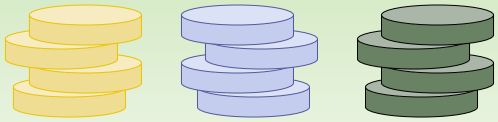
Stacks (and Queues)



Stacks

- Stack: what is it?
- Applications
- Implementations

Stacks and queues

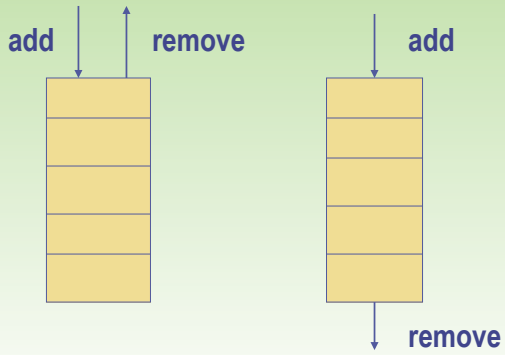
- A **stack** is a very important data structure in computing science.
- A stack is a sequence of elements to which new elements are added to the top of the stack (**push**ed), and from which elements are removed from the top of the stack (**pop**ped).
  - A simple analogy is a stack of plates – you put new plates on top of the current stack and take plates of the top
  - The first plate you put down will be the last plate you pick up as you empty the plate stack

Stacks and queues

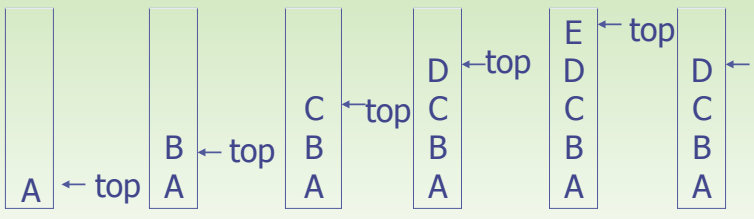
- Stack are sometimes referred to as a **last in, first out** structure (**LIFO**).
- In a **queue**, elements are removed from the opposite end to which they are added.
- Queue sometimes referred to as a **first in, first out** structure (**FIFO**).

Stacks & Queues

- The difference is the remove operation.



Last In First Out



The Stack

- Main stack operations:
  - push(*object*): inserts an element
  - *object* pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - *object* peek() or *object* top() returns the last inserted element without removing it.
  - *integer* size(): returns the number of elements stored
  - *boolean* isEmpty(): indicates whether no elements are stored

Stack Operations

Assume a simple stack of Integer objects.
IntegerStack stack = new IntegerStack();

```
stack.push(12);
stack.push(4);
stack.push( stack.top() + 2 );
stack.pop();
stack.push(stack.top() );
//what are contents of stack?
```

Applications of Stacks

- Direct applications
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

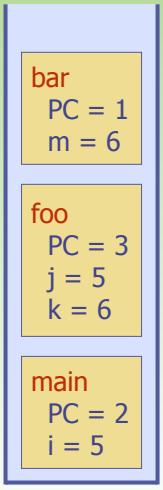
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }


bar(int m) {
  …
  }
```



Implementation: Array Based Stack

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- Last index in the array is the *top*
  - What is index when stack is empty?
- Increment *top* when one element is pushed, decrement after pop

Array-based Stack

```
public int Size() { return top+1; }

public Integer pop() {
  if (isEmpty()) return null;

  Integer onTop = stack[top];
  top = top - 1;

  return onTop;
}
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then return false
  - Limitation of the array-based implementation

```
public boolean push(Integer val)
  {
    if (top+1 == stack.length)
    {
      return false;
    }

    top = top + 1;
    stack[top] = new Integer(val);

    return true;
  }
```

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific problem

Common Stack Error

```
import java.util.Stack;
...

Stack<Integer> s = new Stack<Integer>();

// Put stuff in stack
for(int i = 0; i < 7; i++)
  s.push(i);

// print out contents of stack while emptying it
for(int i = 0; i < s.size(); i++)
  System.out.println(s.pop());

// Output? Why?
```


Corrected Version

```
import java.util.Stack;
...

Stack<Integer> s = new Stack<Integer>();

// Put stuff in stack
for(int i = 0; i < 7; i++)
  s.push(i);

// print out contents of stack while emptying it
int limit = s.size();

for(int i = 0; i < limit; i++)
  System.out.println( s.pop() );

// or

while( !s.isEmpty() )
  System.out.println( s.pop() );
```

Example: Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: (a)((b)){([(c)])}
  - correct: ((a)((b)){([(c)])})
  - incorrect:)((a)){([(b)])}
  - incorrect: ({[a])}
  - incorrect: (a

Parentheses Matching Algorithm

Algorithm parenthMatch(*X*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match

```
Let S be an empty stack
Let n be the length of X
for i=0; i<n; i++ do
   if X[i] is an opening grouping symbol then
        S.push(X[i])
   if X[i] is a closing grouping symbol then
        if S.isEmpty() then
               return false {nothing to match with}
        if S.pop() does not match the type of X[i] then
               return false {wrong type}

return S.isEmpty() {list will be empty if every symbol matched
   or not if some symbols were never matched (true vs false)}
```

Example: HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```