Fundamentals II

ADT, Lists, Stacks and Recurrence

Abstraction

- Abstraction
  - Separates the purpose from implementation
  - Specify what to do, not how to do it!
  - Modularity and Abstraction complement
- Possible to use a module without knowing implementation
  - Specifications for a module are written before implementation
- Think “what” not “how”

Abstraction

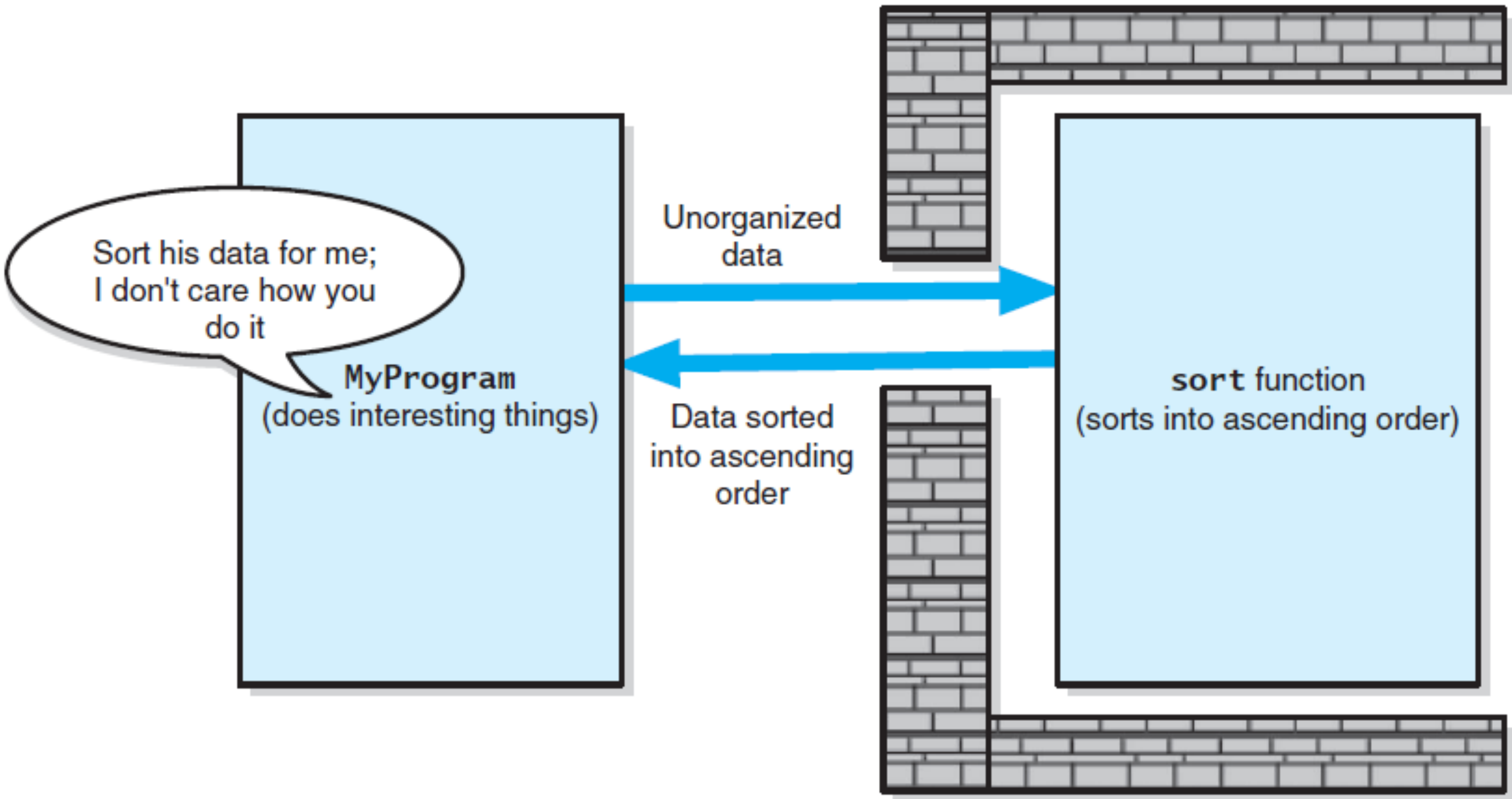


FIGURE Tasks communicate through a slit in wall

Abstraction – Functional & Data

- Functional abstraction:
  - Separate the purpose of a module from its implementation
- Data Abstraction:
  - Focus on operations on data, not the implementation of the operations
  - E.g Array, a fundamental data abstraction. Use it but don't worry about how it is implemented

Abstract Data Type

- A collection of data and
- A set of operations on the data.

- Carefully specify an ADT's operations before you implement them
- Design of an ADT will evolve
- You can use an ADT's operations without knowing their implementation or how data is stored, if you know the operation's spec

Abstract Data Type

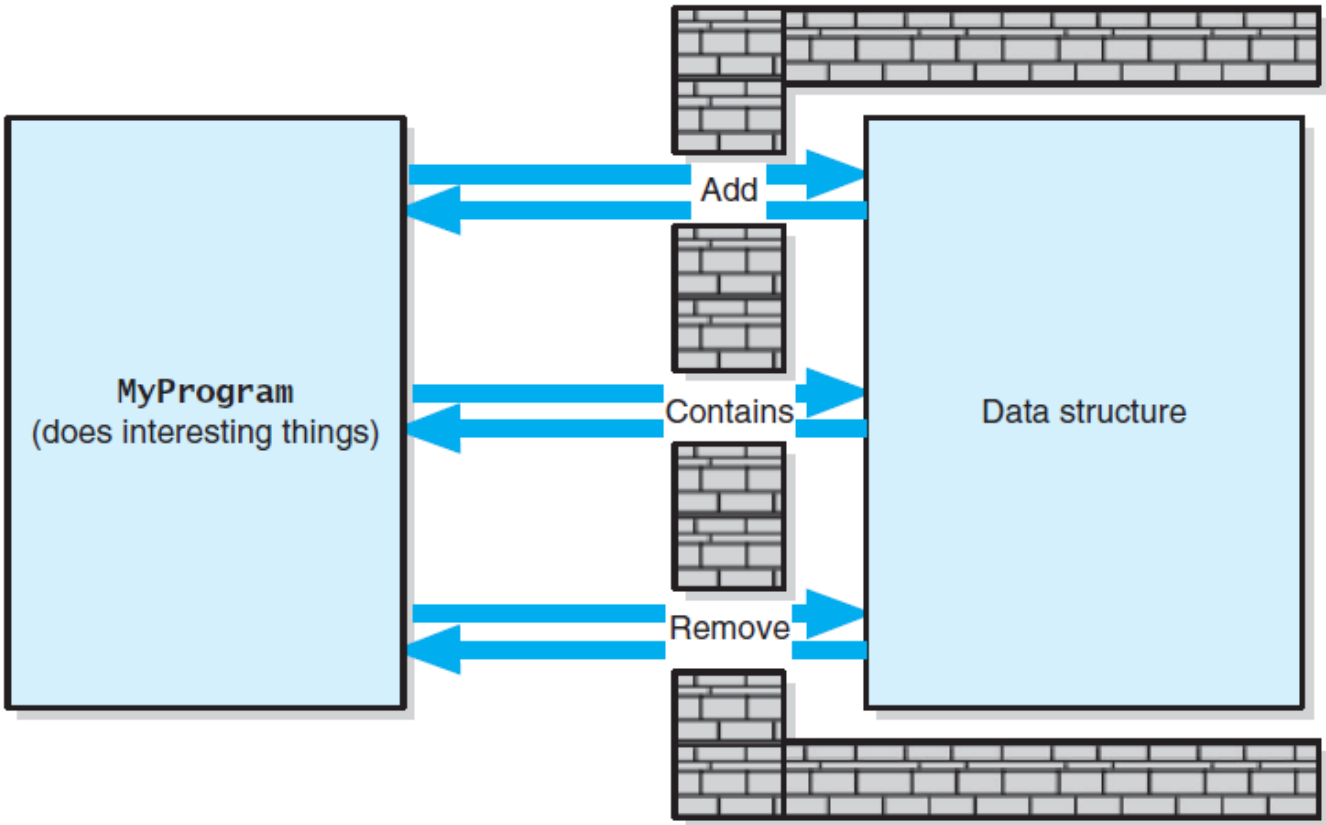


FIGURE A wall of ADT operations isolates a data structure from the program that uses it

Designing an ADT

Ask the questions

- What data does the problem require?
  - Names
  - IDs
  - Numerical data
- What operations will be done on that data?
  - Initialize
  - Display
  - Calculations

ADT vs Data Structure

- Data Structure:
  - A construct that you define within a programming language to store a collection of data
    - E.g. Array of integers or array of objects

The ADT Bag

- A bag is a container
  - Contains finite number of data objects
  - All objects of same type
  - Objects in no particular order
  - Objects may be duplicated

Identifying Behaviors

- Get the number of items currently in the bag.
- See whether the bag is empty.
- Add a given object to bag.
- Remove occurrence of specific object from bag
- Remove all objects from bag.

Identifying Behaviors

- Count the number of times certain object occurs in bag.
- Test whether bag contains particular object.
- Look at all objects in bag.

Identifying Behaviors

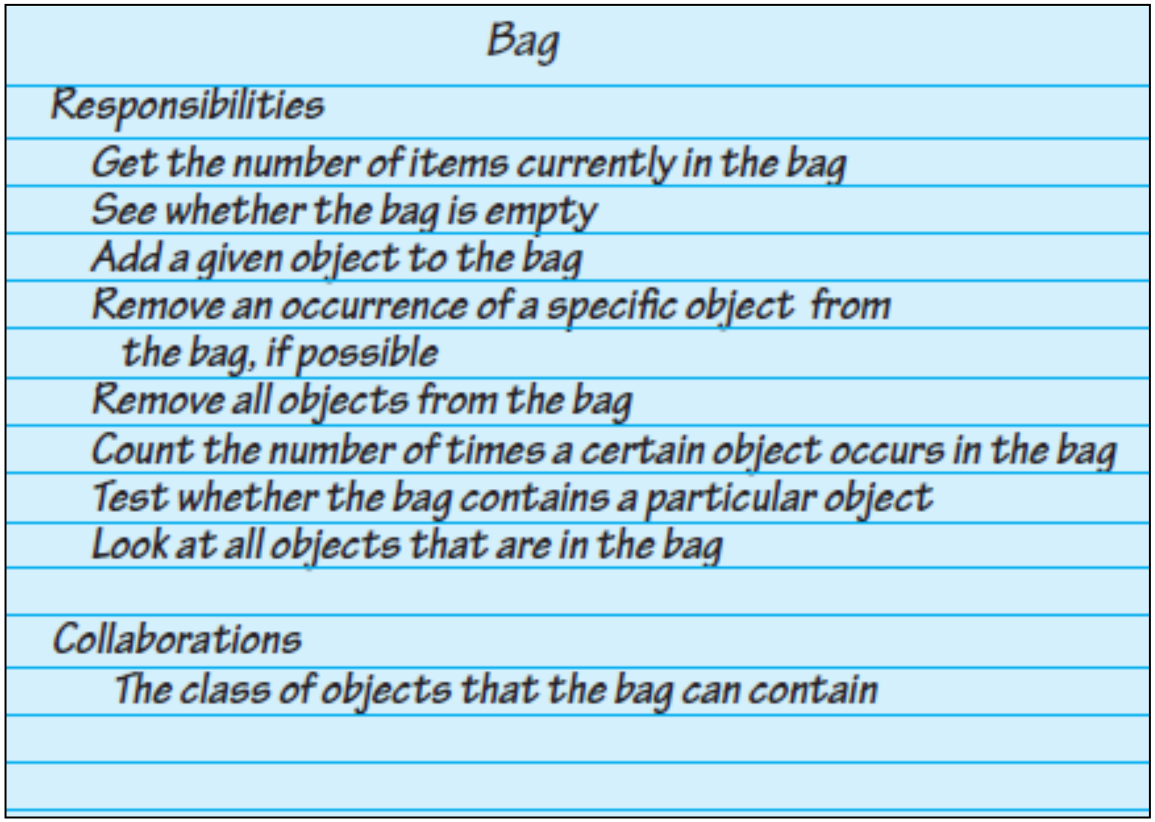


FIGURE A Class-Responsibility-Collaboration (CRC) card for a class **`Bag`**

Specifying Data and Operations

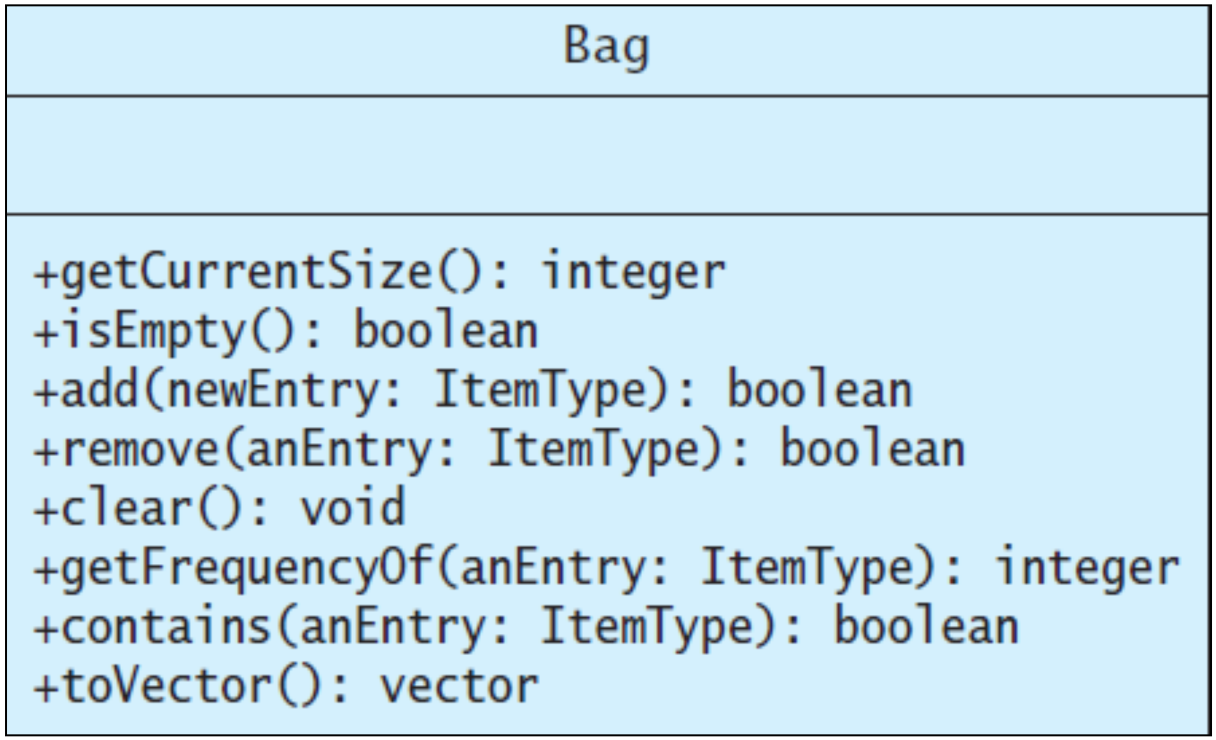


FIGURE 1-7 UML notation for the class **`Bag`**

Lists

Contents

- Specifying the ADT List
- Using the List Operations
- An Interface Template for the ADT List

Specifying the ADT List

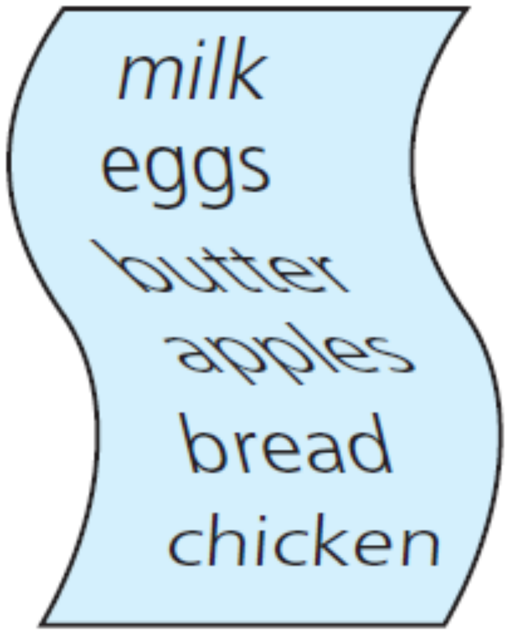


FIGURE 8-1 A grocery list

ADT List Operations

- Test whether a list is empty.
- Get number of entries on a list.
- Insert entry at given position on list.
- Remove entry at given position from list.
- Remove all entries from list.
- Look at (get) entry at given position on list.
- Replace (set) entry at given position on list.

ADT List Operations

List
+isEmpty(): boolean +getLength(): integer +insert(newPosition: integer, newEntry: ItemType): boolean +remove(position: integer): boolean +clear(): void +getEntry(position: integer): ItemType +setEntry(position: integer, newEntry: ItemType): void

FIGURE UML diagram for the ADT list

Abstract Data Type: LIST

- List: A finite number of objects
  - Not necessarily distinct
  - Having the same data type
  - Ordered by their positions determined by client.

Abstract Data Type: LIST

- Operations
  - isEmpty()
  - getLength()
  - insert(newPosition, newEntry)
  - remove(position)
  - clear()
  - getEntry(position)
  - setEntry(position, newEntry)

Using the List Operations

- Displaying the items on a list independent of the implementation

```
// Displays the items on the list aList.
displayList(aList)

   for (position = 1 through aList.getLength())
   {
      dataItem = aList.getEntry(position)
      Display dataItem
   }
```

Using the List Operations

- Replacing an item.

```
// Replaces the ith entry in the list aList with newEntry.
// Returns true if the replacement was successful; otherwise return false.
replace(aList, i, newEntry)

    success = aList.remove(i)
    if (success)
        success = aList.insert(i, newItem)

    return success
```

List Implementations

Contents

- An Array-Based Implementation of the ADT List
- A Link-Based Implementation of the ADT List
- Comparing Implementations

Array-Based Implementation of ADT List

- Recall list operations in UML form

```
+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+setEntry(position: integer, newEntry: ItemType): void
```

The Header File

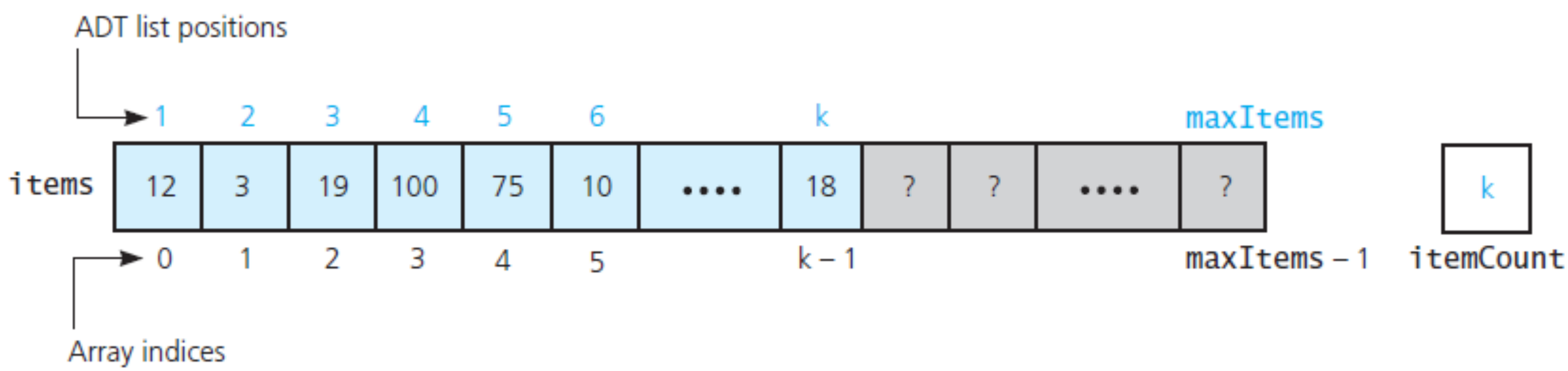


FIGURE 9 An array-based implementation of the ADT list

The Implementation File

- Definition of the method **`insert`**

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition,
                                 const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) &&
                       (newPosition <= itemCount + 1) &&
                       (itemCount < maxItems);
   if (ableToInsert)
   {
      // Make room for new entry by shifting all entries at
      // positions >= newPosition toward the end of the array
      // (no shift if newPosition == itemCount + 1)
      for (int pos = itemCount; pos >= newPosition; pos--)
         items[pos] = items[pos - 1];

      // Insert new entry
      items[newPosition - 1] = newEntry;
      itemCount++;  // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

The Implementation File

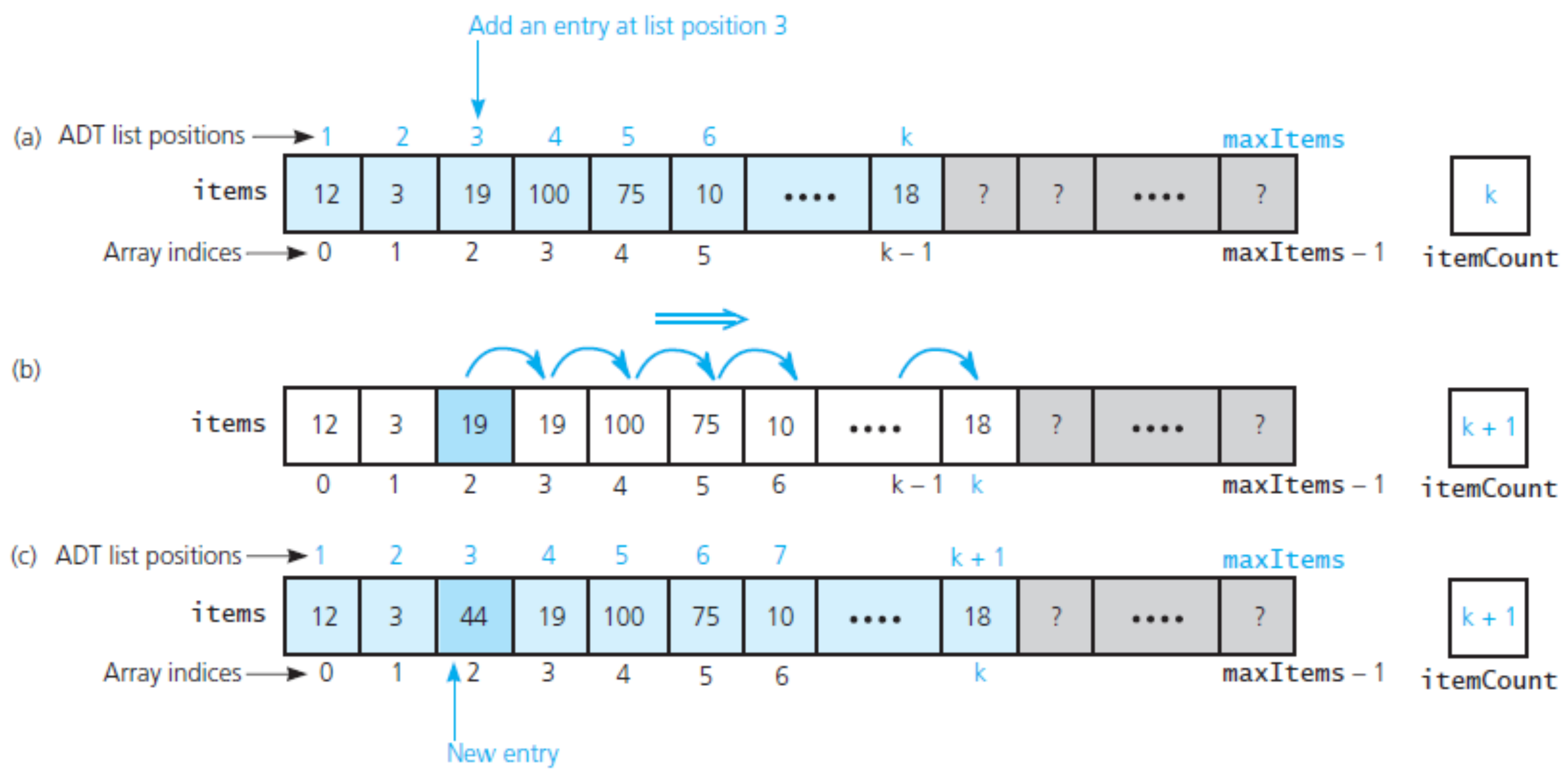


FIGURE Shifting items for insertion: (a) the list before the insertion; (b) copy items to produce room at position 3; (c) the result

The Implementation File

- The definition of **`remove`**

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      // Remove entry by shifting all entries after the one at
      // position toward the beginning of the array
      // (no shift if position == itemCount)
      for (int fromIndex = position, toIndex = fromIndex - 1;
               fromIndex < itemCount; fromIndex++, toIndex++)
         items[toIndex] = items[fromIndex];

      itemCount--;  // Decrease count of entries
   }  // end if

    return ableToRemove;
}  // end remove
```

The Implementation File

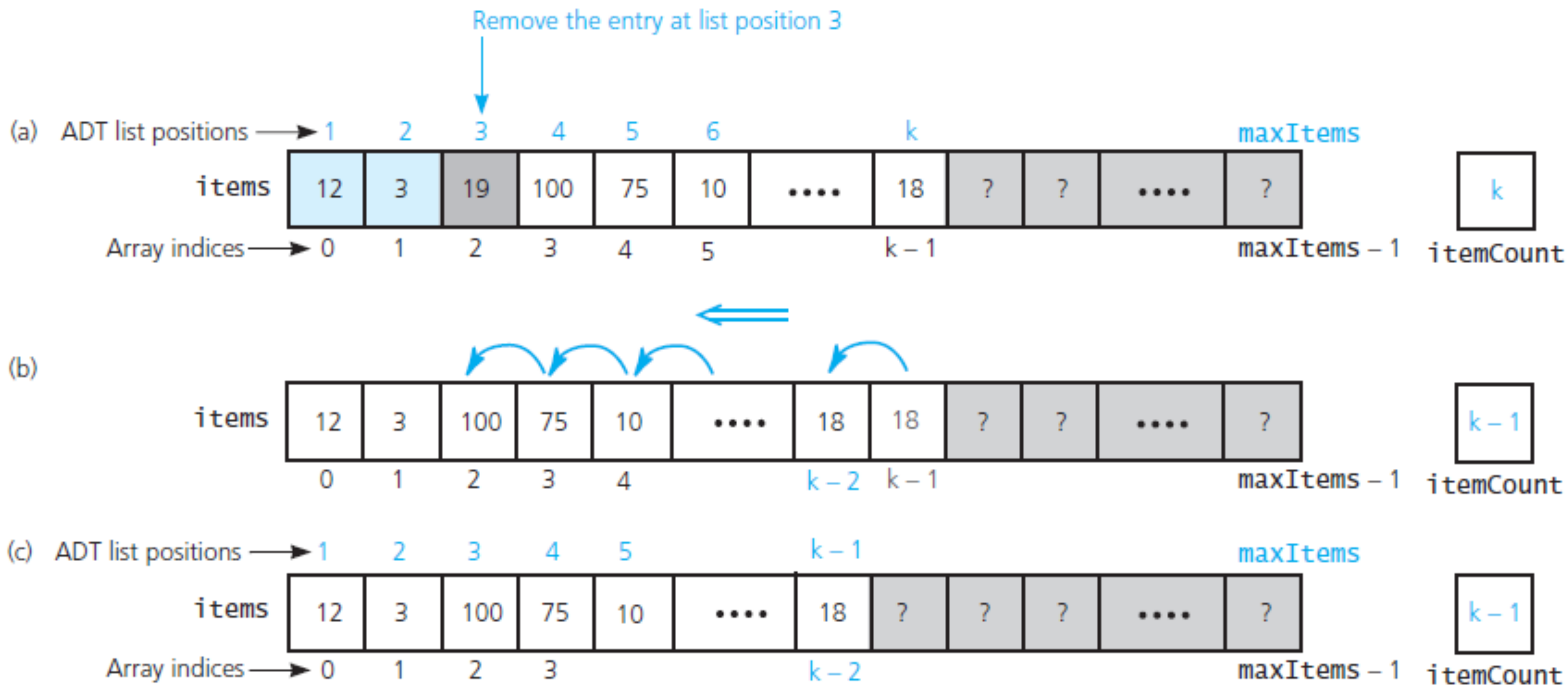


FIGURE 9-3 (a) Deletion can cause a gap; (b) shift items to prevent a gap at position 3; (c) the result

A Link-Based Implementation of the ADT List

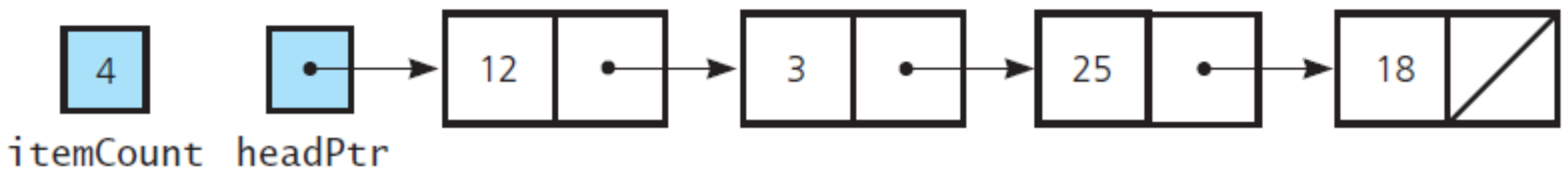


FIGURE A link-based implementation of the ADT list

Stacks

Contents

- The Abstract Data Type Stack
- Simple Uses of a Stack
- Using Stacks with Algebraic Expressions
- Using a Stack to Search a Flight Map
- The Relationship Between Stacks and Recursion

The Abstract Data Type Stack

- Developing an ADT during the design of a solution
- Consider entering keyboard text
  - Mistakes require use of backspace

    `abcdd←←efgg←`
- We seek a programming solution to read these keystrokes

The Abstract Data Type Stack

- Pseudocode of first attempt

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else
        Remove from the ADT (discard) the item that was added most recently
}
```

- Requires
  - Add new item to ADT
  - Remove most recently added item

Specifications for the ADT

- We have identified the following operations:
  - See whether stack is empty.
  - Add a new item to stack.
  - Remove from the stack item added most recently.
  - Get item that was added to stack most recently.
- Stack uses LIFO principle
  - Last In First Out

Specifications for the ADT

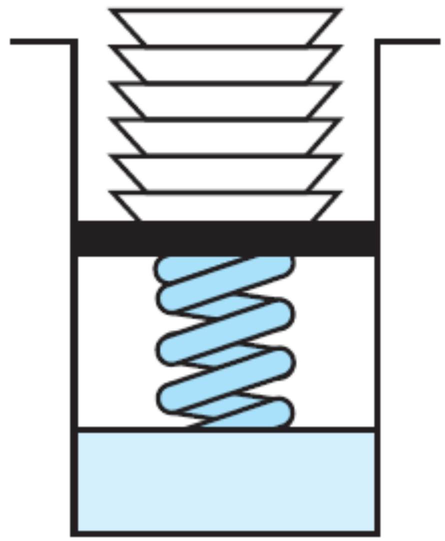


FIGURE A stack of cafeteria plates

Abstract Data Type:

- A finite number of objects
  - Not necessarily distinct
  - Having the same data type
  - Ordered by when they were added
- Operations
  - **`isEmpty()`**
  - **`push(newEntry)`**
  - **`pop()`**
  - **`peek()`**

Abstract Data Type:

Stack
+isEmpty(): boolean +push(newEntry: ItemType): boolean +pop(): boolean +peek(): ItemType

FIGURE UML diagram for the class **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}{l{mn}}op}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 "{"

Simple Uses of a Stack

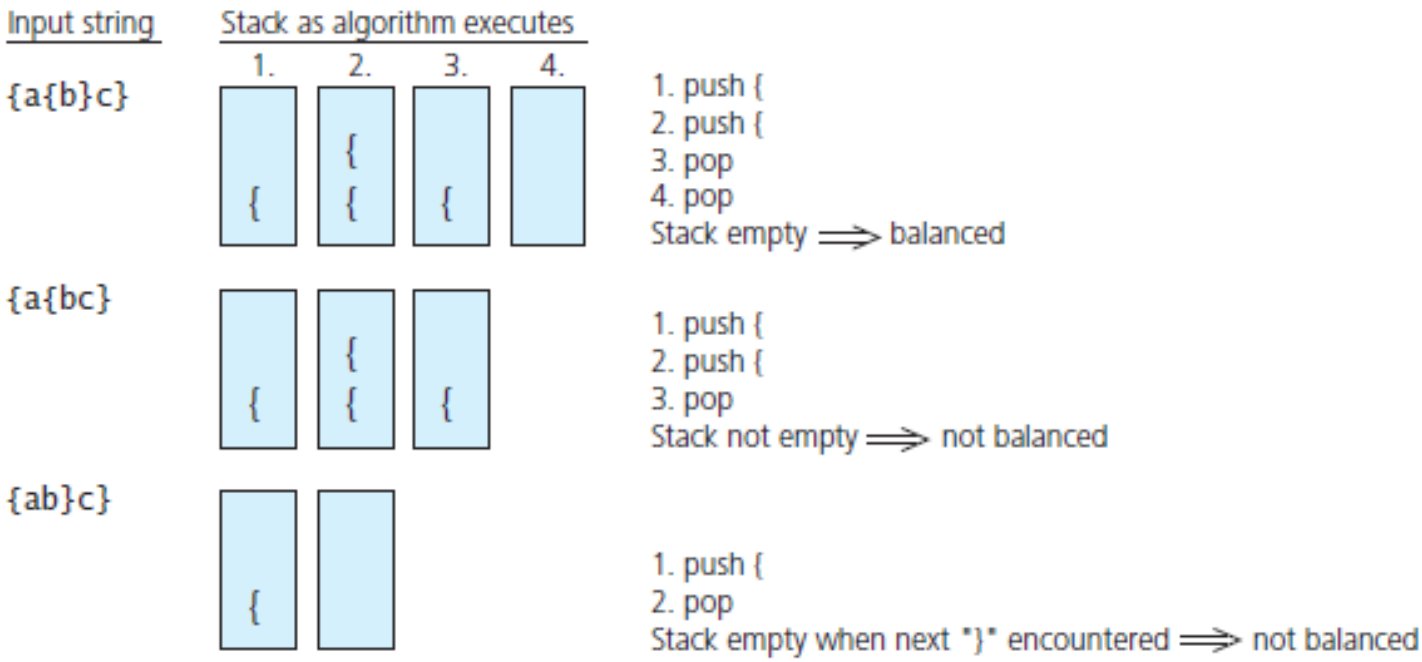


FIGURE Traces of the algorithm that checks for balanced braces

Simple Uses of a Stack

- Recognizing strings in a language
- Consider
  $L = \{s\$s' : s$ is a possibly empty string of characters other than $\$$, $s' =$ reverse(s)$\}$
- A solution using a stack
  - 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, match a popped character off the stack
- View algorithm to verify a string for a given language,

Evaluating Postfix Expressions

- A postfix calculator
  - 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 onto the stack

Using Stacks with Algebraic Expressions

- Evaluating postfix expressions

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

FIGURE The effect of a postfix calculator on a stack when evaluating the expression 2 * (3 + 4)

Evaluating Postfix Expressions

- To evaluate a postfix expression entered as a string of characters
  - Use the same steps as a postfix calculator
  - Simplifying assumptions
    - The string is a syntactically correct postfix expression
    - No unary operators are present
    - No exponentiation operators are present
    - Operands are single lowercase letters that represent integer values

Using Stacks with Algebraic Expressions

- Converting infix expressions to equivalent postfix expressions
- Possible pseudocode solution
- Trace on next slide

```
Initialize postfixExp to the empty string
for (each character ch in the infix expression)
{
    switch (ch)
    {
        case ch is an operand:
            Append ch to the end of postfixExp
            break
        case ch is an operator:
            Save ch until you know where to place it
            break
        case ch is a '(' or a ')':
            Discard ch
            break
    }
}
```

Using Stacks with Algebraic Expressions

ch	aStack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators from stack to
	– (	abcd*+	postfixExp until "("
	–	abcd*+	
/	– /	abcd*+	Copy operators from
e	– /	abcd*+e	stack to postfixExp
		abcd*+e/–	

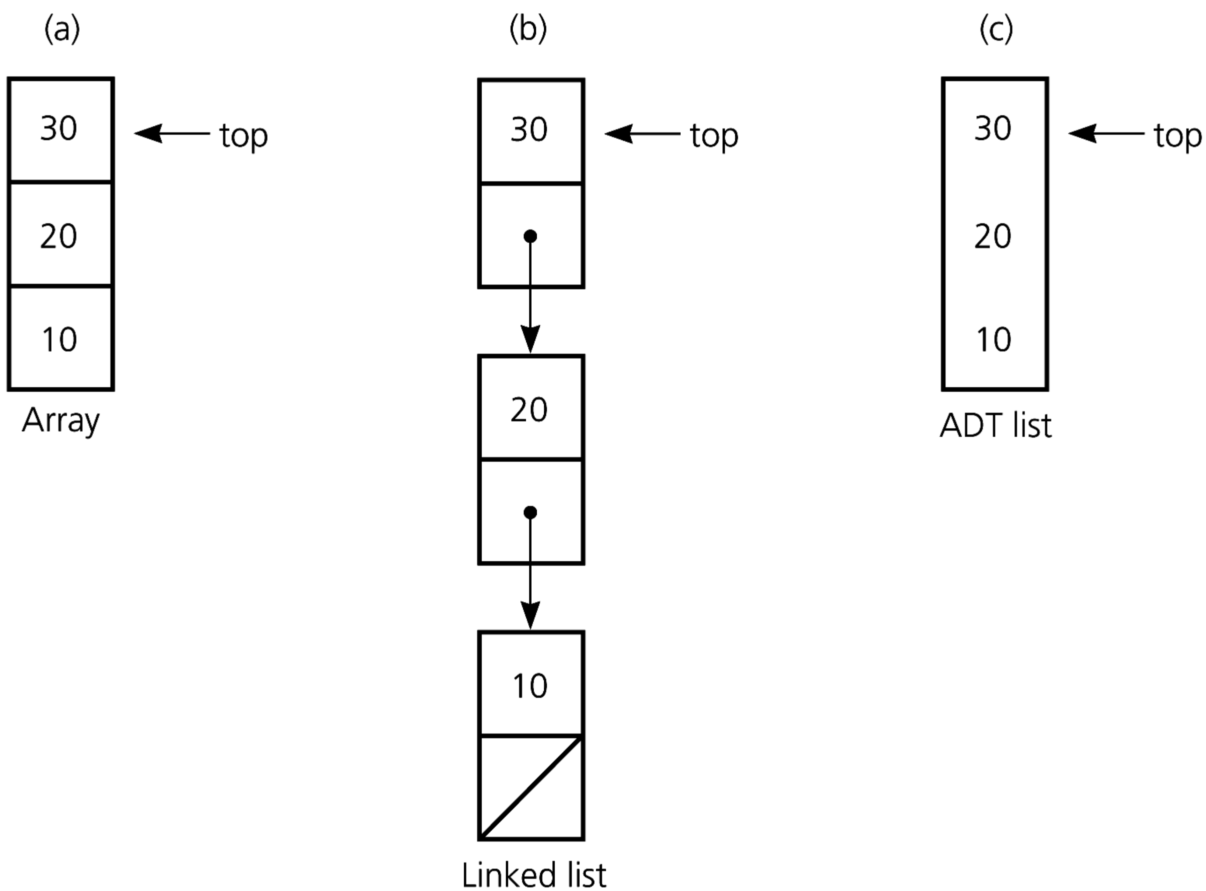
FIGURE 6-5 A trace of the algorithm that converts the infix expression a – (b + c * d) / e to postfix form

Implementations of the ADT Stack

Contents

- An Array-Based Implementation
- A Link-Based implementation
- ADT List

Implementations of the ADT Stack



An Array Based Implementation

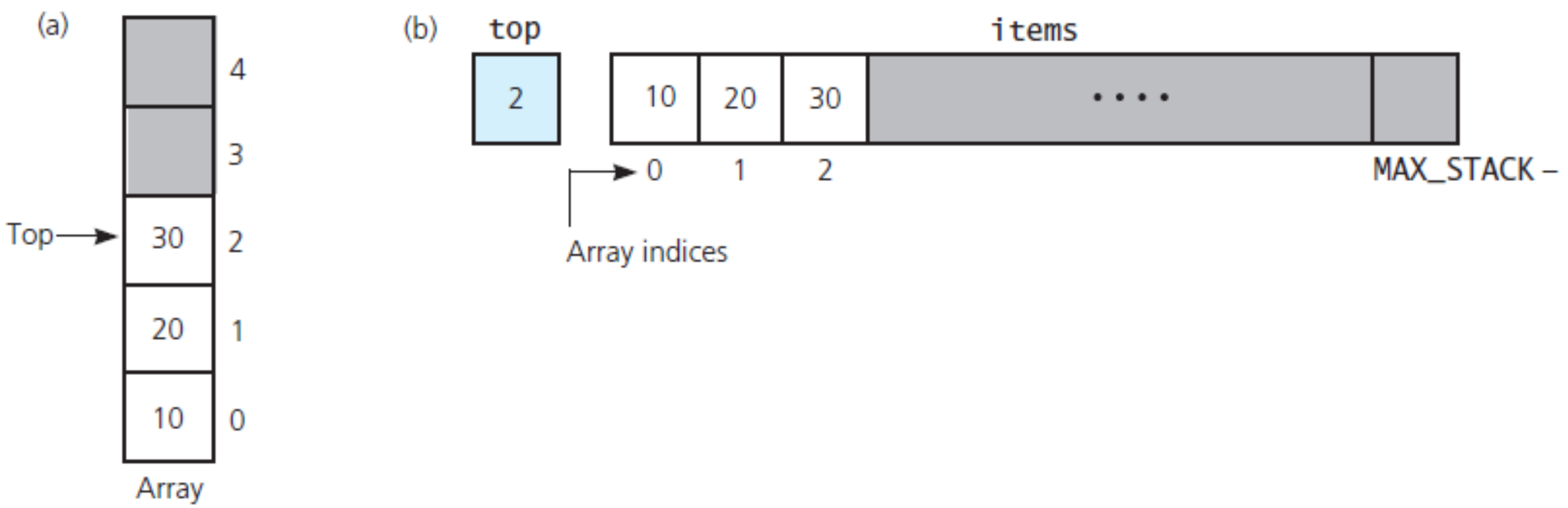


FIGURE 7 Using an array to store a stack's entries: (a) a preliminary sketch; (b) implementation details

A Link-Based implementation

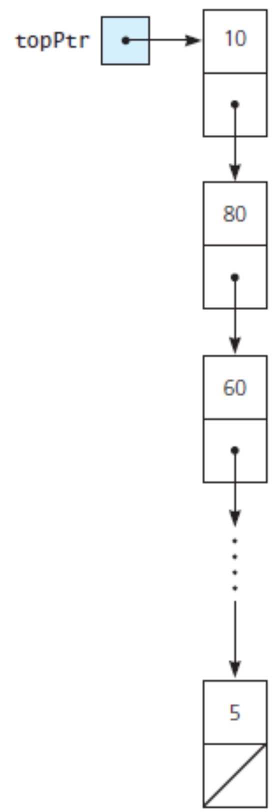
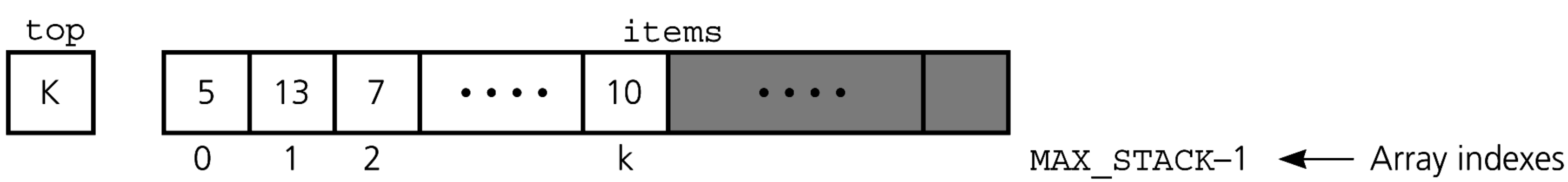


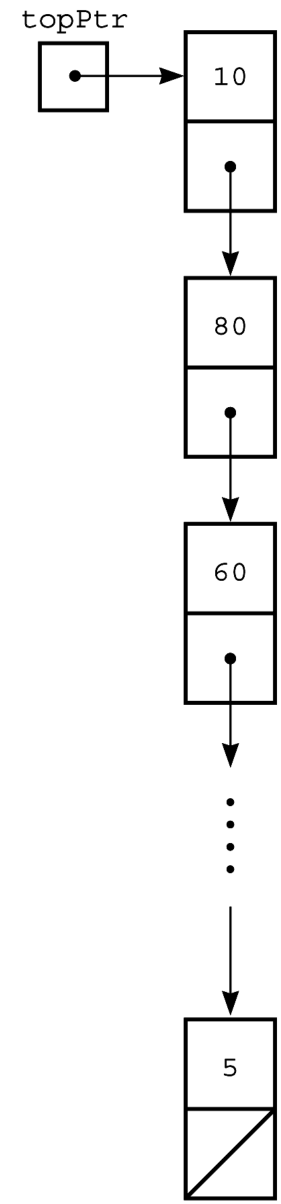
FIGURE A link-based implementation of a stack

An Array-Based Implementation of the ADT Stack

- Stack as a class
- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top` to the top item
- If need to access any element, don't use stack
- Compiler-generated destructor and copy constructor



A Pointer-Based Implementation of the ADT Stack

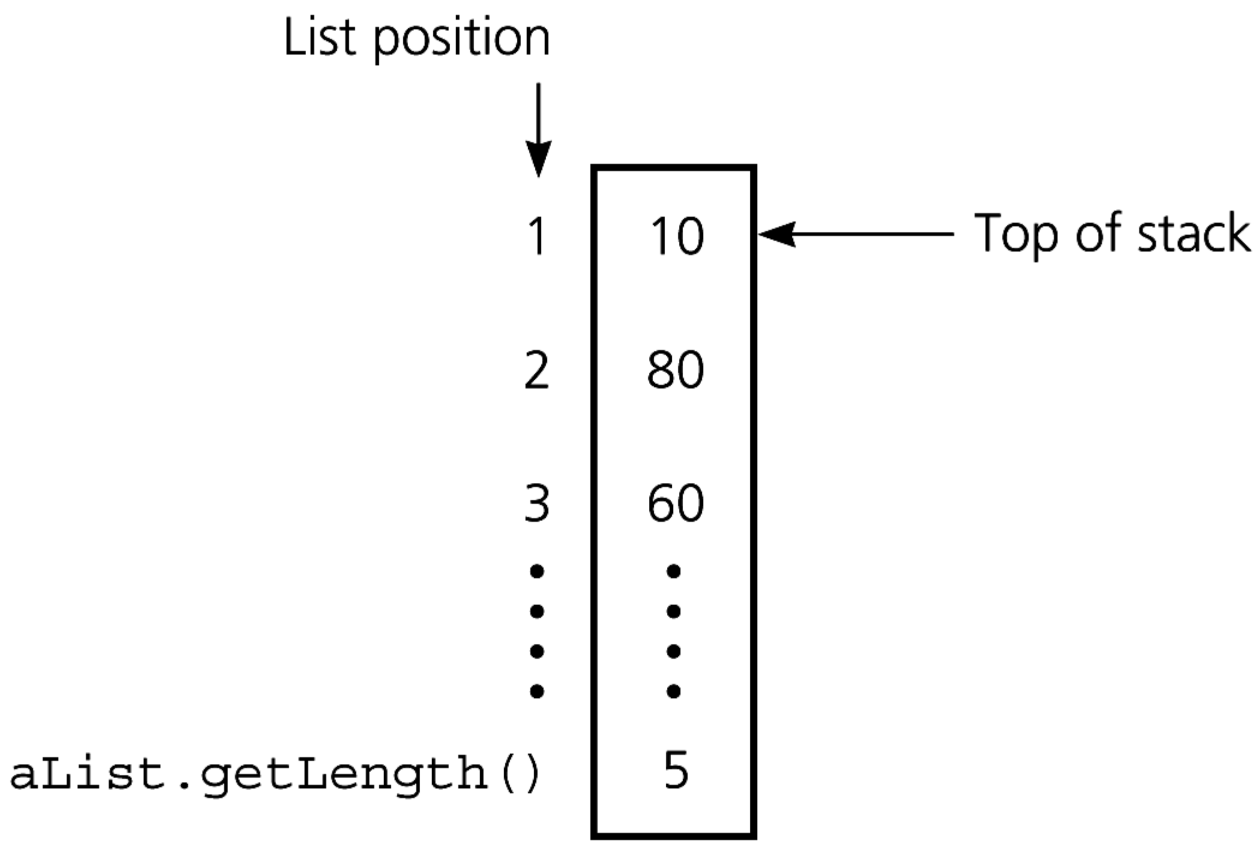


- A pointer-based implementation
  - Enables the stack to grow and shrink dynamically
- `topPtr` is a pointer to the head of a linked list of items
- A copy constructor and destructor must be supplied

An Implementation That Uses the ADT List

- The ADT list can represent the items in a stack
- Let the item in position 1 of the list be the top
  - `push(newItem)`
    - `insert(1, newItem)`
  - `pop()`
    - `remove(1)`
  - `getTop(stackTop)`
    - `retrieve(1, stackTop)`

An Implementation That Uses the ADT List



Comparing Implementations

- Fixed size versus dynamic size
  - A statically allocated array-based implementation
    - Fixed-size stack that can get full
    - Prevents the `push` operation from adding an item to the stack, if the array is full
  - A dynamically allocated array-based implementation or a pointer-based implementation
    - No size restriction on the stack

Comparing Implementations

- A pointer-based implementation vs. one that uses a pointer-based implementation of the ADT list
  - Pointer-based implementation is more efficient
  - ADT list approach reuses an already implemented class
    - Much simpler to write
    - Saves programming time

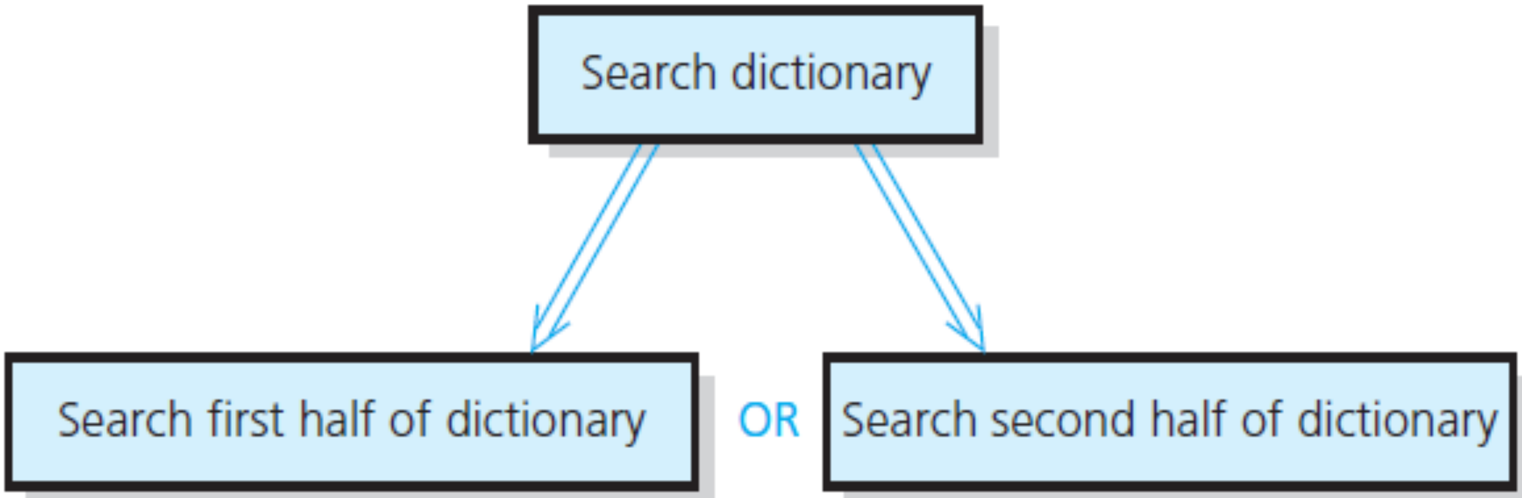
Recursion: The Mirrors

Contents

Recursive Solutions

- Recursion breaks a problem into smaller identical problems
- Some recursive solutions are inefficient, impractical
- Complex problems can have simple recursive solutions

Recursive Solutions



A recursive solution

Recursive Solutions

- A recursive solution calls itself
- Each recursive call solves an identical, smaller problem
- Test for base case enables recursive calls to stop
- Eventually one of smaller calls will be base case

A Recursive Valued Function

The factorial of *n*

```
/** Computes the factorial of the nonnegative integer n.
 @pre  n must be greater than or equal to 0.
 @post  None.
 @return  The factorial of n; n is unchanged. */
int fact(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * fact(n - 1); // n * (n-1)! is n!
}  //  end fact
```

A Recursive Valued Function

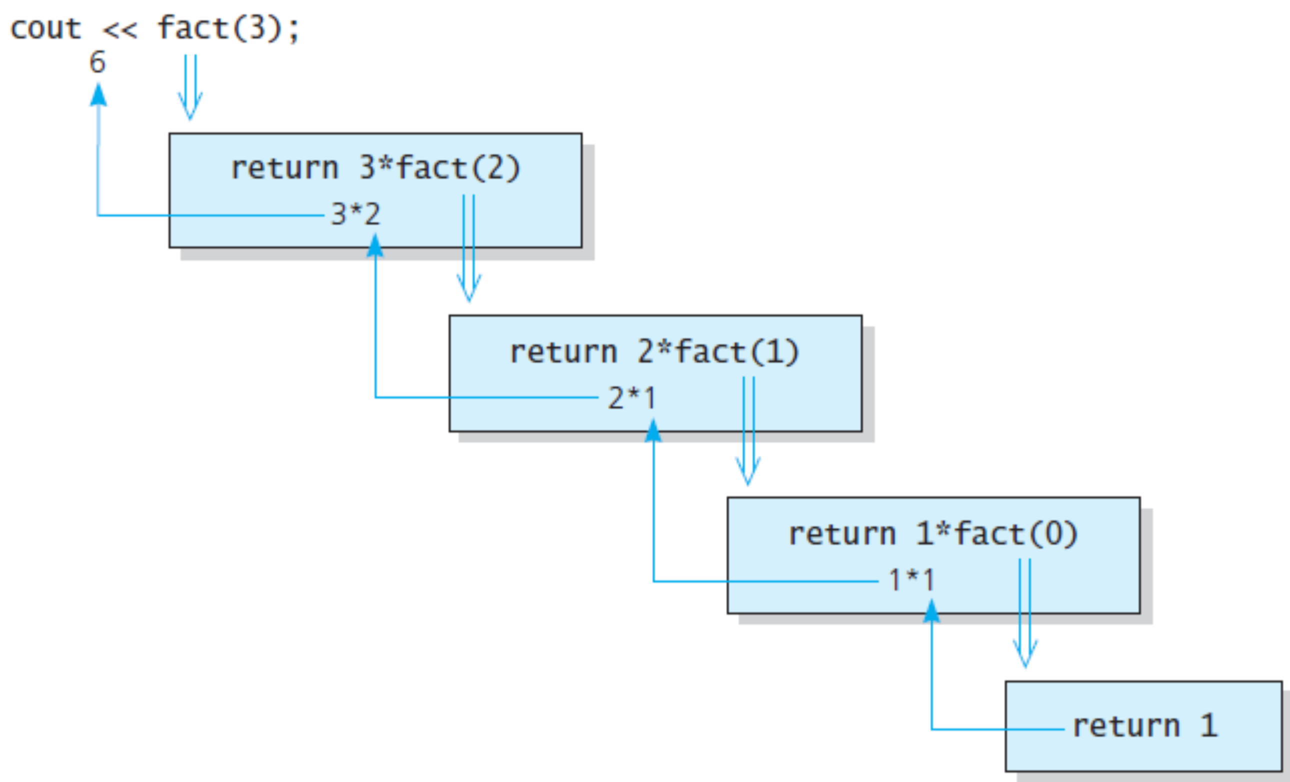


FIGURE **`fact(3)`**

The Box Trace

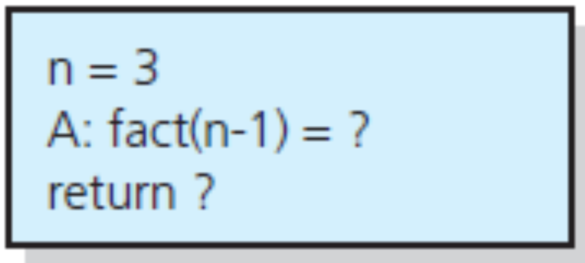
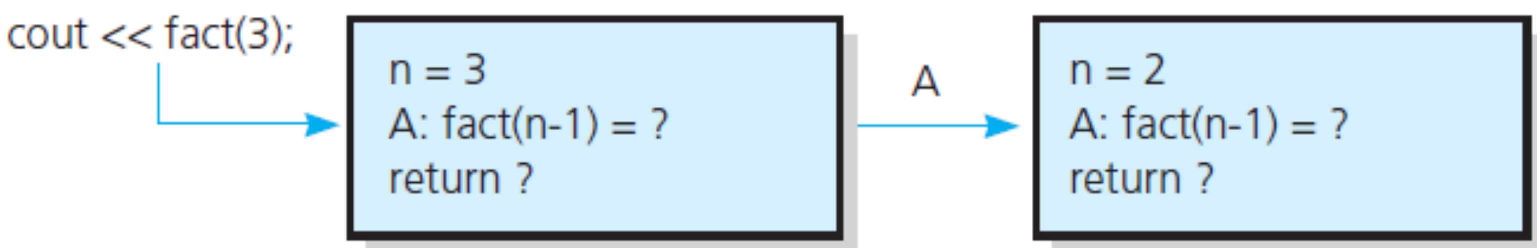


FIGURE 2-3 A box



The Box Trace

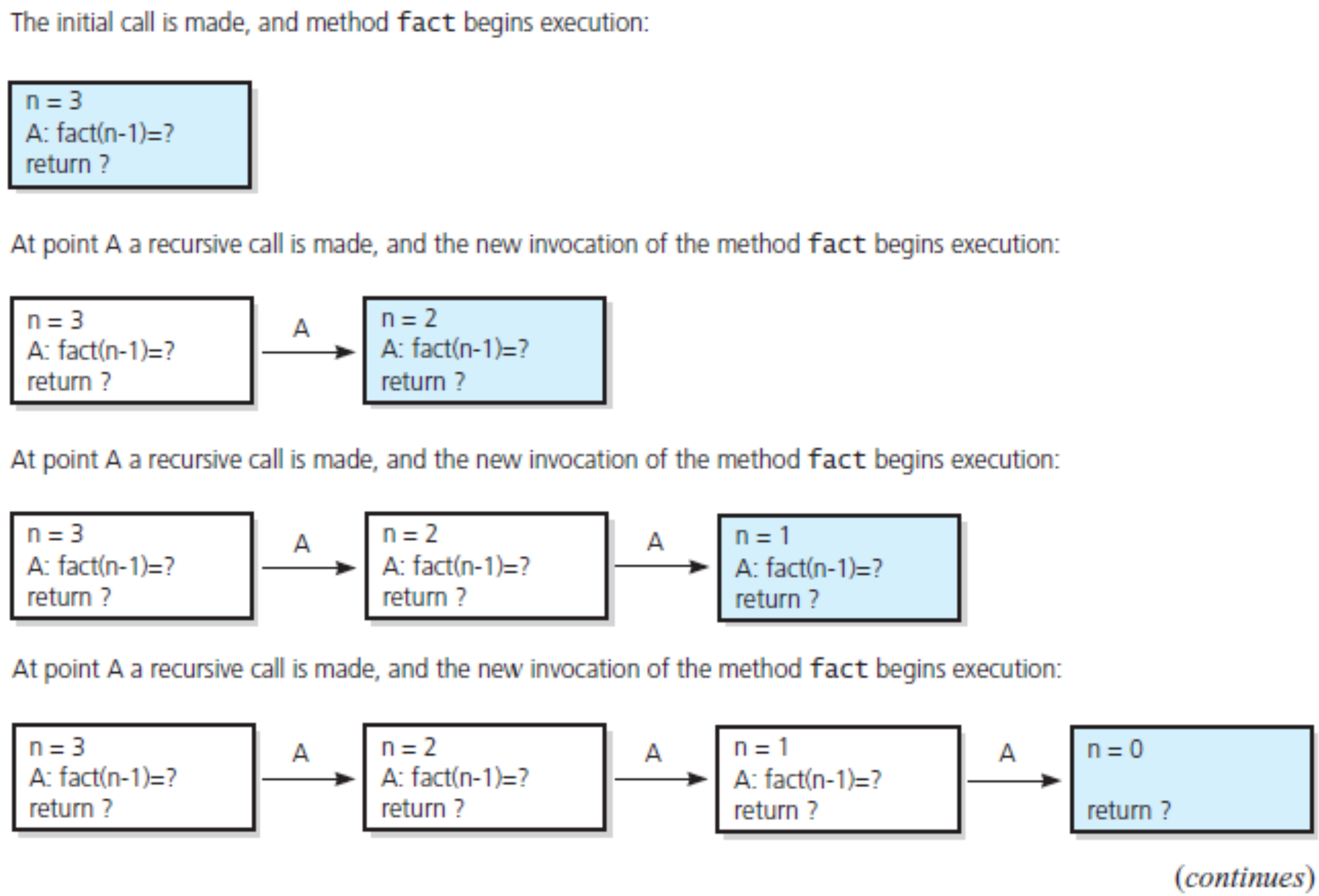


FIGURE 2 Box trace of fact(3)

The Box Trace

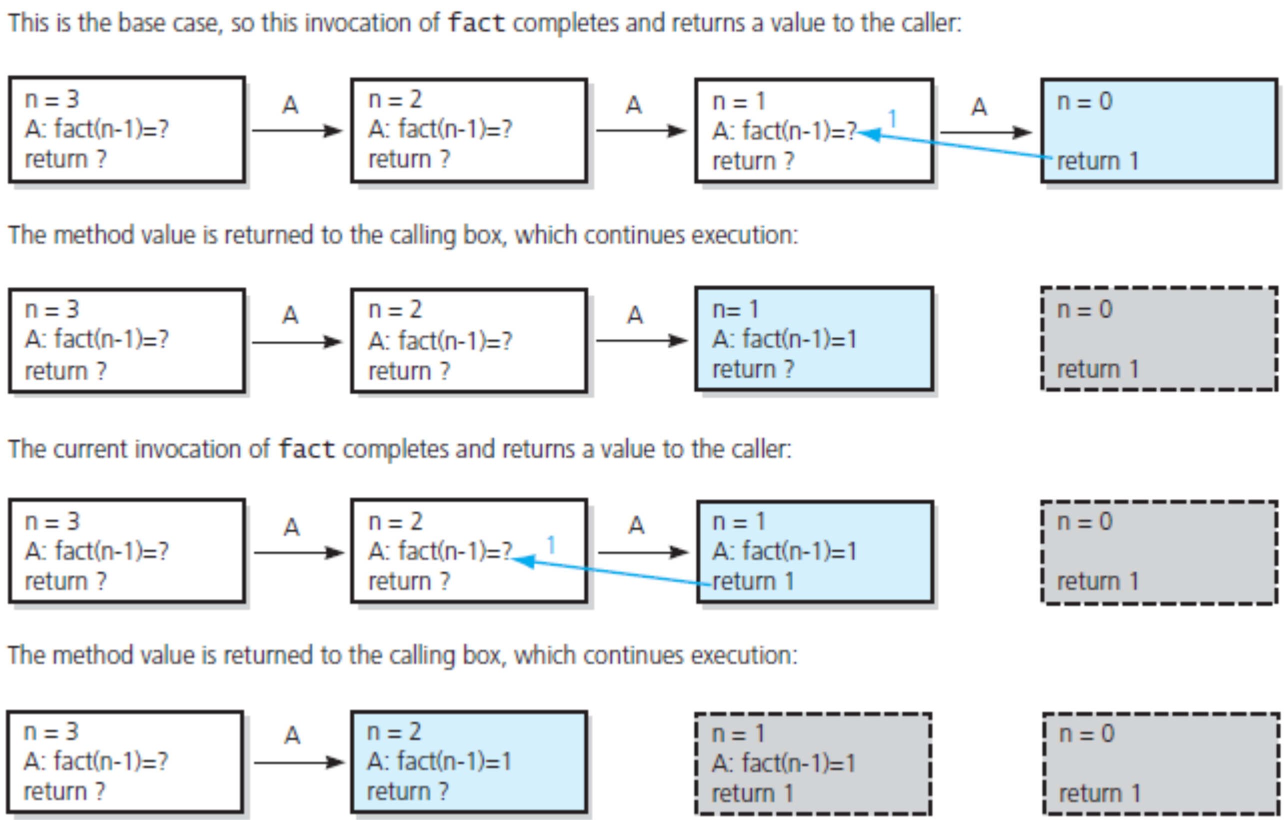


FIGURE 2 Box trace of fact(3) … continued

The Box Trace

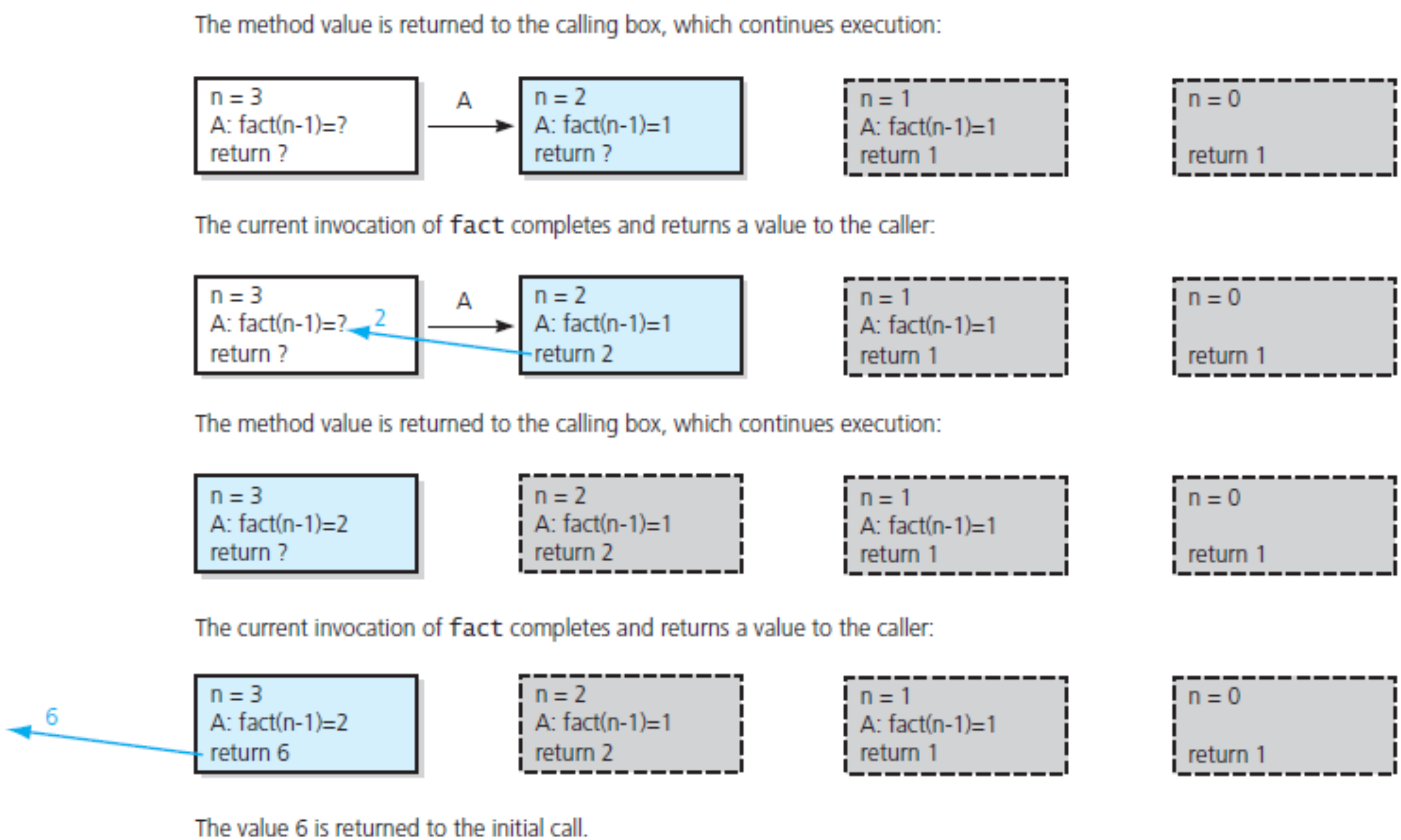


FIGURE 2 Box trace of fact(3) … continued

A Recursive Void Function

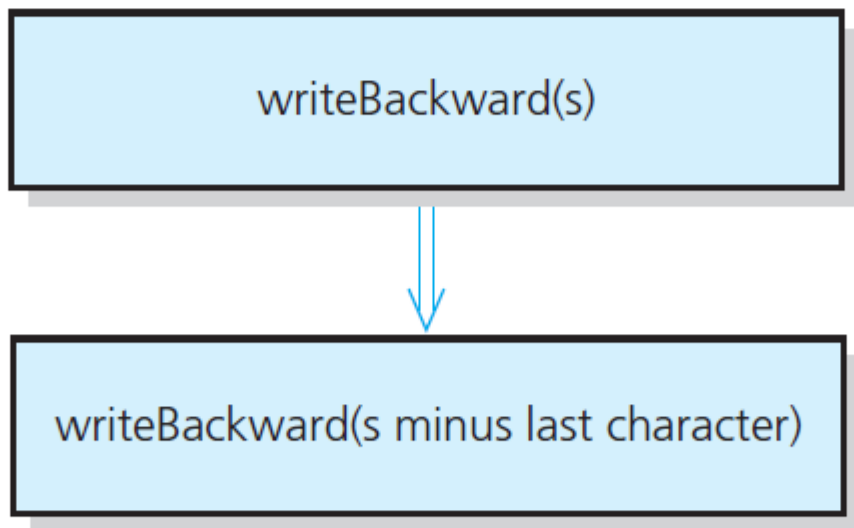


FIGURE 2 A recursive solution

A Recursive Void Function

- The function **`writeBackwards`**

```
/** Writes a character string backward.
 @pre  The string s to write backward.
 @post  None.

  @param s  The string to write backward. */
 void writeBackward(string s)
 {
     int length = s.size(); // Length of string
     if (length > 0)
     {
         // Write the last character
         cout << s.substr(length - 1, 1);

         // Write the rest of the string backward
         writeBackward(s.substr(0, length - 1)); // Point A
     }  // end if

     // length == 0 is the base case - do nothing
 }  // end writeBackward
```

A Recursive Void Function

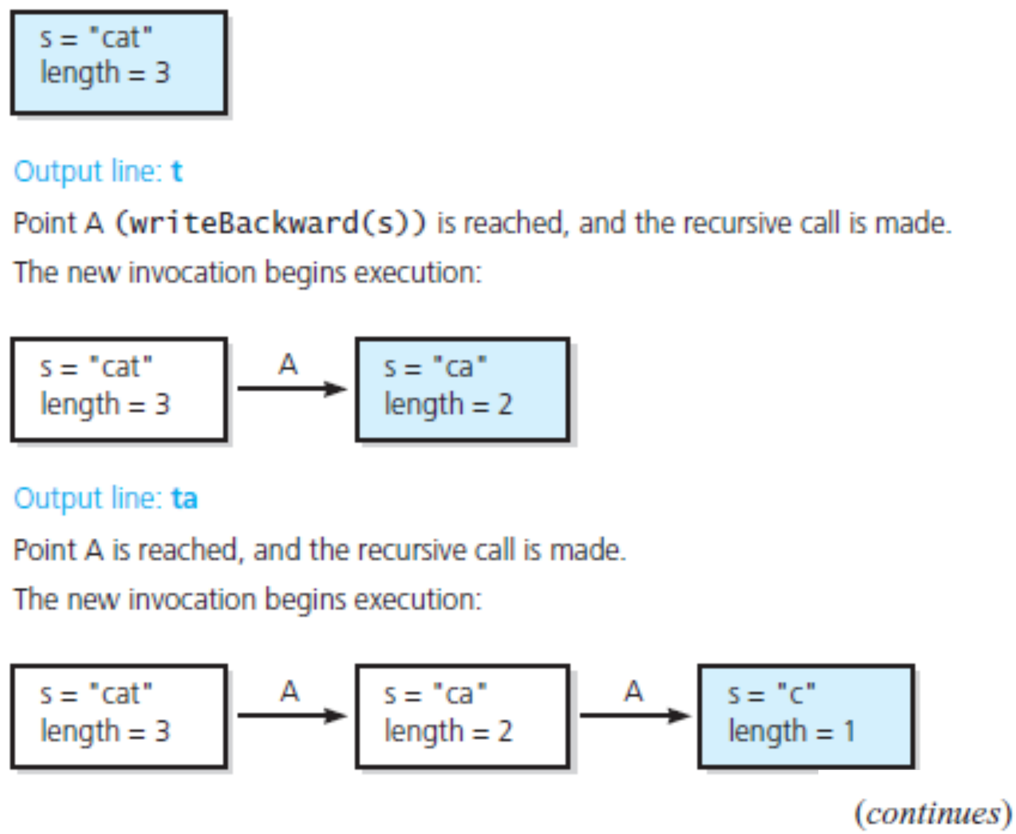


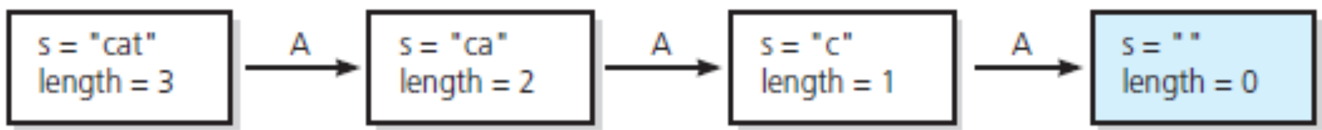
FIGURE Box trace of **`writeBackward("cat")`**

A Recursive Void Function

Output line: **tac**

Point A is reached, and the recursive call is made.

The new invocation begins execution:



This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:

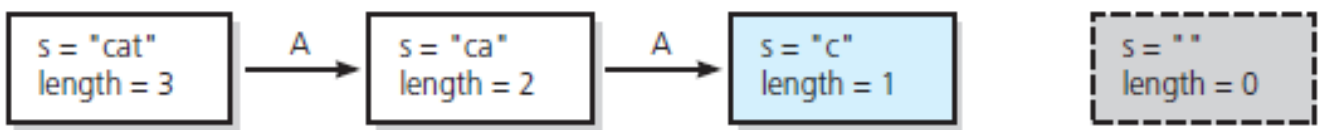


FIGURE 2-7 **`writeBackward("cat")`** continued

A Recursive Void Function

This invocation completes. Control returns to the calling box, which continues execution:

This invocation completes. Control returns to the calling box, which continues execution:

This invocation completes. Control returns to the statement following the initial call.

FIGURE **`writeBackward("cat")`** continued

The Binary Search

- A high-level binary search for the array problem

```
binarySearch(anArray: ArrayType, target: ValueType)

   if (anArray is of size 1)
      Determine if anArray's value is equal to target
   else
   {
      Find the midpoint of anArray
      Determine which half of anArray contains target
      if (target is in the first half of anArray)
         binarySearch(first half of anArray, target)
      else
         binarySearch(second half of anArray, target)
   }
```

The Binary Search

Issues to consider

1. How to pass a half array to recursive call
2. How to determine which half of array has target value
3. What is the base case?
4. How will result of binary search be indicated?

Finding the Largest Value in an Array

- Recursive algorithm

```
if (anArray has only one entry)
   maxArray(anArray) is the entry in anArray
else if (anArray has more than one entry)
   maxArray(anArray) is the maximum of
      maxArray(left half of anArray) and maxArray(right half of anArray)
```

Finding the Largest Value in an Array

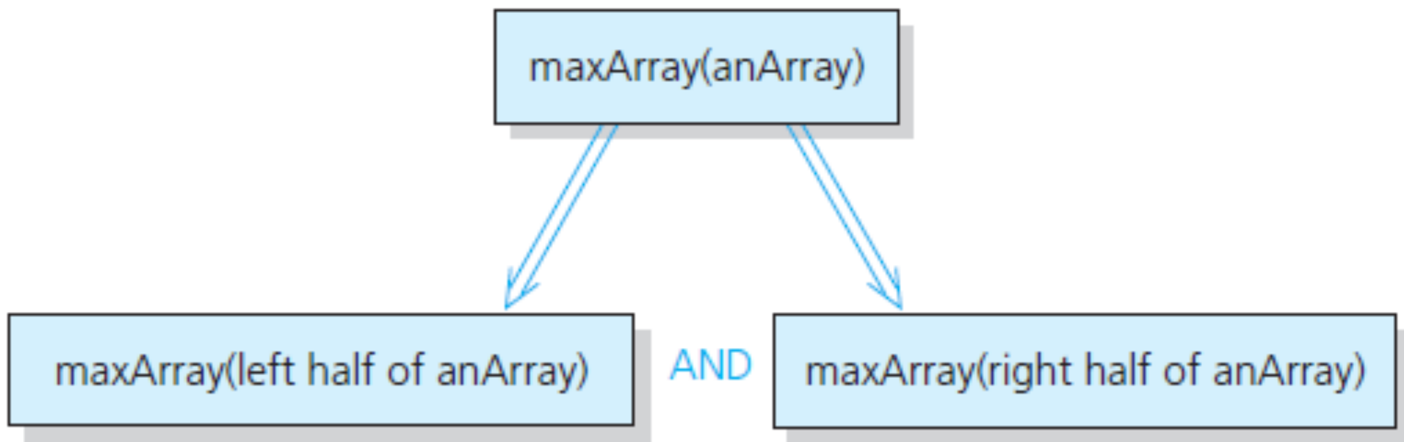


FIGURE 2-12 Recursive solution to the largest-value problem

Finding the Largest Value in an Array

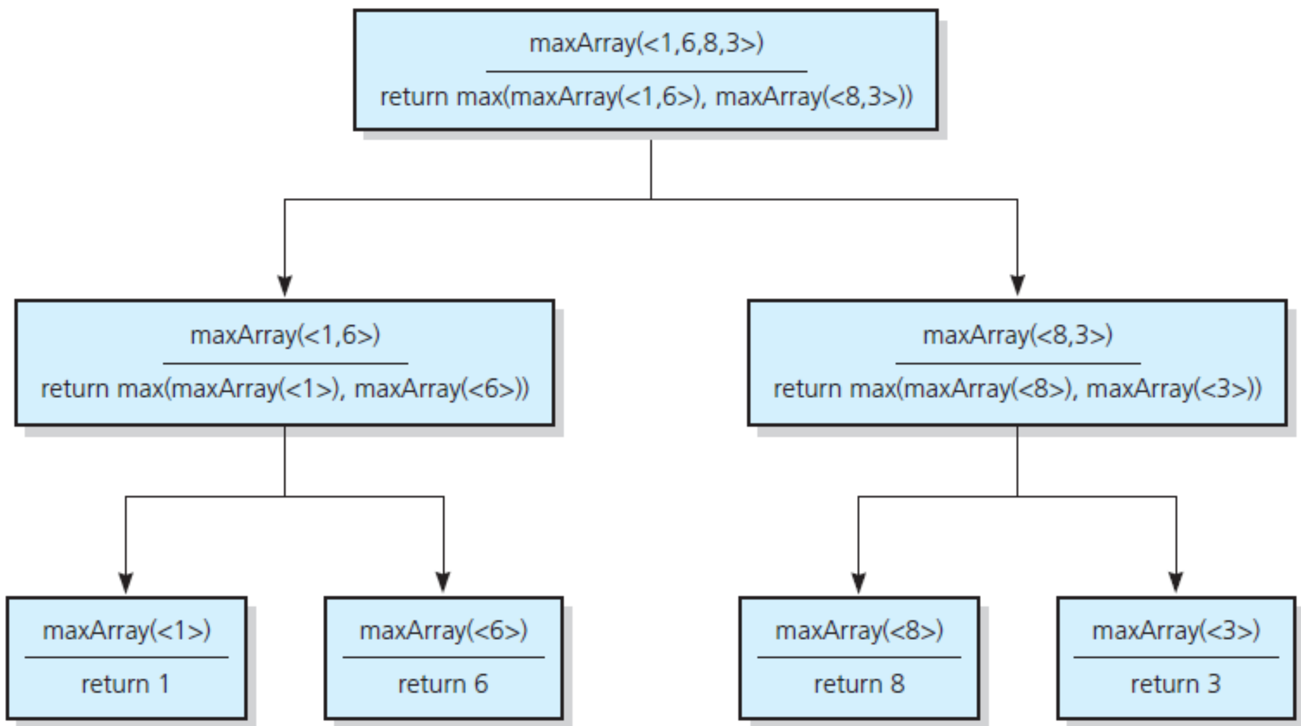


FIGURE 2-13 The recursive calls that **`maxArray(<1,6,8,3>)`** generates

Finding the k^{th} Smallest Value of an Array

The recursive solution proceeds by:

1. Selecting a pivot value in array
2. Cleverly arranging/partitioning, values in array about this pivot value
3. Recursively applying strategy to one of partitions

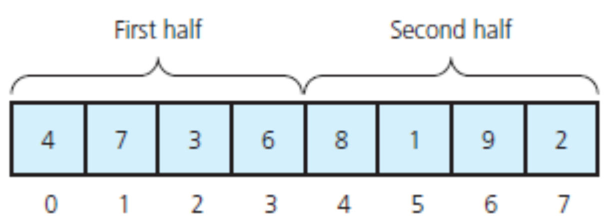


FIGURE A sample array

Finding the k^{th} Smallest Value of an Array

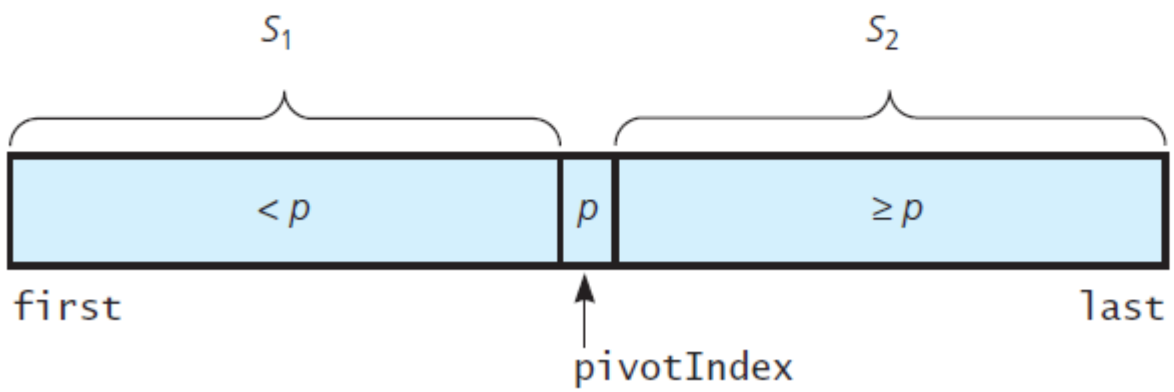


FIGURE 2 A partition about a pivot

Finding the k^{th} Smallest Value of an Array

High level pseudocode solution:

```
// Returns the kth smallest value in anArray[first..last].

kSmall(k: integer, anArray: ArrayType,
      first: integer, last: integer): ValueType

   Choose a pivot value p from anArray[first..last]
   Partition the values of anArray[first..last] about p

   if (k < pivotIndex - first + 1)
      return kSmall(k, anArray, first, pivotIndex - 1)
   else if (k == pivotIndex - first + 1)
      return p
   else
      return kSmall(k - (pivotIndex - first + 1), anArray,
                    pivotIndex + 1, last)
```

Organizing Data
Towers of Hanoi

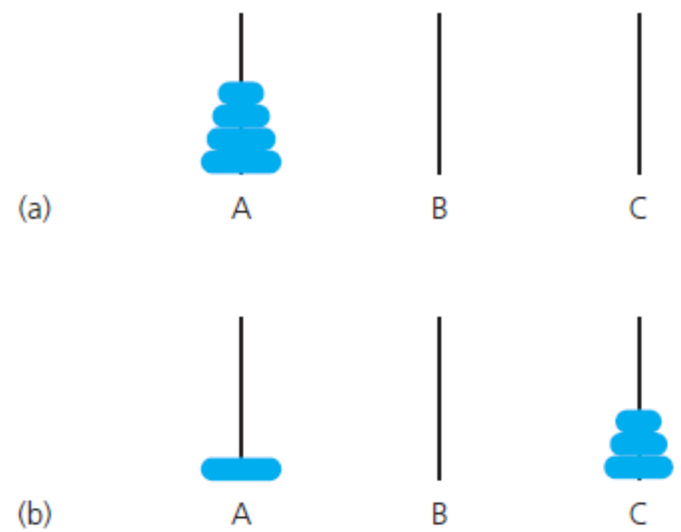


FIGURE 2-16 (a) The initial state;
(b) move n – 1 disks from A to C;

Towers of Hanoi

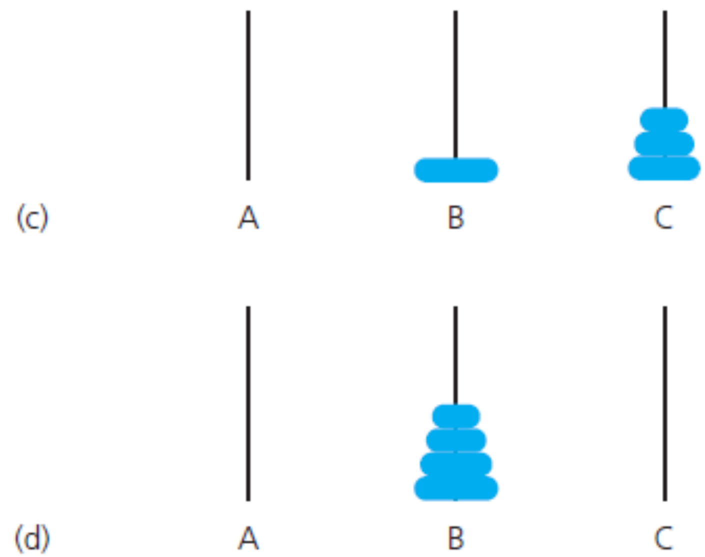


FIGURE 2-16 (c) move 1 disk from A to B;
(d) move n – 1 disks from C to B

Towers of Hanoi

Pseudocode solution:

```
solveTowers(count, source, destination, spare)

   if (count is 1)
      Move a disk directly from source to destination
   else
   {
      solveTowers(count - 1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count - 1, spare, destination, source)
   }
```

Towers of Hanoi

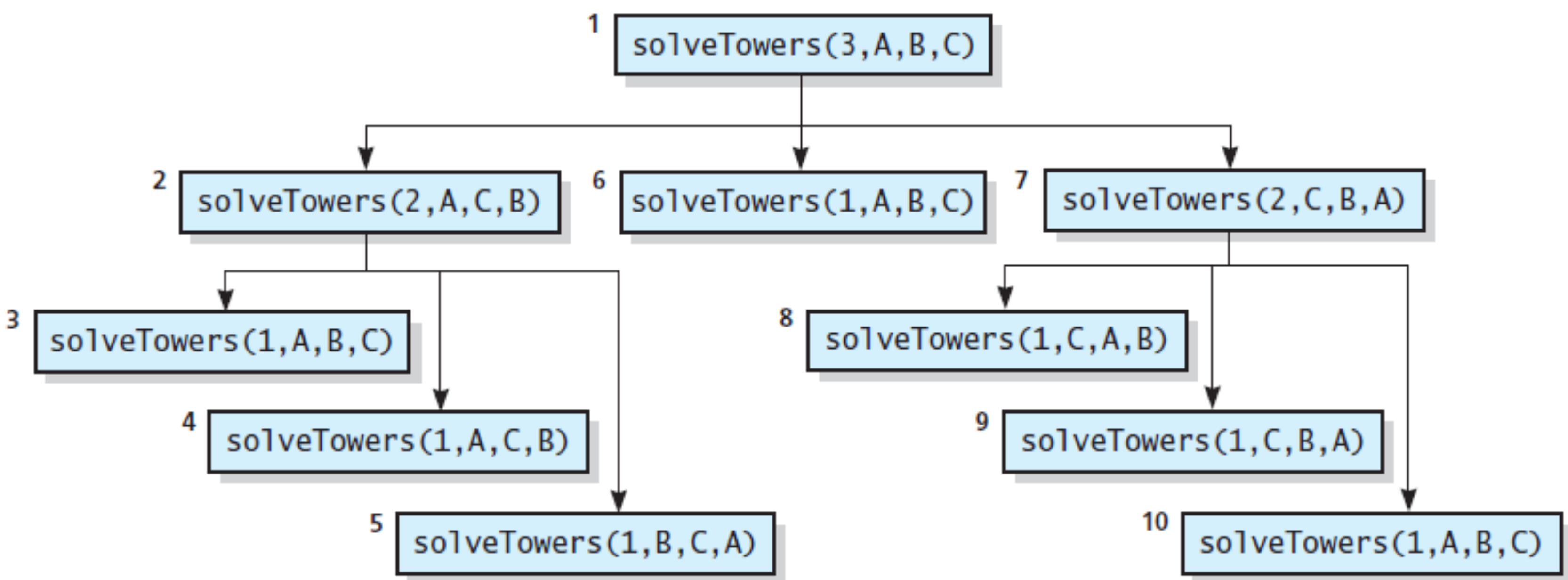


FIGURE 2-17 The order of recursive calls that results from solve **`Towers(3, A, B, C)`**

Towers of Hanoi

- Source code for **`solveTowers`**

```
void solveTowers(int count, char source, char destination, char spare)
{
   if (count == 1)
   {
      cout << "Move top disk from pole " << source
           << " to pole " << destination << endl;
   }
   else
   {
      solveTowers(count - 1, source, spare, destination);  // X
      solveTowers(1, source, destination, spare);          // Y
      solveTowers(count - 1, spare, destination, source);  // Z
   }  // end if
}  // end solveTowers
```

The Fibonacci Sequence (Multiplying Rabbits)

Assumed “facts” about rabbits:

- Rabbits never die.
- A rabbit reaches sexual maturity exactly two months after birth
- Rabbits always born in male-female pairs.
- At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

The Fibonacci Sequence (Multiplying Rabbits)

Month	Rabbit Population
1	One pair
2	One pair, still
3	Two pairs
4	Three pairs
5	Five pairs
6	8 pairs

The Fibonacci Sequence (Multiplying Rabbits)

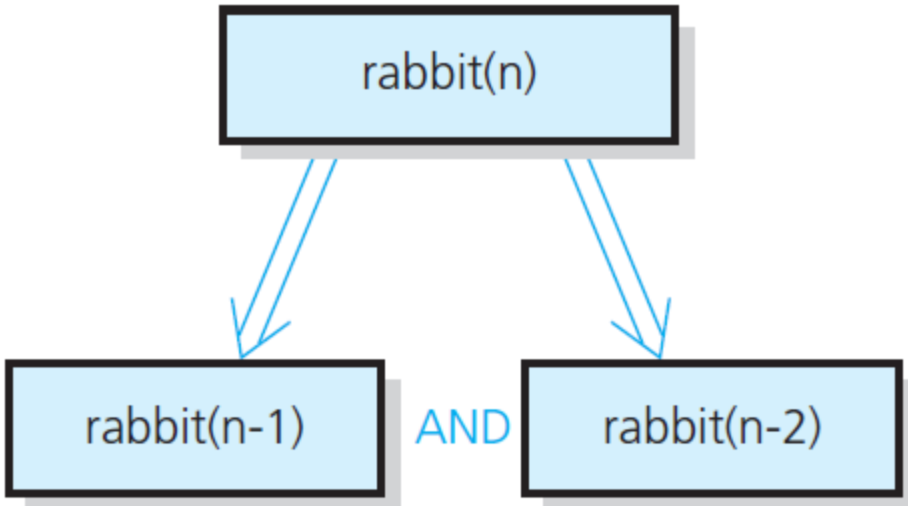


FIGURE 2-18 Recursive solution to the rabbit problem

The Fibonacci Sequence (Multiplying Rabbits)

- A C++ function to compute **`rabbit(n)`**

```
/** Computes a term in the Fibonacci sequence.
  @pre  n is a positive integer.
  @post  None.
  @param n  The given integer.
  @return  The nth Fibonacci number. */
int rabbit(int n)
{
   if (n <= 2)
      return 1;
   else // n > 2, so n - 1 > 0 and n - 2 > 0
      return rabbit(n - 1) + rabbit(n - 2);
}  // end rabbit
```

The Fibonacci Sequence (Multiplying Rabbits)

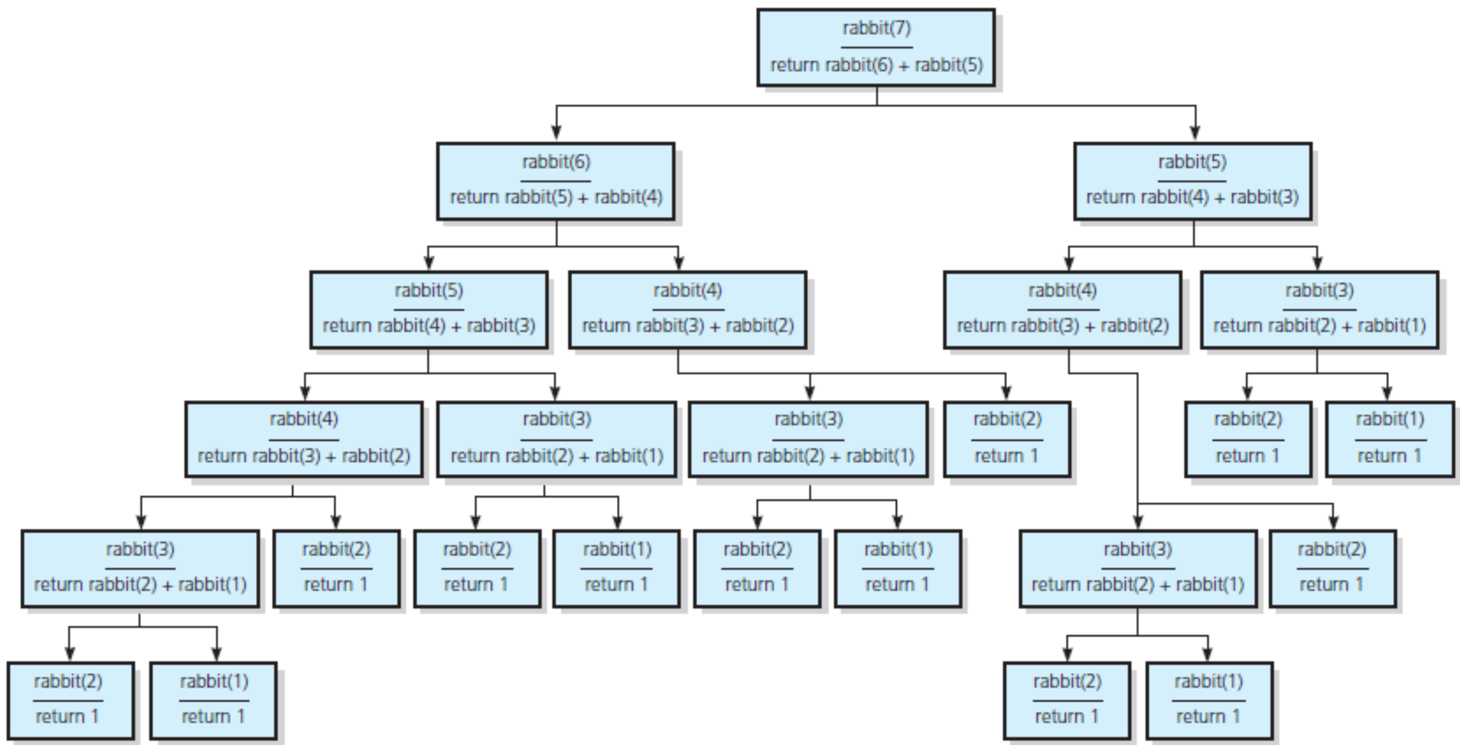


FIGURE 2-19 The recursive calls that rabbit(7) generates

Choosing *k* Out of *n* Things

- Recursive solution:

$$
g(n,k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ g(n-1,k-1) + g(n-1,k) & \text{if } 0 < k < n \end{cases}
$$

Choosing *k* Out of *n* Things

- Recursive function:

```
/** Computes the number of groups of k out of n things.
 @pre  n and k are nonnegative integers.
 @post  None.
 @param n  The given number of things.
 @param k  The given number to choose.
 @return  g(n, k). */
int getNumberOfGroups(int n, int k)
{
   if ( (k == 0) || (k == n) )
      return 1;
   else if (k > n)
      return > 0;
   else
      return g(n - 1, k - 1) + g(n - 1, k);
}  // end getNumberOfGroups
```

Choosing *k* Out of *n* Things

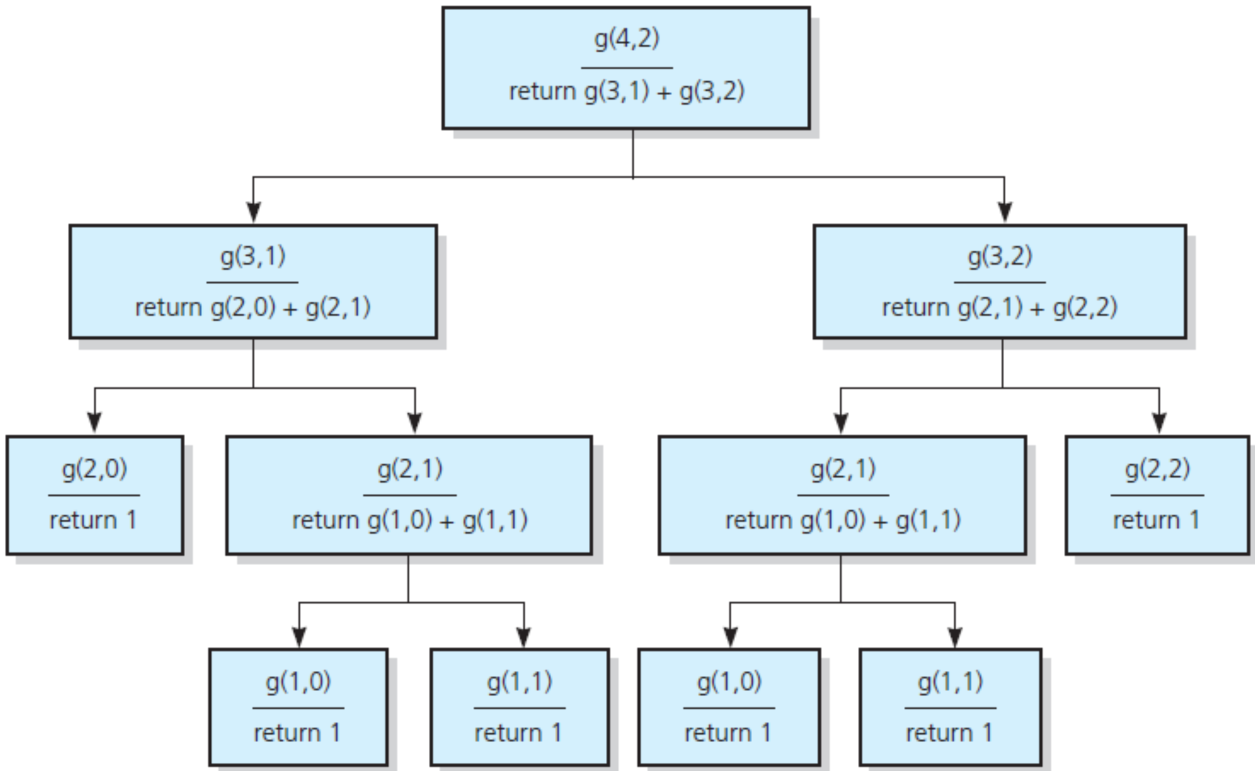


FIGURE 2-20 The recursive calls that g (4, 2) generates

Analysis

- What can we say about the “efficiency” of the recursive solutions?
- When is a recursive solution not a good approach?
  - Conversely: When is it a good approach?

Recursion as a Problem-Solving Technique

Contents

- Mind your Language!
- Backtracking
- The Relationship Between Recursion and Mathematical Induction

Languages and Grammar

- A language is
  - A set of strings of symbols
  - From a finite alphabet.
- C++Programs = {string s : s is a syntactically correct C++ program}
- Grammar states the rules of the language
- A recognition algorithm sees whether a given string is in the language
  - A recognition algorithm for a language is written more easily if the grammar is recursive

The Basics of Grammars

- Special symbols
  - x | y means x or y
  - xy (and sometimes x • y) means x followed by y
  - < word > means any instance of word, where word is a symbol that must be defined elsewhere in the grammar.
- C++Identifiers = {string s : s is a legal C++ identifier}

The Basics of Grammars

- A recognition algorithm sees whether a given string is in the language
  - A recognition algorithm for a language is written more easily if the grammar is recursive

The Basics of Grammars

- A C++ identifier begins with a letter and is followed by zero or more letters and digits
- Language of C++ identifiers

  C++*Ids* = {*w* : *w* is a legal C++ identifier}
- Grammar

  < *identifier* > = < *letter* > | < *identifier* > < *letter* > |
  < *identifier* > < *digit*>

  < *letter* > = a | b | … | z | A | B | …| Z | _

  < *digit* > = 0 | 1 | … | 9

The Basics of Grammars

- Recognition algorithm for the language C++*Ids*

```
isId(in w:string):boolean
  if (w is of length 1)
    if (w is a letter)
      return true
    else
      return false
  else if (the last character of w is a letter or a digit)
      return isId(w minus its last character)
  else
    return false
```

Two Simple Languages

- Palindromes = {string s : s reads the same left to right as right to left}
- Grammar for the language of palindromes:

$$<pal> \ = \text{empty string} \mid <ch> \mid \text{a } <pal> \text{ a} \mid \text{b } <pal> \text{ b} \mid \ldots \mid \text{Z } <pal> \text{ Z}$$

$$<ch> \ = \text{a} \mid \text{b} \mid \ldots \mid \text{z} \mid \text{A} \mid \text{B} \mid \ldots \mid \text{Z}$$

Two Simple Languages

- A recognition algorithm for palindromes

```
// Returns true if the string s of letters is a palindrome; otherwise returns false.
isPalindrome(s: string): boolean

   if (s is the empty string or s is of length 1)
      return true
   else if (s's first and last characters are the same letter)
      return isPalindrome(s minus its first and last characters)
   else
      return false
```

Two Simple Languages: Strings of the Form A^nB^n

- A^nB^n
  - The string that consists of n consecutive A's followed by n consecutive B's
- Language

  $L = \{w : w \text{ is of the form } A^nB^n \text{ for some } n \geq 0\}$
- Grammar

  < *legal-word* > = empty string |

  A < *legal-word* > B

Two Simple Languages: Strings of the form A^nB^n

- Recognition algorithm

isAnBn(in w:string):boolean

    if (the length of w is zero)
        return true
    else if (w begins with the character A and
        ends with the character B)
        return isAnBn(w minus its first and last
    characters)
    else
        return false

Algebraic Expressions

- Compiler must recognize and evaluate algebraic expressions
- Example

  ```
  y = x + z * (w / k + z * (7 * 6));
  ```
- Kinds of algebraic expressions
  - infix
  - prefix
  - postfix

Algebraic Expressions

- infix
  - Binary operator appears between its operands
- prefix
  - Operator appears before its operands
- postfix
  - Operator appears after its operands

Algebraic Expressions

- Advantages of prefix and postfix expressions
  - No precedence rules
  - No association rules
  - No parentheses
  - Simple grammars
  - Straightforward recognition and evaluation algorithms

Fully Parenthesized Expressions

- Fully parenthesized infix expressions
  - Do not require precedence rules or rules for association
  - But are inconvenient for programmers
- Grammar

$$
\begin{aligned}
<\textit{infix}> \; &= \; <\textit{identifier}> \mid \\
&\quad\; (<\textit{infix}><\textit{operator}><\textit{infix}>) \\
<\textit{operator}> \; &= \; + \mid - \mid * \mid / \\
<\textit{identifier}> \; &= \; a \mid b \mid \ldots \mid z
\end{aligned}
$$

Algebraic Expressions

- To convert a fully parenthesized infix expression to a prefix form
  - Move each operator to the position marked by its corresponding open parenthesis
  - Remove the parentheses
  - Example
    - Infix expression: ((a + b) * c)
    - Prefix expression: * + a b c

Algebraic Expressions

- To convert a fully parenthesized infix expression to a postfix form
  - Move each operator to the position marked by its corresponding closing parenthesis
  - Remove the parentheses
  - Example
    - Infix expression: ((a + b) * c)
    - Postfix expression: a b + c *

Prefix Expressions

- Grammar that defines language of all prefix expressions

$$<prefix> \; = \; <identifier> | <operator><prefix><prefix>$$
$$<operator> \; = \; + \; | - | * | /$$
$$<identifier> \; = \; a \,|\, b \,| \ldots | \, z$$

Prefix Expressions

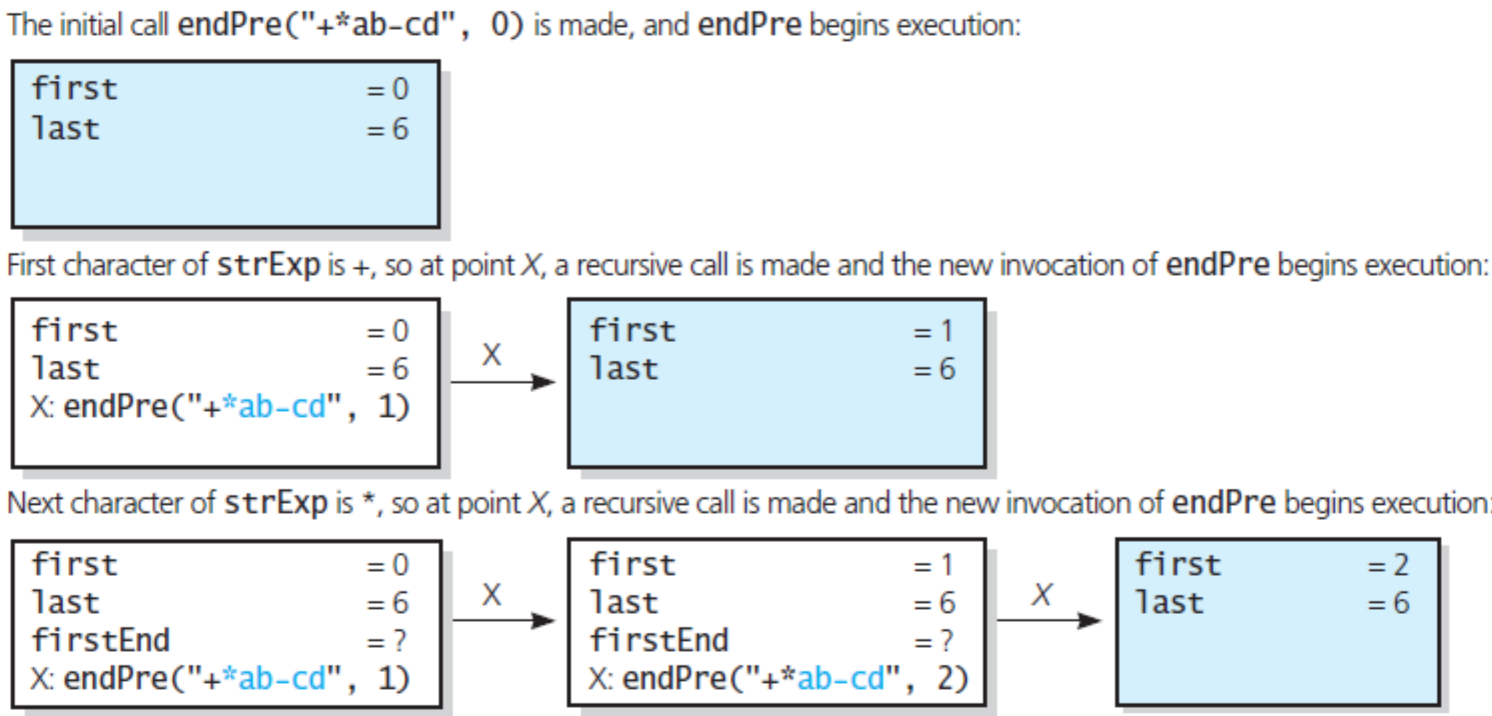


FIGURE Trace of **`endPre( "+/ab-cd",0)`**

Prefix Expressions

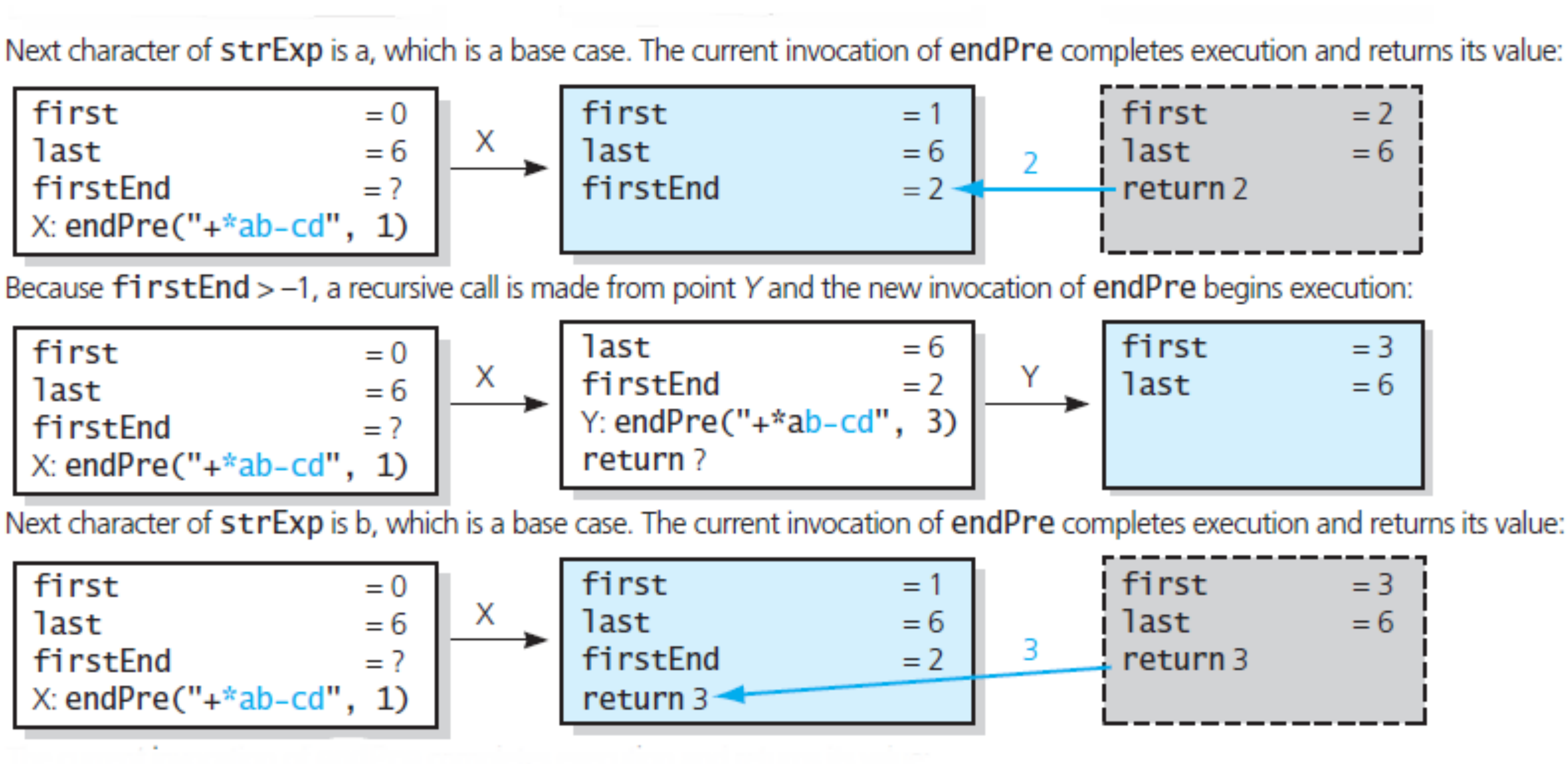


FIGURE 5-3 Trace of **endPre("+/ab-cd",0)**

Prefix Expressions

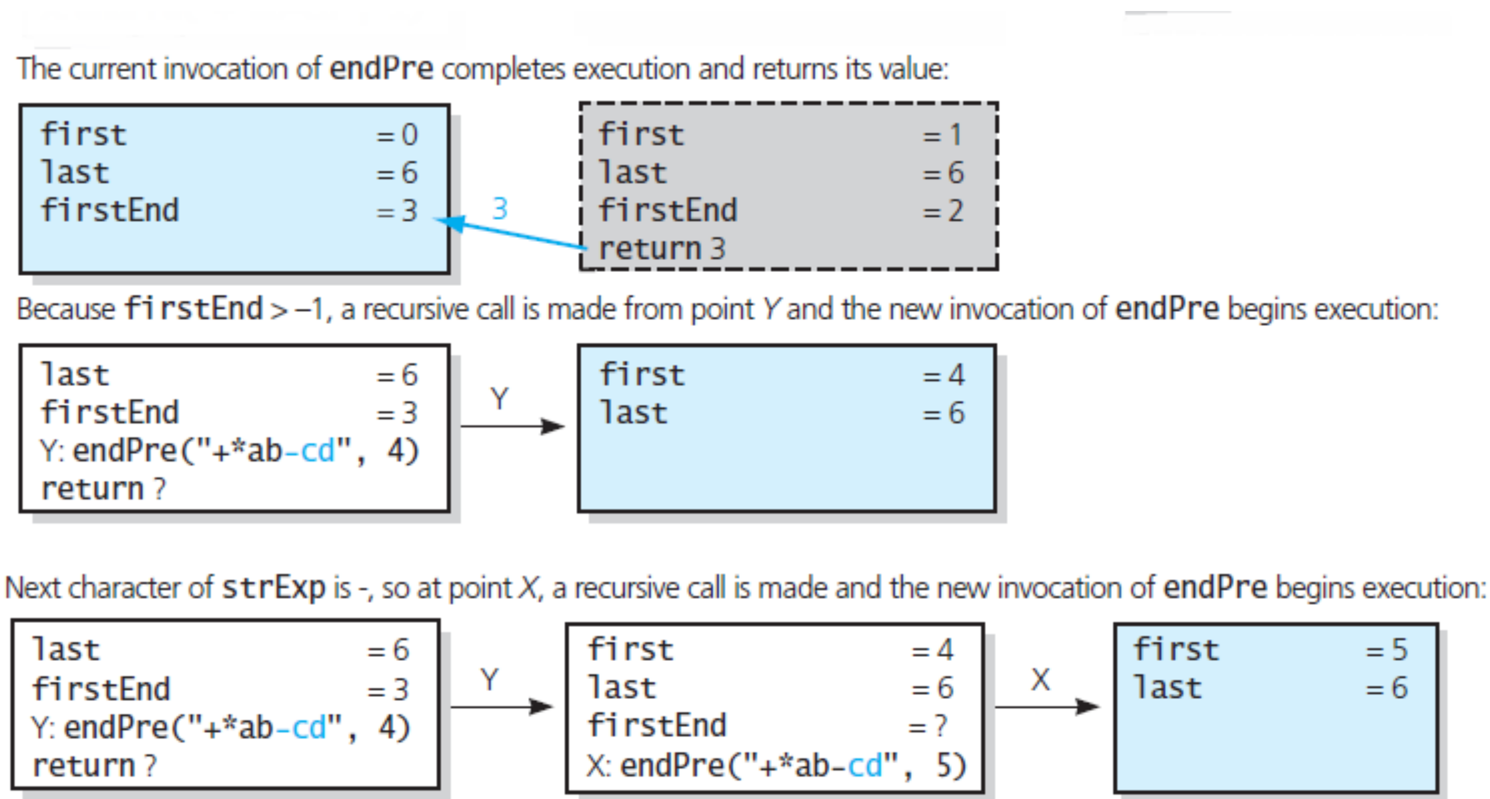


FIGURE 5-3 Trace of `endPre( "+/ab-cd",0)`

Prefix Expressions

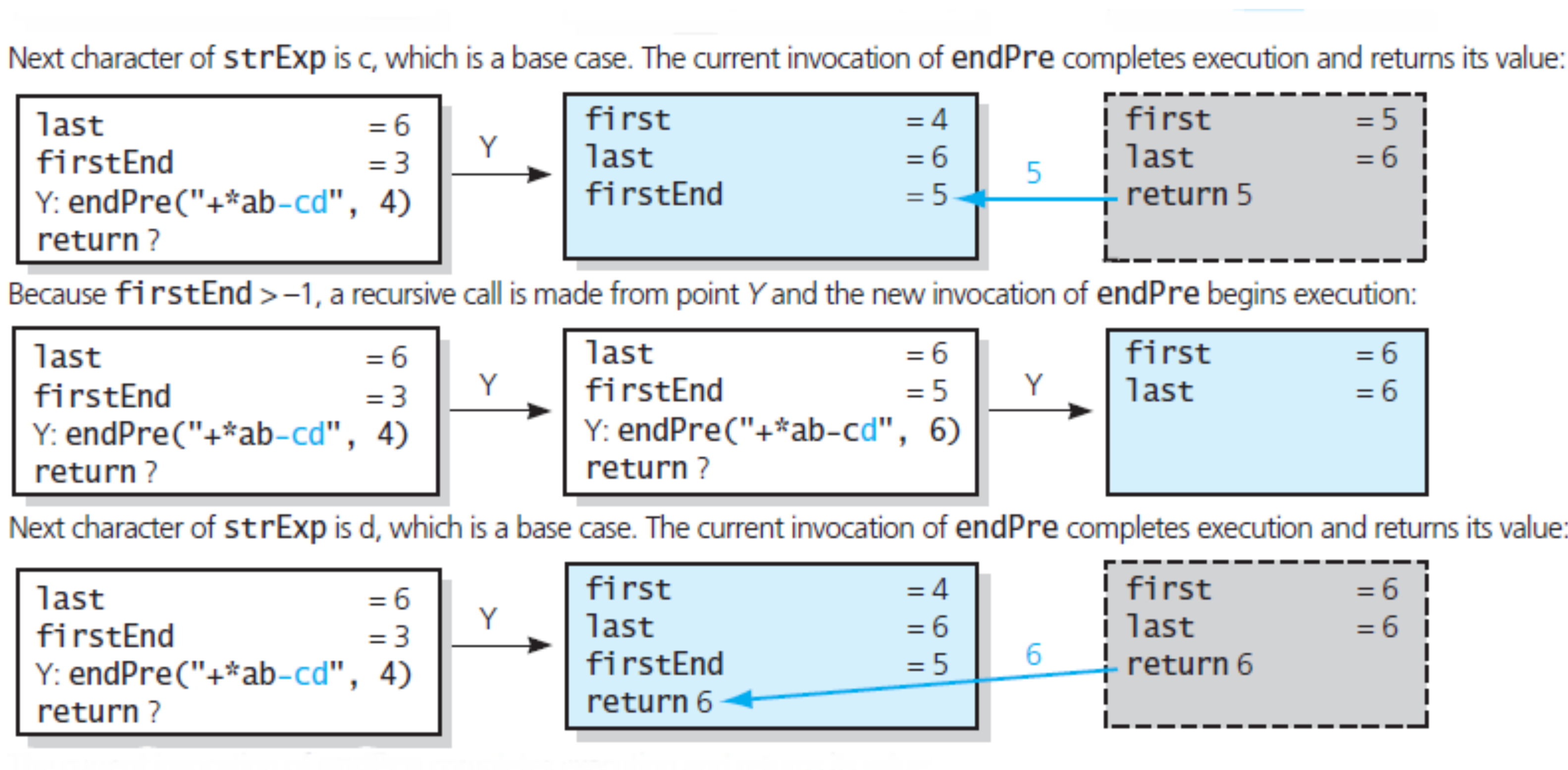


FIGURE 5-3 Trace of **endPre("+/ab-cd",0)**

Prefix Expressions

. . .

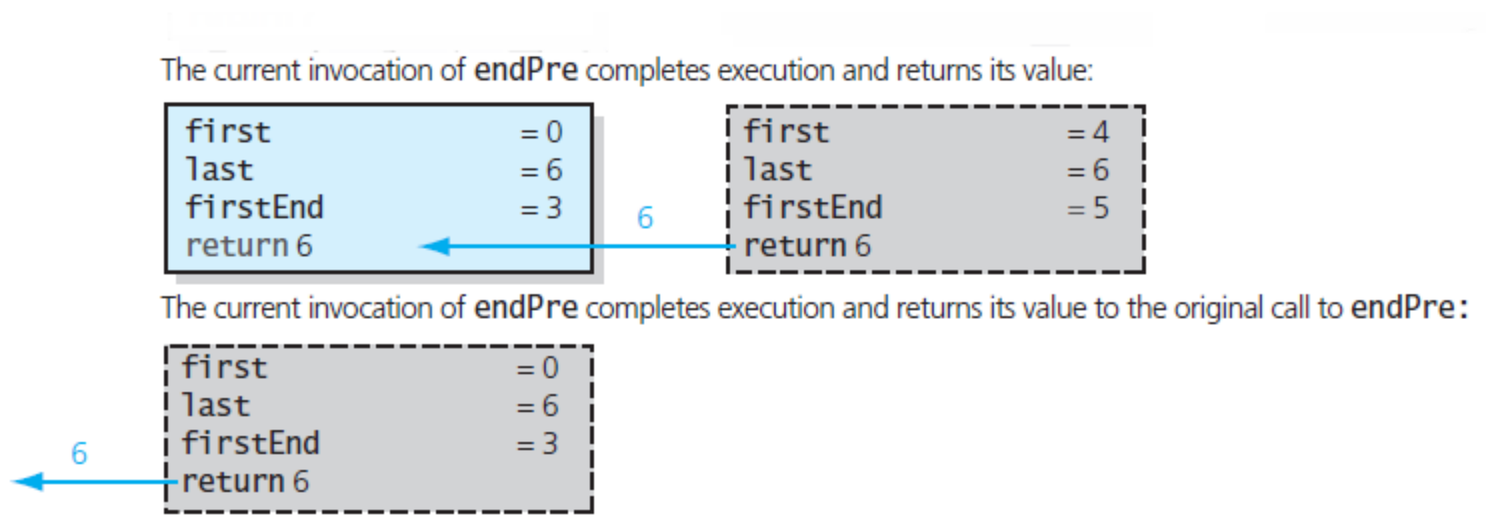


FIGURE 5-3 Trace of **`endPre( "+/ab-cd",0)`**

Postfix Expressions

- Grammar that defines language of postfix expressions

$$\langle postfix \rangle = \langle identifier \rangle | \langle postfix \rangle \langle postfix \rangle \langle operator \rangle$$
$$\langle operator \rangle = + | - | * | /$$
$$\langle identifier \rangle = a \mid b \mid \ldots \mid z$$

Fully Parenthesized Expressions

- Grammar that defines language of fully parenthesized infix expression

$$
\begin{aligned}
<infix> &= <identifier> \,|\, (<infix><operator><infix>) \\
<operator> &= + \,|\, - \,|\, * \,|\, / \\
<identifier> &= a \,|\, b \,|\, \ldots \,|\, z
\end{aligned}
$$

Backtracking

- Consider searching for an airline route
- Input text files that specify all of the flight information for HPAir Company
  - Names of cities HPAir serves
  - Pairs of city names, each pair representing origin and destination of one of HPAir's flights
  - Pairs of city names, each pair representing a request to fly from some origin to some destination

Backtracking

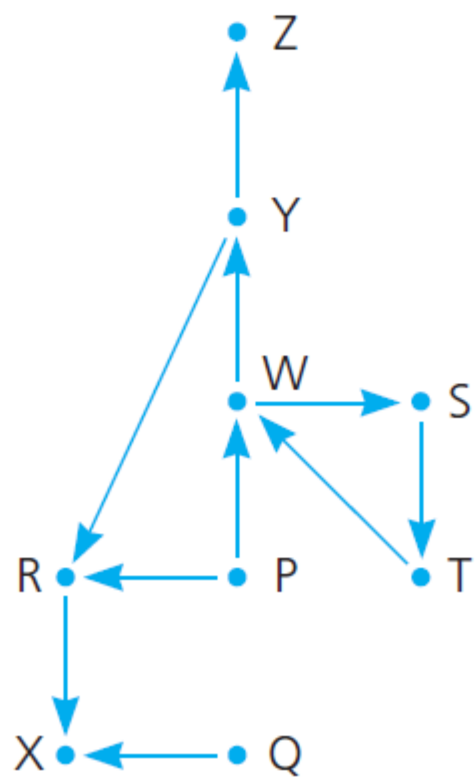


FIGURE 5-4 Flight map for HPAir

Backtracking

- A recursive strategy

To fly from the origin to the destination:

```
    Select a city C adjacent to the origin
    Fly from the origin to city C
    if (C is the destination city)
        Terminate— the destination is reached
    else
        Fly from city C to the destination
```

Backtracking

- Possible outcomes of applying the previous strategy
  1. Eventually reach destination city and can conclude that it is possible to fly from origin to destination.
  2. Reach a city C from which there are no departing flights.
  3. Go around in circles.

The Eight Queens Problem

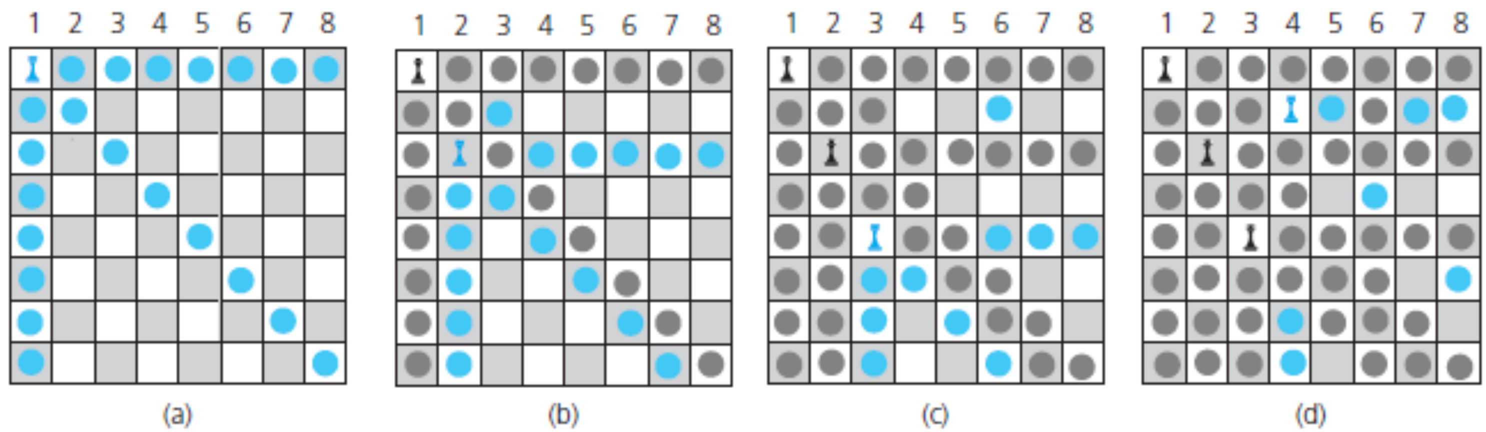


FIGURE 5-7 Placing one queen at a time in each column, and the placed queens' range of attack: (a) the first queen in column 1; (b) the second queen in column 2; (c) the third queen in column 3; (d) the fourth queen in column 4;

The Eight Queens Problem

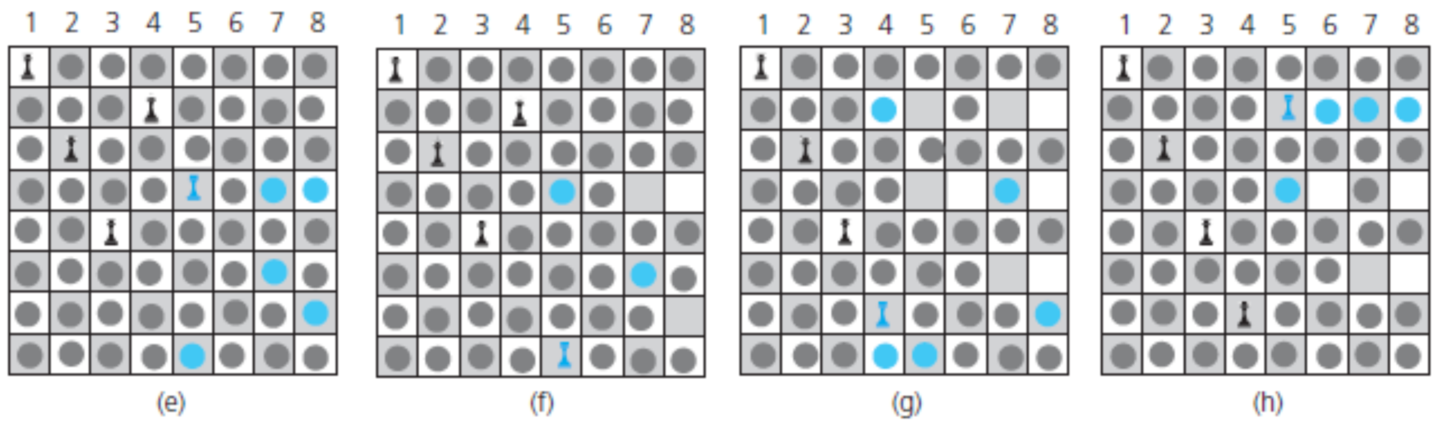


FIGURE 5-7 Placing one queen at a time in each column, and the placed queens' range of attack: (e) five queens can attack all of column 6; (f) backtracking to column 5 to try another square for queen; (g) backtracking to column 4 to try another square for the queen; (h) considering column 5 again

The Eight Queens Problem

- Find pseudocode of algorithm for placing queens in columns,

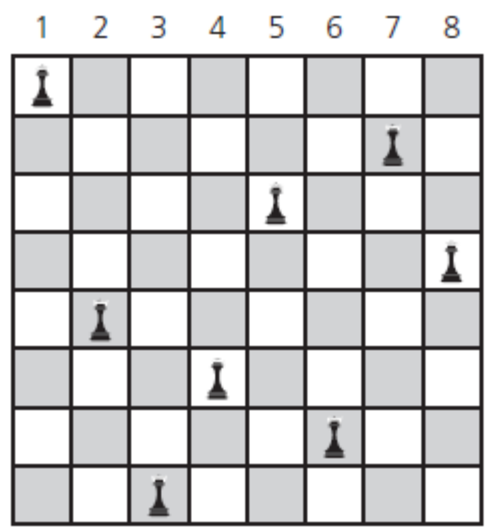


FIGURE A solution to the Eight Queens problem

Recursion and Efficiency

- Inefficiency factors
  - Overhead associated with function calls
  - Inherent inefficiency of some recursive algorithms
- Principle:
  - Do not use recursive solution if inefficient and clear, efficient iterative solution exists

Recursion and Efficiency

- Some recursive solutions are so inefficient that they should not be used
- Factors that contribute to the inefficiency of some recursive solutions
  - Overhead associated with function calls
    - Stack traces etc.,
  - Inherent inefficiency of some recursive algorithms
    - Number of times a function is called

Recursion and Efficiency

- Do not use a recursive solution if it is inefficient and there is a clear, efficient iterative solution
- Recursive solutions have overhead
  - Is it worth the overhead?
  - Simple problems, likely not
    - Counting and other problems in Chp are "simple"
    - Binary search & "Towers of Hanoi" naturally recursive
  - Complex problems, likely yes
    - N-Queens Problem with Backtracking

Recursion and Efficiency

- Is there an optimal point?
  - Simplicity and natural expression of recursive solution, yet efficiency of iterative approach?
  - Use recurrence relation but invoke iteratively
    - See next slides for an example

Recursion and Efficiency

```
/** Computes the nth term in the Fibonacci sequence, iterative
 *  version.
 * @return The nth Fibonacci number. */
int iterativeRabbit(int n)
{
   // Initialize base cases:
   int previous = 1; // Initially rabbit(1)
   int current = 1;  // Initially rabbit(2)
   int next = 1;     // Result when n is 1 or 2

   // Compute next rabbit values when n >= 3
   for (int i = 3; i <= n; ++i)
   {  // current is rabbit(i-1), previous is rabbit(i-2)
      next = current + previous; // rabbit(i)
      previous = current;        // Get ready for
      current = next;            // next iteration
   }  // end for

   return next;
}  // end iterativeRabbit
```

Recursion and Efficiency

- But when is converting recursive to iterative good?
  - Easier if recursive function calls itself once
    - rabbit() calls itself twice, but binarysearch() once
  - Tail recursive, if the recursive function call is the last action
    - writeBackward() --- tail recursive
    - fact() --- not tail recursive

Recursion

```
/** Writes a character string backward.
 * @pre The string s contains size characters, where size >= 0.
 * @post None.
 * @param s  The string to write backward.
 * @param size  The length of s. */
void writeBackward(string s, int size)
{
   if (size > 0)
   {  // write last character
      cout << s.substr(size-1, 1);
      writeBackward(s, size-1);  // write rest
   }  // end if
}  // end writeBackward
```

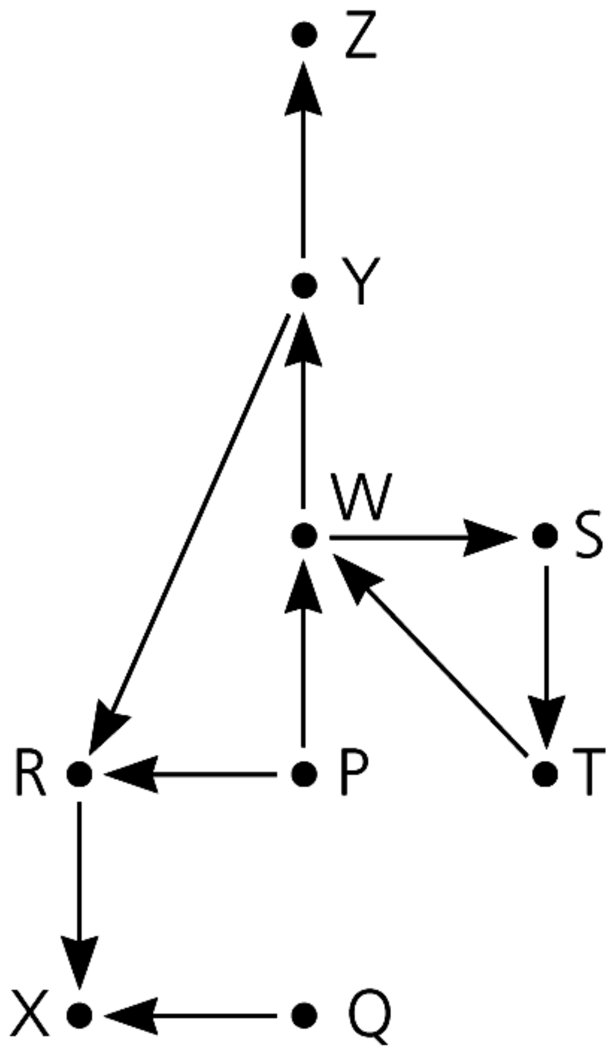
```
/** Writes a character string backward, iterative version.
 * @pre The string s contains size characters, where size >= 0.
 * @param s The string to write backward.
 * @param size The length of s.
 * @post None. */
void writeBackward(string s, int size)
{
   while (size > 0)
   {  cout << s.substr(size-1, 1);
      --size;
   } // end while
} // end writeBackward
```

- Often compilers make the transition automatically

The Relationship Between Stacks and 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

Application: A Search Problem



- Find a path from some point of origin to some destination point
- Exhaustive search:
  - Beginning at the origin the solution will try every possible sequence of flights until either it finds a sequence that gets to the destination city or determines that no such city exists
  - Compare using stacks with recursive solution

Fundamentals III

Tree, Heaps

Trees

Content

- Terminology
- The ADT Binary Tree
- The ADT Binary Search Tree

Terminology

- Use trees to represent relationships
- Trees are hierarchical in nature
  - "Parent-child" relationship exists between nodes in tree.
  - Generalized to ancestor and descendant
  - Lines between the nodes are called edges
- A subtree in a tree is any node in the tree together with all of its descendants

Terminology

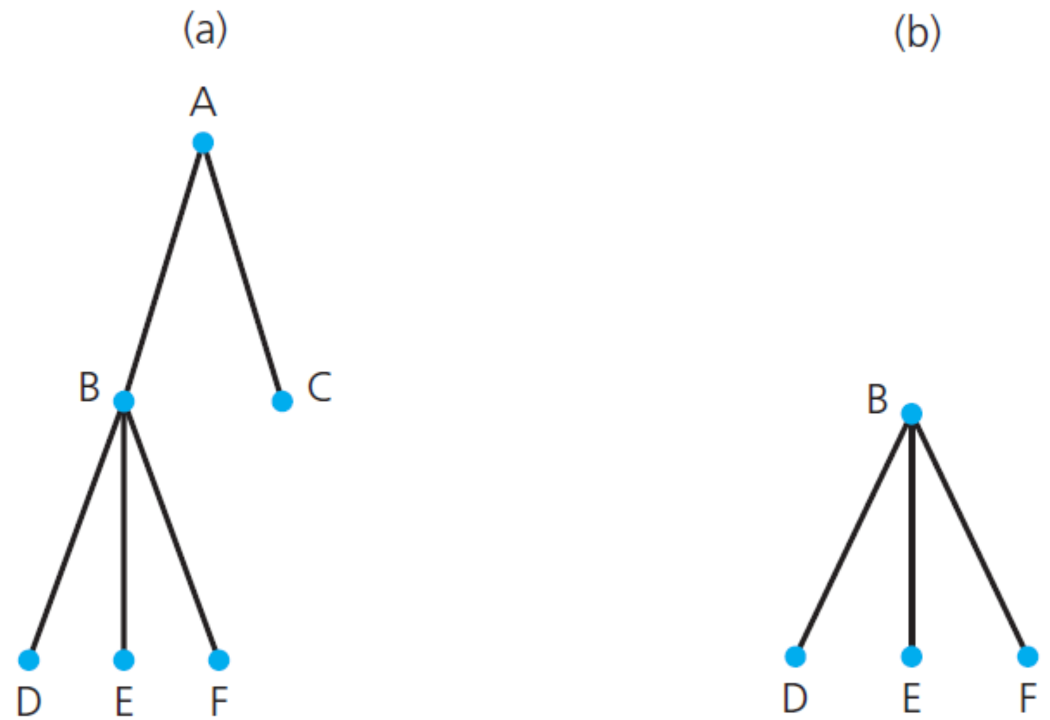


FIGURE (a) A tree;
(b) a subtree of the tree in part a

Terminology

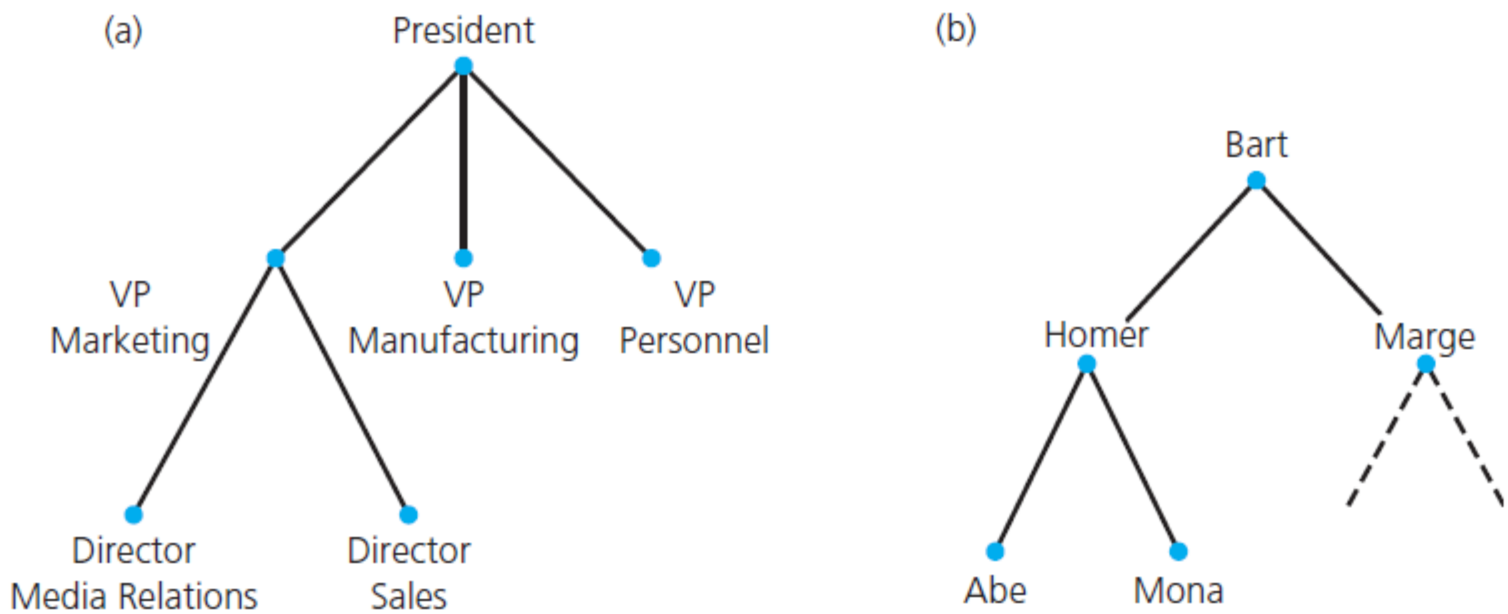


FIGURE 15-2 (a) An organization chart; (b) a family tree

Kinds of Trees

- General Tree
  - Set T of one or more nodes such that T is partitioned into disjoint subsets
  - A single node r, the root
  - Sets that are general trees, called subtrees of r

Kinds of Trees

- n -ary tree
  - set T of nodes that is either empty or partitioned into disjoint subsets:
  - A single node r, the root
  - n possibly empty sets that are n -ary subtrees of r

Kinds of Trees

- Binary tree
  - Set T of nodes that is either empty or partitioned into disjoint subsets
  - Single node r, the root
  - Two possibly empty sets that are binary trees, called left and right sub-trees of r

Example: Algebraic Expressions.

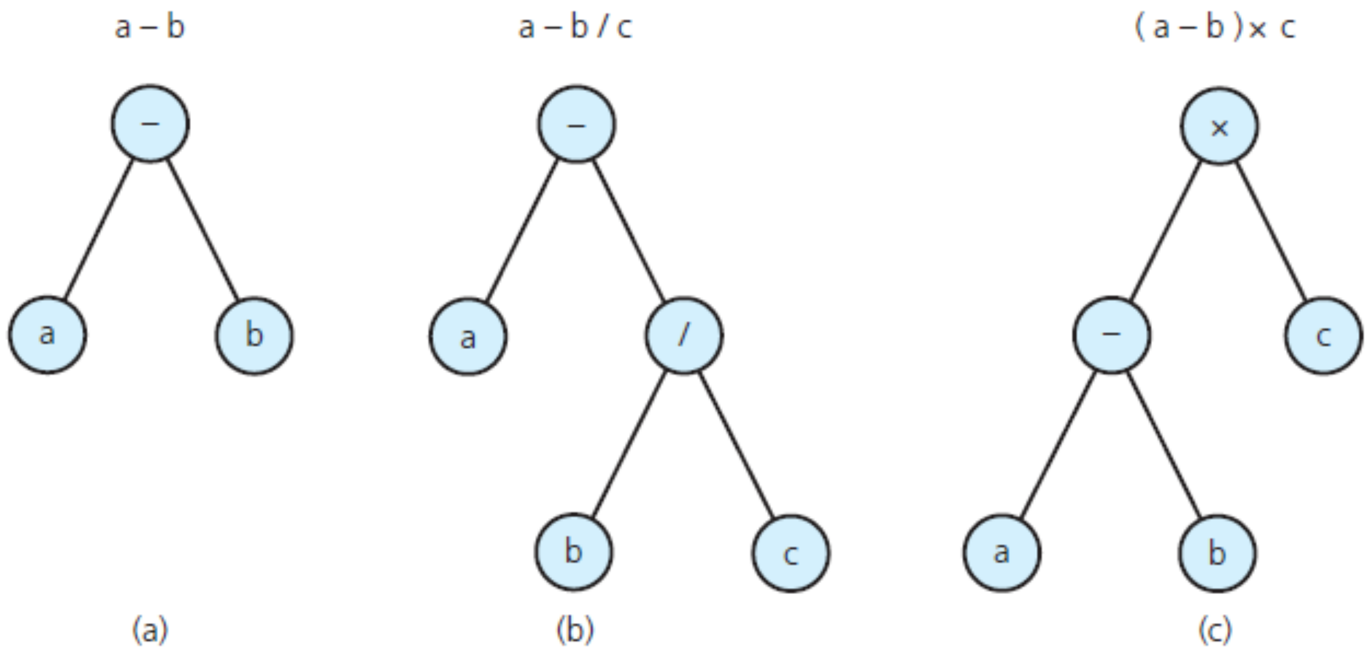


FIGURE Binary trees that represent algebraic expressions

The Height of Trees

- Definition of the level of a node n :
  - If n is the root of T, it is at level 1.
  - If n is not the root of T, its level is 1 greater than the level of its parent.
- Height of a tree T in terms of the levels of its nodes
  - If T is empty, its height is 0.
  - If T is not empty, its height is equal to the maximum level of its nodes.

The Height of Trees

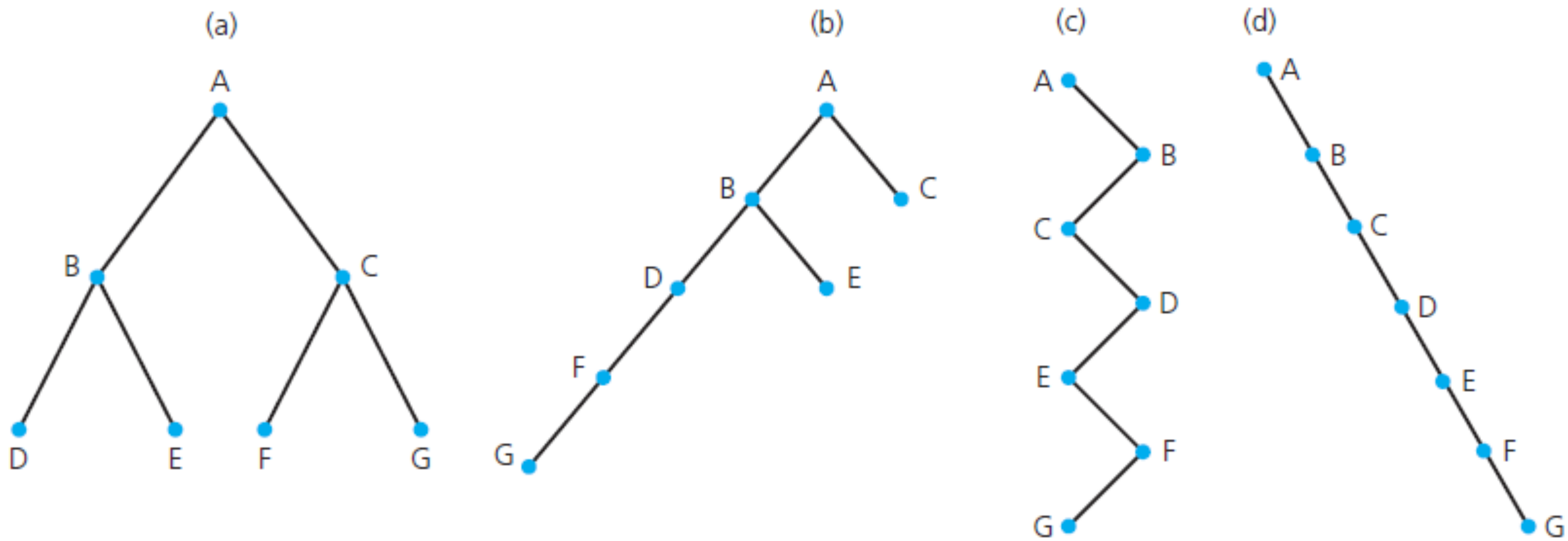


FIGURE Binary trees with the same nodes but different heights

Full, Complete, and Balanced Binary Trees

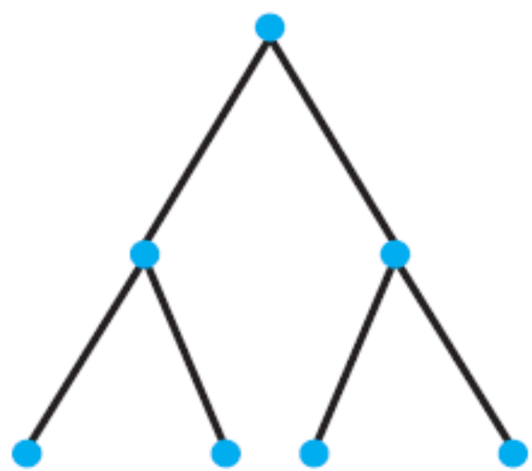


FIGURE A full binary tree of height 3

Full, Complete, and Balanced Binary Trees

- Definition of a full binary tree
  - If T is empty, T is a full binary tree of height 0.
  - If T is not empty and has height h > 0, T is a full binary tree if its root's subtrees are both full binary trees of height h – 1.

Complete Binary Trees

- A binary tree of height h is *complete* if
  - It is full to level $h - 1$, and
  - Level h is filled from left to right

Complete Binary Trees

Another definition:

- A binary tree of height h is *complete* if
  - All nodes at levels <= $h - 2$ have two children each, and
  - When a node at level $h - 1$ has children, all nodes to its left at the same level have two children each, and
  - When a node at level $h - 1$ has one child, it is a left child

Balanced Binary Trees

- A binary tree is *balanced* if the heights of any node's two subtrees differ by no more than 1
- Complete binary trees are balanced
- Full binary trees are complete and balanced

Full, Complete, and Balanced Binary Trees

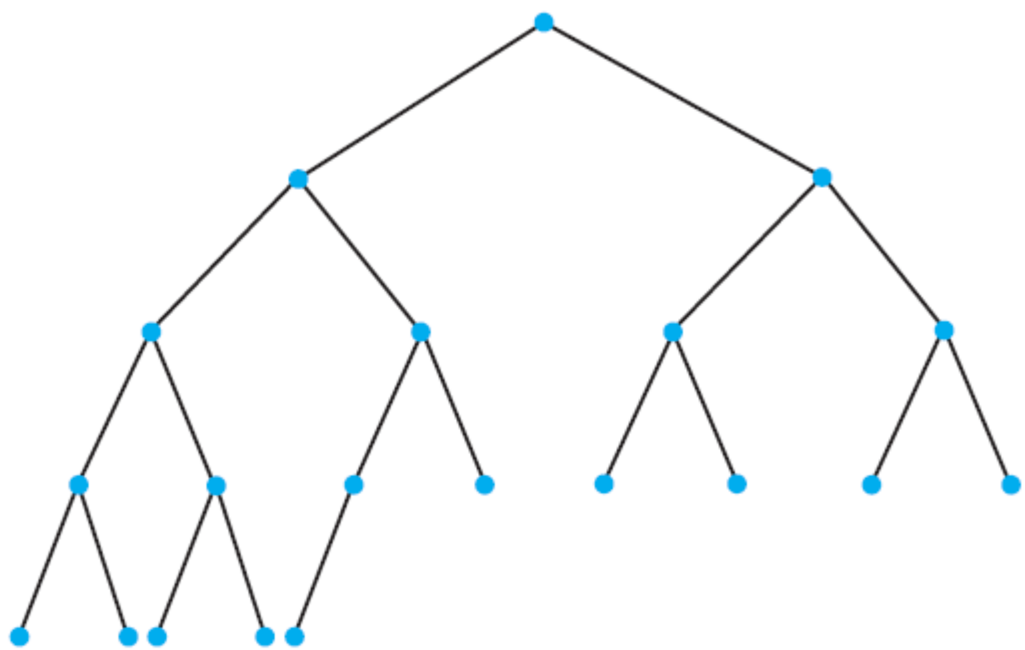


FIGURE A complete binary tree

The Maximum and Minimum Heights of a Binary Tree

- The maximum height of an n -node binary tree is n .

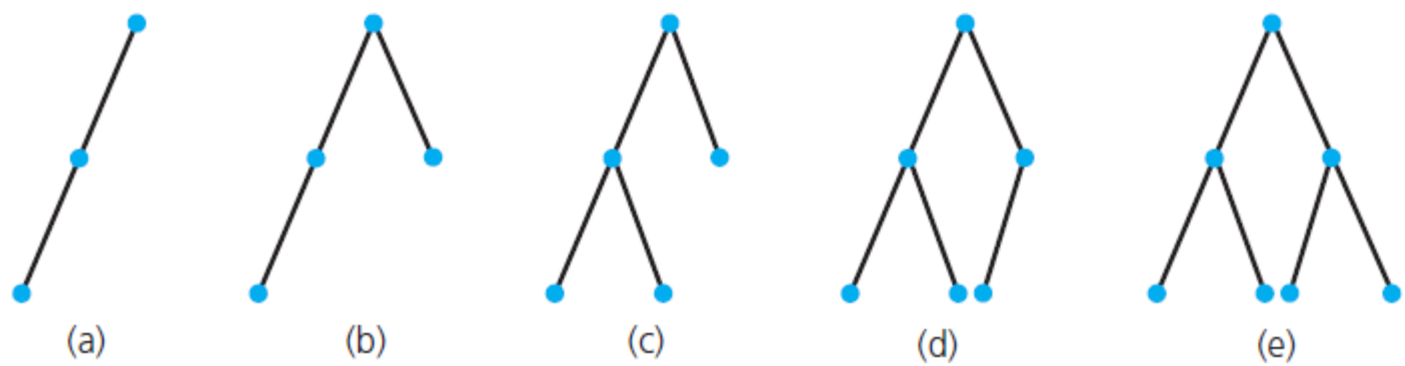
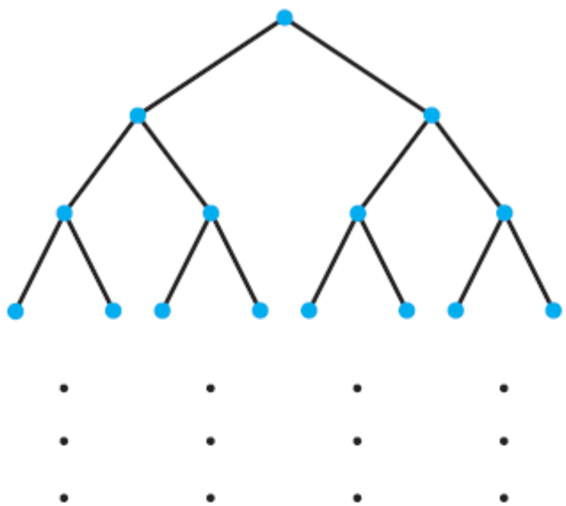


FIGURE Binary trees of height 3

The Maximum and Minimum Heights of a Binary Tree



Level	Number of nodes at this level	Total number of nodes at this level and all previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
⋮	⋮	⋮
h	2^{h-1}	$2^h - 1$

FIGURE Counting the nodes in a full binary tree of height h

Facts about Full Binary Trees

- A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes.
- You cannot add nodes to a full binary tree without increasing its height.
- The maximum number of nodes that a binary tree of height h can have is $2^h - 1$.
- The minimum height of a binary tree with n nodes is $[\log_2 (n + 1)]$

The Maximum and Minimum Heights of a Binary Tree

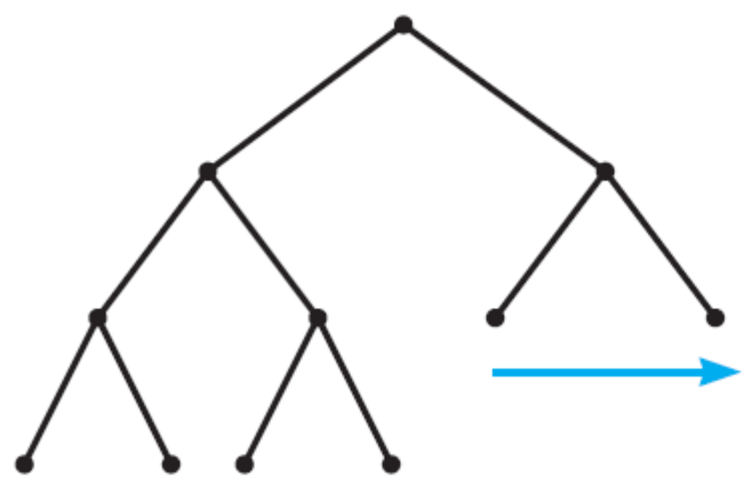


FIGURE Filling in the last level of a tree

Traversals of a Binary Tree

- General form of recursive transversal algorithm

```
if (T is not empty)
{
    Display the data in T's root
    Traverse T's left subtree
    Traverse T's right subtree
}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before visiting its subtrees
    - i. e. Before the recursive calls
- Inorder traversal
  - Visit root between visiting its subtrees
    - i. e. Between the recursive calls
- Postorder traversal
  - Visit root after visiting its subtrees
    - i. e. After the recursive calls

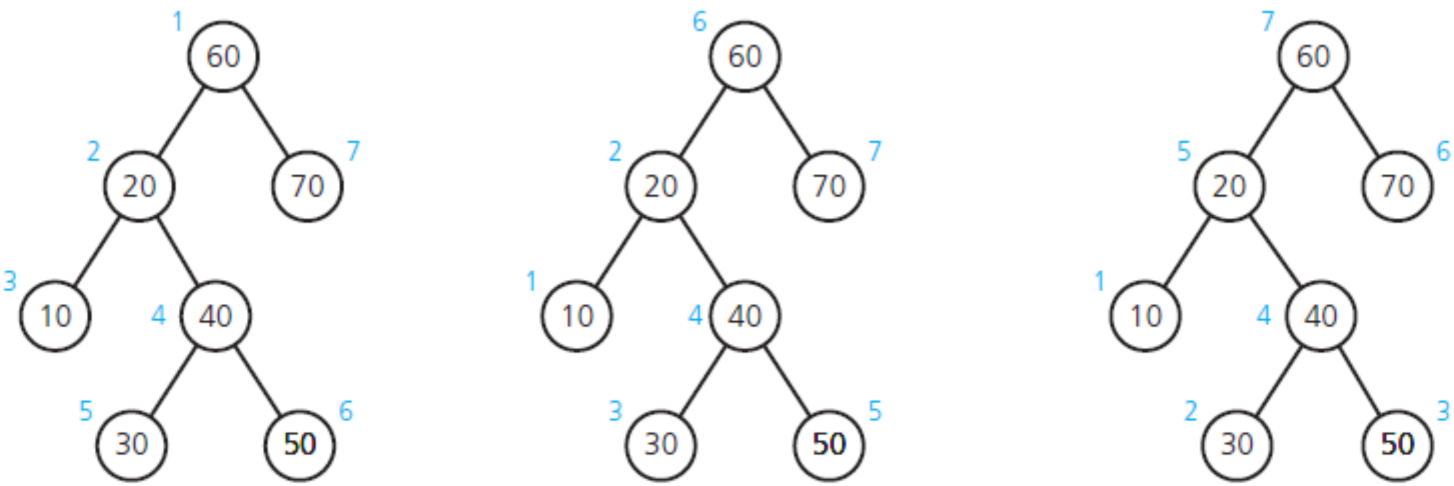
Traversals of a Binary Tree

- Preorder traversal.

```
// Traverses the given binary tree in preorder.
// Assumes that "visit a node" means to process the node's data item.
preorder(binTree: BinaryTree): void

   if (binTree is not empty)
   {
      Visit the root of binTree
      preorder(Left subtree of binTree's root)
      preorder(Right subtree of binTree's root)
   }
```

Traversals of a Binary Tree



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

FIGURE Three traversals of a binary tree

Traversals of a Binary Tree

- Inorder traversal.

```
// Traverses the given binary tree in inorder.
// Assumes that "visit a node" means to process the node's data item.
inorder(binTree: BinaryTree): void

   if (binTree is not empty)
   {
      inorder(Left subtree of binTree's root)
      Visit the root of binTree
      inorder(Right subtree of binTree's root)
   }
```

Traversals of a Binary Tree

- Postorder traversal.

```
// Traverses the given binary tree in postorder.
// Assumes that "visit a node" means to process the node's data item.
postorder(binTree: BinaryTree): void

   if (binTree is not empty)
   {
      postorder(Left subtree of binTree's root)
      postorder(Right subtree of binTree's root)
      Visit the root of binTree
   }
```

Binary Tree Operations

- Test whether a binary tree is empty.
- Get the height of a binary tree.
- Get the number of nodes in a binary tree.
- Get the data in a binary tree's root.
- Set the data in a binary tree's root.
- Add a new node containing a given data item to a binary tree.

Binary Tree Operations

- Remove the node containing a given data item from a binary tree.
- Remove all nodes from a binary tree.
- Retrieve a specific entry in a binary tree.
- Test whether a binary tree contains a specific entry.
- Traverse the nodes in a binary tree in preorder, inorder, or postorder.

Binary Tree Operations

BinaryTree
+isEmpty(): boolean +getHeight(): integer +getNumberOfNodes(): integer +getRootData(): ItemType +setRootData(newData: ItemType): void +add(newData: ItemType): boolean +remove(data: ItemType): boolean +clear(): void +getEntry(anEntry: ItemType): ItemType +contains(data: ItemType): boolean +preorderTraverse(visit(item: ItemType): void): void +inorderTraverse(visit(item: ItemType): void): void +postorderTraverse(visit(item: ItemType): void): void

FIGURE UML diagram for the class BinaryTree

The ADT Binary Search Tree

- ADT binary tree ill suited for search for specific item
- Binary *search* tree solves problem
- Properties of each node, n
  - n's value greater than all values in left subtree T_L
  - n's value less than all values in right subtree T_R
  - Both T_R and T_L are binary search trees.

Binary Search Tree

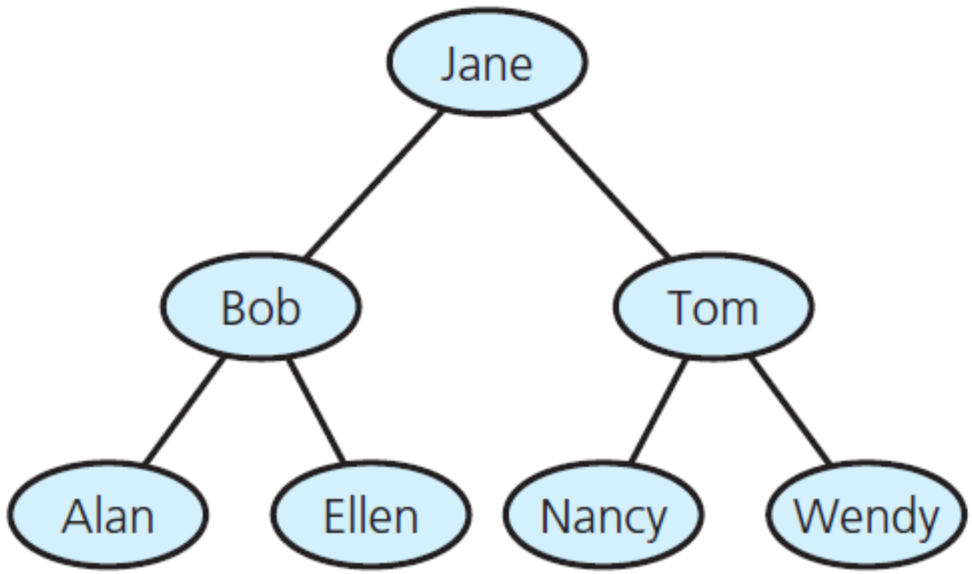


FIGURE A binary search tree of names

The ADT Binary Search Tree

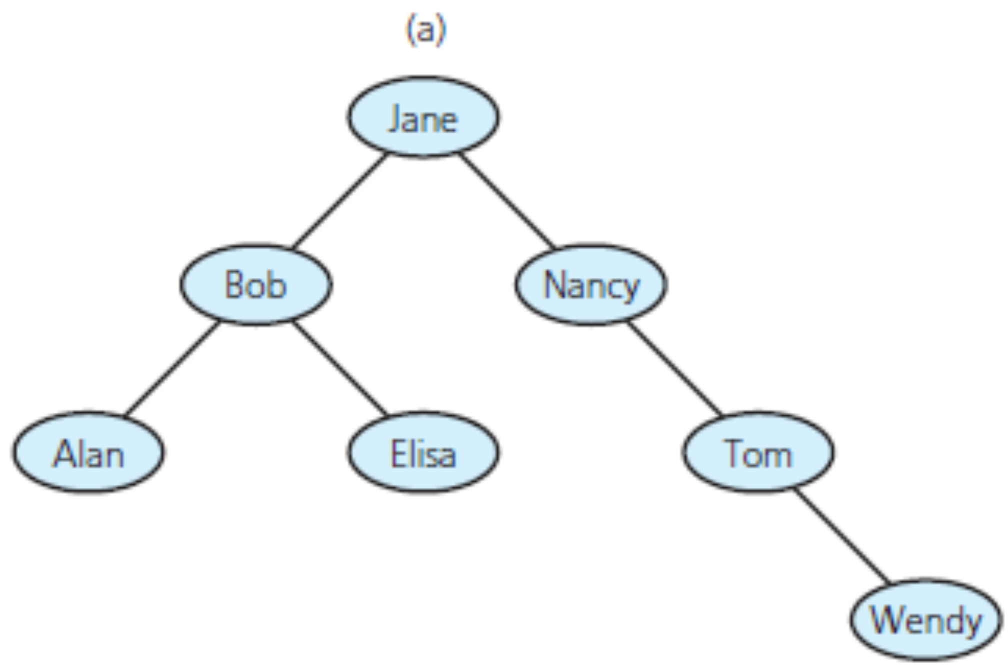


FIGURE Binary search trees with the same data as in Figure

The ADT Binary Search Tree

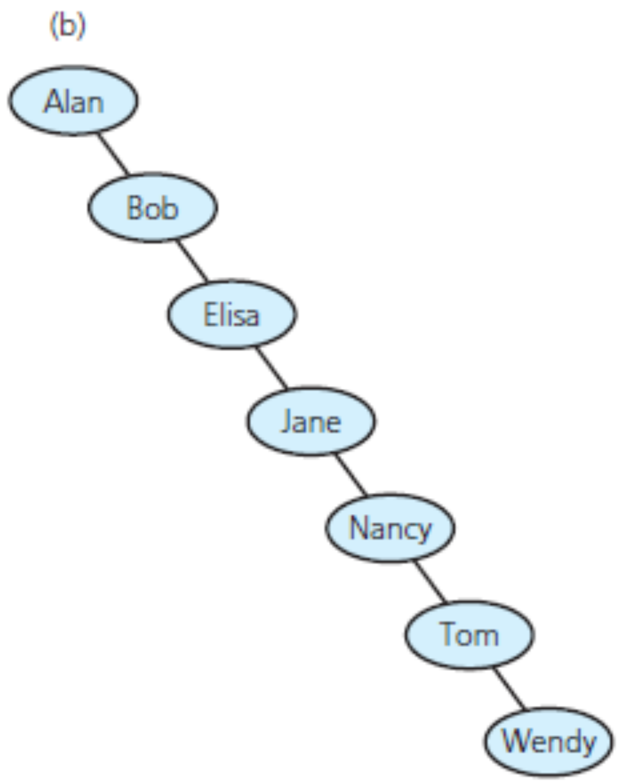


FIGURE Binary search trees
with the same data as in figure before

The ADT Binary Search Tree

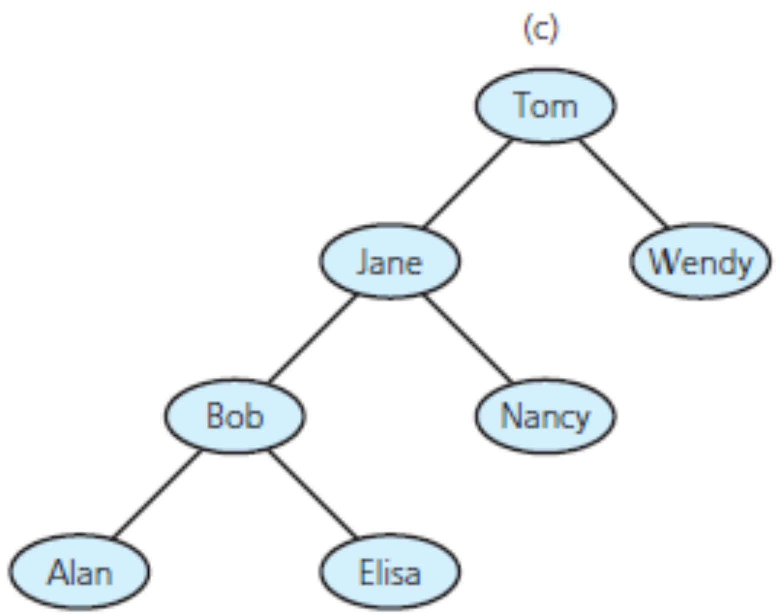


FIGURE Binary search trees
with the same data as before

Binary Search Tree Operations

- Test whether binary search tree is empty.
- Get height of binary search tree.
- Get number of nodes in binary search tree.
- Get data in binary search tree's root.
- Insert new item into binary search tree.
- Remove given item from binary search tree.

Binary Search Tree Operations

- Remove all entries from binary search tree.
- Retrieve given item from binary search tree.
- Test whether binary search tree contains specific entry.
- Traverse items in binary search tree in
  - Preorder
  - Inorder
  - Postorder.

Searching a Binary Search Tree

- Search algorithm for binary search tree

```
// Searches the binary search tree for a given target value.
search(bstTree: BinarySearchTree, target: ItemType)

   if (bstTree is empty)
      The desired item is not found
   else if (target == data item in the root of bstTree)
      The desired item is found
   else if (target < data item in the root of bstTree)
      search(Left subtree of bstTree, target)
   else
      search(Right subtree of bstTree, target)
```

Searching a Binary Search Tree

Alan	Bob	Elisa	Jane	Nancy	Tom	Wendy
0	1	2	3	4	5	6

FIGURE 15-15 An array of names in sorted order

Creating a Binary Search Tree

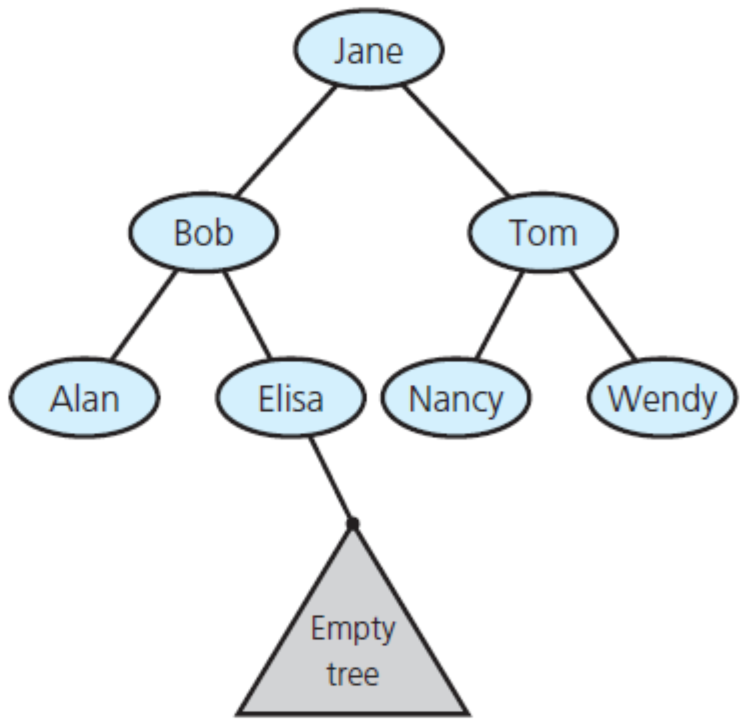


FIGURE 15-16 Empty subtree where the `search` algorithm terminates when looking for Frank

Traversals of a Binary Search Tree

- Algorithm

```
// Traverses the given binary tree in inorder.
// Assumes that "visit a node" means to process the node's data item.
inorder(binTree: BinaryTree): void

   if (binTree is not empty)
   {
      inorder(Left subtree of binTree's root)
      Visit the root of binTree
      inorder(Right subtree of binTree's root)
   }
```

Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Removal	O(log n)	O(n)
Traversal	O(n)	O(n)

FIGURE 15-17 The Big O for the retrieval, insertion, removal, and traversal operations of the ADT binary search tree

Tree Implementations

Array-Based Representation

- Consider required data members

```
TreeNode<ItemType>  tree[MAX_NODES]; // Array of tree nodes
int                 root;            // Index of root
int                 free;            // Index of free list
```

Array-Based Representation

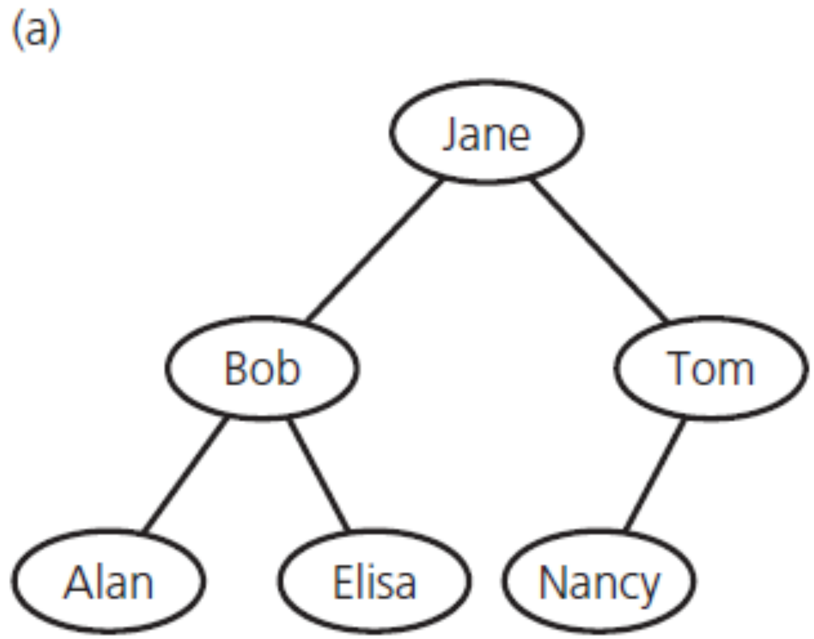


FIGURE 16-1 (a) A binary tree of names;

Array-Based Representation

(b)

tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Elisa	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
•	•	•	•

root: 0

free: 6

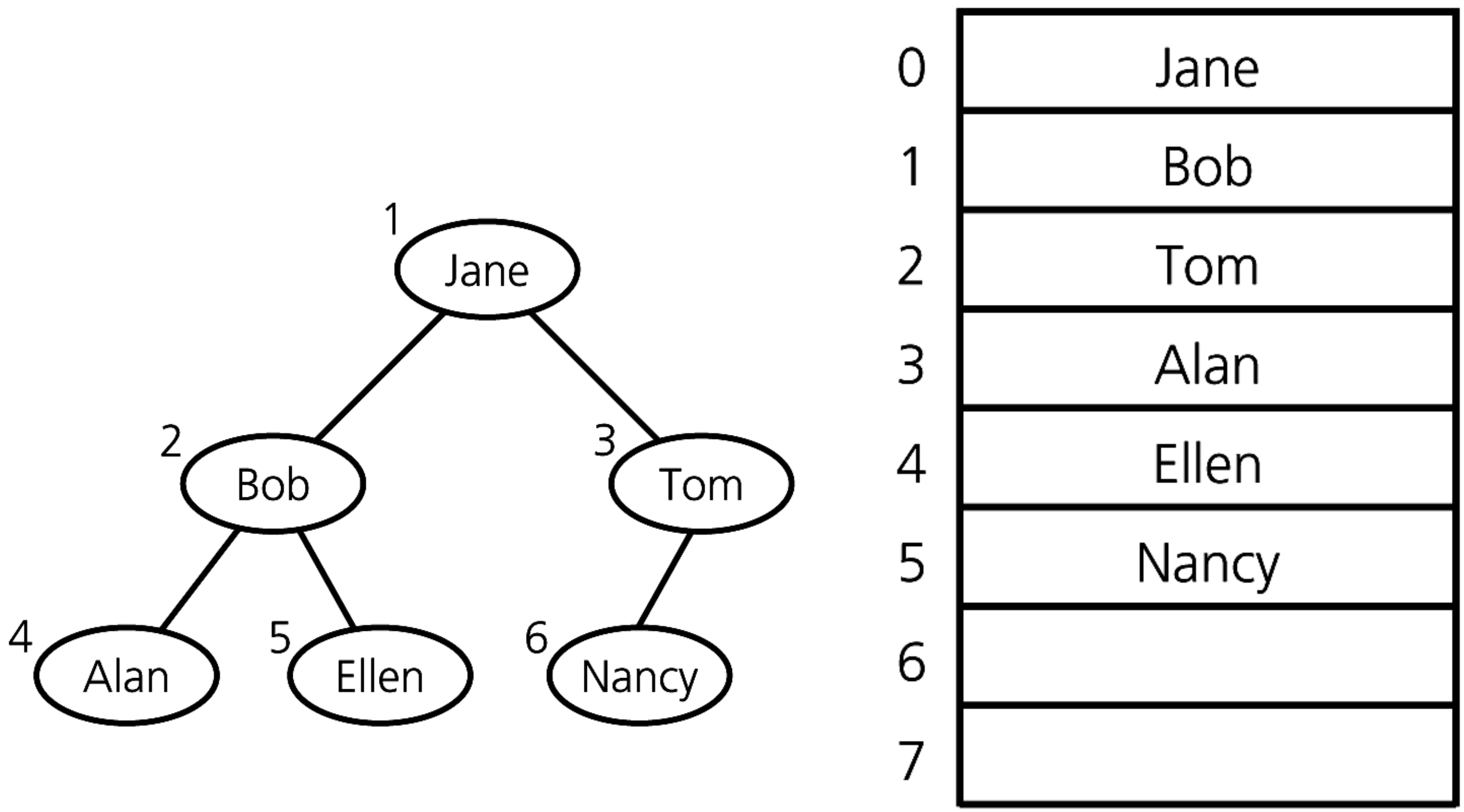
Free list

FIGURE 16-1 (b) its implementation using the array **`tree`**

Array-based Representation of a Complete Binary Tree

- If a binary tree is complete and remains complete
  - A memory-efficient array-based implementation is possible AND attractive

Array-based Representation of a Complete Binary Tree



Heaps

Contents

- The ADT Heap
- An Array-Based Implementation of a Heap
- A Heap Implementation of the ADT Priority Queue
- Heap Sort

The ADT Heap

- Definition
  - A heap is a complete binary tree that either is empty … or …
  - It’s root
    - Contains a value greater than or equal to the value in each of its children, and
    - Has heaps as its subtrees
- Partially Ordered vs Weakly Ordered
    - No relationship between the children of a node

The ADT Heap

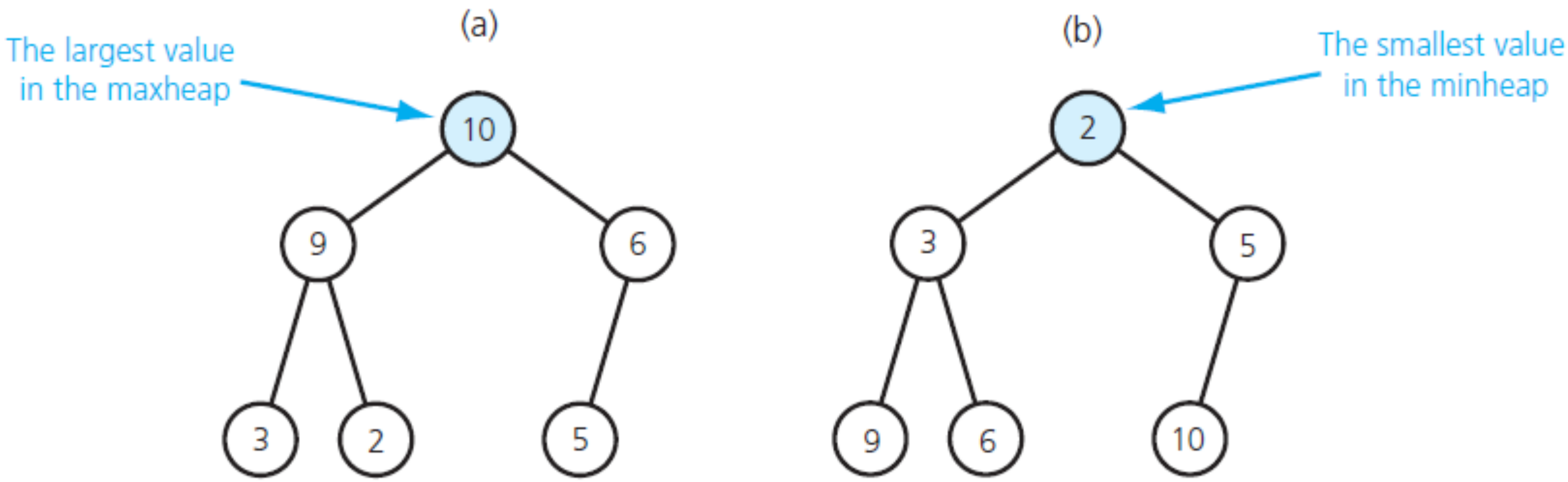


FIGURE 17-1 (a) A maxheap and (b) a minheap

The ADT Heap

Heap
+isEmpty(): boolean +getNumberOfNodes(): integer +getHeight(): integer +peekTop(): ItemType +add(newData: ItemType): boolean +remove(): boolean +clear(): void

FIGURE 17-2 UML diagram for the class Heap

Array-Based Implementation of a Heap

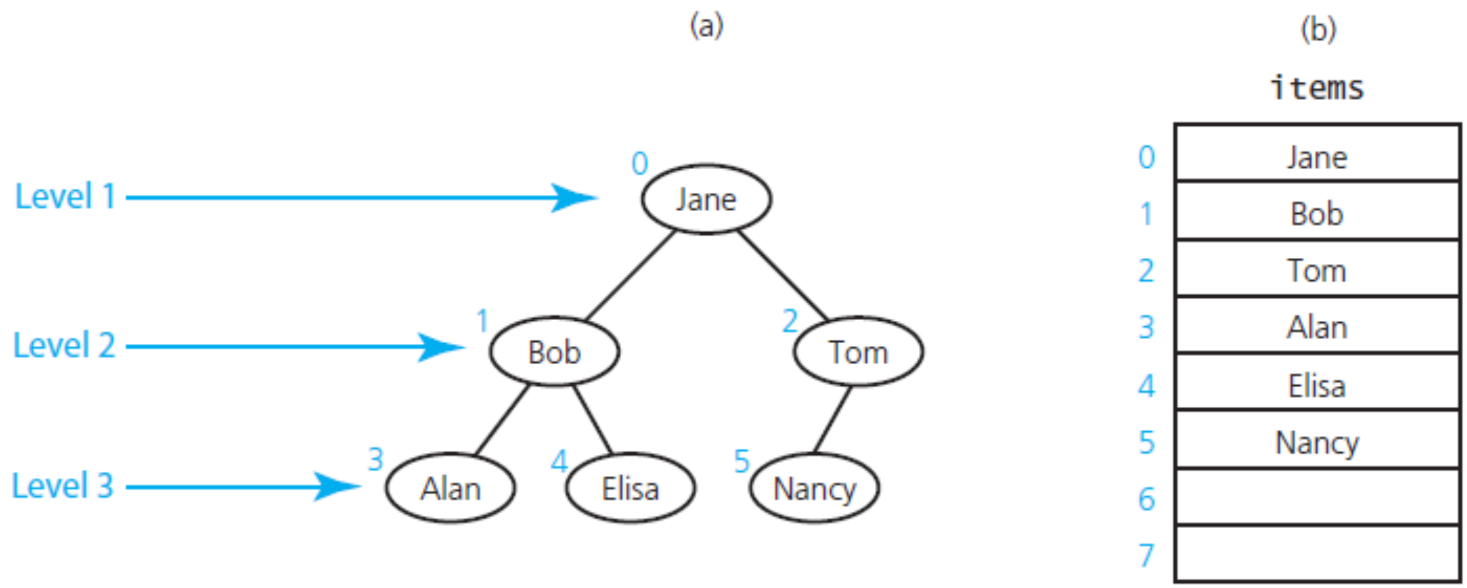


FIGURE 17-3 (a) Level-by-level numbering of a complete binary tree; (b) its array-based implementation

Algorithms for Array-Based Heap Operations

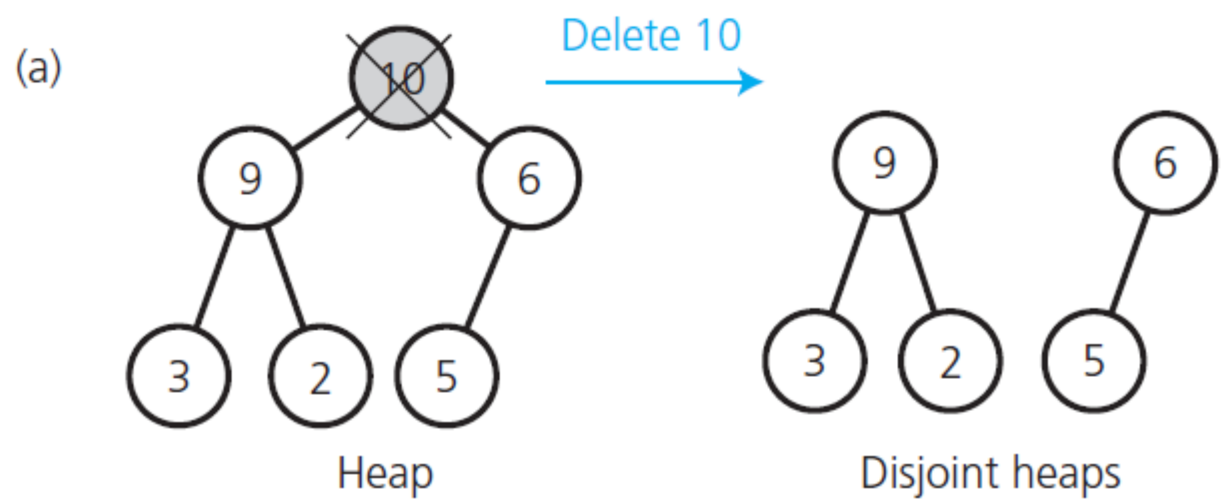


FIGURE 17-4 (a) Disjoint heaps after removing the heap’s root;

Algorithms for Array-Based Heap Operations

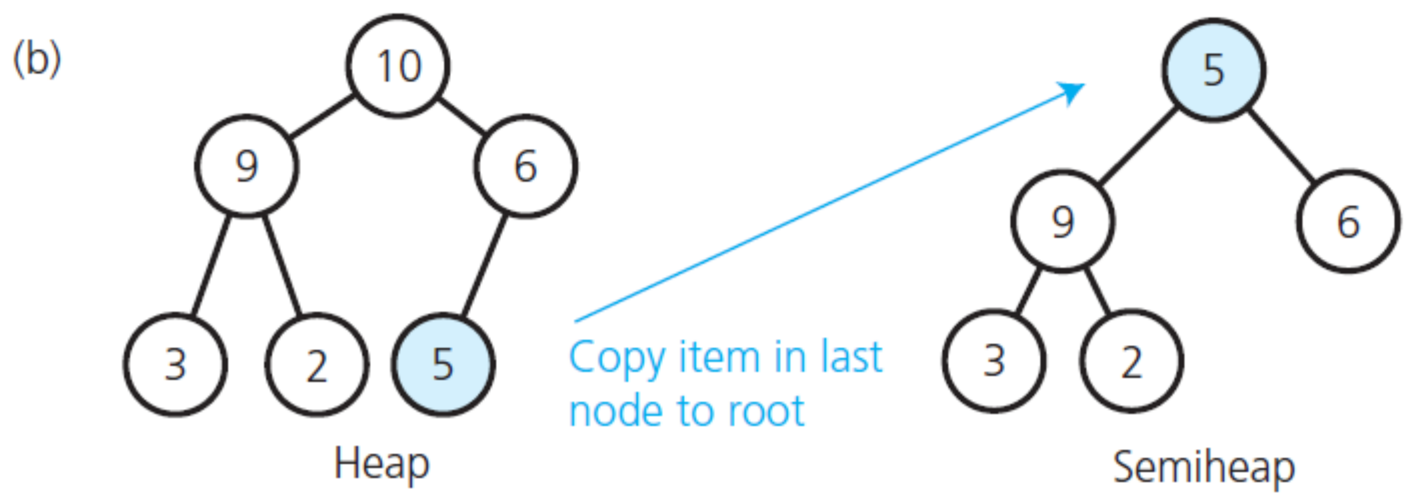


FIGURE 17-4 (b) a semiheap

Algorithms for Array-Based Heap Operations

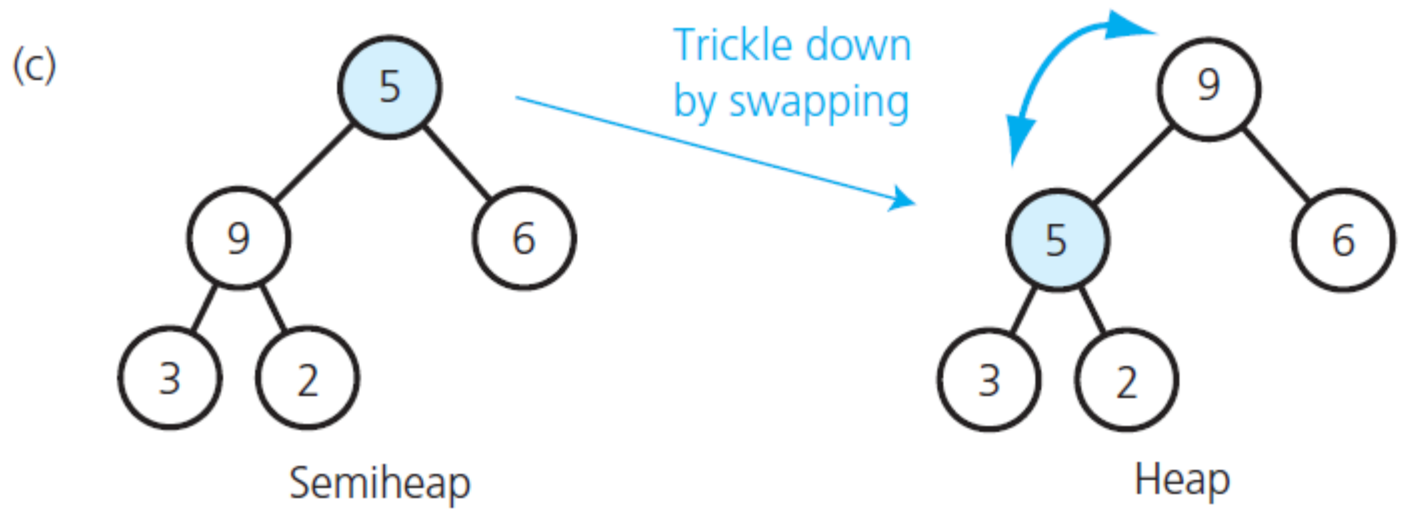


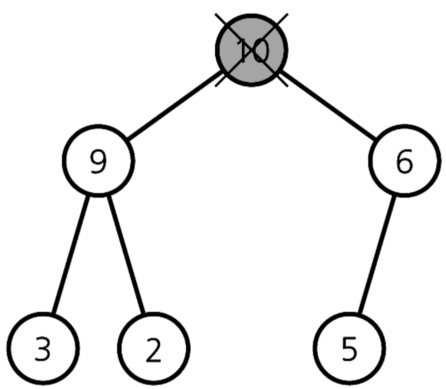
FIGURE 17-4 (c) the restored heap

View algorithm to make this conversion

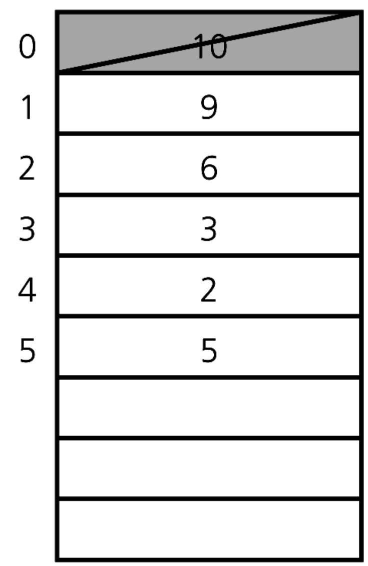
Heaps: `heapDelete`

Strategy

- Step 1: Return the item in the root
  - `rootItem = items[0]`
  - Results in disjoint heaps

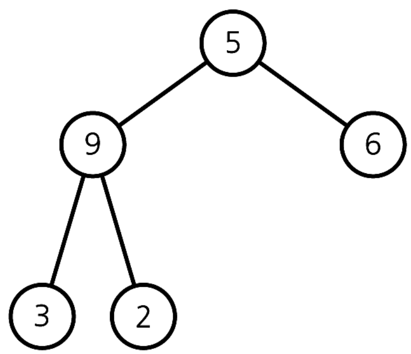


(a)



Heaps: `heapDelete`

- Step 2: Copy the item from the last node into the root: `items[0]= items[size-1]`
- Step 3: Remove the last node: `--size`
  - Results in a semiheap



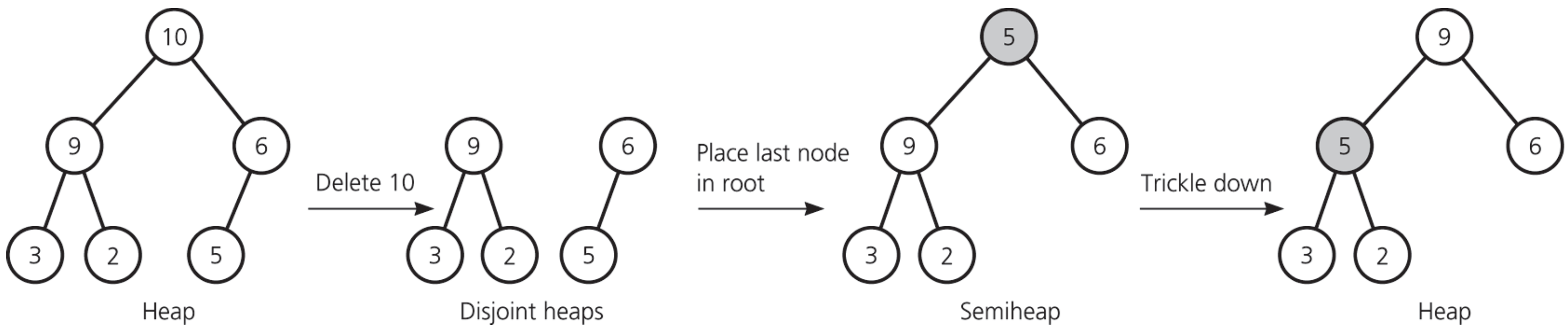
(b)

0	5
1	9
2	6
3	3
4	2

Heaps: `heapDelete`

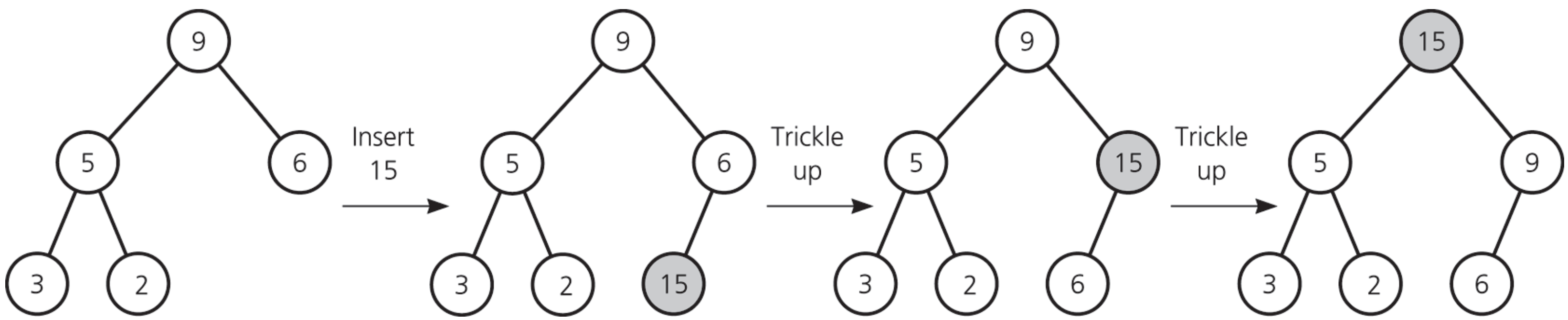
- Step 4: Transform the semiheap back into a heap
  - Use the recursive algorithm `heapRebuild`
    - The root value trickles down the tree until it is not out of place
      - If the root has a smaller search key than the larger of the search keys of its children, swap the item in the root with that of the larger child

Heaps: `heapDelete`



- Efficiency
  - `heapDelete` is O(log n)

Heaps: `heapInsert`



Algorithms for Array-Based Heap Operations

(a)

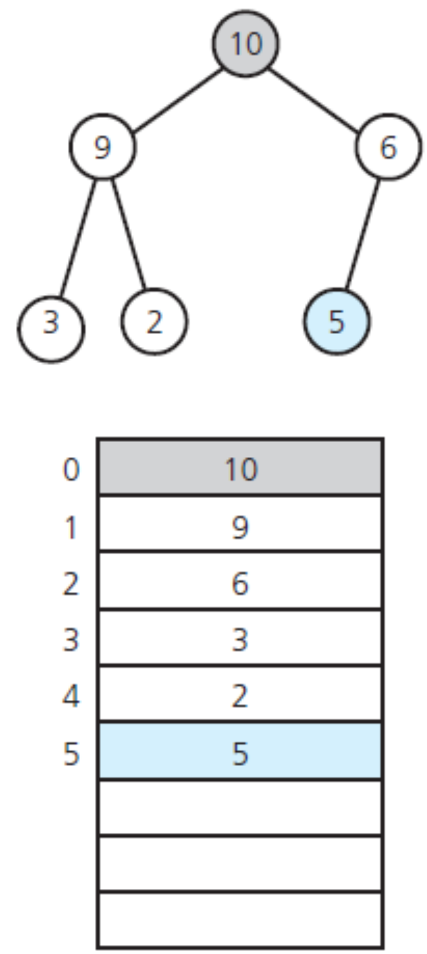


FIGURE 17-5 The array representation of
(a) the heap in Figure 17-4 a;

Algorithms for Array-Based Heap Operations

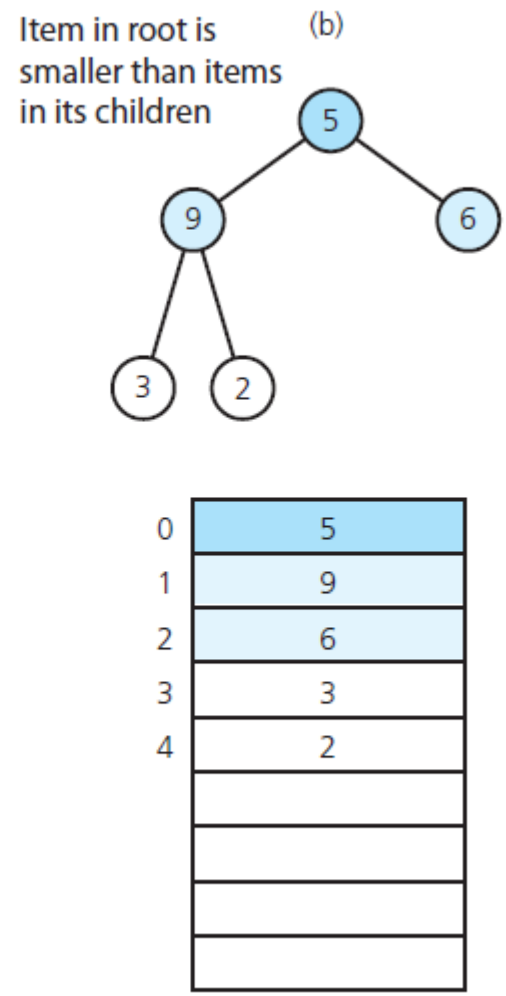


FIGURE 17-5 The array representation of (b) the semiheap in Figure 17-4 b;

Algorithms for Array-Based Heap Operations

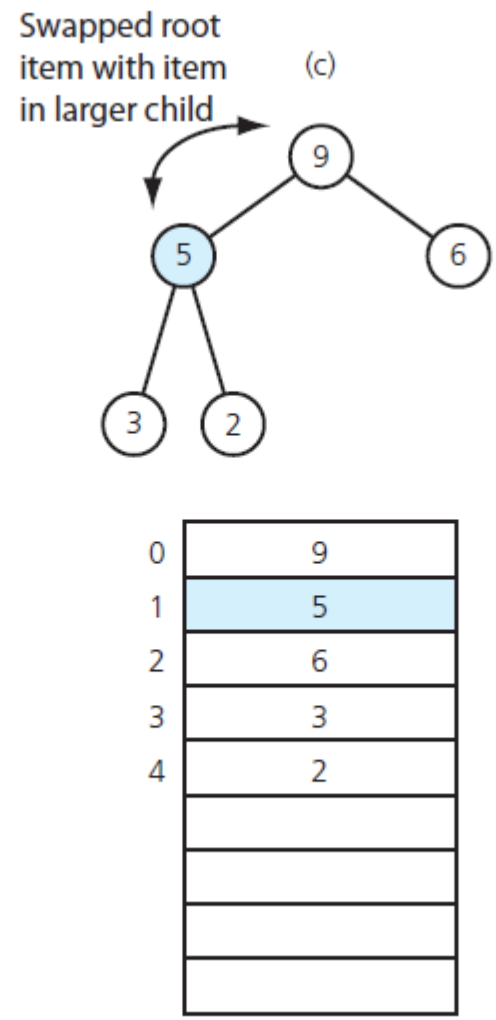


FIGURE 17-5 The array representation of (c) the restored heap in Figure 17-4 c

Algorithms for Array-Based Heap Operations

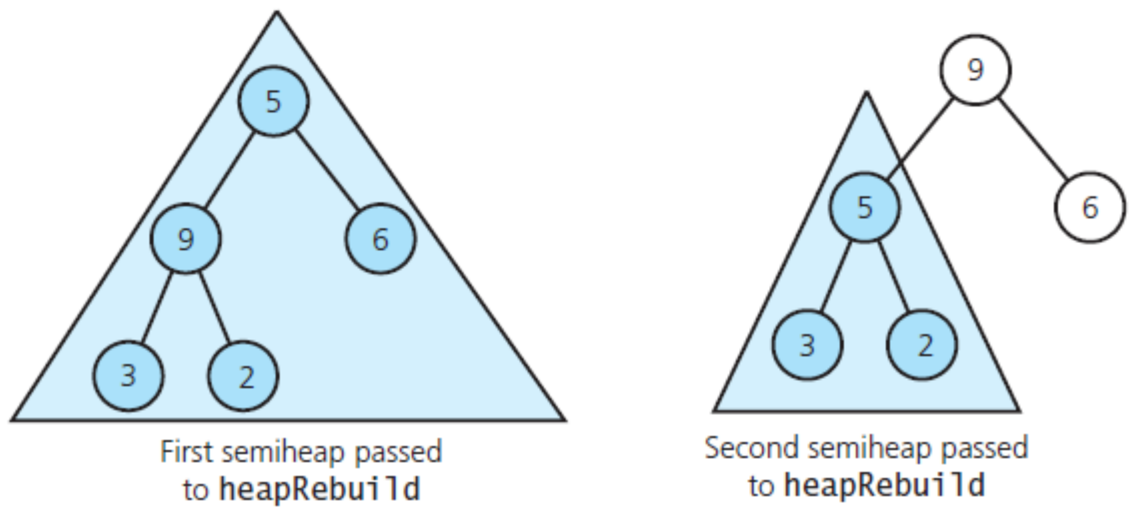


FIGURE 17-6 Recursive calls to **`heapRebuild`**

Algorithms for Array-Based Heap Operations

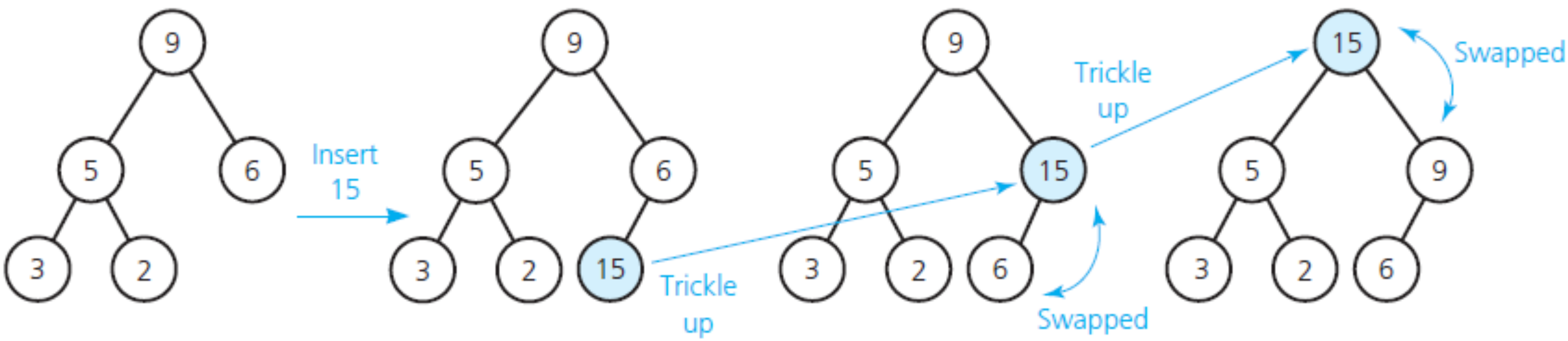


FIGURE 17-7 Insertion into a heap

Algorithms for Array-Based Heap Operations

- Pseudocode for **`add`**

```
// Insert newData into the bottom of the tree
items[itemCount] = newData

// Trickle new item up to the appropriate spot in the tree
newDataIndex = itemCount
inPlace = false
while ( (newDataIndex >= 0) and !inPlace)
{
   parentIndex = (newDataIndex - 1) / 2
   if (items[newDataIndex] < items[parentIndex])
      inPlace = true
   else
   {
      Swap items[newDataIndex ] and items[parentIndex]
      newDataIndex = parentIndex
   }
}
itemCount++
```

The Implementation

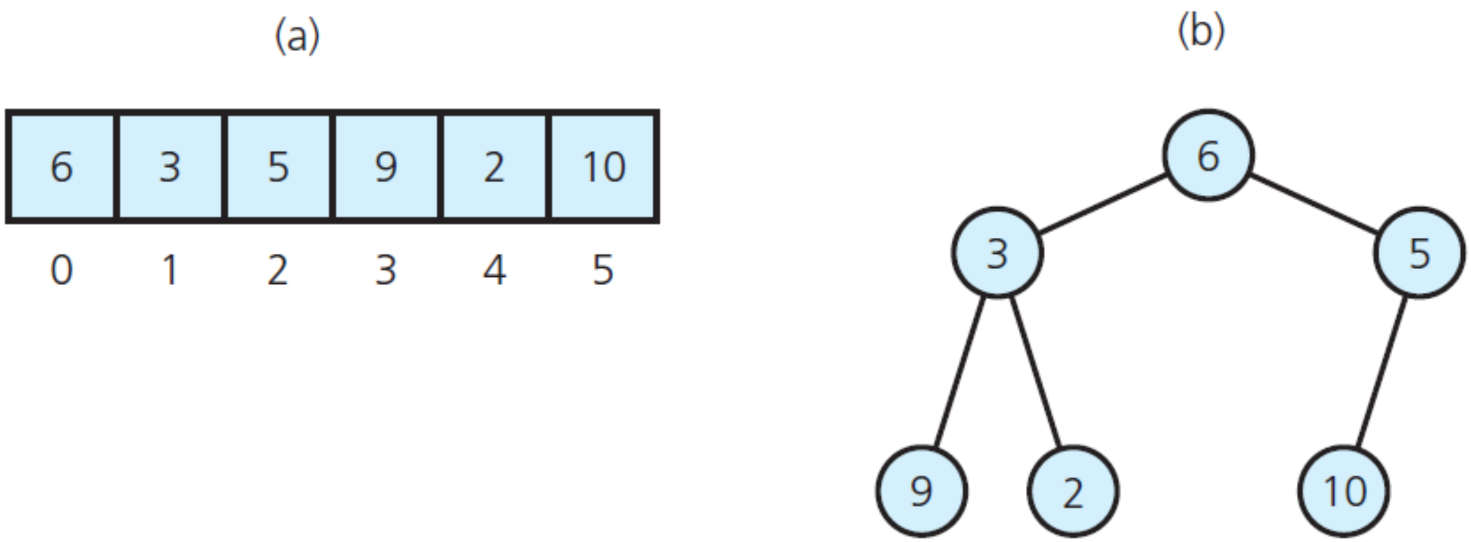


FIGURE 17-8 (a) The initial contents of an array;
(b) the array's corresponding complete binary tree

The Implementation

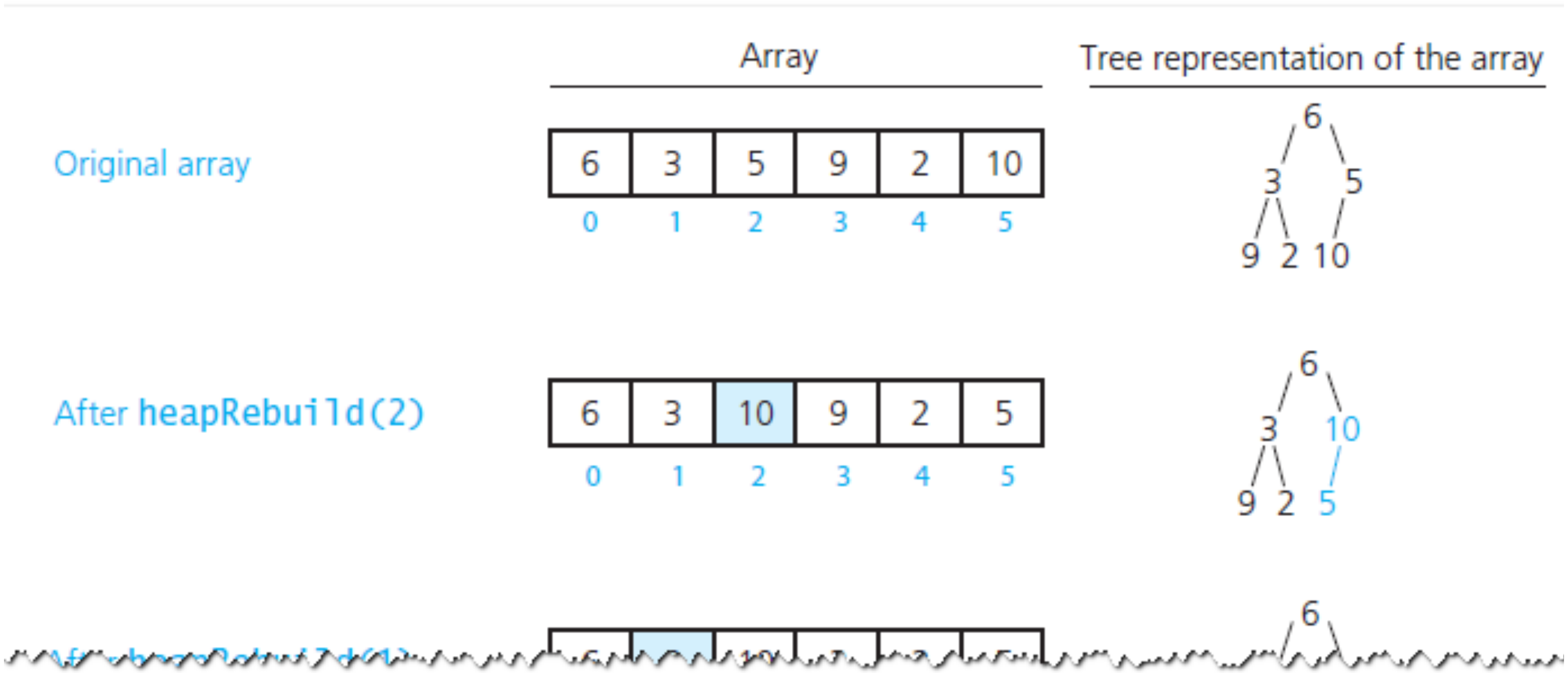


FIGURE 17-9 Transforming an array into a heap

The Implementation

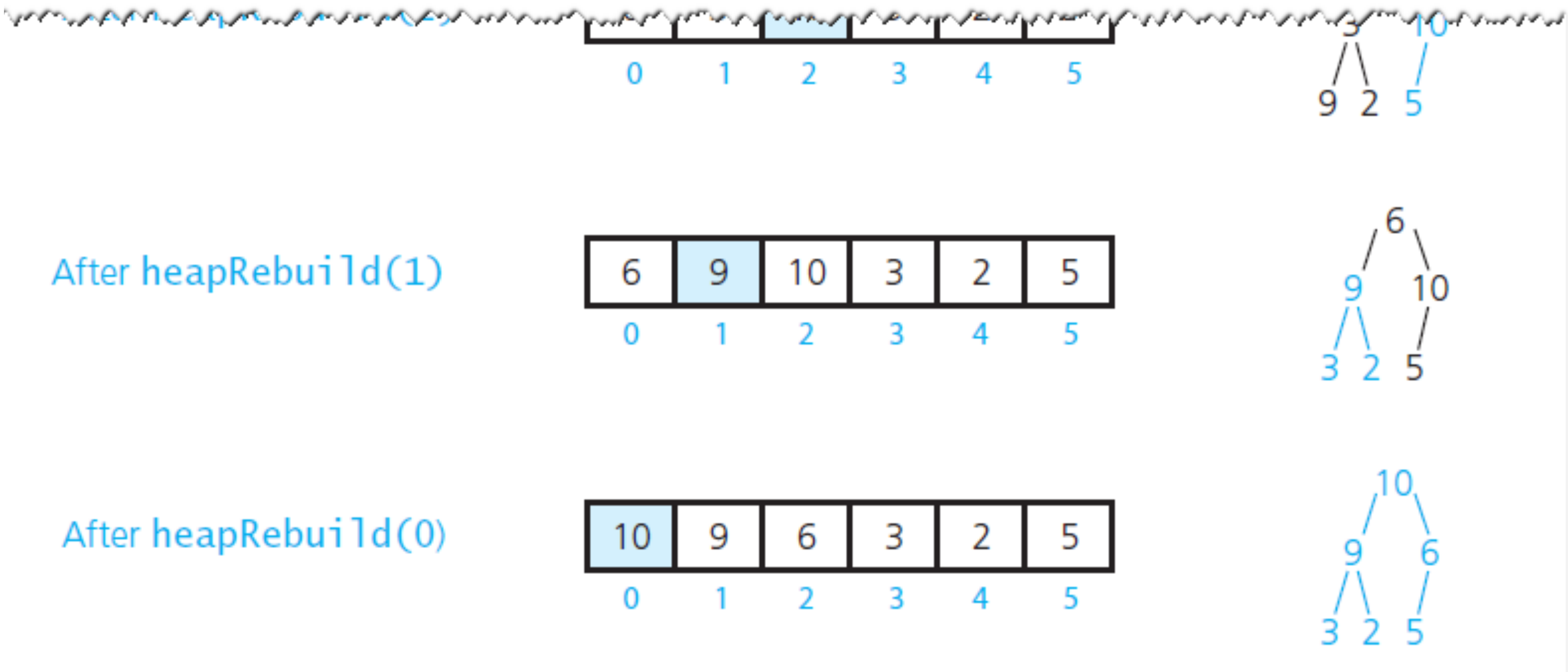


FIGURE 17-9 Transforming an array into a heap

The Implementation

- Method **`heapCreate`**

```
template<class ItemType>
void ArrayMaxHeap<ItemType>::heapCreate()
{
   for (int index = itemCount / 2; index >= 0; index--)
      heapRebuild(index);
}  // end heapCreate
```

The Implementation

- Method **`peekTop`**

```
template<class ItemType>
ItemType ArrayMaxHeap<ItemType>::peekTop() const throw(PrecondViolatedExcep)
{
   if (isEmpty())
      throw PrecondViolatedExcep("Attempted peek into an empty heap.");

   return items[0];
}  // end peekTop
```

Heap Implementation of the ADT Priority Queue

- Using a heap to define a priority queue results in a more time-efficient implementation