Stacks and Queues

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

- Chap. 6: Sections 6.6 and 6.9
- Chap. 7: Section 7.3.3 (using stacks to implement recursion)
- Chap. 11 (applications) and 16 (implementation)

Stack ADT

- A **stack** is a common data structure used in the design of many software systems including operating systems, compilers and in networks.
- A stack is really a special case of the general List ADT introduced in *CompEng 2SH4*.

Stack ADT

- A **stack** is a common data structure used in the design of many software systems including operating systems, compilers and in networks.
- A stack is really a special case of the general List ADT introduced in *CompEng 2SH4*.
  - → Insertions and deletions occur only at one position, called the **top** of the stack.

Stack ADT

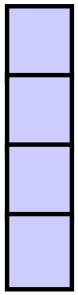
- A **stack** is a common data structure used in the design of many software systems including operating systems, compilers and in networks.
- A stack is really a special case of the general List ADT introduced in *CompEng 2SH4*.
  - → Insertions and deletions occur only at one position, called the **top** of the stack.
  - → Elements are inserted or **pushed** onto the top of the stack

Stack ADT

- A **stack** is a common data structure used in the design of many software systems including operating systems, compilers and in networks.
- A stack is really a special case of the general List ADT introduced in *CompEng 2SH4*.
  - → Insertions and deletions occur only at one position, called the **top** of the stack.
  - → Elements are inserted or **pushed** onto the top of the stack
  - → Only the most recently inserted element can be removed or **popped** from the top of the stack

Stack Definition

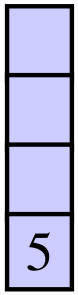
Example: Consider the pseudocode:



```
1  create empty stack
2
3
4
5
```

Stack Definition

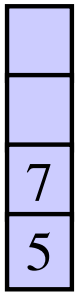
Example: Consider the pseudocode:




```
1  create empty stack
2  push 5
3
4
5
```

Stack Definition

Example: Consider the pseudocode:




```
1  create empty stack
2  push 5
3  push 7
4
5
```

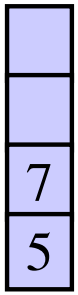
Stack Definition

Example: Consider the pseudocode:


```
1  create empty stack
2  push 5
3  push 7
4  push 3
5
```

Stack Definition

Example: Consider the pseudocode:




```
1  create stack
2  push 5
3  push 7
4  push 3
5  pop                    Returns 3
```

- Stacks are **LIFO** structures – **L**ast **I**n, **F**irst **O**ut

Applications of Stack

- 'Undo' mechanism in text editors
- Syntax check for matching braces
  - $a\{b[c]d\}e$ //correct
  - $a\{b[c]d]e$ // not correct;] does not macth {
- Back/Forward on browsers
- Call stack (information about the active subroutines of a computer program)
- Expression evaluation
- Reversing a word
  - part $\Rightarrow$ trap

Applications of Stack: Matching Braces

$a\{b[c]d\}e$

Character read	Stack contents
a	
{	{
b	{
[	{[
c	{[
]	{
d	{
}	
e	

Stack Operations

- `constructor` - creates a new empty stack.
- `isEmpty` - returns `true` if no element in stack, `false` otherwise.
- `push` - adds a new data item on the top of the stack.
- `pop` - returns and removes the most recently "pushed" element.
- `top` - reads and returns the most recently pushed element.

Stack Operations

- `constructor` - creates a new empty stack.
- `isEmpty` - returns `true` if no element in stack, `false` otherwise.
- `push` - adds a new data item on the top of the stack.
- `pop` - returns and removes the most recently "pushed" element.
- `top` - reads and returns the most recently pushed element.

Java Interface

```
1 public interface Stack<E>{
2     public boolean isEmpty();
3     public E top() throws EmptyStackException;
4     public void push(E e);
5     public E pop() throws EmptyStackException;
6 }
```

Stack Implementation

What data structures can we use to implement a stack?

Stack Implementation

What data structures can we use to implement a stack?

- Array
  - Beginning of array is bottom-of-stack.
  - Use an index to indicate the top-of-stack (TOS).

Stack Implementation

What data structures can we use to implement a stack?

- Array
  - Beginning of array is bottom-of-stack.
  - Use an index to indicate the top-of-stack (TOS).
- Linked-List
  - Top-of-stack is the beginning of the linked list.

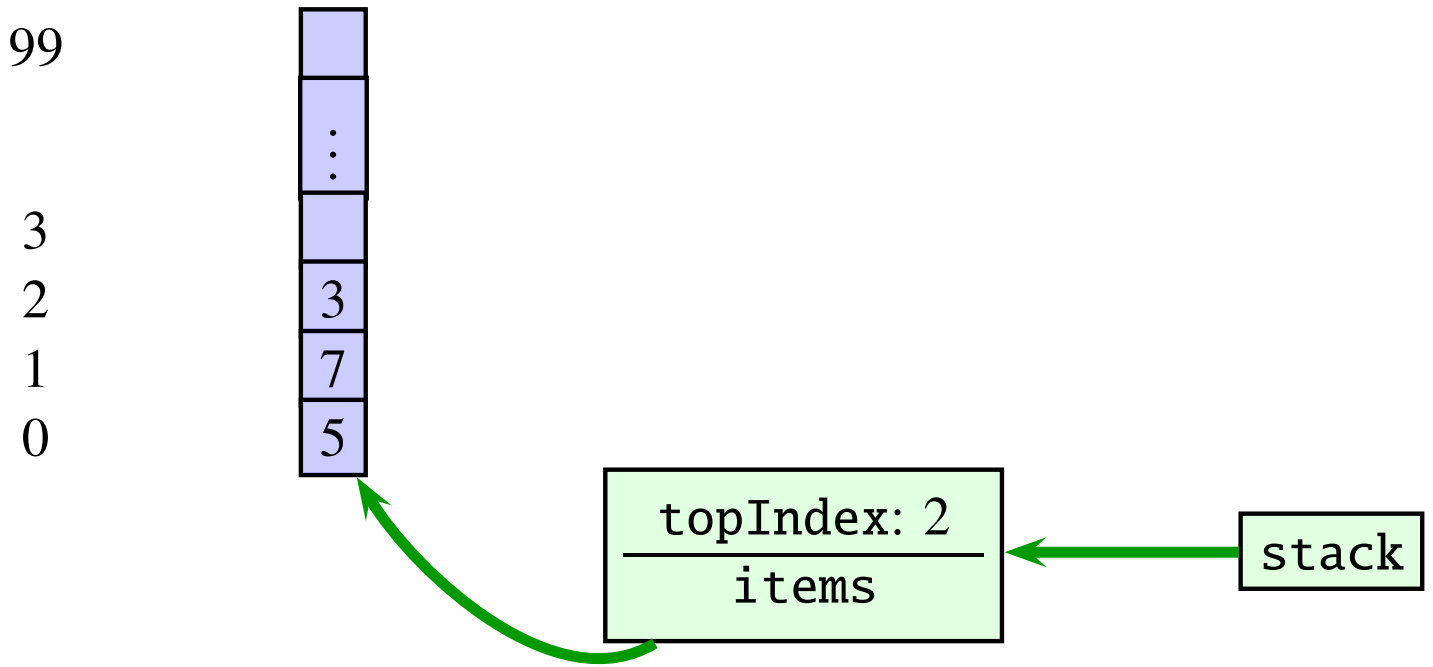
Array Implementation of Stacks

```
1  public class ArrayStack<E> implements Stack<E>{
2      private E[] items; //reference to array of objects of type E
3      private int topIndex = -1; //index of item on top of stack
4
5      //Constructors; each creates an empty stack
6      public ArrayStack(){
7          this(100);
8      }
9      public ArrayStack(int n){
10         items= (E[]) new Object[n];
11     }
12     .....
13 }
```

Array Implementation of Stacks

Example:

```
ArrayStack<Integer> stack = new ArrayStack<Integer>();
stack.push(5); stack.push(7); stack.push(3);
```



Array Implementation of Stacks – `isEmpty`

```
1 public boolean isEmpty(){
2     return(topIndex<0);
3 }
```

→ Variable `topIndex` stores the array index of last element inserted into the stack.

Array Implementation of Stacks – push

```
1  public void push(E e){
2      if(topIndex==items.length-1){
3          E[] newArray = (E[]) new Object[2*items.length];
4          for(int i=0; i<items.length; i++)
5              newArray[i]=items[i];
6          items=newArray;
7      }//end if
8      items[++topIndex] = e;
9  }
```

- Adds the element to the stack.
  - → If there is no room, a double size array is allocated and all elements are copied over.
- Notice the **pre-increment** operator.

Array Implementation of Stacks – top

```
1 public E top() throws EmptyStackException{
2     if(isEmpty())
3         throw new EmptyStackException("Stack underflow");
4     else
5         return(items[topIndex]);
6 }
```

- Returns only the **value** of the element which was last pushed onto the stack.
- Program throws an exception if the user tries to `top` on an empty stack.

Array Implementation of Stacks – pop

```
1 public E pop() throws EmptyStackException{
2     if(isEmpty()){
3         throw new EmptyStackException("Stack underflow");
4     else{
5         E tempE = items[topIndex];
6         items[topIndex--] = null;
7         return(tempE);
8     }
9 }
```

→ Returns the element which was last pushed onto the stack and "removes" it from the stack (decrements `topIndex` index value).

→ Throws exception if you try to `pop` on an empty stack.

→ The object reference in the stack array is set to `null` to let **garbage collection** clean it up.

Run Time

- `isEmpty()`, `pop()` and `top()`: $\Theta(1)$ time.
- A `push()` that does not involve doubling the array: $\Theta(1)$ time.

Run Time

- `isEmpty()`, `pop()` and `top()`: $\Theta(1)$ time.
- A `push()` that does not involve doubling the array: $\Theta(1)$ time.
- A `push` involving doubling the array takes $\Theta(n)$ time. Such a `push` was preceded by at least $n/2$ `pushes` that did not involve doubling the array. The $\Theta(n)$ cost of doubling can be charged over these $n/2$ easy `pushes`, thus effectively increasing the cost of each `push` by only a small amount.
  - → Thus, `push` takes $\Theta(1)$ **amortized** running time (which means **the average cost per push in a sequence of n consecutive `pushes` since the stack was created**).

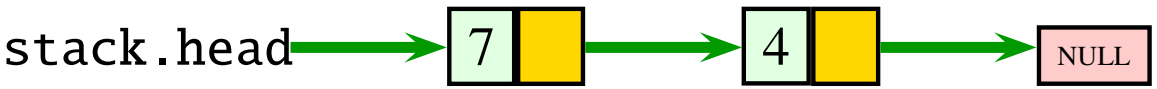
Linked-List Implementation of Stacks

- In a **singly linked list** insertions and deletions **at the start** of the list are **very fast** - $\Theta(1)$ time.
- Let the **top of the stack** be the **beginning** of the list.
- We will consider the implementation **without a dummy header**.

Examples:

- *Empty Stack*

- *Stack with top-of-stack element = 7*



Stack Operations

Recall the `Stack` Interface:

```
1 public interface Stack<E>{
2     public boolean isEmpty();
3     public E top() throws EmptyStackException;
4     public void push(E e);
5     public E pop() throws EmptyStackException;
6 }
```

Linked-List Implementation of Stacks

- Define a class to represent **nodes** in the list.

Example:

```
1  class Node<E>{
2      E element;
3      Node<E> next;
4
5      public Node(E e, Node<E> n){
6          element=e;
7          next=n;
8      }
9  }
```

Linked-List Implementation of Stacks

```
1 public class LLStack<E> implements Stack<E>{
2     private Node<E> head;
3     public LLStack(){head = null;} //creates an empty stack
4     public boolean isEmpty(){ ... }
5     public E top() throws EmptyStackException{ ... }
6     public void push(E e){ ... }
7     public E pop() throws EmptyStackException{ ... }
8 }
```

Linked-List Implementation of Stacks–`isEmpty`

```
1 public boolean isEmpty(){
2     return(head == null);
3 }
```

→ The stack is empty if there are no nodes.

→ Run time: $\Theta(1)$

Linked-List Implementation of Stacks – push

```
1 public void push(E e){
2     head = new Node<E>(e, head);
3 }
```

→ push is an insertion at the beginning of the linked list.

→ $\Theta(1)$ running time.

Linked-List Implementation of Stacks – pop

```
1  public E pop() throws EmptyStackException{
2      if(isEmpty())
3          throw new EmptyStackException("Stack Underflow");
4      else {
5          E e = head.element;          //Element to pop
6          head = head.next;              //Bypass first node
7          return e;
8      }
9  }
```

→ pop deletes the first node of the linked list and returns the element stored there.

→ $\Theta(1)$ running time.

Linked-List Implementation of Stacks – top

```
1 public E top() throws EmptyStackException{
2     if(isEmpty())
3         throw new EmptyStackException("Stack Underflow");
4     else {
5         return(head.element);
6     }
7 }
```

→ Same as pop except that it does not remove an element from the list.

→ $\Theta(1)$ running time.

Linked-List Vs. Array Implementation of Stacks

McMaster University

- **Time**:
  - → `new` commands are typically more expensive operations than arithmetic operations or assignments (in terms of time to execute).
  - → Although both array and linked-list implementations are $\Theta(1)$ for `pop` and `push` (when no array doubling is involved), the **constant is likely larger** for the **linked-list implementation**.

Linked-List Vs. Array Implementation of Stacks

McMaster University

- **Time**:
  - → `new` commands are typically more expensive operations than arithmetic operations or assignments (in terms of time to execute).
  - → Although both array and linked-list implementations are $\Theta(1)$ for `pop` and `push` (when no array doubling is involved), the **constant is likely larger** for the **linked-list implementation**.
- **Space**:
  - → The linked-list implementation requires extra memory - beside stack items it also stores references (links) to list nodes.
  - → The array implementation also requires extra memory - to store the stack items (since a higher capacity array is allocated to have room for future pushes).

Stack Application - Handling Function Calls

- A stack of **activation records** is used to handle method (function) **calls** during program execution.
  - In a sequence of method calls the **last** method invoked is the **first** one to return - LIFO structure.

Stack Application - Handling Function Calls

- A stack of **activation records** is used to handle method (function) **calls** during program execution.
  - In a sequence of method calls the **last** method invoked is the **first** one to return - LIFO structure.
- An activation record (AR) for a method call stores parameters, other local variables, program pointer position after returning.

Stack Application - Handling Function Calls

- A stack of **activation records** is used to handle method (function) **calls** during program execution.
  - In a sequence of method calls the **last** method invoked is the **first** one to return - LIFO structure.
- An activation record (AR) for a method call stores parameters, other local variables, program pointer position after returning.
- When a method is **invoked** an activation record (AR) is **pushed** onto the stack.
- When the method **returns**, an AR is **popped** off the stack.

Stack Application - Implementing Function Calls

Example: Compute 4! using

```
1 public int fact (int n){
2     if(n<=1)
3         return(1);
4     else
5         return(n*fact(n-1));
6 }
```

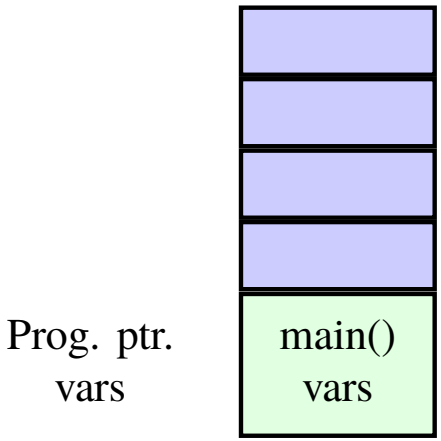
Stack Application: Implementing Function Calls (2)

- What happens when we call `fact(4)` ?

Stack Application: Implementing Function Calls (2)

- What happens when we call `fact(4)` ?

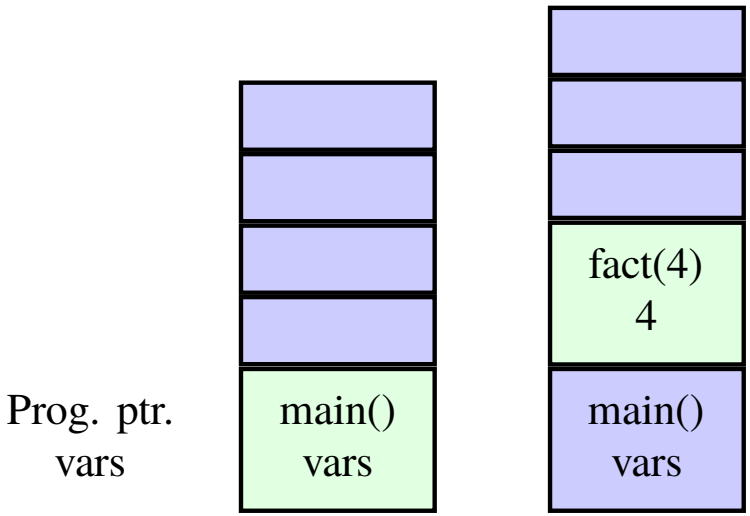
  Call `fact(4)`



Stack Application: Implementing Function Calls (2)

- What happens when we call `fact(4)` ?

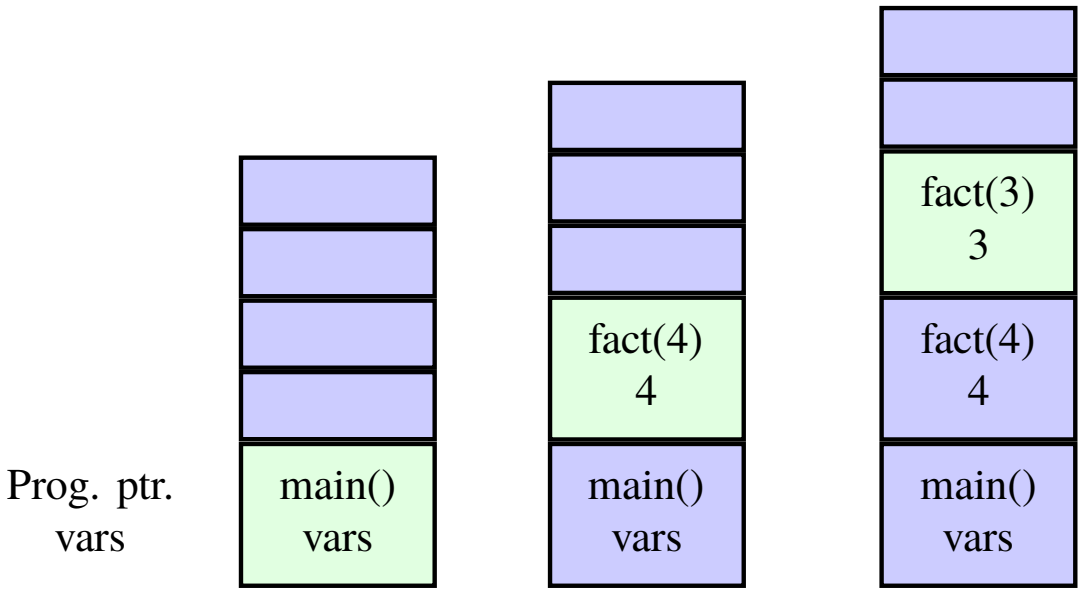
Call `fact(4)` Call `fact(3)`



Stack Application: Implementing Function Calls (2)

- What happens when we call `fact(4)` ?

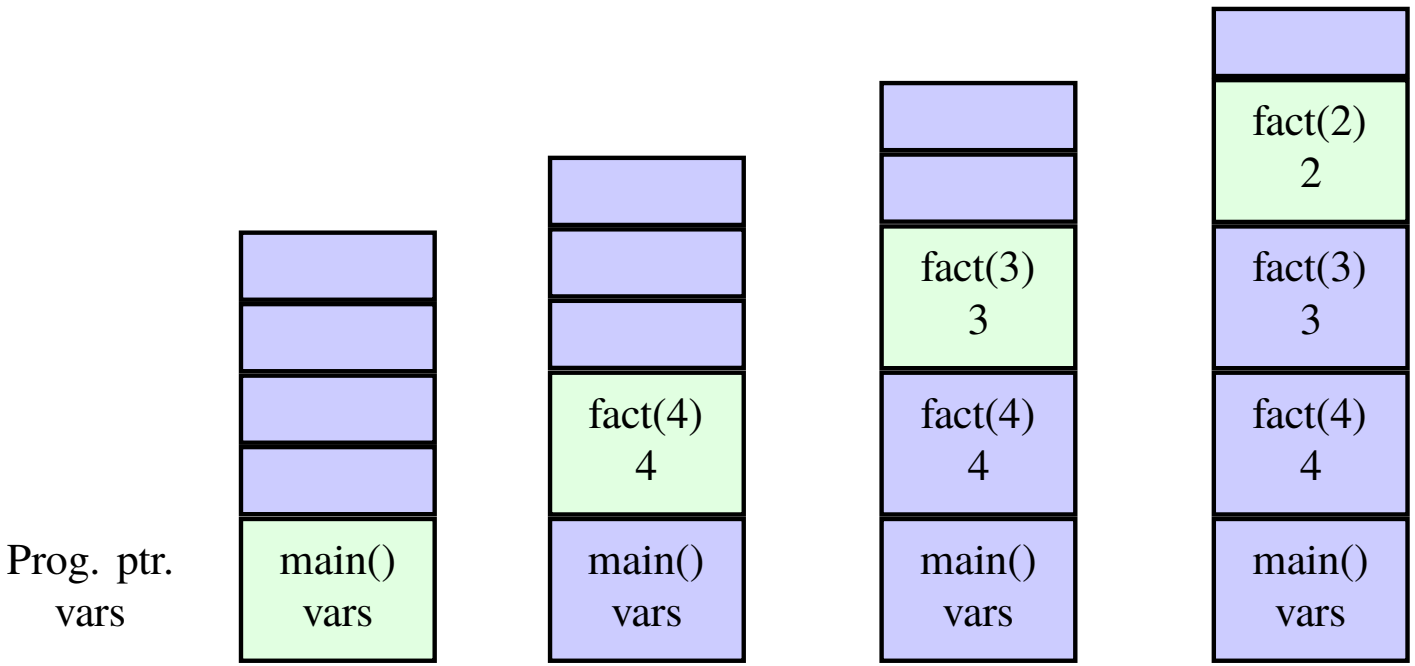
Call `fact(4)` Call `fact(3)` Call `fact(2)` Call `fact(1)`



Stack Application: Implementing Function Calls (2)

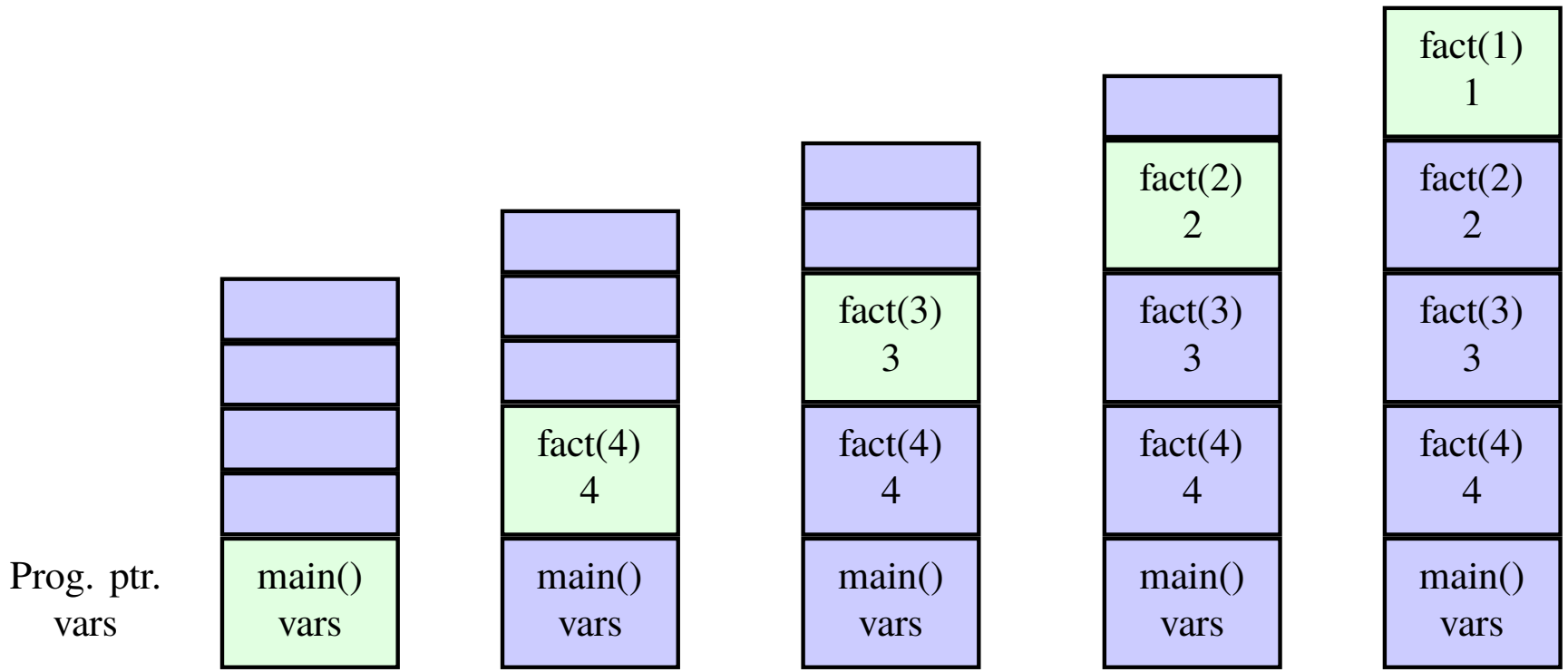
- What happens when we call `fact(4)` ?

Call `fact(4)` Call `fact(3)` Call `fact(2)` Call `fact(1)`



Stack Application: Implementing Function Calls (2)

- What happens when we call `fact(4)` ?

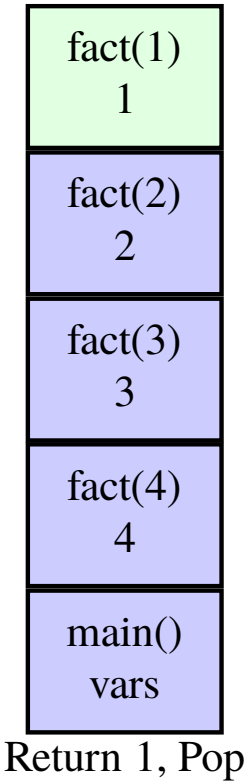


Stack Application: Implementing Function Calls (3)

Now, when the functions return ...

Stack Application: Implementing Function Calls (3)

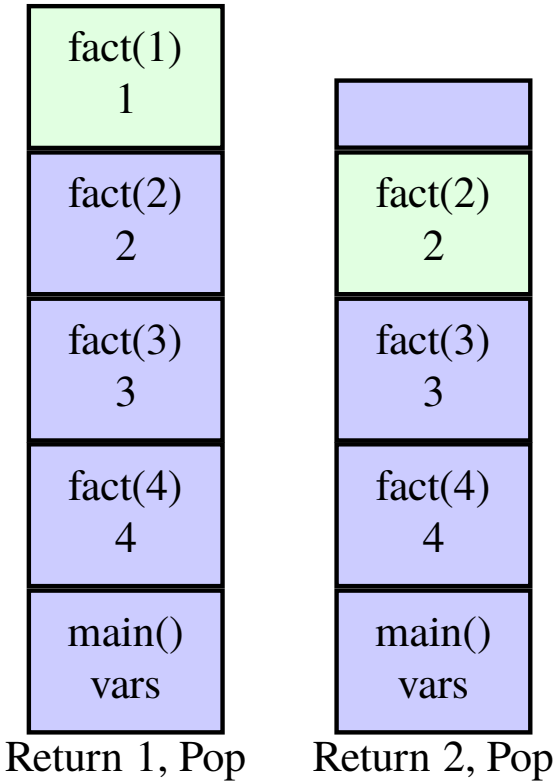
Now, when the functions return ...



Return 1, Pop

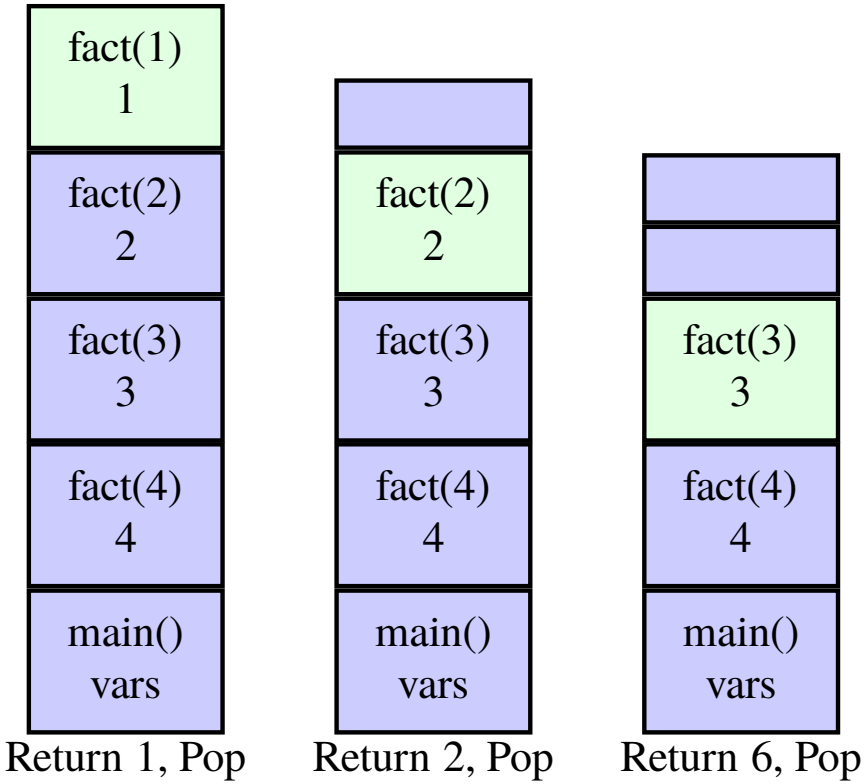
Stack Application: Implementing Function Calls (3)

Now, when the functions return ...



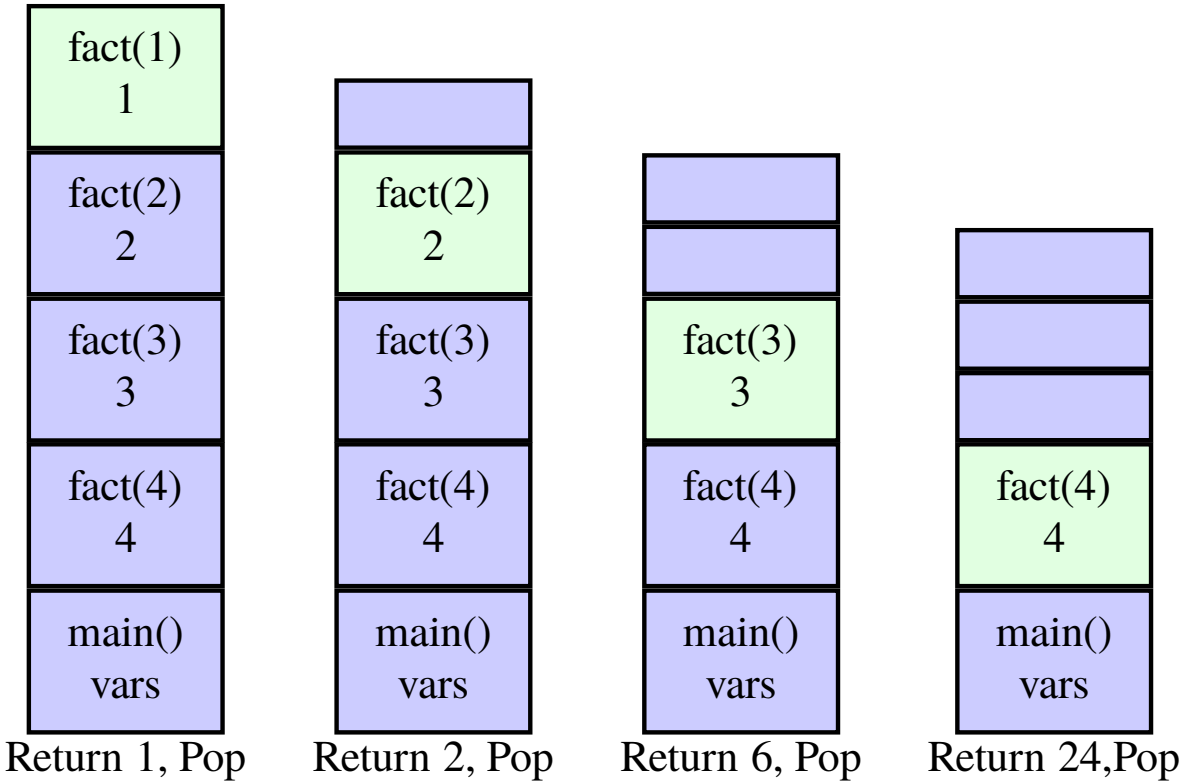
Stack Application: Implementing Function Calls (3)

Now, when the functions return ...



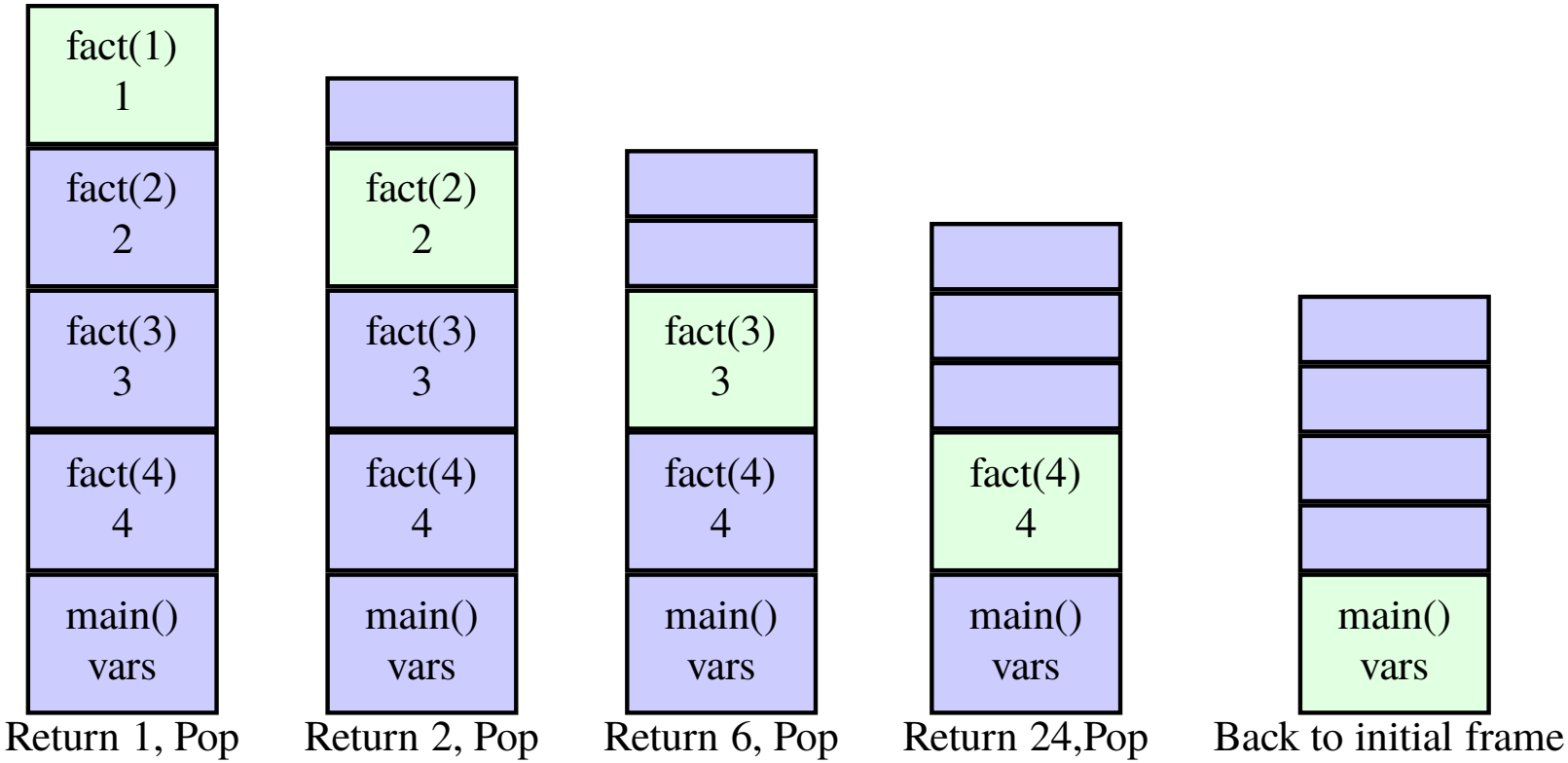
Stack Application: Implementing Function Calls (3)

Now, when the functions return ...



Stack Application: Implementing Function Calls (3)

Now, when the functions return ...



Stack Application: Implementing Function Calls (4)

- In some high-speed systems, these stacks have limited error checking and if you exceed the stack capacity (a.k.a **blow the stack**), which can result in computer instability.
  - A recursive function which calls itself many times (Example: `fact(10000)`)
  - Passing very large arrays by value

Stack Application: Postfix Calculator

Types of notation to illustrate arithmetic expressions:

- **Infix** notation (operator **between** operands)
  → 1 + (2*3)
- **Prefix** notation (operator **before** operands)
  → + 1 * 2 3
- **Postfix** notation (operator **after** operands)
  → 1 2 3 *+

Stack Application: Postfix Calculator

Types of notation to illustrate arithmetic expressions:

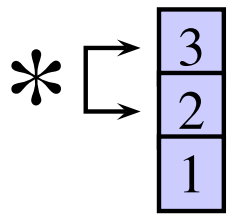
- **Infix** notation (operator **between** operands)
  - → 1 + (2*3)
- **Prefix** notation (operator **before** operands)
  - → + 1 * 2 3
- **Postfix** notation (operator **after** operands)
  - → 1 2 3 *+
  - → Can be evaluated using a **stack** !

Stack Application: Postfix Calculator (2)

- Postfix expressions can be evaluated using a stack.
- The expression is a sequence of **operands** and **operators**. Assume operators are binary.
- The sequence is scanned from left to right.
- When an **operand** is seen it is **pushed** onto the stack.
- When an **operator** is seen, two operands are **popped** from the stack. The operator is **evaluated**. The result is **pushed** back.
- When **done**, stack should contain only one item - the **final result**.
  - → Example: 1 2 3 * +

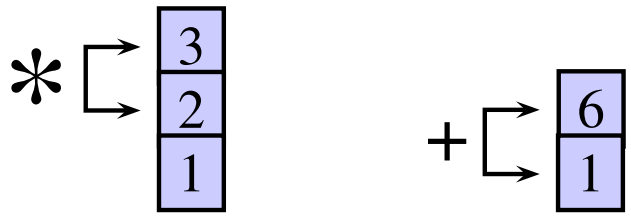
Stack Application: Postfix Calculator (2)

- Postfix expressions can be evaluated using a stack.
- The expression is a sequence of **operands** and **operators**. Assume operators are binary.
- The sequence is scanned from left to right.
- When an **operand** is seen it is **pushed** onto the stack.
- When an **operator** is seen, two operands are **popped** from the stack. The operator is **evaluated**. The result is **pushed** back.
- When **done**, stack should contain only one item - the **final result**.
  - → Example: 1 2 3 * +



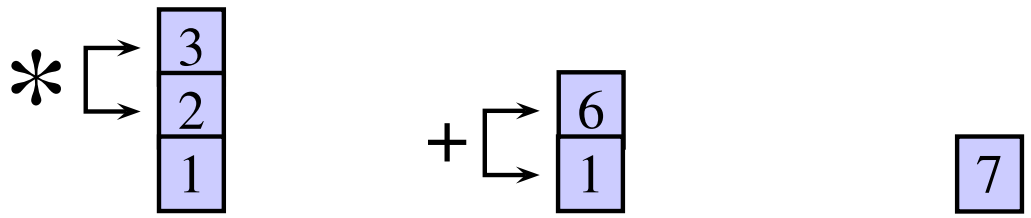
Stack Application: Postfix Calculator (2)

- Postfix expressions can be evaluated using a stack.
- The expression is a sequence of **operands** and **operators**. Assume operators are binary.
- The sequence is scanned from left to right.
- When an **operand** is seen it is **pushed** onto the stack.
- When an **operator** is seen, two operands are **popped** from the stack. The operator is **evaluated**. The result is **pushed** back.
- When **done**, stack should contain only one item - the **final result**.
  - → Example: 1 2 3 * +



Stack Application: Postfix Calculator (2)

- Postfix expressions can be evaluated using a stack.
- The expression is a sequence of **operands** and **operators**. Assume operators are binary.
- The sequence is scanned from left to right.
- When an **operand** is seen it is **pushed** onto the stack.
- When an **operator** is seen, two operands are **popped** from the stack. The operator is **evaluated**. The result is **pushed** back.
- When **done**, stack should contain only one item - the **final result**.
  - → Example: 1 2 3 * +



Stack Application: Postfix Calculator (3)

```
1  String input="1 2 3 * +";   //assume operands are integers
2
3  StringTokenizer st =new StringTokenizer(input); String s;
4
5  Integer x,y,result;
6
7  Stack<Integer> stk = new Stack<Integer>();
8  while(st.hasMoreTokens()){
9      s = st.nextToken();
10     if(s.equals("+") || s.equals("-") || s.equals("*")
11                      || s.equals("/")){
12         x = stk.pop(); y = stk.pop();
13         switch(s.charAt(0)){
14             case '+':
15                 stk.push(y+x); break;
16             case '-':
17                 stk.push(y-x); break;
```

Stack Application: Postfix Calculator (4)

```
1           case '*':
2                   stk.push(y*x); break;
3           case '/':
4                   stk.push(y/x); break;
5           } //end switch
6
7       } else  //token is string representing an integer
8           stk.push(new Integer(s));
9       }
10      result = stk.pop();
11  }
```

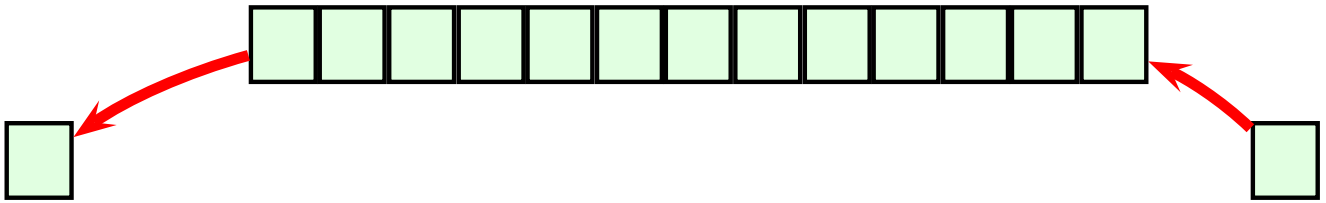
Queue ADT

- The **queue** is a fundamental abstract data type used in operating systems, networks and in simulators.
- Supports insertions and deletions according to the **F**irst **I**n, **F**irst **O**ut principle.
- A model for a (fair!) **waiting line**.

Queue ADT (2)

- Queue ADT - Definition:
  - Description of data: a collection of items
  - Description of operations:
    - **enqueue**: insert a new item.
    - **dequeue**: remove the item which has been in the queue for the longest time.

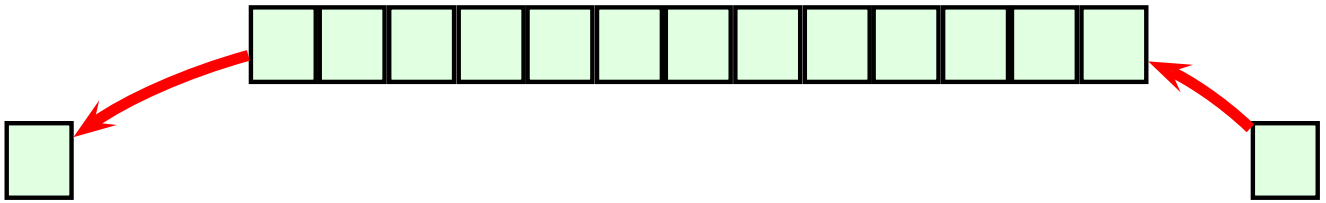
Queue Definition



Remove from front

Insert at end

Queue Definition



Remove from front

Insert at end

- Queues are **FIFO** structures – **F**irst **I**n, **F**irst **O**ut
- These data structures are good models of people waiting in line (with no cutting in line!)
  "First come, First serve".
- A queue is a special case of a list where
  - → Insertions occur only at the **end** of the list (enqueue).
  - → Deletions occur at the **front** of the list (dequeue).

Applications of Queue

- Serving requests on a single shared resource, like a printer
- CPU task scheduling
- Call Center phone systems uses Queues to hold people calling them in an order
- Sending and receiving data through TCP/UDP protocols
- Buffers on MP3 players

Queue Operations

- `constructor` - creates a new empty queue.
- `getSize` - returns the number of elements in the queue.
- `isEmpty` - returns `TRUE` if no element in the queue, `FALSE` otherwise.
- `isFull` - returns `TRUE` if the maximum queue capacity is reached (for the case with max capacity)
- `enqueue` - inserts a new data item.
- `dequeue` - returns and removes the least recently inserted item.

Queue Operations

- `constructor` - creates a new empty queue.
- `getSize` - returns the number of elements in the queue.
- `isEmpty` - returns `TRUE` if no element in the queue, `FALSE` otherwise.
- `isFull` - returns `TRUE` if the maximum queue capacity is reached (for the case with max capacity)
- `enqueue` - inserts a new data item.
- `dequeue` - returns and removes the least recently inserted item.

```
1 public interface Queue<E>{
2     public int getSize();
3     public boolean isEmpty();
4     public boolean isFull();
5     public void enqueue(E e) throws FullQueueException;
6     public E dequeue() throws EmptyQueueException;
7 }
```

Queue Implementation

How can you implement a queue efficiently?

Queue Implementation

How can you implement a queue efficiently?

- Using an array.
  - Circular array implementation.

Queue Implementation

How can you implement a queue efficiently?

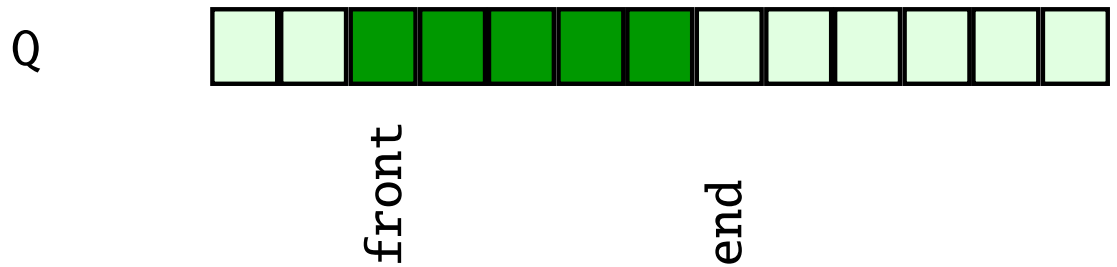
- Using an array.
  - Circular array implementation.
- Using a linked-list
  - Insert elements at the end of the list, and remove elements from front of the list.

Array Implementation of Queues

- Consider the implementation of a queue with a limited capacity.
  - → Allocate a fixed array `Q` to store queue items.

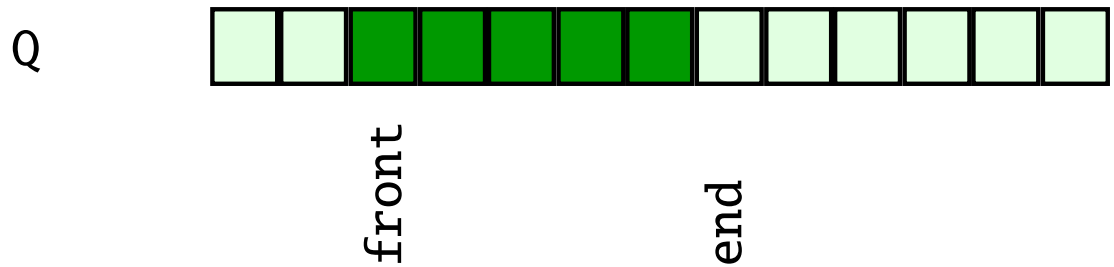
Array Implementation of Queues

- Consider the implementation of a queue with a limited capacity.
  - → Allocate a fixed array `Q` to store queue items.
- Define two indices:
  - → `end` – indicates the end of the queue (index of **next available** position in array).
    - ⇨ when enqueuing **increment** `end`.
  - → `front` – to store the array index of first queue element.
    - ⇨ when dequeuing **increment** `front`.



Array Implementation of Queues

- Consider the implementation of a queue with a limited capacity.
  - → Allocate a fixed array `Q` to store queue items.
- Define two indices:
  - → `end` – indicates the end of the queue (index of **next available** position in array).
    - ⇨ when enqueuing **increment** `end`.
  - → `front` – to store the array index of first queue element.
    - ⇨ when dequeuing **increment** `front`.

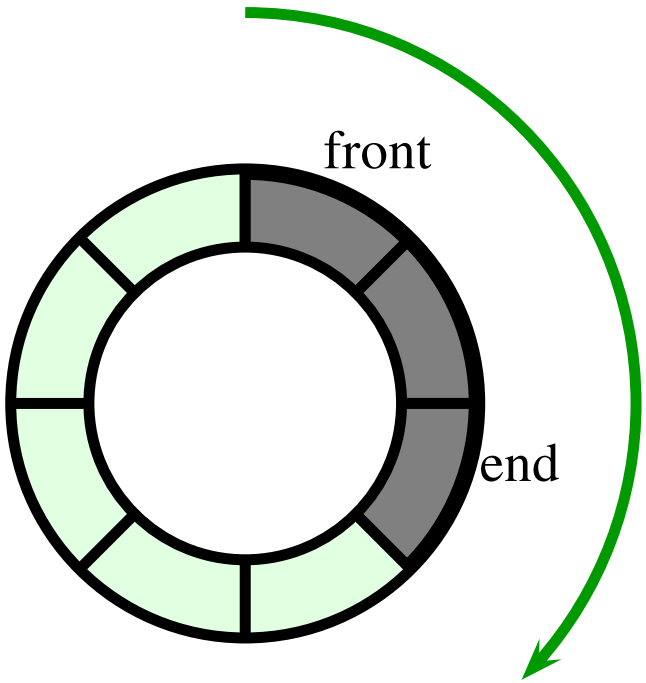


Question: What happens when the indices reach the end of the array?

Array Implementation of Queues (2)

Question: What happens when the indices reach the end of the array?
Answer: **Circular Queue** (or more precisely Circular Array Implementation of the Queue ADT)

- → Imagine that the array is circular, i.e., index 0 is the next index after `Q.length-1`.
- → Increment `front` or `end` **modulo** `Q.length`
  - ⇨ `k=(k+1)% Q.length`
  - ⇨ **Example**: Say `Q.length=5`. If `k=4`, then `(k+1)% 5 = 0`, i.e., the index has wrapped around the end of the array.



Circular Array Implementation of Queues

→ The queue is **empty** when `front==end`.

Circular Array Implementation of Queues

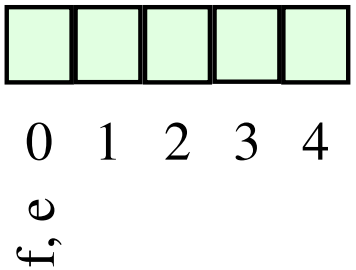
- The queue is **empty** when `front==end`.
- The queue is **full** when the number of items equals `Q.length-1`. Thus, in a full queue one array location is unused.
  - If we allowed the queue to fill the whole array, then for a full queue we would also have `front==back` and we could not distinguish between a full and an empty queue.
  - We will discuss later implementations that do not waste one array location.

Array Implementation of Queues

```
1  public class ArrayQueue<E> implements Queue<E>{
2      public static final int CAPACITY = 1000;  //default array length
3      private E[] Q;       //ref to array of objects of type E
4      private int front = 0;  //indicates front of queue
5      private int end = 0;    //indicates next position after end of queue
6      //Constructors
7      public ArrayQueue(){
8          this(CAPACITY);
9      }
10
11     public ArrayQueue(int cap){
12         Q = (E[])new Object[cap];
13     }
14     .....
15 }
```

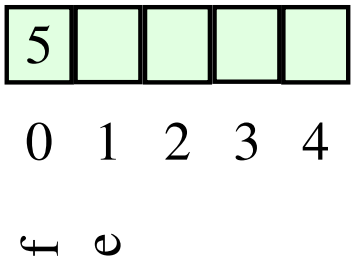
Array Implementation of Queues – Example

```
1  ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2  Integer val;
3  q.enqueue(5); q.enqueue(7);
4  q.enqueue(3); q.enqueue(2);
5  val = q.dequeue(); val=q.dequeue();
6  q.enqueue(8); q.enqueue(9);
```



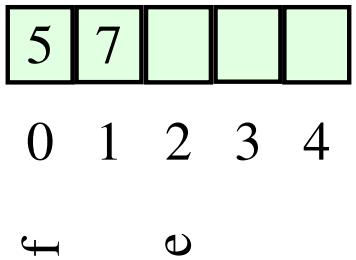
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



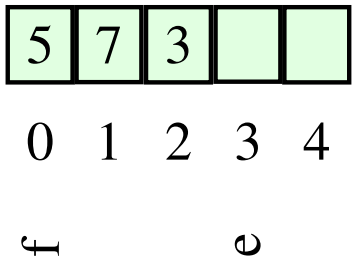
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



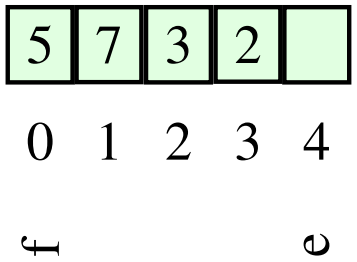
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



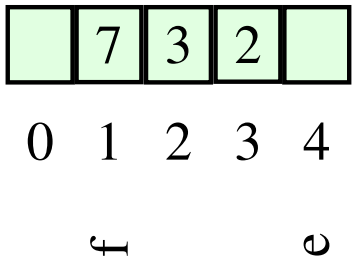
Array Implementation of Queues – Example

```
1  ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2  Integer val;
3  q.enqueue(5); q.enqueue(7);
4  q.enqueue(3); q.enqueue(2);
5  val = q.dequeue(); val=q.dequeue();
6  q.enqueue(8); q.enqueue(9);
```



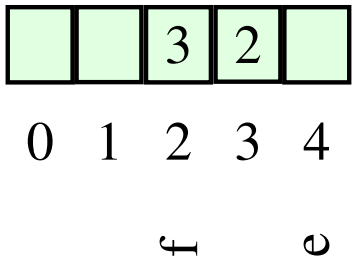
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



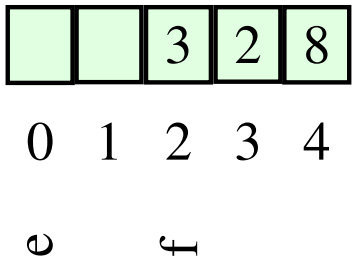
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



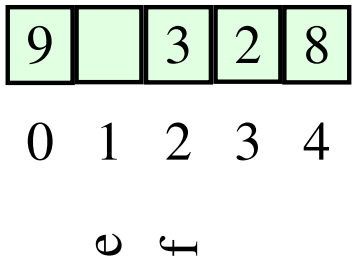
Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



Array Implementation of Queues – Example

```
1    ArrayQueue<Integer> q = new ArrayQueue<Integer>(5);
2    Integer val;
3    q.enqueue(5); q.enqueue(7);
4    q.enqueue(3); q.enqueue(2);
5    val = q.dequeue(); val=q.dequeue();
6    q.enqueue(8); q.enqueue(9);
```



Array Implementation of Queues – `isEmpty`

```
1 public boolean isEmpty(){
2     return(front == end);
3 }
```

→ The queue is empty when the front and back indices are equal

Array Implementation of Queues – `getSize`

```
1 public int getSize(){
2     if (front <= end)
3         return (end-front);
4     else
5         return (Q.length+end-front);
6 }
```

→ The number of elements between the indices is the number of elements in the queue.

→ In second case, add `Q.length` to ensure that the result is non-negative

Array Implementation of Queues – `isFull`

```
1 public boolean isFull(){
2     return(getSize() == Q.length-1);
3 }
```

→ Returns `true` if queue is full.

→ Queue is full if it contains `Q.length`-1 items.

Array Implementation of Queues – enqueue

```
1 public void enqueue(E e) throws FullQueueException{
2     if(isFull())
3         throw new FullQueueException("Queue overflow");
4     else{
5         Q[end]=e;
6         end = (end+1) % Q.length;
7     }
8 }
```

→ Adds the element to the queue and increments the end index with "wraparound".

→ The modulo operation can be expensive and so it may be faster to do a comparison rather than a modulo in some systems, i.e., you can replace Line 6 with:

```
if(++end >= Q.length)
    end = 0;
```

Array Implementation of Queues – dequeue

```
1  public E dequeue() throws EmptyQueueException{
2      E e;
3      if(isEmpty())
4          throw new EmptyQueueException("Error: queue underflow!");
5      else{
6          e = Q[front];
7          Q[front]=null;              // Explicit Nulling
8          if(++front == Q.length)
9              front = 0;
10         return(e);
11     }
12 }
```

→ Returns the element which was in the queue for the longest time and increments the front index with wraparound.

→ The method throws an exception if you try to get an element from an empty queue

→ Using a conditional statement may be faster than using a modulo operation

Array Implementation of Queues

- Notice that `enqueue` and `dequeue` routines are very fast - $\Theta(1)$ running time.
- To allow the queue to grow above the initial capacity, use dynamic array expansion (like for stacks) - `enqueue` takes $\Theta(1)$ **amortized** running time.

Array Implementation of Queues - Notes

- In the implementation presented here, a full queue uses the whole array except for one location.
- **Question**: Can we avoid this waste of space?

Array Implementation of Queues - Notes

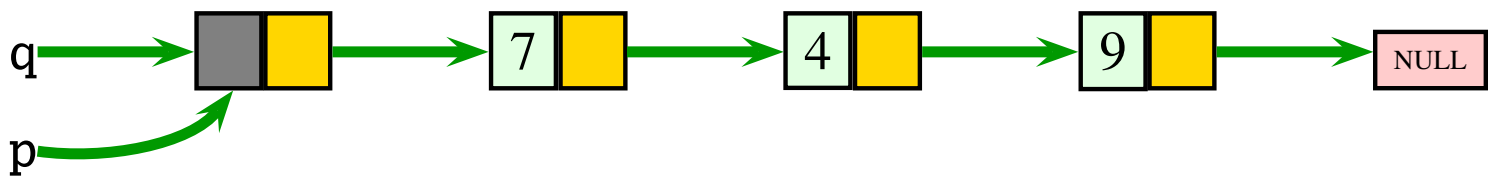
- In the implementation presented here, a full queue uses the whole array except for one location.
- **Question**: Can we avoid this waste of space?
  - → Replace the check for an empty queue by `Q[front]==null` and for a full queue by `front==end && Q[front]!=null` - this solution does not work with an array of primitive variables
  - → Beside `front` and `end` also maintain a variable `queueSize` to store the number of items in the queue - update its value as appropriate after each queue operation.
  - → Maintain only `queueSize` and `end`. To dequeue, compute the array index where the first item is stored, as `(end>=queueSize)?(end-queueSize):(end-queueSize+Q.length)`

Linked-List Implementation of Queues

- You can implement a queue using a linked list with or without a header "dummy" node.
- Insertions are done at the end of the list and deletions are done at the start of the list.
- However, to `enqueue`, we need to traverse the whole list to find the end : takes $\Theta(n)$ operations.

Example: Suppose we already have a linked list and want to find the end of the list.

```
1 for(Node<E> p = q ; p.next ; p=p.next);
```

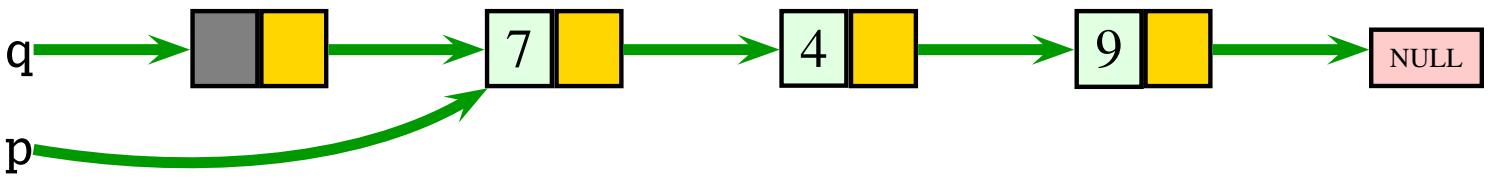


Linked-List Implementation of Queues (2)

- You can implement a queue using a linked list with or without a header "dummy" node.
- Insertions are done at the end of the list and deletions are done at the start of the list.
- However, to `enqueue`, we need to traverse the whole list to find the end : takes $\Theta(n)$ operations.

Example: Suppose we already have a linked list and want to find the end of the list.

```
1 for(Node<E> p = q ; p.next ; p=p.next);
```

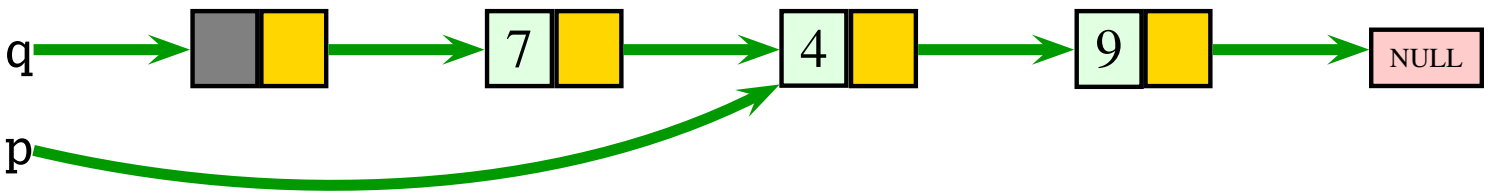


Linked-List Implementation of Queues (3)

- You can implement a queue using a linked list with or without a header "dummy" node.
- Insertions are done at the end of the list and deletions are done at the start of the list.
- However, to `enqueue`, we need to traverse the whole list to find the end : takes $\Theta(n)$ operations.

Example: Suppose we already have a linked list and want to find the end of the list.

```
1 for(Node<E> p = q ; p.next ; p=p.next);
```

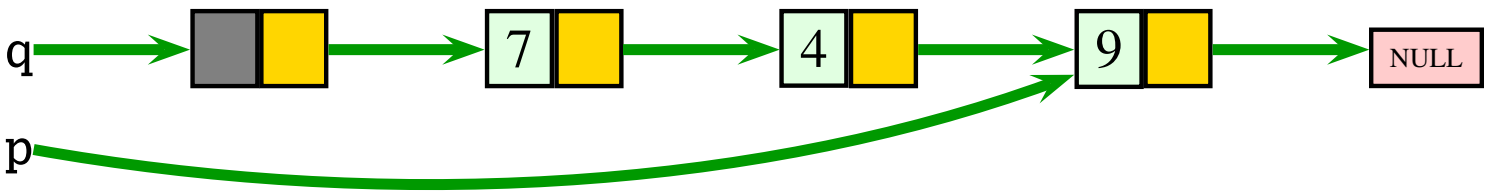


Linked-List Implementation of Queues (4)

- You can implement a queue using a linked list with or without a header "dummy" node.
- Insertions are done at the end of the list and deletions are done at the start of the list.
- However, to `enqueue`, we need to traverse the whole list to find the end : takes $\Theta(n)$ operations.

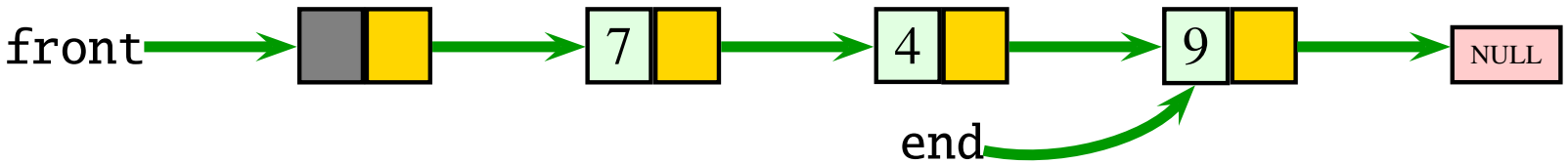
Example: Suppose we already have a linked list and want to find the end of the list.

```
1 for(Node<E> p = q ; p.next ; p=p.next);
```



Linked-List Implementation of Queues (5)

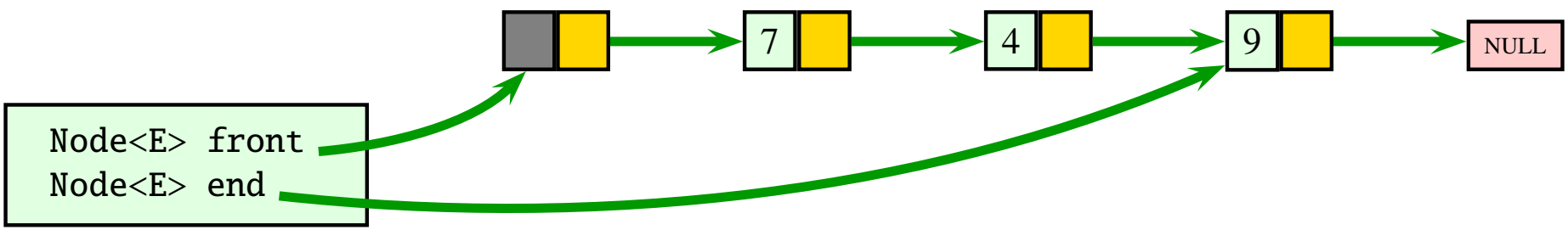
- To speed up the `enqueue` operation, we can add an additional reference to the end of the linked list to make insertions faster.
  - → This does not impact how `dequeue` works
  - → Can add an element to the queue in $\Theta(1)$ time (at the end of the linked list).



Linked-List Implementation of Queues (6)

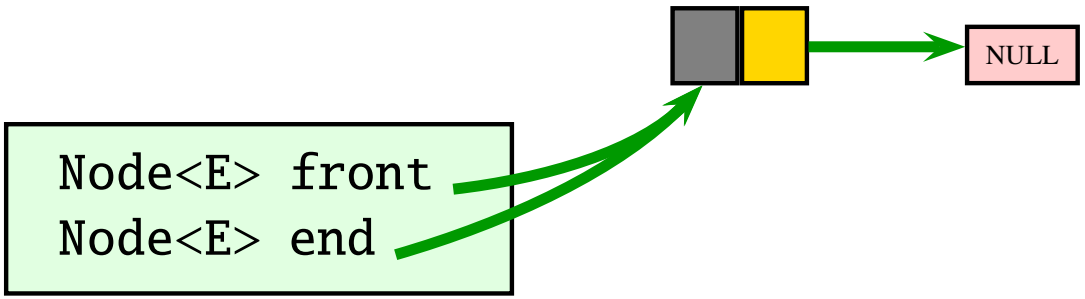
```
1  public class LLQueue<E> implements Queue<E>{
2      private Node<E> front;
3      private Node<E> end;
4
5      public LLQueue(){ front=end= new Node<E>(null,null); }
6      public int getSize(){ ... }
7      public boolean isEmpty(){ ... }
8      public void enqueue(E e){ ... };
9      public E dequeue() throws EmptyQueueException{ ... };
10 }
```

Linked-List Implementation of Queues (7)



Linked-List Implementation of Queues – `constructor`

```
1 public LLQueue(){ front=end= new Node<E>(null,null); }
```



→ Allocate memory for the dummy header node.

→ Let `front` and `end` reference the header.

→ Queue is empty: references `front` and `end` coincide.

Linked-List Implementation of Queues – `isEmpty`

```
1 public boolean isEmpty(){
2     return(front == end);
3 }
```

→ The queue is empty when references `front` and `end` coincide.

→ The queue is never full, i.e., our implementation does not put a limit on the size of the queue (but there is always a practical limit).

Linked-List Implementation of Queues – `getSize`

```
1  public int getSize(){
2      int size=0;
3      Node<E> p=head;
4      while(p.next!=null){  //while p does not point to last node
5          p=p.next;         //advance and increment size
6          size++;
7      }
8      return size;
9  }
```

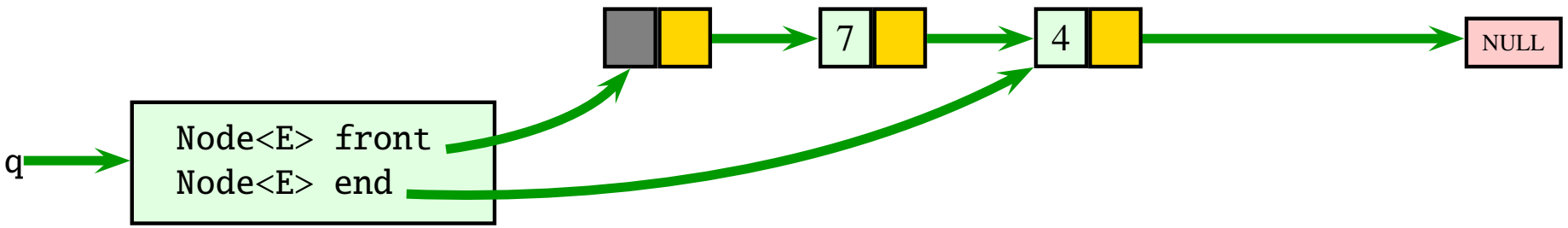
→ Simply traverse the linked list counting how many nodes visited until you fall off the end.

→ This is a relatively expensive operation – $\Theta(n)$

Linked-List Implementation of Queues – enqueue

```
1 public void enqueue(E e){
2     end.next = new Node<E>(e,null);
3     end = end.next;
4 }
```

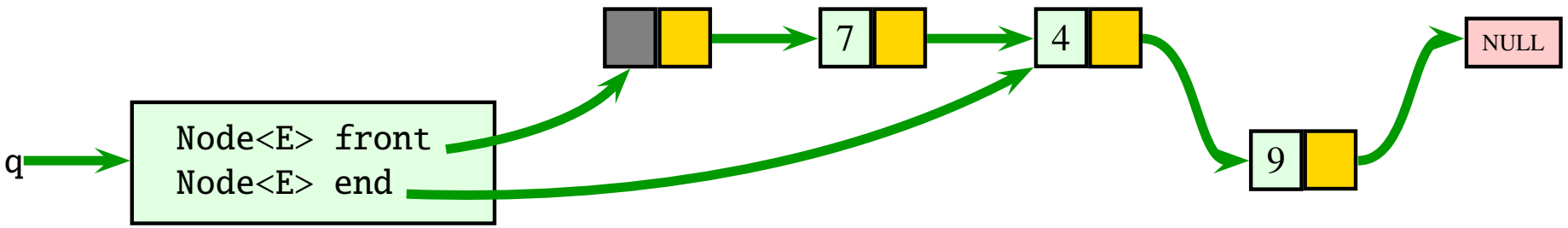
Example: `q.enqueue(new Integer(9));`



Linked-List Implementation of Queues – enqueue

```
1 public void enqueue(E e){
2     end.next = new Node<E>(e,null);
3     end = end.next;
4 }
```

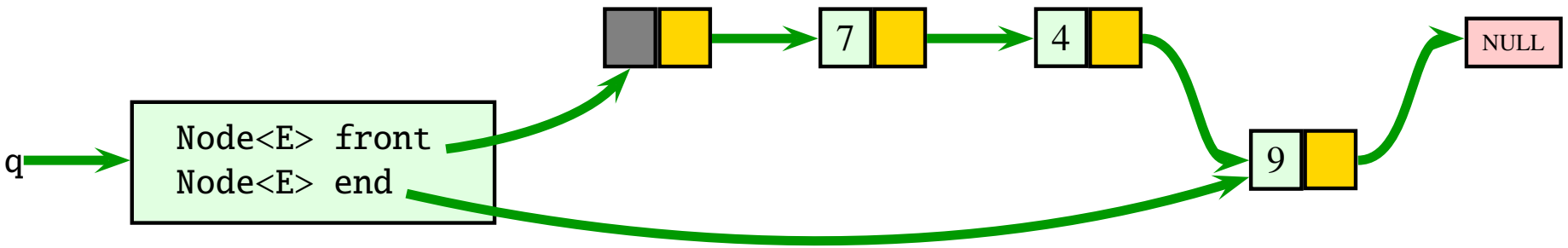
Example: `q.enqueue(new Integer(9));`



Linked-List Implementation of Queues – enqueue

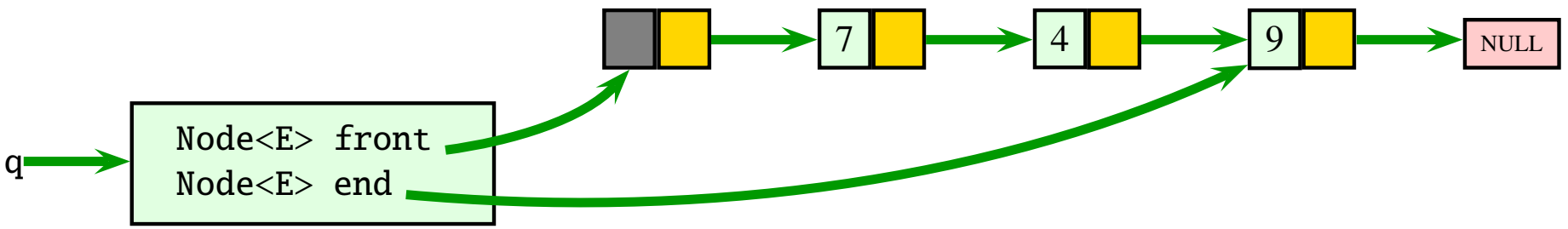
```
1  public void enqueue(E e){
2      end.next = new Node<E>(e,null);
3      end = end.next;
4  }
```

Example: `q.enqueue(new Integer(9));`



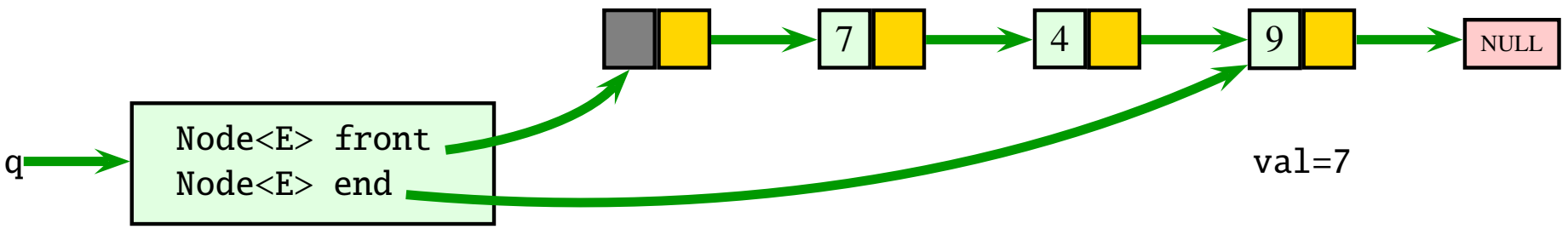
Linked-List Implementation of Queues – dequeue

```
1  public E dequeue() throws EmptyQueueException {
2      if(isEmpty())
3          throw new EmptyQueueException("Queue underflow");
4      else{
5          if(end == front.next)   //Single element in queue
6              end = front;
7          E val = front.next.element;
8          front.next = front.next.next;   //Bypass first node
9          return(val);
10     }
11 }
```



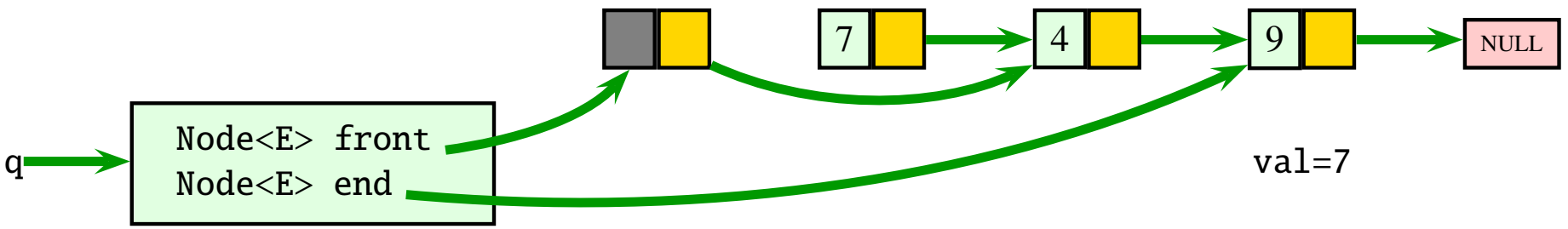
Linked-List Implementation of Queues – dequeue

```
1  public E dequeue() throws EmptyQueueException {
2      if(isEmpty())
3          throw new EmptyQueueException("Queue underflow");
4      else{
5          if(end == front.next)   //Single element in queue
6              end = front;
7          E val = front.next.element;
8          front.next = front.next.next;   //Bypass first node
9          return(val);
10     }
11 }
```



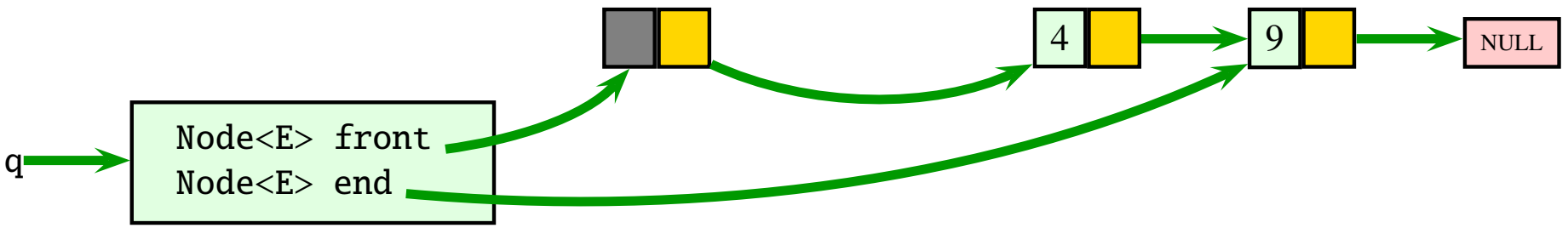
Linked-List Implementation of Queues – dequeue

```
1  public E dequeue() throws EmptyQueueException {
2      if(isEmpty())
3          throw new EmptyQueueException("Queue underflow");
4      else{
5          if(end == front.next)   //Single element in queue
6              end = front;
7          E val = front.next.element;
8          front.next = front.next.next;   //Bypass first node
9          return(val);
10     }
11 }
```



Linked-List Implementation of Queues – dequeue

```
1  public E dequeue() throws EmptyQueueException {
2      if(isEmpty())
3          throw new EmptyQueueException("Queue underflow");
4      else{
5          if(end == front.next)   //Single element in queue
6              end = front;
7          E val = front.next.element;
8          front.next = front.next.next;   //Bypass first node
9          return(val);
10     }
11 }
```



Queue ADT Summary

- **Array Implementation**
  - → Never have overflow in circular queues
  - → However, there is a limit on the number of elements which can be accommodated
  - → Requires little extra overhead to implement the queue

Queue ADT Summary

- **Array Implementation**
  - → Never have overflow in circular queues
  - → However, there is a limit on the number of elements which can be accommodated
  - → Requires little extra overhead to implement the queue
- **Linked-List Implementation**
  - → Can never have a "full" queue, at least not by the design of the implementation
  - → Requires overhead of storing pointers in each node

Queue ADT Summary

- **Array Implementation**
  - → Never have overflow in circular queues
  - → However, there is a limit on the number of elements which can be accommodated
  - → Requires little extra overhead to implement the queue
- **Linked-List Implementation**
  - → Can never have a "full" queue, at least not by the design of the implementation
  - → Requires overhead of storing pointers in each node
- In both cases, insertions and deletions from the queue occur in $\Theta(1)$ operations.

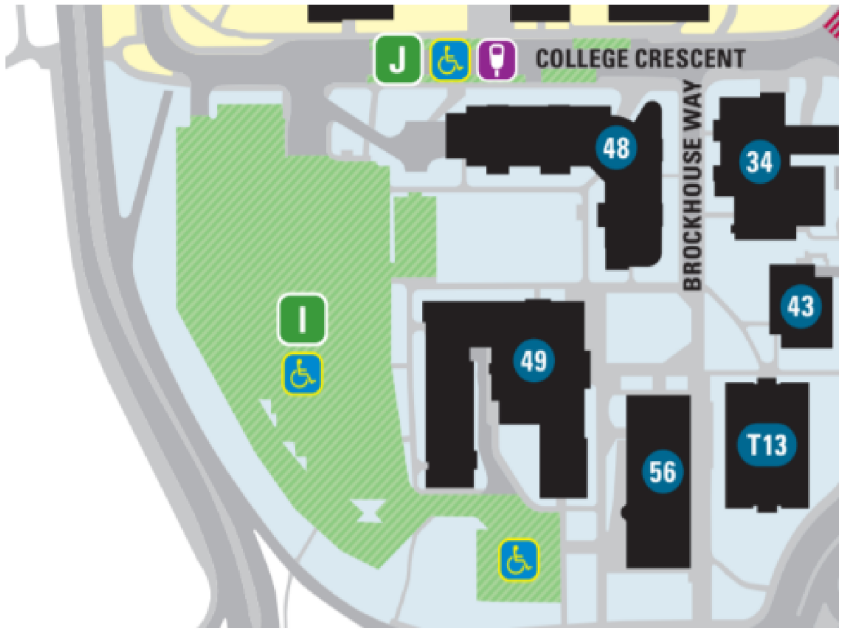
Priority Queues

This is a specific type of queue in which the elements have different **priority**.
Operations:

- `constructor`
- `isEmpty`
- `isFull`
- `enqueueP` - adds an element with a given priority to the queue
- `deQueueP` - returns and removes the **highest priority** element from the queue

Priority Queues – Real Life Example

- McMaster parking waiting list for Lot I
  - Priority
    - Faculy
    - Staff
    - University start date



Wait Lists

Request Date	Status	Type / Location	Position
09/01/[redacted]	Waiting	Lot I	21

McMaster University

Printer Queue

- Simply putting all submitted jobs on a queue might not be a good idea
- Suppose submit a 100 page job and then several 1 page jobs
- Might be reasonable to print small jobs first and then the large job

Priority Queues – Applications

Printer Queue

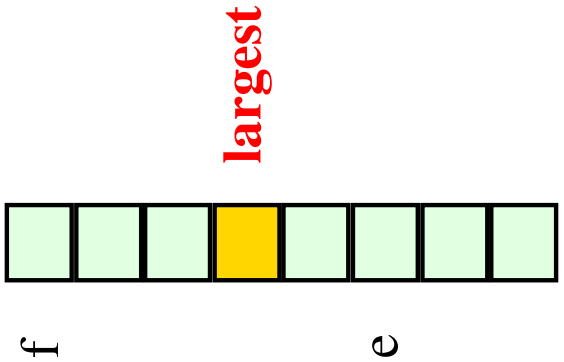
- Simply putting all submitted jobs on a queue might not be a good idea
- Suppose submit a 100 page job and then several 1 page jobs
- Might be reasonable to print small jobs first and then the large job

Multitasking Operating System

- In multitasking OSs, like `UNIX`, a **scheduler** determines when individual processes execute.
- Each process has an assigned priority and placed on in a priority queue
- The scheduler gives high priority processes a greater proportion of processor time
  - For example: The task which handles user input or network traffic may have a higher priority than your email client.

Priority Queue – Implementation

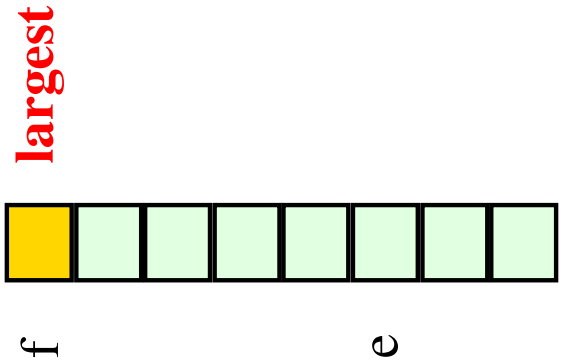
Example: Using an unsorted array



- enqueueP: $\Theta(1)$ (add to end)
- dequeueP: $\Theta(n)$ (to search, delete and then shift elements)

Priority Queue – Implementation

Example: Using a sorted array



- enqueueP: $\Theta(n)$ – need to insert at proper spot in the array
  - $\Theta(\log n)$ to find spot, $\Theta(n)$ to shift elements to make room.
- dequeueP: $\Theta(1)$ – always at the front of the list.

Next Topic ...

In either array or linked-list implementations, there is a trade-off between the performance of **searching** and **insert** (e.g., priority queue).

- In lists we have found that insertions can be easy at the expense of complicated searching.
- If you want good search performance in a list, insertions are difficult
  - e.g., binary search on a sorted list

Next Topic ...

In either array or linked-list implementations, there is a trade-off between the performance of **searching** and **insert** (e.g., priority queue).

- ⇨ In lists we have found that insertions can be easy at the expense of complicated searching.
- ⇨ If you want good search performance in a list, insertions are difficult
  - → e.g., binary search on a sorted list

Next, we will consider a data structure which permits both ...

Trees