LAB ASSIGNMENT 2

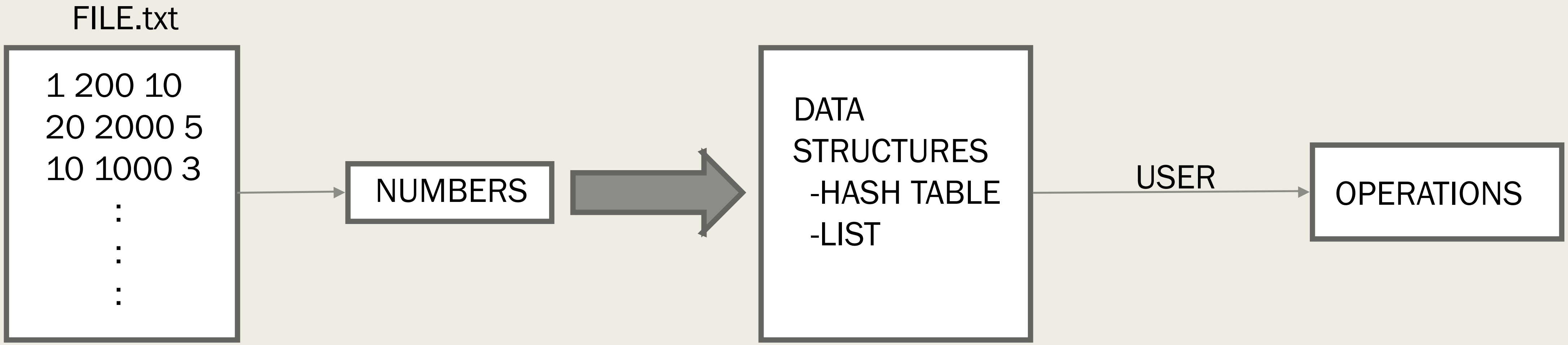
Javier García Jiménez, Isabel Martínez Gómez

INDEX

- Brief explanation
- How the user can interact with the system?
- What ADTs have been used in this program?
- Why Dynamic Data Structures?
- ADT Specifications
- What happens in the memory of the computer?
- UML Class Diagram
- UML Case-Use Diagram
- Explanation of classes
- Program behaviour
- Difficult part of the program
- Running times of operations

Brief Explanation

The work that we are going to introduce to you is a program whose purpouse is to work with a big amount of numbers and compare the behaviour of two data structures, for that, the program extracts series from an external file and generate numbers from that series, storing them and then allowing the user to interact with the system.



FILE.txt

1 200 10
20 2000 5
10 1000 3
:
:
:

NUMBERS

DATA
STRUCTURES
-HASH TABLE
-LIST

USER

OPERATIONS

How the User can interact with the system?

The program gives the user the possibility to do these operations:

- GET THE MAX/MIN NUMBERS
- ADD SERIES
- COUNT DISTINCT ELEMENTS
- SEE 100 BIGGER/LOWER NUMBERS
- SEARCH A NUMBER
- CALCULATE AVERAGE
- RESTART DATA STRUCTURES
- SEARCH NUMBER WITH MOST OCCCURENCES
- FINISH PROGRAM

What ADTs have been used in this program?

These are the ADTs that have been used in this program

- LIST:
  - *Insert*
  - *Length*
  - *Eliminate*
  - *Previous*
  - *Locate*
  - *Empty*
- STACK:
  - *Push*
  - *Pop*
  - *Empty*
- HASH TABLE
  - *Insert*
  - *getTable*
  - *Elminate*
  - *Member*
  - *Search*

Why Dynamic Data Structures?

- Size modification in execution time

- Use of needed memory

It is important to say that static memory is also used:

→ An array inside the Hash Table

ADT SPECIFICATIONS (I)

```
spec STACK[SERIES]
    genres stack, series
    operations
        empty: stack → bool
        pop: stack → serie
        push: stack, serie → stack
endspec
```

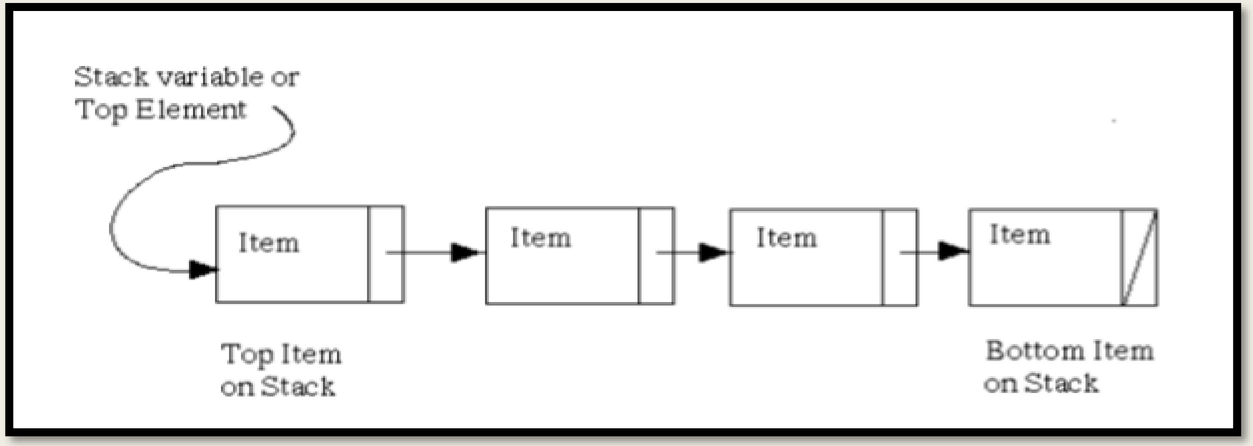
```
spec LIST[INT]
    genres list,int
    operations
        empty: list → bool
        insert: list , int→ list
        previous: list, int → list
        locate: list, int  → int
        lenght: list→ int
        eliminate: list,int → list
endspec
```

ADT SPECIFICATIONS (II)

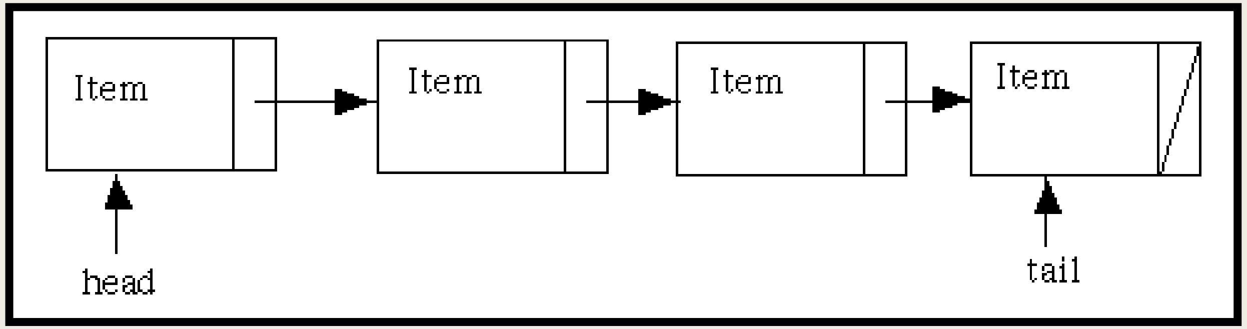
```
Spec HASHTABLE[INT]
    genres hashTable, int
    operations
        insert: hashTable, int →hash table
        member: hashTable, int → bool
        search: hashTable, int → int
        eliminate: hashTable, int→hash
        h: int → int
        getTable: hashTable → int[10]
endspec
```

AND, WHAT HAPPENS IN THE MEMORY OF THE COMPUTER? (I)

Pointer Implementation of Stack

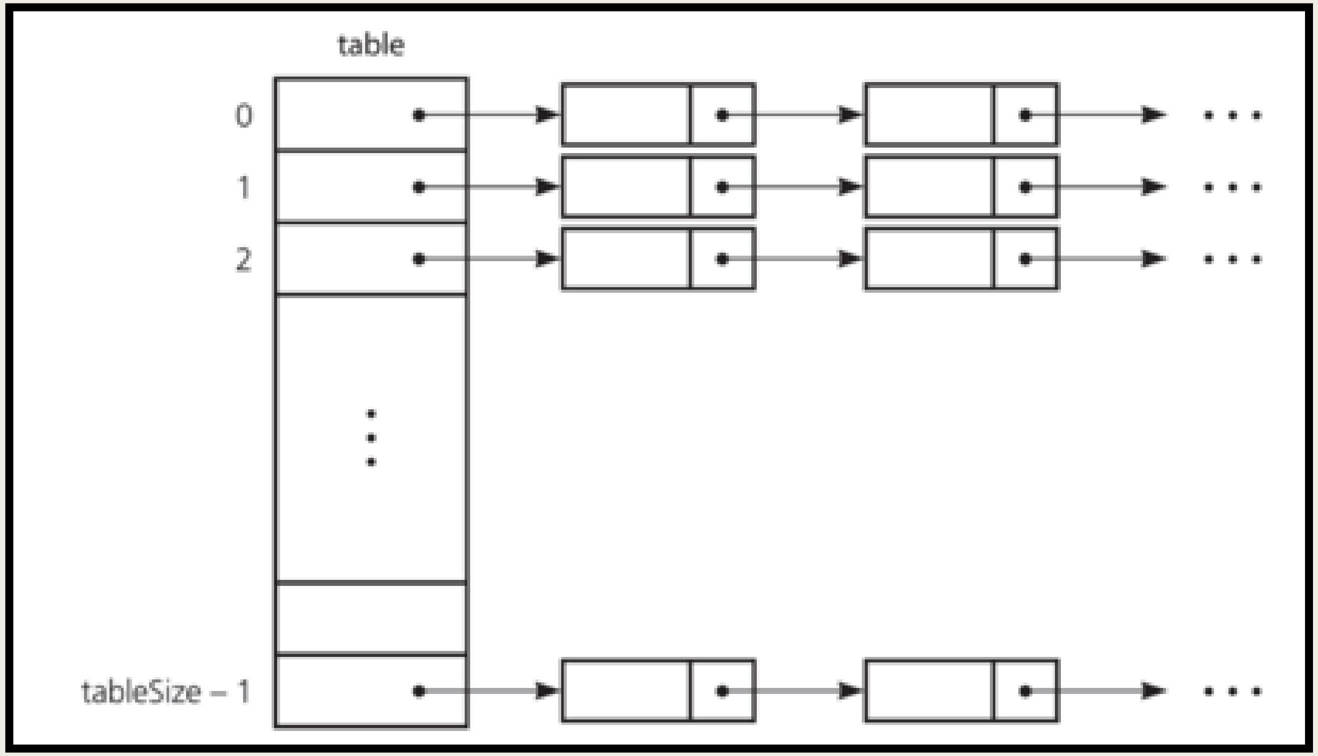


Pointer Implementation of List

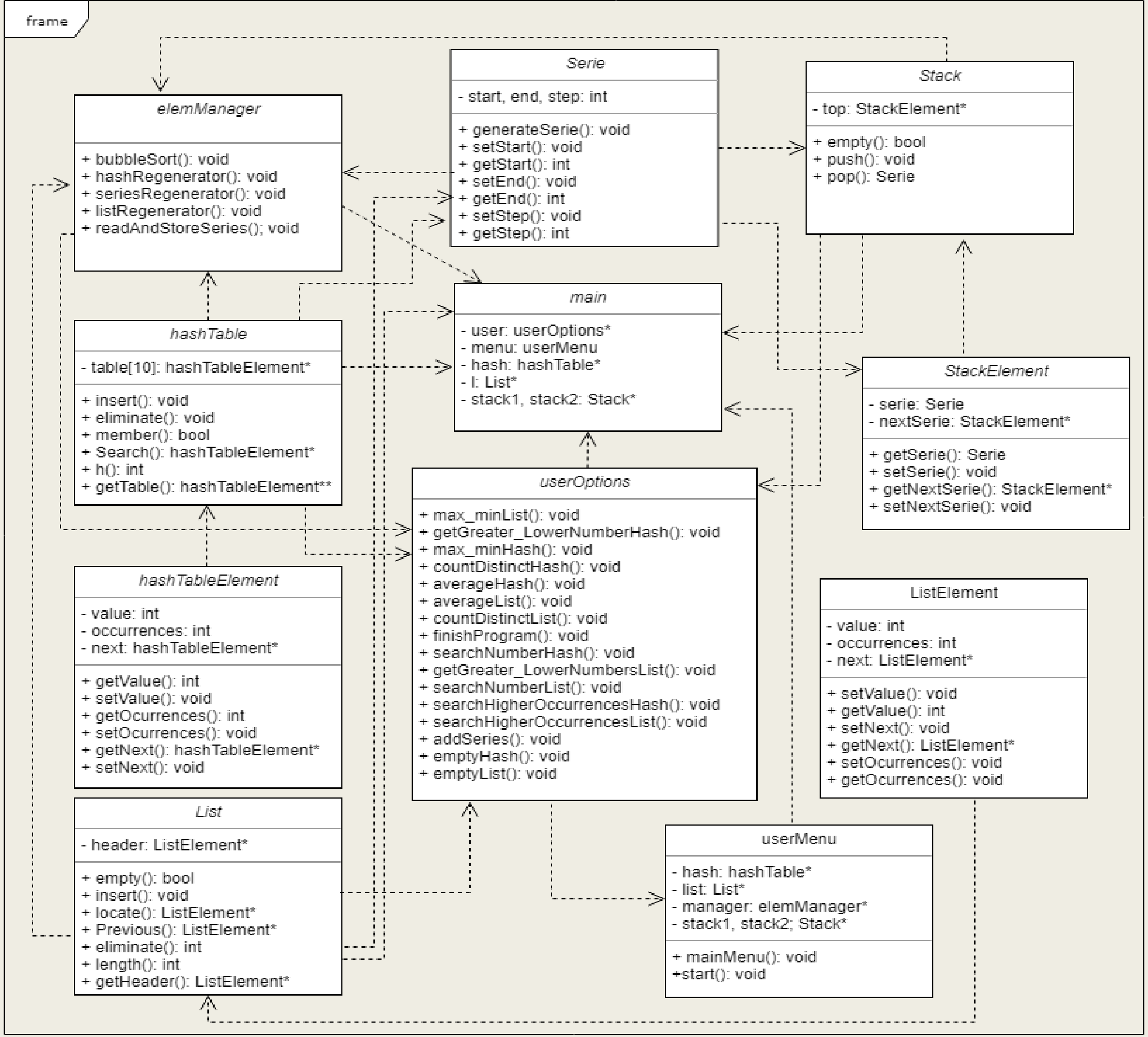


AND, WHAT HAPPENS IN THE MEMORY OF THE COMPUTER? (II)

Pointer Implementation of Hash Table



UML CLASS DIAGRAM



frame

elemManager

+ bubbleSort(): void
+ hashRegenerator(): void
+ seriesRegenerator(): void
+ listRegenerator(): void
+ readAndStoreSeries(); void

Serie

- start, end, step: int

+ generateSerie(): void
+ setStart(): void
+ getStart(): int
+ setEnd(): void
+ getEnd(): int
+ setStep(): void
+ getStep(): int

Stack

- top: StackElement*

+ empty(): bool
+ push(): void
+ pop(): Serie

main

- user: userOptions*
- menu: userMenu
- hash: hashTable*
- l: List*
- stack1, stack2: Stack*

hashTable

- table[10]: hashTableElement*

+ insert(): void
+ eliminate(): void
+ member(): bool
+ Search(): hashTableElement*
+ h(): int
+ getTable(): hashTableElement**

StackElement

- serie: Serie
- nextSerie: StackElement*

+ getSerie(): Serie
+ setSerie(): void
+ getNextSerie(): StackElement*
+ setNextSerie(): void

userOptions

+ max_minList(): void
+ getGreater_LowerNumberHash(): void
+ max_minHash(): void
+ countDistinctHash(): void
+ averageHash(): void
+ averageList(): void
+ countDistinctList(): void
+ finishProgram(): void
+ searchNumberHash(): void
+ getGreater_LowerNumbersList(): void
+ searchNumberList(): void
+ searchHigherOccurrencesHash(): void
+ searchHigherOccurrencesList(): void
+ addSeries(): void
+ emptyHash(): void
+ emptyList(): void

hashTableElement

- value: int
- occurrences: int
- next: hashTableElement*

+ getValue(): int
+ setValue(): void
+ getOcurrences(): int
+ setOcurrences(): void
+ getNext(): hashTableElement*
+ setNext(): void

ListElement

- value: int
- occurrences: int
- next: ListElement*

+ setValue(): void
+ getValue(): int
+ setNext(): void
+ getNext(): ListElement*
+ setOcurrences(): void
+ getOcurrences(): void

List

- header: ListElement*

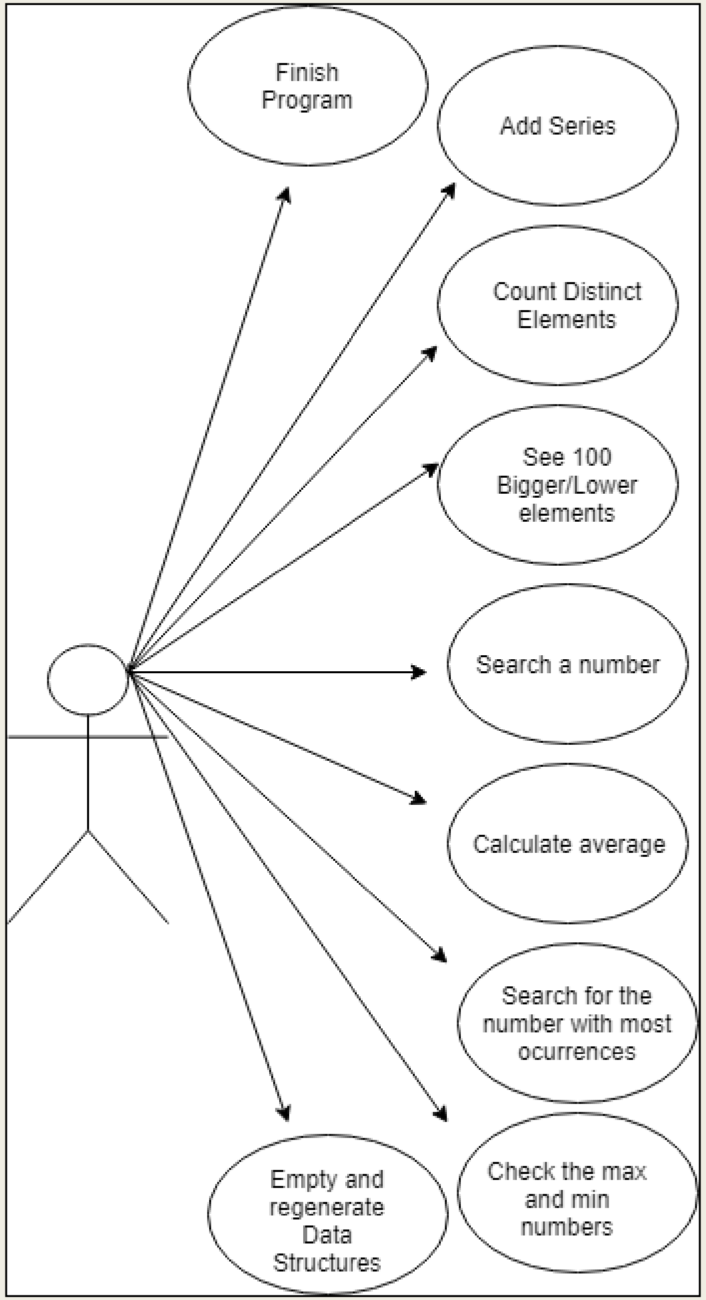
+ empty(): bool
+ insert(): void
+ locate(): ListElement*
+ Previous(): ListElement*
+ eliminate(): int
+ length(): int
+ getHeader(): ListElement*

userMenu

- hash: hashTable*
- list: List*
- manager: elemManager*
- stack1, stack2; Stack*

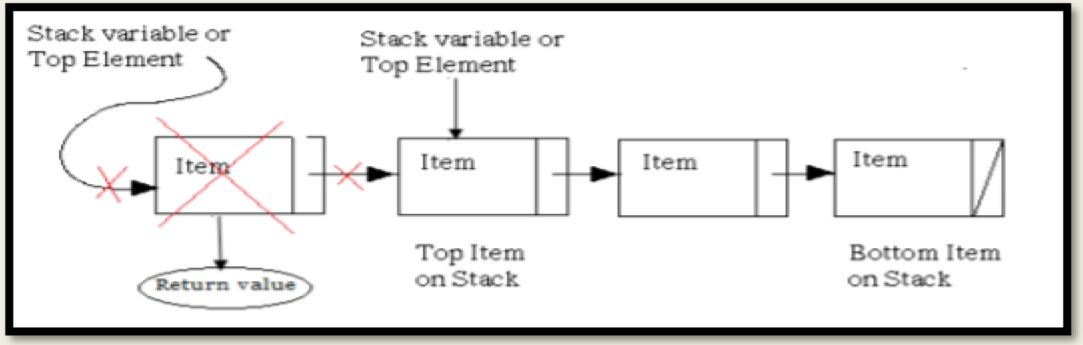
+ mainMenu(): void
+start(): void

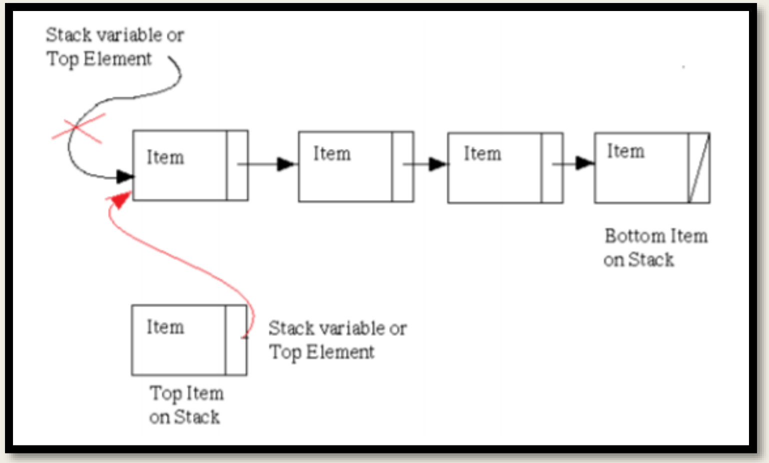
UML CASE-USE DIAGRAM


Finish Program
Add Series
Count Distinct Elements
See 100 Bigger/Lower elements
Search a number
Calculate average
Search for the number with most ocurrences
Check the max and min numbers
Empty and regenerate Data Structures

EXPLANATIONS OF CLASSES (I)

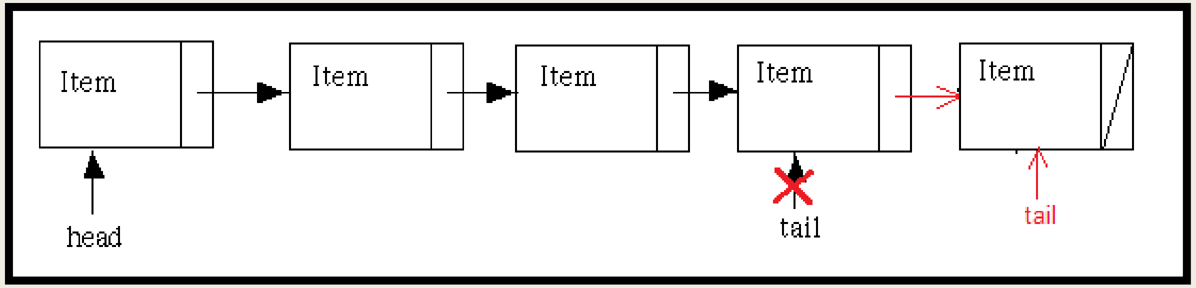
- STACK METHODS
  - *Empty*
  - *Pop*
  - *Push*

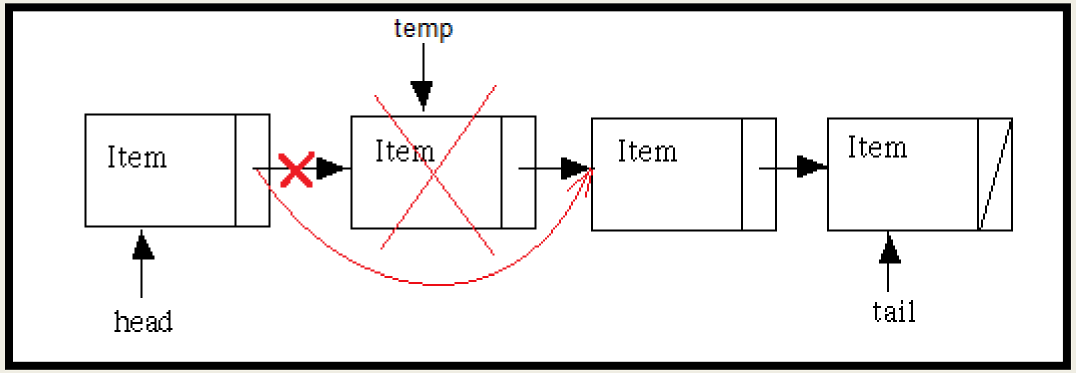


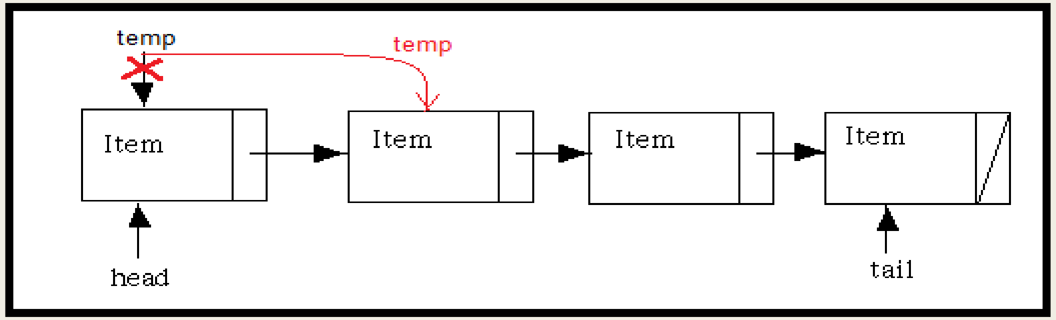


EXPLANATION OF CLASSES (II)

- LIST METHODS:
  - *Insert*
  - *Lenght*
  - *Empty*
  - *Eliminate*
  - *Locate*



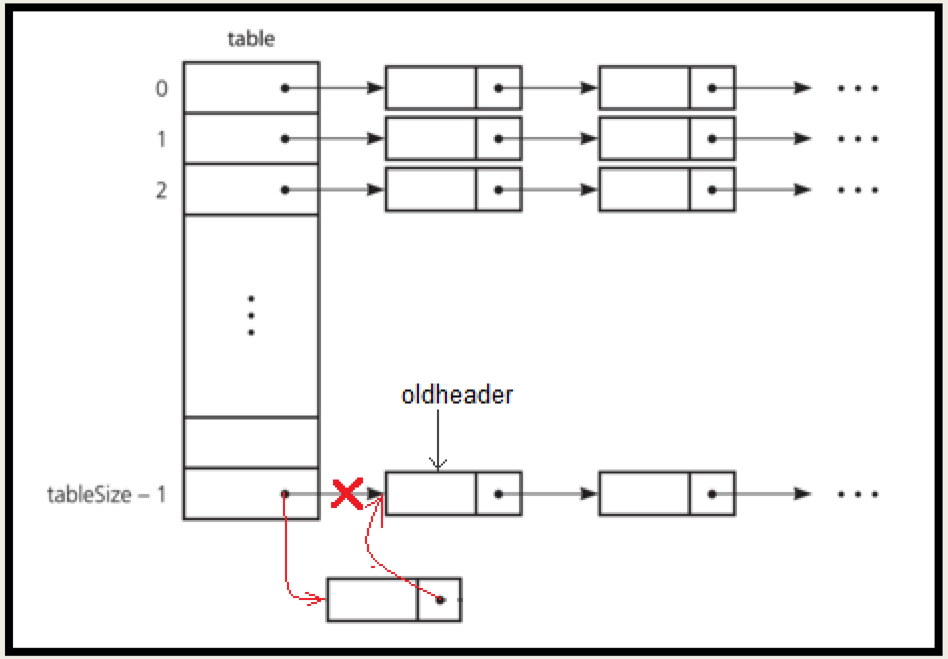


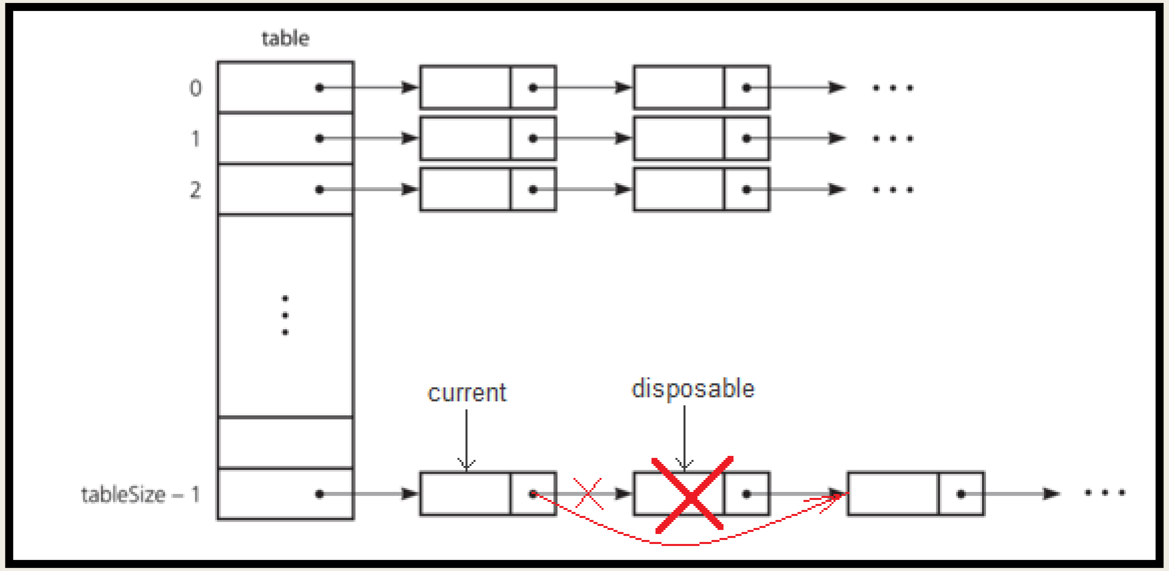


EXPLANATION OF CLASSES (III)

- HASH TABLE METHODS:
  - *Member*
  - *Search*
  - *H*

Insert





Eliminate

EXPLANATION OF THE CLASSES (IV)

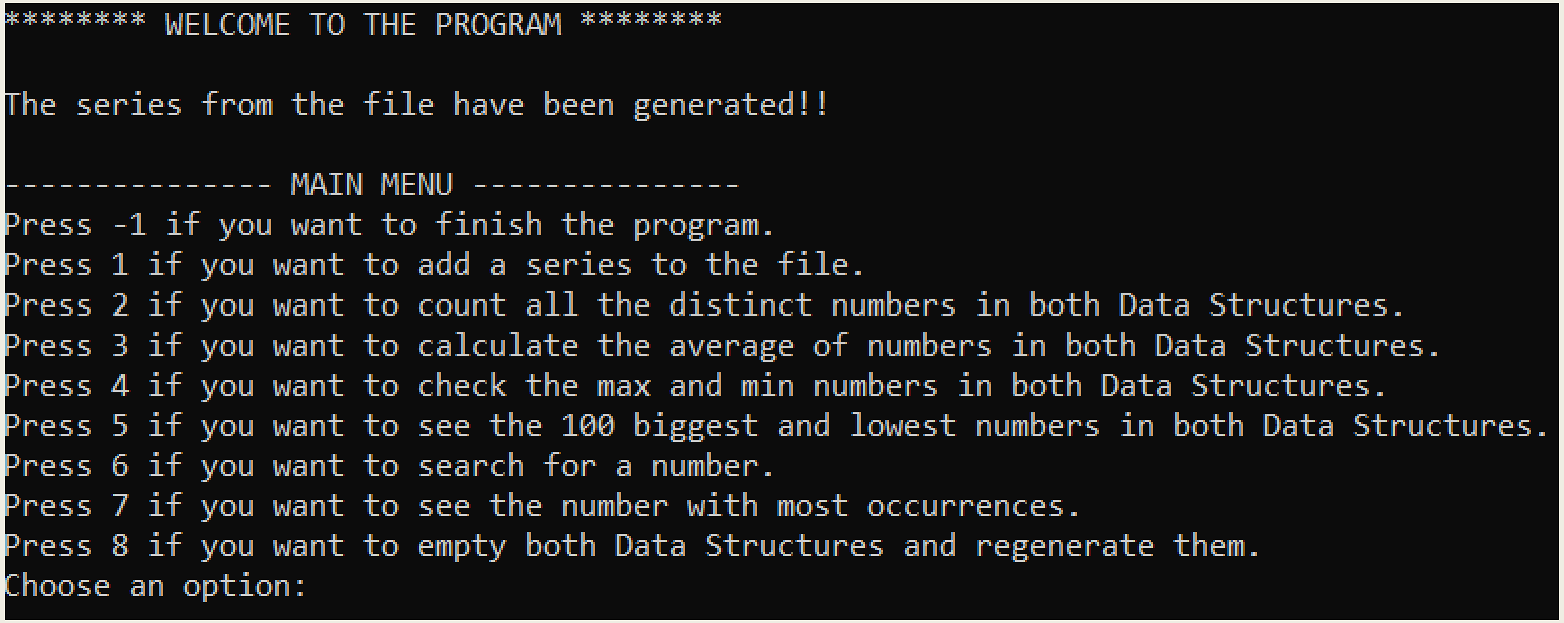
- These are the clases that we have added:
  - *Serie*
  - *elemManager*
  - *mainMenu*
  - *userOptions*

HOW IS THE PROGRAM BEHAVIOUR?

- First, the series are read and stored, then they are generated.

- The numbers created from the series are stored in the hash table and the list

- Finally, the main menu is displayed giving the user the options mentioned before

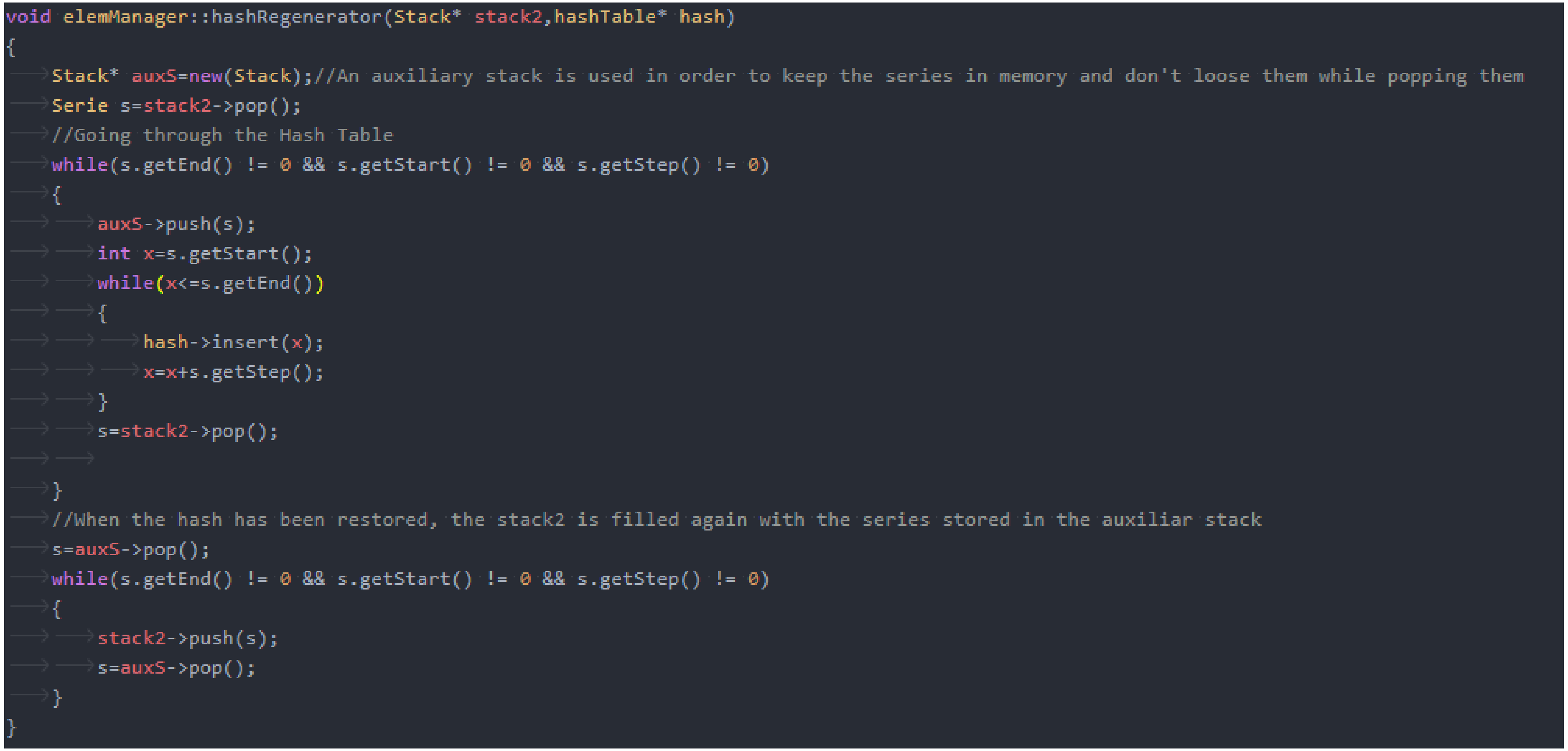


```
******** WELCOME TO THE PROGRAM ********

The series from the file have been generated!!

--------------- MAIN MENU ---------------
Press -1 if you want to finish the program.
Press 1 if you want to add a series to the file.
Press 2 if you want to count all the distinct numbers in both Data Structures.
Press 3 if you want to calculate the average of numbers in both Data Structures.
Press 4 if you want to check the max and min numbers in both Data Structures.
Press 5 if you want to see the 100 biggest and lowest numbers in both Data Structures.
Press 6 if you want to search for a number.
Press 7 if you want to see the number with most occurrences.
Press 8 if you want to empty both Data Structures and regenerate them.
Choose an option:
```

DIFFICULT PART OF THE PROGRAM

hashRegenerator():
This function fills the hash again with numbers from the series, the series are extracted from the second stack and are added to an auxiliary stack that copies the content of the second stack. When the hash has been regenerated, the second stack is filled again with the series stored in the auxiliar stack. This is done in order to keep the series in memory and don’t lose them while refilling the hash.

It’s important to say that there is another function that does the same thing but with the list (listRegenerator).



```cpp
void elemManager::hashRegenerator(Stack* stack2,hashTable* hash)
{
    Stack* auxS=new(Stack);//An auxiliary stack is used in order to keep the series in memory and don't loose them while popping them
    Serie s=stack2->pop();
    //Going through the Hash Table
    while(s.getEnd() != 0 && s.getStart() != 0 && s.getStep() != 0)
    {
        auxS->push(s);
        int x=s.getStart();
        while(x<=s.getEnd())
        {
            hash->insert(x);
            x=x+s.getStep();
        }
        s=stack2->pop();
        
    }
    //When the hash has been restored, the stack2 is filled again with the series stored in the auxiliar stack
    s=auxS->pop();
    while(s.getEnd() != 0 && s.getStart() != 0 && s.getStep() != 0)
    {
        stack2->push(s);
        s=auxS->pop();
    }
}
```

RUNNING TIME OF OPERATIONS

- **Stack:**
  - push : O(1)
  - pop: O(1)
  - empty: O(1)

- **List:**
  - insert: O(n)
  - previous: O(n)
  - empty: O(1)
  - locate: O(n)
  - eliminate: O(n)
  - length: O(n)

- **Hash Table:**
  - member : O(n)
  - insert: O(n)
  - eliminate: O(n)
  - search: O(n)
  - h: O(1)