
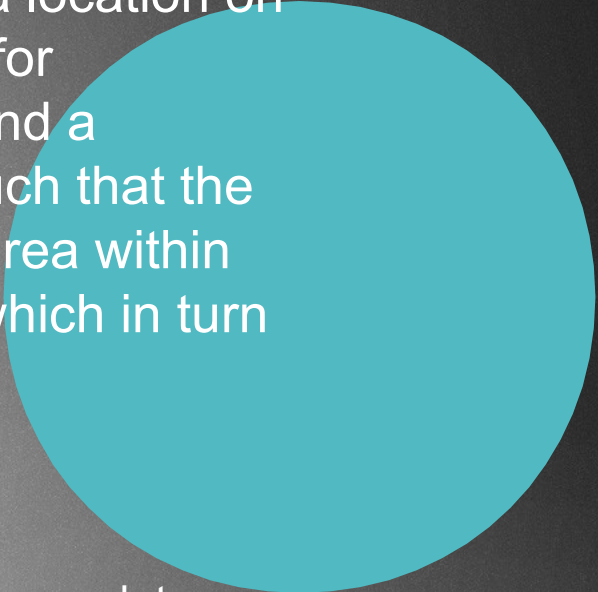


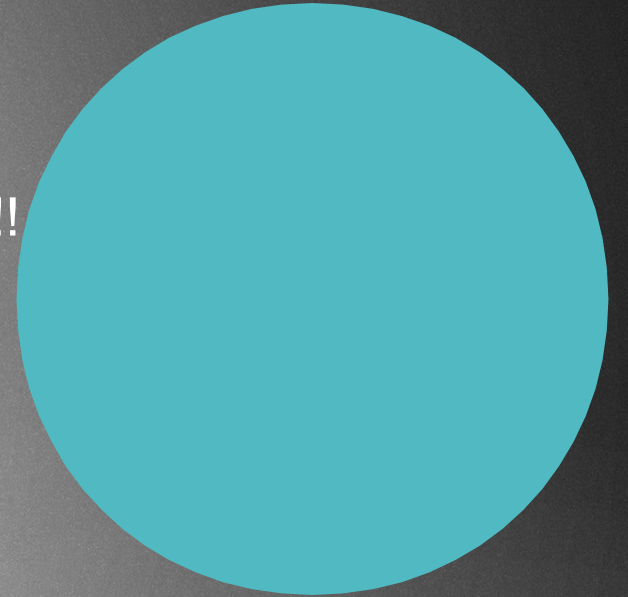
- 
- 
- ▶ Providing oversized data input to a program that does not check the length of input -> such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations

- ▶ „buffer overflow attack ”

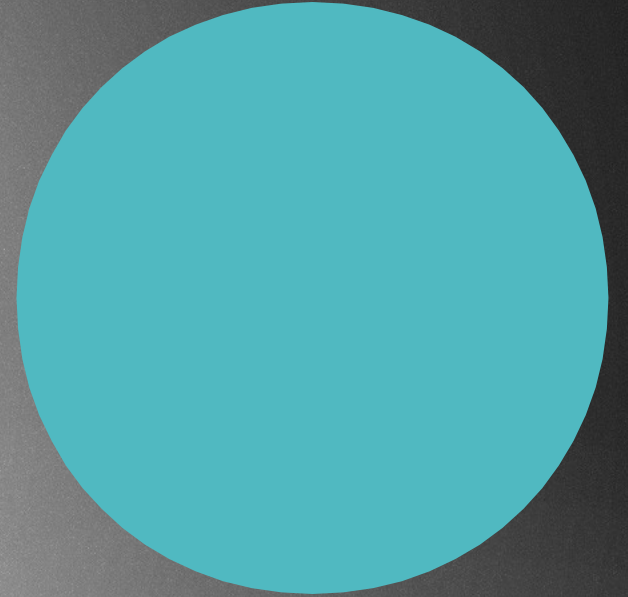
- ▶ If an oversized data item is moved to a stack location that is not large enough to contain it -> return information for procedure calls may be corrupted, causing the program to fail !!!

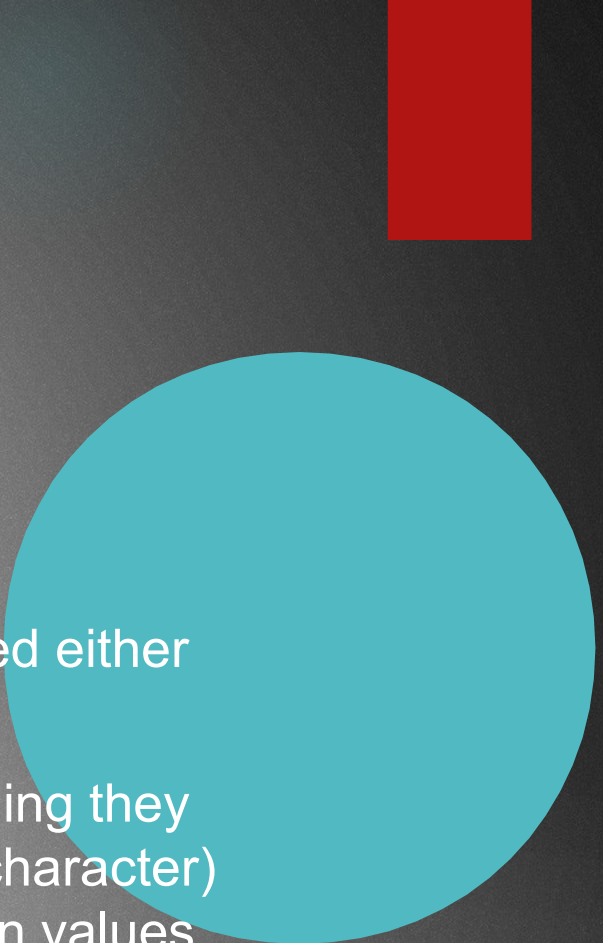
Recursion and stacks

- ▶ Usually: we can avoid using stacks with recursive algorithms
- ▶ BUT the operating system will use a stack in the background !!!

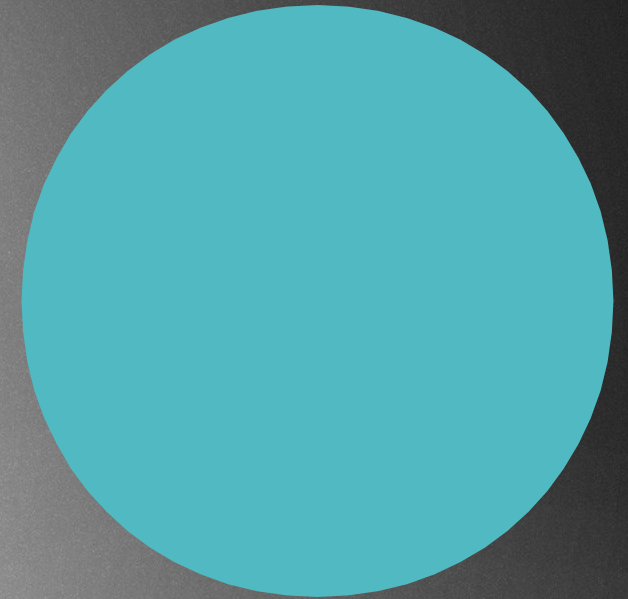
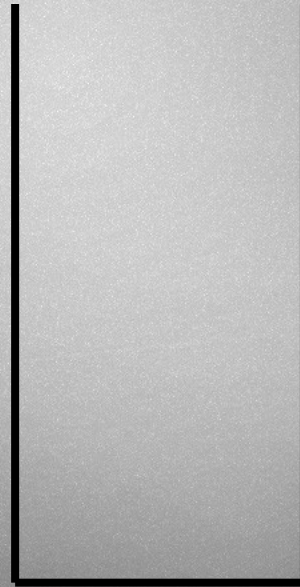


STACK



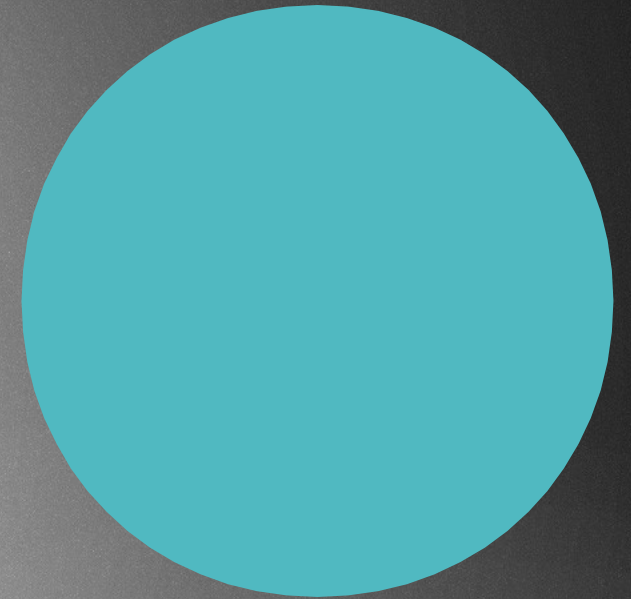
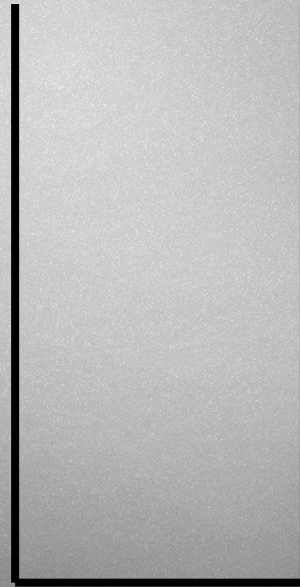
- 
- ▶ It is an abstract data type (interface)
 - ▶ Basic operations: pop(), push() and peek()
 - ▶ **LIFO** structure: last in first out
 - ▶ In most high level languages, a stack can be easily implemented either with arrays or linked lists
 - ▶ A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack

Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$



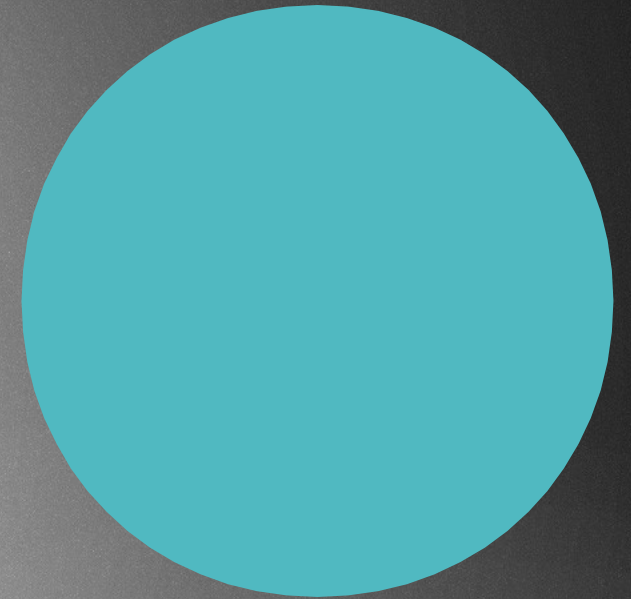
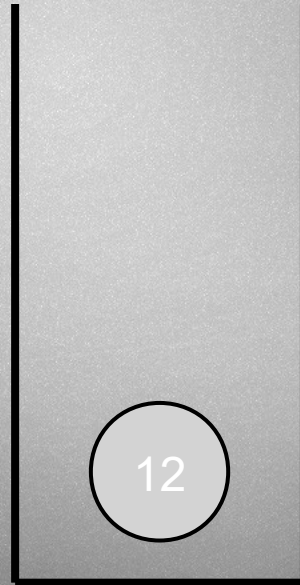
Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$

```
stack.push(12);
```



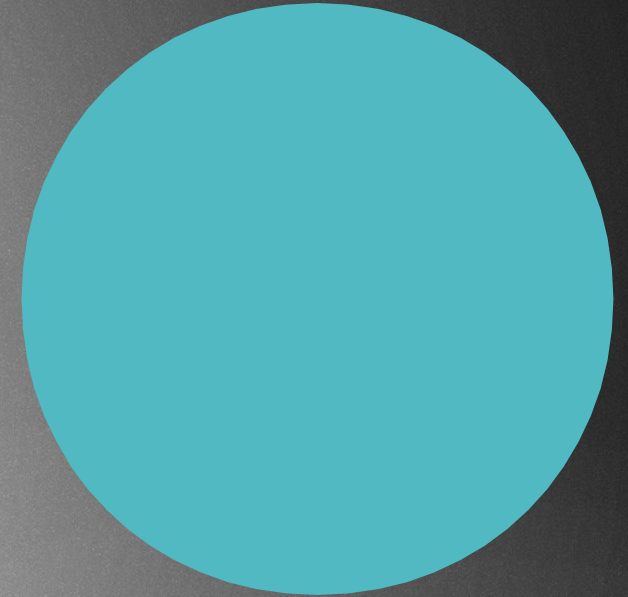
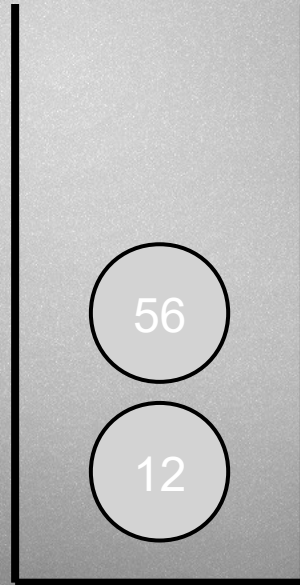
Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$

```
stack.push(56);
```



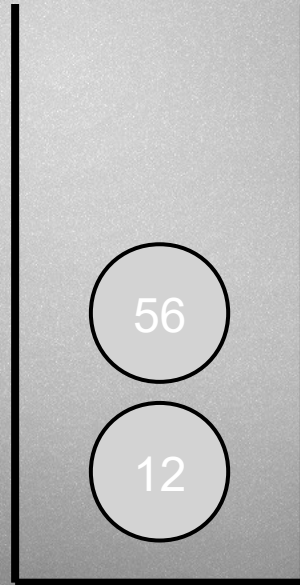
Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$

```
stack.push(56);
```



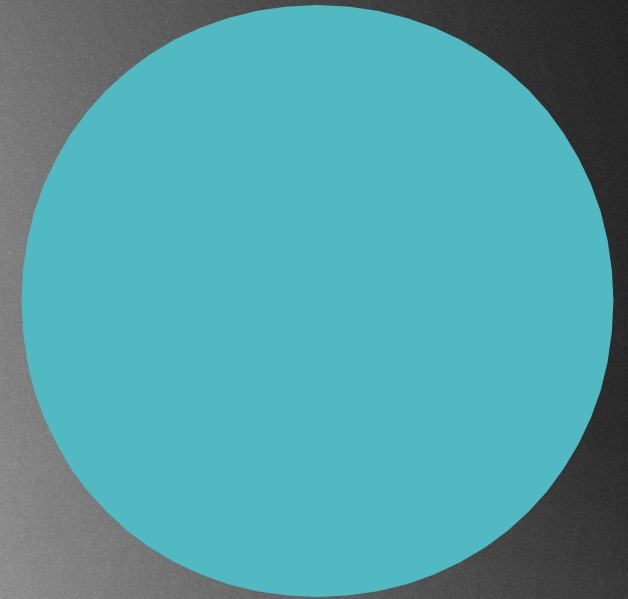
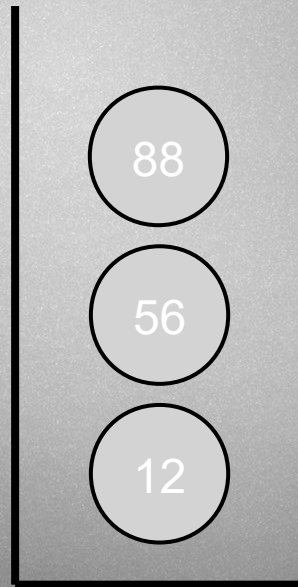
Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$

```
stack.push(88);
```

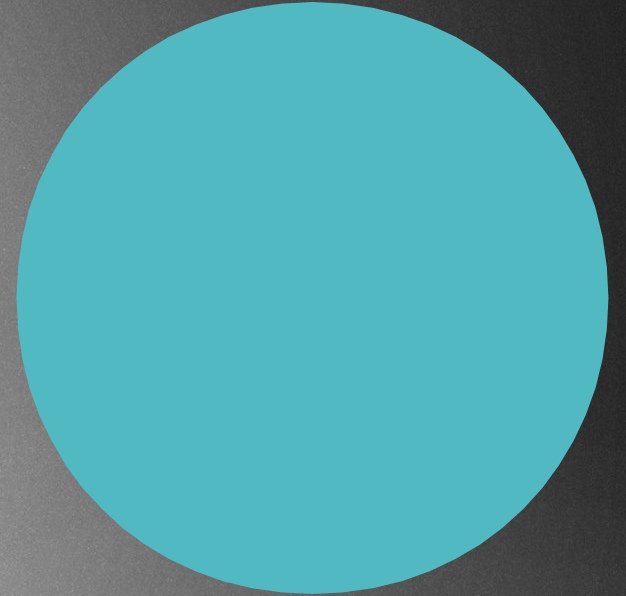
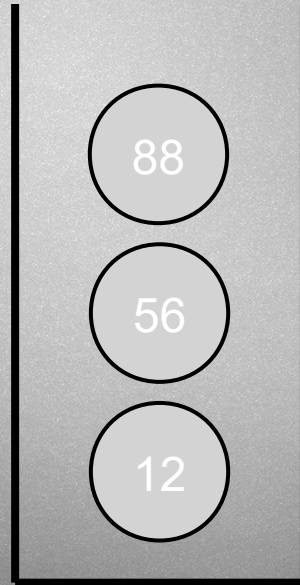


Push operation: put the given item to the top of the stack
Very simple operation, can be done in $O(1)$

```
stack.push(88);
```

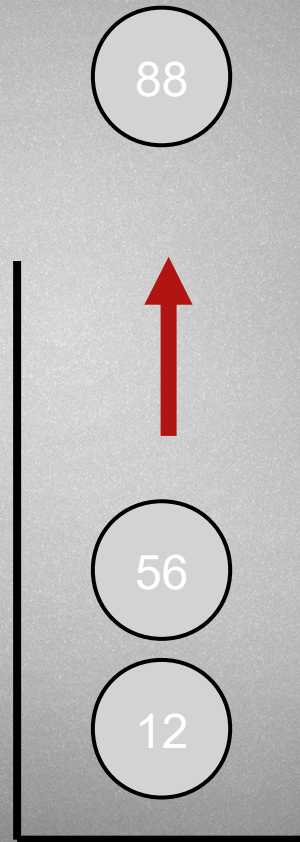


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in $O(1)$

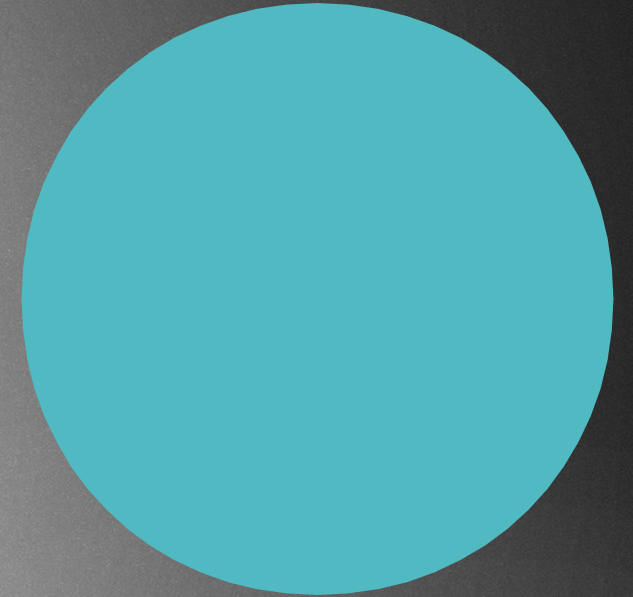
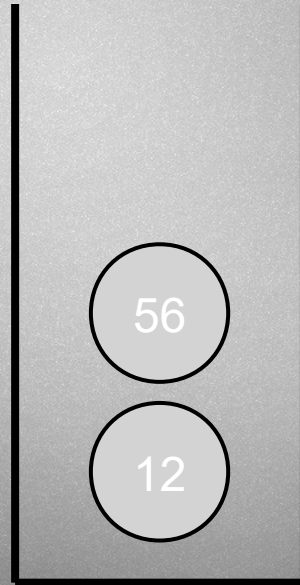


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in $O(1)$

`stack.pop();`

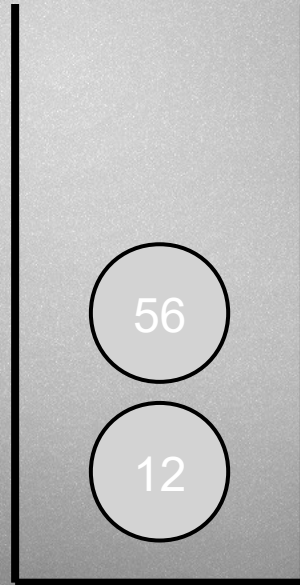


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in $O(1)$



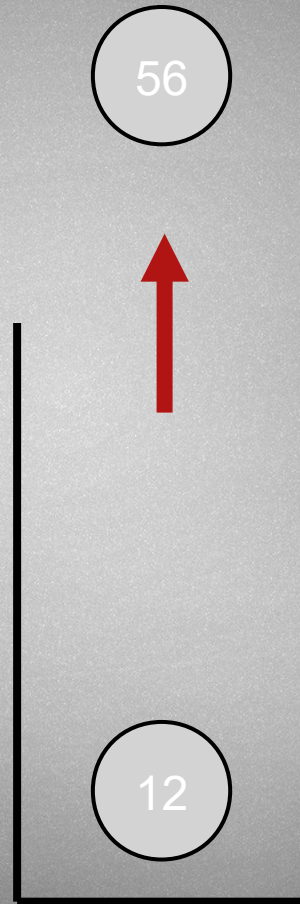
Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in $O(1)$

```
stack.pop();
```

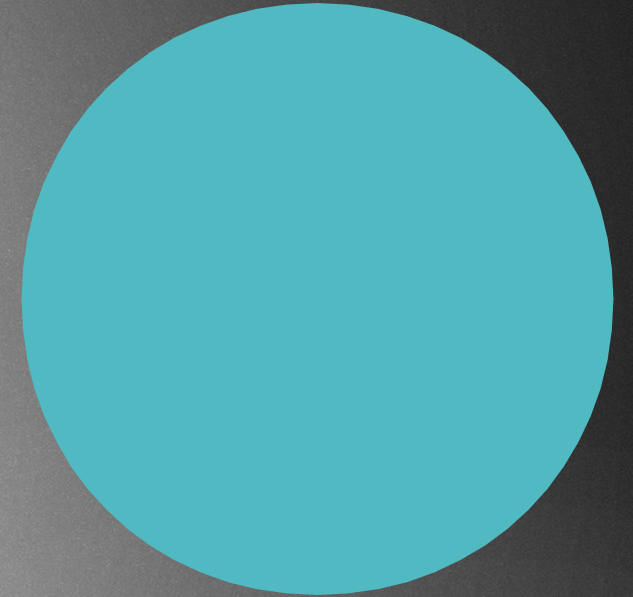
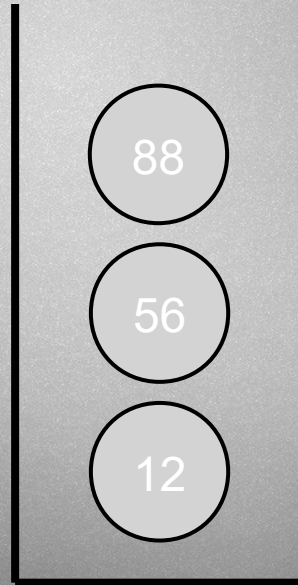


Pop operation: we take the last item we have inserted to the top of the stack (LIFO)
Very simple operation, can be done in $O(1)$

`stack.pop();`

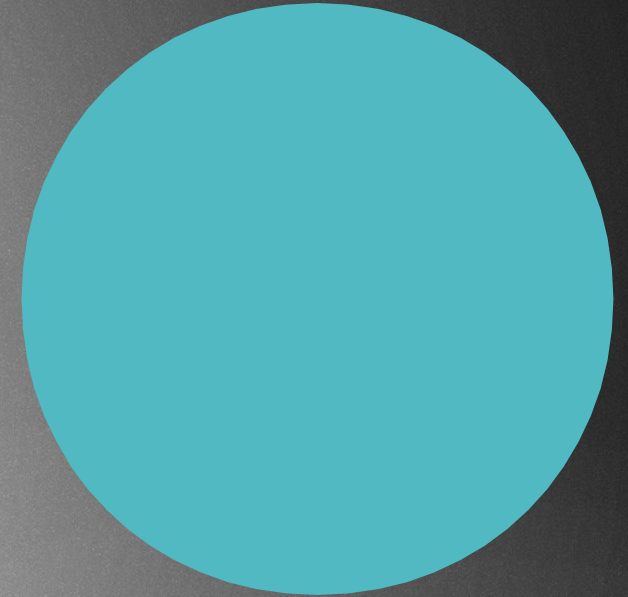
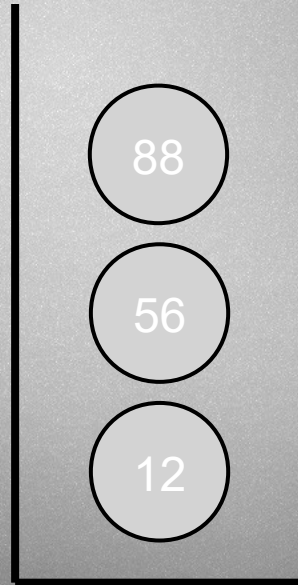


Peek operation: return the item from the top of the stack without removing it
Very simple operation, can be done in $O(1)$



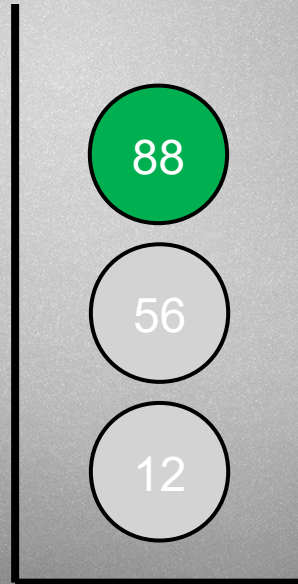
Peek operation: return the item from the top of the stack without removing it
Very simple operation, can be done in $O(1)$

```
stack.peek();
```



Peek operation: return the item from the top of the stack without removing it
Very simple operation, can be done in $O(1)$

```
stack.peek();
```



The peek() method will return 88
but the structure of the stack remains
the same !!!

Applications

- ▶ In stack-oriented programming languages
- ▶ Graph algorithms: depth-first search can be implemented with stacks (or with recursion)
- ▶ Finding Euler-cycles in a graph
- ▶ Finding strongly connected components in a graph



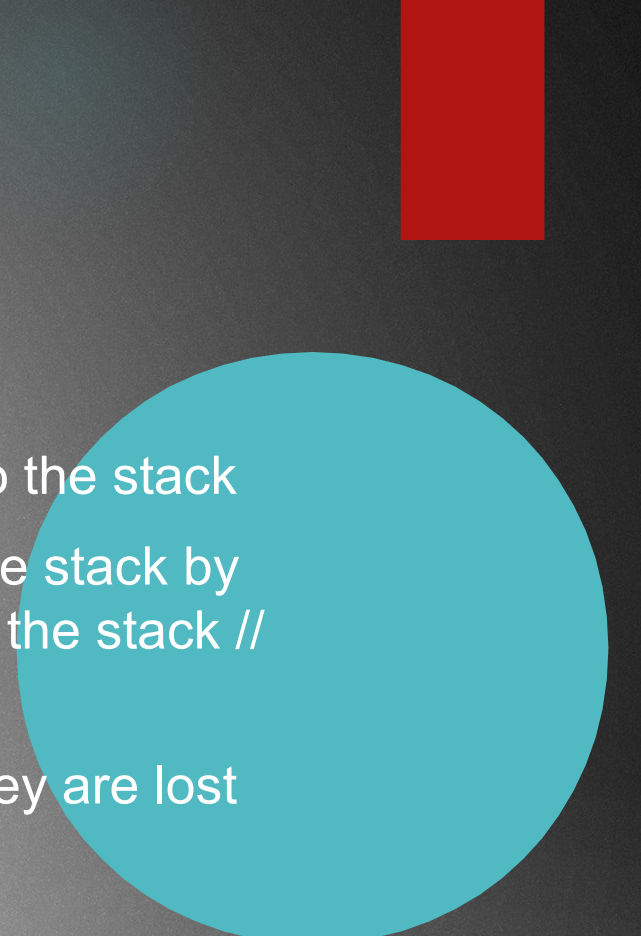
Queue

- ▶ It is an abstract data type (interface)
- ▶ Basic operations: enqueue() and dequeue()
- ▶ FIFO: first in first out
- ▶ It can be implemented with linked lists
- ▶ Applications: BFS



Stack „call stack”

- ▶ Most important application of stacks: stack memory
- ▶ It is a special region of the memory (in the RAM)
- ▶ A call stack is an abstract data type that stores information about the active subroutines / methods / functions of a computer program
- ▶ The details are normally hidden and automatic in high-level programming languages
- ▶ Why is it good?
- ▶ It keeps track of the point to which each active subroutine should return control when it finishes executing
- ▶ Stores temporary variables created by each function

- 
- ▶ Every time a function declares a new variable it is pushed onto the stack
 - ▶ Every time a function exits all of the variables - pushed onto the stack by that function - are freed → all of its variables are popped off of the stack // and lost forever !!!
 - ▶ Local variables: they are on the stack, after function returns they are lost
 - ▶ Stack memory is limited !!!

Heap memory

- ▶ The heap is a region of memory that is not managed automatically for you
- ▶ This is a large region of memory // unlike stack memory
- ▶ **C:** malloc() and calloc() function // with pointers
- ▶ **Java:** reference types and objects are on the heap
- ▶ We have to deallocate these memory chunks: because it is not managed automatically
- ▶ If not: memory leak !!!
- ▶ Slower because of the pointers

stack memory

limited in size

fast access

local variables

space is managed
efficiently by CPU

variables cannot
be resized

heap memory

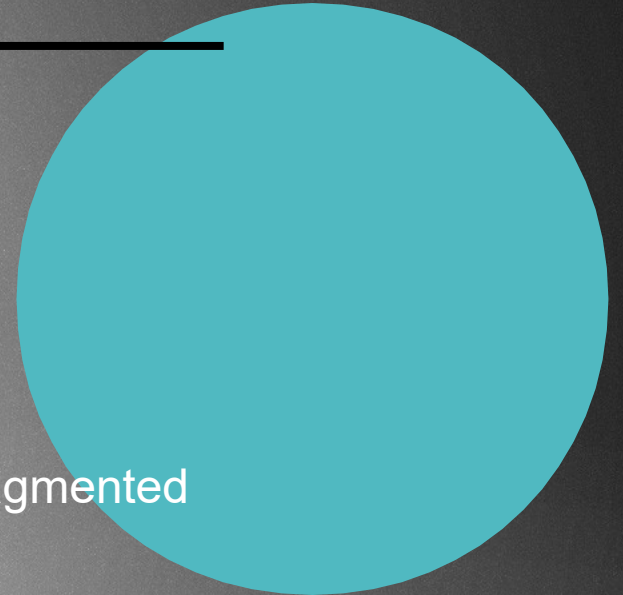
no size limits

slow access

objects

memory may be fragmented

variables can be resized
// realloc()



Stack and recursion

- ▶ There are several situations when recursive methods are quite handy
- ▶ For example: DFS, traversing a binary search tree, looking for an item in a linked list ...
- ▶ What's happening in the background?
- ▶ All the recursive algorithms can be transformed into a simple method with stacks
- ▶ **IMPORTANT:** if we use recursion, the OS will use stacks anyways !!!

Depth-first search

```
public void dfs(Vertex vertex) {  
  
    vertex.setVisited(true);  
    printf(vertex);  
  
    for(Vertex v : vertex.neighbours() ){  
        if( !v.isVisited() ){  
            dfs(v);  
        }  
    }  
}
```

recursion

```
public void dfs(Vertex vertex) {  
  
    Stack stack;  
    stack.push(vertex);  
  
    while( !stack.isEmpty() ){  
  
        actual = stack.pop();  
  
        for(Vertex v : actual .neighbours() ){  
            if( !v.isVisited() ){  
                v.setVisited(true);  
                stack.push(v);  
            }  
        }  
    }  
}
```

iterative approach with stack

Factorial: with recursion

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

This is the factorial function with Recursive implementation

$$n! = n * (n-1) * \dots * 2 * 1$$

For example: $4! = 4*3*2*1 = 24$

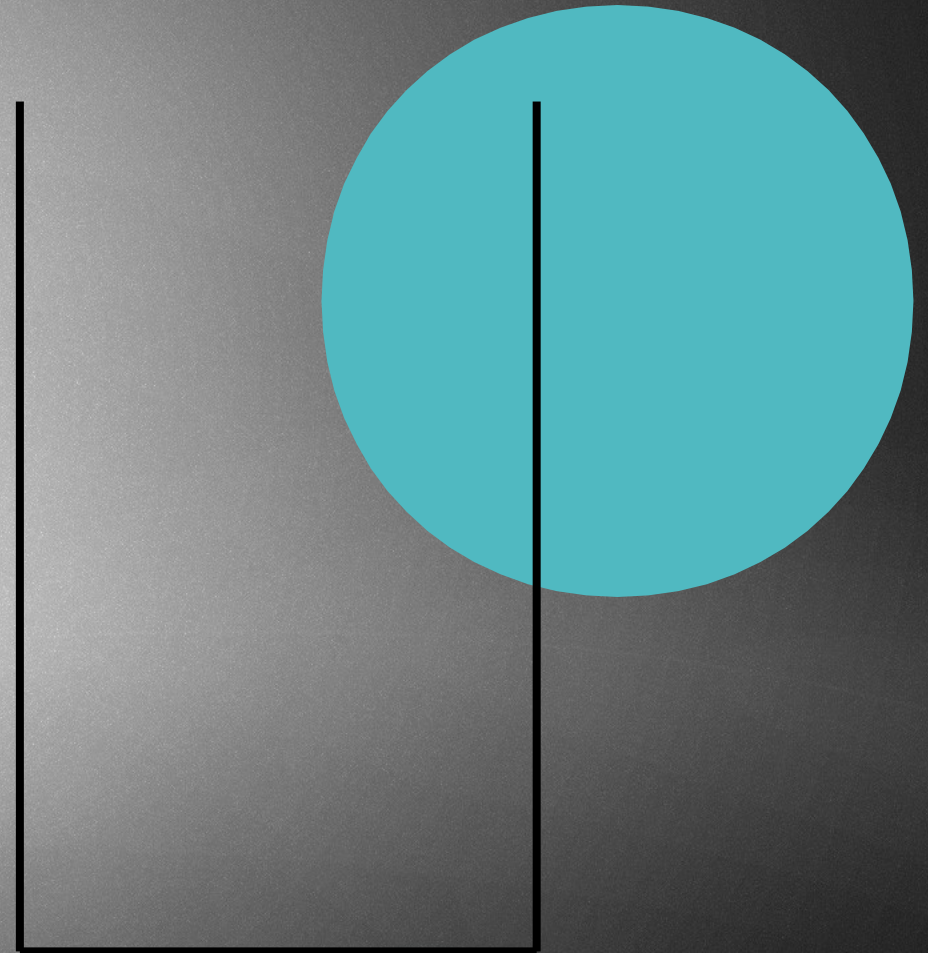
What does it all have to do with stacks? The recursive function calls are pushed onto the stack until we bump into the base case

- we keep backtracking: we know the base case so we know the subsolutions
- if there are too many function calls to be pushed onto the stack: the stack may get full ... no more space left
- stack overflow !!!

Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

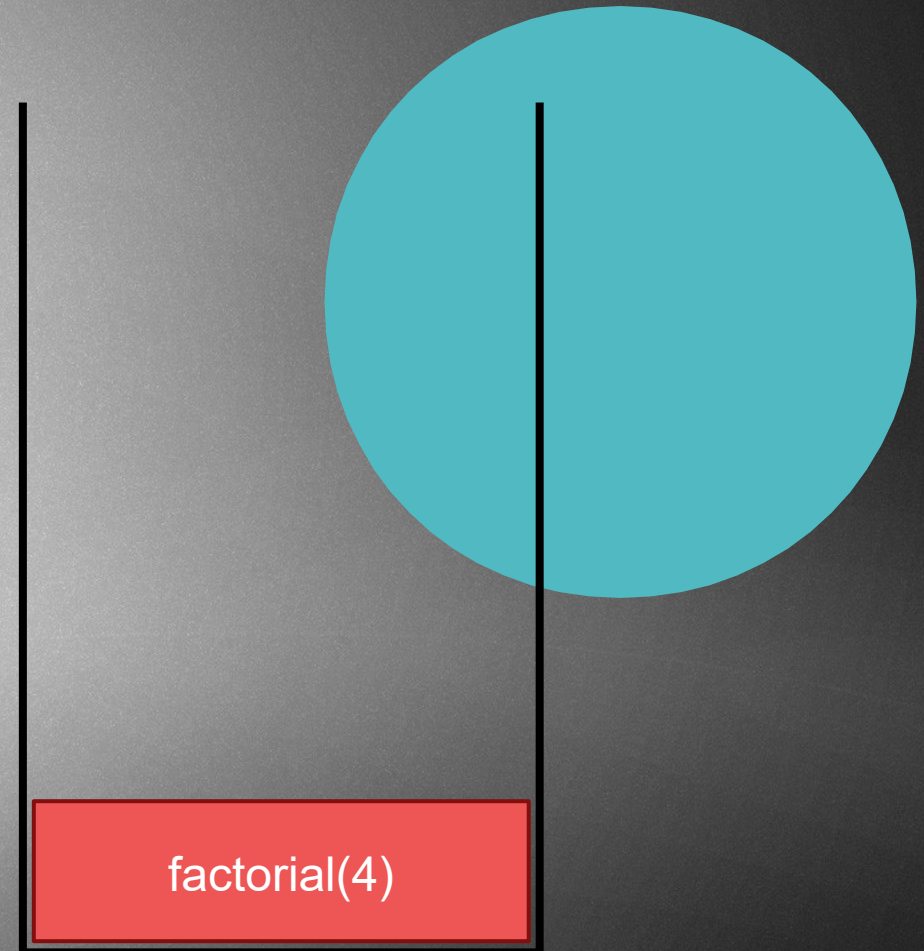
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

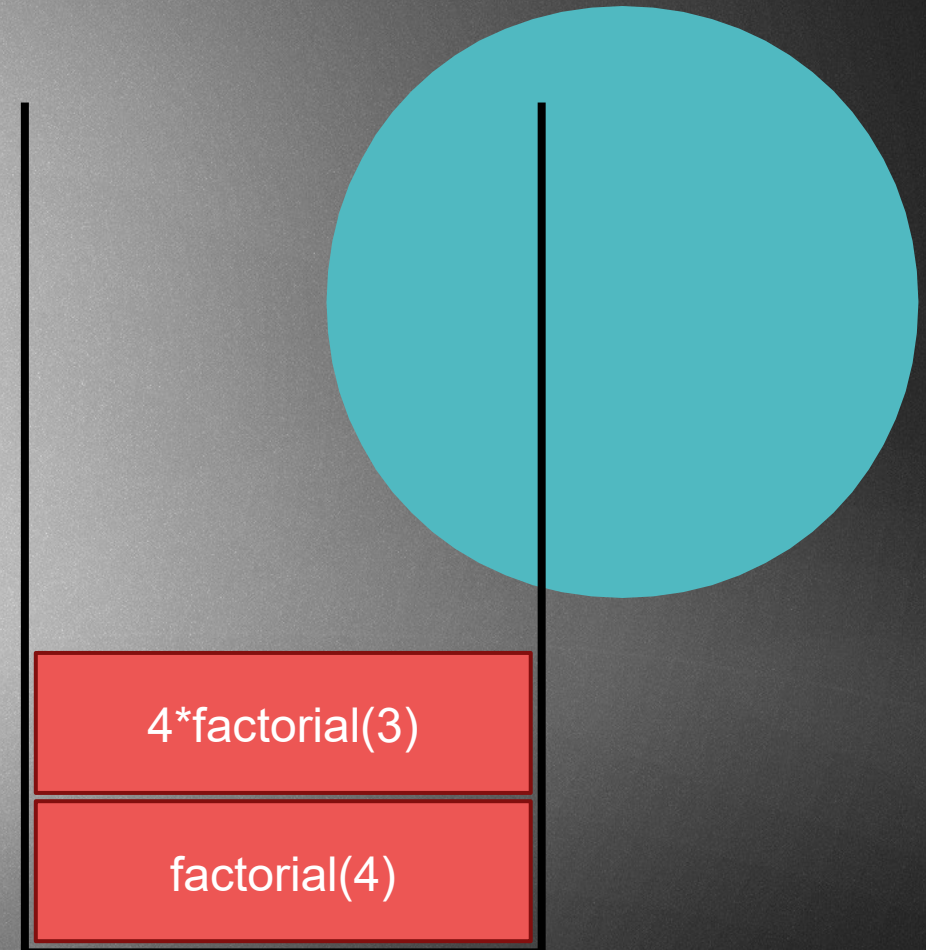
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

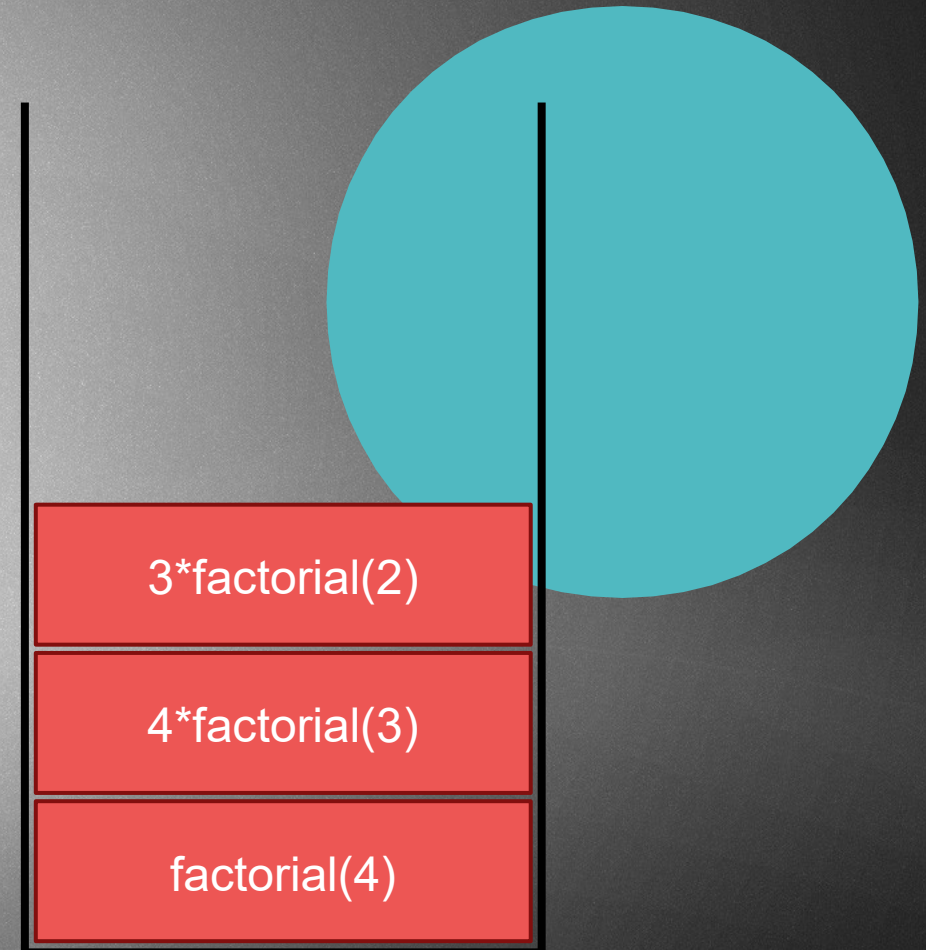
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

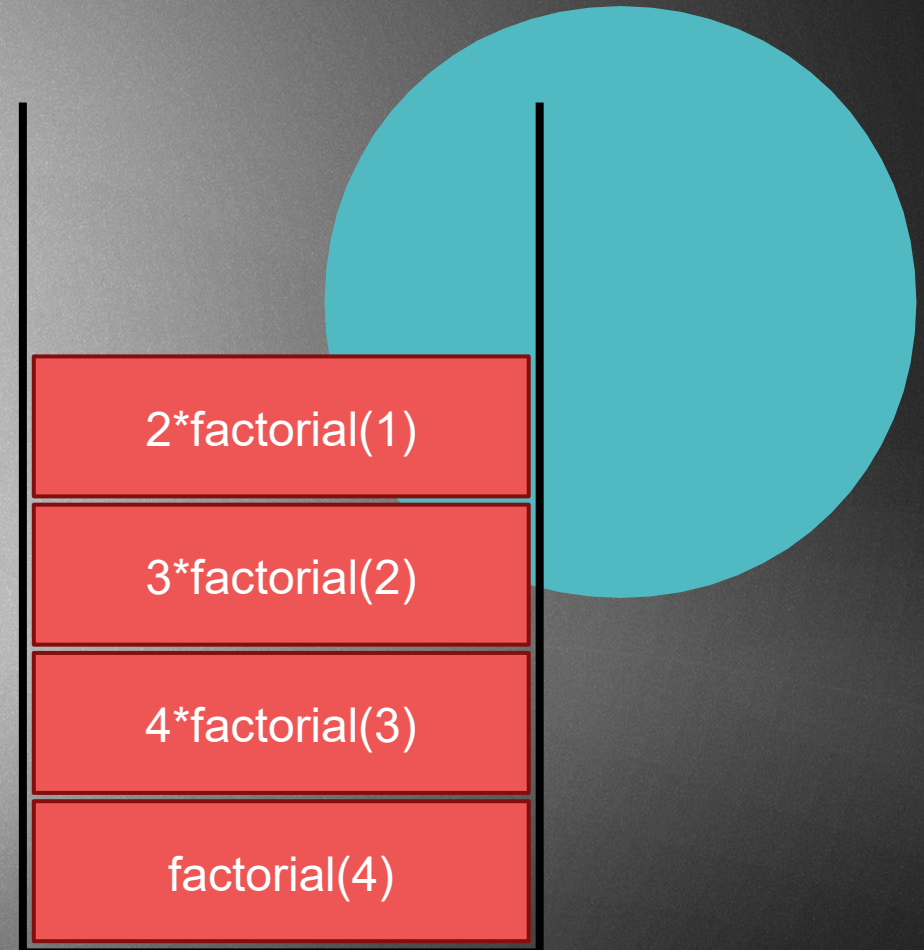
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

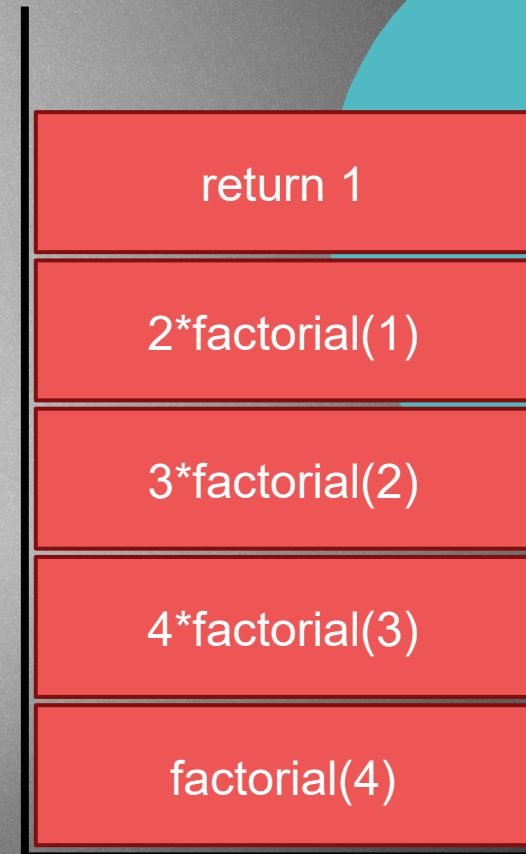
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

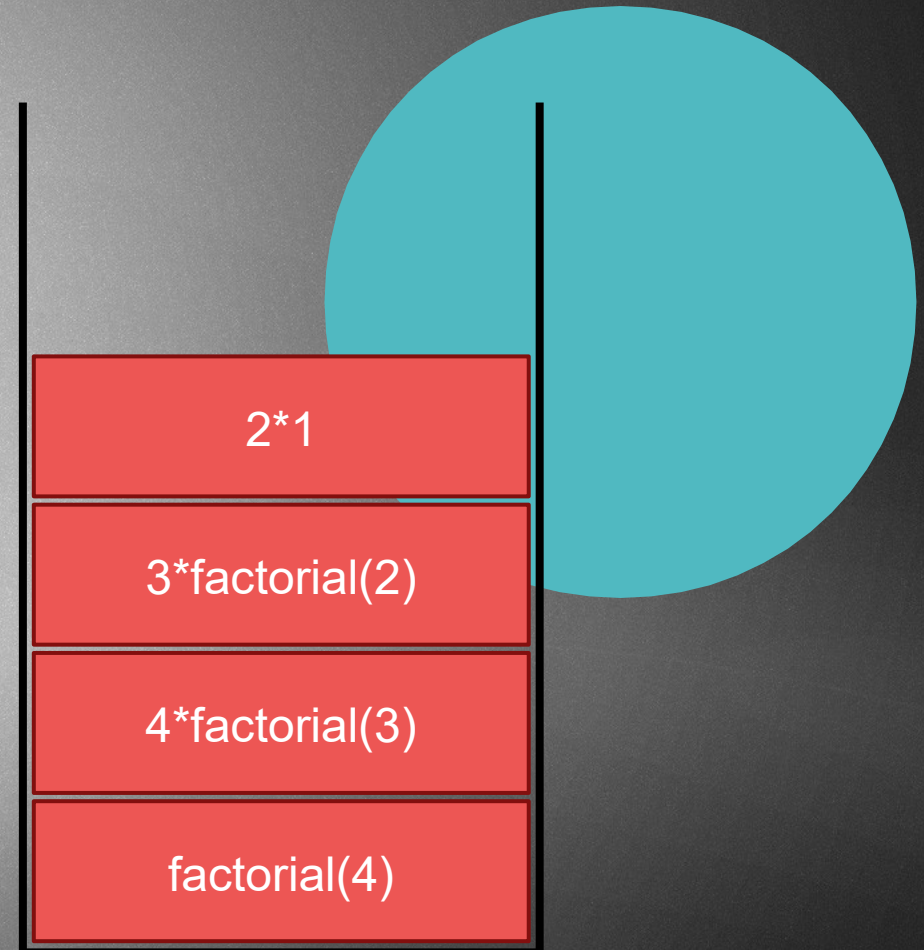
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

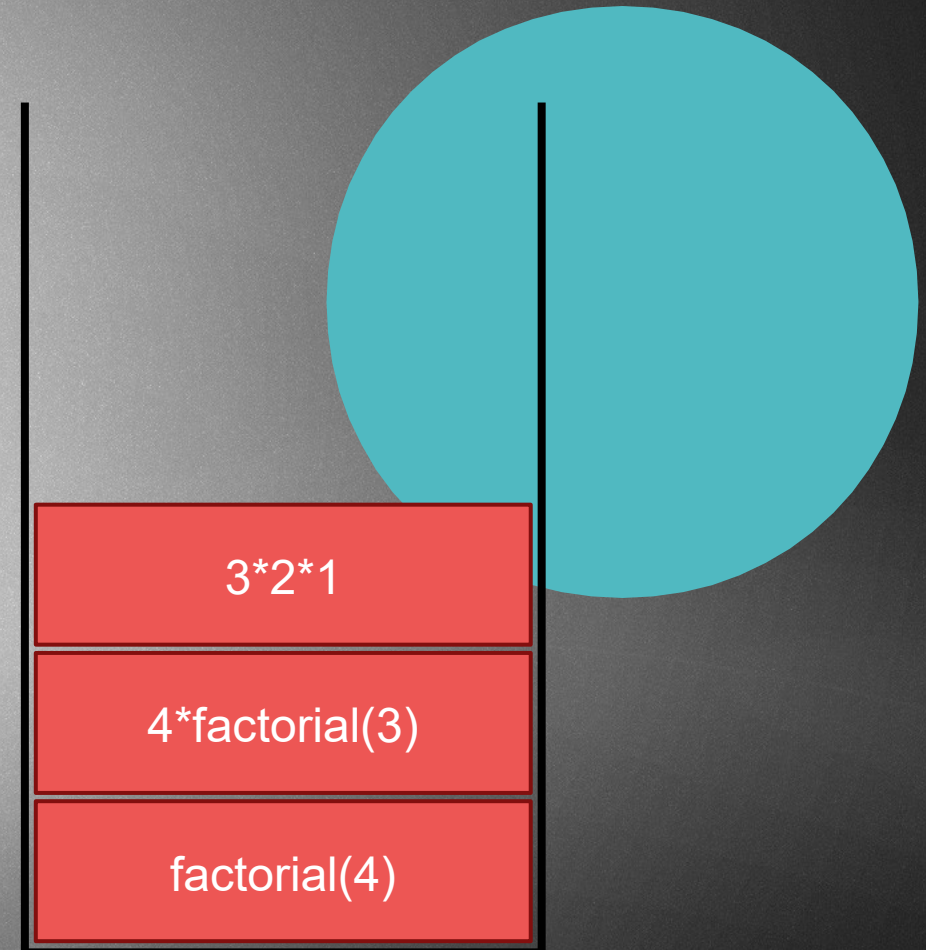
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

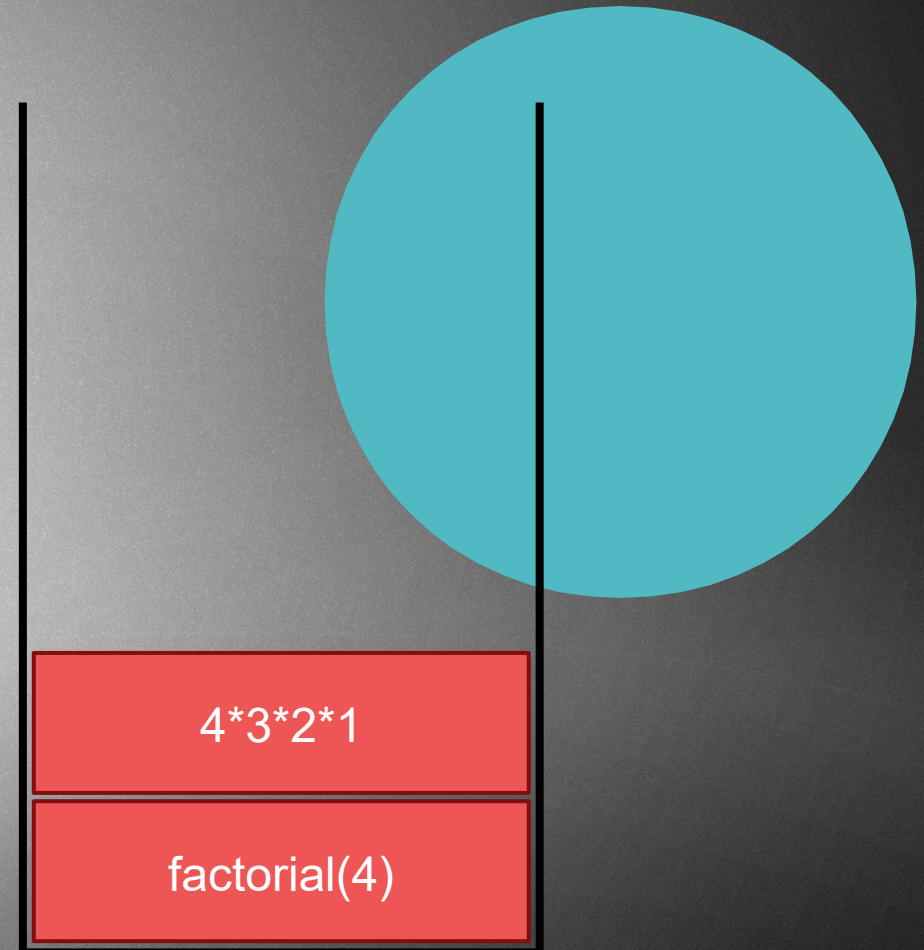
```
int result = factorial(4)
```



Factorial: factorial(4)

```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```

```
int result = factorial(4)
```



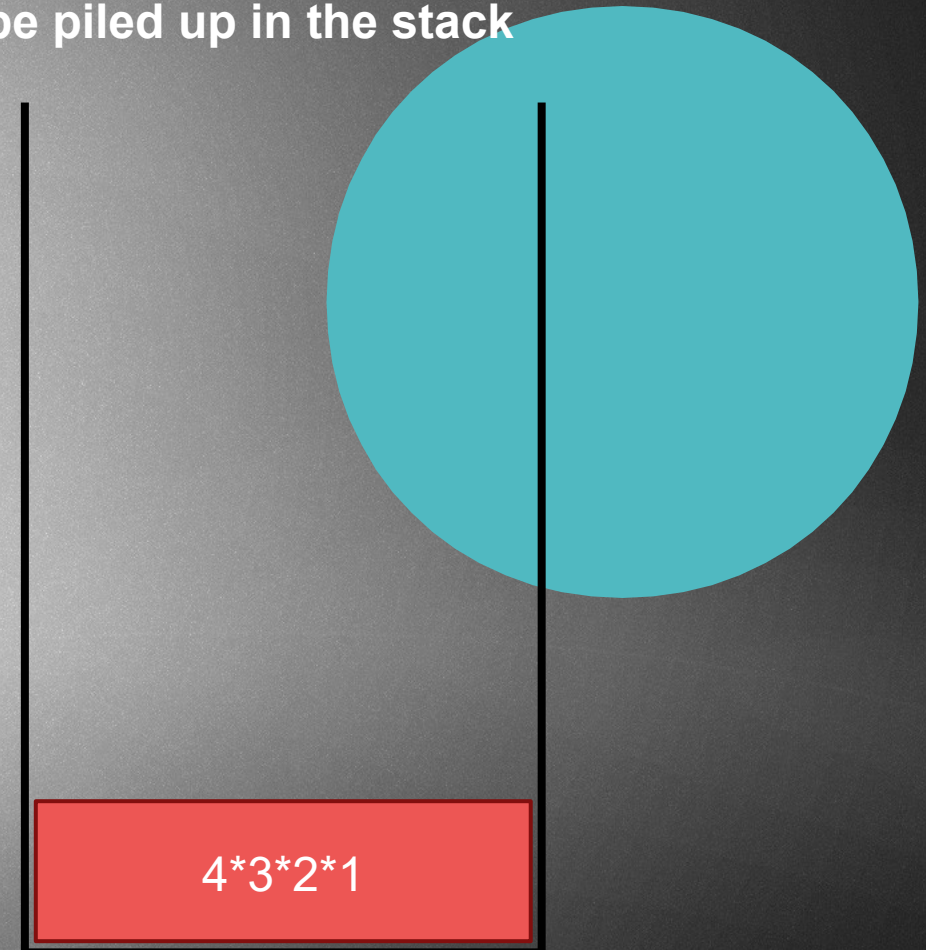
Factorial: factorial(4)

Conclusion: recursive method calls are going to be piled up in the stack

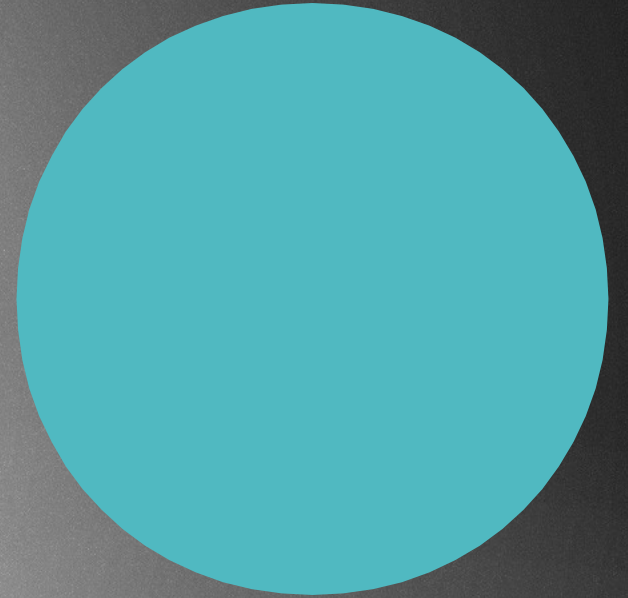
```
public void factorial(int n) {  
    if( n == 0 )  
        return 1;  
  
    return n * factorial(n-1);  
}
```


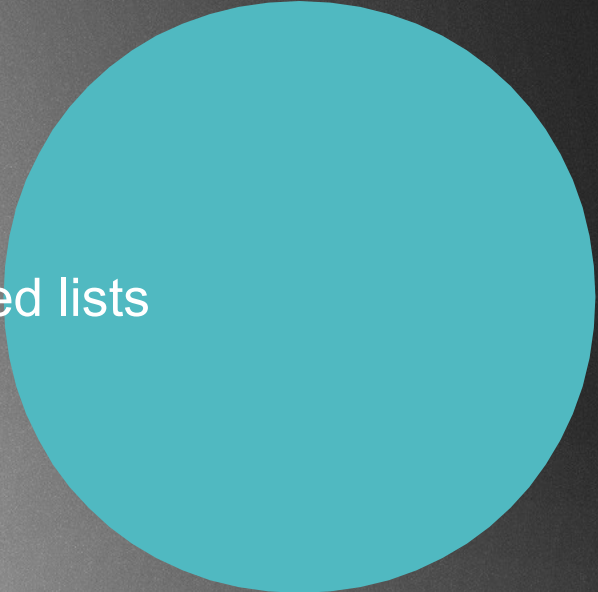
```
int result = factorial(4)
```

Result will be $4*3*2*1 = 24$!!!

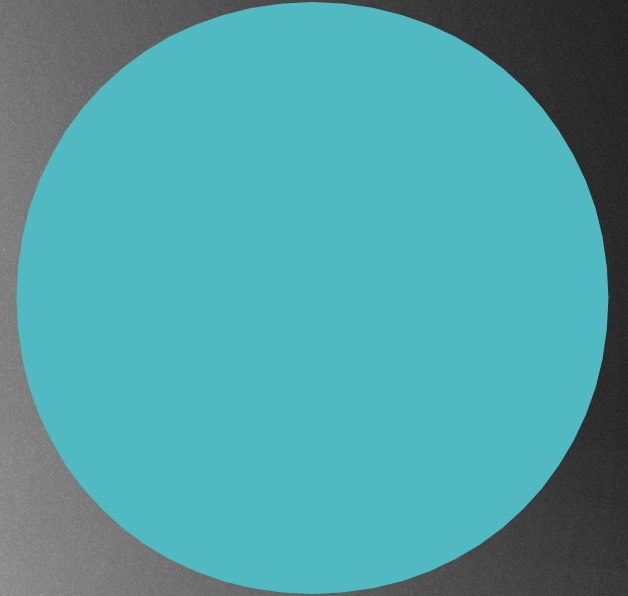
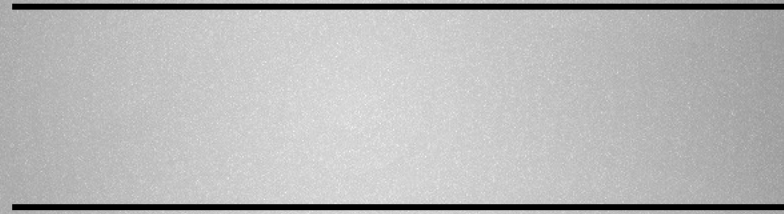


QUEUE



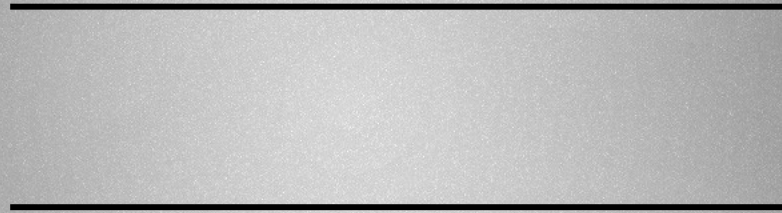
- 
- 
- ▶ It is an abstract data type (interface)
 - ▶ Basic operations: enqueue() and dequeue() , peek()
 - ▶ FIFO structure: first in first out
 - ▶ It can be implemented with dynamic arrays as well as with linked lists
 - ▶ Important when implementing BFS algorithm for graphs

Enqueue operation: we just simply add the new item to the end of the queue



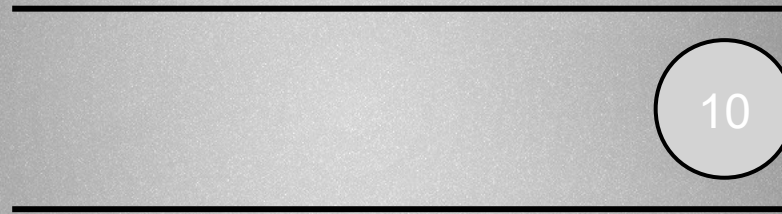
Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(10);
```



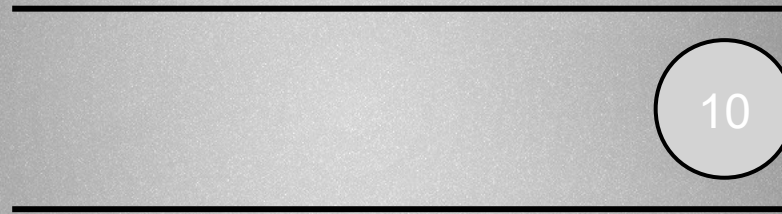
Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(10);
```



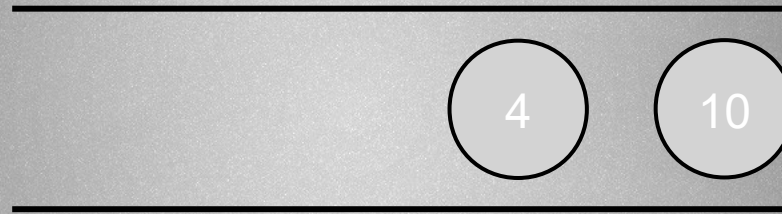
Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(4);
```



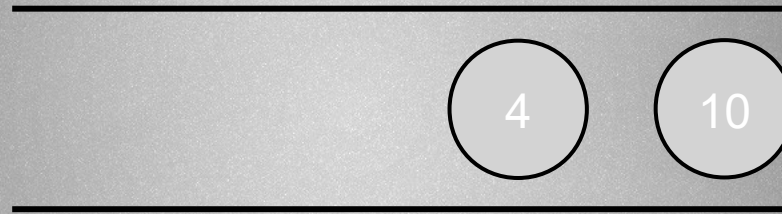
Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(4);
```



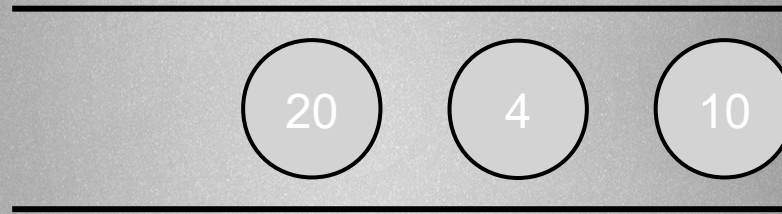
Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(20);
```

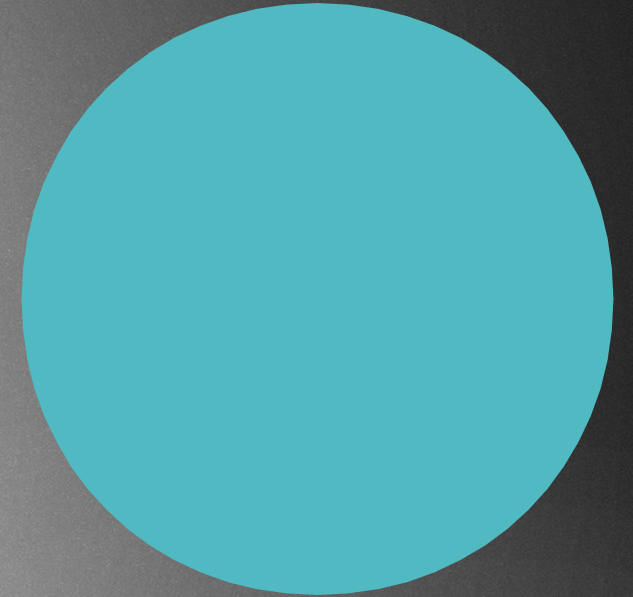
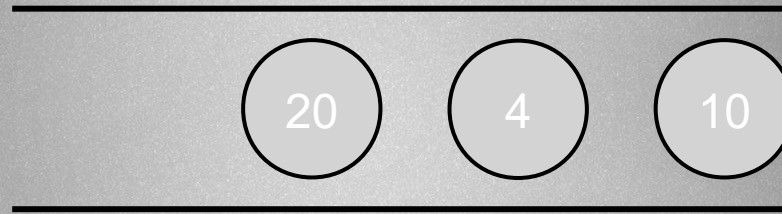


Enqueue operation: we just simply add the new item to the end of the queue

```
queue.enqueue(20);
```

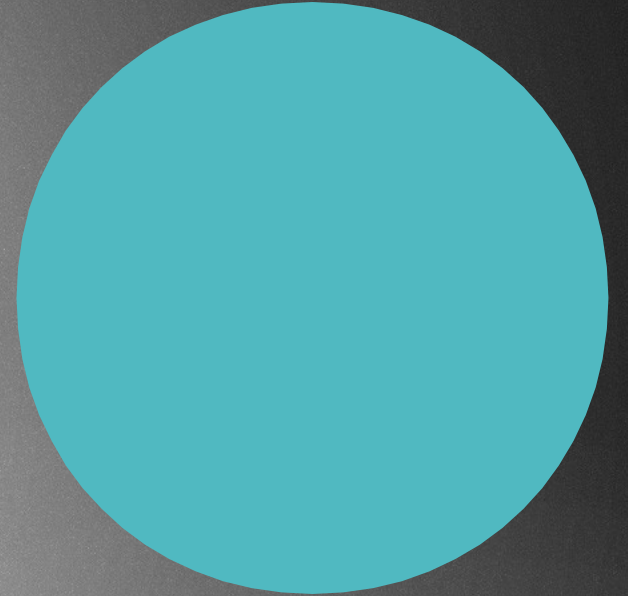
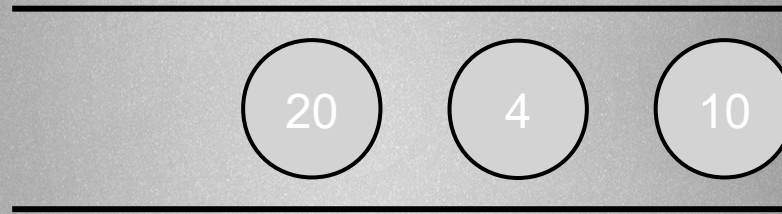


Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure



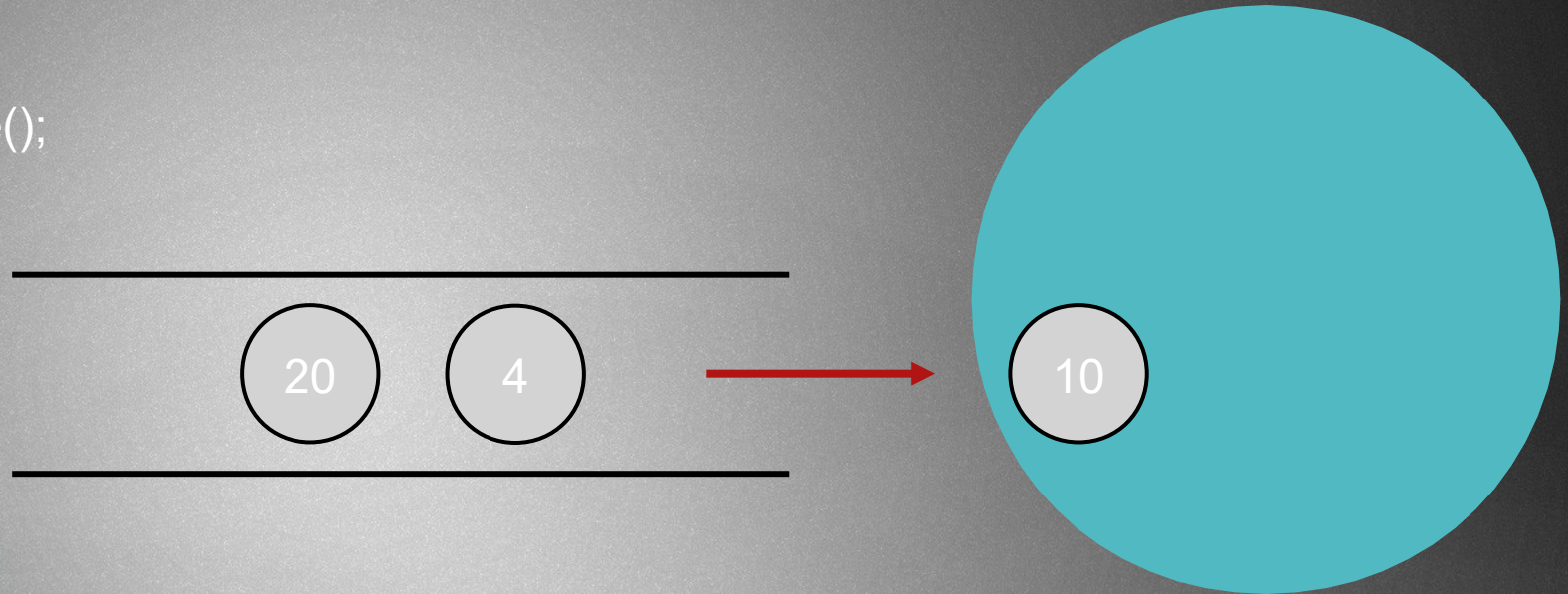
Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.dequeue();
```



Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.dequeue();
```



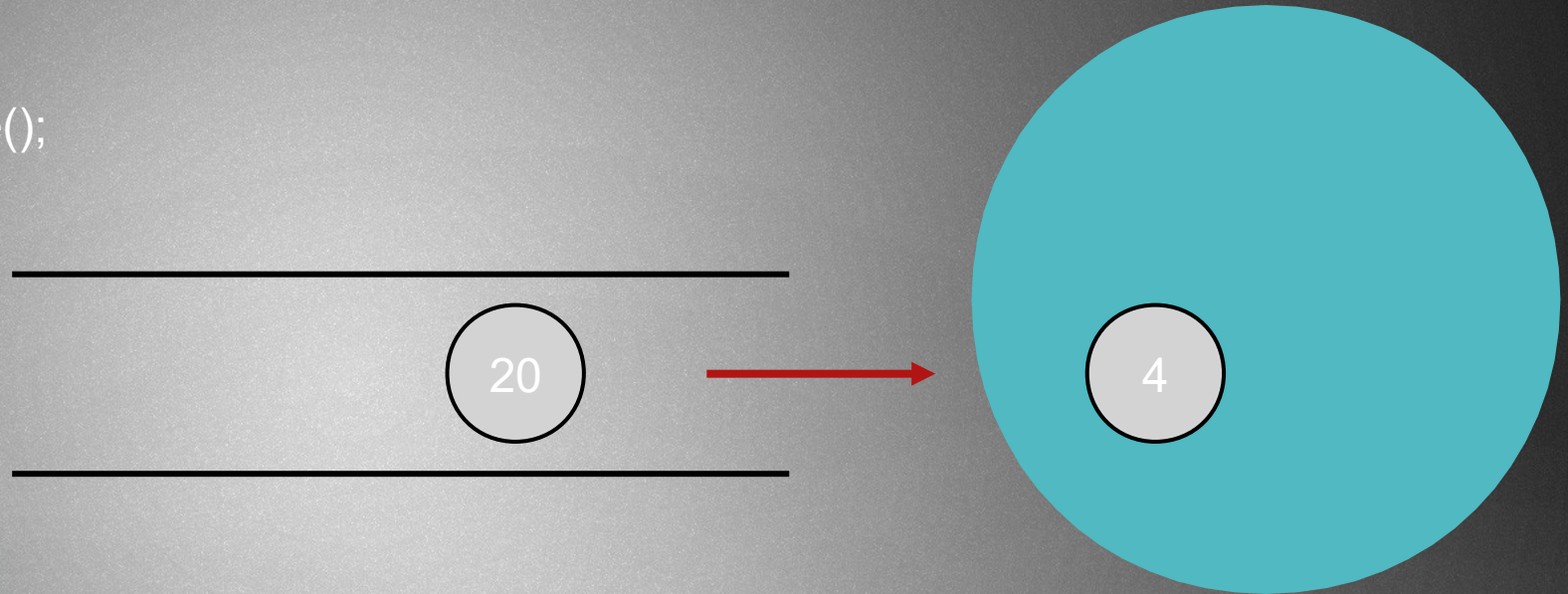
Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.dequeue();
```

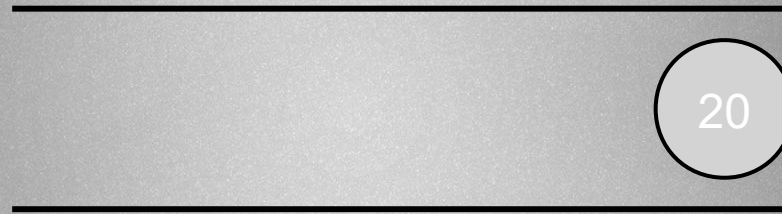


Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure

```
queue.dequeue();
```



Dequeue operation: we just simply remove the item starting at the beginning of the queue // FIFO structure



Applications

- ▶ When a resource is shared with several consumers (threads): we store them in a queue
- ▶ For example: CPU scheduling
- ▶ When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
- ▶ For example: IO buffers
- ▶ Operational research applications or stochastic models relies heavily on queues !!!