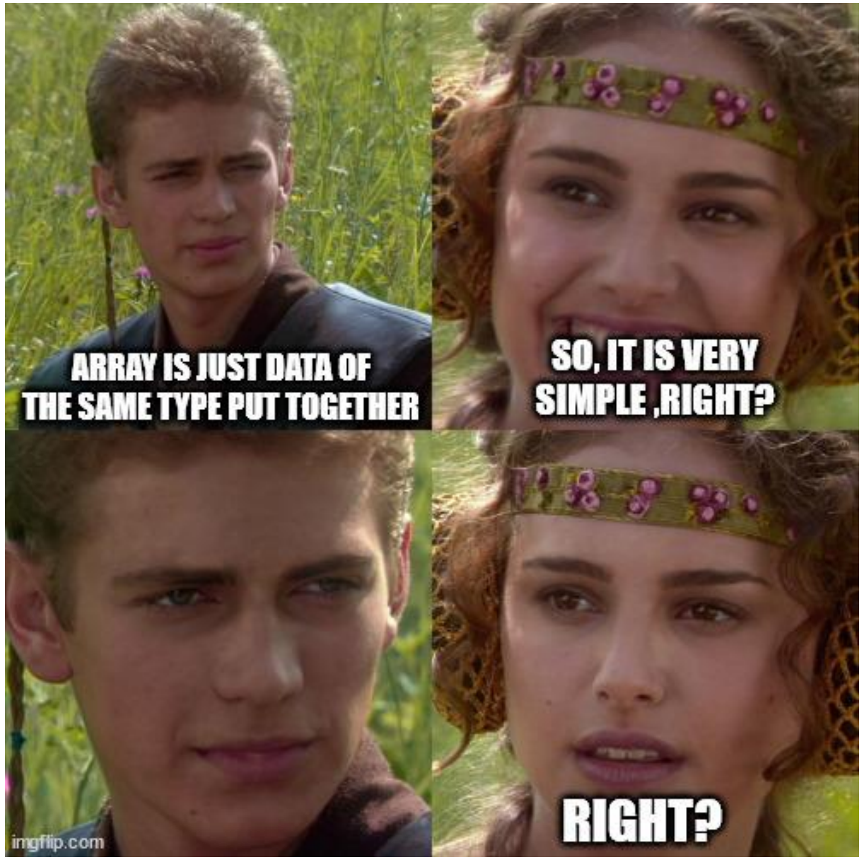
Ch.03

Array

(kmitl) cs-department

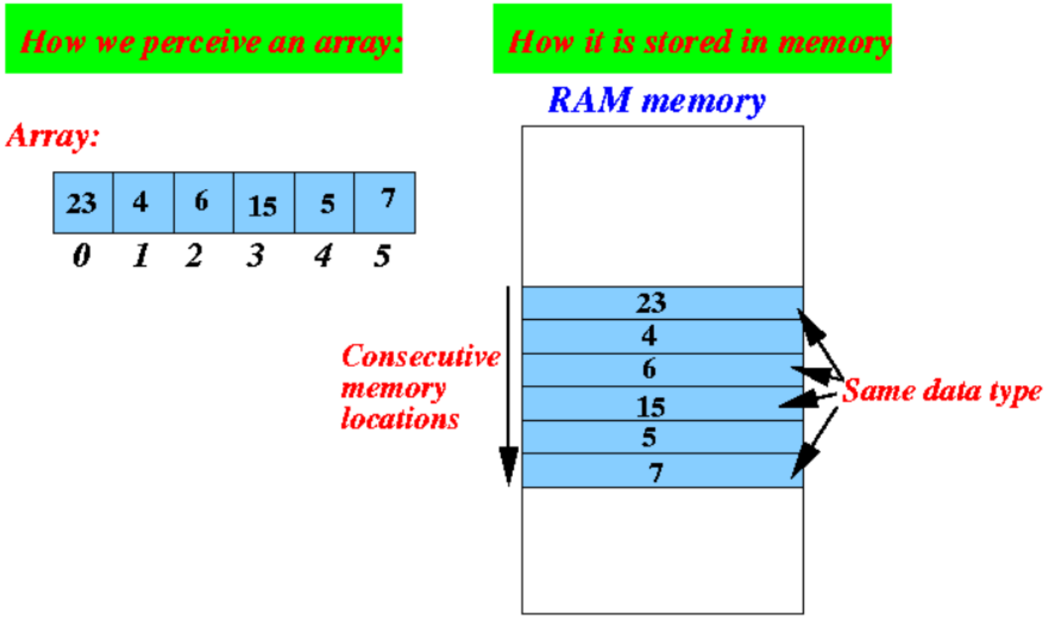
Outline

- Array Introduction
- Declaration and Instantiation
  - Random access (retrieve/update)
  - Add/Insert unordered/ordered array
  - Search in unordered/ordered array
  - Delete unordered/ordered array
- Array Limitation
- Expanding an array



Array Introduction

- One of the most basic data structure.
- An array (hence the name) of data of the same type referred to by a common name.
  - Data are put next to each other, without any space, in the physical memory.
  - In JAVA, the data are in heap (dynamic memory) of JVM.



Declaration and Instantiation

- Starts from variable declaration

```
byte b[];
```

  - *Note that we can write **byte[] b;***
  - *Another note, the follow two lines of code are not the same:*

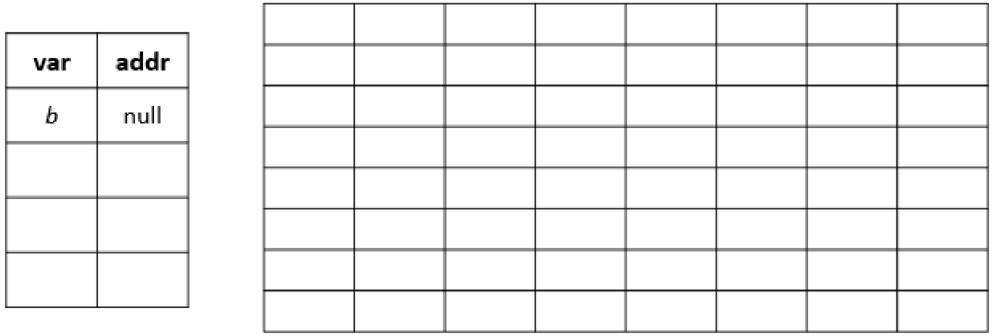
```
byte a[], c; // c is just byte
byte[] b, d; // both b and d are
```

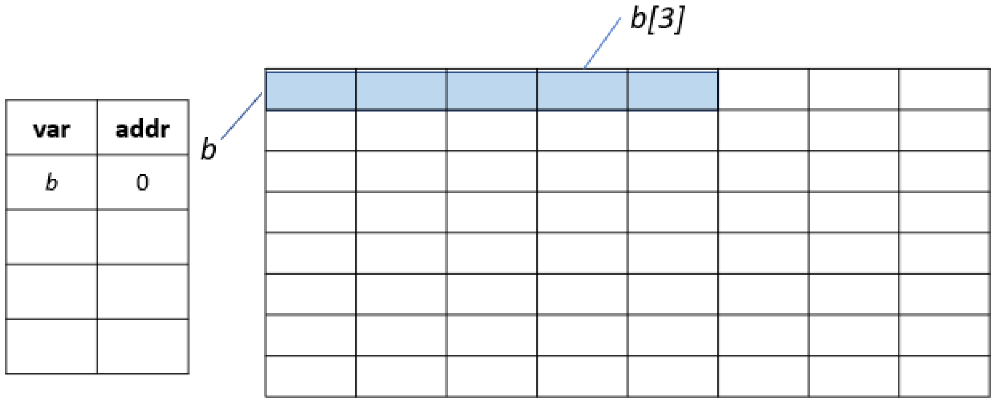
- Now, create it.

```
b = new byte[5];
```

  - data are allocated next to each others.
  - We can refer to them as b[0]..b[4]
- In C/C++ and some other languages address of b is the actual physical memory address. So, we can access memory at the address (b+2) for b[2].

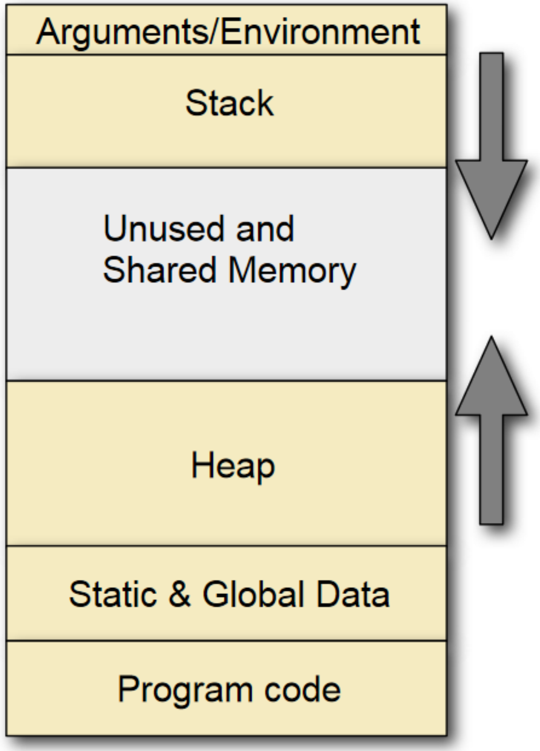
- Let's assume there are 64 bytes of dynamic memory





Array Operations

- We are to analyze the following array operation
  - Random access (retrieve/update)
  - Add/Insert unordered/ordered array
  - Search in unordered/ordered array
  - Delete unordered/ordered array



Process Memory Layout

MyArray.java & ArrayTester.java

MyArray.java

```
public class MyArray {
    int MAX_SIZE = 5;
    int data[] = new int[MAX_SIZE];
    int size = 0;

    // your code here

    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("[");
        for(int i=0; i<size-1; i++) {
            sb.append(data[i]);
            sb.append(",");
        }
        if(size>0) sb.append(data[size-1]);
        sb.append("]");
        return sb.toString();
    }
}
```

What is the Big-O of toString()?

ArrayTester.java

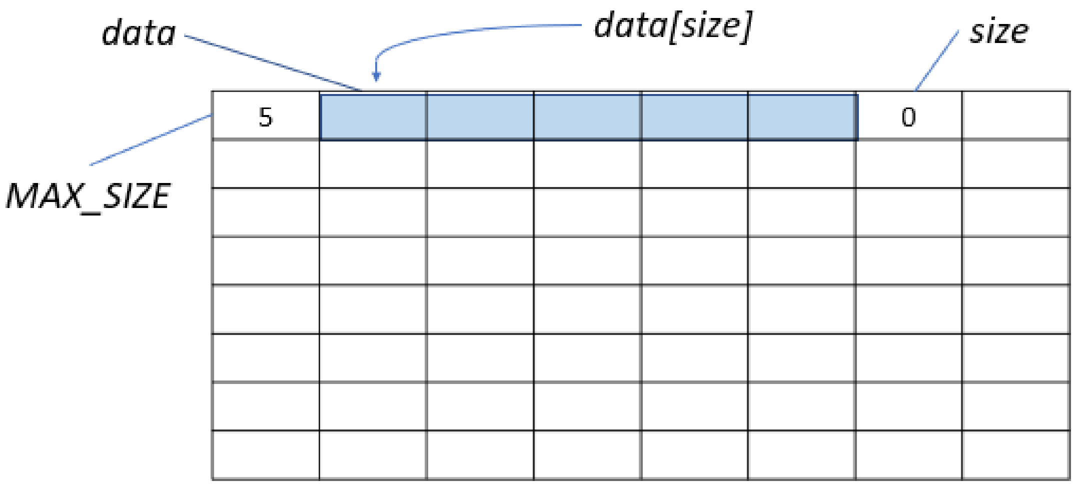
```
public class ArrayTester {

    public static void main(String args[]) {
        MyArray mArray = new MyArray();

        // your test code here

        System.out.print(mArray.toString());
    }
}
```

MyArray Setup

- In practice, we do not care about the actual address.

MyArray.java

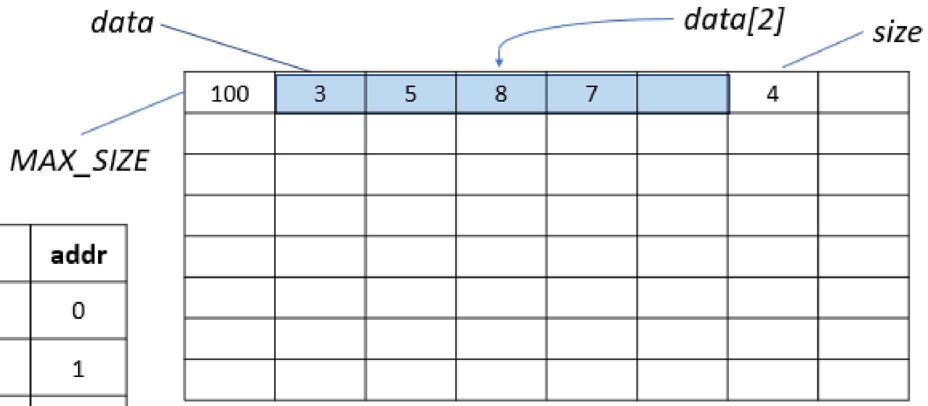
```
public class MyArray {
    int MAX_SIZE = 5;
    int data[] = new int[MAX_SIZE];
    int size = 0;

// more code below here
```



var	addr
MAX_SIZE	0
data	1
size	6

Accessing Array Data

- Random access data in arrays is done by access array at the starting point + index
  - Ex: address of data[2] is at 1+2 = 3
- In MyArray, we encapsulate data by method getAt()/setAt().
  - This way, we can control what happen to our data.
  - The figure illustrates getAt(2) which will return 8.
  - What is the Big-O of setAt()/getAt?



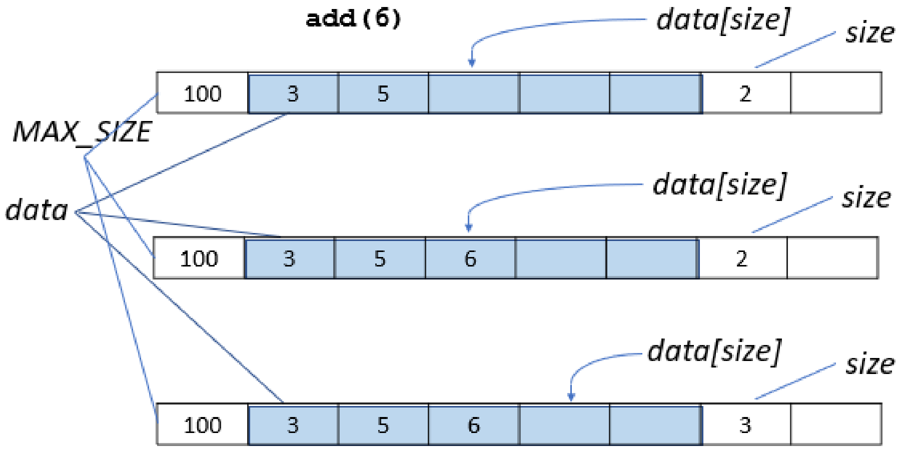
var	addr
MAX_SIZE	0
data	1
size	6

getAt() & setAt()

```
public int getAt(int i) {
    return data[i];
}

public void setAt(int d, int i) {
    data[i] = d;
}
```

Method add(int d)

- We want to add a new data into our array.
- We do not care about order.
- Best place to put is at the end of array.
  - So, we do not have to move any of the old data.



- Let's implement this.

Method add() version 1

```
public void add(int d) {
    data[size]=d;
    size=size+1;
}
```

Method add() more compact version

```
public void add(int d) {
    data[size++]=d;
}
```

Method isFull() and isEmpty()

- Error Checking
  - When to check isFull()/isEmpty()?
  - Three ways to do this:
    - Before calling the method (v1)
    - In method (v2)
    - Check exception (v3)
- We will use methods **add()** and **isFull()** for demonstration
  - The following ideas are also applicable to index out of bound.
- Simple methods, I will just put it here
  - They are both $O(1)$

Method isFull()

`public boolean isFull() {`
`    return size==MAX_SIZE;`
`}`

Method isEmpty()

`public boolean isEmpty() {`
`    return size==0;`
`}`

Demonstration on isFull()

- If you really need performance, you can leave add() as is and do the checking yourself.
- fast(?) but need to worry programmers

Method add() revised

```
public int add(int d) {/* v2 */
    if(isFull())
        return -1;
    data[size++]=d;
    return size;
}
```

ArrayTester.java

```
public class ArrayTester {
    public static void main(String args[]) {
        MyArray mArray = new MyArray();
        /* v1 */
        if(!mArray.isFull())
            mArray.add(5);
        System.out.print(mArray.toString());
    }
}
```

- Another implementation practice is to return boolean, true if successfully added and false otherwise.
- It is a good practice to use isFull() in place of size == MAX_SIZE

Demonstration on isFull() (try – catch)

MyArray.java

```
// other code above

public class IsFullException extends Exception {
    public String toString() {
        return "MyArray is full.";
    }
}

public void add(int d) throws IsFullException {
                       /* v3 */
    if(isFull()) throw new IsFullException();
    data[size++] = d;
}

// other code below
```

ArrayTester.java

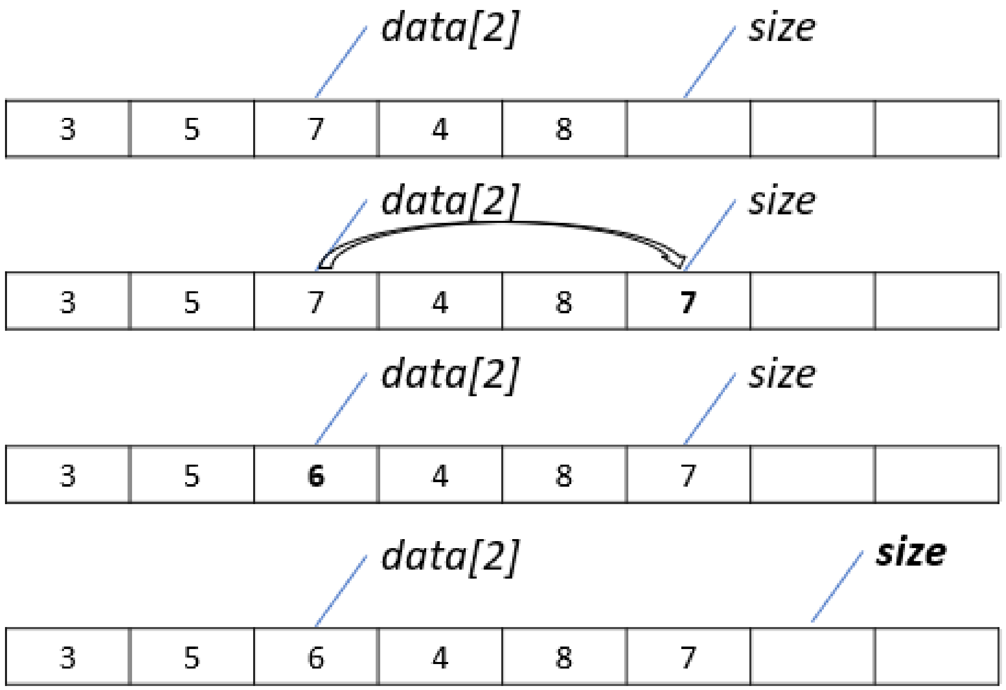
```
public class ArrayTester {

  public static void main(String args[]) {
    MyArray mArray = new MyArray();
    try {
        mArray.add(3);
        mArray.add(7);
        mArray.add(5);
        mArray.add(4);
        mArray.add(6);
        mArray.add(1);
    } catch (MyArray.IsFullException e) {
        // error handling code here
        System.out.println(e.toString());
    }
    System.out.print(mArray.toString());
  }
}
```

- *Slowest of them all, not good for this situation.*
- *A little too advance / too JAVA specific for this class*

Recap

- Array is an array of data of the same type, referred to by a common name.
- Array must be in a continuous memory for speed.
- Array in JAVA is an object, stored in dynamic memory.
- Random access data on an array is $O(1)$
- Adding data to an unordered array is $O(1)$
    - Later, more comprehensive add method

Method insert(int d, int idx)



- Say, if we want to insert 6 at index 2
- If order does not matter, consider the following operations
  1. Move data[2] to the end of array
  2. Put 6 at data[2]
  3. Increate size by one
- Step 1. must be done before step 2, but step 3 can be done at anytime.

- Let's implement this.

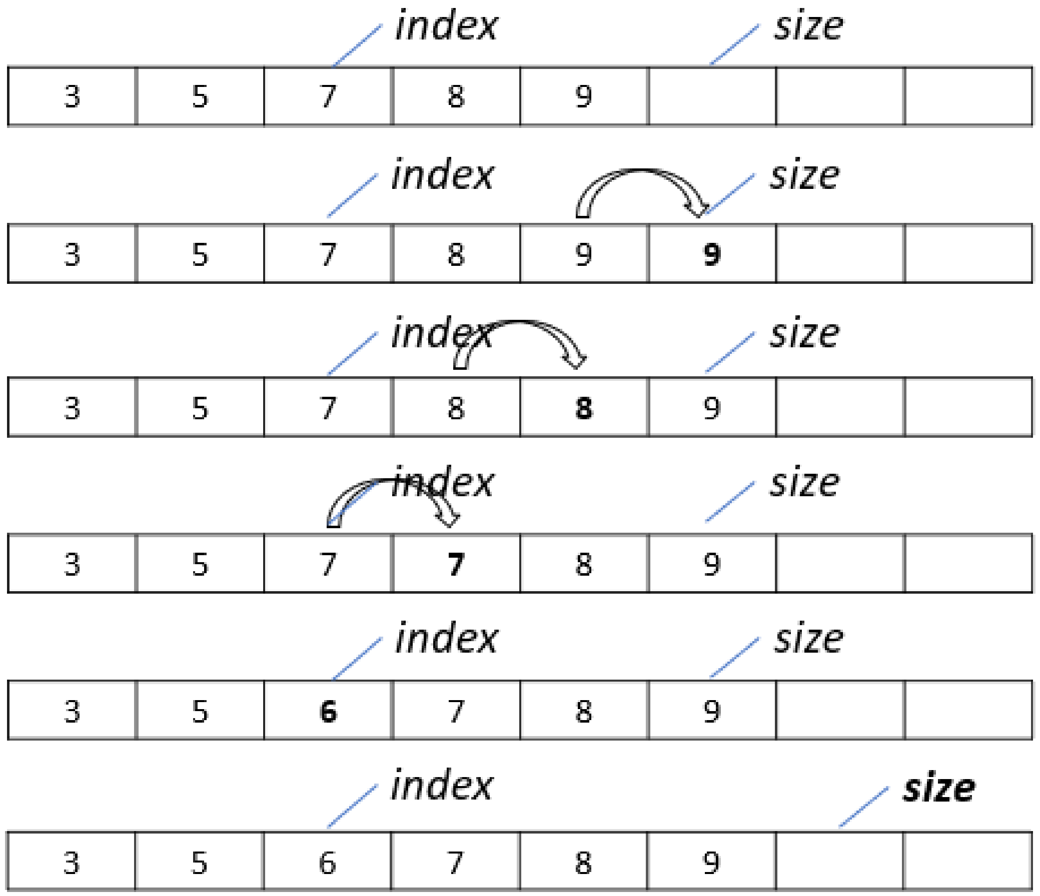
```
public void insert(int d, int index) {
    data[size++]=data[index];
    data[index] = d;
}
```

- What is the Big-O?

Insertion (Ordered Array)

- Right shift so that data[index] is available
  - Remark : the supplied idx must not break the order of data in the array
- Steps (To insert 6 in this ordered array)
  - Move everything >6 to the right one position.
  - Put 6 at the position
  - Increase size by 1

```
public void insert(int d, int index) {
    for(int i=size; i>index; i--) {
        data[i] = data[i-1];
    }
    data[index] = d;
    size++;
}
```



Big-O of Ordered Array Insertion

```
public void insert(int d, int index) {
    for(int i=size; i>index; i--) {
        data[i] = data[i-1];
    }
    data[index] = d;
    size++;
}
```

- Best Case (a.k.a. lucky): insert at the end, $O(1)$
- Worst Case: insert at the index 0, $O(n)$
- Average Case: count everything and average

- For average case, Let's start counting
  - Insert at 0: n copy operations
  - Insert at 1: n-1 copy operations
  - Insert at 2: n-2 copy operations
  - …
  - Insert at n-1: 1 copy operations
  - Insert at n: 0 copy operations

  Note that the other operations are constants, not affecting the Big-O
- Total operations are

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

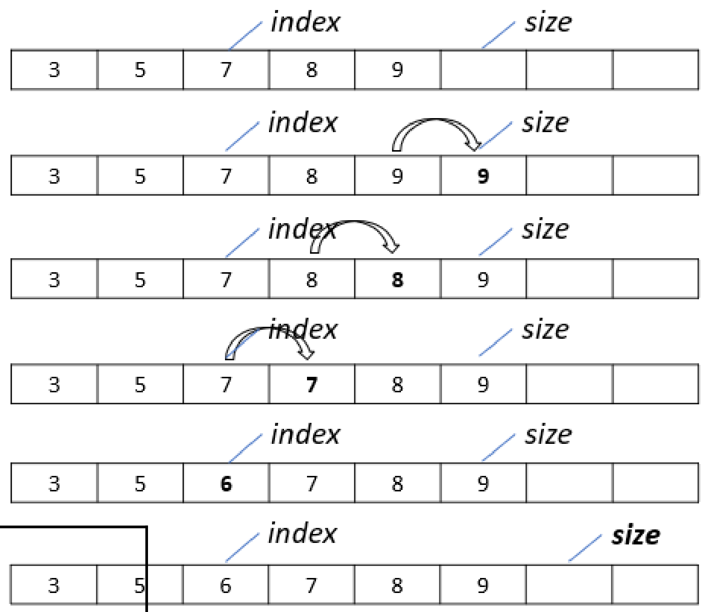
- We insert n times (from 0 to n-1)
- So, the **average** is

$$\frac{n(n+1)}{2} \div n = \frac{(n+1)}{2} \in O(n)$$

Overloaded insert(int d) on ordered array

- Recall, insert(int d, int idx) applies to unordered array
- insert(int d) is to insert into ordered array via computed index
  - No need the index parameter because the index must be computed for consistency.
- Same logic for add(int d) to ordered array
- Steps (insert 6 in this ordered array)
  - (Find where to insert and) Move everything > 6 to the right one position.
  - Put 6 at the position
  - Increase size by 1

```
index = size - 1;
while (data[index] > d) {
    data[index+1] = data[index];
    index--;
}
data[++index] = d;
size++;
```



Method find(int d)

- In an unordered array, best we can do is linear search

Simple implementation

```
public int find(int d) {
    int index=-1;
    for(int i=0; i<size; i++) {
        if(data[i]==d) {
            index = i;
            break;
        }
    }
    return index;
}
```

Compact version

```
public int find(int d) {
    for(int i=0; i<size; i++) {
        if(data[i]==d) {
            return i;
        }
    }
    return -1;
}
```

Efficiency

Best Case: $O(1)$

Worst Case: $O(n)$

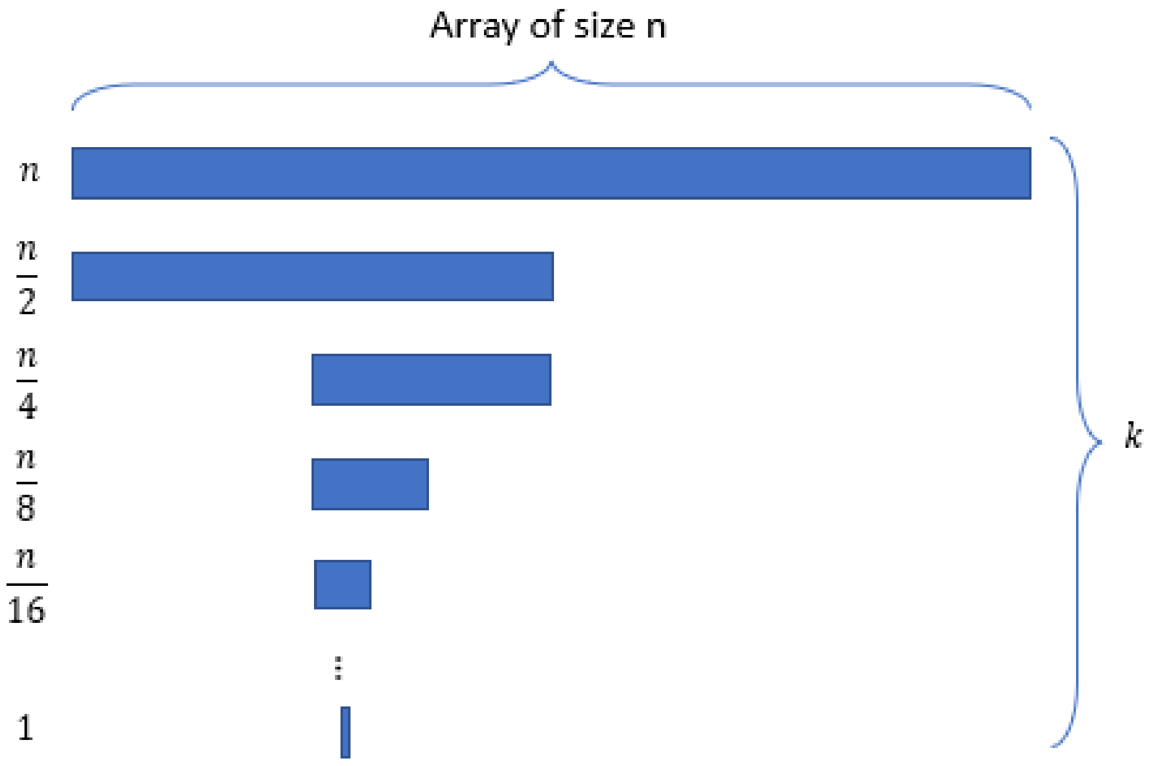
Average Case: $O(n)$

- Big-O is similar to insertion of an ordered array.

Method binarySearch(int d) on Ordered Array

- Binary search on ordered array is by far faster than performing a linear search .

```
public int binarySearch(int d) {
     int a = 0, b=size-1;
     while(a<=b) {
          int m = (a+b)/2;
          if(data[m]==d) return m;
          if(d<data[m]) b = m-1;
          else a = m+1; // d>data[m]
     }
     return -1;
}
```



Big-O of binary search

- Best Case is still O(1)
- Worst Case: we need to count
  - 1st iteration: n
  - 2nd iteration: $\frac{n}{2}$
  - 3rd: $\frac{n}{4}$
  - kth: $\frac{n}{2^{k-1}}$
  - Last iteration: 1

Stop criteria

  - If we run k round, we stop at $\frac{n}{2^{k-1}} = 1$ or $k = \log(n+1)$
  - So, the worst case is k round or $O(\log(n))$

- For average case, Let's start counting
  - The search data is at index $0, 1, 2, 3, \ldots, n-1$, not found
  - We will sum the number of operations of all search and divide it by $n+1$
- There are $[k = \log(n+1)]$:
  - 1 data that need only 1 comparison
  - 2 data that need 2 comparisons
  - 4 data that need 3 comparisons
  - …
  - 2^{k-1} data that need k comparisons

Arkkkk~~

- Total operations

  $= 1*2^0+2*2^1+3*2^2+4*2^3+..+k*2^{k-1}$

Sum								
$1*2^0$	=	2^0						
$+2*2^1$	=	2^1	+ 2^1					
$+3*2^2$	=	2^2	+ 2^2	+ 2^2				
$+4*2^3$	=	2^3	+ 2^3	+ 2^3	+ 2^3			
$+..$	=	…	…	…	…			
$+k*2^{k-1}$	=	2^{k-1}	+ 2^{k-1}	+ 2^{k-1}	+ 2^{k-1}	+	… +	2^{k-1}

Formula

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

$$= \boxed{2^k-1} + \boxed{2^k-1-2^0} + \boxed{2^k-1-2^0-2^1} + \boxed{2^k-1-2^0-2^1-2^2} + \ldots$$

$$+ \boxed{2^k-1-2^0-2^1-2^2-\ldots-2^{k-2}}$$

$$= \mathbf{2^k}-2^0 + \mathbf{2^k}-2^1 + \mathbf{2^k}-2^2 + \mathbf{2^k}-2^3 + \mathbf{2^k}-2^4 + \ldots + \mathbf{2^k}-2^{k-1}$$

$$= \mathbf{k2^k}-(2^0+2^1+2^1+2^1+\ldots+2^{k-1})$$

$$= \mathbf{k2^k}-(2^k-1) = k2^k - 2^k + 1$$

$$= (k-1)2^k+1$$

- So, average = $\frac{(k-1)2^k+1}{n+1} = \frac{(k-1)2^k+1}{2^{k-1}+1} \in O(k) = O(\log n)$

Recap

- Let summarize using a table

Methods	Best case	Worst case	Average case
Add (unordered)	$O(1)$	$O(1)$	$O(1)$
Insert unordered array	$O(1)$	$O(1)$	$O(1)$
Insert ordered array / add (ordered)	$O(1)$	$O(n)$	$O(n)$
Find unordered array	$O(1)$	$O(n)$	$O(n)$
Binary search ordered array	$O(1)$	$O(\log n)$	$O(\log n)$

- Now, only method delete left.

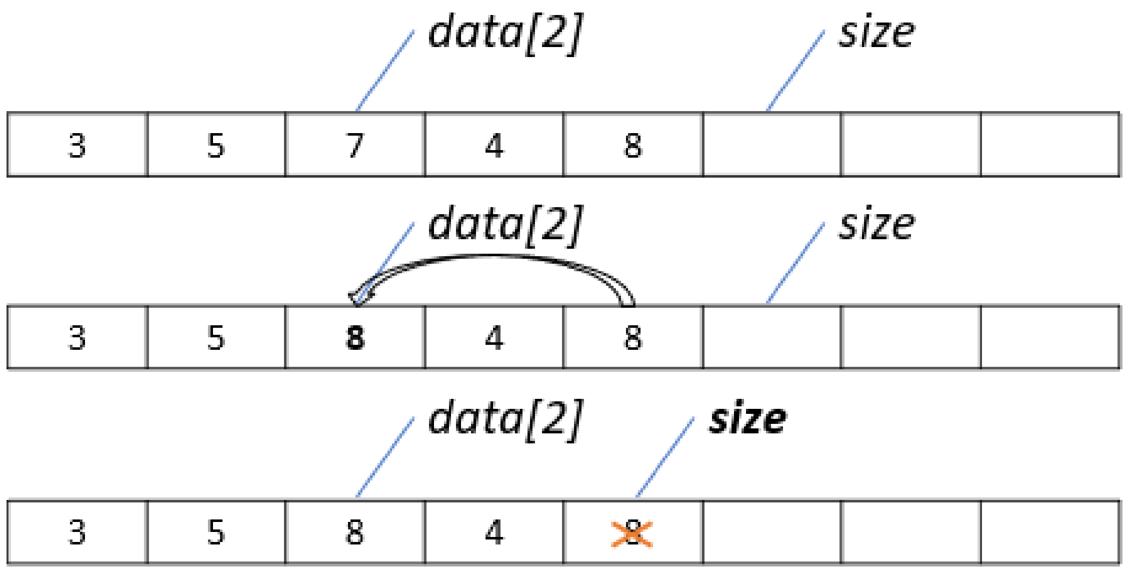
Delete from an Unordered Array delete(int index)

- If order is not important, it can be done in $O(1)$ time.
- If we want to delete 7 at index=2
    1. Copy last data to index 2
    2. Decrease size by one

    Note that the last element will be removed automatically.
- Very simple to implement

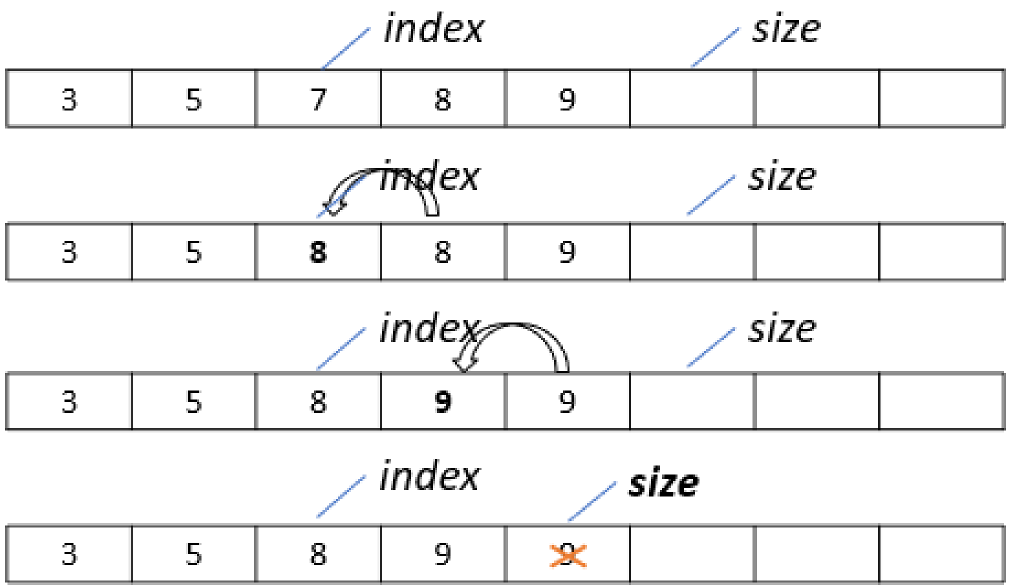
```
public void delete(int index) {
    data[index] = data[--size];
}
```

Note that we skip the searching for data step since that would mean $O(n)$.



Delete from an Ordered Array
delete(int index)

- Very similar to insert into an ordered array.
  - Move everyone to the left, instead of the right
- Steps from delete at index=2
  - Start from the index
  - Move data from the right to the current place
  - Keep going to the right and do the same thing until the end
  - Decrease size by 1
- Big-O is similar to insert into an ordered array.



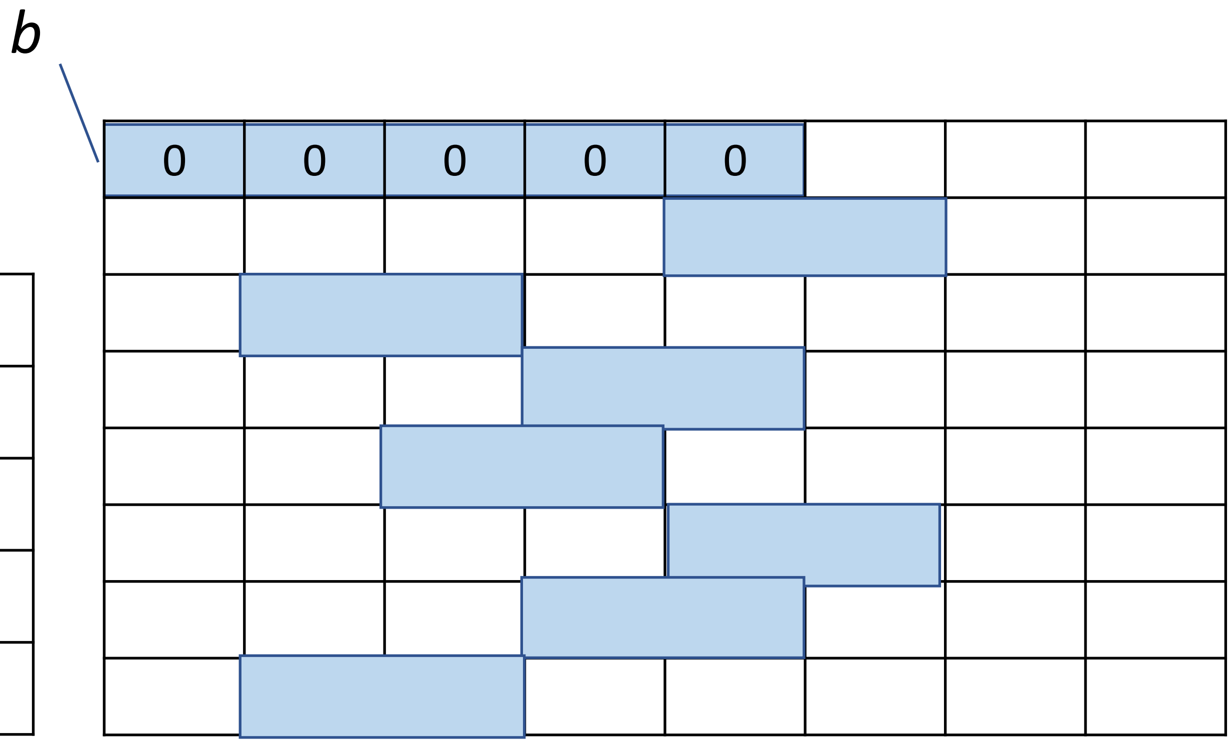
- Let's implement this.

```
public void delete(int index) {
    for(int i=index; i<size-1; i++) {
        data[i] = data[i+1];
    }
    size--;
}
```

Limitation of Array

- Given the shaded areas are occupied.
- It is not possible to
- create an array of size 10.
  - Although we have more than enough memory.

var	addr
b	0



Expanding Array (add())

- Array data structures can not be expanded.
- What we can do is
  - Create a bigger array
  - Copy all data to the new one
  - Point to the new one
  - Delete the old one
- Modified add();

```
public void add(int d) {
    if(isFull()) expands();
    data[size++] = d;
}
```

- Seems too be slow, right?

- Two versions of expands()

```
void expands() { // version A
    MAX_SIZE = MAX_SIZE+M;
    int newData[] = new int[MAX_SIZE];
    System.arraycopy(data,0,newData,0,size);
    data = newData;
    System.gc();
}
```

For sufficiently large M such as M=1000

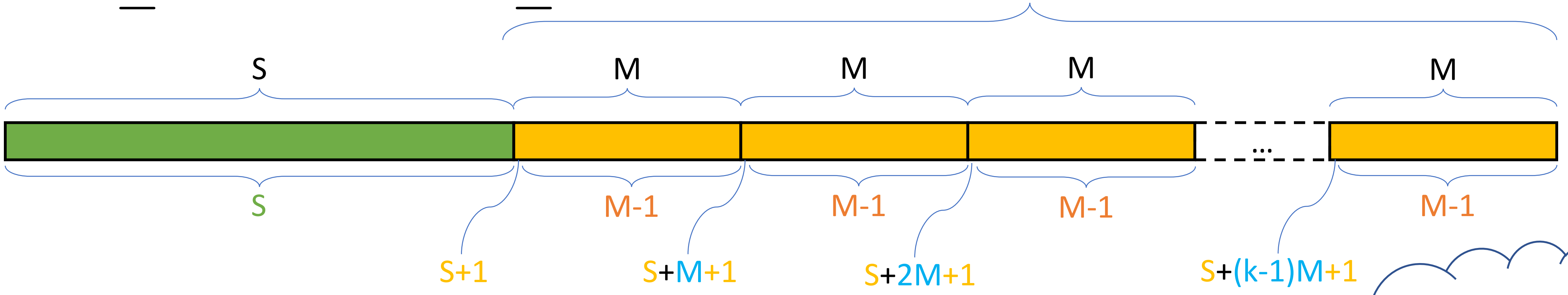
```
void expands() { // version B
    MAX_SIZE = 2*MAX_SIZE;
    int newData[] = new int[MAX_SIZE];
    System.arraycopy(data,0,newData,0,size);
    data = newData;
    System.gc();
}
```

The Big-O

- For `expand()`, Big-O is always $O(n)$
- For `add()`
  - Best case is $O(1)$, we do not have to expand.
  - Worst case is $O(n)$, we must expand and copy n times.
  - Let compute for the average case for both version.

Version A

```
MAX_SIZE = MAX_SIZE+M;
```



$$n = S + kM$$

$$k = \frac{n - S}{M}$$

- To compute average, we assume that data are added n times
  - Add data from index 0 to S-1 take S operations
  - Add data at index S takes S+1 operations, since we need to copy and add a new data.
  - Add data from index S+1 to S+M-1 take M-1 operations
  - Add data at index S+M takes S+M+1 operations
  - And so on..
- Total operation is

$$S + k(S+1) + k(M-1) + [M + 2M + 3M + \cdots + (k-1)M]$$

$$= (k+1)S + kM + \left(\frac{(k-1)k}{2}\right)M$$

Average of Version A

$$n = S + kM$$
$$k = \frac{n - S}{M}$$

- Total Operations

$$Sum = (k+1)S + kM + \left(\frac{(k-1)k}{2}\right)M$$

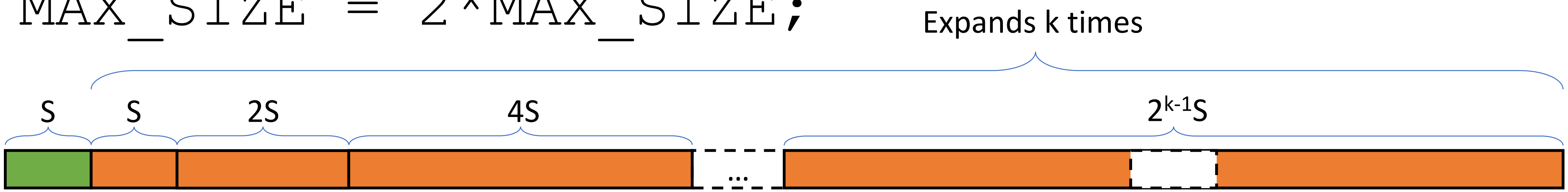
$$Sum = \left(\frac{n-S}{M} + 1\right)S + \frac{n-S}{M}M + \left(\frac{\left(\frac{n-S}{M} - 1\right)\frac{n-S}{M}}{2}\right)M$$

$$Sum = \left(\frac{n-S+M}{M}\right)S + n - S + \left(\frac{(n-S-M)(n-S)}{2M}\right)$$

- Without simplifying it further, we can see that it is $O(n^2)$
- Since average is $\frac{Sum}{n}$, so, the average is $O(n)$.

Version B

```
MAX_SIZE = 2*MAX_SIZE;
```



- First, let compute relationship between n and k

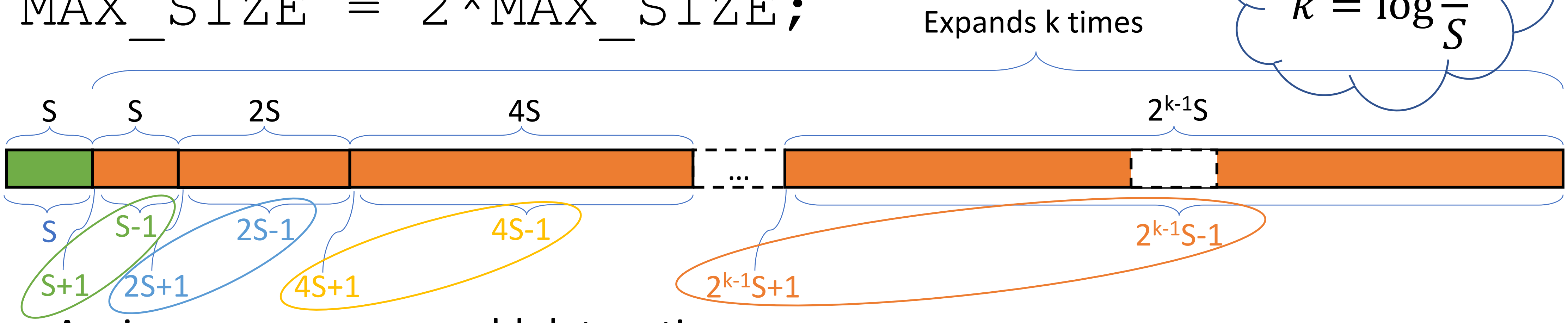
$$n = S + S + 2S + 4S + .. + 2^{k-1}S$$
$$n = (1 + 1 + 2 + 4 + .. + 2^{k-1})S$$
$$n = 2^k S$$
$$k = \log\frac{n}{S}$$

- Let compute average in the next slide

Version B

```
MAX_SIZE = 2*MAX_SIZE;
```

$$n = 2^k S$$
$$k = \log \frac{n}{S}$$



- Again, we assume we add data n times
  - Add data from index 0 to S-1 take S operations
  - Add data at index S takes S+1 operations, from copy and add new data
  - Add data from index S+1 to 2S-1 take S-1 operations
  - Add data at index 2S takes 2S+1 operations
  - And so on..
- The total operations is

$$S + 2S + 4S + 8S + .. + 2^k S = (2^{k+1} - 1)S = 2(2^k S) - S = 2n - S$$

- So, average is $\frac{2n-S}{n} \in O(1)$!

Example challenge

26. Remove Duplicates from Sorted Array

Solved ✓

Easy | Topics | Companies | Hint

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* `nums`.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first
two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (hence
they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first
five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (hence
they are underscores).
```

```java
public class Solution {
  public int removeDuplicates( int[] nums) {
      return 0;
  }
}

/* it is already sorted. Care to swap the first
next value to its correct position? */
```

Example challenge

75. Sort Colors

Solved

Medium | Topics | Companies | Hint

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]

Example 2:

Input: nums = [2,0,1]
Output: [0,1,2]

```
void sortColors(int[] nums) {
  int tmp, low = 0, mid = 0, high = nums.length - 1;
  while (mid <= high) {
    if (nums[mid] == 0) {
      // Swap arr[low] and arr[mid], move both forward
      tmp = nums[low];
      nums[low] = nums[mid];
      nums[mid] = tmp;
      /* use below technique leads to unexpected side effect
         (accessing same index nums[index] both LHS and RHS)
      //  nums[low] = nums[low] + nums[mid];
      //  nums[mid] = nums[low] - nums[mid];
      //  nums[low] = nums[low] - nums[mid];          */
      low++;
      mid++;
    } else if (nums[mid] == 1) {
      mid++;
    } else {
      tmp = nums[mid];
      nums[mid] = nums[high];
      nums[high] = tmp;
      // if the same variable is modified in-place incorrectly,
      // we may accidentally cause corruption,
      // Swap arr[mid] and arr[high], move high backward
      //  nums[mid] = nums[mid] + nums[high];
      //  nums[high] = nums[mid] - nums[high];
      //  nums[mid] = nums[mid] - nums[high];
      high--;
    }
  }
}
```

Summary

- Let summarize all methods of array data structures

Methods	Best case	Worst case	Average case
Add into an array	$O(1)$	$O(1)$	$O(1)$
Insert into an unordered array	$O(1)$	$O(1)$	$O(1)$
Insert / Add into an ordered array	$O(1)$	$O(n)$	$O(n)$
Find in an unordered array	$O(1)$	$O(n)$	$O(n)$
Binary search in an ordered array	$O(1)$	$O(\log n)$	$O(\log n)$
Delete from an unordered array	$O(1)$	$O(1)$	$O(1)$
Delete from an ordered array	$O(1)$	$O(n)$	$O(n)$
Expand an array	$O(\mathrm{n})$	$O(n)$	$O(n)$
Add with expand	$O(1)$	$O(n)$	$O(1)$

version B

an example of amortized analysis