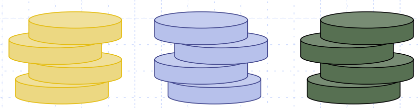
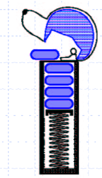
Stacks



Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

The Stack ADT (§4.2)



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - 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 Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack {

  public int size();

  public boolean isEmpty();

  public Object top()
      throws EmptyStackException;

  public void push(Object o);

  public Object pop()
      throws EmptyStackException;
}
```

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - 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

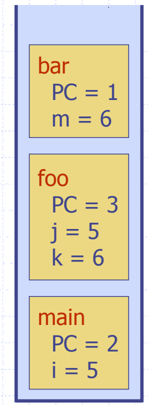
Method Stack in the JVM

- 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
- Allows for **recursion**

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

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

bar(int m) {
  …
}
```

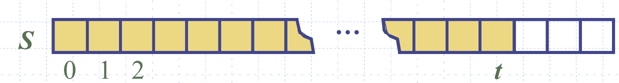


Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

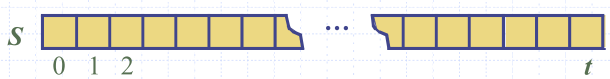
Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm ***push***(*o*)
  if $t = S.length - 1$ **then**
    throw *FullStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



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 exception

Array-based Stack in Java

```
public class ArrayStack
    implements Stack {

  // holds the stack elements
  private Object S[ ];

  // index to top element
  private int top = -1;

  // constructor
  public ArrayStack(int capacity) {
     S = new Object[capacity]);
  }
```

```
public Object pop()
    throws EmptyStackException {
  if isEmpty()
    throw new EmptyStackException
      ("Empty stack: cannot pop");
  Object temp = S[top];
  // facilitates garbage collection
  S[top] = null;
  top = top – 1;
  return temp;
}
```


Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "["
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )]))}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
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
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else 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}
if S.isEmpty() then
    return true {every symbol matched}
else
    return false {some symbols were never matched}
```


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>
```

The Little Boat

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.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm

- Is similar to parentheses matching:

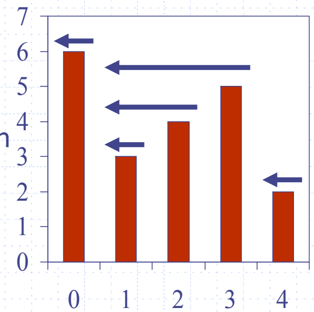
```
import java.util.StringTokenizer;
import datastructures.Stack;
import datastructures.NodeStack;
import java.io.*;
/** Simpli.ed test of matching tags in an HTML document. */
public class HTML { /** Nested class to store simple HTML tags */
    public static class Tag { String name; // The name of this tag
        boolean opening; // Is true i. this is an opening tag
        public Tag() { // Default constructor
            name = "";
            opening = false;
        }
        public Tag(String nm, boolean op) { // Preferred constructor
        name = nm;
        opening = op;
        }
        /** Is this an opening tag? */
        public boolean isOpening() { return opening; }
        /** Return the name of this tag */
        public String getName() {return name; }
    }
    /** Test if every opening tag has a matching closing tag. */
    public boolean isHTMLMatched(Tag[ ] tag) {
        Stack S = new NodeStack(); // Stack for matching tags
        for (int i=0; (i<tag.length) && (tag[i] != null); i++) {
            if (tag[i].isOpening())
            S.push(tag[i].getName()); // opening tag; push its name on the stack
            else {
                if (S.isEmpty()) // nothing to match
                        return false;
                if (!((String) S.pop()).equals(tag[i].getName())) // wrong match
                        return false;
            }
        }
        if (S.isEmpty())
            return true; // we matched everything
        return false; // we have some tags that never were matched
    }
```


Tag Matching Algorithm, cont.

```
public final static int  CAPACITY = 1000;        // Tag array size upper bound
  /* Parse an HTML document into an array of html tags */
  public  Tag[ ]  parseHTML(BufferedReader r) throws  IOException {
      String line; // a line of text
      boolean   inTag =  false    ;              // true iff we are in a tag
       Tag[ ]  tag  =  new  Tag[CAPACITY];  // our tag array (initially all null)
      int  count   =  0             ;            // tag counter
      while  ((line  =  r.readLine()) !=  null)  {
          // Create a string tokenizer for HTML tags (use  <  and  >  as delimiters)
          StringTokenizer st  =  new  StringTokenizer(line,"<> \t",true);
          while  (st.hasMoreTokens())  {
              String token  = (String)  st.nextToken();
              if  (token.equals("<"))  // opening a new HTML tag
                      inTag  =  true;
              else if  (token.equals(">"))   // ending an HTML tag
                      inTag  =  false;
              else if  (inTag)  { // we have a opening or closing HTML tag
                  if  ( (token.length() ==  0)  | | (token.charAt(0) !=  '/') )
                      tag[count++] =  new  Tag(token,  true);  // opening tag
                  else   // ending tag
                      tag[count++] =  new  Tag(token.substring(1),  false);  // skip the
              } // Note: we ignore anything not in an HTML tag
          }
      }
      return  tag;  // our array of tags
  }
  /** Tester method */
  public static void  main(String[ ]  args)  throws  IOException  {
      BufferedReader stdr;          // Standard Input Reader
      stdr  =  new  BufferedReader(new  InputStreamReader(System.in));
      HTML tagChecker  =  new  HTML();
      if  (tagChecker.isHTMLMatched(tagChecker.parseHTML(stdr)))
              System.out.println("The input file is a matched HTML document.");
      else
              System.out.println("The input file is not a matched HTML document.");
      }
}
```

Computing Spans (not in book)

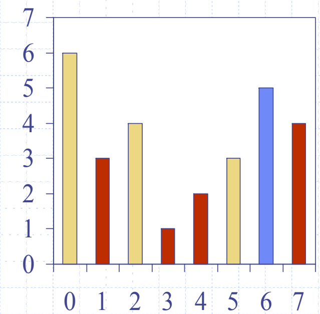
- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \le X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack



Quadratic Algorithm

```
Algorithm spans1(X, n)
   Input array X of n integers
   Output array S of spans of X              #
   S ← new array of n integers               n
   for i ← 0 to n − 1 do                     n
      s ← 1                                  n
      while s ≤ i ∧ X[i − s] ≤ X[i]          1 + 2 + …+ (n − 1)
         s ← s + 1                           1 + 2 + …+ (n − 1)
      S[i] ← s                               n
   return S                                  1
```

- Algorithm *spans1* runs in $O(n^2)$ time

Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                       #
   S ← new array of n integers               n
   A ← new empty stack                       1
   for i ← 0 to n − 1 do                     n
      while (¬A.isEmpty() ∧
            X[A.top()] ≤ X[i] ) do           n
         A.pop()                             n
      if A.isEmpty() then                    n
         S[i] ← i + 1                        n
      else
         S[i] ← i − A.top()                  n
      A.push(i)                              n
   return S                                  1
```