

Objective(s) : understanding handling operations on an array.

task 1:

Implement MyArrayBasic class in package \Lab03\pack with the following methods

-> void add(int d) – append d into an array

+ void insert_unordered(int idx, int d)

– insert value d into the array at position index.

+ int find(int d) – return the index of value d in the array, else -1 (either ordered or unordered)

+ void delete(int index) – delete from ordered array i.e. the order of the data remains unchanged.

+ MyArray(int ... a) – a constructor creating the first MAX_SIZE = a.length

+ int getAt(int index)

+ void setAt(int index)

Demonstrate its mechanism through the demo code

```
package code;
public class MyArrayBasic {
    protected int MAX_SIZE = 5;
    protected int data[] = new int[MAX_SIZE];
    protected int size = 0;

    ...
}
```

```
static private void arrayBasic_demo1() {
    MyArrayBasic demo = new MyArrayBasic(7,6,8,1,2,3);
    System.out.println( demo );
}
static private void arrayBasic_demo2() {
    MyArrayBasic demo = new MyArrayBasic();
    demo.insert_unordered(9,0);
    demo.insert_unordered(7,0);
    demo.insert_unordered(5,0);
    System.out.println( demo );
    System.out.println("5 is at " + demo.find(5));
    System.out.println( demo.getAt(1) );
}
static void arrayBasic_demo_3() {
    MyArrayBasic demo = new MyArrayBasic();
    demo.add(3);
    demo.add(7);
    demo.add(5);
    demo.add(4);
    demo.add(6);
    System.out.println("next add operation
    triggers ArrayIndexOutOfBoundsException");
    demo.add(1);
}
```

task 2: implement `class MyArray` which extends `MyArrayBasic` with the following enhancements:

+ `MyArray()` – a constructor with default `MAX_SIZE = 100_000`

+ `MyArray(int max)` – a constructor with with supplied `MAX_SIZE`;

+ `boolean isFull()` – return true if there is not available cell to insert d (insertion would cause an exception)

+ `boolean isEmpty()` – return true if there is no data in the array (deletion would cause an exception)

- `int [] expandByK(int k)` – implicitly allocate a `k * MAX_SIZE` array to prevent overflow addition (`add()` method)

- `int [] expand()` – default `k = 2` i.e. call `expandByK(2)`; i.e. double the array's capacity

Override `add(int d)`, `insert_unordered(int d, int pos)`, `delete(int d)` to handle its dynamic capacity capability.

```
static void arrayBasic_demo_4(){
    MyArray demo = new MyArray(5);
    demo.delete(0);
    demo.add(3);
    demo.add(7);
    demo.add(5);
    demo.add(4);
    demo.add(6);
    demo.add(1);
    println(demo);
}
```

Task 3 Let's work on binary search.

Given the data in `MyArray` is sorted.

+ `binarySearch(int target)` returns

 If found, Index of the target

 If not found, the negative value of the to be inserted position – (insert point + 1) e.g. -1 for position 0 or -2 for position 1 (after the first element)

```
static void array_demo_5() {
    MyArray demo = new MyArray();
    demo.insert(9,0);
    demo.insert(7,0);
    demo.insert(8,0); // illegal skipped
    demo.insert(5,0);
    System.out.println( demo );
    System.out.println("5 is at " +
        demo.binarySearch(5));
    System.out.println("4 is at " +
        demo.binarySearch(4));
    // return -1 bec. insertion pos for 4 is (-1 +1)
    // * -1 = 0
    int pos = demo.binarySearch(6);
    System.out.println("6 is at " + pos);
    // retrurn -2 bec. insertion pos for 6 is (-2 +1)
    // * -1 = 1
    demo.insert(6, (pos+1));    //[5, 6, 7, 9]
    System.out.println( demo );
}
```

task 4: use

`System.currentTimeMillis()`

to measure time performance.

Notice the time it takes for
each data size.

Run `array_demo_5`

3 times. Write down the result
execution time to the
bellowed table.

If you adjust the size of the
initial N (and step size), note it
to the table as well.

```
static void array_demo_6() {
    System.out.println("small size initialized");
    for (int N = 200_000; N <= 10 * 200_000;
        N += 200_000) {
        long start = System.currentTimeMillis();
        MyArray mArray = new MyArray(200_000/40_000);
        for (int n = 1; n < N; n++)
            mArray.add((int)(Math.random()*1000));
        long time = System.currentTimeMillis() - start;
        System.out.println(N + "\t\t" + time);
    }
    System.out.println("large size initialized");
    for (int N = 200_000; N <= 10 * 200_000;
        N += 200_000) {
        long start = System.currentTimeMillis(); //
        capacity = 100_000
        MyArray mArray = new MyArray();
        for (int n = 1; n < N; n++)
            mArray.add((int)(Math.random()*1000));
        long time = System.currentTimeMillis() - start;
        System.out.println(N + "\t\t" + time);
    }
}
```

N	MyArray(N)			MyArray()		
	1 st	2 nd	3 rd	1 st	2 nd	3 rd
200_000						
400_000						
600_000						
800_000						
1_000_000						
1_200_000						
1_400_000						
1_600_000						
1_800_000						

2_000_000						
-----------	--	--	--	--	--	--

In your opinion,

1. Given the different characteristic between the fixed sized and the expandable array (MyArray(N) vs. MyArray()), which type of array's execution time should be faster?

2. In your opinion, how this experiment should be improved such that the execution time should reflect its true characteristic.

submission: MyArray_XXYYYY.java and this pdf.

Due date: TBA