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

- 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

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.

```
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 \( \text{size}() \)
\[
\text{return } t + 1
\]

Algorithm \( \text{push}(o) \)
\[
\text{if } t = S.\text{length} - 1 \text{ then}
\quad \text{throw } \text{FullStackException}
\text{else}
\quad t \leftarrow t + 1
\quad S[t] \leftarrow o
\]

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.

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.

Computing Spans

- We show how to use a stack as an auxiliary data structure in an algorithm.
- Given 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 \( X[j] \leq X[i] \).
- Spans have applications to financial analysis.
  - E.g., stock at 52-week high.

Quadratic Algorithm

Algorithm \( \text{spans}(X, n) \)
\[
\text{Input array } X \text{ of } n \text{ integers}
\quad S \leftarrow \text{new array of } n \text{ integers}
\quad \text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do}
\quad \quad s \leftarrow 1
\quad \quad \text{while } s \leq i \land X[i-s] \leq X[i] \text{ do}
\quad \quad \quad s \leftarrow s + 1
\quad \quad \text{end while}
\quad \quad S[i] \leftarrow s
\quad \text{end for}
\quad \text{return } S
\]

Algorithm \( \text{spans}(X, n) \) runs in \( O(n^2) \) time.
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.

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.
- How large should the new array be?
  - Incremental strategy: increase the size by a constant \( c \).
  - Doubling strategy: double the size.

Algorithm `push(X)`

\[
\text{if } i = S.length - 1 \text{ then}
\quad A \leftarrow \text{new array of size } \\
\quad \text{for } i \leftarrow 0 \text{ to } i \text{ do}
\quad A[i] \leftarrow S[i]
\quad S[i] \leftarrow \text{i + 1}
\quad S[0] \leftarrow 0
\]

Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly once.
  - Is popped from the stack at most once.
- The statements in the while-loop are executed at most \( n \) times.
- Algorithm `span2` runs in \( O(n) \) time.

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time \( T(n) \) needed to perform a series of \( n \) push operations.
- We assume that we start with an empty stack represented by an array of size 1.
- We call amortized time of a push operation the average time taken by a push over the series of operations, i.e., \( T(n)/n \).

Incremental Strategy Analysis

- We replace the array \( k = \frac{n}{c} \) times.
- The total time \( T(n) \) of a series of \( n \) push operations is proportional to
  \[
  n + c + 2c + 3c + 4c + \cdots + kc = \\
  n + c(1 + 2 + 3 + \cdots + k) = \\
  n + \frac{ck(k + 1)}{2}
  \]
- Since \( c \) is a constant, \( T(n) \) is \( O(n + k^2) \), i.e., \( O(n^2) \).
- The amortized time of a push operation is \( O(n) \).

Doubling Strategy Analysis

- We replace the array \( k = \log_2 n \) times.
- The total time \( T(n) \) of a series of \( n \) push operations is proportional to
  \[
  n + 1 + 2 + 4 + 8 + \cdots + 2^k = \\
  n + 2^k + 1 - 1 = 2n - 1
  \]
- \( T(n) \) is \( O(n) \).
- The amortized time of a push operation is \( O(1) \).
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

```java
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object top() throws EmptyStackException;
    public void push(Object o);
    public Object pop() throws EmptyStackException;
}
```

Array-based Stack in Java

```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;
    }
}
```