

课后答案网，用心为你服务！



[大学答案](#) --- [中学答案](#) --- [考研答案](#) --- [考试答案](#)

最全最多的课后习题参考答案，尽在课后答案网 (www.khdaw.com) !

Khdaw团队一直秉承用心为大家服务的宗旨，以关注学生的学习生活为出发点，

旨在为广大学生朋友的自主学习提供一个分享和交流的平台。

爱校园 (www.aimixiaoyuan.com) 课后答案网 (www.khdaw.com) 淘答案 (www.taodaan.com)

Chapter One: Software Development

1.5 Exercises

1. *Problem analysis and specification* involves determining:
 - output required to solve problem
 - input given
 - other info, such as software usage, feasibility of computer solution

System design involves:

- selecting appropriate data structures to organize the input data
- designing algorithms needed to process the data efficiently

Coding and integration involves

- selecting appropriate programming language
- translating algorithms into well-structured, well-documented, readable code

Verification and validation may involve

- "walking through the code" or, possibly proving correctness of algorithms
- executing the software and correcting errors
- modifying algorithms

System maintenance may involve

- fixing possibly obscure bugs
- modifying software to improve performance
- adding new features, etc.

2. Introductory programming problems are usually well-defined, simple, clearly stated and solvable. Real world problems are generally not well-defined, may be complex, may not have an obvious method of solution.

3. *Top-down* (or modular) design: Partition a problem into simpler subproblems, each of which can be considered individually. Repeat this until the subproblems obtained are simple enough to solve.

Object-oriented design (OOD): Identify a collection of objects, each of which consists of data and operations on the data, that model the real-world objects in the problem and that interact to solve the problem.

4. Encapsulation, inheritance, polymorphism.
5. Structured data types are user-defined structures used to collectively organize data in a given problem. Typical examples include arrays, files, stacks, etc.

6. An algorithm is a procedure that can be executed by a computer. It must be:
 - unambiguous (clear what each instruction is intended to do)
 - simple enough to be done by a computer
 - finite (definite termination)
7. Pseudocode is shorthand notation for representing programs in texts, articles and design phases. It is a mixture of natural language (English) and features and syntax common to high-level languages.
8. Three control structures:
 - *sequential* -- steps performed serially, one after another in established sequence
 - *selection* -- one of several alternative actions is selected and executed
 - *repetition* -- one or more steps is performed repeatedly
9. Typical student programs are small (< 1000 lines), are used infrequently, and are standalone (not part of a system); errors don't have serious consequences and usually quite easy to find and correct; lifetimes of programs are short and thus require little or no maintenance.

Typical real-world programs are large (thousands or millions of lines), used often, and are integrated with other software. Errors may serious — even tragic — results, and may be difficult to find; software may be used for a long period of time and require frequent maintenance.

10. *Syntax errors*: Incorrect syntax (e.g., misplaced or missing semicolons), misspelled words, missing declarations/ The compiler generates error messages that identify the source and type of error.

Run-time errors: Division by zero, integer overflow, missing files, illegal memory access. Execution of the program will be terminated with an error message.

Logical errors: Incorrect coding of algorithm instructions, improper initialization of variables (so garbage values are used), errors in the algorithms (e.g., not checking boundary conditions). The program executes, but incorrect results are produced.

11. Program maintenance is required to fix bugs, accommodate new uses of software (changes in environment), improve performance, or incorporate new features.
12. Many examples can be found on the Internet.

Chapter Two: ADTs -- C-Style types

Exercises 2.2

1. 0000 0000 0110 0011
2. 0001 0100 1010 0000
3. 0000 0000 1111 1111
4. 1111 1111 0000 0001
5. 0000 0100 0000 0000
6. 1111 1100 0000 0000

7. a) 0011 1111 0010 0000 0000 0000 0000 0000
b) same as (a)
8. a) 0100 0001 1100 1101 0000 0000 0000 0000
b) same as (a)
9. a) 0100 0001 0110 1100 1000 0000 0000 0000
b) same as (a)
10. a) 0011 1100 1000 0000 0000 0000 0000 0000
b) same as (a)
11. a) 0011 1101 1100 1100 1100 1100 1100 1100
b) 0011 1101 1100 1100 1100 1100 1100 1101
b) same as (a)
12. a) 0100 0000 0000 0000 1010 0011 1101 0111
b) same as (a)

13. 01000010 01000101
14. 01100010 01100101
15. 01000001 01000010 01001100 01000101
16. 01001110 01101111 00100000 01000111 01101111 00100001
17. 00110001 00110010 00110011 00110100
18. 00110001 00110010 00101110 00110011 00110100

19. 64
20. 28271
21. -16386
22. -16383
23. -26215
24. -21846

25. 3.0
26. 6.0
27. 504.0
28. 1.5
29. 213×2^{36}
30. -3.375

31. null @
32. no
33. false true
34. true true

Exercises 2.3

1. `typedef char byte;`
2. `typedef float Real;`
`typedef float SinglePrecision;`
3. `enum MonthAbbrev {JAN, FEB, MAR, APR, MAY, JUN,`
`JUL, AUG, SEP, OCT, NOV, DEC};`
4. `true`
5. `true`
6. `OCT`
7. `MAR`
8. `OCT`
9. `JUN`
10. `enum Digit {ZERO, ONE, TWO, THREE, FOUR,`
`FIVE, SIX, SEVEN, EIGHT, NINE};`
11. `enum CURRENCY {PENNY = 1, NICKEL = 5, DIME = 10,`
`QUARTER = 25, HALF_DOLLAR = 50, DOLLAR = 100};`

Exercises 2.4

1. 1. Specifies how memory is to be allocated for that variable
2. Associates the variable's name with that memory
3. Initializes that memory with values provided in the declaration (if any).
2. `double * p1, * p2;`
3. `p1 = &d1;`
`p2 = &d2;`
4. Not possible — `i2` is an integer, but `p1` can only store addresses of `doubles`.
5. `int * ptr1 = &i1,`
`* ptr2 = &i2;`
6. `p1 = p2;`
7. `*ptr1 = *ptr2;`
8. `double temp = *p1;`
`*p1 = *p2;`
`*p2 = temp;`
9. `typedef char * CharPointer;`

10. Machine-dependent results (typically 4)
11. Machine-dependent results (typically 4)
12. Machine-dependent results (typically 8)
13. Machine-dependent results (typically 2)
14. Machine-dependent results (typically 4)
15. Machine-dependent results (typically 4)
16. 1
13
43
55
77
99

Chapter Three: Data Structures and Abstract Data Types

Exercises 3.2

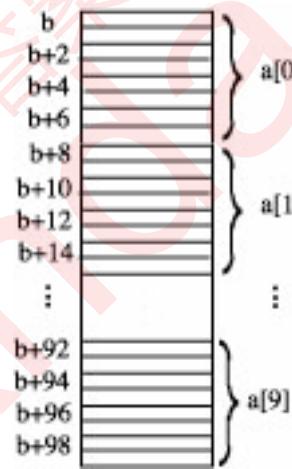
1. 10
2. 80
3. 70
4. 104
5. 28

6. $a[3]: 1003$
7. $a[3]: 1024$
8. $a[4]: 1040$
9. $a[1]: 1004$
10. $a[TUE]: 1008$

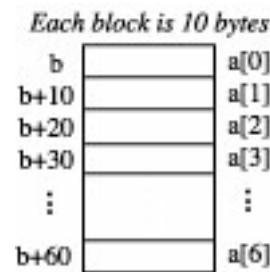
11. $a[i] = \text{base} + i$



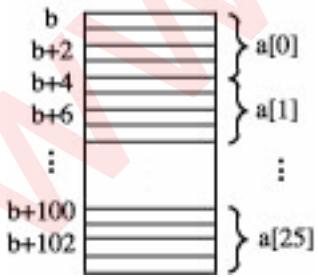
12. $a[i] = \text{base} + 8*i$



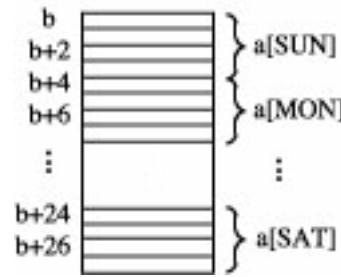
13. $a[i] = \text{base} + 10*i$



14. $a[i] = \text{base} + 10*i$



15. $a[i] = \text{base} + 4*i$



Exercises 3.3

1. 108, 112
2. 164, 356
3. 192, 152
4. 250, 420

5. 116, 112
6. 356, 188
7. 126, 296
8. 330, 290

9. a) base + w*i*N2 + w*j b) base + w*i + w*N1*j, where N1 and N2 are dimensions of 2-D array

10. base + w*i*N2*N3 + w*j*N3 + w*k, where N1, N2 and N3 are dimensions of 3-D array

11. base + w*i*N2*N3*N4 + w*j*N3*N4 + w*k*N4 + w*l

Exercises 3.4

1.

```
int n;
cin >> n;
double * doublePtr = new double[n];
```

2.

```
for (int index = 0; index < n; index++)
    cin >> doublePtr[index];
```

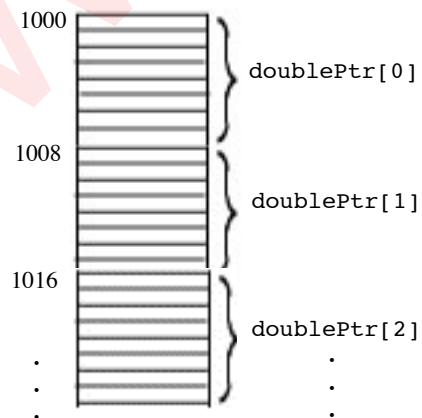
3.

```
double sum = 0.0;
for (int index = 0; index < n; index++)
    sum += doublePtr[index];
cout << "Average is: " << sum/n << endl;
```

4.

```
delete[] doublePtr;
```

5.



6. 4 8 1.1
4 8 2.2
4 8 3.3
4 8 4.4
4 8 5.5

Chapter 4: More about OOP and ADTs—Classes

Exercises 4.5

```
1. class Date
{ ...
private:
    int myMonth, myDay, myYear;
};

2. class PhoneNumber
{ ...
private:
    int myAreaCode,
        myLocalExchange,
        myNumber;
};

3. class Checker
{ ...
private:
    int myRow, myCol;
};

4. class CartesianPoint
{ ...
private:
    double myX,
           myY;
    // ...
};

5. class PolarPoint
{ ...
private:
    double myR,
           myTheta;
    // ...
};

6. enum Suit {HEART, SPADE, DIAMOND, CLUB};
class Card
{ ...
private:
    Suit mySuit;
    int myFace;
};
```

- 7-18. There are obviously many different ways that these classes can be implemented. They are all straightforward but require quite a lot of code. For that reason, only one complete implementation is given here -- for Problem 12, an implementation of the class card in Exercise 6. The others are similar — choose appropriate types for the members of each class and supply member functions to set and retrieve those members, in addition to member functions for I/O, etc. Only some partial solutions are given here.

7. In Date.h:

```
class Date
{
public:
    Date();
    Date(unsigned m, unsigned d, unsigned y);

    void display();
    int getMonth();
    int getDay();
    int getYear();

    void setMonth(unsigned m);
    void setDay(unsigned d);
    void setYear(unsigned y);
    // ...

private:
    int myMonth, myDay, myYear;
};

In Date.cpp:
// Could inline these in Date.h for efficiency

#include <cassert>
using namespace std;

// Definition of default constructor
Date::Date()
myMonth(1), myDay(1), myYear(2004)
{ }

// Definition of explicit-value constructor
Date::Date(unsigned m, unsigned d, unsigned y)
{
    assert( (m == 1 || m == 3 || m == 5 || m == 7 ||
              m == 8 || m == 10 || m == 12) &&
            (1 <= d && d <= 31) ||
            (m == 4 || m == 6 || m == 9 || m == 11) &&
            (1 <= d && d <= 30) ||
            ((m == 2) && (1 <= d && d <= 29) );
    myMonth = m;
    myDay = d;
    myYear = y;
}

// Definition of display()
void Date::display()
{
    cout << myMonth << "/"
        << (myDay < 10 ? "0" : "") << myDay << "/"
        << myYear;
}

// Definition of getMonth()
int Date::getMonth()
{ return myMonth; }
```

```
// Definitions of getDay() and getYear() are similar  
// ...  
  
// Definition of setMonth()  
void Date::setMonth(unsigned m)  
{  
    assert(1 <= m && m <= 12);  
    myMonth = m;  
}  
  
// Definitions of setDay() and setYear() are similar  
// ...
```

8. Same idea as #7

9. In Checker.h:

```
class Checker  
{  
public:  
    void move(int r, int c);  
    int getCol();  
    int getRow();  
    ...  
private:  
    int row, col;  
};
```

In Checker.cpp:

```
// Could inline these in Checker.h for efficiency  
void Checker::move(int r, int c)  
{  
    myRow = r;  
    myCol = c;  
}  
  
int Checker::getCol()  
{return myCol;}  
  
int Checker::getRow()  
{return myRow;}  
  
// ...
```

10. In CartesianPoint.h:

```
#include <iostream>  
using namespace std;  
  
class CartesianPoint  
{  
public:  
    CartesianPoint();  
    CartesianPoint(double xVal, double yVal);  
    double getX() const;  
    double getY() const;  
    void setX(double xVal);  
    void setY(double yVal);
```

```
private:  
    double myX, myY;  
};  
  
In CartesianPoint.cpp:  
// Could inline these in CartesianPoint.h for efficiency  
  
CartesianPoint::CartesianPoint()  
: myX(0), myY(0)  
{}  
  
CartesianPoint::CartesianPoint(double xVal, double yVal)  
: myX(xVal), myY(yVal)  
{}  
  
double CartesianPoint::getX() const  
{ return myX; }  
  
double CartesianPoint::getY() const  
{ return myY; }  
  
void CartesianPoint::setX(double xVal)  
{ myX = xVal; }  
  
void CartesianPoint::setY(double yVal)  
{ myY = yVal; }  
  
ostream & operator<<(ostream & out, const CartesianPoint & point)  
{  
    out << "(" << point.getX() << ", " << point.getY() << ')';  
    return out;  
}  
  
istream & operator>>(istream & in, CartesianPoint & point)  
{  
    char punc;  
    double x, y;  
    in >> punc >> x >> punc >> y >> punc;  
    point.setX(x);  
    point.setY(y);  
    return in;  
}
```

11. Same idea as #10

12.

```
// --- Card.h  
//=====  
  
#ifndef CARD  
#define CARD  
#include <iostream>
```

```
class Card
{
    public:
        Card();
        /*-----
            Default constructor
            Precondition: None
            Postcondition: myValue == mySuit == ' ';
            isValid() will return false.
-----*/
        Card(char value, char suit);
        /*-----
            Explicit-value constructor
            Precondition: value is a valid card value and suit is
            a valid suit.
            Postcondition: myValue == value && mySuit == ' '.
-----*/
        char getValue() const;
        /*-----
            Retrieve the value of this card.
            Precondition: None.
            Postcondition: Value of card (A, 2-9, T, J, Q, or K,
            or space if invalid) is returned.
-----*/
        char getSuit() const;
        /*-----
            Retrieve the suit of this card.
            Precondition: None.
            Postcondition: Suit of card (H, D, S, or C, or space
            if invalid) is returned.
-----*/
        bool isValid(char value, char suit) const;
        /*-----
            Check if value is a valid card value and suit is a
            valid suit.
            Precondition: None.
            Postcondition: true is returned if value and suit are
            valid, and false if not.
-----*/
        void setCard(char value, char suit);
        /*-----
            Change the value and suit of this card.
            Precondition: value is a valid card value and suit is
            a valid suit.
            Postcondition: myValue == value && mySuit == ' '.
-----*/
        void toRandom();
        /*-----
            Set this card to a random card.
            Precondition: None.
            Postcondition: myValue and mySuit are assigned a
            randomly selected value and suit.
-----*/
}
```

```
void print(ostream & out) const;
/*-----
   Outputs this card.
   Precondition: ostream out is open.
   Postcondition: a string representation of this card
   is output to out.
-----*/
void read(istream & in);
/*-----
   Inputs a card.
   Precondition: istream in is open and contains valid
   string representations of a card's value and suit.
   Postcondition: strings have been read from in and
   assigned to myValue and mySuit, provided the
   strings are valid.
-----*/
private:
    char myValue,
        mySuit;
};

inline ostream & operator<<(ostream & out, const Card & card);
/*-----
   Output operator.
   Precondition: ostream out is open.
   Postcondition: a string representation of this card
   is output to out via print().
-----*/
inline istream & operator>>(istream & in, Card & card);
/*-----
   Input operator.
   Precondition: istream in is open.
   Postcondition: strings have been read from in via
   read() and assigned to myValue and mySuit,
   provided the strings are valid.
-----*/
#endif

// --- Card.cpp
//=====
#include <string>           // string
#include <cctype>            // islower(), // toupper()
using namespace std;        // RandomInt
#include "RandomInt.h"       // RandomInt

#include "Card.h"

//--- Definition of default constructor
Card::Card()
: myValue(' '), mySuit(' ')
{ }
```

```
///-- Definition of explicit-value constructor
Card::Card(char value, char suit);
{
    if isValid(value, suit)
    {
        myValue = value;
        mySuit = suit;
    }
}

///-- Definition of getValue()
char getValue() const
{ return myValue; }

///-- Definition of getSuit()
char getSuit() const
{ return mySuit; }

///-- Definition of isValid()
bool isValid(char value, char suit) const
{
    if (islower(value))           // force uppercase
        value = toupper(value);
    if (islower(suit))           // force uppercase
        suit = toupper(suit);

    return (
        (value == 'A') ||          // ace
        ('2' <= value) && (value <= '9') ) ||      // 2 - 9
        (value == 'T') ||          // 10
        (value == 'J') ||          // jack
        (value == 'Q') ||          // queen
        (value == 'K'))           // king
    )
    &&
    (
        (suit == 'H') ||          // hearts
        (suit == 'D') ||          // diamonds
        (suit == 'S') ||          // spades
        (suit == 'C') )           // clubs
    )
}

///-- Definition of setCard()
void Card::setCard(char value, char suit)
{
    if isValid(value, suit)
    {
        myValue = value;
        mySuit = suit;
    }
}

///-- Definition of toRandom()
RandomInt cardNumber(0,51);
// Declared outside all functions so random number
// generator is initialized only once.
```

```
void Card::toRandom()
{
    cardNumber.generate();
    switch ( int(cardNumber) % 13 )           // set random value
    {
        case 0: myValue = 'A'; break;
        case 1: myValue = '2'; break;
        case 2: myValue = '3'; break;
        case 3: myValue = '4'; break;
        case 4: myValue = '5'; break;
        case 5: myValue = '6'; break;
        case 6: myValue = '7'; break;
        case 7: myValue = '8'; break;
        case 8: myValue = '9'; break;
        case 9: myValue = 'T'; break;
        case 10: myValue = 'J'; break;
        case 11: myValue = 'Q'; break;
        case 12: myValue = 'K'; break;
        default: myValue = ' ';
    }

    switch ( int(cardNumber) / 13 )           // set random suit
    {
        case 0: mySuit = 'H'; break;
        case 1: mySuit = 'D'; break;
        case 2: mySuit = 'S'; break;
        case 3: mySuit = 'C'; break;
        default: mySuit = ' ';
    }
}

//-- Definition of print()
void Card::print(ostream & out) const
{
    switch(myValue)
    {
        case 'A': out << "Ace"; break;
        case '2': out << "2"; break;
        case '3': out << "3"; break;
        case '4': out << "4"; break;
        case '5': out << "5"; break;
        case '6': out << "6"; break;
        case '7': out << "7"; break;
        case '8': out << "8"; break;
        case '9': out << "9"; break;
        case 'T': out << "10"; break;
        case 'J': out << "Jack"; break;
        case 'Q': out << "Queen"; break;
        case 'K': out << "King"; break;
        default: out << "ERROR"; break;
    }

    out << ' ';
}
```

```
switch(mySuit)
{
    case 'H':  out << "Hearts";   break;
    case 'D':  out << "Diamonds"; break;
    case 'S':  out << "Spades";   break;
    case 'C':  out << "Clubs";    break;
    default:   out << "ERROR";    break;
}
}

//-- Definition of read()
void Card::read(istream & in)
{
    string inStr;
    char valueChar, suitChar;

    in >> inStr;
    valueChar = inStr[0];
    if (islower(valueChar))
        valueChar = toupper(valueChar);
    else if ( (inStr.length() > 1) && (inStr[1] == '0') )
        valueChar = 'T';
    in >> inStr;
    suitChar = inStr[0];
    if (islower(suitChar))
        suitChar = toupper(suitChar);

    if (isValid(valueChar, suitChar) // Check for legal suit
    {
        myValue = valueChar;
        mySuit = suitChar;
    }
    else
    {
        myValue = ' ';
        mySuit = ' ';
    }
}

//-- Definition of <<
inline ostream & operator<<(ostream & out, const Card & card)
{
    card.print(out);
    return out;
}

//-- Definition of >>
inline istream & operator>>(istream & in, Card & card)
{
    card.read(in);
    return in;
}
```

13.

```
enum HairColor {BALD, BLONDE, BROWN, BRUNETTE, GRAY, RED, WHITE};  
enum EyeColor {BLUE, BROWN, HAZEL, GREEN, GRAY};  
enum MaritalStatus {DIVORCED, MARRIED, SINGLE, WIDOW, WIDOWER};  
class Person  
{  
public:  
    // ... operations ...  
  
private:  
    string myName;  
    short int myBirthmonth, myBirthday, myBirthyear;  
    short int myAge;  
    char myGender;  
    string mySocSecNumber;  
    short int myHeight;  
    float myWeight;  
    string myCity;  
    string myState  
    string myPhoneNumber;  
    HairColor myHairColor;  
    EyeColor myEyeColor;  
    MaritalStatus myMaritalStatus;  
};
```

14.

```
class Inventory  
{  
public:  
    // ... operations ...  
  
private:  
    int myItemNumber;  
    short int myInStock;  
    double myUnitPrice;  
    short int myMinimumInventory;  
    string myItemName;  
};
```

15.

```
class UserId  
{  
public:  
    // ... operations ...  
  
private:  
    int myIDNumber;  
    string myLastName;  
    string myFirstName;  
    string myPassword;  
    short int myResourceLimit;  
    double myResourcesUsed;  
};
```

```
16.
class Student
{
public:
    // ... operations ...

private:
    int myStudentNumber;
    string myLastName;
    string myFirstName;
    char myMiddleInitial;
    string myCity;
    string myState
    string myPhoneNumber;
    char myGender;
    short int myYear;
    string myMajor;
    int myCreditsToDate;
    double myGPA;
};
```

Chapter 5: Standard C++ Input/Output and String Classes

Exercises 5.2

1.

```
int stringCount(const string & str, const string & target)
/*-----
   Find the number of times string str appears in string target.
-----*/
{
    int matches = 0;
    int position = target.find(str);
    while (position != string::npos)
    {
        matches++;
        position = target.find(str, position + 1);
    }
    return matches;
}
```

2.

```
string monthName(int monthNumber)
/*-----
   Convert a month number to its corresponding name.
-----*/
{
    switch(monthNumber)
    {
        case(1) : return "January";
        case(2) : return "February";
        case(3) : return "March";
        case(4) : return "April";
        case(5) : return "May";
        case(6) : return "June";
        case(7) : return "July";
        case(8) : return "August";
        case(9) : return "September";
        case(10): return "October";
        case(11): return "November";
        case(12): return "December";
        default : cout << "\nMonth: illegal month number: "
                  << monthNumber << endl;
                  return "";
    }
}
```

3.

```
#include <cctype>           // isupper(), tolower()

int monthNumber(string monthName)
/*-----
   Convert a month name to its corresponding number.
-----*/
{
    for(int i = 0; i < monthName.length(); i++)
        if(isupper(monthName[i]))
            monthName[i] = tolower(monthName[i]);
```

```
if(monthName == "january")
    return 1;
if(monthName == "february")
    return 2;
if(monthName == "march")
    return 3;
if(monthName == "april")
    return 4;
if(monthName == "may")
    return 5;
if(monthName == "june")
    return 6;
if(monthName == "july")
    return 7;
if(monthName == "august")
    return 8;
if(monthName == "september")
    return 9;
if(monthName == "october")
    return 10;
if(monthName == "november")
    return 11;
if(monthName == "december")
    return 12;
cout << "\nMonthNumber: illegal month name: " << monthName << endl;
return 0;
}

4.
#include <cctype> // islower(), toupper(), isupper(), tolower()

string lowerToUpper(string str )
/*
-----*
   Find uppercase equivalent of a string str.
-----*/
{
    for(int i = 0; i < str.length(); i++)
        if(islower(str[i]))
            str[i] = toupper(str[i]);

    return str;
}

string upperToLower(string str)
/*
-----*
   Find lowercase equivalent of a string str.
-----*/
{
    for(int i = 0; i < str.length(); i++)
        if(isupper(str[i]))
            str[i] = tolower(str[i]);

    return str;
}
```

5.

```
string replaceAll(string str, string substring, string newSubstring )
/*
   Find a string str with all occurrences of substring replaced
   by newSubstring.
*/
{
    int pos = -1;
    for(;;)
    {
        pos = str.find(substring, pos + 1);
        if (pos == string::npos) return str;
        str.replace(pos, substring.length(), newSubstring);
    }
}
```

6.

```
string nameChange(string firstName, string middleName, string lastName)
/*
   Construct a string of the form
       "lastName, firstName, middle-initial."
   from strings firstName, middleName, lastName.
*/
{
    return lastName + ", " + firstName + " "
        + middleName.substr(0,1) + ".";
}
```

7.

```
#include <cctype>      //isspace

string nameChange(string name)
/*
   Construct a string of the form
       "LastName, FirstName MiddleInitial."
   from a string name of the form
       "FirstName MiddleName LastName"
*/
{
    unsigned endFirst,
            startMiddle,
            startLast;
    endFirst = name.find(" ", 0);
    startMiddle = endFirst + 1;
    while (isspace(name[startMiddle]))
        startMiddle++;

    startLast = name.find(" ", startMiddle);
    while (isspace(name[startLast ]))
        startLast++;

    return name.substr(startLast, name.length()-1) + ", "
        + name.substr(0, endFirst + 1) + name.substr(0,1) + ".";
}
```

8.

```
int parseInt(const string & s, bool & success)
/*
 *-----*
 * Convert a string into an integer
 *
 * Precondition: s contains a string of digits, possibly preceded
 * by + or -.
 * Postcondition: Corresponding integer is returned and success is
 * true if conversion is possible; otherwise, -1 is returned
 * and success is false.
 */
{
    success = true;
    char sign = '+';
    string local = s;

    if (s[0] == '+' || s[0] == '-')
    {
        sign = s[0];
        local.erase(0,1);           //strip sign
    }
    bool allDigits = true;

    for (int index = 0; index < local.length(); index++)
        if (!isdigit(local[index]))
    {
        allDigits = false;
        break;
    }

    if (allDigits)
    {
        int holder = atoi(local.data());
        if (sign == '-')
            return -holder;
        else
            return holder;
    }
    else
    {
        success = false;           //indicate poorly formed string
        return -1;
    }
}
```

9.

```
/* Convert string to real number.
 * Note: scientific or decimal format may be used */

#include <cctype>
#include <cstdlib>
```

```
double parseReal(const string & s, bool & success)
/*-----
   Convert a string into a real number.
   Note: scientific or decimal format may be used.

   Precondition: s contains a string of characters in a real value
   Postcondition: Corresponding real value is returned and success is
      true if conversion is possible; otherwise, -1 is returned
      and success is false.
-----*/
{
    success = true;
    char sign = '+';
    string local = s;

    if (s[0] == '+' || s[0] == '-')
    {
        sign = s[0];
        local.erase(0,1);           //strip sign
    }

    bool allDigits = isdigit(local[0]);
    int numE = 0;
    int numP = 0;

    for (int index = 0; index < local.length(); index++)
    {
        if (local[index] == '.')
            numP++;
        else if (local[index] == 'E' || local[index] == 'e')
            numE++;
        else
            allDigits = allDigits && isdigit(local[index]);
        if (numP > 1 || numE > 1) break;
    }
    if (allDigits && numP <= 1 && numE <= 1)
    {
        double holder = atof(local.data());
        if (sign == '-')
            return -holder;
        else
            return holder;
    }
    else
    {
        success = false;          //indicate poorly formed string
        return -1;
    }
}
```

10.

```
bool isAPalindrome(const string & str)
/*
Determine if string str is a palindrome; that is, it does not
change when the order of the characters in the string is reversed.
*/
{
    int strLength = str.length(),
        limit = strLength / 2;

    for (int i = 0; i < limit ; i++)
        if (str[i] != str[strLength-i-1])
            return false;

    return true;
}
```

11.

```
/*
Determine if two strings are anagrams; that is, one string str1
is a permutation of the characters of the other string str2.
*/
bool areAnagrams(string str1, string str2)
{
    string temp = str2;
    int pos;

    for (int i = 0; i < str1.length(); i++)
    {
        pos = temp.find(str1.substr(i,1));

        if (pos == string::npos)
            return false;

        temp.erase(pos, 1);
    }

    return true;
}
```

Chapter 6: Lists

Exercises 6.3

1-6.

```
//---- Polynomial.h -----

#include <iostream>

#ifndef POLYNOMIAL
#define POLYNOMIAL

const int MAX_DEGREE = 100;
typedef double CoefType;
class Polynomial
{
public:
    void read(istream & in);
    void display(ostream & out) const;

    Polynomial operator+(const Polynomial & poly);
    Polynomial operator*(const Polynomial & poly);
    CoefType evaluate(double value);
private:
    int myDegree;
    CoefType myCoeff[MAX_DEGREE + 1];
};

istream & operator>>(istream & in, Polynomial & p);
ostream & operator<<(ostream & out, const Polynomial & p);

#endif

//---- Polynomial.cpp -----
#include <iostream>
#include <cassert>
using namespace std;
#include "Polynomial.h"

//-- Definition of >>
istream & operator>>(istream & in, Polynomial & p)
{
    p.read(in);
    return in;
}

//-- Definition of read()
void Polynomial::read(istream & in)
{
    cout << "Enter the degree (<= " << MAX_DEGREE << "): ";
    in >> myDegree; // the degree of this polynomial
    assert(myDegree <= MAX_DEGREE);
```

```
CoefType co;

// the coefficients in ascending order
cout << "Enter coefficients in ascending order:\n";
for (int index = 0; index <= myDegree; index++)
{
    in >> co;
    myCoeff[index] = co;
}

//-- Definition of <<
ostream & operator<<(ostream & out, const Polynomial & p)
{
    p.display(out);
    return out;
}

//-- Definition of display()
void Polynomial::display(ostream & out) const
{
    for (int index = 0; index < myDegree; index++)
        out << myCoeff[index] << "x^" << index << " + ";

    out << myCoeff[myDegree] << "x^" << myDegree << endl;
}

//-- Definition of evaluate()
CoefType Polynomial::evaluate(CoefType value)
{
    CoefType power = 1,
            result = 0;
    for (int index = 0; index <= myDegree; index++)
    {
        result += myCoeff[index] * power;
        power *= value;
    }
    return result;
}

//-- Definition of +
Polynomial Polynomial::operator+(const Polynomial & b)
{
    Polynomial c;
    if (myDegree < b.myDegree)
    {
        c = b;
        for (int i = 0; i <= myDegree; i++)
            c.myCoeff[i] += myCoeff[i];
    }

    c.myDegree = b.myDegree;
}
else
{
    c = *this;
```

```

        for (int i = 0; i <= b.myDegree; i++)
            c.myCoeff[i] += b.myCoeff[i];

        while (c.myCoeff[c.myDegree] == 0)
            c.myDegree--;
    }
    return c;
}

//-- Definition of *
Polynomial Polynomial::operator*(const Polynomial & b)
{
    Polynomial c;
    c.myDegree = myDegree + b.myDegree;
    for (int i = 0; i <= c.myDegree; i++)
        c.myCoeff[i] = 0;

    for (int i = 0; i <= myDegree; i++)
        for (int j = 0; j <= b.myDegree; j++)
            c.myCoeff[i + j] += myCoeff[i] * b.myCoeff[j];

    return c;
}

```

Exercises 6.4

NOTE: In the following algorithms, `ptr->data` and `ptr->next` refer to the data part and the next part, respectively, of the node pointed to by `ptr`.

- Algorithm to count the nodes in a linked list

```

count = 0;
ptr = first;

while (ptr != null_value)
{
    count++;
    ptr = ptr->next;
}

```

- Algorithm to compute average value in a linked list of real numbers

```

total = 0.0;
count = 0;
ptr = first;

while (ptr != null_value)
{
    count++;
    total += ptr->data;
    ptr = ptr->next;
}

```

```
if (count == 0)
    average = 0.0;
else
    average = total / count;
```

3. Algorithm to append a node at the end of a linked list

```
Get a node pointed to by newPtr;
newPtr->data = item;

if (first == null_value)
{
    first = newPtr;
    newPtr->next = null_value
}
else
{
    ptr = first;
    while (ptr->next != null_value)
        ptr = ptr->next;
    newPtr->next = ptr;
}
```

4. Algorithm to determine if the nodes in a linked list are in non-decreasing order.

```
ptr = first;

if (ptr == null_value ||           //note short-circuit eval
    newPtr->next == null_value)
    return true;
else
{
    nextPtr = ptr->next;
    for(;;)
    {
        if (nextPtr == null_value)
            return true;
        if (ptr->data > nextPtr->data)
            return false;
        // else
        ptr = nextPtr;
        nextPtr = nextPtr->next;
    }
}
```

5. All algorithms in exercises #1 - #4 require a scan of the entire linked list. Hence, they are O(n).

6. Algorithm to search a linked list for a given item.

```
ptr = first;
prevPtr = null_value

for (;;)
{
    if (ptr == null_value)
        return null_value;
    if (ptr->data == item)
        return prevPtr;
    //else
    prevPtr = ptr;
    ptr = ptr->next;
}
```

7. Algorithm to insert a node pointed to by newPtr after the n^{th} node.

```
ptr = first;
num = 1;
Get a node pointed to by newPtr;

if (n==0)
{
    newPtr->next = ptr;
    first = newPtr;
}
else
{
    while (ptr != null_value && num < n)
    {
        num++;
        ptr = ptr->next;
    }

    if (ptr!= null_value)
    {
        newPtr->next = ptr->next;
        ptr = newPtr;
    }
    else
        Signal error: fewer than n elements in list
}
```

8. Algorithm to delete the n^{th} node in a linked list.

```
ptr = first;
num = 1;

if (n==1)
{
    tempPtr = ptr;
    first = ptr->next;
    delete tempPtr;
}
```

```

else
{
    while (ptr != null_value && num < n - 1)
    {
        num++;
        ptr = ptr->next;
    }
    if (ptr != null_value &&           //note short-circuit eval
        ptr->next != null_value)
    {
        tempPtr = ptr->next;
        ptr->next = tempPtr->next;
        Delete node pointed to by tempPtr;
    }
    else
        Signal error: fewer than n elements in list
}

```

9. Algorithm to perform a shuffle-merge of two linked lists

```

Get a node pointed to by mergePtr;      // temporary head node

last = mergePtr;
ptr1 = first1;          // runs through first list
ptr2 = first2;          // runs through second list

while (ptr1 != null_value && ptr2 != null_value)
{
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr1->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr1 = ptr1->next;
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr2->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr2 = ptr2->next;
}

while (ptr1 != null_value)
{
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr1->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr1 = ptr1->next;
}

```

```

while (ptr2 != null_value)
{
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr2->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr2 = ptr2->next;
}

// Delete temporary headnode
tempPtr = mergePtr;
mergePtr = mergePtr->next;
Delete tempPtr;

```

10. Algorithm to perform a shuffle-merge without copying

```

if (first1 == null_value)
    mergePtr = first2;
else
{
    mergePtr = first1;
    ptr1 = first1;           // runs through first list
    ptr2 = first2;           // runs through second list

    while (ptr1 != null_value && ptr2 != null_value)
    {
        tempPtr = ptr1;
        ptr1 = tempPtr->next;
        tempPtr->next = ptr2;
        if (ptr2 != null_value)
            tempPtr = ptr2;
        ptr2 = tempPtr->next;
        tempPtr->next = ptr1;
    }

    if (ptr1 == null_value)
        tempPtr->next = ptr2;
}

```

11. Algorithm to merge two non-decreasing lists

```

Get a node pointed to by mergePtr;      // temporary head node
last = mergePtr;
ptr1 = first1;
ptr2 = first2;
while (ptr1 != null_value && ptr2 != null_value)
{
    Get a node pointed to by tempPtr;

```

```
if (ptr1->data < ptr2->data);
{
    tempPtr->data = ptr1->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr1 = ptr1->next;
}
else
{
    tempPtr->data = ptr2->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr2 = ptr2->next;
}
}

while (ptr1 != null_value)
{
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr1->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr1 = ptr1->next;
}

while (ptr2 != null_value)
{
    Get a node pointed to by tempPtr;
    tempPtr->data = ptr2->data;
    last->next = tempPtr;
    last = tempPtr;
    ptr2 = ptr2->next;
}

tempPtr = mergePtr;
mergePtr = mergePtr->next;

// Delete temporary headnode
Delete node pointed to by tempPtr;
```

12. Algorithm to reverse a linked list without copying.

```
currentPtr = first;
prevPtr = null_value;

while (currentPtr != null_value)
{
    nextPtr = currentPtr->next;
    currentPtr->next = prevPtr;
    prevPtr = currentPtr;
    currentPtr = nextPtr;
}
first = prevPtr;
```

Exercises 6.5

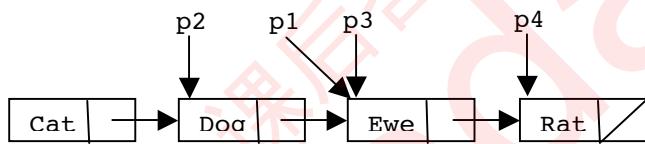
1. 123 456
2. 34 34
3. 34 34
4. Error because the `next` field of `p2` is a null pointer

5. 12 34
34
34
34

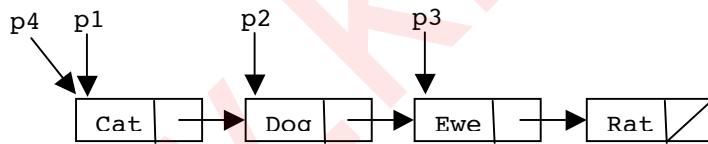
6. 111 222
222
111

7. 34 34
34

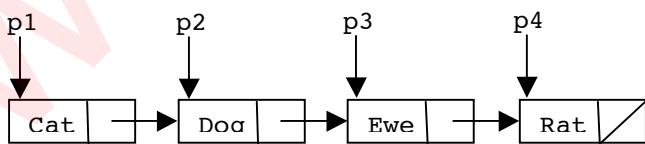
8.



9.

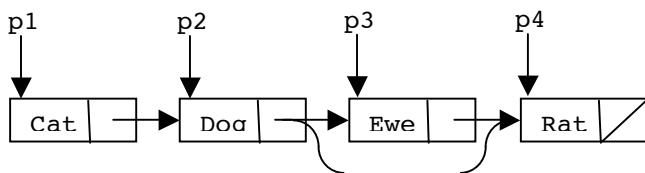


10.

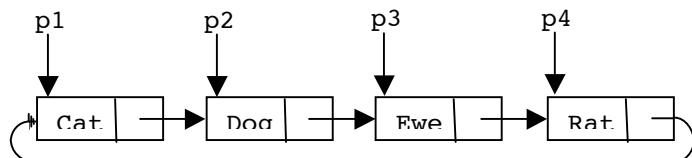


11. `p4->next` is a null pointer, so an error occurs when we attempt to access it's data part.

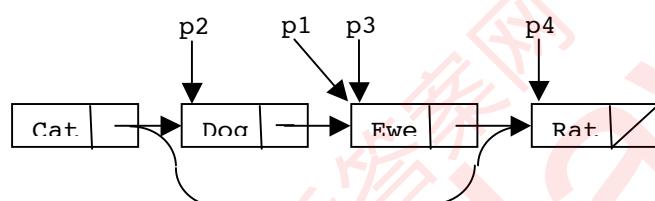
12.



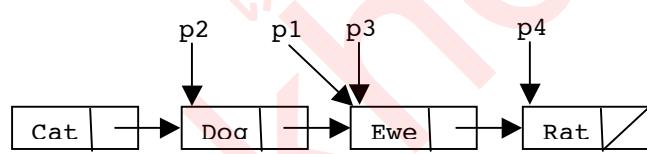
13.



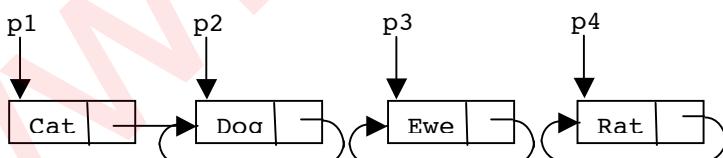
14.



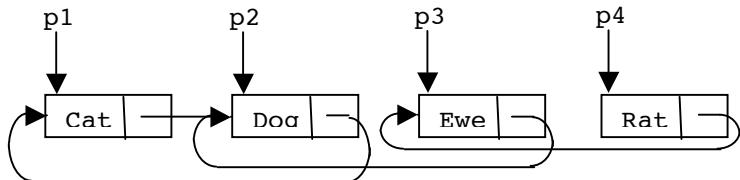
15.



16.



17.



18-21.

```
 //---- List.h ----

#ifndef LINKEDLIST
#define LINKEDLIST

#include <iostream>
using namespace std;

typedef int ElementType;
class List
{
private:
    class Node
    {
    public:
        //----- Node DATA MEMBERS
        ElementType data;
        Node * next;

        //----- Node OPERATIONS
        //--- Default constructor: initializes next member to
        Node()
        : next(0)
        { }

        //--- Explicit-value constructor: initializes data member to dataValue
        //---
        Node(ElementType dataValue)
        : data(dataValue), next(0)
        { }

    }; //--- end of Node class

    typedef Node * NodePointer;

public:
    //----- List OPERATIONS

    List();
    /*-----
       Default constructor: builds an empty List object.

       Precondition: None
       Postcondition: first is 0 and mySize is 0.
    -----*/
    List(const List & origList);
    /*-----
       Copy constructor

       Precondition: A copy of origList is needed.
       Postcondition: A distinct copy of origList has been constructed.
    -----*/
}
```

```
~List();
/*-----
   Destructor

   Precondition: This list's lifetime is over.
   Postcondition: This list has been destroyed.
-----*/



const List & operator=(const List & rightSide);
/*-----
   Assignment operator

   Precondition: This list must be assigned a value.
   Postcondition: A copy of rightSide has been assigned to this list.
-----*/



bool List::empty();
/*-----
   Check if this list is empty

   Precondition: None.
   Postcondition: true is returned if this list is empty, false if not.
-----*/



void insert(ElementType dataVal, int index);
/*-----
   Insert a value into a list at a given index.

   Precondition: index is a valid list index: 0 <= index <= mySize,
   Postcondition: dataVal has been inserted into the list at position
      index, provided index is valid..
-----*/



void erase(int index);
/*-----
   Remove a value from a list at a given index.

   Precondition: index is a valid list index: 0 <= index < mySize
   Postcondition: dataVal at list position index has been removed,
      provided index is valid.
-----*/



int search(ElementType dataVal);
/*-----
   Search for an data value in this list.

   Precondition: None
   Postcondition: Index of node containing dataVal is returned
      if such a node is found, -1 if not.
-----*/
```

```
void display(ostream & out) const;
/*-----
   Display the contents of this list.

   Precondition: ostream out is open
   Postcondition: Elements of this list have been output to out.
-----*/
int nodeCount();
/*-----
   Count the elements of this list.

   Precondition: None
   Postcondition: Number of elements in this list is returned.
-----*/
void reverse();
/*-----
   Reverse this list.

   Precondition: None
   Postcondition: Elements in this list have been reversed.
-----*/
bool ascendingOrder();
/*-----
   Check if the elements of this list are in ascending order.

   Precondition: None
   Postcondition: true is returned if the list elements are in
                  ascending order, false if not.
-----*/
private:
    //---- DATA MEMBERS
    NodePointer first;
    int mySize;
}; //--- end of List class

ostream & operator<<(ostream & out, const List & aList);
istream & operator>>(istream & in, List & aList);

#endif

//---- List.cpp -----
#include <iostream>
using namespace std;

#include "List.h"

//-- Definition of the class constructor
List::List()
: first(0), mySize(0)
{ }
```

```
//-- Definition of the copy constructor
List::List(const List & origList)
{
    mySize = origList.mySize;
    first = 0;
    if (mySize == 0) return;

    List::NodePointer origPtr, lastPtr;
    first = new Node(origList.first->data); // copy first node
    lastPtr = first;
    origPtr = origList.first->next;
    while (origPtr != 0)
    {
        lastPtr->next = new Node(origPtr->data);
        origPtr = origPtr->next;
        lastPtr = lastPtr->next;
    }
}

//-- Definition of the destructor
inline List::~List()
{
    List::NodePointer prev = first,
                    ptr;
    while (prev != 0)
    {
        ptr = prev->next;
        delete prev;
        prev = ptr;
    }
}

// Definition of empty()
bool List::empty()
{
    return mySize == 0;
}

//-- Definition of the assignment operator
const List & List::operator=(const List & rightSide)
{
    mySize = rightSide.mySize;
    first = 0;
    if (mySize == 0) return *this;

    if (this != &rightSide)
    {
        this->~List();
        List::NodePointer origPtr, lastPtr;
        first = new Node(rightSide.first->data); // copy first node
        lastPtr = first;
        origPtr = rightSide.first->next;
```

```
        while (origPtr != 0)
        {
            lastPtr->next = new Node(origPtr->data);
            origPtr = origPtr->next;
            lastPtr = lastPtr->next;
        }
    return *this;
}

//-- Definition of insert()
void List::insert(ElementType dataVal, int index)
{
    if (index < 0 || index > mySize)
    {
        cerr << "Illegal location to insert -- " << index << endl;
        return;
    }

    mySize++;
    List::NodePointer newPtr = new Node(dataVal),
                    predPtr = first;
    if (index == 0)
    {
        newPtr->next = first;
        first = newPtr;
    }
    else
    {
        for(int i = 1; i < index; i++)
            predPtr = predPtr->next;
        newPtr->next = predPtr->next;
        predPtr->next = newPtr;
    }
}

//-- Definition of erase()
void List::erase(int index)
{
    if (index < 0 || index >= mySize)
    {
        cerr << "Illegal location to delete -- " << index << endl;
        return;
    }

    mySize--;
    List::NodePointer ptr,
                    predPtr = first;
    if (index == 0)
    {
        ptr = first;
        first = ptr->next;
        delete ptr;
    }
}
```

```
else
{
    for(int i = 1; i < index; i++)
        predPtr = predPtr->next;
    ptr = predPtr->next;
    predPtr->next = ptr->next;
    delete ptr;
}
}

//-- Definition of search()
int List::search(ElementType dataVal)
{
    int loc;
    List::NodePointer tempP = first;
    for (loc = 0; loc < mySize; loc++)
        if (tempP->data == dataVal)
            return loc;
        else
            tempP = tempP->next;

    return -1;
}

//-- Definition of display()
void List::display(ostream & out) const
{
    List::NodePointer ptr = first;
    while (ptr != 0)
    {
        out << ptr->data << " ";
        ptr = ptr->next;
    }
}

//-- Definition of the output operator
ostream & operator<<(ostream & out, const List & aList)
{
    aList.display(out);
    return out;
}

//-- Definition of nodeCount()
int List::nodeCount()
{
    int count = 0;
    List::NodePointer ptr = first;

    while (ptr != 0)
    {
        count++;
        ptr = ptr->next;
    }
    return count;
}
```

```

//-- Definition of reverse()
void List::reverse()
{
    NodePointer prevP = 0,
                currentP = first,
                nextP;

    while (currentP != 0)
    {
        nextP = currentP->next;
        currentP->next = prevP;
        prevP = currentP;
        currentP = nextP;
    }
    first = prevP;           // new head of (reversed) linked list
}

//-- Definition of ascendingOrder()
bool List::ascendingOrder()
{
    if (mySize <= 1)
        //empty or one element list
        return true;
    //else
    NodePointer prevP = first,
                tempP = first->next;
    while (tempP != 0 && prevP->data <= tempP->data)
    {
        prevP = tempP;
        tempP = tempP->next;
    }
    if (tempP != 0)
        return false;
    // else
        return true;
}

```

Exercises 6.6

1. a. List : B, C, J, P b. Storage pool: M, Z, K, Q, ?, ?
2. first = 4; free = 1

node	data	next
0	J	3
1	Z	6
2	C	5
3	P	-1
4	B	2
5	F	0
6	K	7
7	Q	8
8	?	9
9	?	-1

3. `first = 4; free = 0`

node	data	next
0	J	5
1	Z	6
2	C	3
3	P	-1
4	B	2
5	M	1
6	K	7
7	Q	8
8	?	9
9	?	-1

4. `first = -1; free = 4`

node	data	next
0	J	5
1	Z	6
2	C	3
3	P	0
4	B	2
5	M	1
6	K	7
7	Q	8
8	?	9
9	?	-1

5. `first = 5, free = 2`

node	data	next
0	J	3
1	Z	7
2	C	1
3	K	-1
4	B	0
5	A	4
6	K	8
7	Q	9
8	?	10
9	?	0

NOTE: The following are definitions of function members in a List class whose data member is a pointer first to the first node in the linked list.

```
6. /* Find number of nodes in linked list */
int List::nodeCount()
{
    int count = 0;
    int ptr = first;

    while (ptr != NULL_VALUE)
    {
        count++;
        ptr = node[ptr].next;
    }
    return count;
}

7. /* Check if list in ascending order */
bool List::ascendingOrder()
{
    if (numNodes() <= 1)
        //empty or one element list
        return true;
    //else
    int prevP = first,
        tempP = node[first].next;

    while (tempP != NULL_VALUE && node[prevP].data <= node[tempP].data)
    {
        prevP = tempP;
        tempP = node[tempP].next;
    }

    if (tempP != NULL_VALUE)
        return false;
    // else
        return true;
}

8. /* Find last node */
int List::lastNode()
{
    int currentPtr = first,
        lastPtr = first;

    while (currentPtr != NULL_VALUE)
    {
        lastPtr = currentPtr;
        currentPtr = node[currentPtr].next;
    }
    return lastPtr;
}
```

```
9. /* Reverse the linked list */
void List::reverse()
{
    int prev = NULL_VALUE,
        currentP = first,
        nextP;

    while (currentP != NULL_VALUE)
    {
        nextP = node[currentP].next;
        node[currentP].next = prev;
        prev = currentP;
        currentP = nextP;
    }
    first = prev;
}
```

Chapter 7: Stacks

Exercises 7.2

1. `myTop == 0; myArray` contains 3 elements: 10, 22, 37, ?, ? but note that only element 10 is considered to be in the stack.
2. `myTop == -1; myArray` contains 3 elements: 10, 9, 8, ?, ? but note that `myTop == -1` indicates that the stack is empty.
3. `myTop == 4; myArray` contains 3 elements: 10, 20, 30, 40, 50, but an error occurs for `i = 6` because an attempt is made to push 6 elements onto a stack with capacity 5.
4. `myTop == -1; myArray` contains 1 element: 11, ?, ?, ?, ? but note that, as in #2, the stack is considered empty.
5.

```
bool Stack::full()
{
    return top == STACK_CAPACITY - 1;
}
```
6.

```
StackElement bottom();
/*
 *-----*
 * Retrieve the bottom element of this stack.
 * Precondition: None
 * Postcondition: Bottom element of the stack is returned, unless there
 * was none, in which case a stack-empty message is displayed.
 *-----*
```



```
//Definition of bottom()
StackElement Stack::bottom()
{
    if (myTop >= 0)
        return myArray[0];
    // else
    cerr << "Stack empty: no bottom element -- returning garbage value\n";
    StackElement garbage;
    return garbage;
}
```
7.

```
StackElement bottom(Stack s);
/*
 *-----*
 * Retrieve the bottom element of a stack received as a value parameter.
 * Precondition: None
 * Postcondition: Bottom element of the stack is returned, unless there
 * was none, in which case a stack-empty message is displayed.
 *-----*
```

```
//Definition of bottom():
StackElement bottom(Stack s) // destructive!! Destroys copy of stack.
{
    if (!s.empty())
    [
        StackElement value;

        while (!s.empty())
        {
            value = s.top();
            s.pop();
        }
        return value;           // s now empty
    }
    // else
    cerr << "Stack empty: no bottom element -- returning garbage value\n";
    StackElement garbage;
    return garbage;
}
```

8.

```
StackElement nthElement(int n);
/*
-----*
   Retrieve the n-th element of this stack.
   Precondition: 1 <= n <= number of stack elements
   Postcondition: n-th element of the stack is returned, unless stack
                  has fewer than n elements, in which case an error message is
                  displayed.
-----*/
//Definition of nthElement()
StackElement Stack::nthElement(int n)
{
    StackElement elem;
    int counter = 1;
    while (counter <= n && myTop != -1)
    {
        counter++;
        elem = top();
        pop();
    }

    if (counter == n)
        return elem;
    // else
    cerr << "Stack has no " << n << "-th element"
          " -- returning a garbage value\n";
    StackElement garbage;
    return garbage;
}
```

9.

```
StackElement nthElement(int n);
/*
 *-----*
 * Retrieve the n-th element of this stack.
 * Precondition: 1 <= n <= number of stack elements
 * Postcondition: n-th element of the stack is returned, unless stack
 * has fewer than n elements, in which case an error message is
 * displayed.
 *-----*
```

```
//Definition of nthElement()
StackElement Stack::nthElement(int n)
{
    if (n <= myTop + 1)
        return myArray[myTop + 1 - n];
    //else
    cerr << "Stack has no " << n << "-th element"
        " -- returning a garbage value\n";
    StackElement garbage;
    return garbage;
}
```

10. a) $n = 3$ Possible permutations:

123, 132, 213, 231, 321

Impossible permutations:

312

b) $n = 4$ Possible permutations:

1234, 1243, 1324, 1342, 1432, 2134, 2143, 2314, 2341, 2431, 3214, 3241, 3421, 4321

Impossible permutations (10 out of 24):

1423, 2413, 3124, 3142, 3412, 4123, 4132, 4231, 4213, 4312

c) $n = 5$ Possible permutations:12345, 12354, 12543, 12435, 12453,
13245, 13254, 13452, 13425, 13542,
14325, 14352, 14532, 15432,21345, 21354, 21435, 21453, 21543,
23145, 23154, 23451, 23415, 23541
24315, 24351, 24531, 2543132154, 32145, 32451, 32415, 32541
34215, 34251, 34521, 35421

43215, 43251, 43521, 45321

54321

Impossible permutations (78 out of 120): those remaining

- d) Possible permutations are those in which for any digit d in the permutation, the digits to the right of d that are less than d are in reverse order.

For example, 21354 is possible

21534 is not possible because 3 and 4 are to the right of 5 but are not in reverse order.

It can also be shown that the number of permutations of n cars is

$$C_n = \binom{2n}{n} / (n + 1);$$

or,

$$(2n)! / (n! n! (n + 1))$$

11. Proposed storage method assumes that the two stacks will always be of the same relative fullness. This need not be the case; one stack could become full while the other remains empty.

12.

```
----- DoubleStack.h -----
/*
 *----- A two-stack class:
 *----- One stack operates in usual manner:
 *----- push increments top[0], pop decrements top[0]
 *----- Second stack starts at high end of array:
 *----- push decrements top[1], pop increments top[1],
 *----- Assumption: Error checking is done outside class,
 *----- */

#ifndef DOUBLE_STACK
#define DOUBLE_STACK

#include <iostream>

const int STACK_LIMIT = 20;
typedef int ElementType;
class DoubleStack
{
public:
    //--- Constructor -- builds an empty double stack
    DoubleStack();
    /*----- Constructor
    Postcondition: an empty double stack has been constructed.
    -----*/
    bool empty(int stackNum);
    /*----- Empty stack check
    Precondition: stackNum is 1 or 2
    Postcondition: Returns true if stack stackNum is empty,
    else false.
    -----*/
}
```

```
bool full();
/*-----
   Array-full check
   Postcondition: Returns true if array that stores the two
   stacks is full, else false.
-----*/
ElementType top(int stackNum );
/*-----
   Retrieve top element
   Precondition: stackNum is 1 or 2
   Postcondition: Returns top element of stack stackNum if there
   is one; else a stack-empty message is displayed.
-----*/
void pop(int stackNum );
/*-----
   Remove top element
   Precondition: stackNum is 1 or 2
   Postcondition: Top element of stack stackNum has been removed
   if there was one; else a stack-empty message is displayed.
-----*/
void push(int stackNum , ElementType value);
/*-----
   Add value to a stack
   Precondition: stackNum is 1 or 2
   Postcondition: value has been added to stack stackNum if there
   is room in the array; else execution is terminated.
-----*/
void display(ostream & out);
/*-----
   Output a double Stack
   Precondition: ostream out is open
   Postcondition: Contents of the two stacks have been displayed
   to out.
-----*/
private:
    int          myTop[2];
    ElementType  data[STACK_LIMIT];
};

#endif

//---- DoubleStack.cpp ----

#include <iostream>
using namespace std;
#include "DoubleStack.h"

//-- Definition of constructor
```

```
DoubleStack::DoubleStack()
{
    myTop[0] = -1;
    myTop[1] = STACK_LIMIT;
}

//-- Definition of empty()
bool DoubleStack::empty (int stackNum)
{
    if (stackNum == 1)
        return myTop[0] == -1;
    else
        return myTop[1] == STACK_LIMIT;
}

//-- Definition of full()
bool DoubleStack::full ()
{
    return myTop[0] + 1 == myTop[1];
}

//-- Definition of top()
ElementType DoubleStack::top (int stackNum)
{
    if (stackNum == 1)
        return data[myTop[0]];
    else
        return data[myTop[1]];
}

//-- Definition of pop()
void DoubleStack::pop (int stackNum)
{
    if (stackNum == 1)
        myTop[0]--;
    else
        myTop[1]++;
}

//-- Definition of push()
void DoubleStack::push (int stackNum, ElementType value)
{
    if (full())
    {
        cerr << "No room left for stack elements\n";
        exit(1);
    }
    if (stackNum == 1)
    {
        myTop[0]++;
        data[myTop[0]] = value;
    }
    else
    {
        myTop[1]--;
        data[myTop[1]] = value;
    }
}
```

```
//-- Definition of display()
void DoubleStack::display(ostream & out)
{
    cout << "Stack 1:\n";
    for (int i = 0; i <= myTop[0]; i++)
        cout << data[i] << ' ';
    cout << " <-- Top\n";

    cout << "Stack 2:\n";
    cout << "Top --> ";
    for (int i = myTop[1]; i < STACK_LIMIT; i++)
        cout << data[i] << ' ';
    cout << endl;
}

/*
-----*
           Driver program to test the DoubleStack class.
-----*/
#include <iostream>
using namespace std;

#include "DoubleStack.h"

int main()
{
    DoubleStack s;
    cout << "DoubleStack created.\n" << boolalpha
        << "Stack #1 empty: " << s.empty(1) << endl
        << "Stack #2 empty: " << s.empty(2) << endl;

    cout << "How many elements to add to the stack? ";
    int numItems;
    cin >> numItems;
    for (int i = 1; i <= numItems; i++)
    {
        cout << "Which stack (1 or 2)? ";
        int stNum;
        cin >> stNum;
        s.push(stNum, i);
    }
    cout << "Contents of double stack:\n";
    s.display(cout);

    cout << "Top value in"
        "\nStack 1: " << s.top(1) <<
        "\nStack 2: " << s.top(2) << endl;

    while (!s.empty(1) && !s.empty(2))
    {
        cout << "Popping Stack #1: " << s.top(1) << endl;
        s.pop(1);
        cout << "Popping Stack #2: " << s.top(2) << endl;
        s.pop(2);
    }
}
```

```
cout << "Contents of double stack:\n";
s.display(cout);

cout << "\nStack #1 empty? " << s.empty(1) << endl;
cout << "Stack #2 empty? " << s.empty(2) << endl;
}
```

13.

```
//---- MultiStack.h ----

#ifndef MULTI_STACK
#define MULTI_STACK

#include <iostream>

const int N = 3;           // number of stacks
const int STACK_LIMIT = 5; // initial stack capacity
typedef int ElementType;

class MultiStack
{
public:
    //-- Constructor -- builds an empty multi-stack
    MultiStack();
    /*-----
        Constructor
        Postcondition: an empty multi-stack has been constructed.
    -----*/
    bool empty(int stackNum);
    /*-----
        Empty stack check
        Precondition: 1 <= stackNum <= N
        Postcondition: Returns true if stack stackNum is empty,
                       else false.
    -----*/
    bool full(int stackNum);
    /*-----
        Array-full check
        Postcondition: Returns true if part of array allocated to
                       stack stackNum is full, else false.
    -----*/
    ElementType top(int stackNum);
    /*-----
        Retrieve top element
        Precondition: 1 <= stackNum <= N
        Postcondition: Returns top element of stack stackNum if there
                       is one; else a stack-empty message is displayed.
    -----*/
}
```

```
void pop(int stackNum);
/*-----
   Remove top element
   Precondition: 1 <= stackNum <= N
   Postcondition: Top element of stack stackNum has been removed
      if there was one; else a stack-empty message is displayed.
-----*/
void push(int stackNum , ElementType value);
/*-----
   Add value to a stack
   Precondition: 1 <= stackNum <= N
   Postcondition: value has been added to stack stackNum if there
      is room in the array; else execution is terminated.
-----*/
void display(ostream & out);
/*-----
   Ouput a double Stack
   Precondition: ostream out is open
   Postcondition: Contents of the stacks have been displayed
      to out.
-----*/
private:
    int          myTop[N];
    int          myBottom[N];
    ElementType  data[N*STACK_LIMIT];

    bool move_elem(int stackNum); // Try to move elements of a stack
};

#endif

//---- MultiStack.cpp ----

#include <iostream>
using namespace std;
#include "MultiStack.h"

//-- Definition of constructor
MultiStack::MultiStack()
{
    for (int index = 0; index < N; index++)
    {
        myTop[index] = STACK_LIMIT * index - 1;
        myBottom[index] = myTop[index];
    }
    for (int i = 0; i < N*STACK_LIMIT; i++)
        data[i] = 0;
}

//-- Definition of empty()
bool MultiStack:: empty(int stackNum)
{
    return myTop[stackNum-1] == myBottom[stackNum-1];
}
```

```
/** Definition of full()
bool MultiStack:: full(int stackNum)
{
    if (stackNum< N)
        return myTop[stackNum-1] == myBottom[stackNum];
    else
        return myTop[stackNum-1] == N*STACK_LIMIT - 1;
}

/** Definition of top()
ElementType MultiStack:: top(int stackNum)
{
    if (empty(stackNum))
        cerr << "Stack is empty\n";
    return data[myTop[stackNum-1]];
}

/** Definition of pop()
void MultiStack:: pop(int stackNum)
{
    if (empty(stackNum))
        cerr << "Stack is empty\n";
    else
        myTop[stackNum-1]--;
}

void MultiStack:: push(int stackNum, ElementType item)
{
    if (full(stackNum))
        if(!move_elem(stackNum))
        {
            cerr << "No space left in any of the stacks." << endl;
            exit(1);
        }
    myTop[stackNum-1]++;
    data[myTop[stackNum-1]] = item;
}

void MultiStack::display(ostream & out)
{
    for (int i = 1; i <= N; i++)
    {
        out << "Stack #\n" << i << ":" ;
        out << "Bottom--> ";
        for (int j = myBottom[i-1] + 1; j <= myTop[i-1]; j++)
            out << data[j] << " ";
        out << "<-- Top\n";
    }
}
```

```
bool MultiStack::move_elem(int stackNum)
/*
----- Try to make room in stack stackNum for new value
Precondition: 1 <= stackNum <= N
Postcondition: Stack stackNum's capacity has been increased by
    1 and true returned, if possible; else false returned.
-----*/
{
    bool hole = false;
    int where = stackNum;

    while (where < N-1)// look above for space
    {
        hole = myTop[where] < myBottom[where+1];
        if (hole) break;
        where++;
    }
    if (!hole && myTop[N-1] < N*STACK_LIMIT - 1)
    {
        hole = true;
        where = N-1;
    }

    if (hole) // shift in stacks above
    {
        int upper1 = myTop[where] + 1;
        int lower1 = myBottom[stackNum] + 1;
        for (int index = upper1; index > lower1; index--)
            data[index] = data[index-1];

        while (where >= stackNum) // adjust pointers
        {
            myTop[where]++;
            myBottom[where]++;
            where--;
        }
    }
    else // look below for hole
    {
        where = stackNum-2;
        while(where >= 0)
        {
            hole = myTop[where] < myBottom[where+1];
            if (hole) break;
            where--;
        }
        if (hole)// shift in stacks below
        {
            int lower1 = myBottom[where+1];
            int upper1 = myTop[stackNum-1];
            for (int index = lower1; index < upper1; index++)
                data[index] = data[index+1];

            where++;
        }
    }
}
```

```
        while (where < stackNum) // adjust pointers
    {
        myTop[where]--;
        myBottom[where]--;
        where++;
    }
}
return hole;
}

/*
-----Driver program to test the MultiStack class.
-----*/
#include <iostream>
using namespace std;

#include "MultiStack.h"

int main()
{
    MultiStack s;
    cout << "MultiStack created.\n" << boolalpha;
    for (int i = 1; i <= N; i++)
        cout << "Stack #" << i << " empty: " << s.empty(i) << endl;

    cout << "How many elements to add to the stacks? ";
    int numItems;
    cin >> numItems;
    for (int i = 1; i <= numItems; i++)
    {
        cout << "Which stack (1, ... " << N << ")? ";
        int stNum;
        cin >> stNum;
        s.push(stNum, i);
    }
    cout << "Contents of multi-stack:\n";
    s.display(cout);

    cout << "Top value in\n";
    for (int i = 1; i <= N; i++)
        cout << "Stack #" << i << ": " << s.top(i) << endl;

    bool someStackEmpty = false;
    for (;;)
    {
        for (int i = 1; i <= N; i++)
            if (s.empty(i))
            {
                someStackEmpty = true;
                break;
            }
        if (someStackEmpty) break;
    }
}
```

```

for (int i = 1; i <= N; i++)
{
    cout << "Popping Stack #" << i << ":" << s.top(i) << endl;
    s.pop(i);
}

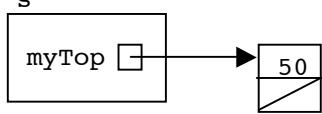
cout << "Contents of multi-stack:\n";
s.display(cout);

for (int i = 1; i <= N; i++)
    cout << "Stack #" << i << " empty: " << s.empty(i) << endl;
}

```

Exercises 7.3

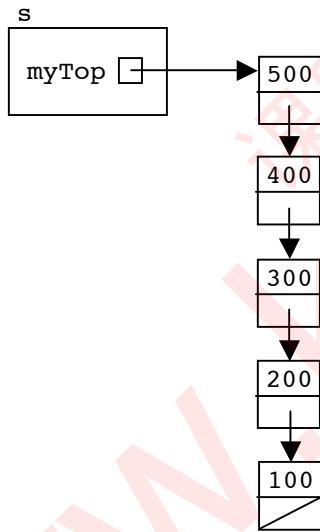
1.



2.



3.



4. Same as 2

5.

```

StackElement bottom();
/*
-----*
   Retrieve the bottom element of this stack.
   Precondition: None
   Postcondition: Bottom element of the stack is returned, unless there
                  was none, in which case a stack-empty message is displayed.
-----*/

```

```

//Definition of bottom()
StackElement Stack::bottom()
{

```

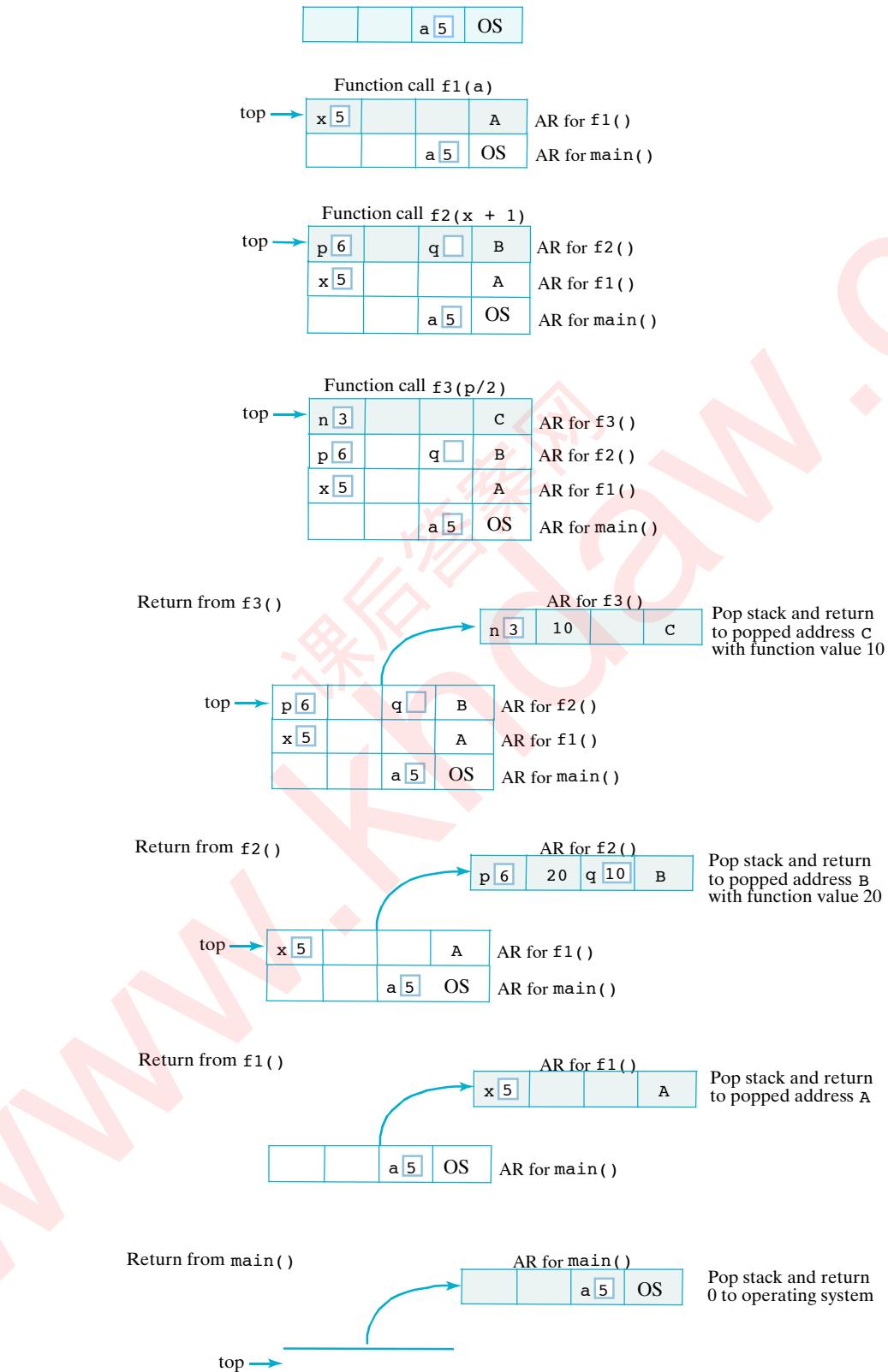
```
if (myTop != 0)
{
    Stack::NodePointer bot = myTop
    while (bot->next != 0)
        bot = bot->next;
    return *bot;
}
// else
cerr << "Stack empty: no bottom element -- returning garbage value\n";
StackElement garbage;
return garbage;
}

6.
StackElement nthElement(int n);
/*-----
   Retrieve the n-th element of this stack.
   Precondition: 1 <= n <= number of stack elements
   Postcondition: n-th element of the stack is returned, unless stack
      has fewer than n elements, in which case an error message is
      displayed.
-----*/
//Definition of nthElement()
StackElement Stack::nthElement(int n)
{
    Stack::NodePointer nptr = myTop;
    int counter = 1;
    while (counter <= n && nptr != 0)
    {
        counter++;
        nptr = nptr->next;
    }

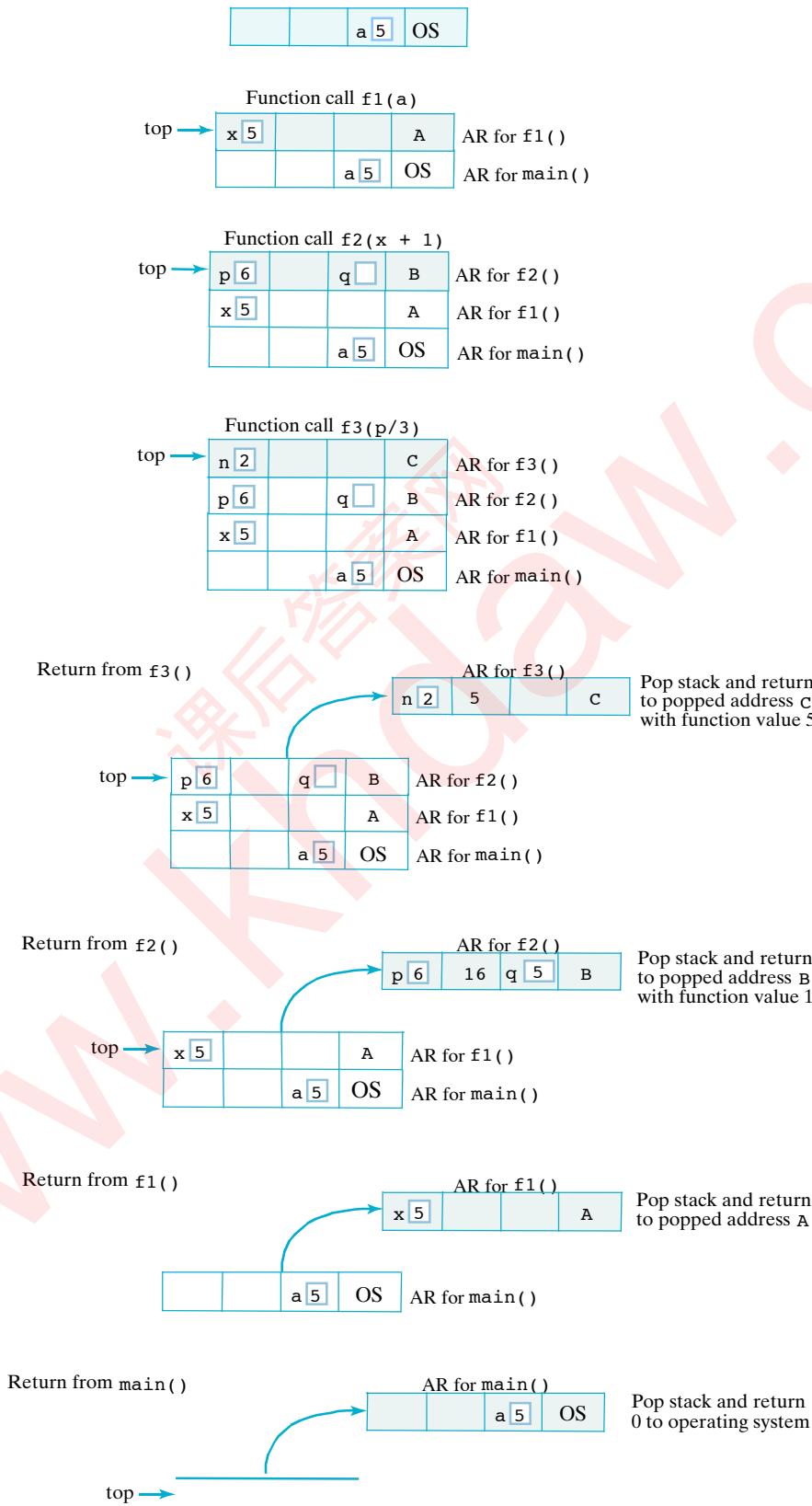
    if (counter == n)
        return *nptr;
    // else
    cerr << "Stack has no " << n << "-th element"
          " -- returning a garbage value\n";
    StackElement garbage;
    return garbage;
}
```

Exercises 7.4

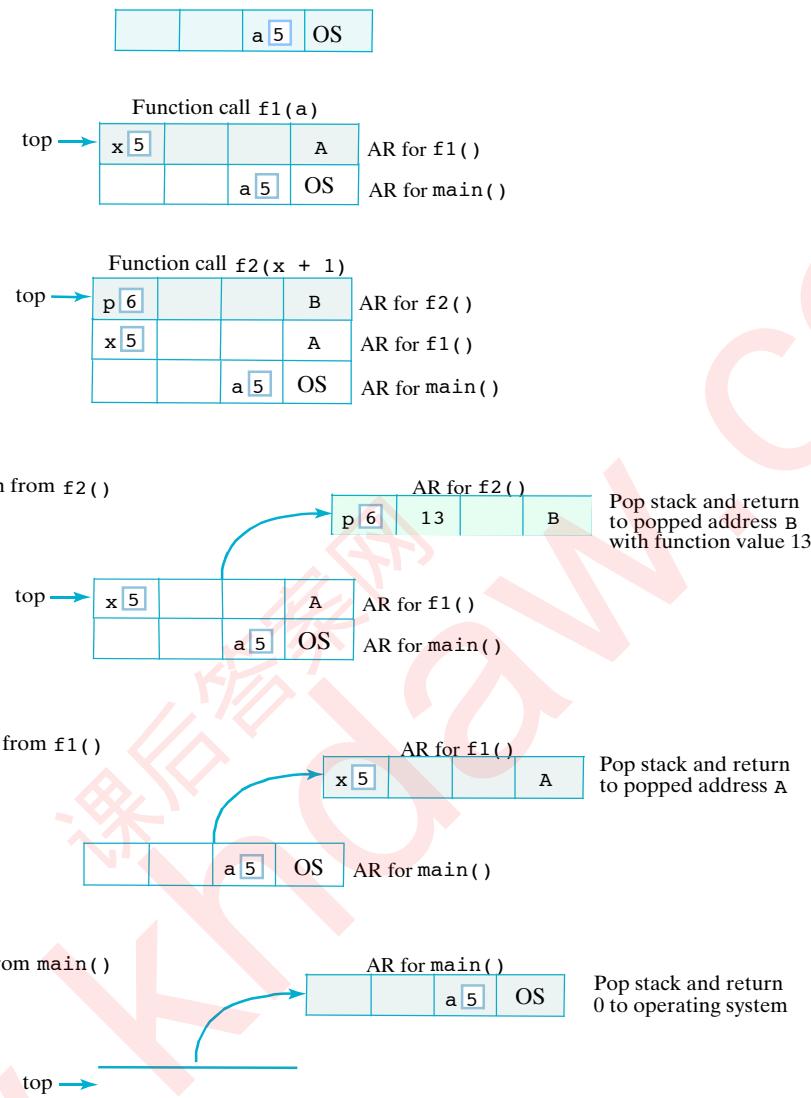
1.



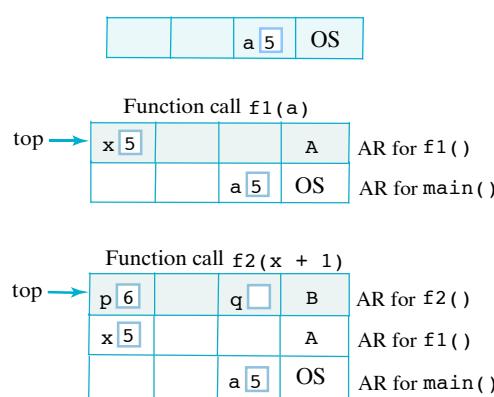
2.

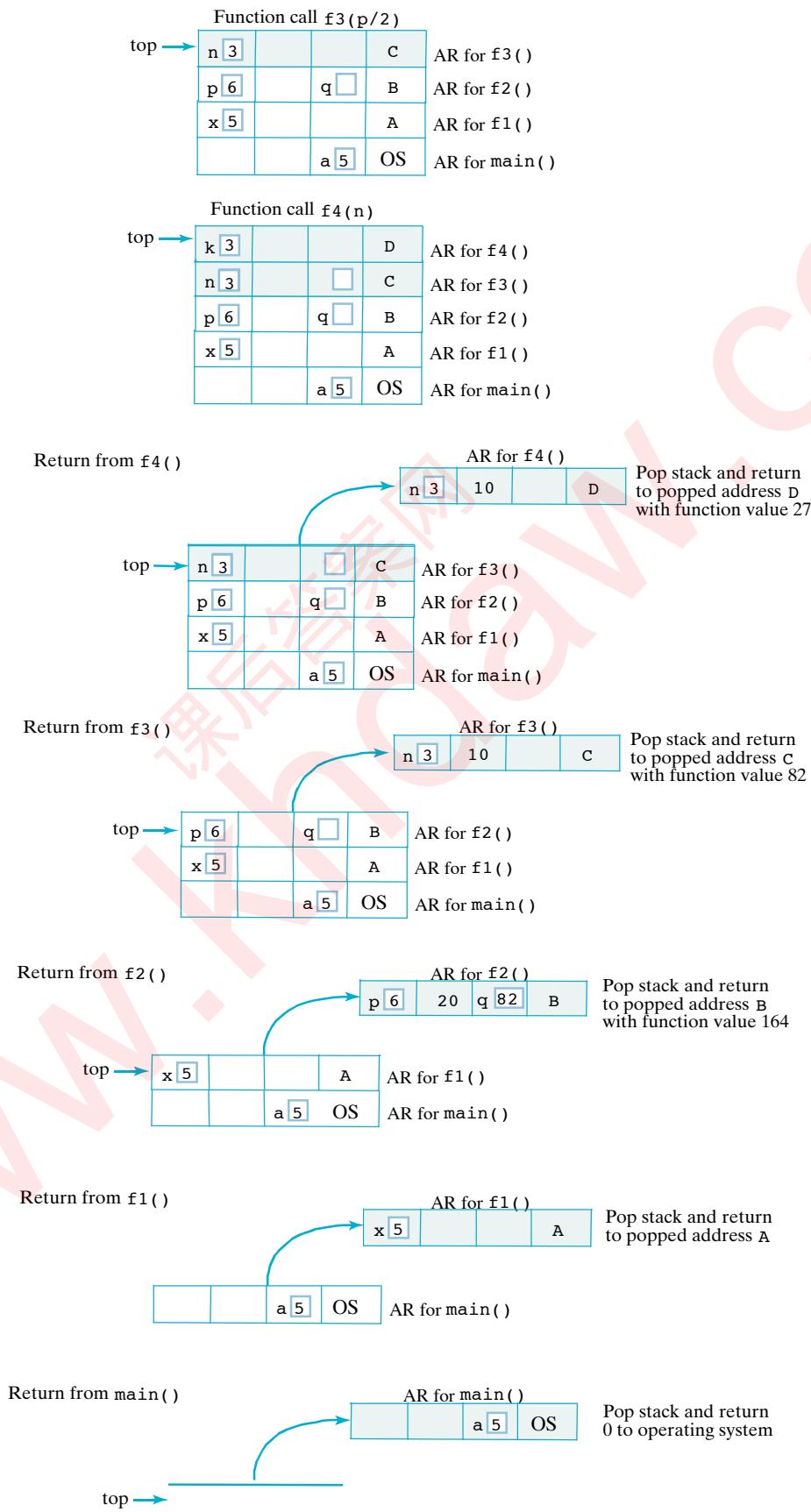


3.

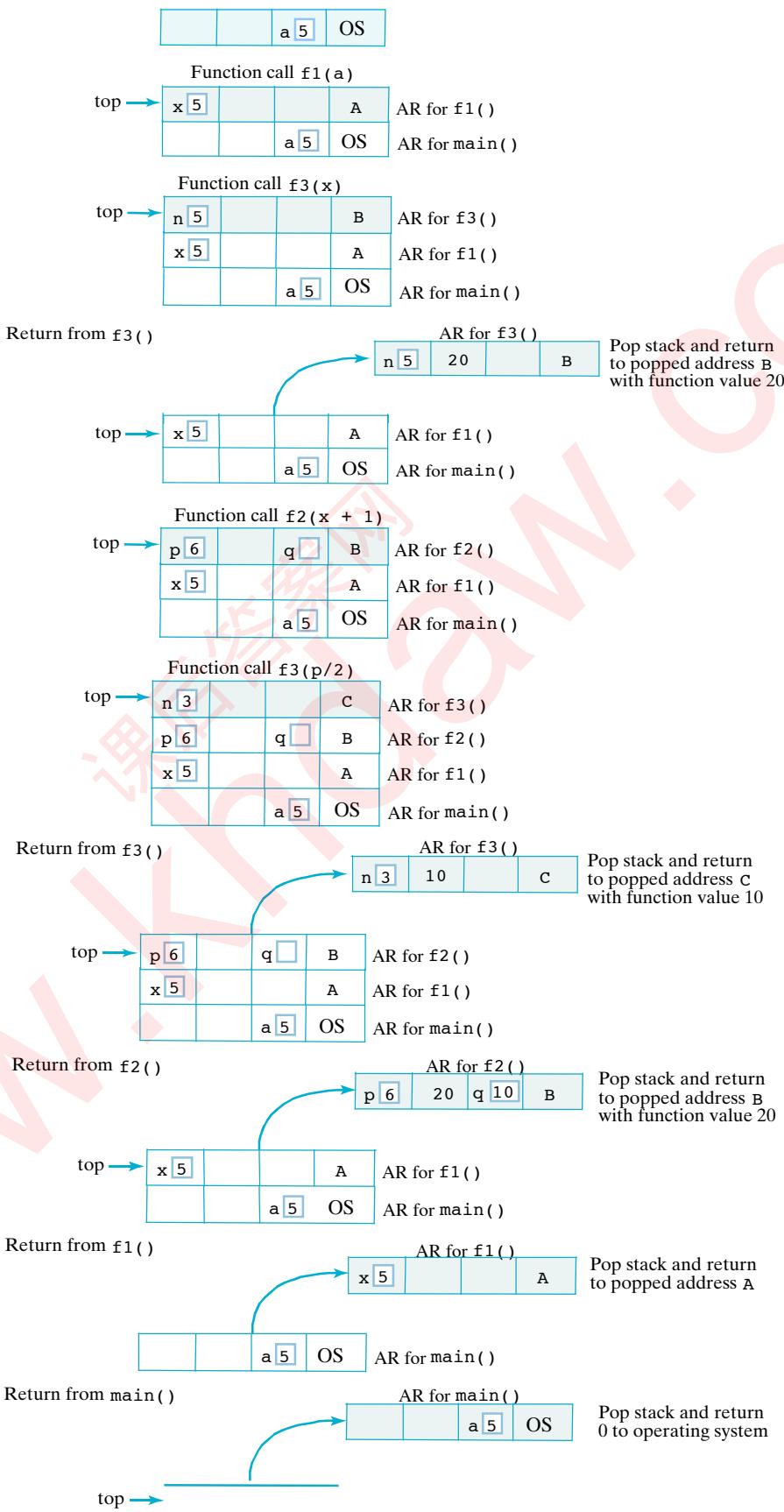


4.

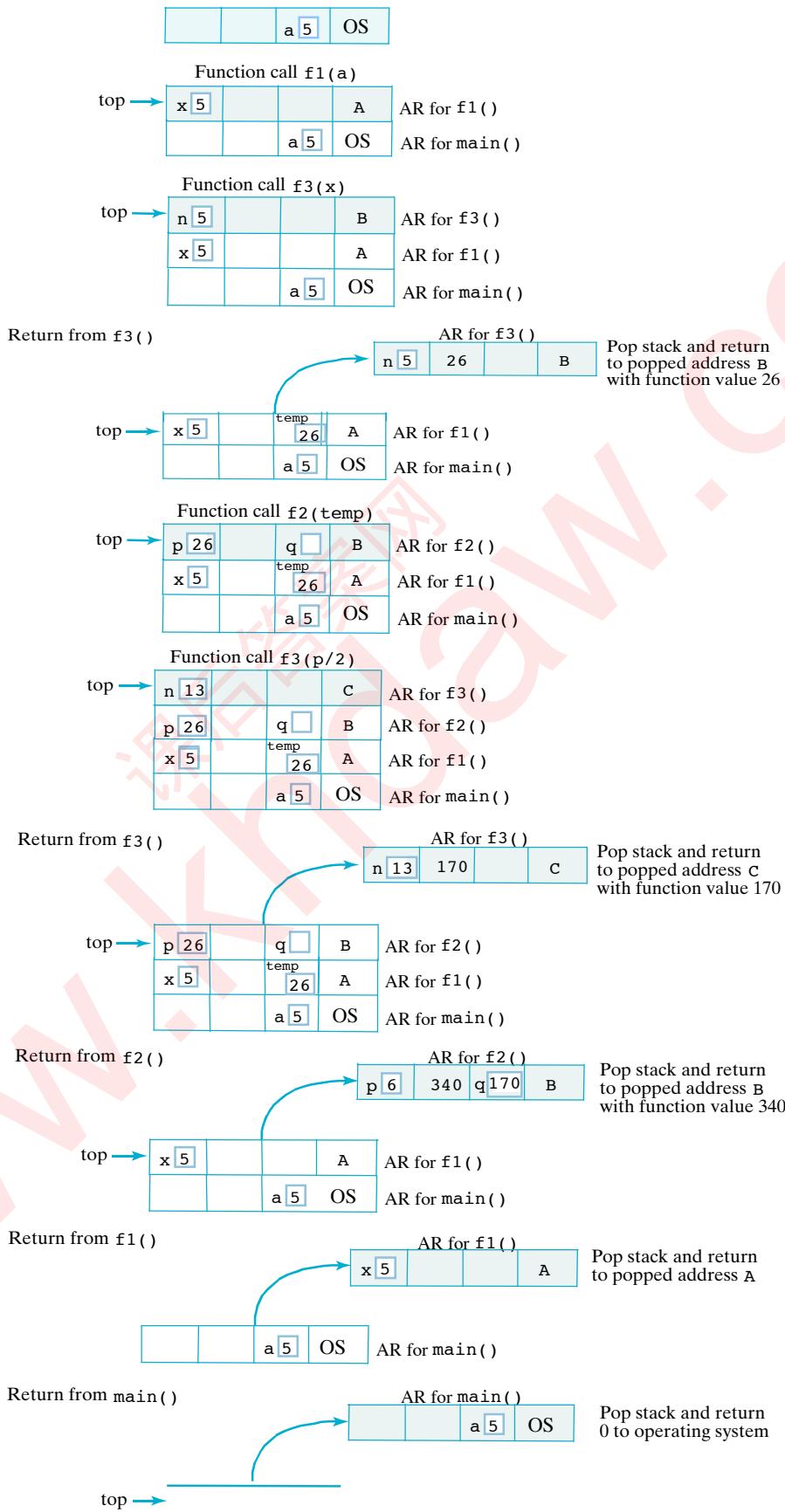




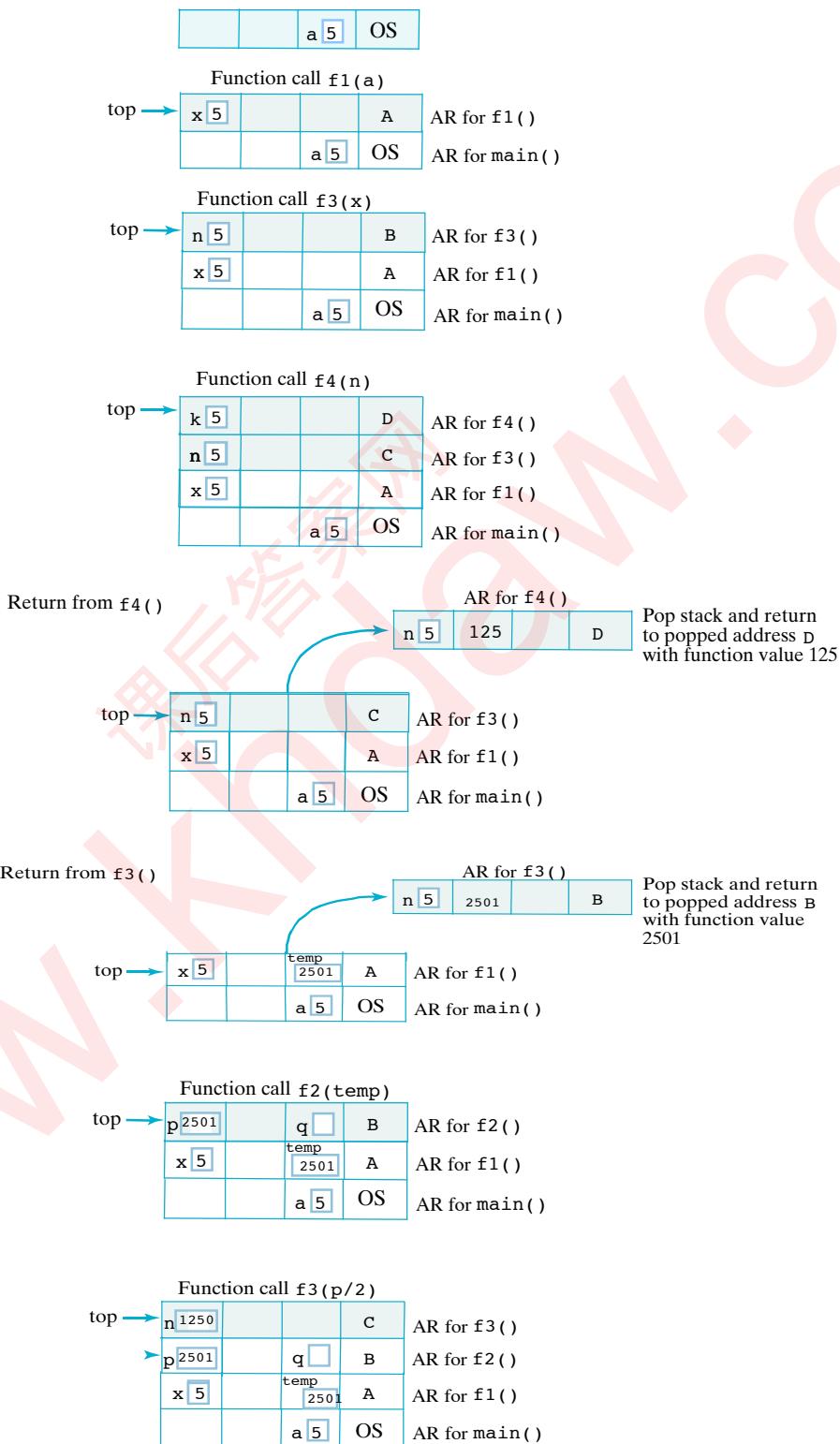
5.

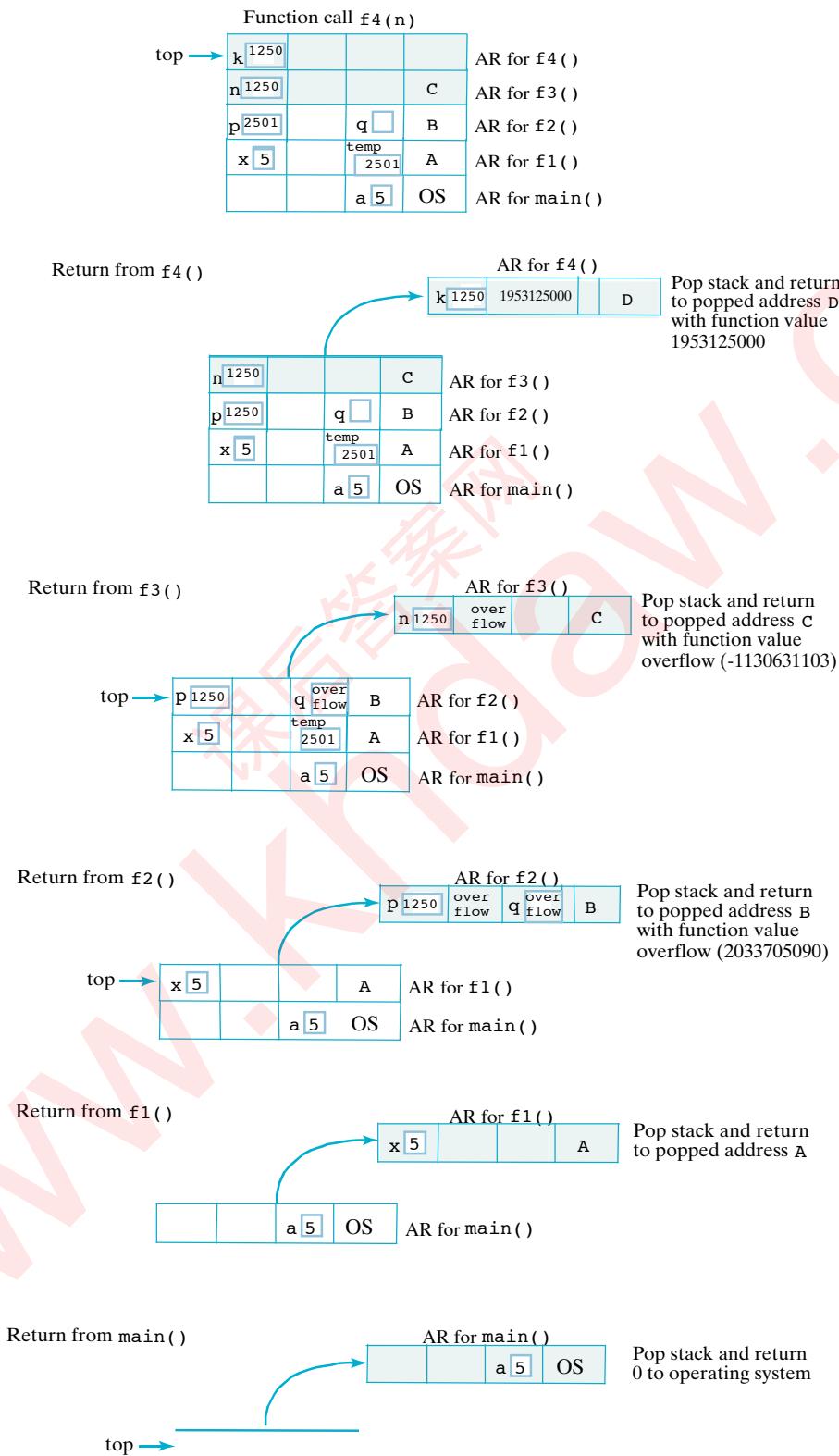


6.



7.





Exercises 7.5

1. $-7 \frac{1}{3} = -7.\overline{3}$

2. -2.0

3. 28.0

4. 12.0

5. 12.0

6. 12.0

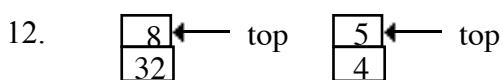
7. 12.0

8. 2.0

9. -2.0

10. 8.0

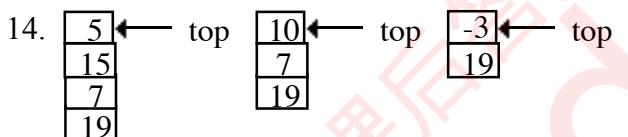
11. 8.0



Value is 20.



Value is -9.



Value is 22.

15. $a b * c + d -$

16. $a b c / + d +$

17. $a b + c / d +$

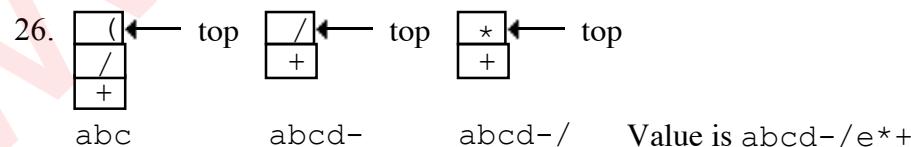
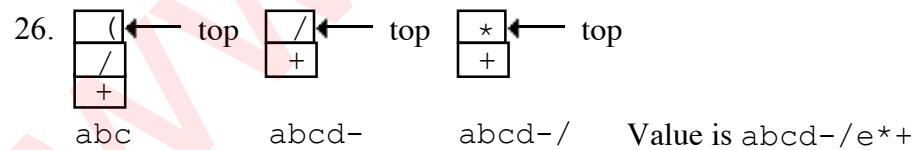
18. $a b c d + / +$

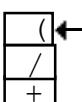
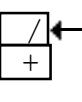
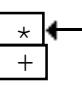
19. $a b + c d + /$

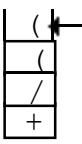
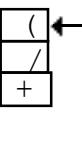
20. $a b - c d e + - *$

21. $a b - c - d - e -$

22. $a b c d e - - - -$



26.  abc  abcd-  abcd- / Value is abcd- / e * +

27.  abc  abcd- abcd- e * abcd- e * / + Value is abcd- e * / + f -

28. $(a - (b + c)) * d$
 29. $(a + b) * (c + d)$
 30. $a * (b - (c + d))$
 31. $(a + b - c) / (e * d)$
 32. $a / b / c / d$
 33. $(a / b) / (c / d)$
 34. $a / ((b / c) / d)$
 35. $a / (b / (c / d))$

36. -15
 37. 9
 38. 5
 39. 15
 40. 1
 41. -5

42. $a \ b \ c \sim + \ast$
 43. $a \ b + c \ d - / \sim$
 44. $a \sim b \sim \ast$
 45. $a \ b \sim c \ d \sim + \ast \sim \sim$
 46. $a \ b \ \&\& \ c \ ||$
 47. $a \ b \ c \ ! \ || \ \&\&$
 48. $a \ b \ \&\& \ !$
 49. $a \ b \ || \ c \ d \ e \ ! \ \&\& \ || \ \&\&$
 50. $a \ b == c \ d == \ ||$
 51. $a \ 3 < a \ 9 > \&\& a \ 0 > \ ! \ ||$
 52. $b \ b * 4 \ a * c * - 0 >= a \ 0 > a \ 0 < \ || \ \&\&$

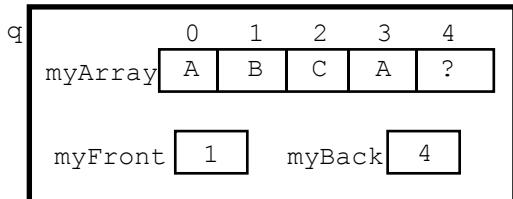
53. $- + \ast \ a \ b \ c \ d$
 54. $+ \ a + / \ b \ c \ d$
 55. $+ / + \ a \ b \ c \ d$
 56. $+ \ a / \ b + c \ d$
 57. $/ + \ a \ b + c \ d$
 58. $\ast - a \ b - c + d \ e$
 59. $- - - - a \ b \ c \ d$

60. - a - b - c - d e
61. -24.5
62. -7.34
63. 8.0
64. -2.0
65. 4.0
66. 2.0
67. 55.0 (After correcton to * + a b - c d)
68. 3
69. (a + b) * (c - d)
70. (a * b) + (c - d)
71. (a - b) - (c - d)
72. a - (b - c) - d
73. a - b - c - d
74. a * b + (c - d) / e
75. (a * b + c) / (d - e)
76. (a + b * c) / (d - e)

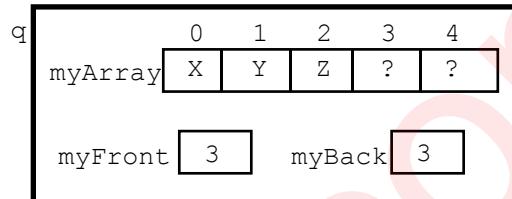
Chapter 8: Queues

Exercises 8.2

1.

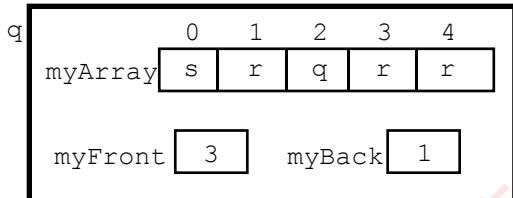


2.

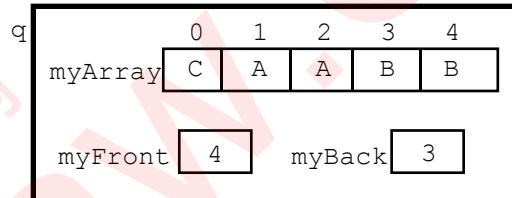


Queue is now empty

3.



4.



Error occurs when $i = 4$. After $ch = 'A'$ is inserted in location 2, $myBack$ is 3 and $myFront$ is 4, which means the queue is full, so the next `enqueue()` operation fails.

5.

```
/*-- DQueue.h -----
This header file defines a Queue data type.
Basic operations:
constructor: Constructs an empty queue
copy constructor: Constructs a copy of a queue
=: Assignment operator
destructor: Destroys a queue
empty: Checks if a queue is empty
enqueue: Modifies a queue by adding a value at the back
front: Accesses the top stack value; leaves queue unchanged
dequeue: Modifies queue by removing the value at the front
display: Displays all the queue elements
-----*/
#include <iostream>
#ifndef DQUEUE
#define DQUEUE
typedef int QueueElement;
```

```
class Queue
{
public:
    //***** Function members *****/
    //***** Constructors *****/

Queue(int numElements = 128);
/*-----
   Construct a Queue object.

Precondition: None.
Postcondition: An empty Queue object has been constructed
   (myFront and myBack are initialized to 0 and myArray
   is an array with numElements (default 128) elements
   of type QueueElement).
-----*/
Queue (const Queue & original);
/*-----
   Copy Constructor

Precondition: original is the queue to be copied and
   is received as a const reference parameter.
Postcondition: A copy of original has been constructed.
-----*/
//***** Destructor *****/
~Queue();
/*-----
   Class destructor

Precondition: None
Postcondition: The dynamic array in the queue has been
   deallocated.
-----*/
//***** Assignment *****/
const Queue & operator= (const Queue & rightHandSide);
/*-----
   Assignment Operator

Precondition: original is the queue to be assigned and
   is received as a const reference parameter.
Postcondition: The current queue becomes a copy of
   original and a const reference to it is returned.
-----*/
bool empty() const;
/*-----
   Check if queue is empty.
Precondition: None
Postcondition: Returns true if queue is empty and
   false otherwise.
-----*/
```

```
void enqueue(const QueueElement & value);
/*-----
   Add a value to a queue.

   Precondition: value is to be added to this queue
   Postcondition: value is added at back of queue provided
      there is space; otherwise, a queue-full message is
      displayed and execution is terminated.
-----*/
void display(ostream & out) const;
/*-----
   Display values stored in the queue.

   Precondition: ostream out is open.
   Postcondition: Queue's contents, from front to back, have
      been output to out.
-----*/
QueueElement front() const;
/*-----
   Retrieve value at front of queue (if any).

   Precondition: Queue is nonempty
   Postcondition: Value at front of queue is returned, unless
      the queue is empty; in that case, an error message is
      displayed and a "garbage value" is returned.
-----*/
void dequeue();
/*-----
   Remove value at front of queue (if any).

   Precondition: Queue is nonempty.
   Postcondition: Value at front of queue has been removed,
      unless the queue is empty; in that case, an error
      message is displayed and execution allowed to proceed.
-----*/
private:
    ***** Data members *****/
    int myFront,           // front
        myBack;           // and back of queue
    int myCapacity;         // capacity of queue
    QueueElement * myArray; // dynamic array to store elements;
                           // empty slot used to distinguish
                           // between empty and full
}; // end of class declaration

#endif

/*-- DQueue.cpp-----
   This file implements Stack member functions.
   Empty slot used to distinguish between empty and full
-----*/
```

```
#include <iostream>
#include <cassert>
#include <new>
using namespace std;

#include "DQueue.h"

//--- Definition of Queue constructor
Queue::Queue(int numElements)
{
    assert (numElements > 0); // check precondition
    myCapacity = numElements; // set queue capacity
                           // allocate array of this capacity
    myArray = new(nothrow) QueueElement[myCapacity];
    if (myArray != 0)          // memory available
        myFront = myBack = 0;
    else
    {
        cerr << "Inadequate memory to allocate queue \n"
            " -- terminating execution\n";
        exit(1);
    }                                // or assert(myArray != 0);
}

//--- Definition of Queue copy constructor
Queue::Queue(const Queue & original)
: myCapacity(original.myCapacity),
  myFront(original.myFront), myBack(original.myBack)
{
    //--- Get new array for copy
    myArray = new(nothrow) QueueElement[myCapacity];
    if (myArray != 0)          // check if memory available
        // copy original's array member into this new array
        for (int i = myFront; i!= myBack; i = (i + 1)%myCapacity)
            myArray[i] = original.myArray[i];
    else
    {
        cerr << "*Inadequate memory to allocate queue ***\n";
        exit(1);
    }
}

//--- Definition of Queue destructor
Queue::~Queue()
{
    delete [] myArray;
}

//--- Definition of assignment operator
const Queue & Queue::operator=(const Queue & rightHandSide)
{
    if (this != &rightHandSide)           // check that not st = st
    {
        //-- Allocate a new array if necessary
        if (myCapacity != rightHandSide.myCapacity)
        {
            delete[] myArray;             // destroy previous array
            myCapacity = rightHandSide.myCapacity; // copy myCapacity
        }
    }
}
```

```
myArray = new QueueElement[myCapacity];
if (myArray == 0) // check if memory available
{
    cerr << "**** Inadequate memory ***\n";
    exit(1);
}

myFront = rightHandSide.myFront; // copy myFront member
myBack = rightHandSide.myBack; // copy myBack member
// copy queue elements
for (int i = myFront; i!= myBack; i= (i + 1)%myCapacity)
    myArray[i] = rightHandSide.myArray[i];
}

return *this;
}

//--- Definition of empty()
inline bool Queue::empty() const
{
    return myFront == myBack;
}

//--- Definition of enqueue()
void Queue::enqueue(const QueueElement & item)
{
    if ((myBack +1)% myCapacity == myFront)
        cerr << "Queue is full: cannot add to queue. Error!! " << endl;
    else
    {
        myArray[myBack] = item;
        myBack = (myBack+ 1) % myCapacity;
    }
}

//--- Definition of front()
QueueElement Queue::front() const
{
    if (myFront == myBack)
    {
        cerr <<"Queue is empty: error! Returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
    else
        return myArray[myFront];
}

//--- Definition of dequeue()
void Queue::dequeue()
{
    if (myFront == myBack)
        cerr <<"Queue is empty: cannot remove from queue: error!\n";
    else
        myFront= (myFront + 1) % myCapacity;
}
```

```
void Queue::display(ostream & out) const
{
    for (int i = myFront; i != myBack; i = (i + 1) % myCapacity)
        cout << myArray[i] << " ";
    cout << endl;
}

6. // Prototype:
bool full() const;
/*-----
   Check if queue is full.

   Precondition: None
   Postcondition: Returns true if queue is full and false otherwise.
-----*/
// Definition:
bool Queue::full()
{
    return myFront == (myBack + 1)% QUEUE_CAPACITY;
}

// Definition:
bool Queue::full()
{
    return myFront == (myBack + 1)% myCapacity;
}

7. // Prototype:
int size() const;
/*-----
   Find number of elements in the queue.
   Precondition: None
   Postcondition: Number of queue elements is returned.
-----*/
// Definition:
int Queue::size() const
{
    if (myFront == myBack)
        return 0;
    else if (myFront > myBack)
        return myBack - myFront + QUEUE_CAPACITY;
    else
        return myBack - myFront;
}

8. // Prototype
int size(Queue q);
/*-----
   Find number of elements in a queue received as a value parameter.
   Precondition: None
   Postcondition: Number of queue elements is returned.
-----*
```

```
// Definition
int size(Queue q)
{
    int count = 0;
    while (!q.empty())
    {
        q.removeQ();
        count++;
    }
    return count;
}

/* Here is a version that preserves the parameter q. */
int size(Queue q)
{
    Queue temp;
    int count = 0;

    while (!q.Empty())
    {
        temp.addQ(q.front());
        q.removeQ();
        count++;
    }
    while (!temp.empty())
    {
        q.addQ(temp.front());
        temp.removeQ();
    }
    return count;
}
```

9.

```
// Prototype:
QueueElement back() const;
/*
   Retrieve the back element of this queue.
   Precondition: None
   Postcondition: Back element of the queue is returned, unless there
                  was none, in which case a queue-empty message is displayed.
-----*/
// Definition:
QueueElement Queue::back() const
{
    if (myFront == myBack)
    {
        cerr << "Error: queue is empty -- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
    //else
    if (myBack == 0)
        return myArray[QUEUE_CAPACITY - 1];
    //else
    return myArray[myBack - 1];
}
```

10.

```
// Prototype:  
QueueElement back();  
/*-----  
   Retrieve the back element of a queue received as a value parameter.  
   Precondition: None  
   Postcondition: Back element of the queue is returned, unless there  
   was none, in which case a queue-empty message is displayed.  
-----*/  
  
// Definition:  
QueueElement back(Queue q)  
{  
    if (q.empty())  
    {  
        cerr << "Error: queue is empty -- returning garbage value\n";  
        QueueElement garbage;  
        return garbage;  
    }  
    //else  
    QueueElement last;  
    while (!q.empty())  
    {  
        last = q.front();  
        q.dequeue();  
    }  
    return last;  
}  
  
//-- Non-destructive version (preserves parameter q)  
QueueElement back(Queue q) const  
{  
    if (q.empty())  
    {  
        cerr << "Error: queue is empty -- returning garbage value\n";  
        QueueElement garbage;  
        return garbage;  
    }  
    //else  
    Queue temp;  
    QueueElement last;  
    while (!q.empty())  
    {  
        last = q.front();  
        temp.addQ(last);  
        q.removeQ();  
    }  
    while (!temp.empty())  
    {  
        q.addQ(temp.front());  
        temp.removeQ();  
    }  
    return last;  
}
```

11.

```
// Prototype:  
QueueElement nthElement(int n);  
/*-----  
   Retrieve the n-th element of a queue.  
   Precondition: 1 <= n <= number of queue elements  
   Postcondition: n-th element of the queue is returned, unless queue  
   has fewer than n elements, in which case an error message is  
   displayed. Also, the elements preceding the n-th element are  
   removed from the queue.  
-----*/  
  
// Definition:  
QueueElement Queue::nthElement(int n)  
{  
    QueueElement elem;  
    while( n > 0 && !empty())  
    {  
        elem = front();  
        removeQ();  
        n--;  
    }  
    if (n > 0)  
    {  
        cerr << "Error: insufficient number of elements in the queue\n";  
        " -- returning garbage value\n";  
        QueueElement garbage;  
        return garbage;  
    }  
    //else  
    return elem;  
}
```

12.

```
// Prototype:  
QueueElement nthElement(int n) const;  
/*-----  
   Retrieve the n-th element of a queue.  
   Precondition: 1 <= n <= number of queue elements  
   Postcondition: n-th element of the queue is returned, unless queue  
   has fewer than n elements, in which case an error message is  
   displayed..  
-----*/  
  
// Definition:  
QueueElement Queue::nthElement(int n) const  
{  
    if (myFront < myBack && myBack - myFront < n  
        || myFront > myBack && QUEUE_CAPACITY - (myFront - myBack) < n)  
    {  
        cerr << "Error: insufficient number of elements in the queue\n";  
        " -- returning garbage value\n";  
        QueueElement garbage;  
        return garbage;  
    }
```

```

    //else
    int index_n = (myFront + n - 1) % QUEUE_CAPACITY;
    return myArray[index_n];
}

```

13. The algorithm is as follows:

1. Create a stack.
2. While the queue is not empty, do the following:
 - a. Remove an item from the queue.
 - b. Push this item onto the stack.
3. While the stack is not empty, do the following:
 - a. Pop an item from the stack.
 - b. Add this item to the queue.

14.

(a) For $n = 3$:

Possible Permutations
123 132 213 231 312
=====

Impossible Permutations
321
=====

(b) For $n = 4$:

Possible Permutations
1234 1324 1342 1423
2134 2143 2314 2341 2413
3124 3142 3412
4123
=====

Impossible Permutations
1243
2431
3214 3241 3421
4132 4312 4321 4213 4231
=====

(c) For $n = 5$:

Possible Permutations
12345 12354 12435 12453 12534
13245 13254 13425 13452 13524
14235 14253 14523
15234

21345 21354 21435 21453 21534
23145 23154 23415 23451 23514
24135 24153 24513
25134

31245 31254 31425 31452 31524
34125 34152 34512
35124

Impossible Permutations
12543
13542
14325 14352 14532
15243 15324 15342 15423 15432

21543
23541
24315 24351 24531
25143 25314 25341 25413 25431

31542
32145 32154 32415 32451 32514 32541
34215 34251 34521
35142 35214 35241 35412 35421

41235 41253 41523	41325 41352 41532
	42135 42153 42315 42351 42513 42531
	43125 43152 43215 43251 43512 43521
45123	45132 45213 45231 45312 45321
51234	51243 51324 51342 51423 51432
	52134 52143 52314 52341 52413 52431
	53124 53142 53214 53241 53412 53421
	54123 54132 54213 54231 54312 54321

- (d) The rule is: for each digit d in the number, the digits to the right of d that are less than d MUST be in ascending order.

15.

```

/* Implementation of Queue class.
Count of elements used to distinguish between empty and full

Add a data member: int myCount; to the private section of
the Queue class declaration.

*/
#include <iostream>
using namespace std;

Queue::Queue()
: myFront(0), myBack(0), myCount(0)
{}

bool Queue::empty() const
{
    return myCount == 0;
}

void Queue::enqueue(const QueueElement & value)
{
    if (myCount < QUEUE_CAPACITY)
    {
        myArray[myBack] = value;
        myBack = (myBack + 1) % QUEUE_CAPACITY;
        myCount++;
    }
    else
    {
        cerr << "**** Queue full -- can't add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n";
        exit(1);
    }
}

QueueElement Queue::front()
{
    if (myCount > 0)
        return myArray[myFront];
    else
    {

```

```
    cerr << "**** Queue is empty -- returning garbage value ***\n";
    QueueElement garbage;
    return garbage;
}
}

void Queue::dequeue()
{
    if (myCount > 0)
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        myCount--;
    }
    else
        cerr << "**** Queue is empty -- can't remove a value ***\n";
}

16.
/* Implementation of Queue class.
Count of elements used to distinguish between empty and full.
No data member myBack is used.

Add a data member: int myCount; to the private section of
the Queue class declaration and remove: int myBack;
*/
#include <iostream>
using namespace std;

Queue::Queue()
: myFront(0), myCount(0)
{}

bool Queue::empty() const
{
    return myCount == 0;
}

void Queue::enqueue(const QueueElement & value)
{
    if (myCount < QUEUE_CAPACITY)
    {
        int back = (myFront + myCount) % QUEUE_CAPACITY;
        myArray[back] = value;
        myCount++;
    }
    else
    {
        cerr << "**** Queue full -- can't add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n";
        exit(1);
    }
}

QueueElement Queue::front()
{
    if (myCount > 0)
        return myArray[myFront];
```

```
else
{
    cerr << "**** Queue is empty -- returning garbage value ***\n";
    QueueElement garbage;
    return garbage;
}
}

void Queue::dequeue()
{
    if (myCount > 0)
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        myCount--;
    }
    else
        cerr << "**** Queue is empty -- can't remove a value ***\n";
}
```

17.

```
/* Implementation of Queue class.
   Full data member used to distinguish between empty and full

   Add a data member:  bool iAmFull;  to the private section of
   the Queue class declaration.

*/
#include <iostream>
using namespace std;

Queue::Queue()
: myFront(0), myCount(0), iAmFull(false)
{ }

bool Queue::empty() const
{
    return (myBack == myFront && !iAmFull);
}

void Queue::enqueue(const QueueElement & item)
{
    if (!iAmFull)
    {
        myArray[myBack] = item;
        myBack = (myBack+ 1) % QUEUE_CAPACITY;
        iAmFull = (myBack == myFront);
    }
    else
    {
        cerr << "**** Queue full -- can't add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n";
        exit(1);
    }
}
```

```

QueueElement Queue::front()
{
    if (!empty())
    {
        return myArray[myFront];
    }
    else
    {
        cerr << "*** Queue is empty -- returning garbage value ***\n";
        QueueElement garbage;
        return garbage;
    }
}

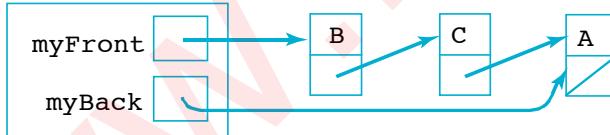
void Queue::dequeue()
{
    if (!empty())
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        iAmFull = false;
    }
    else
        cerr << "Queue is empty: cannot remove from queue. Error!!" << endl;
}

```

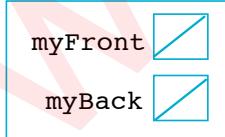
18. This is similar to the use of one buffer for two stacks (Exer. 12 in §7.2): If two queues were to be stored in one array with the front of each being at the ends of the array, then the queues could grow until the backs met in the middle. Then, one of the queues would have to be shifted back to its end. If each queue size is fixed, wraparound within each queue could be employed to avoid shifting elements.

Exercises 8.3

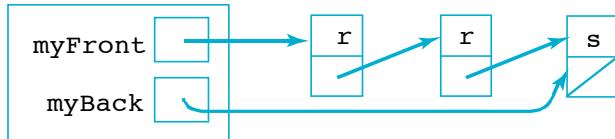
1.



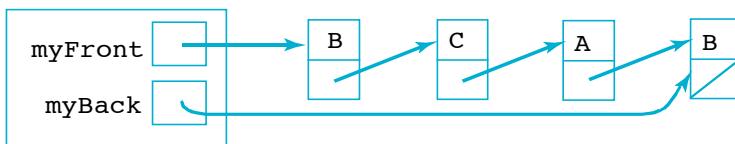
2.



3.



4.



5.

```
// Prototype:
QueueElement back() const;
/*-
 * Retrieve the back element of this queue.
 * Precondition: None
 * Postcondition: Back element of the queue is returned, unless there
 * was none, in which case a queue-empty message is displayed.
 */

```

```
// Definition:
QueueElement Queue::back() const
{
    if (myBack != 0)
        return * myArray;
    //else
    //cerr << "Error: queue is empty -- returning garbage value\n";
    QueueElement garbage;
    return garbage;
}
```

6.

```
// Prototype:
QueueElement nthElement(int n);
/*-
 * Retrieve the n-th element of a queue.
 * Precondition: 1 <= n <= number of queue elements
 * Postcondition: n-th element of the queue is returned, unless queue
 * has fewer than n elements, in which case an error message is
 * displayed. Also, the elements preceding the n-th element are
 * removed from the queue.
 */

```

```
// Definition:
QueueElement Queue::nthElement(int n)
{
    QueueElement elem;
    while( n > 0 && !empty())
    {
        elem = front();
        removeQ();
        n--;
    }
    if (n > 0)
    {
        cerr << "Error: insufficient number of elements in the queue\n";
        " -- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
}
```

```

    //else
    return elem;
}

7.
// Prototype:
QueueElement nthElement(int n) const;
/*-----
   Retrieve the n-th element of a queue.
   Precondition: 1 <= n <= number of queue elements
   Postcondition: n-th element of the queue is returned, unless queue
      has fewer than n elements, in which case an error message is
      displayed..
-----*/
// Definition:
QueueElement Queue::nthElement(int n) const
{
    int count = 0;
    Queue::NodePonter ptr = myFront;

    for (int count = 0; count < n && ptr != 0; count++)
        ptr = ptr->next;

    if (ptr != 0)
        return *ptr;
    //else
    cerr << "Error: insufficient number of elements in the queue\n";
    " -- returning garbage value\n";
    QueueElement garbage;
    return garbage;
}

8.
/*-- CLQueue.h -----*/
This header file defines a Queue data type.
Basic operations:
constructor: Constructs an empty queue
copy constructor: Constructs a copy of a queue
=: Assignment operator
destructor: Destroys a queue
empty: Checks if a queue is empty
enqueue: Modifies a queue by adding a value at the back
front: Accesses the top stack value; leaves queue
       unchanged
dequeue: Modifies queue by removing the value at the
          front
display: Displays all the queue elements

A circular linked list is used to store the queue elements.
-----*/

```

```
#ifndef CLQUEUE
#define CLQUEUE

#include <iostream>

typedef int QueueElement;
class Queue
{
private:
    class Node
    {
public:
    //----- DATA MEMBERS OF Node
    QueueElement data;
    Node * next;

    //----- Node OPERATIONS

    /* --- The Node default class constructor initializes a Node's
       next member.

       Precondition: None
       Postcondition: The next member has been set to 0.
    */
    Node()
    : next(0)
    {}

    /* --- The Node class constructor initializes a Node's data members.

       Precondition: None
       Postcondition: The data and next members have been set to
                      dataValue and 0, respectively.
    */
    Node(DueueElement dataValue)
    : data(dataValue), next(0)
    {}
}; //--- end of Node class

typedef Node * NodePointer;

public:
    ***** Function members *****/
    ***** Constructors *****

    Queue();
    /*-
       Construct a Queue object.

       Precondition: None.
       Postcondition: An empty Queue object has been constructed
                      (myBack is initialized to 0).
    */
}
```

```
Queue (const Queue & original);
/*-----
   Copy Constructor

   Precondition: original is the queue to be copied and
   is received as a const reference parameter.
   Postcondition: A copy of original has been constructed.
-----*/
***** Destructor *****
~Queue();
/*-----
   Class destructor

   Precondition: None
   Postcondition: The linked list in the queue has been
   destroyed.
-----*/
***** Assignment *****
const Queue & operator=(const Queue & rightHandSide);
/*-----
   Assignment Operator

   Precondition: original is the queue to be assigned and
   is received as a const reference parameter.
   Postcondition: The current queue becomes a copy of
   original and a const reference to it is returned.
-----*/
bool empty() const;
/*-----
   Check if queue is empty.
   Precondition: None
   Postcondition: Returns true if queue is empty and
   false otherwise.
-----*/
void enqueue(const QueueElement & value);
/*-----
   Add a value to a queue.

   Precondition: value is to be added to this queue
   Postcondition: value is added at back of queue provided
   memory is available otherwise, a memory-error message
   is displayed and execution is terminated.
-----*/
QueueElement front() const;
/*-----
   Retrieve value at front of queue (if any).

   Precondition: Queue is nonempty
   Postcondition: Value at front of queue is returned, unless
   the queue is empty; in that case, an error message is
   displayed and a "garbage value" is returned.
-----*/
```

```
void dequeue();
/*-----
   Remove value at front of queue (if any).

   Precondition: Queue is nonempty.
   Postcondition: Value at front of queue has been removed,
      unless the queue is empty; in that case, an error
      message is displayed and execution allowed to proceed.
-----*/
void display(ostream & out) const;
/*-----
   Display values stored in the queue.

   Precondition: ostream out is open.
   Postcondition: Queue's contents, from front to back, have
      been output to out.
-----*/
private:
    //***** Data member *****/
    NodePointer myBack;

}; //--- end of Queue class
#endif

/*-- CLQueue.cpp-----
   This file implements Stack member functions.
   A circular linked list with pointer to last node is used to
   store the queue elements.
-----*/
#include <iostream>
using namespace std;

#include "CLQueue.h"

// Definition of constructor
Queue::Queue()
: myBack(0)
{ }

// Definition of empty()
bool Queue::empty() const
{ return myBack == 0; }

// Definition of enqueue()
void Queue::enqueue(const QueueElement & dataVal)
{
    Queue::NodePointer newPtr = new(nothrow) Node(dataVal);
    if (newPtr == 0)
    { cerr << "Out of memory\n"; exit(1); }

    if (myBack == 0)
        newPtr->next = newPtr;
}
```

```
else
{
    newPtr->next = myBack->next;
    myBack->next = newPtr;
}
myBack = newPtr;
}

// Definition of front()
QueueElement Queue::front() const
{
    if (myBack == 0)
    {
        cerr << "Queue is empty: error!  Returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
    // else
    return myBack->next->data;
}

// Definition of dequeue()
void Queue::dequeue()
{
    if (myBack == 0)
        cerr << "Queue is empty: cannot remove from queue: error!\n";
    else
    {
        Queue::NodePointer ptr = myBack->next;
        if (ptr->next == ptr)      // one-element queue becomes empty
            myBack = 0;
        else
            myBack->next = ptr->next;
        delete ptr;
    }
}

// Definition of the destructor
Queue::~Queue()
{
    if (myBack != 0)
    {
        Queue::NodePointer ptr,
                        prev = myBack->next;
        while (prev != myBack)
        {
            ptr = prev->next;
            delete prev;
            prev = ptr;
        }
        delete myBack;
    }
}
```

```
// Definition of the copy constructor
Queue::Queue(const Queue & original)
{
    myBack = 0;
    if (!original.empty())
    {
        Queue::NodePointer origPtr = original.myBack->next,
                           frontPtr, lastPtr;

        frontPtr = new Node(origPtr->data);
        if (frontPtr == 0)
            { cerr << "Out of memory\n"; exit(1); }

        lastPtr = frontPtr;
        while (origPtr != original.myBack)
        {
            origPtr = origPtr->next;
            lastPtr->next = new Node(origPtr->data);
            if (lastPtr == 0)
                { cerr << "Out of memory\n"; exit(1); }

            lastPtr = lastPtr->next;
        }
        lastPtr->next = frontPtr;
        myBack = lastPtr;
    }
}

// Definition of the assignment operator
const Queue & Queue::operator=(const Queue & original)
{
    myBack = 0;
    if (this != &original)
    {
        delete myBack;
        Queue::NodePointer origPtr = original.myBack->next,
                           frontPtr, lastPtr;

        frontPtr = new Node(origPtr->data);
        if (frontPtr == 0)
            { cerr << "Out of memory\n"; exit(1); }

        lastPtr = frontPtr;
        while (origPtr != original.myBack)
        {
            origPtr = origPtr->next;
            lastPtr->next = new Node(origPtr->data);
            if (lastPtr == 0)
                { cerr << "Out of memory\n"; exit(1); }

            lastPtr = lastPtr->next;
        }
        lastPtr->next = frontPtr;
        myBack = lastPtr;
    }
    return *this;
}
```

```

// Definition of the output operators
void Queue::display(ostream & out) const
{
    if (empty()) return;

    Queue::NodePointer ptr = myBack;
    do
    {
        ptr = ptr->next;
        out << ptr->data << " ";
    }
    while (ptr != myBack);

}

inline ostream & operator<<(ostream & out, const Queue & aQueue)
{
    aQueue.display(out);
    return out;
}

// See Programming Problem 18 for a driver

```

Exercises 8.4

1.

```

/*----- Deque.h -----
A deque (double-ended queue) is similar to a queue but additions
and deletions may be performed on either end. Each store/retrieve
operation must specify at which end the operation is to be performed.

Basic operations:
Constructor: Constructs an empty deque
empty: Checks if a deque is empty
add: Modifies a deque by adding a value at one end
retrieve: Retrieve the value at one end; leaves deque unchanged
remove: Modifies a deque by removing the value at one end
display: Displays the deque elements

Class Invariant:
1. The deque elements (if any) are stored in consecutive positions
   in myArray, beginning at position myFront.
2. 0 <= myFront, myBack < DEQUE_CAPACITY
3. Deque's size < DEQUE_CAPACITY
-----*/
#include <iostream>

#ifndef DEQUE
#define DEQUE

const int DEQUE_CAPACITY = 128;
typedef int DequeElement;
enum End {FRONT, BACK};

```

```
class Deque
{
    /***** Function Members *****/
public:
    Deque();
    /*-----
        Construct a Deque object.

        Precondition: None.
        Postcondition: An empty Deque object has been constructed; myFront
            and myBack are initialized to -1 and myArray is an array with
            DEQUE_CAPACITY elements of type DequeElement.
    -----*/
    bool empty() const;
    /*-----
        Check if deque is empty.

        Precondition: None.
        Postcondition: True is returned if the deque is empty and false is
            returned otherwise.
    -----*/
    void add(const DequeElement & value, End where);
    /*-----
        Add a value to a deque.

        Precondition: where is FRONT (0) or BACK (1).
        Postcondition: value is added at end of deque specified by where,
            provided there is space; otherwise, a deque-full message is
            displayed and execution is terminated.
    -----*/
    DequeElement retrieve(End where) const;
    /*-----
        Retrieve value at one end of deque (if any).

        Precondition: Deque is nonempty; where is FRONT (0) or BACK (1).
        Postcondition: Value at at end of deque specified by where is
            returned, unless deque is empty; in that case, an error message
            is displayed and a "garbage value" is returned.
    -----*/
    void remove(End where);
    /*-----
        Remove value at one end of deque (if any).

        Precondition: Deque is nonempty; where is FRONT (0) or BACK (1).
        Postcondition: Value at at end of deque specified by where is
            removed, unless deque is empty; in that case, an error message
            is displayed.
    -----*/
}
```

```
// --- display
void display(ostream & out) const;
/*-----
   Output the values stored in the deque.

   Precondition: ostream out is open.
   Postcondition: Deque's contents have been output to out.
-----*/
***** Data Members *****
private:
    DequeElement myArray[DEQUE_CAPACITY];
    int myFront,
        myBack;

}; // end of class declaration

#endif

//---- Deque.cpp -----
#include <iostream>
#include <cassert>
using namespace std;

#include "Deque.h"

//-- Definition of constructor
Deque::Deque ()
: myFront(0), myBack(0)
{ }

//-- Definition of empty()
bool Deque::empty() const
{
    return myFront == myBack;
}

//-- Definition of add()
void Deque::add(const DequeElement & value, End where)
{
    assert (where == FRONT || where == BACK);
    int newBack = (myBack + 1) % DEQUE_CAPACITY;
    if (newBack == myFront)
    {
        cerr << "Deque is full: cannot add to deque. Error!! " << endl;
        exit(1);
    }

    //else
    if (where == BACK)
    {
        myArray[myBack] = value;
        myBack = newBack;
    }
}
```

```

    else
    {
        int beforeFront = (myFront > 0 ? myFront - 1 : DEQUE_CAPACITY - 1);
        myFront = beforeFront;
        myArray[myFront] = value;
    }
}

//-- Definition of retrieve()
DequeElement Deque::retrieve(End where) const
{
    assert (where == FRONT || where == BACK);
    if (myFront == myBack)
    {
        cerr <<"Deque is empty:Error!!" << endl;
        return myArray[myBack];// some invalid data item;
    }
    else if (where == FRONT)
        return myArray[myFront];
    else
        return myArray[myBack > 0 ? myBack - 1 : DEQUE_CAPACITY - 1];
}

//-- Definition of remove()
void Deque::remove(End where)
{
    assert (where == FRONT || where == BACK);
    if (myFront == myBack)
    {
        cerr <<"Deque empty: Cannot remove an element. Error!!"
            << endl;
        return;
    }
    else if (where == FRONT)
        myFront= (myFront + 1) % DEQUE_CAPACITY;
    else
        myBack = (myBack > 0 ? myBack - 1 : DEQUE_CAPACITY - 1);
    // Result of (myBack - 1) % DEQUE_CAPACITY is implementation-
    // dependent if first operand of % is negative.
}

//-- Definition of display()
void Deque::display(ostream & out) const
{
    cout << "front: " << myFront << " back: " << myBack << endl;
    for (int i = myFront; i != myBack; i = (i + 1) % DEQUE_CAPACITY)
        cout << myArray[i] << " ";
    cout << endl;
}

//-- See Programming Problem 20 for a driver program.

```

2. Implementing a scroll is an easy restriction of the deque class in Exercise 1 — simply restrict the add operation to the front and remove to the back.

Chapter 9: ADT Implementations: Templates and Standard Containers

Exercises 9.3

1.

```
template <typename DataType>
DataType average(DataType a, DataType b)
{
    return (a + b)/2;
}
```

2.

```
template <typename DataType>
DataType max(DataType a, DataType b)
{
    if (a > b)
        return a;
    else
        return b;
```

3.

```
template <typename DataType>
DataType median (DataType a, DataType b, DataType c)
{
    DataType max = a,
            min = b;

    if (a < b)
    {
        max = b;
        min = a;
    }
    if (c > max)
        return max;
    else if (c < min)
        return min;
    else
        return c;
}
```

4.

```
template <typename DataType>
DataType arraySum(DataType x[], int length)
{
    DataType sum = x[0];

    for (int index = 1; index < length; index++)
        sum += x[index];

    return sum;
}
```

5.

```
template <typename DataType>
void arrayMaxMin(DataType x[], int length, DataType & min, DataType & max)
{
    min = max = x[0];
    for (int i = 1; i < length; i++)
    {
        if (x[i] < min)
            min = x[i];
        if (x[i] > max)
            max = x[i];
    }
}
```

6.

```
template <typename DataType>
int search(DataType x[], int length, DataType target)
{
    for (int index = 0; index < length; index++)
        if (x[index] == target)
            return index;

    return -1; // index of -1 denotes not found
}
```

7.

```
/*-- ListT.h --
This header file defines the data type List for processing lists.
Basic operations are:
    Constructor
    empty:    Check if list is empty
    insert:   Insert an item
    erase:    Remove an item
    display:  Output the list
    <<:       Output operator
*/
#include <iostream>

#ifndef LISTT
#define LISTT

const int CAPACITY = 1024;
template <typename ElementType>

class List
{
public:
    ***** Function Members *****/
    /** Class constructor ***/
    List();
    /*-----
        Construct a List object.

    Precondition: None
    Postcondition: An empty List object has been constructed;
                    mySize is 0.
    */
}
```

```
***** empty operation *****
bool empty() const;
/*-----
   Check if a list is empty.

   Precondition: None
   Postcondition: true is returned if the list is empty, false if not.
-----*/
***** insert and erase *****
void insert(ElementType item, int pos);
/*-----
   Insert a value into the list at a given position.

   Precondition: item is the value to be inserted; there is room in
      the array (mySize < CAPACITY); and the position satisfies
      0 <= pos <= mySize.
   Postcondition: item has been inserted into the list at the position
      determined by pos (provided there is room and pos is a legal
      position).
-----*/
void erase(int pos);
/*-----
   Remove a value from the list at a given position.

   Precondition: The list is not empty and the position satisfies
      0 <= pos < mySize.
   Postcondition: element at the position determined by pos has been
      removed (provided pos is a legal position).
-----*/
***** output *****
void display(ostream & out) const;
/*-----
   Display a list.

   Precondition: The ostream out is open.
   Postcondition: The list represented by this List object has been
      inserted into out.
-----*/
private:
***** Data Members *****
int mySize;                                // current size of list stored in myArray
ElementType myArray[CAPACITY];    // array to store list elements

}; //--- end of List class template

//----- Prototype of output operator
template <typename ElementType>
ostream & operator<< (ostream & out, const List<ElementType> & aList);

#endif
```

```
//--- Definition of class constructor
template <typename ElementType>
inline List<ElementType>::List()
: mySize(0)
{ }

//--- Definition of empty()
template <typename ElementType>
inline bool List<ElementType>::empty() const
{
    return mySize == 0;
}

//--- Definition of display()
template <typename ElementType>
inline void List<ElementType>::display(ostream & out) const
{
    for (int i = 0; i < mySize; i++)
        out << myArray[i] << " ";
}

//--- Definition of output operator
template <typename ElementType>
inline ostream & operator<<(ostream & out,
                           const List<ElementType> & aList)
{
    aList.display(out);
    return out;
}

//--- Definition of insert()
template <typename ElementType>
void List<ElementType>::insert(ElementType item, int pos)
{
    if (mySize == CAPACITY)
    {
        cerr << "*** No space for list element -- terminating "
            "execution ***\n";
        exit(1);
    }
    if (pos < 0 || pos > mySize)
    {
        cerr << "*** Illegal location to insert -- " << pos
            << ". List unchanged. ***\n";
        return;
    }

    // First shift array elements right to make room for item

    for(int i = mySize; i > pos; i--)
        myArray[i] = myArray[i - 1];

    // Now insert item at position pos and increase list size
    myArray[pos] = item;
    mySize++;
}
```

```

//--- Definition of erase()
template <typename ElementType>
void List<ElementType>::erase(int pos)
{
    if (mySize == 0)
    {
        cerr << "*** List is empty ***\n";
        return;
    }
    if (pos < 0 || pos >= mySize)
    {
        cerr << "Illegal location to delete -- " << pos
            << ". List unchanged. ***\n";
        return;
    }

    // Shift array elements left to close the gap
    for(int i = pos; i < mySize; i++)
        myArray[i] = myArray[i + 1];

    // Decrease list size
    mySize--;
}

8.
/* QueueT.h contains the declaration of class template Queue.

Basic operations:
    Constructor: Constructs an empty queue
    empty: Checks if a queue is empty
    enqueue: Modifies a queue by adding a value at the back
    front: Accesses the front queue value; leaves queue unchanged
    dequeue: Modifies a queue by removing the value at the front
    display: Displays the queue elements from front to back

Class Invariant:
    1. The queue elements (if any) are stored in consecutive positions
       in myArray, beginning at position myFront.
    2. 0 <= myFront, myBack < QUEUE_CAPACITY
    3. Queue's size < QUEUE_CAPACITY
-----*/
#include <iostream>

#ifndef QUEUET
#define QUEUET

const int QUEUE_CAPACITY = 128;
template <typename QueueElement>

class Queue
{
public:
    //***** Function Members *****/
    //***** Constructor *****/
    Queue();
    /*-----
        Construct a Queue object.

    Precondition: None.

```

```
Postcondition: An empty Queue object has been constructed; myFront
    and myBack are initialized to -1 and myArray is an array with
    QUEUE_CAPACITY elements of type QueueElement.
-----*/
bool empty() const;
/*-----
   Check if queue is empty.

   Precondition: None.
   Postcondition: True is returned if the queue is empty and false is
       returned otherwise.
-----*/
void enqueue(const QueueElement & value);
/*-----
   Add a value to a queue.

   Precondition: value is to be added to this queue.
   Postcondition: value is added to back of queue provided there is
       space; otherwise, a queue-full message is displayed and
       execution is terminated.
-----*/
void display(ostream & out) const;
/*-----
   Output the values stored in the queue.

   Precondition: ostream out is open.
   Postcondition: Queue's contents, from front to back, have been output
       to out.
-----*/
QueueElement front() const;
/*-----
   Retrieve value at front of queue (if any).

   Precondition: Queue is nonempty.
   Postcondition: Value at front of queue is returned, unless queue
       is empty; in that case, an error message is displayed and a
       "garbage value" is returned.
-----*/
void dequeue();
/*-----
   Remove value at front of queue (if any).

   Precondition: Queue is nonempty.
   Postcondition: Value at front of queue has been removed, unless
       queue is empty; in that case, an error message is displayed
       and execution is terminated.
-----*/
private:
    ***** Data Members *****
    int myFront,
        myBack;
    QueueElement myArray[QUEUE_CAPACITY];
```

```
}; // end of class declaration

//--- Definition of Queue constructor
template <typename QueueElement>
inline Queue<QueueElement>::Queue()
: myFront(0), myBack(0)
{}

//--- Definition of empty()
template <typename QueueElement>
inline bool Queue<QueueElement>::empty() const
{
    return (myFront == myBack);
}

//--- Definition of enqueue()
template <typename QueueElement>
void Queue<QueueElement>::enqueue(const QueueElement & value)
{
    int newBack = (myBack + 1) % QUEUE_CAPACITY;
    if (newBack != myFront)      // queue isn't full
    {
        myArray[myBack] = value;
        myBack = newBack;
    }
    else
    {
        cerr << "**** Queue full -- can't add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n";
        exit(1);
    }
}

//--- Definition of display()
template <typename QueueElement>
inline void Queue<QueueElement>::display(ostream & out) const
{
    for (int i = myFront; i != myBack; i = (i + 1)%QUEUE_CAPACITY)
        out << myArray[i] << " ";
    cout << endl;
}

//--- Definition of front()
template <typename QueueElement>
QueueElement Queue<QueueElement>::front() const
{
    if ( !empty() )
        return (myArray[myFront]);
    else
    {
        cerr << "**** Queue is empty -- returning garbage value ***\n";
        QueueElement garbage;
        return garbage;
    }
}
```

```
//--- Definition of dequeue()
template <typename QueueElement>
void Queue<QueueElement>::dequeue()
{
    if ( !empty() )
        myFront = (myFront + 1) % QUEUE_CAPACITY;
    else
    {
        cerr << "*** Queue is empty -- "
            "can't remove a value ***\n";
        exit(1);
    }
}

#endif

9.
//CartesianPoint.h

#ifndef CARTESIANPOINT
#define CARTESIANPOINT

#include <iostream>
using namespace std;

template <typename CoordType>
class CartesianPoint
{
public:
    CartesianPoint();
    CartesianPoint(CoordType xVal, CoordType yVal);
    CoordType getX() const;
    CoordType getY() const;
    void setX(CoordType xVal);
    void setY(CoordType yVal);

private:
    CoordType myX, myY;
};

template<typename CoordType>
inline CartesianPoint<CoordType>::CartesianPoint()
: myX(0), myY(0)
{}

template <typename CoordType>
inline CartesianPoint<CoordType>::CartesianPoint(CoordType xVal,
                                                CoordType yVal)

: myX(xVal), myY(yVal)
{}

template <typename CoordType>
inline CoordType CartesianPoint<CoordType>::getX() const
{ return myX; }
```

```

template <typename CoordType>
inline CoordType CartesianPoint<CoordType>::getY() const
{ return myY; }

template <typename CoordType>
inline void CartesianPoint<CoordType>::setX(CoordType xVal)
{ myX = xVal; }

template <typename CoordType>
inline void CartesianPoint<CoordType>::setY(CoordType yVal)
{ myY = yVal; }

template <typename CoordType>
inline ostream & operator<<(ostream & out,
                           const CartesianPoint<CoordType> & point)
{
    out << "(" << point.getX() << ", " << point.getY() << ')';
    return out;
}

template <typename CoordType>
inline istream & operator>>(istream & in,
                             CartesianPoint<CoordType> & point)
{
    char punc;
    CoordType x, y;
    in >> punc >> x >> punc >> y >> punc;
    point.setX(x);
    point.setY(y);
    return in;
}

#endif

```

Exercises 9.4

1. number[0] = 0 number[1] = 0 number[2] = 1 number[3] = 1 number[4] = 2
number[5] = 2 number[6] = 3 number[7] = 3 number[8] = 4 number[9] = 4
2. w[0] = 0 w[1] = 0 w[2] = 0 w[3] = 0 w[4] = 0
w[5] = 0 w[6] = 0 w[7] = 0 w[8] = 0 w[9] = 0
w[10] = 0 w[11] = 0 w[12] = 1 w[13] = 1 w[14] = 2
w[15] = 2
3. number[0] = 99 number[1] = 33 number[2] = 44 number[3] = 88 number[4] = 22
number[5] = 11 number[6] = 55 number[7] = 66 number[8] = 77
4. number[0] = 0 number[1] = 1 number[2] = 2 number[3] = 3 number[4] = 0
number[5] = 1 number[6] = 2 number[7] = 3 number[8] = 4 number[9] = 5
5. number[0] = 33 number[1] = 33 number[2] = 88 number[3] = 88 number[4] = 11
number[5] = 11 number[6] = 66 number[7] = 66 number[8] = 77
6. number[0] = 99 number[1] = 33 number[2] = 44 number[3] = 88 number[4] = 22
number[5] = 11 number[6] = 55 number[7] = 66 number[8] = 99

```

7. number[0] = 77   number[1] = 33   number[2] = 44   number[3] = 88   number[4] = 22
   number[5] = 11   number[6] = 55   number[7] = 66   number[8] = 99

8. w[0] = 0          w[1] = 0          w[2] = 0          w[3] = 0          w[4] = 0
   w[5] = 0          w[6] = 0          w[7] = 0          w[8] = 0          w[9] = 0
   w[10] = 119        w[11] = 53        w[12] = 64        w[13] = 108        w[14] = 42
   w[15] = 31         w[16] = 75        w[17] = 86        w[18] = 97

9. v[0] = 20          v[1] = 20          v[2] = 20          v[3] = 20          v[4] = 20
   number[0] = 11        number[1] = 55        number[2] = 66        number[3] = 77

10. number0] = 22    number[1] = 11    number[2] = 55    number[3] = 66
    number[4] = 77

11. w[0] = 0          w[1] = 0          w[2] = 0          w[3] = 0          w[4] = 0
    w[5] = 0          w[6] = 0          w[7] = 0          w[8] = 0          w[9] = 0
    w[10] = 100        w[11] = 34        w[12] = 45        w[13] = 89        w[14] = 23
    w[15] = 12         w[16] = 56        w[17] = 67        w[18] = 78

12.
vector<int> v;
for (int i = 1; i < 100; i++)
    v.push_back(i);

13.
vector<int> v;
for (int i = 99; i > 0; i--)
    v.push_back(i);

14.
vector<bool> v(50);
for (int i = 0; i < 50; i++)
    v[i] = (i % 2 == 0);

15.
template <typename T>
bool ascend(const vector<T> & v)
{
    for (int index = 0; index < v.size() - 1; index++)
        if (v[index] >= v[index+1])
            return false;

    return true;
}

16.
template <typename T>
//operators "=" "<" & "-" must be defined for T
T range(const vector<T> & v)
{
    T min = v[0];
    T max = v[0];

```

```

for (int index = 1; index < v.size(); index++)
    if (v[index] < min)
        min = v[index];
    else if (v[index] > max)
        max = v[index];

    return max - min;
}

```

Exercises 9.6

Note: In #1-8, Table is defined by: `typedef vector< vector<double> > Table;`

1.

```

#include <cassert>
using namespace std;

double rowSum(unsigned row, const Table & aTable)
/*-----
   Sum the elements of a given row in a Table object.

   Receive: row, a row index and aTable, a Table object
   Return: the sum of the elements in the row-th row of aTable
-----*/
{
    assert(row < aTable.size());

    double sum = 0.0;
    for (int col = 0; col < aTable[row].size(); col++)
        sum += aTable[row][col];

    return sum;
}

```

2.

```

#include <cassert>
using namespace std;

double columnSum(unsigned row, const Table & aTable)
/*-----
   Sum the elements of a given column in a Table object.

   Receive: col, a column index and aTable, a Table object
   Return: the sum of the elements in the col-th column of aTable
-----*/
{
    assert(col < aTable[0].size());

    double sum = 0.0;
    for (int row = 0; row < aTable.size(); row++)
        sum += aTable[row][col];

    return sum;
}

```

3.

```
#include <cassert>
using namespace std;

double rowAverage(unsigned row, const Table & aTable)
/*-----
   Average the elements of a given row in a Table object.

   Receive: row, a row index and aTable, a Table object
   Return: the average of the elements in the row-th row of aTable
-----*/
{
    assert(row < aTable.size());
    int numValues = aTable[row].size();
    if (numValues > 0)
        return rowSum(row, aTable) / numValues ;
    //else
    cerr << "Row is empty -- returning 0:\n";
    return 0;
}
```

4.

```
#include <cassert>
#include <cmath>
using namespace std;

double rowStdDeviation((unsigned row, const Table & aTable)
/*-----
   Find the standard deviation of the elements of a given row in a
   Table object.

   Receive: row, a row index and aTable, a Table object
   Return: the standard deviation of the elements in the row-th row
   of aTable
-----*/
{
    assert(row < aTable.size());
    int numValues = aTable[row].size();
    if (numValues > 0)
    {
        double mean = rowAverage(row, aTable),
               sum = 0.0;

        for (int j = 0; j < numValues; j++)
            sum += pow(aTable[row][j] - mean, 2);

        return sqrt(sum / numValues);
    }
    //else
    cerr << "Row is empty -- returning 0:\n";
    return 0;
}
```

5.

```
#include <cassert>
using namespace std;

double columnAverage(unsigned col, const Table & aTable)
/*-----
   Average the elements of a given column in a Table object.

   Receive: col, a column index and aTable, a Table object
   Return: the average of the elements in the col-th column of aTable
-----*/
{
    assert(col < aTable[0].size());
    int numValues = aTable.size();
    if (numValues > 0)
        return ColumnSum(col, aTable) / numValues;
    //else
    cerr << "Column is empty -- returning 0:\n";
    return 0;
}
```

6.

```
#include <cassert>
#include <cmath>
using namespace std;
double columnStdDeviation(unsigned col, const Table & aTable)
/*-----
   Find the standard deviation of the elements of a given column
   in a Table object.

   Receive: col, a column index and aTable, a Table object
   Return: the standard deviation of the elements in the col-th
           column of aTable
-----*/
double columnStdDeviation(unsigned col, const Table & aTable)
{
    assert(col < aTable[0].size());
    int numValues = aTable.size();
    if (numValues > 0)
    {
        double mean = columnAverage(row, aTable),
               sum = 0.0;

        for (int i = 0; i < numValues; i++)
            sum += pow(aTable[i][col] - mean, 2);

        return sqrt(sum / numValues);
    }
    //else
    cerr << "Column is empty -- returning 0:\n";
    return 0;
}
```

7.

```
// --- Matrix.h
#include <vector>

#ifndef MATRIX
#define MATRIX

class Matrix
{
public:
    //--- Constructor of rows x cols matrix
    Matrix(int rows = 0, int cols = 0);

    //--- Input/Output
    void read(istream & in);
    void display(ostream & out);

    //--- Addition and multiplication
    Matrix operator+(const Matrix & b);
    Matrix operator*(const Matrix & b);

private:
    int myRows, myCols;
    vector<vector<double> > myData;
};

#endif

// --- Matrix.cpp

#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
#include "Matrix.h"

//--- Definition of constructor
Matrix::Matrix(int rows = 0, int cols = 0)
{
    vector<vector<double> > temp(rows, vector<double>(cols, 0.0));
    myData = temp;
    myRows = rows;
    myCols = cols;
}

//--- Definition of read()
void Matrix::read()
{
    cout << "Number of rows and columns? ";
    cin >> myRows >> myCols;

    cout << "Enter elements rowwise:\n";
    double value;
```

```
for (int i = 0; i < myRows; i++)
{
    vector<double> aRow;
    for (int j = 0; j < myCols; j++)
    {
        cin >> value;
        aRow.push_back(value);
    }
    myData.push_back(aRow);
}

//-- Definition of display()
void Matrix::display()
{
    for (int row = 0; row < myRows; row++)
    {
        for (int col = 0; col < myCols; col++)
            cout << setw(6) << myData[row][col];
        cout << endl;
    }
}

//-- Definition of operator+()
Matrix Matrix::operator+(const Matrix& b)
{
    assert((myRows == b.myRows && myCols == b.myCols));
    Matrix temp(myRows, myCols);

    for (int row = 0; row < myRows; row++)
        for (int col = 0; col < myCols; col++)
            temp.myData[row][col] = myData[row][col] + b.myData[row][col];
    return temp;
}

//-- Definition of operator*()
Matrix Matrix::operator*(const Matrix & b)
{
    assert(myCols == b.myRows);
    Matrix temp(myRows, b.myCols);
    for (int row = 0; row < myRows; row++)
        for (int col = 0; col < b.myCols; rc++)
    {
        double sum = 0;
        for (int k = 0; k < myCols; k++)
            sum += myData[row][k] * b.myData[k][col];

        temp.myData[row][col] = sum;
    }
    return temp;
}
```

Exercises 9.8

Bitset Exercises

These are solutions to the exercises in the supplementary material on bitsets on the Internet
(see <http://cs.calvin.edu/books/c++/ds/2e/WebItems/Chapter09/Bitsets.pdf>)

1. 0101010101010101010101
2. 00110101000101000101
3. 001000000000000000000000
4. 01110101010101010101
5. 01000000010000010000
6. 1111111111111111111111
7. 000000000000000000000000
8. 001110000000
9. 000000000000
10. 000000001010
11. 111111111111
12.

```
#include <iostream>
#include <bitset>
using namespace std;

template<int SetRange>
class Set
{
public:
    Set();
    Set union(const Set & B);
    Set intersection(const Set & B);
    Set complement();

    void add(int);
    void remove(int);
    bool member(int x);
    bool subset(const Set & B);

    void display();
}
```

```
private:  
    bitset<SetRange> myData;  
    int myCardinality;  
};  
  
//-- Definition of Constructor  
template<int SetRange>  
Set<SetRange>::Set()  
{  
    myData.reset(); // all bits set to 0, indicating empty set  
    myCardinality = 0;  
}  
  
//-- Definition of union()  
template<int SetRange>  
Set<SetRange> Set<SetRange>::union(const Set<SetRange> & B)  
{  
    Set<SetRange> temp;  
    temp.myData = myData | B.myData;  
    temp.myCardinality = temp.myData.count();  
    return temp;  
}  
  
//-- Definition of intersection()  
template<int SetRange>  
Set<SetRange> Set<SetRange>::intersection(const Set<SetRange> & B)  
{  
    Set<SetRange> temp;  
    temp.myData = myData & B.myData;  
    temp.myCardinality = temp.myData.count();  
    return temp;  
}  
  
//-- Definition of complement()  
template<int SetRange>  
Set<SetRange> Set<SetRange>::complement()  
{  
    Set<SetRange> temp;  
    temp.myData = myData.flip();  
    temp.myCardinality = temp.myData.count();  
    return temp;  
}  
  
//-- Definition of member()  
template<int SetRange>  
bool Set<SetRange>::member(int x)  
{  
    return myData.test(x-1);  
}  
  
//-- Definition of add()  
template<int SetRange>  
void Set<SetRange>::add(int x)  
{  
    myData.set(x-1, 1);  
}
```

```
///-- Definition of remove()
template<int SetRange>
void Set<SetRange>::remove(int x)
{
    myData.set(x-1, 0);
}

///-- Definition of subset()
template<int SetRange>
bool Set<SetRange>::subset(const Set<SetRange> & B)
{
    return (myData & B.myData) == myData;
}

///-- Definition of display()
template<int SetRange>
void Set<SetRange>::display()
{
    for (int i = 0; i < myData.size(); i++)
        if (myData[i])
            cout << i + 1 << " ";
    cout << endl;
}
```

Valarray Exercises

These are solutions to the exercises in the supplementary material on valarrays on the Internet
(see <http://cs.calvin.edu/books/c++/ds/2e/WebItems/Chapter09/Valarrays.pdf>)

1.

```
#include <valarray>
using namespace std;

void display(const valarray<double> & data)
{
    for (int index = 0; index < data.size(); index++)
        cout << data[index] << " ";
    cout << endl;
}
```

2.

```
#include <valarray>
using namespace std;

void read(valarray<double> & data)
{
    cout << "Input your " << data.size()
        << " real numbers: " << endl;

    for (int index = 0; index < data.size(); index++)
        cin >> data[index];
}
```

3.

```
#include <valarray>
#include <cmath>
using namespace std;

double magnitude(const valarray<double> & data)
{
    double sum = 0.0;
    for (int index = 0; index < data.size(); index++)
        sum += data[index]* data[index];
    return sqrt(sum);
}
```

4.

```
#include <valarray>
using namespace std;

double dotProduct(const valarray<double> & a, const valarray<double> & b)
{
    double sum = 0.0;

    for (int index = 0; index < a.size(); index++)
        sum += a[index]* b[index];

    return sum;
}
```

Chapter 10: ADT Implementations: Recursion, Algorithm Analysis, and Standard Algorithms

Programming Problems

Section 10.1

1.

```
/*
 *-----*
 Driver program to test recursive digit-counting function of
 Exercise 21.

 Input: Nonnegative integers
 Output: Number of digits in each integer
 -----*/
```

```
#include <iostream>
using namespace std;

int numDigits(unsigned n);
/*
 *-----*
 Recursively count the digits in a nonnegative integer -- Exer. 21.

 Precondition: n >= 0
 Postcondition: Number of digits in n is returned.
 -----*/
```

```
int main()
{
    int n;
    for (;;)
    {
        cout << "Enter a nonnegative integer (-1 to stop): ";
        cin >> n;
        if (n < 0) break;
        cout << "# digits = " << numDigits(n) << endl;
    }
}

//-- Definition of numDigits()
int numDigits(unsigned n)
{
    if (n < 10)
        return 1;
    else
        return 1 + numDigits(n / 10);
}
```

- Just replace the function definition in 1 with that in Exercise 22.

3.

```
/*
-----*
Driver program to test reverse-print function of Exercise 23.

Input: Nonnegative integers
Output: The reversal of each integer.
-----*/
```

```
#include <iostream>
using namespace std;

void printReverse(unsigned n);
/*
-----*
Recursively display the digits of a nonnegative integer in reverse
order -- Exer. 23.

Precondition: n >= 0
Postcondition: Reversal of n has been output to cout.
-----*/
```

```
int main()
{
    int n;
    for (;;)
    {
        cout << "\nEnter a nonnegative integer (-1 to stop): ";
        cin >> n;
        if (n < 0) break;
        cout << "Reversal is: ";
        printReverse(n);
    }
}

//-- Definition of printReverse()
void printReverse(unsigned n)
{
    if (n < 10)
        cout << n << endl;
    else
    {
        cout << n % 10;
        printReverse(n / 10);
    }
}
```

4. Just replace the function definition in 3 with that in Exercise 24.

5.

```
/*
-----*
Driver program to test power function of Exercise 25.

Input: Integer exponents and real bases
Output: Each real to that integer power
-----*/
```

```
#include <iostream>
using namespace std;

double power(double x, int n);
/*
-----*
Recursively compute integer powers of real numbers -- Exer. 25.

Precondition: None
Postcondition: n-th power of x is returned.
-----*/
```

```
int main()
{
    double base;
    int exp;
    for (;;)
    {
        cout << "\nEnter a real base and an integer exponent (0 0 to stop): ";
        cin >> base >> exp;

        if (base == 0 && exp == 0) break;

        cout << base << '^' << exp << " = " << power(base, exp) << endl;
    }
}

//-- Definition of power()
double power(double x, int n)
{
    if (n == 0)
        return 1;

    else if (n < 0)
        return power(x, n + 1) / x;

    else
        return power(x, n - 1) * x;
}
```

6. Just replace the function definition in 5 with that in Exercise 26.

7-9.

```
/*
Driver program to test the recursive array reversal, array sum,
and array location functions of Exercise 27-29.

Input: Integer exponents and real bases
Output: Each real to that integer power
-----*/
```

```
#include <iostream>
using namespace std;
```

```
typedef int ElementType;
const int CAPACITY = 100;
typedef ElementType ArrayType[CAPACITY];
```

```
void reverseArray(ArrayType arr, int first, int last);
/*
Recursively reverse an array -- Exer. 27.

Precondition: Array arr has elements in positions first through
last.
Postcondition: Elements in positions first through last of arr
are reversed.
-----*/
```

```
ElementType sumArray(ArrayType arr, int n);
/*
Recursively sum the elements of an array -- Exer. 28.

Precondition: Array arr has n elements.
Postcondition: Sum of the elements is returned
-----*/
```

```
int location(ArrayType arr, int first, int last, ElementType item);
/*
Recursively search the elements of an array -- Exer. 28.

Precondition: Array arr has elements in positions first through
last.
Postcondition: Location of item in arr is returned; -1 if item
isn't found.
-----*/
```

```
void display(ArrayType arr, int n);
/*
Display the elements of an array.

Precondition: Array arr has n elements.
Postcondition: Elements of arr have been output to cout.
-----*/
```

```
int main()
{
    ArrayType x;

    cout << "Enter at most " << CAPACITY << " integers (-1 to stop):\n";
    int item, count;
    for (count = 0; count < CAPACITY; count++)
    {
        cin >> item;
        if (item < 0) break;
        x[count] = item;
    }

    cout << "Original array: ";
    display(x, count);

    reverseArray(x, 0, count - 1);

    cout << "Reversed array: ";
    display(x, count);

    cout << "\nSum of array elements = " << sumArray(x, count) << endl;

    ElementType toFind;
    for(;;)
    {
        cout << "Enter an item to search for (-1 to stop): ";
        cin >> toFind;
        if (toFind < 0) break;

        cout << "Location of item = " << location(x, 0, count - 1, toFind)
            << endl << "where -1 denotes item not found)\n";
    }
}

//-- Definition of reverseArray()
void reverseArray(ArrayType arr, int first, int last)
{
    if (first < last)
    {
        ElementType temp = arr[first];
        arr[first] = arr[last];
        arr[last] = temp;
        reverseArray(arr, first + 1, last - 1);
    }
}

//-- Definition of sumArray()
ElementType sumArray(ArrayType A, int n)
{
    if (n == 0)
```

```
        return 0;
    if (n == 1)
        return A[0];
    else
        return A[n - 1] + sumArray(A, n - 1);
}

//-- Definition of location()
int location(ArrayType arr, int first, int last, ElementType item)
{
    if (first == last && item != arr[first])
        return -1;

    else if (item == arr[last])
        return last;

    else
        return location(arr, first, last-1, item);
}

//-- Definition of display()
void display(ArrayType arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

10.
/*
-----*
   Driver program to test the recursive string reversal function of
   Exercise 30.

   Input: A string
   Output: The reversal of the string
-----*/
#include <iostream>
#include <string>
using namespace std;

string reverse(string word);
/*
-----*
   Recursively reverse a string -- Exer. 30.

   Precondition: None
   Postcondition: String word has been reversed.
-----*/
int main()
{
    string s;
```

```

cout << "Enter a string: ";
getline(cin, s);

cout << "Original string: " << s << endl;
cout << "Reversed string: " << reverse(s) << endl;
}

//-- Definition of reverse()
string reverse(string word)
{
    if (word.length() == 1)
        return word;
    else
        return reverse(word.substr(1, word.length() - 1)) + word[0];
}

```

11. Simply replace the definition of `reverse()` in Problem 10 with the nonrecursive version from Exercise 31.

12.

```

/*-----
   Driver program to test the recursive palindrome checking function
   of Exercise 32.

   Input: An integer
   Output: An indication of whether the integer is a palindrome
-----*/
#include <iostream>
#include <cmath>
using namespace std;

bool palindrome (unsigned number, int numDigits);
/*-----
   Recursively check if an integer is a palindrome -- Exer. 32.

   Precondition: number has numDigits digits.
   Postcondition: True is returned if number is a palindrome, and
   false otherwise.
-----*/

int main()
{
    int x, n;
    cout << "Enter integer & number of digits: ";
    cin >> x >> n;
    cout << "Palindrome? " << (palindrome(x, n) ? "Yes" : "No") << endl;
}

//-- Definition of palindrome()

```

```

bool palindrome (unsigned number, int numDigits)
{
    if (numDigits <= 1)
        return true;
    else
    {
        unsigned powerOf10;
        powerOf10 = (unsigned)pow(10.0, numDigits - 1);
        unsigned firstDigit = number / powerOf10 ;
        unsigned lastDigit = number % 10;

        if (firstDigit != lastDigit)
            return false;
        else
            return palindrome((number % powerOf10) / 10, numDigits - 2);
    }
}

```

13.

```

/*-----
   Driver program to test the recursive palindrome checking function
   of Exercise 32.

   Input: An integer
   Output: An indication of whether the integer is a palindrome
-----*/
#include <iostream>
using namespace std;

int gcd(int a, int b);
/*-----
   Recursively compute gcd of two integers -- Exer. 33.

   Precondition: Not both of a and b are zero.
   Postcondition: GCD of a and b is returned.
-----*/

int main()
{
    int a, b;
    for (;;)
    {
        cout << "\nEnter two integers (0 0 to stop): ";
        cin >> a >> b;
        if (a == 0 && b == 0) break;
        cout << "GCD = " << gcd(a, b) << endl;
    }
}

//-- Definition of gcd()
int gcd(int a, int b)
{

```

```

if (a < 0)
    a = -a;
if (b < 0)
    b = -b;

if (b == 0)
    return a;
else
    return gcd(b, a % b);
}

```

14. Simply replace the recursive function `gcd()` in Problem 13 with the nonrecursive version from Exercise 34.

15.

```

/*-----
Driver program to test the recursive binomial coefficient function
of Exercise 35.

Input: Pairs of integer
Output: The binomial coefficient corresponding to that pair.
-----*/
#include <iostream>
using namespace std;

int recBinomCoef(int n, int k);
/*-----
Recursively compute a binomial coefficient -- Exer. 35.

Precondition: 0 <= k <= n && n != 0.
Postcondition: The binomial coefficient "n above k" is returned.
-----*/

int main()
{
    int a, b;
    for (;;)
    {
        cout << "\nEnter two integers with first >= second (0 0 to stop): ";
        cin >> a >> b;
        if (a == 0 && b == 0) break;

        if (a >= b)
            cout << "C(" << a << ", " << b << ") = " << recBinomCoef(a, b) <<
        endl;
    }
}

//-- Definition of binomCoeff()
int recBinomCoef(int n, int k)

```

```

{
    if (n == k || k == 0)
        return 1;
    else
        return recBinomCoef(n - 1, k - 1) + recBinomCoef(n - 1, k);
}

16.
/*
-----*
Driver program to time the recursive binomial coefficient function
of Exercise 35 and the nonrecursive version of Exercise 36.

Input: Pairs of integer
Output: The binomial coefficient corresponding to that pair and
the times to compute it both recursively and
nonrecursively.
NOTE: Binomial coefficients are being computed as doubles because
large ones need to be computed to achieve nonzero times,
and integer values will overflow.
-----*/
#include <iostream>
using namespace std;
#include "Timer.h"

double recBinomCoef(int n, int k);
/*
-----*
Recursively compute a binomial coefficient -- Exer. 35.

Precondition: 0 <= k <= n && n != 0.
Postcondition: The binomial coefficient "n above k" is returned.
-----*/
double nonrecBinomCoef(int n, int k);
/*
-----*
Iteratively compute a binomial coefficient -- Exer. 36.

Precondition: 0 <= k <= n && n != 0.
Postcondition: The binomial coefficient "n above k" is returned.
-----*/
int main()
{
    int a, b;
    for (;;)
    {
        cout << "\nEnter two integers with first >= second (0 0 to stop): ";
        cin >> a >> b;
        if (a == 0 && b == 0) break;
        if (a >= b)
        {
            double recCoeff,           // recursive binom coeff.

```

```
    nonRecCoeff;      // use double due to integer overflor
Timer tRec, tNonrec;
tRec.start();
recCoeff = recBinomCoef(a, b);
tRec.stop();

tNonrec.start();
nonRecCoeff = nonrecBinomCoef(a, b);
tNonrec.stop();

// Display the times
cout << "C(" << a << ", " << b << ") = " << recCoeff << endl;
cout << "*** Recursive binom. coeff. took: ";
tRec.print(cout);
cout << endl;

cout << "C(" << a << ", " << b << ") = " << nonRecCoeff << endl;
cout << "*** Nonrecursive binom. coeff. took: ";
tNonrec.print(cout);
cout << endl;
}

}

//-- Definition of recBinomCoeff()
double recBinomCoef(int n, int k)
{
    if (n == k || k == 0)
        return 1;
    else
        return recBinomCoef(n - 1, k - 1) + recBinomCoef(n - 1, k);
}

//-- Definition of nonRecBinomCoeff()
double nonrecBinomCoef(int n, int k)
{
    double num = n, denom = 1;
    if (k > n/2)
        k = n - k;
    for (int i = n-1; i >= k + 1; i--)
        num *= i;
    for (int i = 2; i <= n-k; i++)
        denom *= i;

    return num/denom;
}
```

17.

```
/*
-----*
Program to trace recursive calls of a function. Here the function
f() of Exercise 13 is used.

Input: An integer
Output: Trace of recursive calls and returns and the value of
f at that integer
-----*/
```

```
#include <iostream>           // cin, cout, >>, <<
using namespace std;
```

```
void indent(int numSpaces);
/*
-----*
Indent the output numSpaces spaces from current position.

Precondition: None
Postcondition: Output has advanced spaces.
-----*/
```

```
unsigned f(unsigned n);
/*
-----*
Modification of function f of Exercise 13 -- statements are added
to output a trace of the recursive calls to and returns from f().

Precondition: None
Postcondition: Trace of recursive calls and returns was output to
cout and the value of f returned.
-----*/
```

```
int main()
{
    int number;

    cout << "Enter a positive integer: ";
    cin >> number;
    f(number);
}
```

```
//-- Definition of indent()
void indent(int numSpaces)
{
    while (numSpaces--)
        cout << "    ";
}
```

```
//-- Definition of f()
unsigned f(unsigned n)
{
    static int level = 0;      // static variable to control indentation
    if (n < 2)
    {
        indent(level);
```

```
    cout << "f(" << n << ") returns 0\n";
    return 0;
}
else
{
    indent(level);
    cout << "f(" << n << ") = 1 + f(" << (n / 2) << ")\n";
    level++;
    unsigned value = 1 + f(n / 2);
    level--;
    indent(level);
    cout << "f(" << n << ") returns " << value << '\n';
    return value;
}
}

18.
/*
-----*
Program to trace recursive calls of the function printReverse()
of Exercise 23.

Input: An integer
Output: Trace of recursive calls and returns and the reversal of
that integer.
-----*/
#include <iostream>      // cin, cout, >>, <<
using namespace std;

int level = 0;          // global variable to control indentation

void indent(int numSpaces);
/*
-----*
Indent the output numSpaces spaces from current position.

Precondition: None
Postcondition: Output has advanced spaces.
-----*/
void printReverse(unsigned n);
/*
-----*
Recursively display the digits of a nonnegative integer in reverse
order -- Exer. 23.

Precondition: n >= 0
Postcondition: Reversal of n has been output to cout.
-----*/

int main()
{
    int number;

    cout << "Enter a positive integer: ";
    cin >> number;
    printReverse(number);
}
```

```

//-- Definition of indent()
void indent(int numSpaces)
{
    while (numSpaces--)
        cout << "    ";
}

//-- Definition of printReverse()
void printReverse(unsigned number)
{
    static int level = 0;      // static variable to control indentation
    indent(level);
    cout << "printReverse(" << number << "): Output ";
    cout << number % 10;           // output the rightmost digit

    int leftDigits = number / 10;     // leftmost part of Number

    if (leftDigits)                 // inductive step:
    {
        cout << ", then call printReverse(" << leftDigits << ").\n";
        level++;
        printReverse(leftDigits);   // ... output the rest recursively
        level--;
    }
    else                           // anchor case:
    {
        cout << " and \n.";
        cout << endl;             // ... generate a new line
    }

    indent(level);
    cout << "printReverse(" << number << ") returns.\n";
}

```

19.

```

/*
-----*
Program to display the lyrics of the song "Bingo."
Output: lyrics of "Bingo"
-----*/

```

```

#include <iostream>                      // cin, cout, <<, >>
using namespace std;

void printSong(int verseNum);
/*
-----*
Display the lyrics of the song "BINGO" beginning with a given
verse.

Precondition: None
Postcondition: The song beginning with verse verseNum.

```

```
-----*/
```

```
int main()
{
    printSong(1);
}

//-- Definition of printSong()
void printSong(int verseNum)
{
    const char BINGO[] = "BINGO";

    cout << "There was a farmer had a dog;\n"
        "And Bingo was his name-o.\n";

    if (verseNum > 1)
    {
        cout << "(Clap";
        for (int i = 2; i < verseNum; i++)
            cout << ", clap";
        cout << (verseNum < 6 ? "-" : "\n");
    }

    for (int i = verseNum; i <= 5; i++)
        cout << BINGO[i-1] << (i < 5 ? "-" : "\n");

    cout << "And Bingo was his name-o!\n\n";

    if (verseNum < 6)
        printSong(verseNum + 1);
}
```

20.

```
/*-----
Driver program to test a recursive function to display a number
with commas.

Input: An integer
Output: That integer with commas in correct locations
-----*/
```

```
#include <iostream>           // cin, cout, >>, <<
using namespace std;

void printNumberWithCommas(long int number);
/*-----
Display a number with commas in the appropriate locations.

Precondition: None
Postcondition: number, with commas, has been output to cout.
-----*/
```

```
int main()
{
    long int n;

    for (;;)
    {
        cout << "\nEnter an integer n (negative to stop): ";
        cin >> n;
        if (n < 0) break;

        printNumberWithCommas(n);
    }
}

//-- Definition of printNumberWithCommas()
void printNumberWithCommas(long int number)
{
    if (number < 1000)
        cout << number;
    else
    {
        printNumberWithCommas(number / 1000);
        cout << ','
            << (number % 1000) / 100
            << (number % 100) / 10
            << (number % 10);
    }
}
```

21.

```
/*-- BlobGrid.h --
*-----*
     Interface for class BlobGrid
-----*/
```

```
#include <iostream>           // istream, ostream
#include <vector>             // vector<T>
using namespace std;

typedef vector<char> CharRow;
typedef vector<CharRow> CharTable;

class BlobGrid
{
public:
    BlobGrid(unsigned rows, unsigned columns);
/*-----
     Constructs an empty BlobGrid with specified rows and columns.

     Precondition: None
     Postcondition: The constructed BlobGrid has myRows == rows,
                    myColumns == columns, and myGrid is a rows x columns array.
-----*/
```

```
void readGrid(istream & in);
/*-----
   Input a BlobGrid.

   Precondition: istream in is open.
   Postcondition: BlobGrid object has been input and removed from in.
-----*/

void displayGrid(ostream & out) const;
/*-----
   Output a BlobGrid.

   Precondition: ostream out is open.
   Postcondition: BlobGrid object has been output to out.
-----*/

unsigned eatAllBlobs();
/*-----
   Remove all blobs from a BlobGrid.

   Precondition: None
   Postcondition: All blobs have been removed from myGrid and the
                  number of blobs removed is returned.
-----*/

void eatOneBlob(unsigned row, unsigned col);
/*-----
   A recursive function used internally by eatBlobs().

   Precondition: None
   Postcondition: The blob in the specified row and column col has
                  been removed from myGrid.
-----*/

private:
    unsigned myRows,
            myColumns;
    CharTable myGrid;
};

//-- Definition of constructor
inline BlobGrid::BlobGrid(unsigned rows, unsigned columns)
: myRows(rows), myColumns(columns)
{
    CharTable temp(rows, CharRow(columns, '.'));  

    myGrid = temp;
}

inline istream & operator>>(istream & in, BlobGrid & b)
/*-----
   Input operator.

   Precondition: istream in is open.
   Postcondition: BlobGrid object has been input and removed from in
-----*/
```

```
    and in is returned.  
-----*/  
{  
    b.readGrid(in);  
    return in;  
}  
  
inline ostream & operator<<(ostream & out, const BlobGrid & b)  
/*-----  
    Output operator.  
  
    Precondition: ostream out is open.  
    Postcondition: BlobGrid object has been output to out and out is  
        returned.  
-----*/  
{  
    b.displayGrid(out);  
    return out;  
}  
  
//-- BlobGrid.cpp --  
/*-----  
    Implements the operations for class BlobGrid.  
-----*/  
  
#include <iostream>  
using namespace std;  
  
#include "BlobGrid.h"  
  
//-- Definition of readGrid()  
void BlobGrid::readGrid(istream & in)  
{  
    char ch;  
  
    for (int row = 0; row < myRows; row++)  
        for (int col = 0; col < myColumns; col++)  
            in >> myGrid[row][col];  
}  
  
//-- Definition of displayGrid()  
void BlobGrid::displayGrid(ostream& out) const  
{  
    for (int row = 0; row < myRows; row++)  
    {  
        for (int col = 0; col < myColumns; col++)  
            out << myGrid[row][col] << ' ';  
  
        out << endl;  
    }  
}  
  
//-- Definition of eatAllBlobs()
```

```
unsigned BlobGrid::eatAllBlobs()
{
    unsigned number = 0;

    for (int row = 0; row < myRows; row++)
    {
        for (int col = 0; col < myColumns; col++)
        {
            if (myGrid[row][col] == '*')
            {
                eatOneBlob(row, col);
                number++;
            }
        }
    }

    return number;
}

//-- Definition of eatOneBlob()
void BlobGrid::eatOneBlob(unsigned row, unsigned col)
{
    if ( (row >= myRows) || (col >= myColumns) )
        return;

    if (myGrid[row][col] != '*')
        return;

    myGrid[row][col] = '.';

    eatOneBlob(row-1, col);
    eatOneBlob(row+1, col);
    eatOneBlob(row, col-1);
    eatOneBlob(row, col+1);
}

/*
-----*
   Driver program to test class BlobGrid.

   Input: size of grid and locations of '*'s
   Output: The grid and the number of blobs in it.
-----*/
#include <iostream>           // cin, cout, >>, <<
using namespace std;
#include "BlobGrid.h"

int main()
{
    unsigned numRows, numColumns;

    cout << "Enter number of rows and number of columns in grid: ";
    cin >> numRows >> numColumns;
```

```
BlobGrid blobs(numRows, numColumns);

cout << "\nEnter " << numRows << " x " << numColumns
    << " grid of *'s and .'s:\n";
cin >> blobs;

cout << "\nYou entered:\n" << blobs << endl;

cout << blobs.eatAllBlobs() << " blob(s) found.\n";
}
```

Sample Execution:

Enter number of rows and number of columns in grid: 5 10

Enter 5 x 10 grid of '*'s and '.'s:

You entered:

5 blob(s) found.

22.

```
/*
Program that uses a recursive function to count the number of
northeast paths from one point to another in a rectangular grid.

Input: location of B relative to A
Output: the number of northeast paths from A to B
*/
#include <iostream>
using namespace std;

int northeastPaths(int pointsNorth, int pointsEast);
/*
Compute the number of northeast paths from the origin (lower left
corner) to a destination in a grid of points.

Precondition: None.
Postcondition: The number of northeast paths to the destination
pointsNorth units north and pointsEast units east is returned.
*/
int main()
{
    int pointsNorth,
        pointsEast;

    cout << "\nHow many points north of A is B? ";
    cin >> pointsNorth;

    cout << "How many points east of A is B? ";
    cin >> pointsEast;

    cout << "\nThere are " << northeastPaths(pointsNorth, pointsEast)
```

```
    << " northeast paths between A and B.\n";
}

//-- Definition of northeastPaths()
int northeastPaths(int pointsNorth, int pointsEast)
{
    if ( (pointsNorth < 0) || (pointsEast < 0) )
        return 0;
    else if ( (pointsNorth == 0) && (pointsEast == 0) )
        return 1;
    else
        return northeastPaths(pointsNorth - 1, pointsEast) +
               northeastPaths(pointsNorth , pointsEast - 1);
}
```

23.

```
/*
-----*
Program to convert nonnegative integers from base-ten to another
base.

Input: Numbers to be converted and bases
Output: Representations of the numbers in the specified bases.
-----*/
```

```
#include <iostream>
using namespace std;

void convert(unsigned n, unsigned b);
/*
-----*
Recursive function to convert base-ten integers to another base.

Precondition: 2 <= b <= 10.
Postcondition: The base-b representation of n is returned.
-----*/
```

```
int main()
{
    unsigned n, b;
    for (;;)
    {
        cout << "\nEnter number and base (0 0 to stop): ";
        cin >> n >> b;
        if ( b == 0 ) break;

        convert(n, b);
        cout << endl;
    }
}

//-- Definition of convert()
void convert(unsigned n, unsigned b)
{
    if (b < 2 || b > 10)
    {
        cerr << "Only bases in the range 2 - 10 are allowed.\n";
    }
}
```

```
    return;
}

int rem = n % b;
if (n / b != 0)
    convert(n / b, b);
cout << rem;
}

24.
/*-----
   Program to display the prime factorization of a number and
   indicate if it is prime.

   Input: a number
   Output: its prime factorization
-----*/
#include <iostream>           // cin, cout, >>, <<
using namespace std;

bool printPrimeFactorizationOf(long int number);
/*-----
   Recursive function to display the prime factorization of a number
   with prime factors in descending order and check if it is a prime.

   Precondition: number > 0.
   Postcondition: The prime factors of number are displayed in
       descending order and true is returned if number is a prime,
       false if not.
-----*/
int main()
{
    long number;
    for (;;)
    {
        cout << "\nPlease enter a number > 2 to be factored (0 to stop): ";
        cin >> number;

        if (number < 1) break;

        cout << "\nFactorization:\n";

        if (printPrimeFactorizationOf(number))
            cout << " is a prime!";
        cout << endl;
    }
}

//-- Definition of printPrimeFactorizationOf()
bool printPrimeFactorizationOf(long int number)
{
```

```
long int divisor = 2;

while (number % divisor != 0)
    divisor++;

if (number != divisor)
{
    printPrimeFactorizationOf(number / divisor);
    cout << " * " << divisor;
    return false;
}
else
{
    cout << divisor;
    return true;
}
}
```

25.

```
/*
Program to generate permutations.

Input: number of elements to permute
Output: recursively-generated permutations of {1, 2, ... , number}
-----*/
#include <iostream>
using namespace std;

const int MAX_NUMBER = 10;      //limit on number -- max of 10! perms
typedef int ArrayType[MAX_NUMBER + 1];

void printAPerm(ArrayType p);
/*
Display one of the permutations.

Precondition: Array p stores the permutation.
Postcondition: The permutation has been output to cout
-----*/
void permute(ArrayType p, int k, int n);
/*
Generate another permutation.

Precondition: 0 <= k < n
Postcondition: A new permutation has been generated by inserting
k into one of the permutations of 1, 2, . . . , k-1.
-----*/
int main()
{
    ArrayType intPerm;
    int number;

    do
    {
```

```
cout << "Enter number of elements to permute, at most "
    << MAX_NUMBER << ": ";
cin >> number;
}
while (number <=0 || number > MAX_NUMBER);

intPerm[0] = 0;
permute(intPerm, 1, number);
}

//-- Definition of printAPerm()
void printAPerm(ArrayType p)
{
    int index = 0;

    while (p[index] != 0)
    {
        cout << p[index] << " ";
        index = p[index]; //get next number in permutation
    }

    cout << endl;
}

//-- Definition of permute()
void permute(ArrayType p, int k, int n)
{
    int index = 0;
    do
    {
        p[k] = p[index];
        p[index] = k;
        if (k == n)
            printAPerm(p);
        else
            permute(p, k+1, n);
        p[index] = p[k]; // swap back
        index = p[index];
    }
    while (index != 0);
}
```

Sample Execution:

```
Enter number of elements to permute, at most 10: 3
3 2 1
2 3 1
2 1 3
3 1 2
1 3 2
1 2 3
```

Section 10.2

26.

```
/*
-----*
Program to give a "graphical" solution of the Towers of Hanoi
puzzle.

Input: the number of disks to be moved
Output: a sequence of "graphical" moves that solve the puzzle
-----*/
```

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector< vector<int> > peg(3);      // global pegs

vector< vector<int> > lastConfig; // configuration after last move

void move(int n, vector<int> & source,
          vector<int> & destination, vector<int> & spare);
/*
-----*
Display one move in solving the Towers of Hanoi puzzle.

Precondition: n is the number of disks to be moved; source
represents the peg containing the disk to move; destination
represents the peg where to move disk, and spare represents
a peg to store disks temporarily.
Postcondition: A "picture" describing the move has been output
to cout via display().
-----*/
```

```
void display(const vector< vector<int> > & peg);
/*
-----*
Display the elements of the n X 3 matrix representing the
three pegs and disks on them (represented by 1, 2, ..., n),
where n is the number of disks

Precondition: None
Postcondition: Elements of the matrix have been output to cout.
-----*/
```

```
int main()
{
    cout << "This program traces the solution of the Hanoi Towers puzzle.\n"
        "The disks are numbered 1, 2, 3, ... , from smallest to\n"
        "largest and the pegs are horizontal.\n\n";

    cout << "Enter the number of disks: ";
    int numDisks;                      // the number of disks to be moved
    cin >> numDisks;

    for (int i = numDisks; i > 0; i--) //set up source peg
        peg[0].push_back(i);
```

```
display(peg);

lastConfig = peg;
move(numDisks, peg[0], peg[1], peg[2]); // the solution
}

//-- Definition of move()
void move(int n, vector<int> & source,
          vector<int> & destination, vector<int> & spare)
{
    if (n <= 1)                                // anchor
    {
        destination.push_back(source.back());
        source.pop_back();
    }
    else                                         // inductive case
    {
        move(n-1, source, spare, destination);
        display(peg);
        move(1, source, destination, spare);
        display(peg);
        move(n-1, spare, destination, source);
        display(peg);
    }
}

//-- Definition of display()
void display(const vector<vector<int>> & peg)
{
    if (lastConfig == peg) return;

    //else
    for (int n = 0; n < 3; n++)
    {
        cout << "Peg " << n << ":" ;
        for (int i = 0; i < peg[n].size(); i++)
            cout << peg[n][i] << " ";
        cout << endl;
    }
    cout << endl;
    lastConfig = peg;
}
```

Sample Execution:

This program traces the solution of the Hanoi Towers puzzle.
The pegs are horizontal, and the pegs are numbered 1, 2, 3, ...
from smallest to largest.

```
Enter the number of disks: 4
Peg 0: 4 3 2 1
Peg 1:
Peg 2:
```

Peg 0: 4 3 2
Peg 1:
Peg 2: 1

Peg 0: 4 3
Peg 1: 2
Peg 2: 1

Peg 0: 4 3
Peg 1: 2 1
Peg 2:

Peg 0: 4
Peg 1: 2 1
Peg 2: 3

Peg 0: 4 1
Peg 1: 2
Peg 2: 3

Peg 0: 4 1
Peg 1:
Peg 2: 3 2

Peg 0: 4
Peg 1:
Peg 2: 3 2 1

Peg 0:
Peg 1: 4
Peg 2: 3 2 1

Peg 0:
Peg 1: 4 1
Peg 2: 3 2

Peg 0: 2
Peg 1: 4 1
Peg 2: 3

Peg 0: 2 1
Peg 1: 4
Peg 2: 3

Peg 0: 2 1
Peg 1: 4 3
Peg 2:

Peg 0: 2
Peg 1: 4 3
Peg 2: 1

Peg 0:
Peg 1: 4 3 2
Peg 2: 1

Peg 0:
Peg 1: 4 3 2 1
Peg 2:

27. This is similar to the class Expression described in Section 10.2, (the code for which is available on the Internet) and is a quite straightforward but substantial project.

28.

```
/*
-----*
Program to strip comments from C++ source using recursion as the
only repetition structure, except for an input loop, and reading
one character at a time without looking ahead.

Input(keyboard): the name of the file containing the source code
Input(infile): the characters in the file
Output(outfile): the contents of infile with all C++ comments
removed.
-----*/
#include <iostream>
#include <fstream>
#include <cassert>
#include <string>
using namespace std;

void strip(ifstream & fin, char delim, char commChar);
/*
-----*
Comment stripper.

Precondition: ifstream fin is open, delim is '/' or '*', and
commChar is the first character in the comment.
Postcondition: Characters until the end of the comment have been
read and removed from fin.
-----*/
int main()
{
    cout << "Enter name of file containing the source code. \n"
        "Note: ^ must be the last character in the file. ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.data());
    assert(fin.is_open());

    char ch, next;
    for(;;)
    {
        if (fin.eof()) break;
        fin.get(ch);
        if (ch == '^') break;

        if (ch == '/')
        {
            fin.get(ch);
            if (ch == '/' || ch == '*')
            {
                fin.get(next);
                strip(fin, ch, next);
            }
            else
                cout << '/' << ch;
        }
    }
}
```

```
    else
        cout << ch;
    }
    cout << endl;
    fin.close();
}

//-- Definition of strip()
void strip(ifstream & fin, char delim, char commChar)
{
    if (fin.eof()) return;
    char ch;
    bool consecStars = false;
    if (delim == '/') // inline comment
    {
        if (commChar != '\n')

        {
            fin.get(commChar);
            strip(fin, delim, commChar);
        }
        else
        {
            cout << endl;
            return;
        }
    }
    else if (delim == '*') // multiline comment
    {
        if (commChar != '*')
        {
            fin.get(commChar);
            strip(fin, delim, commChar);
        }
        else
        {
            fin.get(ch);
            if (ch != '/')
            {
                commChar = ch ;
                strip(fin, delim, commChar);
            }
            else
            {
                delim = ' ';
                return;
            }
        }
    }
    else
        return;
}
```

29–30. The following program solve Problem 30 and, therefore, Problem 29 as a special case.

```
/*-----
   Program to check whether parentheses in strings are balanced.

   Input: strings
   Output: Indications of whether the parentheses balance.
-----*/
#include <iostream>
#include <string>
using namespace std;

bool parensBalance(string str);
/*-----
   Check if parentheses balance.

   Precondition: String str contains only parentheses
   Postcondition: True is returned if they balance, false if not.
-----*/

int main()
{
    cout << "This program checks whether parentheses in a line of text "
        "balance.\n\n";
    string str;
    for(;;)
    {
        cout << "Enter a string (### to stop):\n";
        getline(cin, str);
        if (str == "###") break;

        cout << "Parentheses are " << (!parensBalance(str) ? "not " : "")
            << "balanced" << endl;
    }
}

//-- Definition of parensBalance()
bool parensBalance(string str)
{
    if (str.size() == 0) return true;      // empty string -- no parent

    int left = str.find('(');
    int right = str.find(')');

    if (left == string::npos && right == string::npos) // no parens
        return true;

    //else
    if (left == string::npos || right == string::npos) // no match
        return false;
```

```

//else
if (left > right)                                // mismatch: ) (
    return false;

// else remove matching parens and recursively check remainder
str.erase(right, 1);
str.erase(left, 1);
return parensBalance(str);
}

```

Section 10.5

31.

```

/*-----
Program to read and store stock prices and then sort them. It then
finds the trading range and a sequence showing how much the price
rose or fell each day.

Input: sequence of prices
Output: trading range (highest - lowest price)
        sequence of daily increase or decrease in price

#include <iostream>           // cin, cout, >>, <<
#include <vector>             // vector
#include <algorithm>          // sort(), min_element(), max_element()
#include <numeric>            // adjacent_difference()
using namespace std;

int main()
{
    double aPrice;
    vector<double>stockPrice;

    cout << "Enter the stock prices (-1 to stop) for:\n";
    for (int day = 1; ; day++)
    {
        cout << "Day " << day << ": ";
        cin >> aPrice;
        if (aPrice < 0) break;

        stockPrice.push_back(aPrice);
    }

    vector<double> sorted = stockPrice;
    sort(sorted.begin(), sorted.end());
    cout << "Sorted stock prices:\n";
    for (int i = 0; i < sorted.size(); i++)
        cout << sorted[i] << endl;

    cout << "\nTrading range = "
        << sorted[sorted.size() - 1] - sorted[0] << endl;
}

```

```

// Alternatively, if data not sorted, can find trading range with
//
//    *max_element(stockPrice.begin(), stockPrice.end()) -
//    (*min_element(stockPrice.begin(), stockPrice.end()))
//
vector<double> change(stockPrice.size());
adjacent_difference(stockPrice.begin(), stockPrice.end(),
                    change.begin());

cout << "\nHow prices rose and fell:\n";
for (int i = 1; i < change.size(); i++)
    cout << showpos << change[i] << endl;
}

```

32.

```

/*-----
Program to read and store production levels, find the lowest,
highest, and average production level, a sequence showing how much
the production level rose or fell each day, and a sequence of
cumulative total production for each day.

Input: sequence of production levels
Output: lowest, highest, and average production level,
        a sequence showing how much production rose or fell each day,
        and a sequence of cumulative total production per day
-----*/
#include <iostream>           // cin, cout, >>, <<
#include <vector>              // vector
#include <algorithm>           // min_element(), max_element()
#include <numeric>              // adjacent_difference(), partial_sum
using namespace std;

int main()
{
    int production;
    vector<int> prodLevel;

    cout << "Enter the production levels (-1 to stop) for:\n";
    for (int day = 1; ; day++)
    {
        cout << "Day " << day << ": ";
        cin >> production;
        if (production < 0) break;

        prodLevel.push_back(production);
    }

    cout << "\nLowest production level = "
        << *min_element(prodLevel.begin(), prodLevel.end())
        << "\nHighest production level = "
        << *max_element(prodLevel.begin(), prodLevel.end())
        << "\nAverage production level = "

```

```

    << accumulate(prodLevel.begin(), prodLevel.end(), 0.0)
        / prodLevel.size()
    << endl;

vector<int> change(prodLevel.size());
adjacent_difference(prodLevel.begin(), prodLevel.end(), change.begin());

cout << "\nHow production levels rose and fell:\n";
for (int i = 1; i < change.size(); i++)
    cout << "Day " << noshowpos << i + 1 << ":" "
        << showpos << change[i] << endl;

vector<int> total(prodLevel.size());
partial_sum(prodLevel.begin(), prodLevel.end(), total.begin());

cout << "\nCumulative production levels:\n";
for (int i = 0; i < total.size(); i++)
    cout << "Day " << noshowpos << i + 1 << ":" "
        << showpos << total[i] << endl;
}

```

33–34.

```

/*
-----*
Program that reads product numbers of products stocked in two
different warehouses, storing these in vector<int>s, and then finds
the intersection and the union of these two lists of numbers.

Input: production levels, week and day numbers
Output: production levels
-----*/
#include <iostream>                      // cin, cout, >>, <<
#include <vector>                         // vector<T>
#include <algorithm>                      // set_intersection(), set_union()
using namespace std;

istream & operator>>(istream & in, vector<int> & v);
/*
-----*
Input operator for vector<int>s.

Precondition: istream in is open.
Postcondition: Elements of v have been input from in.
-----*/
void display(ostream & out,
             vector<int>::iterator begin, vector<int>::iterator end);
/*
-----*
Output function for a subsequence of vector<int>s.

Precondition: ostream out is open.
Postcondition: Elements of v in the range begin up to (but not
               including) end have been output to out.
-----*/
int main()

```

```
{  
    vector<int> prodChicago,  
        prodDetroit;  
  
    cout << "For Chicago, enter product numbers at prompt, "  
        "(-1 to stop):\n";  
    cin >> prodChicago;  
  
    cout << "\nFor Detroit, enter product numbers at prompt, ("  
        "(-1 to stop):\n";  
    cin >> prodDetroit;  
  
    vector<int> prodBoth(prodChicago.size()),  
        prodEither(prodChicago.size() + prodDetroit.size());  
  
    vector<int>::iterator  
        intersectEnd = set_intersection(  
            prodChicago.begin(), prodChicago.end(),  
            prodDetroit.begin(), prodDetroit.end(),  
            prodBoth.begin()),  
  
        unionEnd = set_union(prodChicago.begin(), prodChicago.end(),  
            prodDetroit.begin(), prodDetroit.end(),  
            prodEither.begin());  
  
    cout << "\nProducts in both Chicago and Detroit:\n";  
    display(cout, prodBoth.begin(), intersectEnd);  
    cout << endl;  
  
    cout << "\nProducts in either Chicago or Detroit (or both):\n";  
    display(cout, prodEither.begin(), unionEnd);  
    cout << endl;  
}  
  
//-- Definition of operator>>()  
istream & operator>>(istream & in, vector<int> & v)  
{  
    int x;  
    cout << '>';  
    for (;;) {  
        cin >> x;  
        if (x < 0) break;  
  
        v.push_back(x);  
    }  
    return in;  
}  
  
//-- Definition of display()  
void display(ostream & out,  
            vector<int>::iterator first, vector<int>::iterator end)  
{  
    for (vector<int>::iterator it = first; it != end; it++)  
        cout << *it << " ";  
}
```

35.

```
/*-----*
   Class to model store product numbers and prices.
-----*/
#ifndef PRODUCT
#define PRODUCT

class Product
{
public:
    Product(int number = 0, double price = 0);
    /*-----
       Constructor.

       Precondition: None
       Postcondition: Product object has been constructed with product
                      number 0 and price 0.
    -----*/
    int number() const
    /*-----
       Accessor.

       Precondition: None
       Postcondition: Object's number is returned.
    -----*/
    { return myNumber; }

    double price() const
    /*-----
       Accessor.

       Precondition: None
       Postcondition: Object's price is returned.
    -----*/
    { return myPrice; }

    void read(istream & in);
    /*-----
       Input.

       Precondition: istream in is open.
       Postcondition: Product object's number and price have been input
                      from and removed from in.
    -----*/

private:
    int myNumber;
    double myPrice;
};


```

```
//-- Definition of constructor
inline Product::Product(int number, double price)
: myNumber(number), myPrice(price)
{ }

//-- Definition of read()
inline void Product::read(istream & in)
{
    cout << "Enter product number and price: ";
    in >> myNumber >> myPrice;
}

//-- Definition of <<
inline ostream & operator<<(ostream & out, Product x)
{
    out << "Number: " << x.number() << " Price: " << x.price();
    return out;
}

//-- Definition of ==
inline bool operator==(Product x, Product y)
//-- == means objects have same product number
{
    return x.number() == y.number();
}

//-- Definition of <
inline bool operator<(Product x, Product y)
//-- < means first object's number < second object's number or if they
//-- are equal, first object's price < second object's price.
{
    if (x.number() < y.number())
        return true;
    else if (x.number() == y.number())
        return x.price() < y.price();
    else
        return false;
}

#endif

/*
-----  

Program to read and store product numbers and prices.  

It then allows the operations:  

1. Find price of a product.  

2. Count products with a given price  

3. Remove products whose numbers match that of some other product  

4. Sort the products so product numbers are in ascending order  

5. Display a table of product numbers and prices

```

```
Input: sequence of products (numbers and prices)
Output: results of the above operations
-----*/
#include <iostream>           // cin, cout, >>, <<
#include <vector>             // vector
#include <algorithm>          // find(), count(), unique(), sort()
#include <numeric>             // accumulate()
using namespace std;

#include "Product.h"

int getMenuOption();
/*-----
   Display menu and get a selection from user.

   Precondition: None
   Postcondition: Menu of options has been displayed and option selected
   by user has been returned.
-----*/
int main()
{
    Product p;
    vector<Product> product;

    cout << "Enter the products (zeros to stop):\n";
    for (;;)
    {
        p.read(cin);
        if (p == Product(0,0)) break;

        product.push_back(p);
    }

    vector<Product>::iterator it;
    int option,
        count;
    do
    {
        option = getMenuOption();
        switch(option)
        {
            case 1 : cout << "Enter product number: ";
                      int prodNum; cin >> prodNum;
                      it = find(product.begin(), product.end(), prodNum);
                      if (it == product.end())
                          cout << "Product not found" << endl;
                      else
                          cout << "Price is $" << (*it).price() << endl;
                      break;

            case 2 : cout << "Enter a price: ";
                      double price; cin >> price;
                      count = 0;
                      for (int i = 0; i < product.size(); i++)

```

```
        if (product[i].price() == price)
            count++;
        cout << "There were " << count
            << " products with this price\n";
        break;

case 3 : cout << "We first sort the product list (by product "
           "number).\n";
sort(product.begin(), product.end());

it = unique(product.begin(), product.end());

// Remove extras left at end of list by unique()
while (it != product.end())
    product.pop_back();
cout << "Duplicate product numbers have now been "
    "removed.\n";
break;

case 4 : sort(product.begin(), product.end());
cout << "List has now been sorted by product number.\n";
break;

case 5 : cout << "List of products: \n";
for (int i = 0; i < product.size(); i++)
    cout << product[i] << endl;
break;

case 0 : cout << "Stopping\n";
break;

default: cout << "Invalid option -- try again.";
}
}
while (1 <= option && option <= 5);
}

//-- Definition of getMenuOption
int getMenuOption()
{
    int option;
    do
    {
        cout << "\nSelect an option from the following list:\n";
        cout <<
        "1. Find price of a product\n"
        "2. Count products with a given price\n"
        "3. Remove products whose numbers match that of some others\n"
        "4. Sort the products so product numbers are in ascending order\n"
        "5. Display a table of product numbers and prices\n"
        "0. Stop\n"
        "-->;

        cin >> option;
    }
    while (option < 0 || option > 5);
```

```
    return option;
}

36.
/*-----
   Program to assign letter grades using grading on the curve.
   It provides the following operations:

   1. Find mean score
   2. Find the variance
   3. Find the standard deviation
   4. Find letter grades. Note: Formula in text is modified so
      that all grades are C if standard deviation is 0 (i.e., all
      scores are the same).

   Input: sequence of numeric scores
   Output: mean, variance, std. deviation, and letter grades
-----*/
#include <iostream>           // cin, cout, >>, <<
#include <vector>              // vector
#include <algorithm>           //
#include <numeric>              // accumulate(), inner_product
using namespace std;

double mean(const vector<double> & v);
/*-----
   Find the arithmetic mean of a list of doubles.

   Precondition: None
   Postcondition: Returns the mean of the elements of v.
-----*/
double variance(const vector<double> & v);
/*-----
   Find the variance of a list of doubles.

   Precondition: None
   Postcondition: Returns the variance of the elements of v.
-----*/
double stdDev(const vector<double> & v);
/*-----
   Find the standard deviation of a list of doubles.

   Precondition: None
   Postcondition: Returns the standard deviation of the elements of v.
-----*/
vector<char> letterGrades(const vector<double> & v);
/*-----
   Find letter grades for a list of numeric scores using the method
   of "grading on the curve."

   Precondition: None
-----*/
```

```
Postcondition: A list of letter grades for the scores in v is
    returned.
-----*/
int main()
{
    cout << "Enter scores (-1 to stop): ";
    double aScore;
    vector<double> score;
    for(;;)
    {
        cout << '>';
        cin >> aScore;
        if (aScore < 0) break;

        score.push_back(aScore);
    }

    cout << "The mean is : " << mean(score)
        << "\nThe variance is : " << variance(score)
        << "\nThe standard deviation is : " << stdDev(score) << endl;

    vector<char> letterGrade = letterGrades(score);
    cout << "\nLetter Grades:\n";
    for (int i = 0; i < score.size(); i++)
        cout << score[i] << " --- " << letterGrade[i] << endl;
}

//-- Definition of mean()
double mean(const vector<double> & v)
{
    if (v.empty()) return 0.0;
    return accumulate(v.begin(), v.end(), 0.0) / v.size();
}

//-- Definition of variance()
double variance(const vector<double> & v)
{
    if (v.empty()) return 0.0;

    double theMean = mean(v);
    vector<double> diff = v;
    for (int i = 0; i < diff.size(); i++)
        diff[i] -= theMean;
    return inner_product(diff.begin(), diff.end(), diff.begin(), 0.0)
        / v.size();
}

//-- Definition of stdDev()
double stdDev(const vector<double> & v)
{
    if (v.empty()) return 0.0;
    // else
```

```
    return sqrt(variance(v));
}

//-- Definition of letterGrades()
vector<char> letterGrades(const vector<double> & score)
{
    double theMean = mean(score),
           stdev = stdDev(score),
           DF = theMean - 1.5 *stdev,
           CD = theMean - 0.5 *stdev,
           BC = theMean + 0.5 *stdev,
           AB = theMean + 1.5 *stdev;
    vector<char> grade;

    for (int n = 0; n < score.size(); n++)
        if (score[n] < DF)
            grade.push_back('F');
        else if (score[n] < CD)
            grade.push_back('D');
        else if (stdev == 0 || score[n] < BC)
            grade.push_back('C');
        else if (score[n] < AB)
            grade.push_back('B');
        else
            grade.push_back('A');

    return grade;
}
```

Chapter 11: More Linking Up with Linked Lists

Exercises 11.1

1. The implementation of the `Queue` class using a linked list in Section 8.3 can be easily converted to a class template (see how this was done for the array-based implementation in the solution of Exercise 8 of Section 9.3. This class template can then be easily changed into a class template for priority queues. We need only modify the function member `enqueue()` so that each item is inserted at the proper position for a priority queue and change all occurrences of `Queue` to `PriorityQueue`. The following version assumes that the data value in each node can be used as a priority. If not, a new field should be added to the nodes to store priorities.

```
template <typename ElementType>
void PriorityQueue<T>::enqueue(ElementType dataVal)
{
    PriorityQueue<ElementType>::NodePointer newPtr = new Node(dataVal);

    if (empty())
        myFront = myBack = newPtr;
    else
    {
        LinkedPQueue<ElementType>::NodePointer prev = 0,
                                                current = myFront;
        while (current != 0 && current->data > dataVal)
        {
            prev = current;
            current = current->next;
        }

        if (prev != 0)
        {
            newPtr->next = prev->next;
            prev->next = newPtr;
            if (current == 0) // new last element in priority queue
                myBack = newPtr;
        }
        else // new first element in priority queue
        {
            newPtr->next = myFront;
            myFront = newPtr;
        }
    }
}
```

2. // To search a circular linked list with first node pointed to by *first*.

If *first* == 0
 Return 0. // list is empty, so return null pointer to indicate *item* not found
 Else if *first*->*data* == *item*
 Return *first*. // *item* found in first node

- ```

Else // start search at second node
 a. Set ptr = first->next.
 b. While ptr ≠ first, do the following:
 If (ptr->data) == item
 Return ptr. // item found in node pointed to by ptr
 Else
 Set ptr = ptr->next.
 End while
 c. Return 0. // Entire list traversed with item not found

3. Replace the condition ptr ≠ first in (b) with
 ptr ≠ first and (ptr->data) <= item

4. /* To find nth successor of item pointed to by locPtr in a circular linked list with
 first node pointed to by first. */

```

If *first* == 0 or *locPtr* == 0 // list is empty or item not found in the list  
 Return 0.  
 Else do the following:  
 a. Set *nPtr* = *locPtr*.  
 b. For *count* ranging from 1 to *n*:  
 Set *nPtr* = *nPtr ->next*.  
 c. Return *nPtr*.

5.

```

----- CLDequeT.h -----
This header file contains a class template for a Deque data type.
Basic operations:
 constructor: Constructs an empty deque
 copy constructor: Constructs a copy of a deque
 =: Assignment operator
 destructor: Destroys a deque
 empty: Checks if a deque is empty
 add: Modifies a deque by adding a value at one end
 retrieve: Retrieve the value at one end; leaves deque unchanged
 remove: Modifies a deque by removing the value at one end
 display: Displays the deque elements

A circular linked list is used to store the deque elements.
----- */

```

```

#include <iostream>
#include <cassert>
#include <new>
using namespace std;

#ifndef CLDEQUET
#define CLDEQUET

enum End {FRONT, BACK};


```

```
template <typename DequeElement>
class Deque
{
private:
 class Node
 {
public:
 //----- DATA MEMBERS OF Node
 DequeElement data;
 Node * next;

 //----- Node OPERATIONS

 /* --- The Node default class constructor initializes a Node's
 next member.

 Precondition: None
 Postcondition: The next member has been set to 0.
 */
 Node()
 : next(0)
 {}

 /* --- The Node class constructor initializes a Node's data members.

 Precondition: None
 Postcondition: The data and next members have been set to
 dataValue and 0, respectively.
 */
 Node(DequeElement dataValue)
 : data(dataValue), next(0)
 {}
}; //--- end of Node class

typedef Node * NodePointer;

/***** Function Members *****/
public:
 Deque();
 /*-----
 Construct a Deque object.

 Precondition: None.
 Postcondition: An empty Deque object has been constructed
 (myBack is initialized to 0).
 */
 Deque(const Deque & original);
 /*-----
 Copy Constructor

 Precondition: original is the deque to be copied and
 is received as a const reference parameter.
 Postcondition: A copy of original has been constructed.
 */
}
```

```
***** Destructor *****
~Deque();
/*
----- Class destructor

Precondition: None
Postcondition: The linked list in the deque has been
destroyed.
-----*/
***** Assignment *****
const Deque & operator=(const Deque & rightHandSide);
/*
----- Assignment Operator

Precondition: original is the deque to be assigned and
is received as a const reference parameter.
Postcondition: The current deque becomes a copy of
original and a const reference to it is returned.
-----*/
bool empty() const;
/*
----- Check if deque is empty.

Precondition: None.
Postcondition: True is returned if the deque is empty and false is
returned otherwise.
-----*/
void add(const DequeElement & value, End where);
/*
----- Add a value to a deque.

Precondition: where is FRONT (0) or BACK (1).
Postcondition: value is added at end of deque specified by where,
provided there is space; otherwise, a deque-full message is
displayed and execution is terminated.
-----*/
DequeElement retrieve(End where) const;
/*
----- Retrieve value at one end of deque (if any).

Precondition: Deque is nonempty; where is FRONT (0) or BACK (1).
Postcondition: Value at end of deque specified by where is
returned, unless deque is empty; in that case, an error message
is displayed and a "garbage value" is returned.
-----*/
void remove(End where);
/*
----- Remove value at one end of deque (if any).

Precondition: Deque is nonempty; value of where is
FRONT (0) or BACK (1).
```

```
Postcondition: Value at end of deque specified by where is
 removed, unless deque is empty; in that case, an error message
 is displayed.
-----*/
// --- display
void display(ostream & out) const;
/*-----
 Output the values stored in the deque.

 Precondition: ostream out is open.
 Postcondition: Deque's contents have been output to out.
-----*/
***** Data Members *****/
private:
 NodePointer myBack;
}; // end of class declaration

//-- Definition of constructor
template <typename DequeElement>
inline Deque<DequeElement>::Deque()
: myBack(0)
{ }

//-- Definition of empty()
template <typename DequeElement>
inline bool Deque<DequeElement>::empty() const
{ return myBack == 0; }

//-- Definition of add()
template <typename DequeElement>
void Deque<DequeElement>::add(const DequeElement & value, End where)
{
 assert (where == FRONT || where == BACK);
 Deque<DequeElement>::NodePointer newPtr = new(nothrow) Node(value);
 if (newPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 if (empty())
 {
 newPtr->next = newPtr;
 myBack = newPtr;
 }
 else if (where == BACK)
 {
 newPtr->next = myBack->next;
 myBack->next = newPtr;
 myBack = newPtr;
 }
 else // where == FRONT
 {
 newPtr->next = myBack->next;
 myBack->next = newPtr;
 }
}
```

```
//-- Definition of retrieve()
template <typename DequeElement>
DequeElement Deque<DequeElement>::retrieve(End where) const
{
 assert (where == FRONT || where == BACK);
 if (empty())
 {
 cerr <<"Deque is empty: error! Returning garbage value\n";
 DequeElement garbage;
 return garbage;
 }
 else if (where == BACK)
 return myBack->data;
 else // where == FRONT
 return myBack->next->data;
}

//-- Definition of remove()
template <typename DequeElement>
void Deque<DequeElement>::remove(End where)
{
 assert (where == FRONT || where == BACK);
 if (empty())
 cerr <<"Deque empty: Cannot remove an element. Error!!\n";
 else
 {
 Deque<DequeElement>::NodePointer frontPtr = myBack->next;
 if (frontPtr == myBack) // one-node deque
 myBack = 0; // deque now empty
 else // more than 1 element
 {
 if (where == FRONT)
 {
 myBack->next = frontPtr->next;
 delete frontPtr;
 }
 else // where == BACK
 {
 Deque<DequeElement>::NodePointer ptr = frontPtr;
 while (ptr->next != myBack)
 ptr = ptr->next;
 delete myBack;
 myBack = ptr;
 myBack->next = frontPtr;
 }
 }
 }
}
```

```
//-- Definition of display()
template <typename DequeElement>
void Deque<DequeElement>::display(ostream & out) const
{
 if (!empty())
 {
 Deque<DequeElement>::NodePointer ptr = myBack;
 do
 {
 ptr = ptr->next;
 out << ptr->data << " ";
 }
 while (ptr != myBack);
 }
}

// Definition of <<
template <typename DequeElement>
inline ostream & operator<<(ostream & out, const Deque<DequeElement> aDeque)
{
 aDeque.display(out);
 return out;
}

// Definition of the destructor
template <typename DequeElement>
Deque<DequeElement>::~Deque()
{
 if (myBack != 0)
 {
 Deque<DequeElement>::NodePointer ptr,
 prev = myBack->next;
 while (prev != myBack)
 {
 ptr = prev->next;
 delete prev;
 prev = ptr;
 }
 delete myBack;
 }
}

// Definition of the copy constructor
template <typename DequeElement>
Deque<DequeElement>::Deque(const Deque<DequeElement> & original)
{
 myBack = 0;
 if (!original.empty())
 {
 Deque<DequeElement>::NodePointer origPtr = original.myBack->next,
 frontPtr, lastPtr;

 frontPtr = new Node(origPtr->data);
 if (frontPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 lastPtr = frontPtr;
```

```

while (origPtr != original.myBack)
{
 origPtr = origPtr->next;
 lastPtr->next = new Node(origPtr->data);
 if (lastPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 lastPtr = lastPtr->next;
}
lastPtr->next = frontPtr;
myBack = lastPtr;
}

// Definition of the assignment operator
template <typename DequeElement>
const Deque<DequeElement> & Deque<DequeElement>::operator=(const Deque<DequeElement> & original)
{
 myBack = 0;
 if (this != &original)
 {
 delete myBack;
 Deque<DequeElement>::NodePointer origPtr = original.myBack->next,
 frontPtr, lastPtr;

 frontPtr = new Node(origPtr->data);
 if (frontPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 lastPtr = frontPtr;
 while (origPtr != original.myBack)
 {
 origPtr = origPtr->next;
 lastPtr->next = new Node(origPtr->data);
 if (lastPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 lastPtr = lastPtr->next;
 }
 lastPtr->next = frontPtr;
 myBack = lastPtr;
 }
 return *this;
}
#endif

```

6. Algorithm to perform a shuffle-merge of two circular linked lists with head nodes.  
If there are no head nodes, supply temporary ones.

```

Get a node pointed to by headPtr;
headPtr->next = first;
first = headPtr;

```

```
// and remove it at the end of processing:
first = headPtr->next;
Delete node pointed to by headPtr;

Get a head node pointed to by mergePtr;

last = mergePtr;
ptr1 = first1->next; // runs through first list
ptr2 = first2->next; // runs through second list

while (ptr1 != first1 && ptr2 != first2)
{
 Get a node pointed to by tempPtr;
 tempPtr->data = ptr1->data;
 last->next = tempPtr;
 last = tempPtr;
 ptr1 = ptr1->next;
 Get a node pointed to by tempPtr;
 tempPtr->data = ptr2->data;
 last->next = tempPtr;
 last = tempPtr;
 ptr2 = ptr2->next;
}

while (ptr1 != first1)
{
 Get a node pointed to by tempPtr;
 tempPtr->data = ptr1->data;
 last->next = tempPtr;
 last = tempPtr;
 ptr1 = ptr1->next;
}

while (ptr2 != first2)
{
 Get a node pointed to by tempPtr;
 tempPtr->data = ptr2->data;
 last->next = tempPtr;
 last = tempPtr;
 ptr2 = ptr2->next;
}

// If no head nodes are used, delete headnode in merged list
// tempPtr = mergePtr
// mergePtr = mergePtr->next
// Delete node pointed to by tempPtr
```

7. Algorithm to perform a shuffle-merge of two circular linked lists with head nodes without copying data items into new nodes. If there are no head nodes, proceed as described in Exercise 6.

```

if (first1->next == first1) // empty first list
 mergePtr = first2;
else
{
 mergePtr = first1;
 ptr1 = first1->next; // runs through first list
 ptr2 = first2->next; // runs through second list

 while (ptr1 != first1 && ptr2 != first2)
 {
 tempPtr = ptr1
 ptr1 = tempPtr->next;
 tempPtr->next = ptr2;
 tempPtr = ptr2;
 ptr2 = tempPtr->next;
 if (ptr1 != first1)
 tempPtr->next = ptr1;
 }

 if (ptr1 == first1) // find last node in list
 while ptr2 != first2
 {
 temp = ptr2
 ptr2 = ptr2->next
 }
 temp->next = mergedPtr
 }
}

```

8. Algorithm to return a circular linked list to the storage pool.

```

tempPtr = first;
first = free;
free = tempPtr;

```

9. Algorithm for Josephus problem
1. Get a value for *numSoldiers*.
  2. Store the soldier's numbers/names in a circular linked list.
  3. Generate random integers *n* and *start* with  $1 \leq n, start \leq numSoldiers$ .
  4. Move a pointer *ptr* from the first node of the list to the *start*-th node.
  5. For *turns* ranging from 1 to *numSoldiers* – 1 do the following:
    - a. For *count* ranging from 1 to  $n - 1$  do the following:
      - Let *start* = *start*->*next*.
      - b. Let pointer *out* = *start*->*next*.
      - c. Let pointer *newStart* = *out*->*next*.
      - d. Delete the node pointed to by *out*.
      - e. Let *start* = *newStart*.
  6. Return the name/number of the soldier in the single node remaining in the list.

## Exercises 11.2

1-6:

```
/* PolynomialT.h is the interface for class template Polynomial
for processing sparse (linked) polynomials.
-----*/
#include <iostream>
using namespace std;

#ifndef POLYNOMIAL
#define POLYNOMIAL

template <typename CoefType>
class Polynomial
{
private:
 /*** Term class ***/
 class Term
 {
public:
 /*** Data members ***/
 CoefType coef;
 int expo;
}; //end class Term

/*** Node class ***/
class Node
{
public:
 Term data;
 Node * next;

 //-- Node constructor
 // Creates a Node with given initial values
 Node(CoefType co = 0, int ex = 0, Node * ptr = 0)
 {
 data.coef = co;
 data.expo = ex;
 next = ptr;
 }
}; // end of Node class
typedef Node * NodePointer;

public:
 /*** Function members ***/
 // Operations on polynomials

 Polynomial();
 /*--Constructor-----
 Precondition: None
 Postcondition: A polynomial representing the zero polynomial
 is constructed (myDegree == 0 and empty linked list).
-----*/
}
```

```
~Polynomial();
/*--Destructor-----
 Precondition: None
 Postcondition: A polynomial is destroyed -- nodes in linked list are
 deallocated.
-----*/
void read(istream & in);
/*--Input-----
 Precondition: istream in is open
 Postcondition: coefficients and exponents of polynomial are read from
 in and stored in the linked list.
-----*/
void display (ostream & out);
/*--Output-----
 Precondition: ostream out is open
 Postcondition: Polynomial is output to out.
-----*/
CoefType value(CoefType xValue);
/*--Evaluate-----
 Precondition: None
 Postcondition: Value of polynomial is returned.
-----*/
Polynomial<CoefType> operator+(const Polynomial<CoefType> & secondPoly);
/*--Addition-----
 Precondition: None
 Postcondition: A polynomial representing the sum of
 the current polynomial and secondPoly is returned.
-----*/
Polynomial<CoefType> operator-(const Polynomial<CoefType> & secondPoly);
/*--Addition-----
 Precondition: None
 Postcondition: A polynomial representing the difference of
 the current polynomial and secondPoly is returned.
-----*/
Polynomial<CoefType> operator*(const Polynomial<CoefType> & secondPoly);
/*--Multiplication-----
 Precondition: None
 Postcondition: A polynomial representing the product of
 the current polynomial and secondPoly is returned.
-----*/
Polynomial derivative();
/*--Derivative-----
 Precondition: None
 Postcondition: A polynomial representing the derivative
 of the current polynomial is returned.
-----*/
```

```
bool searchExpo(int anExpo, Polynomial<CoefType>::NodePointer & prev);
/*--searchExpo-----
 Precondition: None
 Postcondition: prev points to predecessor of node containing anExpo
 and true is returned if such a node is found, else false is
 returned.
-----*/
private:
 /*** Data members ***/
 int myDegree;
 NodePointer myTerms; // a linked list with head node;
 // its ElementType is Term
}; // end of Polynomial class template

//--- DEFINITIONS OF FUNCTION MEMBERS

//--- Constructor
template <typename CoefType>
Polynomial<CoefType>::Polynomial()
{
 // 0 polynomial
 myTerms = new Polynomial<CoefType>::Node(); // head node
 if (myTerms == 0)
 { cerr << "Out of memory\n"; exit(1); }
 myDegree = 0;
}

//--- Destructor
template <typename CoefType>
Polynomial<CoefType>::~Polynomial()
{
 Polynomial<CoefType>::NodePointer prev = myTerms,
 ptr = myTerms->next;
 while (ptr != 0)
 {
 delete prev;
 prev = ptr;
 ptr = ptr->next;
 }
 delete myTerms;
}

//--- Input
template <typename CoefType>
void Polynomial<CoefType>::read(istream & in)
{
 cout << "Enter coefficients and exponents with exponents in "
 "increasing order\n(negative expo. to signal end of "
 "polynomial,\nboth coef. & expo. = 0 for zero polynomial):\n";
 (*this).~Polynomial(); // destroy any old polynomial
 myTerms = new Polynomial<CoefType>::Node(); // new head node
 if (myTerms == 0)
 { cerr << "Out of memory\n"; exit(1); }
```

```
CoefType coeff;
int expo;

Polynomial<CoefType>::NodePointer prev = myTerms,
 newPtr = 0;

for (;;)
{
 cin >> coeff >> expo;
 if (expo < 0) return;

 newPtr = new Polynomial<CoefType>::Node(coeff, expo);
 if (newPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 prev->next = newPtr;
 prev = newPtr;
}

//--- Output
template <typename CoefType>
void Polynomial<CoefType>::display(ostream & out)
{
 if (myTerms->data.coef == 0 && myTerms->data.expo == 0 &&
 myTerms->next == 0)
 {
 out << 0 << endl;
 return;
 }

 //--- Not zero polynomial
 Polynomial<CoefType>::NodePointer ptr = myTerms->next;

 while (ptr != 0)
 {
 out << ptr->data.coef;
 if (ptr->data.expo >= 1)
 out << 'x';
 if (ptr->data.expo >= 2)
 out << '^' << ptr->data.expo;
 if (ptr->next != 0)
 out << " + ";
 ptr = ptr->next;
 }
 out << endl;
}

//--- value
template <typename CoefType>
CoefType Polynomial<CoefType>::value(CoefType xValue)
{
 if (myTerms->data.coef == 0 && myTerms->data.expo == 0 &&
 myTerms->next == 0)
 return 0;
```

```
//-- Not zero polynomial
Polynomial<CoefType>::NodePointer ptr = myTerms->next;
double polyValue = 0;
double power = 1.0;
int prevExpo = 0;
while (ptr != 0)
{
 for (int i = prevExpo; i < ptr->data.expo; i++)
 power *= xValue;

 polyValue += ptr->data.coef * power;

 prevExpo = ptr->data.expo;
 ptr = ptr->next;
}

return polyValue;
}

//-- Addition operator
template <typename CoefType>
Polynomial<CoefType> Polynomial<CoefType>::operator+
 (const Polynomial<CoefType> & secondPoly)
{
 Polynomial<CoefType> sumPoly;
 Polynomial<CoefType>::NodePointer // pointers to run
 ptra = myTerms->next, // thru 1st poly,
 ptrb = secondPoly.myTerms->next, // 2nd poly,
 ptrc = sumPoly.myTerms; // sum poly
 int degree = 0;
 while (ptra != 0 || ptrb != 0) // More nodes in
 { // one of polys
 if ((ptrb == 0) || // Copy from 1st poly
 (ptra != 0 && ptra->data.expo < ptrb->data.expo))
 {
 ptrc->next = new Polynomial<CoefType>::
 Node(ptra->data.coef, ptra->data.expo);
 degree = ptra->data.expo;
 ptra = ptra->next;
 ptrc = ptrc->next;
 }
 else if ((ptra == 0) || // Copy from 2nd poly
 (ptrb != 0 && ptrb->data.expo < ptra->data.expo))
 {
 ptrc->next = new Polynomial<CoefType>::
 Node(ptrb->data.coef, ptrb->data.expo);
 degree = ptrb->data.expo;
 ptrb = ptrb->next;
 ptrc = ptrc->next;
 }
 }
}
```

```

 else // Exponents match
 {
 CoefType sum = ptra->data.coef + ptrb->data.coef;
 if (sum != 0) // Nonzero sum --
 { // add to sum poly
 ptrc->next = new Polynomial<CoefType>:::
 Node(sum, ptra->data.expo);
 degree = ptra->data.expo;
 ptrc = ptrc->next;
 }
 ptra = ptra->next; // Move along in
 ptrb = ptrb->next; // 1st & 2nd lists
 }
}
sumPoly.myDegree = degree; // Wrapup
return sumPoly;
}

//-- Subtraction operator
template <typename CoefType>
Polynomial<CoefType> Polynomial<CoefType>::operator-
 (const Polynomial<CoefType> & secondPoly)
{
 Polynomial<CoefType> diffPoly;
 Polynomial<CoefType>::NodePointer // pointers to run
 ptra = myTerms->next, // thru 1st poly,
 ptrb = secondPoly.myTerms->next, // 2nd poly,
 ptrc = diffPoly.myTerms; // difference poly
 int degree = 0;
 while (ptra != 0 || ptrb != 0) // More nodes in
 { // one of polys
 if ((ptrb == 0) || // Copy from 1st poly
 (ptra != 0 && ptra->data.expo < ptrb->data.expo))
 {
 ptrc->next = new Polynomial<CoefType>:::
 Node(ptra->data.coef, ptra->data.expo);
 degree = ptra->data.expo;
 ptra = ptra->next;
 ptrc = ptrc->next;
 }
 else if ((ptra == 0) || // Copy from 2nd poly
 (ptrb != 0 && ptrb->data.expo < ptra->data.expo))
 {
 ptrc->next = new Polynomial<CoefType>:::
 Node(ptrb->data.coef, ptrb->data.expo);
 degree = ptrb->data.expo;
 ptrb = ptrb->next;
 ptrc = ptrc->next;
 }
 }
 else // Exponents match
 {
 CoefType diff = ptra->data.coef - ptrb->data.coef;

```

```

 if (diff != 0) // Nonzero difference --
 { // add to difference poly
 ptrc->next = new Polynomial<CoefType>::

 Node(diff, ptra->data.expo);
 degree = ptra->data.expo;
 ptrc = ptrc->next;
 }
 ptra = ptra->next; // Move along in
 ptrb = ptrb->next; // 1st & 2nd lists
 }
}
diffPoly.myDegree = degree; // Wrapup
return diffPoly;
}

//--- Derivative
template <typename CoefType>
Polynomial<CoefType> Polynomial<CoefType>::derivative()
{
 Polynomial<CoefType> der;
 Polynomial<CoefType>::NodePointer ptr = myTerms->next,
 dPrev = der.myTerms,
 newPtr = 0;
 while (ptr != 0)
 {
 if (ptr->data.expo != 0)
 {
 newPtr = new Polynomial<CoefType>::

 Node(ptr->data.expo * ptr->data.coef, ptr->data.expo - 1);
 if (newPtr == 0)
 { cerr << "Out of memory\n"; exit(1); }

 dPrev->next = newPtr;
 dPrev = newPtr;
 }
 ptr = ptr->next;
 }
 return der;
}

//--- operator*
template <typename CoefType>
Polynomial<CoefType> Polynomial<CoefType>::operator*(
 const
 Polynomial<CoefType> & secondPoly)
{
 Polynomial<CoefType> multPoly;
 Polynomial<CoefType>::NodePointer // pointers to run
 ptra = myTerms->next, // thru 1st poly,
 ptrb, // 2nd poly,
 ptrc = multPoly.myTerms, // difference poly
 prev;

 int expo;
 CoefType prod, coeff;
}

```

```
while (ptrA != 0)
{
 ptrB = secondPoly.myTerms->next;
 while (ptrB != 0)
 {
 expo = ptrA->data.expo + ptrB->data.expo;
 prod = ptrA->data.coef * ptrB->data.coef;
 bool found = multPoly.searchExpo(expo, prev);
 if (found) // multPoly has a node with this exponent
 {
 ptrC = prev->next;
 coeff = ptrC->data.coef + prod;
 if (coeff != 0)
 ptrC->data.coef = coeff;
 else // remove node from multPoly
 {
 prev->next = ptrC->next;
 delete ptrC;
 }
 }
 else // insert new node in c with this exponent
 {
 ptrC = new Polynomial<CoefType>::Node(prod, expo);
 if (ptrC == 0)
 { cerr << "Out of memory\n"; exit(1); }

 ptrC->next = prev->next;
 prev->next = ptrC;
 }
 ptrB = ptrB->next;
 }
 ptrA = ptrA->next;
}

return multPoly;
}

template <typename CoefType>
bool Polynomial<CoefType>::
 searchExpo(int anExpo, Polynomial<CoefType>::NodePointer & prev)
{
 // search for a node with expo == anExpo
 Polynomial<CoefType>::NodePointer ptr = myTerms->next;
 prev = myTerms;

 while (ptr != 0 && ptr->data.expo < anExpo)
 {
 prev = ptr;
 ptr = ptr->next;
 }
 return (ptr != 0 && ptr->data.expo == anExpo);
}

#endif
```

### Exercises 11.3

1. I
  2. R
  3. 0 (null pointer)
  4. p2
  5. p1
  6. I
  7. 0 (null pointer)
  8. R
9. 

```
cout << p1->prev->data << " "
 << p1->data << " "
 << p1->next->data << " "
 << p1->next->next->data << " "
 << p1->next->next->next->data << endl;
```
- or
- ```
ptr = p1->prev;
while (ptr != 0);
{
    cout << ptr->data << " ";
    ptr = ptr->next;
}
```
10.

```
p1->prev->data = 'M';
p1->next->data = 'S';
```
11.

```
temp = p1->next->next;
temp->prev->next = temp->prev->next;
temp->next->prev = temp->prev;
delete temp;
```
12.

```
cout << p2->prev->prev->prev->data << " "
    << p2->prev->prev->data << " "
    << p2->prev->data << " "
    << p2->data << " "
    << p2->next->data << endl;
```
- or
- ```
ptr = p2->prev->prev->prev;
while (ptr != 0);
{
 cout << ptr->data << " ";
 ptr = ptr->next;
}
```
13. 

```
p2->prev->prev->prev->data = 'M';
p2->prev->data = 'S';
```

```
14. temp = p2;
 temp->prev->next = temp->next;
 temp->next->prev = temp->prev;
 delete temp;

15. B
16. E
17. p2
18. p2
19. p1
20. B
21. p1
22. E

23. cout << p1->data << " "
 << p1->next->data << " "
 << p1->next->next->data << " "
 << p1->next->next->next->data << endl;

or
ptr = p1;
do
{
 cout << ptr->data << " ";
 ptr = ptr->next;
}
while (ptr != p1)

24. temp = new Node;
 temp->data = 'L';
 temp->prev = p1;
 temp->next = p1->next;
 p1->next->prev = temp;
 p1->next = temp;
 p1->next->next->data = 'A';

25. temp = p1->prev;
 temp->prev->next = temp->next;
 temp->next->prev = temp->prev;
 delete temp;

26. cout << p2->prev->prev->data << " "
 << p2->prev->data << " "
 << p2->data << " "
 << p2->next->data << endl;
```

or

```

first = ptr = p2->prev->prev;
do
{
 cout << ptr->data << " ";
 ptr = ptr->next;
}
while (ptr != first)

27. temp = new Node;
temp->data = 'L';
temp->prev = p2->prev->prev;
temp->next = p2->prev;
p2->prev->next->next = temp;
p2->prev->prev = temp;
p1->next->next->data = 'A';;

28. temp = p2->next;
temp->prev->next = temp->next;
temp->next->prev = temp->prev;
delete temp;

```

## Exercises 11.4

1-4 .

```

/*-- BigInt.h -----
This header file defines the data type BigInt for processing
nonnegative integers of any size.
Basic operations are:
 Constructor
 +: Addition operator
 <: Less-than operator
 ==: Equal-to operator
 --: Subtraction operator
 *: Multiplication operator
 read(): Read a BigInt object
 display(): Display a BigInt object
 <<, >> : Input and output operators
-----*/
#include <iostream>
#include <iomanip> // setfill(), setw()
#include <cmath> // pow
#include <list>

#ifndef BIGINT
#define BIGINT

const int DIGITS_PER_BLOCK = 3,
 MODULUS = (short int)pow(10.0, DIGITS_PER_BLOCK);
class BigInt
{
public:
 //***** Function Members *****/
 //***** Constructors *****/

```

```
BigInt()
{
}
/*-----
 Default Constructor

 Precondition: None
 Postcondition: list<short int>'s constructor was used to build
 this BigInt object.

-----*/
BigInt(int n);
/*-----
 Construct BigInt equivalent of n.

 Precondition: n >= 0.
 Postcondition: This BigInt is the equivalent of integer n.
-----*/
***** read *****
void read(istream & in);
/*-----
 Read a BigInt.

 Precondition: istream in is open and contains blocks of
 nonnegative integers having at most DIGITS_PER_BLOCK digits
 per block.
 Postcondition: Blocks have been removed from in and added to
 myList.
-----*/
***** display *****
void display(ostream & out) const;
/*-----
 Display a BigInt.

 Precondition: ostream out is open.
 Postcondition: The large integer represented by this BigInt
 object has been formatted with the usual comma separators
 and inserted into ostream out.
-----*/
***** addition operator *****
BigInt operator+(BigInt int2);
/*-----
 Add two BigInts.

 Precondition: int2 is the second addend.
 Postcondition: The BigInt representing the sum of the large
 integer represented by this BigInt object and int2 is
 returned.
-----*/
***** less-than operator *****
bool operator<(BigInt int2);
/*-----
 Compare two BigInts with <.
```

```
Precondition: int2 is the second operand.
Postcondition: true if this BigInt object is < int2.
-----*/

***** equal-to operator *****/
bool operator==(BigInt int2);
/*-----
 Compare two BigInts with ==.

 Precondition: int2 is the second operand.
 Postcondition: true if this BigInt object is equal to int2.
-----*/

***** subtraction operator *****/
BigInt operator-(BigInt int2);
/*-----
 Subtract two BigInts.

 Precondition: int2 is the second operand and this BigInt
 object >= int2.
 Postcondition: The BigInt representing the difference of the
 large integer represented by this BigInt object and int2 is
 returned if the precondition holds, otherwise 0 is returned.
-----*/

***** multiplication operator *****/
BigInt operator*(BigInt int2);
/*-----
 Multiply two BigInts.

 Precondition: int2 is the second operand.
 Postcondition: The BigInt representing the product of the large
 integer represented by this BigInt object and int2 is returned.
-----*/

private:
 /** Data Members ***/
 list<short int> myList;
}; // end of BigInt class declaration

//-- Definition of constructor
inline BigInt::BigInt(int n)
{
 do
 {
 myList.push_front(n % MODULUS);
 n /= MODULUS;
 }
 while (n > 0);
}

//---- Input and output operators
inline istream & operator>>(istream & in, BigInt & number)
{
```

```
 number.read(in);
 return in;
}

inline ostream & operator<<(ostream & out, const BigInt & number)
{
 number.display(out);
 return out;
}

//-- Definition of operator<
inline bool BigInt::operator<(BigInt int2)
{
 return
 (myList.size() < int2.myList.size())
 || (myList.size() == int2.myList.size()
 && myList < int2.myList);
}

//-- Definition of operator==
inline bool BigInt::operator==(BigInt int2)
{
 return myList == int2.myList;
}

/*-- BigInt.cpp-----
 This file implements BigInt member functions.
-----*/
#include <iostream>
using namespace std;
#include "BigInt.h"

//-- Definition of read()
void BigInt::read(istream & in)
{
 static bool instruct = true;
 if (instruct)
 {
 cout << "Enter " << DIGITS_PER_BLOCK << "-digit blocks, separated"
 " by spaces.\nEnter a negative integer in last block to"
 " signal the end of input.\n\n";
 instruct = false;
 }
 short int block;
 for (;;)
 {
 in >> block;
 if (block < 0) return;

 if (block >= MODULUS)
 cerr << "Illegal block -- " << block << " -- ignoring\n";
 else
 myList.push_back(block);
 }
}
```

```
---- Definition of display()
void BigInt::display(ostream & out) const
{
 int blockCount = 0;
 const int BLOCKS_PER_LINE = 20; // number of blocks per line

 for (list<short int>::const_iterator it = myList.begin(); ;)
 {
 out << setfill('0');
 if (blockCount == 0)
 out << setfill(' ');

 if (it == myList.end())
 return;

 out << setw(3) << *it;
 blockCount++ ;

 it++;
 if (it != myList.end())
 {
 out << ',';
 if (blockCount > 0 && blockCount % BLOCKS_PER_LINE == 0)
 out << endl;
 }
 }
}

---- Definition of operator+()
BigInt BigInt::operator+(BigInt int2)
{
 BigInt sum;
 short int first, // a block of this object
 second, // a block of int2
 result, // a block in their sum
 carry = 0; // the carry in adding two blocks

 list<short int>::reverse_iterator // to iterate right to left
 it1 = myList.rbegin(), // through 1st list, and
 it2 = int2.myList.rbegin(); // through 2nd list

 while (it1 != myList.rend() || it2 != int2.myList.rend())
 {
 if (it1 != myList.rend())
 {
 first = *it1;
 it1++ ;
 }
 else
 first = 0;
 if (it2 != int2.myList.rend())
 {
 second = *it2;
 it2++ ;
 }
 }
}
```

```
else
 second = 0;

 short int temp = first + second + carry;
 result = temp % MODULUS;
 carry = temp / MODULUS;
 sum.myList.push_front(result);
}

if (carry > 0)
 sum.myList.push_front(carry);

return sum;
}

//--- Definition of operator-()
BigInt BigInt::operator-(BigInt int2)
{
 if (*this < int2)
 return 0;

 // else
 BigInt diff;
 short int first,
 second, // a block of this object
 // a block of int2

 borrow = 0; // 1 if borrow needed, 0 if not
 int result; // difference for one block

 list<short int>::reverse_iterator // to iterate right to left
 it1 = myList.rbegin(), // through 1st list, and
 it2 = int2.myList.rbegin(); // through 2nd list

 while (it1 != myList.rend() || it2 != int2.myList.rend())
 {
 if (it1 != myList.rend())
 {
 first = *it1;
 it1++;
 }
 else
 first = 0;
 if (it2 != int2.myList.rend())
 {
 second = *it2;
 it2++;
 }
 else
 second = 0;

 result = first - second - borrow;
 if (result < 0)
 {
 result += MODULUS;
 borrow = 1;
 }
 }
}
```

```
 else
 borrow = 0;
 diff myList.push_front(result);
}
while (diff myList.front() == 0)
 diff myList.erase(diff myList.begin()));

return diff;
}

//--- Definition of operator*()
BigInt BigInt::operator*(BigInt int2)
{
 BigInt bigZero = BigInt(0);
 if ((*this) == bigZero || int2 == bigZero)
 return bigZero;

 // Both numbers are nonzero
 BigInt finalResult,
 padZeros,
 partResult;

 short int first, // a block of this object
 second, // a block of int2
 result, // a block in their product
 carry = 0; // the carry in adding two blocks

 for (list<short int>::reverse_iterator it2 = int2 myList.rbegin();
 it2 != int2 myList.rend(); it2++)
 {
 carry = 0;
 second = *it2;
 partResult = padZeros;

 for (list<short int>::reverse_iterator it1 = myList.rbegin();
 it1 != myList.rend(); it1++)
 {
 first = *it1;

 unsigned temp = first * second + carry;
 result = temp % MODULUS;
 carry = temp / MODULUS;

 partResult myList.push_front(result);
 }
 if (carry > 0)
 partResult myList.push_front(carry);

 finalResult = finalResult + partResult;
 padZeros myList.push_back(0);
 }

 return finalResult;
}
```

5.

```
/*-- BigInt.h -----

This header file defines the data type BigInt for processing
integers (positive, 0, or negative) of any size.
Basic operations are:
 Constructors
 +: Addition operator
 <: Less-than operator
 ===: Equal-to operator
 --: Subtraction operator
 -: Unary minus
 *: Multiplication operator
 read(): Read a BigInt object
 display(): Display a BigInt object
 <<, >> : Input and output operators
-----*/

#include <iostream>
#include <iomanip> // setfill(), setw()
#include <cmath> // pow
#include <list>

#ifndef BIGINT
#define BIGINT

const int DIGITS_PER_BLOCK = 3,
 MODULUS = (short int)pow(10.0, DIGITS_PER_BLOCK);
class BigInt
{
public:
 /***** Function Members *****/
 /**** Constructors ****/
 BigInt()
 { }
 /*-----
 Default constructor

 Precondition: None
 Postcondition: list<short int>'s constructor was used to build
 this BigInt object.
-----*/
 BigInt(int n);
 /*-----
 Construct BigInt equivalent of n.

 Precondition: n >= 0.
 Postcondition: This BigInt is the equivalent of integer n.
-----*/
 /**** read ****/
 void read(istream & in);
 /*-----
 Read a BigInt.
-----*/
```

```
Precondition: istream in is open and contains blocks of
 nonnegative integers having at most DIGITS_PER_BLOCK digits
 per block.
Postcondition: Blocks have been removed from in and added to
 myList.
-----*/
***** display *****
void display(ostream & out) const;
/*-----
 Display a BigInt.

Precondition: ostream out is open.
Postcondition: The large integer represented by this BigInt
 object has been formatted with the usual comma separators
 and inserted into ostream out.
-----*/
***** addition operator *****
BigInt operator+(BigInt int2);
/*-----
 Add two BigInts.

Precondition: int2 is the second addend.
Postcondition: The BigInt representing the sum of the large
 integer represented by this BigInt object and int2 is
 returned.
-----*/
***** less-than operator *****
bool operator<(BigInt int2);
/*-----
 Compare two BigInts with <.

Precondition: int2 is the second operand.
Postcondition: true if this BigInt object is < int2.
-----*/
***** equal-to operator *****
bool operator==(BigInt int2);
/*-----
 Compare two BigInts with ==.

Precondition: int2 is the second operand.
Postcondition: true if this BigInt object is equal to int2.
-----*/
***** subtraction operator *****
BigInt operator-(BigInt int2);
/*-----
 Compare two BigInts with ==.

Precondition: int2 is the second operand.
Postcondition: The BigInt representing the difference of the
 large integer represented by this BigInt object.
-----*/
```

```
***** unary minus operator *****
BigInt operator-();
/*-----
 Apply unary minus to BigInt.

 Precondition: None
 Postcondition: The BigInt representing the negative of the large
 integer represented by this BigInt object is returned.
-----*/
***** multiplication operator *****
BigInt operator*(BigInt int2);
/*-----
 Multiply two BigInts.

 Precondition: int2 is the second operand.
 Postcondition: The BigInt representing the product of the large
 integer represented by this BigInt object and int2 is returned.
-----*/
private:
 /** Data Members */
 char mySign;
 list<short int> myList;

 // private utility functions

 ***** less-than operator *****
 bool lessFunc(BigInt int2);
 /*-----
 Compare two BigInts with < disregarding sign.
-----*/
 ***** equal-to operator *****
 bool equalFunc(BigInt int2);
 /*-----
 Compare two BigInts with = disregarding sign.
-----*/

 bool isZero();
 /*-----
 Check if BigInt is zero.
-----*/
 BigInt addFunc(BigInt int2);
 /*-----
 Add two BigInts, disregarding sign.
-----*/
 BigInt subtractFunc(BigInt int2);
 /*-----
 Subtract two BigInts, disregarding sign.
-----*/
```

```
BigInt multiplyFunc(BigInt int2);
/*-----
 Multiply two BigInts, disregarding sign.
-----*/
};

// end of BigInt class declaration

//-- Definition of constructor
inline BigInt::BigInt(int n)
{
 mySign = '+';
 if (n < 0)
 {
 mySign = '-';
 n = -n;
 }
 do
 {
 myList.push_front(n % MODULUS);
 n /= MODULUS;
 }
 while (n > 0);
}

//----- Input and output operators
inline istream & operator>>(istream & in, BigInt & number)
{
 number.read(in);
 return in;
}

inline ostream & operator<<(ostream & out, const BigInt & number)
{
 number.display(out);
 return out;
}

//-- Definition of subtraction operator
inline BigInt BigInt::operator-(BigInt int2)
{
 BigInt result;
 result = (*this) + (-int2);
 return result;
}

//-- Definition of operator*()
inline BigInt BigInt::operator*(BigInt int2)
{
 BigInt product = multiplyFunc(int2);
 product.mySign = (mySign == int2.mySign ? '+' : '-');
 return product;
}
```

```
//-- Definition of operator<
inline bool BigInt::operator<(BigInt int2)
{
 if (mySign != int2.mySign)
 return (mySign == '-') && (int2.mySign == '+');
 // neg. < pos

 // else they have the same sign
 if (mySign == '+')
 return lessFunc(int2);
 // else
 return int2.lessFunc(*this);
}

//-- Definition of operator==
inline bool BigInt::operator==(BigInt int2)
{
 return mySign == int2.mySign && equalFunc(int2);
}

//-- Definition of isZero()
inline bool BigInt::isZero()
{
 return (myList.size() == 1 && myList.front() == 0);
}

//-- Definition of unary minus operator
inline BigInt BigInt::operator-()
{
 BigInt negative;
 negative.myList = myList;
 if (mySign == '+')
 negative.mySign = '-';
 else
 negative.mySign = '+';
 return negative;
}

// Definition of equalFunc()
inline bool BigInt::equalFunc(BigInt int2)
{ return myList == int2.myList; }

// Definition of lessFunc()
inline bool BigInt::lessFunc(BigInt int2)
{
 return
 (myList.size() < int2.myList.size())
 || (myList.size() == int2.myList.size()
 && myList < int2.myList);
}

#endif
```

```
/*-- BigInt.cpp-----
 This file implements BigInt member functions.
-----*/
#include <iostream>
using namespace std;

#include "BigInt.h"

//--- Definition of read()
void BigInt::read(istream & in)
{
 static bool instruct = true;
 if (instruct)
 {
 cout << "Enter the sign of the big integer (+ is optional) followed "
 "by\n" << DIGITS_PER_BLOCK << "-digit blocks, separated"
 " by spaces.\nEnter a negative integer in last block to"
 " signal the end of input.\n\n";
 }

 char sign;
 cin >> sign;
 if (sign == '-')
 mySign = '-';
 else
 {
 mySign = '+';
 if (sign != '+')
 cin.putback(sign);
 }

 short int block;
 for (;;)
 {
 in >> block;
 if (block < 0) return;

 if (block >= MODULUS)
 cerr << "Illegal block -- " << block << " -- ignoring\n";
 else
 myList.push_back(block);
 }
}

//--- Definition of display()
void BigInt::display(ostream & out) const
{
 out << mySign;
 int blockCount = 0;
 const int BLOCKS_PER_LINE = 20; // number of blocks to display per line

 for (list<short int>::const_iterator it = myList.begin(); ;)
 {
 out << setfill('0');
 if (blockCount == 0)
 out << setfill(' ');
 blockCount++;
 if (blockCount % BLOCKS_PER_LINE == 0)
 out << endl;
 }
}
```

```
if (it == myList.end())
 return;

 out << setw(3) << *it;
 blockCount++ ;

 it++;
 if (it != myList.end())
 {
 out << ',';
 if (blockCount > 0 && blockCount % BLOCKS_PER_LINE == 0)
 out << endl;
 }
}

//--- Definition of addFunc()
BigInt BigInt::addFunc(BigInt int2)
{
 BigInt sum;
 short int first, // a block of this object
 second, // a block of int2
 result, // a block in their sum
 carry = 0; // the carry in adding two blocks

 list<short int>::reverse_iterator // to iterate right to left
 it1 = myList.rbegin(), // through 1st list, and
 it2 = int2.myList.rbegin(); // through 2nd list

 while (it1 != myList.rend() || it2 != int2.myList.rend())
 {
 if (it1 != myList.rend())
 {
 first = *it1;
 it1++ ;
 }
 else
 first = 0;
 if (it2 != int2.myList.rend())
 {
 second = *it2;
 it2++ ;
 }
 else
 second = 0;

 short int temp = first + second + carry;
 result = temp % MODULUS;
 carry = temp / MODULUS;
 sum.myList.push_front(result);
 }
 if (carry > 0)
 sum.myList.push_front(carry);

 return sum;
}
```

```
//--- Definition of operator+()
BigInt BigInt::operator+(BigInt int2)
{
 if (isZero())
 return int2;
 if (int2.isZero())
 return *this;

 BigInt answer; // the sum

 // if signs are same, do the addition
 if (mySign == int2.mySign)
 {
 answer = addFunc(int2);
 answer.mySign = mySign;
 }

 // if signs are different, digits the same
 // one is negative of other
 else if (equalFunc(int2))
 return BigInt(0);

 // if signs and digits are different, use subtraction
 else
 {
 if (mySign == '+')
 {
 if (!lessFunc(int2)) // +7 + -4 = +(7 - 4)
 {
 answer = subtractFunc(int2);
 answer.mySign = '+';
 }
 else // +4 + -7 = -(7 - 4)
 {
 answer = int2.subtractFunc(*this);
 answer.mySign = '-';
 }
 }
 else
 {
 if (!lessFunc(int2)) // -7 + +4 = -(7 - 4)
 {
 answer = subtractFunc(int2);
 answer.mySign = '-';
 }
 else // -4 + +7 = +(7 - 4)
 {
 answer = int2.subtractFunc(*this);
 answer.mySign = '+';
 }
 }
 }
 return answer;
}
```

```
//--- Definition of subtractFunc()
BigInt BigInt::subtractFunc(BigInt int2)
{
 BigInt diff;
 short int first, // a block of 1st operand (this object)
 second, // a block of 2nd operand (int2)
 borrow = 0, // 1 if borrow needed, 0 if not
 result; // difference for one block

 list<short int>::reverse_iterator // to iterate right to left
 it1 = myList.rbegin(), // through 1st list, and
 it2 = int2.myList.rbegin(); // through 2nd list

 while (it1 != myList.rend() || it2 != int2.myList.rend())
 {
 if (it1 != myList.rend())
 {
 first = *it1;
 it1++;
 }
 else
 first = 0;
 if (it2 != int2.myList.rend())
 {
 second = *it2;
 it2++;
 }
 else
 second = 0;

 result = first - second - borrow;
 if (result < 0)
 {
 result += MODULUS;
 borrow = 1;
 }
 else
 borrow = 0;
 diff.myList.push_front(result);
 }
 while (diff.myList.front() == 0)
 diff.myList.erase(diff.myList.begin());

 return diff;
}

//--- Definition of multiplyFunc()
BigInt BigInt::multiplyFunc(BigInt int2)
{
 BigInt bigZero = BigInt(0);
 if ((*this) == bigZero || int2 == bigZero)
 return bigZero;

 // Both numbers are nonzero
 BigInt finalResult,
 padZeros,
 partResult;
```

```
short int first, // a block of this object
 second, // a block of int2
 result, // a block in their product
 carry = 0; // the carry in adding two blocks

for (list<short int>::reverse_iterator it2 = int2.myList.rbegin();
 it2 != int2.myList.rend(); it2++)
{
 carry = 0;
 second = *it2;
 partResult = padZeros;

 for (list<short int>::reverse_iterator it1 = myList.rbegin();
 it1 != myList.rend(); it1++)
 {
 first = *it1;

 unsigned temp = first * second + carry;
 result = temp % MODULUS;
 carry = temp / MODULUS;

 partResult.myList.push_front(result);
 }
 if (carry > 0)
 partResult.myList.push_front(carry);

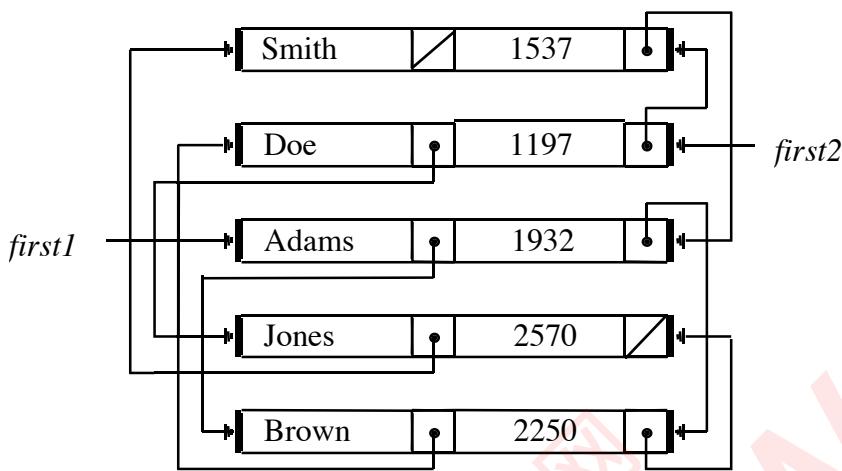
 finalResult = finalResult + partResult;
 padZeros.myList.push_back(0);
}

return finalResult;
}
```

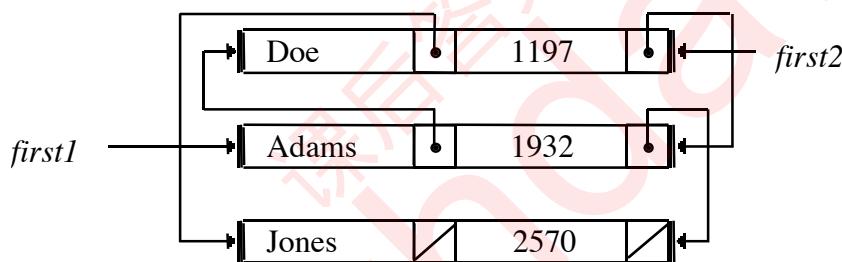
6. This is straightforward, using the ideas and code from Section 11.3. The only list operations needed by the `BigInt` class in Figure 11.3 are:
- a push-back operation, which is an insert-at-the-end operation
  - iterator operations which can be easily replaced by pointer operations (or an iterator inner class can be build) as described in Section 11.3

**Exercises 11.5**

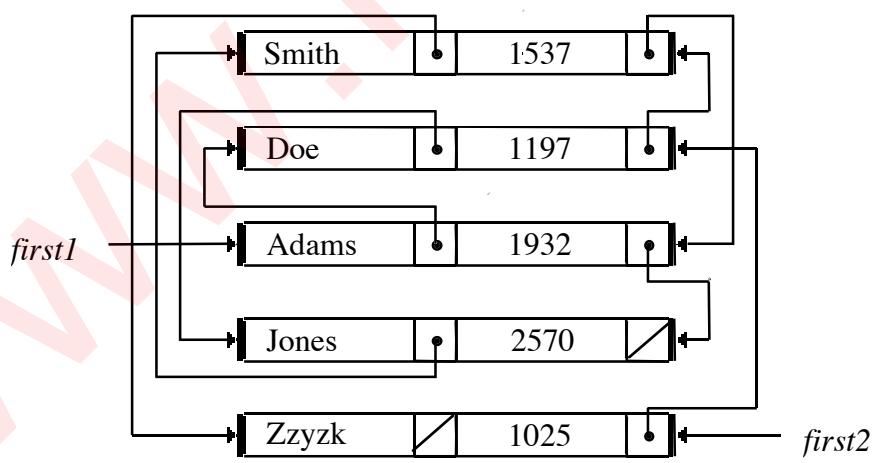
1.



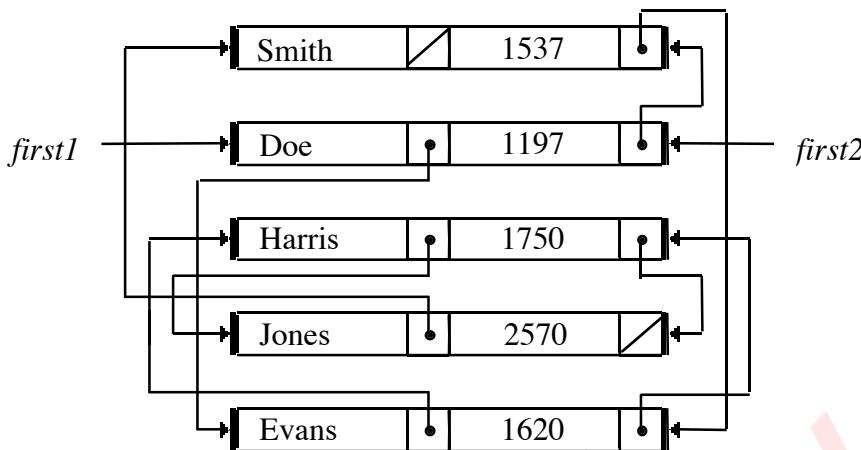
2.



3.



4.



5. A class template for such a multiply-ordered linked list could have the following form:

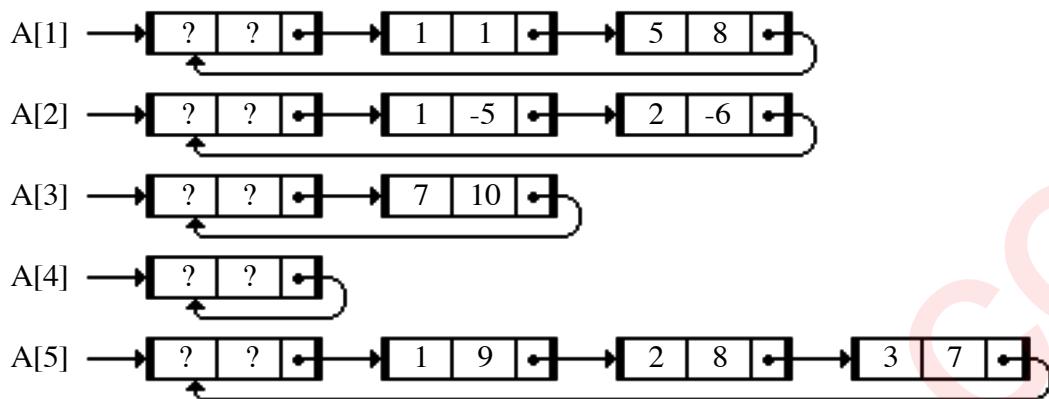
```
template <typename DataType1, typename DataType2>
class MultiplyOrderedList
{
private:
 // Node class
 class Node
 {
public:
 DataType1 data1;
 DataType2 data2;
 Node * next1;
 Node * next2;

 // Node constructor and perhaps other Node ops
 };
 typedef Node * NodePointer;

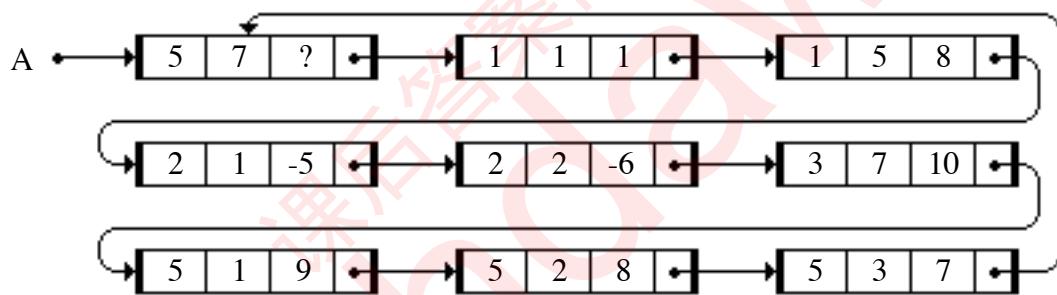
public:
 // List operations declared here

private:
 // Data members
 NodePointer first1, first2;
};
```

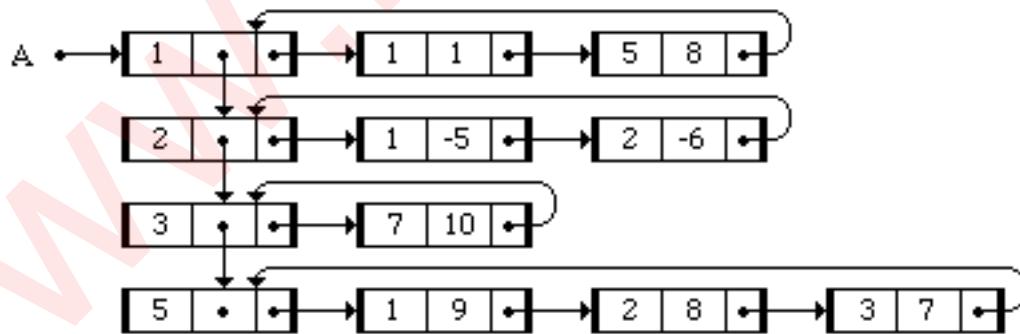
6.



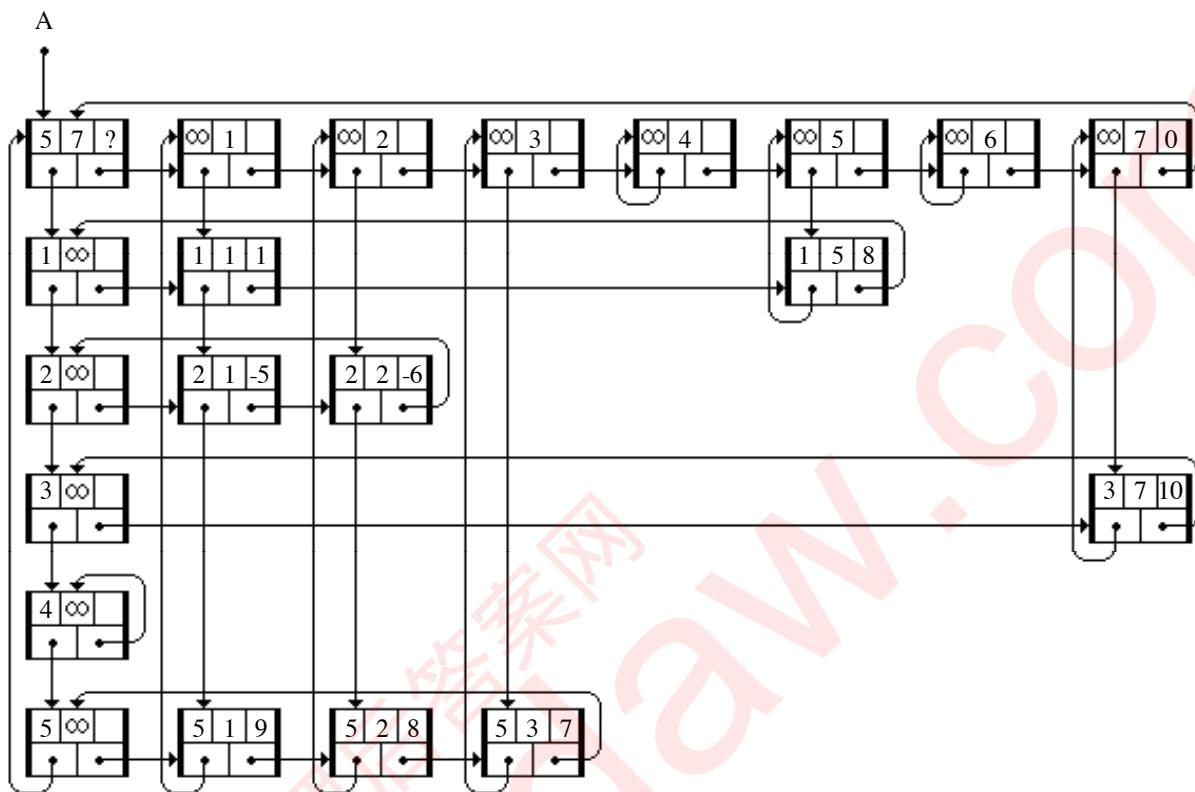
7.



8.



9.



10. See solution to Exercises 18-19.

11.

```

template <typename DataType>
class MatrixLinkedListRep
{
private:
 // Node class
 class Node
 {
public:
 int row,
 col;
 DataType value;
 Node * next;

 // Node constructor and perhaps other Node ops
 Node(int rowNum = 0, int colNum = 0,
 DataType dataVal = 0, Node * ptr = 0)
 {
 row = rowNum;
 col = colNum;
 value = dataVal;
 next = ptr; }
 };

 typedef Node * NodePointer;

```

```
public:
 //-- Constructor
 MatrixLinkedListRep(int numRows = 0, int numColumns = 0)
 {
 myFirst = new Node(numRows, numColumns); // head node
 myFirst ->next = myFirst;
 }

 // -- Other list operations

private:
 // Data members

 NodePointer myFirst;
};

12.
template <typename DataType>
class MatrixLinkedListOfRowListsRep
{
private:
 // Node class
 class Node
 {
 public:
 int col;
 DataType value;
 Node * next;

 // Node constructor and perhaps other Node ops
 Node(int colNum = 0, DataType dataVal, Node * ptr = 0)
 {
 col = colNum;
 value = dataVal;
 next = ptr;
 }
 };
 typedef Node * NodePointer;

 // HeadNode class
 class HeadNode
 {
 public:
 int row;
 NodePointer firstInRow;
 HeadNode * nextHead;

 // HeadNode constructor and perhaps other HeadNode ops
 HeadNode(int rowNum = 0, NodePointer rowPointer, HeadNode * ptr = 0)
 {
 row = rowNum;
 firstInRow = rowPointer;
 nextHead = ptr;
 }
 };
 typedef HeadNode * HeadNodePointer;
```

```
public:
 //-- Constructor
MatrixLinkedListOfRowListsRep()
{
 myFirst = 0;
}
// -- Other list operations

private:
 // Data members
 HeadNodePointer myFirst;
};

13.
template <typename DataType>
class MatrixOrthogonalListRep
{
private:
 // Node class
 class Node
 {
 public:
 int row,
 col;
 DataType value;
 Node * right;
 Node * down;

 // Node constructor and perhaps other Node ops
 Node(int rowNum = 0, int colNum = 0, DataType dataVal,
 Node * rowPtr = 0; Node * colPtr = 0)
 {
 row = rowNum;
 col = colNum;
 value = dataVal;
 right = rowPtr;
 down = colPtr;
 }
 };
 typedef Node * NodePointer;

public:
 //-- Constructor
 MatrixOrthogonalListRep(int numRows = 0, int numColumns = 0)
 {
 myFirst = newNode(numRows, numColumns, 0);
 NodePointer prevptr, newptr;

 // Set up column head nodes
 prevptr = myFirst;
```

```

 for (int i = 1; i <= numColumns; i++)
 {
 newptr = new Node(INFINITY, i, 0);
 newptr->right = newptr->down = newptr;
 prevptr->right = newptr;
 prevptr = newptr;
 }
 newptr->right = firstptr; // make it circular

 // Set up row head nodes
 prevptr = myFirst;
 for (int i = 1; i <= numRows; i++)
 {
 newptr = new Node(INFINITY, i, 0);
 newptr->right = newptr->down = newptr;
 prevptr->down = newptr;
 prevptr = newptr;
 }
 newptr->down = firstptr; // make it circular
}

// -- Other list operations

private:
 static const int INFINITY = INT_MAX;
 // Data members
 NodePointer myFirst;
};

```

14. See solution to Exercises 18-19.

15. // Declare operator+ as a friend function

```

template <typename DataType>
MatrixLinkedListOfRowListsRep<DataType> operator+
 (const MatrixLinkedListOfRowListsRep<DataType> & A,
 const MatrixLinkedListOfRowListsRep<DataType> & B)
{
 MatrixLinkedListOfRowListsRep<DataType>
 C(A.myFirst->row, B.myFirst->row); // sum

 if (A.myFirst->row != B.myFirst->row ||
 A.myFirst->col != B.myFirst->col)
 {
 cerr << "Can't add matrices with different sizes.\n";
 return C;
 }

 MatrixLinkedListOfRowListsRep<DataType>::NodePointer
 Aptr = A.myFirst->next, Bptr = B.myFirst->next,
 Cptr = C.myFirst, temp;
 DataType sum;

```

```
while (Aptr != A.myFirst && Bptr != B.myFirst)
{
 if (Aptr->row == Bptr->row)
 {
 if (Aptr->col == Bptr->col)
 {
 sum = Aptr->value + Bptr->value;
 if (sum != 0)
 {
 temp = new MatrixLinkedListOfRowListsRep<DataType>::Node(Aptr->row, Aptr->col, sum);
 Cptr->next = temp;
 Cptr = temp;
 }
 Aptr = Aptr->next;
 Bptr = Bptr->next;
 }
 else if (Aptr->col < Bptr->col)
 {
 temp = new MatrixLinkedListOfRowListsRep<DataType>::Node(Aptr->row, Aptr->col, Aptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Aptr = Aptr->next;
 }
 else // Aptr->col > Bptr->col
 {
 temp = new MatrixLinkedListOfRowListsRep<DataType>::Node(Bptr->row, Bptr->col, Bptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Bptr = Bptr->next;
 }
 }
 else if (Aptr->row < Bptr->row)
 {
 temp = new MatrixLinkedListOfRowListsRep<DataType>::Node(Aptr->row, Aptr->col, Aptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Aptr = Aptr->next;
 }
 else // (Aptr->row > Bptr->row)
 {
 temp = new MatrixLinkedListOfRowListsRep<DataType>::Node(Bptr->row, Bptr->col, Bptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Bptr = Bptr->next;
 }
}
```

```

while (Aptr != A.myFirst)
{
 temp = new MatrixLinkedListOfRowListsRep<DataType>::
 Node(Aptr->row, Aptr->col, Aptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Aptr = Aptr->next;
}

while (Bptr != B.myFirst)
{
 temp = new MatrixLinkedListOfRowListsRep<DataType>::
 Node(Bptr->row, Bptr->col, Bptr->value);
 Cptr->next = temp;
 Cptr = temp;
 Bptr = Bptr->next;
}

return C;
}

```

16. This is similar to the solution in Exercise 14 (see Ex. 18), but instead of running an index through the array of head nodes, run a pointer through the list of head nodes.

17.

```

template <typename DataType>
class MatrixOrthogonalListRep
{
private:
 // Node class
 class Node
 {
public:
 int row,
 col;
 DataType value;
 Node * right;
 Node * down;

 // Node constructor and perhaps other Node ops
 Node(int rowNum = 0, int colNum = 0, DataType dataVal = 0,
 Node * rowPtr = 0; Node * colPtr = 0)
 {
 row = rowNum;
 col = colNum;
 value = dataVal;
 right = rowPtr;
 down = colPtr;
 }
};

typedef Node * NodePointer;

public:

```

```
//-- Constructor
MatrixOrthogonalListRep(int numRows = 0, int numColumns = 0)
{
 myFirst = newNode(numRows, numColumns);
 NodePointer prevptr, newptr;

 // Set up column head nodes
 prevptr = myFirst;
 for (int i = 1; i <= numColumns; i++)
 {
 newptr = new Node(INFINITY, i);
 newptr->right = newptr->down = newptr;
 prevptr->right = newptr;
 prevptr = newptr;
 }
 newptr->right = firstptr; // make it circular

 // Set up row head nodes
 prevptr = myFirst;
 for (int i = 1; i <= numRows; i++)
 {
 newptr = new Node(INFINITY, i);
 newptr->right = newptr->down = newptr;
 prevptr->down = newptr;
 prevptr = newptr;
 }
 newptr->down = firstptr; // make it circular
}

// -- Other list operations

private:
 static const int INFINITY = INT_MAX;
 // Data members
 NodePointer myFirst;
};

// Declare operator+ as a friend function
template <typename DataType>
MatrixOrthogonalListRep<DataType> operator+
(
 const MatrixOrthogonalListRep<DataType> & A,
 const MatrixOrthogonalListRep<DataType> & B)
{
 MatrixOrthogonalListRep<DataType>
 C(A.myFirst->row, B.myFirst->row); // sum

 if (A.myFirst->row != B.myFirst->row ||
 A.myFirst->col != B.myFirst.col)
 {
 cerr << "Can't add matrices with different sizes.\n";
 return C;
 }

 MatrixOrthogonalListRep<DataType>::NodePointer
 Aptr = A.myFirst->down, Bptr = B.myFirst->down,
 Cdownptr = C.myFirst->down, Crightptr,temp;
 DataType sum;
```

```

while (Aptr != A.myFirst)
{
 Aptr = Aptr->right;
 Bptr = Bptr->right;
 Crightptr = Cdownptr;
 while (Aptr->col != MatrixOrthogonalListRep<DataType>::INFINITY ||
 Bptr->col != MatrixOrthogonalListRep<DataType>::INFINITY)
 {
 if (Aptr->col == Bptr->col)
 {
 sum = Aptr->value + Bptr->value;
 if (sum != 0)
 {
 temp = new MatrixOrthogonalListRep<DataType>:::
 Node(Aptr->row, Aptr->col, sum);
 Crightptr->right = temp;
 Crightptr = temp;
 }
 Aptr = Aptr->right;
 Bptr = Bptr->right;
 }
 else if (Aptr->col < Bptr->col)
 {
 temp = new MatrixOrthogonalListRep<DataType>:::
 Node(Aptr->row, Aptr->col, Aptr->value);
 Crightptr->next = temp;
 Crightptr = temp;
 Aptr = Aptr->right;
 }
 else // Aptr->col > Bptr->col
 {
 temp = new MatrixOrthogonalListRep<DataType>:::
 Node(Bptr->row, Bptr->col, Bptr->value);
 Crightptr->next = temp;
 Crightptr = temp;
 Bptr = Bptr->right;
 }
 }
 Aptr = Aptr->down;
 Bptr = Bptr->down;
 Cdownptr = Cdownptr->down;
}
return C;
}

```

18-19.

```

#include <iostream>
#include <iomanip>
using namespace std;

#ifndef SPARSEMATRIX
#define SPARSEMATRIX

```

```
template <typename DataType>
class SparseMatrix
{
private:
 // Node class
 class Node
 {
public:
 int col;
 DataType value;
 Node * next;

 // Node constructor and perhaps other Node ops
 Node(int colNum = 0, DataType dataVal = 0, Node * ptr = 0)
 {
 col = colNum;
 value = dataVal;
 next = ptr;
 }
};

typedef Node * NodePointer;

public:

 //--- Constructor
 SparseMatrix(int numRows = 0, int numColumns = 0);
 //--- Destructor
 ~SparseMatrix();
 //--- Copy Constructor
 SparseMatrix(const SparseMatrix<DataType> & orig);
 //--- Assignment
 SparseMatrix & operator=(const SparseMatrix<DataType> & orig);

 //--- Addition
 SparseMatrix operator+(SparseMatrix<DataType> right);

 //--- Input/Output
 void read(istream & in);
 void print(ostream & out) const;

 // -- Other list operations

private:
 // Data members
 int myRows, myColumns;
 NodePointer * myArray; // run-time allocated array of lists of nodes
};

//--- Constructor
template <typename DataType>
SparseMatrix<DataType>::SparseMatrix(int numRows = 0, int numColumns = 0)
{
 myRows = numRows;
 myColumns = numColumns;
 myArray = new NodePointer[numRows];
```

```
for (int i = 0; i < myRows; i++)
{
 myArray[i] = new Node(0, 0); // array of pointers to head nodes
 myArray[i]->next = myArray[i];
}
}

//-- Destructor
template <typename DataType>
SparseMatrix<DataType>::~SparseMatrix()
{
 for (int i = 0; i < myRows; i++)
 {
 SparseMatrix<DataType>::NodePointer
 ptr = myArray[i]->next,
 succ;
 while (ptr != myArray[i])
 {
 succ = ptr->next;
 delete ptr;
 ptr = succ;
 }
 delete [] myArray;
 }

 //-- Copy Constructor
 template <typename DataType>
 SparseMatrix<DataType>::
 SparseMatrix(const SparseMatrix<DataType> & orig)
 {
 myRows = orig.myRows;
 myColumns = orig.myColumns;
 myArray = new NodePointer[myRows];
 SparseMatrix<DataType>::NodePointer
 prev,
 last,
 origPtr;

 for (int i = 0; i < myRows; i++)
 {
 myArray[i] = new Node(0, 0); // array of pointers to head nodes
 myArray[i]->next = myArray[i];
 prev = myArray[i];
 origPtr = orig.myArray[i]->next;

 while (origPtr != orig.myArray[i])
 {
 last = new SparseMatrix<DataType>::
 Node(origPtr->col, origPtr->value, myArray[i]);
 prev->next = last;
 prev = last;
 origPtr = origPtr->next;
 }
 }
 }
}
```

```
//-- Assignment
template <typename DataType>
SparseMatrix<DataType> & SparseMatrix<DataType>::operator=(const SparseMatrix<DataType> & orig)
{
 if (this != &orig)
 {
 (*this).~SparseMatrix();
 myRows = orig.myRows;
 myColumns = orig.myColumns;
 myArray = new NodePointer[myRows];
 SparseMatrix<DataType>::NodePointer
 prev,
 last,
 origPtr;

 for (int i = 0; i < myRows; i++)
 {
 myArray[i] = new Node(0, 0); // array of pointers to head nodes
 myArray[i]->next = myArray[i];
 prev = myArray[i];
 origPtr = orig.myArray[i]->next;

 while (origPtr != orig.myArray[i])
 {
 last = new SparseMatrix<DataType>::Node(origPtr->col, origPtr->value, myArray[i]);
 prev->next = last;
 prev = last;
 origPtr = origPtr->next;
 }
 }
 }
 return *this;
}

//-- Addition
template <typename DataType>
SparseMatrix<DataType> SparseMatrix<DataType>::operator+(SparseMatrix<DataType> second)
{
 SparseMatrix<DataType> sum(myRows, myColumns);

 if (myRows != second.myRows || myColumns != second.myColumns)
 cerr << "+ not defined for different size matrices.\n";
 else
 {
 SparseMatrix<DataType>::NodePointer
 ptrSum,
 ptr1, ptr2,
 temp;
 DataType sumEntry;
 int colNumber;
```

```
for (int i = 0; i < myRows; i++)
{
 sum.myArray[i] = new SparseMatrix<DataType>::Node(0, 0);
 sum.myArray[i]->next = sum.myArray[i];
 ptrSum = sum.myArray[i];
 ptr1 = myArray[i]->next;
 ptr2 = second.myArray[i]->next;
 while (ptr1 != myArray[i] && ptr2 != second.myArray[i])
 {
 if (ptr1->col == ptr2->col)
 {
 colNumber = ptr1->col;
 sumEntry = ptr1->value + ptr2->value;
 ptr1 = ptr1->next;
 ptr2 = ptr2->next;
 }
 else if (ptr1->col > ptr2->col)
 {
 colNumber = ptr2->col;
 sumEntry = ptr2->value;
 ptr2 = ptr2->next;
 }
 else // ptr1->col < ptr2->col
 {
 colNumber = ptr1->col;
 sumEntry = ptr1->value;
 ptr1 = ptr1->next;
 }
 if (sumEntry != 0)
 {
 temp = new SparseMatrix<DataType>::
 Node(colNumber, sumEntry, sum.myArray[i]);
 ptrSum->next = temp;
 ptrSum = temp;
 }
 }
 while (ptr1 != myArray[i]) // copy rest of first
 {
 temp = new SparseMatrix<DataType>::
 Node(ptr1->col, ptr1->value, sum.myArray[i]);
 ptrSum->next = temp;
 ptrSum = temp;
 ptr1 = ptr1->next;
 }
 while (ptr2 != second.myArray[i]) // copy rest of second
 {
 temp = new SparseMatrix<DataType>::
 Node(ptr2->col, ptr2->value, sum.myArray[i]);
 ptrSum->next = temp;
 ptrSum = temp;
 ptr2 = ptr2->next;
 }
}
return sum;
}
```

```
//-- Input
template <typename DataType>
void SparseMatrix<DataType>::read(istream & in)
{
 if (myRows == 0 && myColumns == 0)
 {
 cout << "# or rows and columns in matrix? ";
 cin >> myRows >> myColumns;
 }

 cout << "Enter your " << myRows << " x "
 << myColumns << " matrix rowwise.\nColumn"
 " numbers must be in increasing order.\nA negative "
 " column number signals the end of the row.\n\n";

 SparseMatrix<DataType>::NodePointer ptr, prev;
 for (int i = 0; i < myRows; i++)
 {
 myArray[i] = new SparseMatrix<DataType>::Node(0, 0);
 myArray[i]->next = myArray[i];
 prev = myArray[i];

 cout << "Column #s and data values for row " << i << ":\n";
 int col;
 DataType dataVal;
 for (;;)
 {
 in >> col >> dataVal;
 if (col < 0) break;
 if (col >= myColumns)
 cerr << "Column number too large\n";
 else
 {
 ptr = new SparseMatrix<DataType>::Node(col, dataVal, myArray[i]);
 prev->next = ptr;
 prev = ptr;
 }
 }
 }
}

template <typename DataType>
inline istream & operator>>(istream & in,
 SparseMatrix<DataType> & mat)
{ mat.read(in); return in; }

//-- Output
template <typename DataType>
void SparseMatrix<DataType>::print(ostream & out) const
{
 const int OUTPUT_ZONE = 8;

 SparseMatrix<DataType>::NodePointer ptr;
```

```
for (int i = 0; i < myRows; i++)
{
 ptr = myArray[i]->next;
 for (int j = 0; j < myColumns; j++)
 {
 if (ptr == 0 || ptr->col != j)
 out << setw(OUTPUT_ZONE) << 0;
 else
 {
 out << setw(OUTPUT_ZONE) << ptr->value;
 ptr = ptr->next;
 }
 }
 out << endl << endl;
}

template <typename DataType>
inline ostream & operator<<(ostream & out,
 const SparseMatrix<DataType> & mat)
{ mat.print(out); return out; }

#endif

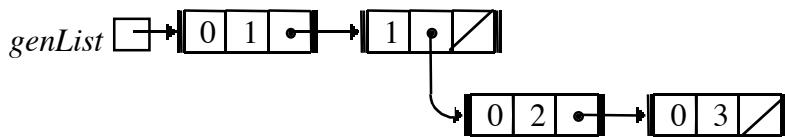
#include "SparseMatrix.h"
#include <iostream>
using namespace std;

int main()
{
 SparseMatrix<int> a(4, 5), b;
 cout << "Enter a: \n";
 cin >> a;
 cout << endl << a << endl;
 cout << "Enter b: \n";
 cin >> b;
 cout << endl << b << endl;

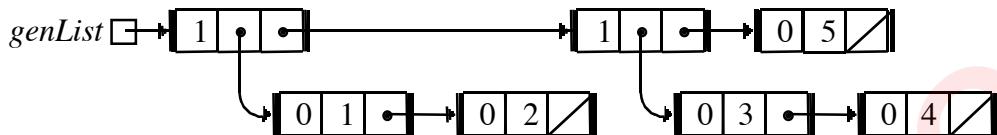
 SparseMatrix<int> c = a;
 cout << "Test copy constructor -- Copy of a:\n";
 cout << c << endl;
 c = b;
 cout << "Test assignment c = b: c =\n";
 cout << c << endl;

 cout << "Test addition -- Enter matrix c to add to b:\n";
 cin >> c;
 cout << "c = \n";
 cout << c << endl;
 cout << "b + c = \n" << b + c << endl;
}
```

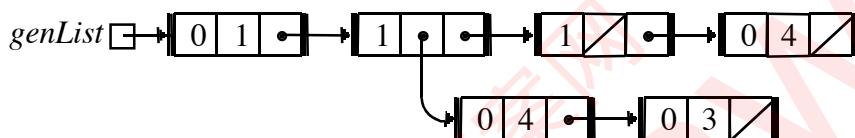
20.



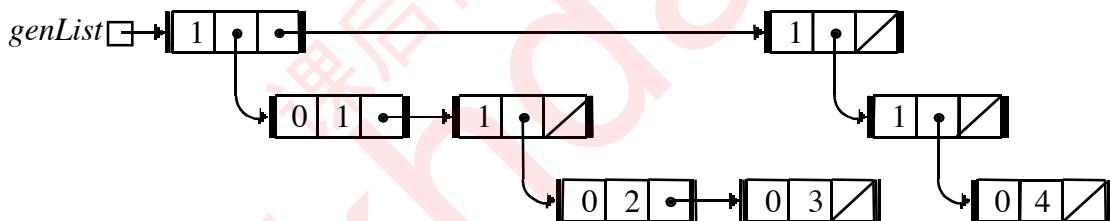
21.



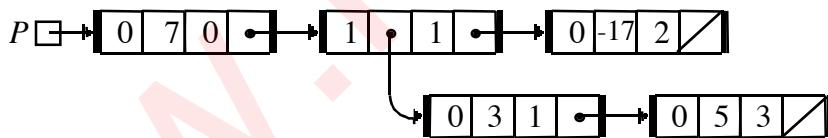
22.



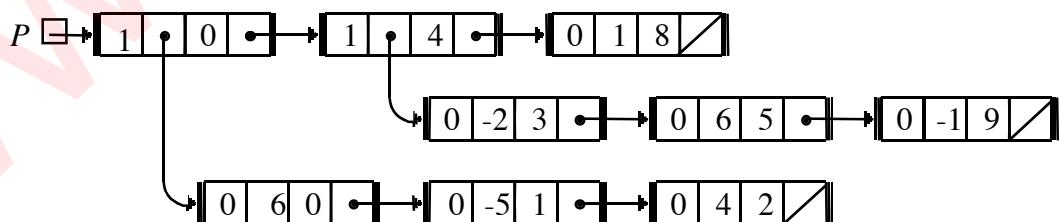
23.



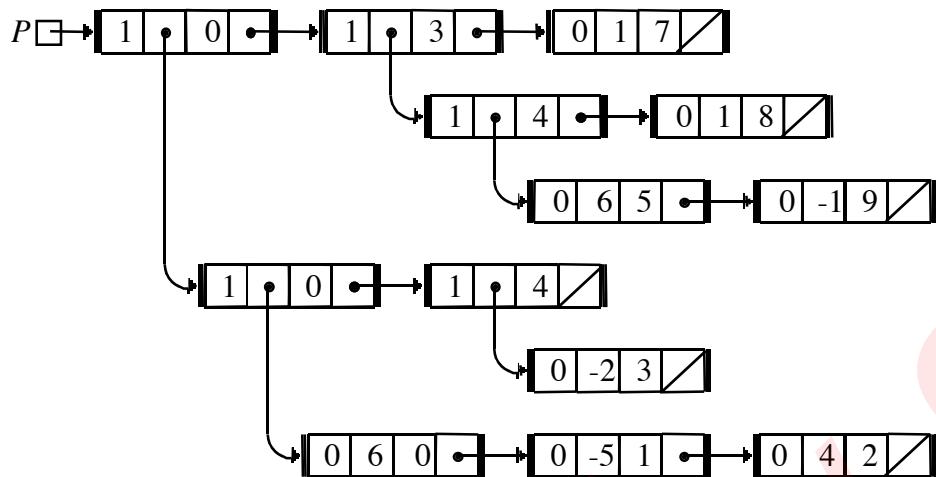
$$24. P(x, y) = 7 + 3xy + 5x^3y - 17y^2$$



$$25. P(x, y) = 6 - 5x + 4x^2 - 2x^3y^4 + 6x^5y^4 - x^9y^4 + y^8$$



$$26. P(x, y, z) = 6 - 5x + 4x^2 - 2x^3y^4 + 6x^5y^4z^3 - x^9y^4z^3 + y^8z^3 + z^7$$



## Chapter 12: Searching: Binary Trees and Hash Tables

### Exercises 12.1

1. 4, 6
2. 4, 1, 2, 3
3. 4, 6, 5
4. 4, 1, 0
5. 4, 6, 7, 8
- 6.

```
template <typename ElementType>
void linearSearch(ElementType x[], ElementType item, int capacity,
 int n, bool & found, int & loc)
/*-----
 Linear search a list stored in an array x for an item, which is added
 at the end of the list, to improve performance.

 Precondition: n is the number of items in the array and is less
 than the array's capacity.
 Postcondition: found is true and loc is the position of item if the
 search is successful; otherwise found is false and loc == n.
-----*/
{
 assert(n < capacity);

 x[n] = item;
 loc = 0;
 while (x[loc] != item)
 loc++;

 found = loc != n;
}

7.
template <typename ElementType>
void linearSearch(ElementType x[], ElementType item, int n,
 bool & found, int & loc)
/*-----
 Linear search an ordered list stored in an array x for an item.

 Precondition: n is the number of items stored in the array.
 Postcondition: found is true and loc is the position of item if the
 search is successful; otherwise found is false.
-----*/
{
 loc = 0;
 while (loc < n && item > x[loc])
 loc++;

 found = (loc < n && x[loc] == item);
}
```

8.

```

template <typename ElementType>
void recLinearSearch(ElementType x[], ElementType item, int n,
 int start, bool & found, int & loc)
/*
-----*
 Recursively linear search a list stored in an array x for an item.

Precondition: n is the number of items in the array and
0 <= start <= n. Initial call is with start = 0.
Postcondition: found is true and loc is the position of item if the
search is successful; otherwise found is false and loc == n.
-----*/
{
 loc = start;
 if (start == n)
 found = false;
 else if (item == x[start])
 found = true;
 else
 recLinearSearch(x, item, n, start + 1, found, loc);
}

```

9.

```

template <typename ElementType>
void moveFrontLinearSearch(Node<ElementType> * first, ElementType item,
 bool & found, Node<ElementType> * locptr);
/*
-----*
 Self-organizing linear search of a linked list stored for an item.
Precondition: first points to first node in the linked list.
Postcondition: found is true and locptr is positioned at item
if the search is successful; otherwise found is false and
locptr is a null pointer.
-----*/
{
 locptr = first;
 if (locptr == 0)
 found = false;
 else if (item == locptr->data)
 found = true;
 else
 recLinkedLinearSearch(first->next, item, found, locptr);
}

```

10. The following solution uses an array to store the list so that its  $O(n)$  computing time caused by having to move array elements in the move-to-the-front operation is very visible. Using a `vector`, we could conceal this by using its `insert()` operation.

```

template <typename ElementType>
void moveFrontLinearSearch(ElementType x[], int size n, ElementType item,
 bool & found, int & loc)
/*
-----*
 Self-organizing linear search of a list stored in an array x for
an item.

Precondition: n is the number of items stored in the array.

```

```

Postcondition: found is true, loc is position of item, and item
 is moved to the front of the list if the search is successful;
 otherwise found is false and loc == n.
-----*/
{
 found = false;
 loc = 0;
 while (!found && loc < n)
 {
 if (item == x[loc])
 found = true;
 else
 loc++;
 }
 if (found) // Move to front of list
 {
 for (int i = loc; i > 0; i--)
 x[i] = x[i - 1];
 x[0] = item;
 }
}

11.
template <typename ElementType>
void moveFrontLinearSearch(NodePointer first,
 const ElementType & item,
 bool & found, NodePointer & locptr)
/*
-----*
 Self-organizing linear search of a linked list for an item.

 Precondition: first points to first node in the linked list.
 Postcondition: found is true, locptr positioned at item, and item
 is moved to the front of the list if the search is successful;

{
 found = false;
 locptr = first;
 NodePointer prev = 0;
 for (;;)
 {
 if (found || locptr == 0) return;
 if (item == locptr->data)
 found = true;
 else
 {
 prevptr = locptr;
 locptr = locptr->next;
 }
 }
 if (found && locptr != first) // Move to front
 {
 prev->next = locptr->next;
 locptr->next = first;
 first = locptr;
 }
}

```

12. In the solution for #10, instead of moving all items up to the location where the item is found, simply swap the found item with its predecessor.
13. In the solution for #11, instead of changing links so node moves to front of list, simply swap the values stored in the nodes pointed to be `prev` and `locptr`.

14.

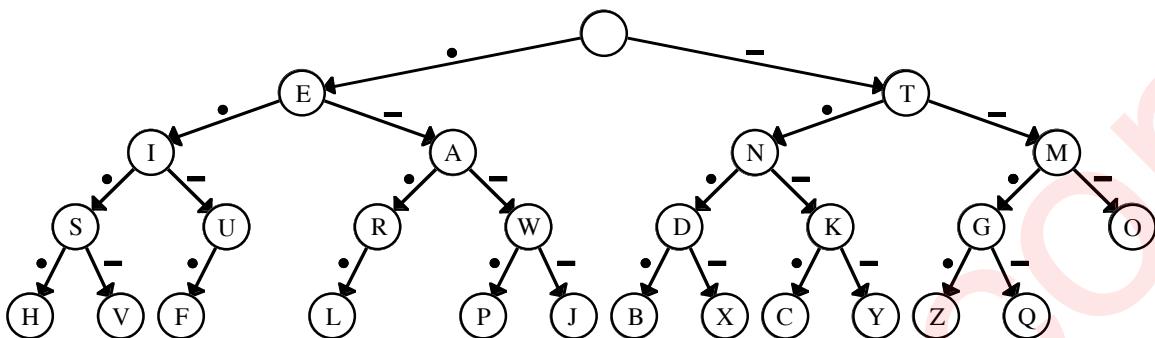
```
template <class ElementType>
void interpolationSearch(const ElementType x[], int n,
 ElementType item, bool & found, long int & loc)
/*-----
 * Linear interpolation of an ordered list stored in an array x for
 * an item.

Precondition: Items in the list are in ascending order; n is
the number of items in the list.
Postcondition: found is true and loc is the position of item
if the search is successful; otherwise found is false.
-----*/
{
 long int first = 0,
 last = n - 1;

 while (x[first] < item && item <= x[last])
 {
 loc = long(first + double(item - x[first]) * (last - first)
 / double(x[last] - x[first]));
 if (item < x[loc])
 last = loc - 1;
 else if (item > x[loc])
 first = loc + 1;
 else // item == x[loc]
 first = loc; // to stop searching;
 }
 found = x[first] == item;
}
```

**Exercises 12.2**

1.



2. H, E, F, G

3. Yes

4. 4

5. A: 3, 2   B: 2, 1   C: 1, 1   D: 1, 0   E, F, G: 0, 0

6. Yes

7.

| i        | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|---|---|---|---|---|---|
| array[i] | A | B | C | D | E | F | G | H |

8. H, I, L, K, G

9. No. Next-to-last level is not completely filled in.

10. 5

11. No. For node E, height of left subtree is 0 and height of right subtree is 2

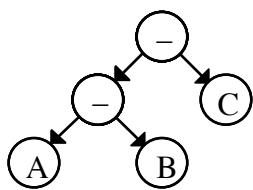
12.

| i        | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|----------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| array[i] | A | B | C | D | E | F | G | H | I | J | K  |    |    |    |    | L  |    |

## Exercises 12.3

1. H, D, B, E, A, F, C, G
2. A, B, D, H, E, C, F, G
3. H, D, E, B, F, G, C, A
4. H, D, I, B, E, L, J, A, F, K, C, G
5. A, B, D, H, I, E, J, L, C, F, K, G
6. H, I, D, L, J, E, B, K, F, G, C, A
7. A, B, C, D, E, F, G, H, I, J, K, L
8. H, D, B, A, C, E, G, F, K, I, J, L
9. A, C, B, F, G, E, D, J, I, L, K, H

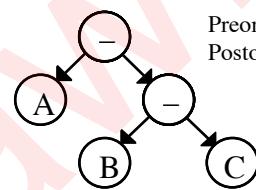
10.



Preorder:  
Postorder:

-- A B C  
A B - C -

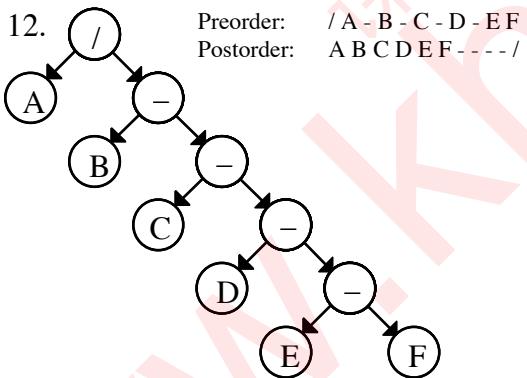
11.



Preorder:  
Postorder:

- A - B C  
A B C --

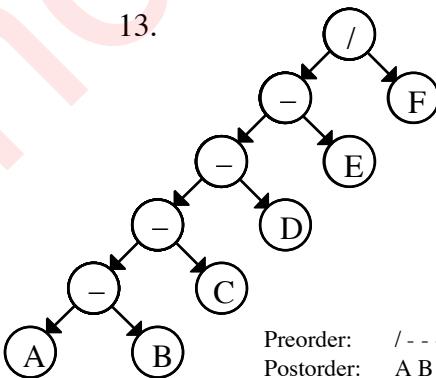
12.



Preorder:  
Postorder:

/ A - B - C - D - E F  
A B C D E F ----- /

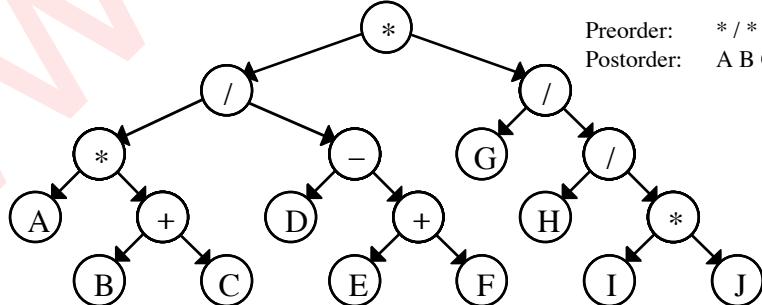
13.



Preorder:  
Postorder:

/ ----- A B C D E F  
A B - C - D - E - F /

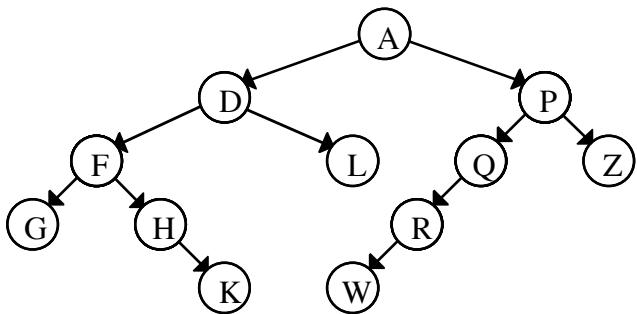
14.



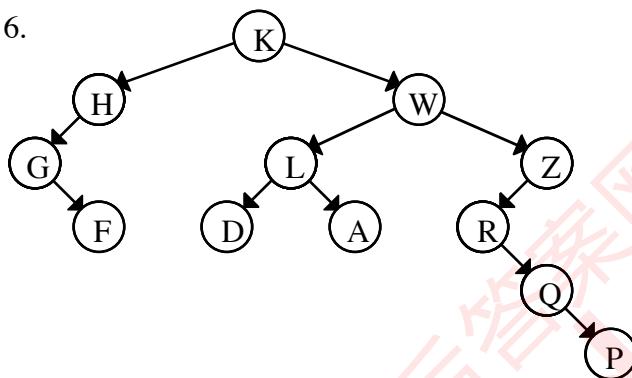
Preorder:  
Postorder:

\* / \* A + B C - D + E F / G / H \* I J  
A B C + \* D E F + - / G H I J \* / \*

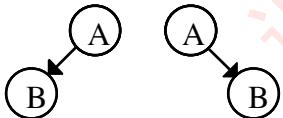
15.



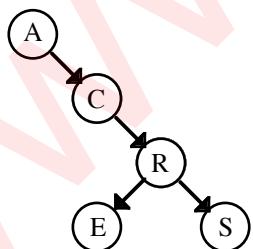
16.



17.

**Exercises 12.4**

1.

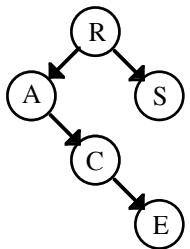


Preorder: A, C, R, E, S

Inorder: A, C, E, R, S

Postorder: E, S, R, C, A

2.

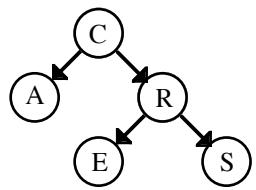


Preorder: R, A, C, E, S

Inorder: A, C, E, R, S

Postorder: E, C, A, S, R

3.

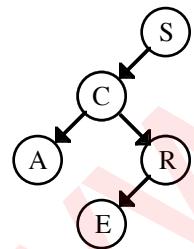


Preorder: C, A, R, E, S

Inorder: A, C, E, R, S

Postorder: A, E, S, R, C

4.

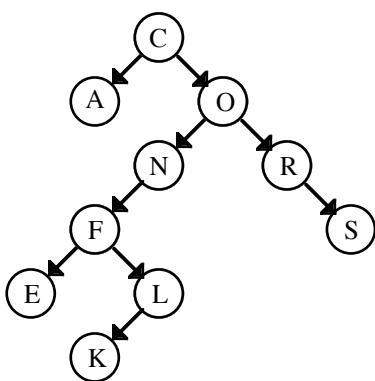


Preorder: S, C, A, R, E

Inorder: A, C, E, R, S

Postorder: A, E, R, C, S

5.

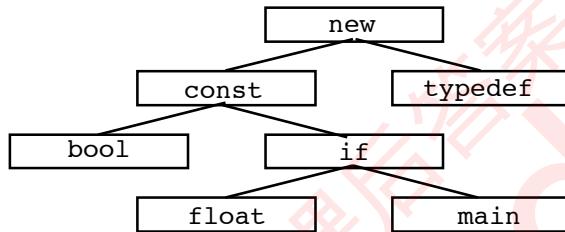


Preorder: C, A, O, N, F, E, L, K, R, S

Inorder: A, C, E, F, K, L, N, O, R, S

Postorder: A, E, K, L, F, N, S, R, O, C

6.

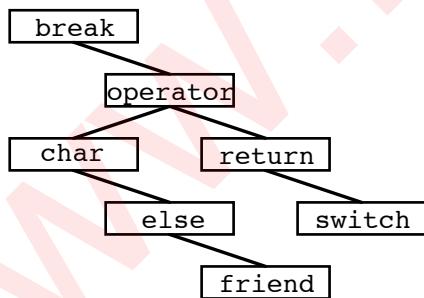


Preorder: new, const, bool, if, float, main, typedef

Inorder: bool, const, float, if, main, new, typedef

Postorder: bool, float, main, if, const, typedef, new

7.

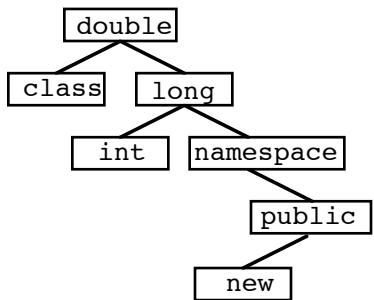


Preorder: break, operator, char, else, friend, return, switch

Inorder: break, char, else, friend, operator, return, switch

Postorder: friend, else, char, switch, return, operator, break

8.

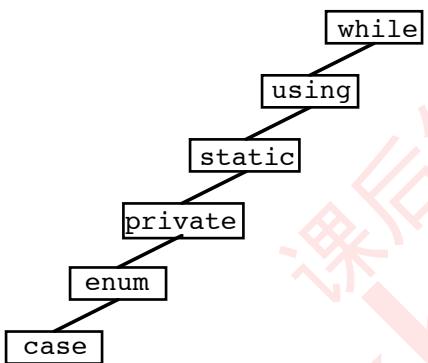


Preorder: double, class, long, int, namespace, public, new

Inorder: class, double, int, long, namespace, new, public

Postorder: class, int, new, public, namespace, long, double

9.

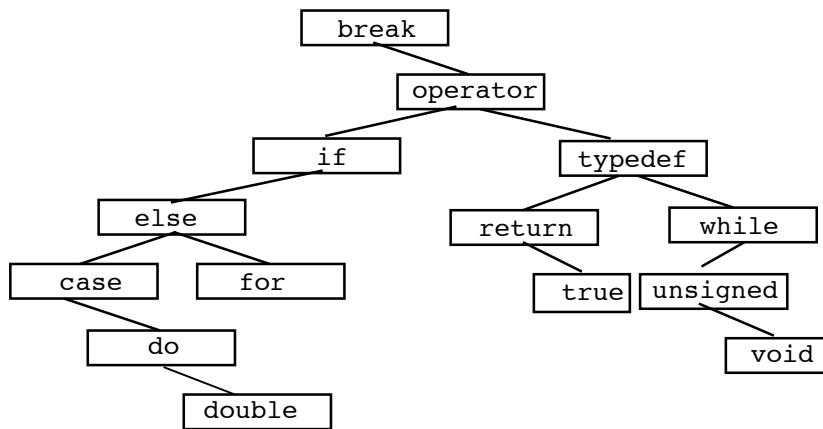


Preorder: while, using, static, private, enum, case

Inorder: case, enum, private, static, using, while

Postorder: same as inorder

10.

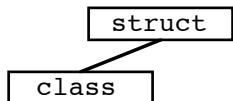


Preorder: break, operator, if, else, case, do, double, for, typedef, return, true, while, unsigned, void

Inorder: break, case, do, double, else, for, if, operator, return, true, typedef, unsigned, void, while

Postorder: double, do, case, for, else, if, true, return, void, unsigned, while, typedef, operator, break

11.

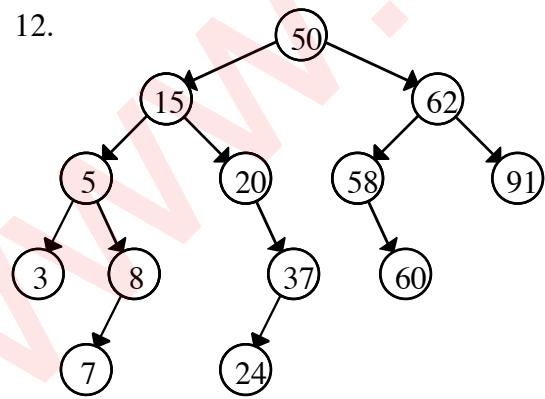


Preorder: struct, class

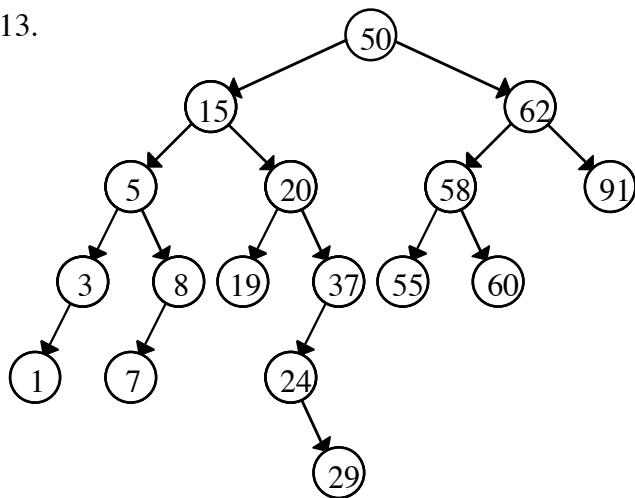
Inorder: class, struct

Postorder: same as inorder

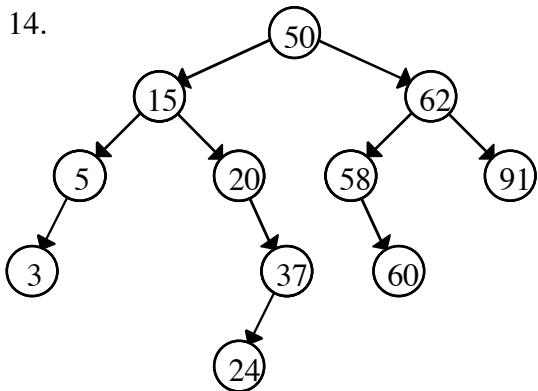
12.



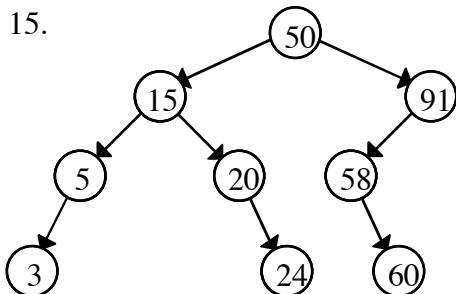
13.



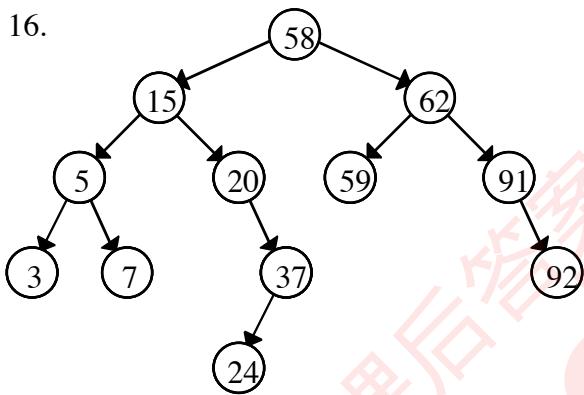
14.



15.



16.



17. 3, 5, 8, 15, 20, 24, 37, 50, 58, 60, 62, 91

18. 50, 15, 5, 3, 8, 20, 37, 24, 62, 58, 60, 91

19. 3, 8, 5, 24, 37, 20, 15, 60, 58, 91, 62, 50

20. LLLL / 3

RUU / 5  
RLU / 8  
RUUU / 15  
RLU / 20  
RLLU / 24  
RUU / 37  
RUUUU / 50  
RLLU / 58  
RLU / 60  
RUUU / 62  
RLU / 91  
RUUUU

21.

```
template <typename DataType>
void BST<DataType>::displayPreorderAux(BinNodePointer ptr,
 ostream & out)
{
 if (ptr != 0)
 {
 out << ptr->data;
 if (ptr->left != 0)
 out << setw(12) << ptr->left->data;
 else
 out << setw(12) << "-";
 if (ptr->right != 0)
 out << setw(12) << ptr->right->data;

 else
 out << setw(12) << "-";
 out << endl;

 displayPreorderAux(ptr->left, out);
 displayPreorderAux(ptr->right, out);
 }
}

#endif
```

22.

```
//-- Find level of item in a BST recursively; returns -1 if not found

template <typename DataType>
inline int BST<DataType>::level(const DataType & item)
{ bool found;
 int itemLevel = levelAux(myRoot, item, found);
 if (!found)
 cerr << item << " not found -- returning -1 for level\n";
 return itemLevel;
}

template <typename DataType>
int BST<DataType>::levelAux(BST<DataType>::BinNodePointer subtreeRoot,
 const DataType & item, bool & found)
{
 if (subtreeRoot == 0)
 {
 found = false;
 return -1;
 }

 int level;
 if (item < subtreeRoot->data)
 {
 level = 1 + levelAux(subtreeRoot->left, item, found);
 if (found)
 return level;
 else
 return -1;
 }
}
```

```
else if (item > subtreeRoot->data)
{
 level = 1 + levelAux(subtreeRoot->right, item, found);
 if (found)
 return level;
 else
 return -1;
}
else
{
 found = true;
 return 0;
}
}
```

23.

```
//-- Find level of item in a BST iteratively; returns -1 if not found

template <typename DataType>
int BST<DataType>::level(const DataType & item)
{
 bool found = false;
 int count = 0;
 BST<DataType>::BinNodePointer ptr = myRoot;

 while (!found && ptr != 0)
 {
 if (item < ptr->data)
 {
 count++;
 ptr = ptr->left;
 }
 else if (item > ptr->data)
 {
 count++;
 ptr = ptr->right;
 }
 else
 found = true;
 }
 if (found)
 return count;
 else
 return -1;
}
```

24.

```
//-- Find height of a BST recursively
inline int BST<DataType>::height()
{ return heightAux(myRoot); }

template <typename DataType>
int BST<DataType>::heightAux(BST<DataType>::BinNodePointer subtreeRoot)
{
 if (subtreeRoot == 0)
```

```

 return 0;
 else if (subtreeRoot->left == 0 && subtreeRoot->right == 0)
 return 1;
 else
 {
 int leftHeight = heightAux(subtreeRoot->left),
 rightHeight = heightAux(subtreeRoot->right);
 if (leftHeight > rightHeight)
 return 1 + leftHeight;
 else
 return 1 + rightHeight;
 }
}

```

25.

```

template <typename DataType>
inline int BST<DataType>::leafCount()
{ return leafCountAux(myRoot); }

template <typename DataType>
int BST<DataType>::leafCountAux(BST<DataType>::BinNodePointer subtreeRoot)
{
 if (subtreeRoot == 0)
 return 0;
 else if (subtreeRoot->left == 0 && subtreeRoot->right == 0)
 return 1;
 else
 return leafCountAux(subtreeRoot->left)
 + leafCountAux(subtreeRoot->right);
}

```

26.

```

/* Nonrecursive version of inorder() -- use a stack to retain
 addresses of nodes. Output statement may be replaced with
 other appropriate action when visiting a node.
*/
#include <iostream>
#include <stack> // or use Stack.h from the text
using namespace std;

template <typename DataType>
void BST<DataType>::nonrecInorder(ostream & out)
{
 stack<BST<DataType>::BinNodePointer> s;
 BST<DataType>::BinNodePointer ptr;
 s.push(myRoot);

 while (!s.empty())
 {
 ptr = s.top();
 s.pop();

 if (ptr != 0)
 {

```

```

 s.push(ptr->right);
 s.push(ptr);
 s.push(ptr->left);
 }
 else if (!s.empty())
 {
 ptr = s.top();
 s.pop();
 out << ptr->data << " ";
 }
}
}

27.
/* Level-by-level traversal -- a queue is used to store
 pointers to nodes. Output statement may be replaced with
 other appropriate action when visiting a node.
*/
#include <iostream>
#include <queue> // or use Queue.h from Chapter 8
using namespace std;

template <typename DataType>
void BST<DataType>::levelByLevel(ostream & out)
{
 queue<BST<DataType>::BinNodePointer> q;
 BST<DataType>::BinNodePointer ptr;
 q.push(myRoot);

 while (!q.empty())
 {
 ptr = q.front();
 q.pop();
 out << ptr->data << " ";
 if (ptr->left != 0)
 q.push(ptr->left);
 if (ptr->right != 0)
 q.push(ptr->right);
 }
}

28.
//--- Recursive version of remove()

template <typename DataType>
void BST<DataType>::remove(const DataType & item)
{ removeAux(item, myRoot); }

//--- Definition of helper function removeAux()
template <typename DataType>
void BST<DataType>::removeAux(const DataType & item,
 BST<DataType>::BinNodePointer & root)
{
 if (root == 0) // empty BST -- item not found
 {

```

```

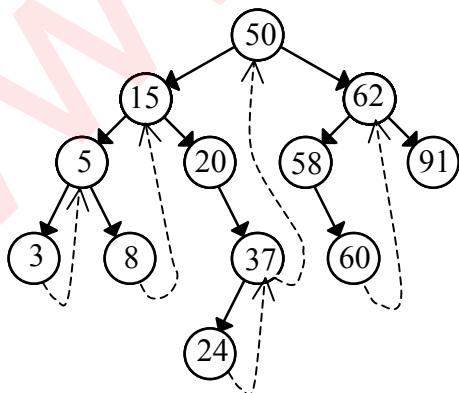
 cerr << "Item not in the BST\n";
 return;
 }
 //else recursively search for the node containing item
 //and remove it from the:
 if (item < root->data) // left subtree
 deleteAux(item, root->left);
 else if (item > root->data) // right subtree
 deleteAux(item, root->right);
 else // item found -- delete node
 {
 BST<DataType>::BinNodePointer ptr; // auxiliary pointer
 if (root->left == 0) // no left child
 {
 ptr = root->right;
 delete root;
 root = ptr;
 }
 else if (root->right == 0) // left child, but no right child
 {
 ptr = root->left;
 delete root;
 root = ptr;
 }
 else // 2 children
 {

 // find inorder successor
 ptr = root->right;
 while (ptr->left != 0)
 ptr = ptr->left;
 // Move contents of successor to the root of the subtree
 // being examined and delete the successor node.
 root->data = ptr->data;
 deleteAux(ptr->data, root->right);
 }
 }
}
}

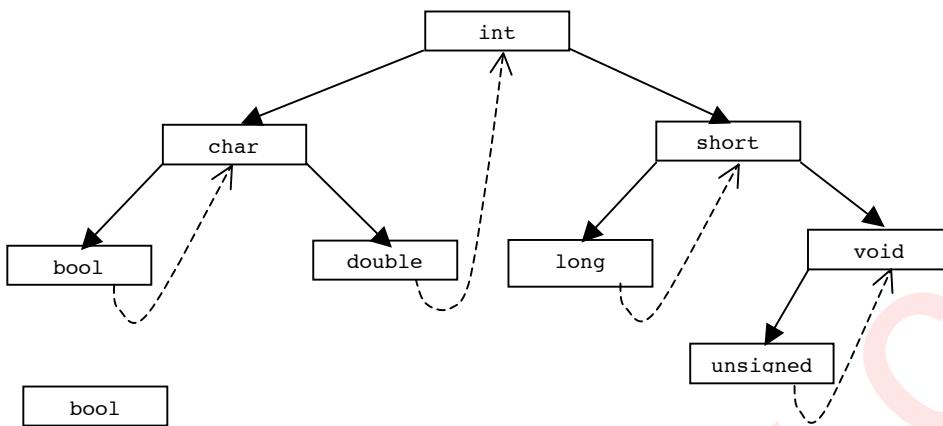
```

## Exercises 12.6

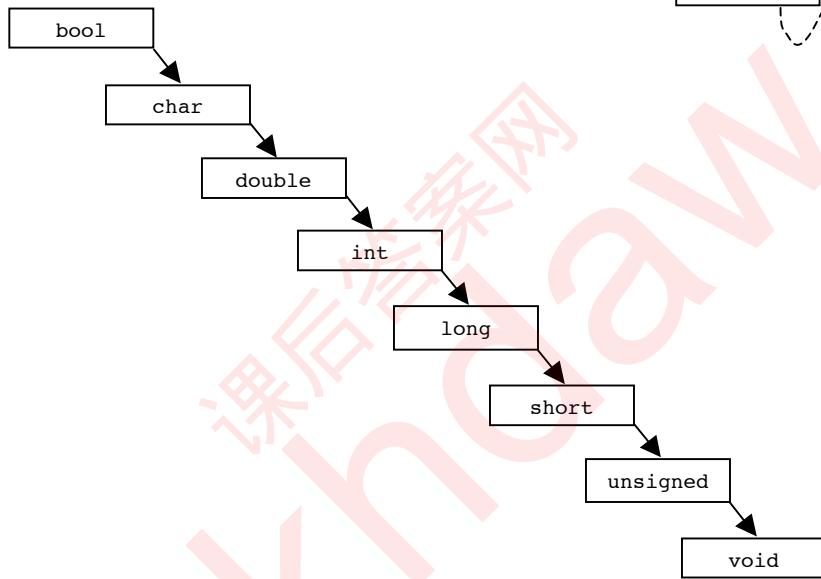
1.



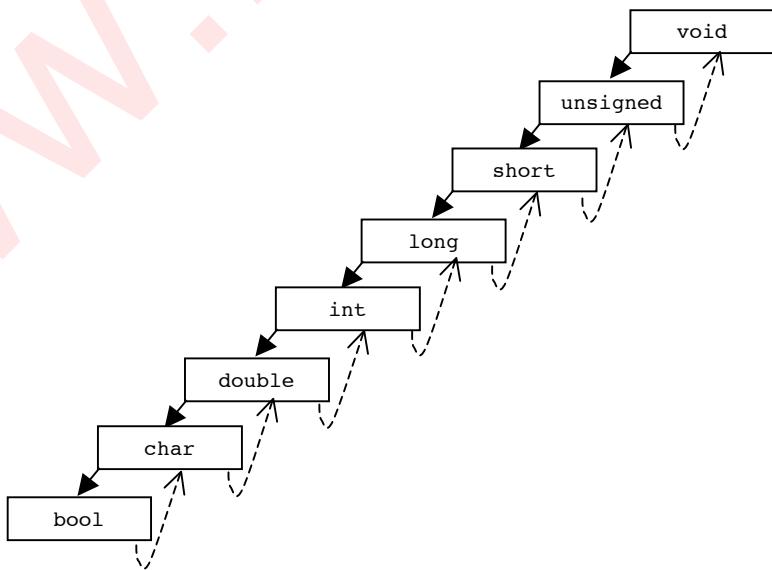
2.



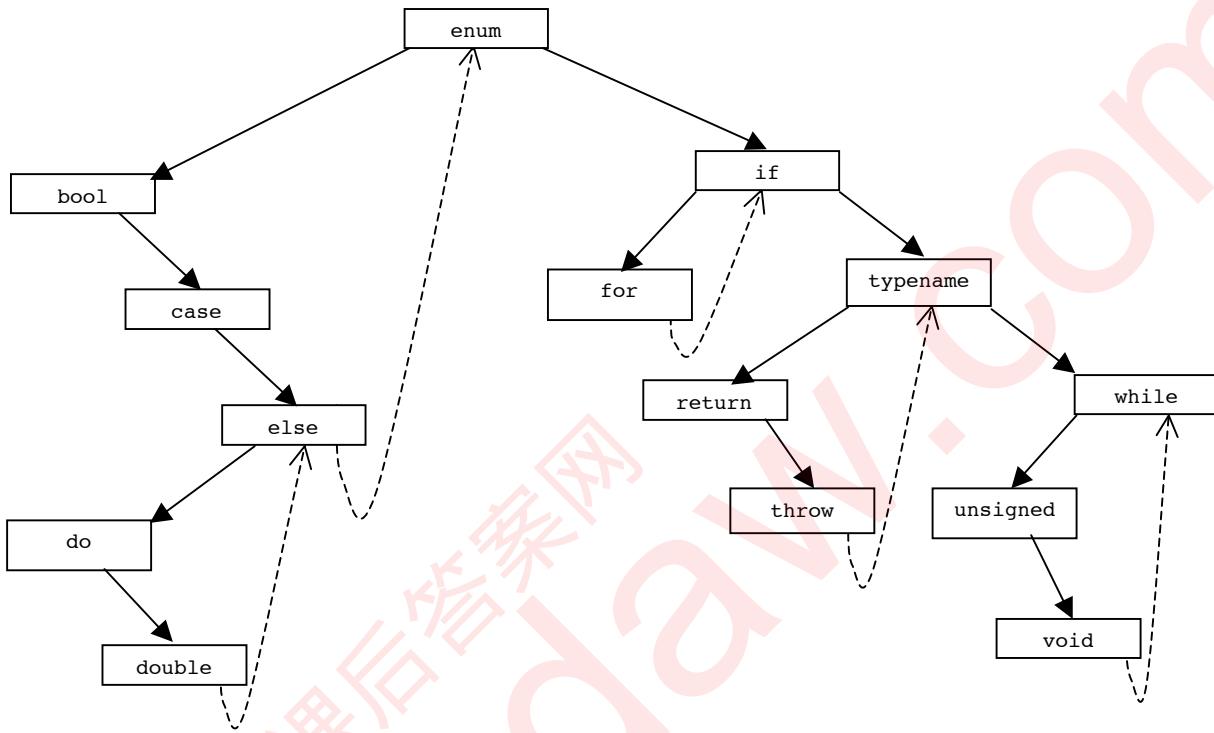
3.



4.



5.



6.

```

/*
 * Inorder traversal of a right-threaded BST.
 * Note that recursion is no longer needed. Also, the output statement
 * may be replaced by appropriate processing of a node.
 */

#include <iostream>

template <typename DataType>
void BST<DataType>::nonrecInorder(ostream & out)
{
 BST<DataType>::BinNodePointer ptr = myRoot;
 while (ptr != 0)
 {
 while (ptr->left != 0)
 ptr = ptr->left;
 out << ptr->data << endl;

 while (ptr->right != 0 && ptr->rightThread)
 {
 ptr = ptr->right;
 out << ptr->data << endl;
 }

 ptr = ptr->right;
 }
}

```

7.

```

/* Right-thread a BST. */

template <typename DataType>
void BST<DataType>::rightThread()
{
 BST<DataType>::BinNodePointer inorderPred = 0
 rightThreadAux(myRoot, inorderPred);
}

template <typename DataType>
void BST<DataType>::rightThreadAux(BST<DataType>::BinNodePointer root,
 BST<DataType>::BinNodePointer & inorderPred)
{
 if (root != 0)
 {
 rightThreadAux(root->left, inorderPred);
 if (inorderPred != 0)
 if (inorderPred->right == 0)
 {
 inorderPred->right = root;
 inorderPred->rightThread = true;
 }

 inorderPred = root,
 rightThreadAux(root->right, inorderPred);
 }
}

```

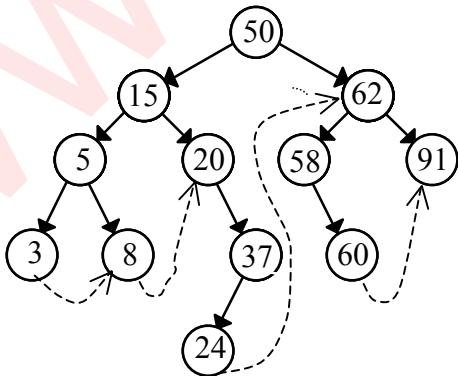
8. Algorithm to thread a BST to facilitate preorder traversal involves replacing the null right links of leaves with their preorder successors. The trick is realizing that the preorder successor of node  $x$  is the right child of the nearest ancestor — all ancestors are visited before a leaf — of  $x$  such that this ancestor's right child is not an ancestor of  $x$ . The algorithm reduces to:

Perform a preorder traversal of the BST.

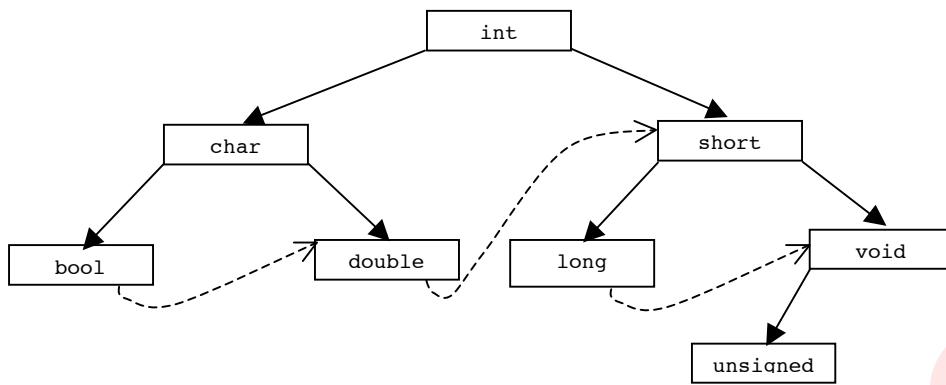
Whenever a leaf node (nil left and right links) is encountered:

Replace the leaf's right link with a thread to its preorder successor.

9.



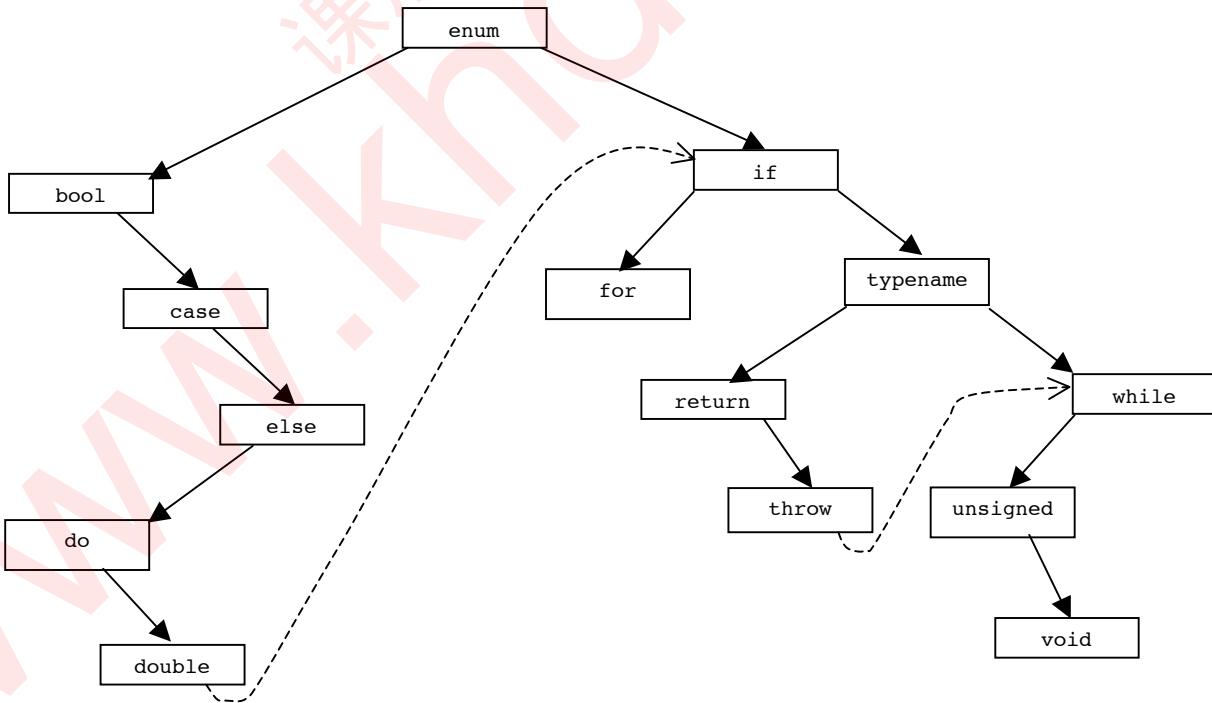
10.



11. There are no right threads for preorder traversal. The tree is simply the underlying BST shown in #3 without any thread replacements.

12. Again, there are no right threads for preorder traversal. The tree is simply the underlying BST shown in #4 without any thread replacements.

13.



14. Algorithm to carry out a preorder traversal of a BST threaded as described in Exercise 8:

1. Set  $ptr = \text{root of BST}$ .
2. Set boolean variable  $done = (\text{ptr is a null pointer})$ .
3. While ( $\text{!done}$ ) do the following:
  - a. Visit the node pointed to by  $ptr$ .
  - b. While ( $ptr \rightarrow \text{left}$  is not a null pointer) do the following:
    - i. Set  $ptr = ptr \rightarrow \text{left}$ .
    - ii. Visit the node pointed to by  $ptr$ .
  - c. If ( $ptr \rightarrow \text{right}$  is not a null pointer):
    - i. Set  $ptr = ptr \rightarrow \text{right}$ .
    - ii. Visit the node pointed to by  $ptr$ .
- Else  
Set  $done = \text{true}$ .

15. Algorithm to find preorder successor of node pointed to by  $ptr$  in an inorder-threaded BST.

If ( $ptr \rightarrow \text{left}$  is not a null pointer) :  
    Set  $preOrdSucc = ptr \rightarrow \text{left}$ .  
Else do the following:

- a. Set  $qptr = ptr$ .
- b. While ( $qptr \rightarrow \text{right}$  is a thread) do the following  
        Set  $qptr = qptr \rightarrow \text{right}$ .
- c. Set  $preOrdSucc = qptr \rightarrow \text{right}$ .

16. Algorithm to carry out a preorder traversal of an inorder-threaded BST.

1. Set  $ptr = \text{root of BST}$ .
2. While ( $ptr$  is not a null pointer) do the following:
  - a. Visit the node pointed to by  $ptr$ .
  - b. Set  $ptr = \text{preorder successor of } ptr$  as determined by algorithm in Exercise 15.
- End while.

17. Algorithm to insert a node into a right-threaded BST.

1. Get a new node pointed to by  $temp$ .
2. Set  $temp \rightarrow \text{data} = item$  to be inserted into the BST, make  $temp \rightarrow \text{left}$  a null pointer, and set  $temp \rightarrow \text{rightThread} = \text{false}$ .
3. If the BST is empty:  
    Set its root =  $temp$  and make  $temp \rightarrow \text{right}$  a null pointer.  
Else do the following:
  - a. Set  $done = \text{false}$ .
  - b. Set  $root = \text{root of the BST}$ .

c. While (*!done*) do the following:

If (*temp->data < root->data*):

    If (*root->left* is a null pointer):

        (i) Set *root->left = temp* and set *root->rightThread = false*.  
        (ii) Set *temp->right = root* and set *temp->rightThread = true*.  
        (iii) Set *done = true*.

    Else

        Set *root = root->left*.

    End if.

Else if (*temp->data > root->data*):

    If (*root->right* is a null pointer):

        (i) Set *root->right = temp* and set *root->rightThread = false*.  
        (ii) Make *temp->right* a null pointer and set *temp->rightThread = false*.  
        (iii) Set *done = true*.

    Else if (*root->rightThread*):

        (i) Set *temp->right = root->right* and *temp->rightThread = true*.  
        (ii) Set *root->right = temp* and *root->rightThread = false*.  
        (iii) Set *done = true*.

    Else

        Set *root = root->right*.

    End if

Else

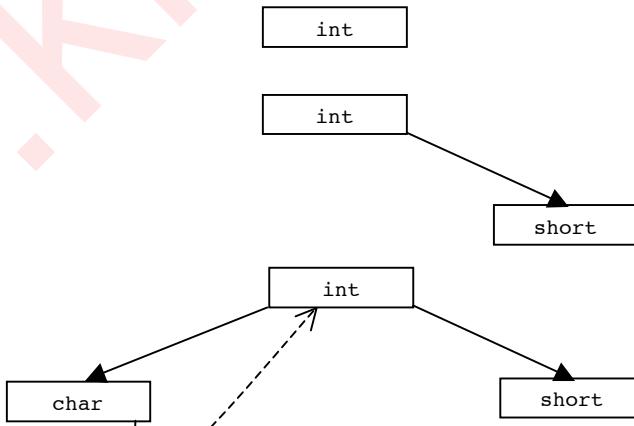
    Display a message that *item* is already in the BST.

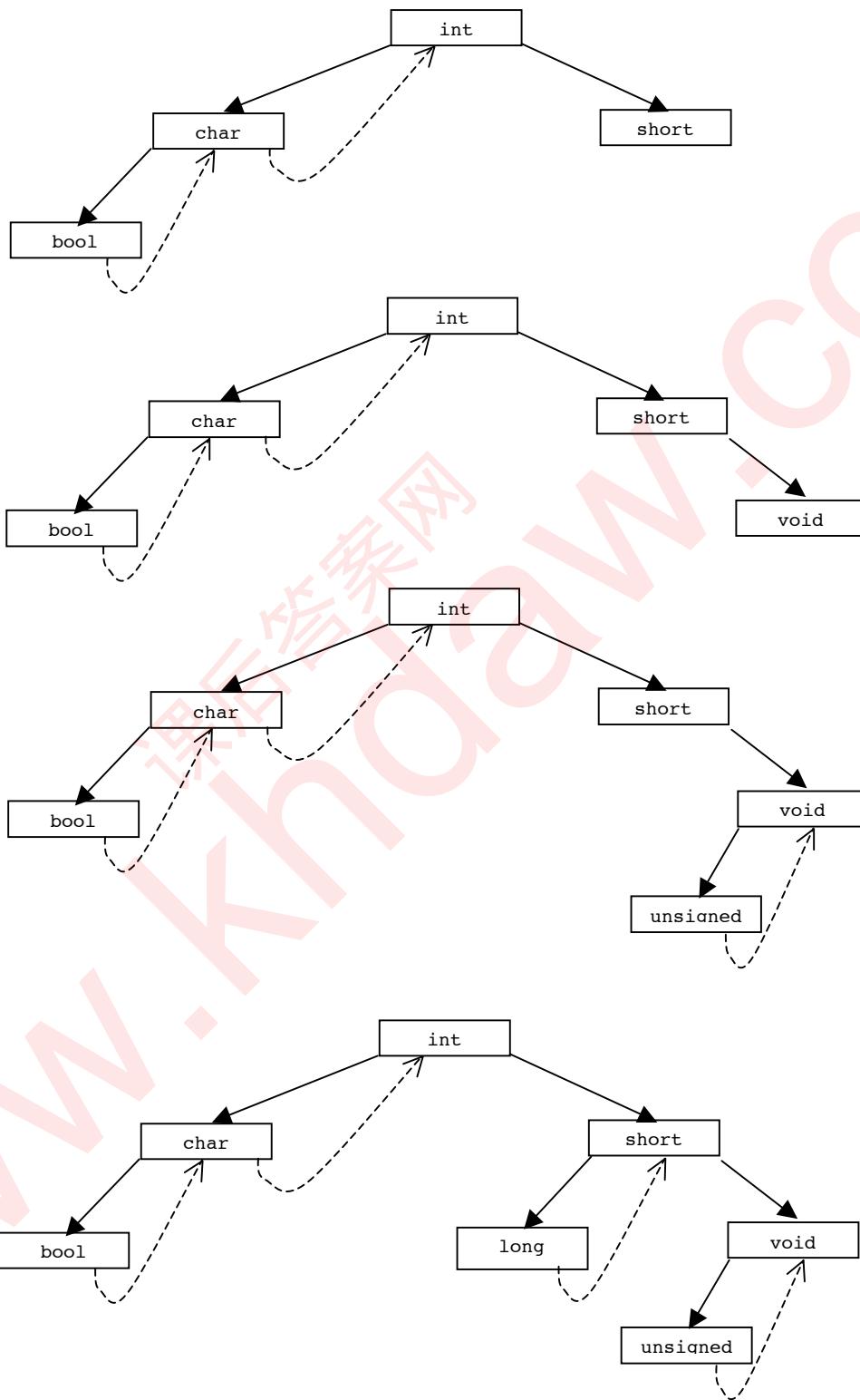
End if.

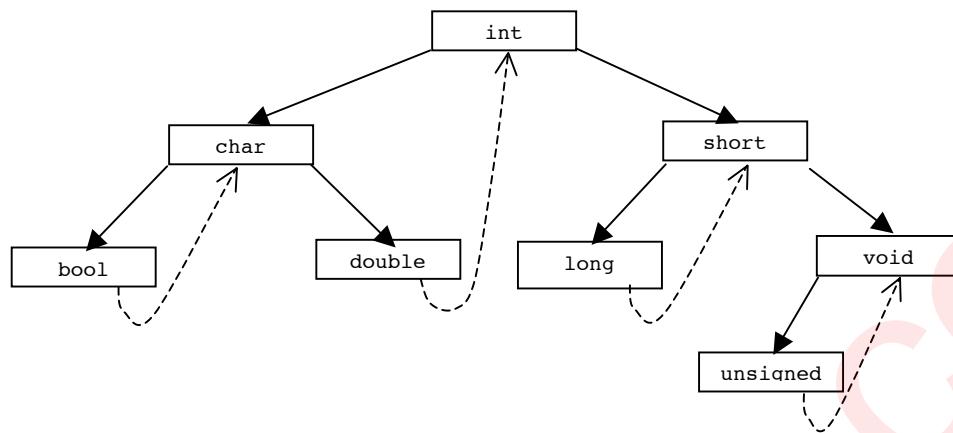
End while.

End if.

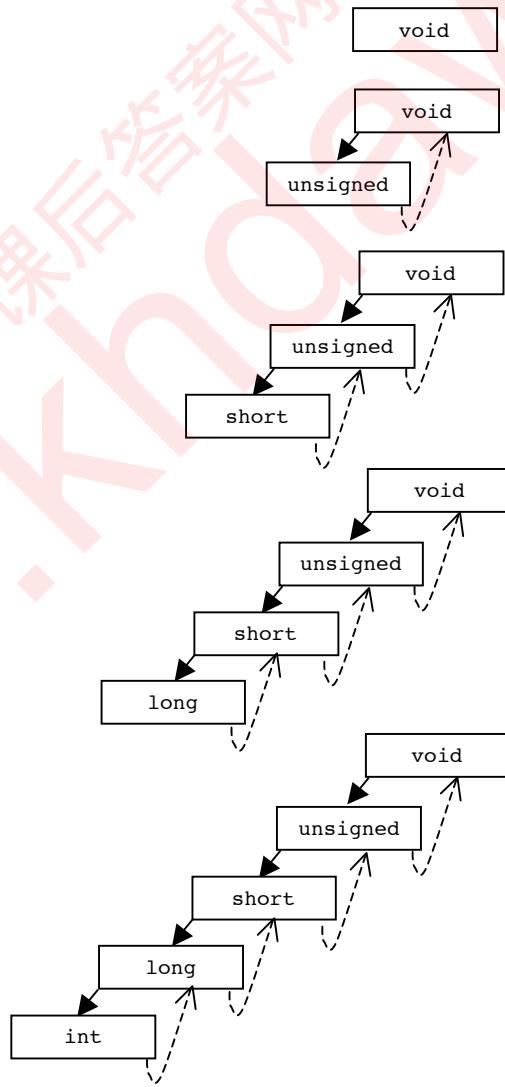
18.

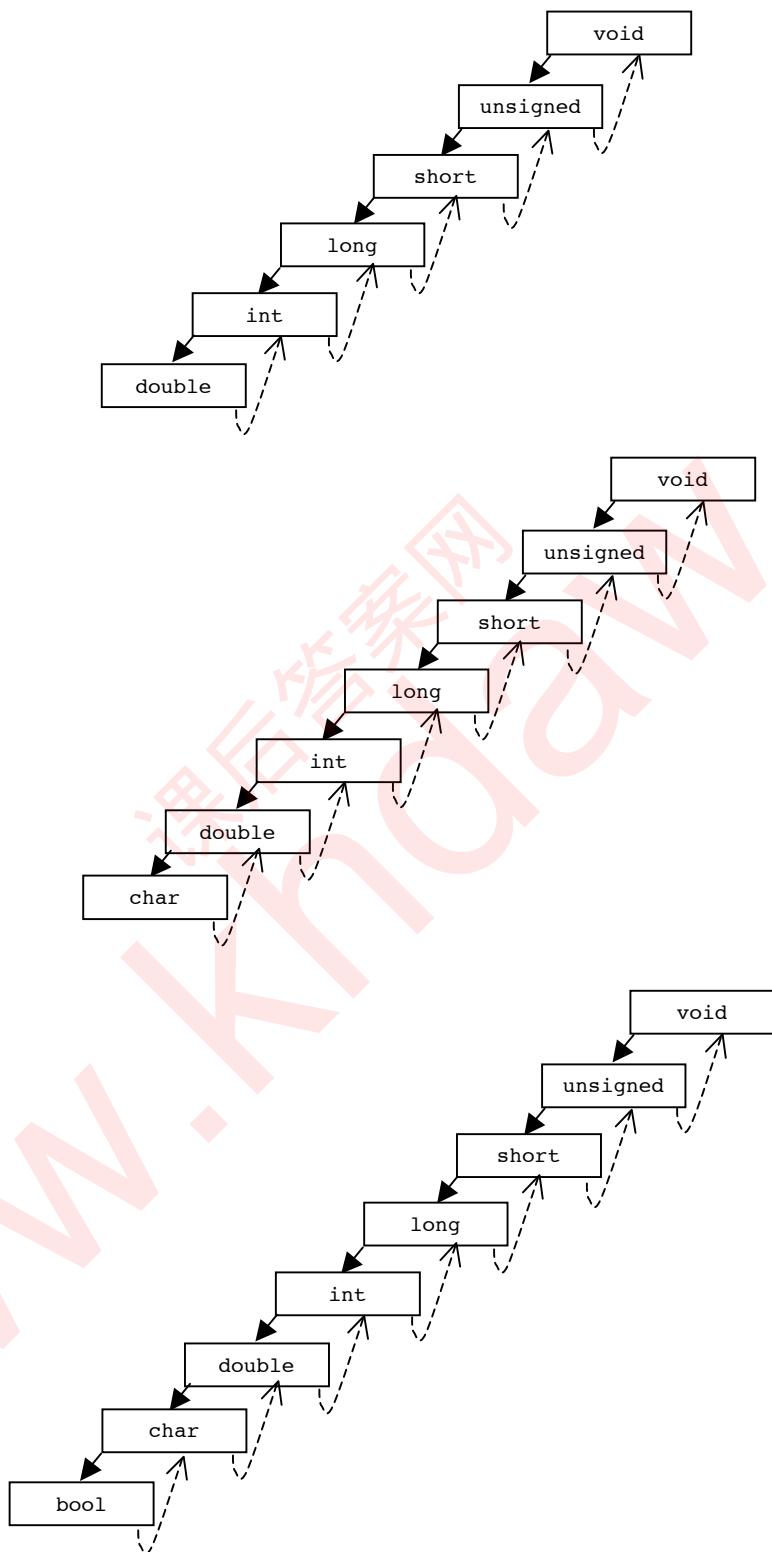






19.





20. Two auxiliary algorithms:

*Algorithm to find inorder successor of a node pointed to by p:*

```

1. Set pSucc = p->right.
2. if (!p->rightThread)
 While (pSucc->left != 0)
 Set pSucc = pSucc->left.
 End while.
End if.
// pSucc points to inorder successor

```

*Algorithm to find inorder predecessor of a node pointed to by p:*

```

1. Set pPred = root of BST.
2. While (pPred->left != 0)
 Set pPred = pPred->left.
End while.
3. Set pSucc = inorder successor of node pointed to by pPred.
4. While (pSucc != p)
 a. Set pPred = pSucc.
 b. Set pSucc = inorder successor of node pointed to by pPred.
End while.
// pPred points to inorder predecessor

```

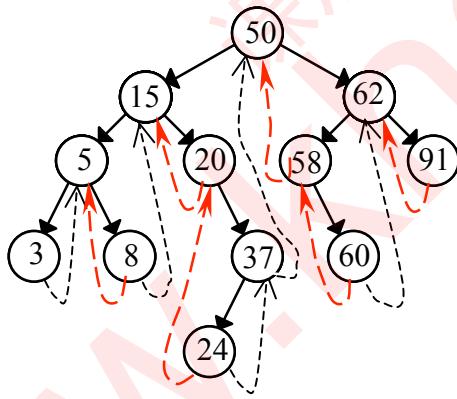
*Algorithm to delete a node from a right-threaded BST, preserving right-threadedness:*

1. Use **search2( )** operation to find node containing *item* to be removed and a pointer *p* to this node and pointer *parent* to its parent.
2. If *item* is not found
 

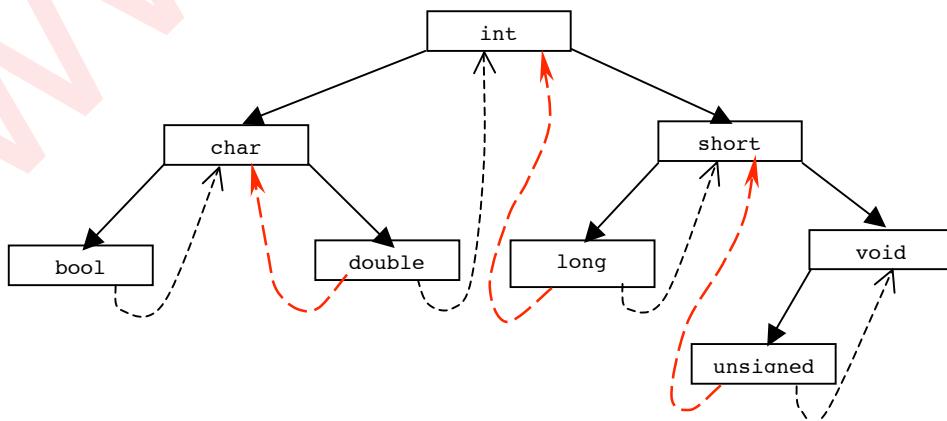
Display an error message and terminate this algorithm.
- // Otherwise proceed with the following.
3. Let *pSucc* point to inorder successor of node pointed to by *p*.
4. If (*p->left != 0 && p->right != 0 && !p->rightThread*) // Node has two children
  - a. // Find parent of inorder successor
    - If (*p->right == pSucc*)
 Set *parent* = *p*.
    - Else
      - i. Set *parent* = *p->right*.
      - ii. While (*parent->left != pSucc*)
 Set *parent* = *parent->left*.
 End while.

- b. Set  $p \rightarrow \text{data} = pSucc \rightarrow \text{data}$ . // Copy successor's data into  $p$ 's node.
  - c. Set  $p = pSucc$ . // We'll delete node pointed to by  $pSucc$  -- it has 0 or 1 child
5. Let  $pPred$  point to the inorder predecessor of node pointed to by  $p$ .
6. If ( $pPred \rightarrow \text{right} == p$ )
  - a. Set  $pPred \rightarrow \text{right} = pSucc$ .
  - b. If ( $pSucc != 0$ ) set  $pPred \rightarrow \text{rightThread} = pSucc \rightarrow \text{rightThread}$ .
 Else do the following:
  - a. Set  $\text{subtree} = p \rightarrow \text{left}$ .
  - b. If ( $\text{subtree} == 0$ )
    - Set  $\text{subtree} = p \rightarrow \text{right}$ .
  - c. If ( $\text{parent} == 0$ )
    - Set BST's root =  $\text{subtree}$ .
  - Else if ( $\text{parent} \rightarrow \text{left} == \text{subtree}$ )
    - Set  $\text{parent} \rightarrow \text{left} = \text{subtree}$ .
  - Else
    - Set  $\text{parent} \rightarrow \text{right} = \text{subtree}$  and  $\text{parent} \rightarrow \text{rightThread} = p \rightarrow \text{rightThread}$ .
 End if
7. Deallocate the node pointed to by  $p$ .

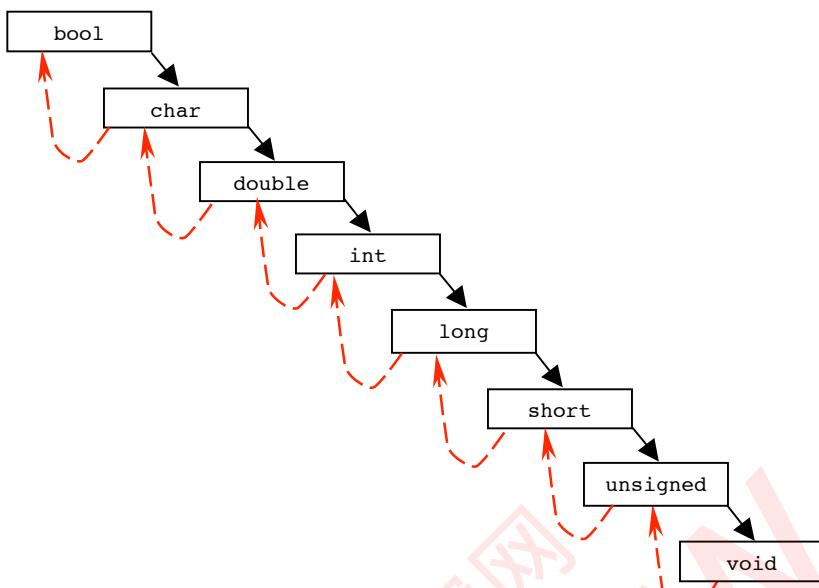
21.



22.

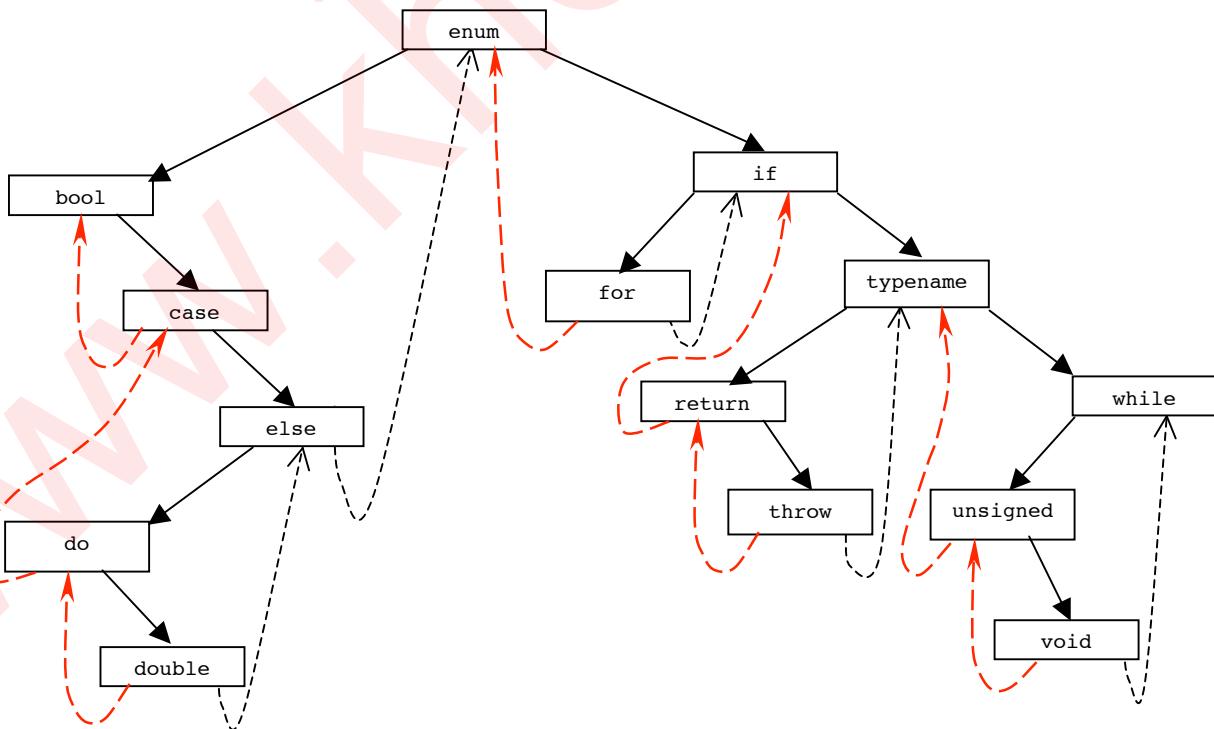


23.



24. The fully-threaded tree is the same as the right-threaded BST in Exercise 4; there are no left threads.

25.



## 26. Algorithm to fully-thread a BST:

Perform an inorder traversal of the BST.

- Whenever a node is visited, set *pred* = current node.
- Whenever a node *x* with a null right pointer is encountered, replace this right link with a thread to the inorder successor of *x*.
- Whenever a node *x* with a nil left pointer is encountered, replace this left link with a thread to *pred*.

## 27. Algorithm to insert a node into a fully-threaded BST:

1. Get a new node pointed to by *temp*.
2. Set *temp*->data = *item* to be inserted, *temp*->leftThread and *temp*->rightThread both false.
3. If the BST is empty:
  - a. Set BST's root = *temp*.
  - b. Make *temp*->left and *temp*->right both null pointers.
  - c. Terminate this algorithm.
- // Otherwise do the following:
- 4.. Set *done* = false and *root* = BST's root.
5. While (*!done*) do the following:
  - a. If (*temp*->data < *root*->data)  
If (*root*->left is null)
    - i. Set *root*->left = *temp*.
    - ii. Set *temp*->right = *root* and *temp*->rightThread = true.
    - iii. Make *temp*->left a null pointer and set *temp*->leftThread = false.
    - iv. Set *done* = true.
  - Else if (*root*->leftThread)
    - i. Set *temp*->left = *root*->left and *temp*->leftThread = true.
    - ii. Set *temp*->right = *root* and *temp*->rightThread = true.
    - iii. Set *root*->left = *temp* and *root*->leftThread = false.
    - iv. Set *done* = true.
  - Else  
Set *root* = *root*->left.
- End if.

```
Else if (temp->data > root->data)
 If (root->right is null)
 i. Set root->right = temp.
 ii. Make temp->right a null pointer and temp->rightThread = false.
 iii. Set temp->left = root and temp->leftThread = true.
 iv. Set done = true.
 Else if (root->rightThread)
 i. Set temp->right = root->right and temp->rightThread = true.
 ii. Set root->right = temp and root->rightThread = false.
 iii. Set temp->left = root and temp->leftThread = true.
 iv. Set done = true.
 Else
 Set root = root->right.
 End if.
Else display a message that item is already in the BST.
```

#### 28. Algorithm to find parent of node $x$ in a fully-threaded BST:

```
1. Set ptr = x.
2. While (ptr != 0 && !ptr->leftThread)
 Set ptr = ptr->left.
 End while.
3. If (ptr != 0 && ptr->left->right != x)
 Set parent = ptr->left.
Else:
 a. Set ptr = x.
 b. While (ptr != 0 && !ptr->rightThread)
 Set ptr = ptr->right.
 End while.
 c. If (ptr is a null pointer)
 Make parent a null pointer.
 Else
 Set parent = ptr->right.
 End if.
End if.
```

**Exercises 12.7**

1.

|    |    |
|----|----|
| 0  | 44 |
| 1  | 12 |
| 2  | 80 |
| 3  | 36 |
| 4  | 26 |
| 5  | 5  |
| 6  | 92 |
| 7  | 59 |
| 8  | 40 |
| 9  | 42 |
| 10 | 60 |

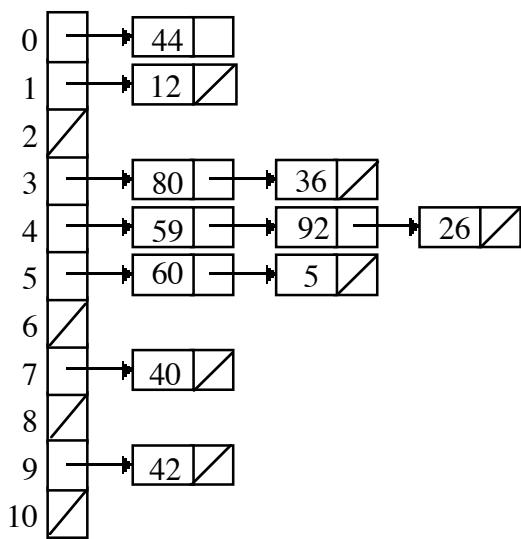
2.

|    |    |
|----|----|
| 0  | 44 |
| 1  | 12 |
| 2  | 36 |
| 3  | 92 |
| 4  | 26 |
| 5  | 5  |
| 6  | 60 |
| 7  | 40 |
| 8  | 59 |
| 9  | 42 |
| 10 | 80 |

3.

|    |    |
|----|----|
| 0  | 44 |
| 1  | 92 |
| 2  | 12 |
| 3  | 36 |
| 4  | 26 |
| 5  | 5  |
| 6  | 59 |
| 7  | 40 |
| 8  | 80 |
| 9  | 42 |
| 10 | 60 |

4.



5.

|    |       |
|----|-------|
| 0  | RATE  |
| 1  | BETA  |
| 2  | MEAN  |
| 3  | WAGE  |
| 4  | FREQ  |
| 5  | SUM   |
| 6  | KAPPA |
| 7  | ALPHA |
| 8  | NUM   |
| 9  | PAY   |
| 10 | BAR   |

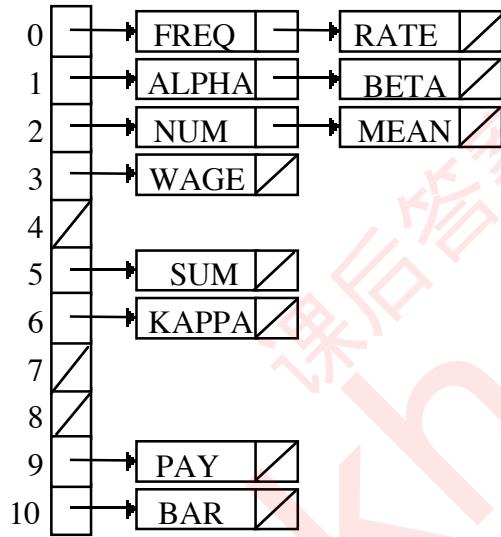
6.

|    |       |
|----|-------|
| 0  | RATE  |
| 1  | BETA  |
| 2  | ALPHA |
| 3  | MEAN  |
| 4  | WAGE  |
| 5  | SUM   |
| 6  | NUM   |
| 7  | KAPPA |
| 8  | PAY   |
| 9  | BAR   |
| 10 | FREQ  |

7.

|    |       |
|----|-------|
| 0  | RATE  |
| 1  | BETA  |
| 2  | ALPHA |
| 3  | BAR   |
| 4  | WAGE  |
| 5  | SUM   |
| 6  | MEAN  |
| 7  | FREQ  |
| 8  | KAPPA |
| 9  | PAY   |
| 10 | NUM   |

8.



9. The following solution uses a `vector< list<ElementType> >` to store the hash table. Converting it to use an array of linked lists(for the linked list class considered in Chapter 6) is straightforward.

```

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <list>
#include <algorithm>

template <typename ElementType>
class HashTable
{
public:

```

```
HashTable(int n = 0);
/*-----
Constructor

Precondition: n >= 0 is the number of items to be stored in the table.
Postcondition: Hash table with n elements (default n = 0) has been
constructed.
-----*/
HashTable(const HashTable & original);
/*-----
Copy Constructor

Precondition: A copy of HashTable object original is needed.
Postcondition: A copy of original has been constructe.
-----*/
~HashTable();
/*-----
Destructor

Precondition: This object must be destroyed.
Postcondition: All dynamically-allocated memory in object has been
reclaimed.
-----*/
void insert(ElementType item, int key);
/*-----
Insert operation

Precondition: item is to be inserted into this hash table; key is an
integer used by hash function to find its location.
Postcondition: item has been inserted into the hash table at the
location determined by applying hash function to key.
-----*/
//--Search hash table for item, with its numeric key
int search(ElementType item, int key);
/*-----
Search operation

Precondition: Hash table is to be searched for item; key is item's
numeric key.
Postcondition: Location of item in table is returned, -1 if not found.
-----*/
//--Display contents of hash table
void display(ostream & out);
/*-----
Output operation

Precondition: ostream out is open; << is defined for ElementType.
Postcondition: Contents of hash table have been output to out.
-----*/
```

```
private:
 vector< list<ElementType> > myTable;
 //--- Hash function for a numeric key
 unsigned hash(int key);
 /*-----
 *-----
 Hash function

 Precondition: None.
 Postcondition: An index of allocation in the hash table is returned.
 -----*/
};

//--- Definition of constructor
template <typename ElementType>
inline HashTable<ElementType>::HashTable(int n)
{
 list<ElementType> emptyList;
 for (int i = 1; i <= n; i++)
 myTable.push_back(emptyList);
}

//--- Definition of copy constructor
template <typename ElementType>
inline HashTable<ElementType>::HashTable(
 const HashTable<ElementType> & original)
{
 myTable = original.myTable;
}

//--- Definition of destructor
template <typename ElementType>
inline HashTable<ElementType>::~HashTable()
{ } // vector and list destructors take care of this

//--- Definition of insert()
template <typename ElementType>
inline void HashTable<ElementType>::insert(ElementType item, int key)
{
 if (search(item, key) < 0)
 myTable[hash(key)].push_front(item);
 else
 cerr << item << " already in table.\n";
}

//--- Definition of search()
template <typename ElementType>
int HashTable<ElementType>::search(ElementType item, int key)
{
 int loc = hash(key);
 if (find(myTable[loc].begin(), myTable[loc].end(), item)
 != myTable[loc].end())
 return loc;
 else
 return -1;
}
```

```
---- Definition of display()
template <typename ElementType>
void HashTable<ElementType>::display(ostream & out)
{
 for (int i = 0; i < myTable.size(); i++)
 {
 out << setw(2) << i << ":" ;
 list<ElementType>::iterator it;
 for (it = myTable[i].begin(); it != myTable[i].end(); it++)
 out << *it << " ";
 out << endl;
 }
}

---- Definition of hash()
template <typename ElementType>
inline unsigned HashTable<ElementType>::hash(int number)
{
 const unsigned
 MULTIPLIER = 25173U,
 ADDEND = 13849U,
 MODULUS = 65536U;
 unsigned x = (MULTIPLIER * number + ADDEND) % MODULUS % myTable.size();
 return x;
}
```

## Chapter 13: Sorting

### Exercises 13.1

1. After first pass, elements of x are: 10 50 70 30 40 60.  
After second pass: 10 30 70 50 40 60.
2. (a) 60 70 50 40 20 10  
70 60 50 40 20 10  
  
(b) 3, since no interchanges occur on the third pass.  
  
(c) Worst case is when elements are in reverse order.
3. (a) After x [4] is positioned: 20 30 40 60 10 50  
After x [5] is positioned: 10 20 30 40 60 50  
  
(b) If the list is in increasing order; no interchanges are required.
4. (a) After one pass: 10 50 60 30 40 70  
After two passes: 10 30 40 50 60 70  
  
(b) Function to perform double-ended selection sort:

```
template <typename ElementType>
void doubleEndedSelectionSort(ElementType x[], int n)
/*
-----*
 Sort a list into ascending order using double-ended selection sort.

 Precondition: array x contains n elements.
 Postcondition: The n elements of array have been sorted into
 ascending order.
-----*/
{
 int minValue, maxValue, minPos, maxPos;

 for (int i = 1; i <= n/2; i++)
 {
 minValue = maxValue = x[i];
 minPos = maxPos = i;

 // find min and max values among x[i], . . . , x[n]
```

```

for (int j = i+1; j <= n-i+1; j++)
{
 if (x[j] < minValue)
 {
 minValue = x[j];
 minPos = j;
 }

 if (x[j] > maxValue)
 {
 maxValue = x[j];
 maxPos = j;
 }
}

// make sure that positioning min value doesn't overwrite max
if (i == maxPos)
 maxPos = minPos;

x[minPos] = x[i];
x[i] = minValue;
x[maxPos] = x[n-i+1];
x[n-i+1] = maxValue;
}
}

```

- (c) Computing time is  $O(n^2)$ .
5. (a) After step 1 executes,  $x[i]$ 's are: 30 50 90 10 60 70 20 100 80 40

There are 5 passes for step 2.  $x[i]$ 's are:

|                    |    |    |    |    |    |    |    |    |    |     |
|--------------------|----|----|----|----|----|----|----|----|----|-----|
| After first pass:  | 10 | 50 | 40 | 20 | 60 | 70 | 30 | 90 | 80 | 100 |
| After second pass: | 10 | 20 | 40 | 30 | 60 | 70 | 50 | 80 | 90 | 100 |
| After third pass:  | 10 | 20 | 30 | 40 | 60 | 70 | 50 | 80 | 90 | 100 |
| After fourth pass: | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| After fifth pass:  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

- (b) The functions below implement Min-Max Sort.

```

template <typename ElementType>
void swap(ElementType x[], int a, int b)
{
 int temp = x[a];
 x[a] = x[b];
 x[b] = temp;
}

```

```
template <typename ElementType>
void minMaxSort(ElementType x[], int n)
{
 // Create rainbow pattern
 for (int i = 1; i <= n/2; i++)
 if (x[i] > x[n + 1 - i])
 swap(x, i, n + 1 - i);

 // Find smallest in first half of list
 for (int i = 1; i <= n/2; i++)
 {
 int minPos = i;
 int minValue = x[i];

 for (int j = i + 1; j <= n/2; j++)
 if (x[j] < minValue)
 {
 minPos = j;
 minValue = x[j];
 }
 // Swap smallest with first element
 x[minPos] = x[i];
 x[i] = minValue;

 // Find largest in last half of list
 int maxPos = n + 1 - i;
 int maxValue = x[maxPos];

 for (int j = n/2 + 1; j <= n + 1 - i; j++)
 if (x[j] > maxValue)
 {
 maxPos = j;
 maxValue = x[j];
 }
 // Swap largest with last element
 x[maxPos] = x[n + 1 - i];
 x[n + 1 - i] = maxValue;

 // Recreate rainbow pattern
 if (x[minPos] > x[n + 1 - minPos])
 swap(x, minPos, n + 1 - minPos);
 if (x[maxPos] < x[n + 1 - maxPos])
 swap(x, maxPos, n + 1 - maxPos);
 }
}
```

6.

```
// Recursive helper function for recSelectionSort() so
// it has same signature as other sorting functions.

template <typename ElementType>
void recSelectionSortAux(ElementType x[], int n, int first)
{
 if (first < n)
 {
 int minPos = first;
 int minValue = x[first];

 for (int j = first + 1; j <= n; j++)
 if (x[j] < minValue)
 {
 minPos = j;
 minValue = x[j];
 }

 x[minPos] = x[first];
 x[first] = minValue;

 recSelectionSortAux(x, n, first + 1);
 }
}

/* recursive SelectionSort */
template <typename ElementType>
void recSelectionSort(ElementType x[], int size)
{ recSelectionSortAux(x, size, 0); }
```

7.

```
// Recursive helper function for recBubbleSort() so
// it has same signature as other sorting functions.
template <typename ElementType>
void recBubbleSortAux(ElementType x[], int numCompares)
{
 if (numCompares > 0)
 {
 int last = 1;
 for (int i = 1; i <= numCompares; i++)
 if (x[i] > x[i + 1])
 {
 int temp = x[i];
 x[i] = x[i + 1];
 x[i + 1] = temp;
 last = i;
 }
 recBubbleSortAux(x, last - 1);
 }
}

/* recursive BubbleSort */
template <typename ElementType>
void recBubbleSort(ElementType x[], int n)
{ recBubbleSortAux(x, n); }
```

8. Bubblesort algorithm for a linked list with head node:

1. If  $\text{first} \rightarrow \text{next} == 0$  // empty list  
terminate this algorithm.  
Else continue with the following:
  2. Initialize  $\text{lastPtr} = 0$ ,  $\text{lastSwap} = \text{first}$ .
  3. While ( $\text{lastSwap} \rightarrow \text{next} != 0$ )  
 $\text{lastSwap} = \text{lastSwap} \rightarrow \text{next}$ . // Initially, put  $\text{lastSwap}$  at next-to-last node
  4. While ( $\text{lastSwap} != \text{first} \rightarrow \text{next}$ )
    - a.  $\text{ptr} = \text{first} \rightarrow \text{next}$
    - b. while ( $\text{ptr} != \text{lastSwap}$ )
      - i. if ( $\text{ptr} \rightarrow \text{data} > \text{ptr} \rightarrow \text{next} \rightarrow \text{data}$ )
        - (1) swap( $\text{ptr} \rightarrow \text{data}$ ,  $\text{ptr} \rightarrow \text{next} \rightarrow \text{data}$ )
        - (2)  $\text{lastPtr} = \text{ptr}$
      - ii.  $\text{ptr} = \text{ptr} \rightarrow \text{next}$
    - c.  $\text{lastSwap} = \text{lastPtr}$
9. (a) Elements of x after each pass:  
 left to right: 30 80 20 60 70 10 90 50 40 100  
 right to left: 10 30 80 20 60 70 40 90 50 100  
 left to right: 10 30 20 60 70 40 80 50 90 100  
 right to left: 10 20 30 40 60 70 50 80 90 100  
 left to right: 10 20 30 40 60 50 70 80 90 100  
 right to left: 10 20 30 40 50 60 70 80 90 100  
 left to right: 10 20 30 40 50 60 70 80 90 100  
 right to left: 10 20 30 40 50 60 70 80 90 100

(b) Algorithm for two way bubble sort:  $O(n^2)$ .

Do the following:

1. Set  $\text{interchanges1}$  and  $\text{interchanges2}$  to false.
2. For  $i = 1$  to  $n - 1$  do the following:  
 If  $x[i] > x[i+1]$  then
  - a. Interchange  $x[i]$  and  $x[i+1]$
  - b. Set  $\text{interchanges1}$  to true.
 End for.
3. If  $\text{noInterchanges1}$  is false then  
 For  $i = n - 1$  downto 1 do the following:  
 If  $x[i+1] < x[i]$  then
  - a. Interchange  $x[i]$  and  $x[i+1]$
  - b. Set  $\text{interchanges2}$  to true.
 End for.

While  $\text{interchanges1}$  or  $\text{interchanges2}$  are true.

10. Algorithm to insertion sort a linked list with head node pointed to by first.

If the list has fewer than 2 elements,

    Terminate this algorithm,

Else do the following:

1.  $ptr = \text{first}$  // points to the predecessor of the place to insert the new node  
 $predPtr = ptr \rightarrow \text{next}$  // points to the predecessor of the next node to be inserted  
 $nextElPtr = predPtr \rightarrow \text{next}$  // points to the node of the next element to be inserted
2. While ( $nextElPtr \neq 0$ )
  - a. While ( $ptr \rightarrow \text{next} \neq nextElPtr \&& ptr \rightarrow \text{next} \rightarrow \text{data} < nextElPtr \rightarrow \text{data}$ )
    - $ptr = ptr \rightarrow \text{next}$
 End while
  - b. If ( $ptr \rightarrow \text{next} \neq nextElPtr$ )
    - i.  $predPtr \rightarrow \text{next} = nextElPtr \rightarrow \text{next}$
    - ii.  $nextElPtr \rightarrow \text{next} = ptr \rightarrow \text{next}$
    - iii.  $ptr \rightarrow \text{next} = nextElPtr$ .
    - iv.  $nextElPtr = predPtr \rightarrow \text{next}$
  - c.  $ptr = first$
 End while

11. (a) Algorithm for binary insertion sort.

For  $index = 1$  to end of array, do

1. If  $x[index - 1] > x[index]$  // Adjustment needed
  - a.  $first = 0, last = index, found = \text{false}$
  - b.  $item = x[index]$
  - c. while ( $\text{!found} \&& last - first > 1$ ) //binary search
    - i.  $loc = (first + last) / 2;$
    - ii. if  $x[index] == x[loc]$   
 $found = \text{true}$
    - else if  $x[index] < x[loc]$   
 $last = loc;$
    - else  
 $first = loc;$
  - d. if  $found$ 
    - i. shift array elements  $x[loc - 1] .. x[index - 1]$  one position to the right
    - ii. set  $x[loc] = item$
  - else if  $item > x[first]$ 
    - i. shift array elements  $x[last] .. x[index - 1]$  one position to the right
    - ii. set  $x[last] = item$
  - else
    - i. shift array elements  $x[first] .. x[index - 1]$  one position to the right
    - ii. set  $x[first] = item$

|     |            |    |     |     |     |     |     |     |     |     |     |
|-----|------------|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| (b) | Insert 90: | 90 | 100 | 60  | 70  | 40  | 20  | 50  | 30  | 80  | 10  |
|     | Insert 60: | 60 | 90  | 100 | 70  | 40  | 20  | 50  | 30  | 80  | 10  |
|     | Insert 70: | 60 | 70  | 90  | 100 | 40  | 20  | 50  | 30  | 80  | 10  |
|     | Insert 40: | 40 | 60  | 70  | 90  | 100 | 20  | 50  | 30  | 80  | 10  |
|     | Insert 20: | 20 | 40  | 60  | 70  | 90  | 100 | 50  | 30  | 80  | 10  |
|     | Insert 50: | 20 | 40  | 50  | 60  | 70  | 90  | 100 | 30  | 80  | 10  |
|     | Insert 30: | 20 | 30  | 40  | 50  | 60  | 70  | 90  | 100 | 80  | 10  |
|     | Insert 80: | 20 | 30  | 40  | 50  | 60  | 70  | 80  | 90  | 100 | 10  |
|     | Insert 10: | 10 | 20  | 30  | 40  | 50  | 60  | 70  | 80  | 90  | 100 |

There were 27 comparisons of list elements.

(b) There would be 35 comparisons of list elements.

12. (a)

40 10 50 30 80 20 60 70 100 90  
10 20 30 40 50 60 70 80 90 100

(b)

```
/*---Incremented Insertion Sort---*/
template <typename ElementType>
void incrInsertSort(ElementType x[], int numElements,
 int start, int gap)
{
 ElementType nextElement;
 int j;

 for (int i = start + gap; i <= numElements; i += gap)
 {
 // Insert x[i] into its proper position among
 // x[start], x[start + gap], . . .

 nextElement = x[i];
 j = i;

 while (j - gap >= start && nextElement < x[j - gap])
 {
 // Shift element gap positions to right to open a spot
 x[j] = x[j - gap];
 j -= gap;
 }
 // Now drop next Element into the open spot
 x[j] = nextElement;
 }
}
```

```

/*--- ShellSort x[1], x[2], ..., x[numElements] ---*/
template <typename ElementType>
void shellSort(ElementType x[], int numElements)
{
 int gap = 1;
 // Find largest value in incr. seq. <= numElements

 while (gap < numElements)
 gap = 3 * gap + 1;

 gap /= 3;
 while (gap >= 1)
 {
 for (int i = 1; i <= gap; i++)
 incrInsertSort(x, numElements, i, gap);
 gap /= 3;
 }
}

```

- 13-14. The following are array-based treesort and supporting routines. Array and linked list implementations are similar. The primary difference is one of traversal: index (for loop) is used for an array. A temporary pointer, continually updated to the contents of the next field (while loop), is used for a linked list.

```

template <typename ElementType>
void treeSort(ElementType x[], int size)
{
 BST<ElementType> tree;
 for (int index = 0; index < size; index++)
 tree.insert(x[index]);

 tree.inOrder(x);
}

```

where `inOrder()` is a function member of `BST` defined as follows:

```

template <typename ElementType>
inline void BST<ElementType>::inOrder(ElementType x[])
{ int index = 0; inOrderAux(root, x, index); }

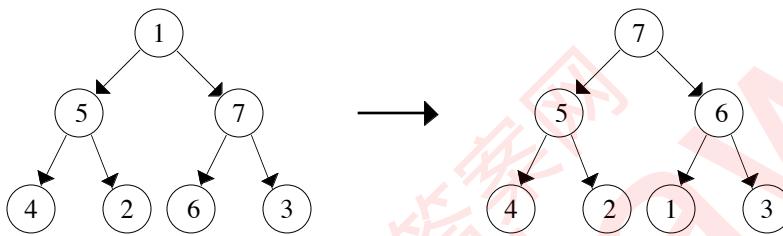
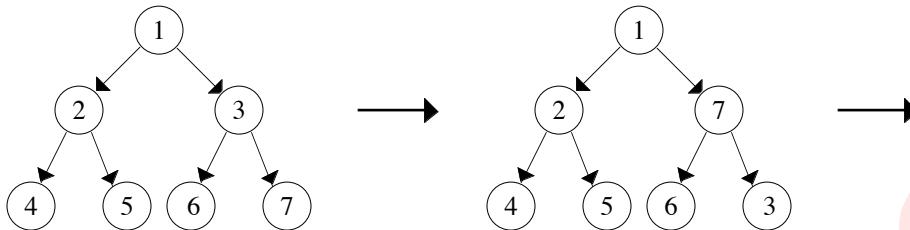
template <typename ElementType>
void BST<ElementType>::inOrderAux(BST<ElementType>::BinNodePointer root,
 ElementType x[], int & index)
{
 if (root != 0)
 {
 inOrderAux(root->left, x, index);
 x[index] = root->data;
 index++;
 inOrderAux(root->right, x, index);
 }
}

```

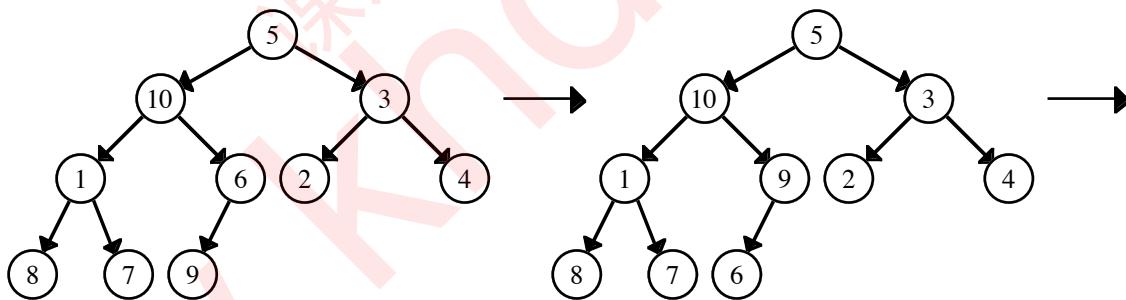
**Exercises 13.2**

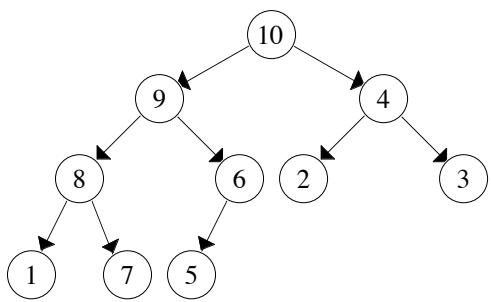
1. The tree is not complete because the next-to-bottom level is not completely full.

2.

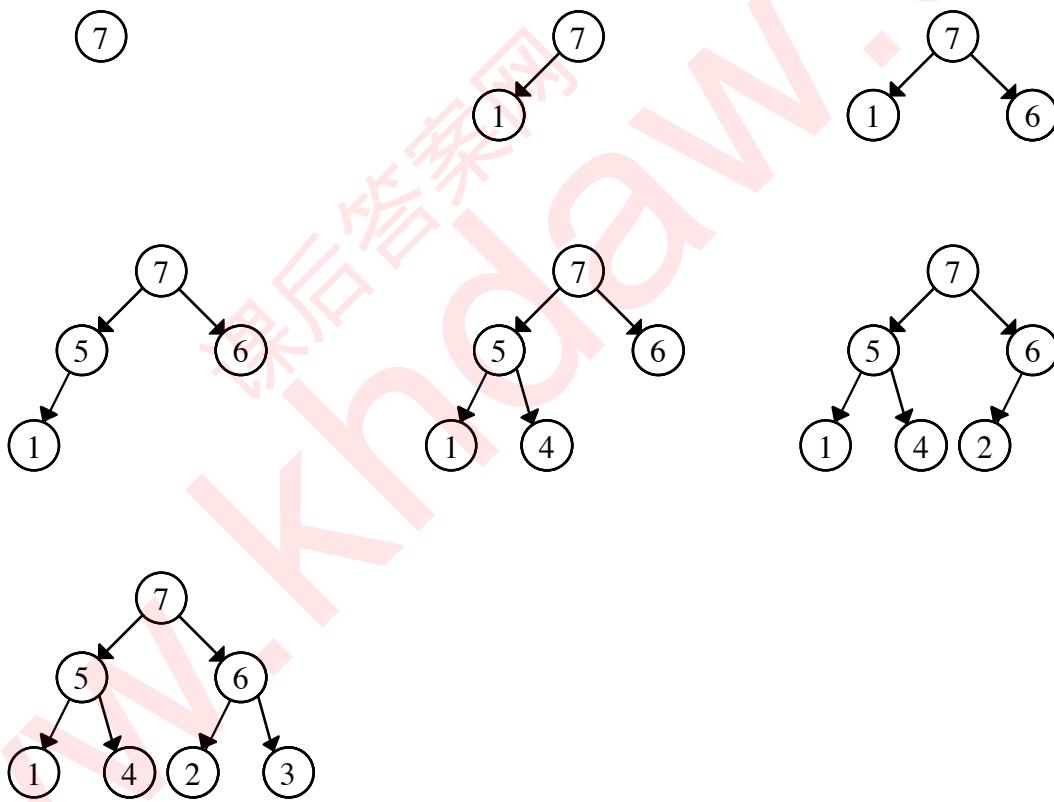


3.

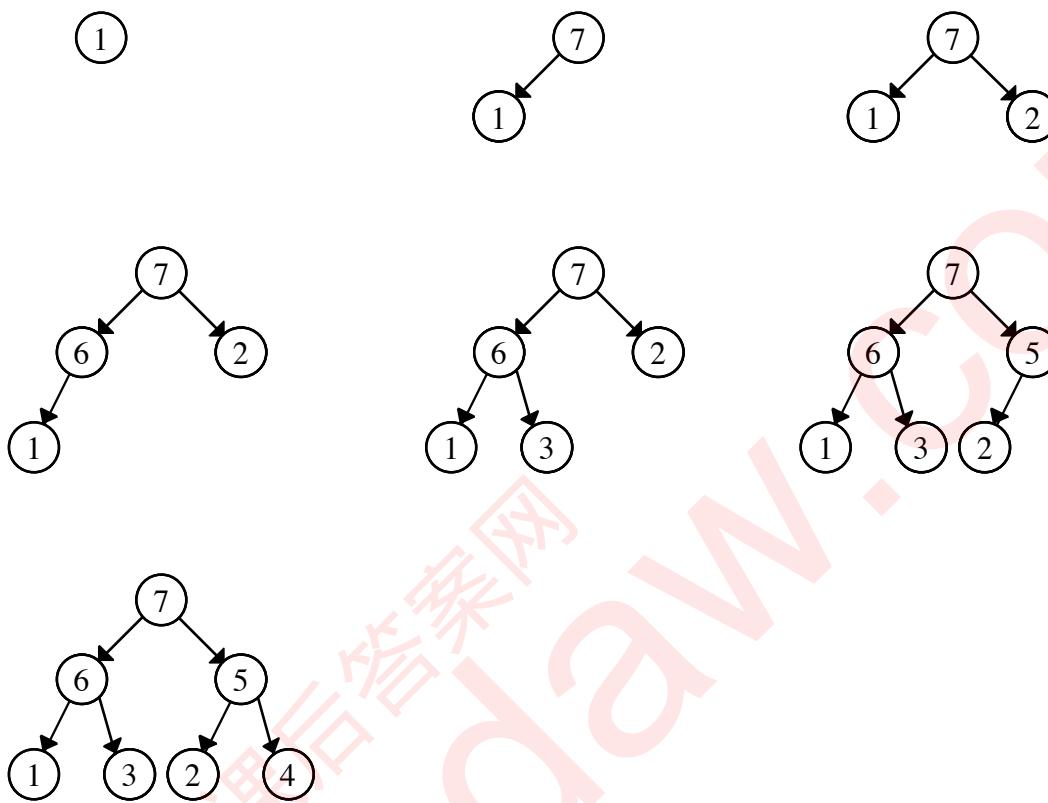




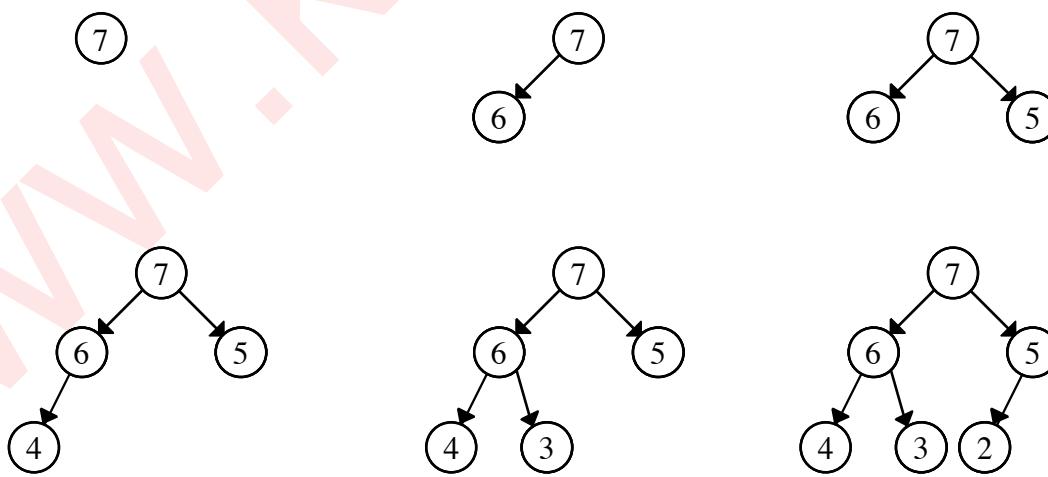
4.

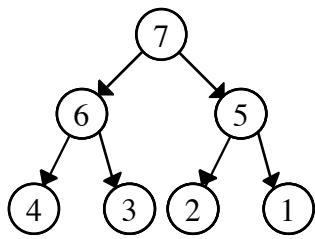


5.

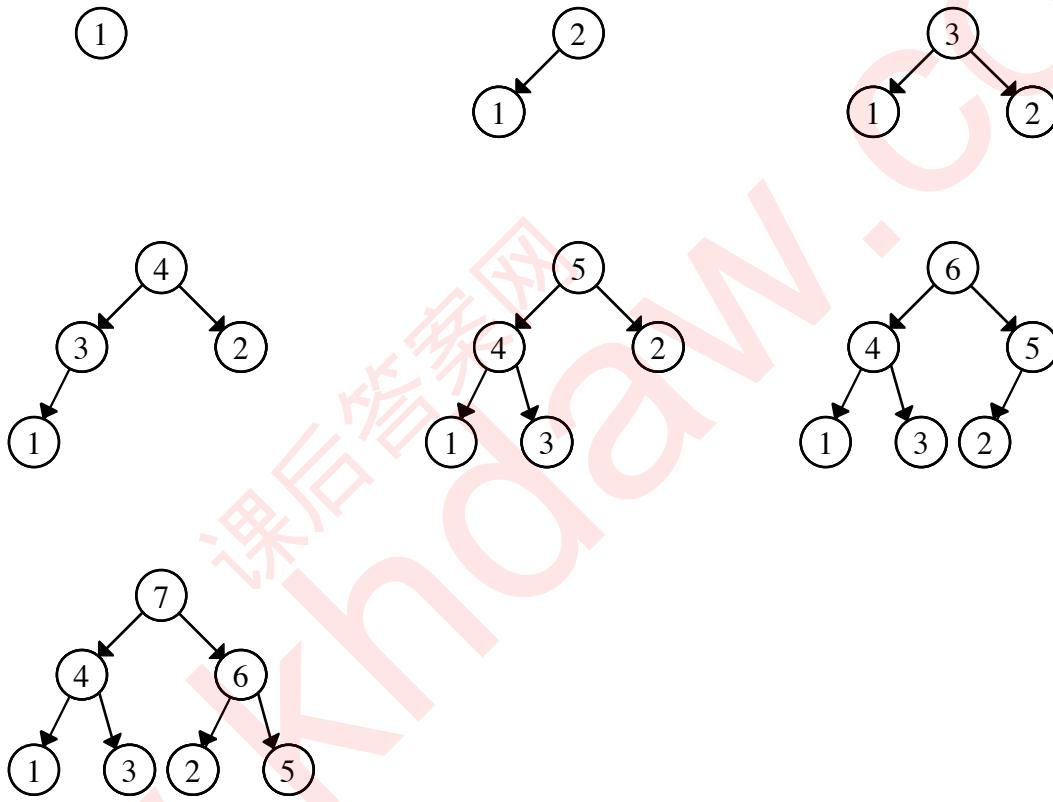


6.

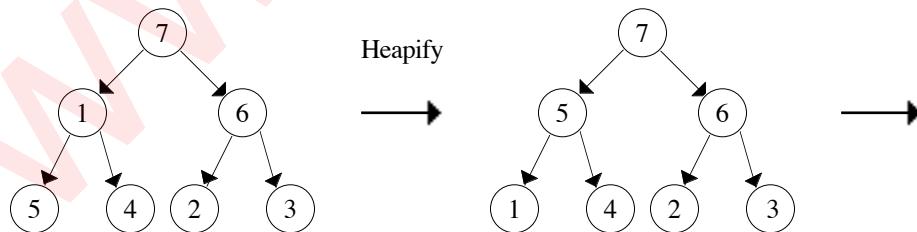


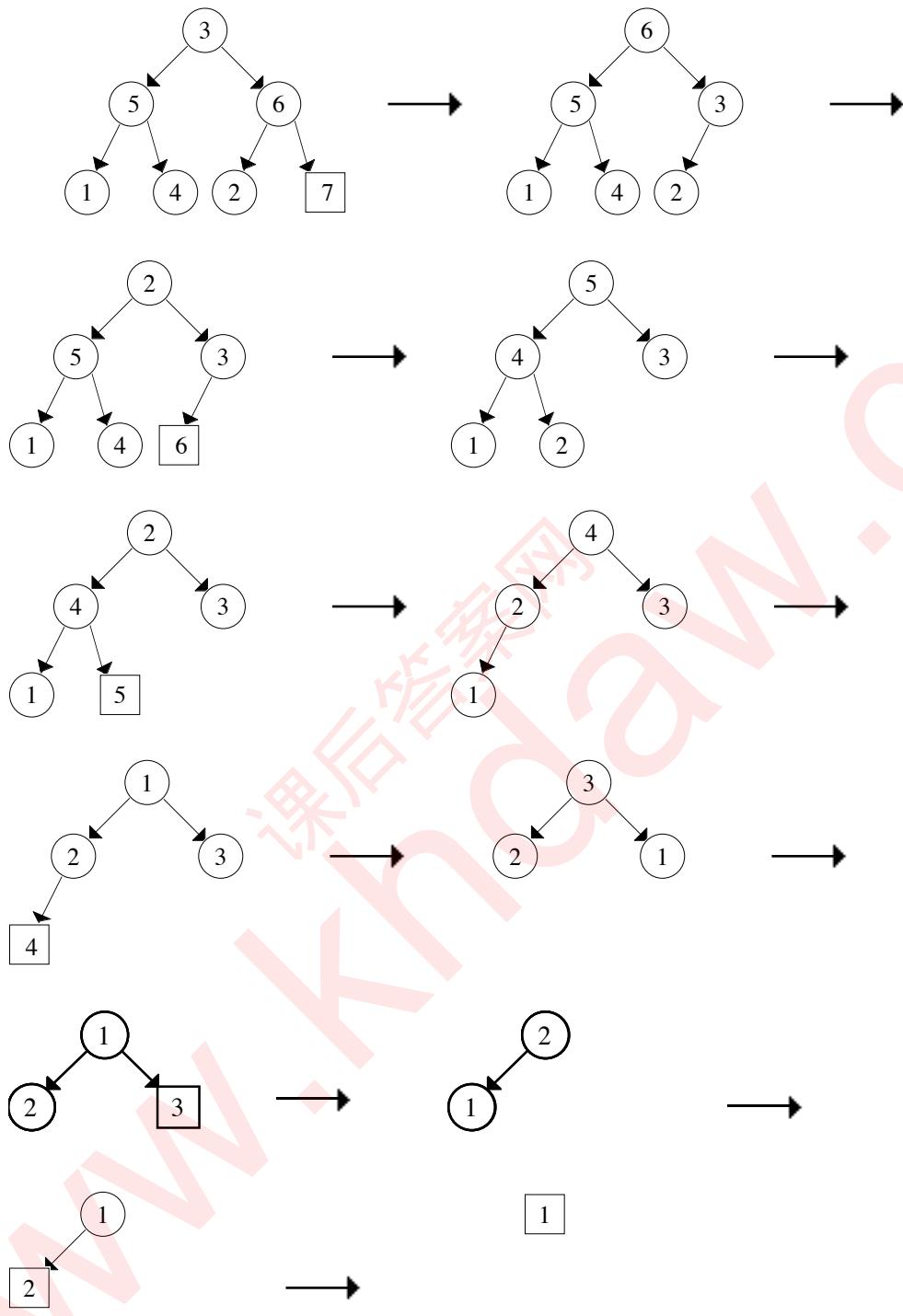


7.

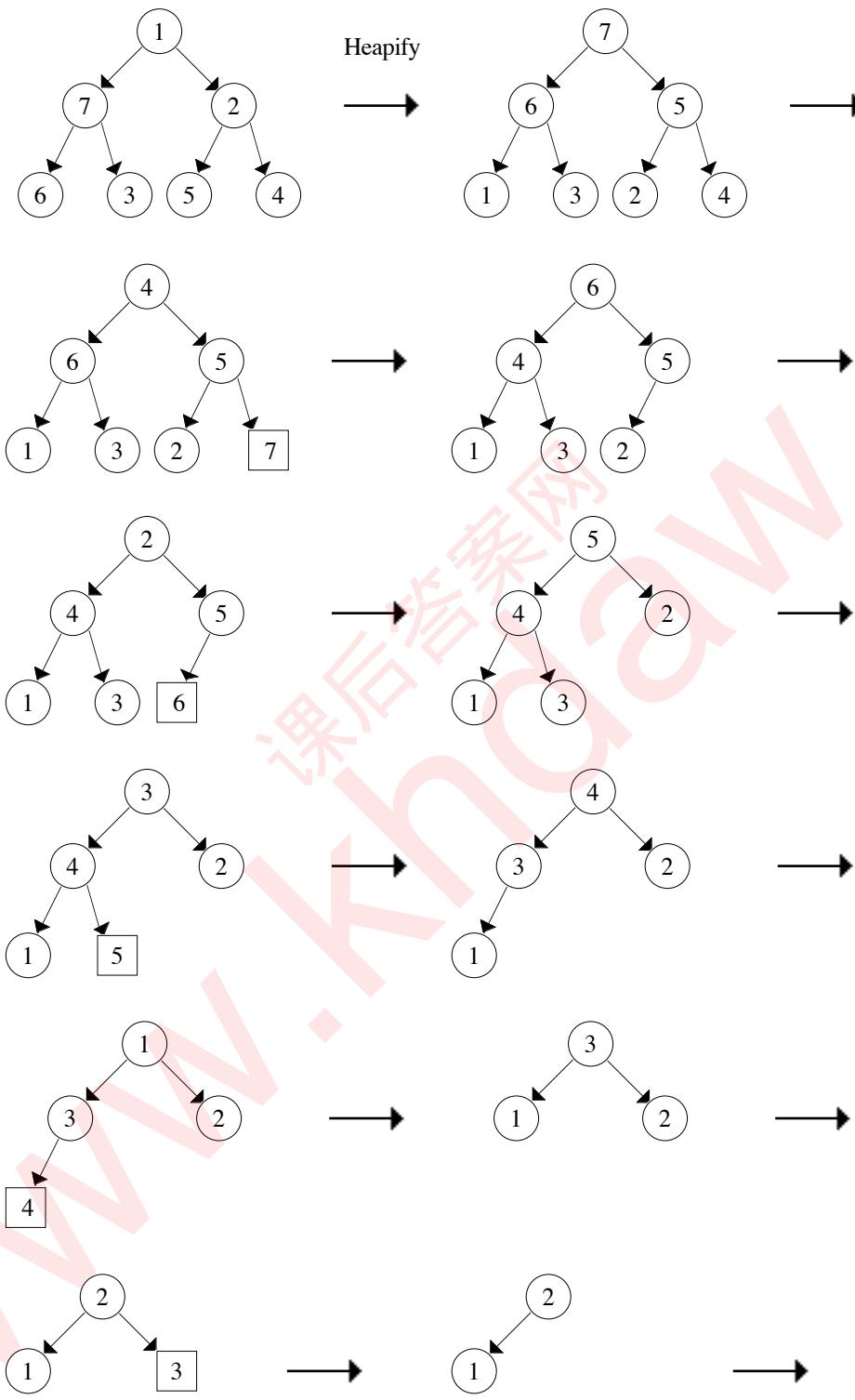


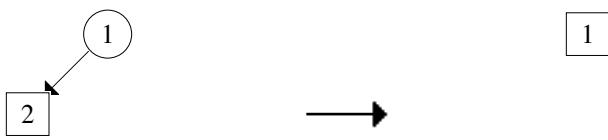
8.



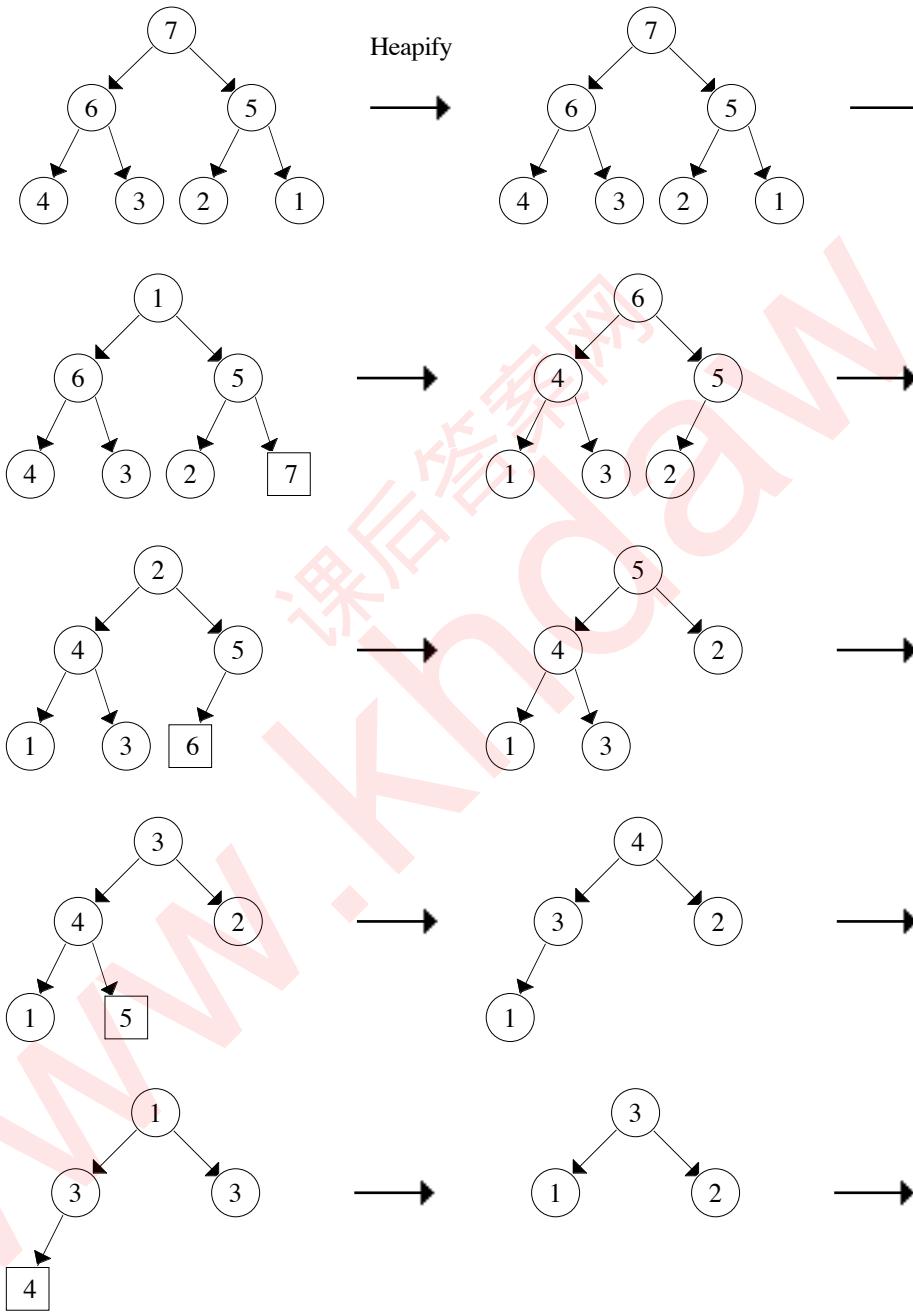


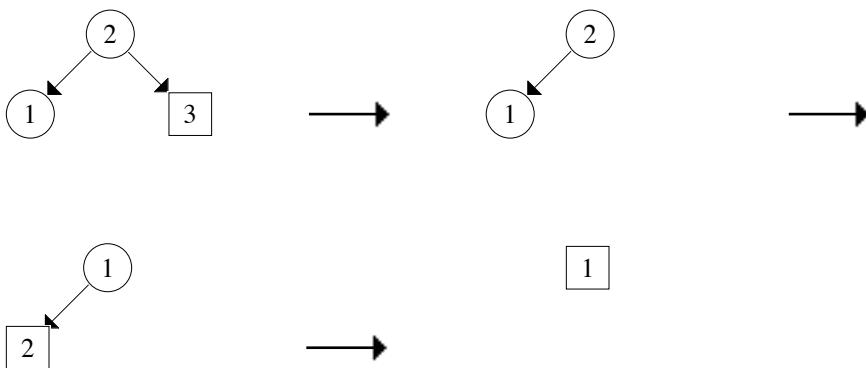
9.



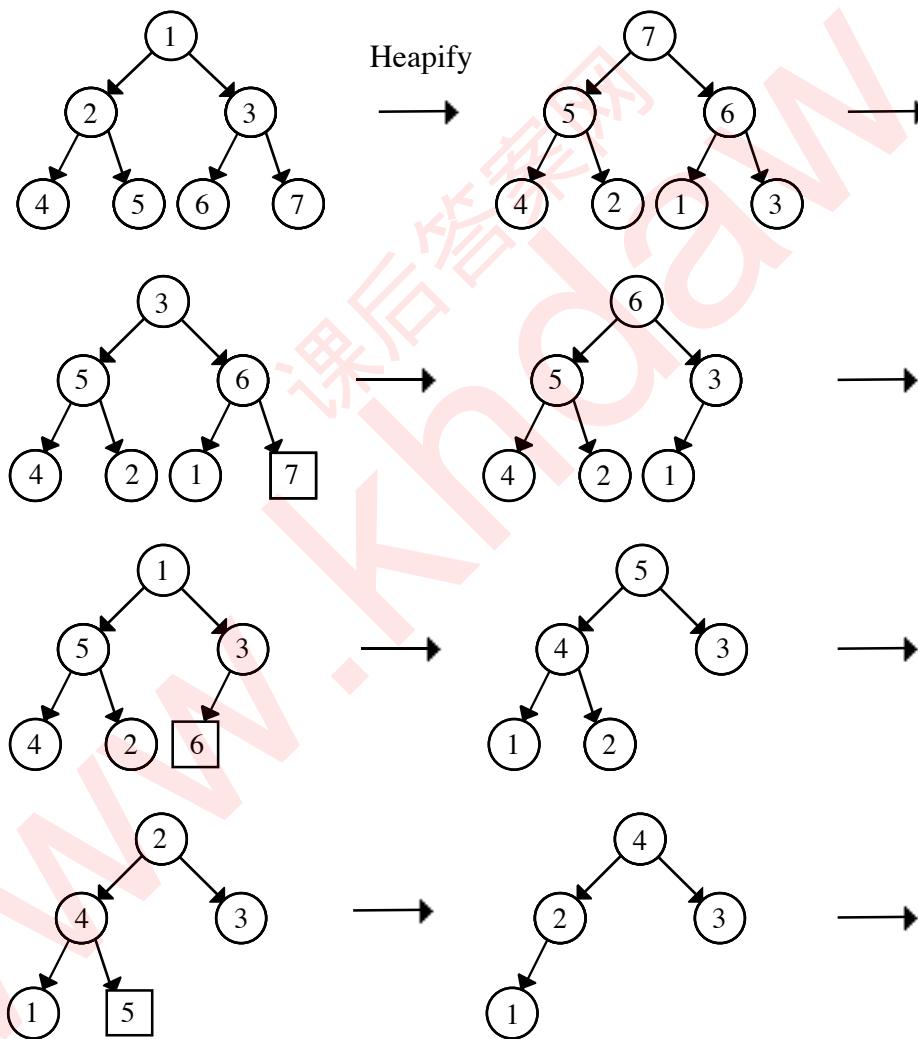


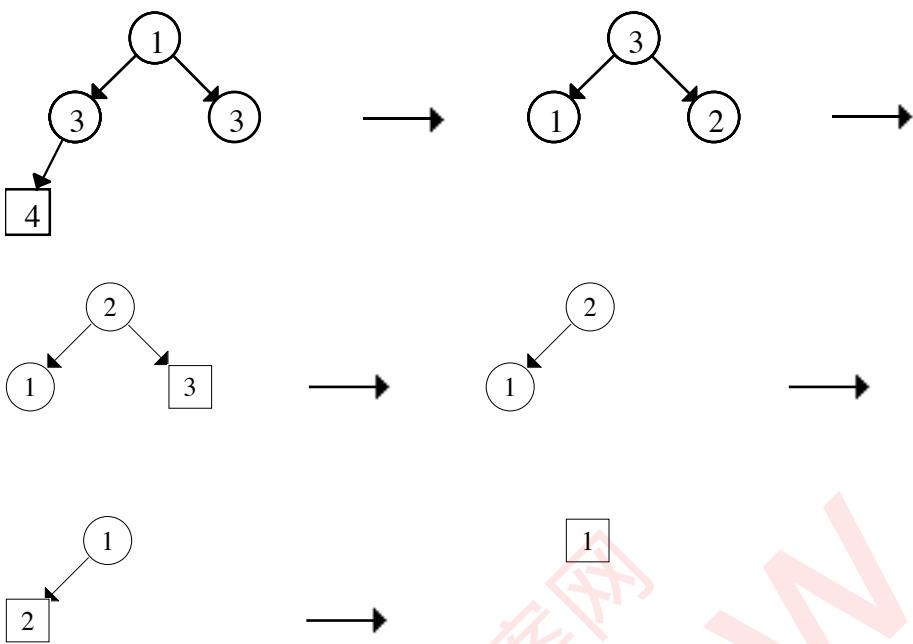
10.





11.





12. Contents of array x after each of the calls:

After first call: 20 15 31 49 67 50 3 10 26  
 After second call: 20 15 50 49 67 31 3 10 26

(The remaining calls produce:

After third call: 20 67 50 49 15 31 3 10 26  
 After fourth call: 67 49 50 26 15 31 3 10 26)

13. Contents of array x after each of the calls:

After first call: 88 77 55 66 22 33 44 99  
 After second call: 77 66 55 44 22 33 88 99

(The remaining calls produce:

After third call: 66 44 55 33 22 77 88 99  
 After fourth call: 55 44 22 33 66 77 88 99  
 After fifth call: 44 33 22 55 66 77 88 99  
 After sixth call: 33 22 44 55 66 77 88 99  
 After seventh call: 22 33 44 55 66 77 88 99

14-15.

```
#include <iostream>
using namespace std;

const int HEAP_CAPACITY = 127;
template <typename ElementType>
class Heap
{
public:
 //---- PUBLIC FUNCTION MEMBERS ----
 Heap();
 /*-----
 Constructor

 Precondition: None.
 Postcondition: An empty heap that can store HEAP_CAPACITY elements has
 been constructed.
 -----*/
 bool empty() const;
 /*-----
 Check if heap is empty.

 Precondition: None.
 Postcondition: True is returned if heap is empty, false if not.
 -----*/
 int getSize() const;
 /*-----
 Return number of elements in heap.

 Precondition: None.
 Postcondition: mySize is returned.
 -----*/
 ElementType * getArray();
 /*-----
 Return array used to store elements of heap.

 Precondition: None.
 Postcondition: myArray is returned.
 -----*/
 void insert(ElementType item);
 /*-----
 Insert operation

 Precondition: mySize < HEAP_CAPACITY.
 Postcondition: item has been inserted into the heap so the result is
 still a heap, provided there is room in myArray; otherwise, a
 heap-full message is displayed and execution is terminated.
 -----*/
```

```
ElementType getMax() const;
/*-----
 Retrieve the largest element in the heap.

 Precondition: Heap is nonempty.
 Postcondition: Largest element is returned if heap is nonempty,
 otherwise a heap-empty message is displayed and a garbage value
 is returned.
-----*/
void removeMax();
/*-----
 Remove the largest element in the heap.

 Precondition: Heap is nonempty.
 Postcondition: Largest element is removed if heap is nonempty and result
 is still a heap; Otherwise a heap-empty message is displayed.
-----*/
void remove(int loc);
/*-----
 Remove the element in location loc.

 Precondition: 1 <= loc <= mySize.
 Postcondition: Element at location loc is removed and result is still a
 heap; otherwise a bad-location message is displayed.
-----*/
//--- Extra Functions to help visualize heaps
private:
 //---- DATA MEMBERS ----
 int mySize;
 ElementType myArray[HEAP_CAPACITY];

 //---- PRIVATE FUNCTION MEMBERS ----
 void percolateDown(int r, int n);
/*-----
 Percolate-down operation
 Precondition: myArray[r], ..., myArray[n] stores a semiheap.
 Postcondition: The semiheap has been converted into a heap.
-----*/
void heapify();
/*-----
 Heapify operation
 Precondition: myArray[1], ..., myArray[mySize] stores a complete binary
 tree.
 Postcondition: The complete binary tree has been converted into a heap.
-----*/
};

//--- Definition of constructor
template <typename ElementType>
inline Heap<ElementType>::Heap()
 : mySize(0)
{ }
```

```
//--- Definition of empty()
template <typename ElementType>
inline bool Heap<ElementType>::empty() const
{ return mySize == 0; }

//--- Definition of getSize()
template <typename ElementType>
inline int Heap<ElementType>::getSize() const
{ return mySize; }

//--- Definition of getArray()
template <typename ElementType>
inline ElementType * Heap<ElementType>::getArray()
{ return myArray; }

//--- Definition of insert()
template <typename ElementType>
void Heap<ElementType>::insert(ElementType item)
{
 if (mySize >= HEAP_CAPACITY)
 {
 cerr << "No more room in heap -- increase its capacity\n";
 exit(1);
 }
 //else
 mySize++;
 myArray[mySize] = item;
 int loc = mySize,
 parent = loc / 2;

 while (parent >= 1 && myArray[loc] > myArray[parent])
 {
 //-- Swap elements at positions loc and parent
 ElementType temp = myArray[loc];
 myArray[loc] = myArray[parent];
 myArray[parent] = temp;
 loc = parent;
 parent = loc/ 2;
 }
}

//--- Definition of getMax()
template <typename ElementType>
ElementType Heap<ElementType>::getMax() const
{
 if (!empty())
 return myArray[1];
 //else
 cerr << "Heap is empty -- garbage value returned\n";
 ElementType garbage;
 return garbage;
}
```

```
//--- Definition of removeMax()
template <typename ElementType>
void Heap<ElementType>::removeMax()
{
 if (!empty())
 remove(1);
 else
 cerr << "Heap is empty -- no element removed";
}

//--- Definition of remove()
template <typename ElementType>
void Heap<ElementType>::remove(int loc)
{
 if (1 <= loc and loc <= mySize)
 {
 myArray[loc] = myArray[mySize];
 mySize--;
 percolateDown(loc, mySize);
 }
 else
 cerr << "Illegal location in heap: " << loc << endl;
}

//--- Definition of percolateDown()
template <typename ElementType>
void Heap<ElementType>::percolateDown(int r, int n)
{
 int c;
 for (c = 2*r; c <= n;)
 {
 if (c < n && myArray[c] < myArray[c+1])
 c++;
 // Interchange node and largest child, if necessary
 // move down to the next subtree.
 if (myArray[r] < myArray[c])
 {
 ElementType temp = myArray[r];
 myArray[r] = myArray[c];
 myArray[c] = temp;
 r = c;
 c *= 2;
 }
 else
 break;
 }
}

//--- Definition of heapify()
template <typename ElementType>
void Heap<ElementType>::heapify()
{
 for (int r = mySize/2; r > 0; r--)
 percolateDown(r, mySize);
}
```

16.

```
#include <iostream>
using namespace std;
#include "Heap.h" // file containing Heap class template

const int PQ_CAPACITY = HEAP_CAPACITY;
template <typename ElementType>
/* < is assumed to be defined for type ElementType so that
 x < y if x's priority < y's priority. */

class PriorityQueue
{
public:

 PriorityQueue();
/*-----
 Constructor

 Precondition: None.
 Postcondition: An empty priority queue that can store PQ_CAPACITY elements
 has been constructed.
-----*/
 bool empty();
/*-----
 Check if priority queue is empty.

 Precondition: None.
 Postcondition: True is returned if priority queue is empty, false if not.
-----*/
 void insert(ElementType item);
/*-----
 Insert operation

 Precondition: mySize < PQ_CAPACITY.
 Postcondition: item has been inserted into the priority queue so the
 result is still a priority queue, provided there is room in myHeap;
 otherwise, a priority-queue-full message is displayed and execution
 is terminated.
-----*/
 ElementType getMax();
/*-----
 Retrieve the largest (i.e., with highest priority) element in the
 priority queue.

 Precondition: Priority queue is nonempty.
 Postcondition: Largest element is returned if priority queue is nonempty,
 otherwise a priority-queue-empty message is displayed and a garbage
 value is returned.
-----*/
```

```
void removeMax();
/*
----- Remove the largest (i.e., with highest priority) element in the
priority queue.

Precondition: Priority queue is nonempty.
Postcondition: Largest element is removed if priority queue is nonempty
and result is still a priority queue; Otherwise a priority-queue-
empty message is displayed.
----- */

void display(ostream & out);
/*
----- Display elements of priority queue.

Precondition: ostream out is open.
Postcondition: Elements of priority queue have been displayed (from
front to back) to out.
----- */

private:
 Heap<ElementType> myHeap;
};

//--- Definition of constructor
template <typename ElementType>
inline PriorityQueue<ElementType>::PriorityQueue()
{
 // Let Heap constructor do the work
}

//--- Definition of empty()
template <typename ElementType>
inline bool PriorityQueue<ElementType>::empty()
{
 myHeap.empty();
}

//--- Definition of insert()
template <typename ElementType>
inline void PriorityQueue<ElementType>::insert(ElementType item)
{
 if(myHeap.getSize() < PQ_CAPACITY)
 myHeap.insert(item);
 else
 {
 cerr << "No more room in priority queue -- increase its capacity\n";
 exit(1);
 }
}
```

```

//--- Definition of getMax()
template <typename ElementType>
ElementType PriorityQueue<ElementType>::getMax()
{
 return myHeap.getMax();
}

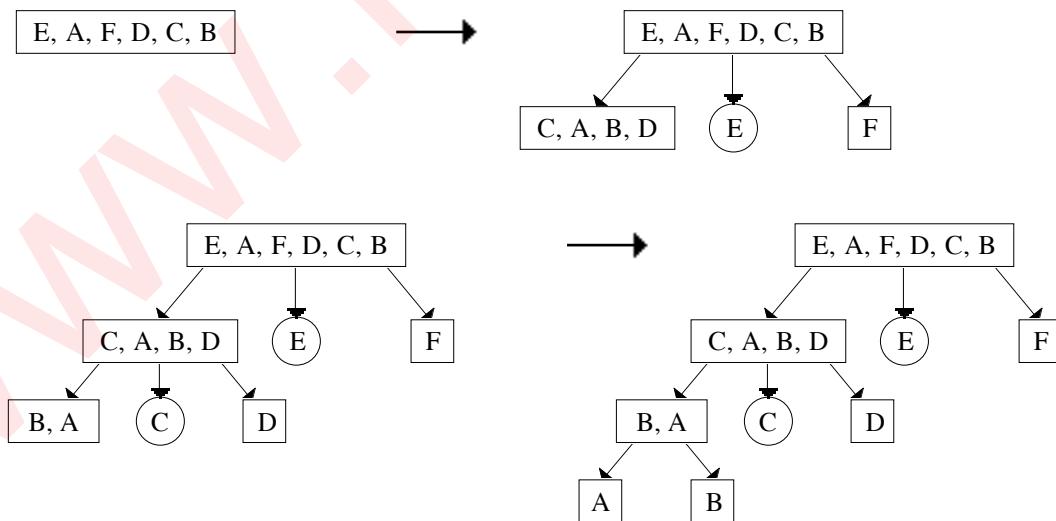
//--- Definition of removeMax()
template <typename ElementType>
void PriorityQueue<ElementType>::removeMax()
{
 myHeap.removeMax();
}

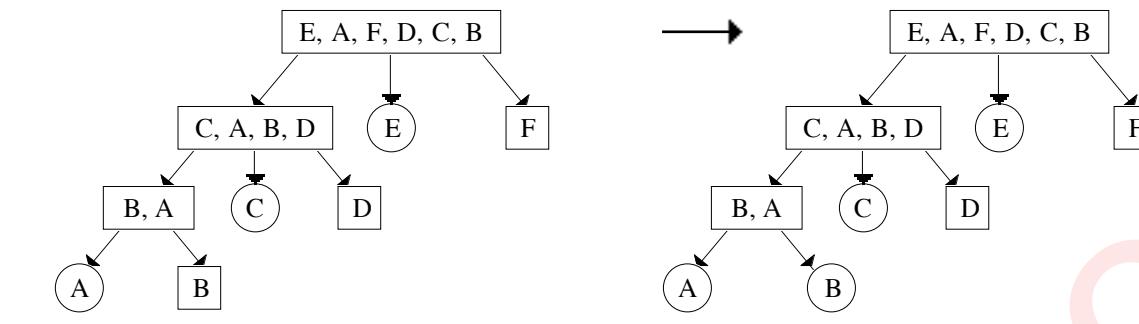
//--- Definition of display()
template <typename ElementType>
void PriorityQueue<ElementType>::display(ostream & out)
{
 for (int i = 1; i <= myHeap.getSize(); i++)
 out << myHeap.getArray()[i] << " ";
 out << endl;
}

```

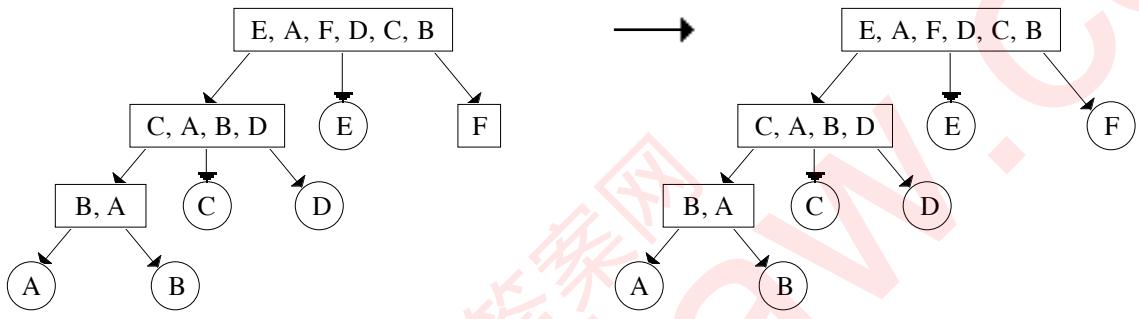
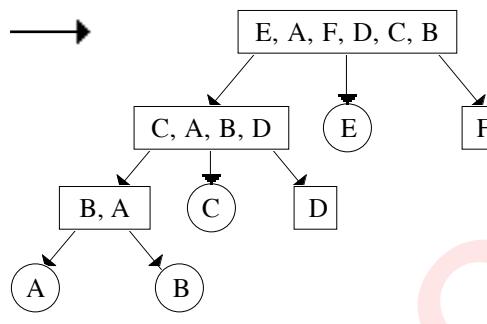
### Exercises 13.3

1. The array elements are 10 20 40 30 45 80 60 70 50 90.
2. The following diagram shows the sequence for trees for Exercise 2. Only the final trees for Exercises 3, 4 and 5 are given.

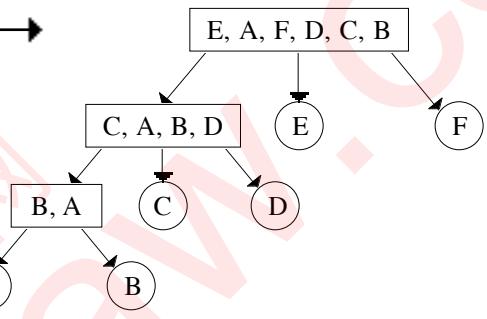




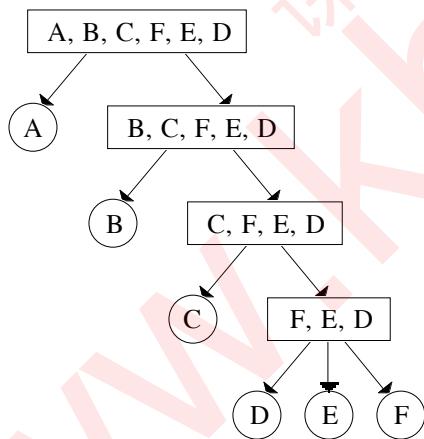
→



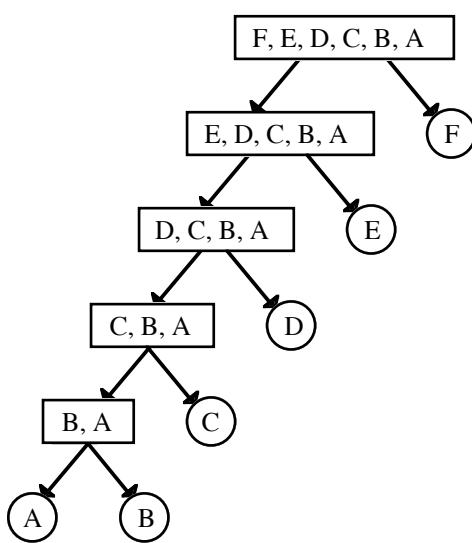
→



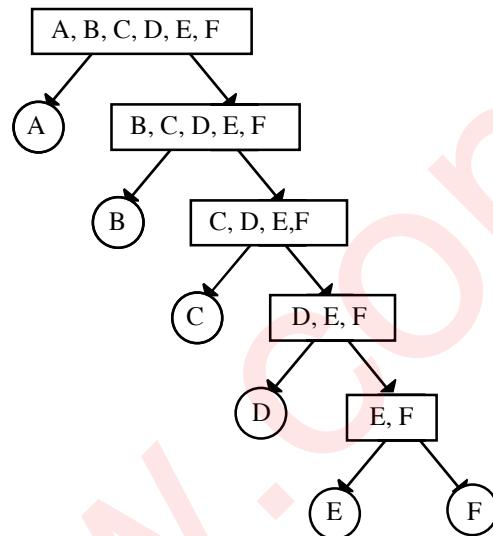
3.



4.



5.



6. In #4, the compound boolean expression prevents the left pointer from going off the right end of the list.

7.

```

template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*
 Modified quicksort of array elements x[first], ..., x[last] so
 they are in ascending order. Small lists (of size < LOWER_BOUND
 are sorted using insertion sort.

 Precondition: < and == are defined for ElementType.
 Note: Client programs call quicksort with first = 1
 and last = n, where n is the list size.
 Postcondition: x[first], ..., x[last] is sorted.
*/
{
 const int LOWER_BOUND = 20;
 if (last - first < LOWER_BOUND) // Small list
 insertionSort(x, first, last);
 else
 {
 int mid = split(x, first, last);
 quicksort(x, first, mid-1);
 quicksort(x, mid+1, last);
 }
}

```

8.

```

template <typename ElementType>
void quicksortAux(ElementType x[], int first, int last)
/*-
 Auxiliary function that does the actual quicksorting.
*/
{
 const int LOWER_BOUND = 20;
 if (last - first >= LOWER_BOUND)
 {
 int mid = split(x, first, last);
 quicksort(x, first, mid-1);
 quicksort(x, mid+1, last);
 }
}

template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*-
 Modified quicksort of array elements x[first], ..., x[last] so
 they are in ascending order. Small lists (of size < LOWER_BOUND
 are left unsorted, and a final insertion sort used at the end.

 Precondition: < and == are defined for ElementType.
 Note: Client programs call quicksort with first = 1
 and last = n, where n is the list size.
 Postcondition: x[first], ..., x[last] is sorted.
*/
{
 quicksortAux(x, first, last);
 insertionSort(x, first, last);
}

```

9.

```

template <typename ElementType>
int split(ElementType x[], int first, int last)
/*-
 Rearrange x[first], ... , x[last] to position pivot.

 Precondition: < and == are defined for ElementType;
 first <= last. Note that this version of split()
 uses the median-of-three rule to select the pivot
 Postcondition: Elements of sublist are rearranged and pos
 returned so x[first],..., x[pos-1] <= pivot and
 pivot < x[pos+1],..., x[last].
*/
{
 int mid = (first + last) / 2;
 ElementType item1 = x[first],
 item2 = x[mid],
 item3 = x[last],
 pivot;

```

```

if ((item2 < item1 && item1 < item3)
 || (item3 < item1 && item1 < item3))
{
 pivot = item1;
 mid = first;
}
else if ((item1 < item2 && item2 < item3)
 || (item3 < item2 && item2 < item1))
 pivot = item2;
else
{
 pivot = item3;
 mid = last;
}

// Put pivot in position first
x[mid] = x[first];
x[first] = pivot;

int left = first;
int right = last;

while (left < right)
{
 while (x[right] > pivot)
 right--;
 while (left < right && x[left] <= pivot)
 left++;
 if (left < right)
 // swap elements at positions left and right
 {
 ElementType temp = x[left];
 x[left] = x[right];
 x[right] = temp;
 }
}

mid = right;
x[first] = x[mid];
x[mid] = pivot;
return mid;
}

```

10.

```

template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*
-----*
Nonrecursive version of quicksort to sort array elements
x[first], ..., x[last] so they are in ascending order.
Uses a stack to store "recursive" calls.

Precondition: < and == are defined for ElementType.
Note: Client programs call quicksort with first = 1
 and last = n, where n is the list size.
Postcondition: x[first], ..., x[last] is sorted.
-----*/

```

```

{
 int mid;
 stack<int> s;

 s.push(first);
 s.push(last);

 while(!s.empty())
 {
 last = s.top();
 s.pop();
 first = s.top();
 s.pop();
 if (first < last)
 {
 mid = split(x, first, last);
 s.push(first);
 s.push(mid-1);
 s.push(mid+1);
 s.push(last);
 }
 }
}

```

11.

```

template <typename ElementType>
int median(ElementType x[], int first, int last, int mid)
/*-----
 Find the median of a list using a quicksort scheme.

 Precondition: < and == are defined for ElementType.
 Note: Client programs call median() with first = 1
 last = n, mid = (n + 1)/2, where n is the list size.
 Postcondition: Index of median element is returned.
-----*/
{
 int pos = split(x, first, last);
 if (pos > mid)
 return median(x, first, pos - 1, mid);
 else if (pos < mid)
 return median(x, pos + 1, last, mid);
 else
 return pos;
}

```

12. This is a simple modification of #11.

Call `median()` with `median(array, 1, n, k)`.

## Exercises 13.4

1. **F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 70 | 22 | 64 | 48 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 13 | 39 | 70 | 64 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 57 | 85 | 22 | 48 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 22 | 70 | 48 | 64 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |    |
|----|----|--|----|----|
| 13 | 57 |  | 22 | 70 |
|----|----|--|----|----|

**F2**

|    |    |  |    |    |
|----|----|--|----|----|
| 39 | 85 |  | 48 | 64 |
|----|----|--|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 39 | 57 | 85 | 22 | 48 | 64 | 70 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 13 | 39 | 57 | 85 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 22 | 48 | 64 | 70 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

2. **F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 99 | 70 | 22 | 48 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |
|----|----|----|----|----|
| 13 | 39 | 99 | 22 | 64 |
|----|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 57 | 85 | 70 | 48 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 70 | 99 | 22 | 48 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |    |  |    |
|----|----|--|----|----|--|----|
| 13 | 57 |  | 70 | 99 |  | 64 |
|----|----|--|----|----|--|----|

**F2**

|    |    |  |    |    |  |
|----|----|--|----|----|--|
| 39 | 85 |  | 22 | 48 |  |
|----|----|--|----|----|--|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 39 | 57 | 85 | 22 | 48 | 70 | 99 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |  |    |
|----|----|----|----|--|----|
| 13 | 39 | 57 | 85 |  | 64 |
|----|----|----|----|--|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 22 | 48 | 70 | 99 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 70 | 85 | 99 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 70 | 85 | 99 |
|----|----|----|----|----|----|----|----|

**F2**

|    |
|----|
| 64 |
|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 | 99 |
|----|----|----|----|----|----|----|----|----|

3. **F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 57 | 99 | 39 | 64 | 57 | 48 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |
|----|----|----|----|----|
| 13 | 57 | 39 | 57 | 70 |
|----|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 22 | 99 | 64 | 48 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 57 | 99 | 39 | 64 | 48 | 57 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |  |    |
|----|----|----|----|--|----|
| 13 | 22 | 39 | 64 |  | 70 |
|----|----|----|----|--|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 57 | 99 | 48 | 57 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 57 | 99 | 39 | 48 | 57 | 64 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |  |    |
|----|----|----|----|--|----|
| 13 | 22 | 57 | 99 |  | 70 |
|----|----|----|----|--|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 39 | 48 | 57 | 64 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 57 | 64 | 99 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 57 | 64 | 99 |
|----|----|----|----|----|----|----|----|

**F2**

|    |
|----|
| 70 |
|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 57 | 64 | 70 | 99 |
|----|----|----|----|----|----|----|----|----|

4. **F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 13 | 39 | 57 | 70 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 22 | 48 | 64 | 85 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |    |
|----|----|--|----|----|
| 13 | 22 |  | 57 | 64 |
|----|----|--|----|----|

**F2**

|    |    |  |    |    |
|----|----|--|----|----|
| 39 | 48 |  | 70 | 85 |
|----|----|--|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 13 | 22 | 39 | 48 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 57 | 64 | 70 | 85 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

5. **F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 85 | 70 | 64 | 57 | 48 | 39 | 22 | 13 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 85 | 64 | 48 | 22 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 70 | 57 | 39 | 13 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 70 | 85 | 57 | 64 | 39 | 48 | 13 | 22 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |    |
|----|----|--|----|----|
| 70 | 85 |  | 39 | 48 |
|----|----|--|----|----|

**F2**

|    |    |  |    |    |
|----|----|--|----|----|
| 57 | 64 |  | 13 | 22 |
|----|----|--|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 57 | 64 | 70 | 85 | 13 | 22 | 39 | 48 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |
|----|----|----|----|
| 57 | 64 | 70 | 85 |
|----|----|----|----|

**F2**

|    |    |    |    |
|----|----|----|----|
| 13 | 22 | 39 | 48 |
|----|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

6. **F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 70 | 22 | 64 | 48 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |  |    |
|----|----|--|----|--|----|
| 13 | 57 |  | 70 |  | 48 |
|----|----|--|----|--|----|

**F2**

|    |    |  |    |    |  |
|----|----|--|----|----|--|
| 39 | 85 |  | 22 | 64 |  |
|----|----|--|----|----|--|

**F1**

|    |    |    |  |    |
|----|----|----|--|----|
| 13 | 57 | 70 |  | 48 |
|----|----|----|--|----|

**F2**

|    |    |  |    |    |
|----|----|--|----|----|
| 39 | 85 |  | 22 | 64 |
|----|----|--|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 39 | 57 | 70 | 85 | 22 | 48 | 64 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |
|----|----|----|----|----|
| 13 | 39 | 57 | 70 | 85 |
|----|----|----|----|----|

**F2**

|    |    |    |
|----|----|----|
| 22 | 48 | 64 |
|----|----|----|

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

7. **F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 57 | 39 | 85 | 99 | 70 | 22 | 48 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |  |    |
|----|----|--|----|
| 13 | 57 |  | 70 |
|----|----|--|----|

**F2**

|    |    |    |  |    |    |    |
|----|----|----|--|----|----|----|
| 39 | 85 | 99 |  | 22 | 48 | 64 |
|----|----|----|--|----|----|----|

**F1**

|    |    |    |
|----|----|----|
| 13 | 57 | 70 |
|----|----|----|

**F2**

|    |    |    |  |    |    |    |
|----|----|----|--|----|----|----|
| 39 | 85 | 99 |  | 22 | 48 | 64 |
|----|----|----|--|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 39 | 57 | 70 | 85 | 99 | 22 | 48 | 64 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 13 | 39 | 57 | 70 | 85 | 99 |
|----|----|----|----|----|----|

**F2**

|    |    |    |
|----|----|----|
| 22 | 48 | 64 |
|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 | 99 |
|----|----|----|----|----|----|----|----|----|

8. **F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 57 | 99 | 39 | 64 | 57 | 48 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |  |    |
|----|----|----|----|--|----|
| 13 | 22 | 57 | 99 |  | 57 |
|----|----|----|----|--|----|

**F2**

|    |    |  |    |    |
|----|----|--|----|----|
| 39 | 64 |  | 48 | 70 |
|----|----|--|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 57 | 64 | 99 | 48 | 57 | 70 |
|----|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 13 | 22 | 39 | 57 | 64 | 99 |
|----|----|----|----|----|----|

**F2**

|    |    |    |
|----|----|----|
| 48 | 57 | 70 |
|----|----|----|

**F**

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 57 | 64 | 70 | 99 |
|----|----|----|----|----|----|----|----|----|

9. **F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

**F1**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

**F2**

**F**

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 39 | 48 | 57 | 64 | 70 | 85 |
|----|----|----|----|----|----|----|----|

10. This is the same as Exercise 5.

11.

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int merge(string & outName, string inName1, string inName2)
/*-----
 Merge sorted subfiles in two different files.

 Precondition: Files named inName1 and inName2 contain sorted subfiles.
 Postcondition: File named outName contains the result of merging
 these sorted subfiles.
-----*/
{
 ofstream f(outName.data());
 ifstream f1(inName1.data()), f2(inName2.data());

 int in1;
 int in2;
 bool inSub1,
 inSub2;

 int numSubfiles = 0;
 int oldone1, oldone2;

 f1 >> in1;
 f2 >> in2;

 while (!f1.eof() && !f2.eof())
 {
 inSub1 = inSub2 = true;

 while (inSub1 && inSub2)
 {
 if (in1 < in2)
 {
 f << in1 << endl;
 oldone1 = in1;
 f1 >> in1;
 inSub1 = !f1.eof() && (oldone1 <= in1);
 }
 else
 {
 f << in2 << endl;
 oldone2 = in2;
 f2 >> in2;
 inSub2 = !f2.eof() && (oldone2 <= in2);
 }
 }
 }
}
```

```
if (inSub2)
 while (inSub2)
 {
 f << in2 << endl;
 oldone2 = in2;
 f2 >> in2;
 inSub2 = !f2.eof() && (oldone2 <= in2);
 }
else
 while (inSub1)
 {
 f << in1 << endl;
 oldone1 = in1;
 f1 >> in1;
 inSub1 = !f1.eof() && (oldone1 <= in1);
 }
numSubfiles++;
}

while (!f1.eof())
{
 f << in1 << endl;
 oldone1 = in1;
 f1 >> in1;
 if (!f1.eof())
 if (oldone1 > in1)
 numSubfiles++;
}

while (!f2.eof())
{
 f << in2 << endl;
 oldone2 = in2;
 f2 >> in2;
 if (!f2.eof())
 if (oldone2 > in2)
 numSubfiles++;
}
numSubfiles++;
return numSubfiles;
}

void split(ifstream & f, string outName1, string outName2)
/*
-----*
 Split file f by writing sorted subfiles alternately to the files
 named outName1 and outName2.

 Precondition: f is open for input.
 Postcondition: Files named outName1 and outName2 contain the result of
 splitting f.
-----*/
{
 ofstream f1(outName1.data()),
 f2(outName2.data());
```

```

bool inSub;
int oldone,
 value;

f >> value;
while (!f.eof())
{
 inSub = true;
 while (inSub)
 {
 f1 << value << endl;
 oldone = value;
 f >> value;
 inSub = (!f.eof()) && (oldone <= value);
 }

 if (!f.eof())
 {
 inSub = true;
 while (inSub)
 {
 f2 << value << endl;
 oldone = value;
 f >> value;
 inSub = (!f.eof()) && (oldone <= value);
 }
 }
}

void mergesort(string filename)
/*
----- Mergesort.

Precondition: None.
Postcondition: File named filename has been sorted into ascending order.
-----*/
{
 int subfiles = 2; // to prime the while loop
 while (subfiles > 1)
 {
 ifstream infile(filename.data());

 string outfile1 = "hold1",
 outfile2 = "hold2";;

 split(infile, outfile1, outfile2);

 subfiles = merge(filename, outfile1, outfile2);
 }
}

```

12. See #11; the process is essentially the same. Now, the *array limits* control the iteration rather than the *end-of-file*.

13. See #11, the process is essentially the same. Now the *end of the linked list* controls the iteration rather than the *end-of-file*.

14. Change mergesort() as follows:

```
void mergesort(string filename)
/*
----- Mergesort.

Precondition: None.
Postcondition: File named filename has been sorted into ascending order.
-----*/
{
 bool firstTime = true;
 int subfiles = 2; // to prime the while loop
 while (subfiles > 1)
 {
 ifstream infile(filename.data());
 string outfile1 = "hold1",
 outfile2 = "hold2";
 if (firstTime)
 {
 firstSplit(infile, outfile1, outfile2);
 firstTime = false;
 }
 else
 split(infile, outfile1, outfile2);
 subfiles = merge(filename, outfile1, outfile2);
 }
}
```

where firstSplit() is:

```
void firstSplit(ifstream & f, string outName1, string outName2)
/*
----- Split file f by copying fixed-size subfiles into an array, quicksorting
these subfiles, and then writing these sorted arrays alternately to the
files named outName1 and outName2.

Precondition: f is open for input.
Postcondition: Files named outName1 and outName2 contain the result of
splitting f.
-----*/
{
 const int SUBFILE_SIZE = 8;
 ElementType internalStore[SUBFILE_SIZE + 1];
 ofstream f1(outName1.data()),
 f2(outName2.data());
 ElementType value;
 int filenum = 1;
```

```

do
{
 int count;
 for (count = 0; count < SUBFILE_SIZE; count++)
 {
 f >> value;
 if (f.eof()) break;
 internalStore[count + 1] = value;
 }

 quicksort(internalStore, 1, count);
 switch (filenum)
 {
 case 1: for (int i = 1; i <= count; i++)
 f1 << internalStore[i] << endl;
 break;
 case 2: for (int i = 1; i <= count; i++)
 f2 << internalStore[i] << endl;
 }
 filenum = 3 - filenum;
}
while (!f.eof());
}

```

15. A three-way merge is similar to a two-way merge, except now the comparison is between three files and the smallest element of the three starts the copying of that subfile to the output file.

### Exercises 13.5

1. 029, 778, 11, 352, 233, 710, 783, 812, 165, 106

Distribute:

|          |          |            |            |          |          |          |          |          |          |
|----------|----------|------------|------------|----------|----------|----------|----------|----------|----------|
| 710      | 011      | 812<br>352 | 783<br>233 |          | 165      | 106      |          | 778      | 029      |
| <b>0</b> | <b>1</b> | <b>2</b>   | <b>3</b>   | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

710, 011, 352, 812, 233, 783, 165, 106, 778, 029

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 812      |          |          |          |          |          |          |          |          |          |
| 011      |          |          |          |          |          |          |          |          |          |
| 106      | 710      | 029      | 233      |          | 352      | 165      | 778      | 783      |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

106, 710, 011, 812, 029, 233, 352, 165, 778, 783

Distribute:

|            |            |          |          |          |          |          |                   |          |          |
|------------|------------|----------|----------|----------|----------|----------|-------------------|----------|----------|
| 029<br>011 | 165<br>106 | 233      | 352      |          |          |          | 783<br>778<br>710 | 812      |          |
| <b>0</b>   | <b>1</b>   | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b>          | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

011, 029, 106, 165, 233, 352, 710, 778, 783, 812

2. 038, 399, 892, 389, 683, 400, 937, 406, 316, 005

Distribute:

|          |          |          |          |          |          |            |          |          |            |
|----------|----------|----------|----------|----------|----------|------------|----------|----------|------------|
| 400      |          | 892      | 683      |          | 005      | 316<br>406 | 937      | 038      | 389<br>399 |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b>   | <b>7</b> | <b>8</b> | <b>9</b>   |

Collect together from left to right, bottom to top:

400, 892, 683, 005, 406, 316, 937, 038, 399, 389

Distribute:

|                   |          |          |            |          |          |          |          |            |            |
|-------------------|----------|----------|------------|----------|----------|----------|----------|------------|------------|
| 406<br>005<br>400 |          |          | 038<br>937 |          |          |          |          | 389<br>683 | 399<br>892 |
| <b>0</b>          | <b>1</b> | <b>2</b> | <b>3</b>   | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b>   | <b>9</b>   |

Collect together from left to right, bottom to top:

400, 005, 406, 316, 937, 038, 683, 389, 892, 399

Distribute:

|            |          |          |                   |            |          |          |          |          |            |
|------------|----------|----------|-------------------|------------|----------|----------|----------|----------|------------|
| 038<br>005 |          |          | 399<br>389<br>316 | 406<br>499 |          |          | 683      |          | 892<br>937 |
| <b>0</b>   | <b>1</b> | <b>2</b> | <b>3</b>          | <b>4</b>   | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b>   |

Collect together from left to right, bottom to top:

005, 038, 316, 389, 399, 400, 406, 683, 892, 937

3. 353, 6, 295, 44, 989, 442, 11, 544, 209, 46

Distribute:

|          |          |          |          |            |          |            |          |          |            |
|----------|----------|----------|----------|------------|----------|------------|----------|----------|------------|
|          | 011      | 442      | 353      | 544<br>044 | 295      | 046<br>006 |          |          | 209<br>989 |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b>   | <b>5</b> | <b>6</b>   | <b>7</b> | <b>8</b> | <b>9</b>   |

Collect together from left to right, bottom to top:

011, 442, 353, 044, 544, 295, 006, 046, 989, 209

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          |          |          |          | 046      |          |          |          |          |          |
| 209      |          |          |          | 544      |          |          |          |          |          |
| 006      | 011      |          |          | 044      |          |          |          |          |          |
|          |          |          |          | 442      | 353      |          |          |          |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

006, 209, 011, 442, 044, 544, 046, 353, 989, 295

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 046      |          |          |          |          |          |          |          |          |          |
| 044      |          |          |          |          |          |          |          |          |          |
| 011      |          | 295      |          |          |          |          |          |          |          |
| 006      |          | 209      | 353      | 442      | 544      |          |          |          |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

006, 011, 044, 046, 209, 295, 353, 442, 544, 989

4. 8745, 7438, 15, 12, 8501, 3642, 8219, 6152, 369, 6166, 8583, 7508, 8717, 8114, 630

Distribute:

|          |          |          |          |          |          |          |          |          |          |      |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|------|
|          |          | 6152     |          |          |          | 0015     |          |          | 7508     | 0369 |
| 0630     | 8501     | 3642     |          |          |          | 8745     | 6166     | 8717     | 7438     | 8219 |
|          |          | 0012     | 3583     | 8114     |          |          |          |          |          |      |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |      |

Collect together from left to right, bottom to top:

0630, 8501, 0012, 3642, 6152, 8583, 8114, 8745, 0015, 6166, 8717, 7438, 7508, 8219, 0369

Distribute:

|          |          |          |          |          |          |          |          |          |          |  |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|--|
|          | 8219     |          |          |          |          |          |          |          |          |  |
| 7508     | 8717     |          |          |          |          |          |          |          |          |  |
|          | 0015     |          |          |          |          |          |          |          |          |  |
| 8501     | 8114     |          | 7438     | 8745     |          | 0369     |          |          |          |  |
|          | 0012     |          | 0630     | 3642     | 6152     | 6166     |          |          |          |  |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |  |

Collect together from left to right, bottom to top:

8501, 7508, 0012, 8114, 0015, 8717, 8219, 0630, 7438, 3642, 8745, 6152, 6166, 0369, 8583

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          | 6166     |          |          |          | 8583     |          |          |          |          |
| 0015     | 6152     |          |          |          | 7508     | 3642     | 8745     |          |          |
| 0012     | 8114     | 8219     | 0369     | 7438     | 8501     | 0630     | 8717     |          |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

0012, 0015, 8114, 6152, 6166, 8219, 0369, 7438, 8501, 7508, 8583, 0630, 3642, 8717, 8745

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          |          |          |          |          |          |          |          | 8745     |          |
| 0630     |          |          |          |          |          |          |          | 8717     |          |
| 0369     |          |          |          |          |          |          |          | 8583     |          |
| 0015     |          |          |          |          |          |          |          | 8501     |          |
| 0012     |          |          | 3642     |          |          | 6166     | 7508     | 8219     |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

0012, 0015, 0369, 0630, 3642, 6152, 6166, 7438, 7508, 8114, 8219, 8501, 8583, 8717, 8745

5. 9001, 78, 8639, 252, 9685, 3754, 4971, 888, 6225, 9686, 6967, 6884, 2, 4370, 131

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          | 0131     |          |          |          |          |          |          |          |          |
| 4971     |          | 0002     |          |          |          |          |          |          |          |
| 4730     | 9001     | 0252     |          |          | 6884     | 6225     |          |          |          |
|          |          |          |          |          | 3754     | 9685     | 9686     | 3937     |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

4370, 9001, 4971, 0131, 0252, 0002, 3754, 6884, 9685, 6225, 9686, 6967, 0078, 0888, 8639

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          |          |          |          |          |          |          |          |          |          |
| 0002     |          |          |          |          |          |          |          |          |          |
| 9001     |          |          | 6225     | 8639     |          |          |          |          |          |
|          |          |          | 0131     |          |          | 3754     |          |          |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

9001, 0002, 6225, 0131, 8639, 0252, 3754, 6967, 4370, 4971, 0078, 6884, 9685, 9686, 0888

Distribute:

|              |          |              |          |          |          |                      |          |              |              |
|--------------|----------|--------------|----------|----------|----------|----------------------|----------|--------------|--------------|
| 0002<br>9001 | 0131     | 0252<br>6225 | 4370     |          |          | 9686<br>9685<br>8639 | 3754     | 0888<br>6884 | 4971<br>6967 |
| <b>0</b>     | <b>1</b> | <b>2</b>     | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b>             | <b>7</b> | <b>8</b>     | <b>9</b>     |

Collect together from left to right, bottom to top:

9001, 0002, 0078, 0131, 6225, 0252, 4370, 8639, 9685, 9686, 3754, 6884, 0888, 6967, 4971

Distribute:

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0888     |          |          |          |          |          | 6967     |          | 9686     |          |
| 0252     |          |          |          |          |          | 6884     |          | 9685     |          |
| 0131     |          |          |          | 4971     |          | 6225     |          | 8639     |          |
| 0078     |          |          | 3754     | 4370     |          |          |          |          | 9001     |
| 0002     |          |          |          |          |          |          |          |          |          |
| <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> |

Collect together from left to right, bottom to top:

0002, 0078, 0131, 0252, 0888, 3754, 4370, 4971, 6225, 6884, 6967, 8639, 9001, 9685, 9686

6. for#, if##, do##, else, case, int#, main (# denotes a blank)

Distribute:

|     |              |     |      |     |                             |
|-----|--------------|-----|------|-----|-----------------------------|
|     | case<br>else |     | main |     | for#<br>if##<br>do#<br>int# |
| ... | e            | ... | n    | ... | blank                       |

Collect together from left to right, bottom to top:

else, case, if##, main, do##, for#, int#

Distribute:

|     |      |     |      |              |      |     |              |
|-----|------|-----|------|--------------|------|-----|--------------|
|     | main |     | for# | case<br>else | int# |     | do##<br>if## |
| ... | i    | ... | r    | s            | t    | ... | blank        |

Collect together from left to right, bottom to top:

main, for#, else, case, int#, if##, do##

Distribute:

|          |     |          |     |          |     |          |          |     |
|----------|-----|----------|-----|----------|-----|----------|----------|-----|
| case     |     | if##     |     | else     |     | int#     | do##     |     |
| main     |     |          |     |          |     |          | for#     |     |
| <b>a</b> | ... | <b>i</b> | ... | <b>l</b> | ... | <b>n</b> | <b>o</b> | ... |

Collect together from left to right, bottom to top:

main, case, if##, else, int#, for#, do##

Distribute:

|     |          |          |          |          |     |          |     |          |
|-----|----------|----------|----------|----------|-----|----------|-----|----------|
|     | case     | do##     | else     | for#     |     | int#     |     | main     |
| ... | <b>c</b> | <b>d</b> | <b>e</b> | <b>f</b> | ... | <b>i</b> | ... | <b>m</b> |
|     |          |          |          |          |     |          |     |          |

Collect together from left to right, bottom to top:

case, do##, else, for#, if##, int#, main

7. while, if###, for##, break, float, bool# (# denotes a blank)

Distribute:

|     |          |     |          |     |          |     |              |  |
|-----|----------|-----|----------|-----|----------|-----|--------------|--|
|     | while    |     | break    |     | float    |     | bool#        |  |
| ... | <b>e</b> | ... | <b>k</b> | ... | <b>t</b> | ... | <b>blank</b> |  |
|     |          |     |          |     |          |     | if###        |  |

Collect together from left to right, bottom to top:

while, break, float, do###, for##, if###, bool#

Distribute:

|          |     |          |     |          |     |              |  |       |
|----------|-----|----------|-----|----------|-----|--------------|--|-------|
| float    |     |          |     |          |     | if###        |  |       |
| break    |     | while    |     | bool#    |     | for##        |  | do### |
| <b>a</b> | ... | <b>l</b> | ... | <b>o</b> | ... | <b>blank</b> |  |       |

Collect together from left to right, bottom to top:

break, float, while, bool#, do###, for##, if###

Distribute:

|          |     |          |     |          |     |          |     |              |
|----------|-----|----------|-----|----------|-----|----------|-----|--------------|
| break    |     | while    |     | bool#    |     | for##    |     | if###        |
|          |     |          |     | float    |     |          |     | do###        |
| <b>a</b> | ... | <b>i</b> | ... | <b>o</b> | ... | <b>r</b> | ... | <b>blank</b> |

Collect together from left to right, bottom to top:

break, while, float, bool#, for##, do###, if###

Distribute:

|     |       |     |       |     |       |     |                         |     |       |     |
|-----|-------|-----|-------|-----|-------|-----|-------------------------|-----|-------|-----|
|     | if### |     | while |     | float |     | do###<br>for##<br>bool# |     | break |     |
| ... | f     | ... | h     | ... | l     | ... | o                       | ... | r     | ... |

Collect together from left to right, bottom to top:  
if###, while, float, bool#, for##, do###, break

Distribute:

|     |       |     |       |     |                |     |       |     |       |     |
|-----|-------|-----|-------|-----|----------------|-----|-------|-----|-------|-----|
|     | break |     | do### |     | for##<br>float |     | if### |     | while |     |
| ... | b     | ... | d     | ... | f              | ... | i     | ... | w     | ... |

Collect together from left to right, bottom to top:  
bool#, break, do###, float, for##, if###, while

8. Selection sort is not stable. Consider the following list of records consisting of an integer and a character:

[2, A], [2, B], [1, C]

Sorting so integers are in ascending order gives:

[1, C], [2, B], [2, A]

The relative order of the 2's has changed.

9. Bubble sort is stable.  
10. Insertion sort is stable.  
11. Heapsort is not stable. See the example from #8.  
12. Quicksort is not stable. See the example from #8.  
13. Binary Mergesort is not stable. Consider this list : [2, A], [2, B], [1, C], [3, C]

Again, sorting so integers are in ascending order gives: [1, C], [2, B], [2, A], [3, C].

The relative order of the 2's has changed.

14. Natural Mergesort is not stable. See example from #13.  
15. Radix sort is stable.

16-17.

```

#include <iostream>
#include <list>
#include <iomanip>
using namespace std;

typedef int ElementType;
void radixSort(list<ElementType> & x, int numDigits, int base)
{
 list<ElementType> * bucket = new list<ElementType>[base];
 int basePower = 1;
 ElementType value;

 for (int pass = 1; pass <= numDigits; pass++)
 {
 while (!x.empty())
 {
 value = x.front();
 x.pop_front();
 int digit = value % (base * basePower) / basePower;
 bucket[digit].push_back(value);
 }

 for (int i = 0; i < base; i++)
 while (!bucket[i].empty())
 {
 value = bucket[i].front();
 x.push_back(value);
 bucket[i].pop_front();
 }
 }

 basePower *= base;

// UNCOMMENT THE FOLLOWING LINES TO TRACE RADIX SORT
/*
#include <iomanip>
cout << pass << ":" ;
for (list<ElementType>::iterator it = x.begin(); it != x.end(); it++)
 cout << setfill('0') << setw(numDigits) << *it << ", ";
cout << endl;
*/
 }
}

```

18. The function in the preceding exercise can be easily modified for this.

```

#include <iostream>
#include <list>
#include <string>
#include <cctype>
using namespace std;

```

```
typedef string ElementType;
void radixSort(list<ElementType> & x, int maxLength)
{
 list<ElementType> * bucket = new list<ElementType>[27];
 ElementType value;

 for (int pass = maxLength - 1; pass >= 0; pass--)
 {
 while (!x.empty())
 {
 value = x.front();
 x.pop_front();

 int charPos;
 if (value[pass] != ' ')
 charPos = int(value[pass]) - int('a');
 else
 charPos = 26;
 bucket[charPos].push_back(value);
 }

 for (int i = 0; i <= 26; i++)
 while (!bucket[i].empty())
 {
 value = bucket[i].front();
 x.push_back(value);
 bucket[i].pop_front();
 }
 }

 // UNCOMMENT THE FOLLOWING LINES TO TRACE RADIX SORT
 /*
 cout << pass << ":" ;
 for (list<ElementType>::iterator it = x.begin(); it != x.end(); it++)
 cout << *it << ", ";
 cout << endl;
 */
}
}
```

## Chapter 14: OOP and ADTs

### Exercises 14.2

1. One can view a derived class as consisting of two components: the portion inherited (data members and function members defined by the base class) and the portion explicitly declared in the derived class definition. Access to each component is governed by the declared accessibility in the class definitions as well as the type of inheritance.

*Public inheritance* allows instances of the derived class to access the public and protected members of its base-class component. For example, `object.publicBaseFunction()` is legal. Moreover, these members retain their visibility for any descendants of the derived class. This form of inheritance provides the most accessibility to the base-class component; only private members of the base class are not directly accessible in the derived class.

*Protected inheritance* allows instances of the derived class to access the public and protected members of its base class component only in a protected manner. Thus, `object.publicBaseFunction()` is *not* legal, but within the derived class implementation, `publicBaseFunction()` would be. Also, any descendants of the derived class would also be able to use the inherited members internally. This form of inheritance allows a derived class to use the public and protected members of the base class in an internal manner but not to open them up to the "outside world."

*Private inheritance* essentially prevents the derived class from accessing any members of the base class externally or from passing on access to these members to any descendants. It is not commonly used. An example use is when one wants to completely redefine the interface of the base class but wants to use that interface internally.

2. An *is-a* relationship refers to a relationship between two objects when one is defined (even if only partially) by the other, i.e., by inheritance. For example, a hunting license is a license. The relationship must be defined by innate characteristics or functionally and *not* by implementation. For example, a stack may be implemented by an array, but it is not correct to say that a stack *is an* array.

A *has-a* relationship refers to containment — when one object contains another. For example, an employee *has a* name.

A *uses-a* relationship refers to interaction between two objects in which one object employs the functionality of the other; they are not related by inheritance or composition.

3. A rectangle *has a* line.
4. A square *is a* rectangle.
5. None of the three relationships.
6. A student *has a* GPA.

7. A student *uses* the library.
8. A professor *is a* university employee.
9. An employee *is a* person.
10. None of the three relationships.
11. A stack *is an* ADT.
12. A binary tree *is a* tree.
13. A binary search tree *is a* binary tree.
14. A vector *is a* container.
15. A computer *is* electronic equipment
16. A computer *uses* an operating system.
17. None of the three relationships.
18. A computer *has a* CPU.

### Exercises 14.5

1.

```
/* Administrator.h -----
 Header file for a class Administrator derived from SalariedEmployee
 that adds the attributes unique to administrators.
 New operations: A constructor and an output operation.
-----*/
#include <iostream>
#include "SalariedEmployee.h"

#ifndef ADMINISTRATOR
#define ADMINISTRATOR

class Administrator : public SalariedEmployee
{
public:
 ***** Function Members *****
 Administrator (long id = 0, string last = "",
 string first = "", char initial = ' ', int dept = 0,
 double salary = 0, double bonus = 0);
/*
-----*
 Administrator constructor.

 Preconditions: None.
 Postconditions: Data members myIdNum, myLastName, myFirstName,
 myMiddleInitial, myDeptCode, and mySalary are initialized by
 the SalariedEmployee constructor; myBonus is initialized to
 bonus (default 0).
-----*/
```

```

-----*/
virtual void display(ostream & out) const;
/*-----
 Output function member.

 Precondition: ostream out is open.
 Postcondition: A text representation of this Administrator object
 has been output to out.
-----*/

// ... Other administrator operations ...

private:
 /***** Data Members *****/
 double myBonus;
};

//-- Definition of constructor
inline double Administrator::Administrator(
 long id, string last, string first, char initial,
 int dept, double salary, double bonus)
: SalariedEmployee (id, last, first, initial, dept, salary),
 myBonus(bonus)
{ }

//--- Definition of Administrator's display()
inline void Administrator::display(ostream & out) const
{
 SalariedEmployee::display(out); //inherited members
 out << "Bonus: $" << myBonus << endl; //local members
}

#endif

2.

/* FactoryWorker.h -----
 Header file for a class FactoryWorker derived from Employee
 that adds the attributes unique to factory workers.
 New operations: A constructor and an output operation.
-----*/
#include <iostream>
#include "Employee.h"

#ifndef FACTORY_WORKER
#define FACTORY_WORKER

class FactoryWorker : public Employee
{
public:
 /***** Function Members *****/

```

```
FactoryWorker(long id = 0, string last = "",
 string first = "", char initial = ' ', int dept = 0,
 double unitPay = 0, int numUnits = 0);
/*-----
 * FactoryWorker constructor.

 Preconditions: None.
 Postconditions: Data members myIdNum, myLastName, myFirstName,
 myMiddleInitial, and myDeptCode, are initialized by the Employee
 constructor; myPayPerUnit and myNumberOfUnits are initialized to
 unitPay (default 0) and numUnits (default 0), respectively.
-----*/

virtual void display(ostream & out) const;
/*-----
 * Output function member.

 Precondition: ostream out is open.
 Postcondition: A text representation of this FactoryWorker object
 has been output to out.
-----*/

// ... Other administrator operations ...

private:
 //***** Data Members *****/
 double myPayPerUnit;
 int myNumberOfUnits;
 double myWages;
};

//-- Definition of constructor
inline double FactoryWorker::FactoryWorker(
 long id, string last, string first, char initial,
 int dept, double unitPay, int numUnits)
: Employee (id, last, first, initial, dept),
myPayPerUnit(unitPay), myNumberOfUnits(numUnits)
{
 myWages = myPayPerUnit * myNumberOfUnits;
}

//--- Definition of FactoryWorker's display()
inline void FactoryWorker::display(ostream & out) const
{
 Employee::display(out); //inherited members
 out << myNumberOfUnits //local members
 << " units at $" << myPayPerUnit
 << " gives wages = $" << myWages << endl;
}

#endif
```

3.

```
/* Salesperson.h -----
 Header file for a class Salesperson derived from SalariedEmployee
 that adds the attributes unique to salespersons.
 New operations: A constructor and an output operation.

#include <iostream>
#include "SalariedEmployee.h"

#ifndef SALESPERSON
#define SALESPERSON

class Salesperson : public SalariedEmployee
{
public:
 ***** Function Members *****/
 Salesperson(long id = 0, string last = "",
 string first = "", char initial = ' ', int dept = 0,
 double salary = 0, double commission = 0);
 /*-----
 Salesperson constructor.

 Preconditions: None.
 Postconditions: Data members myIdNum, myLastName, myFirstName,
 myMiddleInitial, myDeptCode, and mySalary are initialized by
 the SalariedEmployee constructor; myCommission is initialized to
 commission (default 0).

virtual void display(ostream & out) const;
 /*-----
 Output function member.

 Precondition: ostream out is open.
 Postcondition: A text representation of this Salesperson object
 has been output to out.

// ... Other administrator operations ...

private:
 ***** Data Members *****/
 double myCommission;
};

//-- Definition of constructor
inline double Salesperson::Salesperson(
 long id, string last, string first, char initial,
 int dept, double salary, double commission)
: SalariedEmployee (id, last, first, initial, dept, salary),
 myCommission(commission)
{ }
```

```
//--- Definition of Salesperson's display()
inline void Salesperson::display(ostream & out) const
{
 SalariedEmployee::display(out); //inherited members
 out << "Commission: $" //local members
 << myCommission << endl;
}
#endif
```

4.

```
/* Document.h -----
 Header file for an abstract class Document,
 Abstract operations that must be provided by a derived class:
 read(), display()
 Other operations provided:
 <<, >>, constructor
-----*/
#include <iostream>

#ifndef DOCUMENT
#define DOCUMENT

class Document
{
public:
 //***** Function Members *****
 virtual void display(ostream & out) const = 0;
 virtual void read(istream & in) = 0;
 //--- Other operations on documents ---
private:
 //***** Data Members ****/
}; // end of abstract-class declaration

// Output operator
inline ostream & operator<<(ostream & out, const Document & doc)
{
 doc.display(out);
 return out;
}
// Input operator
inline istream & operator>>(istream & in, Document & doc)
{ doc.read(in); return in; }

#endif
```

Then the class License (and others such as Certificate and Report) can be derived from Document. Each must provide definitions of the purely virtual methods read() and display() if the input and output operators are to work correctly. For example:

```

class License : public Document
{
public:
 //***** Function Members *****/
 virtual void display(ostream & out) const;
 //--- Other operations on licenses ---
private:
 //***** Data Members *****/
 long myIdNumber;
 string myLastName,
 myFirstName;
 char myMiddleInitial;
 int myAge;
 Date myBirthDay; // where Date is a user-defined class
 // ... and perhaps other attributes ...
}; // end of class declaration

// Definition of display
void License::display(ostream & out) const
{
 out << myIdNum << " " << myFirstName << " "
 << myMiddleInitial << ". " << myLastName
 << "\nAge: " << myAge << " Birthdate: "
 << myBirthDate; // assumes << defined for Date
}
void License::read(istream & in)
{
 in >> myIdNum >> myFirstName >> myMiddleInitial
 >> my LastName >> myAge >> myBirthDate;
} // assumes >> defined for Data

```

5.

```

/* LookAheadStack.h -----
 Header file for a class template LookAheadStack derived from Stack
 that adds the attributes unique to look-ahead stacks.
 New operations: A constructor and a revised push operation.
-----*/
#include <iostream>
#include "DStackT.h" // The stack class template version from
 // Chapter 8 that uses a dynamic array.
#ifndef LOOK_AHEAD_STACK
#define LOOK_AHEAD_STACK

template <typename ElementType>
class LookAheadStack : public Stack<ElementType>
{
public:
 //***** Function Members *****/

```

```
LookAheadStack();
/*-----
 Construct a LookAheadStack object.

 Preconditions: None.
 Postconditions: An empty look-ahead stack has been constructed.
-----*/
void push(ElementType value);
/*-----
 Add value at top of look-ahead stack (if there is room).

 Preconditions: value is different from top stack element.
 Postconditions: value was added at the top of this LookAheadStack
 provided it is different from top stack element and there is
 space; otherwise, a duplicate-element message is displayed
 in the first case and execution allowed to proceed, and in
 the second case, a stack-full message is displayed and
 execution is terminated.
-----*/
private:
 ***** Data Members *****
}; // end of class template declaration

//--- Definition of Constructor ---
template <typename ElementType>
inline LookAheadStack<ElementType>::LookAheadStack()
: Stack<ElementType>()
{ }

//--- Definition of push() ---
template <typename ElementType>
inline void LookAheadStack<ElementType>::push(const ElementType & value)
{
 if (myArray[myTop] != value)
 Stack<ElementType>::push(value);
 else
 cerr << "Item to be added matches top stack element -- not added\n";
}
#endif
```

**Chapter 15: Trees****Exercises 15.1**

1. Morse is not immediately decodable because its codes do not have unique prefixes.

For example, the given bit string 100001100 can be decoded as:

10 000 110 0  $\Rightarrow$  N S GE

and also as

100 00 11 00  $\Rightarrow$  D I M I

and as several other strings.

2. BAD

3. DEAD

4. CAD

5. BEADED

Note that for #6 and #7, several different codes are possible, depending on whether the nodes are placed in the left or the right subtrees.

6. int 10  
main 11  
while 000  
if 01  
for 001

7. a 111  
b 100  
c 1100  
d 1101  
e 0  
f 1010  
g 10110  
h 10111

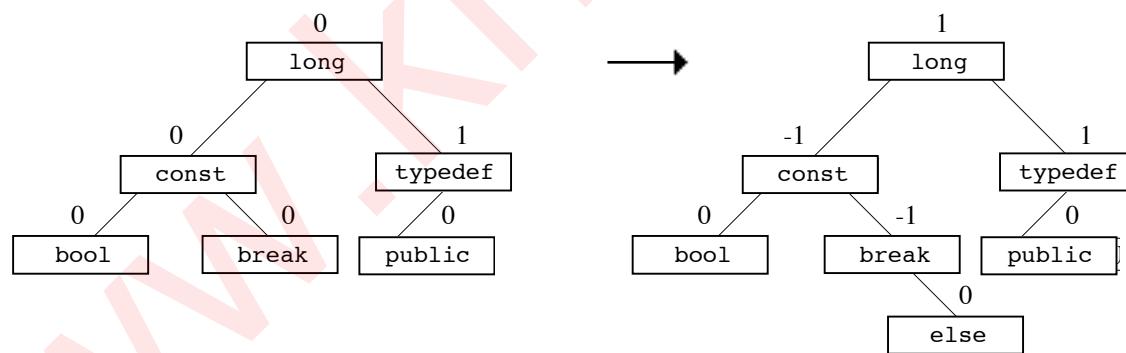
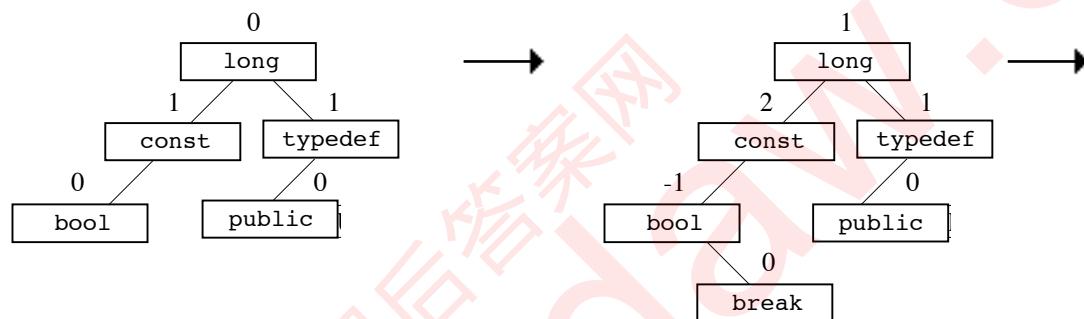
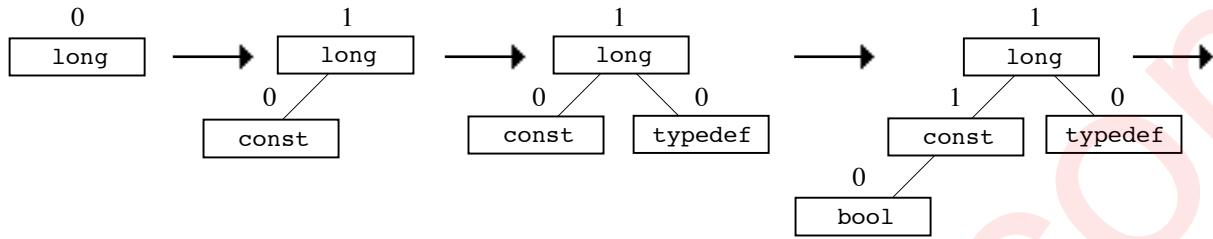
8. 1010 0 0 1101 111 1101 0 111 1010 111 10110 0 1101 10111 111 10110  
F E E D A D E A F A G E D H A G

9. One possible solution is the following:

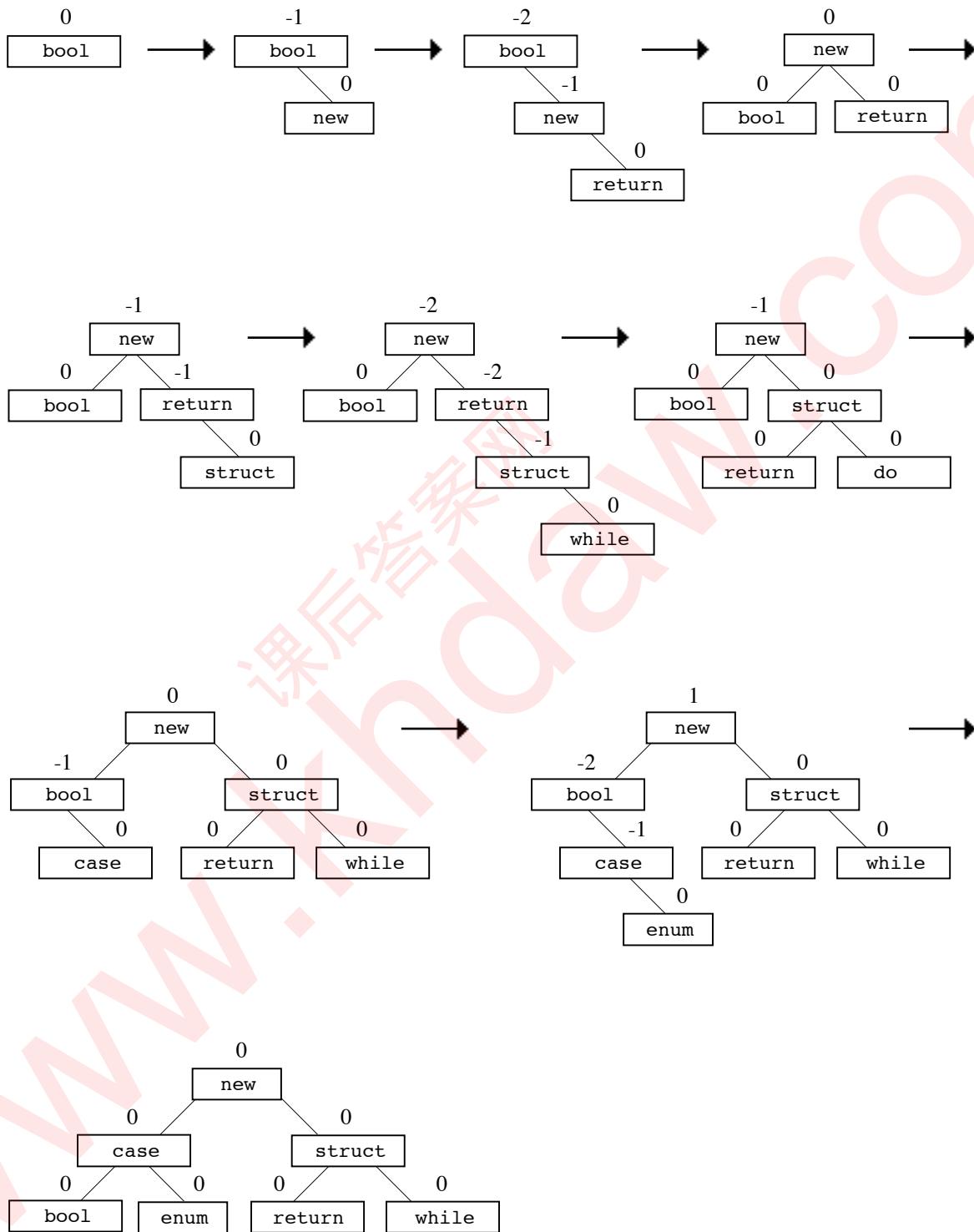
|        |        |
|--------|--------|
| case   | 000011 |
| class  | 011    |
| do     | 01001  |
| else   | 0101   |
| false  | 00010  |
| for    | 100    |
| goto   | 000010 |
| if     | 101    |
| int    | 110    |
| main   | 111    |
| static | 010000 |
| struct | 00000  |
| switch | 010001 |
| true   | 00011  |
| while  | 001    |

**Exercises 15.2**

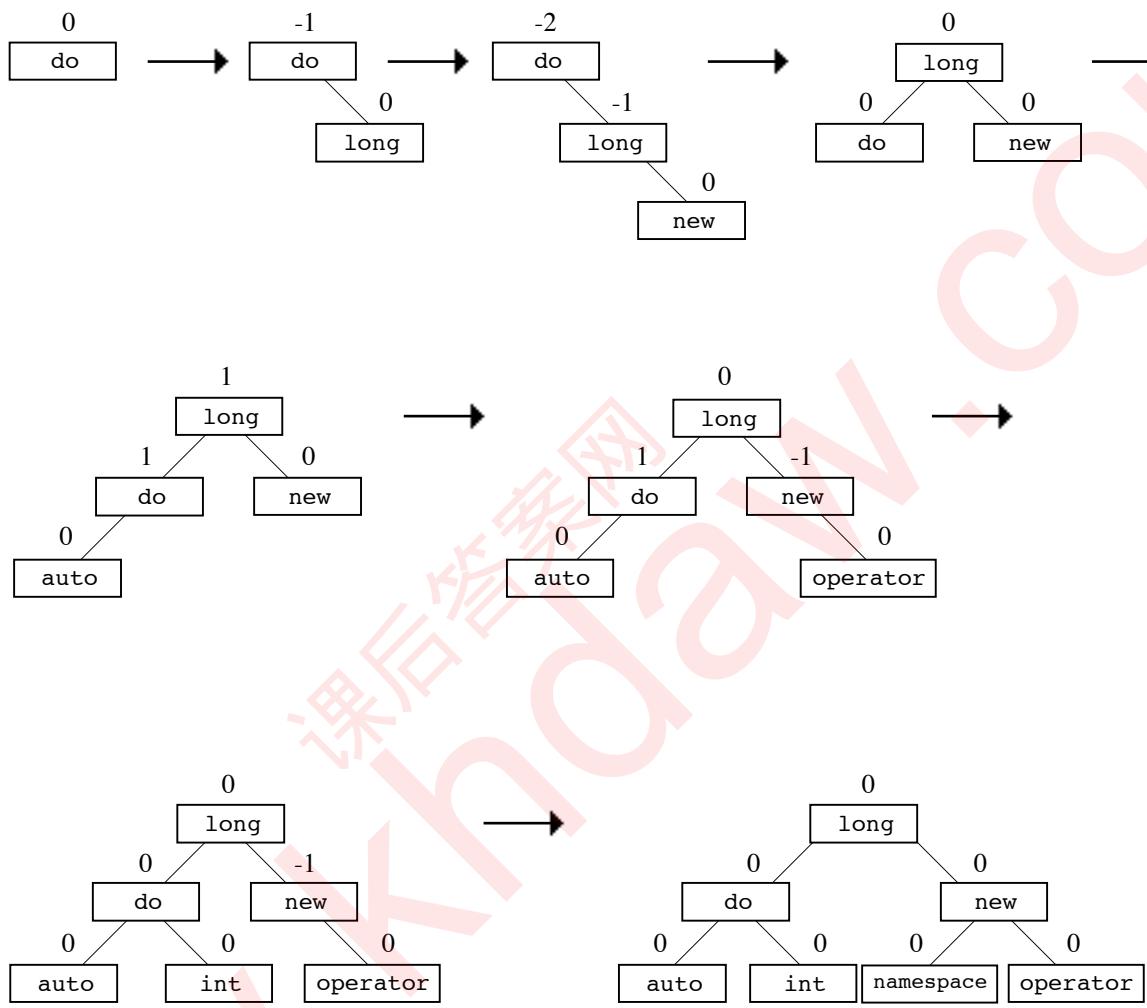
1.



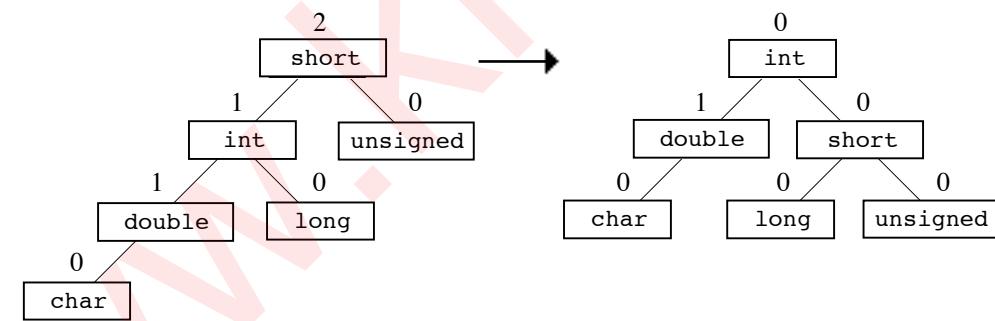
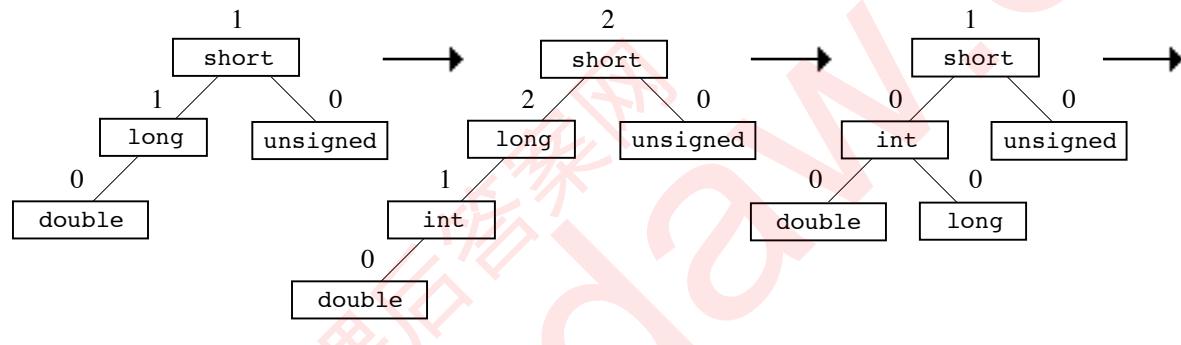
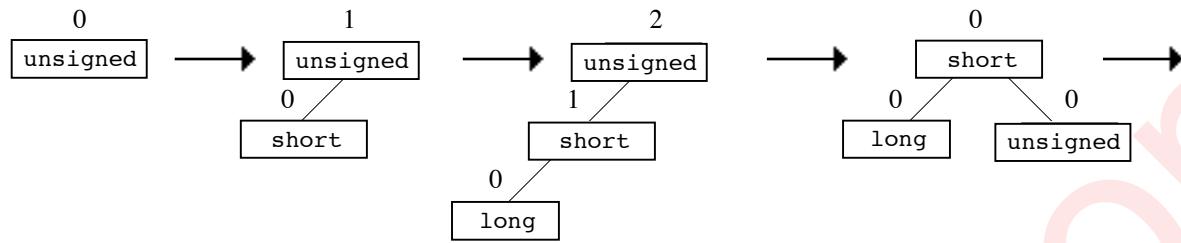
2.



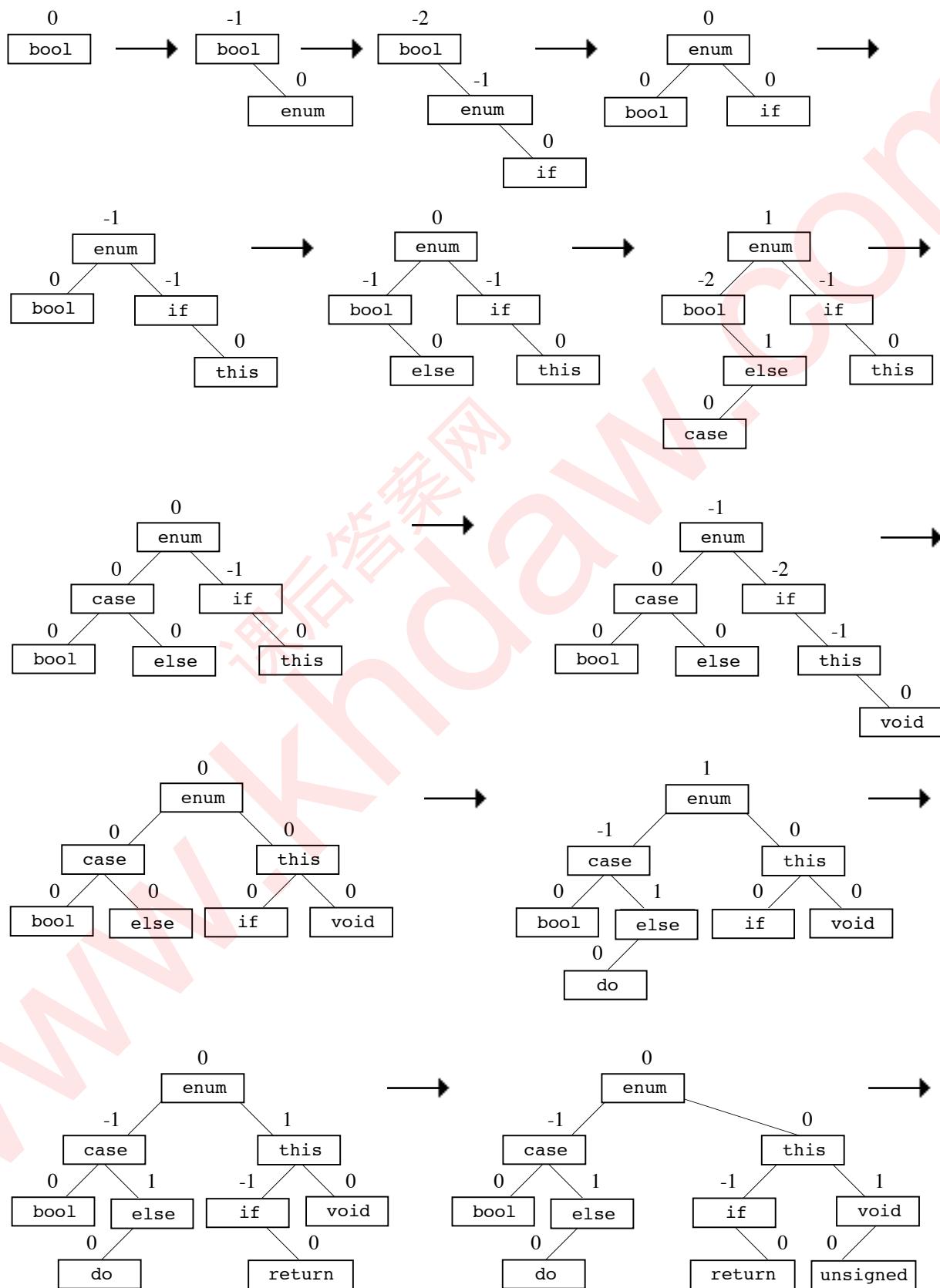
3.

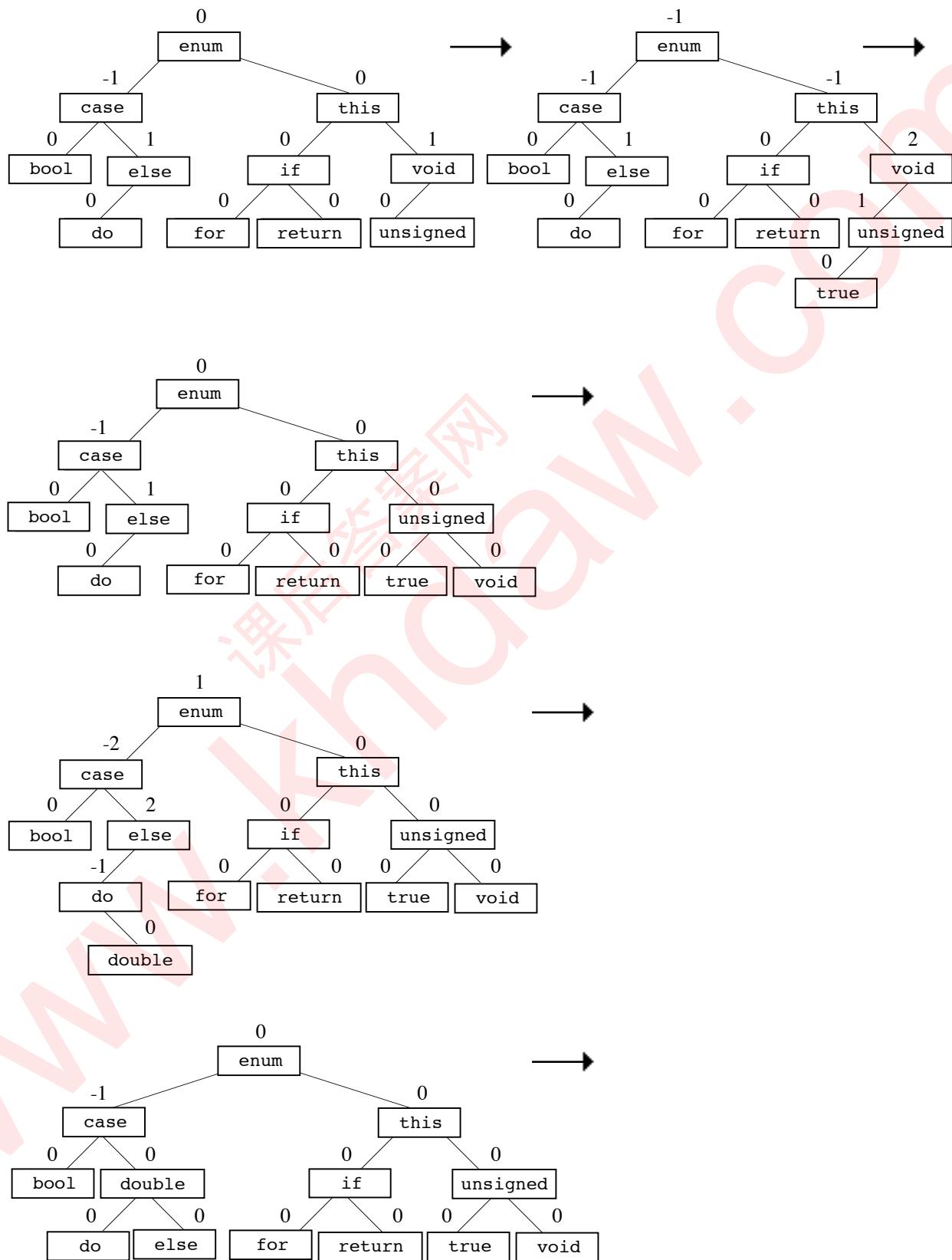


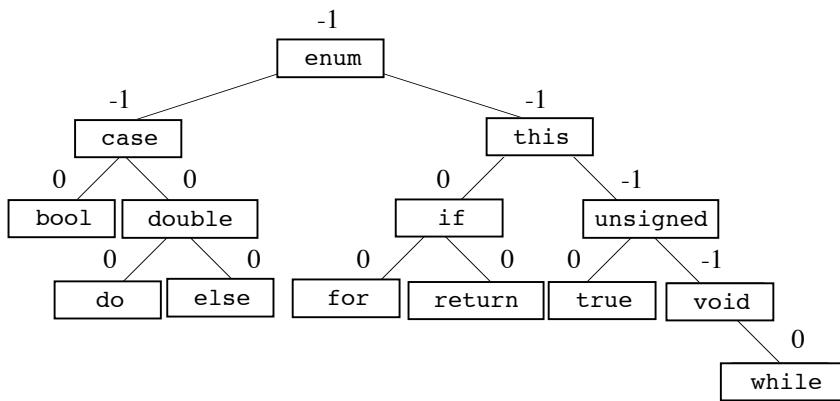
4.



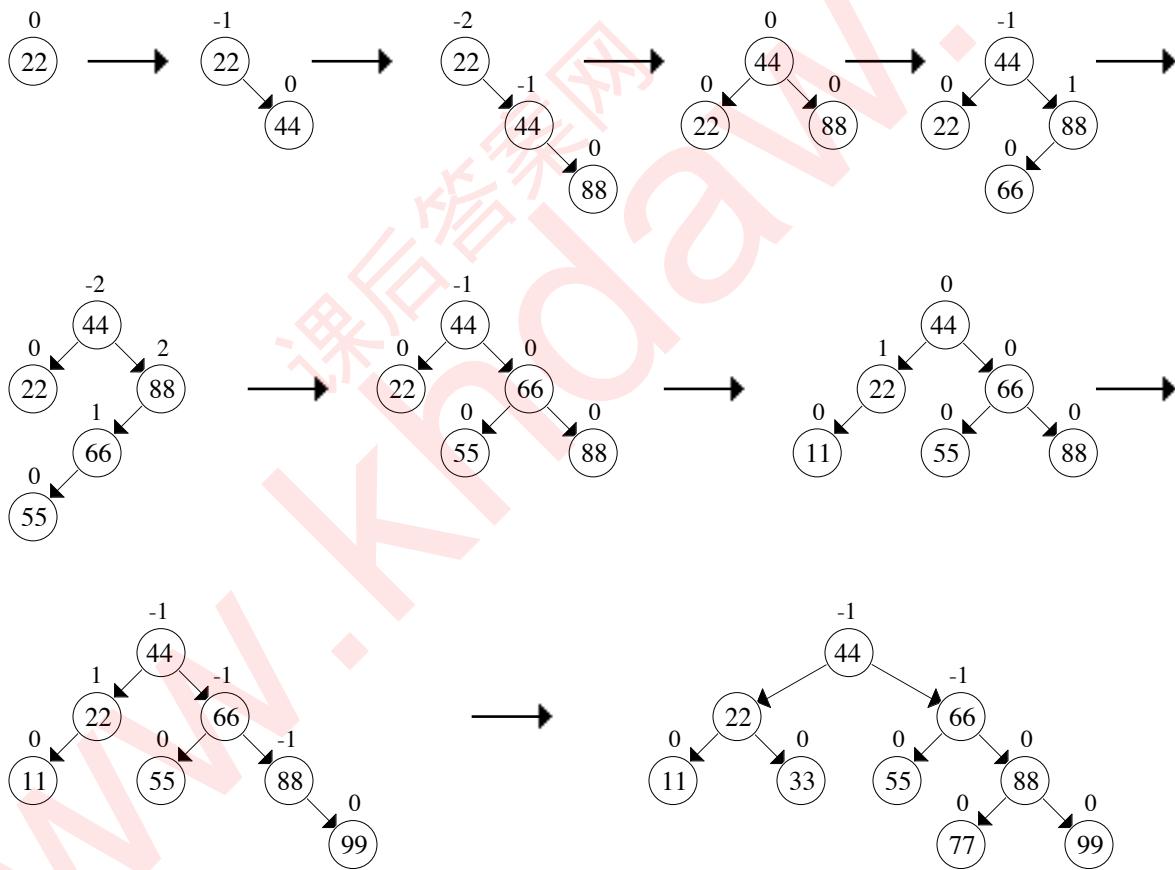
5.



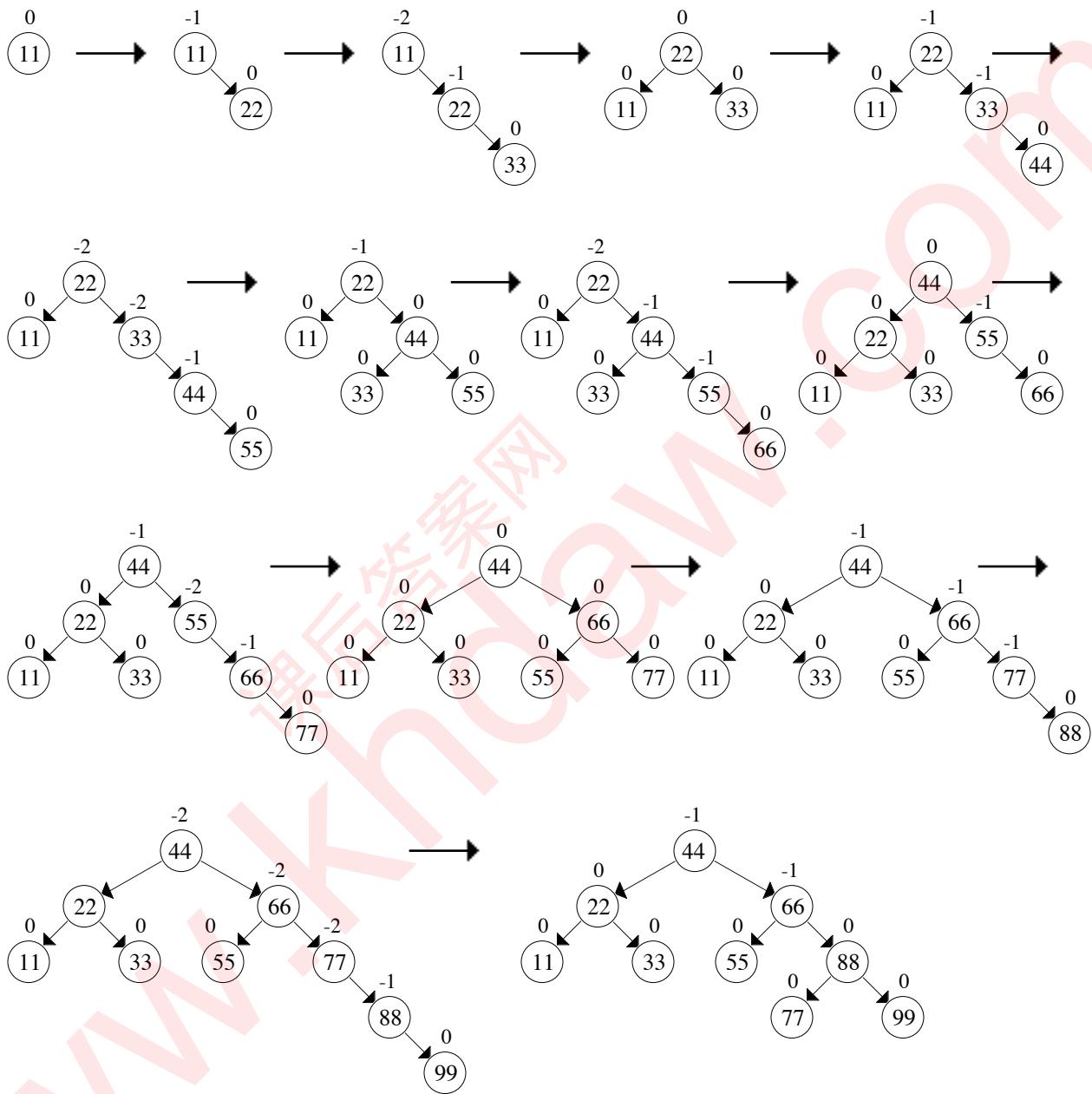




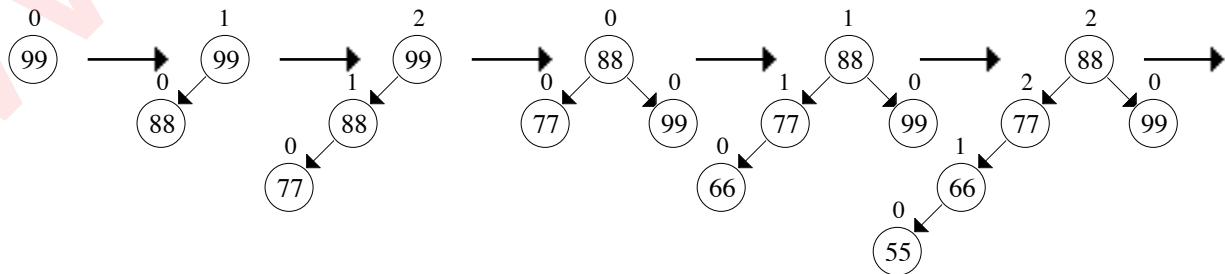
6.

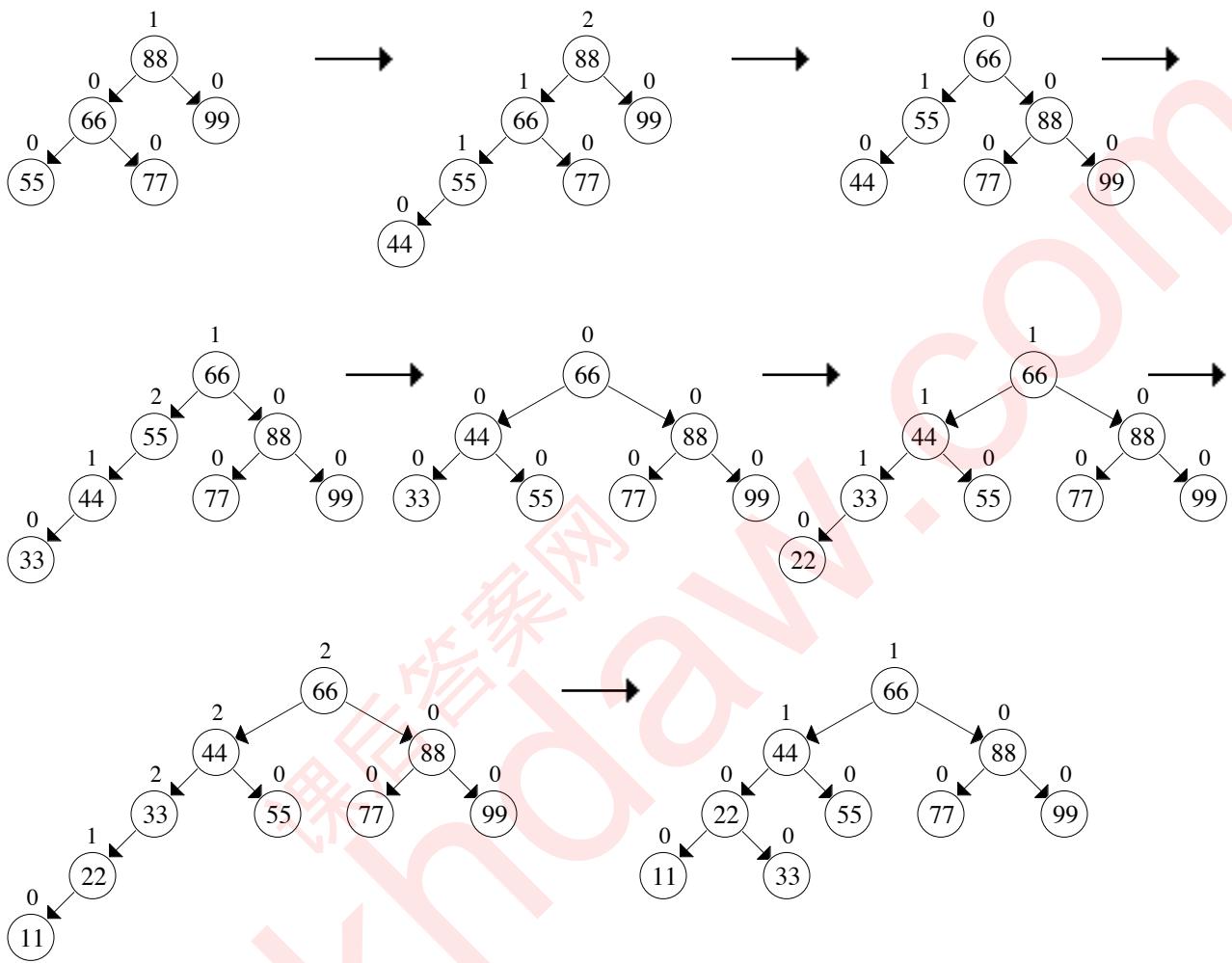


7.

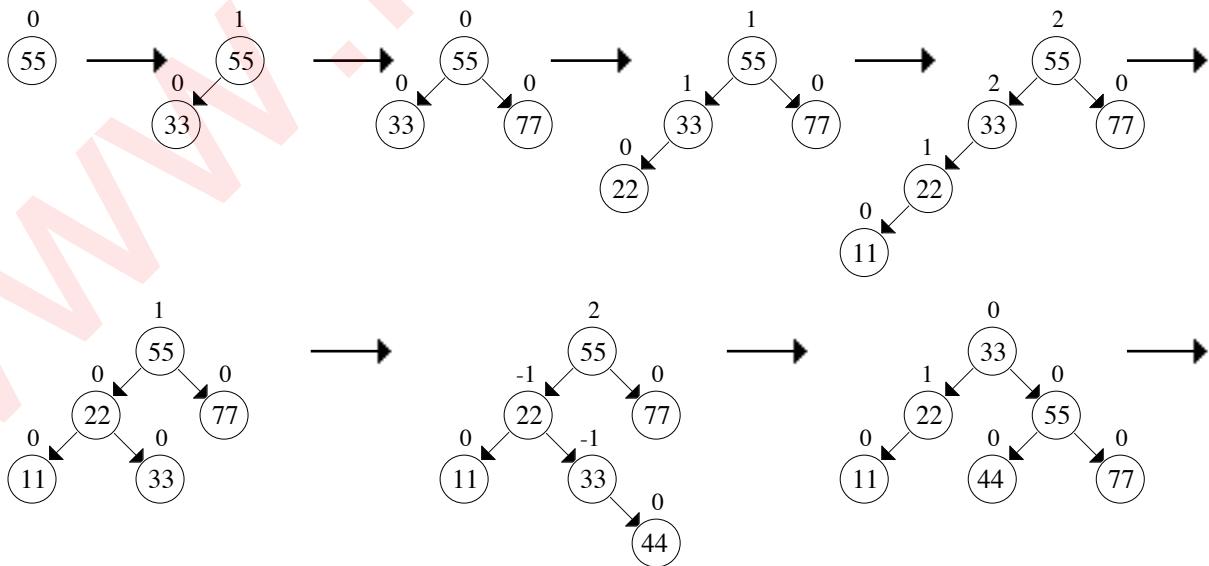


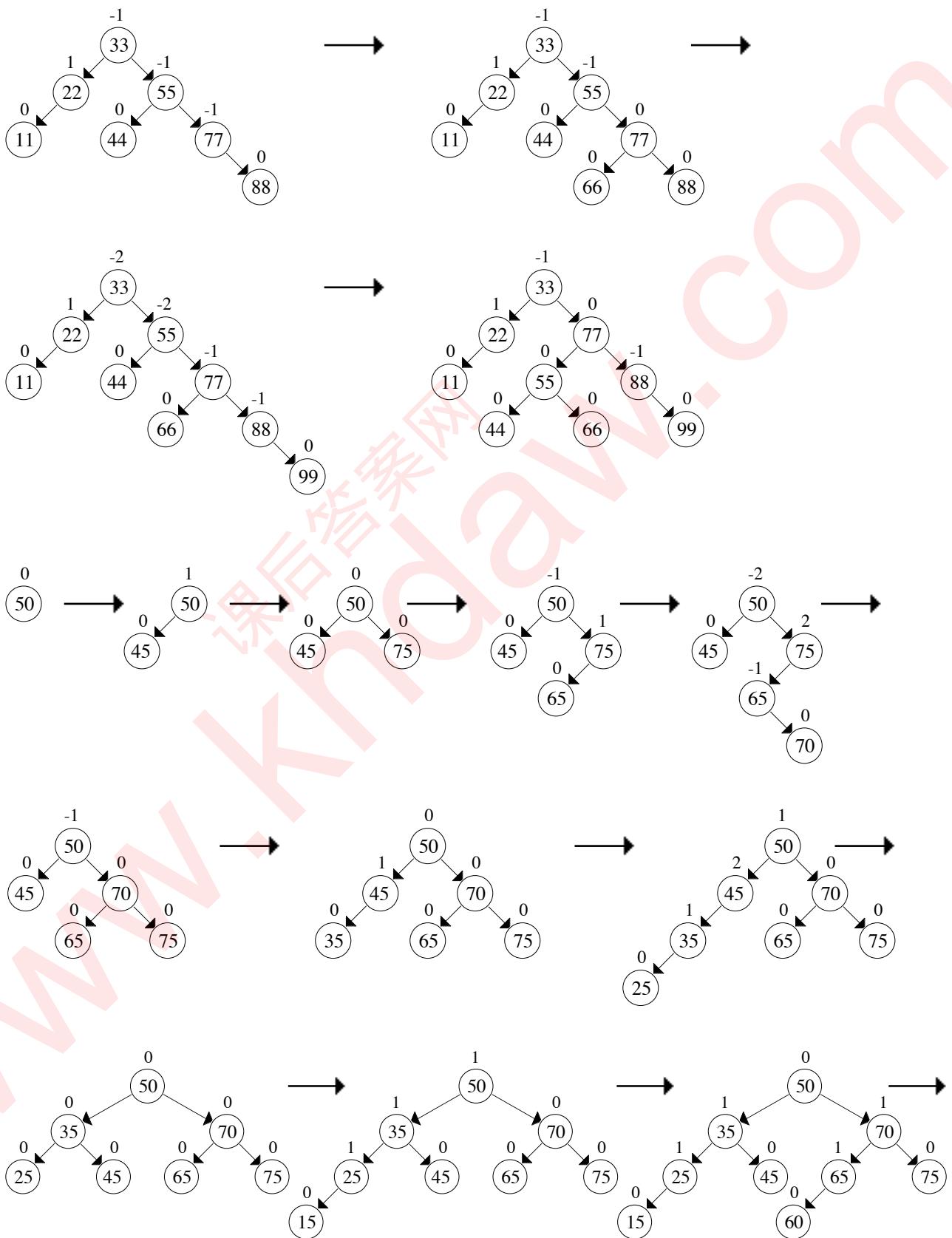
8.

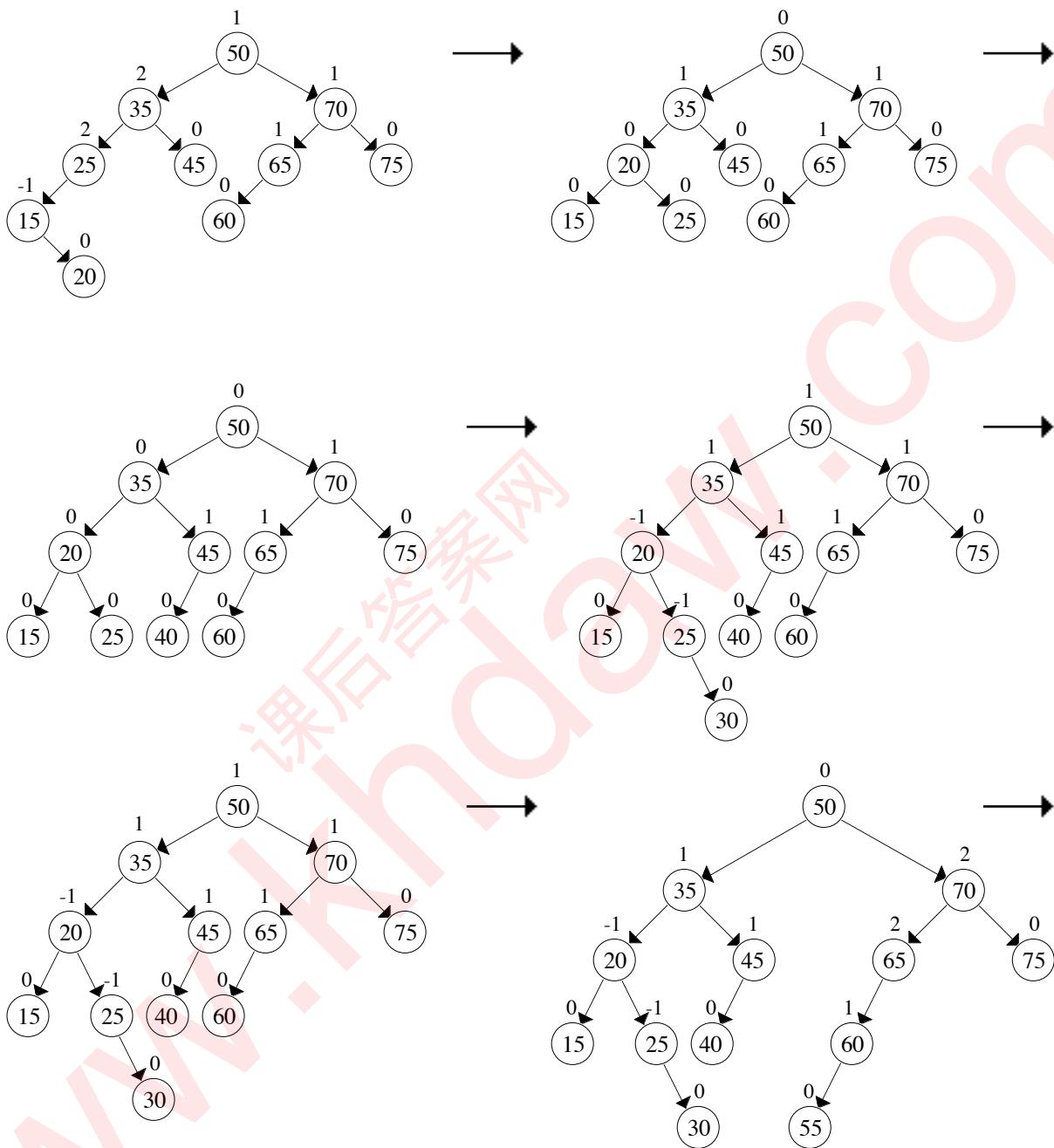


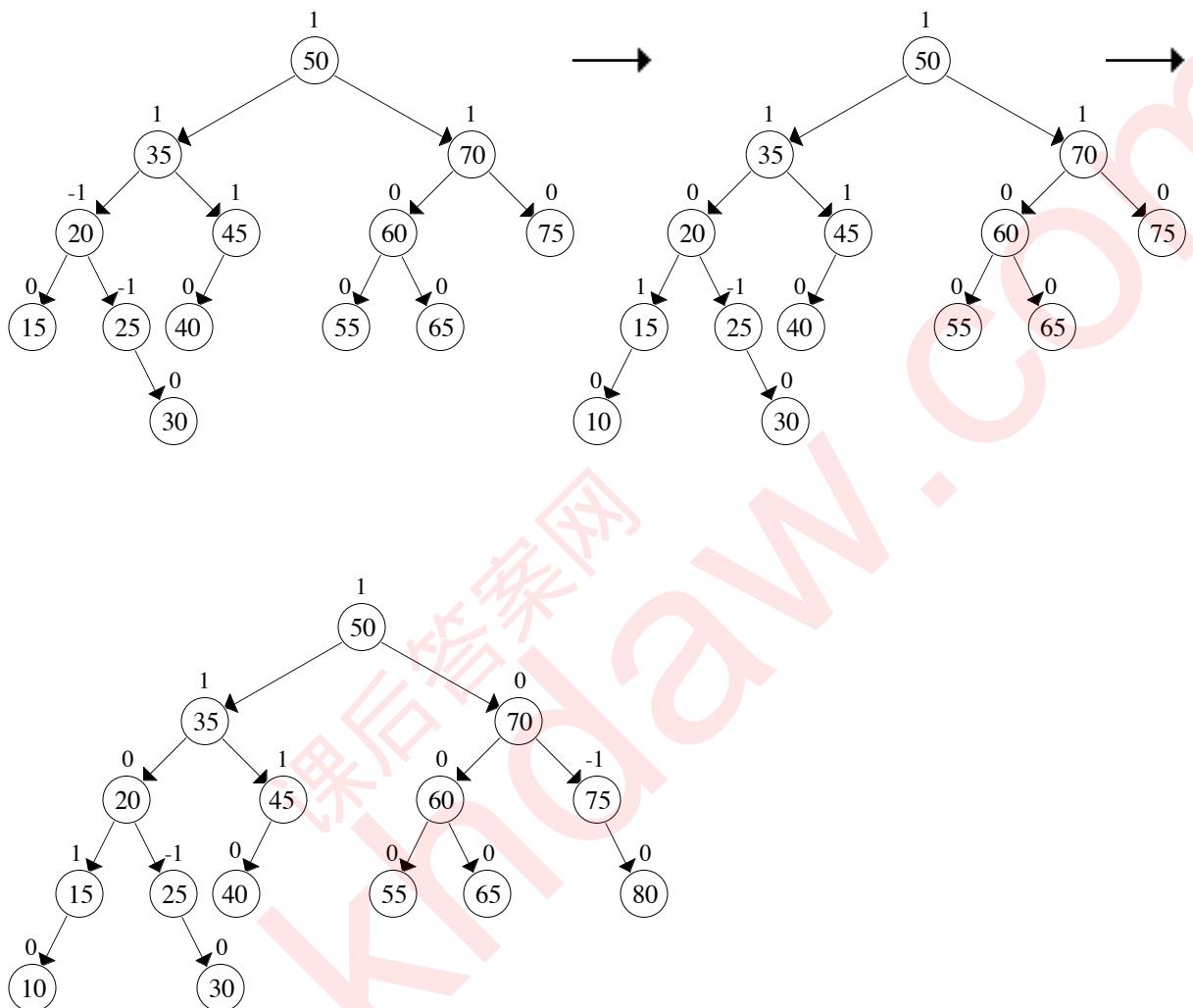


9.



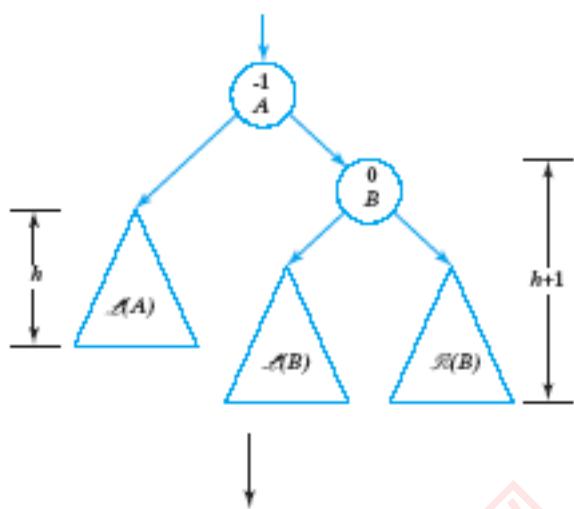




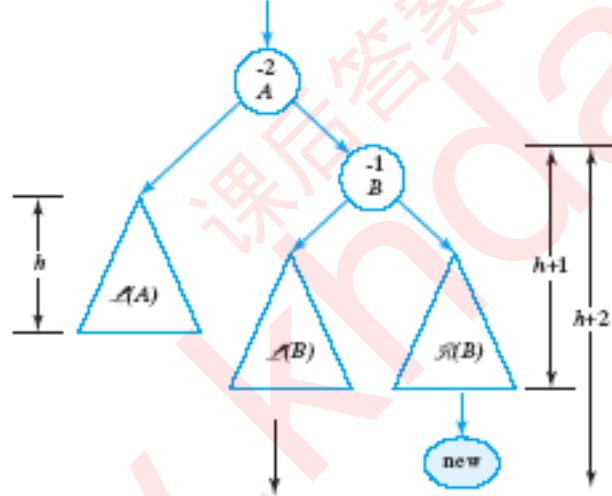


11.

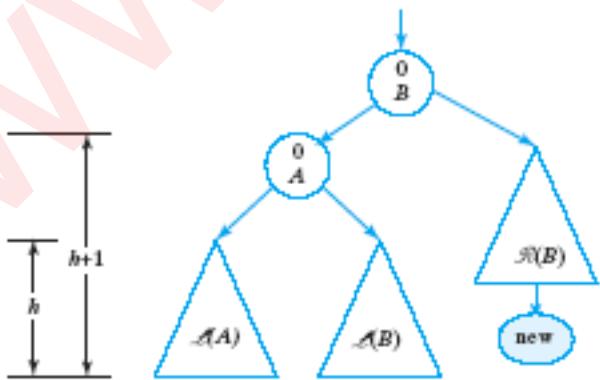
Balanced Subtree Before Insertion



Unbalanced Subtree After Insertion

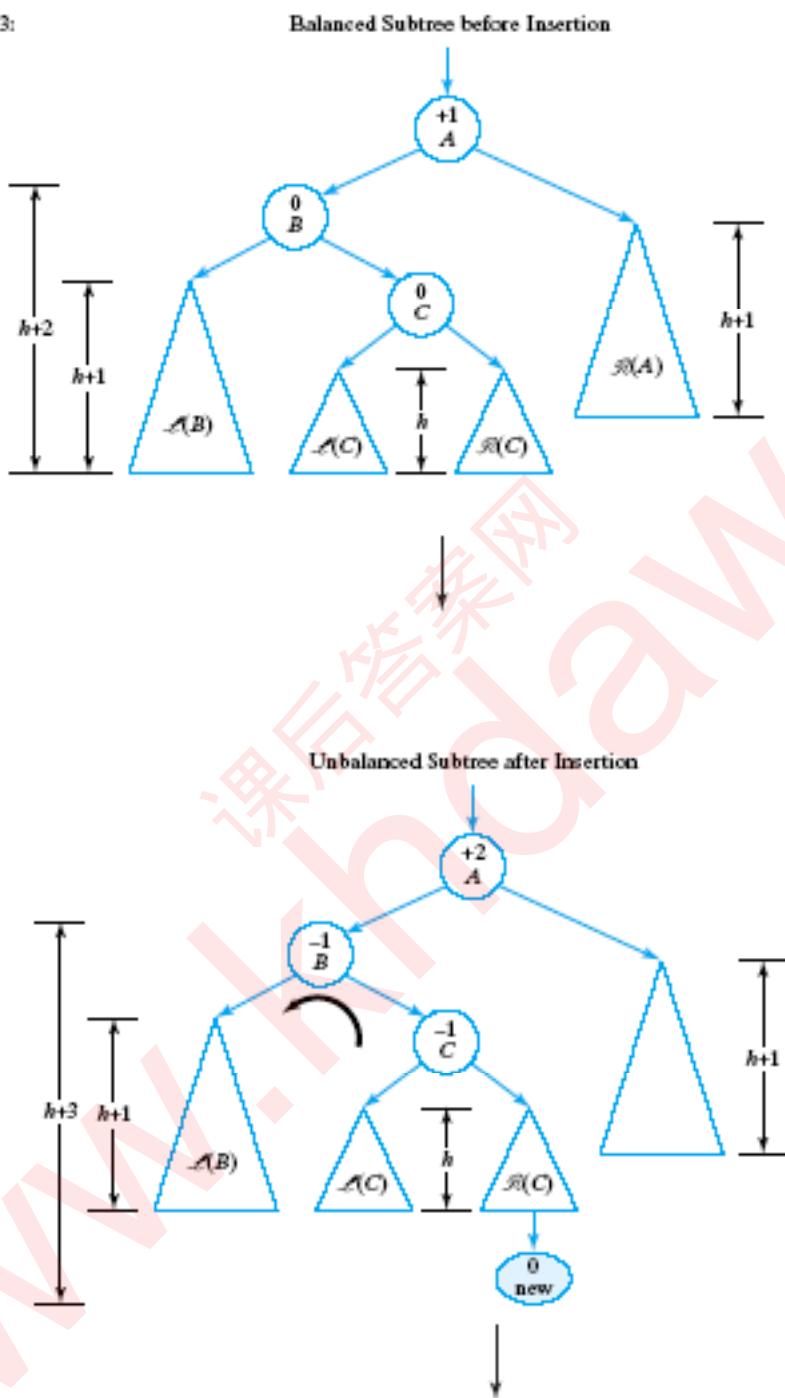


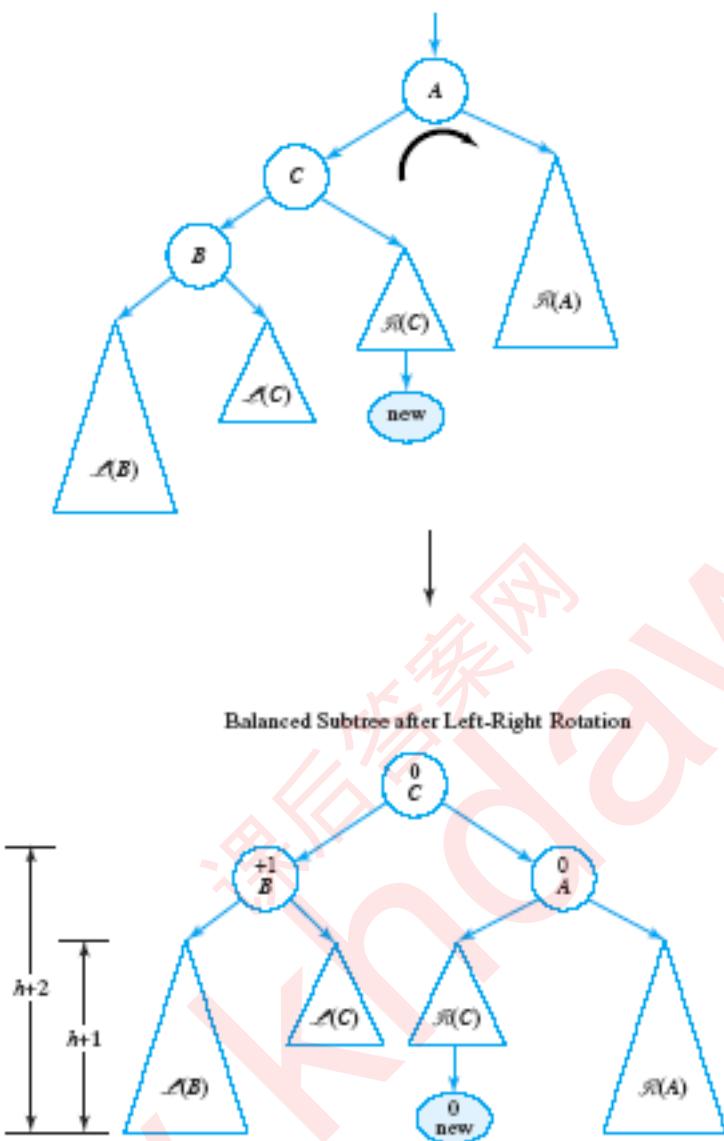
Rebalanced Subtree After Simple Left Rotation



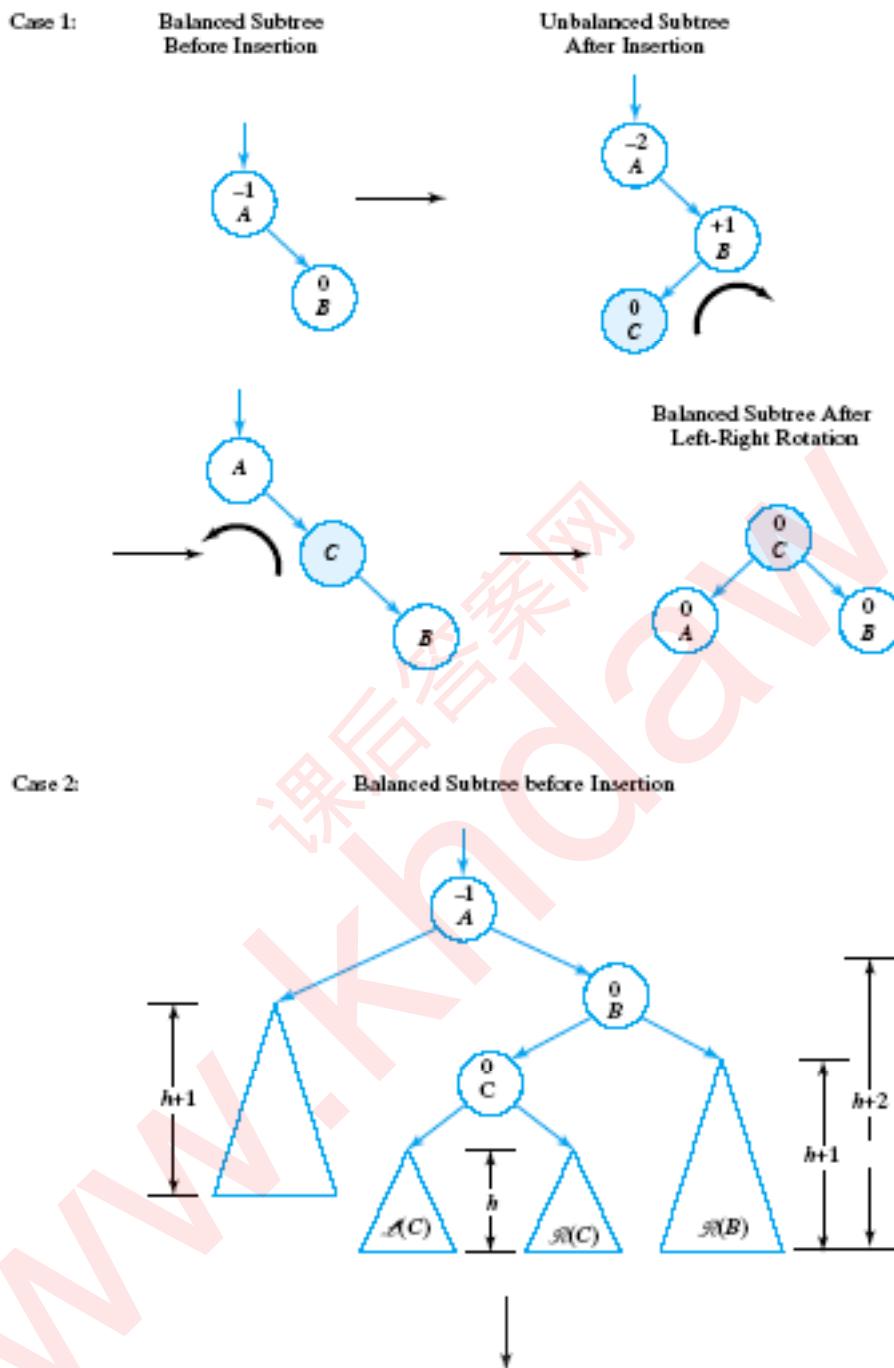
12.

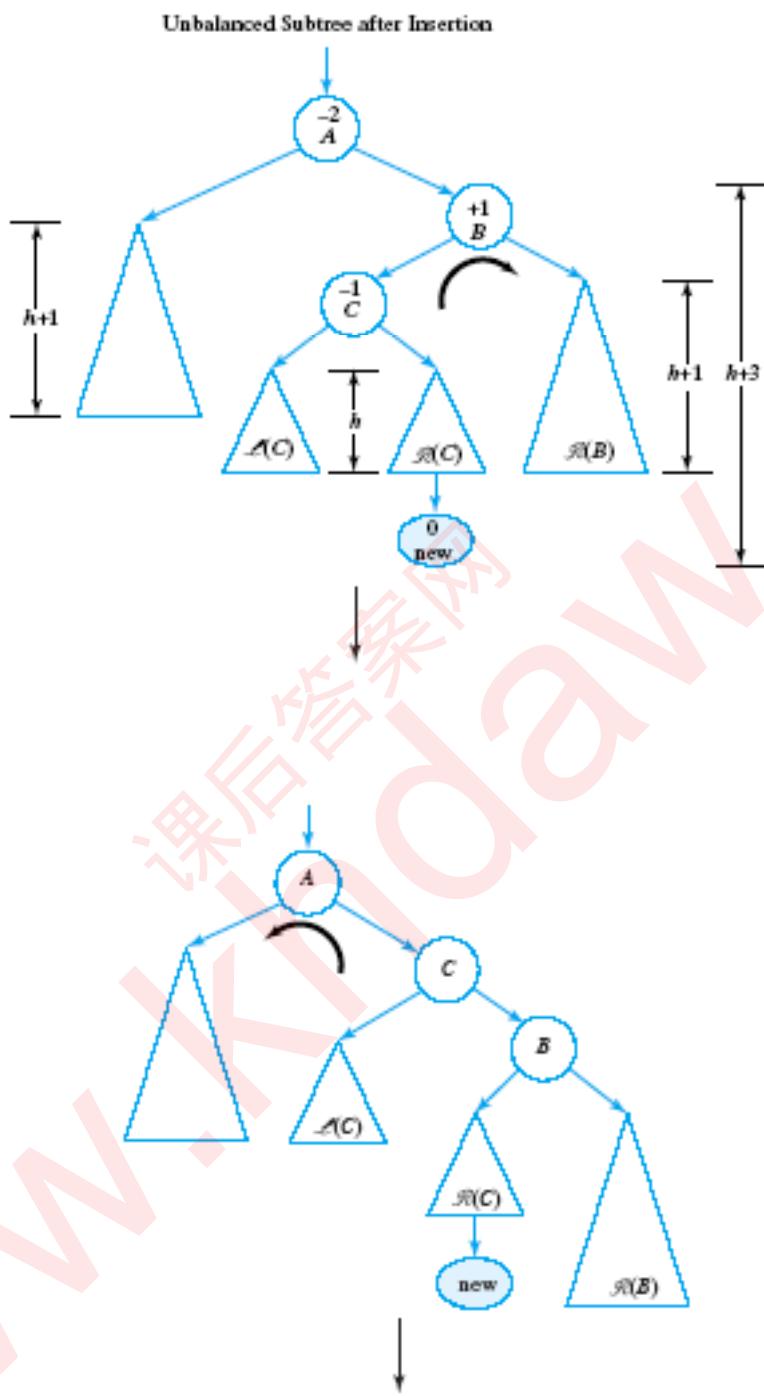
Case 3:



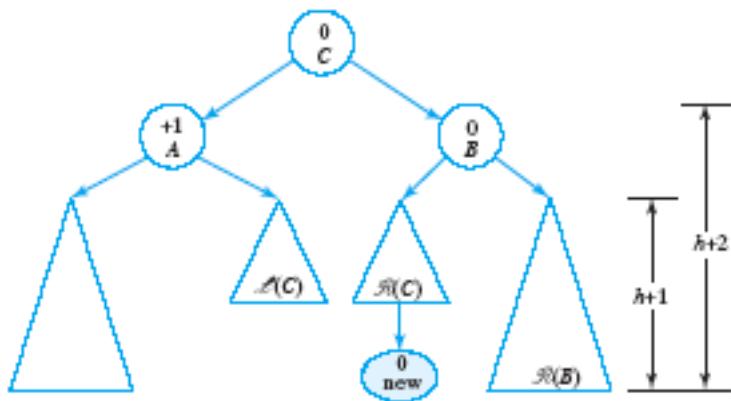


13.



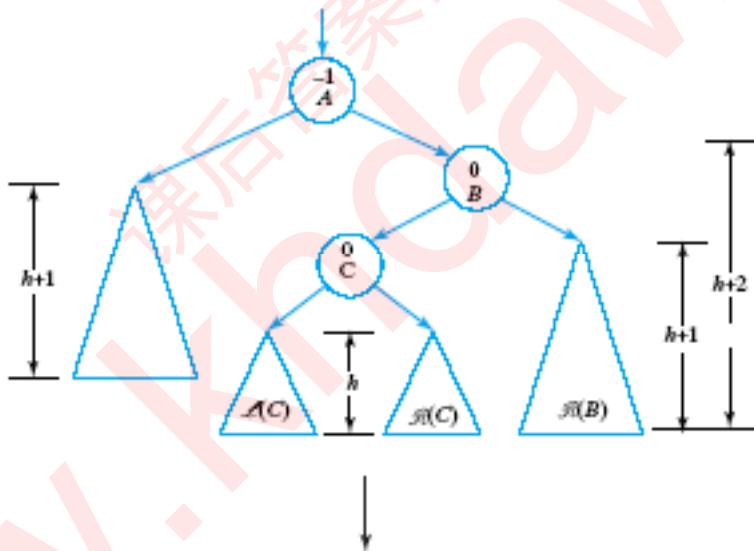


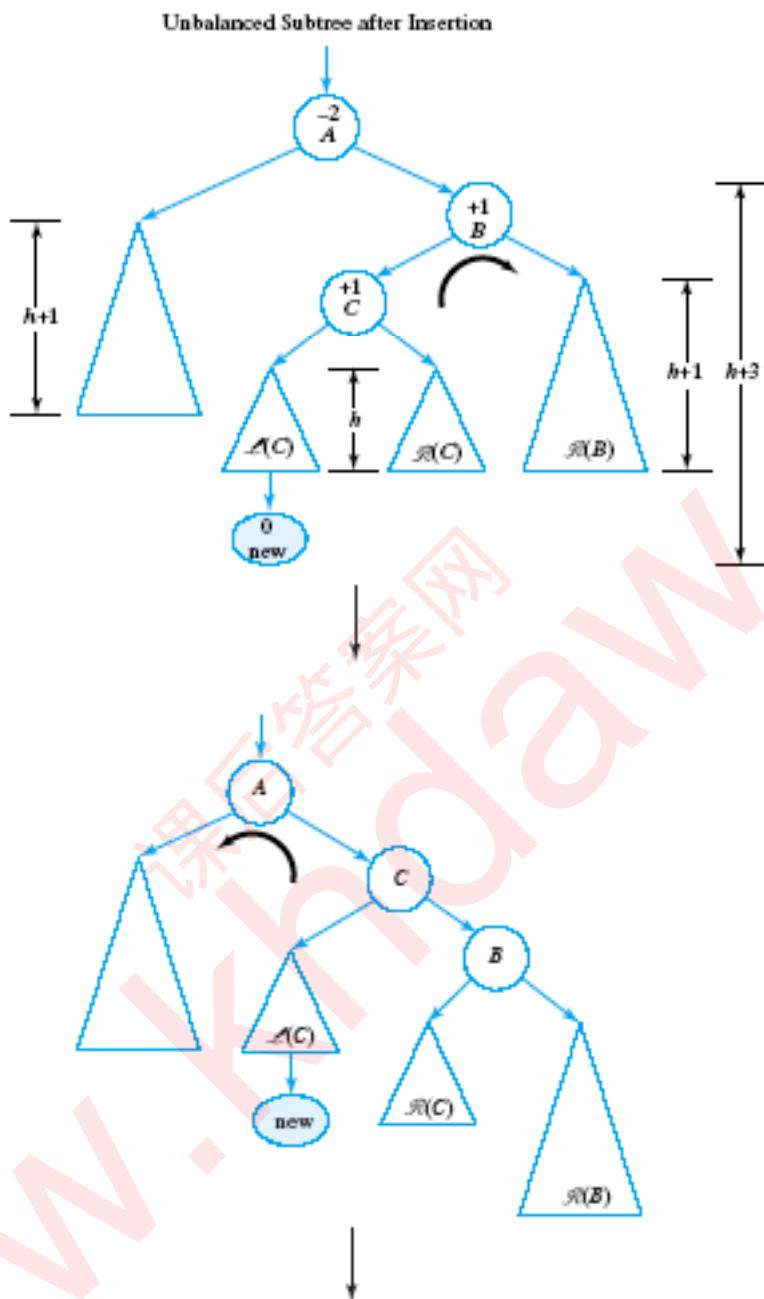
Balanced Subtree after Right-Left Rotation

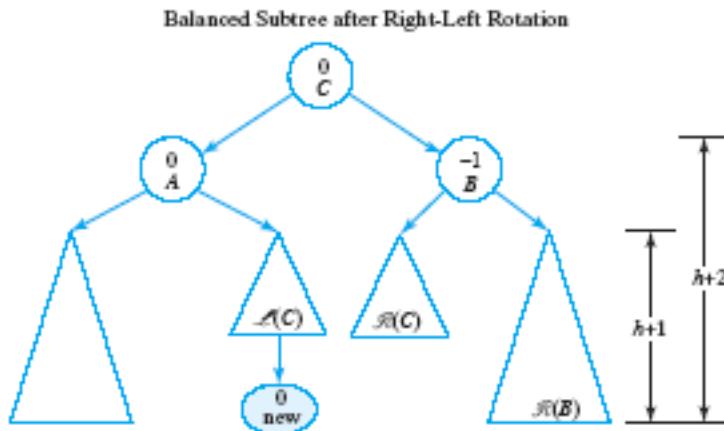


Case 3c

Balanced Subtree before Insertion







14.

```

template <typename ElementType>
class AVLTree
{
public:
 /*** FUNCTION MEMBERS
 // Usual BST operations

private:
 /*** Node ***/

 class AVLNode
 {
 short int balanceFactor;
 ElementType data
 AVLNode * left;
 AVLNode * right;
 }

 //--- AVLNode constructors
 AVLNode()
 {
 balanceFactor = 0;
 left = right = 0;
 }

 AVLNode(ElementType item)
 {
 balanceFactor = 0;
 data = item;
 left = right = 0;
 }
}

```

```

typedef AVLNode * AVLNodePointer;

/** DATA MEMBERS */
AVLNodePointer myRoot;
};

```

The insert operation requires extensive modification from that for BSTs in general. Here is an algorithm:

```

/* Algorithm to insert an item item into an AVL tree */

1. If the BST is empty
 Create a node containing item with both left and right links null and balance factor = 0,
 and return from this algorithm.

// Search for insertion point; ptrA keeps track of the most recent ancestor with balance factor
// ±1 and ptrAParent points to the parent of ptrA. Pointer parent follows locPtr along the
// search path.
2. Set pointers ptrAParent and parent to null, ptrA and locPtr to the root of the BST,
3. While (locPtr != 0) do the following: // Search for insertion point for item
 If (locPtr->balanceFactor != 0)
 a. Set ptrA = locPtr and ptrAParent = parent.
 b. If (item < locPtr->data) // go left
 Set parent = locPtr and locPtr = locPtr->left.
 Else // go right
 Set parent = locPtr and locPtr = locPtr->right.
 Else // item is in the tree
 Display message that item is in the tree and terminate the algorithm.
 End if.
 End if.
 End while.

// item is not in the tree. Insert it as a child of parent.
4. Get a node pointed to by newPtr and containing item in its data part, both left and right
links
set to null pointers, and balance factor = 0.
5. If (item < parent->data) // insert as left child
 Set parent->left = newPtr.
Else // insert as right child
 Set parent->right = newPtr.
End if.

// Rebalance the tree by adjusting balance factors of nodes on the path from ptrA to parent;
// all of these nodes have balance factor 0, which will change to ±1. direction is +1 or
// according as item is inserted in the left subtree or the right subtree of ptrA.

```

6. If (*item* < *ptrA*->*data*)  
    Set *locPtr* = *ptrA*->left, *ptrB* = *p*, and *direction* = +1,  
    Else  
        Set *locPtr* = *ptrA*->right, *ptrB* = *p*, and *direction* = -1,  
    End if.
  7. While (*locPtr* != *newPtr*) do the following:  
    If (*item* < *locPtr*->*data*) // height of left subtree increases by 1  
        Set *locPtr*<balanceFactor = +1 and *locPtr* = *locPtr*->left.  
    Else // height of right subtree increases by 1  
        Set *locPtr*<balanceFactor = -1 and *locPtr* = *locPtr*->right.  
    End if.  
End while.
- // Check if tree is unbalanced
8. If (*ptrA*->balanceFactor == 0) // tree is still balanced  
    Set *ptrA*->balanceFactor = *d* and terminate the algorithm.
  9. If (*ptrA*->balanceFactor + *d* == 0) // tree is still balanced  
    Set *ptrA*->balanceFactor = 0 and terminate the algorithm.
- // Tree is unbalanced; perform suitable rotation
10. If (*direction* = +1) // left imbalance  
    If (*ptrB*->balanceFactor = +1) // LL rotation  
        a. Set *ptrA*->left = *ptrB*->right.  
        b. Set *ptrB*->right = *ptrA*.  
        c. *ptrA*->balanceFactor = *ptrB*->balanceFactor = 0.  
    Else // LR rotation  
        a. Set *ptrC* = *ptrB*->right, *cptrLeft* = *c*->left, *cptrRight* = *c*->right.  
        b. Set *ptrB*->right = *cptrLeft*.  
        c. Set *ptrA*->left = *cptrRight*,  
        d. Set *ptrC*->left = *ptrB*.  
        e. Set *ptrC*->right = *ptrA*.  
        f. If (*ptrC*->balanceFactor == +1) // LR - case 2  
            Set *ptrA*->balanceFactor = -1, *ptrB*->balanceFactor = 0.  
            Else if (*ptrC*->balanceFactor == -1) // LR - case 3  
                Set *ptrB*->balanceFactor = +1, *ptrA*->balanceFactor = 0.  
            Else // LR - case 1  
                i. Set *ptrC*->balanceFactor = 0.  
                ii. Set *ptrB* = *ptrC*,  
            End if.  
    End if.
  - End if.
- Else // --- Right imbalance -- This is symmetric to the above for left imbalance.  
// ---
- End if.

```
// Subtree with root ptrB has been rebalanced and is the new subtree of ptrAParent (for which
// the original
// root was ptrA).
```

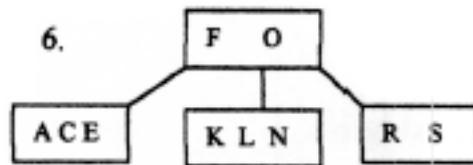
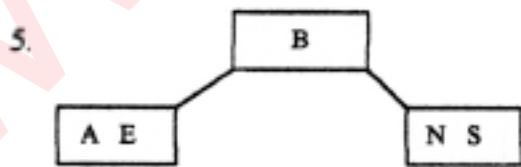
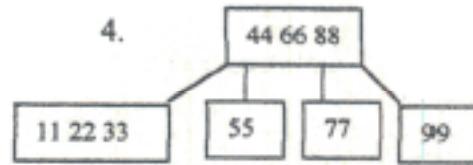
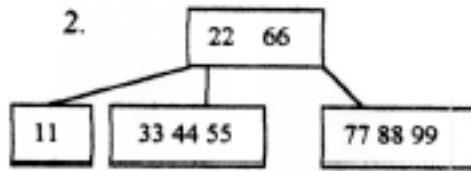
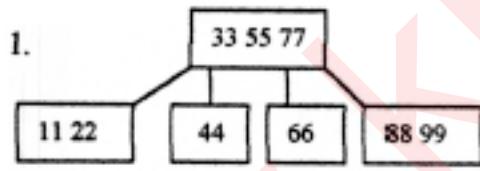
11. If (*ptrAParent* == 0)  
 Set BST's root equal to *ptrB*.  
 Else if (*ptrA* == *ptrAParent*->left)  
 Set *ptrAParent*->left == *ptrB*.  
 Else // *ptrA* == *ptrAParent*->right  
 Set *ptrAParent*->right == *ptrB*.  
 End if.

Deletion from an AVL tree is more difficult. If there are not many deletions, one might use *lazy deletion*: instead of deleting the node and modifying rebalancing the tree, we mark the node as deleted. Then, during operations that require searching the tree, we indicate that the item is not in the tree if the desired node is found but marked. This keeps the balance of a tree from being disturbed by deletions.

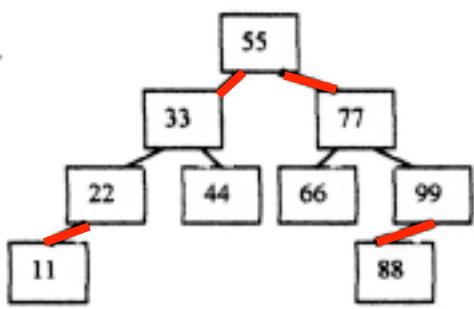
However, if many deletions will be done, then we can delete the node as for a BST and then rebalance the tree. A search of the Internet for AVL deletion will turn up several places where this deletion + rebalancing is described.

See, for example, <http://www.cmcrossroads.com/bradapp/ftp/> for an AVLTree class.

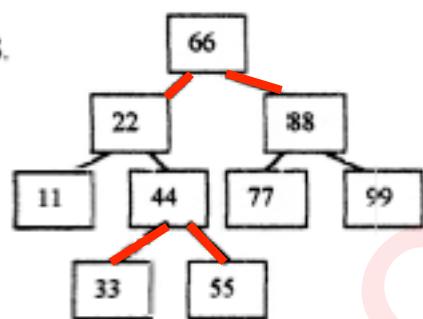
### Exercises 15.3



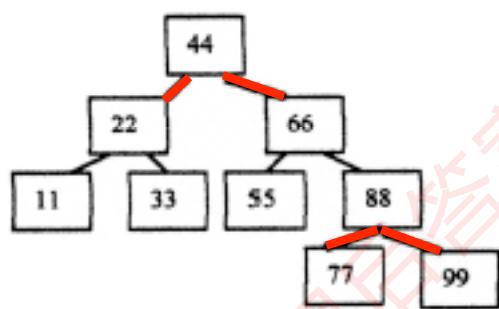
7.



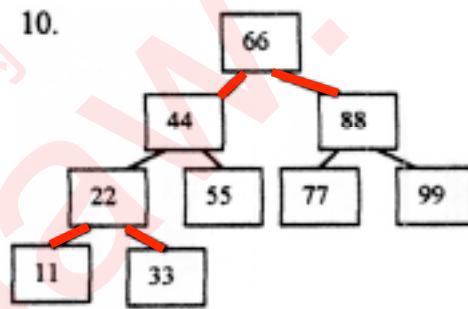
8.



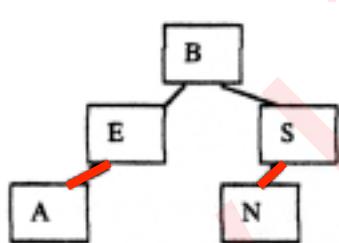
9.



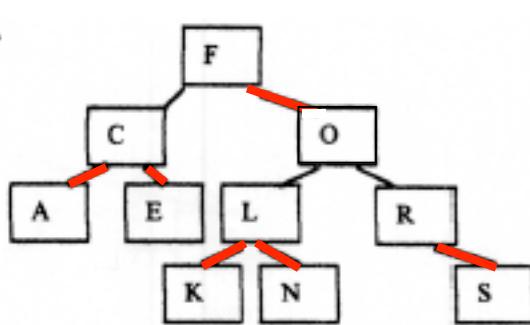
10.



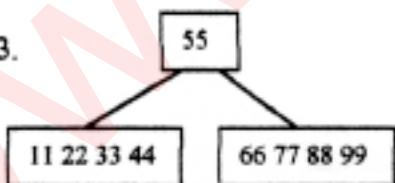
11.



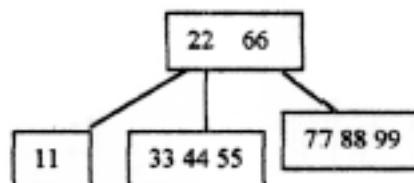
12.



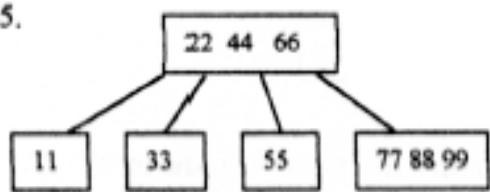
13.



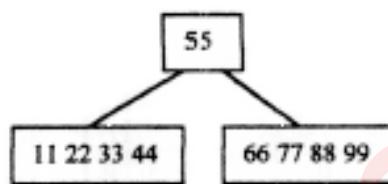
14.



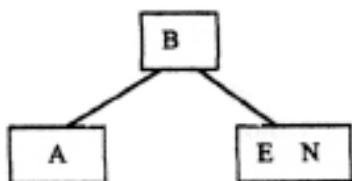
15.



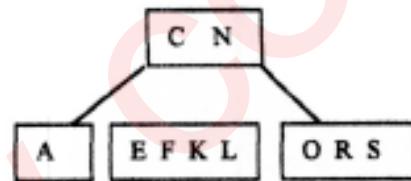
16.



17.



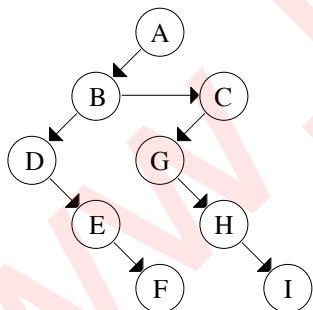
18.



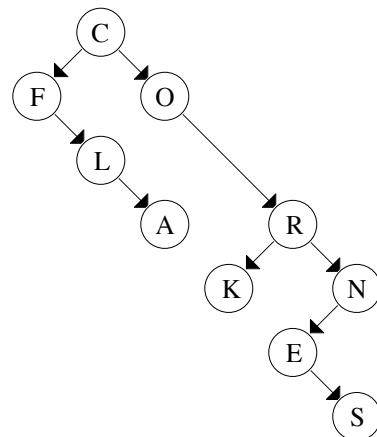
19.



20.



21.



22. An algorithm to delete an element from a 2-3-4 tree must deal with the rebalancing that may be necessary after an element is removed. Recall that insertion sometimes results in a splitting of a node and that split may propagate up or down the tree, depending on whether one is performing a top-down insertion or a bottom-in insertion. In a like manner, deleting an element from a 2-3-4 tree may cause perturbations elsewhere in the tree. There are two main considerations:

- (1) Deletion of an element in a 2-node necessitates the replacement of that value with one from its children.
- (2) Deletion of a 3- or 4-node element may necessitate the merging of children (because in effect, the elements of a 3- or 4-node serve as separator values for their children).

Note that in the latter case, a merge may result in overflow, in which case, we actually need a merge-split operation: merge sibling nodes (causing overflow) and then split the resulting node.

To streamline the deletion process, we reduce the deletion of an arbitrary element to the deletion of an element that is in a leaf node. If the element to be deleted is in a 3-node or 4-node leaf, then after deletion, we have a 2-node or 3-node leaf. No restructuring is then required. To avoid propagation of restructuring, we need to ensure that the element to be deleted is in a 3-node or 4-node leaf before deletion. That is, restructuring is done as we search for the element to be deleted.

The restructuring process rests on the search for the item to be deleted as progressing through only 3-node and 4-nodes. If we move through a 2-node, then restructuring is required. The following steps detail the process. Assume that we are at a node  $p$  and will next move to a child  $q$ .

- If  $p$  is a leaf, the element to be deleted is either in this node or is not in the tree:  
Delete it; no restructuring is needed.
- If  $q$  is a 3-node or 4-node: Move to this node.
- If  $q$  is a 2-node ( $x \& y$ ) and its nearest sibling  $r$  is also a 2-node with children  $x$  and  $y$ :
  - If  $p$  is a 2-node with children  $q$  and  $r$ , merge  $q$ ,  $p$  and  $r$  into a 4-node with children  $x$ ,  $y$ ,  $v$ , and  $w$ .
  - If  $p$  is a 3-node (or 4-node), merge  $q$ ,  $p$ , and  $r$  into a 4-node with children  $x$ ,  $y$ ,  $v$ , and  $w$  (as before), but this 4-node is a child of the 2-node (or 3-node) containing the remaining elements of  $p$ 's original node.
- If  $q$  is a 2-node ( $x \& y$ ) and its nearest sibling  $r$  is also a 3-node (or 4-node): Transform  $q$  into a 3-node by taking the appropriate separator out of  $p$  and one of its children out of  $r$ .

23. An algorithm to delete a node from a red-black tree must deal with the rebalancing that may be necessary after an element is removed. As explored in detail in Exercise 22, restructuring may be confined if one ensures that the node to be deleted is a leaf node, in this case one with a red link from its parent. Again, rebalancing is done as we search for the element to be deleted. As in Exercise 22, the insertion transformations (rotations) are performed in reverse. However, here the cost is cheaper since color changes take care of most transformations.
- 24-25. Developing complete and correct classes for red-black trees and RB trees is a nontrivial task and can be assigned to students who want (or need) a challenge. Source code can be found by searching the Internet; for example, see <http://www.cmcrossroads.com/bradapp/ftp/> .
26. This is an interesting project to challenge students that is not as difficult as those in Exercises 24 and 25. They must decide how to have the user enter the structure of the original tree and store the data in a binary tree as described in the text and then modify the binary tree operations so they perform the desired tree operations.