

LAB 3: CMPS347

|  |  |
| --- | --- |
| **Beirut Arab University Faculty of Science**  **Mathematics and Computer Science Department** | **Course: Data Structure Semester: Fall 2024-2025**  **Lab Sheet # 3**  **Instructor: Dr. Ali El-Zaart, Dr. Abdallah Chakik, Dr. Mahmoud Skafi, and Dr Samer Zant** |

**Question 1:**

Add to the class ArrayInt the following methods:

|  |
| --- |
| **ArrayInt** |
| * **a: int[]** * **currentNb: int** |
| * **ArrayInt( size: int)** * **isEmpty(): boolean** * **isFull() : Boolean** * **InsertElementAtFront(value: int): void** * **InsertElementAtBack(value: int): void** * **InsertElementAtPosition(position: int, value: int): void** * **Display(): void** * **SearchValue(value: int)** * **DeleteElement(value:int)** * **SortArrayDesc(): void** * **SortArrayAscending(): void** * **SelectionSort():void** * **InterpolationSearch(int): Boolean** * **IntervalBinarySearch(value:int): int** |

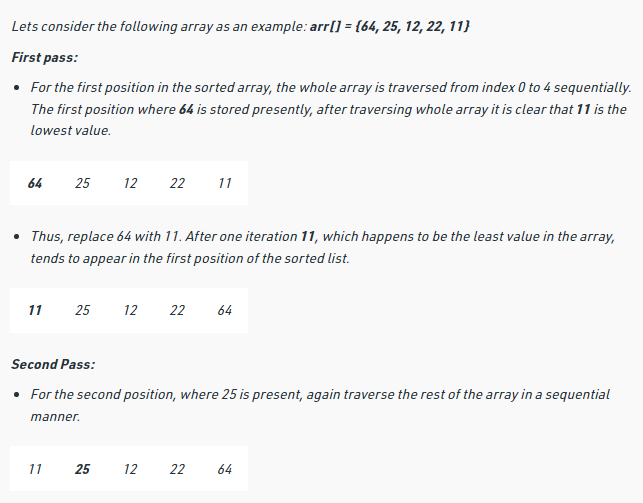
**Attributes** of class **ArrayInt**

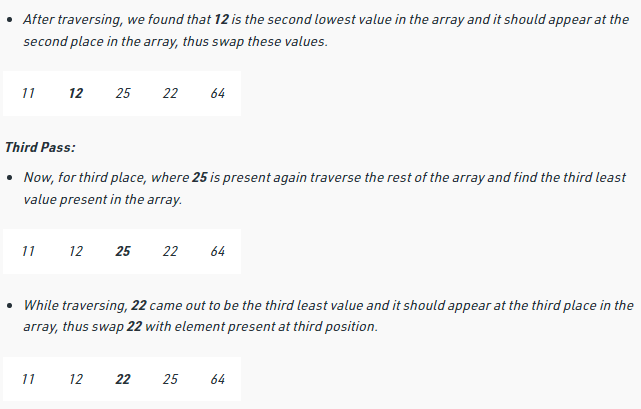
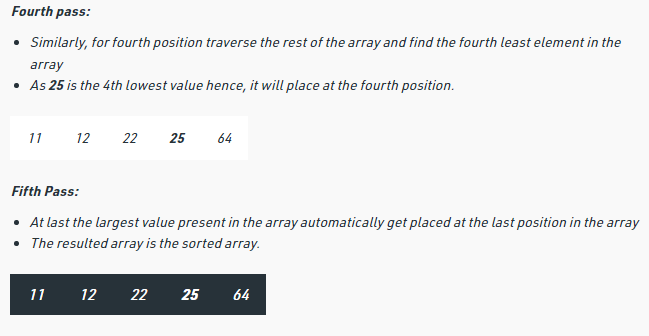
* **a**: an array where numbers are placed
* **currentNb:** number of elements in the array

**Methods** of class **ArrayInt:**

* **ArrayInt():** constructor that initializes the class attributes
* **isEmpty():** returns true if the array is empty and false otherwise
* **isFull():** returns true if the array is full and false otherwise
* **InsertAtFront(int):** adds an element at the front of the array
* **InsertAtPosition(int,int):** adds an element at a certain position
* **InsertAtBack(int):** adds an element at back of the array
* **Display():** prints the elements of the array
* **SearchValue(int):** search a value in the array.
* **DeleteElement(int):** delete a value from the array.
* **SortArrayDesc()** sorts the array elements in an descending order.
* **SortArrayAscending():** sorts the array elements in an ascending order.
* **SelectionSort():** The idea behind selection sort is that we put a list in order by placing each item in turn. In other words, we put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

An example of the selection sort is given in the figure below.



* **InterpolationSearch(int):** return true if the element exits in the array otherwise false. It is an algorithm for searching for a key in an array that has been ordered (sorted) by numerical values assigned to the keys (key values). The idea behind interpolation search is that each time we make a comparison; we ignore part of the list based on the key, until we either find the search term or determine that the term is not in the list. To calculate the key using the following formula:

key = lo +((hi - lo) \* (x - arr[lo])) /(arr[hi ] - arr[lo]);

where

hi: -is higher than the middle item(arr[key])

lo: - is lower than the middle item

x: - is the searching element

If search element is larger, search element is in higher half of the list and repeat our search starting at the point halfway between key and the last item. If it’s smaller, search element is in the lower part. We ignore the upper half of the list and repeat our search starting at the point halfway between the first item and key.

***Example:***

To search for the number **42** in the given array, we should start by looking at the key,12.

key = 0 + ((14-0)\*(42-a[0])/(a[14]-1[0]);

**Lo=0 key hi=14**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | **12** | **13** | **16** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **33** | **35** | **42** | **47** |

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

42 >35, so search the higher half of the array.

Look at the key,13:

**Lo=** key+1;

**hi=s**ize-1

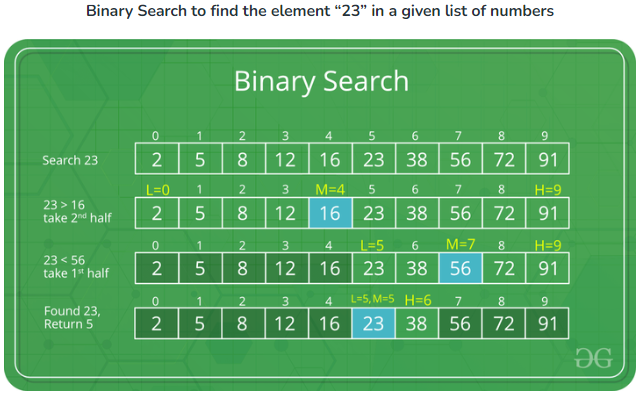
**key**= 13 + ((14 - 13)\*(42-a[13]))/(a[14]-a[13]);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | **12** | **13** | **16** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **33** | **35** | **42** | **47** |

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

42=42, so the search ends. 42 is in the array.

* **IntervalBinarySearch(int):** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example:



Write the main program to test all methods of the class ArrayInt.

# Question 2: Time Complexity

Find Big O for all examples:

***Example 1:***

|  |  |
| --- | --- |
| **Statement** |  |
| i=1; |  |
| sum = 0; |  |
| while (i <= n) { |  |
| j=1; |  |
| while (j <= n) { |  |
| sum = sum + i; |  |
| j = j + 1; |  |
| } |  |
| i = i +1;  } |  |

***Example 2:***

Here is the same code as above but using nested for loops instead of while loops:

|  |  |  |
| --- | --- | --- |
| **Statement** |  |  |
| sum = 0; |  |  |
| for(i= 1; i ≤ n; i++) |  |  |
| for(j= 1; j ≤ n; j++) |  |  |
| sum = sum + i; |  |  |

***Example 3:***

|  |  |  |
| --- | --- | --- |
|  | **Cost** | **Times** |
| for (i=1; i<=n; i++) |  |  |
| for (j=1; j<=i; j++) |  |  |
| for (k=1; k<=j; k++) |  |  |
| x=x+1; |  |  |

***Example 4: Selection sort using for-loops***

|  |  |  |
| --- | --- | --- |
|  | **Cost** | **Times** |
| for (i= n -1; i>=1; i--) |  |  |
| Index=0 |  |  |
| for (j=1; j<=i; j++) |  |  |
| if(x[j] > x[index])  Index=j; |  |  |
| Exchange (x[index], x[i]) |  |  |

***Example 5: Bubble sort using for-loops***

|  |  |  |
| --- | --- | --- |
|  | **Cost** | **Times** |
| for (i= n -1; i>=1; i--) |  |  |
| for (j=1; j<=i; j++) |  |  |
| if(x[j-1] > x[j]) |  |  |
| swap(x[j], x[j-1]) |  |  |

***Example 6:***

What is the time complexity of fun()?

int fun(int n)

{

int count = 0;

for (int i = n; i > 0; i /= 2)

for (int j = 0; j < i; j++)

count += 1;

return count;

}

***Example 7:***

All the following functions use a single for loop and within the for loop, the same set of statements are executed. Consider the following for loops:

A) for(i = 0; i < n; i++)

B) for(i = 0; i < n; i += 2)

C) for(i = 1; i < n; i \*= 2)

D) for(i = n; i <= n; i /= 2)

If n is the size of input(positive), which function is most efficient)?

**Good Luck**