRODUCED < UserData.getId()
ID INTRODUCED < UserData.getId()
ID INTRODUCED > UserData.getId()
Address
UserData
NEW
NodeBinaryTree

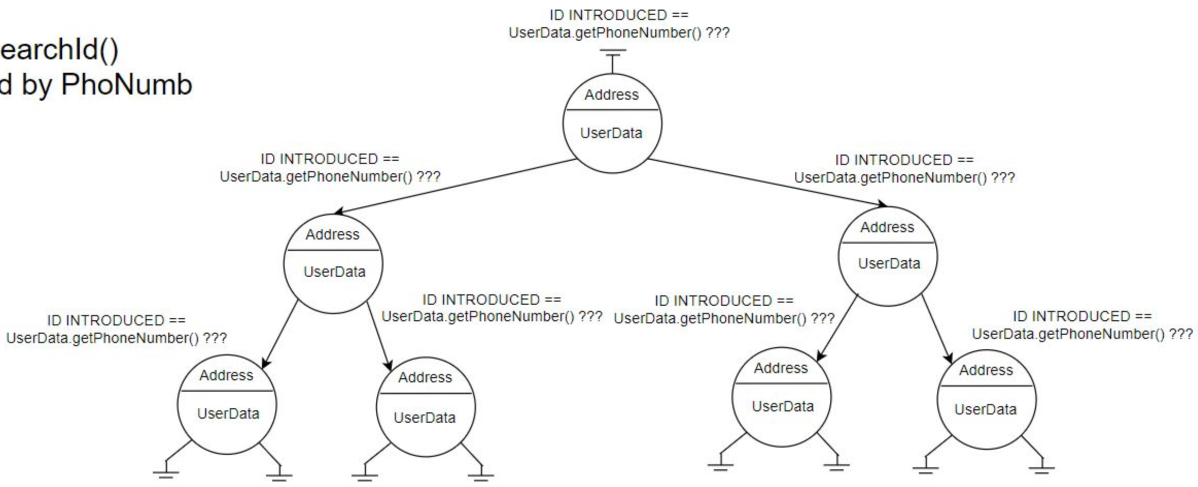
getRoot()

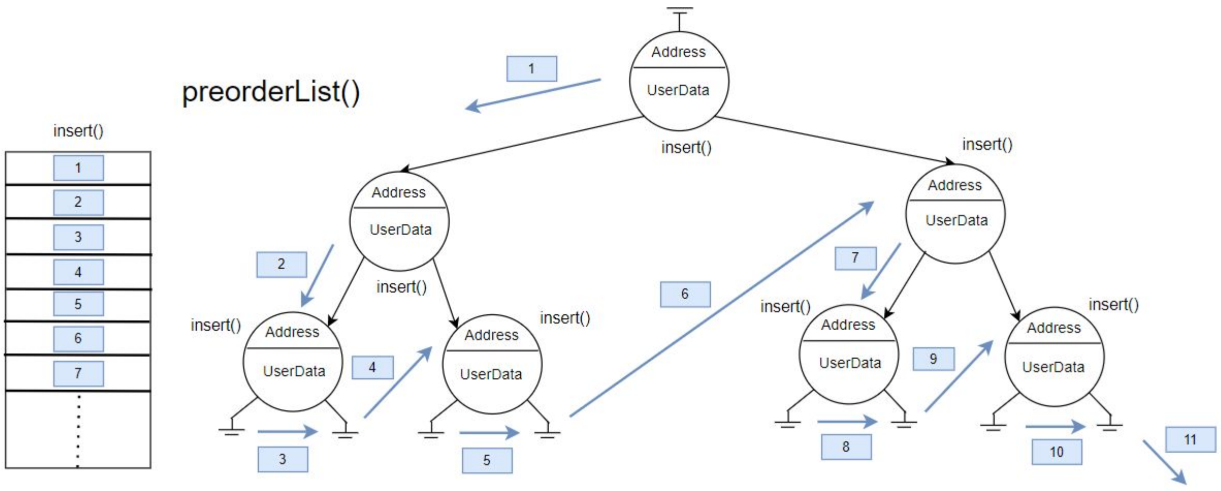


search()
ordered by PhoNumb


PHONE INTRODUCED > UserData.getPhoneNumber()
Address
UserData
PHONE INTRODUCED == UserData.getPhoneNumber()
Return value
Address
UserData

searchId()
ordered by PhoNumb

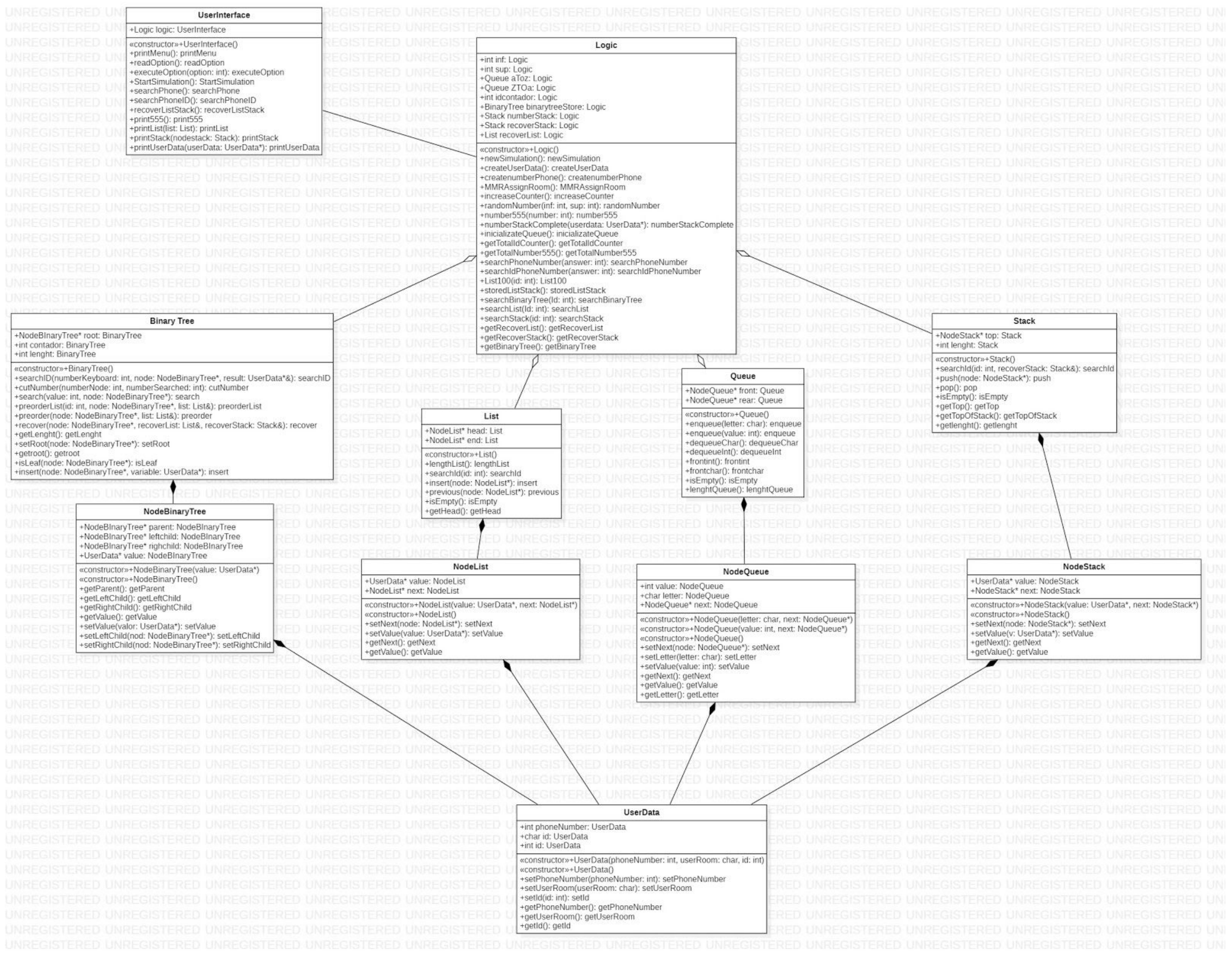

ID INTRODUCED == UserData.getPhoneNumber() ???
Address
UserData
ID INTRODUCED == UserData.getPhoneNumber() ???
ID INTRODUCED == UserData.getPhoneNumber() ???
ID INTRODUCED == UserData.getPhoneNumber() ???
ID INTRODUCED == UserData.getPhoneNumber() ???
ID INTRODUCED == UserData.getPhoneNumber() ???
ID INTRODUCED == UserData.getPhoneNumber() ???


preorderList()
insert()
Address
UserData
1
2
3
4
5
6
7
8
9
10
11

B) UML diagram and class diagram.

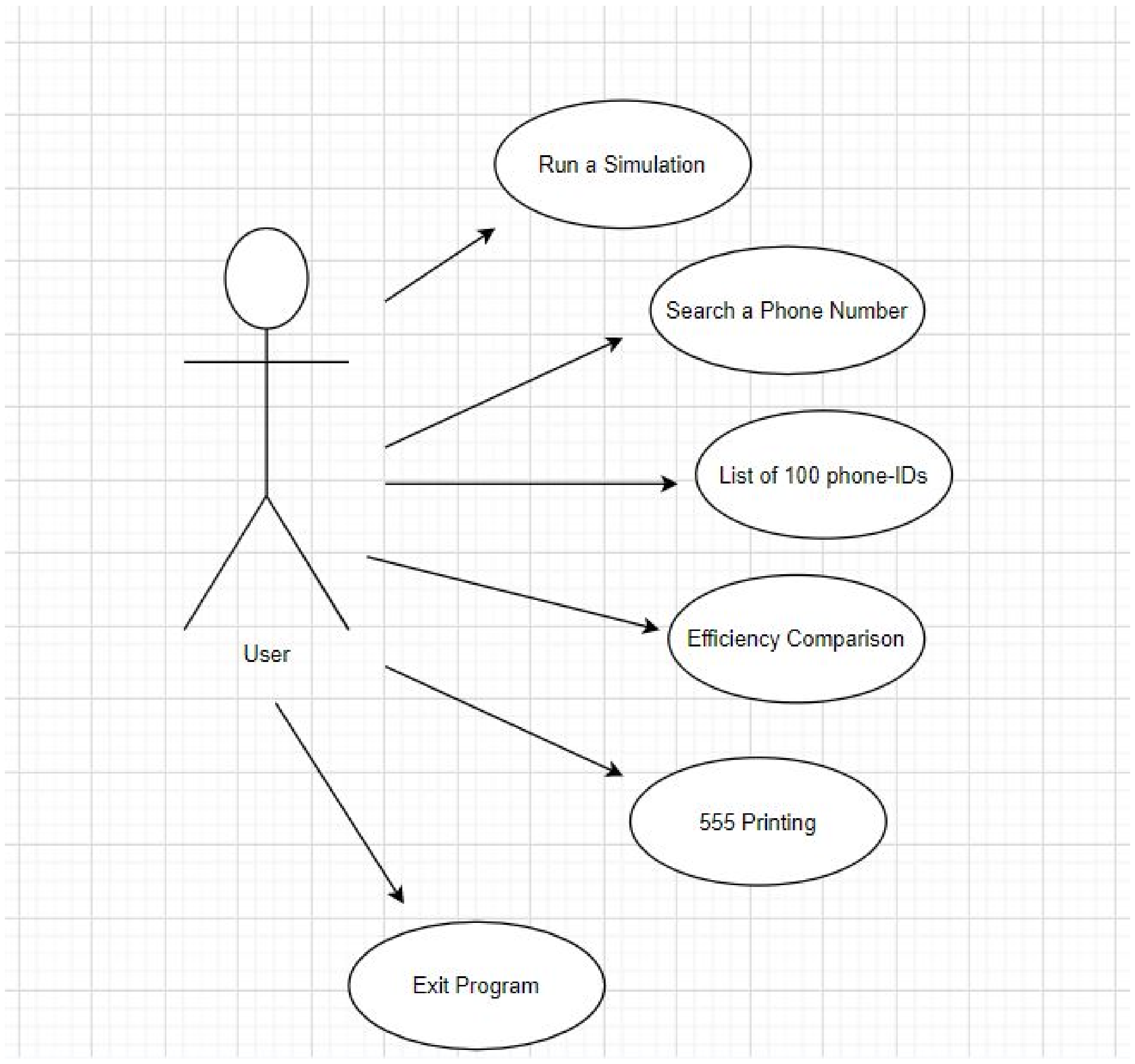
The diagram below shows the class diagram along with the connections between the classes in the program.

The program contains eleven classes: Main, Logic, UserInterface, UserData and the ADTS (Stack, Queue, List, BinaryTree).


UserInterface
+Logic logic: UserInterface
«constructor»+UserInterface()
+printMenu(): printMenu
+readOption(): readOption
+executeOption(option: int): executeOption
+StartSimulation(): StartSimulation
+searchPhone(): searchPhone
+searchPhoneID(): searchPhoneID
+recoverListStack(): recoverListStack
+print555(): print555
+printList(list: List): printList
+printStack(nodestack: Stack): printStack
+printUserData(userData: UserData*): printUserData
Logic
+int inf: Logic
+int sup: Logic
+Queue aToz: Logic
+Queue ZTOa: Logic
+int idcontador: Logic
+BinaryTree binarytreeStore: Logic
+Stack numberStack: Logic
+Stack recoverStack: Logic
+List recoverList: Logic
«constructor»+Logic()
+newSimulation(): newSimulation
+createUserData(): createUserData
+createnumberPhone(): createnumberPhone
+MMRAssignRoom(): MMRAssignRoom
+increaseCounter(): increaseCounter
+randomNumber(inf: int, sup: int): randomNumber
+number555(number: int): number555
+numberStackComplete(userdata: UserData*): numberStackComplete
+inicializateQueue(): inicializateQueue
+getTotalIdCounter(): getTotalIdCounter
+getTotalNumber555(): getTotalNumber555
+searchPhoneNumber(answer: int): searchPhoneNumber
+searchIdPhoneNumber(answer: int): searchIdPhoneNumber
+List100(id: int): List100
+storedListStack(): storedListStack
+searchBinaryTree(Id: int): searchBinaryTree
+searchList(Id: int): searchList
+searchStack(id: int): searchStack
+getRecoverList(): getRecoverList
+getRecoverStack(): getRecoverStack
+getBinaryTree(): getBinaryTree
Binary Tree
+NodeBInaryTree* root: BinaryTree
+int contador: BinaryTree
+int lenght: BinaryTree
«constructor»+BinaryTree()
+searchID(numberKeyboard: int, node: NodeBinaryTree*, result: UserData*&): searchID
+cutNumber(numberNode: int, numberSearched: int): cutNumber
+search(value: int, node: NodeBinaryTree*): search
+preorderList(id: int, node: NodeBinaryTree*, list: List&): preorderList
+preorder(node: NodeBinaryTree*, list: List&): preorder
+recover(node: NodeBinaryTree*, recoverList: List&, recoverStack: Stack&): recover
+getLenght(): getLenght
+setRoot(node: NodeBinaryTree*): setRoot
+getroot(): getroot
+isLeaf(node: NodeBinaryTree*): isLeaf
+insert(node: NodeBinaryTree*, variable: UserData*): insert
Stack
+NodeStack* top: Stack
+int lenght: Stack
«constructor»+Stack()
+searchId(id: int, recoverStack: Stack&): searchId
+push(node: NodeStack*): push
+pop(): pop
+isEmpty(): isEmpty
+getTop(): getTop
+getTopOfStack(): getTopOfStack
+getlenght(): getlenght
Queue
+NodeQueue* front: Queue
+NodeQueue* rear: Queue
«constructor»+Queue()
+enqueue(letter: char): enqueue
+enqueue(value: int): enqueue
+dequeueChar(): dequeueChar
+dequeueInt(): dequeueInt
+frontint(): frontint
+frontchar(): frontchar
+isEmpty(): isEmpty
+lenghtQueue(): lenghtQueue
List
+NodeList* head: List
+NodeList* end: List
«constructor»+List()
+lengthList(): lengthList
+searchId(id: int): searchId
+insert(node: NodeList*): insert
+previous(node: NodeList*): previous
+isEmpty(): isEmpty
+getHead(): getHead
NodeBinaryTree
+NodeBInaryTree* parent: NodeBInaryTree
+NodeBInaryTree* leftchild: NodeBInaryTree
+NodeBInaryTree* righchild: NodeBInaryTree
+UserData* value: NodeBInaryTree
«constructor»+NodeBinaryTree(value: UserData*)
«constructor»+NodeBinaryTree()
+getParent(): getParent
+getLeftChild(): getLeftChild
+getRightChild(): getRightChild
+getValue(): getValue
+setValue(valor: UserData*): setValue
+setLeftChild(nod: NodeBinaryTree*): setLeftChild
+setRightChild(nod: NodeBinaryTree*): setRightChild
NodeList
+UserData* value: NodeList
+NodeList* next: NodeList
«constructor»+NodeList(value: UserData*, next: NodeList*)
«constructor»+NodeList()
+setNext(node: NodeList*): setNext
+setValue(value: UserData*): setValue
+getNext(): getNext
+getValue(): getValue
NodeQueue
+int value: NodeQueue
+char letter: NodeQueue
+NodeQueue* next: NodeQueue
«constructor»+NodeQueue(letter: char, next: NodeQueue*)
«constructor»+NodeQueue(value: int, next: NodeQueue*)
«constructor»+NodeQueue()
+setNext(node: NodeQueue*): setNext
+setLetter(letter: char): setLetter
+setValue(value: int): setValue
+getNext(): getNext
+getValue(): getValue
+getLetter(): getLetter
NodeStack
+UserData* value: NodeStack
+NodeStack* next: NodeStack
«constructor»+NodeStack(value: UserData*, next: NodeStack*)
«constructor»+NodeStack()
+setNext(node: NodeStack*): setNext
+setValue(v: UserData*): setValue
+getNext(): getNext
+getValue(): getValue
UserData
+int phoneNumber: UserData
+char id: UserData
+int id: UserData
«constructor»+UserData(phoneNumber: int, userRoom: char, id: int)
«constructor»+UserData()
+setPhoneNumber(phoneNumber: int): setPhoneNumber
+setUserRoom(userRoom: char): setUserRoom
+setId(id: int): setId
+getPhoneNumber(): getPhoneNumber
+getUserRoom(): getUserRoom
+getId(): getId

In order to get a better view of the Diagram you can have the jpg image in the annexed folder

The diagram below represents the UML use-case diagram with every operation of the user interface.



C)Explanation of the behaviour of the program.

OPTION 1:

The user chooses option 1, this option simulates a random number between 10000 and 50000. The random number that comes out will be the number of times UserDatas are created, each userdata is made up of an id, a phone number and an MMR.

To create the id, we use the "Increase Counter" function, this function adds one each time it is called, this means that if there are 50000 userdate there will be 50000 ids.

To create the MMRs, we have closed two queues that are initialized in the Logic constructor (aTOz AND ZTOa). These two queues, as their names indicate, go from A to Z and from Z to A, the "incializateQueue" method fills these queues with the alphabet. The MMRAssignRoom function will return a random char. This function uses the algorithm of the statement, depending on what percentage we get in the first Random Number we will get one letter or another of the queues; With the char removed, now we must introduce it in the queue, we will use the second Random Number to know which queue to put that char. Once inserted, we return the letter.

To create the phoneNumber, we will use the create "Number Phone" function that will return a phone number. The random of this function will create a random number that will be saved in a string, when the string contains 9 numbers the random will stop working and we will convert said string to int in order to return the phone number.

Once the id, the MMR and the phonenumber have been created, we can now create the userdata, the "create User Data" function will fill in each user data with these 3 data and it will return a full UserData. But before returning it the number phone will go through two functions that are called by "create User Data". The two functions will be "number555" and "numberStackComplete"; the first one verifies if the phone number that was created starts with 555, and if so, the second function will store the userData of that phone number in a stack.

In the "new Simulation" method we will have the userdatas, each of the userDatas will be stored in a BinaryTree (binarytreeStore). Once the tree is full, we will return the number of total IDs that have been created and the number of numbers that start with 555 thanks to the "getTotalNumber555" function.

OPTION 2:
If the user chooses option 2, he will have to enter a phone number of a minimum size of 1 and a maximum of 9. The numbers that he can enter will be from 0 to 9, if he meets these requirements, the "searchPhoneNumber" function will limp said data entered and will call the “search” method of binary tree that will search for the data entered in the tree. Once it finds it, this function will return the user data with that phone number and we print it.

OPTION 3:
If the user chooses option 3, he will have to enter an id, said id must be between 1 and the maximum id. If it meets these requirements, the "searchIdPhoneNumber" function will be called; This function will call the "searchId" function that will return the userdata with the chosen id.

The second part works with the id entered, the "List100" function will be called, which will return the list with 100 userdatas, those 100 userdatas will be the phone numbers following the phone number assigned to the id we have chosen. The one who will search the binary tree and list each user data will be the function “preorder” (search) and “preorderList” (insert).

OPTION 4:

If the user chooses option 4, the first thing we will do is call the “storedListStack” method that will fill in the list and the stack of all the nodes in the binary tree. This function calls “recover” which is the one who will fill it and give it to you. it will pass the list and the full stack.

For the second part we will create a random number, these random numbers will be put into 3 queues, the first queue will be used to search the binary tree for 100 id randoms and print the userdatas that contain those ids. We do the same for the list and the stack previously filled.
Once the 3 blocks of 100 userdatas and their execution times have been printed, option 4 will end.

OPTION 5:

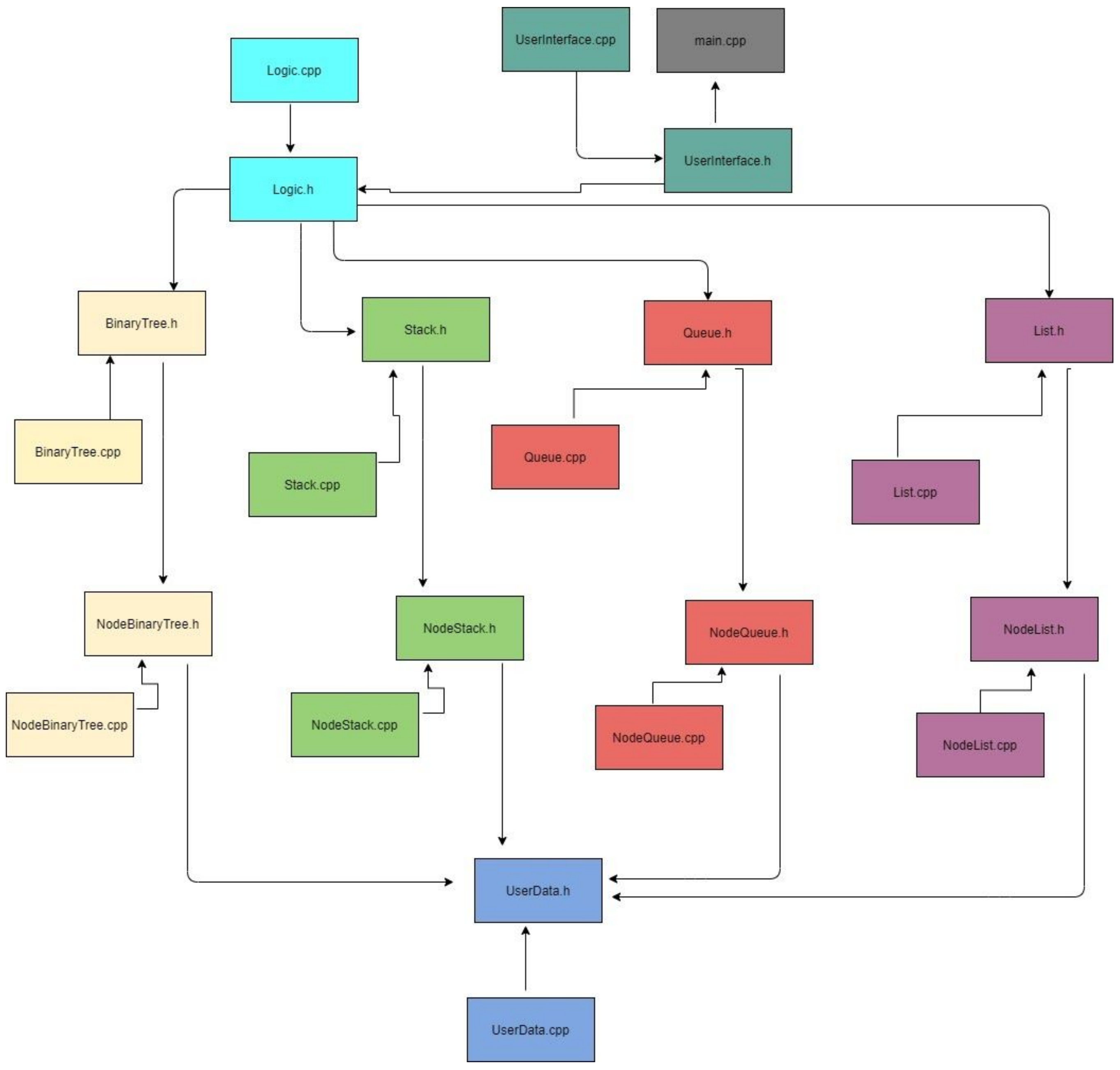
If the user chooses option 5, all 555 numbers will be printed with their ids and their execution time. To do this, we will create a stack that contains all the 555 numbers thanks to the "getTotalNNumber555" function that will return a stack filled with those numbers. The stack that we create pop the userdata and enqueue them in a queue. As we said at the beginning , we remove the userdatas from the queue and print them with their ids and their execution time

For the second part we will make a copy of the queue that we have commented previously and we will get the 555 numbers, we look for these numbers in the binary tree store by calling the "search" option 2, once found we take out their userdatas and print their Id, their phone numbers and their run time.

3. IMPLEMENTATION

A)Diagram with source files and their relations.

The diagram below shows the relations between all the h and cpp files of the program.



B)Explanation of every difficult section.

On Binary Tree the option that searches 100 data using Id order was complicated because the binary tree uses the phone number as a key. So if I try to search an id using the normal search of the tree, the resulting list won't be the desired one. So I came up with an operation that uses linear search that consumes more resources but I think is the only solution.

On Stack I got some difficulties during the operation of searching the id on the stack just because of the behaviour of the stack. (I can only access the element on the top of the stack). So if we want an id that is on the middle of the stack we have to pop the elements that we don't need in an auxiliary stack and when we find it get the userData with that id and push back the elements that we pop before on the original stack.

4. REVIEW

A)Possible enhancements for the solution.

A solution would be the "searchId" method of Binary Tree, since when searching by id, we have to do a linear search of the tree. This search uses a lot of memory and resources, to improve this function the solution would be to create another binary tree that is balanced that the key of the tree is the id

With this I finish the documentation, I hope I have explained myself in the best possible way..

Regards

Lucas Jorge Espejo