Chapter Goals

- Define an abstract data type and discuss its role in algorithm development
- Distinguish between a data type and a data structure
- Distinguish between an array-based implementation and a linked implementation
- Distinguish between an array and a list
Chapter Goals

• Distinguish between an unsorted list and a sorted list
• Distinguish between a selection sort and a bubble sort
• Describe the Quicksort algorithm
• Apply the selection sort, the bubble sort, and the Quicksort to a list of items by hand
• Apply the binary search algorithm
Chapter Goals

• Distinguish between the behavior of a stack and a queue

• Draw the binary search tree that is built from inserting a series of items

• Demonstrate your understanding of the algorithms in this chapter by hand simulating them with a sequence of items
Abstract Data Types

- **Abstract data type** A data type whose properties (data and operations) are specified independently of any particular implementation

  The goal in design is to reduce complexity through abstraction
Abstract Data Types

• In computing, we view data from three perspectives
  – Application level
    • View of the data within a particular problem
  – Logical level
    • An abstract view of the data values (the domain) and the set of operations to manipulate them
  – Implementation level
    • A specific representation of the structure to hold the data items and the coding of the operations in a programming language
Abstract Data Types

• **Data structures**  The implementation of a composite data fields in an abstract data type

• **Containers**  Objects whole role is to hold and manipulate other objects
Array-Based Implementations

• Recall that
  – an array is a named collection of homogeneous items
  – An item’s place within the collection is called an index

• If there is no ordering on the items in the container, we call the container unsorted

• If there is an ordering, we call the container sorted
Array-Based Implementations

Figure 9.1  A list
Array-Based Implementations

Figure 9.2
An unsorted list of integers
Array-Based Implementations

Figure 9.3
A sorted list of integers
• **Linked implementation** An implementation based on the concept of a node

• A node is made up of two pieces of information
  – the item that the user wants in the list, and
  – a pointer to the next node in the list
Linked Implementation

Figure 9.4 Anatomy of a linked list
Figure 9.5 An unsorted linked list
Figure 9.6  A sorted linked list
Figure 9.7  Store a node with info of 67 after current
Figure 9.8 Remove node next(current)
Lists

- List operations
  - Create itself
  - Insert an item
  - Delete an item
  - Print itself
  - Know the number of items it contains

- **Generic data type** (or **class**) A data type or class in which the operations are specified but the type or class of the objects being manipulated is not
Sorting

• Because sorting a large number of elements can be extremely time-consuming, a good sorting algorithm is very desirable

• We present several quite different sorting algorithms
Selection Sort

- List of names
  - Put them in alphabetical order
    - Find the name that comes first in the alphabet, and write it on a second sheet of paper
    - Cross out the name on the original list
    - Continue this cycle until all the names on the original list have been crossed out and written onto the second list, at which point the second list is sorted
Selection Sort (cont.)

• A slight adjustment to this manual approach does away with the need to duplicate space
  – As you cross a name off the original list, a free space opens up
  – Instead of writing the minimum value on a second list, exchange it with the value currently in the position where the crossed-off item should go
Selection Sort

Figure 9.9 Example of a selection sort (sorted elements are shaded)
Bubble Sort

• A selection sort that uses a different scheme for finding the minimum value
  – Starting with the last list element, we compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it
## Bubble Sort

### Example of a bubble sort

<table>
<thead>
<tr>
<th>items</th>
<th>items</th>
<th>items</th>
<th>items</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>Phil</td>
<td>[0]</td>
<td>Phil</td>
<td>[0]</td>
</tr>
</tbody>
</table>

a) First iteration (Sorted elements are shaded.)

<table>
<thead>
<tr>
<th>items</th>
<th>items</th>
<th>items</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>Al</td>
<td>[0]</td>
<td>Al</td>
</tr>
</tbody>
</table>

b) Remaining iterations (Sorted elements are shaded.)

---

**Figure 9.10**
Example of a bubble sort
Quicksort

• Based on the idea that it is faster and easier to sort two small lists than one larger one
  – Given a large stack of final exams to sort by name
  – Pick a splitting value, say L, and divide the stack of tests into two piles, A–L and M–Z
  – note that the two piles do not necessarily contain the same number of tests
  – Then take the first pile and subdivide it into two piles, A–F and G–L
  – This division process goes on until the piles are small enough to be easily sorted by hand
Quicksort

Figure 9.12 Ordering a list using the Quicksort algorithm
Quicksort

If (there is more than one item in list[first]..list[last])
Select splitVal
Split the list so that
list[first]..list[splitPoint−1] ≤ splitVal
list[splitPoint] = splitVal
list[splitPoint+1]..list[last] > splitVal
Quicksort the left half
Quicksort the right half
QuickSort

splitVal = 9

9  20  6  10  14  8  60  11

[last]

[first]

smaller values    larger values

9  8  6  10  14  20  60  11

[last]

[first]

smaller values    larger values

6  8  9  10  14  20  60  11

[first]    [split-Point]    [last]
Set left to first + 1
Set right to last
Do
  Increment left until list[left] > splitVal OR left > right
  Decrement right until list[right] < splitVal OR left > right
  Swap list[left] and list[right]
While (left <= right)
Set splitPoint to right
Swap list[first] and list[right]
Binary Search

- A **sequential search** of a list begins at the beginning of the list and continues until the item is found or the entire list has been searched.

- A **binary search** looks for an item in a list using a divide-and-conquer strategy.
Binary Search

• Binary Search Algorithm
  – Binary search algorithm assumes that the items in the list being searched are sorted
  – The algorithm begins at the middle of the list in a binary search
  – If the item for which we are searching is less than the item in the middle, we know that the item won’t be in the second half of the list
  – Once again we examine the “middle” element (which is really the item 25% of the way into the list)
  – The process continues with each comparison cutting in half the portion of the list where the item might be
Binary Search

Boolean Binary Search (first, last)

If (first > last)
    return false
Else
    Set middle to (first + last)/2
    Set result to item.compareTo(list[middle])
    If (result is equal to 0)
        return true
    Else
        If (result < 0)
            Binary Search (first, middle - 1)
        Else
            Binary Search (middle + 1, last)
### Binary Search

**Figure 9.14  Trace of the binary search**

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>BinarySearch(start, end)</th>
<th>Middle</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>ant</td>
<td>BinarySearch(0, 10)</td>
<td>5</td>
<td>cat &lt; dog</td>
</tr>
<tr>
<td>[1]</td>
<td>cat</td>
<td>BinarySearch(0, 4)</td>
<td>2</td>
<td>cat &lt; chicken</td>
</tr>
<tr>
<td>[2]</td>
<td>chicken</td>
<td>BinarySearch(0, 1)</td>
<td>0</td>
<td>cat &gt; ant</td>
</tr>
<tr>
<td>[3]</td>
<td>cow</td>
<td>BinarySearch(1, 1)</td>
<td>1</td>
<td>cat = cat  Return: true</td>
</tr>
<tr>
<td>[4]</td>
<td>deer</td>
<td>BinarySearch(0, 10)</td>
<td>5</td>
<td>zebra &gt; dog</td>
</tr>
<tr>
<td>[5]</td>
<td>dog</td>
<td>BinarySearch(6, 10)</td>
<td>8</td>
<td>zebra &gt; horse</td>
</tr>
<tr>
<td>[6]</td>
<td>fish</td>
<td>BinarySearch(9, 10)</td>
<td>9</td>
<td>zebra &gt; camel</td>
</tr>
<tr>
<td>[7]</td>
<td>goat</td>
<td>BinarySearch(10, 10)</td>
<td>10</td>
<td>zebra &gt; snake</td>
</tr>
<tr>
<td>[8]</td>
<td>horse</td>
<td>BinarySearch(11, 10)</td>
<td>last &gt; first Return: false</td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td>camel</td>
<td>BinarySearch(0, 10)</td>
<td>5</td>
<td>fish &gt; dog</td>
</tr>
<tr>
<td>[10]</td>
<td>snake</td>
<td>BinarySearch(6, 10)</td>
<td>8</td>
<td>fish &lt; horse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BinarySearch(6, 7)</td>
<td>6</td>
<td>fish = fish Return: true</td>
</tr>
</tbody>
</table>
# Binary Search

<table>
<thead>
<tr>
<th>Length</th>
<th>Sequential Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Base 10</td>
</tr>
<tr>
<td>10</td>
<td>5.5</td>
<td>2.9</td>
</tr>
<tr>
<td>100</td>
<td>50.5</td>
<td>5.8</td>
</tr>
<tr>
<td>1,000</td>
<td>500.5</td>
<td>9.0</td>
</tr>
<tr>
<td>10,000</td>
<td>5000.5</td>
<td>12.0</td>
</tr>
</tbody>
</table>

*Table 9.1 Average Number of Comparisons*
• A **stack** is an abstract data type in which accesses are made at only one end
  – LIFO, which stands for Last In First Out
  – The insert is called **Push** and the delete is called **Pop**
A **Queue** is an abstract data type in which items are entered at one end and removed from the other end
- FIFO, for First In First Out
- Like a waiting line in a bank or supermarket
- No standard queue terminology
  - **Enqueue, Enque, Enq, Enter, and Insert** are used for the insertion operation
  - **Dequeue, Deque, Deq, Delete, and Remove** are used for the deletion operation.
Stacks and Queues

Figure 9.15
Stack and queue visualized as linked structures

(a) A linked stack

(b) A linked queue
Trees

- ADTs such as lists, stacks, and queues are linear in nature
- More complex relationships require more complex structures
Hierarchical structures are called trees.

Binary trees:
- Each node has no more than two children.
- The beginning of the tree is a unique starting node called the root.
- The node to the left of a node, if it exists, is called its left child.
- The node to the right of a node, if it exists, is its right child.
- If a node in the tree has no children, it is called a leaf node.

Figure 9.16 A binary tree
Binary Search Trees

• A **binary search tree** has the **shape property** of a binary tree

• In addition, a binary search tree has a **semantic property**: The value in any node is greater than the value in any node in its left subtree and less than the value in any node in its right subtree
Binary Search Tree

Figure 9.18  A binary search tree
Binary Search Tree

User's data

Pointer to the root of the left subtree  

Pointer to the root of the right subtree

Page 305
**Boolean Binary Search (first, last)**

If (last > first)
   return false

Else
   Set middle to (first + last) / 2
   Set result to list[middle].compareTo(item)
   If (result is equal to 0)
      return true
   Else
      If (result < 0)
         Binary Search (first, middle - 1)
      Else
         Binary Search (middle + 1, last)
Insert (current, item)

If (tree is null)
   Put item in tree

Else
   If (item.compareTo(info(current)) < 0)
      Insert (item, left(current))
   Else
      Insert (item, right(current))
Graphs

- **Graph**  A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other

- **Undirected graph**  A graph in which the edges have no direction

- **Directed graph (Digraph)**  A graph in which each edge is directed from one vertex to another (or the same) vertex
Graphs

Figure 9.21
Examples of graphs

(a) Vertices: People
Edges: Siblings
Graphs

Figure 9.21  Examples of graphs
(b) Vertices: Cities
   Edges: Direct Flights
Graphs

Figure 9.21
Examples of graphs

(c) Vertices: Courses
Edges: Prerequisites