Chapter 5:
Stacks, Queues and Deques

Nancy Amato
Parasol Lab, Dept. CSE, Texas A&M University

Acknowledgement: These slides are adapted from slides provided with Data Structures and Algorithms in C++, Goodrich, Tamassia and Mount (Wiley 2004)
Stacks
Outline and Reading

- The Stack ADT (§5.1.1)
- Array-based implementation (§5.1.4)
- 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 (LIFO) scheme
- Main stack operations:
  - Push(e): inserts element e at the top of the stack
  - pop(): removes and returns the top element of the stack (last inserted element)
  - top(): returns reference to the top element without removing it
- Auxiliary stack operations:
  - size(): returns the number of elements in the stack
  - empty(): a Boolean value indicating whether the stack is empty
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.
Exercise: Stacks

• Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)
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.
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 empty() 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 `push(o)`

```java
if t = S.length - 1 then
    throw FullStackException
else
    t ← t + 1
    S[t] ← o
```
Performance and Limitations
- array-based implementation of stack ADT

• 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
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
    $A[i] \leftarrow S[i]$
    $S \leftarrow A$
    $t \leftarrow t + 1$
  $S[t] \leftarrow o$
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 ← new array of size ...
    for i ← 0 to t do
        A[i] ← S[i]
        S ← A
    t ← t + 1
    S[t] ← 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
- 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;   // stack capacity
        S = new Object[capacity];
        t = -1;
    }
    // … (other functions omitted)
}

isNotEmpty() {
    return (t < 0);
}

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

// … (other functions omitted)
Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node
Stack with a Singly Linked List

- We can implement a stack with a singly linked list.
- The top element is stored at the first node of the list.
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time.

![Diagram of a singly linked list stack with elements]
Exercise

- Describe how to implement a stack using a singly-linked list
  - Stack operations: push(x), pop(), size(), isEmpty()
  - For each operation, give the running time
# Stack Summary

- **Stack Operation Complexity for Different Implementations**

<table>
<thead>
<tr>
<th></th>
<th>Array Fixed-Size</th>
<th>Array Expandable (doubling strategy)</th>
<th>List Singly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pop()</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Push(o)</strong></td>
<td>O(1)</td>
<td>O(n) Worst Case</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(1) Best Case</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(1) Average Case</td>
<td></td>
</tr>
<tr>
<td><strong>Top()</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Size(), isEmpty()</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Queues
Outline and Reading

• The Queue ADT (§5.2.1)
• Implementation with a circular array (§5.2.4)
  • Growable array-based queue
• List-based queue
The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) 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 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 value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of `dequeue` or `front` on an empty queue throws an `EmptyQueueException`
Exercise: Queues

• Describe the output of the following series of queue operations
  • enqueue(8)
  • enqueue(3)
  • dequeue()
  • enqueue(2)
  • enqueue(5)
  • dequeue()
  • dequeue()
  • dequeue()
  • enqueue(9)
  • enqueue(1)
Applications of Queues

• Direct applications
  • Waiting lines
  • 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

\[ Q \]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & r & f & \ldots & \ldots & \ldots \\
\end{array}
\]

wrapped-around configuration

\[ Q \]

\[
\begin{array}{cccccccc}
\ldots & \ldots & \ldots & 0 & 1 & 2 & r & f \\
\end{array}
\]
Queue Operations

- We use the modulo operator (remainder of division)

Algorithm size()
    return \((N - f + r) \mod N\)

Algorithm isEmpty()
    return \((f = r)\)
Queue Operations (cont.)

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

Algorithm enqueue(o)

\[
\begin{cases}
\text{if } \text{size()} = N - 1 \text{ then} \\
\quad \text{throw FullQueueException} \\
\text{else} \\
\quad Q[r] \leftarrow o \\
\quad r \leftarrow (r + 1) \mod N
\end{cases}
\]

\[
Q \begin{array}{cccccccccccccc} 
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
0 & 1 & 2 & f & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
0 & 1 & 2 & r & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\end{array}
\]
• Operation dequeue throws an exception if the queue is empty
• This exception is specified in the queue ADT

Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    \[ o \leftarrow Q[f] \]
    \[ f \leftarrow (f + 1) \mod N \]
  return o
Performance and Limitations
- array-based implementation of queue ADT

• **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
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
Exercise

- Describe how to implement a queue using a singly-linked list
  - Queue operations: enqueue(x), dequeue(), size(), isEmpty()
  - For each operation, give the running time
Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the head of the list
  - The rear element is stored at the tail of the list
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
- NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, i.e., the queue is NEVER full.
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);
};
```
## Queue Summary

- **Queue Operation Complexity for Different Implementations**

<table>
<thead>
<tr>
<th></th>
<th>Array Fixed-Size</th>
<th>Array Expandable (doubling strategy)</th>
<th>List Singly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>enqueue(o)</td>
<td>O(1)</td>
<td>O(n) Worst Case</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(1) Best Case</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(1) Average Case</td>
<td></td>
</tr>
<tr>
<td>front()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Size(), isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
The Double-Ended Queue ADT (§5.3)

- The Double-Ended Queue, or Deque, ADT stores arbitrary objects. (Pronounced ‘deck’)
- Richer than stack or queue ADTs. Supports insertions and deletions at both the front and the end.
- Main deque operations:
  - `insertFirst(object o)`: inserts element `o` at the beginning of the deque
  - `insertLast(object o)`: inserts element `o` at the end of the deque
  - `RemoveFirst()`: removes and returns the element at the front of the queue
  - `RemoveLast()`: removes and returns the element at the end of the queue

- Auxiliary queue operations:
  - `first()`: returns the element at the front without removing it
  - `last()`: returns the element at the front 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 dequeue or front on an empty queue throws an EmptyDequeException
Doubly Linked List

- A doubly linked list provides a natural implementation of the Deque ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Deque ADT takes $O(1)$ time
Performance and Limitations
- doubly linked list implementation of deque ADT

• 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
  • NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, i.e., the deque is NEVER full.
# Deque Summary

- Deque Operation Complexity for Different Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array Fixed-Size</th>
<th>Array Expandable (doubling strategy)</th>
<th>List Singly-Linked</th>
<th>List Doubly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeFirst(), removeLast()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n) for one at list tail, O(1) for other</td>
<td>O(1)</td>
</tr>
<tr>
<td>insertFirst(o), InsertLast(o)</td>
<td>O(1)</td>
<td>O(n) Worst Case</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>first(), last</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Size(), isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>