16.070

Introduction to Computers & Programming

Data structures: Stack ADT, Queue ADT, (Access types), Linked lists
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
The Stack ADT

- A classic data structure giving Last-In, First-Out (LIFO) access to elements of some type.

- Main stack operations:
  - push(object): inserts an element
  - Object pop(): removes and returns the last inserted element

- Auxiliary stack operations:
  - Object top(): returns the last inserted element without removing it
  - Integer size(): returns the number of elements stored
  - Boolean isEmpty(): indicates whether no elements are stored
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 `EmptyStackExcptn`
else
    \( t := t-1 \)
    return \( S[t+1] \)
Array-based Stack

- The array storing the stack elements may become full
- A push operation will then throw an exception

```
Algorithm push(o)
if t=S’length-1 then
    throw FullStackException
else
    t := t + 1
    S[t] := o
```

```
S
0 1 2 ...
t
```
Performance and Limitations

- **Performance**
  - Let N be the number of elements in the stack
  - The space used is O(N)
  - Each operation used is 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
Stack Implementations

- **Array-based**
  - Items “pushed” onto stack are added at next available array location. Requires **IsFull** operation.
  - Top of stack is referenced by an integer index into the array, which contains the index of the most recently pushed item.

- **Linked-List**
  - Items “pushed” onto stack stored in dynamically-allocated nodes added to the front (head) of the list.
  - Top of stack is referenced by the head pointer.

- **Inheritance**
  - Stack class is derived from a List class
  - Internal representation depends on the internals of the List
  - For linked-lists, List must provide equivalent of **InsertFront**
Some Stack Applications

- Direct applications
  - Postfix Expression Evaluation in RPN calculators
  - Page-visited history in a Web browser
  - Undo sequence in a text editor

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
Infix vs. Postfix

Infix Expressions:

5 + 3 + 4 + 1

(5 + 3) * 10

(5 + 3) * (10 - 4)

5 * 3 / (7 - 8)

(b*b - 4*a*c) / (2*a)

Corresponding Postfix:

5 3 + 4 + 1 +

5 3 + 10 *

5 3 + 10 4 - *

5 3 * 7 8 - /

b b * 4 a * c * - 2 a * /
How to Evaluate Postfix

A program can evaluate postfix expressions by reading the expression from left to right and following these simple rules:

- if a number is read, push it on the stack
- if an operator is read, pop two numbers off the stack (the first number popped is the second binary operand)
- apply the operation to the numbers, and push the result back onto the stack
- when the expression is complete, the number on top of stack is the answer

How is a bad postfix expression indicated?
Evaluating infix expressions

- Need 2 stacks. 1 numbers, 1 operators

while tokens available
    if (number) push on number_stack
    if (operator) push on operator_stack
    if ('(') do nothing
    else --must be ')'
        pop 2 numbers and 1 operator
        calculate
        push result on number_stack
    end
The Queue ADT stores arbitrary objects

Insertions and deletions follow the first-in first-out scheme (FIFO)

Insertions are at the rear of the queue and removals are at the front of the queue

Main queue operations:

- `enqueue(object)`: inserts an element at the end of the queue
- `dequeue()`: removes and returns the element at the front of the queue
The Queue ADT

- Auxiliary queue operations:
  - object `front()`: returns the element at the front without removing it
  - integer `size()`: returns the number of elements stored
  - boolean `isEmpty()`: indicates 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)
- Indirect applications
  - Auxiliary data structures 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

- We use the modulo operator

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

Algorithm `isEmpty()`
  return \((f=r)\)

### Normal configuration

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>f</th>
</tr>
</thead>
</table>

### Wrapped-around configuration

| 0 | 1 | 2 | r | f |
Queue Operations

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

Algorithm **enqueue(o)**

```
if size() = N-1 then
    throw FullQException
else
    Q[r] := o
    r := (r+1) mod N
```

---

**Normal configuration**

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

**Wrapped-around configuration**

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

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

Algorithm dequeue()
if isEmpty() then
  throw EmptyQException
else
  o := Q[f]
  f := (f+1) mod N
return o

Normal configuration

\[
\begin{array}{cccccccccccccc}
Q & & & & & & & & & & & & \\
0 & 1 & 2 & f & & & & & & & & & \\
\end{array}
\]

Wrapped-around configuration

\[
\begin{array}{cccccccccccccc}
Q & & & & & & & & & & & & \\
0 & 1 & 2 & r & & & & & & & & & \\
\end{array}
\]
In an **enqueue** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.
A singly linked list is a concrete data structure consisting of a sequence of nodes.

Each node stores:
- Element
- Link to the next node
Queue with a Singly Linked List

- We can implement a queue with a singly 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 Queue ADT takes $O(1)$ time
List ADT

- The list ADT models a sequence of positions storing arbitrary objects
- It allows for insertion and removal in the “middle”

Query methods:
- isFirst(p), isLast(p)

Accessor methods:
- first(), last()
- before(p), after(p)

Update methods:
- replaceElement(p, o), swapElements(p, q)
- insertBefore(p, o), insertAfter(p, o)
- insertFirst(o), insertLast(o)
- remove(p)
A doubly linked list provides a natural implementation of the List ADT

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

Special trailer and header node
### Analysis of Merge Sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MergeSort(A, lower_bound, upper_bound)</code></td>
<td><code>T(n)</code></td>
</tr>
<tr>
<td><code>begin</code></td>
<td></td>
</tr>
<tr>
<td><code>if (lower_bound &lt; upper_bound)</code></td>
<td><code>Θ(1)</code></td>
</tr>
<tr>
<td><code>mid = (lower_bound + upper_bound)/ 2</code></td>
<td><code>Θ(1)</code></td>
</tr>
<tr>
<td><code>MergeSort(A, lower_bound, mid)</code></td>
<td><code>T(n/2)</code></td>
</tr>
<tr>
<td><code>MergeSort(A, mid+1, upper_bound)</code></td>
<td><code>T(n/2)</code></td>
</tr>
<tr>
<td><code>Merge(A, lower_bound, mid,upper_bound)</code></td>
<td><code>Θ(n)</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td></td>
</tr>
</tbody>
</table>

- So `T(n) = Θ(1)` when `n = 1`, and
- `2T(n/2) + Θ(n)` when `n > 1`
- So what (more succinctly) is `T(n)`?
The expression:

\[
T(n) = \begin{cases} 
  c & n = 1 \\
  2T\left(\frac{n}{2}\right) + c & n > 1 
\end{cases}
\]

is a recurrence.

Recurrence: an equation that describes a function in terms of its value on smaller functions.