Stacks
Outline

- The Stack ADT
- Applications of Stacks
- Array-based implementation
- Growable array-based stack
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 o)` inserts element `o`.
  - `pop()` removes and returns the last inserted element.
- Auxiliary stack operations:
  - `top()` returns a reference to the last inserted element without removing it.
  - `size()` returns the number of elements stored.
  - `empty()` returns a Boolean value indicating 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
- Saving local variables when one function calls another, and this one calls another, and so on.

Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures
Let the Hunger Games begin!
C++ Run-time Stack

The C++ run-time system keeps track of the chain of active functions with a stack.

When a function is called, the run-time system pushes on the stack a frame containing:
- Local variables and return value
- Program counter, keeping track of the statement being executed.

When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack.

```c++
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 ← 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 $\text{push}(o)$
\[
\begin{align*}
\text{if } t &= S.\text{length} - 1 \text{ then} \\
\text{throw } &\text{FullStackException} \\
\text{else} \\
& t \leftarrow t + 1 \\
&S[t] \leftarrow o
\end{align*}
\]
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 \textit{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.
- Spans have applications to financial analysis.
  - E.g., stock at 52-week high.
- 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 $X[j] \leq X[i]$.

**Example**

<table>
<thead>
<tr>
<th>X</th>
<th>6</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Naïve Compute Spans Algorithm

Algorithm \( \text{spans1}(X, n) \)

<table>
<thead>
<tr>
<th>Input</th>
<th>array ( X ) of ( n ) integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>array ( S ) of spans of ( X )</td>
</tr>
<tr>
<td>( S )</td>
<td>new array of ( n ) integers</td>
</tr>
<tr>
<td>for ( i ) ← 0 to ( n - 1 ) do</td>
<td>( n )</td>
</tr>
<tr>
<td>( s ) ← 1</td>
<td>( n )</td>
</tr>
<tr>
<td>while ( s \leq i \land X[i - s] \leq X[i] )</td>
<td>( 1 + 2 + \ldots + (n - 1) )</td>
</tr>
<tr>
<td>( s ) ← ( s + 1 )</td>
<td>( 1 + 2 + \ldots + (n - 1) )</td>
</tr>
<tr>
<td>( S[i] ) ← ( s )</td>
<td>( n )</td>
</tr>
<tr>
<td>return ( S )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

[TEAMS] What is the big-Oh?

Algorithm \( \text{spans1} \) runs in \( O(n^2) \) time

Stacks
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 $i$ onto the stack
Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one time
  - 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 $spans2(X, n)$

```plaintext
S ← new array of $n$ integers
A ← new empty stack
for $i ← 0$ to $n - 1$ do
  while (¬A.isEmpty() ∧ $X[top()] \leq X[i]$) do
    $j ← A.pop()$
  if A.isEmpty() then
    $S[i] ← i + 1$
  else
    $j ← top()$
    $S[i] ← i - j$
  $A.push(i)$
return $S$
```

Stacks
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 $push(o)$

```
if $t = S\.length - 1$ then
    $A \leftarrow$ new array of size …
    for $i \leftarrow 0$ to $t$ do
        $A[i] \leftarrow S[i]$
        $S \leftarrow A$
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
```
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 $k = n/c$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc$$

$$= n + c(1 + 2 + 3 + \ldots + k)$$

$$= n + ck(k + 1)/2$$

- 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

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$

$$= n + 2^{k+1} - 1$$

$$= 2n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is
  - $O(1)$
Stack Interface in C++

- Interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Similar to STL vector
- STL construct is stack

```cpp
template <typename Object>
class Stack {
public:
    int size();
    bool empty();
    Object& top() throw(EmptyStackException);
    void push(Object o);
    Object pop() throw(EmptyStackException);
};
```
Array-based Stack in C++

template <typename Object>
class ArrayStack {
private:
    int capacity;       // stack capacity
    Object *S;          // stack array
    int top;            // top of stack
public:
    ArrayStack(int c) {
        capacity = c;
        S = new Object[capacity];
        t = -1;
    }
    bool empty() {
        return (t < 0);
    }
    Object pop() throw(EmptyStackException) {
        if(isEmpty())
            throw EmptyStackException("Access to empty stack");
        return S[t--];
    }
    // … (other functions omitted)