


CSE 203: Stacks & Queues



Dr. Mohammed Eunus Ali
Professor
CSE, BUET

PS. Most of the slides are taken from Robert Sedgewick and Kevin Wayne of Princeton University

Abstract Stack

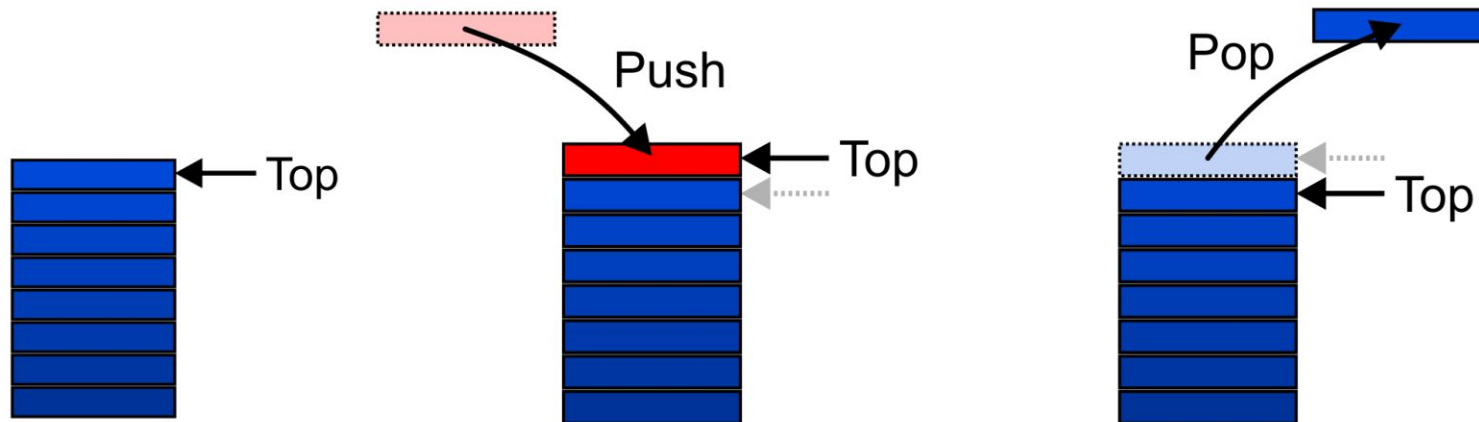
An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:

- Uses an explicit linear ordering
- Insertions and removals are performed individually
- Inserted objects are *pushed onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

Abstract Stack

Also called a *last-in–first-out* (LIFO) behaviour

- Graphically, we may view these operations as follows:



There are two exceptions associated with abstract stacks:

- It is an undefined operation to call either pop or top on an empty stack

Applications

Numerous applications:

- Parsing code:
 - Matching parenthesis
 - XML (e.g., XHTML)
- Tracking function calls
- Dealing with undo/redo operations
- Assembly language

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$

- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

We will look at

- Singly linked lists
- One-ended arrays

Stacks

Stack operations.

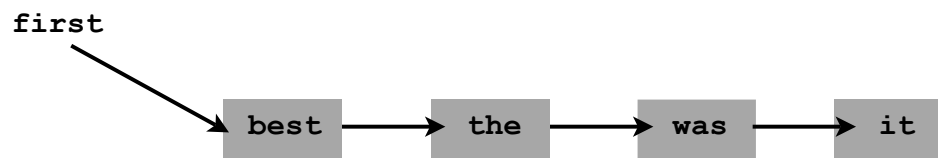
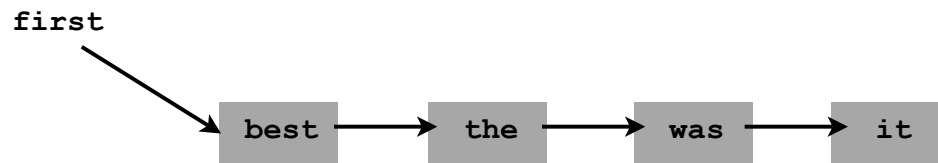
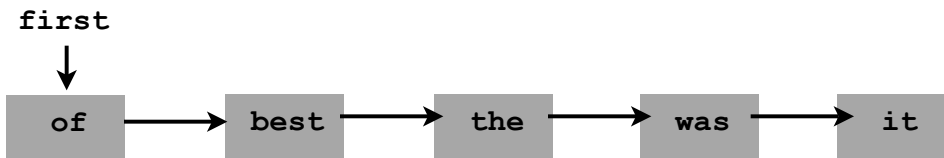
- `push()` **Insert** a new item onto stack.
- `pop()` **Remove** and return the item most recently added.
- `isEmpty()` Is the stack empty?



```
public static void main(String[] args)
{
    StackOfStrings stack = new StackOfStrings();
    while(!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        stack.push(s);
    }
    while(!stack.isEmpty())
    {
        String s = stack.pop();
        StdOut.println(s);
    }
}
```

a sample stack client

Stack pop: Linked-list implementation

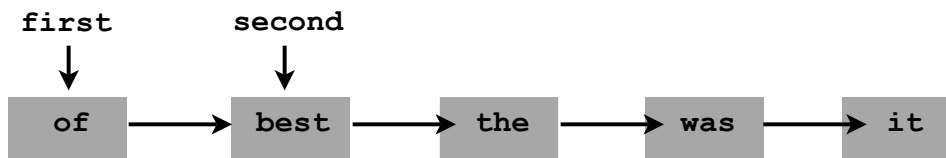
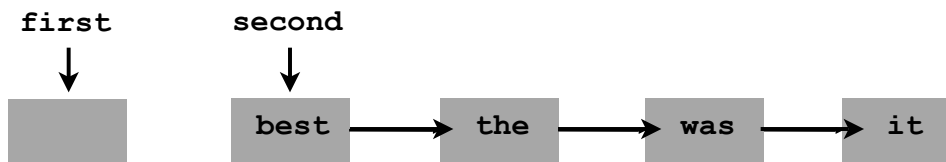
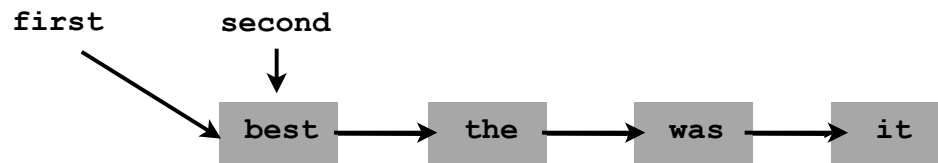
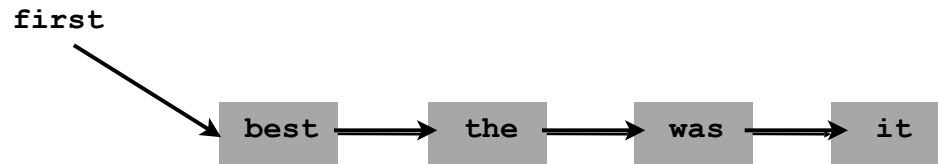


```
item = first.item;
```

```
first = first.next;
```

```
return item;
```

Stack push: Linked-list implementation



```
second = first;
```

```
first = new Node();
```

```
first.item = item;  
first.next = second;
```


Stack: Linked-list implementation

```
public class StackOfStrings
{
    private Node first = null;
    private class Node
    {
        String item;
        Node next;
    }
    public boolean isEmpty()
    { return first == null; }
    public void push(String item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }
    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

← "inner class"

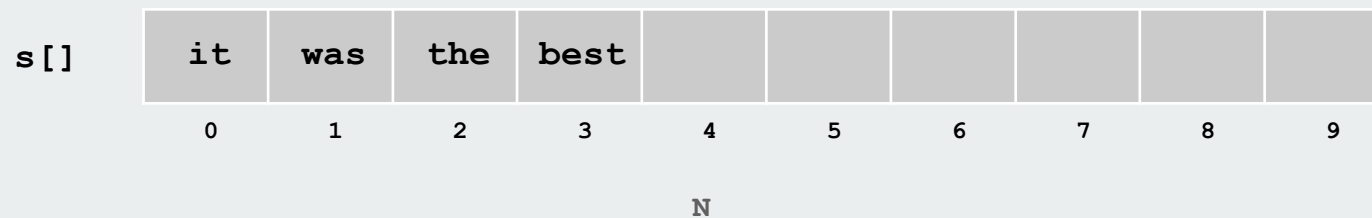
Error conditions?

Example: pop() an empty stack

Stack: Array implementation

Array implementation of a stack.

- Use array `s[]` to store N items on stack.
- `push()` add new item at `s[N]`.
- `pop()` remove item from `s[N-1]`.



Stack: Array implementation

```
public class StackOfStrings
{
    private String[] s;
    private int N = 0;

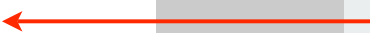
    public StringStack(int capacity)
    { s = new String[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(String item)
    { s[N++] = item; }

    public String pop()
    {
        String item = s[N-1];
        s[N-1] = null;
        N--;
        return item;
    }
}
```

avoid **loitering**
(garbage collector only reclaims memory
if no outstanding references)



Stack array implementation: Dynamic resizing

Q. How to grow array when capacity reached?

Q. How to shrink array (else it stays big even when stack is small)?

First try:

- `push()`: increase size of `s[]` by 1
- `pop()`: decrease size of `s[]` by 1

Too expensive

- Need to copy all of the elements to a new array.
- Inserting N elements: time proportional to $1 + 2 + \dots + N \approx \frac{N^2}{2}$.
↑
infeasible for large N

Need to **guarantee** that array resizing happens **infrequently**

Stack array implementation: Dynamic resizing

Q. How to grow array?

A. Use **repeated doubling**:

if array is full, create a new array of twice the size, and copy items

no-argument
constructor

```
public StackOfStrings()
```

```
public void push(String item)
{
```

```
    s[N++] = item;
```

```
}
```

```
private void resize(int max)
```

```
{
```

```
    String[] dup = new String[max];
```

```
    for (int i = 0; i < N; i++)
```

```
        dup[i] = s[i];
```

```
    s = dup;
```

```
}
```

create new array
copy items to it

Consequence.
 N^2).

Inserting N items takes time proportional to N (not



$$8 + 16 + \dots + N/4 + N/2 + N \approx 2N$$

Stack array implementation: Dynamic resizing

Q. How (and when) to shrink array?

How: create a new array of **half** the size, and copy items.

When (first try): array is half full?

No, causes **thrashing**

← (push-pop-push-pop-... sequence: time proportional to N for each op)

When (solution): array is 1/4 full (then new array is half full).

```
public String pop(String item)
{
    String item = s[--N];
    sa[N] = null;
    if (N == s.length/4)
        resize(s.length/2);
    return item;
}
```

Not a.length/2
to avoid thrashing

Consequences.

- any sequence of N ops takes time proportional to N
- array is always between 25% and 100% full

Stack Implementations: Array vs. Linked List

Stack implementation tradeoffs. Can implement with either array or linked list, and client can use interchangeably. Which is better?

Array.

- Most operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of N operations (starting from empty stack) takes time proportional to N .

Linked list.

- Grows and shrinks gracefully.
- Every operation takes constant time.
- Every operation uses extra space and time to deal with references.

← "amortized" bound

Bottom line: tossup for stacks

but differences are significant when other operations are added

Stack implementations: Array vs. Linked list

Which implementation is more convenient?

array?

linked list?

return count of elements in stack
remove the kth most recently added

sample a random element

Abstract Queue

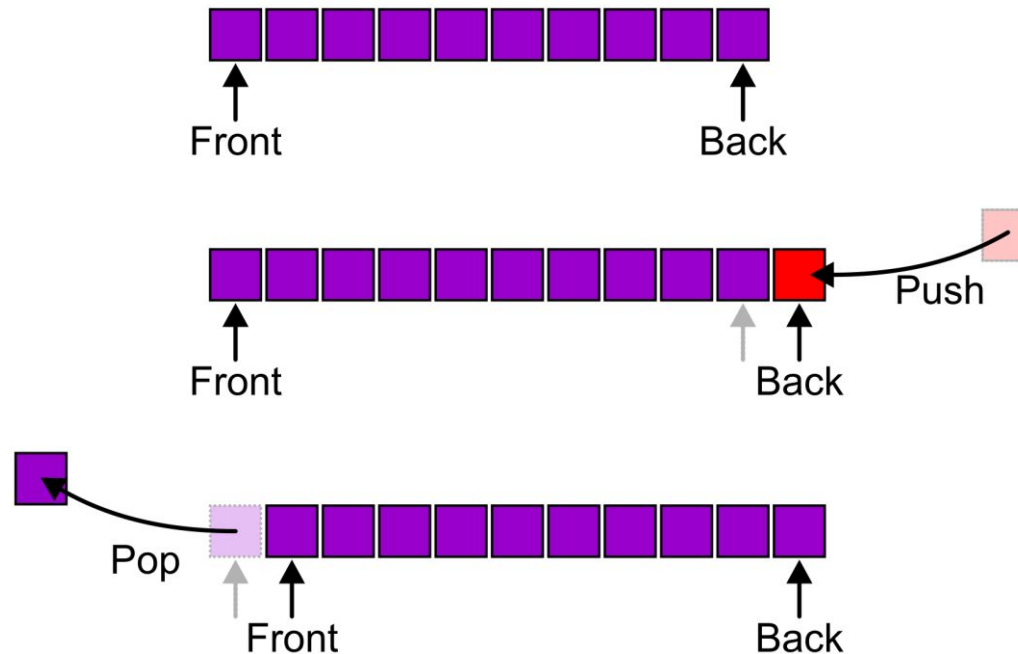
An Abstract Queue (Queue ADT) is an abstract data type that emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- There are no restrictions on objects inserted into (*pushed onto*) the queue—that object is designated the back of the queue
- The object designated as the *front* of the queue is the object which was in the queue the longest
- The remove operation (*popping* from the queue) removes the current *front* of the queue

Abstract Queue

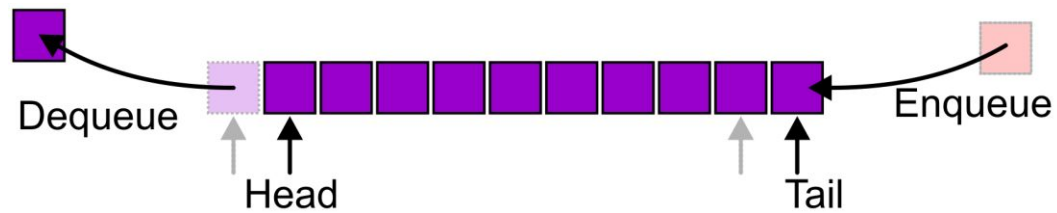
Also called a *first-in–first-out* (FIFO) data structure

- Graphically, we may view these operations as follows:



Abstract Queue

Alternative terms may be used for the four operations on a queue, including:



Abstract Queue

There are two exceptions associated with this abstract data structure:

- It is an undefined operation to call either pop or front on an empty queue

Applications

The most common application is in client-server models

- Multiple clients may be requesting services from one or more servers
- Some clients may have to wait while the servers are busy
- Those clients are placed in a queue and serviced in the order of arrival

Grocery stores, banks, and airport security use queues

The SSH Secure Shell and SFTP are clients

Most shared computer services are servers:

- Web, file, ftp, database, mail, printers, WOW, *etc.*

Implementations

We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

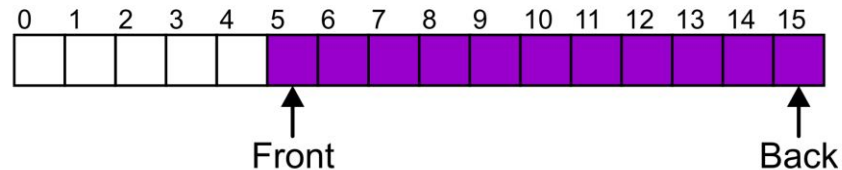
Requirements:

- All queue operations must run in $\Theta(1)$ time

Arrays

Suppose that:

- The array capacity is 16
- We have performed 16 pushes
- We have performed 5 pops
 - The queue size is now 11



- We perform one further push

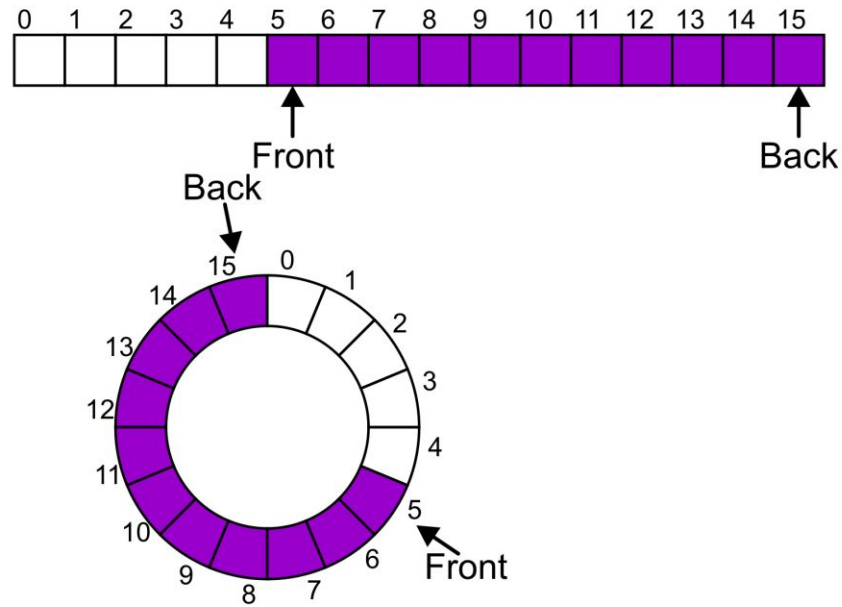
In this case, the array is not full and yet we cannot place any more objects in to the array

Circular Arrays

Instead of viewing the array on the range 0, ..., 15, consider the indices being cyclic:

..., 15, 0, 1, ..., 15, 0, 1, ..., 15, 0, 1, ...

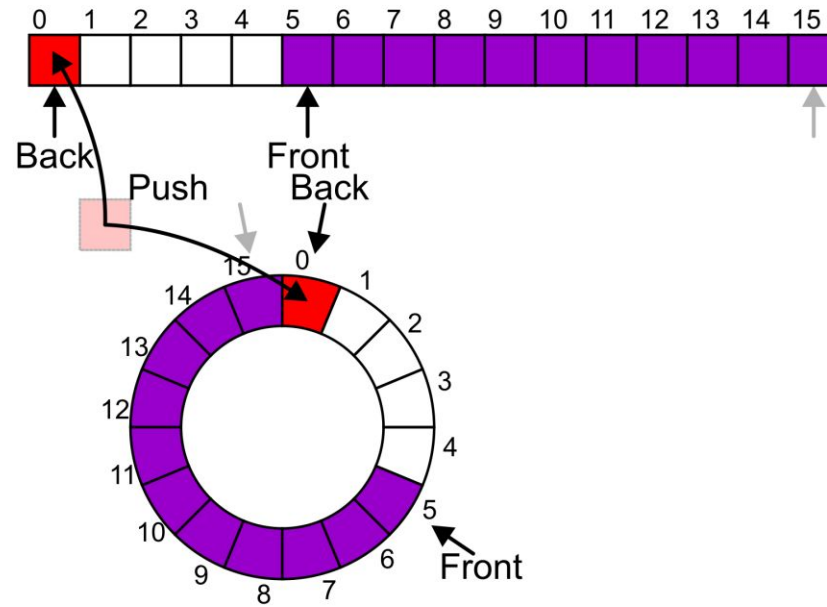
This is referred to as a *circular array*



Circular Array

Now, the next push may be performed in the next available location of the circular array:

```
++iback;  
if ( iback == capacity() ) {  
    iback = 0;  
}
```



Queues

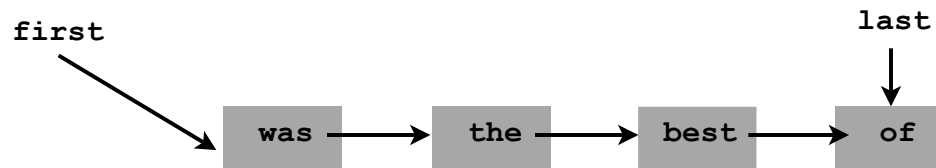
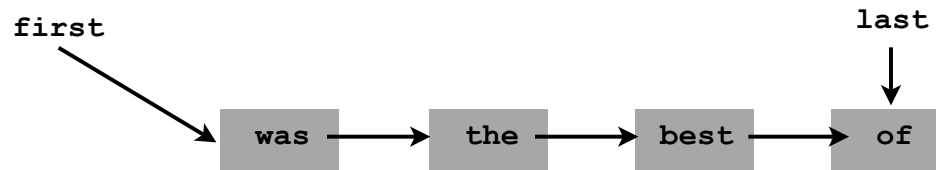
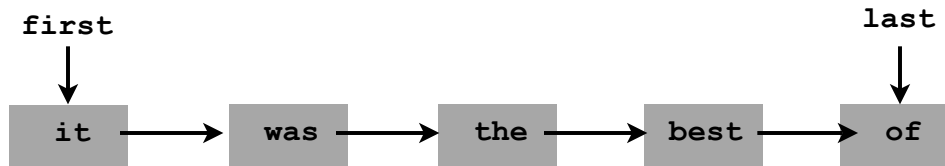
Queue operations.

- `enqueue()` Insert a new item onto queue.
- `dequeue()` Delete and return the item least recently added.
- `isEmpty()` Is the queue empty?

```
public static void main(String[] args)
{
    QueueOfStrings q = new QueueOfStrings();
    q.enqueue("Vertigo");
    q.enqueue("Just Lose It");
    q.enqueue("Pieces of Me");
    q.enqueue("Pieces of Me");
    System.out.println(q.dequeue());
    q.enqueue("Drop It Like It's Hot");
    while(!q.isEmpty())
        System.out.println(q.dequeue());
}
```



Deque: Linked List Implementation



```
item = first.item;
```

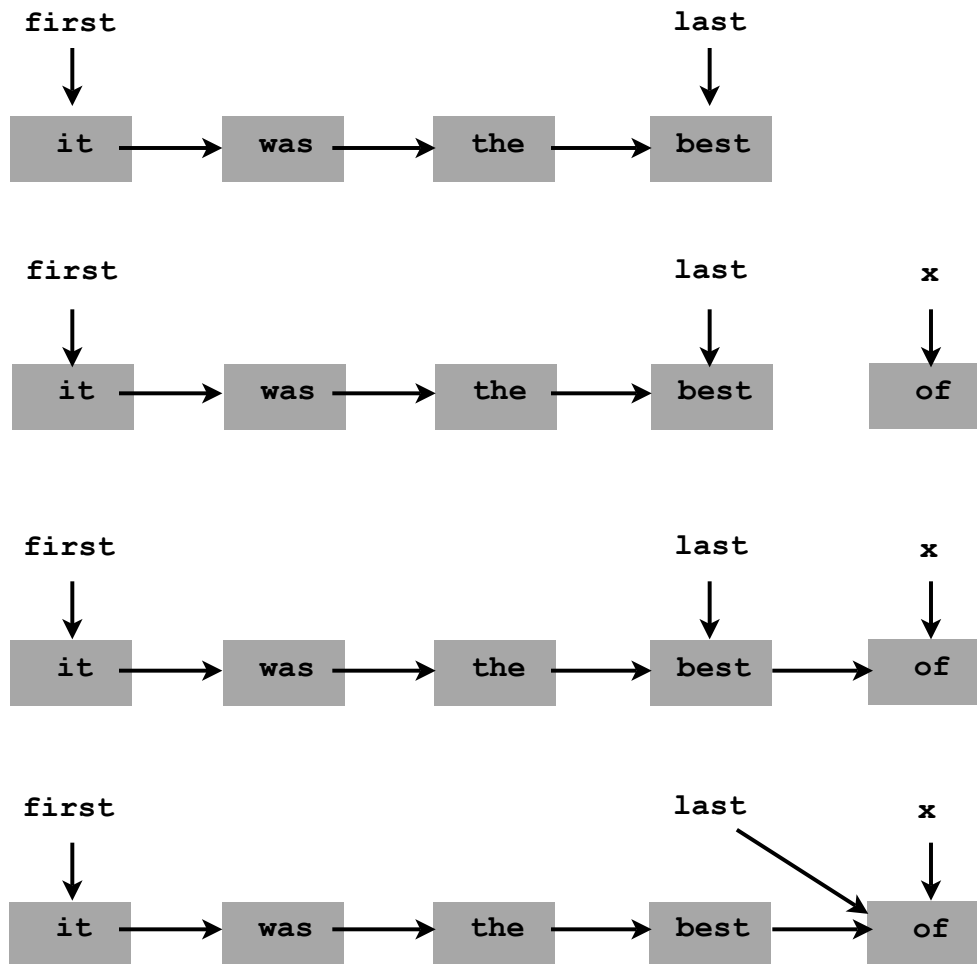
```
first = first.next;
```

```
return item;
```

Aside:

enqueue (pronounced "DQ") means "remove from a queue"

Enqueue: Linked List Implementation



```
x = new Node();  
x.item = item;  
x.next = null;
```

```
last.next = x;
```

```
last = x;
```

Queue: Linked List Implementation

```
public class QueueOfStrings
{
    private Node first;
    private Node last;

    private class Node
    { String item; Node next; }

    public boolean isEmpty()
    { return first == null; }

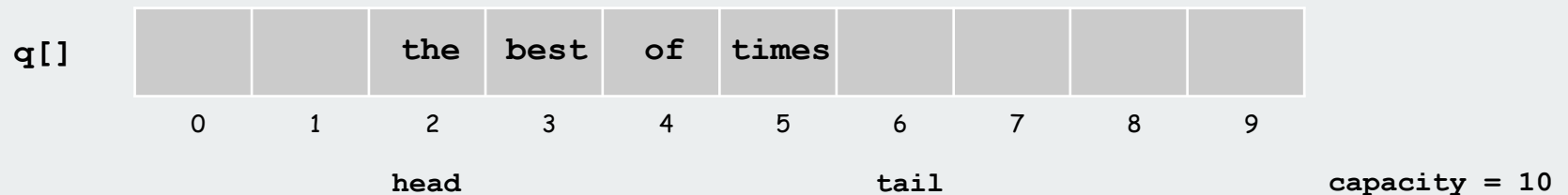
    public void enqueue(String item)
    {
        Node x = new Node();
        x.item = item;
        x.next = null;
        if (isEmpty()) { first = x; last = x; }
        else { last.next = x; last = x; }
    }

    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

Queue: Array implementation

Array implementation of a queue.

- Use array `q[]` to store items on queue.
- `enqueue()`: add new object at `q[tail]`.
- `dequeue()`: remove object from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.



[details: good exercise or exam question]

- › stacks
- › dynamic resizing
- › queues
- › **generics**
- › applications

Generics (parameterized data types)

We implemented: `StackOfStrings`, `QueueOfStrings`.

We also want: `StackOfURLs`, `QueueOfCustomers`, etc?

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and **error-prone**.
- Maintaining cut-and-pasted code is tedious and **error-prone**.

Stack of Objects

We implemented: StackOfStrings,

QueueOfStrings. We also want: StackOfURLs,

QueueOfCustomers, etc?

Attempt 2. Implement a stack with items of type object.

- Casting is required in client.
- Casting is error-prone: **run-time error** if types mismatch.


```
Stack    s = new Stack();  
Apple    a = new Apple();  
Orange   b = new Orange();  
s.push(a);  
s.push(b);  
a = (Apple) (s.pop());
```

run-time error

Generics

Generics. Parameterize stack by a single type.

- Avoid casting in both client and implementation.
- Discover type mismatch errors at **compile-time** instead of run-time.



```
Stack<Apple> s = new Stack<Apple>();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = s.pop();
```

compile-time error

no cast needed in client

Guiding principles.

- Welcome compile-time errors
- Avoid run-time errors

Why?

Generic Stack: Linked List Implementation

```
public class StackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

return item;

```
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(Item item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

Generic type name



Generic stack: array implementation

The way it should be.

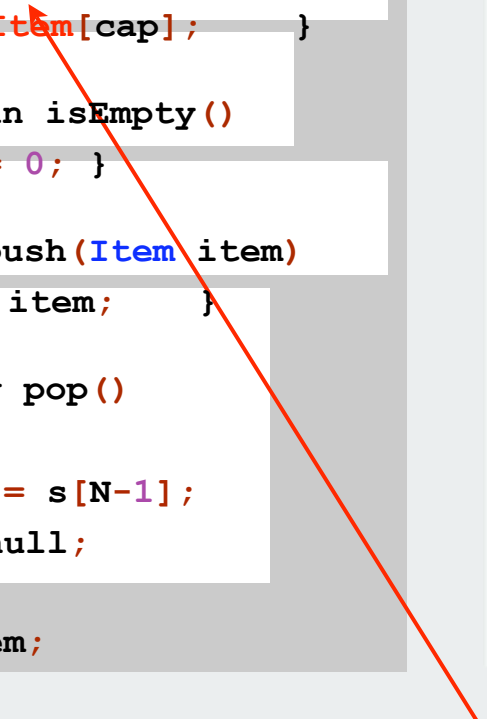
```
public class Stack<Item>
{
    private Item[] s;
    private int N = 0;

    public Stack(int cap)
    { s = new Item[cap]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(Item item)
    { s[N++] = item; }

    public String pop()
    {
        Item item = s[N-1];
        s[N-1] = null;
        N--;
        return item;
    }
}
```



```
public class StackOfStrings
{
    private String[] s;
    private int N = 0;

    public StackOfStrings(int cap)
    { s = new String[cap]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(String item)
    { s[N++] = item; }

    public String pop()
    {
        String item = s[N-1];
        s[N-1] = null;
        N--;
        return item;
    }
}
```

Generic stack: array implementation

The way it is: an **ugly cast** in the implementation.

```
public class Stack<Item>
{
    private Item[] s;
    private int N = 0;
    public Stack(int cap)
    { s = (Item[]) new Object[cap]; }
    public boolean isEmpty()
    { return N == 0; }
    public void push(Item item)
    { s[N++] = item; }
    public String pop()
    {
        Item item = s[N-1];
        s[N-1] = null;
        N--;
        return item;
    }
}
```

← the ugly cast

Number of casts in good code: 0

Generic data types: autoboxing

Generic stack implementation is object-based.

What to do about primitive types?

Wrapper type.

- Each primitive type has a **wrapper** object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```
Stack<Integer> s = new Stack<Integer>();  
s.push(17); // s.push(new Integer(17));  
int a = s.pop(); // int a = ((int) s.pop()).intValue();
```

Bottom line: Client code can use generic stack for **any** type of data

Stack Applications

Real world applications.

- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- Implementing function calls in a compiler.

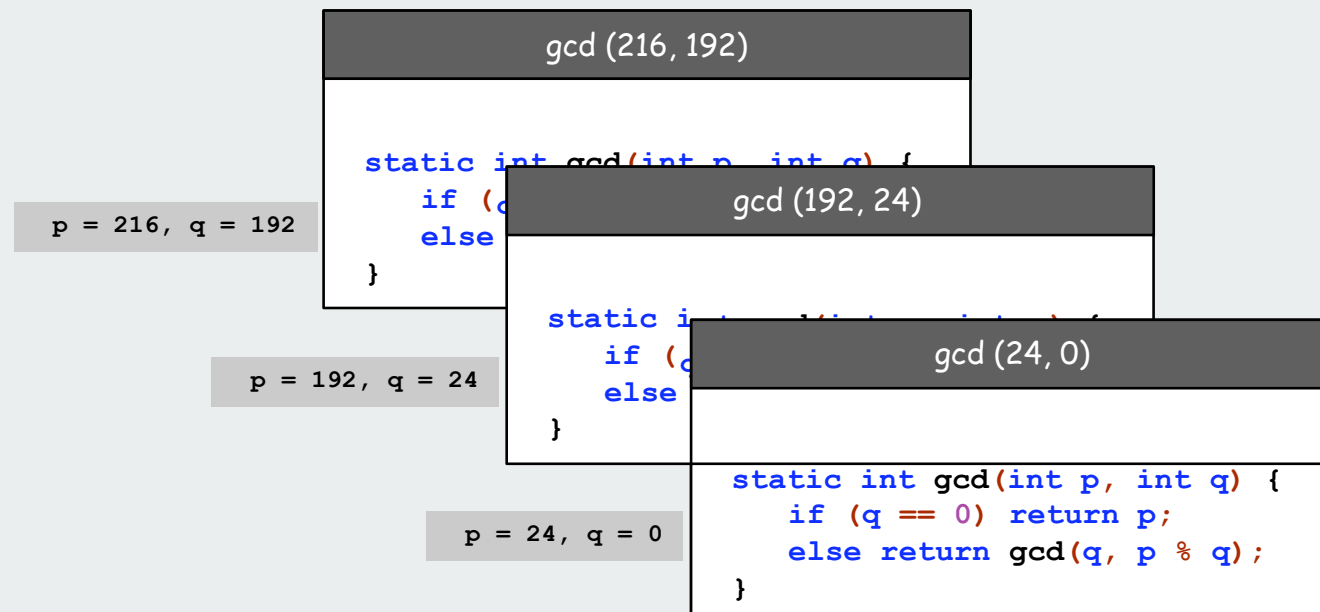
Function Calls

How a compiler implements functions.

- Function call: **push** local environment and return address.
- Return: **pop** return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



Arithmetic Expression Evaluation

Goal. Evaluate infix expressions

(1 + ((2 + 3) * (4 * 5)))

operand operator

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

value stack
operator stack

	(1 + ((2 + 3) * (4 * 5)))
1	+ ((2 + 3) * (4 * 5))
1 +	((2 + 3) * (4 * 5))
1 2 +	+ 3) * (4 * 5))
1 2 + +	3) * (4 * 5))
1 2 3 + +) * (4 * 5))
1 5 +	* (4 * 5))
1 5 + *	(4 * 5))
1 5 4 + *	* 5))
1 5 4 + * *	5))
1 5 4 5 + * *))
1 5 20 + *))
1 100 +)
101	

Arithmetic Expression Evaluation

```
public class Evaluate {
    public static void main(String[] args) {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();

            if (s.equals("(")) ;
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals(")") {
                String op = ops.pop();
                if (op.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Why correct?

When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

$$(1 + ((2 + 3) * (4 * 5)))$$

as if the original input were:

$$(1 + (5 * (4 * 5)))$$

Repeating the argument:

$$\begin{aligned} &(1 + (5 * 20)) \\ &(1 + 100) \\ &101 \end{aligned}$$

Extensions. More ops, precedence order, associativity.

$$1 + (2 - 3 - 4) * 5 * \text{sqrt}(6 + 7)$$

Stack-based programming languages

Observation 1.

Remarkably, the 2-stack algorithm computes the same value if the operator occurs **after** the two values.

(1 ((2 3 +) (4 5 *) *) +)

Observation 2.

All of the parentheses are redundant!

1 2 3 + 4 5 * * +

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, ...

Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

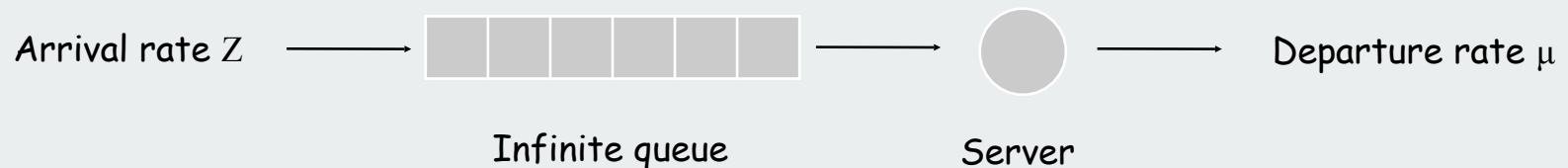
Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

M/D/1 queuing model

M/D/1 queue.

- Customers are serviced at fixed rate of μ per minute.
- Customers arrive according to Poisson process at rate of Z per minute.



- Q. What is average wait time W of a customer?
- Q. What is average number of customers L in system?

M/D/1 queuing model: event-based simulation

```
public class MD1Queue
{
    public static void main(String[] args)
    {
        double lambda = Double.parseDouble(args[0]);    // arrival rate
        double mu      = Double.parseDouble(args[1]);    // service rate
        Histogram hist = new Histogram(60);
        Queue<Double> q = new Queue<Double>();
        double nextArrival = StdRandom.exp(lambda);
        double nextService = 1/mu;
        while (true)
        {
            while (nextArrival < nextService)
            {
                q.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }
            double wait = nextService - q.dequeue();
            hist.addDataPoint(Math.min(60, (int) (wait)));
            if (!q.isEmpty())
                nextService = nextArrival + 1/mu;
            else
                nextService = nextService + 1/mu;
        }
    }
}
```