CH 5: STACKS, QUEUES, AND DEQUES

ACKNOWLEDGEMENT: THE SLIDES ARE PREPARED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO AND JORY DENNY
The Stack ADT (Ch. 5.1.1)
- Array-based implementation (Ch. 5.1.4)
- Growable array-based stack
- Singly list implementation (Ch 5.1.5)
A stack is a container with visibility and access through one end known as top element.

LIFO – Last In First Out Principle

Practical Applications:
- Web browser history of pages: Back button
- Editors: Undo button
- Runtime Stack: Nested Function calls and storage of parameters, variables, return address, etc.

Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures
STACK ADT

- Stores arbitrary objects
- Insertions, deletion, modification and access is allowed through the top element.
- Main Operations:
  - `push(e)` – inserts element e at the top of the stack
  - `pop()` – removes the top element from the stack
  - `top()` – returns the top element without removing it

- Auxiliary operations:
  - `size()` – Number of elements in the stack
  - `empty()` – True if the stack contains no elements
  - Removing or accessing an element in an empty stack returns an exception called `EmptyStackException`. 
EXERCISE

- Show the stack after each of these operations:
  - Push(5)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(8)
  - Pop()
  - Pop()
  - Pop()
  - Pop()
The C++ run-time system keeps track of the chain of active functions with a stack.

- When a function is called, the system pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack.
Array based implementation is the simplest

- Add elements from left to right of the array
- Keep track of the index of the top element in a variable, \( t \)
- Do you observe any problems?

**Algorithm push(e)**

\[
\begin{align*}
\text{Algorithm push}(e) & \\
& t \leftarrow t + 1 \\
& S[t] \leftarrow e
\end{align*}
\]

**Algorithm pop()**

\[
\begin{align*}
\text{Algorithm pop}() & \\
& \text{if empty() then} \\
& & \text{throw StackEmptyException} \\
& \text{else} \\
& & t \leftarrow t - 1 \\
& & \text{return } S[t + 1]
\end{align*}
\]
The array storing the stack elements may become full.

A push operation will then throw a **StackFullException**.

<table>
<thead>
<tr>
<th>Limitation of the array-based implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not intrinsic to the Stack ADT</td>
</tr>
</tbody>
</table>

**Algorithm push(e)**

```java
if size() = S.length then
    throw StackFullException
else
    t ← t + 1
    S[t] ← e
```

**Algorithm size()**

```java
return t + 1
```
Computer Scientists are concerned with describing how long and how much memory an algorithm (computation) takes, known as time and space complexity of the algorithm.

- Described through functions which show how time or space 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
## STACK PERFORMANCE

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array Fixed Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>push(e)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>top()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size(), empty()</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
 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
Instead of throwing an exception while pushing to a filled stack, replace the array with a larger array.

How much to increase to?

- **Incremental strategy:** increase the size by a constant $c$,
  
  $$l \leftarrow S.\text{length} + c$$

- **Doubling strategy:** double the size,
  
  $$l \leftarrow 2 \times S.\text{length}$$

Algorithm `push(e)`

```plaintext
if $t = S.\text{length} - 1$ then
    $A \leftarrow$ new array of length, $l$
    for $i \leftarrow 0$ to $t$ do
        $A[i] \leftarrow S[i]$
        $S \leftarrow A$
    $t \leftarrow t + 1$
    $S[t] \leftarrow e$
```
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., $T(n)/n$. 
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

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

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

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

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

$T(n)$ is $O(n^2)$ so the amortized time of a push is $\frac{O(n^2)}{n} = O(n)$.

Note:

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

$$\frac{O(n)}{n} = O(1)$$
# Stack Performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Array Fixed Capacity</th>
<th>Growable Array (Doubling)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>push(e)</td>
<td>$O(1)$</td>
<td>$O(n)$ Worst Case&lt;br&gt;$O(1)$ Best Case&lt;br&gt;$O(1)$ Average Case</td>
</tr>
<tr>
<td>top()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size(), empty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
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(), size(), empty()
  - For each operation, give the running time
## STACK SUMMARY

<table>
<thead>
<tr>
<th></th>
<th>Array Fixed Capacity</th>
<th>Growable Array (Doubling)</th>
<th>Singly Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>push(e)</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>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>
The Queue ADT (Ch. 5.2.1)
Implementation with a circular array (Ch. 5.2.4)
- Growable array-based queue
- List-based queue
QUEUES

- Container storing arbitrary objects such that insertions allowed at one end called **back** and removal from other end called **front**.
- **FIFO** – First In First Out scheme
- Practical applications:
  - Waiting lines
  - Scheduling – Task, shared resources (printer) access
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
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 operations:
- `enqueue(e)`: inserts an element $e$ at the end of the queue.
- `dequeue()`: removes 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.

Application of `dequeue()` on an empty queue throws `EmptyQueueException`.
Show the queue after each of the following operations:

- enqueue(5)
- enqueue(3)
- dequeue()
- enqueue(2)
- enqueue(8)
- dequeue()
- dequeue()
- dequeue()
- dequeue()
- enqueue(9)
ARRAY BASED QUEUE IMPLEMENTATION

- 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

![Diagram of normal and wrapped-around configurations of an array-based queue](image)
- Use modulo operator (finds the remainder of a division)

**Algorithm size()**

```plaintext```
return \((N - f + r) \mod N\)
```

**Algorithm empty()**

```plaintext```
return \(f = r\)
```

![Queue operations diagram](image-url)
Operation enqueue throws an exception if the array is full.
This exception is implementation-dependent.

Algorithm enqueue(e)

\[
\text{if size()} = N - 1 \text{ then}
\]
\[
\text{throw FullQueueException}
\]
\[
Q[r] \leftarrow e
\]
\[
r \leftarrow r + 1 \mod N
\]
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
- Let $n$ be the number of elements in the queue
- 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 enqueue(e), if the queue is full, similar to growable array-based stack, instead of throwing an exception, we can replace the array with a larger one.
- enqueue(e) has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy
Describe how to implement a queue using a singly-linked list
- Queue operations: enqueue(e), dequeue(), size(), empty()
- For each operation, give the running time
## Queue Summary

<table>
<thead>
<tr>
<th></th>
<th>Array with fixed capacity</th>
<th>Growable array (doubling)</th>
<th>Singly linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
| enqueue(e)           | $O(1)$                     | $O(n)$ Worst Case  
$O(1)$ Best Case  
$O(1)$ Average Case | $O(1)$             |
| front()              | $O(1)$                     | $O(1)$                    | $O(1)$             |
| size(), empty()      | $O(1)$                     | $O(1)$                    | $O(1)$             |
The Double-Ended Queue, or Deque, ADT stores arbitrary objects. (Pronounced ‘deck’) Supports insertions and deletions at both ends: front and back.

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`
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 BASED DEQUE

- 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 capacity</th>
<th>Growable array (doubling)</th>
<th>Singly linked list</th>
<th>Doubly linked list</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>