Stacks and Queues

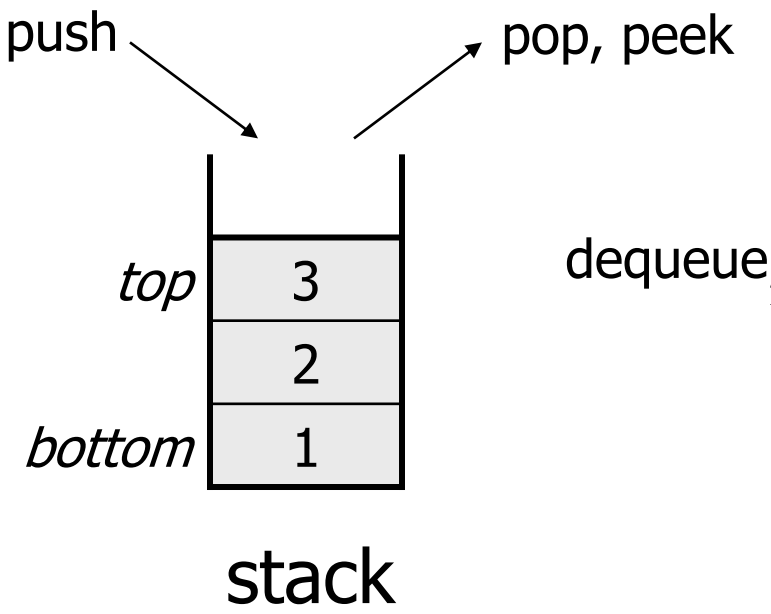
Abstract Data Types (ADTs)

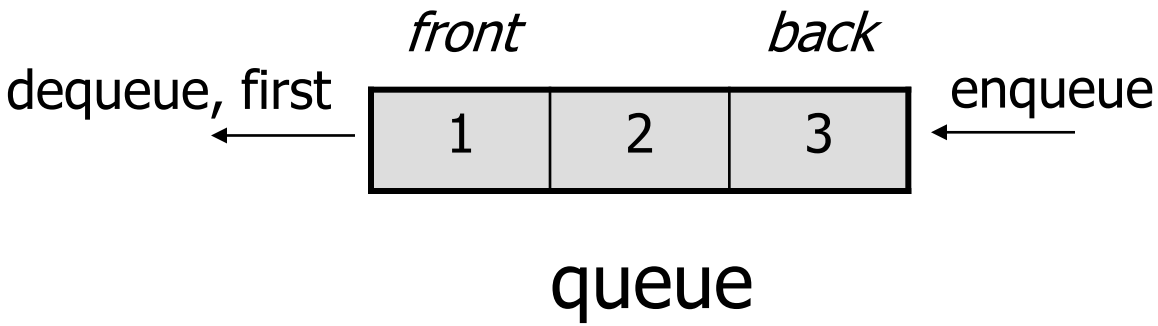
- **Abstract Data Type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it

- Consumers of a stack or queue do not need to know how they are implemented.
  - We just need to understand the idea of the collection and what operations it can perform.
  - Stacks usually implemented with arrays
  - Queues often implemented with a linked list

Stacks and Queues

- Some collections are constrained so clients can only use optimized operations
  - **stack**: retrieves elements in reverse order as added
  - **queue**: retrieves elements in same order as added

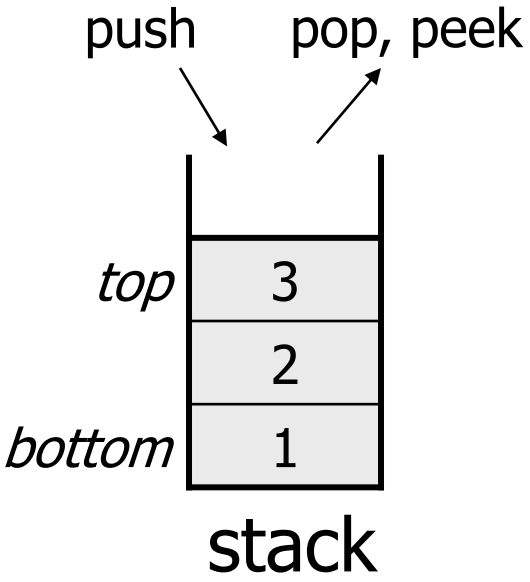




Stacks

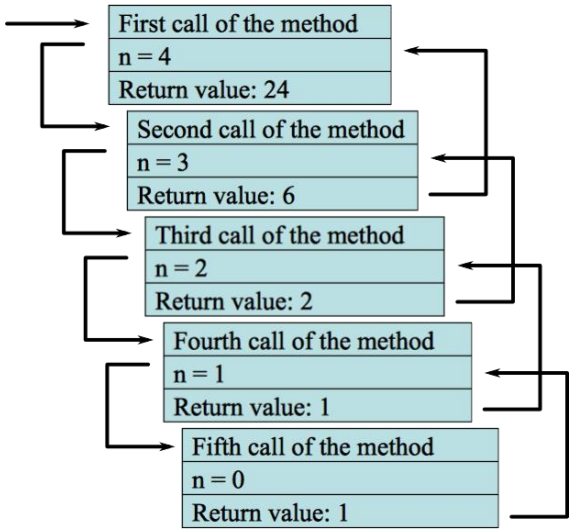
- **Stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").

- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



Stacks in Computer Science

- Programming languages and compilers:
  - method calls are placed onto a stack
    - *call=push*
    - *return=pop*
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations



Stack Limitations

- You cannot loop over a stack in the usual way. Assume we created our own `Stack` class.

```
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, pop each element until the stack is empty.

```
// process (and destroy) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

What Happened to My Stack?

- Suppose we're asked to write a method `max` that accepts a `Stack` of `Integer`s and returns the largest `Integer` in the stack:

```
// Precondition: !s.isEmpty()
public static void max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

  - The algorithm is correct, but what is wrong with the code?

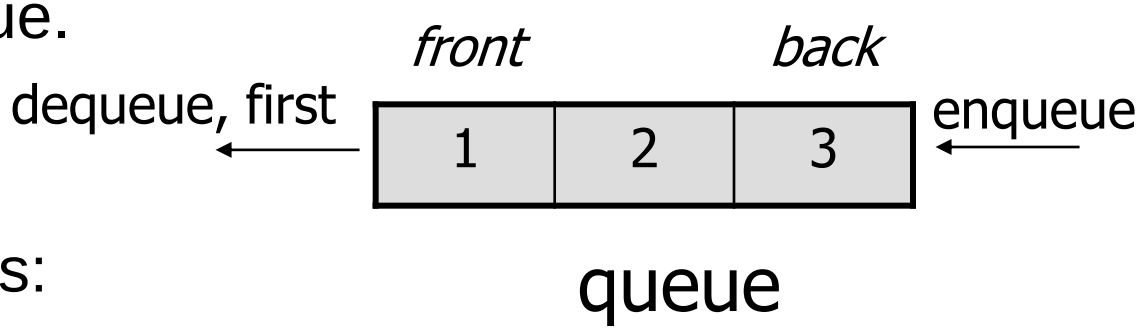
What Happened to My Stack?

- The code destroys the stack in figuring out its answer.
  - To fix this, you must save and restore the stack's contents:

```
public static void max(Stack<Integer> s) {
    Stack<Integer> backup = new Stack<Integer>();
    int maxValue = s.pop();
    backup.push(maxValue);
    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = Math.max(maxValue, next);
    }
    while (!backup.isEmpty()) {    // restore
        s.push(backup.pop());
    }
    return maxValue;
}
```

Queues

- **Queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

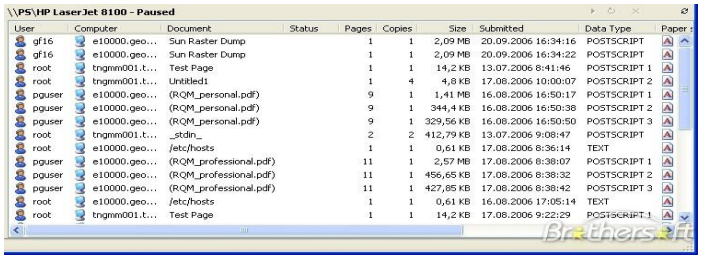


queue

- Basic queue operations:
  - **enqueue** (add): Add an element to the back.
  - **dequeue** (remove): Remove the front element.
  - **first**: Examine the front element.

Queues in Computer Science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send



- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Using Queues

- As with stacks, must pull contents out of queue to view them. Assume we created our own `Queue` class.

```
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.dequeue();
}
```

  - To examine each element exactly once.

```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.dequeue();
    (including possibly re-adding it to the queue)
}
```

    - Why do we need the `size` variable?

Mixing Stacks and Queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<Integer> q = new Queue<Integer>();
q.enqueue(1);
q.enqueue(2);
q.enqueue(3);                          // [1, 2, 3]

Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) {                 // Q -> S
    s.push(q.dequeue());
}
while (!s.isEmpty()) {                 // S -> Q
    q.enqueue (s.pop());
}
System.out.println(q);                 // [3, 2, 1]
```

Java `Stack` Class

`Stack<`**T**`>()`	constructs a new stack with elements of type **T**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<String> s = new Stack<String>();
s.push("a");
s.push("b");
s.push("c");                    // bottom ["a", "b", "c"] top
System.out.println(s.pop()); // "c"
```

- `Stack` has other methods that are off-limits (not efficient)

Java `Queue` Interface

`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);          // front [42, -3, 17] back
System.out.println(q.remove());    // 42
```

- **IMPORTANT**: When constructing a queue, you must use a new `LinkedList` object instead of a new `Queue` object.
  - Because `Queue` is an *interface*.

Exercises

- Write a method `stutter` that accepts a queue of Integers as a parameter and replaces every element of the queue with two copies of that element.
  - `[1, 2, 3]` becomes `[1, 1, 2, 2, 3, 3]`

- Write a method `mirror` that accepts a queue of Strings as a parameter and appends the queue's contents to itself in reverse order.
  - `[a, b, c]` becomes `[a, b, c, c, b, a]`