Chapter 8: Data Abstractions

• 8.1 Data Structure Fundamentals
• 8.2 Implementing Data Structures
• 8.3 A Short Case Study
• 8.4 Customized Data Types
• 8.5 Classes and Objects
• 8.6 Pointers in Machine Language
Basic Data Structures

- Homogeneous array
- Heterogeneous array
- List
  - Stack
  - Queue
- Tree

Figure 8.1 Lists, stacks, and queues

- A list of names
- A stack of books
- A queue of people
**Terminology for Lists**

- **List**: A collection of data whose entries are arranged sequentially
- **Head**: The beginning of the list
- **Tail**: The end of the list

**Terminology for Stacks**

- **Stack**: A list in which entries are removed and inserted only at the head
- **LIFO**: Last-in-first-out
- **Top**: The head of list (stack)
- **Bottom** or **base**: The tail of list (stack)
- **Pop**: To remove the entry at the top
- **Push**: To insert an entry at the top
Terminology for Queues

- **Queue**: A list in which entries are removed at the head and are inserted at the tail
- **FIFO**: First-in-first-out
Terminology for a Tree

- **Tree**: A collection of data whose entries have a hierarchical organization
- **Node**: An entry in a tree
- **Root node**: The node at the top
- **Terminal or leaf node**: A node at the bottom

Terminology for a Tree (continued)

- **Parent**: The node immediately above a specified node
- **Child**: A node immediately below a specified node
- **Ancestor**: Parent, parent of parent, etc.
- **Descendent**: Child, child of child, etc.
- **Siblings**: Nodes sharing a common parent
Terminology for a Tree (continued)

- **Binary tree**: A tree in which every node has at most two children
- **Depth**: The number of nodes in longest path from root to leaf

Figure 8.3  **Tree terminology**
Additional Concepts

- Static Data Structures: Size and shape of data structure does not change
- Dynamic Data Structures: Size and shape of data structure can change
- Pointers: Used to locate data

Figure 8.4  Novels arranged by title but linked according to authorship
Storing Arrays

• Homogeneous arrays
  – **Row-major order** versus **column major order**
  – Address polynomial

• Heterogeneous arrays
  – Components can be stored one after the other in a contiguous block
  – Components can be stored in separate locations identified by pointers

Figure 8.5  The array of temperature readings stored in memory starting at address x
Figure 8.6 A two-dimensional array with four rows and five columns stored in row major order

Figure 8.7 Storing the heterogeneous array Employee

b. Array components stored in separate locations
Storing Lists

- **Contiguous list**: List stored in a homogeneous array
- **Linked list**: List in which each entries are linked by pointers
  - **Head pointer**: Pointer to first entry in list
  - **NIL pointer**: A “non-pointer” value used to indicate end of list

Figure 8.8 **Names stored in memory as a contiguous list**
Figure 8.9 The structure of a linked list

Figure 8.10 Deleting an entry from a linked list
Figure 8.11 Inserting an entry into a linked list

Storing Stacks and Queues

- Stacks usually stored as contiguous lists
- Queues usually stored as **Circular Queues**
  - Stored in a contiguous block in which the first entry is considered to follow the last entry
  - Prevents a queue from crawling out of its allotted storage space
Figure 8.12 A stack in memory

Figure 8.13 A queue implementation with head and tail pointers
Storing Binary Trees

• Linked structure
  – Each node = data cells + two child pointers
  – Accessed via a pointer to root node

• Contiguous array structure

Figure 8.14 A circular queue containing the letters P through V
Figure 8.15  The structure of a node in a binary tree

<table>
<thead>
<tr>
<th>Cells containing the data</th>
<th>Left child pointer</th>
<th>Right child pointer</th>
</tr>
</thead>
</table>

Figure 8.16  The conceptual and actual organization of a binary tree using a linked storage system

Conceptual tree

Actual storage organization

Copyright © 2012 Pearson Education, Inc.
**Figure 8.17** A tree stored without pointers

Conceptual tree

```
A
  /   \
B     C
  /   \
D     E
```

Actual storage organization

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Root node
Nodes in 2nd level of tree
Nodes in 3rd level of tree

**Figure 8.18** A sparse, unbalanced tree shown in its conceptual form and as it would be stored without pointers

Conceptual tree

```
A
  /   \
B     C
    \
    D
    |
    E
```

Actual storage organization

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
</tbody>
</table>
Manipulating Data Structures

• Ideally, a data structure should be manipulated solely by pre-defined procedures.
  – Example: A stack typically needs at least push and pop procedures.
  – The data structure along with these procedures constitutes a complete abstract tool.

Figure 8.19 A procedure for printing a linked list

```
procedure PrintList (List)
CurrentPointer ← head pointer of List.
while (CurrentPointer is not NIL) do
    (Print the name in the entry pointed to by CurrentPointer;
     Observe the value in the pointer cell of the List entry
     pointed to by CurrentPointer, and reassign CurrentPointer
     to be that value.)
```

Copyright © 2012 Pearson Education, Inc.
Case Study

Problem: Construct an abstract tool consisting of a list of names in alphabetical order along with the operations search, print, and insert.
Figure 8.21  The binary search as it would appear if the list were implemented as a linked binary tree

procedure Search(Tree, TargetValue)
    If (root pointer of Tree = NIL)
        then
            (declare the search a failure)
        else
            (execute the block of instructions below that is associated with the appropriate case)
            case 1: TargetValue = value of root node
                    (Report that the search succeeded)
            case 2: TargetValue < value of root node
                    (Apply the procedure Search to see if TargetValue is in the subtree identified by the root’s left child pointer and report the result of that search)
            case 3: TargetValue > value of root node
                    (Apply the procedure Search to see if TargetValue is in the subtree identified by the root’s right child pointer and report the result of that search)
        end if

Figure 8.22  The successively smaller trees considered by the procedure in Figure 8.18 when searching for the letter J
Figure 8.23 **Printing a search tree in alphabetical order**

![Diagram of a search tree with nodes labeled A through J and instructions: 1. Print the left branch in alphabetical order, 2. Print the root node, 3. Print the right branch in alphabetical order.]

Figure 8.24 **A procedure for printing the data in a binary tree**

```plaintext
procedure PrintTree (Tree)
if (Tree is not empty)
    then (Apply the procedure PrintTree to the tree that appears as the left branch in Tree;
           Print the root node of Tree;
           Apply the procedure PrintTree to the tree that appears as the right branch in Tree)
```

Copyright © 2012 Pearson Education, Inc.
Figure 8.25  Inserting the entry M into the list B, E, G, H, J, K, N, P stored as a tree

1a. Search for the new entry until its absence is detected

```
    E
   /|
  B G
 /  |
J   N
```

1b. This is the position in which the new entry should be attached

```
    H
   /|
  B G
 /  |
J   N
   /|
  K M
```

Figure 8.26  A procedure for inserting a new entry in a list stored as a binary tree

```
procedure Insert(Tree, NewValue)
    if (root pointer of Tree = NIL)
        (set the root pointer to point to a new leaf containing NewValue)
    else (execute the block of instructions below that is associated with the appropriate case)
        case 1: NewValue = value of root node
            (Do nothing)
        case 2: NewValue < value of root node
            if (left child pointer of root node = NIL)
                then (set that pointer to point to a new leaf node containing NewValue)
                else (apply the procedure Insert to insert NewValue into the subtree identified by the left child pointer)
        case 3: NewValue > value of root node
            if (right child pointer of root node = NIL)
                then (set that pointer to point to a new leaf node containing NewValue)
                else (apply the procedure Insert to insert NewValue into the subtree identified by the right child pointer)
    end if
```
User-defined Data Type

- A template for a heterogeneous structure
- Example:

```plaintext
define type EmployeeType to be
{char    Name[25];
 int     Age;
 real    SkillRating;
}
```

Abstract Data Type

- A user-defined data type with procedures for access and manipulation
- Example:

```plaintext
define type StackType to be
{int StackEntries[20];
 int StackPointer = 0;
 procedure push(value)
 {StackEntries[StackPointer] ← value;
  StackPointer ← StackPointer + 1;
 }
 procedure pop . . .
}
```
Class

- An abstract data type with extra features
  - Characteristics can be inherited
  - Contents can be encapsulated
  - Constructor methods to initialize new objects

```java
class StackOfIntegers {
    private int[] StackEntries = new int[20];
    private int StackPointer = 0;

    public void push(int NewEntry) {
        if (StackPointer < 20) {
            StackEntries[StackPointer++] = NewEntry;
        }
    }

    public int pop() {
        if (StackPointer > 0) return StackEntries[--StackPointer];
        else return 0;
    }
}
```

Figure 8.27  A stack of integers implemented in Java and C#
Pointers in Machine Language

- **Immediate addressing**: Instruction contains the data to be accessed
- **Direct addressing**: Instruction contains the address of the data to be accessed
- **Indirect addressing**: Instruction contains the location of the address of the data to be accessed

Figure 8.28  Our first attempt at expanding the machine language in Appendix C to take advantage of pointers
Loading a register from a memory cell that is located by means of a pointer stored in a register.