Week 2:
- Stacks
- Queues

Stacks and Queues

• Stacks
  – recursion
  – execution stack
  – backtracking, searching

• Queues
  – simulation
  – systems

ADT Stack

• ADT stack operations
  – Create an empty stack
  – Determine whether a stack is empty
  – Add a new item to the stack
  – Remove from the stack the item that was added most recently
  – Remove all the items from the stack
  – Retrieve from the stack the item that was added most recently

Stacks

• A stack
  – Last-in, first-out (LIFO) property
    • The last item placed on the stack will be the first item removed

Implementations of the ADT Stack

• The ADT stack can be implemented using
  – An array
  – A linked list
  – The ADT list

• StackInterface
  – Provides a common specification for the three implementations
  – StackException
    – Used by StackInterface
    – Extends java.lang.RuntimeException

Stack Interface

```java
public interface StackInterface {
    public boolean isEmpty();
    public void popAll();
    public void push(Object newItem) throws StackException;
    public Object pop() throws StackException;
    public Object peek() throws StackException;
} // end StackInterface
```
Implementations of the ADT Stack

An Array-Based Implementation of the ADT Stack

- StackArrayBased class
  - Implements StackInterface
  - Instances
    - Stacks
      - Private data fields
        - An array of objects called items
        - The index top

A Reference-Based Implementation of the ADT Stack

- A reference-based implementation
  - Required when the stack needs to grow and shrink dynamically
- StackReferenceBased
  - Implements StackInterface
  - top is a reference to the head of a linked list of items

An Implementation That Uses the ADT List

- The ADT list can be used to represent the items in a stack
- If the item in position 1 of a list represents the top of the stack
  - push(newItem) operation is implemented as add(1, newItem)
  - pop() operation is implemented as remove(1)
  - peek() operation is implemented as get(1)

An Implementation That Uses the ADT List
Using the ADT Stack in a Solution

- displayBackward can be refined by using stack operations

```java
public void displayBackward()
{
    while (nextChar != end of input)
    {
        stack.push(char);
    }
    while (! stack.isEmpty())
    {
        print stack.pop();
    }
}
```

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces
    ```
    abc{defg{ijk}{lm}{nop}}qr
    ```
  - An example of unbalanced braces
    ```
    abc{def}}{ghij{kl}m
    ```

Checking for Balanced Braces

- Requirements for balanced braces
  - Each time you encounter a "\[", it matches an already encountered "["
  - When you reach the end of the string, you have matched each "["

Checking for Balanced Braces

![Traces of the algorithm that checks for balanced braces](#)

Recognizing Strings in a Language

- Language L
  - L = {w$w' : w is a possible empty string of characters other than $, w' = reverse(w) }
  - A stack can be used to determine whether a given string is in L
    - Traverse the first half of the string, pushing each character onto a stack
    - Once you reach the $, for each character in the second half of the string, pop a character off the stack
    - Match the popped character with the current character in the string

Algebraic Expressions

- Infix expressions
  - (3+5)*7
- Postfix expressions
  - 3 5 + 7 *
- How about 3+(5*7) in postfix?
  - 3 5 7 * +
Evaluating Postfix Expressions

- A postfix calculator
  - Requires you to enter postfix expressions
    - Example: 2, 3, 4, +, *
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only "to the right" with respect to the operands
  - All parentheses are removed

The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm
The Abstract Data Type Queue

- A queue
  - New items enter at the back, or rear, of the queue
  - Items leave from the front of the queue
  - First-in, first-out (FIFO) property
    - The first item inserted into a queue is the first item to leave

- ADT queue operations
  - Create an empty queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove from the queue the item that was added earliest
  - Remove all the items from the queue
  - Retrieve from the queue the item that was added earliest

- Queues
  - Are appropriate for many real-world situations
    - Example: A line to buy a movie ticket
    - Have many applications in computer science
    - Example: A request to print a document
  - A simulation
    - A study to see how to reduce the wait involved in an application

Pseudocode for the ADT queue operations

createQueue()
// Creates an empty queue.

isEmpty()
// Determines whether a queue is empty

enqueue(newItem) throws QueueException
// Adds newItem at the back of a queue. Throws QueueException if the operation is not successful

decqueue() throws QueueException
// Retrieves and removes the front of a queue. Throws QueueException if the operation is not successful.

decqueueAll()
// Removes all items from a queue

peek() throws QueueException
// Retrieves the front of a queue. Throws QueueException if the retrieval is not successful

Some queue operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue.createQueue()</td>
<td></td>
</tr>
</tbody>
</table>
| queue.enqueue(5) | 5 | front
| queue.enqueue(2) | 5 2 |
| queue.enqueue(7) | 5 2 7 |
| queue.front = queue.peek() | 5 2 7 (queue.front is 5) |
| queue.front = queue.dequeue() | 5 2 7 (queue.front is 5) |
| queue.front = queue.dequeue() | 2 7 (queue.front is 2) |
Simple Applications of the ADT
Queue: Reading a String of Characters

- A queue can retain characters in the order in which they are typed
  
  ```java
  queue.createQueue();
  while (not end of line) {
    Read a new character ch
    queue.enqueue(ch)
  }
  ```

- Once the characters are in a queue, the system can process them as necessary

Recognizing Palindromes

- A palindrome
  - A string of characters that reads the same from left to right as it does from right to left
  - To recognize a palindrome, a queue can be used in conjunction with a stack
    - A stack can be used to reverse the order of occurrences
    - A queue can be used to preserve the order of occurrences

Implementations of the ADT Queue

- A queue can have either
  - An array-based implementation
  - A reference-based implementation

A Reference-Based Implementation

- Possible implementations of a queue
  - A linear linked list with two external references
    - A reference to the front
    - A reference to the back

- A circular linked list with one external reference
  - A reference to the back
A Reference-Based Implementation

Figure 7.4
Inserting an item into a nonempty queue

A Reference-Based Implementation

Figure 7.5
Inserting an item into an empty queue: a) before insertion; b) after insertion

A Reference-Based Implementation

Figure 7.6
Deleting an item from a queue of more than one item

An Array-Based Implementation

Figure 7.7
a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full

An Array-Based Implementation

Figure 7.8
A circular implementation of a queue

An Array-Based Implementation

Figure 7.9
The effect of some operations of the queue in Figure 7-8

• A circular array eliminates the problem of rightward drift
An Array-Based Implementation

- A problem with the circular array implementation
  - `front` and `back` cannot be used to distinguish between queue-full and queue-empty conditions

---

**Figure 7.10a**

- Front passes back when the queue becomes empty

---

**Figure 7.10b**

- Back catches up to front when the queue becomes full

---

An Array-Based Implementation

- To detect queue-full and queue-empty conditions
  - Keep a count of the queue items
- To initialize the queue, set
  - `front` to 0
  - `back` to `MAX_QUEUE - 1`
  - `count` to 0

---

An Array-Based Implementation

- Variations of the array-based implementation
  - Use a flag `full` to distinguish between the full and empty conditions
  - Declare `MAX_QUEUE + 1` locations for the array items, but use only `MAX_QUEUE` of them for queue items

---

An Array-Based Implementation

- Inserting into a queue
  
  ```
  back = (back+1) % MAX_QUEUE;
  items[back] = newItem;
  ++count;
  ```

- Deleting from a queue
  
  ```
  front = (front+1) % MAX_QUEUE;
  --count;
  ```
An Array-Based Implementation

Determining the Efficiency of Algorithms

- Analysis of algorithms
  - Provides tools for contrasting the efficiency of different methods of solution
- A comparison of algorithms
  - Should focus on significant differences in efficiency
  - Should not consider reductions in computing costs due to clever coding tricks

Figure 7.11
A more efficient circular implementation: a) a full queue; b) an empty queue

Determining the Efficiency of Algorithms

- Three difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?
- Algorithm analysis should be independent of
  - Specific implementations
  - Computers
  - Data

The Execution Time of Algorithms

- Counting an algorithm's operations is a way to access its efficiency
  - An algorithm's execution time is related to the number of operations it requires
  - Examples
    - Traversal of a linked list
    - The Towers of Hanoi
    - Nested Loops

Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the problem size
- An algorithm's growth rate
  - Enables the comparison of one algorithm with another
  - Examples
    - Algorithm A requires time proportional to \( n^2 \)
    - Algorithm B requires time proportional to \( n \)
- Algorithm efficiency is typically a concern for large problems only

Algorithm Growth Rates

- Figure 9.1
  - Time requirements as a function of the problem size \( n \)
  - Algorithm A requires \( n^2 \) seconds
  - Algorithm B requires \( n^* n \) seconds

- Figure 9.1
  - Time requirements as a function of the problem size \( n \)
Order-of-Magnitude Analysis and Big O Notation

- Definition of the order of an algorithm
  Algorithm $A$ is order $f(n)$ – denoted $O(f(n))$ – if constants $k$ and $n_0$ exist such that $A$ requires no more than $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$

- Growth-rate function
  - A mathematical function used to specify an algorithm’s order in terms of the size of the problem

- Big O notation
  - Example: $O(f(n))$
  - A notation that uses the capital letter $O$ to specify an algorithm’s order

Order-of-Magnitude Analysis and Big O Notation

<table>
<thead>
<tr>
<th>Function</th>
<th>$10$</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n^2)$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^8$</td>
<td>$10^{10}$</td>
<td>$10^{12}$</td>
<td>$10^{14}$</td>
<td>$10^{16}$</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$10^5$</td>
<td>$10^7$</td>
<td>$10^9$</td>
<td>$10^{11}$</td>
<td>$10^{13}$</td>
<td>$10^{15}$</td>
<td>$10^{17}$</td>
<td>$10^{19}$</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>$2^1$</td>
<td>$2^2$</td>
<td>$2^3$</td>
<td>$2^4$</td>
<td>$2^5$</td>
<td>$2^6$</td>
<td>$2^7$</td>
<td>$2^8$</td>
</tr>
</tbody>
</table>

Figure 9.3a
A comparison of growth-rate functions: a) in tabular form

Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions
  $O(1) < O(\log_2 n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^3) < O(2^n)$

- Properties of growth-rate functions
  - You can ignore low-order terms
  - You can ignore a multiplicative constant in the high-order term
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Order-of-Magnitude Analysis and Big O Notation

- Worst-case and average-case analyses
  - An algorithm can require different times to solve different problems of the same size
  - Worst-case analysis
    - A determination of the maximum amount of time that an algorithm requires to solve problems of size $n$
  - Average-case analysis
    - A determination of the average amount of time that an algorithm requires to solve problems of size $n$

Order-of-Magnitude Analysis and Big O Notation

- Keeping Your Perspective
  - Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency
  - When choosing an ADT’s implementation, consider how frequently particular ADT operations occur in a given application
  - Some seldom-used but critical operations must be efficient
Keeping Your Perspective

- If the problem size is always small, you can probably ignore an algorithm's efficiency
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on large problems

The Efficiency of Searching Algorithms

- Sequential search
  - Strategy
    - Look at each item in the data collection in turn, beginning with the first one
    - Stop when
      - You find the desired item
      - You reach the end of the data collection

The Efficiency of Searching Algorithms

- Sequential search
  - Efficiency
    - Worst case: O(n)
    - Average case: O(n)
    - Best case: O(1)

The Efficiency of Searching Algorithms

- Binary search
  - Strategy
    - To search a sorted array for a particular item
      - Repeatedly divide the array in half
      - Determine which half the item must be in, if it is indeed present, and discard the other half
    - Efficiency
      - Worst case: O(log_2 n)

Binary Search

- A high-level binary search
- Looking for a value 'x' in a sorted array

\[
\begin{array}{c}
1 & a & n \\
\end{array}
\]

- \( x < a \)? Continue on left half else right

Binary Search

- A high-level binary search
  if (anArray is of size |) |
    Determine if anArray's item is equal to value
  else |
    Find the midpoint of anArray
    Determine which half of anArray contains value
    if (value is in the first half of anArray) |
      binarySearch (first half of anArray, value) |
    else |
      binarySearch (second half of anArray, value) |
  // end if
  // end if
Binary Search

- Implementation issues:
  - How will you pass “half of anArray” to the recursive calls to binarySearch?
  - How do you determine which half of the array contains the value?
  - What should the base case(s) be?
  - How will binarySearch indicate the result of the search?

Binary Search

- Searching a number in a list of length n
- How many steps? (worst case)
- “Step” is “list item access”
- \( T(n) = 1 \)
- \( T(n) = k + T(n/2^k) \)
- Stop when: \( n/2^k = 1 \) or \( k = \log n \)
- \( T(n) = O(\log n) \)

Sorting Algorithms and Their Efficiency

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer’s main memory
  - An external sort
    - The collection of data will not fit in the computer’s main memory all at once but must reside in secondary storage
- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects
- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records
Selection Sort

- Selection sort
  - Strategy
    - Select the largest item and put it in its correct place
    - Select the next largest item and put it in its correct place, and so on

Figure 9.4
A selection sort of an array of five integers

<table>
<thead>
<tr>
<th>Initial array:</th>
<th>29 10 14 27 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 1st swap:</td>
<td>10 14 29 27 13</td>
</tr>
<tr>
<td>After 2nd swap:</td>
<td>10 14 29 13 27</td>
</tr>
<tr>
<td>After 3rd swap:</td>
<td>10 14 13 29 27</td>
</tr>
<tr>
<td>After 4th swap:</td>
<td>10 13 14 29 27</td>
</tr>
</tbody>
</table>

Selection Sort

- Analysis
  - Selection sort is $O(n^2)$
- Advantage of selection sort
  - It does not depend on the initial arrangement of the data
- Disadvantage of selection sort
  - It is only appropriate for small $n$

Summary

- Today
  - Stacks (carrano/prichard ch.7)
    - recursion
    - backtracking
    - parsing and evaluating expressions
  - Queues (carrano/prichard ch.8)
    - Simulation
    - Running times, complexity and asymptotics (carrano/prichard 10.1)
- Next week:
  - Trees