HOWDY!

WELCOME TO CSCE 221 – DATA STRUCTURES AND ALGORITHMS
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
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
CH5.
STACKS, QUEUES, AND DEQUES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO
STACKS

• The Stack ADT (Ch. 5.1.1)
• Array-based implementation (Ch. 5.1.4)
• Growable array-based stack
STACKS

- A data structure similar to a neat stack of something, basically only access to top element is allowed — also referred to as LIFO (last-in, first-out) storage

- 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
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
• 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)
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

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

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

bar(int m) {
    ...
}
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5
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 $\text{pop()}$

\[
\text{if empty() then}
\]
\[
\text{throw EmptyStackException}
\]
\[
t \leftarrow t - 1
\]
\[
\text{return } S[t + 1]
\]
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

    t ← t + 1

S[t] ← o
```

![Diagram](image)
NOTE ON ALGORITHM ANALYSIS

• Computer Scientists are concerned with describing how long an algorithm (computation) takes
  • Described through functions which show how time grows as function of input, note that there are no constants!
  • $O(1)$ – Constant time
  • $O(\log n)$ - Logarithmic time
  • $O(n)$ – Linear time
  • $O(n^2)$ – Quadratic time

• More detail in CSCE 222, MATH 302, and/or later in course
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)`

```plaintext
if $t = S.length - 1$ then
  $A \leftarrow$ new array of size ...

  for $i \leftarrow 0$ to $t$ do
    $A[i] \leftarrow S[i]$
    $S \leftarrow A$
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
```
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.

• 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

• Let \( c \) be the constant increase and \( n \) be the number of push operations
• We replace the array \( k = n/c \) times
• The total time \( T(n) \) of a series of \( n \) push operations is proportional to

\[
T(n) = n + c + 2c + 3c + 4c + \ldots + kc
\]

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

\[
= n + c \frac{k(k + 1)}{2}
\]

\[
= O(n + k^2) = O \left(n + \frac{n^2}{c}\right) = O(n^2)
\]

Side note:

\[
1 + 2 + \ldots + k = \sum_{i=0}^{k} i = \frac{k(k + 1)}{2}
\]

• \( T(n) \) is \( O(n^2) \) so the amortized time of a push is \( \frac{O(n^2)}{n} = O(n) \)
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
  \[
  n + 1 + 2 + 4 + 8 + \ldots + 2^k
  = n + 2^{k+1} - 1
  = O(n + 2^k) = O(n + 2^{\log_2 n}) = O(n)
  \]

• \( T(n) \) is \( O(n) \) so the amortized time of a push is
  \[
  \frac{O(n)}{n} = O(1)
  \]
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
EXERCISE

• Describe how to implement a stack using a singly-linked list
  • Stack operations: push\( (x) \), pop\()
  • For each operation, give the running time
<table>
<thead>
<tr>
<th>Method</th>
<th>Array Fixed-Size</th>
<th>Array Expandable (doubling strategy)</th>
<th>List Singly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>push(o)</td>
<td>$O(1)$</td>
<td>$O(n)$ Worst Case $O(1)$ Best Case</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$ Average Case</td>
<td></td>
</tr>
<tr>
<td>top()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size(), empty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
QUEUES

• The Queue ADT (Ch. 5.2.1)

• Implementation with a circular array (Ch. 5.2.4)
  • Growable array-based queue

• List-based queue
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
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(e): inserts element e 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
  • empty(): 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()
  • enqueue(9)
  • enqueue(1)
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

wrapped-around configuration
QUEU E OPERATIONS

• We use the modulo operator (remainder of division)

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

Algorithm isEmpty()
return \(f = r\)
• Operation enqueue throws an exception if the array is full
• This exception is implementation-dependent

**Algorithm enqueue(o)**

```plaintext
if size() = N - 1 then
    throw FullQueueException

Q[r] ← o
r ← r + 1 mod N
```
QUEUE OPERATIONS (CONT.)

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

```
Algorithm dequeue()
if empty() then
    throw EmptyQueueException
o ← Q[f]
f ← 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 *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 enqueue\((e)\), 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

• \texttt{enqueue}(q)\) 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(), empty()
  • For each operation, give the running time
• 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.
## Queue Summary

<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 $O(1)$ Best Case $O(1)$ Average Case</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>front()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size(), empty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
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:
  - `insertFront(e)`: inserts element `e` at the beginning of the deque
  - `insertBack(e)`: inserts element `e` at the end of the deque
  - `eraseFront()`: removes and returns the element at the front of the queue
  - `eraseBack()`: removes and returns the element at the end of the queue

- Auxiliary queue operations:
  - `front()`: returns the element at the front without removing it
  - `back()`: returns the element at the front 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 `dequeue` or `front` on an empty queue throws an `EmptyDequeException`
A doubly linked list provides a natural implementation of the Deque ADT.

Nodes implement Position and store:
- element
- link to previous node
- link to next node

Special trailer and header nodes
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

<table>
<thead>
<tr>
<th></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>eraseFront(), eraseBack()</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>insertFront($o$), insertBack($o$)</td>
<td>$O(1)$</td>
<td>$O(n)$ Worst Case $O(1)$ Best Case $O(1)$ Average Case</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>front(), back()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size(), empty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
INTERVIEW QUESTION 1

• How would you design a stack which, in addition to push and pop, also has a function \texttt{min} which returns the minimum element? \texttt{push, pop and min} should all operate in \(O(1)\) time.

INTERVIEW QUESTION 2

• In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:
  (1) Only one disk can be moved at a time.
  (2) A disk is slid off the top of one tower onto the next tower.
  (3) A disk can only be placed on top of a larger disk.
Write pseudocode to move the disks from the first tower to the last using stacks.