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 \texttt{buy}(stock, shares, price)
  - order \texttt{sell}(stock, shares, price)
  - void \texttt{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
- `isEmpty()`: 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
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]
```

Stacks
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)**

```java
if t = S.length - 1 then
    throw FullStackException
else
    t ← t + 1
    S[t] ← o
```

---

Stacks
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.
- 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.

<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>
 Quadratic Algorithm

Algorithm $spans1(X, n)$

Input array $X$ of $n$ integers

Output array $S$ of spans of $X$

$S \leftarrow$ new array of $n$ integers

for $i \leftarrow 0$ to $n - 1$ do

$s \leftarrow 1$

while $s \leq i \wedge X[i - s] \leq X[i]$

$s \leftarrow s + 1$

$S[i] \leftarrow s$

return $S$

$\diamondsuit$ Algorithm $spans1$ 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 $i$ onto the stack.
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 \( \text{spans2} \) runs in \( O(n) \) time

```
Algorithm \( \text{spans2}(X, n) \) #
\[
S \leftarrow \text{new array of } n \text{ integers} \\
A \leftarrow \text{new empty stack} \\
\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
    \text{while } (\neg A.\text{isEmpty()} \amp X[\text{top()}] \leq X[i]) \text{ do} \\
        j \leftarrow A.\text{pop()} \\
        \text{if } A.\text{isEmpty()} \text{ then} \\
            S[i] \leftarrow i + 1 \\
        \text{else} \\
            j \leftarrow \text{top()} \\
            S[i] \leftarrow i - j \\
    A.\text{push}(i) \\
\text{return } S
\]
```
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 $push(o)$

```plaintext
if $t = S.length - 1$ then
    $A \leftarrow$ new array of size ...
    for $i \leftarrow 0$ to $t$ do
        $A[i] \leftarrow S[i]
        S \leftarrow A$
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
```

Stacks
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

\[
\begin{align*}
  n + 1 + 2 + 4 + 8 + \ldots + 2^k \\
  = n + 2^{k+1} - 1 \\
  = 2n - 1
\end{align*}
\]

- \( T(n) \) 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
- Most similar STL construct is vector

```cpp
template <typename Object>
class Stack {
public:
  int size();
  bool isEmpty();
  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 isEmpty() {
        return (t < 0);
    }

    Object pop() throw(EmptyStackException) {
        if(isEmpty())
            throw EmptyStackException
                (“Access to empty stack”);
        return S[t--];
    }

    // . . . (other functions omitted)
Queues
Outline and Reading

- The Queue ADT
- Implementation with a circular array
- Growable array-based queue
- Queue interface in C++
The Queue ADT

- The Queue ADT stores arbitrary objects.
- Insertions and deletions follow the first-in first-out scheme.
- Insertions are at the rear of the queue and removals are at the front of the queue.
- Main queue operations:
  - `enqueue(Object o)`: inserts an element `o` at the end of the queue.
  - `dequeue()`: removes and returns the element at the front of the queue.

Auxiliary queue operations:
- `front()`: returns the element at the front without removing it.
- `size()`: returns the number of elements stored.
- `isEmpty()`: returns a Boolean indicating whether no elements are stored.

Exceptions:
- Attempting the execution of `dequeue` or `front` on an empty queue throws an `EmptyQueueException`.
Applications of Queues

**Direct applications**
- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

**Indirect applications**
- Auxiliary data structure for algorithms
- Component of other data structures
Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$  index of the front element
  - $r$  index immediately past the rear element
- Array location $r$ is kept empty

**Normal configuration**

```
Q: 0 1 2  f  r
```

**Wrapped-around configuration**

```
Q: 0 1 2  r  f
```
Queue Operations

Hint: use the modulo operator

Algorithm \textit{size}()
\begin{align*}
\text{return } & (N - f + r) \mod N \\
\text{or } & = ((N+r)-f) \mod N
\end{align*}

Algorithm \textit{isEmpty}()
\begin{align*}
\text{return } & (f = r)
\end{align*}

e.g. \( N=17, \ f=4, \ r=14 \)
size\(=(17-4+14) \mod 17 \)
size\(=27 \mod 17 \)
size\(=10 \)
Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full.
- This exception is implementation-dependent.

Algorithm `enqueue(o)`

```java
if size() = N - 1 then
    throw FullQueueException
else
    Q[r] ← o
    r ← (r + 1) mod N
```

![Queue Diagram]

```
0 1 2  f  r
```

![Queue Diagram]

```
0 1 2  r  f
```
Queue Operations (cont.)

- **Operation dequeue** throws an exception if the queue is empty.
- This exception is specified in the queue ADT.

**Algorithm dequeue()**

```java
if isEmpty() then
    throw EmptyQueueException
else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```

![Queue Diagram](image)
Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.
- Similar to what we did for an array-based stack.
- The enqueue operation has amortized running time:
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy
Informal C++ Queue Interface

- Informal C++ interface for our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in STL class

```cpp
template <typename Object>
class Queue {
public:
    int size();
    bool isEmpty();
    Object& front()
        throw(EmptyQueueException);
    void enqueue(Object o);
    Object dequeue()
        throw(EmptyQueueException);
};
```