Structures for Data

Advanced programming for mobile devices

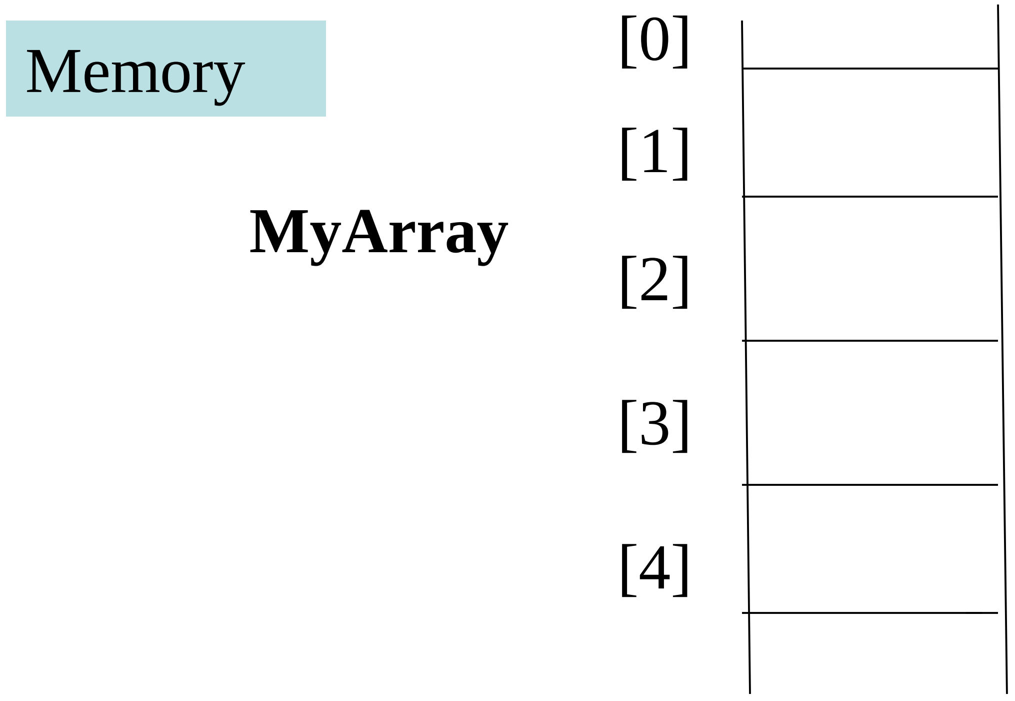
- Why use structures?
  - Organisation
  - Ease of coding
  - Ease of searching
  - Controlling how your data may be accessed by a user
- A requirement for the module – marks!

A simple structure: The Array

What is an array ?

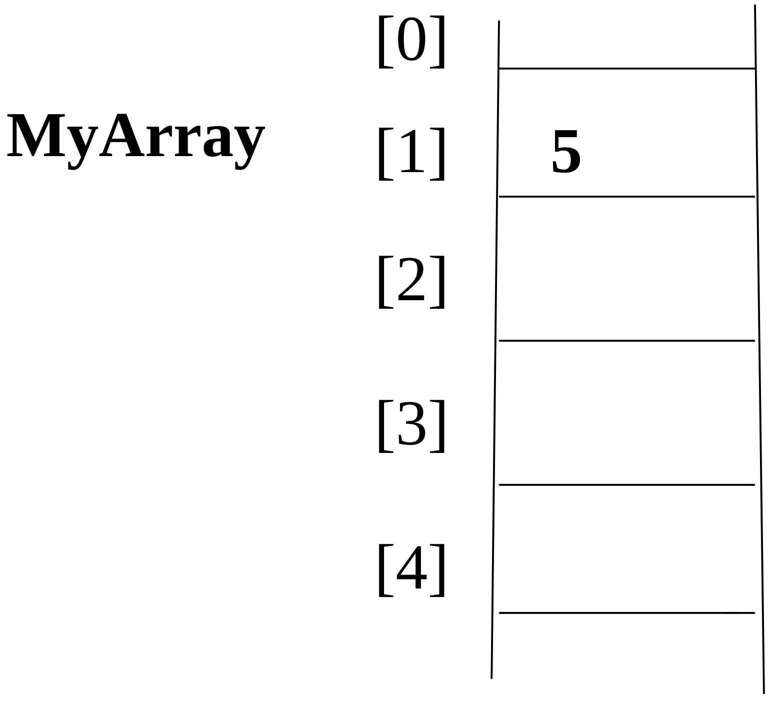
A structure which is one identifier but can have many sub values

int []MyArray = new int[5];



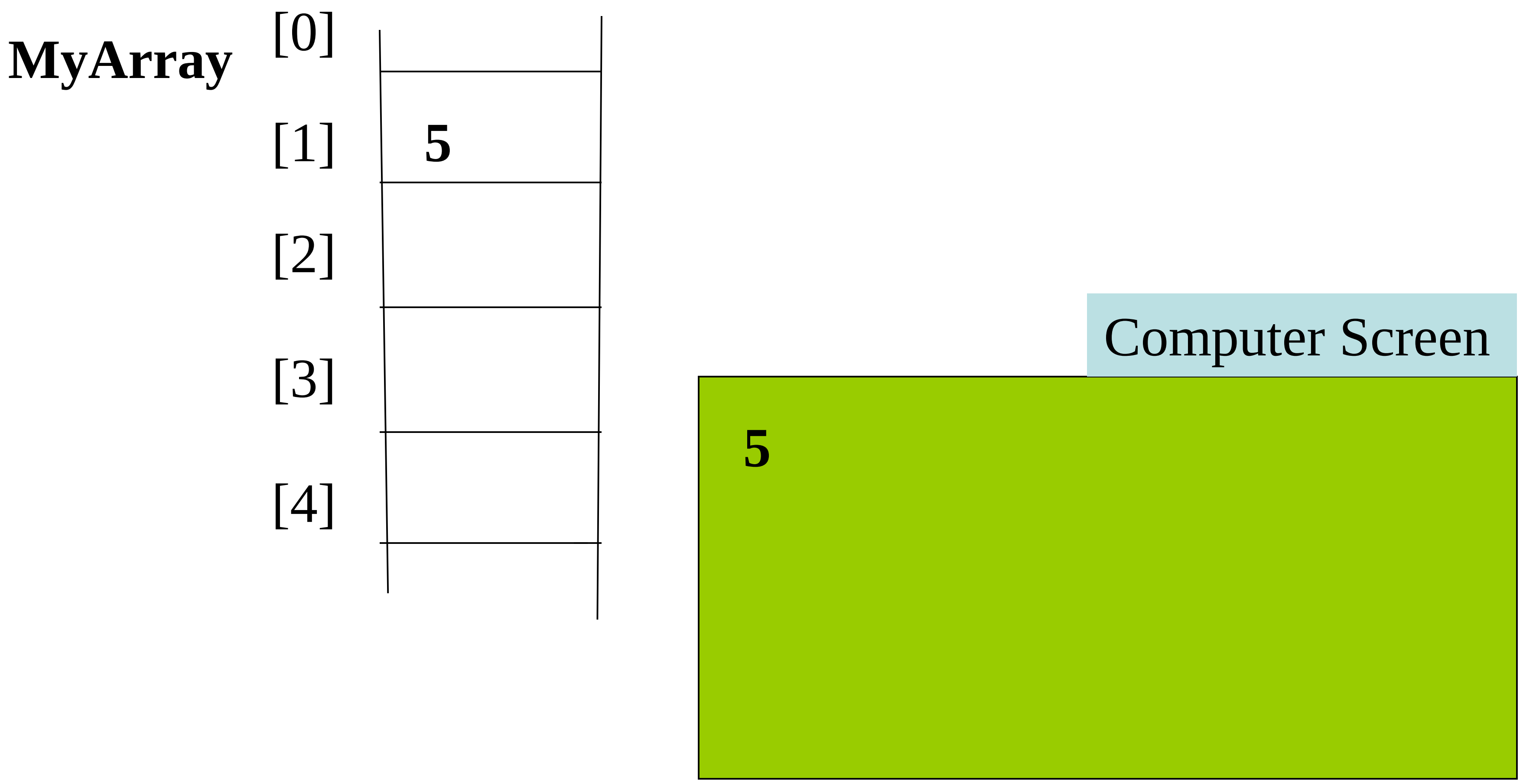
The Index values are used to identify the individual data stores

```
MyArray[1] = 5;
```



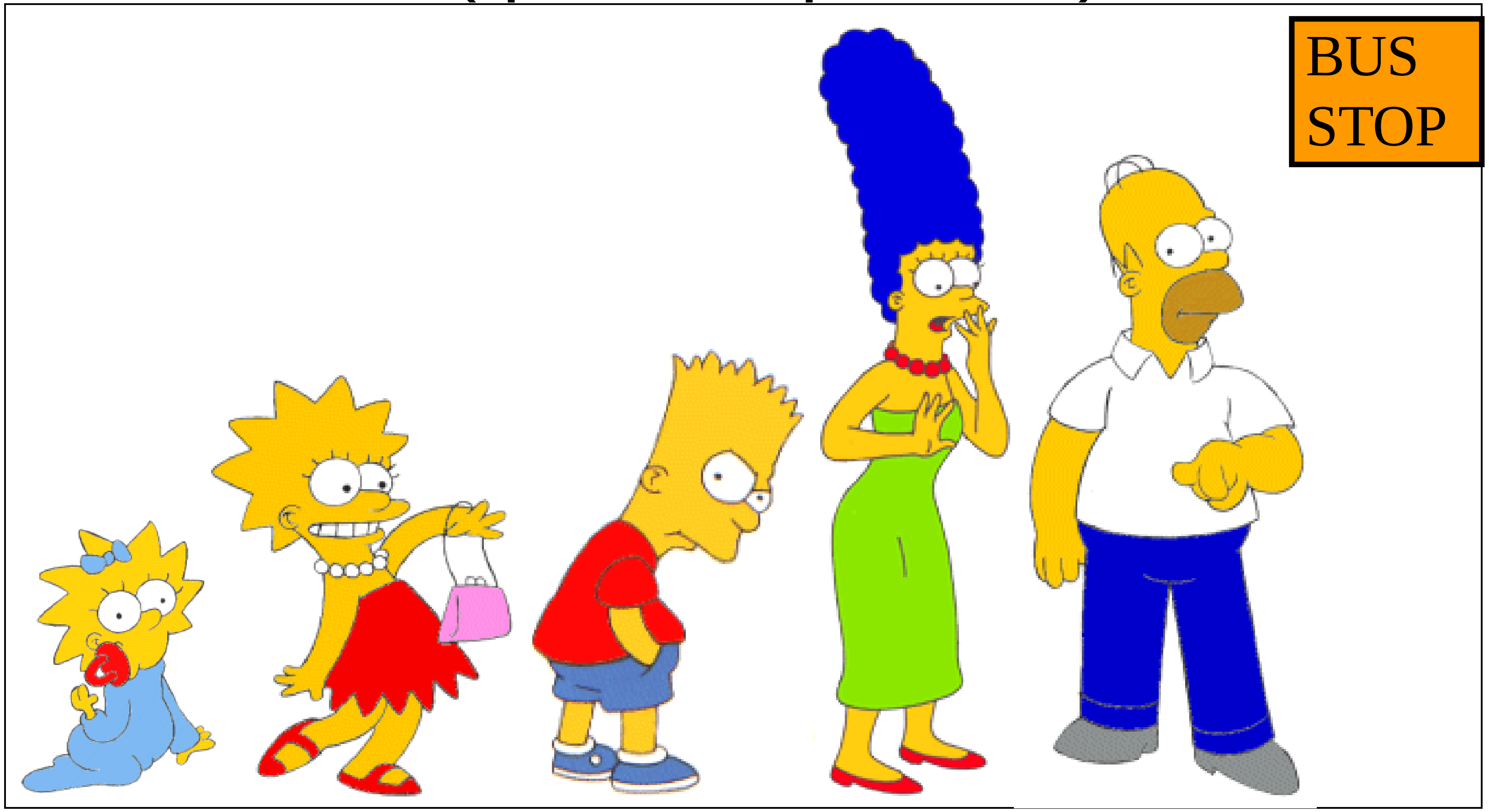
Retrieving Values :

```
Console.Writeln( MyArray [1]);
```



Queue concepts

- Illustration (queue of persons):



Queue concepts

- A **queue** is a first-in-first-out sequence of elements.
  Elements can added only at one end (the **rear** of the queue) and removed only at the other end (the **front** of the queue).
- The **length** of a queue is the number of elements it contains.
- An **empty** queue has length zero.

Queue ADT: requirements

- Requirements:
  1) It must be possible to make a queue empty.
  2) It must be possible to test whether a queue is empty.
  3) It must be possible to obtain the length of a queue.
  4) It must be possible to add an element at the rear of a queue.
  5) It must be possible to remove the front element from a queue.
  6) It must be possible to access the front element in a queue without removing it.

It must be possible to make a queue empty.

A function is needed that will set the values in the array to a value that signifies it is clear and set the front and back of the queue to the first index of the array.

It must be possible to test whether a queue is empty.

A function is required that will test if the queue is empty.

It must be possible to obtain the length of a queue.

A function or value is required that will keep track of the number of values in the array.

It must be possible to add an element at the rear of a queue.

a function is needed that will add an element to the array. The position of the next element will need to be marked. What if we reach the end of the array?

It must be possible to remove the front element from a queue.

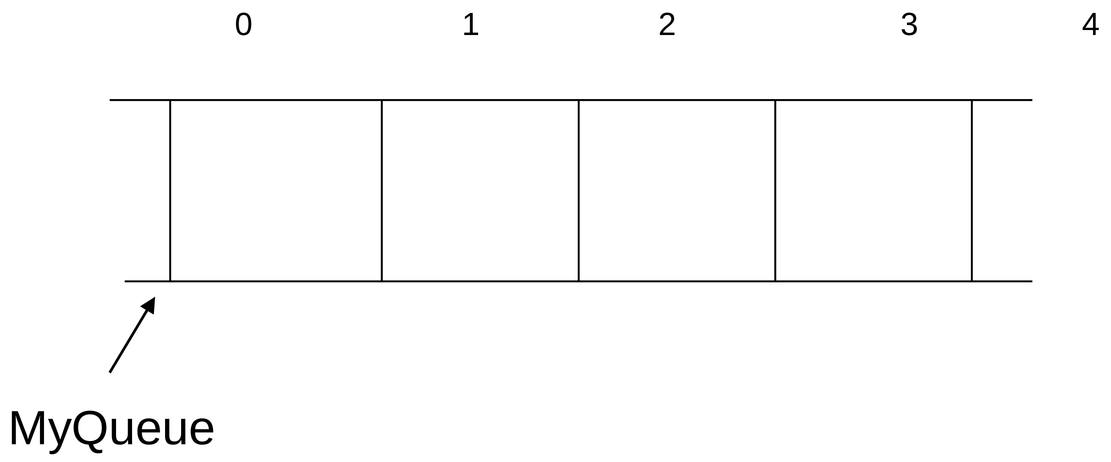
A function is required that will allow the removal of an element. The end of the queue will need to be re-set to the next position. What happens when you reach the end of the queue?

It must be possible to access the front element in a queue without removing it.

A function is required that will read the top element value.

The Queue: Visualisation

```
int []MyQueue = new int[5];
```

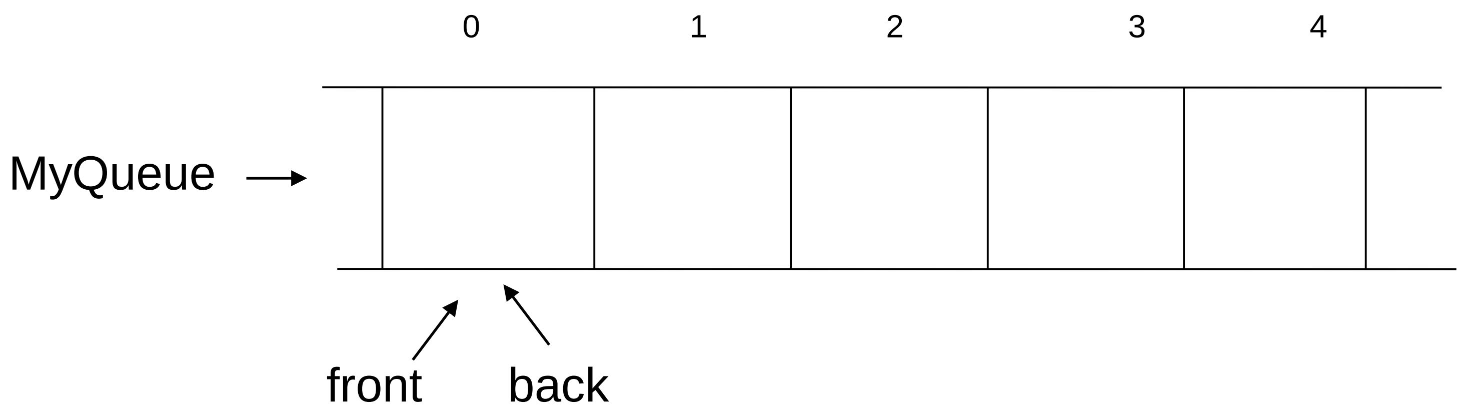


Initialisation

Length = 0

Front = 0

Back = 0

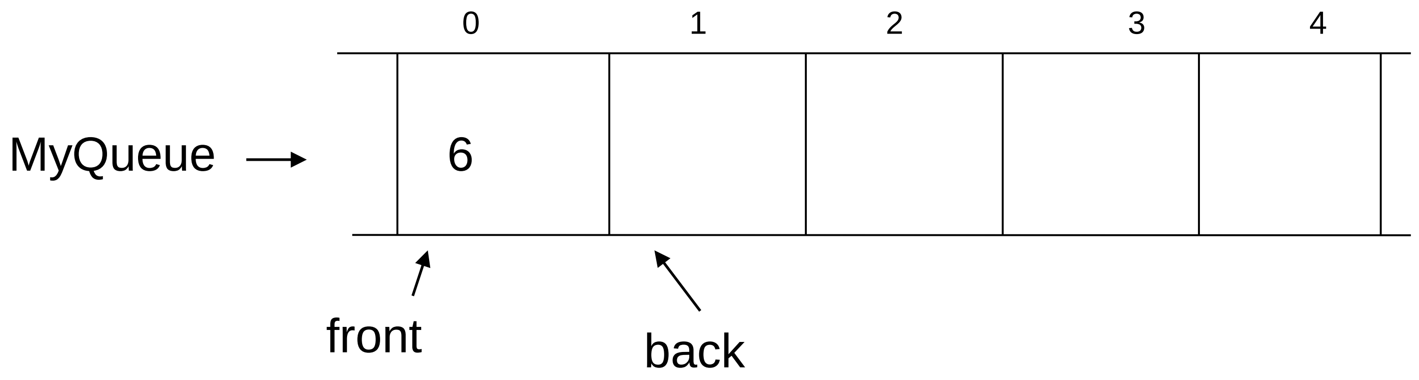


addtoqueue

Assign item to array[back]

Increment back

Increment Length



Front = 0

Back = 1

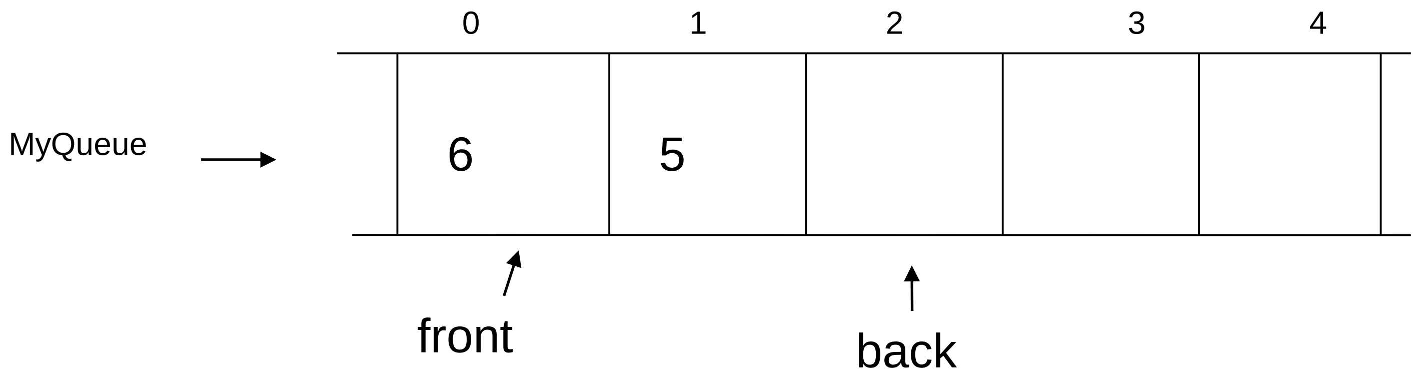
Length = 1

addtoqueue

Assign item to array[back]

Increment back

Increment Length

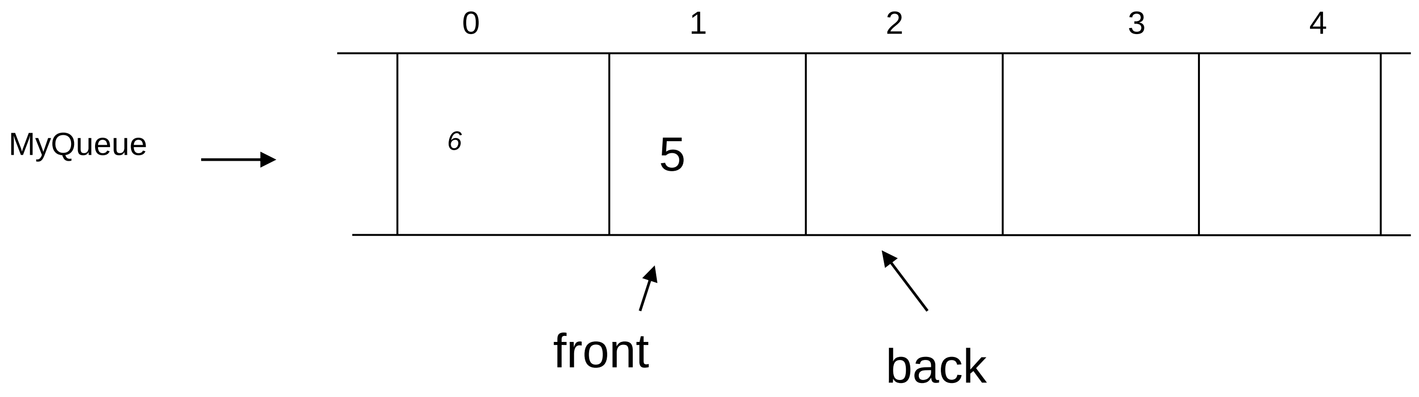


Front = 0

Back = 2

length = 2

Remove from queue



Front = 1

Back = 2

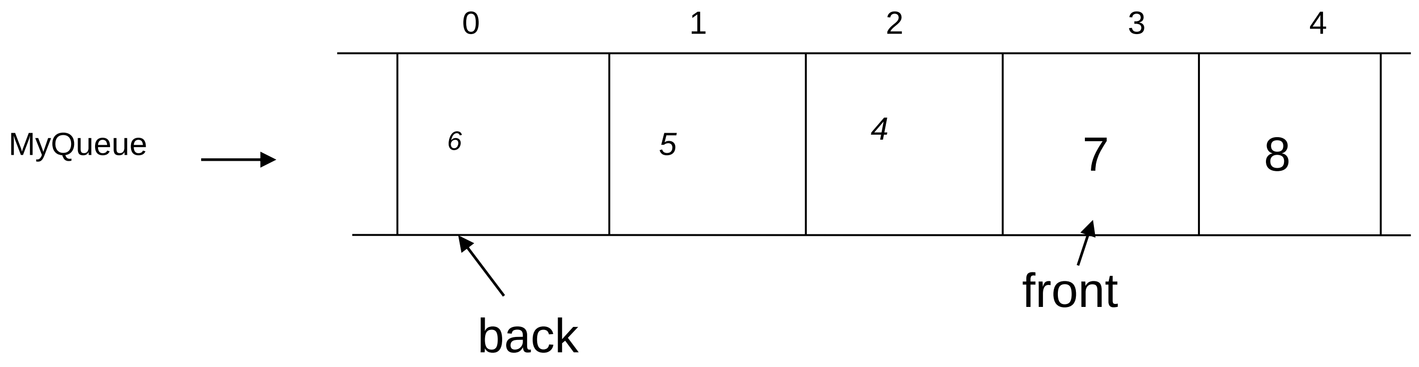
length = 1

Increment front

decrement length

Add to queue?

The array is not full but we have reached the length of the array



Front = 3

Back = 0

length = 2

Assign item to array[back]

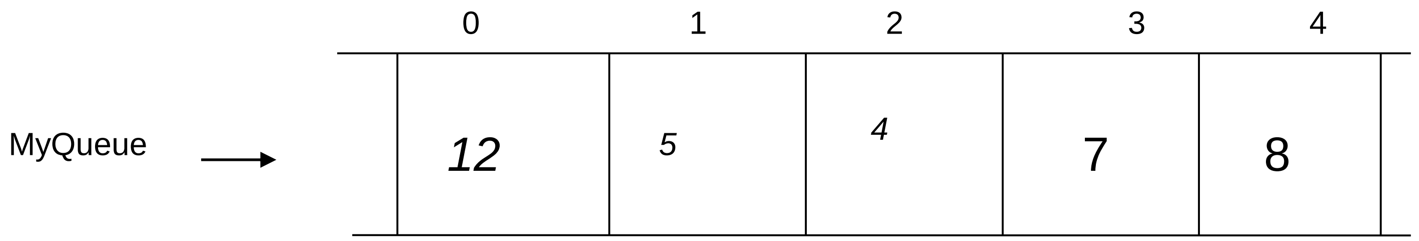
Increment back

Increment length

If back = array length *(i.e. 5)*

Set back to 0

Add to queue



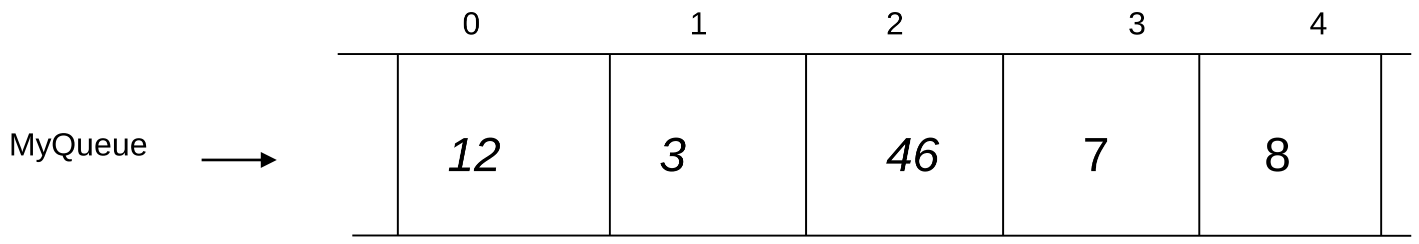
```
Assign item to array[back]
Increment back
Increment length
 If back = array length
        Set back to 0
```

Front = 3

Back = 1

length = 3

Add to queue?



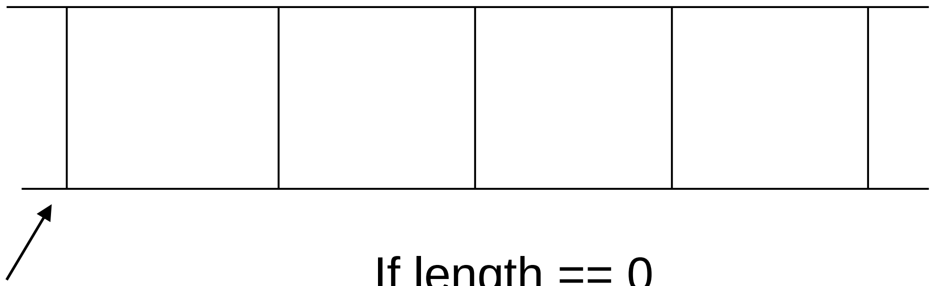
```
If array full
        exit
Assign item to array[back]
Increment back
Increment length
If back = array length
    Set back to 0
```

Front = 3

Back = 3

length = 5

Remove an item

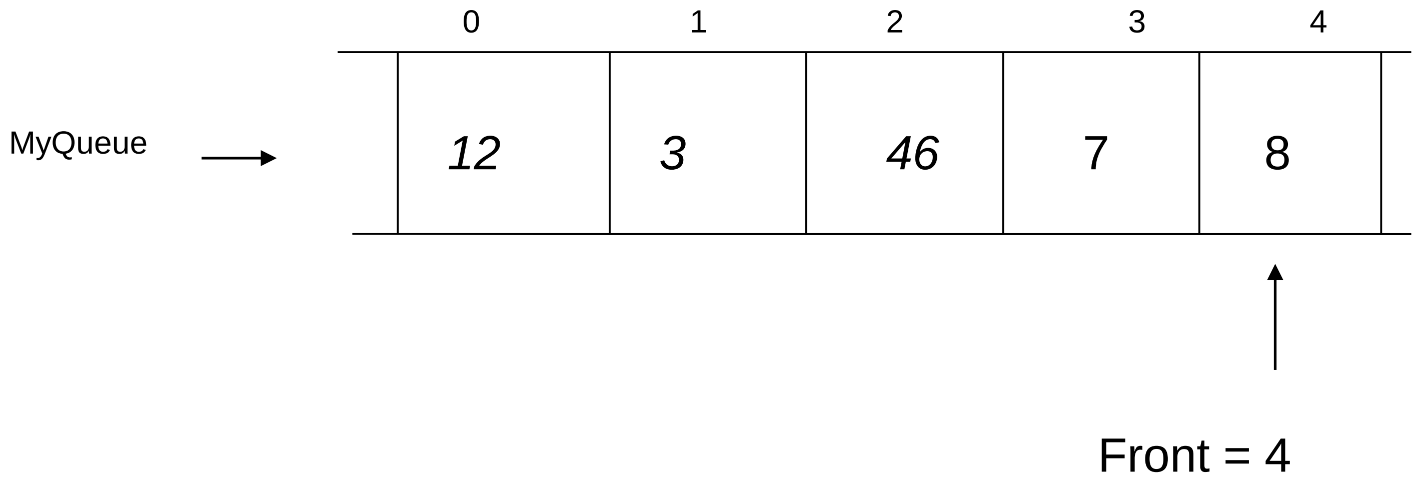


MyQueue

```
If length == 0
        exit
else
        Increment front
        decrement length
```

Size = 0

Remove an item



```
If length == 0
        exit
else
        Increment front
        decrement size
```

```
**If front = array length**
        **set front to 0**
```

- Possible contract:

```
public interface Queue {
    // Each Queue object is a queue whose
elements are integers.
    //////////////// Accessors
////////////////
    public boolean isEmpty ();
    // Return true if and only if this queue is empty.
    public int size ();
    // Return this queue’s length.
    public int getFirst ();
    // Return the element at the front of this queue.
    public void clear ();
    // Make this queue empty.
    public void addLast (int elem);
    // Add elem as the rear element of this queue.
    public int removeFirst ();
    // Remove and return the front element of this
queue.
}
```

- Java implementation:

```
public class Queue
    private int[] elems;
    private int front, rear, length;
    ///////////////// Constructor
/////////////////
    public Queue (int maxLength) {
        elems = new int[maxLength];
        front = rear = length = 0;
    }
```

- Java implementation *(continued)*:

```
      public boolean isEmpty () {
    return (length == 0);
  }
  public int size () {
    return length;
  }
  public int getFirst () {
    if (length == 0)  throw …;
    return elems[front];
  }
```

```
public void clear () {
    front = rear = length = 0;
}
public void addLast (int elem) {
  if (length == elems.length)  throw ...;
  elems[rear++] = elem;
  if (rear == elems.length)  rear = 0;
  length++;
}
```

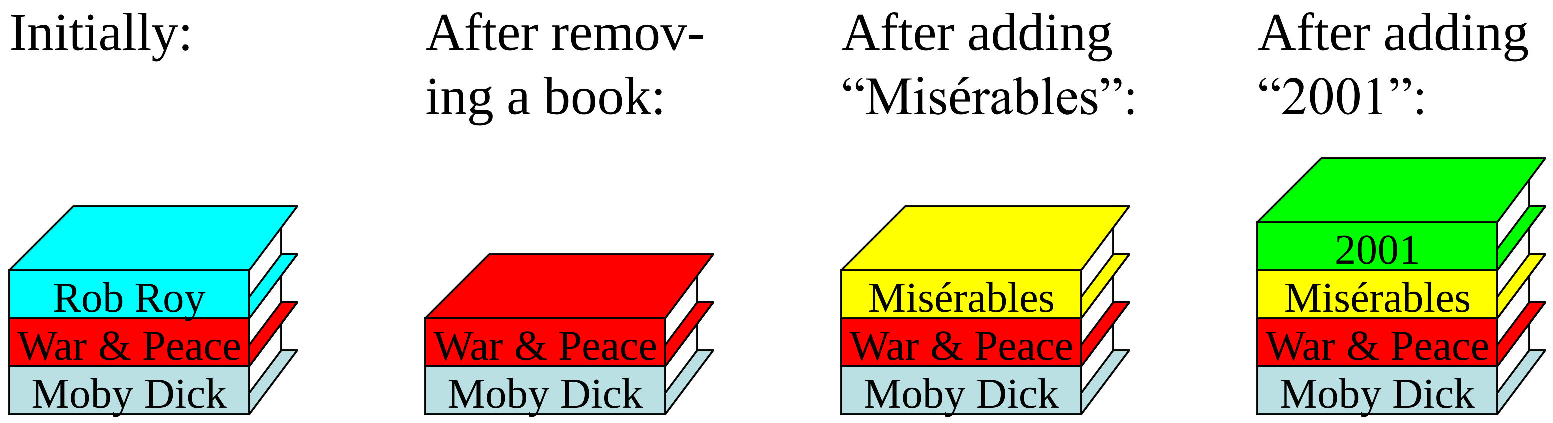
```
public int removeFirst () {
if (length == 0)  throw ...;
  int frontElem = elems[front];
  front++;
if (front == elems.length)
    front = 0;
    return frontElem;
  }
}
```

Stack concepts (1)

- A **stack** is a last-in-first-out sequence of elements.
  Elements can added and removed only at one end
  (the **top** of the stack).
- The **depth** of stack is the number of elements it contains.
- An **empty** stack has depth zero.

Stack concepts (2)

- Illustration (stack of books):



Stack ADT: requirements

- Requirements:
  1) It must be possible to make a stack empty.
  2) It must be possible to add (‘push’) an element to the top of a stack.
  3) It must be possible to remove (‘pop’) the topmost element from a stack.
  4) It must be possible to test whether a stack is empty.
  5) It should be possible to access the topmost element in a stack without removing it.

It must be possible to make a stack empty.

A function is needed that will set the values in the array to a value that signifies it is clear and set the top of the stack to the first index of the array

It must be possible to add (‘push’) an element to the top of a stack.

a function is needed that will add an element to the array. The position of the next element will need to be marked. What if we reach the end of the array?

It must be possible to remove (‘pop’) the topmost element from a stack.

A function is required that will allow the removal of an element. The top of the stack will need to be re-set to the next position. What happens when you reach the end of the stack?

It must be possible to test whether a stack is empty.

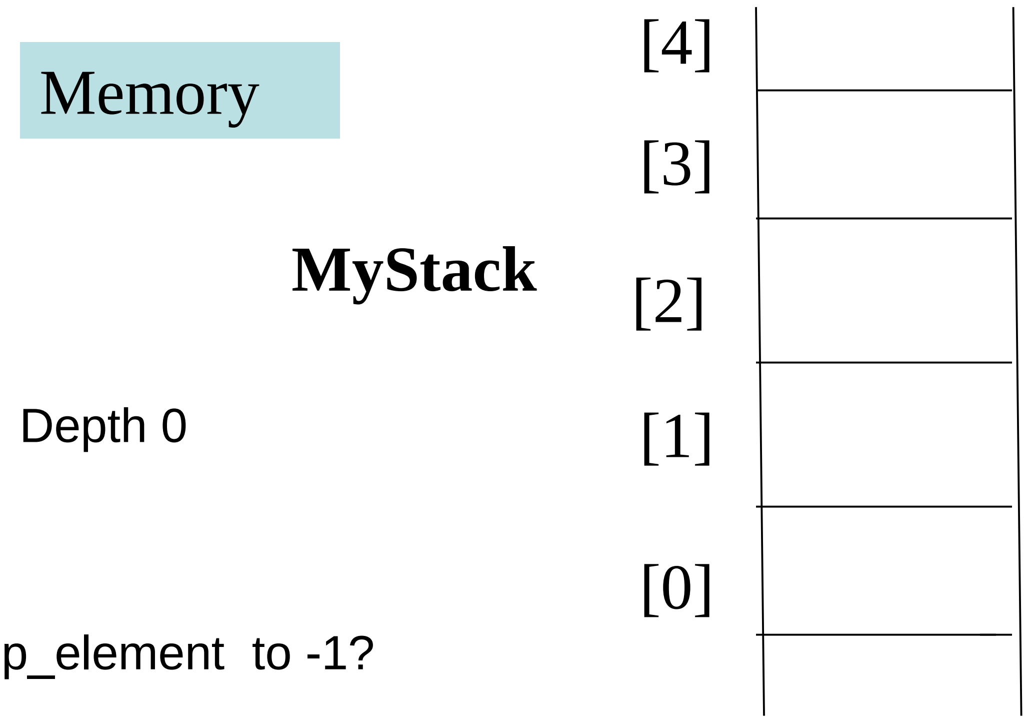
A function is required that will test if the stack is empty – a counter will be required that will keep track of the number of elements in the array.

It should be possible to access the topmost element in a stack without removing it.

A function is required that will read the top element value.

- The stack: visualisation
- 1. create the stack

```
int []MyStack = new int[5];
```



Top element represents the element position at the top of the stack

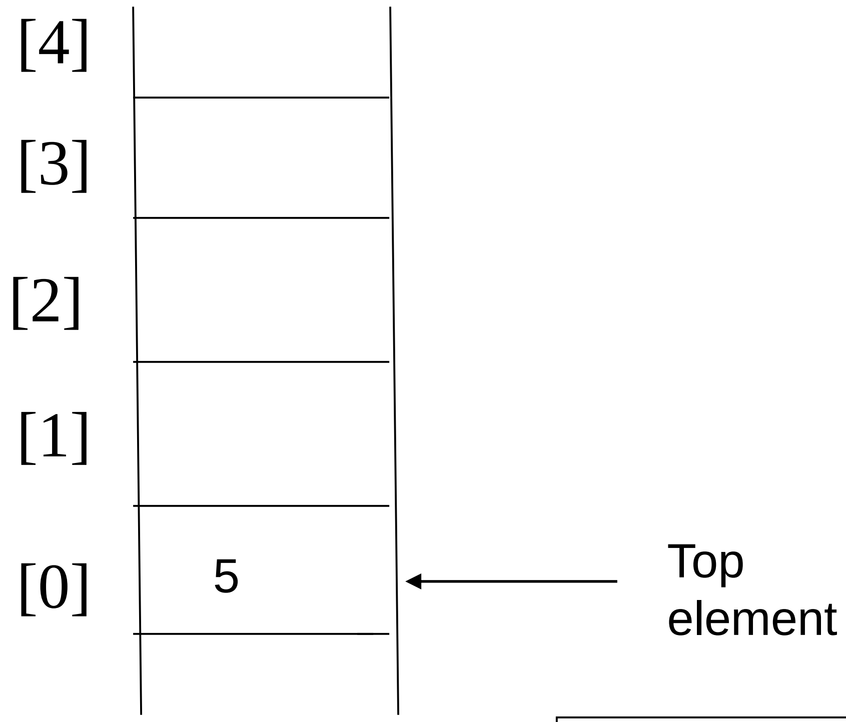
Depth is the position that the next element will be added to. It is also the number of the elements in the array.

initialise top_element to -1?

Add/push an item on the stack

Memory

MyStack



Top_element 0

Depth 1

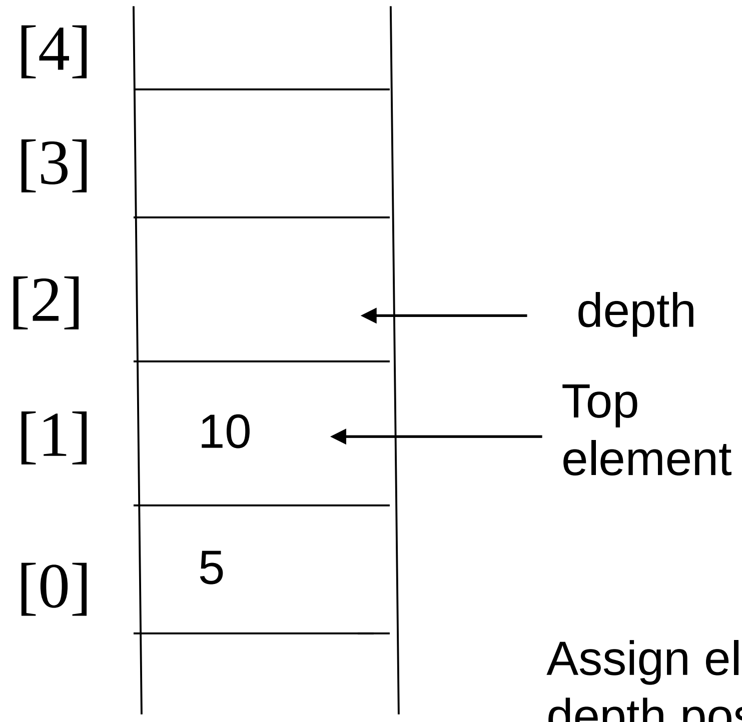
Assign element to depth position

Increment depth

Increment top element

Add/push an item on the stack

Memory



Assign element to depth position

Increment depth

Increment top element

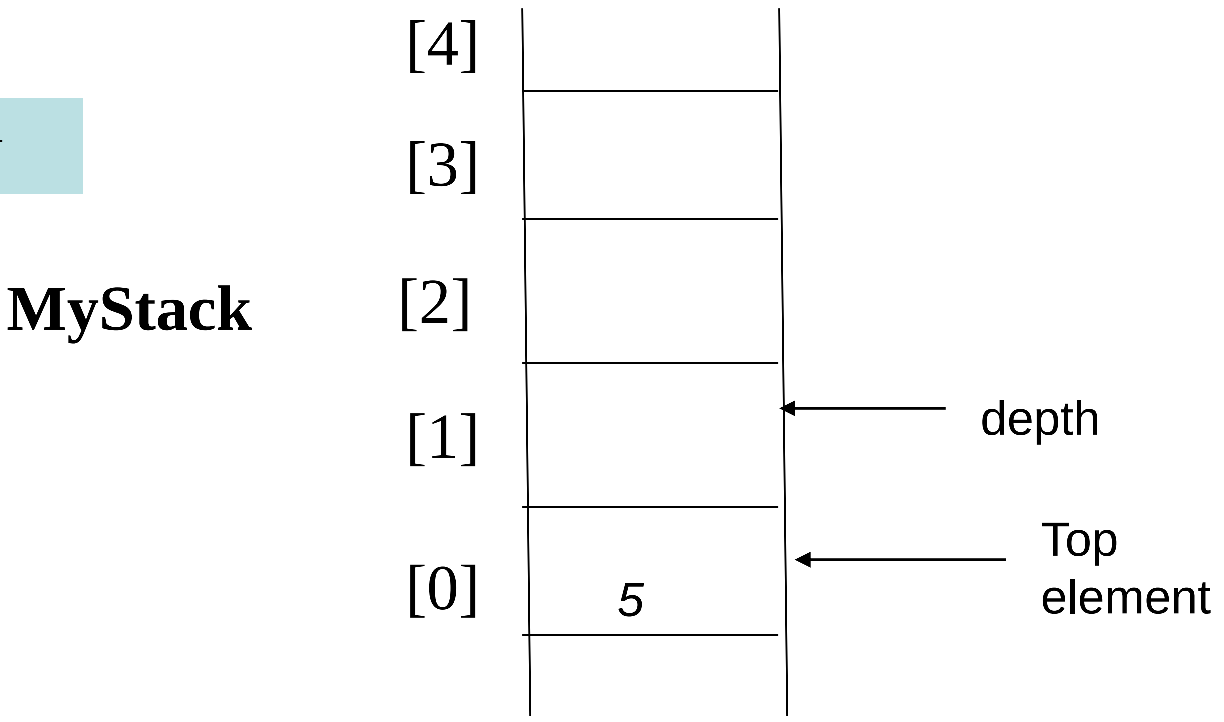
We now have a stack of:

top_element 1

Depth 2

pop an item from the stack

Memory



Depth 1

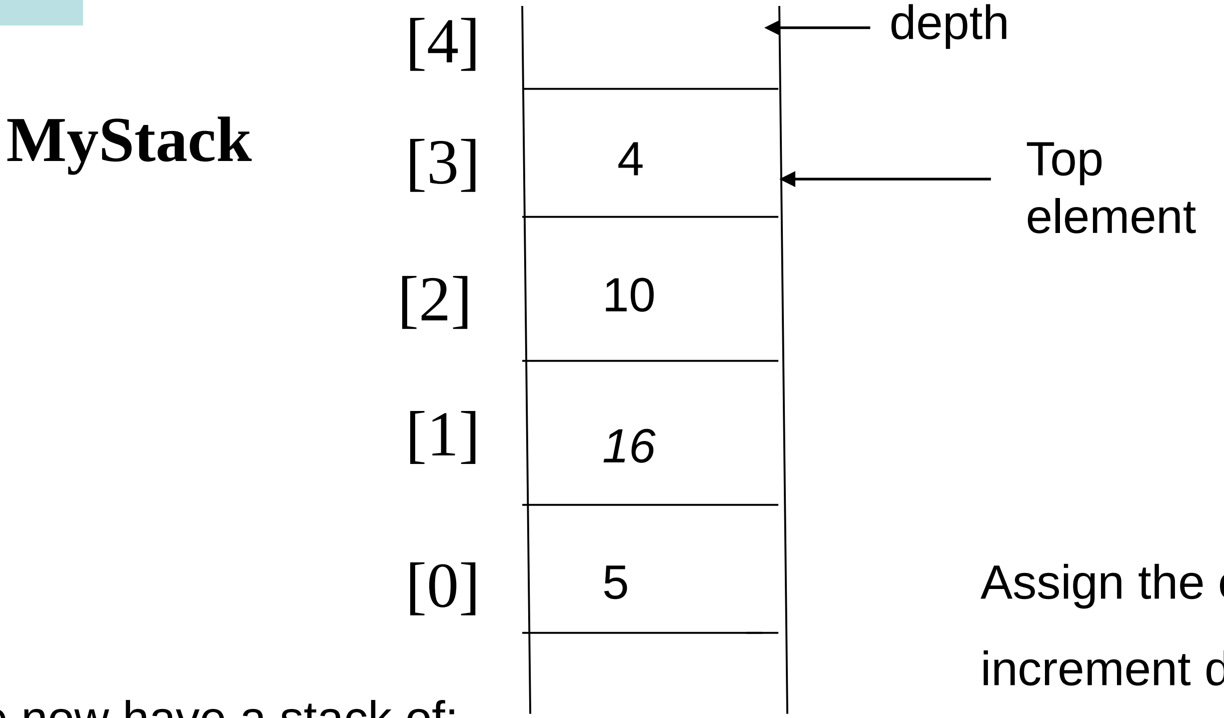
Top-element 0

decrement the top element

Decrement depth

push an item from the stack

Memory



Assign the element

increment depth

We now have a stack of:

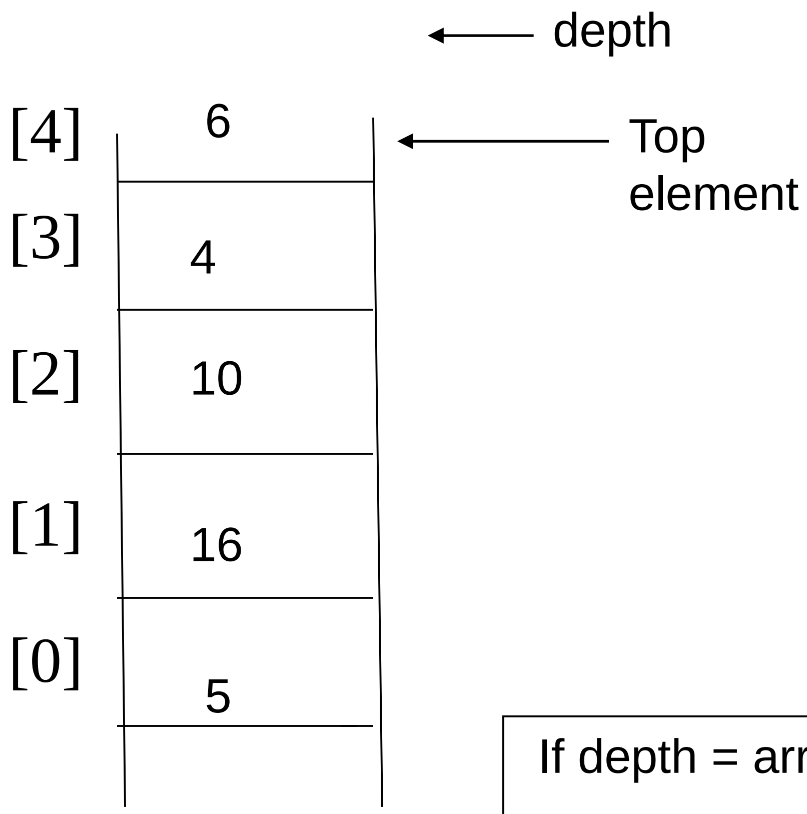
Top_element 3

Depth 4

push an item from the stack

Memory

MyStack



We now have a stack of:

Top_element 4

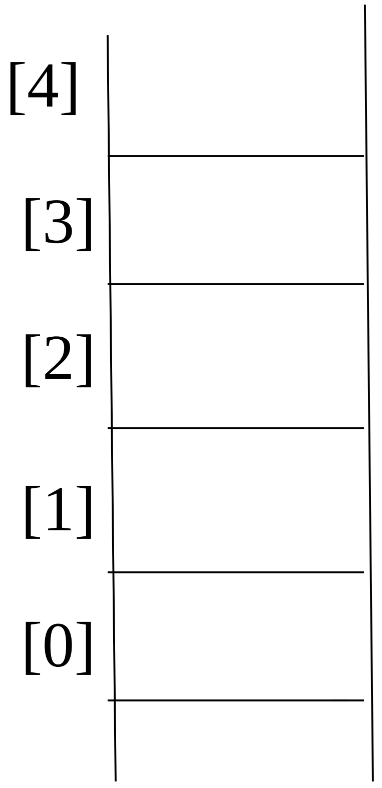
Depth 5

```
If depth = array length
        stack full
Else
        Assign the element
        increment depth
```

Pop from an empty stack

Memory

MyStack



Depth 0

```
If stack not empty
    decrement the top element
    Decrement depth
```

- Points to note:

- To read the top most element without removing
  - If the array is not empty
    - Print the array[topelement]

- To test whether a stack is full:
  - Size will be the same as array.length

- To make a stack empty
  - Set depth to 0
    Set Topelement to -1?

- To test whether a stack is empty:
    - Depth will be 0

Stack ADT: contract (1)

- Possible contract, expressed as a Java interface:

```
public interface Stack {
    // Each Stack object is a stack whose elements are objects.
    //////////////// Accessors ////////////////
    bool isEmpty ();
    // Return true if and only if this stack is empty.
    int getTop ();
    // Return the element at the top of this stack.
```

Stack ADT: contract (2)

- Possible contract (continued):

```
    ///////////////// Transformers /////////////////
    void clear ();
    // Make this stack empty.
    void addTop (int elem);
    // Add elem as the top element of this stack.
    int removeLast ();
    // Remove and return the element at the top of this
stack.
}
```

Implementation using arrays (2)

- Java implementation:

```
public class Stack {
    private int[] elems;
    private int depth;
    ///////////////// Constructor
/////////////////
    public ArrayStack (int maxDepth) {
        elems = new int[maxDepth];
        depth = 0;
    }
```

Implementation using arrays

```
    ////////////////
public boolean isEmpty () {
      return (depth == 0);
    }
    public int getTop () {
      if (depth == 0)
          system.out.println (“stack
                                  empty”);
      else
          return elems[depth-1];
    }
```

Implementation using arrays (4)

```
public void clear () {
      depth = 0;


}
public void addTop (int elem) {
  if (depth == elems.Length)
      system.out.println ("stack
                         full");
  else
    elems[depth++] = elem;
}
```

```
  public int removeLast () {
    if (depth == 0)
      Console.Writeln ("stack
                              empty");


    else{
        int topElem = elems[--depth];
        return topElem;
  }
}
```

In JavaScript:

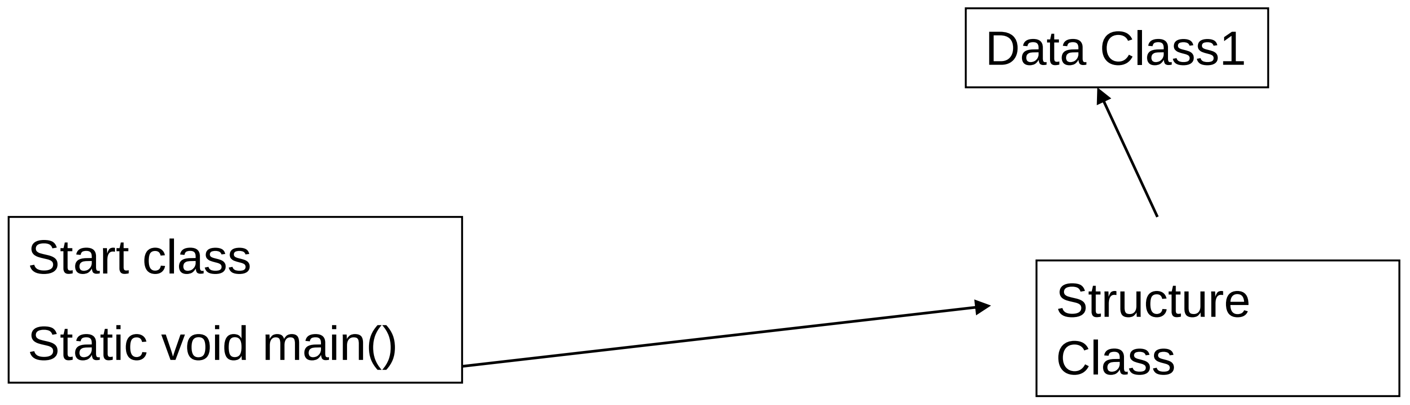
```
Stack.prototype.push = function(data) {
        var size = this._size++;
        this._storage[size] = data;
};
```

```
Stack.prototype.pop = function() {
        var size = this._size,
                var deletedData;

        if (size) {
                deletedData = this._storage[size];

                //  delete this.storage[size];
                this._size--;

                return deletedData;
        }
};
```

Stacks and classes

Example structure

Structure



```
class Car
{
    String Make;
    String Model;
    int Year;
}
```

Stack class

```
public class Stack
{

    int depth;

    int size;

   Car[] Garage;
```

The constructor:

```
Stack(int x)
   {

         depth = 0;

         size = x;

         Garage = new Car[size];

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

                    Garage[i] = new Car();

   }
```

```
void Add_car()
      {

            if (depth == size)

                        System.out.println("The stack is full");

            else
            {

                        System.out.println("Enter Car's Make: ");

                        Garage[depth].Make = EasyIn.getString();

                        System.out.println("Enter Car's Model: ");

                        Garage[depth].Model = EasyIn.getString();

                        System.out.println("Enter Car's Serial Number: ");

                        Garage[depth].Year = EasyIn.getInt();

                        depth++;
                        }


                        }
```

```
void Get_Last_car()
    {

        if (depth == 0)

            System.out.println("The stack is empty.");

        else
        {

            System.out.println("The last car of the stack is: ");

            System.out.println("Make: " + Garage[depth - 1].Make);

            System.out.println("Model: " + Garage[depth - 1].Model);

            System.out.println("Year: " + Garage[depth - 1].Year);

        }

    }
```

```
public void Remove_car()
{

    if (depth == 0)

            System.out.println("The stack is empty.");

    else


            Garage[depth - 1] = null;

    depth--;

}
```

```
public void Clear()
{

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

            Garage[i] = null;

    depth = 0;

}
```

```
public class Program
{

    public static void main(String[] args)
    {
            insert code….




            }



    }
```

```
        {
        Stack MyStack = new Stack(2);

        boolean Continue = true;

        while (Continue)
        {
                System.out.println("1. Add a car: ");

                System.out.println("2. Display last car: ");

                System.out.println("3. Remove a car: ");

                System.out.println("4. Clear the stack: ");

                System.out.println("5. Exit");

                int x = EasyIn.getInt();
}
```

```
switch (x)
    {
case 1:
    {
            MyStack.Add_car();

            break;
    }

case 2:
    {

            MyStack.Get_Last_car();

            break;
    }
```

```
case 3:

{

MyStack.Remove_car();

          break;
                    }
case 4:

{

MyStack.Clear();

break;

}

default:

{

Continue = false;

break;    }          }
}
```

Searching The Array : Developing The Code

Description:

We need to get the item to be searched for from the user.

Each element in the array needs to be checked for a match.

If a match is found we return the position.

If it is found we will report that it is not in the array

```
public int  linearSearchUnsorted (int val)
{ int left = 0;
  int right = MyArray.Length - 1;
  for (int p = left; p <= right; p++)
   {
     if (val==MyArray[p])
          return p; // returns the position if found
   }
   return -1; // return -1 if not found

}
```

Insertion Algorithm

Points to note:

Index from 0 to array.length – 1

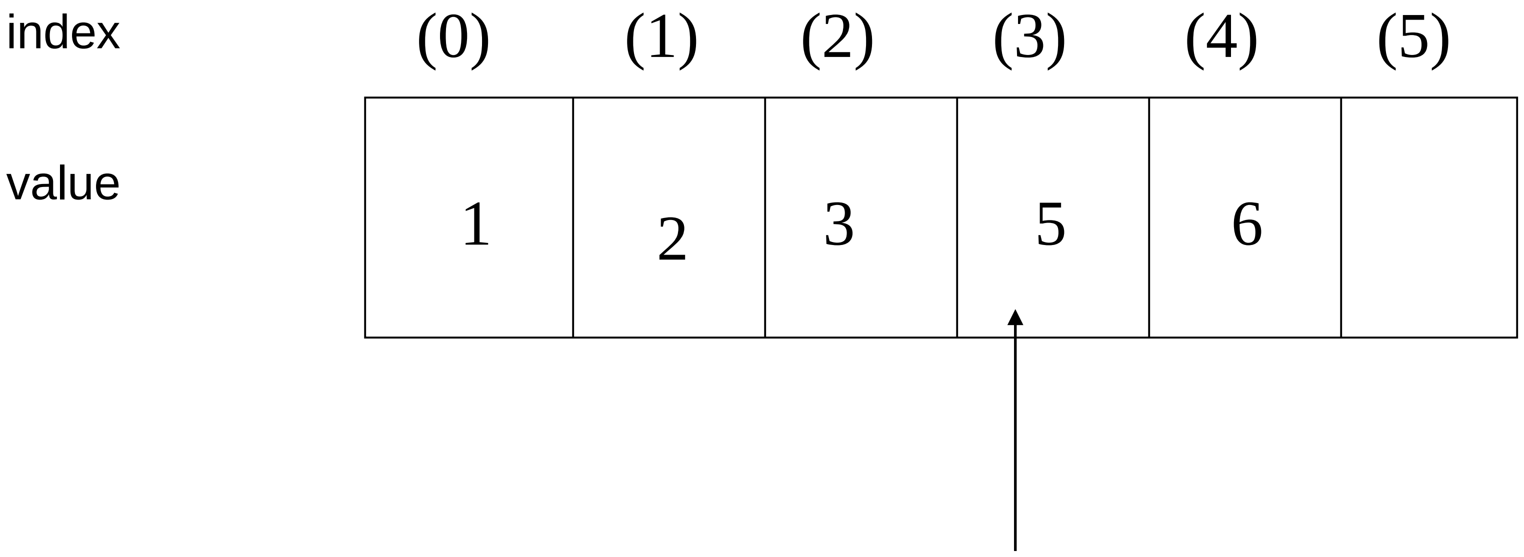
Number of items in the array = depth;

Array to be inserted into is called MyArray

Get item position (pos) to be inserted //use search algorithm or from the user

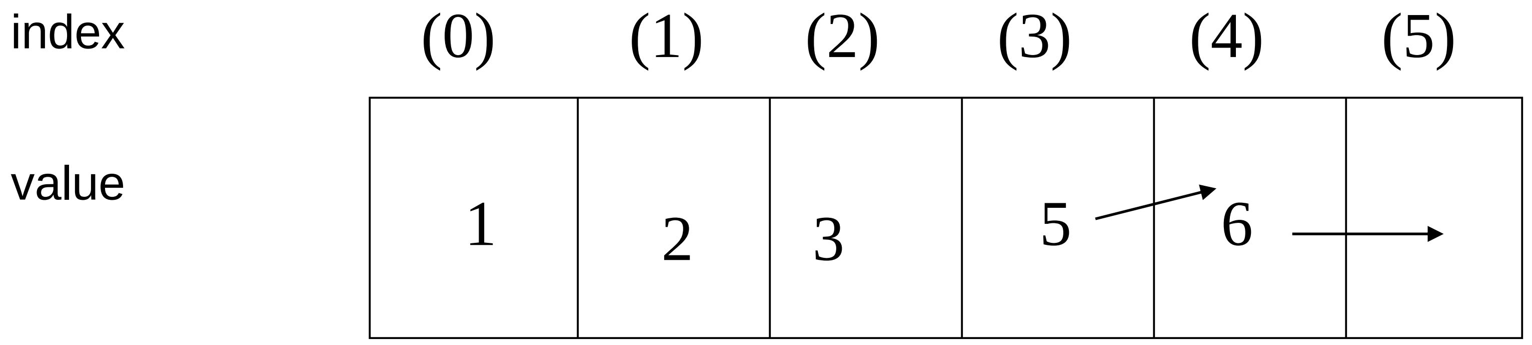
Example code:

```
for (int I = depth;  I>pos; I--)
    MyArray[I] = MyArray[I – 1];
    MyArray[pos] = new_item;
    depth++;
```



Example to insert to index position 3

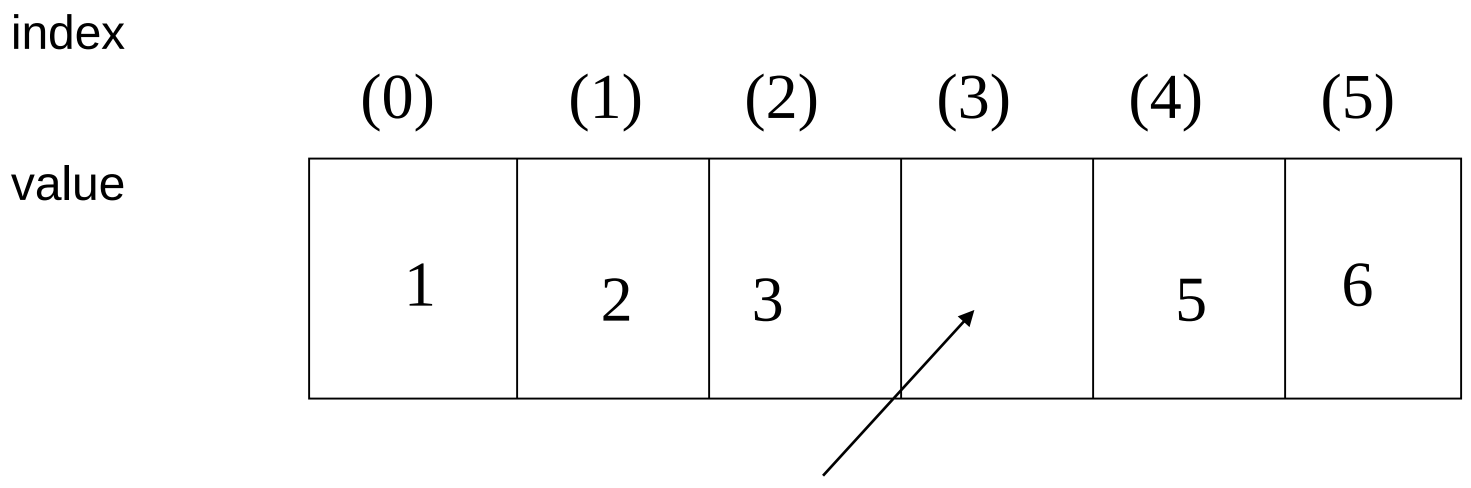
Depth = 5 // i.e. 5 items in the array



```
for (int I = depth;  I>pos; I--)
            MyArray[I] = MyArray[I – 1];
```

In this case depth is 5 and pos is 3 therefore we will be copying to position 5(from 4) and 4 from 3),

When I becomes 3 it is no longer > pos and the loop ends

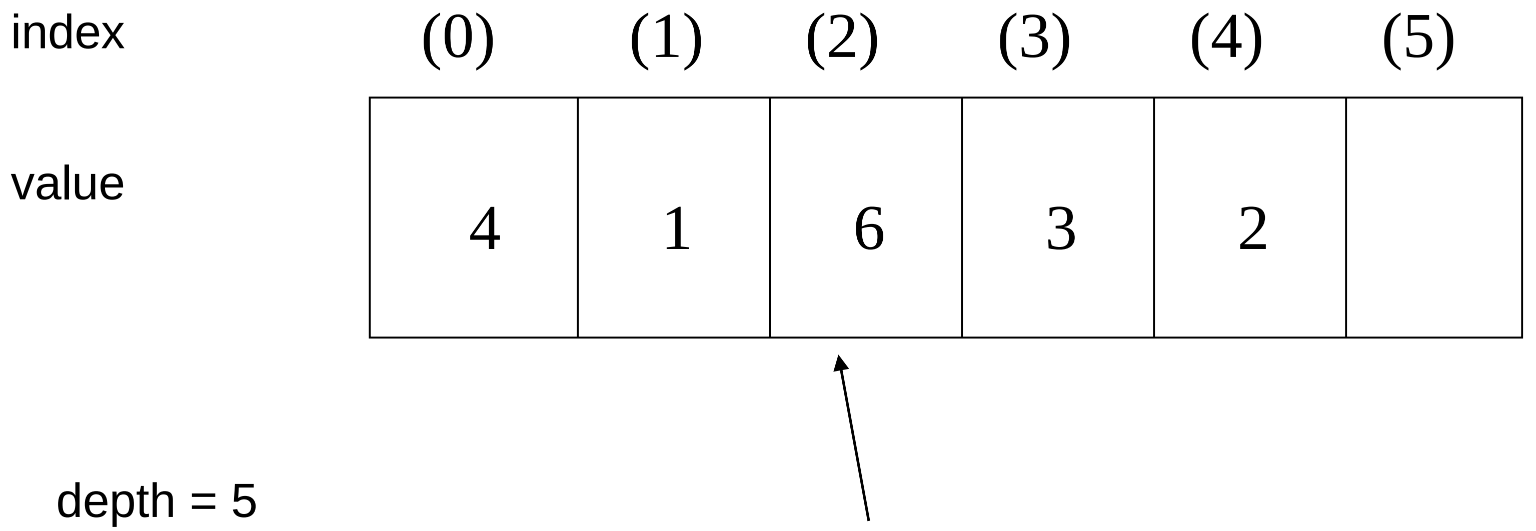


MyArray[pos] = new_item;
depth++;

depth will now take the value 6.

The insertion is complete.

Deletion



depth = 5

Pos = 2

Position to be deleted

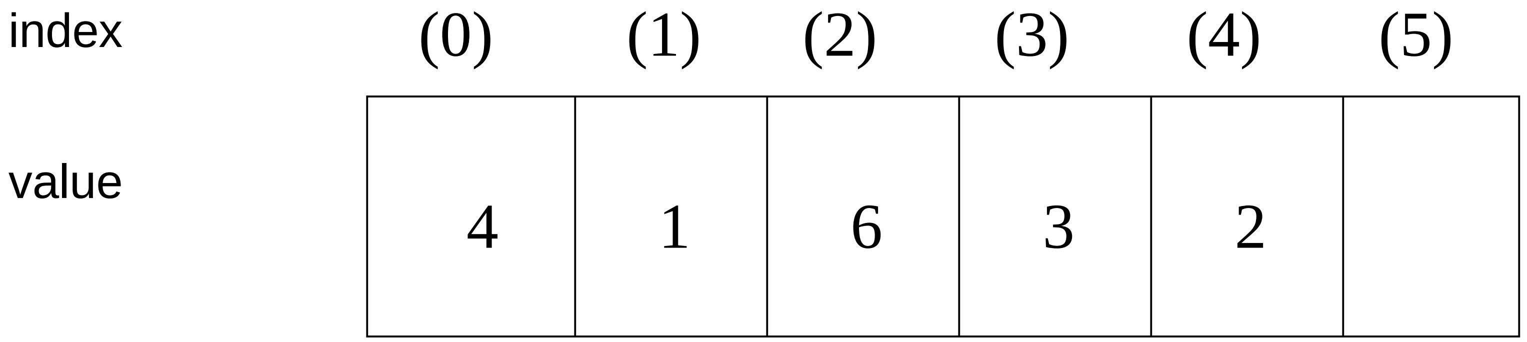
Myarray.length – 1 = 5

```
for (int I = pos;  I<MyArray.length - 1 ; I++)
        MyArray[I] = MyArray[I + 1];
                depth--;
```

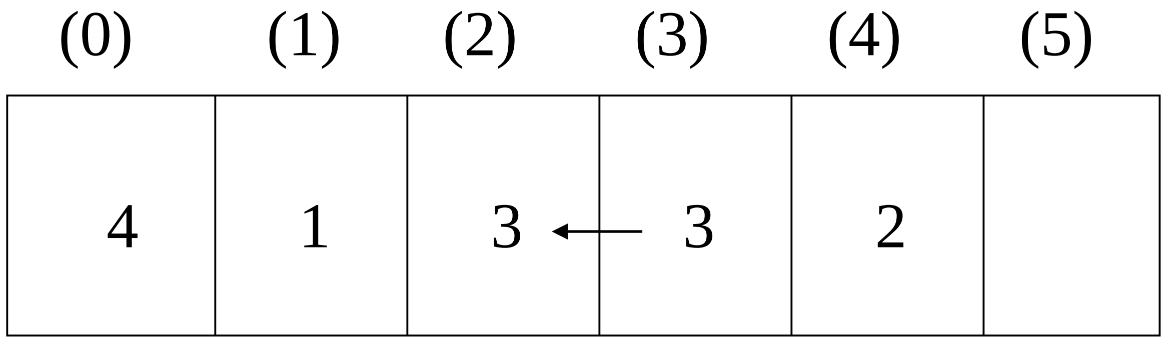
For index positions 2 to 4

This will be performed for each time of the loop

This will be performed after the loop has ended

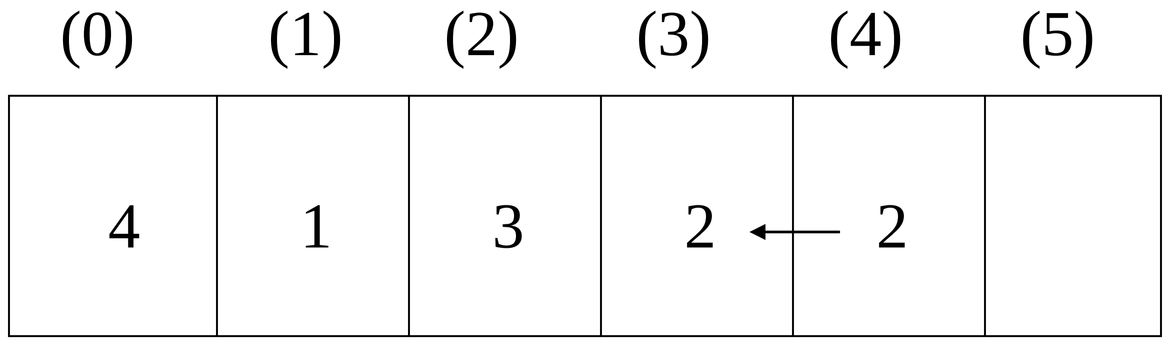


First time in loop i.e. for I = 2



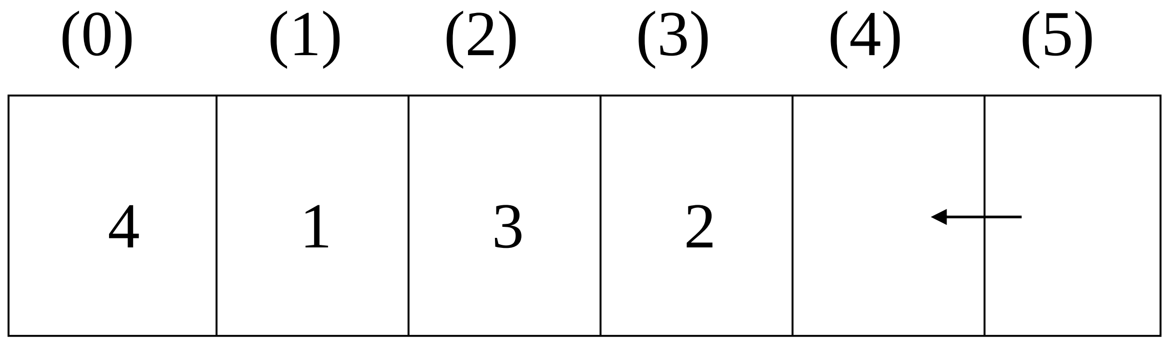
Copy index pos I +1 into pos I

First time in loop i.e. for I = 3



Copy index pos I +1 into pos I

First time in loop i.e. for I = 3



Copy index pos I +1 into pos I

The loop will end.

depth---; Depth takes the value 4

Deletion Algorithm

Points to not:

Index from 0 to array.length – 1

Number of items in the array = depth;

Array to be inserted into is called MyArray

Get item position (pos) to be deleted   //use search algorithm

///If position within bounds

```
if depth == 0
        console.writeln (The array is empty”);
else {
for (int I = pos;  I<MyArray.length - 1 ; I++)
    MyArray[I] = MyArray[I + 1];
         depth--;
}
```