Programming Principles 1

1.2 THE GAME OF LIFE

Exercises 1.2

Determine by hand calculation what will happen to each of the configurations shown in Figure 1.1 over the course of five generations. [Suggestion: Set up the Life configuration on a checkerboard. Use one color of checkers for living cells in the current generation and a second color to mark those that will be born or die in the next generation.]

Answer

TOC
Index
Help

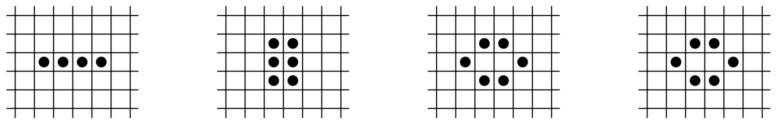
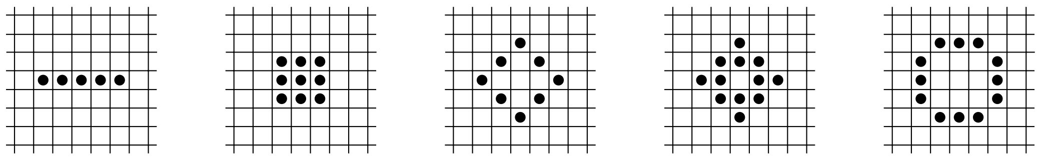
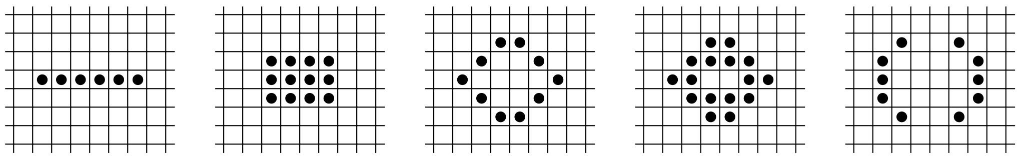


Figure remains stable.

(a)



(b)



(c)

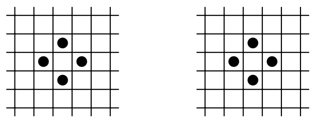
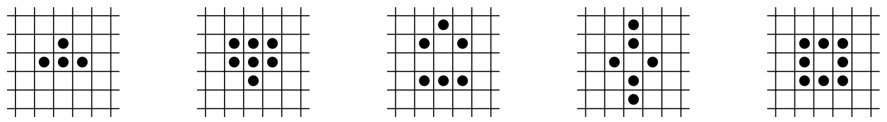


Figure is stable.

(d)


(e)

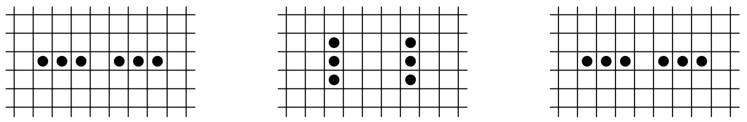
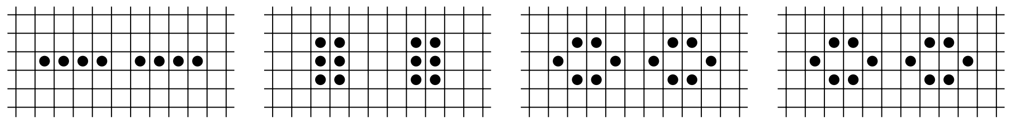
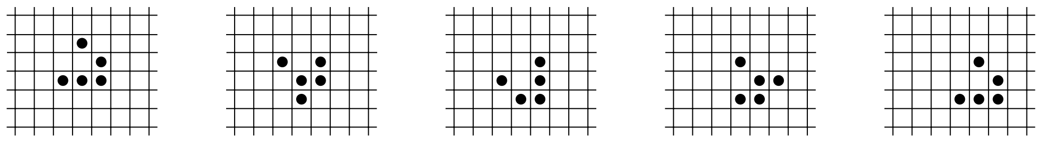


Figure repeats itself.

(f)


(g)


(h)

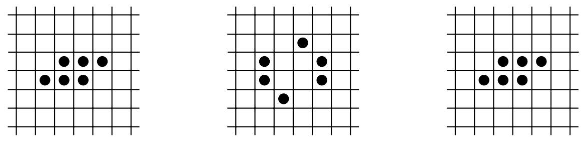
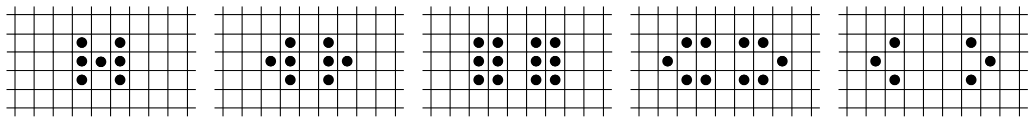
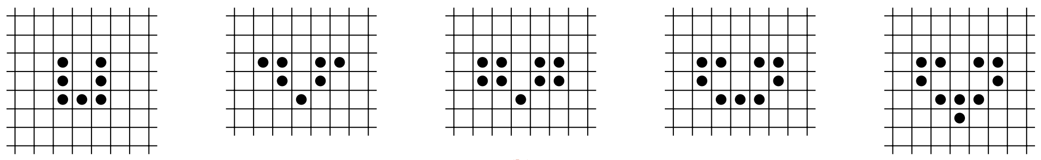


Figure repeats itself.

(i)


(j)


(k)

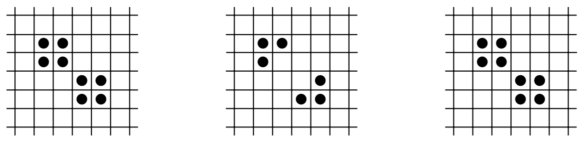


Figure repeats itself.

(l)

1.3 PROGRAMMING STYLE

Exercises 1.3

E1. *What classes would you define in implementing the following projects? What methods would your classes possess?*

(a) *A program to store telephone numbers.*

Answer The program could use classes called Phone_book and Person. The methods for a Phone_book object would include look_up_name, add_person, remove_person. The methods for a Person object would include Look_up_number. Additional methods to initialize and print objects of both classes would also be useful.

(b) *A program to play Monopoly.*

Answer The program could use classes called Game_board, Property, Bank, Player, and Dice. In addition to initialization and printing methods for all classes, the following methods would be useful. The class Game_board needs methods next_card and operate_jail. The class Property needs methods change_owner, look_up_owner, rent, build, mortgage, and unmortgage. The class Bank needs methods pay and collect. The class Player needs methods roll_dice, move_location, buy_property and pay_rent. The class Dice needs a method roll.

(c) *A program to play tic-tac-toe.*

Answer The program could use classes called Game_board and Square. The classes need initialization and printing methods. The class Game_board would also need methods make_move and is_game_over. The class Square would need methods is_occupied, occupied_by, and occupy.

(d) *A program to model the build up of queues of cars waiting at a busy intersection with a traffic light.*

Answer The program could use classes Car, Traffic_light, and Queue. The classes would all need initialization and printing methods. The class Traffic_light would need additional methods change_status and status. The class Queue would need additional methods add_car and remove_car.

E2. *Rewrite the following class definition, which is supposed to model a deck of playing cards, so that it conforms to our principles of style.*

```
    class a {                            //  a deck of cards
int X;  thing Y1[52];  /* X is the location of the top card in the deck.  Y1 lists the cards. */ pub-
lic:  a();
void Shuffle();                          //  Shuffle randomly arranges the cards.
thing d();                               //  deals the top card off the deck
}
           ;
```

Answer

```
class Card_deck {
  Card deck[52];
  int top_card;
 public:
  Card_deck();
  void Shuffle();
  Card deal();
};
```

TOC
Index
Help

E3. *Given the declarations*

```
int a[n][n], i, j;
```

where n *is a constant, determine what the following statement does, and rewrite the statement to accomplish the same effect in a less tricky way.*

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
     a[i][j] = ((i + 1)/(j + 1)) * ((j + 1)/(i + 1));
```

Answer This statement initializes the array a with all 0's except for 1's down the main diagonal. A less tricky way to accomplish this initialization is:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
     if (i == j) a[i][j] = 1;
     else a[i][j] = 0;
```

E4. *Rewrite the following function so that it accomplishes the same result in a less tricky way.*

```
void does_something(int &first, int &second)
{
  first = second − first;
  second = second − first;
  first = second + first;
}
```

Answer The function interchanges the values of its parameters:

```
void swap(int &first, int &second)
/* Pre:  The integers first and second have been initialized.
   Post: The values of first and second have been switched. */
{
   int temp = first;
   first = second;
   second = temp;
}
```

E5. *Determine what each of the following functions does. Rewrite each function with meaningful variable names, with better format, and without unnecessary variables and statements.*

(a)
```
int calculate(int apple, int orange)
{ int peach, lemon;
peach = 0; lemon = 0; if (apple < orange)
peach = orange; else if (orange <= apple)
peach = apple; else { peach = 17;
lemon = 19; }
return(peach);
}
```

Answer The function calculate returns the larger of its two parameters.

```
int larger(int a, int b)
/* Pre:  The integers a and b have been initialized.
   Post: The value of the larger of a and b is returned. */
{
   if (a < b) return b;
   return a;
}
```

(b) *For this part assume the declaration* **typedef float** vector[max];

```
float figure (vector vector1)
{ int loop1, loop4;  float loop2, loop3;
loop1 = 0;  loop2 = vector1[loop1];  loop3 = 0.0;
loop4 = loop1;  for (loop4 = 0;
loop4 < max;  loop4++) { loop1 = loop1 + 1;
loop2 = vector1[loop1 − 1];
loop3 = loop2 + loop3;  } loop1 = loop1 − 1;
loop2 = loop1 + 1;
return(loop2 = loop3/loop2);  }
```

Answer The function figure obtains the mean of an array of floating point numbers.

```
float mean(vector v)
/* Pre:  The vector v contains max floating point values.
   Post: The mean of the values in v is returned. */
{
   float total = 0.0;
   for (int i = 0;  i < max;  i++)
      total += v[i];
   return total/((float) max);
}
```

(c)
```
int question(int &a17, int &stuff)
{ int another, yetanother, stillonemore;
another = yetanother;  stillonemore = a17;
yetanother = stuff;  another = stillonemore;
a17 = yetanother;  stillonemore = yetanother;
stuff = another;  another = yetanother;
yetanother = stuff;  }
```

Answer The function question interchanges the values of its parameters.

```
void swap(int &first, int &second)
/* Pre:  The integers first and second have been initialized.
   Post: The values of first and second have been switched. */
{
   int temp = first;
   first = second;
   second = temp;
}
```

(d)
```
int mystery(int apple, int orange, int peach)
{ if (apple > orange) if (apple > peach) if
(peach > orange) return(peach);  else if (apple < orange)
return(apple);  else return(orange);  else return(apple);  else
if (peach > apple) if (peach > orange) return(orange);  else
return(peach);  else return(apple);  }
```

Answer The function mystery returns the middle value of its three parameters.

```
int median(int a, int b, int c)
/* Pre:   None.
   Post:  Returns the middle value of the three integers a, b, c. */
{
  if (a > b)
    if (c > a) return a;                  //  c > a > b
    else if (c > b) return c;             //  a >= c > b
        else return b;                    //  a > b >= c
  else
    if (c > b) return b;                  //  c > b >= a
    else if (c > a) return c;             //  b >= c > a
        else return a;                    //  b >= a >= c
}
```

E6. *The following statement is designed to check the relative sizes of three integers, which you may assume to be different from each other:*

```
if (x < z) if (x < y) if (y < z) c = 1;  else c = 2;  else
  if (y < z) c = 3;  else c = 4;  else if (x < y)
  if (x < z) c = 5;  else c = 6;  else if (y < z) c = 7;  else
  if (z < x) if (z < y) c = 8;  else c = 9;  else c = 10;
```

(a) *Rewrite this statement in a form that is easier to read.*

Answer

```
if (x < z)
  if (x < y)                              //  x < z and x < y
    if (y < z) c = 1;                     //  x < y < z
    else c = 2;                           //  x < z <= y
  else                                    //  y <= x < z
    if (y < z) c = 3;                     //  y <= x < z
    else c = 4;                           //  impossible
else                                      //  z <= x
  if (x < y)                              //  z <= x < y
    if (x < z) c = 5;                     //  impossible
    else c = 6;                           //  z <= x < y
  else                                    //  z <= x and y <= x
    if (y < z) c = 7;                     //  y < z <= x
        //  z <= y <= x
      if (z < x)                          //  z <= y <= x, z < x
         if (z < y) c = 8;                //  z < y <= x
         else c = 9;                      //  z ==  y < x, impossible
      else c = 10;                        //  y <= z ==  x, impossible
```

(b) *Since there are only six possible orderings for the three integers, only six of the ten cases can actually occur. Find those that can never occur, and eliminate the redundant checks.*

Answer The impossible cases are shown in the remarks for the preceding program segment. After their removal we have:

```
if (x < z)
  if (x < y)                              //  x < z and x < y
    if (y < z) c = 1;                     //  x < y < z
    else c = 2;                           //  x < z <= y
  else c = 3;                             //  y <= x < z
else                                      //  z <= x
  if (x < y) c = 6;                       //  z <= x < y
  else                                    //  z <= x and y <= x
    if (y < z) c = 7;                     //  y < z <= x
    else c = 8;                           //  z <= y <= x
```

TOC
Index
Help

(c) *Write a simpler, shorter statement that accomplishes the same result.*

Answer

```
if ((x < y) && (y < z)) c = 1;
else if ((x < z) && (z < y)) c = 2;
else if ((y < x) && (x < z)) c = 3;
else if ((z < x) && (x < y)) c = 6;
else if ((y < z) && (z < x)) c = 7;
else c = 8;
```

E7. *The following C++ function calculates the cube root of a floating-point number (by the Newton approximation), using the fact that, if y is one approximation to the cube root of x, then*

$$z = \frac{2y + x/y^2}{3}$$

cube roots *is a closer approximation.*

```
float function fcn(float stuff)
{ float april, tim, tiny, shadow, tom, tam, square; int flag;
tim = stuff; tam = stuff; tiny = 0.00001;
if (stuff != 0) do {shadow = tim + tim; square = tim * tim;
tom = (shadow + stuff/square); april = tom/3.0;
if (april*april * april − tam > −tiny) if (april*april*april − tam
      < tiny) flag = 1; else flag = 0; else flag = 0;
if (flag == 0) tim = april; else tim = tam; } while (flag != 1);
if (stuff == 0) return(stuff); else return(april); }
```

(a) *Rewrite this function with meaningful variable names, without the extra variables that contribute nothing to the understanding, with a better layout, and without the redundant and useless statements.*

Answer After some study it can be seen that both stuff and tam play the role of the quantity x in the formula, tim plays the role of y, and tom and april both play the role of z. The object tiny is a small constant which serves as a tolerance to stop the loop. The variable shadow is nothing but $2y$ and square is y^2. The complicated two-line **if** statement checks whether the absolute value $|z^3 - x|$ is less than the tolerance, and the boolean flag is used then only to terminate the loop. Changing all these variables to their mathematical forms and eliminating the redundant ones gives:

```
const double tolerance = 0.00001;
double cube_root(double x)              //   Find cube root of x by Newton method
{
  double y, z;
  y = z = x;
  if (x != 0.0)
    do {
      z = (y + y + x/(y * y))/3.0;
      y = z;
    } while (z * z * z − x > tolerance || x − z * z * z > tolerance);
  return z;
}
```

(b) *Write a function for calculating the cube root of* x *directly from the mathematical formula, by starting with the assignment* y = x *and then repeating*

```
y = (2 * y + (x/(y * y)))/3
```

until abs(y * y * y − x) < 0.00001.

Answer

```
const double tolerance = 0.00001;
double formula(double x)                     // Find cube root of x directly from formula
{
   double y = x;
   if (x != 0.0)
      do {
         y = (y + y + x/(y * y))/3.0;
      } while (y * y * y − x > tolerance || x − y * y * y > tolerance);
   return y;
}
```

(c) *Which of these tasks is easier?*

Answer It is often easier to write a program from scratch than it is to decipher and rewrite a poorly written program.

E8. *The **mean** of a sequence of numbers is their sum divided by the count of numbers in the sequence. The (population) **variance** of the sequence is the mean of the squares of all numbers in the sequence, minus the square of the mean of the numbers in the sequence. The **standard deviation** is the square root of the variance. Write a well-structured C++ function to calculate the standard deviation of a sequence of n floating-point numbers, where n is a constant and the numbers are in an array indexed from 0 to $n-1$, which is a parameter to the function. Use, then write, subsidiary functions to calculate the mean and variance.*

statistics

Answer

```
#include <math.h>
double variance(double v[ ], int n);
double standard_deviation(double v[ ], int n)  //  Standard deviation of v[ ]
{
    return sqrt(variance(v, n));
}
```

This function uses a subsidiary function to calculate the variance.

```
double mean(double v[ ], int n);
double variance(double v[ ], int n)
   //  Find the variance for n numbers in v[ ]
{
   int i;
   double temp;
   double sum_squares = 0.;
   for (i = 0;  i < n;  i++)
      sum_squares += v[i] * v[i];
   temp = mean(v, n);
   return sum_squares/n − temp * temp;
}
```

This function in turn requires another function to calculate the mean.

```
double mean(double v[ ], int n)              //  Find the mean of an array of n numbers
{
    int i;
    double sum = 0.0;
    for (i = 0;  i < n;  i++)
        sum += v[i];
    return sum/n;
}
```

E9. *Design a program that will plot a given set of points on a graph. The input to the program will be a text file, each line of which contains two numbers that are the x and y coordinates of a point to be plotted. The program will use a function to plot one such pair of coordinates. The details of the function involve the specific method of plotting and cannot be written since they depend on the requirements of the plotting equipment, which we do not know. Before plotting the points the program needs to know the maximum and minimum values of x and y that appear in its input file. The program should therefore use another function* bounds *that will read the whole file and determine these four maxima and minima. Afterward, another function is used to draw and label the axes; then the file can be reset and the individual points plotted.*

plotting

(a) *Write the main program, not including the functions.*

Answer

```
#include <fstream.h>
#include "calls.h"
#include "bounds.c"
#include "draw.c"
int main(int argc, char *argv[ ])
        //  Read coordinates from file and plot coordinate pairs.
{
    ifstream file(argv[1]);
    if (file == 0) {
        cout << "Can not open input points file" << endl;
        cout << "Usage:\n\t plotter input_points " << endl;
        exit(1);
    }
    double xmax, xmin;                    //  bounds for x values
    double ymax, ymin;                    //  bounds for y values
    double x, y;                          //  x, y values to plot

    bounds(file, xmax, xmin, ymax, ymin);
    draw_axes(xmax, xmin, ymax, ymin);
    file.seekg(0, ios::beg);              //  reset to beginning of file
    while (!file.eof()) {
        file >> x >> y;
        plot(x, y);
    }
}
```

(b) *Write the function* bounds.

Answer

```
void bounds(ifstream &file, double &xmax, double &xmin,
            double &ymax, double &ymin)
            //  determine maximum and minimum values for x and y
{
    double x, y;
    file >> x >> y;
    xmax = xmin = x;
    ymax = ymin = y;

    while (!file.eof()) {
        file >> x >> y;
        if (x < xmin)
            xmin = x;
        if (x > xmax)
            xmax = x;
        if (y < ymin)
            ymin = y;
        if (y > ymax)
            ymax = y;
    }
}
```

(c) *Write the preconditions and postconditions for the remaining functions together with appropriate documentation showing their purposes and their requirements.*

Answer

```
void draw_axes(double xmax, double xmin,
               double ymax, double ymin)
/* Pre:  The parameters xmin, xmax, ymin, and xmax give bounds for the x and y co-ordinates.
   Post: Draws and labels axes according to the given bounds. */
{
}
void plot(double x, double y)
/* Pre:  The parameters x and y give co-ordinates of a point.
   Post: The point is plotted to the ouput graph. */
{
}
```

1.4 CODING, TESTING, AND FURTHER REFINEMENT

Exercises 1.4

E1. *If you suspected that the Life program contained errors, where would be a good place to insert scaffolding into the main program? What information should be printed out?*

Answer Since much of the program's work is done in neighbor_count, a good place would be within the loops of the update method, just before the **switch** statement. The values of row, col, and neighbor_count could be printed.

E2. *Take your solution to Section 1.3, Exercise E9 (designing a program to plot a set of points), and indicate good places to insert scaffolding if needed.*

Answer Suitable places might be after draw_axes (printing its four parameters) and (with a very small test file to plot) after the call to plot (printing the coordinates of the point just plotted).

E3. *Find suitable black-box test data for each of the following:*

(a) *A function that returns the largest of its three parameters, which are floating-point numbers.*

Answer

easy values:	$(1,2,3)$, $(2,3,1)$, $(3,2,1)$.
typical values:	$(0,0.5,-9.6)$, $(1.3,3.5,0.4)$, $(-2.1,-3.5,-1.6)$.
extreme values:	$(0,0,0)$, $(0,1,1)$, $(1,0,1)$, $(0,0,1)$

(b) *A function that returns the square root of a floating-point number.*

Answer

easy values:	1, 4, 9, 16.
typical values:	0.4836, 56.7, 9762.34.
extreme value:	0.0.
illegal values:	-0.4353, -9.

(c) *A function that returns the least common multiple of its two parameters, which must be positive integers. (The* **least common multiple** *is the smallest integer that is a multiple of both parameters. Examples: The least common multiple of* 4 *and* 6 *is* 12*, of* 3 *and* 9 *is* 9*, and of* 5 *and* 7 *is* 35*.)*

Answer

easy values:	$(3,4)$, $(4,8)$, $(7,3)$.
typical values:	$(7,8)$, $(189,433)$, $(1081,1173)$.
illegal values:	$(7,-6)$, $(0,5)$, $(0,0)$, $(-1,-1)$.

(d) *A function that sorts three integers, given as its parameters, into ascending order.*

Answer

easy values:	$(5,3,2)$, $(2,3,5)$, $(3,5,2)$, $(5,2,3)$, $(-1,-2,-3)$
extreme values:	$(1,1,1)$, $(1,2,1)$, $(1,1,2)$.
typical values:	$(487,-390,20)$, $(0,589,333)$.

(e) *A function that sorts an array* a *containing* n *integers indexed from* 0 *to* n − 1 *into ascending order, where* a *and* n *are both parameters.*

Answer For the number n of entries to be sorted choose values such as 2, 3, 4 (easy values), 0, 1, maximum size of a (extreme values), and −1 (illegal value). Test with all entries of a the same value, the entries already in ascending order, the entries in descending order, and the entries in random order.

E4. *Find suitable glass-box test data for each of the following:*

(a) *The statement*

```
if (a < b) if (c > d) x = 1; else if (c == d) x = 2;
else x = 3; else if (a == b) x = 4; else if (c == d) x = 5;
else x = 6;
```

Answer Choose values for a and b, such as $(1,2)$, $(2,1)$, and $(1,1)$, so that each of a < b, a > b, and a == b holds true. Choose three similar pairs of values for c and d, giving nine sets of test data.

(b) *The* Life *method* neighbor_count(row, col).

Answer Set row in turn to 1, maxrow, and any intermediate value, as well as 0 and maxrow + 1 (as illegal values). Choose col similarly. For each of the legal (row, col) pairs set up the Life object so that the number of living neighbors of (row, col) is each possible value between 0 and 8. Finally, make the cell at (row, col) itself either living or dead. (This process gives 98 sets of test data, provided maxrow and maxrow are each at least 3.)

Programming Projects 1.4

P1. *Enter the Life program of this chapter on your computer and make sure that it works correctly.*

Answer The complete program is implemented in the `life` subdirectory for Chapter 1.

```
#include "../../c/utility.h"
#include "life.h"

#include "../../c/utility.cpp"
#include "life.cpp"

int main()  //  Program to play Conway's game of Life.
/*
Pre:  The user supplies an initial configuration of living cells.
Post: The program prints a sequence of pictures showing the changes in
      the configuration of living cells according to the rules for
      the game of Life.
Uses: The class Life and its methods initialize(), print(), and
      update(); the functions  instructions(),  user_says_yes().
*/
```

```
{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}

const int maxrow = 20, maxcol = 60;    //  grid dimensions

class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int grid[maxrow + 2][maxcol + 2];  //  Allow two extra rows and columns.
   int neighbor_count(int row, int col);
};

void Life::print()
/*
Pre:  The Life object contains a configuration.
Post: The configuration is written for the user.
*/

{
   int row, col;
   cout << "\nThe current Life configuration is:" <<endl;
   for (row = 1; row <= maxrow; row++) {
      for (col = 1; col <= maxcol; col++)
         if (grid[row][col] == 1) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}

int Life::neighbor_count(int row, int col)
/*
Pre:  The Life object contains a configuration, and the coordinates
      row and col define a cell inside its hedge.
Post: The number of living neighbors of the specified cell is returned.
*/

{
   int i, j;
   int count = 0;
   for (i = row - 1; i <= row + 1; i++)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];  //  Increase the count if neighbor is alive.
   count -= grid[row][col];    //  A cell is not its own neighbor.
   return count;
}
```

```
void Life::update()
/*
Pre:  The Life object contains a configuration.
Post: The Life object contains the next generation of configuration.
*/

{
   int row, col;
   int new_grid[maxrow + 2][maxcol + 2];

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         switch (neighbor_count(row, col)) {
         case 2:
            new_grid[row][col] = grid[row][col]; //  Status stays the same.
            break;
         case 3:
            new_grid[row][col] = 1;              //  Cell is now alive.
            break;
         default:
            new_grid[row][col] = 0;              //  Cell is now dead.
         }

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         grid[row][col] = new_grid[row][col];
}

void Life::initialize()
/*
Pre:  None.
Post: The Life object contains a configuration specified by the user.
*/

{
   int row, col;
   for (row = 0; row <= maxrow+1; row++)
      for (col = 0; col <= maxcol+1; col++)
         grid[row][col] = 0;
   cout << "List the coordinates for living cells." << endl;
   cout << "Terminate the list with the the special pair -1 -1" << endl;
   cin >> row >> col;

   while (row != -1 || col != -1) {
      if (row >= 1 && row <= maxrow)
         if (col >= 1 && col <= maxcol)
            grid[row][col] = 1;
         else
            cout << "Column " << col << " is out of range." << endl;
      else
         cout << "Row " << row << " is out of range." << endl;
      cin >> row >> col;
   }
}

void instructions()
/*
Pre:  None.
Post: Instructions for using the Life program have been printed.
*/
```

```
{
 cout << "Welcome to Conway's game of Life." << endl;
 cout << "This game uses a grid of size "
      << maxrow << " by " << maxcol << " in which" << endl;
 cout << "each cell can either be occupied by an organism or not." << endl;
 cout << "The occupied cells change from generation to generation" << endl;
 cout << "according to the number of neighboring cells which are alive."
      << endl;
}
```

P2. *Test the Life program with the examples shown in Figure 1.1.*

Answer See the solution to the exercise in Section 1.2.

P3. *Run the Life program with the initial configurations shown in Figure 1.4. Several of these go through many changes before reaching a configuration that remains the same or has predictable behavior.*

Answer This is a demonstration to be performed by computer.

1.5 PROGRAM MAINTENANCE

Exercises 1.5

E1. *Sometimes the user might wish to run the Life game on a grid smaller than* 20×60*. Determine how it is possible to make* maxrow *and* maxcol *into variables that the user can set when the program is run. Try to make as few changes in the program as possible.*

Answer The integers maxrow and maxcol should become data members of the **class** Life. The method initialize, must now ask for input of the two data members maxrow and maxcol. Upper bounds of 20 and 60 for these integers should be stored in new constants called maxrowbound and maxcolbound. The amended file `life.h` now takes the form.

```
const int maxrowbound = 20, maxcolbound = 60;
      //   bounds on grid dimensions
class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int maxrow, maxcol;
   int grid[maxrowbound + 2][maxcolbound + 2];
        //   allows for two extra rows and columns
   int neighbor_count(int row, int col);
};
```

As noted above, the method initialize needs minor modifications.

E2. *One idea for speeding up the function* Life :: neighbor_count(row, col) *is to delete the hedge (the extra rows and columns that are always dead) from the arrays* grid *and* new_grid*. Then, when a cell is on the boundary,* neighbor_count *will look at fewer than the eight neighboring cells, since some of these are outside the bounds of the grid. To do this, the function will need to determine whether or not the cell* (row, col) *is on the boundary, but this can be done outside the nested loops, by determining, before the loops commence, the lower and upper bounds for the loops. If, for example,* row *is as small as allowed, then the lower bound for the row loop is* row*; otherwise, it is* row − 1*. Determine, in terms of the size of the grid, approximately how many statements are executed by the original version of* neighbor_count *and by the new version. Are the changes proposed in this exercise worth making?*

Answer We need four **if** statements at the beginning of neighbor_count to determine whether (row, col) is on the boundary. This gives a total of $4 \times$ maxrow $\times$ maxcol extra statements. If the cell is in the first or last row or column, but not in a corner, then the nested loops would iterate 3 fewer times. There are $2 \times$ maxrow $+ 2 \times$ maxcol $- 8$ such positions. With the cell in one of the 4 corner positions, the nested loops would iterate 5 fewer times. Hence the total number of statements saved is

$$-4 \times \text{maxrow} \times \text{maxcol} + (2 \times \text{maxrow} + 2 \times \text{maxcol} - 8) + 20.$$

Thus this proposed change actually costs additional work, except for very small values of maxrow and maxcol.

The modified function could be coded as follows.

```
int Life :: neighbor_count(int row, int col)
/* Pre:  The Life object contains a configuration, and the coordinates row and col define a cell
         inside its hedge.
   Post: The number of living neighbors of the specified cell is returned. */
{
   int i, j;
   int count = 0;
   int rowlow = row − 1, rowhigh = row + 1,
       collow = col − 1, colhigh = col + 1;
   if (row == 1) rowlow++;
   if (row == maxrow) rowhigh−−;
   if (col == 1) collow++;
   if (col == maxcol) colhigh−−;
   for (i = rowlow; i <= rowhigh; i++)
      for (j = collow; j <= colhigh; j++)
         count += grid[i][j];              //  Increase the count if neighbor is alive
   count −= grid[row][col];                //  Reduce count, since cell is not its own neighbor
   return count;
}
```

Programming Projects 1.5

P1. *Modify the Life function* initialize *so that it sets up the initial* Life :: grid *configuration by accepting occupied positions as a sequence of blanks and* x*'s in appropriate rows, rather than requiring the occupied positions to be entered as numerical coordinate pairs.*

Answer The following program includes all the changes for projects P1–P6. The changes required for Projects P7 and P8 are system dependent and have not been implemented.

```
#include "../../c/utility.h"
#include "life.h"
#include "../../c/utility.cpp"
#include "life.cpp"

int main()  //  Program to play Conway's game of Life.
/*
Pre:  The user supplies an initial configuration of living cells.
Post: The program prints a sequence of pictures showing the changes in
      the configuration of living cells according to the rules for
      the game of Life.
Uses: The class Life and methods initialize(), print(), and update();
      the functions  instructions(),  user_says_yes().
*/
```

```
{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print(cout);
   bool command;
   do {
      configuration.update();
      configuration.print(cout);
      cout << "Continue viewing without changes? " << endl;
      if (!(command = user_says_yes())) {
         cout << "Do you want to quit? " << flush;
         if (user_says_yes()) command = false;
         else {
            cout << "Do you want help? " << flush;
            if (user_says_yes()) instructions();
            cout << "Do you want to make any changes? " << flush;
            if (user_says_yes()) configuration.edit();
            command = true;
         }
      }
   } while (command);

   cout << "Do you want to save the final position to file? " << flush;
   if (user_says_yes()) {
      char name[1000];
      cout << "Give the save file name: " << flush;
      cin  >> name;
      ofstream f(name);
      configuration.print(f);
   }
}


const int maxrow = 20, maxcol = 60;    //  grid dimensions

class Life {
public:
   void initialize();
   void print(ostream &f);
   void update();
   void edit();
private:
   int grid[maxrow + 2][maxcol + 2];  //  Allow two extra rows and columns.
   int neighbor_count(int row, int col);
   bool step_mode;
};


void Life::edit()
/* Post:  User has edited configuration, and/or step mode */

{
   cout << "Do you want to switch the step mode? " << flush;
   if (user_says_yes()) step_mode = !step_mode;

   cout << "Do you want to change the configuration? " << flush;
   if (!user_says_yes()) return;
   int row, col;
   cout << "List the coordinates for cells to change." << endl;
   cout << "Terminate the list with the the special pair -1 -1" << endl;
   cin >> row >> col;
   while (row != -1 || col != -1) {
      if (row >= 1 && row <= maxrow)
         if (col >= 1 && col <= maxcol)
            grid[row][col] = 1 - grid[row][col];
         else
            cout << "Column " << col << " is out of range." << endl;
      else
         cout << "Row " << row << " is out of range." << endl;
      cin >> row >> col;
   }
}

void Life::print(ostream &f)
/*
Pre:  The Life object contains a configuration.
Post: The configuration is written to stream f for the user.
*/

{
   int row, col;
   f << "The current Life configuration is:" << endl;
   for (row = 1; row <= maxrow; row++) {
      for (col = 1; col <= maxcol; col++)
         if (grid[row][col] == 1) f << '*';
         else f << ' ';
      f << endl;
   }
   f << endl;
}

int Life::neighbor_count(int row, int col)
/*
Pre:  The Life object contains a configuration, and the coordinates
      row and col define a cell inside its hedge.
Post: The number of living neighbors of the specified cell is returned.
*/

{
   int i, j;
   int count = 0;
   for (i = row - 1; i <= row + 1; i++)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];  //  Increase the count if neighbor is alive.
   count -= grid[row][col];    //  A cell is not its own neighbor.
   return count;
}

void Life::update()
/*
Pre:  The Life object contains a configuration.
Post: The Life object contains the next generation of configuration.
*/

{
   int row, col;
   int new_grid[maxrow + 2][maxcol + 2];
```

```
    for (row = 1; row <= maxrow; row++)
       for (col = 1; col <= maxcol; col++)
          switch (neighbor_count(row, col)) {
          case 2:
             new_grid[row][col] = grid[row][col];  //  Status stays the same.
             break;
          case 3:
             if (step_mode &&grid[row][col] == 0)
               cout << "Vivify cell " << row << " " << col << endl;
             new_grid[row][col] = 1;                //  Cell is now alive.
             break;
          default:
             if (step_mode && grid[row][col] == 1)
               cout << "Kill cell " << row << " " << col << endl;
             new_grid[row][col] = 0;                //  Cell is now dead.
          }

    for (row = 1; row <= maxrow; row++)
       for (col = 1; col <= maxcol; col++)
          grid[row][col] = new_grid[row][col];
 }

 void Life::initialize()
 /*
 Pre:  None.
 Post: The Life object contains a configuration specified by the user.
 */

 {
    step_mode = false;
    int row, col;
    char c;
    bool from_file = false;
    ifstream f;

    for (row = 0; row <= maxrow+1; row++)
       for (col = 0; col <= maxcol+1; col++)
          grid[row][col] = 0;

    cout << "Do you want to read input from a prepared file? " << flush;
    if (user_says_yes()) {
       cout << "Enter input file name: " << flush;
       char name[1000];
       cin  >> name;
       f.open(name);
       from_file = true;
       while (f.get() != '\n');
    }
    else {
       while (cin.get() != '\n');
       cout << "Enter a picture of the initial configuration." << endl;
       cout << "Please enter nonblanks to signify living cells,\n";
       cout << "and blanks to signify empty cells.\n";
       cout << "Enter a return (newline) to terminate a row of input.\n";
       cout << "Enter a character $ if all input is complete before the "
            << "last row." << endl;
    }
```

```
   for (row = 1; row <= maxrow; row++) {
      if (!from_file) cout << row << " : " << flush;
      for (col = 1; col <= maxcol; col++) {
         if (from_file) f.get(c);
         else         cin.get(c);
         if (c == '\n' || c == '$') break;
         if (c != ' ') grid[row][col] = 1;
         else          grid[row][col] = 0;
      }
      if (c == '$') break;
      if (from_file) while (c != '\n') f.get(c);
      else while (c != '\n') cin.get(c);
   }
}

void instructions()
/*
Pre:  None.
Post: Instructions for using the Life program have been printed.
*/

{
 cout << "Welcome to Conway's game of Life." << endl;
 cout << "This game uses a grid of size "
      << maxrow << " by " << maxcol << " in which" << endl;
 cout << "each cell can either be occupied by an organism or not." << endl;
 cout << "The occupied cells change from generation to generation" << endl;
 cout << "according to the number of neighboring cells which are alive."
      << endl;
}
```

P2. *Add a feature to the function* initialize *so that it can, at the user's option, either read its initial configuration from the keyboard or from a file. The first line of the file will be a comment giving the name of the configuration. Each remaining line of the file will correspond to a row of the configuration. Each line will contain* x *in each living position and a blank in each dead position.*

Answer See project P1.

P3. *Add a feature to the Life program so that, at termination, it can write the final configuration to a file in a format that can be edited by the user and that can be read in to restart the program (using the feature of Project P2).*

Answer See project P1. The program subdirectories contain a file of test data called FILE.

P4. *Add a feature to the Life program so, at any generation, the user can edit the current configuration by inserting new living cells or by deleting living cells.*

Answer See project P1.

P5. *Add a feature to the Life program so, if the user wishes at any generation, it will display a help screen giving the rules for the Life game and explaining how to use the program.*

Answer See project P1.

P6. *Add a step mode to the Life program, so it will explain every change it makes while going from one generation to the next.*

Answer See project P1.

P7. *Use direct cursor addressing (a system-dependent feature) to make the Life method* print *update the configuration instead of completely rewriting it at each generation.*

Answer The changes required are system dependent and have not been implemented.

P8. *Use different colors in the Life output to show which cells have changed in the current generation and which have not.*

Answer The changes required are system dependent and have not been implemented.

1.6 CONCLUSIONS AND PREVIEW

Programming Projects 1.6

P1. *A* ***magic square*** *is a square array of integers such that the sum of every row, the sum of every column, and sum of each of the two diagonals are all equal.*

(a) *Write a program that reads a square array of integers and determines whether or not it is a magic square.*

(b) *Write a program that generates a magic square by the following method. This method works only when the size of the square is an odd number. Start by placing 1 in the middle of the top row. Write down successive integers 2, 3, . . . along a diagonal going upward and to the right. When you reach the top row (as you do immediately since 1 is in the top row), continue to the bottom row as though the bottom row were immediately above the top row. When you reach the rightmost column, continue to the leftmost column as though it were immediately to the right of the rightmost one. When you reach a position that is already occupied, instead drop straight down one position from the previous number to insert the new one.*

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "square.h"
#include "square.cpp"

int main()
{
  cout << "Magic Square Program" << endl << endl;
  cout << "Enter a square of integers:" << endl;
  Square s;
  s.read();
  if (s.is_magic()) cout << "That square is magic!" << endl;
  else          cout << "That square is not magic." << endl;
  cout << endl << endl << "Generating a 5 x 5 magic square:" << endl;
  cout << "--------------------------------" << endl << endl;
  s.generate(5);
  s.print();
  cout << endl << endl << "Generating a 7 x 7 magic square:" << endl;
  cout << "--------------------------------" << endl << endl;
  s.generate(7);
  s.print();
}

const int max_size = 9;

class Square {
 public:
   void read();
   void print();
   bool is_magic();
   void generate(int n);
 private:
   int sum_row(int i);
   int sum_col(int i);
   int sum_maindiag();
   int sum_other();
   int size;
   int grid[max_size][max_size];
};

void Square::generate(int n)
/*  Pre: n is odd
    Post: a magic n x n square is generated  */

{
   if (n % 2 == 0) {
      cout << "Error: the side must be odd:" << endl;
   }

   size = n;
   if (size < 0 || size > 9) {
      cout << "Error: Side Size --- Out of range." << endl;
      return;
   }

   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         grid[i][j] = 0;

   int count = 0;
   int row = n - 2;
   int col = n / 2 - 1;
   while (count < n * n) {
      if (grid[(row + 1) % n][(col + 1) % n] == 0) {
         row = (row + 1) % n;
         col = (col + 1) % n;
      }
      else {
         row = (row + n - 1) % n;
      }
      count++;
      grid[row][col] = count;
   }
}

void Square::print()
/* Post: The square data is printed. */

{
   for (int i = size - 1; i >= 0; i--) {
      for (int j = 0; j < size; j++) {
         if (grid[i][j] < 10) cout << " ";
         cout << grid[i][j] << " ";
      }
      cout << endl;
   }
}

void Square::read()
/* Post: the square data is read  */
```

```
{
   cout << "Enter the side size: " << flush;
   cin  >> size;
   if (size < 0 || size > 9) {
      cout << "Error: Out of range." << endl;
      return;
   }

   cout << "Now type in the data, row by row." << endl;
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         cin >> grid[i][j];
}

int Square::sum_other()
/*   Post: Returns the non-main diagonal sum   */

{
   int sum = 0;
   for (int j = 0; j < size; j++) sum += grid[size - j - 1][j];
   return sum;
}

int Square::sum_maindiag()
/*   Post: Returns the main diagonal sum   */

{
   int sum = 0;
   for (int j = 0; j < size; j++) sum += grid[j][j];
   return sum;
}

int Square::sum_row(int i)
/*   Pre: i is a valid row of the square
    Post: Returns the row sum of the ith row   */

{
   int sum = 0;
   for (int j = 0; j < size; j++) sum += grid[i][j];
   return sum;
}

int Square::sum_col(int i)
/*   Pre: i is a valid col of the square
    Post: Returns the col sum of the ith col   */

{
   int sum = 0;
   for (int j = 0; j < size; j++) sum += grid[j][i];
   return sum;
}

bool Square::is_magic()
{
   int number = sum_maindiag();
   if (sum_other() != number) return false;
   for (int i = 0; i < size; i++) {
      if (sum_row(i) != number) return false;
      if (sum_col(i) != number) return false;
   }
   return true;
}
```

TOC
Index
Help

P2. ***One-Dimensional Life*** *takes place on a straight line instead of a rectangular grid. Each cell has four neighboring positions: those at distance one or two from it on each side. The rules are similar to those of two-dimensional Life except (1) a dead cell with either two or three living neighbors will become alive in the next generation, and (2) a living cell dies if it has zero, one, or three living neighbors. (Hence a dead cell with zero, one, or four living neighbors stays dead; a living cell with two or four living neighbors stays alive.) The progress of sample communities is shown in Figure 1.6. Design, write, and test a program for one-dimensional Life.*

Answer

```
#include "../../c/utility.h"
#include "life.h"

#include "../../c/utility.cpp"
#include "life.cpp"

int main()  //  Program to play 1-dimensional Life.
/*
Pre:  The user supplies an initial configuration of living cells.
Post: The program prints a sequence of pictures showing the changes in
      the configuration of living cells according to the rules for
      the game of Life.
Uses: The class Life and its methods initialize(), print(), and update();
      the functions  instructions(),  user_says_yes().
*/

{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}

const int maxcol = 60;    //  grid dimensions

class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int grid[maxcol + 4];  //  allows for four extra columns
   int neighbor_count(int col);
};


void Life::print()
/*
Pre:  The Life object contains a configuration.
Post: The configuration is written for the user.
*/
```

```
{
   int col;
   cout << "\nThe current Life configuration is:" <<endl;
      for (col = 2; col <= maxcol + 1; col++)
         if (grid[col] == 1) cout << '*';
         else cout << ' ';
   cout << endl;
}

int Life::neighbor_count(int col)
/*
Pre:  The Life object contains a configuration, and the coordinates
      col defines a cell inside its hedge.
Post: The number of living neighbors of the specified cell is returned.
*/

{
   int j;
   int count = 0;
      for (j = col - 2; j <= col + 2; j++)
         count += grid[j];  //  Increase the count if neighbor is alive.
   count -= grid[col];      //  A cell is not its own neighbor.
   return count;
}

void Life::update()
/*
Pre:  The Life object contains a configuration.
Post: The Life object contains the next generation of configuration.
*/

{
   int col;
   int new_grid[maxcol + 4];
      for (col = 2; col <= maxcol + 1; col++)
         switch (neighbor_count(col)) {
         case 4:
            new_grid[col] = grid[col];  //  Status stays the same.
            break;
         case 3:
            new_grid[col] = 1 - grid[col];  //  Status switches.
            break;
         case 2:
            new_grid[col] = 1;                //  Cell is now alive.
            break;
         default:
            new_grid[col] = 0;                //  Cell is now dead.
         }

      for (col = 2; col <= maxcol + 1; col++)
         grid[col] = new_grid[col];
}

void Life::initialize()
/*
Pre:  None.
Post: The Life object contains a configuration specified by the user.
*/
```

```
{
  int col;
     for (col = 0; col <= maxcol+3; col++)
        grid[col] = 0;
  cout << "List the coordinates for living cells." << endl;
  cout << "Terminate the list with the the special symbol -1" << endl;
  cin  >> col;

  while (col != -1) {
        if (col >= 1 && col <= maxcol)
           grid[col + 1] = 1;
        else
           cout << "Column " << col << " is out of range." << endl;
     cin >> col;
  }
}

void instructions()
/*
Pre:  None.
Post: Instructions for using the Life program have been printed.
*/

{
 cout << "Welcome to 1-dimensional Life." << endl;
 cout << "This game uses a grid of size "
      << maxcol << " in which" << endl;
 cout << "each cell can either be occupied by an organism or not." << endl;
 cout << "The occupied cells change from generation to generation" << endl;
 cout << "according to the number of neighboring cells which are alive."
      << endl;
}
```

P3. (a) *Write a program that will print the calendar of the current year.*

(b) *Modify the program so that it will read a year number and print the calendar for that year. A year is a leap year (that is, February has 29 instead of 28 days) if it is a multiple of 4, except that century years (multiples of 100) are leap years only when the year is divisible by 400. Hence the year 1900 is not a leap year, but the year 2000 is a leap year.*

(c) *Modify the program so that it will accept any date (day, month, year) and print the day of the week for that date.*

(d) *Modify the program so that it will read two dates and print the number of days from one to the other.*

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "calendar.h"
#include "auxil.cpp"
#include "calendar.cpp"

main()
{
   cout << "Calendar testing program." << endl << endl;
   cout << "Printing a calendar for 1999." << endl << endl;
   Calendar x;
   x.print();

   cout << "Enter two dates.  \n"
        << "The difference between them will be printed." << endl;
   print_difference();
}
```

```
class Calendar {
 public:
   int first_day();
   int day_of_year();
   int day_of_time();
   int weekday();
   Calendar();
   Calendar(int d, int m, int y);
   void set_date(int d, int m, int y);
   void get_date(int &d, int &m, int &y);
   void print(int y);
   void print();
 private:
   int day, month, year;
   int month_length(int m, int y);
};


int Calendar::weekday()
{
   return (first_day() + day_of_year() + 6) % 7;
}

int Calendar::month_length(int m, int y)
{
   int add_on = 0;
   if (m == 2 && (y % 4 == 0)) {
      add_on++;
      if ((y % 100 == 0) && (y % 400 != 0)) add_on--;
   }
   return months[m] + add_on;
}

void Calendar::print()
{
   print(year);
}

void Calendar::print(int y)
{
   cout << "\n\n------------  " << y << "  -------------\n\n"
        << endl;
   for (int m = 1; m <= 12; m++) {   // months
      Calendar temp(1, m, y);
      print_header(m);
      print_days(month_length(m, y), temp.weekday());
   }
}

void Calendar::get_date(int &d, int &m, int &y)
{
   d = day;
   m = month;
   y = year;
}
```

TOC
Index
Help

```
void Calendar::set_date(int d, int m, int y)
{
   day = d;
   month = m;
   year = y;
}

Calendar::Calendar(int d, int m, int y)
{
   if (!months_set) set_months();
   set_date(d, m, y);
}

Calendar::Calendar()
{
   if (!months_set) set_months();
   set_date(2, 8, 1999);
}

int Calendar::first_day()
/* Post: The day number of the first day of a given
         year is returned.  Sunday = 0, Monday = 1, etc. */

{
   return (year + (year - 1) / 4 - (year - 1) / 100 +
                    (year - 1) / 400) % 7;
}

int Calendar::day_of_year()
/* Post: The day number in a year is returned
         Jan 1 = day 1, Jan 2 = day 2 etc.              */

{
   int count = 0;
   for (int j = 0; j < month; j++)
      count += month_length(j, year);
   count += day;
   return count;
}

int Calendar::day_of_time()
/* Post: Returns the number of days since the day before
         Jan 1 of the year 0 */

{
   int answer = day_of_year();
   answer += 365 * year;
   answer += (year + 3) / 4;
   answer -= (year - 1) / 100;
   answer += (year - 1) / 400;
   return answer;
}
```

(e) *Using the rules on leap years, show that the sequence of calendars repeats exactly every 400 years.*

Answer Since the rules for leap years repeat only every 400 years, the shortest period over which the sequence of calendars can repeat is 400 years. To show that it does repeat each 400 years, observe that, since $365 = (7 \times 52) + 1$, the calendar advances one day for every year that is not a leap year; that is, if January 1 is a Sunday this year, then it will be a Monday next year, and so on. The calendar advances an additional day for each leap year. In 400 years there would be 100 leap years except that 3 of the 4 century years are not leap years, so there are 97 leap years in 400 years. Hence in 400 years the calendar advances $400 + 97 = 497$ days, and since $497 = 7 \times 71$ is an integer number of weeks, at the end of 400 years the calendar will have returned to where it started.

(f) *What is the probability (over a 400-year period) that the 13th of a month is a Friday? Why is the 13th of the month more likely to be a Friday than any other day of the week? Write a program to calculate how many Friday the 13ths occur in this century.*

REVIEW QUESTIONS

1. *When is it appropriate to use one-letter variable names?*

It is almost always better to avoid single-letter names. Only when imitating a mathematical style (where single-letter variables are the norm) is it appropriate, for example for variables such as x and y in an equation whose meanings are well known and cannot be confused with other variables.

2. *Name four kinds of information that should be included in program documentation.*

(1) A prologue including identification of the program and programmer, purpose of the program, its method, data used by the program and changes it makes, and reference to external documentation; (2) an explanation of each constant, type, and variable; (3) an introduction to each significant section of the program; and (4) an explanation of difficult or tricky code (if any).

3. *What is the difference between external and internal documentation?*

Internal documentation consists of comments in the code itself. External documentation is a user guide, separate explanation, or other document separate from the program.

4. *What are pre- and postconditions?*

Preconditions are statements that are required to be true when a program or function begins. Postconditions are statements that will then be true when the program or function ends.

5. *Name three kinds of parameters. How are they processed in C++?*

Parameters may be *input*, *output*, or *inout*. All three kinds may be reference parameters, although input parameters are usually (but not always) either constant reference parameters or value parameters. Value parameters are only usable as input parameters.

6. *Why should side effects of functions be avoided?*

Side effects should be avoided because they may alter the performance of other functions and make debugging unnecessarily difficult.

7. *What is a program stub?*

Stubs are short dummy functions that temporarily take the place of an actual function so that the calling program can be debugged and tested before the function is written.

8. *What is the difference between stubs and drivers, and when should each be used?*

Stubs are used to test a parent program when the actual function is not yet available. Drivers are shortened, dummy programs that are used to test and debug a completed function.

9. *What is a structured walkthrough?*

In a structured walkthrough the programmer shows the completed program to one or more other programmers and describes completely what happens in the main program as well as in each of the functions. The questions and advice of other people assist the programmer in objectively debugging the program.

10. *What is scaffolding in a program, and when is it used?*

Scaffolding consists of insertions into a program to print out the status of important variables at various stages of program execution. It is used for debugging and commented out (or removed) after the program works correctly.

11. *Name a way to practice defensive programming.*

Place **if** statements at the beginning of functions to verify that the preconditions actually do hold.

12. *Give two methods for testing a program, and discuss when each should be used.*

The ***black-box method*** of program testing is used when the person testing the program is only interested in whether or not the program works, and not in the actual details of its functioning. In ***glass-box testing*** logical structure of the program itself is examined, and, for each alternative that can occur within the program, test data is chosen that will lead to that alternative.

13. *If you cannot immediately picture all details needed for solving a problem, what should you do with the problem?*

You should divide the problem into smaller subproblems, and continue doing so until every detail of each separate subproblem is clear and manageable.

14. *What are preconditions and postconditions of a subprogram?*

Preconditions are statements that are correct when the subprogram begins, and postconditions are statements that are correct after the subprogram has finished. To be useful, these statements should concern the data that the subprogram uses or manipulates.

15. *When should allocation of tasks among functions be made?*

Allocation of tasks among functions should be done after the solution to the problem has been outlined, and while refinements are being made to that solution.

16. *How long should coding be delayed?*

Coding should be delayed until after you have completely and precisely defined the program specifications.

17. *What is program maintenance?*

Program maintenance is reviewing and revising a program according to the changing needs and demands of the user.

18. *What is a prototype?*

A prototype is a rough model of a program that can be used in experiments with alternate specifications for the final design. Prototypes are usually made from pre-existing program blocks that are loosely interfaced for temporary purposes only.

19. *Name at least six phases of the software life cycle and state what each is.*

Ten phases of the software life cycle are listed in Section 1.6.1.

20. *Define software engineering.*

Software engineering is the branch of computer science concerned with techniques of production and maintenance of large software systems. (Many other definitions are possible.)

21. *What are requirements specifications for a program?*

Requirements specifications are the actual requirements given the programmer by the user concerning what the program should do, what commands will be available to the user, the form taken by input and output, documentation requirements, and possible future maintenance changes to the program.

Introduction to Stacks 2

2.1 STACK SPECIFICATIONS

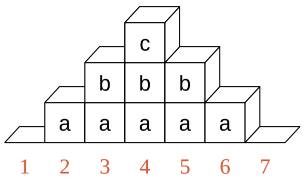
Exercises 2.1

E1. *Draw a sequence of stack frames like Figure 2.2 showing the progress of each of the following segments of code, each beginning with an empty* stack s. *Assume the declarations*

```
#include <stack>
stack<char> s;
char x, y, z;
```

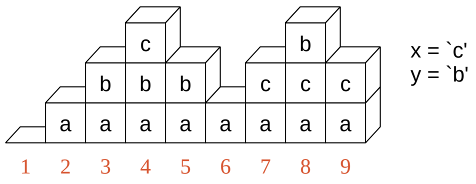
(a)
```
s.push('a');
s.push('b');
s.push('c');
s.pop();
s.pop();
s.pop();
```

Answer



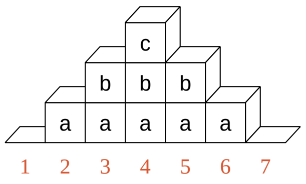
(b)
```
s.push('a');
s.push('b');
s.push('c');
x = s.top();
s.pop();
y = s.top();
s.pop();
s.push(x);
s.push(y);
s.pop();
```

Answer



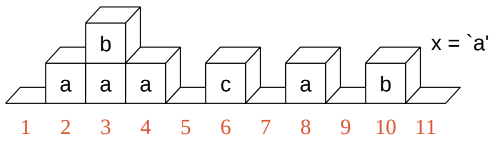
(c)
```
s.push('a');
s.push('b');
s.push('c');
while (!s.empty())
   s.pop();
```

Answer



(d)
```
s.push('a');
s.push('b');
while (!s.empty()) {
   x = s.top();
   s.pop();
}
s.push('c');
s.pop();
s.push('a');
s.pop();
s.push('b');
s.pop();
```

Answer



E2. *Write a program that makes use of a stack to read in a single line of text and write out the characters in the line in reverse order.*

Answer
```
#include <iostream>
#include <stack>
main()
/* Pre:   A line of text is entered by the user.
   Post:  The text is printed in reverse. */
{
   cout << "Enter a line of text to print in reverse." << endl;
   char input;
   stack<char> line;
   while ((input = cin.get()) != '\n') {
      line.push(input);
   }
   while (!line.empty()) {
      cout << line.top();
      line.pop();
   }
   cout << endl;
}
```

E3. *Write a program that reads a sequence of integers of increasing size and prints the integers in decreasing order of size. Input terminates as soon as an integer that does not exceed its predecessor is read. The integers are then printed in decreasing order.*

Answer
```
#include <iostream>
#include <stack>
main()
/* Pre:   An increasing sequence of integers is entered, input is terminated by a final integer that
          is smaller than its predecessor.
   Post:  The sequence is printed in decreasing order. */
{
   cout << "Enter an increasing sequence of integers." << endl;
   cout << "Terminate input with an integer smaller than its predecessor."
        << endl;
   stack<int> sequence;
   int input;
   cin >> input;
   sequence.push(input);
   while (1) {
      cin >> input;
      if (input >= sequence.top()) sequence.push(input);
      else break;
   }
   cout << "In decreasing order, the sequence is:" << endl;
   while (!sequence.empty()) {
      cout << sequence.top() << endl;
      sequence.pop();
   }
}
```

E4. *A stack may be regarded as a railway switching network like the one in Figure 2.3. Cars numbered 1, 2, ..., n are on the line at the left, and it is desired to rearrange (permute) the cars as they leave on the right-hand track. A car that is on the spur (stack) can be left there or sent on its way down the right track, but it can never be sent back to the incoming track. For example, if $n = 3$, and we have the cars 1, 2, 3 on the left track, then 3 first goes to the spur. We could then send 2 to the spur, then on its way to the right, then send 3 on the way, then 1, obtaining the new order 1, 3, 2.*

stack permutations

(a) *For $n = 3$, find all possible permutations that can be obtained.*

Answer There are five permutations:

1 2 3   1 3 2   2 1 3   3 2 1   3 1 2.

(b) *For $n = 4$, find all possible permutations that can be obtained.*

Answer There are 14 such permutations:

1 2 3 4	1 2 4 3	1 3 2 4	1 4 2 3	1 4 3 2	2 1 3 4	2 1 4 3
3 1 2 4	3 2 1 4	4 1 2 3	4 1 3 2	4 2 1 3	4 3 1 2	4 3 2 1

(c) *[Challenging] For general n, find how many permutations can be obtained by using this stack.*

Answer Appendix A, Corollary A.10, shows that the number of permutations of n objects obtainable by a stack is the Catalan number

$$\frac{1}{n+1}\binom{2n}{n}.$$

2.2 IMPLEMENTATION OF STACKS

Exercises 2.2

E1. *Assume the following definition file for a contiguous implementation of an extended stack data structure.*

```
class Extended_stack {
public:
   Extended_stack();
   Error_code pop();
   Error_code push(const Stack_entry &item);
   Error_code top(Stack_entry &item) const;
   bool empty() const;
   void clear();              //  Reset the stack to be empty.
   bool full() const ;        //  If the stack is full, return true; else return false.
   int size() const;          //  Return the number of entries in the stack.
private:
   int count;
   Stack_entry entry[maxstack];
};
```

Write code for the following methods. [Use the private data members in your code.]

(a) clear

Answer

```
void Extended_stack :: clear( )
/* Pre:   None.
   Post:  If the Extended_stack is not empty, all entries are removed. */
{
   count = 0;
}
```

(b) full

Answer

```
bool Extended_stack :: full( ) const
/* Pre:   None.
   Post:  If the Extended_stack is full, true is returned. Otherwise false is returned. */
{
   return count >= maxstack;
}
```

(c) size

Answer

```
int Extended_stack :: size( ) const
/* Pre:   None.
   Post:  Returns the number of entries in the Extended_stack */
{
   return count;
}
```

E2. *Start with the stack methods, and write a function* copy_stack *with the following specifications:*

Error_code copy_stack(Stack &dest, Stack &source);

precondition: None.

postcondition: Stack dest has become an exact copy of Stack source; source is unchanged. If an error is detected, an appropriate code is returned; otherwise, a code of success is returned.

Write three versions of your function:

(a) *Simply use an assignment statement:* dest = source;

Answer

```
Error_code copy_stack(Stack &dest, Stack &source)
/* Pre:   None.
   Post:  Stack dest has become an exact copy of Stack source; source is unchanged. If an error is
          detected, an appropriate code is returned; otherwise, a code of success is returned. */
{
   dest = source;
   return success;
}
```

(b) *Use the* Stack *methods and a temporary* Stack *to retrieve entries from the* Stack source *and add each entry to the* Stack dest *and restore the* Stack source.

Answer
```
Error_code copy_stack(Stack &dest, Stack &source)
/* Pre:  None.
   Post: Stack dest has become an exact copy of Stack source; source is unchanged. If an error is
         detected, an appropriate code is returned; otherwise, a code of success is returned. */
{
  Error_code detected = success;
  Stack temp;
  Stack_entry item;
  while (detected == success && !source.empty()) {
    detected = source.top(item);
    detected = source.pop();
    if (detected == success) detected = temp.push(item);
  }
  while (detected == success && !temp.empty()) {
    detected = temp.top(item);
    detected = temp.pop();
    if (detected == success) detected = source.push(item);
    if (detected == success) detected = dest.push(item);
  }
  return detected;
}
```

(c) *Write the function as a friend*[1] *to the* **class** Stack. *Use the private data members of the* Stack *and write a loop that copies entries from* source *to* dest.

Answer
```
Error_code copy_stack(Stack &dest, Stack &source)
/* Pre:  None.
   Post: Stack dest has become an exact copy of Stack source; source is unchanged. If an error is
         detected, an appropriate code is returned; otherwise, a code of success is returned. */
{
   dest.count = source.count;
   for (int i = 0; i < source.count; i++)
      dest.entry[i] = source.entry[i];
   return success;
}
```

Which of these is easiest to write? Which will run most quickly if the stack is nearly full? Which will run most quickly if the stack is nearly empty? Which would be the best method if the implementation might be changed? In which could we pass the parameter source *as a constant reference?*

Answer The first version is certainly the easiest to write, and it will run the fastest if the stack is nearly full, since the compiler will generate machine code to copy all the entries more quickly than the loop of the third method. If the stack is closer to empty, however, then the third method will be the fastest, since it only copies the occupied entries, whereas the first method copies all positions in the array, occupied or not. Because of all the function calls, the second version will probably be the slowest to run, but it may beat the first version if the stack is nearly empty.

The second version is independent of implementation and hence is best if the implementation may be changed. The first method will fail with a linked implementation, since it will then only set the variables source and dest to point to the same node. Hence no new copy of the stack will be made; source and dest become the *same* stack and changing one will change the other.

E3. *Write code for the following functions. [Your code must use* Stack *methods, but you should not make any assumptions about how stacks or their methods are actually implemented. For some functions, you may wish to declare and use a second, temporary* Stack *object.]*

[1] Friend functions have access to all members of a C++ class, even private ones.

(a) *Function* **bool** full(Stack &s) *leaves the* Stack s *unchanged and returns a value of* **true** *or* **false** *according to whether the* Stack s *is or is not full.*

Answer

```
bool full(Stack &s)
/* Pre:  None
   Post: Leaves the Stack s unchanged. Returns a value of true or false according to whether the
         Stack s is or is not full */
{
   Stack_entry test;
   Error_code outcome;
   outcome = s.push(test);
   if (outcome == success) {
     s.pop();
     return false;
   }
   return true;
}
```

(b) *Function* Error_code pop_top(Stack &s, Stack_entry &t) *removes the top entry from the* Stack s *and returns its value as the output parameter* t.

Answer

```
Error_code pop_top(Stack &s, Stack_entry &t)
/* Pre:  None
   Post: The top entry from the Stack s is removed and returned as value of the output parameter
         t. */
{
   if (s.empty()) return underflow;
   s.top(t);
   s.pop();
   return success;
}
```

(c) *Function* **void** clear(Stack &s) *deletes all entries and returns* s *as an empty* Stack.

Answer

```
void clear(Stack &s)
/* Pre:  None
   Post: Deletes all entries and returns s as an empty Stack. */
{
   while (!s.empty())
      s.pop();
}
```

(d) *Function* **int** size(Stack &s) *leaves the* Stack s *unchanged and returns a count of the number of entries in the* Stack.

Answer

```
int size(Stack &s)
/* Pre:  None
   Post: The Stack s is unchanged. A count of the number of entries in Stack s is returned. */
{
   int count = 0;
   Stack temp;
   Stack_entry item;
   while (!s.empty()) {
      s.top(item);
      s.pop();
      temp.push(item);
      count ++;
   }
   while (!temp.empty()) {
      temp.top(item);
      temp.pop();
      s.push(item);
   }
   return count;
}
```

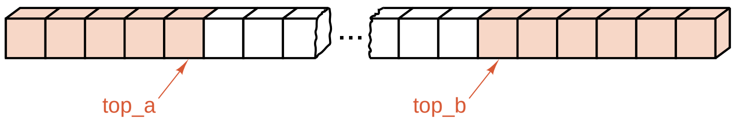
(e) *Function* **void** delete_all(Stack &s, Stack_entry x) *deletes all occurrences (if any) of* x *from* s *and leaves the remaining entries in* s *in the same relative order.*

Answer

```
void delete_all(Stack &s, Stack_entry x)
/* Pre:  None
   Post: All occurrences (if any) of x are removed from s the remaining entries in s remain in the
         same relative order. */
{
   Stack temp;
   Stack_entry item;
   while (!s.empty()) {
      s.top(item);
      s.pop();
      if (item != x) temp.push(item);
   }
   while (!temp.empty()) {
      temp.top(item);
      temp.pop();
      s.push(item);
   }
}
```

E4. *Sometimes a program requires two stacks containing the same type of entries. If the two stacks are stored in separate arrays, then one stack might overflow while there was considerable unused space in the other. A neat way to avoid this problem is to put all the space in one array and let one stack grow from one end of the array and the other stack start at the other end and grow in the opposite direction, toward the first stack. In this way, if one stack turns out to be large and the other small, then they will still both fit, and there will be no overflow until all the space is actually used. Define a new class* Double_stack *that includes (as private data members) the array and the two indices* top_a *and* top_b*, and write code for the methods* Double_stack(), push_a(), push_b(), pop_a(), *and* pop_b() *to handle the two stacks within one* Double_stack.

two coexisting stacks



Answer The class definition is as follows.

```
const int maxstack = 10;          //   small value for testing

class Double_stack {
public:
   Double_stack();
   bool empty_a() const;
   bool empty_b() const;
   bool full() const;             //   Same method checks both stacks for fullness.
   Error_code pop_a();
   Error_code pop_b();
   Error_code stacktop_a(Stack_entry &item) const;
   Error_code stacktop_b(Stack_entry &item) const;
   Error_code push_a(const Stack_entry &item);
   Error_code push_b(const Stack_entry &item);
private:
   int top_a;                                   //  index of top of stack a; −1 if empty
   int top_b;                                   //  index of top of stack b; maxstack if empty
   Stack_entry entry[maxstack];
};
```

The new method implementations follow.

```
Double_stack :: Double_stack( )
/* Pre:   None.
   Post:  The Double_stack is initialized to be empty. */
{
   top_a = −1;
   top_b = maxstack;
}

Error_code Double_stack :: push_a(const Stack_entry &item)
/* Pre:   None.
   Post:  If the Double_stack is not full, item is added to the top of the a-stack of Double_stack.
          If the Double_stack is full, an Error_code of overflow is returned and the Double_stack
          is left unchanged. */
{
   if (top_a >= top_b − 1) return overflow;
   entry[++top_a] = item;
   return success;
}

Error_code Double_stack :: push_b(const Stack_entry &item)
/* Pre:   None.
   Post:  If the Double_stack is not full, item is added to the top of the b-stack of Double_stack.
          If the Double_stack is full, an Error_code of overflow is returned and the Double_stack
          is left unchanged. */
{
   if (top_a >= top_b − 1) return overflow;
   entry[−−top_b] = item;
   return success;
}

Error_code Double_stack :: stacktop_a(Stack_entry &item) const
/* Pre:   None.
   Post:  If the a-stack of the Double_stack is not empty, its top is returned in item. Otherwise
          an Error_code of underflow is returned. */
{
   if (top_a ==  −1) return underflow;
   item = entry[top_a];
   return success;
}
Error_code Double_stack :: stacktop_b(Stack_entry &item) const
/* Pre:   None.
   Post:  If the b-stack of the Double_stack is not empty, its top is returned in item. Otherwise
          an Error_code of underflow is returned. */
{
   if (top_b ==  maxstack) return underflow;
   item = entry[top_b];
   return success;
}
```

```
bool Double_stack :: full( ) const
/* Pre:  None.
   Post: If the Double_stack is full, true is returned. Otherwise false is returned. */
{
   return top_a >= top_b − 1;
}

bool Double_stack :: empty_a( ) const
/* Pre:  None.
   Post: If the a-stack of the Double_stack is empty, true is returned. Otherwise false is re-
         turned. */
{
   return top_a <= −1;
}
bool Double_stack :: empty_b( ) const
/* Pre:  None.
   Post: If the b-stack of the Double_stack is empty, true is returned. Otherwise false is re-
         turned. */
{
   return top_b >= maxstack;
}

Error_code Double_stack :: pop_a( )
/* Pre:  None.
   Post: If the a-stack of the Double_stack is not empty, its top is removed. Otherwise an Er-
         ror_code of underflow is returned. */
{
   if (top_a <= −1) return underflow;
   else −−top_a;
   return success;
}
Error_code Double_stack :: pop_b( )
/* Pre:  None.
   Post: If the b-stack of the Double_stack is not empty, its top is removed. Otherwise an Er-
         ror_code of underflow is returned. */
{
   if (top_b >= maxstack) return underflow;
   else ++top_b;
   return success;
}
```

TOC
Index
Help

Programming Projects 2.2

P1. *Assemble the appropriate declarations from the text into the files* `stack.h` *and* `stack.c` *and verify that* `stack.c` *compiles correctly, so that the* **class** Stack *can be used by future client programs.*

Answer

```
#include   "../../c/utility.h"

typedef float Stack_entry;  //  Use stacks with float entries.
#include   "stack.h"
#include   "../../c/utility.cpp"
#include   "stack.cpp"

main()
/*
Pre:  The user supplies an integer n and n floating point numbers.
Post: The floating point numbers are printed in reverse order.
Uses: The class Stack and its methods
*/
```

```
{
   int n;
   float item;
   Stack numbers;

   cout << " Type in an integer n followed by n floating point numbers.\n"
        << " The numbers will be printed in reverse order." << endl;
   cin  >> n;

   for (int i = 0; i < n; i++) {
      cin >> item;
      if (numbers.push(item) == overflow)
         cout << "\nThe stack is full." << endl;
   }

   cout << "\n\n";
   while (!numbers.empty()) {
      numbers.top(item);
      numbers.pop();
      cout << item << " ";
   }
   cout << endl;
}

const int maxstack = 10;   //  small value for testing

class Stack {
public:
   Stack();
   bool empty() const;
   Error_code pop();
   Error_code top(Stack_entry &item) const;
   Error_code push(const Stack_entry &item);

private:
   int count;
   Stack_entry entry[maxstack];
};


Stack::Stack()
/*
Pre:  None.
Post: The stack is initialized to be empty.
*/

{
   count = 0;
}

Error_code Stack::push(const Stack_entry &item)
/*
Pre:  None.
Post: If the Stack is not full, item is added to the top of the Stack.
      If the Stack is full, an Error_code of overflow is returned and
      the Stack is left unchanged.
*/
```

```
{
   Error_code outcome = success;
   if (count >= maxstack)
      outcome = overflow;
   else
      entry[count++] = item;
   return outcome;
}

Error_code Stack::top(Stack_entry &item) const
/*
Pre:  None.
Post: If the Stack is not empty, the top of
      the Stack is returned in item.  If the Stack
      is empty an Error_code of underflow is returned.
*/

{
   Error_code outcome = success;
   if (count == 0)
      outcome = underflow;
   else
      item = entry[count - 1];
   return outcome;
}

bool Stack::empty() const
/*
Pre:  None.
Post: If the Stack is empty, true is returned.  Otherwise false is returned.
*/

{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}

Error_code Stack::pop()
/*
Pre:  None.
Post: If the Stack is not empty, the top of the Stack is removed.
      If the Stack is empty, an Error_code of underflow is returned.
*/

{
   Error_code outcome = success;
   if (count == 0)
      outcome = underflow;
   else --count;
   return outcome;
}
```

P2. *Write a program that uses a* Stack *to read an integer and print all its prime divisors in descending order. For example, with the integer 2100 the output should be*

prime divisors

7 5 5 3 2 2.

[Hint: The smallest divisor greater than 1 of any integer is guaranteed to be a prime.]

Answer

```
#include   "../../c/utility.h"
typedef int Stack_entry;  //  The program will use stacks with int entries.
#include   "../stack/stack.h"
#include   "../../c/utility.cpp"
#include   "../stack/stack.cpp"

main()
/*
Pre:  The user supplies a positive integer n.
Post: The prime factors of n are printed in decreasing order.
Uses: The class Stack and its methods
*/

{
   int n, original;
   Stack numbers;

   cout << " Type in a positive integer n, whose prime factors\n"
        << " will be printed in decreasing order." << endl;
   cin  >> n;
   original = n;

   int trial_divisor = 2;
   while (n > 1) {
      if (n % trial_divisor)
         trial_divisor++;
      else {
         n /= trial_divisor;
         if (numbers.push(trial_divisor) != success)
            cout << "Warning:  Stack overflow occured " << endl;
      }
   }

   cout << "\nThe prime factors of " << original << " are\n";
   while (!numbers.empty()) {
      int factor;
      numbers.top(factor);
      numbers.pop();
      cout << factor << " ";
   }
   cout << endl;
}
```

2.3 APPLICATION: A DESK CALCULATOR

Exercises 2.3

E1. *If we use the standard library* **class** stack *in our calculator, the method* top() *returns the top entry off the stack as its result. Then the function* do_command *can then be shortened considerably by writing such statements as*

case ′ − ′: numbers.push(numbers.pop() − numbers.pop());

(a) *Assuming that this statement works correctly, explain why it would still be bad programming style.*

Answer The function pop has the side effect of changing the stack, and such side effects can be dangerous. Readability is an important part of good programming style, and writing code such as that displayed above reduces readability and clarity of the function.

(b) *It is possible that two different C++ compilers, both adhering strictly to standard C++, would translate this statement in ways that would give different answers when the program runs. Explain how this could happen.*

Answer Two different C++ compilers could translate the given statement so that either of the two actions numbers.pop() might be done first. One compiler might evaluate the expression as $a - b$, while another might evaluate it as $b - a$.

E2. *Discuss the steps that would be needed to make the calculator process complex numbers.*

Answer It would be necessary to create a **class** Complex to store complex number values and implement the arithmetic operations (addition, subtraction, multiplication, and division) for complex numbers. If these operations are implemented by overloading the standard operator symbols (+, -, *, and /) then the only other change needed is the replacement of the type **double** by Complex wherever it appears in the calculator program.

Programming Projects 2.3

P1. *Assemble the functions developed in this section and make the necessary changes in the code so as to produce a working calculator program.*

Answer The following calculator program makes use of the preceding stack implementation.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

typedef double Stack_entry;
#include "../stack/stack.h"

#include "../stack/stack.cpp"
#include "commands.cpp"

int main()
/*
Post: The program has executed simple arithmetic commands
      entered by the user.
Uses: The class Stack and the functions
introduction, instructions, do_command, and get_command.
*/

{
   Stack stored_numbers;
   introduction();
   instructions();
   while (do_command(get_command(), stored_numbers));
}


void introduction()
{
   cout << "Reverse Polish Calculator Program."
        << "\n----------------------------------"
        << endl << endl;
}
```

```
void instructions()
{
    cout << "User commands are entered to read in and operate on integers."
         << endl;
    cout << "The valid commands are as follows:"   << endl
         << "[Q]uit." << endl
         << "[?] to enter an integer onto a stack."  << endl
         << "[=] to print the top integer in the stack." << endl
         << "[+] [-] [*] [/] are arithmetic operations." << endl
         << "These operations apply to the top pair of stacked integers."
         << endl;
}

char get_command()
{
   char command;
   bool waiting = true;
   cout << "Select command and press <Enter>:";

   while (waiting) {
      cin >> command;
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == '*' || command == '/' ||
          command == 'q' ) waiting = false;


      else {
         cout << "Please enter a valid command:"   << endl
              << "[?]push to stack   [=]print top" << endl
              << "[+] [-] [*] [/]   are arithmetic operations" << endl
              << "[Q]uit." << endl;
      }
   }
   return command;
}

bool do_command(char command, Stack &numbers)
/*
Pre:  The first parameter specifies a valid calculator command.
Post: The command specified by the first parameter has been applied
      to the Stack of numbers given by the second parameter.
      A result of true is returned unless command == 'q'.
Uses: The class Stack.
*/

{
   double p, q;
   switch (command) {
   case '?':
      cout << "Enter a real number: " << flush;
      cin >> p;
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost number" << endl;
      break;
   case '=':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else
         cout << p << endl;
      break;

   case '+':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }

         else {
            numbers.pop();
            if (numbers.push(q + p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

//   Add options for further user commands.

   case '-':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q - p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   case '*':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q * p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;
   case '/':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else if (p == 0.0) {
          cerr << "Division by 0 fails; no action done." << endl;
          numbers.push(p);     //  Restore stack to its prior state.
      }
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q / p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   case 'q':
      cout << "Calculation finished.\n";
      return false;
   }
   return true;
}
```

P2. *Write a function that will interchange the top two numbers on the stack, and include this capability as a new command.*

Answer See the solution to project P4.

P3. *Write a function that will add all the numbers on the stack together, and include this capability as a new command.*

Answer See the solution to project P4.

P4. *Write a function that will compute the average of all numbers on the stack, and include this capability as a new command.*

Answer The following program includes the features specified for projects P2, P3, and P4.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

typedef double Stack_entry;
#include "../stack/stack.h"

#include "../stack/stack.cpp"
#include "auxil.cpp"
#include "commands.cpp"

int main()
/*
Post: The program has executed simple arithmetic
      commands entered by the user.
Uses: The class Stack and the functions
      introduction, instructions, do_command, and get_command.
*/
```

```
{
   Stack stored_numbers;
   introduction();
   instructions();
   while (do_command(get_command(), stored_numbers));
}


void introduction()
{
   cout << "This program implements a reverse Polish calulator"
        << endl;
}

void instructions()
{
   cout << "The valid calculator commands are:"   << endl
        << "[?]push to stack   [=]print top" << endl
        << "[+] [-] [*] [/]   are arithmetic operations" << endl
        << "e[X]change the top two stack entries" << endl
        << "adjoin the [S]um of the stack" << endl
        << "adjoin the [M]ean of the stack" << endl
        << "[Q]uit." << endl;
}

char get_command()
{
   char command;
   bool waiting = true;
   cout << "Select command and press <Enter>:";

   while (waiting) {
      cin >> command;
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == '*' || command == '/' ||
          command == 'x' || command == 's' || command == 'm' ||
          command == 'q' ) waiting = false;

      else {
         cout << "Please enter a valid command:"   << endl
              << "[?]push to stack   [=]print top" << endl
              << "[+] [-] [*] [/]   are arithmetic operations" << endl
              << "e[X]change the top two stack entries" << endl
              << "adjoin the [S]um of the stack" << endl
              << "adjoin the [M]ean of the stack" << endl
              << "[Q]uit." << endl;
      }
   }
   return command;
}

bool do_command(char command, Stack &numbers)
/*
Pre:  The first parameter specifies a valid calculator command.
Post: The command specified by the first parameter has been applied
      to the Stack of numbers given by the second parameter.
      A result of true is returned unless command == 'q'.
Uses: The class Stack.
*/
```

```
{
   double p, q;
   int counter;
   switch (command) {
   case '?':
      cout << "Enter a real number: " << flush;
      cin >> p;
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost number" << endl;
      break;

   case 'm':
      p = sum_stack(numbers);
      counter = count_stack(numbers);
      if (counter <= 0)
         cout << "Warning: Stack empty, it has no mean" << endl;
      else if (numbers.push(p / ((double) counter)) == overflow)
         cout << "Warning: Stack full, lost result" << endl;
      break;

   case 's':
      p = sum_stack(numbers);
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost result" << endl;
      break;

   case '=':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else
         cout << p << endl;
      break;

   case 'x':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }

         else {
            numbers.pop();
            if (numbers.push(p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
            if (numbers.push(q) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;
   case '+':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }

         else {
            numbers.pop();
            if (numbers.push(q + p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

//   Add options for further user commands.

   case '-':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q - p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   case '*':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q * p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;
   case '/':
      if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
      else if (p == 0.0) {
          cerr << "Division by 0 fails; no action done." << endl;
          numbers.push(p);     //  Restore stack to its prior state.
      }
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }
         else {
            numbers.pop();
            if (numbers.push(q / p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   case 'q':
      cout << "Calculation finished.\n";
      return false;
   }
   return true;
}


double sum_stack(Stack &numbers)
/*
Post: The sum of the entries in a Stack is returned,
      and the Stack is restored to its original state.
Uses: The class Stack.
*/

{
   double p, sum = 0.0;
   Stack temp_copy;
   while (!numbers.empty()) {
       numbers.top(p);
       temp_copy.push(p);
       sum += p;
       numbers.pop();
   }

   while (!temp_copy.empty()) {
       temp_copy.top(p);
       numbers.push(p);
       temp_copy.pop();
   }
   return sum;
}

int count_stack(Stack &numbers)
/*
Post: The number of the entries in a Stack is returned,
      and the Stack is restored to its original state.
Uses: The class Stack.
*/
```

```
{
   double p;
   int counter = 0;
   Stack temp_copy;
   while (!numbers.empty()) {
       numbers.top(p);
       temp_copy.push(p);
       counter++;
       numbers.pop();
   }

   while (!temp_copy.empty()) {
       temp_copy.top(p);
       numbers.push(p);
       temp_copy.pop();
   }
   return counter;
}
```

2.4 APPLICATION: BRACKET MATCHING

Programming Projects 2.4

P1. *Modify the bracket checking program so that it reads the whole of an input file.*

Answer See the solution to project P3 below.

P2. *Modify the bracket checking program so that input characters are echoed to output, and individual unmatched closing brackets are identified in the output file.*

Answer See the solution to project P3 below.

P3. *Incorporate C++ comments and character strings into the bracket checking program, so that any bracket within a comment or character string is ignored.*

Answer Note: In order to bracket-check an actual C++ program, we must also allow for character constants, such as ‘"’ and for other sorts of brackets such as the template brackets < … > .

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

typedef char Stack_entry;
#include "../stack/stack.h"
#include "../stack/stack.cpp"
bool echo_input = true;
#include "auxil.cpp"

int main()
/*
Post: The program has notified the user of any bracket
mismatch in the standard input file.
Uses: The class Stack.
*/

{
   introduction();
   instructions();
```

TOC
Index
Help

```
Stack openings;
char symbol;
bool is_matched = true;
int line_len = 0;
bool short_comment = false;
bool long_comment = false;
bool string = false;
char last_symbol = ' ';

while (is_matched && !cin.eof()) {
   symbol = cin.get();

   if (symbol == '\n') {
      line_len = 0;
      if (short_comment) short_comment = false;
   }
   else line_len++;

   if (symbol == '/' && !short_comment && !string) {
      if (long_comment) {
         if (last_symbol == '*') long_comment = false;
      }
      else if (last_symbol == '/') short_comment = true;
   }

   if (symbol == '*' && last_symbol == '/' && !short_comment && !string)
      long_comment = true;

   if (!short_comment && !long_comment && !string && symbol == '"')
      string = true;
   else if (!short_comment && !long_comment &&
              string && symbol == '"' && last_symbol != '\\')
      string = false;

   last_symbol = symbol;

   if (symbol != EOF && echo_input) cout << symbol;

   if (!string && !short_comment && !long_comment) {
      if (symbol == '{' || symbol == '(' || symbol == '[')
         openings.push(symbol);

      if (symbol == '}' || symbol == ')' || symbol == ']') {
         if (openings.empty()) {
            if (echo_input) cout << "  <";
            else {
               for (int i = 1; i < line_len; i++) cout << " ";
               cout << "^" << endl;
               for (int j = 1; j < line_len; j++) cout << " ";
               cout << "|";
            }
            cout << "------ Error " << endl;
            cout << "Unmatched closing bracket " << symbol
                 << " detected." << endl;
            is_matched = false;
         }
```

```
                    else {
                       char match;
                       openings.top(match);
                       openings.pop();
                       is_matched = (symbol == '}' && match == '{')
                                    || (symbol == ')' && match == '(')
                                    || (symbol == ']' && match == '[');
                       if (!is_matched) {
                          if (echo_input) cout << "  <";
                          else {
                             for (int i = 1; i < line_len; i++) cout << " ";
                             cout << "^" << endl;
                             for (int j = 1; j < line_len; j++) cout << " ";
                             cout << "|";
                          }
                          cout << "------ Error " << endl;
                          cout << "Bad match " << match << symbol << endl;
                       }
                    }
                 }
              }
           }
           if (!openings.empty())
              cout << "Unmatched opening bracket(s) detected." << endl;
        }


        void introduction()
        {
           cout << "This program implements a bracket checking program"
                << endl;
        }

        void instructions()
        {
           cout << "Enter text for checking." << endl
                << "End input with an EOF (a Ctrl-D character)." << endl
                << "Input will be echoed with bracket mismatches flagged."
                << endl;

           cout << "Do you want to turn off the input echo? ";
           echo_input = !user_says_yes();
        }
```

2.5 ABSTRACT DATA TYPES AND THEIR IMPLEMENTATIONS

Exercises 2.5

E1. *Give a formal definition of the term extended stack as used in Exercise E1 of Section 2.2.*

Answer

Definition An Extended_stack of elements of type T is a finite sequence of elements of T together with the operations

1. *Construct* the extended stack, leaving it empty.
2. Determine whether the extended stack is *empty* or not.
3. Determine whether the extended stack is *full* or not.
4. Find the *size* of the extended stack.
5. *Push* a new entry on top of the extended stack, provided the extended stack is not full.
6. Retrieve the *top* entry in the extended stack, provided the extended stack is not empty.
7. *Pop* (and remove) the entry from the top of the extended stack , provided the extended stack is not empty.
8. *Clear* the extended stack to make it empty.

E2. *In mathematics the* ***Cartesian product*** *of sets* $T_1, T_2, \dots, T_n$ *is defined as the set of all* n*-tuples* $(t_1, t_2, \dots, t_n)$, *where* t_i *is a member of* T_i *for all* $i, 1 \le i \le n$. *Use the Cartesian product to give a precise definition of a class.*

Answer A ***class*** consists of members, t_i of type T_i for $i = 1, 2, \dots, n$ and is the Cartesian product of the corresponding sets $T_1 \times T_2 \times \dots \times T_n$.

REVIEW QUESTIONS

1. *What is the standard library?*

   The standard library is available in implementations of ANSI C++. It provides system-dependent information, such as the maximum exponent that can be stored in a floating-point type and input and output facilities. In addition, the standard library provides an extensive set of data structures and method implementations.

2. *What are the methods of a stack?*

   push pop top empty initialization

3. *What are the advantages of writing the operations on a data structure as methods?*

   The advantages of writing the operations on a data structure as methods are that more complicated functions can be performed by combining these methods; the programmer does not have to worry about the details involved in the methods already written; and the programming style becomes clearer.

4. *What are the differences between information hiding and encapsulation?*

   Information hiding is the programming strategy of designing classes and functions so that their data and implementations are not needed by client code. Encapsulation goes further by making the data and implementations of classes completely unavailable to client code.

5. *Describe three different approaches to error handling that could be adopted by a C++ class.*

   Errors could be reported by returning an error code to clients. Alternatively, errors could be signalled by throwing exceptions for clients to catch. Finally, a class could respond to errors by printing warning messages or terminating program execution.

6. *Give two different ways of implementing a generic data structure in C++.*

In C++, generic data structures can be implemented either as templates or in terms of an unspecified data type that is selected by client code with a **typedef** statement.

7. *What is the reason for using the reverse Polish convention for calculators?*

The advantage of a reverse Polish calculator is that any expression, no matter how complicated, can be specified without the use of parentheses.

8. *What two parts must be in the definition of any abstract data type?*

The two parts are :

1. A description of how components are related to one another.
2. A statement of the methods that can be performed on the elements of the abstract data type.

9. *In an abstract data type, how much is specified about implementation?*

An abstract data type specifies nothing concerning the actual implementation of the structure.

10. *Name (in order from abstract to concrete) four levels of refinement of data specification.*

The four levels are :

1. The *abstract* level.
2. The *data structure* level.
3. The *implementation* level.
4. The *application* level.

Queues 3

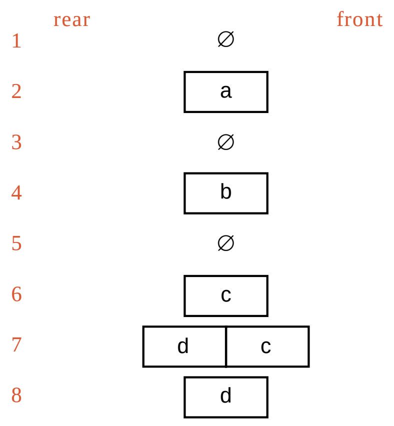
3.1 DEFINITIONS

Exercises 3.1

E1. *Suppose that* q *is a* Queue *that holds characters and that* x *and* y *are character variables. Show the contents of* q *at each step of the following code segments.*

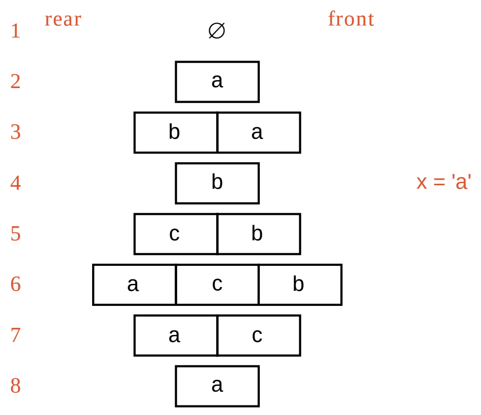
(a)
```
Queue q;
q.append('a');
q.serve();
q.append('b');
q.serve();
q.append('c');
q.append('d');
q.serve();
```

Answer



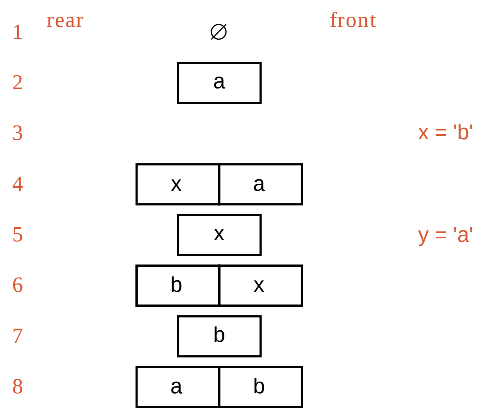
(b)
```
Queue q;
q.append('a');
q.append('b');
q.retrieve(x);
q.serve();
q.append('c');
q.append(x);
q.serve();
q.serve();
```

Answer



(c)
```
Queue q;
q.append('a');
x = 'b';
q.append('x');
q.retrieve(y);
q.serve();
q.append(x);
q.serve();
q.append(y);
```

Answer



E2. *Suppose that you are a financier and purchase 100 shares of stock in Company X in each of January, April, and September and sell 100 shares in each of June and November. The prices per share in these months were*

accounting

Jan	*Apr*	*Jun*	*Sep*	*Nov*
$10	*$30*	*$20*	*$50*	*$30*

Determine the total amount of your capital gain or loss using **(a)** *FIFO (first-in, first-out) accounting and* **(b)** *LIFO (last-in, first-out) accounting [that is, assuming that you keep your stock certificates in* **(a)** *a queue or* **(b)** *a stack]. The 100 shares you still own at the end of the year do not enter the calculation.*

Answer Purchases and sales:

January:	purchase	$100 \times \$10 = \$1000$
April:	purchase	$100 \times \$30 = \$3000$
June:	sell	$100 \times \$20 = \$2000$
September:	purchase	$100 \times \$50 = \$5000$
November:	sell	$100 \times \$30 = \$3000$

With FIFO accounting, the 100 shares sold in June were those purchased in January, so the profit is $2000 − $1000 = $1000. Similarly, the November sales were the April purchase, both at $300, so the transaction breaks even. *Total profit* = $1000. With LIFO accounting, June sales were April purchases giving a loss of $1000 ($2000 − $3000), and November sales were September purchases giving a loss of $2000 ($3000 − $5000). *Total loss* = $3000.

E3. *Use the methods for stacks and queues developed in the text to write functions that will do each of the following tasks. In writing each function, be sure to check for empty and full structures as appropriate. Your functions may declare other, local structures as needed.*

(a) *Move all the entries from a* Stack *into a* Queue.

Answer The following procedures use both stacks and queues. Where both queues and stacks are involved in the solution, entries will be of type Entry, as opposed to Stack_entry or Queue_entry. This can be accomplished by adding the type declarations **typedef** Entry Stack_entry; and **typedef** Entry Queue_entry; to the implementations. The type of Entry is specified by client code in the usual way.

```
Error_code stack_to_queue(Stack &s, Queue &q)
/* Pre:  The Stack s and the Queue q have the same entry type.
   Post: All entries from s have been moved to q. If there is not enough room in q to hold all
         entries in s return a code of overflow, otherwise return success. */
```

```
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = q.append(item);
      if (outcome == success) s.pop();
   }
   return (outcome);
}
```

(b) *Move all the entries from a* Queue *onto a* Stack.

Answer

```
Error_code queue_to_stack(Stack &s, Queue &q)
/* Pre:  The Stack s and the Queue q have the same entry type.
   Post: All entries from q have been moved to s. If there is not enough room in s to hold all
         entries in q return a code of overflow, otherwise return success. */
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !q.empty()) {
      q.retrieve(item);
      outcome = s.push(item);
      if (outcome == success) q.serve();
   }
   return (outcome);
}
```

(c) *Empty one* Stack *onto the top of another* Stack *in such a way that the entries that were in the first* Stack *keep the same relative order.*

Answer

```
Error_code move_stack(Stack &s, Stack &t)
/* Pre:  None.
   Post: All entries from s have been moved in order onto the top of t. If there is not enough
         room in t to hold these entries return a code of overflow, otherwise return success. */
{
   Error_code outcome = success;
   Entry item;
   Stack temp;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = temp.push(item);
      if (outcome == success) s.pop();
   }
   while (outcome == success && !temp.empty()) {
      temp.top(item);
      outcome = t.push(item);
      if (outcome == success) temp.pop();
   }
   while (!temp.empty()) {                    //  replace any entries to s that can not fit on t
      temp.top(item);
      s.push(item);
   }
   return (outcome);
}
```

TOC
Index
Help

(d) *Empty one* Stack *onto the top of another* Stack *in such a way that the entries that were in the first* Stack *are in the reverse of their original order.*

Answer
```
Error_code reverse_move_stack(Stack &s, Stack &t)
/* Pre:  None.
   Post: All entries from s have been moved in reverse order onto the top of t. If there is not
         enough room in t to hold these entries return a code of overflow, otherwise return
         success. */
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = t.push(item);
      if (outcome == success) s.pop();
   }
   return (outcome);
}
```

(e) *Use a local* Stack *to reverse the order of all the entries in a* Queue.

Answer
```
Error_code reverse_queue(Queue &q)
/* Pre:  None.
   Post: All entries from q have been reversed. */
{
   Error_code outcome = success;
   Entry item;
   Stack temp;
   while (outcome == success && !q.empty()) {
      q.retrieve(item);
      outcome = temp.push(item);
      if (outcome == success) q.serve();
   }
   while (!temp.empty()) {
      temp.top(item);
      q.append(item);
      temp.pop();
   }
   return (outcome);
}
```

(f) *Use a local* Queue *to reverse the order of all the entries in a* Stack.

Answer
```
Error_code reverse_stack(Stack &s)
/* Pre:  None.
   Post: All entries from s have been reversed. */
{
   Error_code outcome = success;
   Entry item;
   Queue temp;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = temp.append(item);
      if (outcome == success) s.pop();
   }
```

```
      while (!temp.empty()) {
         temp.retrieve(item);
         s.push(item);
         temp.serve();
      }
      return (outcome);
   }
```

3.3 CIRCULAR IMPLEMENTATION OF QUEUES IN C++

Exercises 3.3

E1. *Write the remaining methods for queues as implemented in this section:*

(a) empty

Answer

```
bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return count == 0;
}
```

(b) retrieve

Answer

```
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

E2. *Write the remaining methods for extended queues as implemented in this section:*

(a) full

Answer

```
bool Extended_queue :: full() const
/* Post: Return true if the Extended_queue is full; return false otherwise. */
{
   return count == maxqueue;
}
```

(b) clear

Answer

```
void Extended_queue :: clear()
/* Post: All entries in the Extended_queue have been deleted; the Extended_queue is empty. */
{
   count = 0;
   front = 0;
   rear = maxqueue − 1;
}
```

(c) serve_and_retrieve

Answer

```
Error_code Extended_queue :: serve_and_retrieve(Queue_entry &item)
/* Post: Return underflow if the Extended_queue is empty. Otherwise remove and copy the
         item at the front of the Extended_queue to item. */
{
   if (count == 0) return underflow;
   else {
      count−−;
      item = entry[front];
      front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   }
   return success;
}
```

E3. *Write the methods needed for the implementation of a queue in a linear array when it can be assumed that the queue can be emptied when necessary.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                  //  small value for testing
class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
{
   rear = −1;
   front = 0;
}
bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear < front;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full, then empty the Queue
         before adding item and return an Error_code of overflow. */
{
   Error_code result = success;
   if (rear == maxqueue − 1) {
      result = overflow;
      rear = −1;
      front = 0;
   }
   entry[++rear] = item;
   return result;
}
```

```
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (rear < front) return underflow;
   front = front + 1;
   return success;
}
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (rear < front) return underflow;
   item = entry[front];
   return success;
}
```

E4. *Write the methods to implement queues by the simple but slow technique of keeping the front of the queue always in the first position of a linear array.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                      //   small value for testing
class Queue {
public:
   Queue( );
   bool empty( ) const;
   Error_code serve( );
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int rear;                                  //   front == 0
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   rear = −1;
}
bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear < 0;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an Error_code of
         overflow. */
{
   if (rear == maxqueue − 1) return overflow;
   entry[++rear] = item;
   return success;
}
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
```

```
{
   if (rear < 0) return underflow;
   for (int i = 0; i < rear; i++)
      entry[i] = entry[i + 1];
   rear−−;
   return success;
}
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (rear < 0) return underflow;
   item = entry[0];
   return success;
}
```

E5. *Write the methods to implement queues in a linear array with two indices* front *and* rear, *such that, when* rear *reaches the end of the array, all the entries are moved to the front of the array.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                    //  small value for testing
class Queue {
public:
   Queue( );
   bool empty( ) const;
   Error_code serve( );
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   rear = −1;
   front = 0;
}
bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear < front;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the rear is at or immediately before the end
         of the Queue, move all entries back to the start of the Queue before appending. If the
         Queue is full return an Error_code of overflow. */
{
   if (rear == maxqueue − 1 || rear == maxqueue − 2)
     for (int i = 0; i <= rear − front; i++) {
        entry[i] = entry[i + front];
        rear = rear − front;
        front = 0;
     }
   if (rear == maxqueue − 1) return overflow;
   entry[++rear] = item;
   return success;
}
Error_code Queue :: serve()
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (rear < front) return underflow;
   front = front + 1;
   return success;
}
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (rear < front) return underflow;
   item = entry[front];
   return success;
}
```

E6. *Write the methods to implement queues, where the implementation does not keep a count of the entries in the queue but instead uses the special conditions*

$$\text{rear} = -1 \qquad \textit{and} \qquad \text{front} = 0$$

to indicate an empty queue.

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                   //   small value for testing

class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
{
   rear = − 1;
   front = 0;
}

bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear == −1;
}
```

TOC
Index
Help

```
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an Error_code of
         overflow and leave the Queue unchanged. */
{
   if (!empty() && (rear + 1) % maxqueue == front) return overflow;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}

Error_code Queue :: serve()
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (empty()) return underflow;
   if (rear == front) {
      rear = −1;
      front = 0;
   }
   else
      front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}

Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (empty()) return underflow;
   item = entry[front];
   return success;
}
```

E7. *Rewrite the methods for queue processing from the text, using a flag to indicate a full queue instead of keeping a count of the entries in the queue.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                    //   small value for testing

class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
   bool is_empty;
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
```

```
{
   rear = −1;
   front = 0;
   is_empty = true;
}

bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return is_empty;
}

Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an Error_code of
         overflow and leave the Queue unchanged. */
{
   if (!empty( ) && (rear + 1) % maxqueue == front) return overflow;
   is_empty = false;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}

Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (empty( )) return underflow;
   if (rear == front) is_empty = true;
   front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}

Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (empty( )) return underflow;
   item = entry[front];
   return success;
}
```

E8. *Write methods to implement queues in a circular array with one unused entry in the array. That is, we consider that the array is full when the rear is two positions before the front; when the rear is one position before, it will always indicate an empty queue.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                    //  small value for testing

class Queue {
public:
   Queue( );
   bool empty( ) const;
   Error_code serve( );
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   rear = −1;
   front = 0;
}

bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return (rear + 1) % maxqueue == front;
}

Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an Error_code of
         overflow and leave the Queue unchanged. */
{
   if ((rear + 2) % maxqueue == front) return overflow;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}

Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (empty( )) return underflow;
   front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}

Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (empty( )) return underflow;
   item = entry[front];
   return success;
}
```

deque *The word **deque** (pronounced either "deck" or "DQ") is a shortened form of **double-ended queue** and denotes a list in which entries can be added or removed from either the first or the last position of the list, but no changes can be made elsewhere in the list. Thus a deque is a generalization of both a stack and a queue. The fundamental operations on a deque are* append_front, append_rear, serve_front, serve_rear, retrieve_front, *and* retrieve_rear.

E9. *Write the class definition and the method implementations needed to implement a deque in a linear array.*

Answer The class definition for a Deque implementation is as follows.

```
const int maxqueue = 10;                    //   small value for testing
```

```
class Deque {
public:
   Deque();
   bool empty() const;
   Error_code serve_front();
   Error_code serve_rear();
   Error_code append_front(const Queue_entry &item);
   Error_code append_rear(const Queue_entry &item);
   Error_code retrieve_front(Queue_entry &item) const;
   Error_code retrieve_rear(Queue_entry &item) const;
protected:
   int rear;                                    //   front stays at 0
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Deque :: Deque()
/* Post: The Deque is initialized to be empty. */
{
   rear = −1;
}
bool Deque :: empty() const
/* Post: Return true if the Deque is empty, otherwise return false. */
{
   return rear < 0;
}
Error_code Deque :: append_front(const Queue_entry &item)
/* Post: item is added to the front of the Deque. If the Deque is full return an Error_code of
         overflow. */
{
   if (rear == maxqueue − 1) return overflow;
   for (int i = rear + 1; i > 0; i−−)
      entry[i] = entry[i − 1];
   rear++;
   entry[0] = item;
   return success;
}
Error_code Deque :: serve_front()
/* Post: The front of the Deque is removed. If the Deque is empty return an Error_code of
         underflow. */
{
   if (rear < 0) return underflow;
   for (int i = 0; i < rear; i++)
      entry[i] = entry[i + 1];
   rear−−;
   return success;
}
Error_code Deque :: retrieve_front(Queue_entry &item) const
/* Post: The front of the Deque retrieved to the output parameter item. If the Deque is empty
         return an Error_code of underflow. */
{
   if (rear < 0) return underflow;
   item = entry[0];
   return success;
}
```

```
Error_code Deque :: append_rear(const Queue_entry &item)
/* Post: item is added to the rear of the Deque. If the Deque is full return an Error_code of
         overflow. */
{
   if (rear == maxqueue − 1) return overflow;
   entry[++rear] = item;
   return success;
}
Error_code Deque :: serve_rear( )
/* Post: The rear of the Deque is removed. If the Deque is empty return an Error_code of
         underflow. */
{
   if (rear < 0) return underflow;
   rear −−;
   return success;
}
Error_code Deque :: retrieve_rear(Queue_entry &item) const
/* Post: The rear of the Deque retrieved to the output parameter item. If the Deque is empty
         return an Error_code of underflow. */
{
   if (rear < 0) return underflow;
   item = entry[rear];
   return success;
}
```

E10. *Write the methods needed to implement a deque in a circular array. Consider the* **class** Deque *as derived from the* **class** Queue. *(Can you hide the* Queue *methods from a client?)*

Answer The class definition for a Deque implementation follows. The use of private inheritance hides the Queue methods from clients.

```
class Deque: private Queue {
public:
   Deque( );
   bool empty( ) const;
   Error_code serve_front( );
   Error_code serve_rear( );
   Error_code append_front(const Queue_entry &item);
   Error_code append_rear(const Queue_entry &item);
   Error_code retrieve_front(Queue_entry &item) const;
   Error_code retrieve_rear(Queue_entry &item) const;
};
```

The method implementations follow.

```
Deque :: Deque( )
/* Post: The Deque is initialized to be empty. */
{
   Queue :: Queue( );
}
bool Deque :: empty( ) const
/* Post: Return true if the Deque is empty, otherwise return false. */
{
   return count <= 0;
}
Error_code Deque :: append_front(const Queue_entry &item)
/* Post: item is added to the front of the Deque. If the Deque is full return an Error_code of
         overflow. */
```

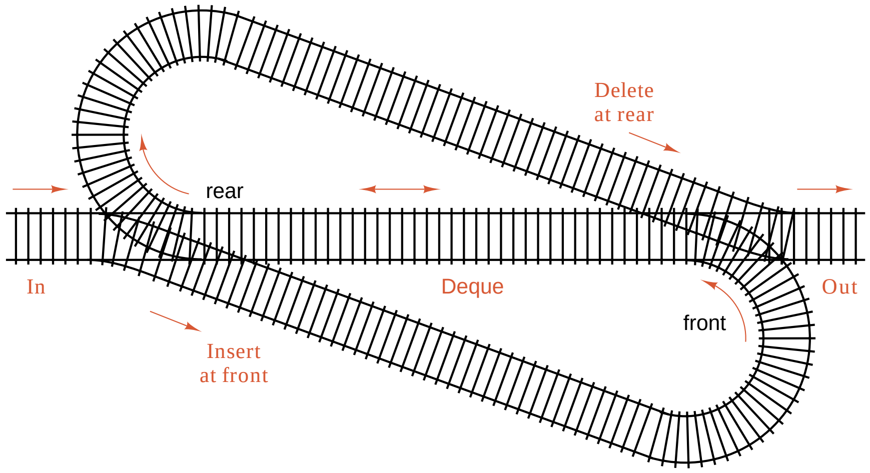
```
{
   if (count >= maxqueue) return overflow;
   front = (front + maxqueue − 1) % maxqueue;
   entry[front] = item;
   count ++;
   return success;
}
Error_code Deque :: serve_front( )
/* Post: The front of the Deque is removed. If the Deque is empty return an Error_code of
         underflow. */
{
   return serve( );
}
Error_code Deque :: retrieve_front(Queue_entry &item) const
/* Post: The front of the Deque retrieved to the output parameter item. If the Deque is empty
         return an Error_code of underflow. */
{
   return retrieve(item);
}
Error_code Deque :: append_rear(const Queue_entry &item)
/* Post: item is added to the rear of the Deque. If the Deque is full return an Error_code of
         overflow. */
{
   return append(item);
}
Error_code Deque :: serve_rear( )
/* Post: The rear of the Deque is removed. If the Deque is empty return an Error_code of
         underflow. */
{
   if (count <= 0) return underflow;
   rear = (rear + maxqueue − 1) % maxqueue;
   count −−;
   return success;
}
Error_code Deque :: retrieve_rear(Queue_entry &item) const
/* Post: The rear of the Deque retrieved to the output parameter item. If the Deque is empty
         return an Error_code of underflow. */
{
   if (count <= 0) return underflow;
   item = entry[rear];
   return success;
}
```

E11. *Is it more appropriate to implement a deque in a linear array or in a circular array? Why?*

Answer As with a queue, it is more appropriate to think of a deque as implemented with a circular array. The deque is subject to the same problem of growing in a direction past the end, or the beginning, of its array without occupying all available positions. The reasons in the text for using a circular array to implement a queue also apply to the implementation of a deque.

E12. *Note from Figure 2.3 that a stack can be represented pictorially as a spur track on a straight railway line. A queue can, of course, be represented simply as a straight track. Devise and draw a railway switching network that will represent a deque. The network should have only one entrance and one exit.*

Answer



E13. *Suppose that data items numbered 1, 2, 3, 4, 5, 6 come in the input stream in this order. That is, 1 comes first, then 2, and so on. By using (1) a queue and (2) a deque, which of the following rearrangements can be obtained in the output order? The entries also leave the deque in left-to-right order.*

(a) *1 2 3 4 5 6*	**(b)** *2 4 3 6 5 1*	**(c)** *1 5 2 4 3 6*
(d) *4 2 1 3 5 6*	**(e)** *1 2 6 4 5 3*	**(f)** *5 2 6 3 4 1*

Answer We use the convention that the items arrive from left to right, that is 1 through 6, and leave from left to right, as listed in (a) through (f):

Output Order	**queue**	**Deque**
(a) 1 2 3 4 5 6	√	√
(b) 2 4 3 6 5 1	X	√
(c) 1 5 2 4 3 6	X	√
(d) 4 2 1 3 5 6	X	√
(e) 1 2 6 4 5 3	X	√
(f) 5 2 6 3 4 1	X	X

The most difficult of these results is that ordering (f) cannot be obtained by a deque. For suppose that it could be obtained. Since 1 goes into the deque at the beginning and stays there until the end, it effectively divides the deque into two stacks for processing the remaining entries. Since 5 is first out, the numbers 2, 3, and 4 are all put into one or the other of these stacks. Since 2 comes out before 3 or 4, it is on top of one of the stacks, but 2 is received first, so it must be on the bottom of its stack. Hence 3 and 4 both go onto the other stack. Since 3 comes in first, 4 must go out first. Hence the order shown, with 3 before 4, cannot be obtained by a deque.

Programming Project 3.3

P1. *Write a function that will read one line of input from the terminal. The input is supposed to consist of two parts separated by a colon ':'. As its result, your function should produce a single character as follows:*

N	*No colon on the line.*
L	*The left part (before the colon) is longer than the right.*
R	*The right part (after the colon) is longer than the left.*
D	*The left and right parts have the same length but are different.*
S	*The left and right parts are exactly the same.*

Examples:

Input	*Output*
Sample Sample	N
Left:Right	R
Sample:Sample	S

Use either a queue or an extended queue to keep track of the left part of the line while reading the right part.

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
typedef char Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"

int main()
/*
Post: Implements Project P1 of Section 3.3
*/

{
   Extended_queue left_queue;
   Extended_queue right_queue;
   cout << "Enter two pieces of text, on a single line, separated by a :"
        << endl;
   char c;
   while ((c = cin.get()) != '\n' && c != ':')
      left_queue.append(c);
   if (c == '\n') cout << "N" << endl;
   else {
      while ((c = cin.get()) != '\n' && c != ':')
         right_queue.append(c);
      if (left_queue.size() > right_queue.size()) cout << "L" << endl;
      else if (left_queue.size() < right_queue.size()) cout << "R" << endl;
      else {
         char l = ' ', r = ' ';
         while (!left_queue.empty()) {
            left_queue.serve_and_retrieve(l);
            right_queue.serve_and_retrieve(r);
            if (l != r) break;
         }
         if (l == r) cout << "S" << endl;
         else cout << "D" << endl;
      }
   }
}
```

3.4 DEMONSTRATION AND TESTING

Programming Projects 3.4

P1. *Complete the menu-driven demonstration program for manipulating an* Extended_queue *of characters, by implementing the function* get_command *and completing the function* do_command.

Answer The program of Project P2 includes these functions.

P2. *Write a menu-driven demonstration program for manipulating a deque of characters, similar to the* Extended_queue *demonstration program.*

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
typedef char Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../3e10/e10.h"
#include "../3e10/e10.cpp"

void introduction();
void help();
char get_command();
bool do_command(char, Deque &);

int main()
/*
Post: Accepts commands from user as a menu-driven demonstration program for
      the class Deque.
Uses: The class Deque and the
      functions introduction, get_command, and do_command.
*/

{
   Deque test_queue;
   introduction();
   while (do_command(get_command(), test_queue));
}

void introduction()
/*
Post: Writes out an introduction and instructions for the user.
*/

{
   cout << endl << "\t\tDeque Testing Program" << endl << endl
        << "The program demonstrates a deque of " << endl
        << "single character keys. " << endl

        << "The deque can hold a maximum of "
        << maxqueue << " characters." << endl << endl

        << "Valid commands are to be entered at each prompt." << endl
        << "Both upper and lower case letters can be used." << endl
        << "At the command prompt press H for help." << endl << endl;
}

void help()
/*
Post: A help screen for the program is printed, giving the meaning of each
      command that the user may enter.
*/

{
   cout << endl
        << "This program allows the user to enter one command" << endl
        << "(but only one) on each input line." << endl
        << "For example, if the command S is entered, then" << endl
        << "the program will serve the front of the queue." << endl
        << endl
     << " The valid commands are:" << endl
     << "A - Append the next input character to the rear." << endl
     << "P - Push the next input character to the front." << endl
     << "S - Serve the front of the queue" << endl
     << "X - Extract the rear of the queue" << endl
     << "R - Retrieve and print the front entry." << endl
     << "W - Retrieve and write the rear entry." << endl
     << "H - This help screen" << endl
     << "Q - Quit" << endl

     << "Press <Enter> to continue." << flush;
   char c;
   do {
      cin.get(c);
   } while (c != '\n');
}

char get_command()
/*
Post: Gets a valid command from the user and,
      after converting it to lower case if necessary,
      returns it.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:" << flush;
   while (true) {
      do {
         cin.get(c);
      } while (c == '\n'); //  c is now a command character.

      do {                 //  Skip remaining characters on the line.
         cin.get(d);
      } while (d != '\n');

      c = tolower(c);
      if (c == 'a' || c == 'p' || c == 's' || c == 'x' ||
          c == 'r' || c == 'w' || c == 'h' || c == 'q')
         return c;
      else
         cout << "Please enter a valid command or H for help:"
              << endl;
   }
}

bool do_command(char c, Deque &test_queue)
/*
Pre:  c represents a valid command.
Post: Performs the given command c on the Deque test_queue.
      Returns false if c == 'q', otherwise returns true.
Uses: The class Deque.
*/

{
   bool continue_input = true;
   Queue_entry x;
   switch (c) {
   case 'w':
      if (test_queue.retrieve_rear(x) == underflow)
         cout << "Deque is empty." << endl;
      else
         cout << endl
              << "The last entry is: " << x
              << endl;
      break;

   case 'r':
      if (test_queue.retrieve_front(x) == underflow)
         cout << "Deque is empty." << endl;
      else
         cout << endl
              << "The first entry is: " << x
              << endl;
      break;

   case 'q':
      cout << "Deque demonstration finished." << endl;
      continue_input = false;
      break;

   case 'x':
      if (test_queue.serve_rear() == underflow)
         cout << "Serve failed, the Queue is empty." << endl;
      break;

   case 's':
      if (test_queue.serve_front() == underflow)
         cout << "Serve failed, the Queue is empty." << endl;
      break;

   case 'p':
      cout << "Enter new key to insert:" << flush;
      cin.get(x);
      if (test_queue.append_front(x) != success)
         cout << "Operation failed: because the deque is full" << endl;
      break;

   case 'a':
      cout << "Enter new key to insert:" << flush;
      cin.get(x);
      if (test_queue.append_rear(x) != success)
         cout << "Operation failed: because the deque is full" << endl;
      break;

   case 'h':
      help();
      break;
   }
   return continue_input;
}

const int maxqueue = 10; //  small value for testing
```

```
class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int count;
   int front, rear;
   Queue_entry entry[maxqueue];
};

Queue::Queue()
/*
Post: The Queue is initialized to be empty.
*/

{
   count = 0;
   rear = maxqueue - 1;
   front = 0;
}

bool Queue::empty() const
/*
Post: Return true if the Queue is empty, otherwise
      return false.
*/

{
   return count == 0;
}

Error_code Queue::append(const Queue_entry &item)
/*
Post: item is added to the rear of the Queue. If the Queue is full
      return an Error_code of overflow and leave the Queue unchanged.
*/

{
   if (count >= maxqueue) return overflow;
   count++;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}

Error_code Queue::serve()
/*
Post: The front of the Queue is removed. If the Queue
      is empty return an Error_code of underflow.
*/

{
   if (count <= 0) return underflow;
   count--;
   front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}
```

TOC
Index
Help

```
Error_code Queue::retrieve(Queue_entry &item) const
/*
Post: The front of the Queue retrieved to the output
      parameter item. If the Queue
      is empty return an Error_code of underflow.
*/

{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

The code for the deque implementation that follows is taken directly from the solution to Exercise E10 of Section 3.3.

```
class Deque: private Queue {
public:
   Deque();
   bool empty() const;
   Error_code serve_front();
   Error_code serve_rear();
   Error_code append_front(const Queue_entry &item);
   Error_code append_rear(const Queue_entry &item);
   Error_code retrieve_front(Queue_entry &item) const;
   Error_code retrieve_rear(Queue_entry &item) const;
};


Deque::Deque()
/*
Post: The Deque is initialized to be empty.
*/
{ }

bool Deque::empty() const
/*
Post: Return true if the Deque is empty, otherwise return false.
*/
{return count <= 0;}

Error_code Deque::append_front(const Queue_entry &item)
/*
Post: item is added to the front of the Deque.  If the Deque is full
      return an Error_code of overflow.
*/

{
   if (count >= maxqueue) return overflow;
   front = (front + maxqueue - 1) % maxqueue;
   entry[front] = item;
   count ++;
   return success;
}

Error_code Deque::serve_front()
/*
Post: The front of the Deque is removed. If the Deque
      is empty return an Error_code of underflow.
*/
{return serve();}
```

```
Error_code Deque::retrieve_front(Queue_entry &item) const
/*
Post: The front of the Deque retrieved to the output parameter item.
      If the Deque is empty return an Error_code of underflow.
*/
{return retrieve(item);}

Error_code Deque::append_rear(const Queue_entry &item)
/*
Post: item is added to the rear of the Deque. If the Deque is full
      return an Error_code of overflow.
*/
{return append(item);}

Error_code Deque::serve_rear()
/*
Post: The rear of the Deque is removed. If the Deque
      is empty return an Error_code of underflow.
*/

{
   if (count <= 0) return underflow;
   rear = (rear + maxqueue - 1) % maxqueue;
   count --;
   return success;
}

Error_code Deque::retrieve_rear(Queue_entry &item) const
/*
Post: The rear of the Deque retrieved to the output parameter item.
      If the Deque is empty return an Error_code of underflow.
*/

{
   if (count <= 0) return underflow;
   item = entry[rear];
   return success;
}
```

TOC
Index
Help

3.5 APPLICATION OF QUEUES: SIMULATION

Programming Projects 3.5

P1. *Combine all the functions and methods for the airport simulation into a complete program. Experiment with several sample runs of the airport simulation, adjusting the values for the expected numbers of planes ready to land and take off. Find approximate values for these expected numbers that are as large as possible subject to the condition that it is very unlikely that a plane must be refused service. What happens to these values if the maximum size of the queues is increased or decreased?*

Answer All parts of the program appear in the text. The results of the experiments will vary depending on the number of time units for which the simulation is run. If the maximum size of each queue is increased, the resulting values will be larger. If the maximum size is decreased, the values will be smaller. If the size of the queue is increased, then more planes can wait to be served.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "plane.h"
#include "plane.cpp"
```

```
typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "runway.h"
#include "runway.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   cerr << "This program simulates an airport with only one runway." << endl
        << "One plane can land or depart in each unit of time." << endl;
   cerr << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cerr << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {

      cerr << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cerr << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;

      if (acceptable && arrival_rate + departure_rate > 1.0)
         cerr << "Safety Warning: This airport will become saturated. "
              << endl;
   } while (!acceptable);
}

void run_idle(int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/

{
   cout << time << ": Runway is idle." << endl;
}
```

```
int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation
      is to run, the expected number of planes arriving, the expected
      number of planes departing per time interval, and the maximum
      allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.
Uses: Classes Runway, Plane, Random and functions run_idle, initialize.
*/

{
   int end_time;               //  time to run simulation
   int queue_limit;            //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;

   Runway small_airport(queue_limit);
   for (int current_time = 0; current_time < end_time; current_time++) {
                               //  loop over time intervals
      int number_arrivals = variable.poisson(arrival_rate);
                               //  current arrival requests
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (small_airport.can_land(current_plane) != success)
            current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
                              //  current departure requests
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (small_airport.can_depart(current_plane) != success)
            current_plane.refuse();
      }

      Plane moving_plane;
      switch (small_airport.activity(current_time, moving_plane)) {
        //  Let at most one Plane onto the Runway at current_time.
      case land:
         moving_plane.land(current_time);
         break;

      case take_off:
         moving_plane.fly(current_time);
         break;

      case idle:
         run_idle(current_time);
      }
   }
   small_airport.shut_down(end_time);
}


enum Plane_status {null, arriving, departing};
```

```
class Plane {
public:
   Plane();
   Plane(int flt, int time, Plane_status status);
   void refuse() const;
   void land(int time) const;
   void fly(int time) const;
   int started() const;

private:
   int flt_num;
   int clock_start;
   Plane_status state;
};


enum Runway_activity {idle, land, take_off};

class Runway {
public:
   Runway(int limit);
   Error_code can_land(const Plane &current);
   Error_code can_depart(const Plane &current);
   Runway_activity activity(int time, Plane &moving);
   void shut_down(int time) const;

private:
   Extended_queue landing;
   Extended_queue takeoff;
   int queue_limit;
   int num_land_requests;     // number of planes asking to land
   int num_takeoff_requests;  // number of planes asking to take off
   int num_landings;          // number of planes that have landed
   int num_takeoffs;          // number of planes that have taken off
   int num_land_accepted;     // number of planes queued to land
   int num_takeoff_accepted;  // number of planes queued to take off
   int num_land_refused;      // number of landing planes refused
   int num_takeoff_refused;   // number of departing planes refused
   int land_wait;             // total time of planes waiting to land
   int takeoff_wait;          // total time of planes waiting to take off
   int idle_time;             // total time runway is idle
};


Plane::Plane(int flt, int time, Plane_status status)
/*
Post:  The Plane data members flt_num, clock_start,
and state are set to the values of the parameters flt,
time and status, respectively.
*/

{
   flt_num = flt;
   clock_start = time;
   state = status;
   cout << "Plane number " << flt << " ready to ";
   if (status == arriving)
      cout << "land." << endl;
   else
      cout << "take off." << endl;
}
```

TOC
Index
Help

```
Plane::Plane()
/*
Post:  The Plane data members flt_num, clock_start,
state are set to illegal default values.
*/

{
   flt_num = -1;
   clock_start = -1;
   state = null;
}

void Plane::refuse() const
/*
Post: Processes a Plane wanting to use Runway, when
the Queue is full.
*/

{
   cout << "Plane number " << flt_num;
   if (state == arriving)
      cout << " directed to another airport" << endl;
   else
      cout << " told to try to takeoff again later" << endl;
}

void Plane::land(int time) const
/*
Post: Processes a Plane that is landing at the specified time.
*/

{
   int wait = time - clock_start;
   cout << time << ": Plane number " << flt_num << " landed after "
        << wait << " time unit" << ((wait == 1) ? "" : "s")
        << " in the landing queue." << endl;
}

void Plane::fly(int time) const
/*
Post: Process a Plane that is taking off at the specified time.
*/

{
   int wait = time - clock_start;
   cout << time << ": Plane number " << flt_num << " took off after "
        << wait << " time unit" << ((wait == 1) ? "" : "s")
        << " in the takeoff queue." << endl;
}

int Plane::started() const
/*
Post: Return the time that the Plane entered the airport system.
*/

{
   return clock_start;
}
```

```
Runway::Runway(int limit)
/*
Post: The Runway data members are initialized to record no
      prior Runway use and to record the limit on queue sizes.
*/

{
   queue_limit = limit;
   num_land_requests = num_takeoff_requests = 0;
   num_landings = num_takeoffs = 0;
   num_land_refused = num_takeoff_refused = 0;
   num_land_accepted = num_takeoff_accepted = 0;
   land_wait = takeoff_wait = idle_time = 0;
}

Error_code Runway::can_land(const Plane &current)
/*
Post: If possible, the Plane current is added to the
      landing Queue; otherwise, an Error_code of overflow is
      returned. The Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Error_code result;

   if (landing.size() < queue_limit)
      result = landing.append(current);
   else
      result = fail;

   num_land_requests++;

   if (result != success)
      num_land_refused++;
   else
      num_land_accepted++;

   return result;
}

Error_code Runway::can_depart(const Plane &current)
/*
Post:  If possible, the Plane current is added to the
takeoff Queue; otherwise, an Error_code of overflow is
returned. The Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Error_code result;

   if (takeoff.size() < queue_limit)
      result = takeoff.append(current);
   else
      result = fail;

   num_takeoff_requests++;
   if (result != success)
      num_takeoff_refused++;
   else
      num_takeoff_accepted++;
   return result;
}

Runway_activity Runway::activity(int time, Plane &moving)
/*
Post: If the landing Queue has entries, its front
      Plane is copied to the parameter moving
and a result  land is returned. Otherwise,
if the takeoff Queue has entries, its front
Plane is copied to the parameter moving
and a result  takeoff is returned. Otherwise,
idle is returned. Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Runway_activity in_progress;
   if (!landing.empty()) {
      landing.retrieve(moving);
      land_wait += time - moving.started();
      num_landings++;
      in_progress = land;
      landing.serve();
   }

   else if (!takeoff.empty()) {
      takeoff.retrieve(moving);
      takeoff_wait += time - moving.started();
      num_takeoffs++;
      in_progress = take_off;
      takeoff.serve();
   }

   else {
      idle_time++;
      in_progress = idle;
   }
   return in_progress;
}

void Runway::shut_down(int time) const
/*
Post: Runway usage statistics are summarized and printed.
*/

{
   cout << "Simulation has concluded after " << time
        << " time units." << endl
        << "Total number of planes processed "
        << (num_land_requests + num_takeoff_requests) << endl

        << "Total number of planes asking to land "
        << num_land_requests << endl

        << "Total number of planes asking to take off "
        << num_takeoff_requests << endl
```

```
          << "Total number of planes accepted for landing "
          << num_land_accepted << endl

          << "Total number of planes accepted for takeoff "
          << num_takeoff_accepted << endl

          << "Total number of planes refused for landing "
          << num_land_refused << endl

          << "Total number of planes refused for takeoff "
          << num_takeoff_refused << endl

          << "Total number of planes that landed "
          << num_landings << endl

          << "Total number of planes that took off "
          << num_takeoffs << endl

          << "Total number of planes left in landing queue "
          << landing.size() << endl

          << "Total number of planes left in takeoff queue "
          << takeoff.size() << endl;

     cout << "Percentage of time runway idle "
          << 100.0 * (( float ) idle_time) / (( float ) time) << "%" << endl;

     cout << "Average wait in landing queue "
          << (( float ) land_wait) / (( float ) num_landings)
          << " time units";
     cout << endl << "Average wait in takeoff queue "
          << (( float ) takeoff_wait) / (( float ) num_takeoffs)
          << " time units" << endl;

     cout << "Average observed rate of planes wanting to land "
          << (( float ) num_land_requests) / (( float ) time)
          << " per time unit" << endl;

     cout << "Average observed rate of planes wanting to take off "
          << (( float ) num_takeoff_requests) / (( float ) time)
          << " per time unit" << endl;
}
```

P2. *Modify the simulation to give the airport two runways, one always used for landings and one always used for takeoffs. Compare the total number of planes that can be served with the number for the one-runway airport. Does it more than double?*

Answer The Runway and Plane classes developed in the text are also suitable for this project. A driver for this project follows.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../airport/plane.h"
#include "../airport/plane.cpp"
typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "../airport/runway.h"
#include "../airport/runway.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
```

```
void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   cout << "This program simulates an airport with two runways." << endl
        << "One runway is used for landings, one for departures." << endl
        << "One plane can land or depart in each unit of time." << endl;

   cout << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cout << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {

      cout << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cout << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;
   } while (!acceptable);
}

void run_idle(char *runway_id, int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/

{
   cout << time << ": " << runway_id << " Runway is idle."
        << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation is to
      run, the expected number of planes arriving, the expected number
      of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.
Uses: Classes Runway, Plane, Random and
      functions run_idle, initialize.
*/
```

```
{
   int end_time;              //  time to run simulation
   int queue_limit;           //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;

   Runway arrivals(queue_limit);   // set up the two runways.
   Runway departures(queue_limit);

   for (int current_time = 0; current_time < end_time; current_time++) {
      int number_arrivals = variable.poisson(arrival_rate);
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (arrivals.can_land(current_plane) != success)
            current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (departures.can_depart(current_plane) != success)
            current_plane.refuse();
      }

      Plane arriving_plane;
      switch (arrivals.activity(current_time, arriving_plane)) {
      case land:
         arriving_plane.land(current_time);
         break;

      case take_off:
         cout << "WARNING: Unexpected, catastrophic program failure!"
              <<  endl;
         break;

      case idle:
         run_idle("Arrival", current_time);
      }
      Plane departing_plane;
      switch (departures.activity(current_time, departing_plane)) {
      case take_off:
         departing_plane.fly(current_time);
         break;

      case land:
         cout << "WARNING: Unexpected, catastrophic program failure!"
              <<  endl;
         break;

      case idle:
         run_idle("Departure", current_time);
      }
   }
   cout << "\n\n----Arrival Runway statistics----\n " << endl;
   arrivals.shut_down(end_time);
   cout << "\n\n----Departure Runway statistics----\n " << endl;
   departures.shut_down(end_time);
}
```

P3. *Modify the simulation to give the airport two runways, one usually used for landings and one usually used for takeoffs. If one of the queues is empty, then both runways can be used for the other queue. Also, if the landing queue is full and another plane arrives to land, then takeoffs will be stopped and both runways used to clear the backlog of landing planes.*

Answer This simulation needs to use extra runway methods that query the runways about the sizes of their queues. A suitably modified Runway class is:

```
enum Runway_activity {idle, land, take_off};

class Runway {
public:
   Runway(int limit);
   Error_code can_land(const Plane &current);
   Error_code can_depart(const Plane &current);
   Runway_activity activity(int time, Plane &moving);
   void shut_down(int time) const;
   int arrival_size();
   int departure_size();

private:
   Extended_queue landing;
   Extended_queue takeoff;
   int queue_limit;
   int num_land_requests;     //  number of planes asking to land
   int num_takeoff_requests;  //  number of planes asking to take off
   int num_landings;          //  number of planes that have landed
   int num_takeoffs;          //  number of planes that have taken off
   int num_land_accepted;     //  number of planes queued to land
   int num_takeoff_accepted;  //  number of planes queued to take off
   int num_land_refused;      //  number of landing planes refused
   int num_takeoff_refused;   //  number of departing planes refused
   int land_wait;             //  total time of planes waiting to land
   int takeoff_wait;          //  total time of planes waiting to take off
   int idle_time;             //  total time runway is idle
};


int Runway::departure_size()
/*
Post:  Returns size of the departure queue.
Uses: class Extended_queue.
*/

{
   return takeoff.size();
}

int Runway::arrival_size()
/*
Post:  Returns size of the arrival queue.
Uses: class Extended_queue.
*/

{
   return landing.size();
}
```

```
Runway::Runway(int limit)
/*
Post: The Runway data members are initialized to record no
      prior Runway use and to record the limit on queue sizes.
*/

{
   queue_limit = limit;
   num_land_requests = num_takeoff_requests = 0;
   num_landings = num_takeoffs = 0;
   num_land_refused = num_takeoff_refused = 0;
   num_land_accepted = num_takeoff_accepted = 0;
   land_wait = takeoff_wait = idle_time = 0;
}

Error_code Runway::can_land(const Plane &current)
/*
Post: If possible, the Plane current is added to the
      landing Queue; otherwise, an Error_code of overflow is
      returned. The Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Error_code result;
   if (landing.size() < queue_limit)
      result = landing.append(current);
   else
      result = fail;

   num_land_requests++;

   if (result != success)
      num_land_refused++;
   else
      num_land_accepted++;

   return result;
}

Error_code Runway::can_depart(const Plane &current)
/*
Post: If possible, the Plane current is added to the
      takeoff Queue; otherwise, an Error_code of overflow is
      returned. The Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Error_code result;

   if (takeoff.size() < queue_limit)
      result = takeoff.append(current);
   else
      result = fail;
   num_takeoff_requests++;

   if (result != success)
      num_takeoff_refused++;
   else
      num_takeoff_accepted++;
   return result;
}
```

```
Runway_activity Runway::activity(int time, Plane &moving)
/*
Post: If the landing Queue has entries, its front
      Plane is copied to the parameter moving
      and a result  land is returned. Otherwise,
      if the takeoff Queue has entries, its front
      Plane is copied to the parameter moving
      and a result  takeoff is returned. Otherwise,
      idle is returned. Runway statistics are updated.
Uses: class Extended_queue.
*/

{
   Runway_activity in_progress;
   if (!landing.empty()) {
      landing.retrieve(moving);
      land_wait += time - moving.started();
      num_landings++;
      in_progress = land;
      landing.serve();
   }

   else if (!takeoff.empty()) {
      takeoff.retrieve(moving);
      takeoff_wait += time - moving.started();
      num_takeoffs++;
      in_progress = take_off;
      takeoff.serve();
   }

   else {
      idle_time++;
      in_progress = idle;
   }
   return in_progress;
}

void Runway::shut_down(int time) const
/*
Post: Runway usage statistics are summarized and printed.
*/

{
   cout << "Simulation has concluded after " << time
        << " time units." << endl
        << "Total number of planes processed "
        << (num_land_requests + num_takeoff_requests) << endl

        << "Total number of planes asking to land "
        << num_land_requests << endl

        << "Total number of planes asking to take off "
        << num_takeoff_requests << endl

        << "Total number of planes accepted for landing "
        << num_land_accepted << endl

        << "Total number of planes accepted for takeoff "
        << num_takeoff_accepted << endl

        << "Total number of planes refused for landing "
        << num_land_refused << endl
```

```
          << "Total number of planes refused for takeoff "
          << num_takeoff_refused << endl

          << "Total number of planes that landed "
          << num_landings << endl

          << "Total number of planes that took off "
          << num_takeoffs << endl

          << "Total number of planes left in landing queue "
          << landing.size() << endl

          << "Total number of planes left in takeoff queue "
          << takeoff.size() << endl;

     cout << "Percentage of time runway idle "
          << 100.0 * (( float ) idle_time) / (( float ) time) << "%" << endl;

     cout << "Average wait in landing queue "
          << (( float ) land_wait) / (( float ) num_landings)
          << " time units";

     cout << endl << "Average wait in takeoff queue "
          << (( float ) takeoff_wait) / (( float ) num_takeoffs)
          << " time units" << endl;

     cout << "Average observed rate of planes wanting to land "
          << (( float ) num_land_requests) / (( float ) time)
          << " per time unit" << endl;

     cout << "Average observed rate of planes wanting to take off "
          << (( float ) num_takeoff_requests) / (( float ) time)
          << " per time unit" << endl;
}
```

The driver is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../airport/plane.h"
#include "../airport/plane.cpp"

typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "runway.h"
#include "runway.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/
```

```
{
   cerr << "This program simulates an airport with two runways." << endl
        << "One runway is normally used for landings, "
        << "the other is normally for departures." << endl
        << "One plane can land or depart on each runway "
        << "in each unit of time." << endl;

   cerr << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cerr << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {
      cerr << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cerr << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;
   } while (!acceptable);
}

void run_idle(char *runway_id, int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/

{
   cout << time << ": " << runway_id << " Runway is idle." << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation is
      to run, the expected number of planes arriving, the expected number
      of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.
Uses: Classes Runway, Plane, Random and functions run_idle, initialize.
*/

{
   int end_time;            //  time to run simulation
   int queue_limit;         //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;

   Runway arrivals(queue_limit);   // set up the two runways.
   Runway departures(queue_limit);
   for (int current_time = 0; current_time < end_time; current_time++) {
      int number_arrivals = variable.poisson(arrival_rate);
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (departures.departure_size() == 0 &&
              departures.arrival_size() == 0)
                 departures.can_land(current_plane);
         else if (arrivals.can_land(current_plane) != success)
            if (departures.can_land(current_plane) != success)
               current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (arrivals.departure_size() == 0 &&
              arrivals.arrival_size() == 0)
                 arrivals.can_depart(current_plane);
         else if (departures.can_depart(current_plane) != success)
            current_plane.refuse();
      }

      Plane moving_plane;
      switch (arrivals.activity(current_time, moving_plane)) {
      case land:
         moving_plane.land(current_time);
         break;

      case take_off:
         cout << "On arrival runway: ";
         moving_plane.fly(current_time);
         break;

      case idle:
         run_idle("Arrival", current_time);
      }

      switch (departures.activity(current_time, moving_plane)) {
      case take_off:
         moving_plane.fly(current_time);
         break;

      case land:
         cout << "On departure runway: ";
         moving_plane.land(current_time);
         break;

      case idle:
         run_idle("Departure", current_time);
      }
   }

   cout << "\n\n----Arrival Runway statistics----\n " << endl;
   arrivals.shut_down(end_time);
   cout << "\n\n----Departure Runway statistics----\n " << endl;
   departures.shut_down(end_time);
}
```

With the solution program, the number of planes served successfully will be slightly less than double that of the basic one-runway airport, because, depending on the input parameters, one of the runways will sometimes be idle.

P4. *Modify the simulation to have three runways, one always reserved for each of landing and takeoff and the third used for landings unless the landing queue is empty, in which case it can be used for takeoffs.*

Answer The Runway class of Project P3 remains suitable for this project. The driver is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../airport/plane.h"
#include "../airport/plane.cpp"

typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "runway.h"
#include "runway.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   cerr << "This program simulates an airport with three runways." << endl
        << "One runway is used for landings, "
        << "a second is for departures." << endl
        << "The third runway is for overflow traffic.  It gives priority "
        << "to arrivals.\nOne plane can land or depart on each runway "
        << "in each unit of time." << endl;

   cerr << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cerr << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {
      cerr << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cerr << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;
   } while (!acceptable);
}
```

```
void run_idle(char *runway_id, int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/
{
   cout << time << ": " << runway_id << " Runway is idle." << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation is to
      run, the expected number of planes arriving, the expected number
      of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runways at each time interval, and prints out a
      summary of airport operation at the conclusion.
Uses: Classes Runway, Plane, Random and functions run_idle, initialize.
*/

{
   int end_time;            //  time to run simulation
   int queue_limit;         //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;

   Runway arrivals(queue_limit);   // set up the 3 runways.
   Runway departures(queue_limit);
   Runway overflow(queue_limit);

   for (int current_time = 0; current_time < end_time; current_time++) {
      int number_arrivals = variable.poisson(arrival_rate);
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (arrivals.can_land(current_plane) != success)
            if (overflow.can_land(current_plane) != success)
               current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (departures.can_depart(current_plane) != success)
            if (overflow.arrival_size() == 0)
               if (overflow.can_depart(current_plane) != success)
                    current_plane.refuse();
      }

      Plane moving_plane;
      switch (arrivals.activity(current_time, moving_plane)) {
      case land:
         cout << "Landing runway: ";
         moving_plane.land(current_time);
         break;

      case take_off:
         cout << "Landing runway: ";
         cout << "WARNING: catastophic failure!" << endl;
         break;
      case idle:
         run_idle("Arrival", current_time);
      }
      switch (departures.activity(current_time, moving_plane)) {
      case take_off:
         cout << "Takeoff runway: ";
         moving_plane.fly(current_time);
         break;
      case land:
         cout << "Takeoff runway: ";
         cout << "WARNING: catastophic failure!" << endl;
         break;
      case idle:
         run_idle("Departure", current_time);
      }
      switch (overflow.activity(current_time, moving_plane)) {
      case take_off:
         cout << "Overflow runway: ";
         moving_plane.fly(current_time);
         break;
      case land:
         cout << "Overflow runway: ";
         moving_plane.land(current_time);
         break;
      case idle:
         run_idle("Overflow", current_time);
      }
   }
   cout << "\n\n----Arrival Runway statistics----\n " << endl;
   arrivals.shut_down(end_time);
   cout << "\n\n----Departure Runway statistics----\n " << endl;
   departures.shut_down(end_time);
   cout << "\n\n----Overflow Runway statistics----\n " << endl;
   overflow.shut_down(end_time);
}
```

TOC
Index
Help

P5. *Modify the original (one-runway) simulation so that when each plane arrives to land, it will have (as one of its data members) a (randomly generated) fuel level, measured in units of time remaining. If the plane does not have enough fuel to wait in the queue, it is allowed to land immediately. Hence the planes in the landing queue may be kept waiting additional units, and so may run out of fuel themselves. Check this out as part of the landing function, and find about how busy the airport can become before planes start to crash from running out of fuel.*

Answer The class Runway is unmodified from the text (and project P1). The class Plane is modified to account for fuel:

```
enum Plane_status {null, arriving, departing, emergency};
class Plane {
public:
   Plane();
   Plane(int flt, int time, Plane_status status);
   void refuse() const;
   void land(int time) const;
   void fly(int time) const;
   int started() const;
   Plane_status get_status() const;
private:
   int flt_num;
   int clock_start;
   Plane_status state;
   int fuel;
};


int crashes;
Random dice;

Plane::Plane(int flt, int time, Plane_status status)
/*
Post: The Plane data members flt_num, clock_start,
      and state are set to the values of the parameters flt,
      time and status, respectively.
*/

{
   flt_num = flt;
   clock_start = time;
   state = status;
   if (status == arriving) {
      fuel = dice.random_integer(0, 20);
      if (fuel <= 1) status = state = emergency;
   }

   cout << "Plane number " << flt << " ready to ";
   if (status == arriving)
      cout << "land." << endl;
   else if (status == departing)
      cout << "take off." << endl;
   else if (status == emergency)
      cout << "land in a fuel EMERGENCY." << endl;
}

Plane_status Plane::get_status() const
{
   return state;
}

Plane::Plane()
/*
Post: The Plane data members flt_num, clock_start,
      state are set to illegal default values.
*/

{
   flt_num = -1;
   clock_start = -1;
   state = null;
}

void Plane::refuse() const
/*
Post: Processes a Plane wanting to use Runway, when the Queue is full.
*/
```

```
{
   cout << "Plane number " << flt_num;
   if (state == arriving || state == emergency) {
      cout << " directed to another airport" << endl;
      if (fuel <= 3) {
         cout << "Unfortunately it didn't have enough fuel and crashed.";
         cout << endl;
         cout << "plane CRASH \n\n";
         crashes++;
      }
   }
   else
      cout << " told to try to takeoff again later" << endl;
}

void Plane::land(int time) const
/*
Post: Processes a Plane that is landing at the specified time.
*/

{
   int wait = time - clock_start;
   if (wait > fuel) {
      cout << "\n plane CRASH \n";
      cout << time << ": Plane number " << flt_num
           << " ran out of fuel crashed after "
           << wait << " time unit" << ((wait == 1) ? "" : "s")
           << " in the landing queue." << endl;
       crashes++;
   }

   else cout << time << ": Plane number " << flt_num << " landed after "
             << wait << " time unit" << ((wait == 1) ? "" : "s")
             << " in the landing queue." << endl;
}

void Plane::fly(int time) const
/*
Post: Process a Plane that is taking off at the specified time.
*/

{
   int wait = time - clock_start;
   cout << time << ": Plane number " << flt_num << " took off after "
        << wait << " time unit" << ((wait == 1) ? "" : "s")
        << " in the takeoff queue." << endl;
}

int Plane::started() const
/*
Post: Return the time that the Plane entered the airport system.
*/

{
   return clock_start;
}
```

The driver is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include "plane.h"
#include "plane.cpp"

typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "../airport/runway.h"
#include "../airport/runway.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   crashes = 0;
   cerr << "This program simulates an airport with only one runway." << endl
        << "One plane can land or depart in each unit of time." << endl;
   cerr << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cerr << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {

      cerr << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cerr << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;

      if (acceptable && arrival_rate + departure_rate > 1.0)
         cerr << "Safety Warning: This airport will become saturated."
              << endl;

   } while (!acceptable);
}

void run_idle(int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/
```

```
{
   cout << time << ": Runway is idle." << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation is
      to run, the expected number of planes arriving, the expected number
      of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.  Emergency measures
      are taken for planes without fuel.
Uses: Classes Runway, Plane, Random and functions run_idle, initialize.
*/

{
   int end_time;             //  time to run simulation
   int queue_limit;          //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;
   Plane moving_plane;
   bool emergency_action;

   Runway small_airport(queue_limit);
   for (int current_time = 0; current_time < end_time; current_time++) {
      emergency_action = false;
      int number_arrivals = variable.poisson(arrival_rate);
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (!emergency_action && current_plane.get_status() == emergency) {
            cout << "EMERGENCY normal action suspended at airport " << endl;
            emergency_action = true;
            moving_plane = current_plane;  // it can land at once
         }
         else if (small_airport.can_land(current_plane) != success)
            current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (small_airport.can_depart(current_plane) != success)
            current_plane.refuse();
      }

      if (emergency_action) moving_plane.land(current_time);
      else switch (small_airport.activity(current_time, moving_plane)) {
      case land:
         moving_plane.land(current_time);
         break;

      case take_off:
         moving_plane.fly(current_time);
         break;
      case idle:
         run_idle(current_time);
      }
   }

   cout << "\n\n------------ Runway Statistics ---------------\n" << endl;
   small_airport.shut_down(end_time);
   cout << "\n\n------------------------------------------\n" << endl;
   cout << "The airport allowed " << crashes << " planes to crash." << endl;
}
```

P6. *Write a stub to take the place of the random-number function. The stub can be used both to debug the program and to allow the user to control exactly the number of planes arriving for each queue at each time unit.*

Answer A reimplementation of the random class with stubs in place of the random number generation methods is provided by:

```
#include <limits.h>
const int max_int = INT_MAX;
#include <math.h>
#include <time.h>

Random::Random(bool pseudo)
/*
Post: The values of seed, add_on, and multiplier
      are initialized.  The seed is initialized randomly only if
      pseudo == false.
*/
{
   if (pseudo) seed = 1;
   else seed = time(NULL) % max_int;
   multiplier = 2743;
   add_on = 5923;
}

double Random::random_real()
/*
Post: A random real number between 0 and 1 is returned.
*/
{
   cerr << "\nSupply a random real (between 0 and 1): " << flush;
   double temp;
   cin >> temp;
   return temp;
}

int Random::random_integer(int low, int high)
/*
Post: A random integer between low and high (inclusive) is returned.
*/
{
   if (low > high) return random_integer(high, low);
   cerr << "\nSupply a random integer (between " << low << " and "
        << high <<"): " << flush;
   int temp;
   cin >> temp;
   return temp;
}
```

```
int Random::poisson(double mean)
/*
Post: A random integer, reflecting a Poisson distribution
      with parameter mean, is returned.
*/

{
   cerr << "\nSupply a random integer (around " << mean
        << "): " << flush;
   int temp;
   cin >> temp;
   return temp;
}

int Random::reseed()
/*
Post: The seed is replaced by a psuedorandom successor.
*/

{
   seed = seed * multiplier + add_on;
   return seed;
}
```

A driver program is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "random.cpp"
#include "../5p5/plane.h"
#include "../5p5/plane.cpp"

typedef Plane Queue_entry;
#include "../queue/queue.h"
#include "../queue/queue.cpp"
#include "../extqueue/extqueue.h"
#include "../extqueue/extqueue.cpp"
#include "../airport/runway.h"
#include "../airport/runway.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   crashes = 0;
   cerr << "This program simulates an airport with only one runway." << endl
        << "One plane can land or depart in each unit of time." << endl;
   cerr << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;
   cerr << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {
      cerr << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cerr << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;

      if (acceptable && arrival_rate + departure_rate > 1.0)
         cerr << "Safety Warning: This airport will become saturated."
              << endl;

   } while (!acceptable);
}

void run_idle(int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/

{
   cout << time << ": Runway is idle." << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation
      is to run, the expected number of planes arriving, the expected
      number of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.  Emergency measures
      are taken for planes without fuel.
Uses: Classes Runway, Plane, Random and functions run_idle, initialize.
*/

{
   int end_time;             //  time to run simulation
   int queue_limit;          //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;
   Plane moving_plane;
   bool emergency_action;
```

TOC
Index
Help

```
      Runway small_airport(queue_limit);
      for (int current_time = 0; current_time < end_time; current_time++) {
         emergency_action = false;
         int number_arrivals = variable.poisson(arrival_rate);
         for (int i = 0; i < number_arrivals; i++) {
            Plane current_plane(flight_number++, current_time, arriving);
            if (!emergency_action && current_plane.get_status() == emergency) {
               cout << "EMERGENCY normal action suspended at airport " << endl;
               emergency_action = true;
               moving_plane = current_plane;  // it can land at once
            }
            else if (small_airport.can_land(current_plane) != success)
               current_plane.refuse();
         }

         int number_departures= variable.poisson(departure_rate);
         for (int j = 0; j < number_departures; j++) {
            Plane current_plane(flight_number++, current_time, departing);
            if (small_airport.can_depart(current_plane) != success)
               current_plane.refuse();
         }

         if (emergency_action) moving_plane.land(current_time);
         else switch (small_airport.activity(current_time, moving_plane)) {
         case land:
            moving_plane.land(current_time);
            break;

         case take_off:
            moving_plane.fly(current_time);
            break;

         case idle:
            run_idle(current_time);
         }
      }
      cout << "\n\n------------ Runway Statistics ---------------\n" << endl;
      small_airport.shut_down(end_time);
      cout << "\n\n-------------------------------------------\n" << endl;
      cout << "The airport allowed " << crashes << " planes to crash." << endl;
   }
```

REVIEW QUESTIONS

1. *Define the term queue. What operations can be done on a queue?*

A queue is a data structure where insertions can only be appended to the rear of the list, and deletions can only be served from the front of the list (***FIFO***). The operations that can be performed on a queue are: append, serve, construction, empty, and retrieve.

2. *How is a circular array implemented in a linear array?*

A circular array is implemented by wrapping positions past the end of a linear array (used to implement the circular array) around to the beginning of the linear array. The % operator may be used to find the locations within the circular array.

3. *List three different implementations of queues.*

There are eight different implementations listed on page 88.

4. *Explain the difference between has-a and is-a relationships between classes.*

A pair of classes exhibit an is-a relationship if one is publicly derived from the other. A pair of classes exhibit a has-a relationship if one of the classes includes a member that is an object from the other class.

5. *Define the term simulation.*

A simulation is a computer model of an activity which is used to study that activity.

Linked Stacks and Queues 4

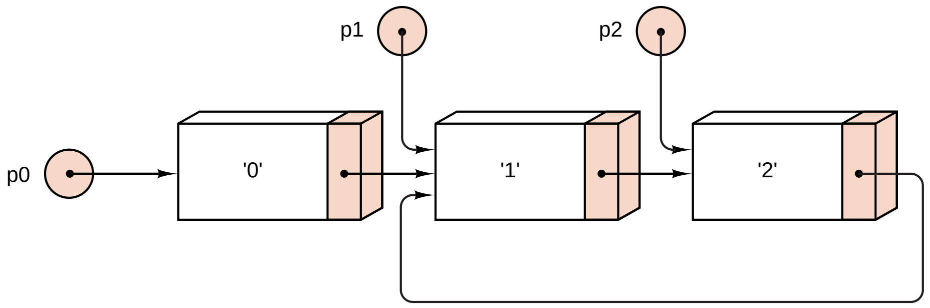
4.1 POINTERS AND LINKED STRUCTURES

Exercises 4.1

E1. *Draw a diagram to illustrate the configuration of linked nodes that is created by the following statements.*

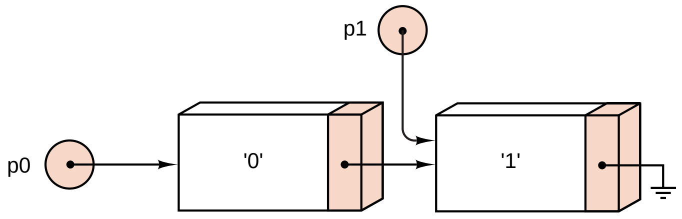
```
Node *p0 = new Node('0');
Node *p1 = p0->next = new Node('1');
Node *p2 = p1->next = new Node('2', p1);
```

Answer



E2. *Write the C++ statements that are needed to create the linked configuration of nodes shown in each of the following diagrams. For each part, embed these statements as part of a program that prints the contents of each node (both* data *and* next*), thereby demonstrating that the nodes have been correctly linked.*

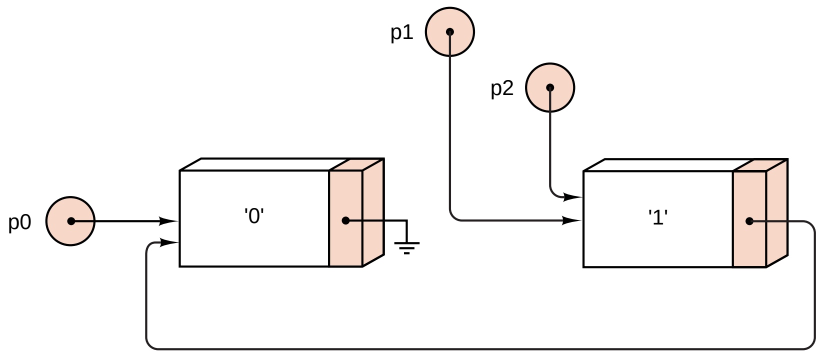
(a)



Answer

```
Node *p0 = new Node('0');
Node *p1 = p0->next = new Node('1');
```

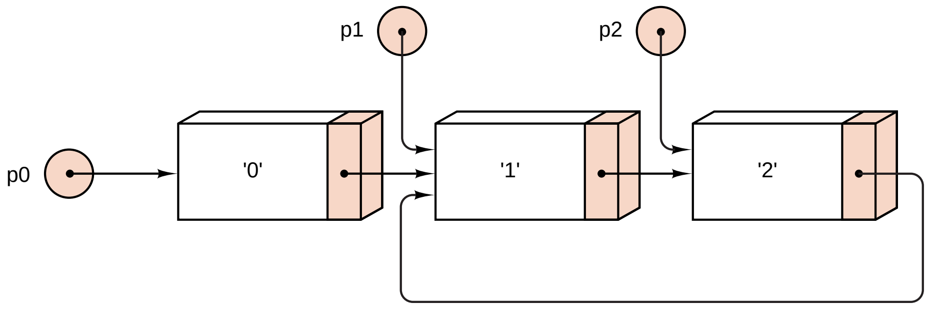
(b)



Answer

```
Node *p0 = new Node('0');
Node *p1 = new Node('1', p0);
Node *p2 = p1;
```

(c)



Answer

```
Node *p0 = new Node('0');
Node *p1 = p0->next = new Node('1');
Node *p2 = p1->next = new Node('2', p1);
```

4.2 LINKED STACKS

Exercises 4.2

E1. *Explain why we cannot use the following implementation for the method* push *in our linked* Stack.

```
Error_code Stack :: push(Stack_entry item)
{
   Node new_top(item, top_node);
   top_node = new_top;
   return success;
}
```

Answer There is a type incompatibility in the assignment statement top_node = new_top. Moreover, even if we corrected this incompatibility with the assignment top_node = &new_top, the statically acquired node would disappear at the end of the function, and the remaining nodes of the Stack would become garbage.

E2. *Consider a linked stack that includes a method* size*. This method* size *requires a loop that moves through the entire stack to count the entries, since the number of entries in the stack is not kept as a separate member in the stack record.*

(a) *Write a method* size *for a linked stack by using a loop that moves a pointer variable from node to node through the stack.*

Answer

```
int Stack :: size( ) const
/* Post: Return the number of entries in the Stack. */
{
  Node *temp = top_node;
  int count = 0;
  while (temp != NULL) {
    temp = temp ->next;
    count ++;
  }
  return count;
}
```

(b) *Consider modifying the declaration of a linked stack to make a stack into a structure with two members, the top of the stack and a counter giving its size. What changes will need to be made to the other methods for linked stacks? Discuss the advantages and disadvantages of this modification compared to the original implementation of linked stacks.*

Answer In the constructor Stack, the counter would need to be initialized. The method empty could now just check the value of the counter to see if it contains the value 0, and size would now only have to access the current value of the counter member. As well, the methods for pushing and popping the stack would now include statements to increment or decrement the counter, as required. The primary advantage of this modification is that it is more in line with the contiguous stack implementation and reduces the time required to check the size of the stack. The disadvantages include the addition of an extra member to maintain in the stack and the additional coding required, especially considering the limited use of such a member in a linked implementation. The advantages and disadvantages would depend to a large degree on how frequently the function size is used.

Programming Project 4.2

P1. *Write a demonstration program that can be used to check the methods written in this section for manipulating stacks. Model your program on the one developed in Section 3.4 and use as much of that code as possible. The entries in your stack should be characters. Your program should write a one-line menu from which the user can select any of the stack operations. After your program does the requested operation, it should inform the user of the result and ask for the next request. When the user wishes to push a character onto the stack, your program will need to ask what character to push.*

Use the linked implementation of stacks, and be careful to maintain the principles of information hiding.

Answer Driver program:

```
#include "../../c/utility.h"

void help()
/*
PRE:  None.
POST: Instructions for the Stack operations have been printed.
*/

{
     cout << "The legal commands available are\n";
     cout << "\t[P]ush (char) [T]op    [D]elete (pop) \n"
             "\t[Q]uit        [?]help     \n" << endl;
}

#include <string.h>
// include auxiliary data structures
```

```
typedef char Node_entry;
#include "../nodes/node.h"
typedef Node_entry Stack_entry;
#include "../linkstac/stack.h"
#include "../nodes/node.cpp"
#include "../linkstac/stack.cpp"

char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands for the
      Stack demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("ptdq?",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

char get_char()
{
   char c;
   cin >> c;
   return c;
}

int do_command(char c, Stack &test)
/*
PRE:  The Stack has been created and c is a valid Stack
      operation.
POST: The command c has been executed.
USES: The class Stack.
*/

{
   char x;
   switch (c) {
   case '?': help();
     break;

   case 'p':
      cout << "Enter character to push:";
      x = get_char();
      test.push(x);
      break;
   case 'd':
      if (!test.empty()) test.pop();
      break;

   case 't':
      if (test.empty()) cout <<"The stack is empty." << endl;
      else {
         test.top(x);
         cout << "The top of the stack is: " << x << endl;
      }
      break;

   case 'q':
      cout << "Linked Stack demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/*
PRE:  None.
POST: A linked Stack demonstration has been performed.
USES: get_command, do_command, class Stack (linked implementation)
*/

{
   cout << "Demonstration program for a linked stack of character data."
        << endl;
   help();
   Stack test;
   while (do_command(get_command(), test));
}
```

Header file `stack.h`:

```
class Stack {
public:
//  Standard Stack methods
   Stack();
   bool empty() const;
   Error_code push(const Stack_entry &item);
   Error_code pop();
   Error_code top(Stack_entry &item) const;
//  Safety features for linked structures
   ~Stack();
   Stack(const Stack &original);
   void operator =(const Stack &original);
protected:
   Node *top_node;
};
```

Header file `node.h`:

```
struct Node {
//  data members
   Node_entry entry;
   Node *next;
//  constructors
   Node();
   Node(Node_entry item, Node *add_on = NULL);
};
```

Code files:

```
#include "../../4/linkstac/stack1.cpp"

Stack::Stack(const Stack &original) //  copy constructor
/*
Post: The Stack is initialized as a copy of Stack original.
*/
{
   Node *new_copy, *original_node = original.top_node;
   if (original_node == NULL) top_node = NULL;
   else {                          //  Duplicate the linked nodes.
      top_node = new_copy = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
}

void Stack::operator = (const Stack &original) //  Overload assignment
/*
Post: The Stack is reset as a copy of Stack original.
*/
{
   Node *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                          //  Duplicate the linked nodes
      new_copy = new_top = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())             // Clean out old Stack entries
      pop();
   top_node = new_top;         // and replace them with new entries.
}

Stack::~Stack() //  Destructor
/*
Post: The Stack is cleared.
*/
{
   while (!empty())
      pop();
}

Node::Node()
{
   next = NULL;
}
```

```
Node::Node(Node_entry item, Node *add_on)
{
   entry = item;
   next = add_on;
}
```

4.3 LINKED STACKS WITH SAFEGUARDS

Exercises 4.3

E1. *Suppose that* x, y, *and* z *are* Stack *objects. Explain why the overloaded assignment operator of* Section 4.3.2 *cannot be used in an expression such as* x = y = z. *Modify the prototype and implementation of the overloaded assignment operator so that this expression becomes valid.*

Answer The expression x = y = z is compiled as x = (y = z). The assignment inside the parentheses has **void** type (since this is what we have returned from the overloaded assignment operator) and therefore the outer assignment has mismatched arguments. In order to correct this, the assignment operator should return a reference to a Stack. A modified assignment operator could be implemented as follows.

```
Stack &Stack :: operator = (const Stack &original)  //   Overload assignment
/* Post: The Stack is reset as a copy of Stack original. */
{
   Node *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                                      //   Duplicate the linked nodes
      new_copy = new_top = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())                            //   Clean out old Stack entries
      pop();
   top_node = new_top;                         //   and replace them with new entries.
   return *this;
}
```

E2. *What is wrong with the following attempt to use the copy constructor to implement the overloaded assignment operator for a linked* Stack*?*

```
void Stack :: operator = (const Stack &original)
{
   Stack new_copy(original);
   top_node = new_copy.top_node;
}
```

How can we modify this code to give a correct implementation?

Answer The assignment top_node = new_copy.top_node; leaves the former nodes of the left hand side of the assignment operator as garbage. Moreover, when the function ends, the statically allocated Stack new_copy will be destructed, this will destroy the newly assigned left hand side of the assignment operator. We can avoid both problems by swapping the pointers top_node and new_copy.top_node as in the following.

```
void Stack :: operator = (const Stack &original)
/* Post: The Stack is reset as a copy of Stack original. */
{
   Stack new_copy(original);
   Node *temp = top_node;
   top_node = new_copy.top_node;
   new_copy.top_node = temp;            //  Make sure old stack value is deleted
                                        //  and that new stack value is kept.
}
```

4.4 LINKED QUEUES

Exercises 4.4

E1. *Write the following methods for linked queues:*

(a) *the method* empty,

Answer

```
bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return front == NULL;
}
```

(b) *the method* retrieve,

Answer

```
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue is reported in item. If the Queue is empty return an Error_code
         of underflow and leave the Queue unchanged. */
{
   if (front == NULL) return underflow;
   item = front->entry;
   return success;
}
```

(c) *the destructor,*

Answer

```
Queue :: ~Queue( )
{
   while (!empty( ))
      serve( );
}
```

(d) *the copy constructor,*

Answer

```
Queue :: Queue(const Queue &copy)
{
   Node *copy_node = copy.front;
   front = rear = NULL;
   while (copy_node != NULL) {
      append(copy_node->entry);
      copy_node = copy_node->next;
   }
}
```

(e) *the overloaded assignment operator.*

Answer

```
void Queue :: operator = (const Queue &copy)
{
   while (!empty())
      serve();
   Node *copy_node = copy.front;
   while (copy_node != NULL) {
      append(copy_node->entry);
      copy_node = copy_node->next;
   }
}
```

E2. *Write an implementation of the* Extended_queue *method* full*. In light of the simplicity of this method in the linked implementation, why is it still important to include it in the linked* **class** Extended_queue*?*

Answer If the abstract data type Extended_queue is to remain truly implementation independent, all of its original functions must remain available to an application program. This maintains the general integrity of the ADT.

```
bool Extended_queue :: full() const
{
   return false;
}
```

E3. *Write the following methods for the linked* **class** Extended_queue*:*

(a) clear*;*

Answer

```
void Extended_queue :: clear()
{
   while (!empty())
      serve();
}
```

(b) serve_and_retrieve*;*

Answer

```
Error_code Extended_queue :: serve_and_retrieve(Queue_entry &item)
/* Post: The front of the Queue is removed and reported in item. If the Queue is empty return
         an Error_code of underflow and leave the Queue unchanged. */
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   item = old_front->entry;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}
```

E4. *For a linked* Extended_queue*, the function* size *requires a loop that moves through the entire queue to count the entries, since the number of entries in the queue is not kept as a separate member in the class. Consider modifying the declaration of a linked* Extended_queue *to add a count data member to the class. What changes will need to be made to all the other methods of the class? Discuss the advantages and disadvantages of this modification compared to the original implementation.*

Answer The most dramatic change will be in size, where a single member count will give the size. The methods append and serve must include statements to increment or decrement the counter, as required, while the function empty need merely check for a Queue size of 0. The constructor and the method clear will also require an additional statement to set the counter to 0. As in the case of the linked stack, it is questionable whether the additional code is justified, given the limited use of the counter. Much would depend on the frequency with which the function size is called.

E5. *A **circularly linked list**, illustrated in Figure 4.14, is a linked list in which the node at the tail of the list, instead of having a* NULL *pointer, points back to the node at the head of the list. We then need only one pointer* tail *to access both ends of the list, since we know that* tail->next *points back to the head of the list.*

(a) *If we implement a queue as a circularly linked list, then we need only one pointer* tail *(or* rear*) to locate both the front and the rear. Write the methods needed to process a queue stored in this way.*

Answer The definition for the **class** Queue is as follows.

```
class Queue {
public:
//    standard Queue methods
   Queue( );
   bool empty( ) const;
   Error_code append(const Queue_entry &item);
   Error_code serve( );
   Error_code retrieve(Queue_entry &item) const;
//    safety features for linked structures
   ~Queue( );
   Queue(const Queue &original);
   void operator = (const Queue &original);
protected:
   Node *tail;
};
```

The implementations of the methods follow.

```
bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return tail == NULL;
}

Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   tail = NULL;
}

Error_code Queue :: append(const Queue_entry &item)
/* Post: Add item to the rear of the Queue and return a code of success or return a code of
         overflow if dynamic memory is exhausted. */
{
   Node *new_rear = new Node(item);
   if (new_rear == NULL) return overflow;
   if (tail == NULL) {
      tail = new_rear;
      tail->next = tail;
   }
   else {
      new_rear->next = tail->next;
      tail->next = new_rear;
      tail = new_rear;
   }
   return success;
}
```

```
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue is reported in item. If the Queue is empty return an Error_code
         of underflow and leave the Queue unchanged. */
{
   if (tail == NULL) return underflow;
   item = (tail->next)->entry;
   return success;
}

Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty, return an Error_code of
         underflow. */
{
   if (tail == NULL) return underflow;
   Node *old_front = tail->next;
   if (tail == old_front) tail = NULL;
   else tail->next = old_front->next;
   delete old_front;
   return success;
}

Queue :: ~Queue( )
{
   while (!empty( ))
      serve( );
}

Queue :: Queue(const Queue &copy)
{
   tail = NULL;
   if (copy.tail == NULL) return;
   Node *copy_node = (copy.tail)->next;
   do {
      append(copy_node->entry);
      copy_node = copy_node->next;
   } while (copy_node != (copy.tail)->next);
}

void Queue :: operator = (const Queue &copy)
{
   while (!empty( ))
      serve( );
   tail = NULL;
   if (copy.tail == NULL) return;
   Node *copy_node = (copy.tail)->next;
   do {
      append(copy_node->entry);
      copy_node = copy_node->next;
   } while (copy_node != (copy.tail)->next);
}
```

(b) *What are the disadvantages of implementing this structure, as opposed to using the version requiring two pointers?*

Answer Extra effort is required to manipulate the queue when there is only one pointer. For instance, removing the entry at the front of the queue requires looking for the node that follows the tail, as compared to the straightforward clarity of looking for the node pointed to by the front pointer. In general, the savings to be had by the use of one fewer pointer is not worth the extra programming effort or the lack of clarity in the operations on the queue.

Programming Projects 4.4

P1. *Assemble specification and method files, called* queue.h *and* queue.c*, for linked queues, suitable for use by an application program.*

Answer The driver program (also for Project P2) is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
typedef char Node_entry;
#include "../nodes/node.h"
#include "../nodes/node.cpp"
typedef Node_entry Queue_entry;
#include "queue.h"
#include "queue.cpp"
#include "extqueue.h"
#include "extqueue.cpp"

void introduction();
void help();
char get_command();
int do_command(char, Extended_queue &);

main()
/*
PRE:  None.
POST: A linked Queue demonstration has been performed.
USES: get_command, do_command, class Extended_queue
      (linked implementation)
*/

{
   Extended_queue test_queue;
   introduction();
   while (do_command(get_command(), test_queue));
}

void introduction()
/*
PRE:  None.
POST: Instructions for the Queue testing program are printed
*/

{
   cout << "\n\t\tExtended Queue Testing Program\n\n"
      << "The program demonstrates an extended queue of "
      << "single character keys. \nAdditions occur at "
      << "the end of the queue, while deletions can only "
      << "be\ndone at the front. The queue can hold a "
      << "any number of characters.\n\n"
      << "Valid commands are to be entered at each prompt.\n"
      << "Both upper and lower case letters can be used.\n"
      << "At the command prompt press H for help.\n\n";
}

void help()
/*
PRE:  None.
POST: Instructions for the Queue operations have been printed.
*/
```

```
{
   char c;
   cout << "\nThis program allows one command to be entered on each line.\n"
    << "For example, if the command S is entered at the command line\n"
    << "then the program will serve the front of the queue."
    << "\nValid commands are:\n"
    << "\tS - Serve the front of the extended queue\n"
    << "\t# - The current size of the extended queue\n"
    << "\tA - Append the next input character to the extended queue\n"
    << "\tC - Clear the extended queue (same as delete)\n"
    << "\tH - This help screen\n"
    << "\tQ - Quit\n"
    << "\tP - Print the extended queue\n"
    << "\tR - Retrieve and print the front entry of the extended queue\n"
    << "Press <Enter> to continue.";
   do {
      cin.get(c);
   } while (c != '\n');
}

char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands for the
      Queue demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if (c == 's' || c == '#' || c == 'a' || c == 'c' ||
         c == 'h' || c == 'q' || c == 'p' || c == 'r') {
         return (c);
      }
      cout << "Please enter a valid command or H for help:";
      cout << "\n\t[S]erve  entry\t[P]rint queue\t[#] size of queue\n"
         << "\t[C]lear queue\t[R]irst entry\t[A]ppend entry\n"
         << "\t[H]elp\t[Q]uit.\n";
   }
}

int do_command(char c, Extended_queue &test_queue)
/*
PRE:  The Extended_queue has been created and c is a valid Queue
      operation.
POST: The command c has been executed.
USES: The class Extended_queue.
*/
```

```
{
   Queue_entry x;
   switch (c) {
   case 'r':
      if (test_queue.retrieve(x) == fail) cout << "Queue is empty.\n";
      else cout << "\nThe first entry is: " << x << endl;
      break;

   case 's':
      if (test_queue.empty()) cout << "Queue is empty.\n";
      else test_queue.serve();
      break;

   case 'a':
      if (test_queue.full()) cout << "Sorry, queue is full.\n";
      else {
         cout << "Enter new key to insert:";
         cin.get(x);
         test_queue.append(x);
      }
      break;

   case 'c':
      test_queue.clear();
      cout << "Queue is cleared.\n";
      break;

   case '#':
      cout << "The size of the queue is " << test_queue.size() << "\n";
      break;

   case 'h':
      help();
      break;

   case 'q':
      cout << "Extended queue demonstration finished.\n";
      return (0);
   case 'p':
      int sz = test_queue.size();
      if (sz == 0) cout << "Queue is empty.\n";
      else {
         cout << "\nThe queue contains:\n";
         for (int i = 0; i < sz; i++) {
            test_queue.retrieve(x);
            test_queue.append(x);
            cout << "  " << x;
            test_queue.serve();
         }
         cout << endl;
      }
      break;
   //  additional cases cover other commands
   }
   return (1);
}
```

The (linked) Queue implementation follows. Header file:

```
class Queue {
public:
//  standard Queue methods
   Queue();
   bool empty() const;
   Error_code append(const Queue_entry &item);
   Error_code serve();
   Error_code retrieve(Queue_entry &item) const;
//  safety features for linked structures
   ~Queue();
   Queue(const Queue &original);
   void operator =(const Queue &original);
protected:
   Node *front, *rear;
};
```

Code file:

```
bool Queue::empty() const
/*
Post: Return true if the Queue is empty,
      otherwise return false.
*/
{
   return front == NULL;
}

Queue::Queue()
/*
Post: The Queue is initialized to be empty.
*/
{
   front = rear = NULL;
}

Error_code Queue::append(const Queue_entry &item)
/*
Post: Add item to the rear of the Queue and
      return a code of success or return a code
      of overflow if dynamic memory is exhausted.
*/
{
   Node *new_rear = new Node(item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}

Error_code Queue::retrieve(Queue_entry &item) const
/*
Post: The front of the Queue is reported
      in item. If the Queue
      is empty return an Error_code
      of underflow and leave the Queue unchanged.
*/
```

```
{
   if (front == NULL) return underflow;
   item = front->entry;
   return success;
}

Error_code Queue::serve()
/*
Post: The front of the Queue is removed.  If the Queue
is empty, return an Error_code of underflow.
*/

{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}

Queue::~Queue()
{
   while (!empty())
      serve();
}

Queue::Queue(const Queue &copy)
{
   Node *copy_node = copy.front;
   front = rear = NULL;
   while (copy_node != NULL) {
      append(copy_node->entry);
      copy_node = copy_node->next;
   }
}

void Queue::operator =(const Queue &copy)
{
   while (!empty())
      serve();
   Node *copy_node = copy.front;
   while (copy_node != NULL) {
      append(copy_node->entry);
      copy_node = copy_node->next;
   }
}
```

P2. *Take the menu-driven demonstration program for an* Extended_queue *of characters in Section 3.4 and substitute the linked* Extended_queue *implementation files for the files implementing contiguous queues. If you have designed the program and the classes carefully, then the program should work correctly with no further change.*

Answer The driver program for Project P1 works just as well here. The (linked) extended queue implementation follows. Header file:

```
class Extended_queue: public Queue {
public:
   bool full() const;
   int size() const;
   void clear();
   Error_code serve_and_retrieve(Queue_entry &item);
};
```

Code file:

```
Error_code Extended_queue::serve_and_retrieve(Queue_entry &item)
/*
Post: The front of the Queue is removed and reported
      in item. If the Queue
      is empty return an Error_code
      of underflow and leave the Queue unchanged.
*/

{
   if (front == NULL) return underflow;
   Node *old_front = front;
   item = old_front->entry;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}

bool Extended_queue::full() const
{
   return false;
}

void Extended_queue::clear()
{
   while (!empty())
      serve();
}

int Extended_queue::size() const
/*
Post: Return the number of entries in the Extended_queue.
*/

{
   Node *window = front;
   int count = 0;
   while (window != NULL) {
      window = window->next;
      count++;
   }
   return count;
}
```

P3. *In the airport simulation developed in Section 3.5, replace the implementations of contiguous queues with linked versions. If you have designed the classes carefully, the program should run in exactly the same way with no further change required.*

Answer This program only differs from that of Section 3.5 in its inclusion of a different set of files for a Queue implementation.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../3/airport/plane.h"
#include "../../3/airport/plane.cpp"
typedef Plane Node_entry;
#include "../nodes/node.h"
#include "../nodes/node.cpp"
typedef Node_entry Queue_entry;
#include "../linkdq/queue.h"
#include "../linkdq/queue.cpp"
#include "../linkdq/extqueue.h"
#include "../linkdq/extqueue.cpp"
#include "../../3/airport/runway.h"
#include "../../3/airport/runway.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/*
Pre:  The user specifies the number of time units in the simulation,
      the maximal queue sizes permitted,
      and the expected arrival and departure rates for the airport.
Post: The program prints instructions and initializes the parameters
      end_time, queue_limit, arrival_rate, and departure_rate to
      the specified values.
Uses: utility function user_says_yes
*/

{
   cout << "This program simulates an airport with only one runway." << endl
        << "One plane can land or depart in each unit of time." << endl;
   cout << "Up to what number of planes can be waiting to land "
        << "or take off at any time? " << flush;
   cin  >> queue_limit;

   cout << "How many units of time will the simulation run?" << flush;
   cin  >> end_time;

   bool acceptable;
   do {
      cout << "Expected number of arrivals per unit time?" << flush;
      cin  >> arrival_rate;
      cout << "Expected number of departures per unit time?" << flush;
      cin  >> departure_rate;

      if (arrival_rate < 0.0 || departure_rate < 0.0)
         cerr << "These rates must be nonnegative." << endl;
      else
         acceptable = true;

      if (acceptable && arrival_rate + departure_rate > 1.0)
         cerr << "Safety Warning: This airport will become saturated. "
              << endl;

   } while (!acceptable);
}
```

```
void run_idle(int time)
/*
Post: The specified time is printed with a message that the runway is idle.
*/

{
   cout << time << ": Runway is idle." << endl;
}

int main()     //  Airport simulation program
/*
Pre:  The user must supply the number of time intervals the simulation
      is to run, the expected number of planes arriving, the expected
      number of planes departing per time interval, and the
      maximum allowed size for runway queues.
Post: The program performs a random simulation of the airport, showing
      the status of the runway at each time interval, and prints out a
      summary of airport operation at the conclusion.
Uses: Classes Runway, Plane, Random and
      functions run_idle, initialize.
*/

{
   int end_time;             //  time to run simulation
   int queue_limit;          //  size of Runway queues
   int flight_number = 0;
   double arrival_rate, departure_rate;
   initialize(end_time, queue_limit, arrival_rate, departure_rate);
   Random variable;

   Runway small_airport(queue_limit);
   for (int current_time = 0; current_time < end_time; current_time++) {
        //  loop over time intervals
      int number_arrivals = variable.poisson(arrival_rate);
        //  current arrival requests
      for (int i = 0; i < number_arrivals; i++) {
         Plane current_plane(flight_number++, current_time, arriving);
         if (small_airport.can_land(current_plane) != success)
            current_plane.refuse();
      }

      int number_departures= variable.poisson(departure_rate);
        //  current departure requests
      for (int j = 0; j < number_departures; j++) {
         Plane current_plane(flight_number++, current_time, departing);
         if (small_airport.can_depart(current_plane) != success)
            current_plane.refuse();
      }

      Plane moving_plane;
      switch (small_airport.activity(current_time, moving_plane)) {
        //  Let at most one Plane onto the Runway at current_time.
      case land:
         moving_plane.land(current_time);
         break;

      case take_off:
         moving_plane.fly(current_time);
         break;
      case idle:
         run_idle(current_time);
      }
   }
   small_airport.shut_down(end_time);
}
```

4.5 APPLICATION: POLYNOMIAL ARITHMETIC

Exercise 4.5

E1. *Discuss the steps that would be needed to extend the polynomial calculator so that it would process polynomials in several variables.*

Answer Most of the operations can be implemented for polynomials in several variables with only obvious changes. The major exception is division. Consider, for example, the polynomials x^2y and xy^2. If one were divided into the other, what would the quotient and remainder be? (Remember that the degree of the remainder must be strictly less than that of the divisor.) The precise mathematical reason why the usual division will not work is that the polynomials in more than one variable over the real numbers do not form a Euclidean ring. In regard to data representation, if a bound on the number of variables is given, then the node type representing a term of the polynomial can be expanded to include as many exponent members as there are variables. The nodes should be arranged so that their exponent members are decreasing under lexicographic order; that is, one term should come before a second provided that in the first place where their exponent members differ, the first term has a larger exponent. If there were no bound on the number of variables, then some kind of linked queue would be required to keep the exponents, and the data representation would become much more complicated (and probably not very useful).

Programming Projects 4.5

P1. *Assemble the functions developed in this section and make the necessary changes in the code so as to produce a working skeleton for the calculator program, one that will read, write, and add polynomials. You will need to supply the functions* get_command(), introduction()*, and* instructions()*.*

Answer The solutions to Projects P1, P2, P3, P4, and P5 are implemented as follows. The driver is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "term.h"
typedef Term Node_entry;
#include "../nodes/node.h"
#include "../nodes/node.cpp"
typedef Node_entry Queue_entry;
#include "../linkdq/queue.h"
#include "../linkdq/queue.cpp"
#include "../linkdq/extqueue.h"
#include "../linkdq/extqueue.cpp"
#include "polynomi.h"
#include "polynomi.cpp"
typedef Polynomial Stack_entry;
#include "../../2/stack/stack.h"
#include "../../2/stack/stack.cpp"

void introduction()
/*
PRE:  None.
POST: An introduction to the program Polynomial Calculator is printed.
*/
```

```
{
 cout << "Polynomial Calculator Program." << endl
      << "This program simulates a polynomial calculator that works on a\n"
      << "stack and a list of operations.  It models a reverse Polish\n"
      << "calculator where operands are entered before the operators. An\n"
      << "example to add two polynomials and print the answer is ?P?Q+= .\n"
      << "P and Q are the operands to be entered, + is add, and = is\n"
      << "print result. The result is put onto the calculator's stack.\n\n";
}

void instructions()
/*
PRE:  None.
POST: Prints out the instructions and the operations allowed on the
      calculator.
*/

{
  cout << "\nEnter a string of instructions in reverse Polish form.\n"
       << "The allowable instructions are:\n\n"
       << "?:Read       +:Add           =:Print      -:Subtract\n"
       << "*:Multiply   q:Quit          /:Divide        h:Help\n\n";
}

char get_command()
/*
PRE:  None.
POST: A legal command is read from the user and returned.
*/

{
   char command, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(command);
      } while (command == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == 'h' || command == '*' ||
          command == '/' || command == 'q' || command == 'p' ||
          command == 'h') {
         return (command);
      }
      cout << "Please enter a valid command:" << endl;
      instructions();
   }
}

bool do_command(char command, Stack &stored_polynomials)
/*
Pre:  The first parameter specifies a valid
      calculator command.
```

```
Post: The command specified by the first parameter
      has been applied to the Stack of Polynomial
      objects given by the second parameter.
      A result of true is returned unless
      command == 'q'.
Uses: The classes Stack and Polynomial.
*/

{
   Polynomial p, q, r;
   switch (command) {
   case '?':
      p.read();
      if (stored_polynomials.push(p) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
      break;

   case '=':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         p.print();
      }
      break;

   case '+':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         stored_polynomials.pop();
         if (stored_polynomials.empty()) {
            cout << "Stack has just one polynomial" << endl;
            stored_polynomials.push(p);
         }

         else {
            stored_polynomials.top(q);
            stored_polynomials.pop();
            r.equals_sum(q, p);
            if (stored_polynomials.push(r) == overflow)
               cout << "Warning: Stack full, lost polynomial" << endl;
         }
      }
      break;

   //   Add options for further user commands.
```

```
case '/':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         if (r.equals_quotient(q, p) != success) {
            cerr << "Division by 0 fails; no action done." << endl;
            stored_polynomials.push(q);
            stored_polynomials.push(p);
         }
         else if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

case '-':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         r.equals_difference(q, p);
         if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

case '*':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();

      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
```

TOC
Index
Help

```
            else {
               stored_polynomials.top(q);
               stored_polynomials.pop();
               r.equals_product(q, p);
               if (stored_polynomials.push(r) == overflow)
                  cout << "Warning: Stack full, lost polynomial" << endl;
            }
         }
         break;

      case 'h':
         instructions();
         break;

      case 'q':
         cout << "Calculation finished." << endl;
         return false;
   }
   return true;
}

int main()
/*
The program has executed simple polynomial arithmetic
commands entered by the user.
Uses: The classes Stack and Polynomial and the functions
      introduction, instructions, do_command, and
      get_command.
*/

{
   Stack stored_polynomials;
   introduction();
   instructions();
   while (do_command(get_command(), stored_polynomials));
}
```

The polynomial class, including the methods required in projects P2, P3, P4, P5 is implemented as follows. Auxiliary header file:

```
struct Term {
   int degree;
   double coefficient;
   Term (int exponent = 0, double scalar = 0);
};

Term::Term(int exponent, double scalar)
/*
Post: The Term is initialized with the given coefficient and exponent,
      or with default parameter values of 0.
*/
{
   degree = exponent;
   coefficient = scalar;
}
```

Header file:

```
class Polynomial: private Extended_queue {  //  Use private inheritance.
public:
   void read();
   void print() const;
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   Error_code equals_quotient(Polynomial p, Polynomial q);
   int degree() const;
private:
   void mult_term(Polynomial p, Term t);
};
```

Code file:

```
void Polynomial::mult_term(Polynomial p, Term t)
{
   clear();
   while (!p.empty()) {
      Term pterm;
      p.serve_and_retrieve(pterm);
      Term answer_term(pterm.degree + t.degree,
                       t.coefficient * pterm.coefficient);
      append(answer_term);
   }
}

int Polynomial::degree() const
/*
Post: If the Polynomial is identically 0, a result of -1 is returned.
      Otherwise the degree of the Polynomial is returned.

*/

{
   if (empty()) return -1;
   Term lead;
   retrieve(lead);
   return lead.degree;
}

void Polynomial::print() const
/*
Post: The Polynomial is printed to cout.
*/

{
   Node *print_node = front;
   bool first_term = true;

   while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (first_term) {  //  In this case, suppress printing an initial '+'.
         first_term = false;
         if (print_term.coefficient < 0) cout << "- ";
      }

      else if (print_term.coefficient < 0) cout << " - ";
      else cout << " + ";
      double r = (print_term.coefficient >= 0)
                 ? print_term.coefficient : -(print_term.coefficient);
      if (r != 1) cout << r;
      if (print_term.degree > 1) cout << " X^" << print_term.degree;
      if (print_term.degree == 1) cout << " X";
      if (r == 1 && print_term.degree == 0) cout << " 1";
      print_node = print_node->next;
   }
   if (first_term)
      cout << "0";  //  Print 0 for an empty Polynomial.
   cout << endl;
}

void Polynomial::read()
/*
Post: The Polynomial is read from cin.
*/

{
   clear();
   double coefficient;
   int last_exponent, exponent;
   bool first_term = true;

   cout << "Enter the coefficients and exponents for the polynomial, "
        << "one pair per line.  Exponents must be in descending order."
        << endl
        << "Enter a coefficient of 0 or an exponent of 0 to terminate."
        << endl;

   do {
     cout << "coefficient? " << flush;
     cin  >> coefficient;
     if (coefficient != 0.0) {

        cout << "exponent? " << flush;
        cin  >> exponent;
        if ((!first_term && exponent >= last_exponent) || exponent < 0) {
           exponent = 0;
           cout << "Bad exponent: Polynomial ends without its last term."
                << endl;
        }

        else {
           Term new_term(exponent, coefficient);
           append(new_term);
           first_term = false;
        }
        last_exponent = exponent;
     }
   } while (coefficient != 0.0 && exponent != 0);
}

void Polynomial::equals_sum(Polynomial p, Polynomial q)
/*
Post: The Polynomial object
is reset as the sum of the two parameters.
*/
```

```
{
   clear();
   while (!p.empty() || !q.empty()) {
      Term p_term, q_term;
      if (p.degree() > q.degree()) {
         p.serve_and_retrieve(p_term);
         append(p_term);
      }

      else if (q.degree() > p.degree()) {
         q.serve_and_retrieve(q_term);
         append(q_term);
      }

      else {
         p.serve_and_retrieve(p_term);
         q.serve_and_retrieve(q_term);
         if (p_term.coefficient + q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                             p_term.coefficient + q_term.coefficient);
            append(answer_term);
         }
      }
   }
}

void Polynomial::equals_difference(Polynomial p, Polynomial q)
{
   clear();
   Term p_term, q_term;
   while (!p.empty() || !q.empty()) {
      if (p.degree() > q.degree()) {
         p.serve_and_retrieve(p_term);
         append(p_term);
      }
      else if (q.degree() > p.degree()) {
         q.serve_and_retrieve(q_term);
         q_term.coefficient = -q_term.coefficient;
         append(q_term);
      }
      else {
         p.serve_and_retrieve(p_term);
         q.serve_and_retrieve(q_term);
         if (p_term.coefficient - q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                             p_term.coefficient - q_term.coefficient);
            append(answer_term);
         }
      }
   }
}

void Polynomial::equals_product(Polynomial p, Polynomial q)
```

TOC
Index
Help

```
{
   clear();
   Term p_term, term;
   Polynomial partial_prod, old_sum, new_sum;
   while (p.serve_and_retrieve(p_term) == success) {
      partial_prod.mult_term(q, p_term);
      new_sum.equals_sum(old_sum, partial_prod);
      old_sum = new_sum;
   }
   while (new_sum.serve_and_retrieve(term) == success) append(term);
}

Error_code Polynomial::equals_quotient(Polynomial p, Polynomial q)
{
   clear();
   if (q.empty()) return (fail);    //  divide by 0
   if (p.degree() < q.degree()) return (success);
   Term p_term, q_term, term;
   Polynomial partial_prod, remainder, tail;
   p.serve_and_retrieve(p_term);
   q.retrieve(q_term);
   Term lead_term(p_term.degree - q_term.degree,
                  p_term.coefficient / q_term.coefficient);
   append(lead_term);
   partial_prod.mult_term(q, lead_term);
   partial_prod.serve_and_retrieve(term);
   remainder.equals_difference(p, partial_prod);
   tail.equals_quotient(remainder, q);
   while (tail.serve_and_retrieve(term) == success) append(term);
   return (success);
}
```

TOC
Index
Help

P2. *Write the* Polynomial *method* equals_difference *and integrate it into the calculator.*

Answer See the solution to Project P1.

P3. *Write an auxiliary function*

void Polynomial :: mult_term(Polynomial p, Term t)

that calculates a Polynomial *object by multiplying* p *by the single* Term t.

Answer See the solution to Project P1.

P4. *Use the function developed in the preceding problem, together with the* Polynomial *method* equals_sum, *to write the* Polynomial *method* equals_product, *and integrate the resulting method into the calculator.*

Answer See the solution to Project P1.

P5. *Write the* Polynomial *method* equals_quotient *and integrate it into the calculator.*

Answer See the solution to Project P1.

P6. *Many reverse Polish calculators use not only a stack but also provide memory locations where operands can be stored. Extend the project to provide an array to store polynomials. Provide additional commands to store the top of the stack into an array entry and to push the polynomial in an array entry onto the stack. The array should have 100 entries, and all 100 positions should be initialized to the zero polynomial when the program begins. The functions that access the array should ask the user which entry to use.*

Answer An augmented polynomial calculator that admits the operations described in projects P6, P7, P8, P9, P10, P11, and P12 is implemented by:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../polycalc/term.h"
typedef Term Node_entry;
#include "../nodes/node.h"
#include "../nodes/node.cpp"
typedef Node_entry Queue_entry;
#include "../linkdq/queue.h"
#include "../linkdq/queue.cpp"
#include "../linkdq/extqueue.h"
#include "../linkdq/extqueue.cpp"
#include "polynomi.h"
#include "polynomi.cpp"
typedef Polynomial Stack_entry;
#include "../../2/stack/stack.h"
#include "../../2/stack/stack.cpp"
#include "auxil.cpp"

Polynomial memory[100];   // global array for calculator memory

void introduction()
/*
PRE:  None.
POST: An introduction to the program Polynomial Calculator is printed.
*/

{
 cout << "Polynomial Calculator Program." << endl
      << "This program simulates a polynomial calculator that works on a\n"
      << "stack and a list of operations.  It models a reverse Polish\n"
      << "calculator where operands are entered before the operators. An\n"
      << "example to add two polynomials and print the answer is ?P?Q+= .\n"
      << "P and Q are the operands to be entered, + is add, and = is\n"
      << "print result. The result is put onto the calculator's stack.\n\n";
}

void instructions()
/*
PRE:  None.
POST: Prints out the instructions and the operations allowed on the
      calculator.
*/

{
  cout << "\nEnter a string of instructions in reverse Polish form.\n"
       << "The allowable instructions are:\n\n"
       << "?:Read       +:Add           =:Print       -:Subtract\n"
       << "*:Multiply   q:Quit          /:Divide      h:Help\n\n"
       << "$:Evaluate m:Get from memory ~:Derivative  s:Save to memory\n"
       << "d:Discard    a:Add all       i:Interchange %:Remainder\n";
}

char get_command()
/*
PRE:  None.
POST: A legal command is read from the user and returned.
*/
```

TOC
Index
Help

```
{
   char command, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(command);
      } while (command == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == 'h' || command == '*' ||
          command == '/' || command == 'q' || command == 'p' ||
          command == 's' || command == 'm' || command == 'd' ||
          command == 'i' || command == 'a' || command == '~' ||
          command == '$' || command == '%' || command == 'h') {
         return (command);
      }
      cout << "Please enter a valid command:" << endl;
      instructions();
   }
}

bool do_command(char command, Stack &stored_polynomials)
/*
Pre:  The first parameter specifies a valid
      calculator command.
Post: The command specified by the first parameter
      has been applied to the Stack of Polynomial
      objects given by the second parameter.
      A result of true is returned unless
      command == 'q'.
Uses: The classes Stack and Polynomial.
*/

{
   Polynomial p, q, r;
   int slot;
   double argument;

   switch (command) {
   case '?':
      p.read();
      if (stored_polynomials.push(p) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
      break;

   case '=':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         p.print();
      }
      break;
```

```
case '+':
   if (stored_polynomials.empty())
      cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }

      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         r.equals_sum(q, p);
         if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

//   Add options for further user commands.

case '/':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         if (r.equals_quotient(q, p) != success) {
            cerr << "Division by 0 fails; no action done." << endl;
            stored_polynomials.push(q);
            stored_polynomials.push(p);
         }
         else if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;
```

```
case '-':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         r.equals_difference(q, p);
         if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

case '*':
   if (stored_polynomials.empty()) cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }
      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         r.equals_product(q, p);
         if (stored_polynomials.push(r) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

case 'h':
   instructions();
   break;

case 's':
   if (stored_polynomials.empty())
      cout << "There is no polynomial to store" << endl;
   else {
      cout << "\nAt which memory slot do you want to store?" << endl;
      do {
        cout << "Enter a choice between 0 and 99" << endl;
        cin >> slot;
      } while (slot < 0 || slot > 99);
      stored_polynomials.top(memory[slot]);
   }
   break;
```

```
case 'm':
   cout << "\nWhich memory slot do you want to stack?" << endl;
   do {
     cout << "Enter a choice between 0 and 99" << endl;
     cin >> slot;
   } while (slot < 0 || slot > 99);
   stored_polynomials.push(memory[slot]);
   break;

case 'd':
   stored_polynomials.pop();
   break;

case 'i':
   if (stored_polynomials.empty())
      cout << "Stack empty" << endl;
   else {
      stored_polynomials.top(p);
      stored_polynomials.pop();
      if (stored_polynomials.empty()) {
         cout << "Stack has just one polynomial" << endl;
         stored_polynomials.push(p);
      }

      else {
         stored_polynomials.top(q);
         stored_polynomials.pop();
         if (stored_polynomials.push(p) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
         if (stored_polynomials.push(q) == overflow)
            cout << "Warning: Stack full, lost polynomial" << endl;
      }
   }
   break;

case 'a':
   p = sum_stack(stored_polynomials);
   if (stored_polynomials.push(p) == overflow)
      cout << "Warning: Stack full, lost polynomial" << endl;
   break;

case '*':
   if (!stored_polynomials.empty()) {
      stored_polynomials.top(p);
      q.equals_deriv(p);
      if (stored_polynomials.push(q) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
   }
   break;

case '$':
   if (!stored_polynomials.empty()) {
      stored_polynomials.top(p);
      cout << "Give an argument at which to evaluating the polynomial: "
           << flush;
      cin >> argument;
      cout << p.evaluate(argument) << endl;
   }
   break;
   case '%':
      if (stored_polynomials.empty()) cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         stored_polynomials.pop();
         if (stored_polynomials.empty()) {
            cout << "Stack has just one polynomial" << endl;
            stored_polynomials.push(p);
         }
         else {
            stored_polynomials.top(q);
            stored_polynomials.pop();
            if (r.equals_remainder(q, p) != success) {
               cerr << "Division by 0 fails; no action done." << endl;
               stored_polynomials.push(q);
               stored_polynomials.push(p);
            }
            else if (stored_polynomials.push(r) == overflow)
               cout << "Warning: Stack full, lost polynomial" << endl;
         }
      }
      break;

   case 'q':
      cout << "Calculation finished." << endl;
      return false;
   }
   return true;
}

int main()
/*
The program has executed simple polynomial arithmetic
commands entered by the user.
Uses: The classes Stack and Polynomial and the functions
      introduction, instructions, do_command, and
      get_command.
*/

{
   Stack stored_polynomials;
   introduction();
   instructions();
   while (do_command(get_command(), stored_polynomials));
}
```

The function `do_command` in `main.cpp` covers the solution to projects P6, P7, P8.

P7. *Write a function that will discard the top polynomial on the stack, and include this capability as a new command.*

Answer See the solution to Project P6.

P8. *Write a function that will interchange the top two polynomials on the stack, and include this capability as a new command.*

Answer See the solution to Project P6.

P9. *Write a function that will add all the polynomials on the stack together, and include this capability as a new command.*

Answer See the solution to Project P6, together with:

```
Polynomial sum_stack(Stack &data)
/*
Post: The sum of the entries in a Stack is returned,
      and the Stack is restored to its original state.
*/

{
   Polynomial p, sum, temp;
   Stack temp_copy;
   while (!data.empty()) {
       data.top(p);
       temp_copy.push(p);
       temp.equals_sum(sum,p);
       sum = temp;
       data.pop();
   }
   while (!temp_copy.empty()) {
       temp_copy.top(p);
       data.push(p);
       temp_copy.pop();
   }
   return sum;
}
```

P10. *Write a function that will compute the derivative of a polynomial, and include this capability as a new command.*

Answer An augmented polynomial class with new methods that implement solutions to Projects P10, P11, and P12 follows. Header file:

```
class Polynomial: private Extended_queue {  //  Use private inheritance.
public:
   void read();
   void print() const;
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   Error_code equals_quotient(Polynomial p, Polynomial q);
   int degree() const;
   void equals_deriv(Polynomial p);
   double evaluate(double x);
   Error_code equals_remainder(Polynomial p, Polynomial q);
private:
   void mult_term(Polynomial p, Term t);
};
```

Code file:

```
void Polynomial::mult_term(Polynomial p, Term t)
{
   clear();
   while (!p.empty()) {
      Term pterm;
      p.serve_and_retrieve(pterm);
      Term answer_term(pterm.degree + t.degree,
                       t.coefficient * pterm.coefficient);
      append(answer_term);
   }
}

int Polynomial::degree() const
/*
Post: If the Polynomial is identically 0, a result of -1 is returned.
      Otherwise the degree of the Polynomial is returned.
*/

{
   if (empty()) return -1;
   Term lead;
   retrieve(lead);
   return lead.degree;
}

void Polynomial::print() const
/*
Post: The Polynomial is printed to cout.
*/

{
   Node *print_node = front;
   bool first_term = true;

   while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (first_term) {  //  In this case, suppress printing an initial '+'.
         first_term = false;
         if (print_term.coefficient < 0) cout << "- ";
      }

      else if (print_term.coefficient < 0) cout << " - ";
      else cout << " + ";

      double r = (print_term.coefficient >= 0)
                 ? print_term.coefficient : -(print_term.coefficient);
      if (r != 1) cout << r;
      if (print_term.degree > 1) cout << " X^" << print_term.degree;
      if (print_term.degree == 1) cout << " X";
      if (r == 1 && print_term.degree == 0) cout << " 1";
      print_node = print_node->next;
   }
   if (first_term)
      cout << "0";  //  Print 0 for an empty Polynomial.
   cout << endl;
}

void Polynomial::read()
/*
Post: The Polynomial is read from cin.
*/
```

TOC
Index
Help

```
{
   clear();
   double coefficient;
   int last_exponent, exponent;
   bool first_term = true;

   cout << "Enter the coefficients and exponents for the polynomial, "
        << "one pair per line.  Exponents must be in descending order."
        << endl
        << "Enter a coefficient of 0 or an exponent of 0 to terminate."
        << endl;

   do {

      cout << "coefficient? " << flush;
      cin  >> coefficient;
      if (coefficient != 0.0) {

         cout << "exponent? " << flush;
         cin  >> exponent;
         if ((!first_term && exponent >= last_exponent) || exponent < 0) {
            exponent = 0;
            cout << "Bad exponent: Polynomial ends without its last term."
                 << endl;
         }

         else {
            Term new_term(exponent, coefficient);
            append(new_term);
            first_term = false;
         }
         last_exponent = exponent;
      }
   } while (coefficient != 0.0 && exponent != 0);
}

void Polynomial::equals_sum(Polynomial p, Polynomial q)
/*
Post: The Polynomial object
      is reset as the sum of the two parameters.
*/

{
   clear();
   while (!p.empty() || !q.empty()) {
      Term p_term, q_term;
      if (p.degree() > q.degree()) {
         p.serve_and_retrieve(p_term);
         append(p_term);
      }

      else if (q.degree() > p.degree()) {
         q.serve_and_retrieve(q_term);
         append(q_term);
      }
```

```
         else {
            p.serve_and_retrieve(p_term);
            q.serve_and_retrieve(q_term);
            if (p_term.coefficient + q_term.coefficient != 0) {
               Term answer_term(p_term.degree,
                                p_term.coefficient + q_term.coefficient);
               append(answer_term);
            }
         }
      }
   }

   void Polynomial::equals_difference(Polynomial p, Polynomial q)
   {
      clear();
      Term p_term, q_term;
      while (!p.empty() || !q.empty()) {
         if (p.degree() > q.degree()) {
            p.serve_and_retrieve(p_term);
            append(p_term);
         }
         else if (q.degree() > p.degree()) {
            q.serve_and_retrieve(q_term);
            q_term.coefficient = -q_term.coefficient;
            append(q_term);
         }
         else {
            p.serve_and_retrieve(p_term);
            q.serve_and_retrieve(q_term);
            if (p_term.coefficient - q_term.coefficient != 0) {
               Term answer_term(p_term.degree,
                                p_term.coefficient - q_term.coefficient);
               append(answer_term);
            }
         }
      }
   }

   void Polynomial::equals_product(Polynomial p, Polynomial q)
   {
      clear();
      Term p_term, term;
      Polynomial partial_prod, old_sum, new_sum;
      while (p.serve_and_retrieve(p_term) == success) {
         partial_prod.mult_term(q, p_term);
         new_sum.equals_sum(old_sum, partial_prod);
         old_sum = new_sum;
      }
      while (new_sum.serve_and_retrieve(term) == success) append(term);
   }

   Error_code Polynomial::equals_quotient(Polynomial p, Polynomial q)
```

```
{
   clear();
   if (q.empty()) return (fail);     //  divide by 0
   if (p.degree() < q.degree()) return (success);
   Term p_term, q_term, term;
   Polynomial partial_prod, remainder, tail;
   p.serve_and_retrieve(p_term);
   q.retrieve(q_term);
   Term lead_term(p_term.degree - q_term.degree,
                  p_term.coefficient / q_term.coefficient);
   append(lead_term);
   partial_prod.mult_term(q, lead_term);
   partial_prod.serve_and_retrieve(term);
   remainder.equals_difference(p, partial_prod);
   tail.equals_quotient(remainder, q);
   while (tail.serve_and_retrieve(term) == success) append(term);
   return success;
}

void Polynomial::equals_deriv(Polynomial p)
{
   clear();
   Term pterm;
   while (p.serve_and_retrieve(pterm) == success) {
      Term answer_term(pterm.degree - 1, pterm.degree * pterm.coefficient);
      if (pterm.degree > 0) append(answer_term);
   }
}

double Polynomial::evaluate(double x)
{
   double  answer = 0.0;
   Polynomial p = *this;
   Term pterm;
   while (p.serve_and_retrieve(pterm) == success) {
      double addon = 1.0;
      for (int i = 0; i < pterm.degree; i++)
         addon *= x;
      addon *= pterm.coefficient;
      answer += addon;
   }
   return answer;
}

Error_code Polynomial::equals_remainder(Polynomial p, Polynomial q)
{
   Polynomial div;
   if (div.equals_quotient(p, q) != success) return fail;
   Polynomial pro;
   pro.equals_product(div, q);
   equals_difference(p, pro);
   return success;
}
```

P11. *Write a function that, given a polynomial and a real number, evaluates the polynomial at that number, and include this capability as a new command.*

Answer See the solution to Project P6 and P10.

P12. *Write a new method* equals_remainder *that obtains the remainder when a first* Polynomial *argument is divided by a second* Polynomial *argument. Add a new user command* % *to the calculator program to call this method.*

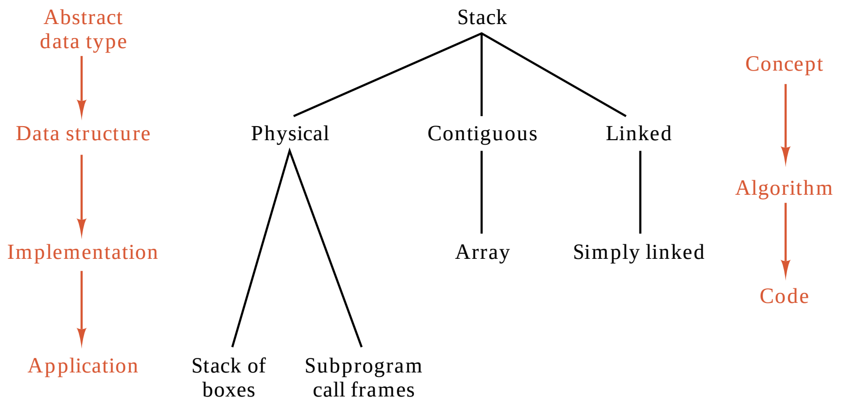
Answer See the solution to Project P6 and P10.

4.6 ABSTRACT DATA TYPES AND THEIR IMPLEMENTATIONS

Exercises 4.6

E1. *Draw a diagram similar to that of Figure 4.16 showing levels of refinement for a stack.*

Answer



E2. *Give a formal definition of the term* ***deque****, using the definitions given for stack and queue as models. Recall that entries may be added to or deleted from either end of a deque, but nowhere except at its ends.*

Answer

Definition A ***Deque*** of elements of type T is a finite sequence of elements of T together with the operations

1. *Construct* the deque, leaving it empty.
2. Determine whether the deque is *empty* or not.
3. *Append* a new entry onto the rear of the deque, provided the deque is not full.
4. *Append* a new entry onto the front of the deque, provided the deque is not full.
5. *Retrieve* the front entry in the deque, provided the deque is not empty.
6. *Retrieve* the rear entry in the deque, provided the deque is not empty.
7. *Serve* (and remove) the entry from the front of the deque, provided the deque is not empty.
8. *Serve* (and remove) the entry from the rear of the deque, provided the deque is not empty.

REVIEW QUESTIONS

1. *Give two reasons why dynamic memory allocation is valuable.*

The use of dynamic memory prevents unnecessary overflow problems caused by running out of space in arrays. Moreover, by using dynamic memory we avoid the unused memory that would result from overly large array declaration.

2. *What is garbage?*

A dynamic object that is not referenced by any pointer is garbage.

3. *Why should uninitialized pointers be set to* NULL*?*

The common programming error of dereferencing a pointer that does not store a usable machine address is much easier to trace, and much less harmful if the uninitialized pointer has been set to NULL.

4. *What is an alias and why is it dangerous?*

An alias is a second name for an object. Aliases are dangerous because they provide a means to change or destroy an object without any obvious use of its primary name.

5. *Why is it important to return an* Error_code *from the* push *method of a linked* Stack*?*

To maintain the general integrity of the Stack ADT.

6. *Why should we always add a destructor to a linked data structure?*

Otherwise garbage will be generated every time an object of the data structure goes out of scope.

7. *How is a copy constructor used and why should a copy constructor be included in a linked data structure?*

A copy constructor of a class is applied whenever an object of the class is initialized from another object of the class (for example, when passing class objects as value parameters). If a class has no explicitly defined copy constructor, a standard copy constructor is generated by the C++ compiler. If this default copy constructor is applied to a linked data structure, the copying will have reference semantics. In order to make sure that linked structures are copied with value semantics, a copy constructor is necessary.

8. *Why should a linked data structure be implemented with an overloaded assignment operator?*

If a class has no explicitly defined overloaded assignment operator, a standard assignment operator is generated by the C++ compiler. If this default assignment operator is applied to a linked data structure, the assignment will have reference semantics. In order to make sure that linked structures are assigned with value semantics, an explicit overloaded assignment operator is necessary.

9. *Discuss some problems that occur in group programming projects that do not occur in individual programming projects. What advantages does a group project have over individual projects?*

All planning for the overall project must be done with the group and any changes to be made must be cleared with the cooperation of the entire group, which may not be easy. Also, chances are, all group members will not complete their work at the same time, so there will be delays in some programmers' work that depends on the outcome of another programmer's work. All parts of the program must be made to integrate once they are completed, which brings into play possible variations in programming style and methods among the group members that may present problems. One of the obvious advantages to working as a group, however, is the division of the workload on major projects into smaller, more manageable problems. The use of data abstraction and top-down design can prevent some of the possible problems encountered in group programming projects.

10. *In an abstract data type, how much is specified about implementation?*

An abstract data type specifies nothing concerning the actual implementation of the structure.

11. *Name (in order from abstract to concrete) four levels of refinement of data specification.*

The four levels are :

1. The *abstract* level.

2. The *data structure* level.

3. The *implementation* level.

4. The *application* level.

Recursion 5

5.1 INTRODUCTION TO RECURSION

Exercises 5.1

E1. *Consider the function $f(n)$ defined as follows, where n is a nonnegative integer:*

$$f(n) = \begin{cases} 0 & \text{if } n = 0; \\ f(\tfrac{1}{2}n) & \text{if } n \text{ is even, } n > 0; \\ 1 + f(n-1) & \text{if } n \text{ is odd, } n > 0. \end{cases}$$

Calculate the value of $f(n)$ for the following values of n.

(a) $n = 1$. **(c)** $n = 3$. **(e)** $n = 100$.

Answer 1 2 3

(b) $n = 2$. **(d)** $n = 99$. **(f)** $n = 128$.

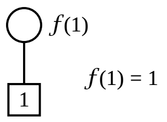
Answer 1 4 1

E2. *Consider the function $f(n)$ defined as follows, where n is a nonnegative integer:*

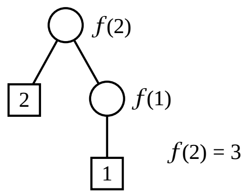
$$f(n) = \begin{cases} n & \text{if } n \le 1; \\ n + f(\tfrac{1}{2}n) & \text{if } n \text{ is even, } n > 1; \\ f(\tfrac{1}{2}(n+1)) + f(\tfrac{1}{2}(n-1)) & \text{if } n \text{ is odd, } n > 1. \end{cases}$$

For each of the following values of n, draw the recursion tree and calculate the value of $f(n)$.

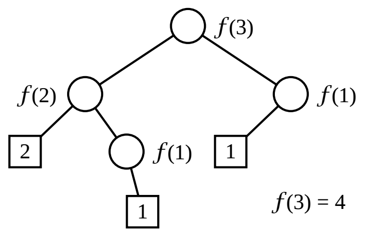
(a) $n = 1$. *Answer* 1



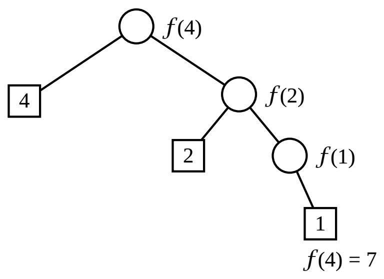
(b) $n = 2$. *Answer* 3



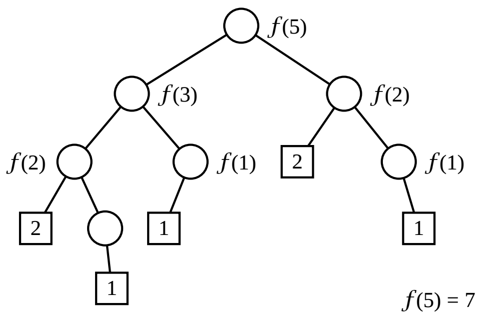
(c) $n = 3$. *Answer* 4



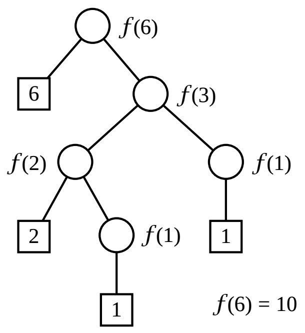
(d) $n = 4$. *Answer* 7



(e) $n = 5$. *Answer* 7



(f) $n = 6$. *Answer* 10



Programming Projects 5.1

P1. *Compare the running times for the recursive factorial function written in this section with a nonrecursive function obtained by initializing a local variable to 1 and using a loop to calculate the product $n! = 1 \times 2 \times \cdots \times n$. To obtain meaningful comparisons of the CPU time required, you will probably need to write a loop in your driver program that will repeat the same calculation of a factorial several hundred times. Integer overflow will occur if you attempt to calculate the factorial of a large number. To prevent this from happening, you may declare n and the function value to have type* **double** *instead of* **int**.

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

double factorial_recursive(int n) {
   if (n <= 0) return 1.0;
   return ((double) n) * factorial_recursive(n - 1);
}

double factorial_iterative(int n) {
   double answer = 1.0;
   for (int i = 1; i <= n; i++)
      answer *= ((double) i);
   return answer;
}

int main()
{
   cout << "Timing program for the iterative and recursive factorial "
        << "functions. \nThe value of n! is calculated repeatedly, "
        << "and timing data is printed." << endl << endl;
   int n, trials;
   cout << "Enter a value of n and a number of trials: " << flush;
   cin  >> n >> trials;
   Timer clock;
   for (int i = 0; i < trials; i++) factorial_recursive(n);
   cout << "Recursive timing data: " << clock.elapsed_time()
        << " seconds" << endl;
   clock.reset();
   for (int j = 0; j < trials; j++) factorial_iterative(n);
   cout << "Iterative timing data: " << clock.elapsed_time()
        << " seconds" << endl;
}
```

P2. *Confirm that the running time for the program* hanoi *increases approximately like a constant multiple of 2^n, where n is the number of disks moved. To do this, make* disks *a variable, comment out the line that writes a message to the user, and run the program for several successive values of* disks*, such as 10, 11, . . . , 15. How does the CPU time change from one value of* disks *to the next?*

Answer The CPU time will approximately double each time disks is increased by 1. A driver for timing the towers of hanoi program is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
#include "hanoi.cpp"
```

```
int main()
{
   int disks;
   cout << "Timing program for towers of hanoi." << endl;
   while (true) {
     cout << " Do you want to continue? ";
     if (!user_says_yes()) break;
     cout << "How many disks? " << flush;
     cin >> disks;
     Timer clock;
     move(disks, 0, 1, 2);
     cout << "Hanoi timing data: " << clock.elapsed_time()
          << " seconds for " << disks << " disks." << endl;
   }
}
```

A modified (non-printing) function for the Towers of Hanoi is:

```
#include   <iostream.h>

const int disks = 6;  // Make this constant much smaller to run program.
void move(int count, int start, int finish, int temp);

/*
Pre:  None.
Post: The simulation of the Towers of Hanoi has terminated.
*/

main()
{
   move(disks, 1, 3, 2);
}

void move(int count, int start, int finish, int temp)
{
   if (count > 0) {
      move(count - 1, start, temp, finish);
      cout << "Move disk " << count << " from " << start
           << " to " << finish << "." << endl;
      move(count - 1, temp, finish, start);
   }
}
```

5.2 PRINCIPLES OF RECURSION

Exercises 5.2

E1. *In the recursive calculation of F_n, determine exactly how many times each smaller Fibonacci number will be calculated. From this, determine the order-of-magnitude time and space requirements of the recursive function. [You may find out either by setting up and solving a recurrence relation (top-down approach), or by finding the answer in simple cases and proving it more generally by mathematical induction (bottom-up approach).]*

Answer We shall use $C(n,k)$ to denote the number of times F_k is calculated on the way to calculating F_n. The following relation then holds:

$$\begin{aligned} C(n,k) &= C(n-1,k)+C(n-2,k) \\ C(k,k) &= 0 \\ C(k+1,k) &= 1. \end{aligned}$$

Therefore, $C(n,k) = F_{n-k}$. From this the time required can be calculated as follows:

$$\begin{aligned}\text{time} &= \sum_{k=0}^{n-1} C(n,k) = \sum_{k=0}^{n-1} F_{n-k} = \sum_{k=1}^{n} F_k \\ &= \frac{1}{\sqrt{5}} \sum_{k=1}^{n} \phi^k \approx \frac{1}{\sqrt{5}} \frac{\phi^{n-1}}{\phi - 1}.\end{aligned}$$

Therefore, the time required is proportional to ϕ^{n-1}, where $\phi = \frac{1}{2}(1+\sqrt{5}) \approx 1.618$. The space required is proportional to the height of the recursion tree for F_n and is therefore proportional to n.

E2. *The **greatest common divisor** (gcd) of two positive integers is the largest integer that divides both of them. Thus, for example, the gcd of 8 and 12 is 4, the gcd of 9 and 18 is 9, and the gcd of 16 and 25 is 1.*

(a) *Write a nonrecursive function* **int** gcd(**int** x, **int** y), *where* x *and* y *are required to be positive integers, that searches through the positive integers until it finds the largest integer dividing both* x *and* y.

Answer
```
int gcd(int x, int y)
/* Pre:   Integers x and y are both positive.
   Post:  The greatest commond divisor of x and y is returned. */
{
   int gcd = 1;
   int test = 2;
   while (test < x && test < y) {
     if ((x % test == 0) && (y % test == 0))
        gcd = test;
     test ++;
   }
   return gcd;
}
```

(b) *Write a recursive function* **int** gcd(**int** x, **int** y) *that implements **Euclid's algorithm**: If* y = 0, *then the gcd of* x *and* y *is* x; *otherwise the gcd of* x *and* y *is the same as the gcd of* y *and* x % y.

Answer
```
int gcd(int x, int y)
/* Pre:   Integers x and y are both at least 0, and they are not both equal to 0.
   Post:  The greatest commond divisor of x and y is returned. */
{
   if (y == 0) return x;
   return gcd(y, x % y);
}
```

(c) *Rewrite the function of part (b) into iterative form.*

Answer
```
int gcd(int x, int y)
/* Pre:   Integers x and y are both at least 0, and they are not both equal to 0.
   Post:  The greatest commond divisor of x and y is returned. */
{
   int temp;
   while (y > 0) {
      temp = x;
      x = y;
      y = temp % y;
   }
   return x;
}
```

(d) *Discuss the advantages and disadvantages of each of these methods.*

Answer The function in (a) is an immediate consequence of the definition and so may be conceptually easier than Euclid's algorithm. It will, however, usually be much slower than Euclid's algorithm.

E3. *The binomial coefficients may be defined by the following recurrence relation, which is the idea of* ***Pascal's triangle****. The top of Pascal's triangle is shown in Figure 5.11.*

134

$$C(n,0)=1 \quad \textit{and} \quad C(n,n)=1 \qquad \textit{for } n \ge 0.$$
$$C(n,k)=C(n-1,k)+C(n-1,k-1) \qquad \textit{for } n > k > 0.$$

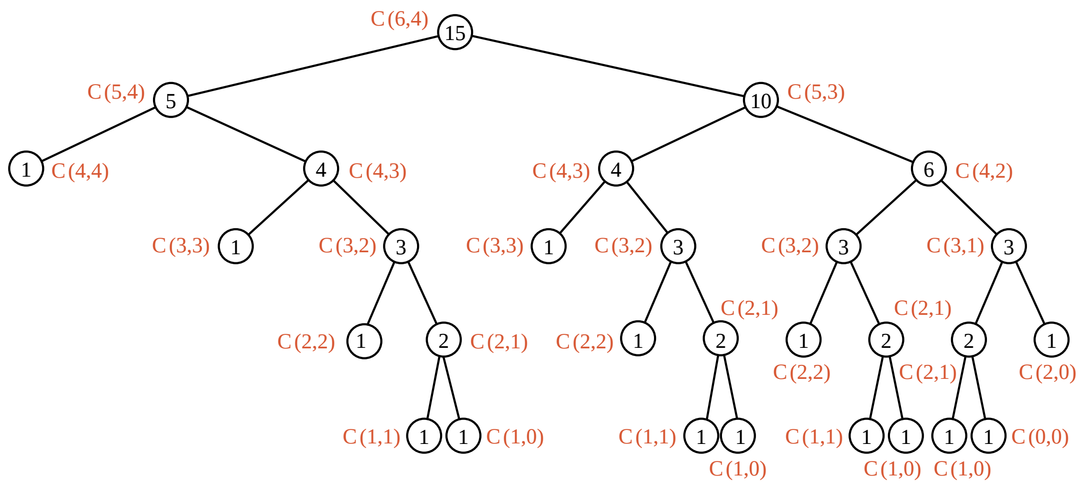
(a) *Write a recursive function to generate $C(n,k)$ by the foregoing formula.*

Answer

```
int binomial(int n, int k)
/* Pre:  Integers n and k are both at least 0, and k is at most n.
   Post: The binomial coefficient C(n, k) is returned. */
{
   if (k == 0 || k == n) return 1;
   return binomial(n − 1, k) + binomial(n − 1, k − 1);
}
```

(b) *Draw the recursion tree for calculating $C(6,4)$.*

Answer



(c) *Use a square array with n indicating the row and k the column, and write a nonrecursive program to generate Pascal's triangle in the lower left half of the array, that is, in the entries for which $k \le n$.*

Answer

```
const int maxindex = 12;                  //   bound on n
int binomial(int n, int k)
/* Pre:  Integers n and k are both at least 0, and k is at most n.
   Post: The binomial coefficient C(n, k) is returned. */
{
   int array[maxindex + 1][maxindex + 1];
   for (int i = 0; i <= n; i++) {
      array[i][0] = array[i][i] = 1;
      for (int j = 1; j < i && j <= k; j++)
         array[i][j] = array[i − 1][j] + array[i − 1][j − 1];
   }
   return array[n][k];
}
```

(d) *Write a nonrecursive program that uses neither an array nor a stack to calculate $C(n,k)$ for arbitrary $n \geq k \geq 0$.*

Answer

```
int binomial(int n, int k)
/* Pre:  Integers n and k are both at least 0, and k is at most n.
   Post: The binomial coefficient C(n, k) is returned. */
{
   int numerator = 1, denominator = 1;
   for (int i = 0;  i < k;  ) {
      numerator *= n − i;
      denominator *= ++i;
   }
   return numerator/denominator;
}
```

(e) *Determine the approximate space and time requirements for each of the algorithms devised in parts (a), (c), and (d).*

Answer To calculate $C(n,k)$ the recursive function in part (a) requires space proportional to n and time proportional to $C(n,k) = n!/(k!(n-k)!)$; the function in part (c) generating the array requires constant time for each entry, hence time proportional to n^2 for the entire array, and space proportional to n^2; and the nonrecursive function (d) requires time proportional to k and constant space.

E4. ***Ackermann's function***, *defined as follows, is a standard device to determine how well recursion is implemented on a computer.*

$$\begin{array}{ll}
A(0,n) = n+1 & \text{for } n \geq 0. \\
A(m,0) = A(m-1,1) & \text{for } m > 0. \\
A(m,n) = A(m-1, A(m,n-1)) & \text{for } m > 0 \text{ and } n > 0.
\end{array}$$

(a) *Write a recursive function to calculate Ackermann's function.*

Answer

```
int ackermann(int m, int n)
/* Pre:  Integers n and k are both at least 0.
   Post: The Ackermann function A(m, n) is returned. */
{
   if (m ==  0) return n + 1;
   if (n ==  0) return ackermann(m − 1, 1);
   return ackermann(m − 1, ackermann(m, n − 1));
}
```

(b) *Calculate the following values. If it is impossible to obtain any of these values, explain why.*

$$\begin{array}{lllll}
A(0,0) & A(0,9) & A(1,8) & A(2,2) & A(2,0) \\
A(2,3) & A(3,2) & A(4,2) & A(4,3) & A(4,0)
\end{array}$$

Answer

$$\begin{array}{llll}
A(0,0) = 1 & A(0,9) = 10 & A(1,8) = 10 & A(2,2) = 7 \\
A(2,0) = 3 & A(2,3) = 9 & A(3,2) = 29 & A(4,0) = 13
\end{array}$$

$A(4,2)$ and $A(4,3)$ are too large to calculate.

(c) *Write a nonrecursive function to calculate Ackermann's function.*

Answer We must use a stack to keep track of the parameters for each recursive call. Since the value of the function can appear as a parameter in a recursive call, we can simplify the algorithm by, at each pass, first popping the parameters from the stack and then pushing the function value or new pair of parameters onto the stack. [This idea and the resulting algorithm were contributed by K. W. Smillie.]

TOC
Index
Help

```
typedef int Stack_entry;
#include "stack.h"
#include "stack.c"
int ackermann(int m, int n)
/* Pre:  Integers n and k are both at least 0.
   Post: The Ackermann function A(m, n) is returned. */
{
   Stack s;
   s.push(m);
   s.push(n);
   while (1) {
      s.top(n); s.pop();
      if (s.top(m) == underflow) return n; s.pop();
      if (m == 0) s.push(n + 1);
      else if (n == 0) {
         s.push(m - 1);
         s.push(1);
      }
      else {
         s.push(m - 1);
         s.push(m);
         s.push(n - 1);
      }
   }
}
```

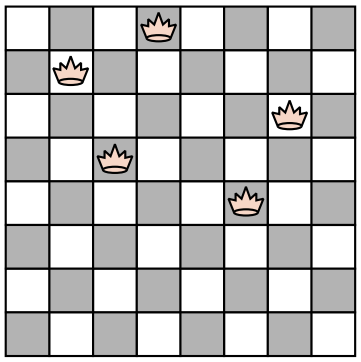
5.3 BACKTRACKING: POSTPONING THE WORK

Exercises 5.3

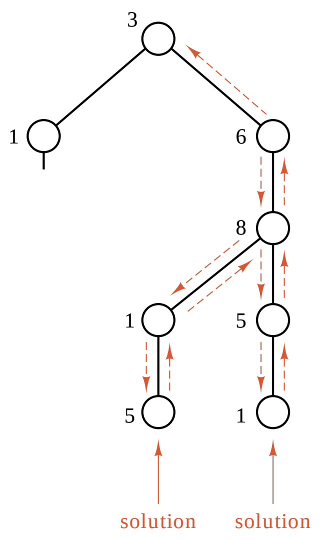
E1. *What is the maximum depth of recursion in the function* solve_from*?*

Answer The maximum depth of recursion occurs when the solution for the eight-queens problem has been found. Hence the maximum depth of recursion is eight.

E2. *Starting with the following partial configuration of five queens on the board, construct the recursion tree of all situations that the function* solve_from *will consider in trying to add the remaining three queens. Stop drawing the tree at the point where the function will backtrack and remove one of the original five queens.*

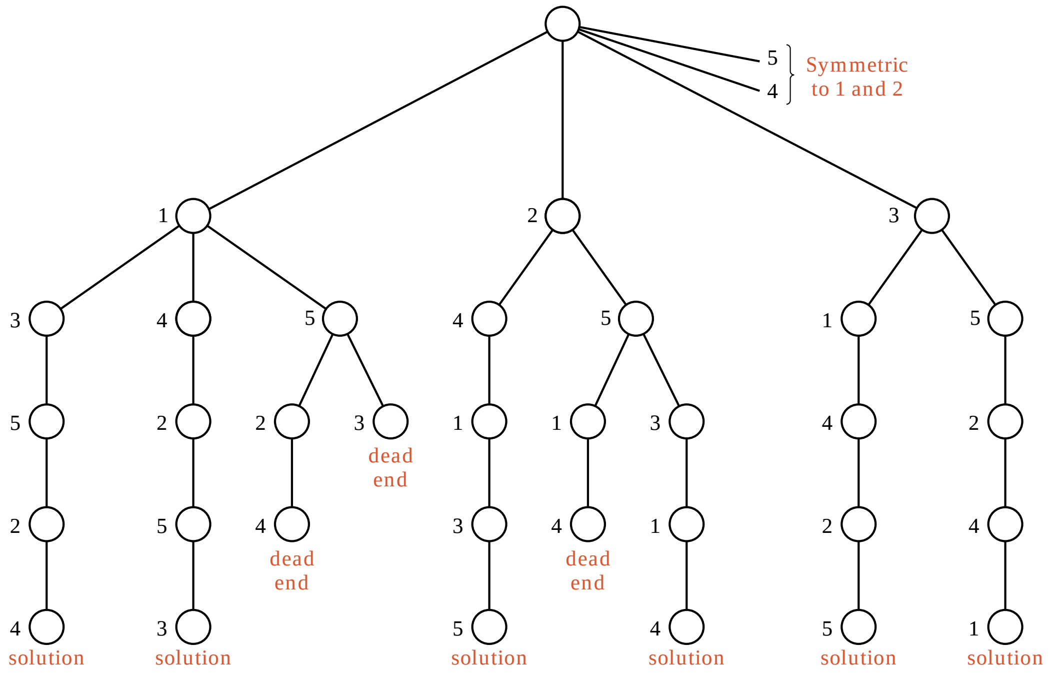


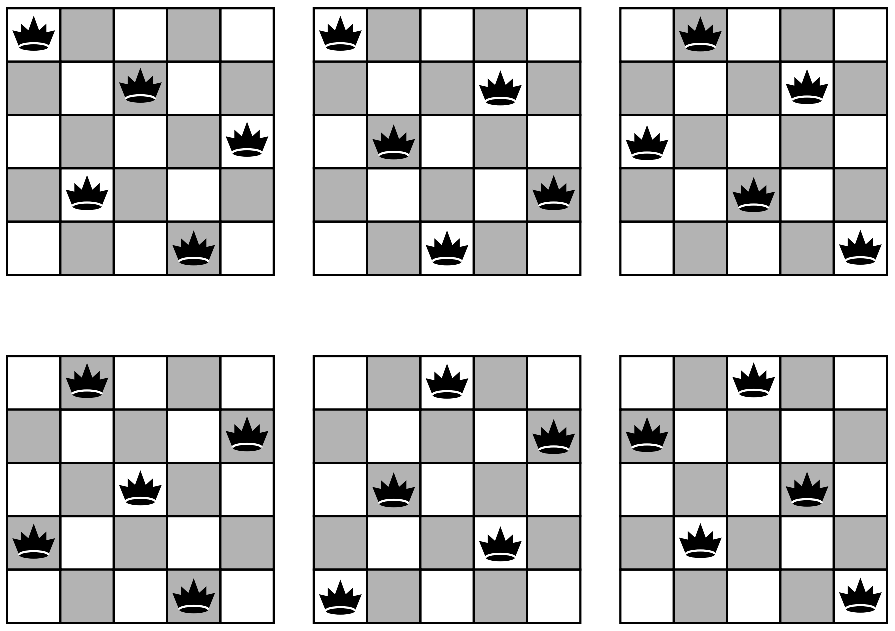
Answer



E3. *By performing backtracking by hand, find all solutions to the problem of placing five queens on a* 5×5 *board. You may use the left-right symmetry of the first row by considering only the possibilities when the queen in row 1 is in one of columns 1, 2, or 3.*

Answer There are six such solutions:





Programming Projects 5.3

P1. *Run the eight-queens program on your computer:*

(a) *Write the missing* Queens *methods.*

(b) *Find out exactly how many board positions are investigated by including a counter that is incremented every time function* solve_from *is started. [Note that a method that placed all eight queens before it started checking for guarded squares would be equivalent to eight calls to* solve_from*.]*

(c) *Run the program for the number of queens ranging from 4 to 15. Try to find a mathematical function that approximates the number of positions investigated as a function of the number of queens.*

Answer The complete program appears in the software. Here is the method print.

```
void Queens :: print( ) const
{
   int i, j;
   for (i = 0; i < board_size; i++)
      cout << "--";
   cout << "--\n";
   for (i = 0; i < board_size; i++) {
      cout << queen_in_row[i];
      for (j = 0; j < board_size; j++)
         if (j == queen_in_row[i]) cout << " Q";
         else cout << " .";
      cout << endl;
   }
}
```

The function solve_from is called 1965 times (equivalent to only about 246 complete trials of eight queens). The Queens class is implemented in the directory .. / QUEENS. Both versions from the text are implemented there, the files corresponding to the first (simpler) implementation are prefixed with the letter s. The files in .. / QUEENS provide a solution to Project P1.

Here is the first (simpler) implementation from the text. Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "squeens.h"
#include "squeens.cpp"

main()
{
   int board_size;
   print_information();
   cout << "What is the size of the board? " << flush;
   cin >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << "The number must be between 0 and " << max_board << endl;
   else {
      Queens configuration(board_size);
         //   Initialize an empty configuration.
      solve_from(configuration);
         //  Find all solutions extending configuration.
   }
}
```

Header file:

```
const int max_board = 30;
```

```
class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int  board_size; // dimension of board = maximum number of queens
private:
   int  count;      // current number of queens = first unoccupied row
   bool queen_square[max_board][max_board];
};
```

Class declaration for simple implementation:

```
Queens::Queens(int size)
/*
Post: The Queens object is set up as an empty
      configuration on a chessboard with size squares in
      each row and column.
*/

{
   board_size = size;
   count = 0;
   for (int row = 0; row < board_size; row++)
      for (int col = 0; col < board_size; col++)
         queen_square[row][col] = false;
}

void Queens::print() const
{
   int row, col;
   for (col = 0; col < board_size; col++)
      cout << "--";
   cout << "--\n";
   for (row = 0; row < board_size; row++) {
      for (col = 0; col < board_size; col++)
         if (queen_square[row][col])
            cout << " Q";
         else
            cout << " .";
      cout << endl;
   }
}

bool Queens::unguarded(int col) const
/*
Post: Returns true or false according as the square in the first
      unoccupied row (row count) and column col is not guarded
      by any queen.
*/

{
   int i;
   bool ok = true; // turns false if we find a queen in column or diagonal
   for (i = 0; ok && i < count; i++)
      ok = !queen_square[i][col];             // Check upper part of column
   for (i = 1; ok && count - i >= 0 && col - i >= 0; i++)
      ok = !queen_square[count - i][col - i]; // Check upper-left diagonal
   for (i = 1; ok && count - i >= 0 && col + i < board_size; i++)
      ok = !queen_square[count - i][col + i]; // Check upper-right diagonal

   return ok;
}

void Queens::insert(int col)
/*
Pre:  The square in the first unoccupied row (row count) and column col
      is not guarded by any queen.
Post: A queen has been inserted into the square at row count and column
      col; count has been incremented by 1.
*/

{
   queen_square[count++][col] = true;
}

void Queens::remove(int col)
/*
Pre:   There is a queen in the square in row count - 1 and column col.
Post: The above queen has been removed; count has been decremented by 1.
*/

{
   queen_square[--count][col] = false;
}

void print_information()
{
   cout << "This program determines all the ways to place n queens\n"
        << "on an n by n chessboard, where n <= " << max_board << endl;
}

bool Queens::is_solved() const
/*
Post: Returns true if the number of queens
      already placed equals board_size, otherwise return false.
*/

{
   if (count == board_size) return true;
   else return false;
}

void solve_from(Queens &configuration)
{
   if (configuration.is_solved()) configuration.print();
   else
      for (int col = 0; col < configuration.board_size; col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);
            solve_from(configuration); // Recursively continue.
            configuration.remove(col);
         }
}
```

Here is the second (more sophisticated) implementation from the text. Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "queens.h"
#include "queens.cpp"

int main()
/*
Pre:  The user enters a valid board size.
Post: All solutions to the n-queens puzzle for the selected
      board size are printed.
Uses: The class Queens and the recursive function solve_from.
*/
{
   int board_size;
   print_information();
   cout << "What is the size of the board? " << flush;
   cin  >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << "The number must be between 0 and " << max_board << endl;
   else {
      Queens configuration(board_size); // Initialize empty configuration.
      solve_from(configuration);
                                  // Find all solutions extending configuration.
   }
}
```

Header file:

```
const int max_board = 30;

class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int board_size;
private:
   int  count;
   bool col_free[max_board];
   bool upward_free[2 * max_board - 1];
   bool downward_free[2 * max_board - 1];
   int  queen_in_row[max_board];  // column number of queen in each row
};
```

Class declaration for second implementation:

```
Queens::Queens(int size)
/*
Post: The Queens object is set up as an empty
      configuration on a chessboard with size squares in
      each row and column.
*/
```

```
{
   board_size = size;
   count = 0;
   for (int i = 0; i < board_size; i++) col_free[i] = true;
   for (int j = 0; j < (2 * board_size - 1); j++) upward_free[j] = true;
   for (int k = 0; k < (2 * board_size - 1); k++) downward_free[k] = true;
}

void Queens::print() const
{
   int i, j;
   for (i = 0; i < board_size; i++)
      cout << "--";
   cout << "--\n";
   for (i = 0; i < board_size; i++) {
      cout << queen_in_row[i];
      for (j = 0; j < board_size; j++)
         if (j == queen_in_row[i]) cout << " Q";
         else cout << " .";
      cout << endl;
   }
}

bool Queens::unguarded(int col) const
/*
Post: Returns true or false according as the square in the first
      unoccupied row (row count) and column col is not guarded by any queen.
*/

{
   return  col_free[col]
           && upward_free[count + col]
           && downward_free[count - col + board_size - 1];
}

void Queens::insert(int col)
/*
Pre:  The square in the first unoccupied row (row count) and column col
      is not guarded by any queen.
Post: A queen has been inserted into the square at row count and column
      col; count has been incremented by 1.
*/

{
   queen_in_row[count] = col;
   col_free[col] = false;
   upward_free[count + col] = false;
   downward_free[count - col + board_size - 1] = false;
   count++;
}

void Queens::remove(int col)
/*
Pre:  There is a queen in the square in row count - 1 and column col.
Post: The above queen has been removed; count has been decremented by 1.
*/
```

```
{
   count--;
   col_free[col] = true;
   upward_free[count + col] = true;
   downward_free[count - col + board_size - 1] = true;
}

void print_information()
{
   cout << "This program determines all the ways to place n queens\n"
        << "on an n by n chessboard, where n <= " << max_board << endl;
}

bool Queens::is_solved() const
/*
Post: Returns true if the number of queens
      already placed equals board_size, otherwise return false.
*/

{
   if (count == board_size) return true;
   else return false;
}

void solve_from(Queens &configuration)
/*
Pre:  The Queens configuration represents a partially completed
      arrangement of non-attack-ing queens on a chessboard.
Post: All n-queens solutions that extend the given configuration
      are printed.
      The configuration is restored to its initial state.
Uses: The class Queens and the function solve_from, recursively.
*/

{
   if (configuration.is_solved()) configuration.print();
   else
      for (int col = 0; col < configuration.board_size; col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);
            solve_from(configuration);  // Recursively continue.
            configuration.remove(col);
         }
}
```

P2. *A* ***superqueen*** *can make not only all of a queen's moves, but it can also make a knight's move. (See Project P4.) Modify Project P1 so it uses superqueens instead of ordinary queens.*

Answer The following solution is based on the first (simple) n-queens solution in the text. Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "superq.h"
#include "superq.cpp"
```

```
main()
{
   int board_size;
   print_information();
   cout << "Placing Superqueens." << endl << endl;
   cout << "What is the size of the board? " << flush;
   cin >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << "The number must be between 0 and " << max_board << endl;
   else {
      Queens configuration(board_size);
                                  // Initialize an empty configuration.
      solve_from(configuration);
                         // Find all solutions extending configuration.
   }
}
```

Header file:

```
const int max_board = 30;

class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int  board_size; // dimension of board = maximum number of queens
private:
   int  count;      // current number of queens = first unoccupied row
   bool queen_square[max_board][max_board];
};
```

Code file:

```
Queens::Queens(int size)
/*
Post: The Queens object is set up as an empty
      configuration on a chessboard with size squares in
      each row and column.
*/

{
   board_size = size;
   count = 0;
   for (int row = 0; row < board_size; row++)
      for (int col = 0; col < board_size; col++)
         queen_square[row][col] = false;
}

void Queens::print() const
```

TOC
Index
Help

```
{
   int row, col;
   for (col = 0; col < board_size; col++)
      cout << "--";
   cout << "--\n";
   for (row = 0; row < board_size; row++) {
      for (col = 0; col < board_size; col++)
         if (queen_square[row][col])
            cout << " Q";
         else
            cout << " .";
      cout << endl;
   }
}

bool Queens::unguarded(int col) const
/*
Post: Returns true or false according as the square in the first
      unoccupied row (row count) and column col is not guarded
      by any queen.
*/

{
   int i;
   bool ok = true; // turns false if we find a queen in column or diagonal

   for (i = 0; ok && i < count; i++)
      ok = !queen_square[i][col];              // Check upper part of column
   for (i = 1; ok && count - i >= 0 && col - i >= 0; i++)
      ok = !queen_square[count - i][col - i]; // Check upper-left diagonal
   for (i = 1; ok && count - i >= 0 && col + i < board_size; i++)
      ok = !queen_square[count - i][col + i]; // Check upper-right diagonal
   if (ok && col >= 2 && count >= 1) ok = !queen_square[count - 1][col - 2];
   if (ok && col >= 1 && count >= 2) ok = !queen_square[count - 2][col - 1];
   if (ok && count >= 2 && col < board_size - 1)
       ok = !queen_square[count - 2][col + 1];
   if (ok && count >= 1 && col < board_size - 2)
       ok = !queen_square[count - 1][col + 2];

   return ok;
}

void Queens::insert(int col)
/*
Pre:  The square in the first unoccupied row (row count) and column col
      is not guarded by any queen.
Post: A queen has been inserted into the square at row count and column
      col; count has been incremented by 1.
*/

{
   queen_square[count++][col] = true;
}

void Queens::remove(int col)
/*
Pre:  There is a queen in the square in row count - 1 and column col.
Post: The above queen has been removed; count has been decremented by 1.
*/
```

```
{
   queen_square[--count][col] = false;
}

void print_information()
{
   cout << "This program determines all the ways to place n queens\n"
        << "on an n by n chessboard, where n <= " << max_board << endl;
}

bool Queens::is_solved() const
/*
Post: Returns true if the number of queens
      already placed equals board_size, otherwise return false.
*/

{
   if (count == board_size) return true;
   else return false;
}

void solve_from(Queens &configuration)
{
   if (configuration.is_solved()) configuration.print();
   else
      for (int col = 0; col < configuration.board_size; col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);
            solve_from(configuration); // Recursively continue.
            configuration.remove(col);
         }
}
```

P3. *Describe a rectangular maze by indicating its paths and walls within an array. Write a backtracking program to find a way through the maze.*

Answer The program directories contain a pair of data files for mazes; the file AMAZING can be solved, whereas NOMAZE cannot. Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#define MAXROW  81
#define MAXCOL  81

#include "maze.h"
#include "maze.cpp"

int main(int argc, char *argv[])
{
   char file[1000];
   cout  << "Running the maze solving program." << endl << endl;
   if (argc != 2) {
      cout << "Please specify a maze file" << flush;
      cin  >> file;
   }
   else strcpy(file, argv[1]);
   Maze m;
   m.read(file);
   m.print();
   if (m.solve(1,1))
      cout << endl << "--------- solution------------";
   else cout << endl << " **** Impossible  ****** ";
   cout << endl << endl;
   m.print();
}
```

Header file:

```
const int max_row = 81, max_col = 81;
class Maze {
 public:
   void read(char *maze_file);
   bool solve(int x, int y);
   void print();
 private:
   char maze[max_row][max_col];
   int rowsize, colsize;
};
```

Class declaration for `Maze` class:

```
void Maze::read(char *maze_file)
/*
Post: read the data defining the maze
*/

{
   int i, j;

   ifstream inmaze(maze_file);
   if (inmaze == 0) {
      cerr << "Couldn’t open maze file " << maze_file << endl;
      return;
   }
   inmaze >> rowsize >> colsize;
   while (inmaze.get() != ’\n’);     // skip newline
   for (i = 0; i <= colsize; i++) {  // Hedge
      maze[0][i] = ’H’;
      maze[rowsize+1][i] = ’H’;
   }
   for (i = 0; i <= rowsize; i++) {
      maze[i][0] = ’H’;
      maze[i][colsize+1] = ’H’;
   }

   for (i = 1; i <= rowsize; i++) {
      for (j = 1; j <= colsize; j++) {
         switch(inmaze.get()) {
         case ’*’:
         case ’ ’:
            maze[i][j] = ’*’;
         break;
```

TOC
Index
Help

```
          default:
             maze[i][j] = 'H';
          break;
          }
       }
       while (inmaze.get() != '\n');     // skip newline
    }
}
bool Maze::solve(int i, int j)
/*
Post: attempt to find a path through the maze from coordinate (i,j)
*/
{
   bool finish = false;
   maze[i][j] = '-';
   if (i == rowsize && j == colsize)
      return true;  // because we're done
   if (!finish && maze[i+1][j] == '*')
      finish = solve(i+1, j);
   if (!finish && maze[i][j+1] == '*')
      finish = solve(i, j+1);
   if (!finish && maze[i-1][j] == '*')
      finish = solve(i-1, j);
   if (!finish && maze[i][j-1] == '*')
      finish = solve(i, j-1);
   if (!finish)
      maze[i][j] = '+';
   return finish;
}
void Maze::print()
/*
Post: prints the solution
*/
{
   int i, j;
   for (i = 1; i <= rowsize; i++) {
      for (j = 1; j <= colsize; j++)
         switch (maze[i][j]) {
         case '-':
            cout << "-";
         break;
         case '+':
            cout << "+";
         break;
         case '*':
            cout << "*";
         break;
         case 'H':
            cout << "H";
         break;
         }
     cout << endl;
   }
}
```

P4. *Another chessboard puzzle (this one reputedly solved by* Gauss *at the age of four) is to find a sequence of moves by a knight that will visit every square of the board exactly once. Recall that a knight's move is to jump two positions either vertically or horizontally and one position in the perpendicular direction. Such a move can be accomplished by setting x to either 1 or 2, setting y to $3 - x$, and then changing the first coordinate by $\pm x$ and the second by $\pm y$ (provided that the resulting position is still on the board). Write a backtracking program that will input an initial position and search for a knight's tour starting at the given position and going to every square once and no square more than once. If you find that the program runs too slowly, a good method is to order the list of squares to which it can move from a given position so that it will first try to go to the squares with the least accessibility, that is, to the squares from which there are the fewest knight's moves to squares not yet visited.*

Answer The class Horse includes methods to place and remove knights from a chessboard and to search recursively for knight's tours and magic knight's tours. Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "knight.h"
#include "knight.cpp"

main()
{
   int board_size;
   print_information();
   cout << "What is the size of the board? " << flush;
   cin >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << "The number must be between 0 and " << max_board << endl;
   else {
      Horse configuration(board_size); // Initialize empty configuration.
      cout << "In which row and column (1.." << board_size << ")"
           << " do you want\nthe first knight?: " << flush;
      int row, col;
      cin >> row >> col;
      configuration.insert(row + 1, col + 1);
      solve_from(configuration);   //  Find all solutions
   }
}
```

Definition of class Horse:

```
const int max_board = 9;

class Horse {
public:
   Horse (int size);
   bool is_solved() const;
   void print() const;
   void magic_print() const;
   bool unguarded(int col) const;
   void insert(int h, int v);
   void remove();
   int  board_size; //   dimension of board
private:
   int  count;      //   current number of knights
   bool horse_square[max_board + 4][max_board + 4];
   int  hhorses[max_board * max_board];
   int  vhorses[max_board * max_board];
friend void solve_from(Horse &configuration);
friend void solve_magic(Horse &configuration);
};
```

```
int horiz[8];  // horizontal components of knight moves
int vert[8];   // vertical components of knight moves
```

Implementation of class Horse:

```
Horse::Horse(int size)
/*
Post: The Horse object is set up as an empty
      configuration on a chessboard with size squares in
      each row and column.
*/

{
   board_size = size;
   count = 0;
   int row, col;
   for (row = 0; row < board_size + 4; row++)
      for (col = 0; col < board_size + 4; col++)
         horse_square[row][col] = false;
   for (row = 0; row < 2; row++)    // Make a wide hedge
      for (col = 0; col < board_size + 3; col++)
         horse_square[col][board_size + 2 + row]
               = horse_square[col][row]
               = horse_square[board_size + 2 + row][col]
               = horse_square[row][col]
               =  true;
// finally initialize the table of legal knight moves:
   horiz[0] = -2; horiz[1] = -1; horiz[2] = 1; horiz[3] = 2;
   horiz[7] = -2; horiz[6] = -1; horiz[5] = 1; horiz[4] = 2;
    vert[0] = -1;  vert[1] = -2; vert[2] = -2; vert[3] = -1;
    vert[7] =  1;  vert[6] =  2; vert[5] =  2; vert[4] =  1;
}

void Horse::magic_print() const
{
   // begin by checking for magic, then call print.
   int x = hhorses[0], y = vhorses[0];
   int x1 = hhorses[count - 1], y1 = vhorses[count - 1];
   int d = (x - x1) * (x - x1) + (y - y1) * (y - y1);
   if (d != 5) return;
   print();
}

void Horse::print() const
{
   int print_out[max_board][max_board];
   int i, j, row, col;
   for (i = 0; i < board_size; i++)
    for (j = 0; j < board_size; j++)
      print_out[i][j] = -1;

   for (i = 0; i < count; i++)
      print_out[vhorses[i] - 2][hhorses[i] - 2] = i;
   for (row = 0; row < board_size; row++) {
      for (col = 0; col < board_size; col++) {
         if (0 <= print_out[row][col] && print_out[row][col] <=9)
            cout << " ";
         cout << print_out[row][col] << " ";
      }
      cout << endl << endl;
   }
}

void Horse::insert(int h, int v)
{
   horse_square[h][v] = true;
   hhorses[count] = h;
   vhorses[count] = v;
   count++;
}

void Horse::remove()
/*
Pre:   There is a queen in the square in row count - 1 and column col.
Post: The above queen has been removed; count has been decremented by 1.
*/

{
   count--;
   horse_square[hhorses[count]][vhorses[count]] = false;
}

void print_information()
{
   cout << "This program determines all the ways to place n knights\n"
        << "on an n by n chessboard, where n <= " << max_board << endl;
}

bool Horse::is_solved() const
/*
Post: Returns true if the number of knights
already placed equals board_size * board_size, otherwise return false.
*/

{
   return count == board_size * board_size;
}

void solve_from(Horse &configuration)
{
   if (configuration.is_solved()) configuration.print();
   else {
      int h = configuration.hhorses[configuration.count - 1],
          v = configuration.vhorses[configuration.count - 1];
      for (int move = 0; move < 8; move++)
         if (!configuration.horse_square[h + horiz[move]][v + vert[move]])
         {
            configuration.insert(h + horiz[move], v + vert[move]);
            solve_from(configuration); //   Recursively continue
            configuration.remove();
         }
   }
}
```

TOC
Index
Help

```
void solve_magic(Horse &configuration)
{
   if (configuration.is_solved()) configuration.magic_print();
   else {
      int h = configuration.hhorses[configuration.count - 1],
          v = configuration.vhorses[configuration.count - 1];
      for (int move = 0; move < 8; move++)
         if (!configuration.horse_square[h + horiz[move]][v + vert[move]])
         {
            configuration.insert(h + horiz[move], v + vert[move]);
            solve_magic(configuration); //   Recursively continue
            configuration.remove();
         }
   }
}
```

P5. *Modify the program from Project P4 so that it numbers the squares of the chessboard in the order they are visited by the knight, starting with 1 in the square where the knight starts. Modify the program so that it finds a* ***magic*** *knight's tour, that is, a tour in which the resulting numbering of the squares produces a magic square. [See Section 1.6, Project P1(a) for the definition of a magic square.]*

Answer See the solution to Project P4 for the class Horse. Driver program:

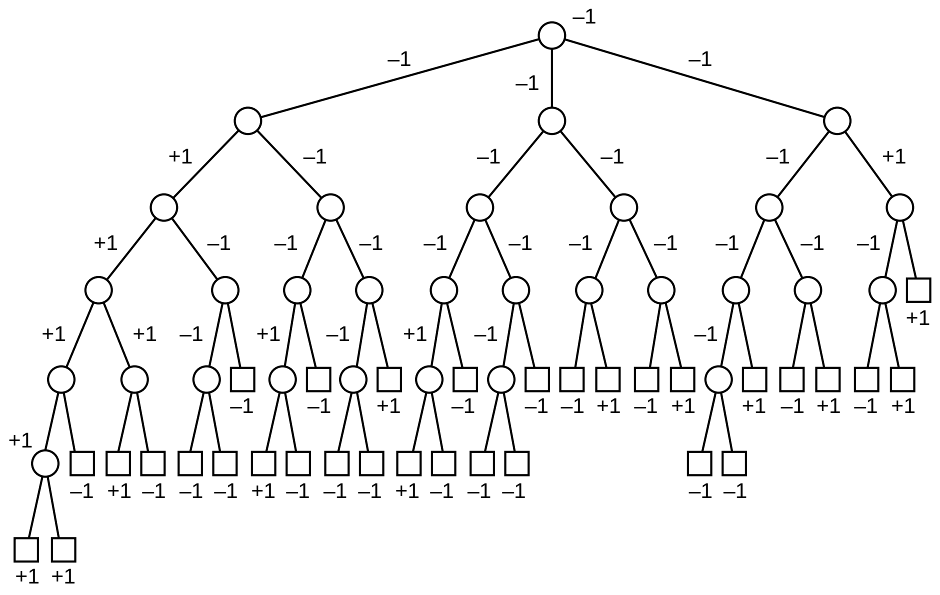
```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "knight.h"
#include "knight.cpp"

main()
{
 int board_size;
 cout << "This program determines all the MAGIC ways to place n knights\n"
      << "on an n by n chessboard, where n <= " << max_board << endl;
 cout << "What is the size of the board? " << flush;
 cin >> board_size;
 if (board_size < 0 || board_size > max_board)
    cout << "The number must be between 0 and " << max_board << endl;
 else {
    Horse configuration(board_size); // Initialize an empty configuration.
    cout << "In which row and column (1.." << board_size << ")"
         << " do you want\nthe first knight?: " << flush;
    int row, col;
    cin >> row >> col;
    configuration.insert(row + 1, col + 1);
    solve_magic(configuration);   //  Find all solutions
 }
}
```

5.4 TREE-STRUCTURED PROGRAMS: LOOK-AHEAD IN GAMES

Exercises 5.4

E1. *Assign values of +1 for a win by the first player and −1 for a win by the second player in the game of Eight, and evaluate its game tree by the minimax method, as shown in Figure 5.16.*

Answer

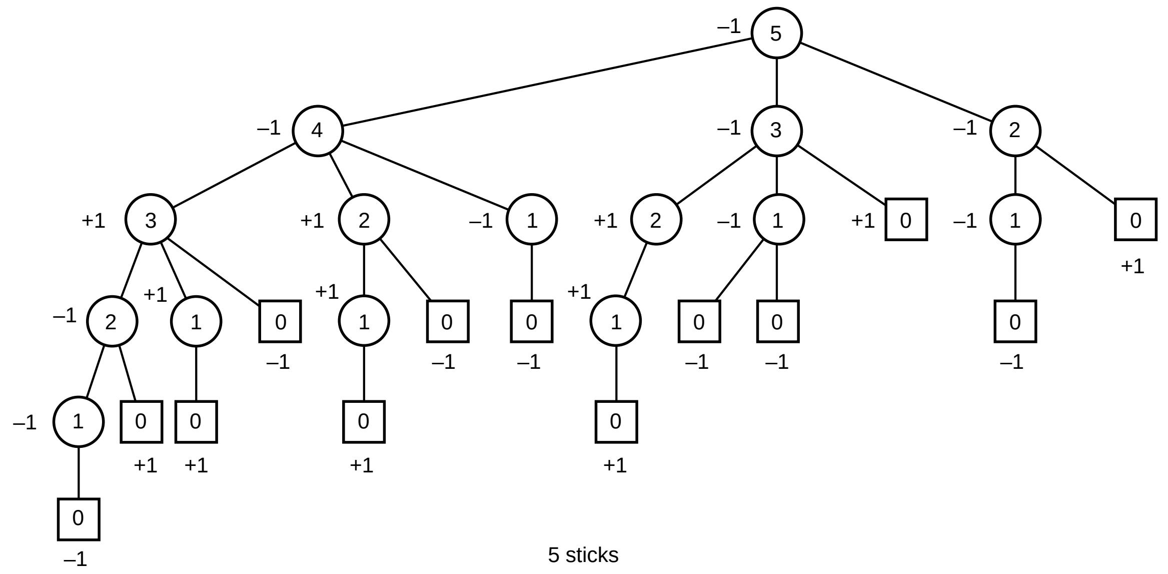


E2. *A variation of the game of Nim begins with a pile of sticks, from which a player can remove 1, 2, or 3 sticks at each turn. The player must remove at least 1 (but no more than remain on the pile). The player who takes the last stick loses. Draw the complete game tree that begins with*

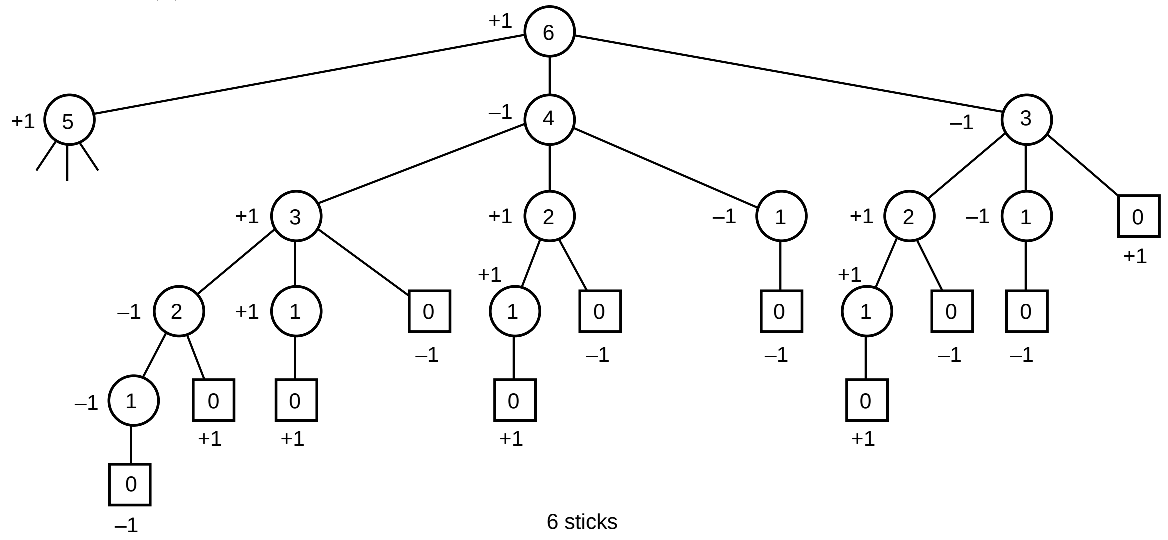
(a) *5 sticks* **(b)** *6 sticks.*

Assign appropriate values for the leaves of the tree, and evaluate the other nodes by the minimax method.

Answer **(a)** 5 sticks:

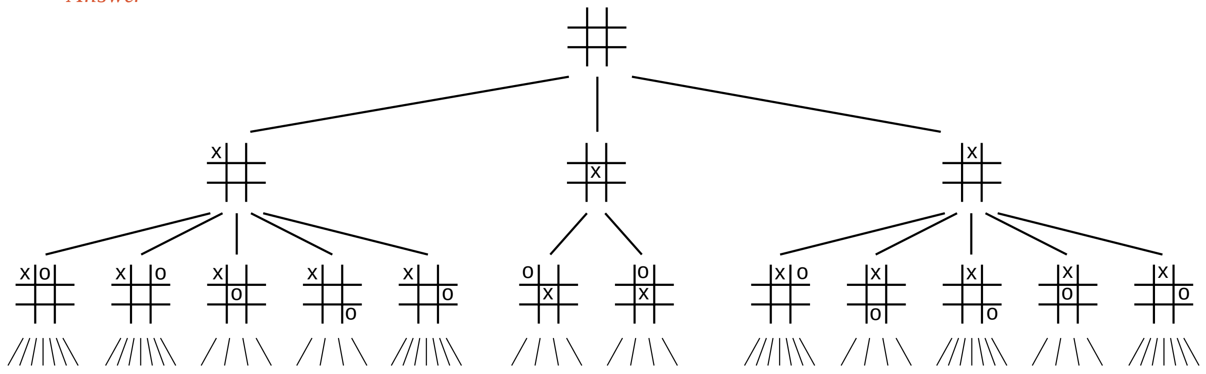

5 sticks

(b) 6 sticks:


6 sticks

E3. *Draw the top three levels (showing the first two moves) of the game tree for the game of tic-tac-toe (noughts and crosses), and calculate the number of vertices that will appear on the fourth level. You may reduce the size of the tree by taking advantage of symmetries: At the first move, for example, show only three possibilities (the center square, a corner, or a side square) rather than all nine. Further symmetries near the root will reduce the size of the game tree.*

Answer



Programming Projects 5.4

P1. *Write a main program and the* Move *and* Board *class implementations to play Eight against a human opponent.*

Answer The main program to play the game and implement look-ahead is identical to that used for Tic-Tac-Toe. Here it is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "move.h"
#include "move.cpp"
typedef Move Stack_entry;   //  Type for Stack entries.
#include "../../2/stack/stack.h"
#include "../../2/stack/stack.cpp"
#include "board.h"
#include "board.cpp"

int look_ahead(const Board &game, int depth, Move &recommended)
/*
Pre:  Board game represents a legal game position.
Post: An evaluation of the game, based on looking ahead
      depth moves, is returned.  The best move that can be found
      for the mover is recorded as Move recommended.
Uses: The classes Stack, Board, and Move, together with
      function look_ahead recursively.
*/

{
   if (game.done() || depth == 0)
      return game.evaluate();

   else {
      Stack moves;
      game.legal_moves(moves);
      int value, best_value = game.worst_case();

      while (!moves.empty()) {
         Move try_it, reply;
         moves.top(try_it);
         Board new_game = game;
         new_game.play(try_it);
         value = look_ahead(new_game, depth - 1, reply);
         if (game.better(value, best_value)) {
            //  try_it is the best move yet found
            best_value = value;
            recommended = try_it;
         }
         moves.pop();
      }
      return best_value;
   }
}
```

```
main()
{
   Board game;
   Move next_move;
   game.instructions();
   cout << " How much lookahead?";
   int looka;
   cin >> looka;
   while (!game.done()) {
      if (looka > 0)
         look_ahead(game, looka, next_move);
      else {
         Stack moves;
         game.legal_moves(moves);
         moves.top(next_move);
      }
      game.play(next_move);
      game.print();
   }
}
```

Definition of class Board:

```
class Board {
public:
   Board();
   bool done() const;
   void print() const;
   void instructions() const;
   bool better(int value, int old_value) const;
   void play(Move try_it);
   int worst_case() const;
   int evaluate() const;
   int legal_moves(Stack &moves) const;
private:
   int count, last;
   int moves_done;
   int the_winner() const;
};
```

Definition of class Move:

```
//  class for an eight move
class Move {
public:
   Move();
   Move(int x);
   int value;
};
```

Implementation of class Board:

```
Board::Board()
/*
Post: The Board is initialized as empty.
*/
{
   count = last = moves_done = 0;
}
```

```
bool Board::done() const
/*
Post: Return true if the game is over; because
      a player has already won
*/

{
   return the_winner() > 0;
}

void Board::print() const
/*
Post: The Board is printed.
*/

{
   cout << " Current sum : " << count
        << " Last move was: " << last << endl;
}

void Board::instructions() const
/*
Post: Instructions for Eight are printed.
*/

{
   cout << " This game plays Eight\n";
}

bool Board::better(int value, int old_value) const
/*
Post: Return true if the next player to move prefers a game
      situation value rather than a game situation old_value.
*/

{
   return (moves_done % 2 && (value < old_value)) ||
          (!(moves_done % 2) && value > old_value);
}

void Board::play(Move try_it)
/*
Post: The Move try_it is played on the Board.
*/

{
   count += try_it.value;
   last = try_it.value;
   moves_done++;
}

int Board::worst_case() const
/*
Post: Return the value of a worst case scenario for the mover.
*/

{
   if (moves_done % 2) return 10;
   else return -10;
}
```

```
int Board::the_winner() const
/*
Post: Return either a value of 0 for a position where neither player
      has won, a value of 1 if the first player has won,
      or a value of 2 if the second player has won.
*/

{
   if (count < 8) return 0;
   if (count > 8) {
     if (moves_done % 2!= 0) return 2;
     return 1;
   }
   if (moves_done % 2!= 0) return 1;
   return 2;
}

int Board::evaluate() const
/*
Post: Return either a value of 0 for a position where neither player
      has won, a positive value between 1 and 9 if the first player
      has won, or a negative value between -1 and -9 if the second
      player has won.
*/

{
   int winner = the_winner();
   if (winner == 1) return 10 - moves_done;
   else if (winner == 2) return moves_done - 10;
   else return 0;
}

int Board::legal_moves(Stack &moves) const
/*
Post: The parameter Stack moves is set up to contain all
      possible legal moves on the Board.
*/

{
   int number = 0;
   while (!moves.empty()) moves.pop();
   for (int i = 1; i < 4; i++) if (i != last) {
      Move can_play(i);
      moves.push(can_play);
      number++;
   }
   return number;
}
```

Implementation of class Move:

```
Move::Move()
/*
Post: The Move is initialized to an illegal, default value.
*/
{
   value = 0;
}
```

```
Move::Move(int r)
/*
Post: The Move is initialized to the given value.
*/
{
   value = r;
}
```

P2. *If you have worked your way through the tree in Figure 5.17 in enough detail, you may have noticed that it is not necessary to obtain the values for all the vertices while doing the minimax process, for there are some parts of the tree in which the best move certainly cannot appear.*

Let us suppose that we work our way through the tree starting at the lower left and filling in the value for a parent vertex as soon as we have the values for all its children. After we have done all the vertices in the two main branches on the left, we find values of 7 and 5, and therefore the maximum value will be at least 7. When we go to the next vertex on level 1 and its left child, we find that the value of this child is 3. At this stage, we are taking minima, so the value to be assigned to the parent on level 1 cannot possibly be more than 3 (it is actually 1). Since 3 is less than 7, the first player will take the leftmost branch instead, and we can exclude the other branch. The vertices that, in this way, need never be evaluated are shown within dotted lines in color in Figure 5.18.

alpha-beta pruning

The process of eliminating vertices in this way is called ***alpha-beta pruning****. The Greek letters α (alpha) and β (beta) are generally used to denote the cutoff points found.*

Modify the function look_ahead *so that it uses alpha-beta pruning to reduce the number of branches investigated. Compare the performance of the two versions in playing several games.*

Answer In our method, alpha is a game value that is so bad for the next player that any position with this value or a more extreme one can be ignored. Similarly, beta is a game value for the next but one player (i.e., the previous player) that is so bad that any game with this value or a more extreme value can be ignored.

An implementation of look-ahead with alpha-beta pruning is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../minimax/move.h"
#include "../minimax/move.cpp"
typedef Move Stack_entry;   //  Type for Stack entries.
#include "../../2/stack/stack.h"
#include "../../2/stack/stack.cpp"
#include "board.h"
#include "board.cpp"

int look_ahead(const Board &game, int depth, Move &recommended,
     int alpha, int beta)     // alpha is considered too "terrible" to
                              // analyze by the mover
                              // beta is considered too "terrible" to
                              // analyze by the non-mover
/*
Pre:  Board game represents a legal game position.
Post: An evaluation of the game, based on looking ahead
      depth moves, is returned.  The best move that can be found
      for the mover is recorded as Move recommended.

      This implementation incorporates alpha-beta pruning; the changes
      from regular lookahead are marked with alpha-beta comments.

Uses: The classes Stack, Board, and Move, together with
      function look_ahead recursively.
*/
```

```
{
   if (game.done() || depth == 0)
      return game.evaluate();

   else {
      Stack moves;
      game.legal_moves(moves);
      int value, best_value = game.worst_case();

      while (!moves.empty()) {
         Move try_it, reply;
         moves.top(try_it);
         Board new_game = game;
         new_game.play(try_it);
         value = look_ahead(new_game, depth - 1, reply, beta, alpha);
         if (game.better(value, best_value)) {
                           //  try_it is the best move yet found
            best_value = value;
            recommended = try_it;
            if (!game.better(beta, value))          // prune the tree
               while (!moves.empty()) moves.pop();
            if (game.better(value, alpha)) alpha = value;
         }
         moves.pop();
      }
      return best_value;
   }
}

main()
{
   Board game;
   Move next_move;
   game.instructions();

   cout << "Tic Tac Toe with lookahead and alpha beta pruning."
        << endl << endl;

   cout << " How much lookahead?";
   int looka;
   cin >> looka;
   while (!game.done()) {
      if (looka > 0)
         look_ahead(game, looka, next_move,
                             game.worst_case(), game.best_case());
      else {
         Stack moves;
         game.legal_moves(moves);
         moves.top(next_move);
      }
      game.play(next_move);
      game.print();
   }
}
```

A new board method, best_case(), is now required, so this is added to the Tic_Tac_Toe Board class, in the

```
class Board {
public:
   Board();
   bool done() const;
   void print() const;
   void instructions() const;
   bool better(int value, int old_value) const;
   void play(Move try_it);
   int worst_case() const;
   int best_case() const;
   int evaluate() const;
   int legal_moves(Stack &moves) const;
private:
   int squares[3][3];
   int moves_done;
   int the_winner() const;
};


Board::Board()
/*
Post: The Board is initialized as empty.
*/

{
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         squares[i][j] = 0;
   moves_done = 0;
}

bool Board::done() const
/*
Post: Return true if the game is over; either because
      a player has already won
      or because all nine squares have been filled.
*/

{
   return moves_done == 9 || the_winner() > 0;
}

void Board::print() const
/*
Post: The Board is printed.
*/

{
   int i, j;
   for (i = 0; i < 5; i++) cout << "-"; cout << "\n";
   for (i = 0; i < 3; i++) {
      for (j = 0; j < 3; j++)
         if (!squares[i][j]) cout << " ";
         else if (squares[i][j] == 1) cout << "X";
         else cout << "O";
      cout << "\n";
   }
}
```

```
void Board::instructions() const
/*
Post: Instructions for tic-tac-toe are printed.
*/
{
   cout << " This game plays Tic Tac To \n";
}

bool Board::better(int value, int old_value) const
/*
Post: Return true if the next player to move prefers a game
      situation value rather than a game situation old_value.
*/
{
   return (moves_done % 2 && (value < old_value)) ||
          (!(moves_done % 2) && value > old_value);
}

void Board::play(Move try_it)
/*
Post: The Move try_it is played on the Board.
*/
{
   squares[try_it.row][try_it.col] = moves_done % 2 + 1;
   moves_done++;
}

int Board::worst_case() const
/*
Post: Return the value of a worst case scenario for the mover.
*/
{
   if (moves_done % 2) return 10;
   else return -10;
}

int Board::best_case() const
/*
Post: Return the value of a worst case scenario for the mover.
*/
{
   if (moves_done % 2) return -10;
   else return 10;
}

int Board::the_winner() const
/*
Post: Return either a value of 0 for a position where neither
      player has won, a value of 1 if the first player has won,
      or a value of 2 if the second player has won.
*/
{
   int i;
   for (i = 0; i < 3; i++)
      if (squares[i][0] && squares[i][0] == squares[i][1]
                        && squares[i][0] == squares[i][2])
            return squares[i][0];
   for (i = 0; i < 3; i++)
      if (squares[0][i] && squares[0][i] == squares[1][i]
                        && squares[0][i] == squares[2][i])
            return squares[0][i];

   if (squares[0][0] && squares[0][0] == squares[1][1]
                     && squares[0][0] == squares[2][2])
            return squares[0][0];

   if (squares[0][2] && squares[0][2] == squares[1][1]
                     && squares[2][0] == squares[0][2])
            return squares[0][2];
   return 0;
}

int Board::evaluate() const
/*
Post: Return either a value of 0 for a position where neither player
      has won, a positive value between 1 and 9 if the first player
      has won, or a negative value between -1 and -9 if the second
      player has won,
*/

{
   int winner = the_winner();
   if (winner == 1) return 10 - moves_done;
   else if (winner == 2) return moves_done - 10;
   else return 0;
}

int Board::legal_moves(Stack &moves) const
/*
Post: The parameter Stack moves is set up to contain all
      possible legal moves on the Board.
*/

{
   int count = 0;
   while (!moves.empty()) moves.pop();
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         if (squares[i][j] == 0) {
            Move can_play(i,j);
            moves.push(can_play);
            count++;
         }
   return count;
}
```

REVIEW QUESTIONS

1. *Define the term divide and conquer.*

This is the method of solving problems by dividing the main problem into two or more subproblems that are similar in nature, but smaller in size. Solutions are then obtained to the subproblems and combined to produce the solution to the larger problem.

2. *Name two different ways to implement recursion.*

Recursion can be implemented either by using multiple processors or stacks.

3. *What is a re-entrant program?*

These are large system programs that keep their instructions in one storage area and their data and variable addresses in another. This allows a number of users on a time-sharing system to use the same program (such as the text editor) while keeping only one copy of the instructions in memory, but a separate data area for each user.

4. *How does the time requirement for a recursive function relate to its recursion tree?*

The time requirement is related to the number of times functions are done and therefore to the total number of vertices in the tree.

5. *How does the space requirement for a recursive function relate to its recursion tree?*

The space requirement is only that of the storage areas on the path from a single vertex back to the root and hence is related to the height of the tree.

6. *What is tail recursion?*

Tail recursion occurs when the last executed statement of a function is a recursive call.

7. *Describe the relationship between the shape of the recursion tree and the efficiency of the corresponding recursive algorithm.*

A well-balanced, bushy tree reflects a recursive process that can work efficiently, while a tall, thin tree signifies an inefficient process.

8. *What are the major phases of designing recursive algorithms?*

The first step is to derive a general algorithm for the solution of the problem at hand. Secondly, the stopping rule must be determined. The next step if to verify that the recursion will terminate in both general and extreme cases. Finally, devise a recursion tree for the algorithm.

9. *What is concurrency?*

Concurrency occurs when a number of processes take place simultaneously.

10. *What important kinds of information does the computer system need to keep while implementing a recursive function call?*

Firstly, the computer must keep track of the address where the call was made from. Also, it must store the addresses and values of all local variables.

11. *Is the removal of tail recursion more important for saving time or for saving space?*

There is little difference in execution time when tail recursion is removed, but there is a noticeable saving in storage space.

12. *Describe backtracking as a problem-solving method.*

In backtracking, you construct partial solutions to a problem, continuing to add pieces to the puzzle until you either solve it or come to a point where nothing further can be done. At this stage, you *backtrack* by removing the most previously added part, and trying a different approach until, finally, you come across a full solution.

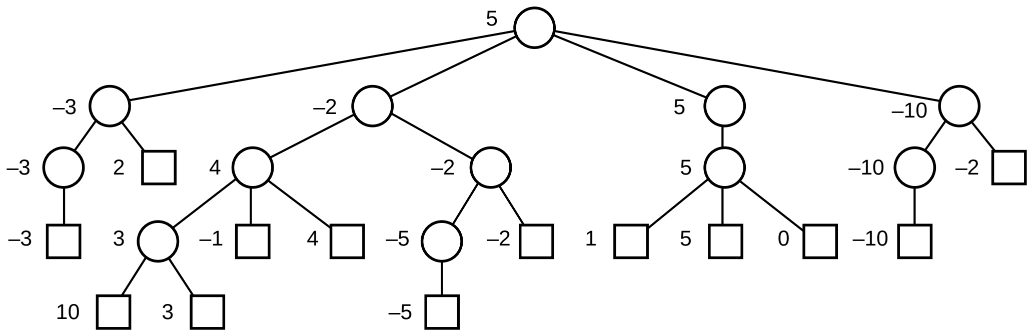
13. *State the pigeonhole principle.*

If you have n pigeons and n pigeonholes, and no more than one pigeon ever goes in the same hole, then there must be a pigeon in every hole.

14. *Explain the minimax method for finding the value of a game.*

As the levels alternate in a game tree, we take either the path with the largest or smallest value. The value at the root is either the maximum or the minimum value of all the subtrees emanating from the root that we are currently evaluating.

15. *Determine the value of the following game tree by the minimax method.*



Lists and Strings 6

6.1 LIST DEFINITION

Exercises 6.1

Given the methods for lists described in this section, write functions to do each of the following tasks. Be sure to specify the preconditions and postconditions for each function. You may use local variables of types List *and* List_entry, *but do not write any code that depends on the choice of implementation. Include code to detect and report an error if a function cannot complete normally.*

E1. Error_code insert_first(**const** List_entry &x, List &a_list) *inserts entry* x *into position 0 of the* List a_list.

Answer

```
Error_code insert_first(const List_entry &x, List &a_list)
/* Post: Entry x is inserted at position 0 of List a_list. */
{
   return a_list.insert(0, x);
}
```

E2. Error_code remove_first(List_entry &x, List &a_list) *removes the first entry of the* List a_list, *copying it to* x.

Answer

```
Error_code remove_first(List_entry &x, List &a_list)
/* Post: A code of underflow is returned if List a_list is empty. Otherwise, the first entry of List
         a_list is removed and reported as x. */
{
   return a_list.remove(0, x);
}
```

E3. Error_code insert_last(**const** List_entry &x, List &a_list) *inserts* x *as the last entry of the* List a_list.

Answer

```
Error_code insert_last(const List_entry &x, List &a_list)
/* Post: Parameter x is inserted as the last entry of the List a_list. */
{
   return a_list.insert(a_list.size( ), x);
}
```

E4. Error_code remove_last(List_entry &x, List &a_list) *removes the last entry of* a_list*, copying it to* x.

Answer
```
Error_code remove_last(List_entry &x, List &a_list)
/* Post: A code of underflow is returned if List a_list is empty. Otherwise, the last entry of List
         a_list is removed and reported as x. */
{
  return a_list.remove(a_list.size() − 1, x);
}
```

E5. Error_code median_list(List_entry &x, List &a_list) *copies the central entry of the* List a_list *to* x *if* a_list *has an odd number of entries; otherwise, it copies the left-central entry of* a_list *to* x.

Answer
```
Error_code median_list(List_entry &x, List &a_list)
/* Post: A code of underflow is returned if List a_list is empty. Otherwise, the median entry of
         List a_list is reported as x. */
{
  return a_list.retrieve((a_list.size() − 1)/2, x);
}
```

E6. Error_code interchange(**int** pos1, **int** pos2, List &a_list) *interchanges the entries at positions* pos1 *and* pos2 *of the* List a_list.

Answer
```
Error_code interchange(int pos1, int pos2, List &a_list)
/* Post: Any entries at positions pos1 and pos2 of List a_list are interchanged. If either entry is
         missing a code of range_error is returned. */
{
  List_entry entry1, entry2;
  Error_code outcome = a_list.retrieve(pos1, entry1);
  if (outcome == success)
    a_list.retrieve(pos2, entry2);
  if (outcome == success)
    a_list.replace(pos1, entry2);
  if (outcome == success)
    a_list.replace(pos2, entry1);
  return outcome;
}
```

E7. **void** reverse_traverse_list(List &a_list, **void** (*visit)(List_entry &)) *traverses the* List a_list *in reverse order (from its last entry to its first).*

Answer
```
void reverse_traverse_list(List &a_list,
                           void (*visit)(List_entry &))
/* Post: The List a_list is traversed, in reverse order, and the function *visit is applied to all
         entries. */
{
  List_entry item;
  for (int i = a_list.size() − 1; i >= 0; i−−) {
    a_list.retrieve(i, item);
    (*visit)(item);
  }
}
```

E8. Error_code copy(List &dest, List &source) *copies all entries from* source *into* dest*;* source *remains unchanged. You may assume that* dest *already exists, but any entries already in* dest *are to be discarded.*

Answer

```
Error_code copy(List &dest, List &source)
/* Post: All entries are copied from source into dest; source remains unchanged. A code
         of fail is returned if a complete copy cannot be made. */
{
   List_entry item;
   Error_code outcome = success;
   while (!dest.empty()) dest.remove(0, item);
   for (int i = 0; i < source.size(); i++) {
       if (source.retrieve(i, item) != success) outcome = fail;
       if (dest.insert(i, item) != success) outcome = fail;
   }
   return outcome;
}
```

E9. Error_code join(List &list1, List &list2) *copies all entries from* list1 *onto the end of* list2*;* list1 *remains unchanged, as do all the entries previously in* list2.

Answer

```
Error_code join(List &list1, List &list2)
/* Post: All entries from list1 are copied onto the end of list2. A code of overflow is returned if
         list2 is filled up before the copying is complete. */
{
   List_entry item;
   for (int i = 0; i < list1.size(); i++) {
       list1.retrieve(i, item);
       if (list2.insert(list2.size(), item) != success)
              return overflow;
   }
   return success;
}
```

E10. **void** reverse(List &a_list) *reverses the order of all entries in* a_list.

Answer

```
void reverse(List &a_list)
/* Post: Reverses the order of all entries in a_list. A code of fail is returned in case the reversal
         cannot be completed. */
{
   List temp;
   List_entry item;
   Error_code outcome = success;
   for (int i = 0; i < a_list.size(); i++) {
       a_list.retrieve(i, item);
       if (temp.insert(i, item) != success)
           outcome = fail;
   }
   for (int j = 0; j < a_list.size(); j++) {
       temp.retrieve(j, item);
       a_list.replace(a_list.size() − 1 − j, item);
   }
}
```

E11. Error_code split(List &source, List &oddlist, List &evenlist) *copies all entries from* source *so that those in odd-numbered positions make up* oddlist *and those in even-numbered positions make up* evenlist. *You may assume that* oddlist *and* evenlist *already exist, but any entries they may contain are to be discarded.*

Answer
```
Error_code split(List &source, List &oddlist, List &evenlist)
/* Post: Copies all entries from source so that those in odd-numbered positions make up oddlist
         and those in even-numbered positions make up evenlist. Returns an error code of
         overflow in case either output list fills before the copy is complete. */
{
   List_entry item;
   Error_code outcome = success;
   for (int i = 0; i < source.size(); i++) {
      source.retrieve(i, item);
      if (i % 2 != 0) {
         if (oddlist.insert(oddlist.size(), item) == overflow)
            outcome = overflow;
      }
      else
         if (evenlist.insert(evenlist.size(), item) == overflow)
            outcome = overflow;
   }
   return outcome;
}
```

6.2 IMPLEMENTATION OF LISTS

Exercises 6.2

E1. *Write C++ functions to implement the remaining operations for the contiguous implementation of a* List, *as follows:*

(a) *The constructor* List

Answer
```
template <class List_entry>
List<List_entry> :: List()
/* Post: The List is initialized to be empty. */
{
   count = 0;
}
```

(b) clear

Answer
```
//   clear: clear the List.
/* Post: The List is cleared. */
template <class List_entry>
void List<List_entry> :: clear()
{
   count = 0;
}
```

(c) empty

Answer
```
//   empty: returns non-zero if the List is empty.
/* Post: The function returns true or false according as the List is empty or not. */
template <class List_entry>
bool List<List_entry> :: empty() const
{
   return count <= 0;
}
```

TOC
Index
Help

(d) full

Answer

```
// full: returns non-zero if the List is full.
/* Post: The function returns true or false according as the List is full or not. */
template <class List_entry>
bool List<List_entry> :: full( ) const
{
   return count >= max_list;
}
```

(e) replace

Answer

```
template <class List_entry>
Error_code List<List_entry> :: replace(int position, const List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is replaced by x, all other entries remain unchanged. Otherwise
         the function fails with an error code of range_error. */
{
   if (position < 0 || position >= count) return range_error;
   entry[position] = x;
   return success;
}
```

(f) retrieve

Answer

```
template <class List_entry>
Error_code List<List_entry> :: retrieve(int position, List_entry &x) const
/* Post: If the List is not full and 0 ≤ position < n, where n is the number of entries in the List,
         the function succeeds: The entry in position is copied to x. Otherwise the function fails
         with an error code of range_error. */
{
   if (position < 0 || position >= count) return range_error;
   x = entry[position];
   return success;
}
```

(g) remove

Answer

```
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly in position. Otherwise the function fails with a diagnostic error code. */
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
{
   if (count == 0) return underflow;
   if (position < 0 || position >= count) return range_error;
   x = entry[position];
   count−−;
   while (position < count) {
      entry[position] = entry[position + 1];
      position++;
   }
   return success;
}
```

E2. *Write C++ functions to implement the constructors (both forms) for singly linked and doubly linked* Node *objects.*

Answer The constructors for singly linked nodes are as follows.

```
template <class List_entry>
Node<List_entry> :: Node( )
{
   next = NULL;
}
template <class List_entry>
Node<List_entry> :: Node (List_entry data, Node<List_entry> *link = NULL)
{
   entry = data;
   next = link;
}
```

The constructors for doubly linked nodes are as follows.

```
template <class List_entry>
Node<List_entry> :: Node( )
{
   next = back = NULL;
}
template <class List_entry>
Node<List_entry> :: Node (List_entry data, Node<List_entry> *link_back = NULL,
                                           Node<List_entry> *link_next = NULL)
{
   entry = data;
   back = link_back;
   next = link_next;
}
```

E3. *Write C++ functions to implement the following operations for the (first) simply linked implementation of a list:*

(a) *The constructor* List

Answer

```
template <class List_entry>
List<List_entry> :: List( )
/* Post: The List is initialized to be empty. */
{
   count = 0;
   head = NULL;
}
```

(b) *The copy constructor*

Answer

```
template <class List_entry>
List<List_entry> :: List(const List<List_entry> &copy)
/* Post: The List is initialized to copy the parameter copy. */
{
   count = copy.count;
   Node<List_entry> *new_node, *old_node = copy.head;
   if (old_node == NULL) head = NULL;
   else {
      new_node = head = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry);
         new_node = new_node->next;
      }
   }
}
```

(c) *The overloaded assignment operator*

Answer

```
template <class List_entry>
void List<List_entry> :: operator = (const List<List_entry> &copy)
/* Post: The List is assigned to copy a parameter */
{
   List new_copy(copy);
   clear( );
   count = new_copy.count;
   head = new_copy.head;
   new_copy.count = 0;
   new_copy.head = NULL;
}
```

(d) *The destructor* ~List

Answer

```
template <class List_entry>
List<List_entry> :: ~List( )
/* Post: The List is empty: all entries have been removed. */
{
   clear( );
}
```

(e) clear

Answer

```
template <class List_entry>
void List<List_entry> :: clear( )
/* Post: The List is cleared. */
{
   Node<List_entry> *p, *q;
   for (p = head; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   head = NULL;
}
```

(f) size

Answer

```
template <class List_entry>
int List<List_entry> :: size( ) const
/* Post: The function returns the number of entries in the List. */
{
   return count;
}
```

(g) empty

Answer

```
template <class List_entry>
bool List<List_entry> :: empty( ) const
/* Post: The function returns true or false according as the List is empty or not. */
{
   return count <= 0;
}
```

(h) full

Answer

```
template <class List_entry>
bool List<List_entry> :: full( ) const
/* Post: The function returns 1 or 0 according as the List is full or not. */
{
   return false;
}
```

(i) replace

Answer

```
template <class List_entry>
Error_code List<List_entry> :: replace(int position, const List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is replaced by x, all other entries remain unchanged. Otherwise
         the function fails with an error code of range_error. */
{
   Node<List_entry> *current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   current->entry = x;
   return success;
}
```

(j) retrieve

Answer

```
template <class List_entry>
Error_code List<List_entry> :: retrieve(int position, List_entry &x) const
/* Post: If the List is not full and 0 ≤ position < n, where n is the number of entries in the List,
         the function succeeds: The entry in position is copied to x. Otherwise the function fails
         with an error code of range_error. */
{
   Node<List_entry> *current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   x = current->entry;
   return success;
}
```

(k) remove

Answer

```
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly in position. Otherwise the function fails with a diagnostic error code. */
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
{
   Node<List_entry> *prior, *current;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;
   if (position > 0) {
      prior = set_position(position − 1);
      current = prior->next;
      prior->next = current->next;
   }
   else {
      current = head;
      head = head->next;
   }
   x = current->entry;
   delete current;
   count−−;
   return success;
}
```

(l) traverse

Answer

```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: The action specified by function (*visit) has been performed on every entry of the List,
         beginning at position 0 and doing each in turn. */
{
   Node<List_entry> *q;
   for (q = head; q; q = q->next)
      (*visit)(q->entry);
}
```

E4. *Write* remove *for the (second) implementation of simply linked lists that remembers the last-used position.*

Answer

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry at position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly at position. Otherwise the function fails with a diagnostic error code. */
{
   Node<List_entry> *old_node;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;
   if (position > 0) {
      set_position(position − 1);
      old_node = current->next;
      current->next = old_node->next;
   }
   else {
      old_node = head;
      current = head = old_node->next;
      current_position = 0;
   }
   x = old_node->entry;
   delete old_node;
   count−−;
   return success;
}
```

E5. *Indicate which of the following functions are the same for doubly linked lists (as implemented in this section) and for simply linked lists. For those that are different, write new versions for doubly linked lists. Be sure that each function conforms to the specifications given in* Section 6.1.

(a) *The constructor* List
(b) *The copy constructor*
(c) *The overloaded assignment operator*
(d) *The destructor* ~List
(e) clear
(f) size
(g) empty
(h) full
(i) replace
(j) insert
(k) retrieve
(l) remove
(m) traverse

Answer Parts (f) through (h) all remain the same as for simply linked lists. Part (j) appears in the text; parts (a), (b), (c), (d), (e), (i), (k), (l) and (m) are listed here.

```
template <class List_entry>
List<List_entry> :: List( )
/* Post: The List is initialized to be empty. */
{
   count = 0;
   current = NULL;
   current_position = −1;
}

//    List: a copy constructor
/* Post: The List is initialized to copy the parameter copy. */
template <class List_entry>
List<List_entry> :: List(const List<List_entry> &copy)
{
   count = copy.count;
   current_position = copy.current_position;
   Node<List_entry> *new_node, *old_node = copy.current;
   if (old_node == NULL) current = NULL;
   else {
      new_node = current = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry, new_node);
         new_node = new_node->next;
      }
      old_node = copy.current;
      new_node = current;
      while (old_node->back != NULL) {
         old_node = old_node->back;
         new_node->back = new Node<List_entry>(old_node->entry, NULL, new_node);
         new_node = new_node->back;
      }
   }
}

//    List: overloaded assignment
/* Post: The List is assigned to copy a parameter */
template <class List_entry>
void List<List_entry> :: operator = (const List<List_entry> &copy)
{
   List new_copy(copy);
   clear( );
   count = new_copy.count;
   current_position = new_copy.current_position;
   current = new_copy.current;
   new_copy.count = 0;
   new_copy.current_position = 0;
   new_copy.current = NULL;
}

//    ~List: a destructor to clear the List.
/* Post: The List is empty: all entries have been removed. */
template <class List_entry>
List<List_entry> :: ~List( )
{
   clear( );
}

template <class List_entry>
void List<List_entry> :: clear( )
/* Post: The List is cleared. */
{
   Node<List_entry> *p, *q;
   if (current == NULL) return;
   for (p = current->back; p; p = q) {
      q = p->back;
      delete p;
   }
   for (p = current; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   current = NULL;
   current_position = −1;
}

template <class List_entry>
Error_code List<List_entry> :: replace(int position, const List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is replaced by x, all other entries remain unchanged. Otherwise
         the function fails with an error code of range_error. */
{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   current->entry = x;
   return success;
}

template <class List_entry>
Error_code List<List_entry> :: retrieve(int position, List_entry &x) const
/* Post: If the List is not full and 0 ≤ position < n, where n is the number of entries in the List,
         the function succeeds: The entry in position is copied to x. Otherwise the function fails
         with an error code of range_error. */
{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   x = current->entry;
   return success;
}
```

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly in position. Otherwise the function fails with a diagnostic error code. */
{
   Node<List_entry> *old_node, *neighbor;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   old_node = current;
   if (neighbor = current->back) neighbor->next = current->next;
   if (neighbor = current->next) {
      neighbor->back = current->back;
      current = neighbor;
   }
   else {
      current = current->back;
      current_position−−;
   }
   x = old_node->entry;
   delete old_node;
   count−−;
   return success;
}

template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: The action specified by function *visit has been performed on every entry of the List,
         beginning at position 0 and doing each in turn. */
{
   Node<List_entry> *to_visit = current;
   if (to_visit != NULL)                                    //  Ignore empty lists.
      for ( ; to_visit->back; to_visit = to_visit->back)   //  Find the beginning of List.
         ;
   for ( ; to_visit; to_visit = to_visit->next)
      (*visit)(to_visit->entry);
}
```

Programming Projects 6.2

P1. *Prepare a collection of files containing the declarations for a contiguous list and all the functions for list processing.*

Answer Definition file:

```
const int max_list = 5000;

template <class List_entry>
class List {
public:
```

```
//  methods of the List ADT
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);

protected:
//  data members for a contiguous list implementation
   int count;
   List_entry entry[max_list];
};
```

Implementation file:

```
template <class List_entry>
List<List_entry>::List()
/*
Post: The List is initialized to be empty.
*/

{
   count = 0;
}

//  clear:  clear the List.

/*
Post: The List is cleared.
*/

template <class List_entry>
void List<List_entry>::clear()
{
   count = 0;
}

template <class List_entry>
int List<List_entry>::size() const
/*
Post: The function returns the number of entries in the List.
*/
{
   return count;
}

//  empty: returns non-zero if the List is empty.

/*
Post: The function returns true or false
      according as the List is empty or not.
*/

template <class List_entry>
bool List<List_entry>::empty() const
{
   return count <= 0;
}
```

```
//  full: returns non-zero if the List is full.
/*
Post: The function returns true or false
      according as the List is full or not.
*/
template <class List_entry>
bool List<List_entry>::full() const
{
   return count >= max_list;
}
template <class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/*
Post: The action specified by function (*visit) has been performed
      on every entry of the List, beginning at position 0 and doing
      each in turn.
*/
{
   for (int i = 0; i < count; i++)
      (*visit)(entry[i]);
}
template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/*
Post: If the List is not full and 0 <= position <= n,
      where n is the number of entries in the List,
      the function succeeds:
      Any entry formerly at
      position and all later entries have their
      position numbers increased by 1 and
      x is inserted at position of the List.
Else: The function fails with a diagnostic error code.
*/
{
   if (full())
      return overflow;
   if (position < 0 || position > count)
      return range_error;
   for (int i = count - 1; i >= position; i--)
      entry[i + 1] = entry[i];
   entry[position] = x;
   count++;
   return success;
}
template <class List_entry>
Error_code List<List_entry>::retrieve(int position, List_entry &x) const
/*
Post: If the List is not full and 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is copied to x.
      Otherwise the function fails with an error code of range_error.
*/
```

```
{
   if (position < 0 || position >= count) return range_error;
   x = entry[position];
   return success;
}

template <class List_entry>
Error_code List<List_entry>::replace(int position, const List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is replaced by x,
      all other entries remain unchanged.
      Otherwise the function fails with an error code of range_error.
*/

{
   if (position < 0 || position >= count) return range_error;
   entry[position] = x;
   return success;
}

/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is removed
      from the List, and the entries in all later positions
      have their position numbers decreased by 1.
      The parameter x records a copy of
      the entry formerly in position.
      Otherwise the function fails with a diagnostic error code.
*/

template <class List_entry>
Error_code List<List_entry>::remove(int position, List_entry &x)
{
   if (count == 0) return underflow;
   if (position < 0 || position >= count) return range_error;

   x = entry[position];
   count--;
   while (position < count) {
      entry[position] = entry[position + 1];
      position++;
   }
   return success;
}
```

Driver program:

```
#include   "../../c/utility.h"
#include   "list.h"
#include   "list.cpp"

void write_entry(char &c)
{
   cout << c;
}
```

```
main()
{
   char x;
   List<char> c_list;   //  a list of characters, initialized empty

   cout << "List is empty, it has " << c_list.size() << " items.\n"
        << "Enter characters and the program adds them to the list.\n"
        << "Use Enter key (newline) to terminate the input.\n When "
        << "size() is 3, the element will be inserted at the front of "
        << "the list.\n The other ones will appear at the back.\n";
   while (!c_list.full() && (x = cin.get()) != '\n')
      if (c_list.size() == 3) c_list.insert(0, x);
      else c_list.insert(c_list.size(), x);
   cout << "The list has " << c_list.size() << " items.\n";
   cout << "The function c_list.full(), got " << c_list.full();
   if (c_list.full()) cout << " because the list is full.\n";
   else cout << " because the list is NOT full.\n";
   cout << "c_list.empty(), expecting 0, got "
        << c_list.empty() << ".\n";
   cout << "c_list.traverse(write_entry) output:  ";
   c_list.traverse(write_entry);
   cout << "\n";
}
```

P2. *Write a menu-driven demonstration program for general lists, based on the one in Section 3.4. The list entries should be characters. Use the declarations and the functions for contiguous lists developed in Project P1.*

Answer See Project P2 for the List class. Menu-driven demonstration program:

```
#include "../../c/utility.h"

void introduction()
/*
POST: An introduction has been written to the terminal.
*/

{
  cout << "\n\nList Demonstration Program.\n\n"
       << "This program is intended for verifying list operations. The"
       << "\nlists are defined to have single character entries which\n"
       << "also act as the keys.\n\n" << endl;

  cout << "This version works with a contiguous list implementation "
       << endl;

  cout << "\nInsertions and deletions must be specified by list position."
       << "\nThe HEAD of the list is located at position 0 " << endl;

  cout << "Valid commands are shown at the prompt.\n"
       << "Both upper and lower case letters can be used.\n";
}

void help()
/*
PRE:  None.
POST: Instructions for the list operations have been printed.
*/
```

```
{
    cout <<  "Valid commands are:\n"
         <<  "\tA - Append value to the end of the list.\n"
         <<  "\tI - Insert value into the list.\n"
         <<  "\tD - Delete value from the list.\n"
         <<  "\tR - Replace a value in the list.\n"
         <<  "\tF - Fetch an entry from the list.\n"
         <<  "\tT - Traverse the list and print each entry.\n"
         <<  "\tS - The current size of the list.\n"
         <<  "\t[H]elp    [Q]uit." << endl;
}

#include <string.h>
char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands for
      the list demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("aidrftshq",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}

char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include list data structure

#include   "../contlist/list.h"
#include   "../contlist/list.cpp"
```

```
int do_command(char c, List<char> &test_list)
/*
PRE:  The list has been created and the command c is a valid list
      operation.
POST: The command has been executed.
USES: All the functions that perform list operations.
*/

{
   char x;
   int position;
   switch (c) {
   case 'h': help();
     break;

   case 's':
      cout << "The size of the list is " << test_list.size() << "\n";
      break;

   case 'a':
      position = test_list.size();
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;

   case 'i':
      cout << "Enter a list position to insert at: " << flush;
      cin >> position;
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;

   case 'r':
      cout << "Enter a list position to replace at: " << flush;
      cin >> position;
      cout << "Enter new character to use: " << flush;
      x = get_char();
      test_list.replace(position, x);
      break;

   case 'd':
      cout << "Enter a list position to remove from: " << flush;
      cin >> position;
      test_list.remove(position, x);
      break;

   case 'f':
      cout << "Enter a list position to fetch: " << flush;
      cin >> position;
      if (test_list.retrieve(position, x) == success)
         cout << "Retrieved entry is: " << x << endl;
      else
         cout << "Failed to retrieve such an entry." << endl;
      break;

   case 't':
      test_list.traverse(write_ent);
      cout << endl;
      break;
   case 'q':
      cout << "Extended list demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/*
PRE:  None.
POST: A list demonstration has been performed.
USES: get_command, do_command, List methods
*/

{
   introduction();
   help();
   List<char> test_list;
   while (do_command(get_command(), test_list));
}
```

P3. *Create a collection of files containing declarations and functions for processing linked lists.*

(a) *Use the simply linked lists as first implemented.*

Answer Definition file:

```
template <class Node_entry>
struct Node {
//  data members
   Node_entry entry;
   Node<Node_entry> *next;
//  constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link = NULL);
};

template <class List_entry>
class List {
public:
//  Specifications for the methods of the list ADT go here.

   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);

//  The following methods replace compiler-generated defaults.
   ~List();
   List(const List<List_entry> &copy);
   void operator =(const List<List_entry> &copy);
protected:
//  Data members for the linked list implementation now follow.
   int count;
   Node<List_entry> *head;
//  The following auxiliary function is used to locate list positions
   Node<List_entry> *set_position(int position) const;
};

template <class List_entry>
Node<List_entry>::Node()
{
   next = NULL;
}

template <class List_entry>
Node<List_entry>::Node (List_entry data, Node<List_entry> *link = NULL)
{
   entry = data;
   next = link;
}
```

Implementation file:

```
template <class List_entry>
List<List_entry>::List()
/*
Post: The List is initialized to be empty.
*/
{
   count = 0;
   head = NULL;
}

template <class List_entry>
void List<List_entry>::clear()
/*
Post: The List is cleared.
*/
{
   Node<List_entry> *p, *q;

   for (p = head; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   head = NULL;
}

template <class List_entry>
int List<List_entry>::size() const
/*
Post: The function returns the number of entries in the List.
*/
{
   return count;
}

template <class List_entry>
bool List<List_entry>::empty() const
/*
Post: The function returns true or false according as the
      List is empty or not.
*/
```

```
{
   return count <= 0;
}

template <class List_entry>
bool List<List_entry>::full() const
/*
Post: The function returns 1 or 0 according as the
      List is full or not.
*/

{
   return false;
}

template <class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/*
Post: The action specified by function (*visit) has been performed
      on every entry of the List, beginning at position 0 and doing
      each in turn.
*/

{
   Node<List_entry> *q;

   for (q = head; q; q = q->next)
      (*visit)(q->entry);
}

template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/*
Post: If the List is not full and 0 <= position <= n,
      where n is the number of entries in the List, the function succeeds:
      Any entry formerly at position and all later entries have their
      position numbers increased by 1, and x is inserted at position
      of the List.
Else: The function fails with a diagnostic error code.
*/

{
   if (position < 0 || position > count)
      return range_error;
   Node<List_entry> *new_node, *previous, *following;
   if (position > 0) {
      previous = set_position(position - 1);
      following = previous->next;
   }
   else following = head;
   new_node = new Node<List_entry>(x, following);
   if (new_node == NULL)
      return overflow;
   if (position == 0)
      head = new_node;
   else
      previous->next = new_node;
   count++;
   return success;
}
```

```
template <class List_entry>
Error_code List<List_entry>::retrieve(int position, List_entry &x) const
/*
Post: If the List is not full and 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is copied to x.
      Otherwise the function fails with an error code of range_error.
*/
{
   Node<List_entry> *current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   x = current->entry;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::replace(int position, const List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is replaced by x,
      all other entries remain unchanged.
      Otherwise the function fails with an error code of range_error.
*/
{
   Node<List_entry> *current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   current->entry = x;
   return success;
}

/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is removed
      from the List, and the entries in all later positions
      have their position numbers decreased by 1.
      The parameter x records a copy of
      the entry formerly in position.
      Otherwise the function fails with a diagnostic error code.
*/

template <class List_entry>
Error_code List<List_entry>::remove(int position, List_entry &x)
{
   Node<List_entry> *prior, *current;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;

   if (position > 0) {
      prior = set_position(position - 1);
      current = prior->next;
      prior->next = current->next;
   }
   else {
      current = head;
      head = head->next;
   }

   x = current->entry;
   delete current;
   count--;
   return success;
}

template <class List_entry>
Node<List_entry> *List<List_entry>::set_position(int position) const
/*
Pre:  position is a valid position in the List;
      0 <= position < count.
Post: Returns a pointer to the Node in position.
*/

{
   Node<List_entry> *q = head;
   for (int i = 0; i < position; i++) q = q->next;
   return q;
}

template <class List_entry>
List<List_entry>::~List()
/*
Post: The List is empty: all entries have been removed.
*/

{
   clear();
}

template <class List_entry>
List<List_entry>::List(const List<List_entry> &copy)
/*
Post: The List is initialized to copy the parameter copy.
*/

{
   count = copy.count;
   Node<List_entry> *new_node, *old_node = copy.head;

   if (old_node == NULL) head = NULL;
   else {
      new_node = head = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry);
         new_node = new_node->next;
      }
   }
}

template <class List_entry>
void List<List_entry>::operator =(const List<List_entry> &copy)
/*
Post: The List is assigned to copy a parameter
*/
```

```
{
   List new_copy(copy);
   clear();
   count = new_copy.count;
   head = new_copy.head;
   new_copy.count = 0;
   new_copy.head = NULL;
}
```

Simple driver program:

```
#include   "../../c/utility.h"
#include   "../../c/utility.cpp"
#include   "list.h"
#include   "list.cpp"

void write_entry(char &c)
{
   cout << c;
}

main()
{
  char x;
  List<char> c_list;  //  a list of characters, initialized empty

  cout << "List is empty, it has " << c_list.size() << " items.\n";
  cout << "Enter characters and the program adds them to the list.\n";
  cout << "Use Enter key (newline) to terminate the input.\n";
  cout << "When ListSize() is 3, the element will be inserted at the ";
  cout << "front of the list.\n The others will appear at the back.\n";
  while (!c_list.full() && (x = cin.get()) != '\n')
     if (c_list.size() == 3) c_list.insert(0, x);
     else c_list.insert(c_list.size(), x);
  cout << "The list has " << c_list.size() << " items.\n";
  cout << "The function c_list.full(), got " << c_list.full();
  if (c_list.full()) cout << " because the list is full.\n";
  else cout << " because the list is NOT full.\n";
  cout << "c_list.empty(), expecting 0, got " << c_list.empty() << ".\n";
  cout << "c_list.traverse(write_entry) output:  ";
  c_list.traverse(write_entry);
  cout << "\n";
  c_list.clear();
  cout << "Cleared the list, printing its contents:\n";
  c_list.traverse(write_entry);
  cout << "\n";
  cout << "Enter characters and the program adds them to the list.\n";
  cout << "Use Enter key (newline) to terminate the input.\n";
  cout << "When ListSize() is < 3, the element will be inserted at the ";
  cout << "front of the list.\n The others will appear at the back.\n";
  while (!c_list.full() && (x = cin.get()) != '\n')
     if (c_list.size() < 3) c_list.insert(0, x);
     else c_list.insert(c_list.size(), x);
  c_list.traverse(write_entry);
  cout << "\n";
}
```

(b) *Use the simply linked lists that maintain a pointer to the last-used position.*

Answer Definition file:

```
template <class List_entry>
struct Node {
   List_entry entry;
   Node<List_entry> *next;
   Node();
   Node(List_entry, Node<List_entry> *link = NULL);
};

template <class List_entry>
class List {
public:

   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);
   ~List();
   List(const List<List_entry> &copy);
   void operator =(const List<List_entry> &copy);

//  Add specifications for the methods of the list ADT.
//  Add methods to replace the compiler-generated defaults.

protected:
//  Data members for the linked-list implementation with
//  current position follow:
   int count;
   mutable int current_position;
   Node<List_entry> *head;
   mutable Node<List_entry> *current;

//  Auxiliary function to locate list positions follows:
   void set_position(int position) const;
};

template <class List_entry>
Node<List_entry>::Node()
{
   next = NULL;
}

template <class List_entry>
Node<List_entry>::Node (List_entry data, Node<List_entry> *link = NULL)
{
   entry = data;
   next = link;
}
```

Implementation file:

TOC
Index
Help

```
template <class List_entry>
List<List_entry>::List()
/*
Post: The List is initialized to be empty.
*/

{
   count = 0;
   current = head = NULL;
   current_position = -1;
}

template <class List_entry>
void List<List_entry>::clear()
/*
Post: The List is cleared.
*/

{
   Node<List_entry> *p, *q;

   for (p = head; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   current = head = NULL;
   current_position = -1;
}

template <class List_entry>
int List<List_entry>::size() const
/*
Post: The function returns the number of entries in the List.
*/

{
   return count;
}

template <class List_entry>
bool List<List_entry>::empty() const
/*
Post: The function returns true or false according as the List
      is empty or not.
*/

{
   return count <= 0;
}

template <class List_entry>
bool List<List_entry>::full() const
/*
Post: The function returns true or false according as the
      List is full or not.
*/

{
   return false;
}
```

```
template <class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/*
Post: The action specified by function *f has been performed on every
      entry of the List, beginning at position 0 and doing each in turn.
*/

{
   Node<List_entry> *to_visit;

   for (to_visit = head; to_visit; to_visit = to_visit->next)
      (*visit)(to_visit->entry);
}

template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/*
Post: If the List is not full and 0 <= position <= n,
      where n is the number of entries in the List,
      the function succeeds:

      Any entry formerly at
      position and all later entries have their
      position numbers increased by 1 and
      x is inserted at position in the List.

Else: the function fails with a diagnostic error code.
*/

{
   Node<List_entry> *new_node;

   if (position < 0 || position > count) return range_error;
   new_node = new Node<List_entry>;

   if (new_node == NULL) return fail;
   new_node->entry = x;

   if (position == 0) {
      new_node->next = head;
      current = head = new_node;
      current_position = 0;
   } else {
      set_position(position - 1);
      new_node->next = current->next;
      current->next = new_node;
   }
   count++;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::retrieve(int position, List_entry &x) const
/*
Post: If the List is not full and 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry at position is copied to x.
      Otherwise the function fails with an error code of range_error.
*/
```

TOC
Index
Help

```
{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   x = current->entry;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::replace(int position, const List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry at position is replaced by x,
      all other entries remain unchanged.
      Otherwise the function fails with an error code of range_error.
*/

{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   current->entry = x;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::remove(int position, List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry at position is removed
      from the List, and the entries in all later positions
      have their position numbers decreased by 1.
      The parameter x records a copy of
      the entry formerly at position.
      Otherwise the function fails with a diagnostic error code.
*/

{
   Node<List_entry> *old_node;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;

   if (position > 0) {
      set_position(position - 1);
      old_node = current->next;
      current->next = old_node->next;
   }

   else {
      old_node = head;
      current = head = old_node->next;
      current_position = 0;
   }

   x = old_node->entry;
   delete old_node;
   count--;
   return success;
}
```

```
template <class List_entry>
void List<List_entry>::set_position(int position) const
/*
Pre:  position is a valid position in the List:
      0 <= position < count.
Post: The current Node pointer references the Node at position.
*/

{
   if (position < current_position) { // must start over at head of list
      current_position = 0;
      current = head;
   }
   for ( ; current_position != position; current_position++)
      current = current->next;
}

template <class List_entry>
List<List_entry>::~List()
/*
Post: The List is empty: all entries have been removed.
*/

{
   clear();
}

template <class List_entry>
List<List_entry>::List(const List<List_entry> &copy)
/*
Post: The List is initialized to copy the parameter copy.
*/

{
   count = copy.count;
   current_position = 0;
   Node<List_entry> *new_node, *old_node = copy.head;

   if (old_node == NULL) current =  head = NULL;
   else {
      new_node = current = head = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry);
         new_node = new_node->next;
      }
      set_position(copy.current_position);
   }
}

template <class List_entry>
void List<List_entry>::operator =(const List<List_entry> &original)
/*
Post: The List is assigned to copy a parameter
*/
```

TOC
Index
Help

```
{
   List new_copy(original);
   clear();
   count = new_copy.count;
   current_position = new_copy.current_position;
   head = new_copy.head;
   current = new_copy.current;
   new_copy.count = 0;
   new_copy.current_position = 0;
   new_copy.head = NULL;
   new_copy.current = NULL;
}
```

Simple driver program:

```
#include   "../../c/utility.h"
#include   "list.h"
#include   "list.cpp"

void write_entry(char &c)
{
   cout << c;
}

main()
{
   char x;
   List<char> c_list;  //  a list of characters, initialized empty

   cout << "List is empty, it has " << c_list.size() << " items.\n";
   cout << "Enter characters and the program adds them to the list.\n";
   cout << "Use Enter key (newline) to terminate the input.\n";
   cout << "When ListSize() is 3, the element will be inserted at the ";
   cout << "front of the list.\n The others will appear at the back.\n";
   while (!c_list.full() && (x = cin.get()) != '\n')
      if (c_list.size() == 3) c_list.insert(0, x);
      else c_list.insert(c_list.size(), x);
   cout << "The list has " << c_list.size() << " items.\n";
   cout << "The function c_list.full(), got " << c_list.full();
   if (c_list.full()) cout << " because the list is full.\n";
   else cout << " because the list is NOT full.\n";
   cout << "c_list.empty(), expecting 0, got " << c_list.empty() << ".\n";
   cout << "c_list.traverse(write_entry) output:  ";
   c_list.traverse(write_entry);
   cout << "\n";
   c_list.clear();
   cout << "Cleared the list, printing its contents:\n";
   c_list.traverse(write_entry);
   cout << "\n";
   cout << "Enter characters and the program adds them to the list.\n";
   cout << "Use Enter key (newline) to terminate the input.\n";
   cout << "When ListSize() is < 3, the element will be inserted at the ";
   cout << "front of the list.\n The others will appear at the back.\n";
   while (!c_list.full() && (x = cin.get()) != '\n')
      if (c_list.size() < 3) c_list.insert(0, x);
      else c_list.insert(c_list.size(), x);
   c_list.traverse(write_entry);
   cout << "\n";
}
```

(c) *Use doubly linked lists as implemented in this section.*

Answer Definition file:

```
template <class Node_entry>
struct Node {
//  data members
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//  constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};

template <class List_entry>
class List {
public:

   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);
   ~List();
   List(const List<List_entry> &copy);
   void operator =(const List<List_entry> &copy);

//  Add specifications for methods of the list ADT.
//  Add methods to replace compiler generated defaults.
protected:
//  Data members for the doubly-linked list implementation follow:
   int count;
   mutable int current_position;
   mutable Node<List_entry> *current;

//  The auxiliary function to locate list positions follows:
   void set_position(int position) const;
};

template <class List_entry>
Node<List_entry>::Node()
{
   next = back = NULL;
}

template <class List_entry>
Node<List_entry>::Node (List_entry data,
                        Node<List_entry> *link_back = NULL,
                        Node<List_entry> *link_next = NULL)
{
   entry = data;
   back = link_back;
   next = link_next;
}
```

Implementation file:

```
template <class List_entry>
List<List_entry>::List()
/*
Post: The List is initialized to be empty.
*/

{
   count = 0;
   current = NULL;
   current_position = -1;
}

template <class List_entry>
void List<List_entry>::clear()
/*
Post: The List is cleared.
*/

{
   Node<List_entry> *p, *q;

   if (current == NULL) return;
   for (p = current->back; p; p = q) {
      q = p->back;
      delete p;
   }

   for (p = current; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   current = NULL;
   current_position = -1;
}

template <class List_entry>
int List<List_entry>::size() const
/*
Post: The function returns the number of entries in the List.
*/

{
   return count;
}

template <class List_entry>
bool List<List_entry>::empty() const
/*
Post: The function returns true or false according as the
      List is empty or not.
*/

{
   return count <= 0;
}
```

```
template <class List_entry>
bool List<List_entry>::full() const
/*
Post: The function returns true or false according as the
      List is full or not.
*/

{
   return false;
}

template <class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/*
Post: The action specified by function *visit has been
      performed on every entry of the List, beginning at
      position 0 and doing each in turn.
*/

{
   Node<List_entry> *to_visit = current;

   if (to_visit != NULL)   //  Ignore empty lists.
      for ( ; to_visit->back; to_visit = to_visit->back)
      //  Find the beginning of List.
         ;
   for ( ; to_visit; to_visit = to_visit->next)
      (*visit)(to_visit->entry);
}

template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/*
Post: If the List is not full and 0 <= position <= n,
      where n is the number of entries in the List,
      the function succeeds:

      Any entry formerly at
      position and all later entries have their
      position numbers increased by 1 and
      x is inserted at position of the List.

Else: the function fails with a diagnostic error code.
*/

{
   Node<List_entry> *new_node, *following, *preceding;

   if (position < 0 || position > count) return range_error;

   if (position == 0) {
      if (count == 0) following = NULL;
      else {
         set_position(0);
         following = current;
      }
      preceding = NULL;
   }
   else {
      set_position(position - 1);
      preceding = current;
      following = preceding->next;
   }
   new_node = new Node<List_entry>(x, preceding, following);

   if (new_node == NULL) return overflow;
   if (preceding != NULL) preceding->next = new_node;
   if (following != NULL) following->back = new_node;
   current = new_node;
   current_position = position;
   count++;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::retrieve(int position,
                                      List_entry &x) const
/*
Post: If the List is not full and 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is copied to x.
      Otherwise the function fails with an error code of range_error.
*/

{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   x = current->entry;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::replace(int position, const List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is replaced by x,
      all other entries remain unchanged.
      Otherwise the function fails with an error code of range_error.
*/

{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   current->entry = x;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::remove(int position, List_entry &x)
/*
```

```
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is removed
      from the List, and the entries in all later positions
      have their position numbers decreased by 1.
      The parameter x records a copy of
      the entry formerly in position.
      Otherwise the function fails with a diagnostic error code.
*/

{
   Node<List_entry> *old_node, *neighbor;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;

   set_position(position);
   old_node = current;
   if (neighbor = current->back) neighbor->next = current->next;
   if (neighbor = current->next) {
      neighbor->back = current->back;
      current = neighbor;
   }

   else {
      current = current->back;
      current_position--;
   }

   x = old_node->entry;
   delete old_node;
   count--;
   return success;
}

template <class List_entry>
void List<List_entry>::set_position(int position) const
/*
Pre:  position is a valid position in the List:
      0 <= position < count.
Post: The current Node pointer references the Node at position.
*/

{
   if (current_position <= position)
      for ( ; current_position != position; current_position++)
         current = current->next;
   else
      for ( ; current_position != position; current_position--)
         current = current->back;
}

//  ~List:  a destructor to clear the List.

/*
Post: The List is empty: all entries have been removed.
*/
```

TOC
Index
Help

```
template <class List_entry>
List<List_entry>::~List()
{
   clear();
}

//  List:  a copy constructor

/*
Post: The List is initialized to copy the parameter copy.
*/

template <class List_entry>
List<List_entry>::List(const List<List_entry> &copy)
{
   count = copy.count;
   current_position = copy.current_position;
   Node<List_entry> *new_node, *old_node = copy.current;

   if (old_node == NULL) current = NULL;
   else {
      new_node = current = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry, new_node);
         new_node = new_node->next;
      }

      old_node = copy.current;
      new_node = current;
      while (old_node->back != NULL) {
         old_node = old_node->back;
         new_node->back = new Node<List_entry>(old_node->entry,
                                               NULL, new_node);
         new_node = new_node->back;
      }
   }
}

//  List:  overloaded assignment

/*
Post: The List is assigned to copy a parameter
*/

template <class List_entry>
void List<List_entry>::operator =(const List<List_entry> &copy)
{
   List new_copy(copy);

   clear();
   count = new_copy.count;

   current_position = new_copy.current_position;
   current = new_copy.current;
   new_copy.count = 0;
   new_copy.current_position = 0;
   new_copy.current = NULL;
}
```

Simple driver program:

TOC
Index
Help

```
#include   "../../c/utility.h"
#include   "list.h"
#include   "list.cpp"

//  test List mechanism
void write_entry(char &c)
{
   cout << c;
}

main()
{
  char x;
  List<char> c_list;  //  a list of characters, initialized empty

  cout << "List is empty, it has " << c_list.size() << " items.\n";
  cout << "Enter characters and the program adds them to the list.\n";
  cout << "Use Enter key (newline) to terminate the input.\n";
  cout << "When ListSize() is 3, the element will be inserted at the ";
  cout << "front of the list.\n The others will appear at the back.\n";
  while (!c_list.full() && (x = cin.get()) != '\n')
     if (c_list.size() == 3) c_list.insert(0, x);
     else c_list.insert(c_list.size(), x);
  cout << "The list has " << c_list.size() << " items.\n";
  cout << "The function c_list.full(), got " << c_list.full();
  if (c_list.full()) cout << " because the list is full.\n";
  else cout << " because the list is NOT full.\n";
  cout << "c_list.empty(), expecting 0, got " << c_list.empty() << ".\n";
  cout << "c_list.traverse(write_entry) output:  ";
  c_list.traverse(write_entry);
  cout << "\n";
  c_list.clear();
  cout << "Cleared the list, printing its contents:\n";
  c_list.traverse(write_entry);
  cout << "\n";
  cout << "Enter characters and the program adds them to the list.\n";
  cout << "Use Enter key (newline) to terminate the input.\n";
  cout << "When ListSize() is < 3, the element will be inserted at the ";
  cout << "front of the list.\n The others will appear at the back.\n";
  while (!c_list.full() && (x = cin.get()) != '\n')
     if (c_list.size() < 3) c_list.insert(0, x);
     else c_list.insert(c_list.size(), x);
  c_list.traverse(write_entry);
  cout << "\n";
}
```

P4. *In the menu-driven demonstration program of Project P2, substitute the collection of files with declarations and functions that support linked lists (from Project P3) for the files that support contiguous lists (Project P1). If you have designed the declarations and the functions carefully, the program should operate correctly with no further change required.*

Answer The programs for each implementation are identical, apart from their included list implementations and a message that describes the implementation being used. Only the program for simply linked lists is given here.

```
#include "../../c/utility.h"
```

```
void introduction()
/*
POST: An introduction has been written to the terminal.
*/

{
  cout << "\n\nList Demonstration Program.\n\n"
       << "This program is intended for verifying list operations. The\n"
       << "lists are defined to have single character entries which\n"
       << "also act as the keys.\n\n" << endl;

  cout << "This version works with a linked list implementation " << endl;

  cout << "\nInsertions and deletions must be specified by list position."
       << "\nThe HEAD of the list is located at position 0 " << endl;

  cout << "Valid commands are shown at the prompt.\n"
       << "Both upper and lower case letters can be used.\n";
}

void help()
/*
PRE:  None.
POST: Instructions for the list operations have been printed.
*/

{
    cout <<  "Valid commands are:\n"
         <<  "\tA - Append value to the end of the list.\n"
         <<  "\tI - Insert value into the list.\n"
         <<  "\tD - Delete value from the list.\n"
         <<  "\tR - Replace a value in the list.\n"
         <<  "\tF - Fetch an entry from the list.\n"
         <<  "\tT - Traverse the list and print each entry.\n"
         <<  "\tS - The current size of the list.\n"
         <<  "\t[H]elp    [Q]uit." << endl;
}

#include <string.h>
char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands for the
      list demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if(strchr("aidrftshq",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}

char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include list data structure

#include   "../linklist/list.h"
#include   "../linklist/list.cpp"

int do_command(char c, List<char> &test_list)
/*
PRE:  The list has been created and the command c is a valid list
      operation.
POST: The command has been executed.
USES: All the functions that perform list operations.
*/

{
   char x;
   int position;
   switch (c) {
   case 'h': help();
      break;

   case 's':
      cout << "The size of the list is " << test_list.size() << "\n";
      break;

   case 'a':
      position = test_list.size();
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;

   case 'i':
      cout << "Enter a list position to insert at: " << flush;
      cin >> position;
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;
   case 'r':
      cout << "Enter a list position to replace at: " << flush;
      cin >> position;
      cout << "Enter new character to use: " << flush;
      x = get_char();
      test_list.replace(position, x);
      break;

   case 'd':
      cout << "Enter a list position to remove from: " << flush;
      cin >> position;
      test_list.remove(position, x);
      break;

   case 'f':
      cout << "Enter a list position to fetch: " << flush;
      cin >> position;
      if (test_list.retrieve(position, x) == success)
         cout << "Retrieved entry is: " << x << endl;
      else
         cout << "Failed to retrieve such an entry." << endl;
      break;

   case 't':
      test_list.traverse(write_ent);
      cout << endl;
      break;

   case 'q':
      cout << "Extended list demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/*
PRE:  None.
POST: A list demonstration has been performed.
USES: get_command, do_command, List methods
*/

{
   introduction();
   help();
   List<char> test_list;
   while (do_command(get_command(), test_list));
}
```

P5. (a) *Modify the implementation of doubly linked lists so that, along with the pointer to the last-used position, it will maintain pointers to the first node and to the last node of the list.*

Answer The package has been adapted from Projects P3 and P4 to make use of pointers to the head and tail of the list. The major change was made in function `set_position` which now determines the closest start point to the target before running down the list to find it. The methods `insert` and `remove` include minor changes in order to update the positions of the additional pointers. The constructor must also accomodate the additional data members.

Definition:

```
template <class Node_entry>
struct Node {
//  data members
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//  constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};

template <class List_entry>
class List {
public:

   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);
   ~List();
   List(const List<List_entry> &copy);
   void operator =(const List<List_entry> &copy);

//  Add specifications for methods of the list ADT.
//  Add methods to replace compiler generated defaults.
protected:
//  Data members for the doubly-linked list implementation follow:
   int count;
   mutable int current_position;
   mutable Node<List_entry> *current;
   Node<List_entry> *first, *last;

//  The auxiliary function to locate list positions follows:
   void set_position(int position) const;
};

template <class List_entry>
Node<List_entry>::Node()
{
   next = back = NULL;
}

template <class List_entry>
Node<List_entry>::Node (List_entry data,
                        Node<List_entry> *link_back = NULL,
                        Node<List_entry> *link_next = NULL)
{
   entry = data;
   back = link_back;
   next = link_next;
}
```

Implementation:

```
template <class List_entry>
List<List_entry>::List()
/*
Post: The List is initialized to be empty.
*/
{
   count = 0;
   first = last = current = NULL;
   current_position = -1;
}

template <class List_entry>
void List<List_entry>::clear()
/*
Post: The List is cleared.
*/
{
   Node<List_entry> *p, *q;

   if (current == NULL) return;
   for (p = current->back; p; p = q) {
      q = p->back;
      delete p;
   }

   for (p = current; p; p = q) {
      q = p->next;
      delete p;
   }
   count = 0;
   first = last = current = NULL;
   current_position = -1;
}

template <class List_entry>
int List<List_entry>::size() const
/*
Post: The function returns the number of entries in the List.
*/
{
   return count;
}

template <class List_entry>
bool List<List_entry>::empty() const
/*
Post: The function returns true or false according as the
      List is empty or not.
*/
{
   return count <= 0;
}

template <class List_entry>
bool List<List_entry>::full() const
/*
Post: The function returns true or false according as the
      List is full or not.
*/
```

```
{
   return false;
}

template <class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/*
Post: The action specified by function *visit has been
      performed on every entry of the List, beginning at
      position 0 and doing each in turn.
*/

{
   Node<List_entry> *to_visit = first;
   for ( ; to_visit; to_visit = to_visit->next)
      (*visit)(to_visit->entry);
}

template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/*
Post: If the List is not full and 0 <= position <= n,
      where n is the number of entries in the List, the function succeeds:
      Any entry formerly at position and all later entries have their
      position numbers increased by 1 and x is inserted at position
      of the List.
Else: the function fails with a diagnostic error code.
*/

{
   Node<List_entry> *new_node, *following, *preceding;

   if (position < 0 || position > count) return range_error;

   if (position == 0) {
      if (count == 0) following = NULL;
      else {
         set_position(0);
         following = current;
      }
      preceding = NULL;
   }

   else {
      set_position(position - 1);
      preceding = current;
      following = preceding->next;
   }
   new_node = new Node<List_entry>(x, preceding, following);

   if (new_node == NULL) return overflow;
   if (preceding != NULL) preceding->next = new_node;
   if (following != NULL) following->back = new_node;
   current = new_node;
   current_position = position;
   if (position == 0) first = current;
   if (position == count) last = current;
   count++;
   return success;
}
```

```
template <class List_entry>
Error_code List<List_entry>::retrieve(int position, List_entry &x) const
/*
Post: If the List is not full and 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is copied to x.
      Otherwise the function fails with an error code of range_error.
*/
{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   x = current->entry;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::replace(int position, const List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is replaced by x,
      all other entries remain unchanged.
      Otherwise the function fails with an error code of range_error.
*/
{
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   current->entry = x;
   return success;
}

template <class List_entry>
Error_code List<List_entry>::remove(int position, List_entry &x)
/*
Post: If 0 <= position < n,
      where n is the number of entries in the List,
      the function succeeds:
      The entry in position is removed
      from the List, and the entries in all later positions
      have their position numbers decreased by 1.
      The parameter x records a copy of
      the entry formerly in position.
      Otherwise the function fails with a diagnostic error code.
*/
{
   Node<List_entry> *old_node, *neighbor;
   if (count == 0) return fail;
   if (position < 0 || position >= count) return range_error;

   set_position(position);
   old_node = current;
   if (neighbor = current->back) neighbor->next = current->next;
   if (neighbor = current->next) {
      neighbor->back = current->back;
      current = neighbor;
   }
   else {
      last = current = current->back;
      current_position--;
   }

   x = old_node->entry;
   delete old_node;
   count--;
   if (position == 0) first = current;
   return success;
}

template <class List_entry>
void List<List_entry>::set_position(int position) const
/*
Pre:  position is a valid position in the List:
      0 <= position < count.
Post: The current Node pointer references the Node at position.
*/

{
   if (position < current_position - position) {
       current_position = 0;
       current = first;
   }
   else if (count - position < position - current_position) {
       current_position = count - 1;
       current = last;
   }

   if (current_position <= position)
      for ( ; current_position != position; current_position++)
         current = current->next;
   else
      for ( ; current_position != position; current_position--)
         current = current->back;
}

//  ~List:  a destructor to clear the List.

/*
Post: The List is empty: all entries have been removed.
*/

template <class List_entry>
List<List_entry>::~List()
{
   clear();
}

//  List:  a copy constructor

/*
Post: The List is initialized to copy the parameter copy.
*/

template <class List_entry>
List<List_entry>::List(const List<List_entry> &copy)
{
   count = copy.count;
   current_position = copy.current_position;
   Node<List_entry> *new_node, *old_node = copy.current;
```

TOC
Index
Help

```
   if (old_node == NULL) current = NULL;
   else {
      new_node = current = new Node<List_entry>(old_node->entry);
      while (old_node->next != NULL) {
         old_node = old_node->next;
         new_node->next = new Node<List_entry>(old_node->entry, new_node);
         new_node = new_node->next;
      }

      old_node = copy.current;
      new_node = current;
      while (old_node->back != NULL) {
         old_node = old_node->back;
         new_node->back =
                 new Node<List_entry>(old_node->entry, NULL, new_node);
         new_node = new_node->back;
      }
   }
}

//  List:  overloaded assignment

/*
Post: The List is assigned to copy a parameter
*/

template <class List_entry>
void List<List_entry>::operator =(const List<List_entry> &copy)
{
   List new_copy(copy);

   clear();
   count = new_copy.count;

   current_position = new_copy.current_position;
   current = new_copy.current;
   new_copy.count = 0;
   new_copy.current_position = 0;
   new_copy.current = NULL;
}
```

(b) *Use this implementation with the menu-driven demonstration program of Project P2 and thereby test that it is correct.*

Answer Modified demonstration program:

```
#include "../../c/utility.h"

void introduction()
/*
POST: An introduction has been written to the terminal.
*/

{
  cout << "\n\nList Demonstration Program.\n\n"
       << "This program is intended for verifying list operations. The\n"
       << "lists are defined to have single character entries which\n"
       << "also act as the keys.\n\n" << endl;

  cout << "This version works with a doubly linked list implementation "
       << "(with first and last Node pointers)." << endl;
  cout << "\nInsertions, deletions must be specified by list position.\n"
       << "The HEAD of the list is located at position 0 " << endl;

  cout << "Valid commands are shown at the prompt.\n"
       << "Both upper and lower case letters can be used.\n";
}

void help()
/*
PRE:  None.
POST: Instructions for the list operations have been printed.
*/

{
    cout <<  "Valid commands are:\n"
         <<  "\tA - Append value to the end of the list.\n"
         <<  "\tI - Insert value into the list.\n"
         <<  "\tD - Delete value from the list.\n"
         <<  "\tR - Replace a value in the list.\n"
         <<  "\tF - Fetch an entry from the list.\n"
         <<  "\tT - Traverse the list and print each entry.\n"
         <<  "\tS - The current size of the list.\n"
         <<  "\t[H]elp    [Q]uit." << endl;
}

#include <string.h>
char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands
      for the list demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("aidrftshq",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}
```

```
char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include list data structure
#include   "list.h"
#include   "list.cpp"

int do_command(char c, List<char> &test_list)
/*
PRE:  The list has been created and the command c is a valid list
      operation.
POST: The command has been executed.
USES: All the functions that perform list operations.
*/

{
   char x;
   int position;
   switch (c) {
   case 'h': help();
     break;

   case 's':
      cout << "The size of the list is " << test_list.size() << "\n";
      break;

   case 'a':
      position = test_list.size();
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;

   case 'i':
      cout << "Enter a list position to insert at: " << flush;
      cin >> position;
      cout << "Enter new character to insert: " << flush;
      x = get_char();
      test_list.insert(position, x);
      break;

   case 'r':
      cout << "Enter a list position to replace at: " << flush;
      cin >> position;
      cout << "Enter new character to use: " << flush;
      x = get_char();
      test_list.replace(position, x);
      break;

   case 'd':
      cout << "Enter a list position to remove from: " << flush;
      cin >> position;
      test_list.remove(position, x);
      break;
   case 'f':
      cout << "Enter a list position to fetch: " << flush;
      cin >> position;
      if (test_list.retrieve(position, x) == success)
         cout << "Retrieved entry is: " << x << endl;
      else
         cout << "Failed to retrieve such an entry." << endl;
      break;

   case 't':
      test_list.traverse(write_ent);
      cout << endl;
      break;

   case 'q':
      cout << "Extended list demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/*
PRE:  None.
POST: A list demonstration has been performed.
USES: get_command, do_command, List methods
*/

{
   introduction();
   help();
   List<char> test_list;
   while (do_command(get_command(), test_list));
}
```

(c) *Discuss the advantages and disadvantages of this variation of doubly linked lists in comparison with the doubly linked lists of the text.*

Answer The main advantage of using additional pointers to the head and tail of the list is that it makes accessing nodes an extremely fast operation. When the list is small, this feature is of little consequence but, when the list is large, access speed becomes critical. On average, if we assume random accessing, head and tail pointers will reduce access time by 50%. (Consider that the current pointer is now dividing the list length.) If accesses flipflop between the head and the tail (which, in practice, are by far the most frequent points of operation), the saving in access time will be dramatic.

The main disadvantage is an increased use of data stack memory space (in order to store the two extra fields). Again, this factor becomes relevant only when the list becomes large. There is some increase in complexity in the programming.

Overall the advantages far outweigh the disadvantages.

P6. **(a)** *Write a program that will do addition, subtraction, multiplication, and division for arbitrarily large integers. Each integer should be represented as a list of its digits. Since the integers are allowed to be as large as you like, linked lists will be needed to prevent the possibility of overflow. For some operations, it is useful to move backwards through the list; hence, doubly linked lists are appropriate. Multiplication and division can be done simply as repeated addition and subtraction.*

(b) *Rewrite* multiply *so that it is not based on repeated addition but on standard multiplication where the first multiplicand is multiplied with each digit of the second multiplicand and then added.*

(c) *Rewrite the divide operation so that it is not based on repeated subtraction but on long division. It may be necessary to write an additional function that determines if the dividend is larger than the divisor in absolute value.*

Answer This directory contains the following impelmentation files:
MAIN.CPP a driver
BIGINT.H BIGINT.CPP a Big integer class
README - this one.

The multiplication and division functions for part (a) are called `simple_mult` and `simple_div`, repspectively, and they are called with the commands '`m`' and '`d`' in the demonstration program. The multiply and divide functions implemented for the demonstration program commands '`*`' and '`/`' are the more efficient methods for parts (b) and (c).

Demonstratin program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../doubly/list.h"
#include "../doubly/list.cpp"
#include "bigint.h"
#include "bigint.cpp"
typedef Int Stack_entry;
#include "../../2/stack/stack.h"
#include "../../2/stack/stack.cpp"

void introduction()
/*
PRE:  None.
POST: An introduction to the program BigInt Calculator is printed.
*/

{
 cout << "BigInt Calculator Program." << endl
  << "This program simulates a big integer calculator that works on a\n"
  << "stack and a list of operations. The model is of a reverse Polish\n"
  << "calculator where operands are entered before the operators. An\n"
  << "example to add two integers and print the answer is ?P?Q+= .\n"
  << "P and Q are the operands to be entered, + is add, and = is\n"
  << "print result. The result is put onto the calculator's stack.\n\n";
}

void instructions()
/*
PRE:  None.
POST: Prints out the instructions and the operations allowed on the
      calculator.
*/

{
  cout << "\nEnter a string of instructions in reverse Polish form.\n"
       << "The allowable instructions are:\n\n"
       << "?:Read       +:Add            =:Print      -:Subtract\n"
       << "*:Multiply   q:Quit           /:Divide        h:Help\n"
       << "[M]ultiply (slow way)         [D]ivide (slow way)\n\n";
}

char get_command()
/*
PRE:  None.
POST: A legal command is read from the user and returned.
*/
```

```
{
   char command, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(command);
      } while (command == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == 'h' || command == '*' ||
          command == '/' || command == 'q' || command == 'p' ||
          command == 'm' || command == 'd' || command == 'h') {
         return (command);
      }
      cout << "Please enter a valid command:" << endl;
      instructions();
   }
}

bool do_command(char command, Stack &stored_ints)
/*
Pre:  The first parameter specifies a valid
      calculator command.
Post: The command specified by the first parameter
      has been applied to the Stack of Int
      objects given by the second parameter.
      A result of true is returned unless
      command == 'q'.
Uses: The classes Stack and Int.
*/

{
   Int p, q, r;
   switch (command) {
   case '?':
      p.read();
      if (stored_ints.push(p) == overflow)
         cout << "Warning: Stack full, lost integer" << endl;
      break;

   case '=':
      if (stored_ints.empty())
         cout << "Stack empty" << endl;
      else {
         stored_ints.top(p);
         p.print();
         cout << endl;
      }
      break;
```

TOC
Index
Help

```
case '+':
   if (stored_ints.empty())
      cout << "Stack empty" << endl;
   else {
      stored_ints.top(p);
      stored_ints.pop();
      if (stored_ints.empty()) {
         cout << "Stack has just one integer" << endl;
         stored_ints.push(p);
      }

      else {
         stored_ints.top(q);
         stored_ints.pop();
         r.equals_sum(q, p);
         if (stored_ints.push(r) == overflow)
            cout << "Warning: Stack full, lost integint" << endl;
      }
   }
   break;

case '/':
case 'd':
   if (stored_ints.empty()) cout << "Stack empty" << endl;
   else {
      stored_ints.top(p);
      stored_ints.pop();
      if (stored_ints.empty()) {
         cout << "Stack has just one integer." << endl;
         stored_ints.push(p);
      }
      else {
         stored_ints.top(q);
         stored_ints.pop();
         if (command == '/')
           if (r.equals_quotient(q, p) != success) {
              cerr << "Division by 0 fails; no action done." << endl;
              stored_ints.push(q);
              stored_ints.push(p);
           }
           else {
              if (stored_ints.push(r) == overflow)
               cout << "Warning: Stack full, lost integer. " << endl;
           }
         else
           if (r.simple_div(q, p) != success) {
              cerr << "Division by 0 fails; no action done." << endl;
              stored_ints.push(q);
              stored_ints.push(p);
           }
           else if (stored_ints.push(r) == overflow)
              cout << "Warning: Stack full, lost integer. " << endl;
      }
   }
   break;
   case '-':
      if (stored_ints.empty()) cout << "Stack empty" << endl;
      else {
         stored_ints.top(p);
         stored_ints.pop();
         if (stored_ints.empty()) {
            cout << "Stack has just one integer." << endl;
            stored_ints.push(p);
         }
         else {
            stored_ints.top(q);
            stored_ints.pop();
            r.equals_difference(q, p);
            if (stored_ints.push(r) == overflow)
               cout << "Warning: Stack full, lost integer." << endl;
         }
      }
      break;

   case '*':
   case 'm':
      if (stored_ints.empty()) cout << "Stack empty" << endl;
      else {
         stored_ints.top(p);
         stored_ints.pop();
         if (stored_ints.empty()) {
            cout << "Stack has just one integer." << endl;
            stored_ints.push(p);
         }
         else {
            stored_ints.top(q);
            stored_ints.pop();
            if (command == '*') r.equals_product(q, p);
            else r.simple_mult(q, p);
            if (stored_ints.push(r) == overflow)
               cout << "Warning: Stack full, lost integer." << endl;
         }
      }
      break;

   case 'h':
      instructions();
      break;

   case 'q':
      cout << "Calculation finished." << endl;
      return false;
   }
   return true;
}

int main()
/*
The program has executed simple Int arithmetic
commands entered by the user.
Uses: The classes Stack and Int and the functions
      introduction, instructions, do_command, and
      get_command.
*/
```

```
{
   List<int> temp, temp1; temp1 = temp;
   Stack stored_ints;
   introduction();
   instructions();
   while (do_command(get_command(), stored_ints));
}
```

Class definition:

```
class Int: private List<int>{  //  Use private inheritance.
public:
   void read();
   void print() const;
   void equals_sum(const Int &p, const Int &q);
   void equals_difference(const Int &p, const Int &q);
   void equals_product(const Int &p, const Int &q);
   void simple_mult(const Int &p, const Int &q);
   Error_code equals_quotient(const Int &p, const Int &q);
   Error_code simple_div(const Int &p, const Int &q);
private:
   void clean_up();
   bool negative;
   bool is_less_than(const Int &p);
};
```

Class implementation:

```
void Int::clean_up()
/*
Post: The Int is placed in standard form as a list of
      decimal digits
*/

{
   int carry = 0;
   for (int i = 0; i < size(); i++) {
      int digit;
      retrieve(i, digit);
      digit += carry;
      carry = 0;
      if (digit < -9 || digit > 9) {
         carry = digit / 10;
         digit = digit - 10 * carry;
      }
      replace(i, digit);
   }

   while (carry != 0) {
      insert(size(), carry % 10);
      carry = carry / 10;
   }

   int lead = 0;
   while (!empty() && lead == 0) {
      retrieve(size() - 1, lead);
      if (lead == 0) remove(size() - 1, lead);
   }
```

TOC
Index
Help

```
   if (lead < 0) negative = true;
   else          negative = false;

   carry = 0;
   if (negative) {
      for (int i= 0; i < size(); i++) {
         int digit;
         retrieve(i, digit);
         digit += carry;
         carry = 0;
         if (digit > 0) {
            digit -= 10;
            carry = 1;
         }
         replace(i, digit);
      }
   }

   else {
      for (int i= 0; i < size(); i++) {
         int digit;
         retrieve(i, digit);
         digit += carry;
         carry = 0;
         if (digit < 0) {
            digit += 10;
            carry = -1;
         }
         replace(i, digit);
      }
   }

   do {
      retrieve(size() - 1, lead);
      if (lead == 0) remove(size() - 1, lead);
   } while (lead == 0 && !empty());
}

bool Int::is_less_than(const Int &p)
/*
Post: Returns true if and only if the Int is less than the Int p
*/

{
   Int q;
   q.equals_difference(*this, p);
   return q.negative;
}

void Int::print() const
/*
Post: The Int is printed to cout.
*/
```

```
{
   if (empty()) {
      cout << "0";
      return;
   }
   int digit;
   if (negative) {
      cout << "-";
      for (int i = size() - 1; i >= 0; i--) {
         retrieve(i, digit);
         cout << -digit;
      }
   }

   else for (int i = size() - 1; i >= 0; i--) {
         retrieve(i, digit);
         cout << digit;
   }
}

void Int::read()
/*
Post: The Int is read from cin.
*/

{
   clear();
   char c;
   do {
     c = cin.get();
   } while (c != '-' && !('0' <= c && c <= '9'));
   if (c == '-') negative = true;
   else negative = false;

   if (negative) {
     do {
        c = cin.get();
        insert (0, '0' - c);
     } while ('0' <= c && c <= '9');
   }

   else {
     insert (0, c - '0');
     do {
        c = cin.get();
        insert (0, c - '0');
     } while ('0' <= c && c <= '9');
   }

   int digit;
   remove(0, digit);
   clean_up();
}

void Int::equals_sum(const Int &p, const Int &q)
/*
Post: The Int object
      is reset as the sum of the two parameters.
*/
```

```
{
   clear();
   int i = 0;
   while (i < p.size() && i < q.size()) {
      int digit1, digit2;
      p.retrieve(i, digit1);
      q.retrieve(i, digit2);
      insert(i, digit1 + digit2);
      i++;
   }

   while (i < p.size()) {
      int digit;
      p.retrieve(i, digit);
      insert(i, digit);
      i++;
   }

   while (i < q.size()) {
      int digit;
      q.retrieve(i, digit);
      insert(i, digit);
      i++;
   }
   clean_up();
}

void Int::equals_difference(const Int &p, const Int &q)
{
   clear();
   int i = 0;
   while (i < p.size() && i < q.size()) {
      int digit1, digit2;
      p.retrieve(i, digit1);
      q.retrieve(i, digit2);
      insert(i, digit1 - digit2);
      i++;
   }

   while (i < p.size()) {
      int digit;
      p.retrieve(i, digit);
      insert(i, digit);
      i++;
   }

   while (i < q.size()) {
      int digit;
      q.retrieve(i, digit);
      insert(i, -digit);
      i++;
   }
   clean_up();
}
```

```
void Int::equals_product(const Int &p, const Int &q)
{
   clear();
   int digit, digit1 = 0, digit2 = 0;
   for (int i = 0; i < p.size(); i++) {
      p.retrieve(i, digit1);
      for (int j = 0; j < q.size(); j++) {
         q.retrieve(j, digit2);
         if (size() > i + j) {
            retrieve(i + j, digit);
            replace(i + j, digit + digit1 * digit2);
         }
         else {
            insert (i + j, digit1 * digit2);
         }
      }
   }
   clean_up();
}

Error_code Int::equals_quotient(const Int &p, const Int &q)
{
   clear();
   if (q.empty()) return (fail);     //  divide by 0
   if (p.size() < q.size()) return (success);  // answer is just 0

   bool sign = false;
   Int zero, p_pos, q_pos, estimate;
   if (p.negative) {
      sign = !sign;
      p_pos.equals_difference(zero, p);
   }
   else p_pos = p;

   if (q.negative) {
      sign = !sign;
      q_pos.equals_difference(zero, q);
   }
   else q_pos = q;

   if (p_pos.is_less_than(q_pos)) return (success); // answer is 0

   int p_term, q_term;
   p_pos.retrieve(p_pos.size() - 1, p_term);
   q_pos.retrieve(q_pos.size() - 1, q_term);

   if (p_term >= q_term) {
     int i;
     for (i = 0; i < p_pos.size() - q_pos.size(); i++)
        estimate.insert(i, 0);
     estimate.insert(i, p_term / q_term);
   }

   else {
     int i;
     for (i = 0; i < p_pos.size() - q_pos.size() - 1; i++)
        estimate.insert(i, 0);
     estimate.insert(i, (10 * p_term) / q_term);
   }
```

TOC
Index
Help

```
   Int partial_prod;
   partial_prod.equals_product(estimate, q_pos);

   Int remainder;
   remainder.equals_difference(p_pos, partial_prod);

   Int add_on;
   add_on.equals_quotient(remainder, q_pos);
   if (!sign) equals_sum(estimate, add_on);
   else {
      Int answer;
      answer.equals_sum(estimate, add_on);
      equals_difference(zero, answer);
   }
   return success;
}

Error_code Int::simple_div(const Int &p, const Int &q)
{
   clear();
   if (q.empty()) return (fail);    //  divide by 0

   bool sign = false;
   Int p_pos, q_pos, answer, zero;
   if (p.negative) {
      sign = !sign;
      p_pos.equals_difference(zero, p);
   }
   else p_pos = p;

   if (q.negative) {
      sign = !sign;
      q_pos.equals_difference(zero, q);
   }
   else q_pos = q;

   Int one;
   one.insert(0, 1);
   one.negative = false;

   while (!p_pos.is_less_than(q_pos)) {
      Int temp1, temp2;
      temp1.equals_sum(answer, one);
      answer = temp1;
      temp2.equals_difference(p_pos, q_pos);
      p_pos = temp2;
   }

   if (!sign) equals_sum(zero, answer);
   else equals_difference(zero, answer);
   return success;
}

void Int::simple_mult(const Int &p, const Int &q)
{
   clear();
   Int one;
   one.insert(0, 1);
   one.negative = false;
```

```
    Int zero, answer, counter;
    bool sign = false;
    Int q_pos;
    if (q.negative) {
       sign = !sign;
       q_pos.equals_difference(zero, q);
    }
    else q_pos = q;

    while (counter.is_less_than(q_pos)) {
       Int temp1, temp2;
       temp1.equals_sum(counter, one);
       counter = temp1;
       temp2.equals_sum(p, answer);
       answer = temp2;
    }

    if (!sign) equals_sum(zero, answer);
    else equals_difference(zero, answer);
}
```

6.3 STRINGS

Exercises 6.3

E1. *Write implementations for the remaining* String *methods.*

(a) *The constructor* String()

Answer

```
String :: String()
/* Post: A new empty String object is created. */
{
   length = 0;
   entries = new char[length + 1];
   strcpy(entries, "");
}
```

(b) *The destructor* ~String()

Answer

```
String :: ~String()
/* Post: The dynamically aquired storage of a String is deleted. */
{
   delete [ ]entries;
}
```

(c) *The copy constructor* String(**const** String &copy)

Answer

```
String :: String(const String &copy)
/* Post: A new String object is created to match copy. */
{
   length = strlen(copy.entries);
   entries = new char[length + 1];
   strcpy(entries, copy.entries);
}
```

(d) *The overloaded assignment operator.*

Answer
```
void String :: operator = (const String &copy)
/* Post: A String object is assigned the value of the String copy. */
{
   if (strcmp(entries, copy.entries) != 0) {
      delete [ ]entries;
      length = strlen(copy.entries);
      entries = new char[length + 1];
      strcpy(entries, copy.entries);
   }
}
```

E2. *Write implementations for the following* String *comparison operators:*

$$> \quad < \quad >= \quad <= \quad !=$$

Answer The overloaded comparison operators (including a test for equality of strings) are implemented as follows.

```
bool operator == (const String &first, const String &second)
/* Post: Return true if the String first agrees with String second. Else: Return false. */
{
   return strcmp(first.c_str( ), second.c_str( )) == 0;
}

bool operator > (const String &first, const String &second)
/* Post: Return true if the String first is lexicographically later than String second. Else: Return
         false. */
{
   return strcmp(first.c_str( ), second.c_str( )) > 0;
}

bool operator < (const String &first, const String &second)
/* Post: Return true if the String first is lexicographically earlier than String second. Else: Return
         false. */
{
   return strcmp(first.c_str( ), second.c_str( )) < 0;
}

bool operator >= (const String &first, const String &second)
/* Post: Return true if the String first is not lexicographically earlier than String second. Else:
         Return false. */
{
   return strcmp(first.c_str( ), second.c_str( )) >= 0;
}

bool operator <= (const String &first, const String &second)
/* Post: Return true if the String first is not lexicographically later than String second. Else:
         Return false. */
{
   return strcmp(first.c_str( ), second.c_str( )) <= 0;
}

bool operator != (const String &first, const String &second)
/* Post: Return true if the String first is not equal to String second. Else: Return false. */
{
   return strcmp(first.c_str( ), second.c_str( )) != 0;
}
```

E3. *Write implementations for the remaining* String *processing functions.*

(a) **void** strcpy(String &copy, **const** String &original);

Answer

```
void strcpy(String &s, String &t)
/* Post: Copies the value of String t to String s. */
{
   s = t;
}
```

(b) **void** strncpy(String &copy, **const** String &original, **int** n);

Answer

```
void strncpy(String &s, const String &t, unsigned len)
/* Post: Copies the first len characters of String t to make String s. */
{
   const char *temp = t.c_str( );
   char *copy = new char[len + 1];
   strncpy(copy, temp, len);
   copy[len] = 0;
   s = copy;
   delete [ ]copy;
}
```

(c) **int** strstr(**const** String &text, **const** String &target);

Answer

```
int strstr(String s, String t)
/* Post: Returns the index of the first copy of String t as a substring of String s. Else: Return
         −1. */
{
   int answer;
   const char * content_s = s.c_str( );
   char *p = strstr((char *) content_s, t.c_str( ));
   if (p == NULL) answer = −1;
   else answer = p − content_s;
   return answer;
}
```

E4. *A **palindrome** is a string that reads the same forward as backward; that is, a string in which the first character equals the last, the second equals the next to last, and so on. Examples of palindromes include* ′radar′ *and*

palindromes

′ABLE WAS I ERE I SAW ELBA′.

Write a C++ function to test whether a String *object passed as a reference parameter represents a palindrome.*

Answer

```
bool palindrome(String &to_test)
/* Post: Returns true if to_test represents a palindrome. */
{
   const char *content = to_test.c_str( );
   int l = strlen(content);
   for (int i = 0; i < l/2; i++)
      if (content[i] != content[l − 1 − i]) return false;
   return true;
}
```

Programming Projects 6.3

P1. *Prepare a file containing implementations of the* String *methods and the functions for* String *processing. This file should be suitable for inclusion in any application program that uses strings.*

Answer Class definition:

```
class String {
public:                                  //  methods of the string ADT
   String();
   ~String();
   String (const String &copy);          //  copy constructor
   String (const char * copy);           //  conversion from C-string
   String (List<char> &copy);            //  conversion from List

   void operator =(const String &copy);
   const char *c_str() const;            //  conversion to C-style string

protected:

   char *entries;
   int length;
};

bool operator ==(const String &first, const String &second);
bool operator  >(const String &first, const String &second);
bool operator  <(const String &first, const String &second);
bool operator >=(const String &first, const String &second);
bool operator <=(const String &first, const String &second);
bool operator !=(const String &first, const String &second);
```

Class implementation:

```
String::String (const char *in_string)
/*
Pre:  The pointer in_string references a C-string.
Post: The String is initialized by the C-string in_string.
*/

{
   length = strlen(in_string);
   entries = new char[length + 1];
   strcpy(entries, in_string);
}

String::String (List<char> &in_list)
/*
Post: The String is initialized by the character List in_list.
*/

{
   length = in_list.size();
   entries = new char[length + 1];
   for (int i = 0; i < length; i++) in_list.retrieve(i,entries[i]);
   entries[length] = '\0';
}

const char*String::c_str() const
/*
Post: A pointer to a legal C-string object matching
      the String is returned.
*/
```

TOC
Index
Help

```
{
   return (const char *) entries;
}

String::String(const String &copy)
/*
Post: A new String object is created to match copy.
*/

{
   length = strlen(copy.entries);
   entries = new char[length + 1];
   strcpy(entries, copy.entries);
}

String::String()
/*
Post: A new empty String object is created.
*/

{
   length = 0;
   entries = new char[length + 1];
   strcpy(entries, "");
}

String::~String()
/*
Post: The dynamically aquired storage of a String is deleted.
*/

{
   delete []entries;
}

void String:: operator =(const String &copy)
/*
Post: A String object is assigned the value of the String copy.
*/

{
   if (strcmp(entries, copy.entries) != 0) {
      delete []entries;
      length = strlen(copy.entries);
      entries = new char[length + 1];
      strcpy(entries, copy.entries);
   }
}

bool operator ==(const String &first, const String &second)
/*
Post: Return true if the String first agrees with
      String second.  Else: Return false.
*/

{
   return strcmp(first.c_str(), second.c_str()) == 0;
}

bool operator >(const String &first, const String &second)
/*
Post: Return true if the String first is lexicographically
      later than String second.  Else: Return false.
*/
```

```
{
   return strcmp(first.c_str(), second.c_str()) > 0;
}

bool operator <(const String &first, const String &second)
/*
Post: Return true if the String first is lexicographically
      earlier than String second.  Else: Return false.
*/

{
   return strcmp(first.c_str(), second.c_str()) < 0;
}

bool operator >=(const String &first, const String &second)
/*
Post: Return true if the String first is not lexicographically
      earlier than String second.  Else: Return false.
*/

{
   return strcmp(first.c_str(), second.c_str()) >= 0;
}

bool operator <=(const String &first, const String &second)
/*
Post: Return true if the String first is not lexicographically
      later than String second.  Else: Return false.
*/

{
   return strcmp(first.c_str(), second.c_str()) <= 0;
}

bool operator !=(const String &first, const String &second)
/*
Post: Return true if the String first is not
      equal to String second.  Else: Return false.
*/

{
   return strcmp(first.c_str(), second.c_str()) != 0;
}

int strstr(String s, String t)
/*
Post: Returns the index of the first copy of String
      as a substring of String .  Else: Return -1.
*/

{
   int answer;
   const char * content_s = s.c_str();
   char *p = strstr((char *) content_s, t.c_str());
   if (p == NULL) answer = -1;
   else answer = p - content_s;
   return answer;
}
```

```
String read_in(istream &input, int &terminator)
/*
Post: Return a String read (as characters terminated
      by a newline or an end of file character)
      from an istream parameter.
      The terminating character is recorded
      with the output parameter terminator.
*/

{
   List<char> temp;
   int size = 0;

   while ((terminator = input.peek()) != EOF &&
          (terminator = input.get()) != '\n')
      temp.insert(size++, terminator);
   String answer(temp);
   return answer;
}

String read_in(istream &input)
/*
Post: Return a String read (as characters terminated
      by a newline or an end-of-file character)
      from an istream parameter.
*/

{
   List<char> temp;
   int size = 0;

   char c;
   while ((c = input.peek()) != EOF && (c = input.get()) != '\n')
      temp.insert(size++, c);
   String answer(temp);
   return answer;
}

void write(String &s)
/*
Post: The String parameter s is written to cout.
*/

{
   cout << s.c_str() << endl;
}

void strcpy(String &s, String &t)
/*
Post: Copies the value of String  to String s.
*/

{
   s = t;
}

void strncpy(String &s, const String &t, unsigned len)
/*
Post: Copies the first len characters of String  to make String s.
*/
```

```
{
   const char *temp = t.c_str();
   char *copy = new char[len + 1];
   strncpy(copy, temp, len);
   copy[len] = 0;
   s = copy;
   delete []copy;
}

void strcat(String &add_to, const String &add_on)
/*
Post: The function concatenates String add_on
      onto the end of String add_to.
*/

{
   const char *cfirst = add_to.c_str();
   const char *csecond = add_on.c_str();
   char *copy = new char[strlen(cfirst) + strlen(csecond) + 1];
   strcpy(copy, cfirst);
   strcat(copy, csecond);
   add_to = copy;
   delete []copy;
}
```

Simple test program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../doubly/list.h"
#include "../doubly/list.cpp"
#include <string.h>

#include "string.h"
#include "string.cpp"

void copy_string(String &s, String t)
{
   s = t;
}

main()
{
   String s, t("second string "), u = t;
   char * junk2;

   cout << s.c_str() << " " << t.c_str() << " " << u.c_str() << endl;

   write(s);
   write(t);
   write(u);
   cout << "\n";

   s = u;
   copy_string(t, s);

   write(s);
   write(t);
   write(u);
   cout << "\n";
   cout << strlen(s.c_str()) << "\n";
   const char *junk = s.c_str();
   cout << junk << "\n";
   junk2 = "first string";
   s = junk2;
   write(s);
   write(t);
   write(u);
   cout << "\n";
}
```

P2. *Different authors tend to employ different vocabularies, sentences of different lengths, and paragraphs of different lengths. This project is intended to analyze a text file for some of these properties.*

(a) *Write a program that reads a text file and counts the number of words of each length that occurs, as well as the total number of words. The program should then print the mean (average) length of a word and the percentage of words of each length that occurs. For this project, assume that a word consists entirely of (uppercase and lowercase) letters and is terminated by the first non-letter that appears.*

text analysis

Answer The program uses the `String` class developed in Project P1 together with further functions.
A demonstration program for the projects:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../doubly/list.h"
#include "../doubly/list.cpp"
#include "../strings/string.h"
#include "../strings/string.cpp"
#include "auxil.cpp"
#include "projA.cpp"
#include "projB.cpp"
#include "projC.cpp"

int main(int argc, char *argv[]) // count, values of cmnd-line arguments
/*
Post: Reads an input file of text, and computes simple statistics
      about the text.  The input file can be specified by the user
      or can be given as a command line argument.
*/

{
   cout << "Running text statistics program. " << endl << endl;

   char file_name[1000];
   if (argc >= 2)
      strcpy(file_name, argv[1]);
   else {
      cout << "Type in the input text file name: " << flush;
      cin >> file_name;
   }

   ifstream file_in(file_name);   //  Declare and open the input stream.
   if (file_in == 0) {
      cout << "Can't open input file " << file_name << endl;
      exit (1);
   }
   bool proceed = true;
   List<String> lines;
   int line_number;
   int terminal_char;

   while (proceed) {
      String in_string = read_in(file_in, terminal_char);
      if (terminal_char == EOF) {
         proceed = false;
         if (strlen(in_string.c_str()) > 0) {
            tokenize(in_string);
            lines.insert(line_number++, in_string);
         }
      }

      else {
         tokenize(in_string);
         lines.insert(line_number++, in_string);
      }
    }

    char command;
    do {
      command = ' ';
      cout << "Choose one of the following options: " << endl;
      cout << "[Q]uit [D]ump tokens project[A] project[B] project[C]: "
           << flush;

      while (!('a' <= command && command <='z' ||
                   'A' <= command && command <= 'Z')) {
         cout >> command;
      }

      switch (command) {
         case 'd': case 'D': {
               String s;
               for (int i = 0; i < line_number; i++) {
                  lines.retrieve(i, s);
                  cout << s.c_str() << endl;
               }
            }
            break;

         case 'a': case 'A':
            a_count(lines);
            break;

         case 'b': case 'B':
            b_count(lines);
            break;

         case 'c': case 'C':
            c_count(lines);
            break;
      }
    } while (command != 'q' && command != 'Q');
 }
```

Auxiliary String functions:

```
void tokenize(String &s)
/*
Pre:  The parameter s represents a line of text
Post: Spaces have been inserted around all non-alphabetic characters
      of the String s
*/

{
   const char *temp = s.c_str();
   char *new_temp = new char[3 * strlen(temp) + 1];
   char *q = (char *) temp;
   char *p = new_temp;

   while (*q != '\0') {
      if (('a' <= *q && *q <= 'z') ||
          ('A' <= *q && *q <= 'Z') )  *(p++) = *(q++);
      else {
         *(p++) = ' ';
         *(p++) = *(q++);
         *(p++) = ' ';
      }
   }

   *p = '\0';
   s = (String) new_temp;
   delete []new_temp;
}

Error_code get_word(const String &s, int n, String &t)
/*
Pre: The input parameter is a tokenized String.
Post:  The nth Token of s is copied to the output parameter t
*/

{
   if (n < 0) return fail;
   const char *temp = s.c_str();
   char *q = (char *) temp;
   char *p;

   while (n-- >= 0) {
      while (*q == ' ') q++;
      if (*q == '\0') return fail;
      p = q;
      while (*q != '\0' && *q != ' ') q++;
   }
   char old = *q;
   *q = '\0';
   t = p;
   *q = old;
   return success;
}

bool is_word(const String &s)
/*
Pre:  The input parameter s is a single token string
Post: True is returned if and only if s is a word
      (that is a string of alphabetic characters).
*/
```

```
{
   const char *temp = s.c_str();
   char *q = (char *) temp;
   if (('a' <= *q && *q <= 'z') || ('A' <= *q && *q <= 'Z'))
       return true;
   return false;
}

bool is_period(const String &s)
/*
Pre: The input parameter s is a single token string
Post: True is returned if and only if s terminates a sentence
*/

{
   const char *temp = s.c_str();
   char *q = (char *) temp;
   if (*q == '.' || *q == '?' || *q == '!')
       return true;
   return false;
}
```

Additional function:

```
void a_count(const List<String> &lines)
/*
Pre:  The List lines is a list of text lines.
Post: The statistics on lines for project P2(a)
      have been printed.
*/

{
   int word_count = 0;
   int length_count = 0;
   int lengths[22];
   int i;

   for (i = 0; i < 22; i++) lengths[i] = 0;
   for (i = 0; i < lines.size(); i++) {
      String s, word;
      lines.retrieve(i, s);

      int j = 0;
      while (get_word(s, j++, word) == success) {
         if (is_word(word)) {
            word_count++;
            int len = strlen(word.c_str());
            length_count += len;
            if (len <= 20) lengths[len] ++;
            else lengths[21]++;
         }
      }
   }

   cout << "Total number of words: " << word_count << endl;
   cout << "Average word length: "
        << ((double) length_count) / ((double) word_count) << endl;
   for (i = 0; i < 22; i++) {
       if (lengths[i] > 0) {
          double percent = 100.0 * (((double) lengths[i]) /
                                    ((double) word_count));
          cout << "Words of length ";
          if (i < 21) cout << i;
          else cout << "greater than 20";
          cout << " form " << percent <<"% of the file." << endl;
       }
   }
}
```

(b) *Modify the program so that it also counts sentences and prints the total number of sentences and the mean number of words per sentence. Assume that a sentence terminates as soon as one of the characters period (.), question mark (?), or exclamation point (!) appears.*

Answer

```
void b_count(const List<String> &lines)
/*
Pre:  The List lines is a list of text lines.
Post: The statistics on lines for project P2(b)
      have been printed.
*/

{
   a_count(lines);

   int sentence_count = 0;
   int word_count = 0;
   for (int i = 0; i < lines.size(); i++) {
      String s, word;
      lines.retrieve(i, s);
      int j = 0;

      while (get_word(s, j++, word) == success) {
         if (is_word(word))
            word_count++;
         else if (is_period(word))
            sentence_count++;
      }
   }

   cout << "Total number of sentences: " << sentence_count << endl;
   cout << "Average sentence length: "
        << ((double) word_count) / ((double) sentence_count) << endl;
}
```

(c) *Modify the program so that it counts paragraphs and prints the total number of paragraphs and the mean number of words per paragraph. Assume that a paragraph terminates when a blank line or a line beginning with a blank character appears.*

Answer

```
void c_count(const List<String> &lines)
/*
Pre:  The List lines is a list of text lines.
Post: The statistics on lines for project P2(c)
      have been printed.
*/

{
   a_count(lines);
   b_count(lines);
```

```
      int para_count = 0;
      int word_count = 0;
      String word;
      for (int i = 0; i < lines.size(); i++) {
         String s;
         lines.retrieve(i, s);

         const char *content = s.c_str();
         if (strlen(content) == 0)
            para_count++;

         else if (strlen(content) == 1 && content[0] == ' ')
            para_count++;

         else if (content[0] == ' ' && content[1] == ' ')
            para_count++;

         else if (i == lines.size() - 1)
            para_count++;

         int j = 0;
         while (get_word(s, j++, word) == success) {
            if (is_word(word))
               word_count++;
         }
      }

      cout << "Total number of paragraphs: " << para_count << endl;
      cout << "Average paragraph length: "
           << ((double) word_count) / ((double) para_count) << " words"
           << endl;
   }
```

6.4 APPLICATION: A TEXT EDITOR

Programming Projects 6.4

P1. *Supply the following functions; test and exercise the text editor.*

(a) next_line
(b) previous_line
(c) goto_line
(d) substitute_line
(e) write_file

Answer Driver program:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../doubly/list.h"
#include "../doubly/list.cpp"
#include "../strings/string.h"
#include "../strings/string.cpp"
#include "editor.h"
#include "editor.cpp"
```

```
int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Names of input and output files are given as command-line arguments.
Post: Reads an input file that contains lines (character strings),
      performs simple editing operations on the lines, and
      writes the edited version to the output file.
Uses: methods of class Editor
*/

{
   if (argc != 3) {
      cout << "Usage:\n\t edit  inputfile  outputfile" << endl;
      exit (1);
   }

   ifstream file_in(argv[1]);   //  Declare and open the input stream.
   if (file_in == 0) {
      cout << "Can't open input file " << argv[1] << endl;
      exit (1);
   }

   ofstream file_out(argv[2]);   //  Declare and open the output stream.
   if (file_out == 0) {
      cout << "Can't open output file " << argv[2] << endl;
      exit (1);
   }

   Editor buffer(&file_in, &file_out);
   while (buffer.get_command())
      buffer.run_command();
}
```

Class definition:

```
class Editor:public List<String> {
public:
   Editor(ifstream *file_in, ofstream *file_out);
   bool get_command();
   void run_command();
private:
   ifstream *infile;
   ofstream *outfile;
   char user_command;

//  auxiliary functions
   Error_code next_line();
   Error_code previous_line();
   Error_code goto_line();
   Error_code insert_line();
   Error_code substitute_line();
   Error_code change_line();
   void read_file();
   void write_file();
   void find_string();
};
```

Class implementation:

```
Error_code Editor::next_line()
{
   if (current->next == NULL) return fail;
   else {
      current = current->next;
      current_position++;
      return success;
   }
}

Error_code Editor::previous_line()
{
   if (current->back == NULL) return fail;
   else {
      current = current->back;
      current_position--;
      return success;
   }
}

Error_code Editor::goto_line()
{
   int line_number;
   cout << " Go to what line number? " << flush;
   cin >> line_number;
   if (line_number > size() - 1) return fail;
   else {
      set_position(line_number);
      return success;
   }
}

Error_code Editor::insert_line()
/*
Post: A string entered by the user is inserted as a
      user-selected line number.
Uses: String and Editor methods and functions.
*/

{
   int line_number;
   cout << " Insert what line number? " << flush;
   cin  >> line_number;
   while (cin.get() != '\n');
   cout << " What is the new line to insert? " << flush;
   String to_insert = read_in(cin);
   return insert(line_number, to_insert);
}

Error_code Editor::substitute_line()
{
   int line_number;
   cout << " Substitute what line number? " << flush;
   cin >> line_number;
   while (cin.get() != '\n');
   cout << " What is the replacement line? " << flush;
   String to_insert = read_in(cin);
   return replace(line_number, to_insert);
}
```

```
Error_code Editor::change_line()
/*
Pre:  The Editor is not empty.
Post: If a user-specified string appears in the
      current line, it is replaced by a new user-selected string.
Else: an Error_code is returned.
Uses: String and Editor methods and functions.
*/

{
   Error_code result = success;
   cout << " What text segment do you want to replace? " << flush;
   String old_text = read_in(cin);
   cout << " What new text segment do you want to add in? " << flush;
   String new_text = read_in(cin);

   int the_index = strstr(current->entry, old_text);
   if (the_index == -1) result = fail;
   else {
      String new_line;
      strncpy(new_line, current->entry, the_index);
      strcat(new_line, new_text);
      const char *old_line = (current->entry).c_str();
      strcat(new_line, (String)
         (old_line + the_index + strlen(old_text.c_str())));
      current->entry = new_line;
   }
   return result;
}

void Editor::read_file()
/*
Pre:  Either the Editor is empty or the user authorizes
      the command.
Post: The contents of *infile are read to the Editor.
      Any prior contents of the Editor are overwritten.
Uses: String and Editor methods and functions.
*/

{
   bool proceed = true;
   if (!empty()) {
      cout << "Buffer is not empty; the read will destroy it." << endl;
      cout << " OK to proceed? " << endl;
      if (proceed = user_says_yes()) clear();
   }

  int line_number = 0, terminal_char;
  while (proceed) {
      String in_string = read_in(*infile, terminal_char);
      if (terminal_char == EOF) {
         proceed = false;
         if (strlen(in_string.c_str()) > 0)
            insert(line_number , in_string);
      }
      else insert(line_number++, in_string);
   }
}
```

```
void Editor::write_file()
{
   while (previous_line() == success);
   do {
      *outfile << (current->entry).c_str() << endl;
   } while (next_line() == success);
}

void Editor::find_string()
/*
Pre:   The Editor is not empty.
Post: The current line is advanced until either it contains a copy
      of a user-selected string or it reaches the end of the Editor.
      If the selected string is found, the corresponding line is
      printed with the string highlighted.
Uses: String and Editor methods and functions.
*/

{
   int the_index;

   cout << "Enter string to search for:" << endl;
   String search_string = read_in(cin);
   while ((the_index = strstr(current->entry, search_string)) == -1)
      if (next_line() != success) break;
   if (the_index == -1) cout << "String was not found.";
   else {
      cout << (current->entry).c_str() << endl;
      for (int i = 0; i < the_index; i++)
         cout << " ";
      for (int j = 0; j < strlen(search_string.c_str()); j++)
         cout << "^";
   }
   cout << endl;
}

bool Editor::get_command()
/*
Post: Sets member user_command; returns true
      unless the user's command is q.
Uses: C library function tolower.
*/

{
   if (current != NULL)
      cout << current_position << " : "
           << current->entry.c_str() << "\n??" << flush;
   else
      cout << "File is empty. \n??" << flush;

   cin >> user_command;    //  ignores white space and gets command
   user_command = tolower(user_command);
   while (cin.get() != '\n')
      ;                       //  ignore user's enter key
   if (user_command == 'q')
      return false;
   else
      return true;
}
```

```
void Editor::run_command()
/*
Post: The command in user_command has been performed.
Uses: methods and auxiliary functions of the class Editor,
      the class String, and the String processing functions.
*/

{
   String temp_string;

   switch (user_command) {

   case 'b':
      if (empty())
         cout << " Warning: empty buffer " << endl;
      else
         while (previous_line() == success)
            ;
      break;

   case 'c':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else if (change_line() != success)
         cout << " Error: Substitution failed " << endl;
      break;

   case 'd':
      if (remove(current_position, temp_string) != success)
         cout << " Error: Deletion failed " << endl;
      break;

   case 'e':
      if (empty())
         cout << " Warning: empty buffer " << endl;
      else
         while (next_line() == success)
            ;
      break;

   case 'f':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else
         find_string();
      break;

   case 'g':
      if (goto_line() != success)
         cout << " Warning: No such line" << endl;
      break;

   case '?':
   case 'h':
      cout << "Valid commands are: b(egin) c(hange) d(el) e(nd)" << endl
           << "f(ind) g(o) h(elp) i(nsert) l(ength) n(ext) p(rior) "
           << endl
           << "q(uit) r(ead) s(ubstitue) v(iew) w(rite) " << endl;
      break;
   case 'i':
      if (insert_line() != success)
         cout << " Error: Insertion failed " << endl;
      break;

   case 'l':
      cout << "There are " << size() << " lines in the file." << endl;

      if (!empty())
         cout << "Current line length is "
              << strlen((current->entry).c_str()) << endl;
      break;

   case 'n':
      if (next_line() != success)
         cout << " Warning: at end of buffer" << endl;
      break;

   case 'p':
      if (previous_line() != success)
         cout << " Warning: at start of buffer" << endl;
      break;

   case 'r':
      read_file();
      break;

   case 's':
      if (substitute_line() != success)
         cout << " Error: Substitution failed " << endl;
      break;

   case 'v':
      traverse(write);
      break;

   case 'w':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else
         write_file();
      break;

   default :
      cout << "Press h or ? for help or enter a valid command: ";
   }
}

Editor::Editor(ifstream *file_in, ofstream *file_out)
/*
Post: Initialize the Editor members infile and outfile
with the parameters.
*/
{
   infile = file_in;
   outfile = file_out;
}
```

P2. *Add a feature to the text editor to put text into two columns, as follows. The user will select a range of line numbers, and the corresponding lines from the buffer will be placed into two queues, the first half of the lines in one, and the second half in the other. The lines will then be removed from the queues, one at a time from each, and combined with a predetermined number of blanks between them to form a line of the final result. (The white space between the columns is called the gutter.)*

Answer The main editor program is identical to that for Project P1.

The following modified editor class has an extra method called `gutter` to put text into two-column format.

```
class Editor:public List<String> {
public:
   Editor(ifstream *file_in, ofstream *file_out);
   bool get_command();
   void run_command();
private:
   ifstream *infile;
   ofstream *outfile;
   char user_command;

//  auxiliary functions
   Error_code next_line();
   Error_code previous_line();
   Error_code goto_line();
   Error_code insert_line();
   Error_code substitute_line();
   Error_code change_line();
   void read_file();
   void write_file();
   void find_string();
   Error_code gutter();
};
```

The following implementation incorporates the `gutter()` method as well as instructions to run it.

```
Error_code Editor::gutter()
/*
Post:  A user specified range of lines are combined
       into two columns with a user specified gutter.
*/

{
   int begin_line, end_line, mid_line, guttering;
   cout << "Start two column text at what line number? " << flush;
   cin >> begin_line;
   while (cin.get() != '\n');
   cout << "End two column text at what line number? " << flush;
   cin >> end_line;
   while (cin.get() != '\n');
   cout << "What is the minimal gutter width? " << flush;
   cin >> guttering;
   while (cin.get() != '\n');

   if (begin_line < 0 || end_line < begin_line || end_line >= size()
          || guttering < 0)
      return range_error;
   List<String> first_half, second_half;
   String s2, s;
   mid_line = (begin_line + end_line) / 2;
   int long_line = 0;
   int i;
   for (i = end_line; i >= begin_line; i--) {
      remove(i, s);
      if (i > mid_line) second_half.insert(0, s);
      else {
         if (strlen(s.c_str()) > long_line)
                  long_line = strlen(s.c_str());
         first_half.insert(0, s);
      }
   }
   if (first_half.size() > second_half.size()) {
      first_half.remove(first_half.size() - 1, s);
      insert(begin_line, s);
   }
   for (i = first_half.size() - 1; i >= 0; i --) {
      first_half.remove(i, s);
      second_half.remove(i, s2);
      int l = 1 + guttering + long_line - strlen(s.c_str());
      char *gutter_piece = new char[l];
      for (int j = 0; j < l - 1; j++) gutter_piece[j] = ' ';
      gutter_piece[l - 1] = '\0';
      String gut = gutter_piece;
      strcat(gut, s2);
      strcat(s, gut);
      insert(begin_line, s);
      delete []gutter_piece;
   }
   return success;
}

Error_code Editor::next_line()
{
   if (current->next == NULL) return fail;
   else {
      current = current->next;
      current_position++;
      return success;
   }
}

Error_code Editor::previous_line()
{
   if (current->back == NULL) return fail;
   else {
      current = current->back;
      current_position--;
      return success;
   }
}
```

```
Error_code Editor::goto_line()
{
   int line_number;
   cout << " Go to what line number? " << flush;
   cin >> line_number;
   if (line_number > size() - 1) return fail;
   else {
      set_position(line_number);
      return success;
   }
}

Error_code Editor::insert_line()
/*
Post: A string entered by the user is inserted as a
      user-selected line number.
Uses: String and Editor methods and functions.
*/

{
   int line_number;
   cout << " Insert what line number? " << flush;
   cin  >> line_number;
   while (cin.get() != '\n');
   cout << " What is the new line to insert? " << flush;
   String to_insert = read_in(cin);
   return insert(line_number, to_insert);
}

Error_code Editor::substitute_line()
{
   int line_number;
   cout << " Substitute what line number? " << flush;
   cin >> line_number;
   while (cin.get() != '\n');
   cout << " What is the replacement line? " << flush;
   String to_insert = read_in(cin);
   return replace(line_number, to_insert);
}

Error_code Editor::change_line()
/*
Pre:  The Editor is not empty.
Post: If a user-specified string appears in the
      current line, it is replaced by a new user-selected string.
Else: an Error_code is returned.
Uses: String and Editor methods and functions.
*/

{
   Error_code result = success;
   cout << " What text segment do you want to replace? " << flush;
   String old_text = read_in(cin);
   cout << " What new text segment do you want to add in? " << flush;
   String new_text = read_in(cin);
   int the_index = strstr(current->entry, old_text);
   if (the_index == -1) result = fail;
   else {
      String new_line;
      strncpy(new_line, current->entry, the_index);
      strcat(new_line, new_text);
      const char *old_line = (current->entry).c_str();
      strcat(new_line, (String)
         (old_line + the_index + strlen(old_text.c_str())));
      current->entry = new_line;
   }
   return result;
}

void Editor::read_file()
/*
Pre:  Either the Editor is empty or the user authorizes
      the command.
Post: The contents of *infile are read to the Editor.
      Any prior contents of the Editor are overwritten.
Uses: String and Editor methods and functions.
*/

{
   bool proceed = true;
   if (!empty()) {
      cout << "Buffer is not empty; the read will destroy it." << endl;
      cout << " OK to proceed? " << endl;
      if (proceed = user_says_yes()) clear();
   }

  int line_number = 0, terminal_char;
  while (proceed) {
      String in_string = read_in(*infile, terminal_char);
      if (terminal_char == EOF) {
         proceed = false;
         if (strlen(in_string.c_str()) > 0)
             insert(line_number , in_string);
      }
      else insert(line_number++, in_string);
   }
}

void Editor::write_file()
{
   while (previous_line() == success);
   do {
      *outfile << (current->entry).c_str() << endl;
   } while (next_line() == success);
}

void Editor::find_string()
/*
Pre:  The Editor is not empty.
Post: The current line is advanced until either it contains a copy
      of a user-selected string or it reaches the end of the Editor.
      If the selected string is found, the corresponding line is
      printed with the string highlighted.
Uses: String and Editor methods and functions.
*/
```

```
{
   int the_index;

   cout << "Enter string to search for:" << endl;
   String search_string = read_in(cin);
   while ((the_index = strstr(current->entry, search_string)) == -1)
      if (next_line() != success) break;
   if (the_index == -1) cout << "String was not found.";
   else {
      cout << (current->entry).c_str() << endl;
      for (int i = 0; i < the_index; i++)
         cout << " ";
      for (int j = 0; j < strlen(search_string.c_str()); j++)
         cout << "^";
   }
   cout << endl;
}

bool Editor::get_command()
/*
Post: Sets member user_command; returns true
      unless the user's command is q.
Uses: C library function tolower.
*/

{
   if (current != NULL)
      cout << current_position << " : "
           << current->entry.c_str() << "\n??" << flush;
   else
      cout << "File is empty. \n??" << flush;

   cin >> user_command;    //  ignores white space and gets command
   user_command = tolower(user_command);
   while (cin.get() != '\n')
      ;                      //  ignore user's enter key
   if (user_command == 'q')
      return false;
   else
      return true;
}

void Editor::run_command()
/*
Post: The command in user_command has been performed.
Uses: methods and auxiliary functions of the class Editor,
      the class String, and the String processing functions.
*/

{
   String temp_string;

   switch (user_command) {

   case '2':
      if (empty())
         cout << " Warning: empty buffer " << endl;
      else
         gutter();
      break;
```

TOC
Index
Help

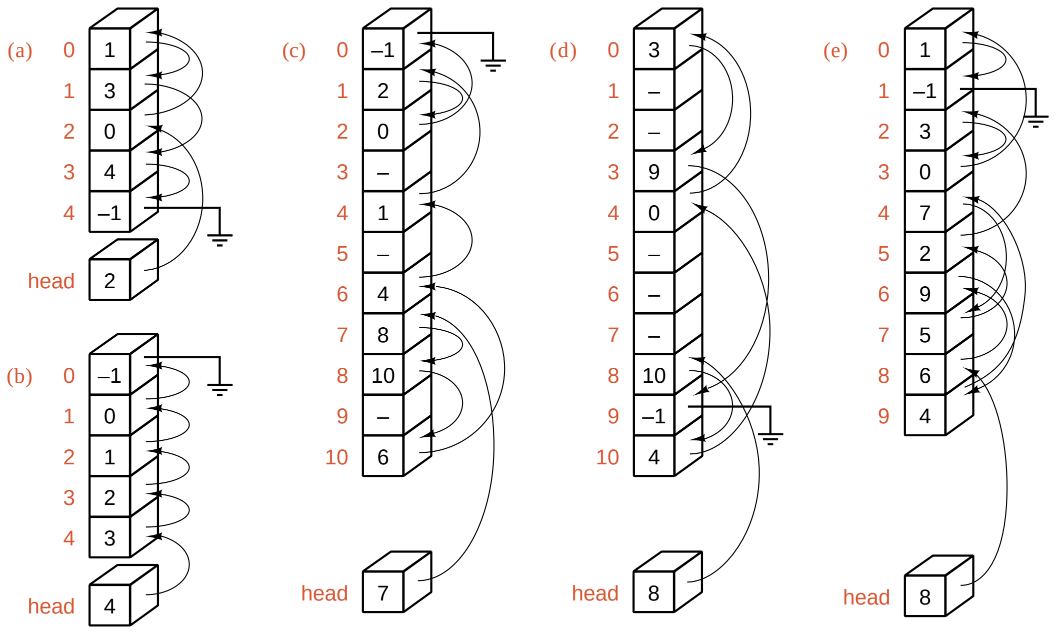
```
case 'b':
   if (empty())
      cout << " Warning: empty buffer " << endl;
   else
      while (previous_line() == success)
         ;
   break;

case 'c':
   if (empty())
      cout << " Warning: Empty file" << endl;
   else if (change_line() != success)
      cout << " Error: Substitution failed " << endl;
   break;

case 'd':
   if (remove(current_position, temp_string) != success)
      cout << " Error: Deletion failed " << endl;
   break;

case 'e':
   if (empty())
      cout << " Warning: empty buffer " << endl;
   else
      while (next_line() == success)
         ;
   break;

case 'f':
   if (empty())
      cout << " Warning: Empty file" << endl;
   else
      find_string();
   break;

case 'g':
   if (goto_line() != success)
      cout << " Warning: No such line" << endl;
   break;

case '?':
case 'h':
   cout << "Valid commands are: b(egin) c(hange) d(el) e(nd)" << endl
    << "f(ind) g(o) h(elp) i(nsert) l(ength) n(ext) p(rior) " << endl
    << "q(uit) r(ead) s(ubstitue) v(iew) w(rite)  (make) 2 (columns)"
    << endl;
   break;

case 'i':
   if (insert_line() != success)
      cout << " Error: Insertion failed " << endl;
   break;

case 'l':
   cout << "There are " << size() << " lines in the file." << endl;

   if (!empty())
      cout << "Current line length is "
           << strlen((current->entry).c_str()) << endl;
   break;
   case 'n':
      if (next_line() != success)
         cout << " Warning: at end of buffer" << endl;
      break;

   case 'p':
      if (previous_line() != success)
         cout << " Warning: at start of buffer" << endl;
      break;

   case 'r':
      read_file();
      break;

   case 's':
      if (substitute_line() != success)
         cout << " Error: Substitution failed " << endl;
      break;

   case 'v':
      traverse(write);
      break;

   case 'w':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else
         write_file();
      break;

   default :
      cout << "Press h or ? for help or enter a valid command: ";
   }
}

Editor::Editor(ifstream *file_in, ofstream *file_out)
/*
Post: Initialize the Editor members infile and outfile
      with the parameters.
*/

{
   infile = file_in;
   outfile = file_out;
}
```

6.5 LINKED LISTS IN ARRAYS

Exercises 6.5

E1. *Draw arrows showing how the list entries are linked together in each of the following* next *node tables.*

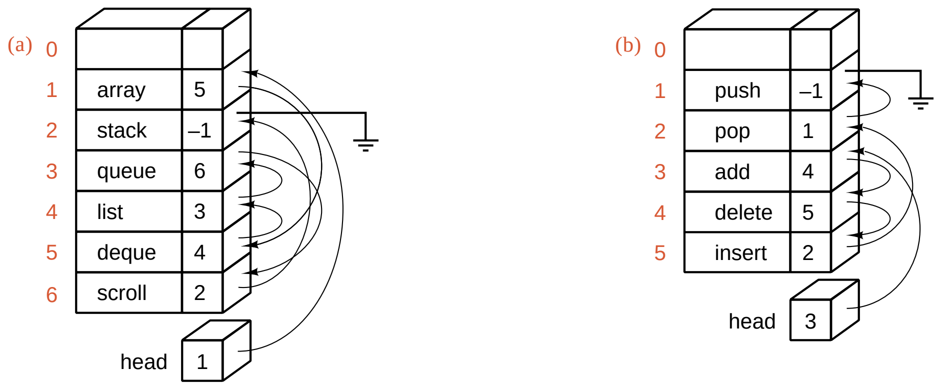
Answer

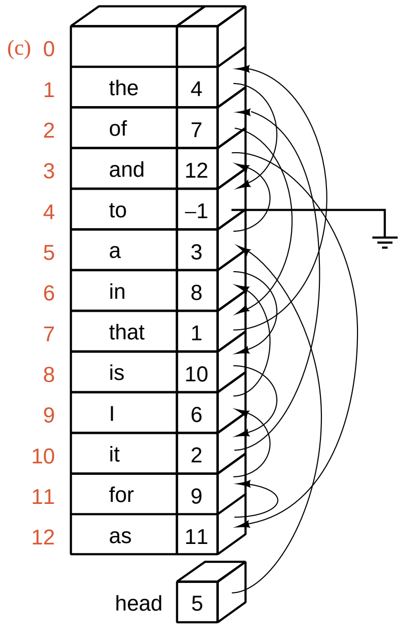


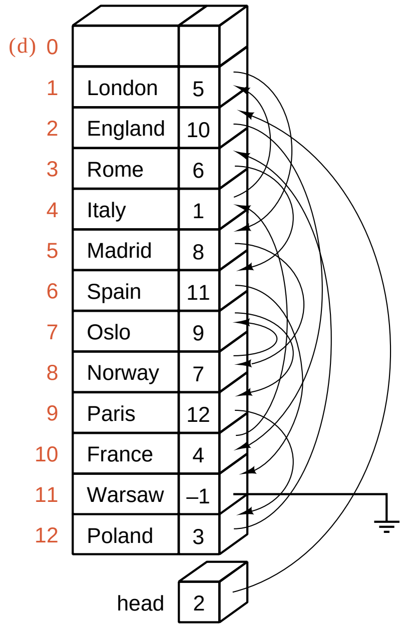
E2. *Construct* next *tables showing how each of the following lists is linked into alphabetical order. Also, in each case, give the value of the variable* head *that starts the list.*

(a)	*1*	array	*(c)*	*1*	the	*(d)*	*1*	London
	2	stack		*2*	of		*2*	England
	3	queue		*3*	and		*3*	Rome
	4	list		*4*	to		*4*	Italy
	5	deque		*5*	a		*5*	Madrid
	6	scroll		*6*	in		*6*	Spain
				7	that		*7*	Oslo
(b)	*1*	push		*8*	is		*8*	Norway
	2	pop		*9*	I		*9*	Paris
	3	add		*10*	it		*10*	France
	4	remove		*11*	for		*11*	Warsaw
	5	insert		*12*	as		*12*	Poland

Answer

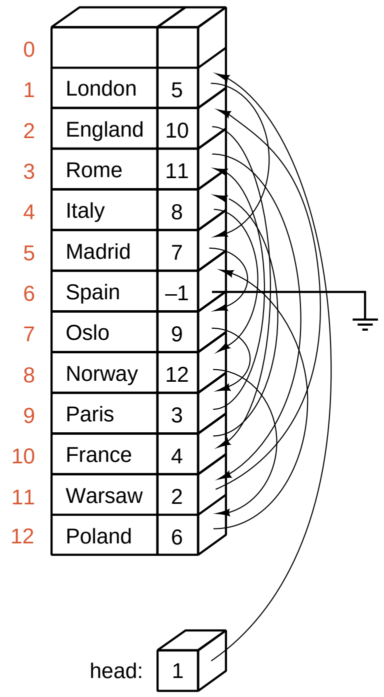






E3. *For the list of cities and countries in part (d) of the previous question, construct a* next *node table that produces a linked list, containing all the cities in alphabetical order followed by all the countries in alphabetical order.*

Answer



E4. *Write versions of each of the following functions for linked lists in arrays. Be sure that each function conforms to the specifications given in Section 6.1 and the declarations in this section.*

(a) set_position

Answer

```
template <class List_entry>
index List<List_entry> :: set_position(int position) const
/* Pre:  position is a valid position in the List; 0 ≤ position < count.
   Post: Returns the index of the Node in position of the List. */
```

```
{
   index n = head;
   for (int i = 0; i < position; i++, n = workspace[n].next);
   return n;
}
```

(b) List *(a constructor)*

Answer

```
template <class List_entry>
List<List_entry>::List()
/* Post: The List is initialized to be empty. */
{
   head = −1;
   count = 0;
   available = −1;
   last_used = −1;
}
```

(c) clear

Answer

```
template <class List_entry>
void List<List_entry>::clear()
/* Post: The List is cleared. */
{
   index p, q;
   for (p = head; p != −1; p = q) {
      q = workspace[p].next;
      delete_node(p);
   }
   count = 0;
   head = −1;
}
```

(d) empty

Answer

```
template <class List_entry>
bool List<List_entry>::empty() const
/* Post: The function returns true or false according as the List is empty or not. */
{
   return count <= 0;
}
```

(e) full

Answer

```
template <class List_entry>
bool List<List_entry>::full() const
/* Post: The function returns true or false according as the List is full or not. */
{
   return count >= max_list;
}
```

(f) size

Answer

```
template <class List_entry>
int List<List_entry>::size() const
/* Post: The function returns the number of entries in the List. */
{
   return count;
}
```

(g) retrieve

Answer

```
template <class List_entry>
Error_code List<List_entry> :: retrieve(int position, List_entry &x) const
/* Post: If the List is not full and 0 ≤ position < n, where n is the number of entries in the List,
         the function succeeds: The entry in position is copied to x. Otherwise the function fails
         with an error code of range_error. */
{
   index current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   x = workspace[current].entry;
   return success;
}
```

(h) remove

Answer

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly in position. Otherwise the function fails with a diagnostic error code. */
{
   index prior, current;
   if (count == 0) return underflow;
   if (position < 0 || position >= count) return range_error;
   if (position > 0) {
      prior = set_position(position − 1);
      current = workspace[prior].next;
      workspace[prior].next = workspace[current].next;
   }
   else {
      current = head;
      head = workspace[head].next;
   }
   x = workspace[current].entry;
   delete_node(current);
   count−−;
   return success;
}
```

(i) replace

Answer

```
template <class List_entry>
Error_code List<List_entry> :: replace(int position, const List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is replaced by x, all other entries remain unchanged. Otherwise
         the function fails with an error code of range_error. */
{
   index current;
   if (position < 0 || position >= count) return range_error;
   current = set_position(position);
   workspace[current].entry = x;
   return success;
}
```

(j) current_position

Answer

```
template <class List_entry>
int List<List_entry> :: current_position(index n) const
/* Post: Returns the position number of the Node stored at index n, or −1 if there no such
        Node. */
{
   int i = 0;
   for (index m = head;  m != −1;  m = workspace[m].next, i++)
      if (m ==  n) return i;                    //  position number of Node at index n
   return −1;
}
```

E5. *It is possible to implement a doubly linked list in a workspace array by using only one index* next. *That is, we do not need to keep a separate field* back *in the nodes that make up the workspace array to find the backward links. The idea is to put into* workspace[current] *not the index of the next entry on the list but, instead, a member* workspace[current].difference *giving the index of the next entry minus the index of the entry preceding* current. *We also must maintain two pointers to successive nodes in the list, the* current *index and the index* previous *of the node just before* current *in the linked list. To find the next entry of the list, we calculate*

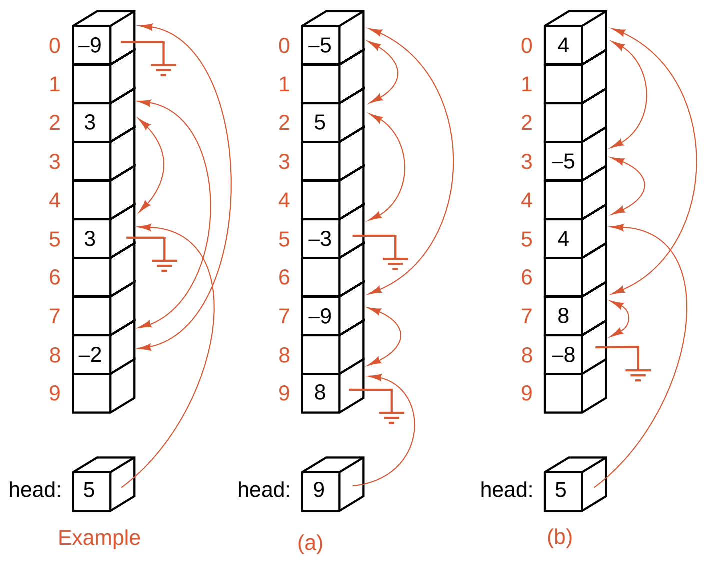
workspace[current].difference + previous;

Similarly, to find the entry preceding previous, *we calculate*

current − workspace[previous].difference;

An example of such a list is shown in the first part of the following diagram. Inside each box is shown the value stored in difference; *on the right is the corresponding calculation of index values.*

TOC
Index
Help



(a) *For the doubly linked list shown in the second part of the preceding diagram, show the values that will be stored in* list.head *and in the* difference *fields of occupied nodes of the workspace.*

Answer See the preceding diagram.

(b) *For the values of* list.head *and* difference *shown in the third part of the preceding diagram, draw links showing the corresponding doubly linked list.*

Answer See the preceding diagram.

(c) *With this implementation, write the function* set_position.

Answer We shall suppose that the following definitions and class declaration are available.

```
typedef int index;
const int max_list = 7;                    //  small value for testing purposes

template <class List_entry>
class Node {
public:
   List_entry entry;
   index difference;
};

template <class List_entry>
class List {
public:
//  Methods of the list ADT
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);
protected:
//  Data members
   Node<List_entry> workspace[max_list];
   index available, last_used;
   mutable index current, previous;
   int count;
   mutable int current_position;
//  Auxiliary member functions
   index new_node();
   void delete_node(index n);
   void set_position(int position) const;
};
```

The auxiliary member function set_position can be coded as follows.

```
template <class List_entry>
void List<List_entry>::set_position(int position) const
/* Pre:  position is a valid position in the List: 0 ≤ position < count.
   Post: The current Node pointer references the Node at position. */
{
   index temp;
   if (current_position <= position)
      for ( ; current_position != position; current_position++) {
         temp = current;
         current = workspace[current].difference + previous;
         previous = temp;
      }
   else
      for ( ; current_position != position; current_position−−) {
         temp = previous;
         previous = current − workspace[previous].difference;
         current = temp;
      }
}
```

(d) *With this implementation, write the function* insert.

Answer

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of entries in the List,
         the function succeeds: Any entry formerly at position and all later entries have their
         position numbers increased by 1 and x is inserted at position of the List.
         Else: the function fails with a diagnostic error code. */
{
   index newnode, following, preceding;
   if (position < 0 || position > count) return range_error;
   if (position == 0) {
      if (count == 0) following = −1;
      else {
         set_position(0);
         following = current;
      }
      preceding = −1;
   }
   else {
      set_position(position − 1);
      preceding = current;
      following = workspace[current].difference + previous;
   }
   newnode = new_node( );
   if (newnode == −1) return overflow;
   workspace[newnode].entry = x;
   workspace[newnode].difference = following − preceding;
   if (preceding != −1) workspace[preceding].difference += newnode − following;
   if (following != −1) workspace[following].difference −= newnode − preceding;
   current = newnode;
   current_position = position;
   previous = preceding;
   count++;
   return success;
}
```

(e) *With this implementation, write the function* remove.

Answer

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
/* Post: If 0 ≤ position < n, where n is the number of entries in the List, the function succeeds:
         The entry in position is removed from the List, and the entries in all later positions have
         their position numbers decreased by 1. The parameter x records a copy of the entry
         formerly in position. Otherwise the function fails with a diagnostic error code. */
```

```
{
   index old_node, preceding, following;
   if (count ==  0) return fail;
   if (position < 0 || position >= count) return range_error;
   set_position(position);
   old_node = current;
   preceding = previous;
   following = workspace[current].difference + previous;
   if (preceding != −1) workspace[preceding].difference += following − old_node;
   if (following != −1) {
      workspace[following].difference −= previous − old_node;
      current = following;
   }
   else {
      current = preceding;
      current_position−−;
      if (current_position >= 0)
         previous = following − workspace[current].difference;
   }
   x = workspace[old_node].entry;
   delete_node(old_node);
   count−−;
   return success;
}
```

6.6 APPLICATION: GENERATING PERMUTATIONS

TOC
Index
Help

Programming Projects 6.6

P1. *Complete a version of the permutation-generation program that uses one of the general list implementations by writing its main program and a function* process_permutation *that prints the permutation at the terminal. After testing your program, suppress printing the permutations and include the CPU timer functions provided with your compiler. Compare the performance of your program with each of the list implementations in* Section 6.2. *Also compare the performance with that of the optimized program written in the text.*

Answer The driver program for the various methods is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../doubly/list.h"
#include "../doubly/list.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include "permlist.cpp"
#include "camp.cpp"
#include "permutat.cpp"

void intro()
/*
PRE:  None.
POST: A program introduction has been printed.
*/
```

```
{
    cout <<  "\nPermutation generation program.\n\n"
         <<  "Given a user supplied integer representing n distinct\n"
         <<  "elements the program produces n! permutations.\n"
         <<  "Printing the permutations is optional." << endl;
}

void mode_screen()
/*
PRE:  None.
POST: A mode option menu has been printed.
*/

{
  cout <<  "\nSix modes of operation are available:\n\n"
   <<  "1) Dynamic (with print) - generation using dynamic linked list.\n"
   <<  "2) Dynamic (no print) - generation using dynamic linked list.\n"
   <<  "3) Camp (print) - campanology variant using the above.\n"
   <<  "4) Camp (no print).\n"
   <<  "5) Text Optimal (print) - array implementation as used in text."
   <<  "6) Text Optimal (no print)."
   <<  endl;
}
/*************************************************************************/

int main()
/*
PRE:  None.
POST: The permutation program has been executed.
*/

{
   intro();
   bool keep_going = true;
   int degree, mode;
   while (keep_going) {
        List<int> permutation;
        cout << "Number of elements to permute? ";
        cin >> degree;
        if (degree < 1) {
            cout << "Number must be at least 1." << endl;
            continue;
        }

        mode_screen();
        cout << "Enter the mode." << endl;
        cin  >> mode;
        Timer clock;
        clock.reset();

        if (mode == 1) permute(1, degree, permutation);
        if (mode == 2) permute0(1, degree, permutation);
        if (mode == 3 || mode == 4) {
           bool *forward = new bool[degree + 1];
           for (int i = 0; i <= degree; i++) forward[i] = true;
           camp(1, degree, permutation, (mode % 2) == 1, forward);
        }
```

```
            if (mode == 5 || mode == 6) {
               int *permutation = new int[degree + 1];
               permutation[0] = 0;
               permute_quick(1, degree, permutation, (mode %2 == 1));
            }

            double time = clock.elapsed_time();
            cout << "Time taken to permute was " << time <<" seconds"
                 << endl;
            cout << "Do you wish to try again\?: " << flush;
            keep_going = user_says_yes();
      }

      cout <<  "\nPermutation demonstration program finished.\n"
           <<  endl;
}
```

The driver can call on the following code for the generic List based method.

```
void print_entry(int &x)
{
   cout << x << " ";
}

void process_permutation(List < int > &permutation)
/*
Pre:  permutation contains a permutation in linked form.
Post: The permutation has been printed to the terminal.
*/

{
   permutation.traverse(print_entry);
   cout << endl;
}

void permute(int new_entry, int degree, List<int> &permutation)
/*
Pre:  permutation contains a permutation with entries in positions 1
      through new_entry - 1.
Post: All permutations with degree entries, built from the given
      permutation, have been constructed and processed.
Uses: permute recursively, process_permutation, and List functions.
*/

{
   for (int current = 0; current < permutation.size() + 1; current++) {
      permutation.insert(current, new_entry);
      if (new_entry == degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation.remove(current, new_entry);
   }
}
```

```
void permute0(int new_entry, int degree, List<int> &permutation)
/*
Pre:  permutation contains a permutation with entries in positions 1
m     through new_entry - 1.
Post: All permutations with degree entries, built from the given
      permutation, have been constructed but not processed.
Uses: permute recursively, process_permutation, and List functions.
*/

{
   for (int current = 0; current < permutation.size() + 1; current++) {
      permutation.insert(current, new_entry);
      if (new_entry != degree)
         permute0(new_entry + 1, degree, permutation);
      permutation.remove(current, new_entry);
   }
}
```

The driver can call on the following code for the optimized method.

```
void process_permutat(int *permutation)
/*
Pre:  permutation is in linked form.
Post: The permutation has been printed to the terminal.
*/

{
   int current = 0;
   while (permutation[current] != 0) {
      cout << permutation[current] << " ";
      current = permutation[current];
   }
   cout << endl;
}

void permute_quick(int new_entry, int degree,
                   int *permutation, bool do_print)
/*
Pre:  permutation contains a linked permutation with entries in
      positions 1 through new_entry - 1.
Post: All permutations with degree entries, built from the given
      permutation, have been constructed and processed.
Uses: Functions permute (recursively) and process_permutation.
*/

{
   int current = 0;
   do {
      permutation[new_entry] = permutation[current];
      permutation[current] = new_entry;
      if (new_entry == degree) {
        if (do_print) process_permutat(permutation);
      }
      else
         permute_quick(new_entry + 1, degree, permutation, do_print);
      permutation[current] = permutation[new_entry];
      current = permutation[current];
   } while (current != 0);
}
```

TOC
Index
Help

P2. *Modify the general version of* `permute` *so that the position occupied by each number does not change by more than one to the left or to the right from any permutation to the next one generated. [This is a simplified form of one rule for campanology (ringing changes on church bells).]*

Answer This is an interesting but very frustrating project. Unfortunately, we are comparing chalk with cheese in the time demonstration program. The time difference between the methods is primarily due to implementation and the data structures themselves rather than the difference of the algorithms. The campanological method, if used recursively, will produce very nearly the same time as the dynamic text method. Both require the calculation of all factorial subsets. For example, for $n = 6$ we are making $1! + 2! + 3! + 4! + 5! + 6! = 1 + 2 + 6 + 24 + 120 + 720 = 873$ nodes.

It is only if you flip to nonrecursive methods that the campanological method will speed up. The text optional method seems almost fraudulent. It would require a second indexing data structure to map anything that does not have an inherent ordering property. This is very different to the list implementations which are capable of changing themselves rather than just an index.

The driver can call on the following code for the campanological generic `List`-based method.

```
bool forwards = false;

void camp(int new_entry, int degree, List<int> &permutation,
          bool do_print, bool *forwards)
/*
PRE:  perm contains a permutation with entries 1 through newentry - 1.
POST: All permutations with degree entries, built from the given
      permutation, have been constructed and processed. This uses the
      campanological algorithm (once started only neighbours change).
*/

{
    forwards[new_entry] = !forwards[new_entry];
    if (forwards[new_entry])
       for (int current = 0; current <= permutation.size(); current++) {
          permutation.insert(current, new_entry);
          if (new_entry == degree) {
             if (do_print) process_permutation(permutation);
          }
          else
             camp(new_entry + 1, degree, permutation, do_print, forwards);
          permutation.remove(current, new_entry);
       }
    else
       for (int current = permutation.size(); current >= 0; current--) {
          permutation.insert(current, new_entry);
          if (new_entry == degree) {
             if (do_print) process_permutation(permutation);
          }
          else
             camp(new_entry + 1, degree, permutation, do_print, forwards);
          permutation.remove(current, new_entry);
       }
}
```

REVIEW QUESTIONS

1. *Which of the operations possible for general lists are also possible for queues? for stacks?*

Operations on queues:

insertion (at tail only)
deletion (at head only)
retrieval (from head only)
empty

Operations on stacks:

insertion (at top only)
deletion (at top only)
retrieval (from top only)
empty

2. *List three operations possible for general lists that are not allowed for either stacks or queues.*

Insertion or value changes to arbitrary positions in the list, deletion from anywhere in the list are not allowed on either stacks or queues.

3. *If the items in a list are integers (one word each), compare the amount of space required altogether if (a) the list is kept contiguously in an array 90 percent full, (b) the list is kept contiguously in an array 40 percent full, and (c) the list is kept as a linked list (where the pointers take one word each).*

Consider that the array can hold max integers. Therefore, the contiguous lists (no matter how much of it is used) uses max words of memory. The linked list will occupy two words of memory for each integer stored due to the use of links. Therefore, a linked list of $\frac{9}{10}$ max integers would occupy $\frac{9}{5}$ max words of memory and a linked list of $\frac{4}{10}$ max integers would occupy $\frac{4}{5}$ max words of memory. This would imply that using linked lists is only a saving if the number of items stored, in this case, is less than half of the array allocation.

4. *Repeat the comparisons of the previous exercise when the items in the list are entries taking 200 words each.*

The contiguous lists (no matter how much of it is used) uses 200 max words of memory. The linked list will occupy 201 words of memory for each entry stored due to the use of links. Therefore, a linked list of $\frac{9}{10}$ max entries would occupy $\frac{1809}{10}$ max $\approx$ 181 max words of memory and a linked list of $\frac{4}{10}$ max entries would occupy $\frac{402}{5}$ max $\approx$ 80 max words of memory, both a significant saving from the contiguous implementation.

5. *What is the major disadvantage of linked lists in comparison with contiguous lists?*

One of the major disadvantages of linked lists as compared to contiguous lists is that the links themselves require the equivalent of one word of memory space, which could be used for storage of other data. Secondly, linked lists are not suited to random access. Also, it may require slightly more computer time to access a linked node than it would a item in contiguous storage. Lastly, manipulating pointers may be more confusing for beginning programmers and may require more programming effort.

6. *What are the major disadvantages of C-strings?*

C-strings must conform to a collection of conventions that are not enforced by the compiler or language. Any deviation from these conventions is likely to lead to serious errors.

7. *What are some reasons for implementing linked lists in arrays with indices instead of in dynamic memory with pointers?*

A few languages do not provide for dynamic memory allocation used by linked lists and therefore require the implementation of arrays. Some applications, where the same data is sometimes best treated as a linked list and other times as a contiguous list, also require implementing linked lists in arrays.

Searching 7

7.2 SEQUENTIAL SEARCH

Exercises 7.2

E1. *One good check for any algorithm is to see what it does in extreme cases. Determine what sequential search does when*

(a) *there is only one item in the list.*

Answer Sequential search, when performed on a list of size one, iterates through the main loop once, regardless of whether the key is found or not, performing only one key comparison.

(b) *the list is empty.*

Answer If sequential search is invoked with an empty list, the main loop will not iterate and no key comparisons will occur.

(c) *the list is full.*

Answer If sequential search is invoked with a full list, the main loop will iterate either n times, for an unsuccessful search, or an average of $\frac{1}{2}(n+1)$ times for a successful search, performing one key comparison with each iteration.

E2. *Trace sequential search as it searches for each of the keys present in a list containing three items. Determine how many comparisons are made, and thereby check the formula for the average number of comparisons for a successful search.*

Answer Let the list contain the three keys a, b, and c. First perform a search for the key a by invoking sequential_search, setting found to false and position to 0. The while loop will execute, found will be set to true and the loop will terminate after one key comparison. Upon searching for key b the while loop will execute twice before finding the target key, thus performing two key comparisons. A search for the key c will require three key comparisons. The average of these numbers is $\frac{1}{3}(1+2+3) = 2$. This equals $\frac{1}{2}(n+1)$ with $n = 3$. Note that a search for a key that does not appear in the list requires n key comparisons.

E3. *If we can assume that the keys in the list have been arranged in order (for example, numerical or alphabetical order), then we can terminate unsuccessful searches more quickly. If the smallest keys come first, then we can terminate the search as soon as a key greater than or equal to the target key has been found. If we assume that it is equally likely that a target key not in the list is in any one of the $n+1$ intervals (before the first key, between a pair of successive keys, or after the last key), then what is the average number of comparisons for unsuccessful search in this version?*

Answer Assuming that it is equally likely that a target key not in the list belongs in any one of the $n+1$ intervals, that is, it is equally likely to terminate the search after 1, 2, ..., n comparisons, the average number of key comparisons is

$$\frac{1+2+\cdots+n+n}{n+1}.$$

This is equal to

$$\frac{\frac{1}{2}n(n+1)+n}{n+1} = \frac{1}{2}n + 1 - \frac{1}{n+1}.$$

E4. *At each iteration, sequential search checks two inequalities, one a comparison of keys to see if the target has been found, and the other a comparison of indices to see if the end of the list has been reached. A good way to speed up the algorithm by eliminating the second comparison is to make sure that eventually key* target *will be found, by increasing the size of the list and inserting an extra item at the end with key* target. *Such an item placed in a list to ensure that a process terminates is called a* ***sentinel***. *When the loop terminates, the search will have been successful if* target *was found before the last item in the list and unsuccessful if the final sentinel item was the one found.*

sentinel

Write a C++ function that embodies the idea of a sentinel in the contiguous version of sequential search using lists developed in Section 6.2.2.

Answer

```
Error_code sequential_search(List<Record> &the_list,
                             const Record &target, int &position)
/* Post: If an entry in the_list has key equal to target, then return success and the output param-
         eter position locates such an entry within the list. Otherwise return not_present and
         position becomes invalid. */
{
   Record data;
   int s = the_list.size();
   the_list.insert(s, target);
   for (position = 0; position <= s; position++) {
      the_list.retrieve(position, data);
      if (data == target) break;
   }
   the_list.remove(s, data);
   if (position < s) return success;
   return not_present;
}
```

E5. *Find the number of comparisons of keys done by the function written in Exercise E4 for*

(a) *unsuccessful search.*

Answer As the repeat loop iterates until the sentinel is found, $n+1$ key comparisons are required for an unsuccessful search, where n is the length of the original list.

(b) *best successful search.*

Answer The best successful search, occurring if the target is in the first position, requires one key comparison.

(c) *worst successful search.*

Answer The worst successful search, occurring if the target key is in the last position, requires n key comparisons.

(d) *average successful search.*

Answer The average successful search requires $\frac{1}{2}(n+1)$ key comparisons.

Programming Projects 7.2

P1. *Write a program to test sequential search and, later, other searching methods using lists developed in Section 6.2.2. You should make the appropriate declarations required to set up the list and put keys into it. The keys are the odd integers from 1 to n, where the user gives the value of n. Then successful searches can be tested by searching for odd integers, and unsuccessful searches can be tested by searching for even integers. Use the function* test_search *from the text to do the actual testing of the search function. Overload the key comparison operators so that they increment the* counter. *Write appropriate* introduction *and* print_out *functions and a menu driver. For now, the only options are to fill the list with a user-given number of entries, to test* sequential_search, *and to quit. Later, other searching methods could be added as further options.*

Find out how many comparisons are done for both unsuccessful and successful searches, and compare these results with the analyses in the text.

Run your program for representative values of n, such as $n = 10$, $n = 100$, $n = 1000$.

Answer Here is a menu-driven main program for Projects P1 and P3:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../2/key.h"
#include "../2/key.cpp"
typedef Key Record;
#include "../2/sequenti.cpp"
#include "../2/test.cpp"

void intro()
{
   cout << "Testing program for sequential search.\n"
        << "User options are:\n"
        << "[H]elp  [Q]uit  [F]ill list   [T]est sequential search "
        << endl;
}

main()
/*
Post: Sequential search has been tested for a user
      selected list size, with the function test_search
*/

{
   intro();
   int n, searches = 100;
   char command = ' ';
   List<Record> the_list;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, T: " << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           cout << "User options are:\n"
                << "[H]elp  [Q]uit  [F]ill list  [T]est sequential search"
                << endl;
         break;

         case 't': case 'T':
           test_search(searches, the_list);
         break;

         case 'f': case 'F':
           the_list.clear();
           cout << "Enter an upper bound n for the size of list entries:"
                << flush;
           cin  >> n;
           for (int i = 1; i <= n; i += 2)
              if (the_list.insert(the_list.size(), i) != success)
                  cout << " Overflow in filling list." << endl;
         break;
      }
   }
}
```

Because we have used list operations throughout, we can use either contiguous or linked lists with no change other than the `#include` directive for the list implementation. Indeed since we have included a linked implementation in Project P1, it is also a solution to Project P3.

Other programs, functions, and methods, including the class `Key`, the sequential search program, and the `test_search` program are taken from the text and shown below.

Definition of `Key` class:

```
class Key {
   int key;
public:
   static int comparisons;
   Key (int x = 0);
   int the_key() const;
};

bool operator ==(const Key &x,const Key &y);
bool operator >(const Key &x,const Key &y);
bool operator <(const Key &x,const Key &y);
bool operator >=(const Key &x,const Key &y);
bool operator <=(const Key &x,const Key &y);
bool operator !=(const Key &x,const Key &y);
```

Implementation of `Key` class:

```
int Key::comparisons = 0;

int Key::the_key() const
{
   return key;
}
```

```
Key::Key (int x)
{
   key = x;
}

bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}

bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}

bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}

bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}

bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}
```

Sequential search function:

```
Error_code sequential_search(const List<Record> &the_list,
                             const Key &target, int &position)
/*
Post: If an entry in the_list has key equal to target, then return
      success and the output parameter position locates such an entry
      within the list.

      Otherwise return not_present and position becomes invalid.
*/

{
   int s = the_list.size();
   for (position = 0; position < s; position++) {
      Record data;
      the_list.retrieve(position, data);
      if ((Key) data == target) return success;
   }
   return not_present;
}
```

Main driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "key.h"
#include "key.cpp"
typedef Key Record;
#include "sequenti.cpp"
#include "test.cpp"

main()
{
   int list_size = 20, searches = 100;
   cout << "Timing and testing of sequential search on a list of size 20"
        << endl;

   int i;
   List<Record> the_list;
   for (i = 0; i < list_size; i++)
      if (the_list.insert(i, 2 * i + 1) != success) {
         cout << " Overflow in filling list." << endl;
      }
   test_search(searches, the_list);
}
```

Testing program:

```
void print_out(char *comment, float t, int comp_ct, int searches)
{
   float average;
   cout << comment << " Search Statistics: " << endl;
   cout << " Time for " << searches << " comparisons was " << t << endl;
   average = (( float ) comp_ct) / (( float ) searches);
   cout << " Average number of comparisons per search was " << average
        << endl;
}

void test_search(int searches, List<Record> &the_list)
/*
Pre:  None.
Post: The number of key comparisons and CPU time for a
      sequential searching funtion
      have been calculated.
Uses: Methods of the classes List, Random, and Timer,
      together with an output function print_out
*/

{
  int list_size = the_list.size();
  if (searches <= 0 || list_size < 0) {
     cout << " Exiting test: " << endl
          << " The number of searches must be positive." << endl
          << " The number of list entries must exceed 0." << endl;
     return;
  }
  int i, target, found_at;
  Key::comparisons = 0;
  Random number;
  Timer clock;
  for (i = 0; i < searches; i++) {
     target = 2 * number.random_integer(0, list_size - 1) + 1;
     if (sequential_search(the_list, target, found_at) == not_present)
        cout << "Error: Failed to find expected target " << target << endl;
  }
  print_out("Successful", clock.elapsed_time(), Key::comparisons, searches);
  Key::comparisons = 0;
  clock.reset();
  for (i = 0; i < searches; i++) {
     target = 2 * number.random_integer(0, list_size);
     if (sequential_search(the_list, target, found_at) == success)
        cout << "Error: Found unexpected target " << target
             << " at " << found_at << endl;
  }
  print_out("Unsuccessful", clock.elapsed_time(),
                            Key::comparisons, searches);
}
```

P2. *Take the driver program written in Project P1 to test searching functions, and insert the version of sequential search that uses a sentinel (see Exercise E4). For various values of n, determine whether the version with or without a sentinel is faster. By experimenting, find the cross-over point between the two versions, if there is one. That is, for what value of n is the extra time needed to insert a sentinel at the end of a list of size n about the same as the time needed for extra comparisons of indices in the version without a sentinel?*

sentinel search

Answer A modified driver that allows for a regular sequential search or a sentinel based sequential search is implemented as:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../2/key.h"
#include "../2/key.cpp"
typedef Key Record;
#include "../2/sequenti.cpp"
#include "sentinel.cpp"
#include "test.cpp"

void intro()
{
   cout << "Testing program for sequential search.\n"
        << "User options are:\n"
        << "[H]elp  [Q]uit  [F]ill list \n"
        << "test [R]egular sequential search \n"
        << "test sequential search with [S]entinel\n"
        << endl;
}
```

```
main()
/*
Post: Sequential search has been tested for a user
      selected list size, with the function test_search
*/

{
   intro();
   int n, searches = 100;

   char command = ' ';
   List<Record> the_list;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, R, S: " << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           cout << "User options are:\n"
                << "[H]elp  [Q]uit  [F]ill list \n"
                << "test [R]egular sequential search \n"
                << "test sequential search with [S]entinel\n"
                << endl;
         break;

         case 'r': case 'R':
           test_search(searches, the_list, 'r');
         break;

         case 's': case 'S':
           test_search(searches, the_list, 's');
         break;

         case 'f': case 'F':
           the_list.clear();
           cout << "Enter an upper bound n for the size of list entries:"
                << flush;
           cin  >> n;
           for (int i = 1; i <= n; i += 2)
              if (the_list.insert(the_list.size(), i) != success)
                  cout << " Overflow in filling list." << endl;
         break;
      }
   }
}
```

The implementation of sentinel search is:

```
Error_code sentinel_search(List<Record> &the_list,
                             const Record &target, int &position)
/*
Post: If an entry in the_list has key equal to target, then return
      success and the output parameter position locates such an entry
      within the list.
      Otherwise return not_present and position becomes invalid.
*/

{
   Record data;
   int s = the_list.size();
   the_list.insert(s, target);
   for (position = 0; position <= s; position++) {
      the_list.retrieve(position, data);
      if (data == target) break;
   }
   the_list.remove(s, data);
   if (position < s) return success;
   return not_present;
}
```

A modified testing function is in:

```
void print_out(char *comment, float t, int comp_ct, int searches)
{
   float average;
   cout << comment << " Search Statistics: " << endl;
   cout << " Time for " << searches << " comparisons was " << t << endl;
   average = (( float ) comp_ct) / (( float ) searches);
   cout << " Average number of comparisons per search was " << average
        << endl;
}

void test_search(int searches, List<Record> &the_list, char method)
/*
Pre:   None.
Post: The number of key comparisons and CPU time for a
      sequential searching funtion have been calculated.
Uses: Methods of the classes List, Random, and Timer,
      together with an output function print_out
*/
{
   int list_size = the_list.size();
   if (searches <= 0 || list_size < 0) {
      cout << " Exiting test: " << endl
           << " The number of searches must be positive." << endl
           << " The number of list entries must exceed 0." << endl;
      return;
   }
   int i, target, found_at;
   Key::comparisons = 0;
   Random number;
   Timer clock;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (method == 'r') {
       if (sequential_search(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
      }
      else if (method == 's')
       if (sentinel_search(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                               Key::comparisons, searches);
   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (method == 'r') {
       if (sequential_search(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
      }
      else if (method == 's')
       if (sentinel_search(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(),
                             Key::comparisons, searches);
}
```

The Key class and other programs are as in Project P1.

linked sequential search

P3. *What changes are required to our sequential search function and testing program in order to operate on simply linked lists as developed in Section 6.2.3? Make these changes and apply the testing program from Project P1 for linked lists to test linked sequential search.*

Answer The solution is included with that of Project P1.

7.3 BINARY SEARCH

Exercises 7.3

E1. *Suppose that* the_list *contains the integers 1, 2, . . . , 8. Trace through the steps of* binary_search_1 *to determine what comparisons of keys are done in searching for each of the following targets:* **(a)** *3,* **(b)** *5,* **(c)** *1,* **(d)** *9,* **(e)** *4.5.*

Answer **(a)** Position three of the list is examined first; the key in that position is greater than the target; and the top of the sublist reduces to three. Position one is then examined and the bottom of the sublist is increased to two because the target is larger than the key in position one. Position two is examined and, since the key is not greater than the target, the top is reduced to two. The loop stops as top is no longer larger than bottom. The comparison outside the loop verifies that the target has been found. A total of four key comparisons were done.

(b) In searching for the key 5, the positions three, five, and four of the list are compared to the target (four key comparisons).

(c) In searching for the key 1, the positions three, one, and zero of the list are compared to the target (four key comparisons).

(d) In searching, unsuccessfully, for the key 9, the positions three, five, six, and (outside the loop) seven of the list are compared to the target (four key comparisons).

(e) In searching, unsuccessfully, for the key 4.5, the positions three, five, four, and (outside the loop) four again of the list are compared to the target (four key comparisons).

E2. *Repeat Exercise E1 using* binary_search_2.

Answer **(a)** In searching for the key 3, the key in position three is compared to the target, found to be not equal, compared again, found to be less than the target, and the top of the sublist is reduced to two. The key in position one of the list is compared to the target, found to be not equal, compared again, found to be not less than the target and the bottom of the sublist is increased to two. The key position two is compared to the target, found to be equal, and the comparisons terminate after five comparisons of keys.

(b) In searching for the key 5, the keys in positions three, five, and four are compared before equality is found (five key comparisons).

(c) In searching for the key 1, the keys in positions three, one, and zero are compared before equality is found (five key comparisons).

(d) In searching for the key 9, the keys in positions three, five, six, and seven are compared before bottom exceeds top and the comparisons terminate (eight key comparisons).

(e) In searching for the key 4.5, the keys in positions three, five, and four are compared before bottom exceeds top and the comparisons terminate (six key comparisons).

E3. *[Challenging] Suppose that L_1 and L_2 are ordered lists containing n_1 and n_2 integers, respectively.*

(a) *Use the idea of binary search to describe how to find the median of the $n_1 + n_2$ integers in the combined lists.*

Answer Without loss of generality we assume that $n_1 \le n_2$, for, if not, we can interchange the two lists before starting. Let x_1 and x_2 be the medians of L_1 and L_2, respectively, so that x_1 appears in position $\lceil n_1/2 \rceil - 1$ and x_2 in position $\lceil n_2/2 \rceil - 1$ of their respective lists. Suppose, first, that $x_1 \le x_2$. The median of the combined list must be somewhere between x_1 and x_2, inclusive, since no more than half the elements are less than the smaller of the two medians and no more than half are larger than the larger median. Hence no element strictly to the left of x_1 in L_1 could be the median, nor could any element to the right of x_2 in L_2. Since $n_1 \le n_2$, we can delete all elements strictly to the left of x_1 from L_1 and an equal number from the right end of L_2 and, in doing so, we will not have changed the median.

When $x_1 > x_2$ similar conditions hold, and we can delete the elements strictly to the right of x_1 from L_1 and an equal number from the left end of L_2.

In the same way as binary search, we continue this process, at each step removing almost half the elements from L_1 and an equal number from L_2. When L_1 has length 2, however, this process may remove no further elements, since the (left) median of L_1 is its first element. It turns out, however, that when L_1 has even length and $x_1 \le x_2$, then x_1 itself may be deleted without changing the median. Hence we modify the method so that, instead of deleting the elements strictly on one side of x_1, we delete $\lfloor n_1/2 \rfloor$ elements from L_1 and from L_2 at each stage. This process terminates when the length of L_1 is reduced to 1.

It is, finally, easy to find the median of a list with one extra element adjoined. Let *a* be the unique element in L_1, let *b* be the element of L_2 in position $\lfloor n_2/2 \rfloor$ (so *b* is the left median if n_2 is even and is one position left of the median if n_2 is odd), and let *c* be the element of L_2 in position $\lfloor n_2/2 \rfloor + 1$. It is easy to check that if $a \le b$ then b is the median of the combined list, else if $a \le c$ then a is the median, else c is the median.

(b) *Write a function that implements your method.*

Answer
```
Error_code find_median(const Ordered_list &l1, const Ordered_list &l2,
                       Record &median)
/* Pre:  The ordered lists l1,l2 are not empty and l2 contains at least as many entries as l1.
   Post: The median of the two lists combined in order is returned as the output parameter
         median. */
{
   int reduce;                          //  removed from both ends at each iteration
   int mid1, bottom1, top1;
   int mid2, bottom2, top2;
   Record a, b, c;                      //  used to check values in short lists
   if (l1.size( ) <= 0 || l1.size( ) > l2.size( ))
      return fail;
   bottom1 = bottom2 = 0;
   top1 = l1.size( ) − 1;
   top2 = l2.size( ) − 1;
   /* Invariant: median is either between bottom1 and top1 in l1 or between bottom2 and top2
      in l2, inclusive; median is the median of the combined lists from bottom1 to top1 and
      bottom2 to top2. */
```

```
    Record data1, data2;
    while (top1 > bottom1) {                /* while l1 has at least two entries          */
        mid1 = (bottom1 + top1)/2;
        l1.retrieve(mid1, data1);
        mid2 = (bottom2 + top2)/2;
        l2.retrieve(mid2, data2);
        reduce = (top1 − bottom1 + 1)/2;
        if (data1 < data2) {
            top2 −= reduce;
            bottom1 += reduce;              //  Reduce each list by the same amount
        } else {
            bottom2 += reduce;
            top1 −= reduce;
        }
    }
    //  The preceding loop terminates when l1 has length exactly 1.
    if (top1 != bottom1) {
        cout << "Unexpected program error." << endl;
        return fail;
    }
    l1.retrieve(top1, a);
    //  if l2 has only one element; median is the smaller.
    if (bottom2 == top2) {
        l2.retrieve(top2, b);
        if (a <= b) median = a;
        else median = b;
    } else {
    //  l2 has more than one element; find entry to left of median.
        mid2 = (bottom2 + top2 − 1)/2;
        l2.retrieve(mid2, b);               //  at or to the left of median in l2
        l2.retrieve(mid2 + 1, c);           //  at median or right median in l2
        if (a <= b) median = b;
        else if (a <= c) median = a;
        else median = c;
    }
    return success;
}
```

Programming Projects 7.3

P1. *Take the driver program of Project P1 of Section 7.2 (page 277), and make* binary_search_1 *and* binary_search_2 *the search options. Compare their performance with each other and with sequential search.*

Answer A driver that implements the menu options for Projects P1 and P2 is provided by:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../2/key.h"
#include "../2/key.cpp"
typedef Key Record;
```

```
#include "../3/ordlist.h"
#include "../3/ordlist.cpp"
#include "../3/binary1.cpp"
#include "../3/binary2.cpp"
#include "../3/rbinary1.cpp"
#include "../3/rbinary2.cpp"
#include "test.cpp"

void intro()
{
   cout << "Testing program for binary search.\n"
        << "User options are:\n"
        << "[H]elp  [Q]uit  [F]ill list \n"
        << "test binary search version [1]\n"
        << "test binary search version [2]\n"
        << "test recursive binary search version [O]ne\n"
        << "test recursive binary search version [T]wo\n"
        << endl;
}

main()
/*
Post: Binary search has been tested for a user
      selected list size, with the function test_search
*/

{
   intro();
   int n, searches = 100;

   char command = ' ';
   Ordered_list the_list;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, 1, 2, O, T: " << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           cout << "User options are:\n"
                << "[H]elp  [Q]uit  [F]ill list \n"
                << "test binary search version [1]\n"
                << "test binary search version [2]\n"
                << "test recursive binary search version [O]ne\n"
                << "test recursive binary search version [T]wo\n"
                << endl;
         break;

         case '1': case '2': case 'o': case 't':
           test_search(searches, the_list, command);
         break;

         case 'T': case 'O':
           test_search(searches, the_list, command + 'a' - 'A');
         break;
         case 'f': case 'F':
           the_list.clear();
           cout << "Enter an upper bound n for the size of list entries:"
                << flush;
           cin  >> n;
           for (int i = 1; i <= n; i += 2)
              if (the_list.insert(the_list.size(), i) != success)
                  cout << " Overflow in filling list." << endl;
         break;
      }
   }
}
```

This program uses a testing function:

```
void print_out(char *comment, float t, int comp_ct, int searches)
{
   float average;
   cout << comment << " Search Statistics: " << endl;
   cout << " Time for " << searches << " comparisons was " << t << endl;
   average = (( float ) comp_ct) / (( float ) searches);
   cout << " Average number of comparisons per search was " << average
        << endl;
}

void test_search(int searches, Ordered_list &the_list, char method)
/*
Pre:  None.
Post: The number of key comparisons and CPU time for a searching method
      have been calculated.
Uses: Methods of the classes Ordered_list, Random, Timer,
      and an output function print_out.
*/

{
   int list_size = the_list.size();
   if (searches <= 0 || list_size < 0) {
      cout << " Exiting test: " << endl
           << " The number of searches must be positive." << endl
           << " The number of list entries must exceed 0." << endl;
      return;
   }

   int i, target, found_at;
   Random number;
   Timer clock;

   switch (method) {
   case 'o':
      cout << "\n\nRecursive Binary Search 1" << endl;
      Key::comparisons = 0;
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size - 1) + 1;
         if (run_recursive_binary_1(the_list, target, found_at) ==
                                                             not_present)
            cout << "Error: Failed to find expected target " << target
                 << endl;
      }
      print_out("Successful", clock.elapsed_time(),
                               Key::comparisons, searches);
      Key::comparisons = 0;
      clock.reset();
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size);
         if (run_recursive_binary_1(the_list, target, found_at) == success)
            cout << "Error: Found unexpected target " << target
                 << " at " << found_at << endl;
      }
      print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                                searches);
   break;

   case '1':
      cout << "\n\nNon-recursive Binary Search 1" << endl;
      Key::comparisons = 0;
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size - 1) + 1;
         if (binary_search_1(the_list, target, found_at) == not_present)
            cout << "Error: Failed to find expected target " << target
                 << endl;
      }
      print_out("Successful", clock.elapsed_time(),
                                Key::comparisons, searches);

      Key::comparisons = 0;
      clock.reset();
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size);
         if (binary_search_1(the_list, target, found_at) == success)
            cout << "Error: Found unexpected target " << target
                 << " at " << found_at << endl;
      }
      print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
          searches);
   break;

   case 't':
      cout << "\n\nRecursive Binary Search 2" << endl;
      Key::comparisons = 0;
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size - 1) + 1;
         if (run_recursive_binary_2(the_list, target, found_at) ==
                                                           not_present)
            cout << "Error: Failed to find expected target " << target
                 << endl;
      }
      print_out("Successful", clock.elapsed_time(),
                                Key::comparisons, searches);
      Key::comparisons = 0;
      clock.reset();
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size);
         if (run_recursive_binary_2(the_list, target, found_at) == success)
            cout << "Error: Found unexpected target " << target
                 << " at " << found_at << endl;
      }
      print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                                                 searches);
   break;

   case '2':
      cout << "\n\nNon-recursive Binary Search 2" << endl;
      Key::comparisons = 0;
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size - 1) + 1;
         if (binary_search_2(the_list, target, found_at) == not_present)
            cout << "Error: Failed to find expected target " << target
                 << endl;
      }
      print_out("Successful", clock.elapsed_time(),
                              Key::comparisons, searches);

      Key::comparisons = 0;
      clock.reset();
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size);
         if (binary_search_2(the_list, target, found_at) == success)
            cout << "Error: Found unexpected target " << target
                 << " at " << found_at << endl;
      }
      print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                searches);
   break;
   }
}
```

The remaining code generally comes from the text, but for convenience is listed below.
Key definition:

```
class Key {
   int key;
public:
   static int comparisons;
   Key (int x = 0);
   int the_key() const;
};

bool operator ==(const Key &x,const Key &y);
bool operator >(const Key &x,const Key &y);
bool operator <(const Key &x,const Key &y);
bool operator >=(const Key &x,const Key &y);
bool operator <=(const Key &x,const Key &y);
bool operator !=(const Key &x,const Key &y);
```

Ordered_list definition:

```
class Ordered_list:public List<Record>{
public:
   Ordered_list();
   Error_code insert(const Record &data);
   Error_code insert(int position, const Record &data);
   Error_code replace(int position, const Record &data);
};
```

Key implementation:

```
int Key::comparisons = 0;

int Key::the_key() const
{
   return key;
}

Key::Key (int x)
{
   key = x;
}

bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}

bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}

bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}

bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}

bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}
```

`Ordered_list` implementation:

```
Ordered_list::Ordered_list()
{
}
```

```
Error_code Ordered_list::insert(const Record &data)
/*
Post: If the Ordered_list is not full,
      the function succeeds: The Record data is
      inserted into the list, following the
      last entry of the list with a strictly lesser key
      (or in the first list position if no list
      element has a lesser key).

Else: the function fails with the diagnostic Error_code overflow.
*/

{
   int s = size();
   int position;
   for (position = 0; position < s; position++) {
      Record list_data;
      retrieve(position, list_data);
      if (data >= list_data) break;
   }
   return List<Record>::insert(position, data);
}

Error_code Ordered_list::insert(int position, const Record &data)
/*
Post: If the Ordered_list is not full,
      0 <= position <= n,
      where n is the number of entries in the list,
      and the Record ata can be inserted at
      position in the list, without disturbing
      the list order, then
      the function succeeds:
      Any entry formerly in
      position and all later entries have their
      position numbers increased by 1 and
      data is inserted at position of the List.

Else: the function fails with a diagnostic Error_code.
*/

{
   Record list_data;
   if (position > 0) {
      retrieve(position - 1, list_data);
      if (data < list_data)
         return fail;
   }
   if (position < size()) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
   }
   return List<Record>::insert(position, data);
}
```

```
Error_code Ordered_list::replace(int position, const Record &data)
{
   Record list_data;
   if (position > 0) {
      retrieve(position - 1, list_data);
      if (data < list_data)
         return fail;
   }
   if (position < size()) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
   }
   return List<Record>::replace(position, data);
}
```

Non-recursive binary search, first version:

```
Error_code binary_search_1 (const Ordered_list &the_list,
                            const Key &target, int &position)
/*
Post: If a Record in the_list  has Key equal
      to target, then position locates
      one such entry and a code of success is returned.
      Otherwise, not_present is returned and position is
      undefined.
Uses: Methods for classes List and Record.
*/

{
   Record data;
   int bottom = 0, top = the_list.size() - 1;

   while (bottom < top) {
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data < target)
         bottom = mid + 1;
      else
         top = mid;
   }

   if (top < bottom) return not_present;
   else {
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
```

Non-recursive binary search, second version:

```
Error_code binary_search_2(const Ordered_list &the_list,
                           const Key &target, int &position)
/*
Post: If a Record in the_list   has key equal
      to target, then position locates
      one such entry and a code of success is returned.
      Otherwise, not_present is returned and position is
      undefined.
Uses: Methods for classes Ordered_list and Record.
*/

{
   Record data;
   int bottom = 0, top = the_list.size() - 1;
   while (bottom <= top) {
      position = (bottom + top) / 2;
      the_list.retrieve(position, data);
      if (data == target) return success;
      if (data < target) bottom = position + 1;
      else top = position - 1;
   }
   return not_present;
}
```

P2. *Incorporate the recursive versions of binary search (both variations) into the testing program of Project P1 of Section 7.2 (page 277). Compare the performance with the nonrecursive versions of binary search.*

Answer The following replace the non-recursive versions of binary search. Recursive binary search, first version:

```
Error_code recursive_binary_1(const Ordered_list &the_list,
           const Key &target, int bottom, int top, int &position)
/*
Pre:  The indices bottom to top define the
      range in the list to search for the target.
Post: If a Record in the range of locations
      from bottom to top in the_list has key equal
      to target, then position locates
      one such entry and a code of success is returned.
      Otherwise, the Error_code of not_present is returned
      and position becomes undefined.
Uses: recursive_binary_1 and methods of the classes
List and Record.
*/
```

```
{
   Record data;
   if (bottom < top) {              // List has more than one entry.
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data < target)            // Reduce to top half of list.
        return recursive_binary_1(the_list, target, mid + 1, top, position);
      else                          // Reduce to bottom half of list.
        return recursive_binary_1(the_list, target, bottom, mid, position);
   }
   else if (top < bottom)
      return not_present;           // List is empty.
   else {                           // List has exactly one entry.
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
Error_code run_recursive_binary_1(const Ordered_list &the_list,
                               const Key &target, int &position)
{
   return recursive_binary_1(the_list, target, 0,
                             the_list.size() - 1, position);
}
```

Recursive binary search, second version:

```
Error_code recursive_binary_2(const Ordered_list &the_list,
           const Key &target, int bottom, int top, int &position)
/*
Pre:  The indices bottom to top define the
      range in the list to search for the target.
Post: If a Record in the range
      from bottom to top in the_list  has key equal
      to target, then position locates
      one such entry, and a code of success is returned.
      Otherwise, not_present is returned, and position is undefined.
Uses: recursive_binary_2, together with methods from the classes
      Ordered_list and Record.
*/
{
  Record data;
  if (bottom <= top) {
    int mid = (bottom + top) / 2;
    the_list.retrieve(mid, data);
    if (data == target) {
       position = mid;
       return success;
    }
    else if (data < target)
      return recursive_binary_2(the_list, target, mid+1, top, position);
    else
      return recursive_binary_2(the_list, target, bottom, mid-1, position);
  }
  else return not_present;
}
```

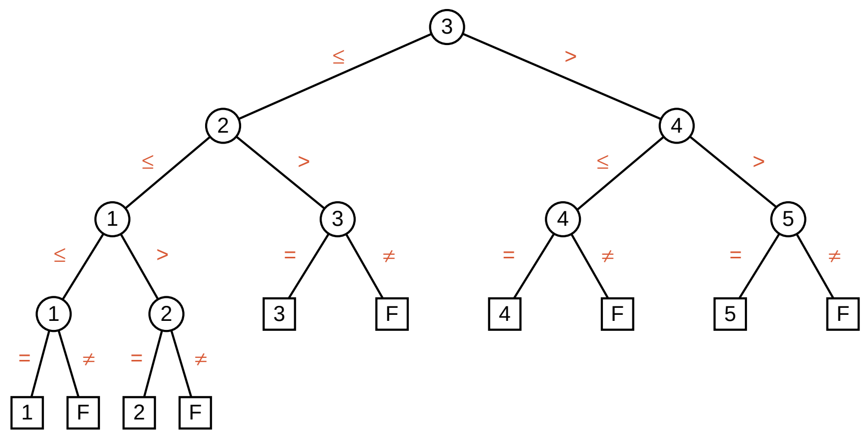
```
Error_code run_recursive_binary_2(const Ordered_list &the_list,
                                  const Key &target, int &position)
{
   return recursive_binary_2(the_list, target, 0,
                             the_list.size() - 1, position);
}
```

7.4 COMPARISON TREES

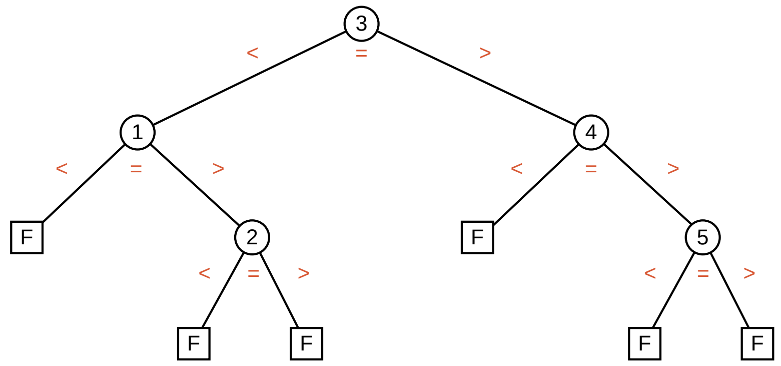
Exercises 7.4

E1. *Draw the comparison trees for (i)* binary_search_1 *and (ii)* binary_search_2 *when* **(a)** $n = 5$, **(b)** $n = 7$, **(c)** $n = 8$, **(d)** $n = 13$. *Calculate the external and internal path lengths for each of these trees, and verify that the conclusion of Theorem 7.3 holds.*

Answer **(a)**



$n = 5$
Binary 1

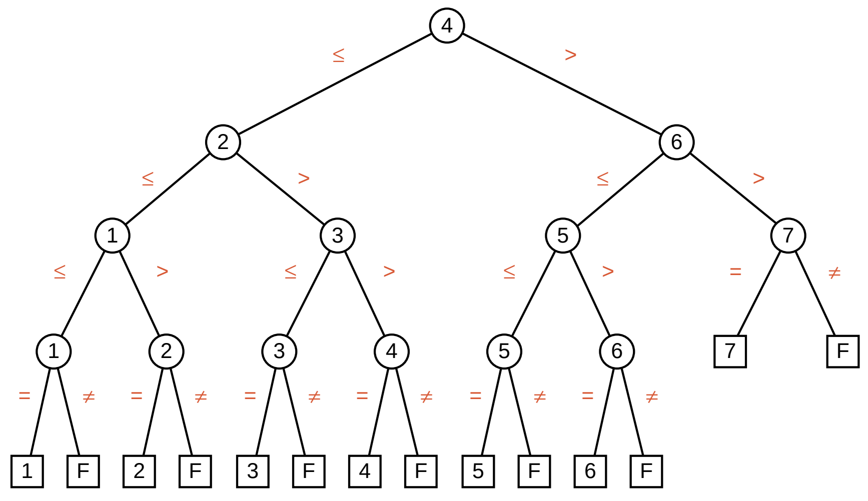


Binary 2

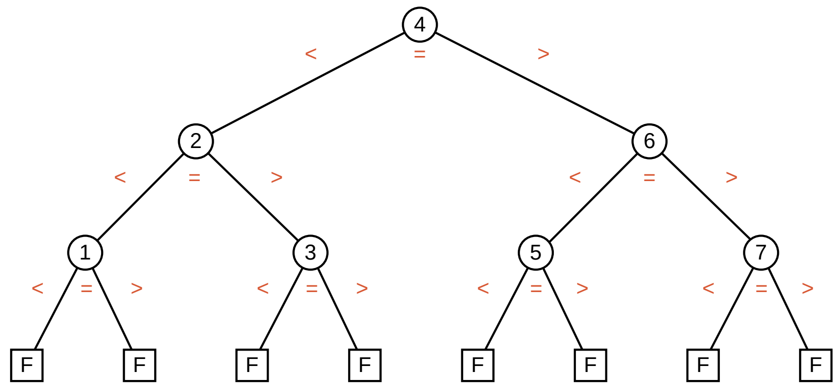
For binary_search_1 we have $E = 34$, $I = 16$, $q = 9$, and $34 = 16 + 2 \times 9$. For binary_search_2 we have $E = 16$, $I = 6$, $q = 5$, and $16 = 6 + 2 \times 5$.

TOC
Index
Help

(b)



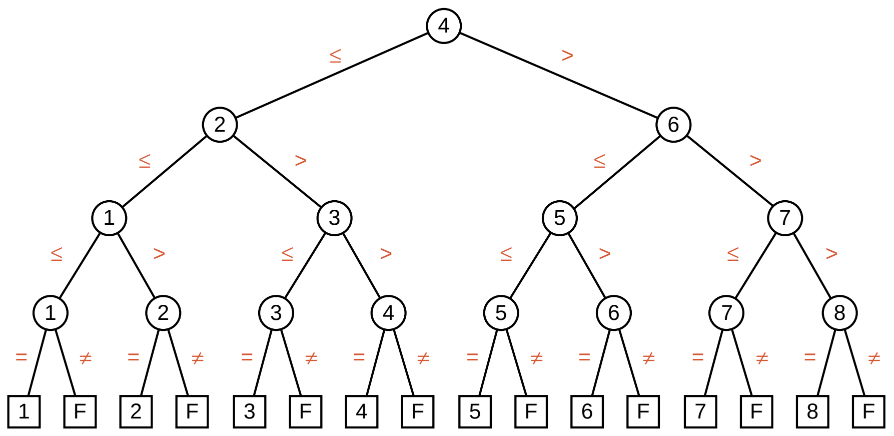
$n = 7$
Binary 1



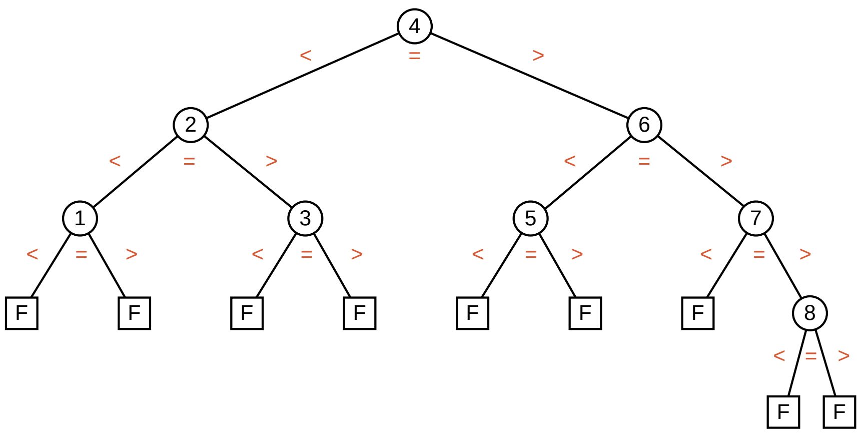
Binary 2

For binary_search_1 we have $E = 54$, $I = 28$, $q = 13$, and $54 = 28 + 2 \times 13$. For binary_search_2 we have $E = 24$, $I = 10$, $q = 7$, and $24 = 10 + 2 \times 7$.

(c)



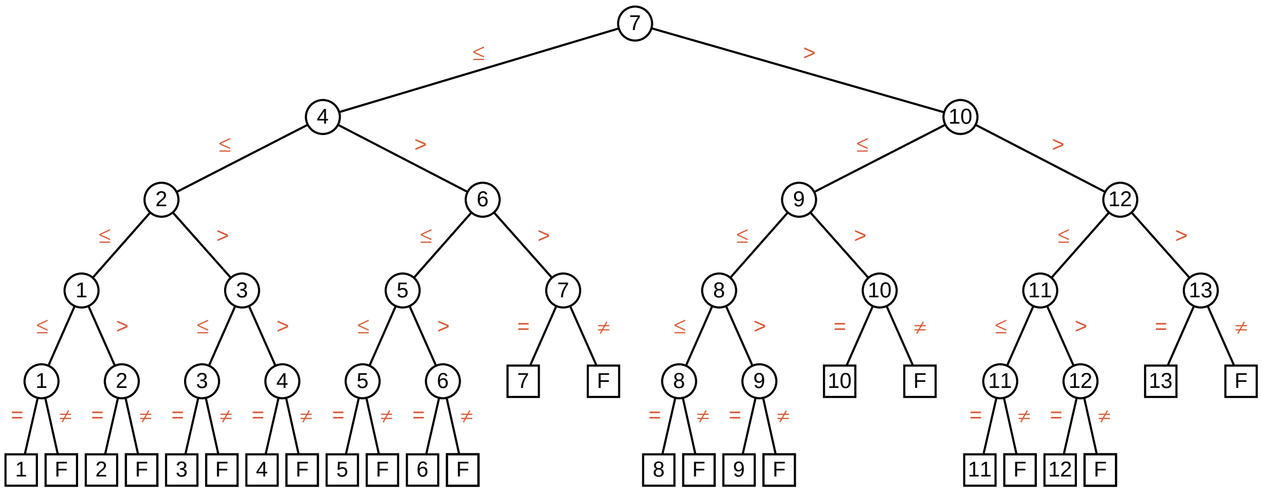
$n = 8$
Binary 1



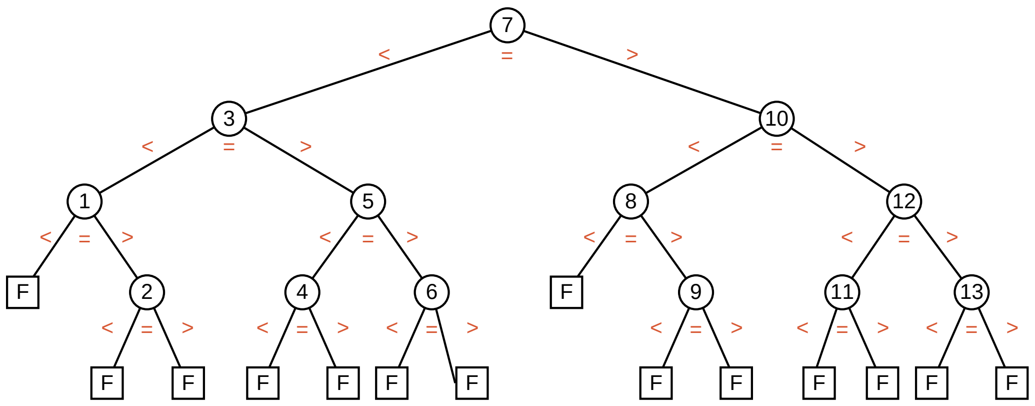
Binary 2

For binary_search_1 we have $E = 64$, $I = 34$, $q = 15$, and $64 = 34 + 2 \times 15$. For binary_search_2 we have $E = 29$, $I = 13$, $q = 8$, and $29 = 13 + 2 \times 8$.

(d)



n = 13
Binary 1



Binary 2

For binary_search_1 we have $E = 124$, $I = 74$, $q = 25$, and $124 = 74 + 2 \times 25$. For binary_search_2 we have $E = 54$, $I = 28$, $q = 13$, and $54 = 28 + 2 \times 13$.

TOC
Index
Help

E2. *Sequential search has less overhead than binary search, and so may run faster for small n. Find the break-even point where the same number of comparisons of keys is made between* sequential_search *and* binary_search_1*. Compute in terms of the formulas for the number of comparisons done in the average successful search.*

Answer

	Sequential Search	*Binary Search*
	$\frac{1}{2}(n+1)$	$\lg n + 1$
$n = 1$	1	1
$n = 2$	1.5	2
$n = 3$	2	2.6
$n = 4$	2.5	3
$n = 5$	3	3.3
$n = 6$	3.5	3.6
$n = 7$	4	3.8

From $n = 7$ on, binary search (version 1) does fewer comparisons of keys.

E3. *Suppose that you have a list of 10,000 names in alphabetical order in an array and you must frequently look for various names. It turns out that 20 percent of the names account for 80 percent of the retrievals. Instead of doing a binary search over all 10,000 names every time, consider the possibility of splitting the list into two: a high-frequency list of 2000 names and a low-frequency list of the remaining 8000 names. To look up a name, you will first use binary search on the high-frequency list, and 80 percent of the time you will not need to go on to the second stage, where you use binary search on the low-frequency list. Is this scheme worth the effort? Justify your answer by finding the number of comparisons done by* binary_search_1 *for the average successful search, both in the new scheme and in a binary search of a single list of 10,000 names.*

Answer A binary search on a single table of 10,000 names (using binary_search_1) takes about $\lg 10000 + 1 \approx 14.3$ comparisons. A search on 2000 names takes about $\lg 2000 + 1 \approx 12.0$ comparisons, and about $\lg 8000 + 1 \approx 14.0$ comparisons with 8000 names. The separate table method takes about 12.0 comparisons for 80% of all access and $12.0 + 14.0 = 26.0$ comparisons for the other 20% of accesses. Combining these comparisons with their probabilities gives an average number of comparisons of about $0.80 \times 12.0 + 0.20 \times 26.0 = 14.8$. Using one larger table hence provides a faster average access than using separate tables.

E4. *Use mathematical induction on n to prove that, in the comparison tree for* binary_search_1 *on a list of n entries, $n > 0$, all the leaves are on levels $\lfloor \lg 2n \rfloor$ and $\lceil \lg 2n \rceil$. [Hence, if n is a power of 2 (so that $\lg 2n$ is an integer), then all the leaves are on one level; otherwise, they are all on two adjacent levels.]*

Answer For $n = 1$, the loop in binary_search_1 does not iterate, and only one comparison of keys is done checking for equality of the unique entry in the list with the target. Hence the comparison tree consists of the root and its two children, which are both leaves on level 1. Hence the result is correct for $n = 1$.

For $n > 1$, the first key comparison in binary_search_1 divides the list into two sublists of sizes $r = \lceil n/2 \rceil$ and $s = \lfloor n/2 \rfloor$, and the comparison tree for n consists of the root together with two comparison (sub)trees for binary_search_1 on lists of lengths r and s. By induction hypothesis, the first of these has its leaves on levels $\lfloor \lg 2r \rfloor$ and $\lceil \lg 2r \rceil$, and the second has its leaves on levels $\lfloor \lg 2s \rfloor$ and $\lceil \lg 2s \rceil$. To evaluate these four expressions for levels, we consider two cases.

If n is even, then $r = s = n/2$, and the four expressions for levels reduce to $\lfloor \lg n \rfloor$ and $\lceil \lg n \rceil$.

If n is odd, then $2r = n + 1$ and $2s = n - 1$, and again all four expressions for levels reduce to $\lfloor \lg n \rfloor$ and $\lceil \lg n \rceil$.

Finally, the levels are increased by 1 to account for the comparison of keys at the root of the whole tree. Hence all leaves are on levels $\lfloor \lg n \rfloor + 1 = \lfloor \lg 2n \rfloor$ and $\lceil \lg n \rceil + 1 = \lceil \lg 2n \rceil$, which was to be proved.

E5. *If you modified binary search so that it divided the list not essentially in half at each pass, but instead into two pieces of sizes about one-third and two-thirds of the remaining list, then what would be the approximate effect on its average count of comparisons?*

Answer Let $f(n)$ be the average number of key comparisons done for a list of length n. If we apply the method to a list of length $3n$, it will first do one comparison, and then, with probability $\frac{1}{3}$, it will search for the target in the first third of the list, which will require $f(n)$ comparisons. With probability $\frac{2}{3}$ it will search the latter part of the list, requiring $f(2n)$ comparisons. We hence have the equation

$$f(3n) = 1 + \tfrac{1}{3}f(n) + \tfrac{2}{3}f(2n).$$

Let us try to find a solution of the same form as that for binary search, $f(n) = a\lg n + b$, where we must find a and b. Substituting this expression into the equation gives

$$a\lg 3n + b = 1 + \tfrac{1}{3}(a\lg n + b) + \tfrac{2}{3}(a\lg 2n + b),$$

which simplifies to $a\lg 3 = 1 + \frac{2}{3}a$, from which we obtain $a = 1/(\lg 3 - \frac{2}{3}) \approx 1.0890$. Since we have $f(1) = 1$ we get $b = 1$, so that the average number of key comparisons is

$$f(n) \approx 1.0890\lg n + 1.$$

Hence this method does about 8.9 percent more comparisons than ordinary binary search.

Programming Projects 7.4

P1. **(a)** *Write a "ternary" search function analogous to* binary_search_2 *that examines the key one-third of the way through the list, and if the target key is greater, then examines the key two-thirds of the way through, and thus in any case at each pass reduces the length of the list by a factor of three.* **(b)** *Include your function as an additional option in the testing program of Project P1 of Section 7.2 (page 277), and compare its performance with other methods.*

Answer A driver that implements the menu options for Projects P1 and P2 is provided by:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../2/key.h"
#include "../2/key.cpp"
typedef Key Record;

#include "../3/ordlist.h"
#include "../3/ordlist.cpp"
#include "../3/binary1.cpp"
#include "../3/binary2.cpp"
#include "../3/rbinary1.cpp"
#include "../3/rbinary2.cpp"
#include "ternary.cpp"
#include "sequ.cpp"
#include "hybrid.cpp"
#include "test.cpp"
```

```
void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit  [F]ill list \n"
        << "[0] test sequential search\n"
        << "test binary search version [1]\n"
        << "test binary search version [2]\n"
        << "test recursive binary search version [O]ne\n"
        << "test recursive binary search version [T]wo\n"
        << "test h[Y]brid search\n"
        << endl;
}

void intro()
{
   cout << "Testing program for search methods."
        << endl;
}

main()
/*
Post: Binary search has been tested for a user
      selected list size, with the function test_search
*/

{
   intro();
   help();
   int n, searches = 100;

   char command = ' ';
   Ordered_list the_list;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, 0, 1, 2, 3, O, T, Y: " << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           help();
         break;

         case '0': case '1': case '2': case '3': case 'o':
         case 't': case 'y':
           test_search(searches, the_list, command);
         break;

         case 'T': case 'O': case 'Y':
           test_search(searches, the_list, command + 'a' - 'A');
         break;

         case 'f': case 'F':
           the_list.clear();
           cout << "Enter an upper bound n for the size of list entries:"
                << flush;
           cin  >> n;
           for (int i = 1; i <= n; i += 2)
              if (the_list.insert(the_list.size(), i) != success)
                  cout << " Overflow in filling list." << endl;
         break;
      }
   }
}
```

This program uses a testing function:

```
void print_out(char *comment, float t, int comp_ct, int searches)
{
   float average;
   cout << comment << " Search Statistics: " << endl;
   cout << " Time for " << searches << " comparisons was " << t << endl;
   average = (( float ) comp_ct) / (( float ) searches);
   cout << " Average number of comparisons per search was " << average
        << endl;
}

void test_search(int searches, Ordered_list &the_list, char method)
/*
Pre:  None.
Post: The number of key comparisons and CPU time for a searching method
      have been calculated.
Uses: Methods of the classes Ordered_list, Random, Timer,
      and an output function print_out.
*/

{
   int list_size = the_list.size();
   if (searches <= 0 || list_size < 0) {
      cout << " Exiting test: " << endl
           << " The number of searches must be positive." << endl
           << " The number of list entries must exceed 0." << endl;
      return;
   }

   int i, target, found_at;
   Random number;
   Timer clock;

   switch (method) {
   case '0':
      cout << "\n\nSequential Search" << endl;
      Key::comparisons = 0;
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size - 1) + 1;
         if (sequential_search(the_list, target, found_at) == not_present)
            cout << "Error: Failed to find expected target " << target
                 << endl;
      }
      print_out("Successful", clock.elapsed_time(),
                               Key::comparisons, searches);

      Key::comparisons = 0;
      clock.reset();
      for (i = 0; i < searches; i++) {
         target = 2 * number.random_integer(0, list_size);
         if (sequential_search(the_list, target, found_at) == success)
            cout << "Error: Found unexpected target " << target
                 << " at " << found_at << endl;
      }
      print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
           searches);
   break;
```

```
case 'y':
   cout << "\n\nHybrid Search" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (hybrid(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                           Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (hybrid(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
       searches);
break;

case '3':
   cout << "\n\nTernary Search" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (ternary(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                           Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (ternary(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
       searches);
break;
```

```
case 'o':
   cout << "\n\nRecursive Binary Search 1" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (run_recursive_binary_1(the_list, target, found_at) ==
                                                        not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                        Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (run_recursive_binary_1(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                        searches);
break;

case '1':
   cout << "\n\nNon-recursive Binary Search 1" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (binary_search_1(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                        Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (binary_search_1(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
       searches);
break;
```

```
      case 't':
         cout << "\n\nRecursive Binary Search 2" << endl;
         Key::comparisons = 0;
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size - 1) + 1;
            if (run_recursive_binary_2(the_list, target, found_at) ==
                                                          not_present)
               cout << "Error: Failed to find expected target " << target
                    << endl;
         }
         print_out("Successful", clock.elapsed_time(),
                                 Key::comparisons, searches);

         Key::comparisons = 0;
         clock.reset();
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size);
            if (run_recursive_binary_2(the_list, target, found_at) == success)
               cout << "Error: Found unexpected target " << target
                    << " at " << found_at << endl;
         }
         print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                                            searches);
      break;

      case '2':
         cout << "\n\nNon-recursive Binary Search 2" << endl;
         Key::comparisons = 0;
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size - 1) + 1;
            if (binary_search_2(the_list, target, found_at) == not_present)
               cout << "Error: Failed to find expected target " << target
                    << endl;
         }
         print_out("Successful", clock.elapsed_time(),
                                 Key::comparisons, searches);

         Key::comparisons = 0;
         clock.reset();
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size);
            if (binary_search_2(the_list, target, found_at) == success)
               cout << "Error: Found unexpected target " << target
                    << " at " << found_at << endl;
         }
         print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                   searches);
      break;
      }
   }
```

An implementation for ternary search is:

```
Error_code ternary(const Ordered_list &the_list,
                           const Key &target, int &position)
/*
Post: If a Record in the_list has key equal
      to target, then position locates
      one such entry and a code of success is returned.
      Otherwise, not_present is returned and position is
      undefined.
Uses: Methods for classes Ordered_list and Record.
*/

{
   Record data;
   int bottom = 0, top = the_list.size() - 1;

   while (bottom < top) {
      int low_mid = (2 * bottom + top) / 3;
      the_list.retrieve(low_mid, data);
      if (data < target) {
         int high_mid = (bottom + 2 * top) / 3;
         the_list.retrieve(high_mid, data);
         if (data < target) bottom = high_mid + 1;
         else {
            bottom = low_mid + 1;
            top = high_mid;
         }
      }
      else
         top = low_mid;
   }

   if (top < bottom) return not_present;
   else {
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
```

Code for other classes and functions has appeared with earlier projects.

P2. **(a)** *Write a program that will do a "hybrid" search, using* binary_search_1 *for large lists and switching to sequential search when the search is reduced to a sufficiently small sublist. (Because of different overhead, the best switch-over point is not necessarily the same as your answer to Exercise E2.)* **(b)** *Include your function as an additional option in the testing program of Project P1 of Section 7.2 (page 277), and compare its performance to other methods.*

Answer An implementation for hybrid search is:

```
Error_code hybrid(const Ordered_list &the_list,
                             const Key &target, int &position)
/*
Post: If an entry in the_list has key equal to target, then return
      success and the output parameter position locates such an entry
      within the list.
      Otherwise return not_present and position becomes invalid.
*/
{
   int s = the_list.size();
   if (s <= 6) return sequential_search(the_list, target, position);
   else return binary_search_1(the_list, target, position);
}
```

The main program and other code is the same as Project P1.

7.5 LOWER BOUNDS

Exercise 7.5

E1. *Suppose that, like* binary_search_2*, a search algorithm makes three-way comparisons. Let each internal node of its comparison tree correspond to a successful search and each leaf to an unsuccessful search.*

(a) *Use Lemma 7.5 to obtain a theorem like Theorem 7.6 giving lower bounds for worst and average case behavior for an unsuccessful search by such an algorithm.*

Answer Suppose that an algorithm uses three-way comparisons of keys to search for a target in a list. If there are k different failure nodes, then in its worst case the algorithm must make at least $\lceil \lg k \rceil$ three-way comparisons of keys, and in its average case, it must make at least $\lg k$ three-way comparisons of keys.

(b) *Use Theorem 7.4 to obtain a similar result for successful searches.*

Answer Denote the external path length of a 2-tree by E, the internal path length by I, and let q be the number of successful searches. Then:

$$E = I + 2q \text{ and } I = E - 2q.$$

Note that in a 2-tree $E = I + 1$, $q = k - 1$, and $E \ge k \lg k$. Therefore, the average number of three-way comparisons is

$$\frac{I}{q} = \frac{E}{q} - 2 \ge \frac{q+1}{q} \lg(q+1) - 2 > \lg q - 2.$$

(c) *Compare the bounds you obtain with the analysis of* binary_search_2.

Answer By comparing bounds with the analysis of binary_search_2 it can be shown that binary_search_2 is essentially optimal in its class of algorithms.

Programming Project 7.5

P1. **(a)** *Write a program to do interpolation search and verify its correctness (especially termination). See the references at the end of the chapter for suggestions and program analysis.* **(b)** *Include your function as another option in the testing program of Project P1 of Section 7.2 (page 277) and compare its performance with the other methods.*

Answer An implementation of interpolation search is:

```
Error_code interpolation(const Ordered_list &the_list,
                         const Key &target, int &position)
/*
Post: If a Record in the_list   has key equal
      to target, then position locates
      one such entry and a code of success is returned.
      Otherwise, not_present is returned and position is
      undefined.
Uses: Methods for classes Ordered_list and Record.
*/
```

```
{
   Record data, data_b, data_t;

   int bottom = 0, top = the_list.size() - 1;
   the_list.retrieve(bottom, data_b);
   the_list.retrieve(top, data_t);

   while (bottom <= top) {
      double position_ratio =
       ((double) (target.the_key() - data_b.the_key())) /
         ((double) (data_t.the_key() - data_b.the_key()));
      double estimate = ((double) (top - bottom)) * position_ratio +
                           ((double) bottom);
      position = (int) (0.5 + estimate);
      if (position < bottom) position = bottom;
      if (position > top) position = top;
      the_list.retrieve(position, data);
      if (data == target) return success;
      if (data < target) {
         bottom = position + 1;
         the_list.retrieve(bottom, data_b);
      }
      else {
         top = position - 1;
         the_list.retrieve(top, data_t);
      }
   }
   return not_present;
}
```

A menu-driven driver is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../2/key.h"
#include "../2/key.cpp"
typedef Key Record;

#include "../3/ordlist.h"
#include "../3/ordlist.cpp"
#include "../3/binary1.cpp"
#include "../3/binary2.cpp"
#include "../3/rbinary1.cpp"
#include "../3/rbinary2.cpp"
#include "../4p1p2/ternary.cpp"
#include "../4p1p2/sequ.cpp"
#include "../4p1p2/hybrid.cpp"
#include "interp.cpp"
#include "test.cpp"
```

```
void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit  [F]ill list \n"
        << "[0] test sequential search\n"
        << "test binary search version [1]\n"
        << "test binary search version [2]\n"
        << "test [I]nterpolation search \n"
        << "test recursive binary search version [O]ne\n"
        << "test recursive binary search version [T]wo\n"
        << "test h[Y]brid search\n"
        << endl;
}

void intro()
{
   cout << "Testing program for search methods."
        << endl;
   help();
}

main()
/*
Post: Binary search has been tested for a user
      selected list size, with the function test_search
*/

{
   intro();
   int n, searches = 100;

   char command = ' ';
   Ordered_list the_list;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, 0, 1, 2, 3, I, O, T, Y: "
           << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           help();
         break;

         case '0': case '1': case '2': case '3': case 'o': case 't':
         case 'y': case 'i':
           test_search(searches, the_list, command);
         break;

         case 'T': case 'O': case 'Y': case 'I':
           test_search(searches, the_list, command + 'a' - 'A');
         break;
```

```
          case 'f': case 'F':
            the_list.clear();
            cout << "Enter an upper bound n for the size of list entries:"
                 << flush;
            cin  >> n;
            for (int i = 1; i <= n; i += 2)
               if (the_list.insert(the_list.size(), i) != success)
                   cout << " Overflow in filling list." << endl;
          break;
       }
    }
}
```

A file with auxiliary functions is:

```
void print_out(char *comment, float t, int comp_ct, int searches)
{
   float average;
   cout << comment << " Search Statistics: " << endl;
   cout << " Time for " << searches << " comparisons was " << t << endl;
   average = (( float ) comp_ct) / (( float ) searches);
   cout << " Average number of comparisons per search was " << average
        << endl;
}

void test_search(int searches, Ordered_list &the_list, char method)
/*
Pre:   None.
Post: The number of key comparisons and CPU time for a searching method
      have been calculated.
Uses: Methods of the classes Ordered_list, Random, Timer,
      and an output function print_out.
*/

{
  int list_size = the_list.size();
  if (searches <= 0 || list_size < 0) {
     cout << " Exiting test: " << endl
          << " The number of searches must be positive." << endl
          << " The number of list entries must exceed 0." << endl;
     return;
  }

  int i, target, found_at;
  Random number;
  Timer clock;

  switch (method) {
  case 'i':
     cout << "\n\nInterpolation Search" << endl;
     Key::comparisons = 0;
     for (i = 0; i < searches; i++) {
        target = 2 * number.random_integer(0, list_size - 1) + 1;
        if (interpolation(the_list, target, found_at) == not_present)
           cout << "Error: Failed to find expected target " << target
                << endl;
     }
     print_out("Successful", clock.elapsed_time(),
                            Key::comparisons, searches);
```

```
         Key::comparisons = 0;
         clock.reset();
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size);
            if (interpolation(the_list, target, found_at) == success)
               cout << "Error: Found unexpected target " << target
                    << " at " << found_at << endl;
         }
         print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
             searches);
      break;

      case '0':
         cout << "\n\nSequential Search" << endl;
         Key::comparisons = 0;
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size - 1) + 1;
            if (sequential_search(the_list, target, found_at) == not_present)
               cout << "Error: Failed to find expected target " << target
                    << endl;
         }
         print_out("Successful", clock.elapsed_time(),
                                 Key::comparisons, searches);

         Key::comparisons = 0;
         clock.reset();
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size);
            if (sequential_search(the_list, target, found_at) == success)
               cout << "Error: Found unexpected target " << target
                    << " at " << found_at << endl;
         }
         print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
             searches);
      break;

      case 'y':
         cout << "\n\nHybrid Search" << endl;
         Key::comparisons = 0;
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size - 1) + 1;
            if (hybrid(the_list, target, found_at) == not_present)
               cout << "Error: Failed to find expected target " << target
                    << endl;
         }
         print_out("Successful", clock.elapsed_time(),
                   Key::comparisons, searches);

         Key::comparisons = 0;
         clock.reset();
         for (i = 0; i < searches; i++) {
            target = 2 * number.random_integer(0, list_size);
            if (hybrid(the_list, target, found_at) == success)
               cout << "Error: Found unexpected target " << target
                    << " at " << found_at << endl;
         }
         print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
             searches);
      break;
```

```
case '3':
   cout << "\n\nTernary Search" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (ternary(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                            Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (ternary(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
        searches);
break;

case 'o':
   cout << "\n\nRecursive Binary Search 1" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (run_recursive_binary_1(the_list, target, found_at) ==
                                                        not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                            Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (run_recursive_binary_1(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                            searches);
break;
```

```
case '1':
   cout << "\n\nNon-recursive Binary Search 1" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (binary_search_1(the_list, target, found_at) == not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
                        Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (binary_search_1(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
       searches);
break;

case 't':
   cout << "\n\nRecursive Binary Search 2" << endl;
   Key::comparisons = 0;
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size - 1) + 1;
      if (run_recursive_binary_2(the_list, target, found_at) ==
                                                     not_present)
         cout << "Error: Failed to find expected target " << target
              << endl;
   }
   print_out("Successful", clock.elapsed_time(),
            Key::comparisons, searches);

   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (run_recursive_binary_2(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
                                 searches);
break;
```

```
  case '2':
     cout << "\n\nNon-recursive Binary Search 2" << endl;
     Key::comparisons = 0;
     for (i = 0; i < searches; i++) {
        target = 2 * number.random_integer(0, list_size - 1) + 1;
        if (binary_search_2(the_list, target, found_at) == not_present)
           cout << "Error: Failed to find expected target " << target
                << endl;
     }
     print_out("Successful", clock.elapsed_time(),
                        Key::comparisons, searches);

     Key::comparisons = 0;
     clock.reset();
     for (i = 0; i < searches; i++) {
        target = 2 * number.random_integer(0, list_size);
        if (binary_search_2(the_list, target, found_at) == success)
           cout << "Error: Found unexpected target " << target
                << " at " << found_at << endl;
     }
     print_out("Unsuccessful", clock.elapsed_time(), Key::comparisons,
             searches);
  break;
  }
}
```

The interpolation-search function was developed directly from the function binary_search_1 with the only change being the method of assigning mid in the while loop. As was shown in the text, the while loop will terminate provided bottom $\le$ mid $<$ top is always true, and the **if** statement after the calculation of mid forces this invariant to be true. Hence the algorithm will terminate after a finite number of iterations.

To verify that interpolation search converges to the target (if present) we study the following relation, which is used to calculate mid:

$$\frac{\text{mid} - \text{bottom}}{\text{top} - \text{bottom}} = \frac{\text{target} - \text{entry[bottom]}}{\text{entry[top]} - \text{entry[bottom]}}.$$

Note that if the keys were evenly distributed this relation would locate the target immediately. If not, then additional iterations will occur. Assume that target is initially within the range of entry[bottom] to entry[top], that is, entry[bottom] $\le$ target $\le$ entry[top]. Since target $\ge$ entry[bottom], the right hand side of the displayed equation is nonnegative, which implies that mid $\ge$ bottom. Similarly, mid $\le$ top, since target $\le$ entry[top], so the right hand side of the displayed equation does not exceed 1. If, however, target is outside the specified range, or if target = entry[top], mid will not satisfy these criteria and therefore a statement must be introduced to force mid within the range. This statement makes the function terminate for an unsuccessful search. In case target = entry[top], the assignment bottom : = mid + 1 will be repeated until the loop terminates, whereupon the target is compared with the top entry, as it should be.

7.6 ASYMPTOTICS

Exercises 7.6

E1. *For each of the following pairs of functions, find the smallest integer value of $n > 1$ for which the first becomes larger than the second. [For some of these, use a calculator to try various values of n.]*

(a) n^2 *and* $15n + 5$

Answer $n = 16: \quad n^2 = 256, \quad 15n + 5 = 245.$

(b) 2^n *and* $8n^4$

Answer $n = 21:$ $2^n = 2097152,$ $8n^4 = 1555848.$

(c) $0.1n$ *and* $10 \lg n$

Answer $n = 997:$ $0.1n = 99.70,$ $10 \lg n \approx 99.61.$

(d) $0.1n^2$ *and* $100n \lg n$

Answer $n = 13747:$ $0.1n^2 = 18898000.09,$ $100n \lg n \approx 18897766.09$

E2. *Arrange the following functions into increasing order; that is, $f(n)$ should come before $g(n)$ in your list if and only if $f(n)$ is $O(g(n))$.*

100000	$(\lg n)^3$	2^n
$n \lg n$	$n^3 - 100n^2$	$n + \lg n$
$\lg \lg n$	$n^{0.1}$	n^2

Answer The functions arranged in increasing order are:

1. 100000	**4.** $n^{0.1}$	**7.** n^2
2. $\lg \lg n$	**5.** $n + \lg n$	**8.** $n^3 - 100n^2$
3. $(\lg n)^3$	**6.** $n \lg n$	**9.** 2^n

E3. *Divide the following functions into classes so that two functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n)$ is $\Theta(g(n))$. Arrange the classes from the lowest order of magnitude to the highest. [A function may be in a class by itself, or there may be several functions in the same class.]*

5000	$(\lg n)^5$	3^n
$\lg n$	$n + \lg n$	n^3
$n^2 \lg n$	$n^2 - 100n$	$4n + \sqrt{n}$
$\lg \lg n^2$	$n^{0.3}$	n^2
$\lg n^2$	$\sqrt{n^2 + 4}$	2^n

Answer We group the functions in increasing order of magnitude on the following rows. (Functions of the same order of magnitude are listed on a row together.)

5000
$\lg \lg n^2$
$\lg n,\ \lg n^2$
$(\lg n)^5$
$n^{0.3}$
$n + \lg n,\ \sqrt{n^2+4},\ 4n + \sqrt{n}$
$n^2 - 100n,\ n^2$
$n^2 \lg n$
n^3
2^n
3^n

E4. *Show that each of the following is correct.*

(a) $3n^2 - 10n \lg n - 300$ *is* $\Theta(n^2)$.

Answer We have:

$$\lim_{n\to\infty} \frac{3n^2 - 10n \lg n - 300}{n^2} = \lim_{n\to\infty} \left(3 - \frac{10 \lg n}{n} - \frac{300}{n^2}\right) = 3 - 0 - 0 = 3$$

(b) $4n \lg n + 100n - \sqrt{n+5}$ *is* $\Omega(n)$.

Answer We have:

$$\lim_{n\to\infty} \frac{4n\lg n + 100n - \sqrt{n+5}}{n} = \lim_{n\to\infty}\left(4\lg n + 100 - \sqrt{\frac{n+5}{n^2}}\right) = \infty + 100 - 0 = \infty$$

(c) $4n \lg n + 100n - \sqrt{n+5}$ *is* $o(\sqrt{n^3})$.

Answer We have:

$$\lim_{n\to\infty} \frac{4n\lg n + 100n - \sqrt{n+5}}{\sqrt{n^3}} = \lim_{n\to\infty}\left(\frac{4\lg n}{\sqrt{n}} + \frac{100}{\sqrt{n}} - \sqrt{\frac{n+5}{n^3}}\right) = 0 + 0 - 0 = 0$$

(d) $(n-5)(n + \lg n + \sqrt{n})$ *is* $O(n^2)$.

Answer We have:

$$\lim_{n\to\infty} \frac{(n-5)(n+\lg n+\sqrt{n})}{n^2} = \lim_{n\to\infty}\left(\frac{n-5}{n} + \frac{(n-5)\lg n}{n^2} + \sqrt{\frac{n}{n^4}}\right) = 1 + 0 + 0 = 1$$

(e) $\sqrt{n^2 + 5n + 12}$ *is* $\Theta(n)$.

Answer We have:

$$\lim_{n\to\infty} \frac{\sqrt{n^2+5n+12}}{n} = \lim_{n\to\infty}\sqrt{\frac{n^2+5n+12}{n^2}} = \sqrt{1+0+0} = 1$$

E5. *Decide whether each of the following is correct or not.*

(a) $(3\lg n)^3 - 10\sqrt{n} + 2n$ *is* $O(n)$.
(b) $(3\lg n)^3 - 10\sqrt{n} + 2n$ *is* $\Omega(\sqrt{n})$.
(c) $(3\lg n)^3 - 10\sqrt{n} + 2n$ *is* $o(n \log n)$.
(d) $\sqrt{n^2 - 10n + 100}$ *is* $\Omega(n)$.
(e) $3n - 10\sqrt{n} + \sqrt{n \lg n}$ *is* $O(n)$.
(f) $2^n - n^3$ *is* $\Omega(n^4)$.
(g) $\sqrt{3n - 12n\lg n - 2n^3 + n^4}$ *is* $\Theta(n^2)$.
(h) $(n+10)^3$ *is* $O((n-10)^3)$.

Answer All of the statements are correct.

E6. *Suppose you have programs whose running times in microseconds for an input of size n are $1000\lg n$, $100n$, $10n^2$, and 2^n. Find the largest size n of input that can be processed by each of these programs in* **(a)** *one second,* **(b)** *one minute,* **(c)** *one day, and* **(d)** *one year.*

Answer For a program with running time $1000\lg n$ we can process inputs of the following sizes in the indicated time periods: **(a)** 2, **(b)** 2^{60}, **(c)** 2^{86400}, and **(d)** $2^{31536000}$.

For a program with running time $100n$ we can process inputs of the following sizes in the indicated time periods: **(a)** 10, **(b)** 600, **(c)** 864000, and **(d)** 315360000.

For a program with running time $10n^2$ we can process inputs of the following sizes in the indicated time periods: **(a)** 10, **(b)** 77, **(c)** 2939, and **(d)** 56156.

For a program with running time 2^n we can process inputs of the following sizes in the indicated time periods: **(a)** 9, **(b)** 15, **(c)** 26, and **(d)** 34.

E7. *Prove that a function $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$*

Answer The function $f(n)$ is $\Theta(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is both finite and nonzero. This is the case if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is nonzero. In other words, this is the case if and only if $f(n)$ is both $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

E8. *Suppose that $f(n)$ is $\Theta(g(n))$ and $h(n)$ is $o(g(n))$. Prove that $f(n)+h(n)$ is $\Theta(g(n))$.*

Answer We have:

$$\lim_{n\to\infty} \frac{f(n)+h(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)} + \frac{h(n)}{g(n)}\right) = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)} + 0\right) = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right)$$

Hence, $\lim_{n\to\infty} \frac{f(n)+h(n)}{g(n)}$ is finite and nonzero.

E9. *Find functions $f(n)$ and $h(n)$ that are both $\Theta(n)$ but $f(n)+h(n)$ is not $\Theta(n)$.*

Answer Take $f(n) = n$ and $h(n) = -n$.

E10. *Suppose that $f(n)$ is $O(g(n))$ and $h(n)$ is $O(g(n))$. Prove that $f(n)+h(n)$ is $O(g(n))$.*

Answer We have:

$$\lim_{n\to\infty} \frac{f(n)+h(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)} + \frac{h(n)}{g(n)}\right)$$

Hence, $\lim_{n\to\infty} \frac{f(n)+h(n)}{g(n)}$ is finite.

E11. *Show that the relation O is transitive; that is, from the assumption that $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, prove that $f(n)$ is $O(h(n))$. Are any of the other relations o, Θ, and Ω transitive? If so, which one(s)?*

Answer We have:

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) \lim_{n\to\infty} \left(\frac{g(n)}{h(n)}\right)$$

If both limits on the right hand side are finite, then so is the left hand limit, and in this case, we deduce that $f(n)$ is $O(h(n))$. All of the relations o, Θ, and Ω are transitive.

E12. *Show that the relation Θ is symmetric; that is, from the assumption that $f(n)$ is $\Theta(g(n))$ prove that $g(n)$ is $\Theta(f(n))$. Are any of the other relations o, O, and Ω symmetric? If so, which one(s)?*

Answer We have:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{\lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right)}$$

If the limit on the right hand side is finite and nonzero, then so is the left hand limit, and in this case, we deduce that $g(n)$ is $\Theta(f(n))$. None of the other relations is symmetric.

E13. *Show that the relation Ω is reflexive; that is, prove that any function $f(n)$ is $\Omega(f(n))$. Are any of the other relations o, O, and Θ reflexive? If so, which one(s)?*

Answer We have:

$$\lim_{n\to\infty} \frac{f(n)}{f(n)} = 1$$

Since the value on the right hand side is nonzero, we deduce that $f(n)$ is $\Omega(f(n))$. The relations O and Θ are also reflexive, but o is not.

E14. *A relation is called an **equivalence relation** if it is reflexive, symmetric, and transitive. On the basis of the three preceding exercises, which (if any) of o, O, Θ, and Ω is an equivalence relation?*

Answer Only the relation Θ is an equivalence relation?

E15. *Suppose you are evaluating computer programs by running them without seeing the source code. You run each program for several sizes n of input and obtain the operation counts or times shown. In each case, on the basis of the numbers given, find a constant c and a function $g(n)$ (which should be one of the seven common functions shown in Figure 7.10) so that $cg(n)$ closely approximates the given numbers.*

(a)

n:	*10*	*50*	*200*	*1000*
count:	*201*	*998*	*4005*	*19987*

(b)

n:	*10*	*100*	*1000*	*10000*
count:	*3*	*6*	*12*	*24*

(c)

n:	*10*	*20*	*40*	*80*
count:	*10*	*40*	*158*	*602*

(d)

n:	*10*	*11*	*12*	*13*
count:	*3*	*6*	*12*	*24*

Answer Functions that closely approximate the given sets of data are:

(a) $20n$ **(b)** $\frac{3}{2}2^{\log_{10} n} = 1.5 \times n^{\log_{10} 2}$ **(c)** $n^2/10$ **(d)** $\frac{3}{1024}2^n$.

E16. *Two functions $f(n)$ and $g(n)$ are called* ***asymptotically equal*** *if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$

in which case we write $f(n) \asymp g(n)$. Show each of the following:

(a) $3x^2 - 12x + 7 \asymp 3x^2$.

Answer We have:

$$\lim_{n\to\infty} \frac{3n^2 - 12n + 7}{3n^2} = \lim_{n\to\infty} \left(1 - \frac{4}{n} + \frac{7}{3n^2}\right) = 1 - 0 + 0 = 1$$

(b) $\sqrt{x^2+1} \asymp x$.

Answer We have:

$$\lim_{n\to\infty} \frac{\sqrt{n^2+1}}{n} = \sqrt{\lim_{n\to\infty} \frac{n^2+1}{n^2}} = 1$$

(c) *If $f(n) \asymp g(n)$ then $f(n)$ is $\Theta(g(n))$.*

Answer If $f(n) \asymp g(n)$ then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. In particular, the limit is finite so that $f(n)$ is $\Theta(g(n))$.

(d) *If $f(n) \asymp g(n)$ and $g(n) \asymp h(n)$, then $f(n) \asymp h(n)$.*

Answer We have:

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = \lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) \lim_{n\to\infty} \left(\frac{g(n)}{h(n)}\right)$$

If both limits on the right hand side have value 1, then so does the left hand limit, and in this case, we have $f(n) \asymp h(n)$.

TOC
Index
Help

Programming Project 7.6

P1. *Write a program to test on your computer how long it takes to do* $n \lg n$, n^2, n^5, 2^n, *and* $n!$ *additions for* $n = 5, 10, 15, 20$.

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
#include <math.h>

void test(long n)
/*
Test: time how long it takes to perform n additions
*/

{
    long i;
    int result = 0;

    cout << n << " iterations" << endl;
    Timer clock;
    for (i = 0; i < n; i++)
        result = result + result;

    cout << "Elapsed time: " << clock.elapsed_time() << "seconds"
         << endl;
}

long factorial(int n)
/*
factorial: calculate n!
*/

{
    int i;
    long p = 1;

    for (i = 2; i <= n; i++)
        p = p * i;

    return p;
}

int main()
/*
test how long it take to perform different number of additions
*/

{
    int runs, n;
    long value;     /* number of additions to calculate */

    /* execute for n = 5, 10, 15, 20    */
    for (runs = 1; runs < 5; runs++) {
        n = runs * 5;
        cout << "Running test with n = " << n << endl;

        cout << "n lg n - ";
        value = (long) (n * log((double) n) / log((double) 2));
        test(value);
```

```
            cout << "n ^ 2 - ";
            value = n * n;
            test(value);

            cout << "2 ^ n - ";
            value = (long) pow((double)2, (double)n);
            test(value);

            cout << "n ^ 5 - ";
            value = (long) pow((double)n, (double)5);
            test(value);

            cout << "n! - ";
            value = factorial(n);
            test(value);
        }
    }
```

REVIEW QUESTIONS

1. *Name three conditions under which sequential search of a list is preferable to binary search.*

Under the conditions **(1)** where the list is not arranged in a sorted order, **(2)** where the list contains very few items, or **(3)** where the list is in linked implementation, it is preferable to use sequential search and not binary search.

2. *In searching a list of n items, how many comparisons of keys are done, on average, by* **(a)** sequential_search, **(b)** binary_search_1, *and* **(c)** binary_search_2*?*

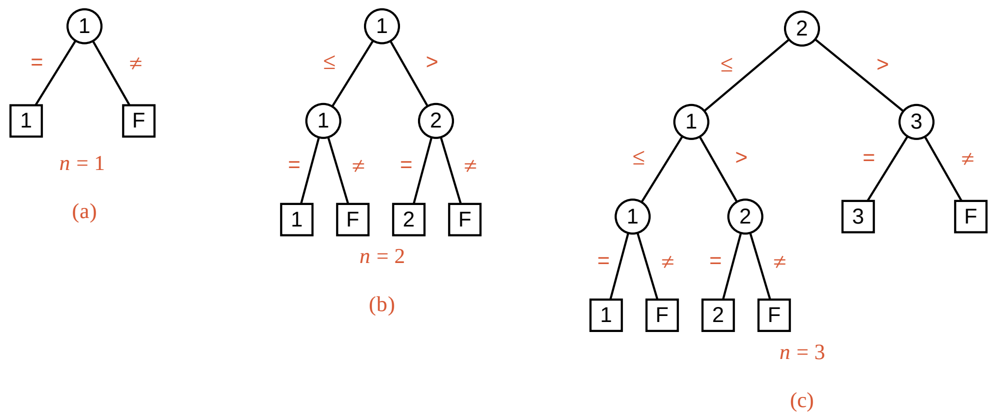
The average successful sequential_search requires approximately $\frac{1}{2}(n+1)$ comparisons and $n+1$ comparisons for an unsuccessful search. An application of binary_search_1 requires $\lg n + 1$ comparisons of the average search, whether the search is successful or not. An application of binary_search_2 requires $2\lg n - 3$ comparisons of the average successful search and $2\lg n$ comparisons for an unsuccessful search.

	Successful search	*Unsuccessful search*
sequential_search	$\frac{1}{2}(n+1)$	$n+1$
binary_search_1	$\lg n + 1$	$\lg n + 1$
binary_search_2	$2\lg n - 3$	$2\lg n$

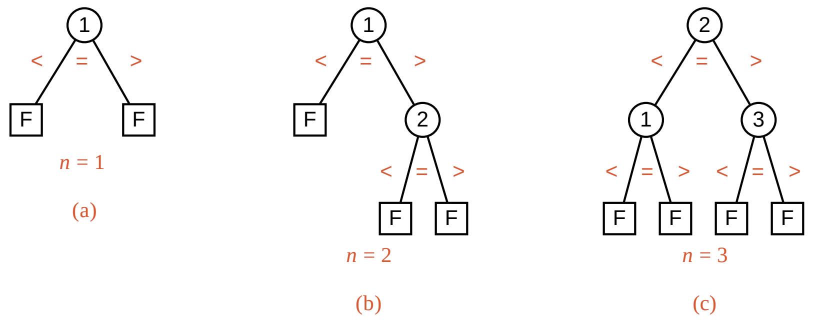
3. *Why was binary search implemented only for contiguous lists, not for linked lists?*

Binary search works by dividing the list into sublists until a sublist of size zero or one is created. This requires access to the center of the sublist at each iteration, an action very easy to do with contiguous lists but requires far too much work with linked lists which are sequentially accessed.

4. *Draw the comparison tree for* binary_search_1 *for searching a list of length* **(a)** *1,* **(b)** *2, and* **(c)** *3.*



5. *Draw the comparison tree for* binary_search_2 *for searching a list of length* **(a)** *1,* **(b)** *2, and* **(c)** *3.*



6. *If the height of a 2-tree is 3, what are* **(a)** *the largest and* **(b)** *the smallest number of vertices that can be in the tree?*

   A 2-tree of height 3 can contain a maximum of 7 and a minimum of 5 vertices.

7. *Define the terms internal and external path length of a 2-tree. State the path length theorem.*

   The internal path length of a 2-tree is the sum, over all vertices that are not leaves, of the number of branches from the root to the vertex. The external path length of a 2-tree is the sum of the number of branches traversed in going from the root once to every leaf in the tree. The path length theorem states that where the external path length of a 2-tree is denoted by E, the internal path length by I, the number of vertices that are not leaves by q, then $E = I + 2q$.

8. *What is the smallest number of comparisons that any method relying on comparisons of keys must make, on average, in searching a list of n items?*

   Any searching algorithm that uses a comparison of keys to search for a target in a list will have an average of at least $\lg n$ comparisons to make with each search.

9. *If* binary_search_2 *does 20 comparisons for the average successful search, then about how many will it do for the average unsuccessful search, assuming that the possibilities of the target less than the smallest key, between any pair of keys, or larger than the largest key are all equally likely?*

   binary_search_2 does an average of $2\lg n - 3$ comparisons for the average successful search and $2\lg n$ comparisons for the average unsuccessful search. Therefore, if the average successful search requires 20 comparisons, the average unsuccessful search would require 23 comparisons.

10. *What is the purpose of the big-O notation?*

    The purpose of the big-O notation is to suppress minor details of operation costs and produce algorithm analyses valid for large cases.

TOC
Index
Help

Sorting 8

8.2 INSERTION SORT

Exercises 8.2

E1. *By hand, trace through the steps insertion sort will use on each of the following lists. In each case, count the number of comparisons that will be made and the number of times an entry will be moved.*

(a) *The following three words to be sorted alphabetically:*

triangle square pentagon

Answer

triangle	square	pentagon
square	triangle	square
pentagon	pentagon	triangle

There are 5 moves in total and 3 comparisons.

(b) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in increasing order*

Answer In this case, there would be no moves made and 2 comparisons.

(c) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in decreasing order*

Answer

triangle (3)	square (4)	pentagon (5)
square (4)	triangle (3)	square (4)
pentagon (5)	pentagon (5)	triangle (3)

There are 5 moves in total and 3 comparisons.

(d) *The following seven numbers to be sorted into increasing order:*

26 33 35 29 19 12 22

Answer

26	26	26	26	19	12	12
$\boxed{33}$	33	33	29	26	19	19
35	$\boxed{35}$	35	33	29	26	22
29	29	$\boxed{29}$	35	33	29	26
19	19	19	$\boxed{19}$	35	33	29
12	12	12	12	$\boxed{12}$	35	33
22	22	22	22	22	$\boxed{22}$	35

In the above table, the entries in each column shown in $\boxed{\text{boxes}}$ are the ones currently being processed by insertion sort. There are a total of 19 entries moved and 19 comparisons made.

(e) *The same seven numbers in a different initial order, again to be sorted into increasing order:*

12 19 33 26 29 35 22

Answer

12	12	12	12	12	12	12
$\boxed{19}$	19	19	19	19	19	19
33	$\boxed{33}$	33	26	26	26	22
26	26	$\boxed{26}$	33	29	29	26
29	29	29	$\boxed{29}$	33	33	29
35	35	35	35	$\boxed{35}$	35	33
22	22	22	22	22	$\boxed{22}$	35

There are a total of 9 moves and 12 comparisons in this case.

(f) *The following list of 14 names to be sorted into alphabetical order:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

Tim	Dot	Dot	Dot	Dot	Dot	Dot	Amy	Amy	Amy	Amy	Amy	Amy	Amy
$\boxed{\text{Dot}}$	Tim	Eva	Eva	Eva	Eva	Eva	Dot	Dot	Ann	Ann	Ann	Ann	Ann
Eva	$\boxed{\text{Eva}}$	Tim	Roy	Roy	Kim	Guy	Eva	Eva	Dot	Dot	Dot	Dot	Dot
Roy	Roy	$\boxed{\text{Roy}}$	Tim	Tim	Roy	Kim	Guy	Guy	Eva	Eva	Eva	Eva	Eva
Tom	Tom	Tom	$\boxed{\text{Tom}}$	Tom	Tim	Roy	Kim	Jon	Guy	Guy	Guy	Guy	Guy
Kim	Kim	Kim	Kim	$\boxed{\text{Kim}}$	Tom	Tim	Roy	Kim	Jon	Jim	Jim	Jim	Jan
Guy	Guy	Guy	Guy	Guy	$\boxed{\text{Guy}}$	Tom	Tim	Roy	Kim	Jon	Jon	Jon	Jim
Amy	Amy	Amy	Amy	Amy	Amy	$\boxed{\text{Amy}}$	Tom	Tim	Roy	Kim	Kay	Kay	Jon
Jon	Jon	Jon	Jon	Jon	Jon	Jon	$\boxed{\text{Jon}}$	Tom	Tim	Roy	Kim	Kim	Kay
Ann	Ann	Ann	Ann	Ann	Ann	Ann	Ann	$\boxed{\text{Ann}}$	Tom	Tim	Roy	Ron	Kim
Jim	Jim	Jim	Jim	Jim	Jim	Jim	Jim	Jim	$\boxed{\text{Jim}}$	Tom	Tim	Roy	Ron
Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	$\boxed{\text{Kay}}$	Tom	Tim	Roy
Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	$\boxed{\text{Ron}}$	Tom	Tim
Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	$\boxed{\text{Jan}}$	Tom

In the diagram above, the entries in each column shown in $\boxed{\text{boxes}}$ are the ones currently being processed by the insertion sort function. In this example, there will be a total of 61 comparisons and 73 assignments made.

E2. *What initial order for a list of keys will produce the worst case for insertion sort in the contiguous version? In the linked version?*

Answer The worst case for the contiguous version of insertion sort is when the keys are input in reversed order. In the linked version, the worse case is when the list is in sorted order.

E3. *How many key comparisons and entry assignments does contiguous insertion sort make in its worst case?*

Answer As stated in the solution to Exercise E2, the worst case for contiguous insertion sort is when the list is given in reversed order. This would require $k-1$ comparisons and $k+1$ assignments for the k^{th} entry in the list, with n keys being checked, giving a worst case comparison count of

$$\sum_{k=2}^{n}(k-1) = \frac{1}{2}(n-1)n.$$

Counting each key moved gives a total assignment count in this case of $\frac{1}{2}(n-1)n$.

E4. *Modify the linked version of insertion sort so that a list that is already sorted, or nearly so, will be processed rapidly.*

Answer The linked insertion sort function may be used by reversing the key comparisons from $<$ to $>$ and vice versa. This will produce a list sorted into reverse order. The list can then be reversed in one more pass to produce the proper order.

TOC
Index
Help

Programming Projects 8.2

P1. *Write a program that can be used to test and evaluate the performance of insertion sort and, later, other methods. The following outline should be used.*

test program for sorting

(a) *Create several files of integers to be used to test sorting methods. Make files of several sizes, for example, sizes 20, 200, and 2000. Make files that are in order, in reverse order, in random order, and partially in order. By keeping all this test data in files (rather than generating it with random numbers each time the testing program is run), the same data can be used to test different sorting methods, and hence it will be easier to compare their performance.*

(b) *Write a menu-driven program for testing various sorting methods. One option is to read a file of integers into a list. Other options will be to run one of various sorting methods on the list, to print the unsorted or the sorted list, and to quit. After the list is sorted and (perhaps) printed, it should be discarded so that testing a later sorting method will start with the same input data. This can be done either by copying the unsorted list to a second list and sorting that one, or by arranging the program so that it reads the data file again before each time it starts sorting.*

(c) *Include code in the program to calculate and print (1) the CPU time, (2) the number of comparisons of keys, and (3) the number of assignments of list entries during sorting a list. Counting comparisons can be achieved, as in Section 7.2, by overloading the comparison operators for the class* Key *so that they increment a counter. In a similar way we can overload the assignment operator for the class* Record *to keep a count of assignments of entries.*

Answer A Class Key that keeps track of key assignments and comparisons is defined in:

```
class Key {
   int key;
public:
   static int comparisons;
   static int assignments;
   static void initialize();
   static int counter();
   Key ( int x = 0 );
   int the_key() const;
   Key &operator =(const Key &y);
};
bool operator ==(const Key &y,const Key &x);
bool operator !=(const Key &y,const Key &x);
bool operator >=(const Key &y,const Key &x);
bool operator <=(const Key &y,const Key &x);
bool operator >(const Key &y,const Key &x);
bool operator <(const Key &y,const Key &x);
```

Implementation:

```
int Key::comparisons = 0;
int Key::assignments = 0;
void Key::initialize()
{
   comparisons = 0;
}
int Key::counter()
{
   return comparisons;
}
Key::Key (int x)
{
   key = x;
}
Key &Key::operator =(const Key &x)
{
   Key::assignments++;
   key = x.key;
   return *this;
}
bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}
bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}
bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}
```

```
bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}

bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}

int Key::the_key() const
{
   return key;
}
```

(d) *Use the contiguous list package as developed in Section 6.2.2, include the contiguous version of insertion sort, and assemble statistics on the performance of contiguous insertion sort for later comparison with other methods.*

Answer Class specifications:

```
template <class Record>
class Sortable_list: public List<Record> {
public:
   void insertion_sort();
   void insertion_binary();
   void selection_sort();
   void shell_sort();
   void quick_sort();
   void heap_sort();
   void scan_sort();
   void bubble_sort();

private:
   void sort_interval(int start, int increment);
   void swap(int low, int high);
   int max_key(int low, int high);
   int partition(int low, int high);
   void recursive_quick_sort(int low, int high);
   void insert_heap(const Record &current, int low, int high);
   void build_heap();

public:
   void merge_sort();  // Section 7, Project P4

private:
   void merge(int low, int high);
   void recursive_merge_sort(int low, int high);

public:
   void quick_sort(int offset);  // Section 8, Project P3b

private:
   void recursive_quick_sort(int low, int high, int offset);
public:
   void hybrid(int offset);  // Section 8, Project P3a

private:
   void recursive_hybrid(int low, int high, int offset);
   void insertion_sort(int low, int high);
};
```

Implementation:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include   "../../6/contlist/list.h"
#include   "../../6/contlist/list.cpp"
#include "key.h"
#include "key.cpp"
#include "../contlist/record.h"
#include "../contlist/record.cpp"

void write_entry(Record &c)
{
   cout << ((Key) c).the_key() << " ";
}

#include "cont.h"   // sortable list specification
#include "../contlist/insert.cpp"
#include "../contlist/selctcon.cpp"
#include "../contlist/shell.cpp"
#include "../contlist/quickcon.cpp"
#include "../contlist/heap.cpp"
#include "ins_p2.cpp"
#include "scan.cpp"
#include "bubble.cpp"
#include "../7p4/merge.cpp"
#include "../8p3/hybrid_a.cpp"
#include "../8p3/hybrid_b.cpp"

void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit  (re)[F]ill list \n"
        << "write [D]ata  write sorted [O]utput \n"

        << "[0] insertion sort --- Project 2P1d\n"
        << "[1] selection sort --- Project 3P1\n"
        << "[2] shell sort     --- Project 4P2\n"
        << "[3] quick sort\n"
        << "[4] heap sort\n"
        << "[5] insertion, with binary search --- project 2P2\n"
        << "[6] scan sort   --- Project 2P3\n"
        << "[7] bubble sort --- Project 2P4\n"
        << "[8] merge sort  --- Project 7P4\n"
        << "[9] hybrid sort, many small insert sorts  --- Project 8P3b\n"
        << "[b] hybrid sort, one insert sort          --- Project 8P3b\n"
        << endl;
}
```

```
void intro()
{
   cout << "Testing program for sorting methods for a contiguous list."
        << endl;
   help ();
}

main()
{
   List<Record> s; List<Record> t = s;  //  help out a poor old compiler
   intro();

   int n;
   Random dice;
   Error_code report;
   Record target;
   Sortable_list<Record> the_list;
   Sortable_list<Record> copy_list;
   char command = ' ';

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, O, D, "
           << "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, b: "
           << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           help();
         break;

         case 'd': case 'D':
            cout << "\nUnsorted list \n";
            the_list.traverse(write_entry);
            cout << endl;
         break;

         case 'o': case 'O':
            cout << "\nLast sorted list \n";
            copy_list.traverse(write_entry);
            cout << endl;
         break;

         case '0': case '1': case '2': case '3': case '4': case '5':
         case '6': case '7': case '8': case '9': case 'b': case 'B': {
           copy_list = the_list;
           Key::comparisons = 0;
           Key::assignments = 0;

           Timer clock;
           switch (command) {
              case '0':
              cout << "Insertion Sort ";
              copy_list.insertion_sort();
            break;

              case '1':
              cout << "Selection Sort ";
              copy_list.selection_sort();
            break;
              case '2':
              cout << "     Shell Sort ";
              copy_list.shell_sort();
            break;

              case '3':
              cout << "     Quick Sort ";
              copy_list.quick_sort();
            break;

              case '4':
              cout << "      Heap Sort ";
              copy_list.heap_sort();
            break;

              case '5':
              cout << "      Insertion Sort with bianry search ";
              copy_list.insertion_binary();
            break;

              case '6':
              cout << "      Scan Sort ";
              copy_list.scan_sort();
            break;

              case '7':
              cout << "      Bubble Sort ";
              copy_list.bubble_sort();
            break;

              case '8':
              cout << "      Merge Sort ";
              copy_list.merge_sort();
            break;

              case '9':
              cout << "      Hybrid Sort ";
              cout << "At what size list do you want to switch method: "
                   << flush;
              cin  >> n;
              copy_list.hybrid(n);
            break;

              case 'b': case 'B':
              cout << "      Hybrid Sort ";
              cout << "At what size list do you want to switch method: "
                   << flush;
              cin  >> n;
              copy_list.quick_sort(n);
            break;

           }
           cout << "Time: " << clock.elapsed_time() << " seconds.\n"
                << "Comparisons: " << Key::comparisons << "\n"
                << "Assignments: " << Key::assignments
                << endl;
         }
         break;
```

```
         case 'f': case 'F':
           the_list.clear();
           cout << "How many list entries would you like? "
                << flush;
           cin  >> n;
           for (int i = 0; i < n; i++) {
              target = dice.random_integer(0, 10 * n);
              report = the_list.insert(i, target);
              if (report == overflow) {
                 cout << "Available list space filled up at " << i
                      << " entries." << endl;
                 break;
              }
              if (report != success) i--;
           }
         break;
      }  // end of outer switch statement
   }     // end of outer while statement
}
```

(e) *Use the linked list package as developed in Section 6.2.3, include the linked version of insertion sort, assemble its performance statistics, and compare them with contiguous insertion sort. Why is the count of entry assignments of little interest for this version?*

Answer Class specifications:

```
template <class Record>
class Sortable_list:public List<Record> {
public:   //  Add prototypes for sorting methods here.

   void insertion_sort();
   void merge_sort();
   void selection_sort();

private:
   Node<Record> *divide_from(Node<Record> *sub_list);
   Node<Record> *merge(Node<Record> *first, Node<Record> *second);
   void recursive_merge_sort(Node<Record> *&sub_list);
   void swap(int low, int high);
   int max_key(int low, int high);

public:
   void merge7P2();  // Sec 7 Project P2

private:
   Node<Record> *divide7P2(Node<Record> *sub_list, int l1, int l2);
   Node<Record> *merge7P2(Node<Record> *first, Node<Record> *second);
   void recursive_merge7P2(Node<Record> *&sub_list, int length);

public:
   void merge_sort7P3a();  // Sec 7 Project P3a

private:
   Node<Record> *divide_from7P3a(Node<Record> *sub_list);
   Node<Record> *merge7P3a(Node<Record> *first, Node<Record> *second);

public:
   void merge_sort7P3b();  // Sec 7 Project P3b

private:
   Node<Record> *divide_from7P3b(Node<Record> *sub_list, int &length);
   Node<Record> *merge7P3b(Node<Record> *first, Node<Record> *second);
```

TOC
Index
Help

```
public:                      // Sec 8 Project P2
   void quick_sort();

private:
   void recursive_quick_sort(Node<Record> *&first_node,
                                  Node<Record> *&last_node);
   void partition(Node<Record> *sub_list,
        Node<Record> *&first_small, Node<Record> *&last_small,
         Node<Record> *&first_large, Node<Record> *&last_large);
};
```

Implementation:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"

#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include "key.h"
#include "key.cpp"
#include "../contlist/record.h"
#include "../contlist/record.cpp"

void write_entry(Record &c)
{
   cout << ((Key) c).the_key() << " ";
}

#include "link.h"  // sortable list specification
#include "../linklist/insert.cpp"
#include "../linklist/merge.cpp"
#include "../3p1p2/select.cpp"
#include "../7p1p2/merge.cpp"
#include "../7p3/merge_a.cpp"
#include "../7p3/merge_b.cpp"
#include "../8p1p2/quick.cpp"

void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit  (re)[F]ill list \n"
        << "write [D]ata  write sorted [O]utput \n"

        << "[0] insertion sort --- Project 2P1e\n"
        << "[1] merge sort     --- Project 7P1\n"
        << "[2] selection sort --- Project 3P2\n"
        << "[3] modified merge sort  --- Project 7P2\n"
        << "[4] natural merge sort   --- Project 7P3a\n"
        << "[5] natural merge sort   --- Project 7P3b\n"
        << "[6] quick sort           --- Project 8P2\n"
        << endl;
}
```

```
void intro()
{
   cout << "Testing program for sorting methods for a linked list."
        << endl;
   help ();
}

main()
{
   List<Record> s;
   List<Record> t = s;  t = s; //  Help out a poor old compiler.
   intro();

   int n;
   Random dice;
   Error_code report;
   Record target;
   Sortable_list<Record> the_list;
   Sortable_list<Record> copy_list;
   char command = ' ';

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, O, D, 0, 1, 2, 3, 4, 5, 6: "
           << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           help();
         break;

         case 'd': case 'D':
            cout << "\nUnsorted list \n";
            the_list.traverse(write_entry);
            cout << endl;
         break;

         case 'o': case 'O':
            cout << "\nLast sorted list \n";
            copy_list.traverse(write_entry);
            cout << endl;
         break;

         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6':  {
           copy_list = the_list;
           Key::comparisons = 0;
           Key::assignments = 0;
           Timer clock;
           switch (command) {
              case '0':
              cout << "Insertion Sort ";
              copy_list.insertion_sort();
            break;

              case '1':
              cout << "Merge Sort ";
              copy_list.merge_sort();
            break;
```

```
                  case '2':
                  cout << "Linked Selection Sort ";
                  copy_list.selection_sort();
                break;

                  case '3':
                  cout << "Modified Merge Sort ";
                  copy_list.merge7P2();
                break;

                  case '4':
                  cout << "Natural Merge Sort Project 3a ";
                  copy_list.merge_sort7P3a();
                break;

                  case '5':
                  cout << "Natural Merge Sort Project 3b ";
                  copy_list.merge_sort7P3b();
                break;

                  case '6':
                  cout << "Quick Sort Project ";
                  copy_list.quick_sort();
                break;
              }
              cout << "Time: " << clock.elapsed_time() << " seconds.\n"
                   << "Comparisons: " << Key::comparisons << "\n"
                   << "Assignments: " << Key::assignments
                   << endl;
            }
            break;

            case 'f': case 'F':
              the_list.clear();
              cout << "How many list entries would you like? "
                   << flush;
              cin  >> n;
              for (int i = 0; i < n; i++) {
                 target = dice.random_integer(0, 10 * n);
                 report = the_list.insert(i, target);
                 if (report == overflow) {
                    cout << "Available list space filled up at " << i
                         << " entries." << endl;
                    break;
                 }
                 if (report != success) i--;
              }
            break;
         }  // end of outer switch statement
      }     // end of outer while statement
}
```

Refer to the comparison tables in Project P6 of Section 8.10 for timings, key comparisons, and assignments.

binary insertion sort

P2. *Rewrite the contiguous version of the function* insertion_sort *so that it uses binary search to locate where to insert the next entry. Compare the time needed to sort a list with that of the original function* insertion_sort. *Is it reasonable to use binary search in the linked version of* insertion_sort? *Why or why not?*

Answer

```
template <class Record>
void Sortable_list<Record>::insertion_binary()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.
*/

{
   int first_unsorted;  //  position of first unsorted entry
   int position;        //  searches sorted part of list
   Record current;      //  holds the entry temporarily removed from list

   for (first_unsorted = 1; first_unsorted < count; first_unsorted++) {
      current = entry[first_unsorted];
      int low = 0, mid, high = first_unsorted - 1;
      while (high > low) {
         mid = (high + low) / 2;
         if (entry[mid] < current) low = mid + 1;
         else                      high = mid;
      }

      if (entry[high] < current) position = high + 1;
      else                       position = high;
      int index = first_unsorted;
      do {
         entry[index] = entry[index - 1];
         index--;                              //  position is empty.
      } while (index > 0 && index > position);
      entry[position] = current;
   }
}
```

Contrary to intuition, this version actually runs slower than ordinary insertion sort. After the search for the proper position in which to insert the key, you still must move all the entries. It would not be reasonable to use this function with linked lists due to the fact that, as with the binary search itself, random access to the list is required.

P3. *There is an even easier sorting method, which, instead of using two pointers to move through the list, uses only one. We can call it* ***scan sort****, and it proceeds by starting at one end and moving forward, comparing adjacent pairs of keys, until it finds a pair out of order. It then swaps this pair of entries and starts moving the other way, continuing to swap pairs until it finds a pair in the correct order. At this point it knows that it has moved the one entry as far back as necessary, so that the first part of the list is sorted, but, unlike insertion sort, it has forgotten how far forward has been sorted, so it simply reverses direction and sorts forward again, looking for a pair out of order. When it reaches the far end of the list, then it is finished.*

scan sort

(a) *Write a C++ program to implement scan sort for contiguous lists. Your program should use only one position variable (other than the list's* count *member), one variable of type* entry *to be used in making swaps, and no other local variables.*

Answer

```
template <class Record>
void Sortable_list<Record>::scan_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
```

```
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.
*/
{
   int i = 1;
   if (size() >= 2) do {
      if (entry[i] >= entry[i - 1]) i++;
      else {
         swap(i, i - 1);
         if (i > 0) i--;
      }
   } while (i != size());
}
```

(b) *Compare the timings for your program with those of* insertion_sort.

Answer Refer to the comparison tables in Project P6 of Section 8.10.

P4. *A well-known algorithm called* ***bubble sort*** *proceeds by scanning the list from left to right, and whenever a pair of adjacent keys is found to be out of order, then those entries are swapped. In this first pass, the largest key in the list will have "bubbled" to the end, but the earlier keys may still be out of order. Thus the pass scanning for pairs out of order is put in a loop that first makes the scanning pass go all the way to* count*, and at each iteration stops it one position sooner.* **(a)** *Write a C++ function for bubble sort.* **(b)** *Find the performance of bubble sort on various kinds of lists, and compare the results with those for insertion sort.*

bubble sort

Answer

```
template <class Record>
void Sortable_list<Record>::bubble_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.
*/
{
   for (int top = size() - 2; top >= 0; top--)
      for (int i = 0; i <= top; i++)
         if (entry[i] > entry[i + 1]) swap(i, i + 1);
}
```

The function does $1 + 2 + \cdots + (n-1) = \frac{1}{2}(n-1)n$ comparisons of keys and about $\frac{1}{4}n^2$ swaps of entries (See Knuth, vol. 3, pp. 108–110). Refer to the comparison table in Project P6 of Section 8.10 for the figures on test runs of this function.

8.3 SELECTION SORT

Exercises 8.3

E1. *By hand, trace through the steps selection sort will use on each of the following lists. In each case, count the number of comparisons that will be made and the number of times an entry will be moved.*

(a) *The following three words to be sorted alphabetically:*

triangle square pentagon

Answer

triangle	pentagon	pentagon
square	square	square
pentagon	triangle	triangle

There will be three comparisons and one swap.

(b) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in increasing order*

Answer All entries are in their proper positions so no movement will occur. There will be 3 comparisons.

(c) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in decreasing order*

Answer The results will be the same as those in Part (a) of this question.

(d) *The following seven numbers to be sorted into increasing order:*

26 33 35 29 19 12 22

Answer

26	26	26	$\boxed{26}$	19	$\boxed{19}$	12
33	$\boxed{33}$	12	12	12	$\boxed{12}$	19
$\boxed{35}$	22	22	22	$\boxed{22}$	22	22
29	29	$\boxed{29}$	$\boxed{19}$	26	26	26
19	19	$\boxed{19}$	29	29	29	29
12	$\boxed{12}$	33	33	33	33	33
$\boxed{22}$	35	35	35	35	35	35

In this diagram, the two entries in each column shown in $\boxed{\text{boxes}}$ are the ones that will be swapped during that pass of the function. Only one boxed entry in a column means that the entry is currently in the proper position. There will be 21 comparisons and 5 swaps done here.

(e) *The same seven numbers in a different initial order, again to be sorted into increasing order:*

12 19 33 26 29 35 22

Answer

12	12	12	12	12	12	12
19	19	19	19	19	$\boxed{19}$	19
33	$\boxed{33}$	22	22	$\boxed{22}$	22	22
26	26	26	$\boxed{26}$	26	26	26
29	29	$\boxed{29}$	29	29	29	29
$\boxed{35}$	$\boxed{22}$	33	33	33	33	
$\boxed{22}$	35	35	35	35	35	35

(f) *The following list of 14 names to be sorted into alphabetical order:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

Tim	$\boxed{\text{Tim}}$	Ron	$\boxed{\text{Ron}}$	Jim	Jim	Jim	$\boxed{\text{Jim}}$	Guy	$\boxed{\text{Guy}}$	Ann	Ann	$\boxed{\text{Ann}}$	Amy
Dot	Dot	Dot	Dot	Dot	Dot	Dot	Dot	Dot	Dot	Dot	$\boxed{\text{Dot}}$	$\boxed{\text{Amy}}$	Ann
Eva	Eva	Eva	Eva	Eva	Eva	Eva	Eva	Eva	Eva	$\boxed{\text{Eva}}$	$\boxed{\text{Amy}}$	Dot	Dot
Roy	Roy	$\boxed{\text{Roy}}$	Kay	Kay	$\boxed{\text{Kay}}$	$\boxed{\text{Jon}}$	Amy	Amy	Amy	$\boxed{\text{Amy}}$	Eva	Eva	Eva
$\boxed{\text{Tom}}$	Jan	Jan	Jan	Jan	Jan	Jan	Jan	$\boxed{\text{Jan}}$	$\boxed{\text{Ann}}$	Guy	Guy	Guy	Guy
Kim	Kim	Kim	Kim	$\boxed{\text{Kim}}$	Ann	Ann	Ann	$\boxed{\text{Ann}}$	Jan	Jan	Jan	Jan	Jan
Guy	Guy	Guy	Guy	Guy	Guy	Guy	$\boxed{\text{Guy}}$	Jim	Jim	Jim	Jim	Jim	Jim
Amy	Amy	Amy	Amy	Amy	Amy	$\boxed{\text{Amy}}$	Jon	Jon	Jon	Jon	Jon	Jon	Jon
Jon	Jon	Jon	Jon	Jon	$\boxed{\text{Jon}}$	Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay
Ann	Ann	Ann	Ann	$\boxed{\text{Ann}}$	Kim	Kim	Kim	Kim	Kim	Kim	Kim	Kim	Kim
Jim	Jim	Jim	$\boxed{\text{Jim}}$	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron
Kay	Kay	$\boxed{\text{Kay}}$	Roy	Roy	Roy	Roy	Roy	Roy	Roy	Roy	Roy	Roy	Roy
Ron	$\boxed{\text{Ron}}$	Tim	Tim	Tim	Tim	Tim	Tim	Tim	Tim	Tim	Tim	Tim	Tim
$\boxed{\text{Jan}}$	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom	Tom

In this diagram, the two entries in each column shown in $\boxed{\text{boxes}}$ are the ones that will be swapped during that pass of the function. There will be a total of 13 swaps and 91 comparisons done for this list.

E2. *There is a simple algorithm called* ***count sort*** *that begins with an unsorted list and constructs a new, sorted list in a new array, provided we are guaranteed that all the keys in the original list are different from each other. Count sort goes through the list once, and for each record scans the list to count how many records have smaller keys. If c is this count, then the proper position in the sorted list for this key is c. Determine how many comparisons of keys will be done by count sort. Is it a better algorithm than selection sort?*

Answer Count sort will scan through the original list of n entries n times, making the total comparison count n^2. Thus, this algorithm requires about twice as many key comparisons as selection sort.

TOC
Index
Help

Programming Projects 8.3

P1. *Run the test program written in Project P1 of Section 8.2 (page 328), to compare selection sort with insertion sort (contiguous version). Use the same files of test data used with insertion sort.*

Answer For the driver programs, see Project P1 of Section 8.2. For all test results, refer to the table in Project P6 of Section 8.10. Here is contiguous selection sort, from the text.

```
template <class Record>
void Sortable_list<Record>::swap(int low, int high)
/*
Pre:  low and high are valid positions in the Sortable_list.
Post: The entry at position low is swapped with the entry at position high.
Uses: The contiguous List implementation of ?list_ch?.
*/

{
   Record temp;
   temp = entry[low];
   entry[low] = entry[high];
   entry[high] = temp;
}
```

```
template <class Record>
int Sortable_list<Record>::max_key(int low, int high)
/*
Pre:  low and high are valid positions in the Sortable_list and
      low <= high.
Post: The position of the entry between low and high with the largest
      key is returned.
Uses: The class Record.
      The contiguous List implementation of ?list_ch?.
*/

{
   int largest, current;
   largest = low;
   for (current = low + 1; current <= high; current++)
      if (entry[largest] < entry[current])
         largest = current;
   return largest;
}

template <class Record>
void Sortable_list<Record>::selection_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: max_key, swap.
*/

{
   for (int position = count - 1; position > 0; position--) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

P2. *Write and test a linked version of selection sort.*

Answer For the driver programs, see Project P1 of Section 8.2. For all test results, refer to the table in Project P6 of Section 8.10.

```
template <class Record>
void Sortable_list<Record>::swap(int low, int high)
/*
Pre:  low and high are valid positions in the Sortable_list.
Post: The entry at position low is swapped with the entry at position high.
Uses: The linked List implementation of Chapter 6.
*/

{
   Record low_r, high_r;
   retrieve(low, low_r);
   retrieve(high, high_r);
   replace(low, high_r);
   replace(high, low_r);
}
```

```
template <class Record>
int Sortable_list<Record>::max_key(int low, int high)
/*
Pre:  low and high are valid positions in the Sortable_list and
      low <= high.
Post: The position of the entry between low and high with the largest
      key is returned.
Uses: The class Record.
      The linked List implementation of Chapter 6.
*/

{
   int largest, current;
   largest = low;
   Record big, consider;
   retrieve(low, big);
   for (current = low + 1; current <= high; current++) {
      retrieve(current, consider);
      if (big < consider) {
         largest = current;
         big = consider;
      }
   }
   return largest;
}

template <class Record>
void Sortable_list<Record>::selection_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: max_key, swap.
*/

{
   for (int position = count - 1; position > 0; position--) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

8.4 SHELL SORT

Exercises 8.4

E1. *By hand, sort the list of 14 names in the "unsorted" column of Figure 8.7 using Shell sort with increments of* **(a)** *8, 4, 2, 1 and* **(b)** *7, 3, 1. Count the number of comparisons and moves that are made in each case.*

Answer

(a)

Incr.	*8*	*4*	*2*	*1*
Tim	Jon	Jon	Eva	Amy
Dot	Ann	Ann	Amy	Ann
Eva	Eva	Eva	Guy	Dot
Roy	Kay	Amy	Ann	Eva
Tom	Ron	Ron	Jim	Guy
Kim	Jan	Dot	Dot	Jan
Guy	Guy	Guy	Jon	Jim
Amy	Amy	Kay	Jan	Jon
Jon	Tim	Tim	Ron	Kay
Ann	Dot	Jan	Kay	Kim
Jim	Jim	Jim	Tim	Ron
Kay	Roy	Roy	Kim	Roy
Ron	Tom	Tom	Tom	Tim
Jan	Kim	Kim	Roy	Tom

(b)

Incr.	*7*	*3*	*1*
Tim	Amy	Amy	Amy
Dot	Dot	Dot	Ann
Eva	Ann	Ann	Dot
Roy	Jim	Eva	Eva
Tom	Kay	Jan	Guy
Kin	Kim	Jon	Jan
Guy	Guy	Guy	Jim
Amy	Tim	Kay	Jon
Jon	Jon	Kim	Kay
Ann	Eva	Jim	Kim
Jim	Roy	Roy	Ron
Kay	Tom	Tom	Roy
Ron	Ron	Ron	Tim
Jan	Jan	Tim	Tom

For the increments given in part (a), there will be 61 comparisons and 69 assignments made. With the increments in part (b), there will be 48 comparisons and 49 assignments.

E2. *Explain why Shell sort is ill suited for use with linked lists.*

Answer Shell sort requires random access to the list.

Programming Projects 8.4

P1. *Rewrite the method* insertion_sort *to serve as the function* sort_interval *embedded in* shell_sort.

Answer

```
template <class Record>
void Sortable_list<Record>::sort_interval(int start, int increment)
/*
Pre:  start is a valid list position and increment is a valid
      stepping length.
Post: The entries of the Sortable_list have been rearranged so that the
      keys in all the entries are sorted into nondecreasing order based
      on certain positions calculated by start and increment.
Uses: Methods for classes Record and Key.
      The contiguous List implementation of ?list_ch?.
*/

{
   int first_unsorted;    //  position of first unsorted entry
   int place;             //  searches sorted part of list
   Record current;        //  used to swap entries
   for (first_unsorted = start + increment; first_unsorted < count;
                                            first_unsorted += increment)
      if (entry[first_unsorted] < entry[first_unsorted - increment]) {
         place = first_unsorted;
         current = entry[first_unsorted];
                          //  Pull entry[first_unsorted] out of the list.
         do {     //  Shift all previous entries one place down the list
            entry[place] = entry[place - increment];
                                        // until the proper place is found.
            place -= increment;
                         // Position place is now available for insertion.
         } while (place > start && entry[place - increment] > current);
         entry[place] = current;
      }
}

template <class Record>
void Sortable_list<Record>::shell_sort()
/*
Post: The entries of the Sortable_list have been rearranged
      so that the keys in all the entries are sorted into
      nondecreasing order.
Uses: sort_interval
*/

{
   int increment,    //  spacing of entries in sublist
       start;        //  starting point of sublist
   increment = count;

   do {
      increment = increment / 3 + 1;
      for (start = 0; start < increment; start++)
         sort_interval(start, increment);  //  modified insertion sort
   } while (increment > 1);
}
```

P2. *Test* shell_sort *with the program of Project P1 of Section 8.2 (page 328), using the same data files as for insertion sort, and compare the results.*

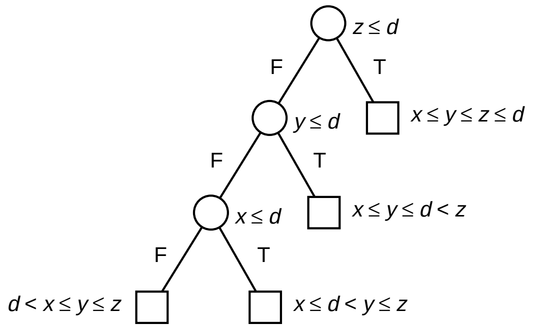
Answer Refer to the table in Project P6 of Section 8.10 for comparative results.

8.5 LOWER BOUNDS

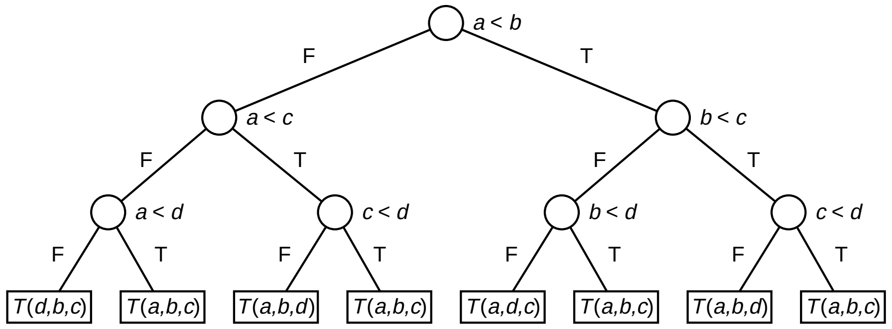
Exercises 8.5

E1. *Draw the comparison trees for* **(a)** *insertion sort and* **(b)** *selection sort applied to four objects.*

Answer **(a)** Take the tree for insertion sort, $n = 3$, from Figure 8.8 and replace each leaf by the following tree, where x, y, and z denote a, b, and c in the increasing order specified for the leaf in Figure 8.8.



(b) Take the following tree and replace each leaf $T(x, y, z)$ by the tree shown for selection sort, $n = 3$, in Figure 8.8, with a, b, c replaced by x, y, z, respectively, and then x, y, z, replaced by a, b, c in the order shown.



E2. **(a)** *Find a sorting method for four keys that is optimal in the sense of doing the smallest possible number of key comparisons in its worst case.* **(b)** *Find how many comparisons your algorithm does in the average case (applied to four keys). Modify your algorithm to make it come as close as possible to achieving the lower bound of* $\lg 4! \approx 4.585$ *key comparisons. Why is it impossible to achieve this lower bound?*

Answer **(a)** The worst-case lower bound for sorting four keys is $\lceil \lg 24 \rceil = 5$. Both insertion and selection sorts (see answer to Exercise E1) do 6 comparisons in the worst case. The following method sorts four keys a, b, c, and d in 5 comparisons in every case.

```
void four_sort(Key &a, Key &b, Key &c, Key &d)
{
  if (a > b) swap(a, b);               //  Now know a ≤ b.
  if (c > d) swap(c, d);               //  Now know c ≤ d
  if (b > d) swap(b, d);               //  Now know that d is largest.
  if (a > c) swap(a, c);               //  Now know that a is smallest.
  if (b > c) swap(b, c)                //  Finally compare the middle keys.
}
```

(b) This function always makes 5 comparisons, significantly more than the average-case lower bound of $\lg 24 \approx 4.585$. It is impossible to achieve this lower bound, since the average number of comparisons for any method is calculated by taking the external path length of its comparison tree (an integer) and dividing by the number of possibilities, $4! = 24$. Hence we must increase to the next multiple of $\frac{1}{24}$, which is $\frac{111}{24} = 4.625$. Even this bound cannot be attained. To see why, note that the first three comparisons done in any sorting method can, together, distinguish at most 8 cases. With 24 cases to start with, after the first three comparisons we still must distinguish among at least $\frac{24}{8} = 3$ cases with further comparisons. The next comparison can distinguish one of these cases, but the other two will require yet one more. Hence distinguishing one of three (equally likely) cases requires $1\frac{2}{3}$ comparisons on average, and the minimum needed to sort four keys is at least $4\frac{2}{3} = \frac{112}{24} \approx 4.667$.

Insertion sort (see the tree in Exercise E1) achieves $\frac{118}{24} \approx 4.917$ on average. The following method achieves the lower bound of $4\frac{2}{3}$ comparisons on average. This algorithm is a special case of one given by L. R. Ford and S. M. Johnson, "A tournament problem," *American Mathematical Monthly* 66 (1959), 387–389. The Ford–Johnson algorithm achieves the best possible worst-case bounds for lists of lengths 12 or less, 20, and 21.

```
void optimal_four_sort(Key &a, Key &b, Key &c, Key &d)
{
  if (a > b) swap(a, b);                  //  Now know that a ≤ b.
  if (c > d) swap(c, d);                  //  Now know that c ≤ d.
  if (b > d) {                            //  swap to make d largest.
    swap(b, d);
    swap(a, c) ;                          //  Maintain a ≤ b and c ≤ d.
  }                                       //  Now know that a ≤ b ≤ d and c ≤ d.
  if (c < b) {                          //  Insert c in the proper place among the first three.
    swap(b, c);                           //  Move b right to make room for c.
    if (c < a) swap(a, c);
  }
}
```

E3. *Suppose that you have a shuffled deck of 52 cards, 13 cards in each of 4 suits, and you wish to sort the deck so that the 4 suits are in order and the 13 cards within each suit are also in order. Which of the following methods is fastest?*

(a) *Go through the deck and remove all the clubs; then sort them separately. Proceed to do the same for the diamonds, the hearts, and the spades.*

Answer This process would require about 247 steps.

(b) *Deal the cards into 13 piles according to the rank of the card. Stack these 13 piles back together and deal into 4 piles according to suit. Stack these back together.*

Answer This process would require 52 steps to deal the cards into the 13 piles, 13 steps to stack the piles in order, 52 steps to deal them into 4 piles, and 3 steps to stack the piles. Thus, this method (similar to the process used by distribution sort, seen later in this chapter) would be the fastest with a total of only 120 steps.

(c) *Make only one pass through the cards, by placing each card in its proper position relative to the previously sorted cards.*

Answer This method is similar to insertion sort, and requires $\frac{1}{4} \times 52^2 = 676$ steps.

Programming Projects 8.5

The sorting projects for this section are specialized methods requiring keys of a particular type, pseudorandom numbers between 0 and 1. Hence they are not intended to work with the testing program devised for other methods, nor to use the same data as the other methods studied in this chapter.

P1. *Construct a list of n pseudorandom numbers strictly between 0 and 1. Suitable values for n are 10 (for debugging) and 500 (for comparing the results with other methods). Write a program to sort these numbers into an array via the following* ***interpolation sort****. First, clear the array (to all 0). For each number from the old list, multiply it by n, take the integer part, and look in that position of the table. If that position is 0, put the number there. If not, move left or right (according to the relative size of the current number and the one in its place) to find the position to insert the new number, moving the entries in the table over if necessary to make room (as in the fashion of insertion sort). Show that your algorithm will really sort the numbers correctly. Compare its running time with that of the other sorting methods applied to randomly ordered lists of the same size.*

interpolation sort

Answer The class definitions for keys and records with real number entries follow. Class Key:

TOC
Index
Help

```
class Key {
   double key;
public:
   static int comparisons;
   static int assignments;
   static void initialize();
   static int counter();
   Key (double x = 0.0);
   double the_key() const;
   Key &operator =(const Key &y);
};
bool operator ==(const Key &y,const Key &x);
bool operator !=(const Key &y,const Key &x);
bool operator >=(const Key &y,const Key &x);
bool operator <=(const Key &y,const Key &x);
bool operator >(const Key &y,const Key &x);
bool operator <(const Key &y,const Key &x);
```

Implementation:

```
int Key::comparisons = 0;
int Key::assignments = 0;
void Key::initialize()
{
   comparisons = 0;
}
int Key::counter()
{
   return comparisons;
}
Key::Key (double x)
{
   key = x;
}
Key &Key::operator =(const Key &x)
{
   Key::assignments++;
   key = x.key;
   return *this;
}
bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}
bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}
bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}
```

```
bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}

bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}

double Key::the_key() const
{
   return key;
}
```

Class Record:

```
class Record {
   Key key;
public:
   Record(double x);
   Record();
   operator Key() const;                //   cast to Key
};
```

Implementation:

```
Record::Record()
{
   key = 0;
}

Record::Record(double x)
{
   key = x;
}

Record::operator Key() const
{
   return key;
}
```

A driver for testing the sorting methods is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"
#include <time.h>
#include "../../c/timer.h"
#include "../../c/timer.cpp"
```

```
#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include "key.h"
#include "key.cpp"
#include "record.h"
#include "record.cpp"

void write_entry(Record &c)
{
   cout << ((Key) c).the_key() << " ";
}

#include "link.h"  // sortable list specification
#include "interp.cpp"
#include "distrib.cpp"

void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit  (re)[F]ill list \n"
        << "write [D]ata  write sorted [O]utput \n"

        << "[0] interpolation sort --- Project 5P1\n"
        << "[1] distribution sort ---  Project 5P2\n"
        << endl;
}

void intro()
{
   cout << "Testing program for sorting methods for a linked list."
        << endl;
   help ();
}

main()
{
   List<Record> s; List<Record> t = s;  t = s;
   intro();                       //  help out a poor old compiler

   int n;
   Random dice;
   Error_code report;
   Record target;
   Sortable_list<Record> the_list;
   Sortable_list<Record> copy_list;
   char command = ' ';

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, F, O, D, 0, 1: " << flush;
      cin  >> command;
      switch (command) {
         case 'h': case 'H':
           help();
         break;

         case 'd': case 'D':
            cout << "\nUnsorted list \n";
            the_list.traverse(write_entry);
            cout << endl;
         break;
         case 'o': case 'O':
            cout << "\nLast sorted list \n";
            copy_list.traverse(write_entry);
            cout << endl;
         break;

         case '0': case '1': {
           copy_list = the_list;
           Key::comparisons = 0;
           Key::assignments = 0;
           Timer clock;
           switch (command) {
              case '0':
              cout << "Interpolation Sort ";
              copy_list.interpolation();
           break;

              case '1':
              cout << "Distribution Sort ";
              cout << "How long would you like the chains to be on average?"
                   << flush;
              cin  >> n;
              copy_list.distribution(copy_list.size() / n + 1);
           break;

           }
           cout << "Time: " << clock.elapsed_time() << " seconds.\n"
                << "Comparisons: " << Key::comparisons << "\n"
                << "Assignments: " << Key::assignments
                << endl;
         }
         break;

         case 'f': case 'F':
           the_list.clear();
           cout << "How many list entries would you like? "
                << flush;
           cin  >> n;
           for (int i = 0; i < n; i++) {
              target = dice.random_real();
              report = the_list.insert(i, target);
              if (report == overflow) {
                 cout << "Available list space filled up at " << i
                      << " entries." << endl;
                 break;
              }
              if (report != success) i--;
           }
         break;
      }  // end of outer switch statement
   }     // end of outer while statement
}
```

A class definition allowing for the sorting methods is:

```
template <class Record>
class Sortable_list:public List<Record> {
public:   //  Add prototypes for sorting methods here.
   void interpolation();
   void distribution(int number_chains);
   void insertion_sort();
};
```

Interpolation sort is implemented as:

```
#include "auxil.cpp"

template <class Record>
void Sortable_list<Record>::interpolation()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
*/

{
   Record *array = new Record[size()];
   Record data;

   int i;
   for (i = 0; i < size(); i++) array[i] = -1.0;

   for (i = 0; i < size(); i++) {
      retrieve(i, data);
      int estimate = (int) (((Key) data).the_key() * size());
      if (array[estimate] < 0.0) array[estimate] = data;
      else if (data < array[estimate])
              insert_left(data, estimate, array, size());
           else
              insert_right(data, estimate, array, size());
    }

    for (i = 0; i < size(); i++)
       replace(i, array[i]);
    delete []array;
}


void push_right(Record k, int p, Record numbers[], int max);

void push_left(Record k, int p, Record numbers[], int max)
/*
push_left: push Records to the left to make room for k.
           push Records to the right if no room to the left
*/

{
        Record x;
        for ( ; p >= 0 && !(numbers[p] < 0.); p--) {
                x = numbers[p];
                numbers[p] = k;
                k = x;
        }
        if (p >= 0 && numbers[p] < 0.0) numbers[p] = k;
        else push_right(k, 0, numbers, max);
}
```

```
void push_right(Record k, int p, Record numbers[], int max)
/*
push_right: push Records to the right to make room for k.
            push Records to the left if no room to the right
*/

{
        Record x;

        for ( ; p <= max - 1 && !(numbers[p] < 0.); p++) {
                x = numbers[p];
                numbers[p] = k;
                k = x;
        }
        if (p <= max - 1 && numbers[p] < 0.0) numbers[p] = k;
        else push_left(x, max - 1, numbers, max);
}

void insert_left(Record k, int p, Record numbers[], int max)
/*
insert_left: insert Record to the left of position p if
             possible, moving Records to the left if necessary
*/

{
     for ( ; p > 0 && numbers[p] > k; p--);
     if (numbers[p] < 0.0) numbers[p] = k;
     else {
        if (p > 0) push_left(k, p, numbers, max);
        else push_right(k, p, numbers, max);
     }
}

void insert_right(Record k, int p, Record numbers[], int max)
/*
insert_left: insert Record to the right of position p if
             possible, moving Records to the right if necessary
*/

{
     for ( ; p < max - 1 && numbers[p] < k && !(numbers[p] < 0.0); p++);
     if (numbers[p] < 0.0) numbers[p] = k;
     else {
        if (p < max - 1) push_right(k, p, numbers, max);
        else push_left(k, p, numbers, max);
     }
}
```

P2. *[suggested by B. Lee] Write a program to perform a linked distribution sort, as follows. Take the keys to be pseudorandom numbers between 0 and 1, as in the previous project. Set up an array of linked lists, and distribute the keys into the linked lists according to their magnitude. The linked lists can either be kept sorted as the numbers are inserted or sorted during a second pass, during which the lists are all connected together into one sorted list. Experiment to determine the optimum number of lists to use. (It seems that it works well to have enough lists so that the average length of each list is about 3.)*

linked distribution sort

Answer Distribution sort is implemented as:

```
#include "../linklist/insert.cpp"
template <class Record>
void Sortable_list<Record>::distribution(int max)
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
*/

{
    Record data;

    Sortable_list<Record> *chains = new Sortable_list<Record>[max];
    int i;
    for (i = 0; i < size(); i++) {  // form the chains
       retrieve(i, data);
       int estimate = (int) (((Key) data).the_key() * max);
       while (estimate >= max) estimate--;
       chains[estimate].insert(0, data);
     }

    for (i = 0; i < max; i++)       // sort the chains
       chains[i].insertion_sort();

    int k = 0;                      // reform the original list (sorted)
    for (i = 0; i < max; i++)
       for (int j = 0; j < chains[i].size(); j++) {
          chains[i].retrieve(j, data);
          replace(k++, data);
       }
    delete []chains;
}
```

TOC
Index
Help

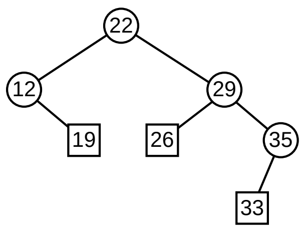
8.6 DIVIDE-AND-CONQUER SORTING

Exercises 8.6

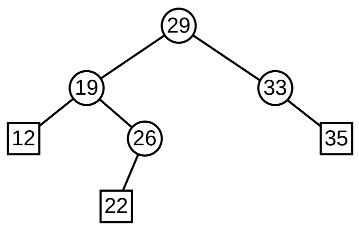
E1. *Apply quicksort to the list of seven numbers considered in this section, where the pivot in each sublist is chosen to be* **(a)** *the last number in the sublist and* **(b)** *the center (or left-center) number in the sublist. In each case, draw the tree of recursive calls.*

Answer

(a)



(b)



E2. *Apply mergesort to the list of 14 names considered for previous sorting methods:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

$\boxed{\text{Tim Dot Eva Roy Tom Kim Guy}}$ $\boxed{\text{Amy Jon Ann Jim Kay Ron Jan}}$

$\boxed{\text{Tim Dot Eva Roy}}$ $\boxed{\text{Tom Kim Guy}}$ $\boxed{\text{Amy Jon Ann Jim}}$ $\boxed{\text{Kay Ron Jan}}$

$\boxed{\text{Tim Dot}}\boxed{\text{Eva Roy}}$ $\boxed{\text{Tom Kim}}\boxed{\text{Guy}}$ $\boxed{\text{Amy Jon}}\boxed{\text{Ann Jim}}$ $\boxed{\text{Kay Ron}}\boxed{\text{Jan}}$

$\boxed{\text{Dot Tim}}\boxed{\text{Eva Roy}}$ $\boxed{\text{Kim Tom}}\boxed{\text{Guy}}$ $\boxed{\text{Amy Jon}}\boxed{\text{Ann Jim}}$ $\boxed{\text{Kay Ron}}\boxed{\text{Jan}}$

$\boxed{\text{Dot Eva Roy Tim}}$ $\boxed{\text{Guy Kim Tom}}$ $\boxed{\text{Amy Ann Jim Jon}}$ $\boxed{\text{Jan Kay Ron}}$

$\boxed{\text{Dot Eva Guy Kim Roy Tim Tom}}$ $\boxed{\text{Amy Ann Jan Jim Jon Kay Ron}}$

$\boxed{\text{Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom}}$

E3. *Apply quicksort to this list of 14 names, and thereby sort them by hand into alphabetical order. Take the pivot to be* **(a)** *the first key in each sublist and* **(b)** *the center (or left-center) key in each sublist.*

Answer **(a)**

$\boxed{\text{Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan}}$

$\boxed{\text{Dot Eva Roy Kim Guy Amy Jon Ann Jim Kay Ron Jan}}$ Tim $\boxed{\text{Tom}}$

$\boxed{\text{Amy Ann}}$ Dot $\boxed{\text{Eva Roy Kim Guy Jon Jim Kay Ron Jan}}$ Tim Tom

Amy $\boxed{\text{Ann}}$ Dot Eva $\boxed{\text{Roy Kim Guy Jon Jim Kay Ron Jan}}$ Tim Tom

Amy Ann Dot Eva $\boxed{\text{Kim Guy Jon Jim Kay Ron Jan}}$ Roy Tim Tom

Amy Ann Dot Eva $\boxed{\text{Guy Jon Jim Kay Jan}}$ Kim $\boxed{\text{Ron}}$ Roy Tim Tom

Amy Ann Dot Eva Guy $\boxed{\text{Jon Jim Kay Jan}}$ Kim Ron Roy Tim Tom

Amy Ann Dot Eva Guy $\boxed{\text{Jim Jan}}$ Jon $\boxed{\text{Kay}}$ Kim Ron Roy Tim Tom

Amy Ann Dot Eva Guy $\boxed{\text{Jan}}$ Jim Jon Kay Kim Ron Roy Tim Tom

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

(b)

$\boxed{\text{Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan}}$

$\boxed{\text{Dot Eva Amy Ann}}$ Guy $\boxed{\text{Tim Roy Tom Kim Jon Jim Kay Ron Jan}}$

$\boxed{\text{Dot Amy Ann}}$ Eva Guy $\boxed{\text{Jim Jan}}$ Jon $\boxed{\text{Tim Roy Tom Kim Kay Ron}}$

Amy $\boxed{\text{Dot Ann}}$ Eva Guy $\boxed{\text{Jan}}$ Jim Jon $\boxed{\text{Tim Roy Kim Kay Ron}}$ Tom

Amy $\boxed{\text{Ann}}$ Dot Eva Guy Jan Jim Jon $\boxed{\text{Kay}}$ Kim $\boxed{\text{Tim Roy Ron}}$ Tom

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim $\boxed{\text{Ron}}$ Roy $\boxed{\text{Tim}}$ Tom

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

TOC
Index
Help

E4. *In both divide-and-conquer methods, we have attempted to divide the list into two sublists of approximately equal size, but the basic outline of sorting by divide-and-conquer remains valid without equal-sized halves. Consider dividing the list so that one sublist has size 1. This leads to two methods, depending on whether the work is done in splitting one element from the list or in combining the sublists.*

(a) *Split the list by finding the entry with the largest key and making it the sublist of size 1. After sorting the remaining entries, the sublists are combined easily by placing the entry with largest key last.*

(b) *Split off the last entry from the list. After sorting the remaining entries, merge this entry into the list.*

Show that one of these methods is exactly the same method as insertion sort and the other is the same as selection sort.

Answer The first method is that used by selection sort. It works by finding the largest key and placing it at the end of the list. The second method is that used by insertion sort, which at each stage takes the next unsorted item and inserts it into the proper place relative to the previously sorted items.

8.7 MERGESORT FOR LINKED LISTS

Exercises 8.7

E1. *An article in a professional journal stated, "This recursive process [mergesort] takes time $O(n \log n)$, and so runs 64 times faster than the previous method [insertion sort] when sorting 256 numbers." Criticize this statement.*

Answer The author of the above statement is abusing the big-O notation, since the constants of proportionality are ignored. This notation is used for generalized analysis of algorithms for sufficiently large values of n. Recall the analysis of merge sort:

$$n \lg n - 1.1583n + 1 \approx 1752 \text{ for } n = 256$$

Recall the analysis for the number of comparisons done by insertion sort:

$$\tfrac{1}{4}n^2 - \tfrac{3}{4}n = 16192 \text{ for } n = 256.$$

It seems, through the above results, that merge sort performs less than ten times faster than insertion sort with a list of size 256. Besides, even if the operation counts were $n \lg n$ and n^2, the author's arithmetic is wrong: For $n = 256 = 2^8$ we have $n \lg n = 2^{11}$ and $n^2 = 2^{16}$ so the ratio would be $2^5 = 32$.

E2. *The count of key comparisons in merging is usually too high, since it does not account for the fact that one list may be finished before the other. It might happen, for example, that all entries in the first list come before any in the second list, so that the number of comparisons is just the length of the first list. For this exercise, assume that all numbers in the two lists are different and that all possible arrangements of the numbers are equally likely.*

(a) *Show that the average number of comparisons performed by our algorithm to merge two ordered lists of length 2 is $\frac{8}{3}$. [Hint: Start with the ordered list 1, 2, 3, 4. Write down the six ways of putting these numbers into two ordered lists of length 2, and show that four of these ways will use 3 comparisons, and two will use 2 comparisons.]*

Answer If the first list contains a and b with $a < b$ and the second contains c and d with $c < d$, then the possible orders for the combined list are

$$a < b < c < d \qquad a < c < b < d \qquad a < c < d < b$$

$$c < a < b < d \qquad c < a < d < b \qquad c < d < a < b.$$

The first and last of these will use only two comparisons, since both the elements of one list are greater than all those in the other, and the other four will use three comparisons. Hence the average number of comparisons is

$$\frac{1}{6}(2 \times 2 + 4 \times 3) = \frac{8}{3}.$$

(b) *Show that the average number of comparisons done to merge two ordered lists of length 3 is 4.5.*

Answer Even with lists of length 3 it becomes too long to list all possible orders (there are 20), so we change our point of view. Let the *combined* list consist of the elements

$$a < b < c < d < e < f$$

and to consider the original sequences think of each of these elements as labeled with a 1 or a 2 according to the list from which it comes. The number of ways of labeling three of the elements with 1 is $\binom{6}{3} = 20$ and the remaining elements will then be labeled 2, so there are 20 such sequences. The number of sequences that will require 5 comparisons is the number for which the last two elements, e and f, come from different lists. There are then 2 ways to choose e and f, and the remaining elements can be labeled in $\binom{4}{2} = 6$ ways, for a total of 12 choices. The sequences needing 4 comparisons are those in which e and f come from the same list and d from the other. This number is $2 \times \binom{3}{1} = 6$. If a sequence requires only 3 comparisons, then d, e, and f all come from the same list (and a, b, c from the other). There are 2 such sequences. The average number of comparisons needed is therefore

$$\frac{1}{20}(5 \times 12 + 4 \times 6 + 3 \times 2) = 4.5$$

.

(c) *Show that the average number of comparisons done to merge two ordered lists of length 4 is 6.4.*

Answer For similar reasons as part (b), the total number of sequences to consider is $\binom{8}{4} = 70$. The number of sequences for each number of comparisons is:

7 comparisons:	$2 \times \binom{6}{3} = 40$ sequences;	280 total comparisons;
6 comparisons:	$2 \times \binom{5}{3} = 20$ sequences;	120 total comparisons;
5 comparisons:	$2 \times \binom{4}{3} = 8$ sequences;	40 total comparisons;
4 comparisons:	$2 \times \binom{3}{3} = 2$ sequences;	8 total comparisons;

Hence the average number of comparisons is

$$\frac{1}{70}(280 + 120 + 40 + 8) = \frac{448}{70} = 6.4.$$

(d) *Use the foregoing results to obtain the improved total count of key comparisons for mergesort.*

Answer As in the text, we assume that the number n of entries to sort is a power of 2. By using a count of $2m - 1$ comparisons in merging two lists of length m, the book obtains a total count of $n \lg n - n + 1$. Part (a) allows us to reduce the count for sublists of length 2 from 3 comparisons to $\frac{8}{3}$ comparisons. There are $\frac{n}{4}$ pairs of sublists of length 2, so the total count can be reduced by $\frac{n}{12}$. Similarly, part (c) reduces the count by 0.6 for each of the $\frac{n}{8}$ pairs of sublists of length 4, for a total reduction of $0.075n$. Together, the total count is reduced to

$$n \lg n - 1.158333n + 1.$$

(e) *Show that, as m tends to infinity, the average number of comparisons done to merge two ordered lists of length m approaches $2m - 2$.*

Answer We shall, more generally, find the exact number of comparisons done by our merge algorithm, on average, in merging two sorted lists of length $m > 0$. As in parts (b) and (c), the total number of labelings of ordered sequences is $\binom{2m}{m}$. If the last k elements of the sequence are labeled the same and the preceding element is labeled differently, $1 \le k \le m$, then the merge algorithm will perform exactly $2m - k$ comparisons, and there are exactly $2\binom{2m-k-1}{m-1}$ such labelings. Hence the average number of comparisons done is exactly

$$\frac{\sum_{k=1}^{m}\left((2m - k)\times 2 \times \binom{2m-k-1}{m-1}\right)}{\binom{2m}{m}}.$$

By expanding the combinations as factorials we obtain

$$\sum_{k=1}^{m} \frac{2(2m-k)(2m-k-1)!m!m!}{(m-k)!(m-1)!(2m)!} = \frac{2m(m!)^2}{(2m)!}\sum_{k=1}^{m}\binom{2m-k}{m}.$$

This final sum is in the form of a well-known identity for the binomial coefficients (see, for example, Knuth, volume 1, pp. 54–55, formula (11)). The sum evaluates to $\binom{2m}{m-1}$, giving a final result of

$$\frac{2m^2}{m+1}$$

comparisons of keys required, on average, to merge two sorted lists of length m. To obtain the limit, we observe that

$$\frac{2m^2}{m+1} = 2m - 2 + \frac{2}{m} - \frac{2}{m^2} + \cdots,$$

and, as $m \to \infty$, the sum of all except the first two terms goes to 0.

TOC
Index
Help

E3. *[Very challenging] The straightforward method for merging two contiguous lists, by building the merged list in a separate array, uses extra space proportional to the number of entries in the two lists but can be written to run efficiently, with time proportional to the number of entries. Try to devise a merging method for contiguous lists that will require as little extra space as possible but that will still run in time (linearly) proportional to the number of entries in the lists. [There is a solution using only a small, constant amount of extra space. See the references.]*

fixed-space linear-time merging

Implementation of the algorithm given in the references requires substantial effort. Readers are invited to contribute a working version written in C++.

Programming Projects 8.7

P1. *Implement mergesort for linked lists on your computer. Use the same conventions and the same test data used for implementing and testing the linked version of insertion sort. Compare the performance of mergesort and insertion sort for short and long lists, as well as for lists nearly in correct order and in random order.*

Answer Refer to the linked driver program in Project P1 of Section 8.2 and to the results tables in Project P6 of Section 8.10.

```
template <class Record>
Node<Record> *Sortable_list<Record>::divide_from(Node<Record> *sub_list)
/*
Post: The list of nodes referenced by sub_list has been reduced
      to its first half, and a pointer to the first node in the
      second half of the sublist is returned.  If the sublist has
      an odd number of entries, then its first half will be one
      entry larger than its second.
Uses: The linked List implementation of Chapter 6.
*/

{

   Node<Record> *position, //  traverses the entire list
                *midpoint, //  moves at half speed of position to midpoint
                *second_half;

   if ((midpoint = sub_list) == NULL) return NULL;  //  List is empty.
   position = midpoint->next;
   while (position != NULL) {// Move position twice for midpoint's one move.
      position = position->next;
      if (position != NULL) {
         midpoint = midpoint->next;
         position = position->next;
      }
   }
   second_half = midpoint->next;
   midpoint->next = NULL;
   return second_half;
}

template <class Record>
Node<Record> *Sortable_list<Record>::merge(Node<Record> *first,
                                          Node<Record> *second)
/*
Pre:  first and second point to ordered lists of nodes.
Post: A pointer to an ordered list of nodes is returned.
      The ordered list contains all entries that were referenced by
      first and second.  The original lists of nodes referenced
      by first and second are no longer available.
```

```
Uses: Methods for Record class; the linked
      List implementation of Chapter 6.
*/
{
   Node<Record> *last_sorted; //  points to the last node of sorted list
   Node<Record> combined;     //  dummy first node, points to merged list

   last_sorted = &combined;
   while (first != NULL && second != NULL) {//Attach node with smaller key
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;   //  Advance to the next unmerged node.
      }

      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }

//  After one list ends, attach the remainder of the other.
   if (first == NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}

template <class Record>
void Sortable_list<Record>::recursive_merge_sort(Node<Record> *&sub_list)
/*
Post: The nodes referenced by sub_list have been
      rearranged so that their keys
      are sorted into nondecreasing order.  The pointer
      parameter sub_list is reset to point
      at the node containing the smallest key.
Uses: The linked List implementation of Chapter 6;
      the functions divide_from, merge, and recursive_merge_sort.
*/
{
   if (sub_list != NULL && sub_list->next != NULL) {
      Node<Record> *second_half = divide_from(sub_list);
      recursive_merge_sort(sub_list);
      recursive_merge_sort(second_half);
      sub_list = merge(sub_list, second_half);
   }
}

template <class Record>
void Sortable_list<Record>::merge_sort()
/*
Post: The entries of the sortable list have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The linked List implementation of Chapter 6 and
      recursive_merge_sort.
*/
```

```
{
   recursive_merge_sort(head);
}
```

P2. *Our mergesort program for linked lists spends significant time locating the center of each sublist, so that it can be broken in half. Implement the following modification that will save some of this time. Rewrite the* divide_from *function to use a second parameter giving the length of original list of nodes. Use this to simplify and speed up the subdivision of the lists. What modifications are needed in the functions* merge_sort() *and* recursive_merge_sort()*?*

Answer

```
template <class Record>
Node<Record> *Sortable_list<Record>::divide7P2(Node<Record> *sub_list,
                                               int length1, int length2)
/*
Post: The list of nodes referenced by sub_list has been reduced
      to its first half, and a pointer to the first node in the
      second half of the sublist is returned.  If the sublist
      has an odd number of entries, then its first half will be
      one entry larger than its second.
Uses: The linked List implementation of Chapter 6.
*/

{
   if  (length2 == 0) return NULL;       //  Second half empty.

   Node<Record> *position = sub_list, *second_half;
   int  i = 0;
   while (i < length1 - 1) {
      position = position->next;
      i++;
   }
   second_half = position->next;
   position->next = NULL;
   return second_half;
}

template <class Record>
Node<Record> *Sortable_list<Record>::merge7P2(Node<Record> *first,
                                           Node<Record> *second)
/*
Pre:  first and second point to ordered lists of nodes.
Post: A pointer to an ordered list of nodes is returned.
      The ordered list contains all entries that were referenced by
      first and second.  The original lists of nodes referenced
      by first and second are no longer available.
Uses: Methods for Record class; the linked
      List implementation of Chapter 6.
*/

{
   Node<Record> *last_sorted;  //  points to the last node of sorted list
   Node<Record> combined;      //  dummy first node, points to merged list

   last_sorted = &combined;
   while (first != NULL && second != NULL) {// Attach node with smaller key
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;  //  Advance to the next unmerged node.
      }
      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }
//  After one list ends, attach the remainder of the other.
   if (first == NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}

template <class Record>
void Sortable_list<Record>::recursive_merge7P2
                              (Node<Record> *&sub_list, int length)
/*
Post: The nodes referenced by sub_list have been
      rearranged so that their keys
      are sorted into nondecreasing order.  The pointer
      parameter sub_list is reset to point
      at the node containing the smallest key.
Uses: The linked List implementation of Chapter 6;
      the functions divide7P2, merge7P2, and recursive_merge7P2.
*/
{
   if (length > 1) {
      int length2 = length / 2, length1 = length - length2;
      Node<Record> *second_half = divide7P2(sub_list, length1, length2);
      recursive_merge7P2(sub_list, length1);
      recursive_merge7P2(second_half, length2);
      sub_list = merge7P2(sub_list, second_half);
   }
}

template <class Record>
void Sortable_list<Record>::merge7P2()
/*
Post: The entries of the sortable list have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The linked List implementation of Chapter 6 and
      recursive_merge7P2.
*/
{
   recursive_merge7P2(head, size());
}
```

TOC
Index
Help

P3. *Our mergesort function pays little attention to whether or not the original list was partially in the correct order. In **natural mergesort** the list is broken into sublists at the end of an increasing sequence of keys, instead of arbitrarily at its halfway point. This exercise requests the implementation of two versions of natural mergesort.*

natural mergesort

(a) *In the first version, the original list is traversed only once, and only two sublists are used. As long as the order of the keys is correct, the nodes are placed in the first sublist. When a key is found out of order, the first sublist is ended and the second started. When another key is found out of order, the second sublist is ended, and the second sublist merged into the first. Then the second sublist is repeatedly built again and*

one sorted list

merged into the first. When the end of the original list is reached, the sort is finished. This first version is simple to program, but as it proceeds, the first sublist is likely to become much longer than the second, and the performance of the function will degenerate toward that of insertion sort.

Answer

```
template <class Record>
Node<Record> *Sortable_list<Record>::divide_from7P3a(Node<Record> *sub_list)
/*
Post: The list of nodes referenced by sub_list has been reduced to its
      first increasing segment, and a pointer to the first node in the
      remainder is returned.
Uses: The linked List implementation of Chapter 6.
*/

{
   Node<Record> *position = sub_list, *subsequent;
   if (position == NULL) return NULL;  //  List is empty.
   while (true) {
      subsequent = position->next;
      if (subsequent == NULL || subsequent->entry < position->entry)
         break;

      position = subsequent;
   }
   position->next = NULL;
   return subsequent;
}

template <class Record>
Node<Record> *Sortable_list<Record>::merge7P3a(Node<Record> *first,
                                              Node<Record> *second)
/*
Pre:  first and second point to ordered lists of nodes.
Post: A pointer to an ordered list of nodes is returned.
      The ordered list contains all entries that were referenced by
      first and second.  The original lists of nodes referenced
      by first and second are no longer available.
Uses: Methods for Record class; the linked
      List implementation of Chapter 6.
*/

{
   Node<Record> *last_sorted; //  points to the last node of sorted list
   Node<Record> combined;     //  dummy first node, points to merged list

   last_sorted = &combined;
   while (first != NULL && second != NULL) { // Attach node with smaller key
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;   //  Advance to the next unmerged node.
      }

      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }
//  After one list ends, attach the remainder of the other.
   if (first == NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}

template <class Record>
void Sortable_list<Record>::merge_sort7P3a()
/*
Post: The entries of the sortable list have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The linked List implementation of Chapter 6.
*/

{
   Node<Record> *part2 = divide_from7P3a(head);
   while (part2 != NULL) {
      Node<Record> *part3 = divide_from7P3a(part2);
      head = merge7P3a(head, part2);
      part2 = part3;
   }
}
```

(b) *The second version ensures that the lengths of sublists being merged are closer to being equal and, therefore, that the advantages of divide and conquer are fully used. This method keeps a (small) auxiliary array containing (1) the lengths and (2) pointers to the heads of the ordered sublists that are not yet merged. The entries in this array should be kept in order according to the length of sublist. As each (naturally ordered) sublist is split from the original list, it is put into the auxiliary array. If there is another list in the array whose length is between half and twice that of the new list, then the two are merged, and the process repeated. When the original list is exhausted, any remaining sublists in the array are merged (smaller lengths first) and the sort is finished.*

several sorted lists

There is nothing sacred about the ratio of 2 in the criterion for merging sublists. Its choice merely ensures that the number of entries in the auxiliary array cannot exceed $\lg n$ *(prove it!). A smaller ratio (required to be greater than 1) will make the auxiliary table larger, and a larger ratio will lessen the advantages of divide and conquer. Experiment with test data to find a good ratio to use.*

Answer For the second version we set up an array of sublists not yet merged, and we require that each sublist in the array be about twice as long as the last sublist. Hence if there are k sublists in the array, the last must have length about 2^k. Since the number n of entries in the entire list is at most 2^k, we obtain that the total number of sublists in the auxiliary array cannot exceed $\lg n$.

```
template <class Record>
Node<Record> *Sortable_list<Record>::divide_from7P3b(Node<Record> *sub_list,
                                                     int &length_sub)
/*
Post: The list of nodes referenced by sub_list has been reduced to its
      first increasing segment, and a pointer to the first node in the
      remainder is returned.
Uses: The linked List implementation of Chapter 6.
*/
```

```
{
   Node<Record> *position = sub_list, *subsequent;
   length_sub = 0;
   if (position == NULL) return NULL;  //  List is empty.
   while (true) {
      length_sub++;
      subsequent = position->next;
      if (subsequent == NULL || subsequent->entry < position->entry)
         break;

      position = subsequent;
   }
   position->next = NULL;
   return subsequent;
}

template <class Record>
Node<Record> *Sortable_list<Record>::merge7P3b(Node<Record> *first,
                                                Node<Record> *second)
/*
Pre:  first and second point to ordered lists of nodes.
Post: A pointer to an ordered list of nodes is returned.
      The ordered list contains all entries that were referenced by
      first and second.  The original lists of nodes referenced
      by first and second are no longer available.
Uses: Methods for Record class; the linked
      List implementation of Chapter 6.
*/

{
   Node<Record> *last_sorted; //  points to the last node of sorted list
   Node<Record> combined;     //  dummy first node, points to merged list

   last_sorted = &combined;
   while (first != NULL && second != NULL) { // Attach node with smaller key
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;   //  Advance to the next unmerged node.
      }

      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }

//  After one list ends, attach the remainder of the other.
   if (first == NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}
```

```
template <class Record>
void Sortable_list<Record>::merge_sort7P3b()
/*
Post: The entries of the sortable list have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The linked List implementation of Chapter 6.
*/

{
   List<int> lengths;
   List<Node<Record>*> heads;

   Node<Record> *part = head, *part2, *old_part;
   int length, old_length;
   while (part != NULL) {
      Node<Record> *part2 = divide_from7P3b(part, length);
      int i = 0;
      while (true) {
         if (lengths.retrieve(i, old_length) != success) {
            lengths.insert(i, length);
            heads.insert(i, part);
            break;

         }
         if (2 * old_length < length) i++;
         else if (2 * length < old_length) {
            lengths.insert(i, length);
            heads.insert(i, part);
            break;

         }
         else {
            length += old_length;
            heads.remove(i, old_part);
            lengths.remove(i, old_length);
            part= merge7P3b(part, old_part);
         }
      }
      part = part2;
   }

   if (!heads.empty()) {
      heads.remove(0, head);
      while (!heads.empty()) {
         heads.remove(0, part);
         head = merge7P3b(head, part);
      }
   }
}
```

contiguous mergesort

P4. *Devise a version of mergesort for contiguous lists. The difficulty is to produce a function to merge two sorted lists in contiguous storage. It is necessary to use some additional space other than that needed by the two lists. The easiest solution is to use two arrays, each large enough to hold all the entries in the two original lists. The two sorted sublists occupy different parts of the same array. As they are merged, the new list is built in the second array. After the merge is complete, the new list can, if desired, be copied back into the first array. Otherwise, the roles of the two arrays can be reversed for the next stage.*

Answer
```
template <class Record>
void Sortable_list<Record>::merge(int low, int high)
{
    Record *temp = new Record[high - low + 1];
    int index = 0;
    int index1 = low, mid = (low + high) / 2, index2 = mid + 1;
    while (index1 <= mid && index2 <= high) {
       if (entry[index1] < entry[index2])
          temp[index++] = entry[index1++];
       else
          temp[index++] = entry[index2++];
    }
    while (index1 <= mid)
       temp[index++] = entry[index1++];
    while (index2 <= high)
       temp[index++] = entry[index2++];
    for (index = low; index <= high; index++)
       entry[index] = temp[index -low];
    delete []temp;
}

template <class Record>
void Sortable_list<Record>::recursive_merge_sort(int low, int high)
/*
Post: The entries of the sortable list between
      index low and high have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of Chapter 6.
*/

{
   if (high > low) {
      recursive_merge_sort(low, (high + low) / 2);
      recursive_merge_sort((high + low) / 2 + 1, high);
      merge(low, high);
   }
}

template <class Record>
void Sortable_list<Record>::merge_sort()
/*
Post: The entries of the sortable list have been rearranged so that
      their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of Chapter 6.
*/

{
   recursive_merge_sort(0, size() - 1);
}
```

8.8 QUICKSORT FOR CONTIGUOUS LISTS

Exercises 8.8

E1. *How will the quicksort function (as presented in the text) function if all the keys in the list are equal?*

Answer With all keys equal, the partition function will scan through the entire sublist making no swaps other than the initial swap of the pivot before scanning and swapping the pivot with itself after the scanning. The pivot position will always be the same as the first position of the sublist passed to the function and recursive_quick_sort will then be called again with a sublist one entry smaller. Therefore, quicksort will require roughly twice as many comparisons, $\frac{1}{2}n^2 + O(n)$, as insertion sort.

E2. *[Due to KNUTH] Describe an algorithm that will arrange a contiguous list whose keys are real numbers so that all the entries with negative keys will come first, followed by those with nonnegative keys. The final list need not be completely sorted. Make your algorithm do as few movements of entries and as few comparisons as possible. Do not use an auxiliary array.*

Answer The function partition can be easily modified to use 0 as its pivot and will then place all negative keys before all positive keys.

E3. *[Due to HOARE] Suppose that, instead of sorting, we wish only to find the m^{th} smallest key in a given list of size n. Show how quicksort can be adapted to this problem, doing much less work than a complete sort.*

Answer

```
template <class Record> void Sortable_list<Record> ::
        recursive_find_rank_key(int low, int high, int m)
/* Pre:   low and high are valid positions in the Sortable_list. m is between low and high
   Post:  Places the m^th smallest key in position m in the list without completely sorting the list
   Uses:  The contiguous List implementation of Chapter 6
{
   int pivot_position;
   if (low < high) {                              //   Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      if (m < pivot_position)
         recursive_find_rank_key(low, pivot_position − 1, m);
      else if (m > pivot_position)
         recursive_find_rank_key(pivot_position + 1, high, m);
   }
}
template <class Record> void Sortable_list<Record> :: find_rank_key(int m)
/* Post: Places the m^th smallest key in position m in the list without completely sorting the list */
{
   recursive_find_rank_key(0, count − 1, m);
}
```

E4. *Given a list of integers, develop a function, similar to the partition function, that will rearrange the integers so that either all the integers in even-numbered positions will be even or all the integers in odd-numbered positions will be odd. (Your function will provide a proof that one or the other of these goals can always be attained, although it may not be possible to establish both at once.)*

Answer

```
void even_odd(List<int> & the_list)
/* Pre:
   Post: either all even positions of the List the_list store even numbers or all odd positions of
         List the_list store odd integers. */
{
   int s = the_list.size( );
   int even_pos = 0, odd_pos = 1;
   while (odd_pos < s && even_pos < s) {
      int data_e, data_o;
      the_list.retrieve(even_pos, data_e);
      the_list.retrieve(odd_pos, data_o);
      if (data_e % 2 ==  0) {
         even_pos += 2;
         the_list.retrieve(even_pos, data_e);
      }
```

```
        else {
          while (odd_pos < s && (data_o % 2 != 0)) {
            odd_pos += 2;
            the_list.retrieve(odd_pos, data_o);
          }
          if (odd_pos < s) {
            swap(the_list, even_pos, odd_pos);
            odd_pos += 2;
            even_pos += 2;
          }
        }
      }
    }
```

E5. *A different method for choosing the pivot in quicksort is to take the median of the first, last, and central keys of the list. Describe the modifications needed to the function* quick_sort *to implement this choice. How much extra computation will be done? For $n = 7$, find an ordering of the keys*

$$1, 2, \ldots, 7$$

that will force the algorithm into its worst case. How much better is this worst case than that of the original algorithm?

Answer The selection of pivot must be changed in the function partition by using the function median to select the median of the three keys and proceeds as follows:

```
  swap(low, median(low, high, (low + high)/2)); //  swap pivot into first location
  pivot = entry[low];                          //  First entry is now pivot.
```

The function median is a private auxiliary member function of the **class** Sortable_list with the following implementation.

```
template <class Record>
int Sortable_list<Record>::median(int a, int b, int c)
/* Post: The median of the three parameter values is returned. */
{
    if (entry[a] > entry[b])
      if (entry[c] > entry[a]) return a;        //  c > a > b
      else if (entry[c] > entry[b]) return c;   //  a > c > b
           else return b;                       //  a >= b > c
    else if (entry[c] > entry[a])
      if (entry[c] > entry[b]) return b;        //  c > b > a
      else return c;                            //  b >= c > a
     else return a;                             //  b >= a >= c
}
```

The same amount of computation, of (low + high)/2, is done; however, a call to the new function median accounts for an average of three key comparisons with each call.

The worst case for this sorting method involves splitting n entries into sublists of size 2 and $n - 2$. This is true because the median of three of the key in the list is guaranteed to have at least one key less than it. The following ordering of seven keys will produce a worst case:

$$1 \quad 6 \quad 4 \quad 2 \quad 5 \quad 7 \quad 3.$$

The first call to partition will select 2 as the pivot, placing 2 and 1 in the first sublist and 4, 6, 5, 7, and 3 in the second. The larger sublist will be partitioned using 4 as a pivot, placing 4 and 3 in the first sublist and 5, 7, and 6 in the second. The larger sublist will then be partitioned using 6 as the pivot. The maximum depth of recursion with this method is about $n/2$ whereas the method presented in the text would have a maximum depth of about n.

E6. *A different approach to the selection of pivot is to take the mean (average) of all the keys in the list as the pivot. The resulting algorithm is called* ***meansort***.

meansort

(a) *Write a function to implement meansort. The partition function must be modified, since the mean of the keys is not necessarily one of the keys in the list. On the first pass, the pivot can be chosen any way you wish. As the keys are then partitioned, running sums and counts are kept for the two sublists, and thereby the means (which will be the new pivots) of the sublists can be calculated without making any extra passes through the list.*

Answer See the reference listed at the end of the chapter.

(b) *In meansort, the relative sizes of the keys determine how nearly equal the sublists will be after partitioning; the initial order of the keys is of no importance, except for counting the number of swaps that will take place. How bad can the worst case for meansort be in terms of the relative sizes of the two sublists? Find a set of n integers that will produce the worst case for meansort.*

Answer The list consisting of $1, 2, 4, \dots, 2^{n-1}$ will produce the worst case for meansort. Since the sum of all except the last entry is less than the last entry alone, at each stage meansort will partition the list into sublists one of which has length one.

E7. *[Requires elementary probability theory] A good way to choose the pivot is to use a random-number generator to choose the position for the next pivot at each call to* recursive_quick_sort. *Using the fact that these choices are independent, find the probability that quicksort will happen upon its worst case.* **(a)** *Do the problem for $n = 7$.* **(b)** *Do the problem for general n.*

Answer The worst case of quicksort occurs when every pivot chosen is either the largest or smallest key in its sublist. With $n = 7$, the probability of using either the largest or smallest key as the pivot is $\frac{2}{7}$, with $n = 6$ the probability is $\frac{2}{6}$. Since these occurrences are independent, the probability of all of these occurring, the worst case, is the product of all probabilities of using either the largest or smallest key as the pivot. The total probability that quicksort happen upon its worst case with $n = 7$ is:

$$\frac{2}{7} \times \frac{2}{6} \times \frac{2}{5} \times \frac{2}{4} \times \frac{2}{3} \times \frac{2}{2} = \frac{64}{5040} = \frac{4}{315}.$$

For a general list of size n, the chance that quicksort will happen upon its worst case is:

$$\frac{2}{n} \times \frac{2}{n-1} \times \dots \times \frac{2}{2} = \frac{2^{n-1}}{n!}.$$

E8. *At the cost of a few more comparisons of keys, the partition function can be rewritten so that the number of swaps is reduced by a factor of about 3, from $\frac{1}{2}n$ to $\frac{1}{6}n$ on average. The idea is to use two indices moving from the ends of the lists toward the center and to perform a swap only when a large key is found by the low position and a small key by the high position. This exercise outlines the development of such a function.*

(a) *Establish two indices* i *and* j, *and maintain the invariant property that all keys before position* i *are less than or equal to the pivot and all keys after position* j *are greater than the pivot. For simplicity, swap the pivot into the first position, and start the partition with the second element. Write a loop that will increase the position* i *as long as the invariant holds and another loop that will decrease* j *as long as the invariant holds. Your loops must also ensure that the indices do not go out of range, perhaps by checking that* i $\le$ j. *When a pair of entries, each on the wrong side, is found, then they should be swapped and the loops repeated. What is the termination condition of this outer loop? At the end, the pivot can be swapped into its proper place.*

Answer
```
template <class Record>
int Sortable_list<Record> :: partition(int low, int high)
/* Pre:  low and high are valid positions of the Sortable_list, with low <= high.
   Post: The center (or left center) entry in the range between indices low and high of the
         Sortable_list has been chosen as a pivot. All entries of the Sortable_list between indices
         low and high, inclusive, have been rearranged so that those with keys less than the
         pivot come before the pivot and the remaining entries come after the pivot. The final
         position of the pivot is returned.
   Uses: swap(int i, int j) (interchanges entries in positions i and j of a Sortable_list), the contigu-
         ous List implementation of Chapter 6
{
   Record pivot;
   int i = low, j = high + 1;                //  used to scan through the list
   swap(low, (low + high)/2);
   pivot = entry[low];                       //  First entry is now pivot.
   while (i < j) {
      do {
         i++;
      } while (i < j && entry[i] <= pivot);
      do {
         j−−;
      } while (i < j && entry[j] > pivot);
      if (i < j) swap(i, j);
   }
   swap(low, i − 1);                         //  Put the pivot into its proper position.
   return i − 1;
}
```

(b) *Using the invariant property, verify that your function works properly.*

Answer At the beginning of the loop, all entries between low and i, inclusive, have keys less than or equal to pivot; all entries between j and high, inclusive, have keys greater than pivot. It also holds that i < high and j > low. After the two inner **do** ... **while** loops, entries may have been found that violate the first condition (the invariant), but after they are swapped the invariants become true again. Hence the invariants remain true provided that i < j, and in the contrary case the main loop terminates. Therefore, the function works properly.

(c) *Show that each swap performed within the loop puts two entries into their final positions. From this, show that the function does at most $\frac{1}{2}n + O(1)$ swaps in its worst case for a list of length n.*

Answer With each iteration of the main **while** repeat loop i increases by at least one and j decreases by at least one. This implies that the main loop will iterate no more than $\frac{1}{2}n$ times. Once this loop has terminated, the entries, other than the pivot and the entry in position $i-1$, are no longer disturbed. Therefore, no more than $\frac{1}{2}n + O(1)$ swaps are done by the partition function.

(d) *If, after partitioning, the pivot belongs in position p, then the number of swaps that the function does is approximately the number of entries originally in one of the p positions at or before the pivot, but whose keys are greater than or equal to the pivot. If the keys are randomly distributed, then the probability that a particular key is greater than or equal to the pivot is $\frac{1}{n}(n-p-1)$. Show that the average number of such keys, and hence the average number of swaps, is approximately $\frac{p}{n}(n-p)$. By taking the average of these numbers from $p = 1$ to $p = n$, show that the number of swaps is approximately $\frac{n}{6} + O(1)$.*

Answer Without loss of generality we assume that the keys are the integers from 1 to n. The number of swaps that will have been made in one call to partition is the number of keys in positions 1 through $p-1$ (i.e., before p) that are greater than p, plus one more if the position p is not p itself before the partition. The probability that any particular key is greater than p is $\frac{(n-p+1)}{n}$. To obtain the number of swaps expected we shall add these probabilities. The answer, however, is only approximate, since we do not know whether one key is greater than p is independent of the same property for another key. We obtain:

$$\frac{(p-1)(n-p+1)}{n}+\frac{n-1}{n}=\frac{(n+2)p-p^2-2}{n}.$$

We must now take the average of all these numbers, since p is random, by adding them from $p=1$ to $p=n$. The calculation requires the identities in Appendix A.1, and the result is $\frac{n}{6}+1-\frac{7}{n}$ swaps, which we approximate to $\frac{n}{6}+O(1)$ swaps.

(e) *The need to check to make sure that the indices* i *and* j *in the partition stay in bounds can be eliminated by using the pivot key as a sentinel to stop the loops. Implement this method in your function. Be sure to verify that your function works correctly in all cases.*

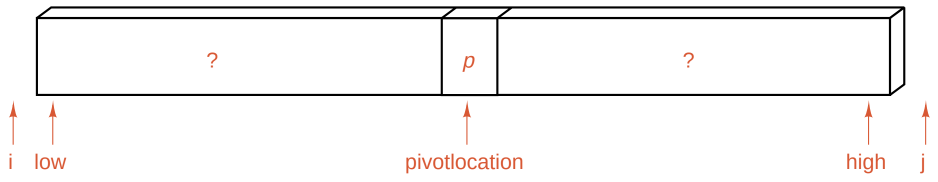
Answer

```
template <class Record>
int Sortable_list<Record> :: partition(int low, int high)
/* Pre:  low and high are valid positions of the Sortable_list, with low <= high.
   Post: The center (or left center) entry in the range between indices low and high of the
         Sortable_list has been chosen as a pivot. All entries of the Sortable_list between indices
         low and high, inclusive, have been rearranged so that those with keys less than the
         pivot come before the pivot and the remaining entries come after the pivot. The final
         position of the pivot is returned.
   Uses: swap(int i, int j) (interchanges entries in positions i and j of a Sortable_list), the contigu-
         ous List implementation of Chapter 6
{
  int pivot_position = (low + high)/2;
  Record pivot = entry[pivot_position];
  int i = low − 1, j = high + 1;               //  used to scan through the list
  while (i < j) {
    do {
      i++;
    } while (entry[i] < pivot);
    do {
      j−−;
    } while (entry[j] > pivot);
    if (i < j) {
      swap(i, j);
      if (pivot_position ==  i) pivot_position = j;
      else if (pivot_position ==  j) pivot_position = i;
    }
  }
  if (pivot_position < j) {
    swap(pivot_position, i − 1);
    pivot_position = i − 1;
  }
  else if (pivot_position > i) {
    swap(pivot_position, j + 1);
    pivot_position = j + 1;
  }
  return pivot_position;
}
```

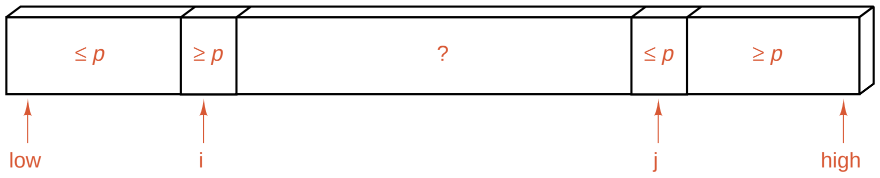
To establish formally that this function works correctly, we shall use the method of invariant assertions introduced in The invariants for the outer repeat loop are:

At the beginning of the loop, all entries between low *and* i*, inclusive, have keys less than or equal to* pivot*; and all entries between* j *and* high*, inclusive, have keys greater than or equal to* pivot.

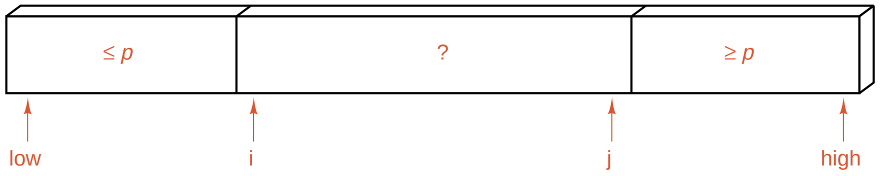
The partition function begins by initializing variables as follows:



At the first iteration of the outer loop these invariants are vacuously true, since i < low, so there are no entries in the specified range, and j > high, so there are no entries between. Now assume that the invariants are true at the beginning of some iteration. After the two inner repeat loops, entries may have been found that violate the invariants, as follows:

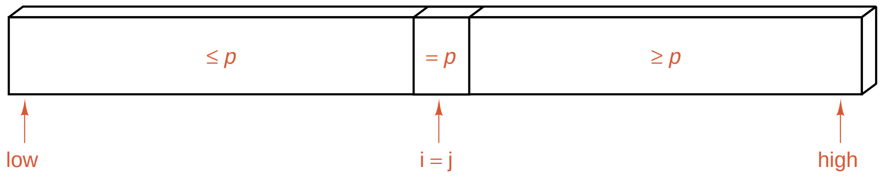


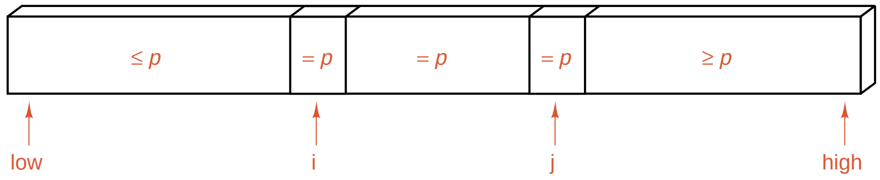
When they are swapped the invariants become true again, as follows:



Hence the invariants remain true as long as the swap is done, that is, as long as i < j. In the contrary case, i ≥ j, the outer loop terminates. Hence the proof is complete.

When the outer loop terminates, all entries strictly before i have keys less than or equal to pivot, and entry i has key greater than or equal to pivot. All entries strictly after j have keys greater than or equal to pivot, and entry j has key less than or equal to pivot. Moreover, we know that j ≤ i. The pivot (which was originally at pivot_position) might have been swapped to anywhere in the list. If it is before j then it is now swapped into position i − 1, the last position guaranteed to be less than or equal to pivot. If it comes after i then it is swapped to position j + 1, the first position guaranteed to be at least pivot. If it comes between j and i, inclusive, then it is already in position to partition the list properly. These situations are illustrated in the following diagrams.





(f) *[Due to* WIRTH*] Consider the following simple and "obvious" way to write the loops using the pivot as a sentinel:*

```
do {
  do { i = i + 1; } while (entry[i] < pivot);
  do { j = j − 1; } while (entry[j] > pivot);
  swap(i, j);
} while (j > i);
```

Find a list of keys for which this version fails.

Answer Suppose that the pivot value is 3 and the list has been reduced to the keys 3 1 2, with i pointing to 3 and j to 2 after the two inner do . . . while loops. After the entries are swapped, the list will be 2 1 3, and the do . . . while loop on i will move i so that both i and j point to the last entry, 3. Next, the do . . . while loop on j will move j one entry back to 1. The final swap will change the list to 2 3 1, and the outer loop terminates with the entry 1 on the wrong side of the pivot.

Programming Projects 8.8

P1. *Implement quicksort (for contiguous lists) on your computer, and test it with the program from Project P1 of Section 8.2 (page 328). Compare its performance with that of all the other sorting methods you have studied.*

Answer Refer to the test result tables in Project P6 of Section 8.10.

```
template <class Record>
int Sortable_list<Record>::partition(int low, int high)
/*
Pre:  low and high are valid positions of the Sortable_list,
      with low <= high.
Post: The center (or left center) entry in the range between indices
      low and high of the Sortable_list
      has been chosen as a pivot.  All entries of the Sortable_list
      between indices low and high, inclusive, have been
      rearranged so that those with keys less than the pivot come
      before the pivot and the remaining entries come
      after the pivot.  The final position of the pivot is returned.
Uses: swap(int i, int j) (interchanges entries in positions
      i and j of a Sortable_list),
      the contiguous List implementation of ?list_ch?, and
      methods for the class Record.
*/

{
   Record pivot;
   int i,            //  used to scan through the list
       last_small;   //  position of the last key less than pivot
   swap(low, (low + high) / 2);
   pivot = entry[low];   //  First entry is now pivot.
   last_small = low;
   for (i = low + 1; i <= high; i++)

/*
At the beginning of each iteration of this loop, we have the following
conditions:

      If low < j <= last_small then entry[j].key < pivot.

      If last_small < j < i then entry[j].key >= pivot.
*/
```

```
      if (entry[i] < pivot) {
         last_small = last_small + 1;
         swap(last_small, i);  //  Move large entry to right, small to left.
      }
   swap(low, last_small);      //  Put the pivot into its proper position.
   return last_small;
}

template <class Record>
void Sortable_list<Record>::recursive_quick_sort(int low, int high)
/*

Pre:  low and high are valid positions in the Sortable_list.
Post: The entries of the Sortable_list have been
      rearranged so that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort, and partition.
*/

{
   int pivot_position;
   if (low < high) {   //   Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      recursive_quick_sort(low, pivot_position - 1);
      recursive_quick_sort(pivot_position + 1, high);
   }
}

template <class Record>
void Sortable_list<Record>::quick_sort()
/*
Post: The entries of the Sortable_list have been rearranged so
      that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort.
*/

{
   recursive_quick_sort(0, count - 1);
}
```

P2. *Write a version of quicksort for linked lists, integrate it into the linked version of the testing program from Project P1 of Section 8.2 (page 328), and compare its performance with that of other sorting methods for linked lists.*

linked quicksort

Use the first entry of a sublist as the pivot for partitioning. The partition function for linked lists is somewhat simpler than for contiguous lists, since minimization of data movement is not a concern. To partition a sublist, you need only traverse it, deleting each entry as you go, and then add the entry to one of two lists depending on the size of its key relative to the pivot.

Since partition *now produces two new lists, you will, however, require a short additional function to recombine the sorted sublists into a single linked list.*

Answer

```
template <class Record>
void Sortable_list<Record>::partition(Node<Record> *sub_list,
        Node<Record> *&first_small, Node<Record> *&last_small,
         Node<Record> *&first_large, Node<Record> *&last_large)
```

```
{
   Record pivot = sub_list->entry;
   sub_list = sub_list->next;
   Node<Record> small, large;
   last_small = &small;
   last_large = &large;
   while (sub_list!= NULL) {
      if (sub_list->entry > pivot)
         last_large = last_large->next = sub_list;
      else
         last_small = last_small->next = sub_list;
      sub_list = sub_list->next;
   }
   last_large->next = last_small->next = NULL;
   if ((first_small = small.next) == NULL) last_small = NULL;
   if ((first_large = large.next) == NULL) last_large = NULL;
}

template <class Record>
void Sortable_list<Record>::recursive_quick_sort(Node<Record> *&first_node,
                                                 Node<Record> *&last_node)
{
   if (first_node != NULL && first_node != last_node) {
                              //   Otherwise, no sorting is needed.
      Node<Record> *first_small, *last_small, *first_large, *last_large;
      partition(first_node, first_small, last_small,
                            first_large, last_large);
      recursive_quick_sort(first_small, last_small);
      recursive_quick_sort(first_large, last_large);

      if (last_large != NULL) last_node = last_large;
      else last_node = first_node;

      first_node->next = first_large;

      if (first_small != NULL) {
          last_small->next = first_node;
          first_node = first_small;
      }
   }
}

template <class Record>
void Sortable_list<Record>::quick_sort()
/*
Post: The entries of the Sortable_list have been rearranged so
      that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort.
*/

{
   if (head == NULL || head->next ==NULL) return;
   recursive_quick_sort(head, head->next);
}
```

P3. *Because it may involve more overhead, quicksort may be inferior to simpler methods for short lists. Through experiment, find a value where, on average for lists in random order, quicksort becomes more efficient than insertion sort. Write a hybrid sorting function that starts with quicksort and, when the sublists are sufficiently short, switches to insertion sort. Determine if it is better to do the switch-over within the recursive function or to terminate the recursive calls when the sublists are sufficiently short to change methods and then at the very end of the process run through insertion sort once on the whole list.*

Answer Both methods of this hybrid sort perform very quickly and each performs at very similar speed. The first method performs insertion sort only on lists of size switch_point or smaller, but these sorts are performed many times. The second method does only one insertion sort with the entire list, but the list is almost sorted and the insertion sort function works fairly quickly. Both methods require few modifications, as listed in the following. The first method requires a modification to the insertion sort function itself to ensure that only part of the list is sorted.

The implementation where the insertion sorts are applied to all small parts of the list as they are produced by quicksort is:

```
template <class Record>
void Sortable_list<Record>::insertion_sort(int low, int high)
/*
Post: The entries of the Sortable_list between low and high
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.  */

{
   int first_unsorted; //  position of first unsorted entry
   int position;       //  searches sorted part of list
   Record current;     //  holds the entry temporarily removed from list

   for (first_unsorted = low + 1; first_unsorted <= high; first_unsorted++)
      if (entry[first_unsorted] < entry[first_unsorted - 1]) {
         position = first_unsorted;
         current = entry[first_unsorted];
         do {
            entry[position] = entry[position - 1];
            position--;                          //  position is empty.
         } while (position > low && entry[position - 1] > current);
         entry[position] = current;
      }
}

template <class Record>
void Sortable_list<Record>::recursive_hybrid(int low, int high, int offset)
/*
Pre:  low and high are valid positions in the Sortable_list.
Post: The entries of the Sortable_list have been
      rearranged so that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort, and partition.
*/
```

```
{
   int pivot_position;
   if (low + offset < high) {   //   Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      recursive_hybrid(low, pivot_position - 1, offset);
      recursive_hybrid(pivot_position + 1, high, offset);
   }
   else insertion_sort(low, high);
}

template <class Record>
void Sortable_list<Record>::hybrid(int offset)
/*
Post: The entries of the Sortable_list have been rearranged so
      that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of Chapter 6,
      recursive_quick_sort.
*/

{
   recursive_hybrid(0, count - 1, offset);
}
```

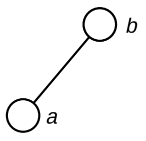
The implementation where the insertion sort is applied once at the end of the method is:

```
template <class Record>
void Sortable_list<Record>::recursive_quick_sort(int low, int high,
                                                 int offset)
/*

Pre:  low and high are valid positions in the Sortable_list.
Post: The entries of the Sortable_list have been
      rearranged so that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort, and partition.
*/

{
   int pivot_position;
   if (low + offset < high) {   //   Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      recursive_quick_sort(low, pivot_position - 1, offset);
      recursive_quick_sort(pivot_position + 1, high, offset);
   }
}

template <class Record>
void Sortable_list<Record>::quick_sort(int offset)
/*
Post: The entries of the Sortable_list have been rearranged so
      that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      recursive_quick_sort.
*/

{
   recursive_quick_sort(0, count - 1, offset);
   insertion_sort();
}
```

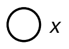
8.9 HEAPS AND HEAPSORT

Exercises 8.9

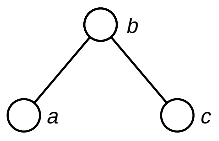
E1. *Show the list corresponding to each of the following trees under the representation that the children of the entry in position k are in positions $2k+1$ and $2k+2$. Which of these are heaps? For those that are not, state the position in the list at which the heap property is violated.*



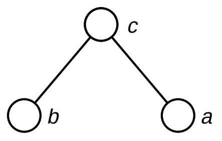
(a)



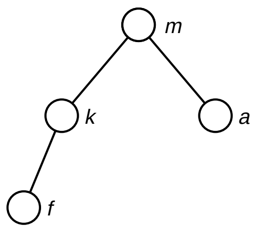
(b)



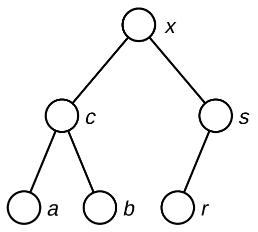
(c)



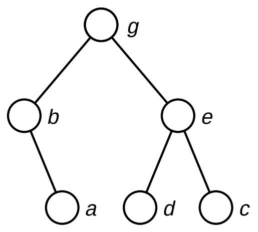
(d)



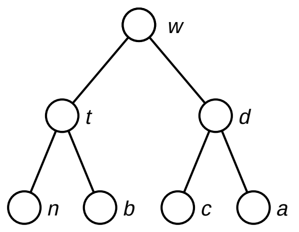
(e)



(f)



(g)



(h)

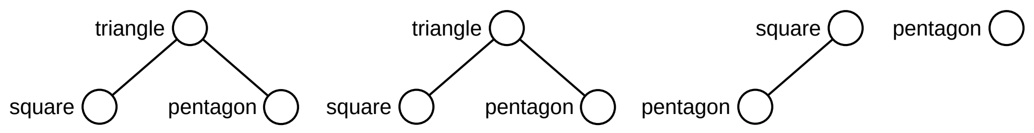
Answer The corresponding lists are: **(a)** b a **(b)** x **(c)** b a c **(d)** c b a **(e)** m k a f **(f)** x c s a b r **(g)** g b e – a d c **(h)** w t d n b c a. All these trees are heaps with exception of tree **(c)**, for which key *b* in the root is less than *c*, and tree **(g)**, for which there is one unoccupied entry in the list.

E2. *By hand, trace the action of* heap_sort *on each of the following lists. Draw the initial tree to which the list corresponds, show how it is converted into a heap, and show the resulting heap as each entry is removed from the top and the new entry inserted.*

(a) *The following three words to be sorted alphabetically:*

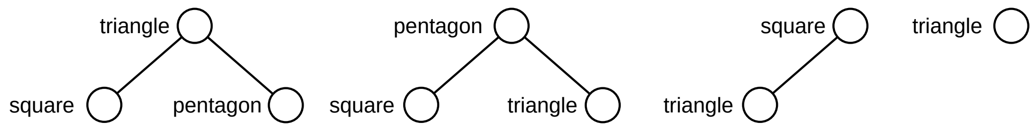
triangle square pentagon

Answer



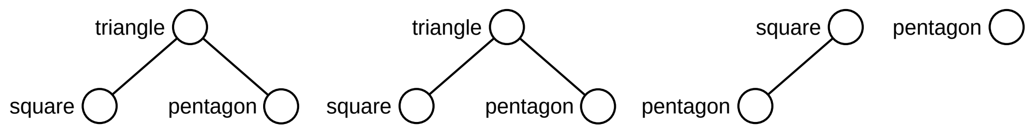
(b) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in increasing order*

Answer



(c) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in decreasing order*

Answer

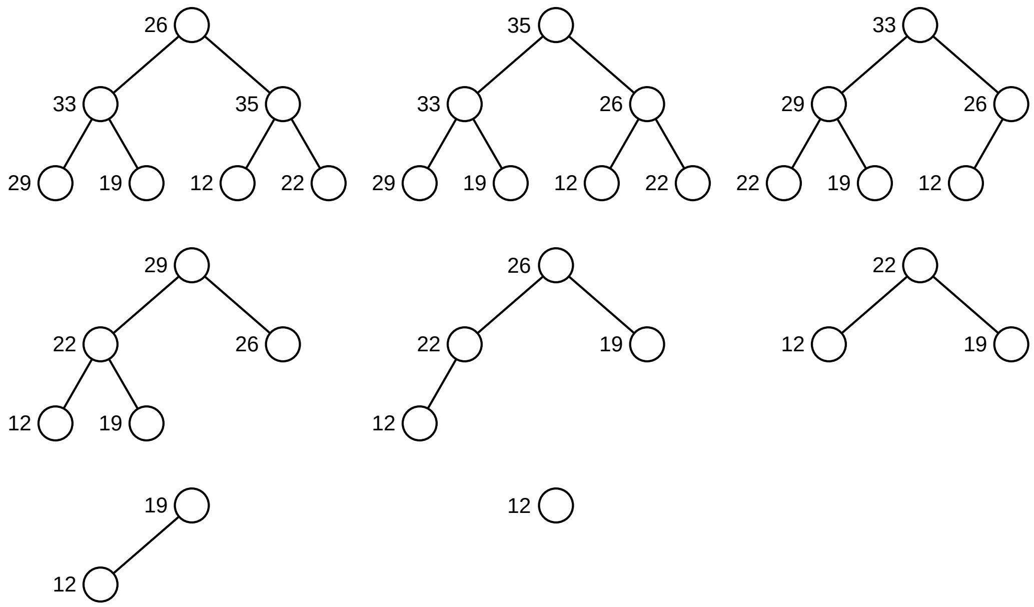


TOC
Index
Help

(d) *The following seven numbers to be sorted into increasing order:*

26 33 35 29 19 12 22

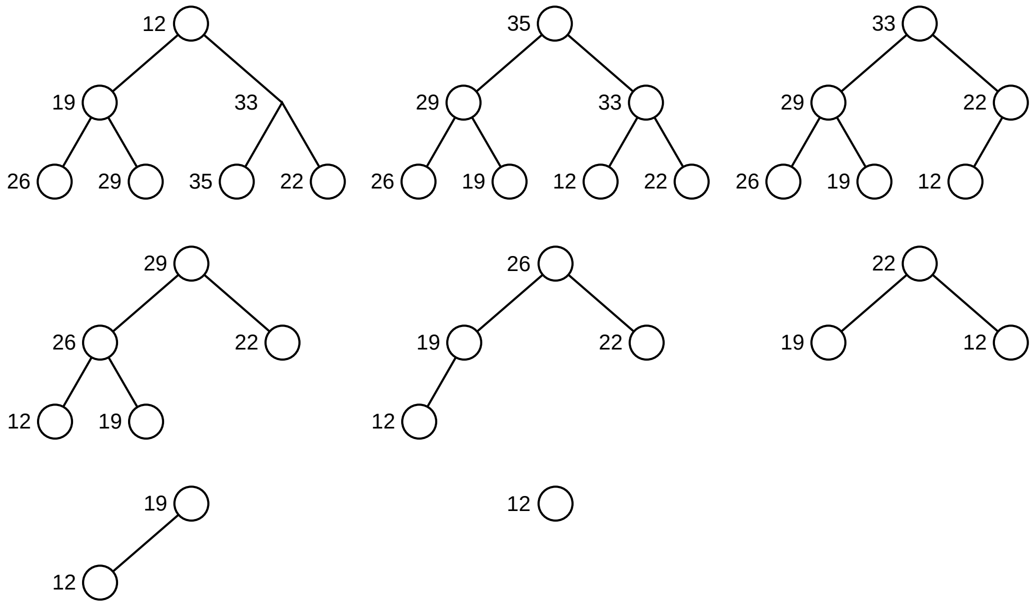
Answer



(e) *The same seven numbers in a different initial order, again to be sorted into increasing order:*

12 19 33 26 29 35 22

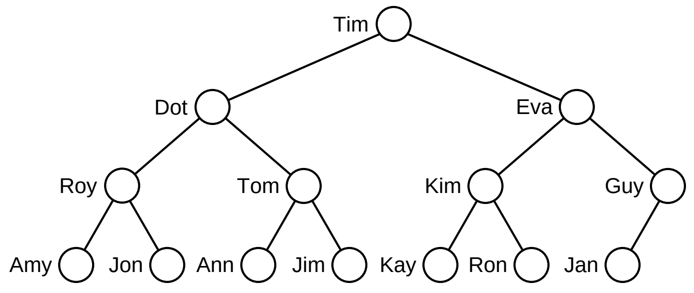
Answer

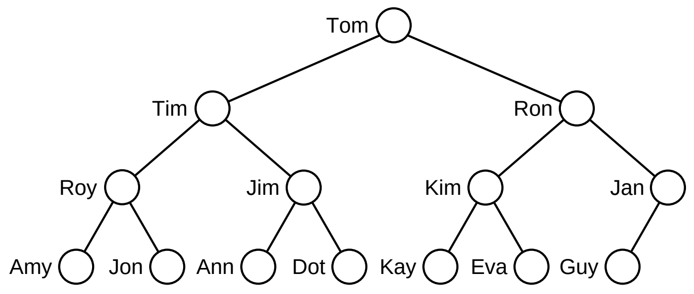


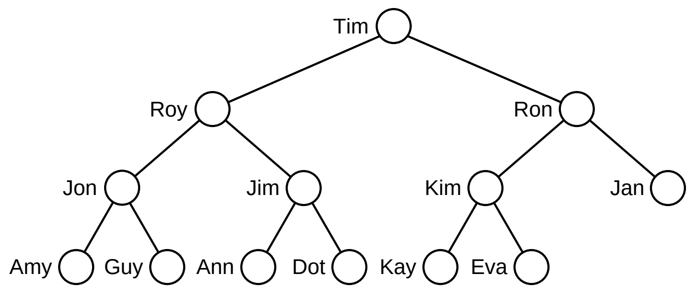
(f) *The following list of 14 names to be sorted into alphabetical order:*

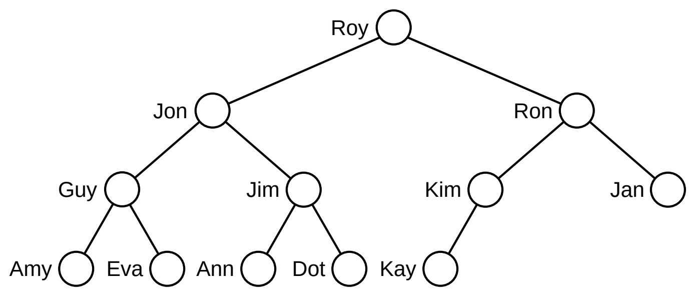
Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

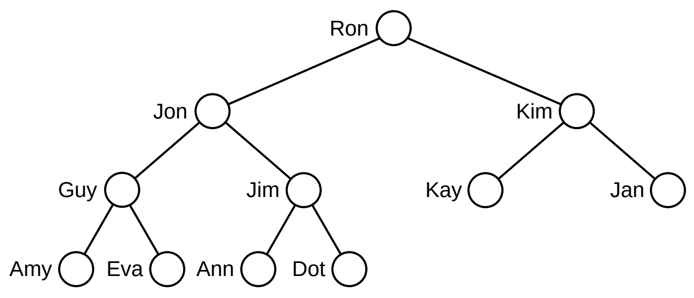
Answer

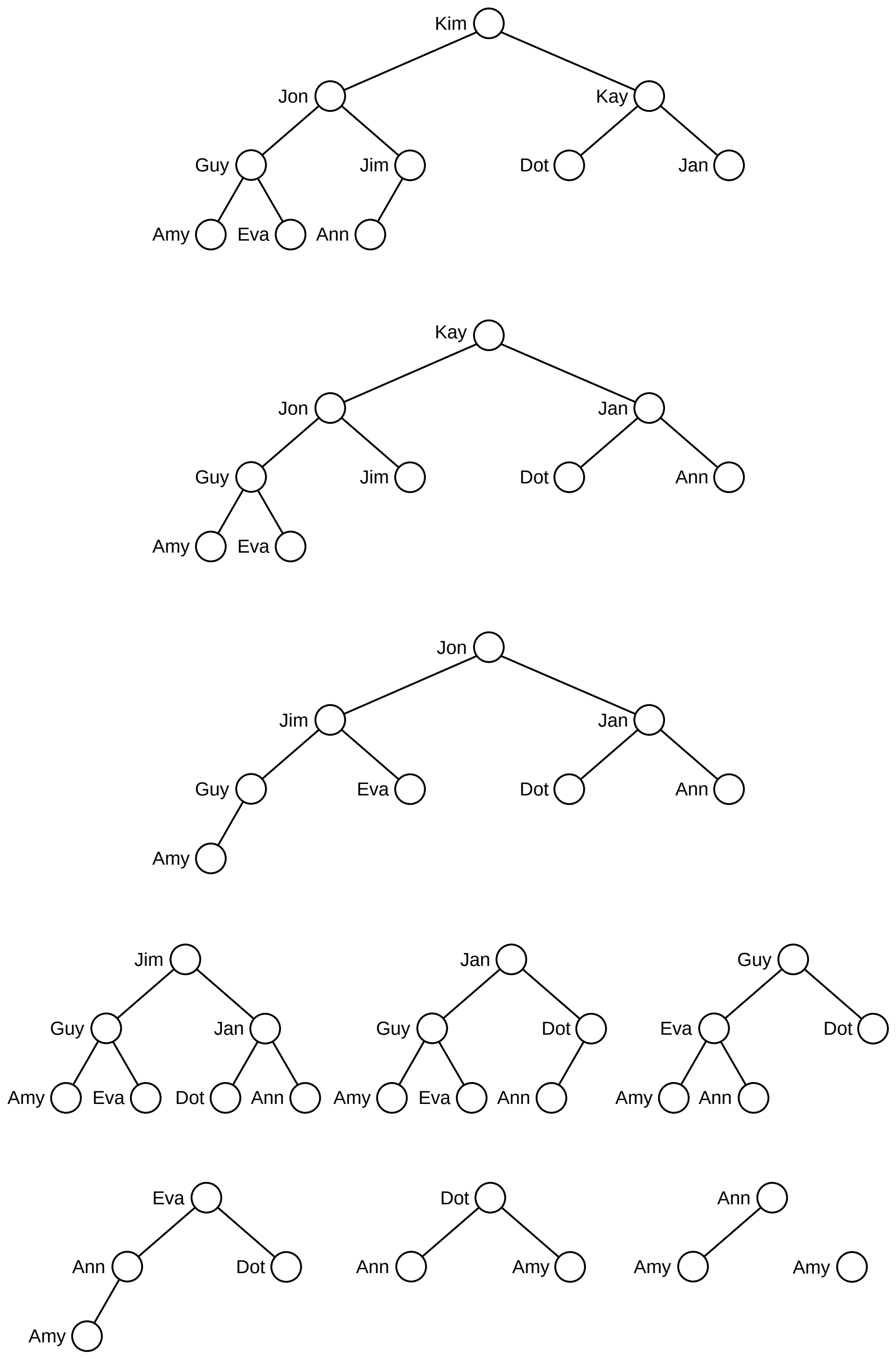

Tim
Dot
Eva
Roy
Tom
Kim
Guy
Amy
Jon
Ann
Jim
Kay
Ron
Jan


Tom
Tim
Ron
Roy
Jim
Kim
Jan
Amy
Jon
Ann
Dot
Kay
Eva
Guy


Tim
Roy
Ron
Jon
Jim
Kim
Jan
Amy
Guy
Ann
Dot
Kay
Eva


Roy
Jon
Ron
Guy
Jim
Kim
Jan
Amy
Eva
Ann
Dot
Kay


Ron
Jon
Kim
Guy
Jim
Kay
Jan
Amy
Eva
Ann
Dot



TOC
Index
Help

E3. (a) *Design a function that will insert a new entry into a heap, obtaining a new heap. (The function* insert_heap *in the text requires that the root be unoccupied, whereas, for this exercise, the root will already contain the entry with largest key, which must remain in the heap. Your function will increase the count of entries in the list.)*

Answer

```
template <class Record>
void Sortable_list<Record>::append_heap(const Record &x)
/* Post: The Record x has been added to the heap maintaining heap properties. */
```

```
{
   int low = (size() − 1)/2;
   insert(size(), x);                          //   add x at the end of the heap
                         //   now restore heap properties
   while (low >= 0) {
      Record add_in = entry[low];
      insert_heap(add_in, low, size() − 1);
      low = (low + 1)/2 − 1;
   }
}
```

(b) *Analyze the time and space requirements of your function.*

Answer The function used to add an item to the heap requires a constant amount of memory for the temporary storage of the new item plus loop control variables. The while loop iterates $\lg n$ times calling insert_heap, an $O(\log n)$ function, each time. The total time required is therefore $O((\log n)^2)$.

E4. (a) *Design a function that will delete the entry with the largest key (the root) from the top of the heap and restore the heap properties of the resulting, smaller list.*

Answer

```
template <class Record>
Error_code Sortable_list<Record> :: remove_heap_root()
/* Post: The root of the heap has been discarded and heap properties have been restored. */
{
   if (count <= 0) return underflow;
   insert_heap(entry[count − 1], 0, size() − 2);
   count−−;
   return success;
}
```

(b) *Analyze the time and space requirements of your function.*

Answer This function performs only one call to the function insert_heap requiring $O(\log n)$ time and only a small constant amount of memory to execute.

E5. (a) *Design a function that will delete the entry with index i from a heap and restore the heap properties of the resulting, smaller list.*

Answer

```
template <class Record>
Error_code Sortable_list<Record> :: remove(int i)
/* Post: The entry of the heap at position i has been discarded and heap properties have been
         restored. */
{
   if (size() <= 0) return underflow;
   if (i < 0 || i >= size()) return range_error;
   insert_heap(entry[size() − 1], i, size() − 2);
   count−−;
   int parent = (i − 1)/2;
   while (parent >= 0) {
      insert_heap(entry[parent], parent, size() − 1);
      parent = (parent + 1)/2 − 1;
   }
   return success;
}
```

(b) *Analyze the time and space requirements of your function.*

Answer This function requires only a small constant amount of memory. The main **while** loop will iterate $O(\log n)$ times calling insert_heap, which requires $O(\log n)$ time for each call. The algorithm therefore requires $O((\log n)^2)$ time for execution.

E6. *Consider a heap of n keys, with x_k being the key in position k (in the contiguous representation) for $0 \le k < n$. Prove that the height of the subtree rooted at x_k is the greatest integer not exceeding $\lg(n/(k+1))$, for all k satisfying $0 \le k < n$. [Hint: Use "backward" induction on k, starting with the leaves and working back toward the root, which is x_0.]*

Answer We denote the greatest integer not exceeding x by $\lfloor x \rfloor$ and the least integer not less than x by $\lceil x \rceil$. According to the conditions for the contiguous representation of a heap, a node x_i is a leaf if and only if $2i+1 \ge n$, which means $(i+1) \ge \lceil (n+1)/2 \rceil$. For $\lceil (n+1)/2 \rceil \le (k+1) \le n$, we have $\frac{1}{2} < (k+1)/n \le 1$. Therefore, $-1 < \lg((k+1)/n) \le 0$, so $0 \le \lg(n/(k+1)) < 1$. This implies that $\lfloor \lg(n/(k+1)) \rfloor = 0$, the height of the subtree rooted at x_k. Now suppose the result holds for all $k > f$, where $0 \le f \le \lceil (n+1)/2 \rceil - 2 = \lfloor n/2 \rfloor - 1$. We must show that the result holds for f. The height of the subtree rooted at x_f is one greater than the height of the subtree rooted at x_{2f+1}. By induction hypothesis, the height of the tree rooted at x_{2f+1} is $\lfloor \lg(n/(2f+2)) \rfloor$, and we therefore have $1 + \lfloor \lg(n/(2f+2)) \rfloor = \lfloor \lg(n/(f+1)) \rfloor$ for the height of a tree rooted at x_f. Therefore, the height of the subtree rooted at x_k is $\lfloor \lg(n/(k+1)) \rfloor$ for all k, $0 \le k < n$.

E7. *Define the notion of a ternary heap, analogous to an ordinary heap except that each node of the tree except the leaves has three children. Devise a sorting method based on ternary heaps, and analyze the properties of the sorting method.*

Answer A ternary heap is a completely balanced ternary tree of height h, in which all leaves are at a distance h or $h-1$ from the root, such that the keys in all descendants of a node are smaller than the key in the node itself. Furthermore, all leaves at level h are as far to the left as possible. A ternary heap can be stored by levels in a one-dimensional array to yield a linear representation of the tree. The children of the key in the i^{th} position are the keys in positions $3i+1$, $3i+2$, and $3i+3$.

The outline of heapsort given in the text will work for ternary heapsort, but the restoration process in insert_heap must be modified slightly by comparing x_j to the largest of its as many as three children. In this case x_j is a leaf of the heap $x_0, x_1, \ldots, x_l$ if and only if $j > \lfloor (l-1)/3 \rfloor$. The only other change needed to obtain the algorithm for ternary heapsort is to replace (count/2 − 1) by (count/3 − 1) in the **for** loop in build_heap.

If h is the height of the subtree rooted at x_f, then the modified insert_heap will perform at most $3h$ key comparisons and at most h swaps of items. We then find that $h = \lfloor \log_3 (l+1)/(f+1) \rfloor$. Therefore, the build_heap section of ternary heapsort requires at most

$$\sum_{i=1}^{\lfloor \frac{1}{3}n \rfloor} \left\lfloor \log_3 \frac{n}{i} \right\rfloor \le \sum_{i=1}^{\lfloor \frac{1}{3}n \rfloor} \log_3 \frac{n}{i} = \left\lfloor \frac{1}{3}n \right\rfloor \log_3 n - \log_3 \left\lfloor \frac{1}{3}n \right\rfloor ! = O(n)$$

swaps and hence $O(n)$ key comparisons. Similarly, the main loop of the heap_sort method requires at most

$$(n-1) + \sum_{i=2}^{n} \lfloor \log_3 (i-1) \rfloor = n \log_3 n + O(n)$$

swaps and hence only $3n \log_3 n + O(n)$ key comparisons, since the largest of the three children of a node can be found in two comparisons.

Programming Project 8.9

P1. *Implement heapsort (for contiguous lists) on your computer, integrate it into the test program of Project P1 of Section 8.2 (page 328), and compare its performance with all the previous sorting algorithms.*

Answer For test results, see the tables in Project P6 of Section 8.10.

TOC
Index
Help

```
template <class Record>
void Sortable_list<Record>::insert_heap(const Record &current,
                                        int low, int high)
/*
Pre:  The entries of the Sortable_list between indices low + 1 and high,
      inclusive, form a heap. The entry in position low will be discarded.
Post: The entry current has been inserted into the Sortable_list
      and the entries rearranged
      so that the entries between indices low and high, inclusive,
      form a heap.
Uses: The class Record, and the contiguous List implementation of
      ?list_ch?.
*/

{
   int large;     //  position of child of entry[low] with the larger key

   large = 2 * low + 1;//  large is now the left child of low.
   while (large <= high) {
      if (large < high && entry[large] < entry[large + 1])
         large++;    //  large is now the child of low with the largest key.
      if (current >= entry[large])
         break;         //  current belongs in position low.

      else {            //  Promote entry[large] and move down the tree.
         entry[low] = entry[large];
         low = large;
         large = 2 * low + 1;
      }
   }
   entry[low] = current;
}

template <class Record>
void Sortable_list<Record>::build_heap()
/*
Post: The entries of the Sortable_list have been rearranged so
      that it becomes a heap.
Uses: The contiguous List implementation of ?list_ch?, and insert_heap.
*/

{
   int low;   //  All entries beyond the position low form a heap.
   for (low = count / 2 - 1; low >= 0; low--) {
      Record current = entry[low];
      insert_heap(current, low, count - 1);
   }
}

template <class Record>
void Sortable_list<Record>::heap_sort()
/*
Post: The entries of the Sortable_list have been rearranged so
      that their keys are sorted into nondecreasing order.
Uses: The contiguous List implementation of ?list_ch?,
      build_heap, and insert_heap.
*/
```

```
{
   Record current;    //  temporary storage for moving entries
   int last_unsorted; //  Entries beyond last_unsorted have been sorted.
   build_heap();      //  First phase:  Turn the list into a heap.
   for (last_unsorted = count - 1; last_unsorted > 0; last_unsorted--) {
      current = entry[last_unsorted]; // Extract last entry from the list.
      entry[last_unsorted] = entry[0];     //  Move top of heap to the end
      insert_heap(current, 0, last_unsorted - 1);  //  Restore the heap
   }
}
```

8.10 REVIEW: COMPARISON OF METHODS

Exercises 8.10

E1. *Classify the sorting methods we have studied into one of the following categories: (a) The method does not require access to the entries at one end of the list until the entries at the other end have been sorted; (b) The method does not require access to the entries that have already been sorted; (c) The method requires access to all entries in the list throughout the process.*

Answer The methods are (a) Insertion sort and merge sort, (b) Selection sort and heap sort, and (c) Quicksort and Shell sort.

E2. *Some of the sorting methods we have studied are not suited for use with linked lists. Which ones, and why not?*

Answer Most sorting methods studied can be written for linked lists without great difficulty. Shell sort and heapsort are the exceptions, as they require random access to the list entries at different intervals.

E3. *Rank the sorting methods we have studied (both for linked and contiguous lists) according to the amount of extra storage space that they require for indices or pointers, for recursion, and for copies of the entries being sorted.*

Answer The sorting methods insertion, selection, Shell, and heapsort all require very little extra storage. The linked version of merge sort requires storage on the system stack due to recursion; however, the depth of recursion is limited to lg n. Quicksort requires anywhere between lg n and n extra storage on the system stack due to recursion. The method that would take the most extra memory would be the contiguous version of merge sort, which requires an auxiliary list for merging as well as the system stack for recursion.

E4. *Which of the methods we studied would be a good choice in each of the following applications? Why? If the representation of the list in contiguous or linked storage makes a difference in your choice, state how.*

(a) *You wish to write a general-purpose sorting program that will be used by many people in a variety of applications.*

Answer Shell sort is one of the faster of the sorting methods covered; it does not suffer very much during a worst case; it performs fairly well with almost sorted list; and it does not require very much extra storage. It is therefore a good general purpose sorting method. If the list is linked, mergesort would be excellent.

(b) *You wish to sort 1000 numbers once. After you finish, you will not keep the program.*

Answer The best choice would be either insertion or Shell sort. As shown in the comparison tables in question E6 in this section, insertion sort on a list of 1000 items required 6.1 seconds, while Shell sort took 0.31 seconds on the same list. Since you are not going to keep the program, the best choice would be something that is easy to code. Insertion sort would best fit this description. Therefore, the choice comes down to coding time versus running time, with each of these two methods showing definite advantages over all others.

(c) *You wish to sort 50 numbers once. After you finish, you will not keep the program.*

Answer In this case, the most important consideration is ease of coding. Therefore, insertion or selection sort would be equally viable choices.

(d) *You need to sort 5 entries in the middle of a long program. Your sort will be called hundreds of times by the long program.*

Answer Insertion sort works with few comparisons and swaps for very small lists, such as $n = 5$.

(e) *You have a list of 1000 keys to sort in high-speed memory, and key comparisons can be made quickly, but each time a key is moved, a corresponding 500 block file on disk must also be moved, and doing so is a slow process.*

Answer Selection sort achieves the minimum number of swaps.

(f) *There is a twelve foot long shelf full of computer science books all catalogued by number. A few of these have been put back in the wrong places by readers, but rarely are the books more than one foot from where they belong.*

Answer Insertion sort works well with lists that are close to being sorted.

(g) *You have a stack of 500 library index cards in random order to sort alphabetically.*

Answer Quicksort works well under random conditions and with structure that essentially contiguous, such as a stack of cards. In working by hand, it is often quicker to partition the cards into several piles, not just two.

(h) *You are told that a list of 5000 words is already in alphabetical order, but you wish to check it to make sure, and sort any words found out of order.*

Answer Insertion sort is optimal for sorted lists.

E5. *Discuss the advantages and disadvantages of designing a general sorting function as a hybrid between quicksort and Shell sort. What criteria would you use to switch from one to the other? Which would be the better choice for what kinds of lists?*

Answer Neither quicksort nor Shell sort is very good for small lists, so it makes little sense to form a hybrid with the two. It would be much better to form a hybrid between quicksort and insertion sort, as in as insertion sort works well with small lists. A hybrid between Shell sort and insertion sort makes no sense, since Shell sort already does insertion sort as its last step.

E6. *Summarize the results of the test runs of the sorting methods of this chapter for your computer. Also include any variations of the methods that you have written as exercises. Make charts comparing the following:*

(a) *the number of key comparisons.*

Answer

Method	10	50	100	500	1000
Insertion	33	711	2553	63987	255025
Linked Insertion	33	712	2559	64017	254993
Insertion / sentinel	33	711	2553	63987	255026
Binary Insertion	28	246	602	4276	9577
Scan Sort	57	1373	5007	127475	509053
Bubble Sort	45	1225	4950	124750	499500
Selection	45	1225	4950	124750	499500
Linked Selection	45	1227	4955	124960	499980
Shell Sort	30	329	720	6066	13414
Linked Merge	21	221	547	3939	9023
Merge with counter	21	224	541	4209	9619
Natural Merge 1	34	544	1902	43188	169530
Natural Merge 2	26	258	610	6275	17721
Merge / contiguous	21	222	544	3896	8735
Quicksort	19	258	643	4630	10028
Linked Quicksort	30	387	885	6757	13340
Hybrid 1	33	302	774	5506	11220
Hybrid 2	42	351	873	6005	12219
Heapsort	42	428	1040	7408	16864

Number of comparisons done by various sorting methods

(b) *the number of assignments of entries.*

Answer

Method	10	50	100	500	1000
Insertion	38	748	2640	63474	256013
Insertion / sentinel	39	749	2641	64475	256014
Binary Insertion	42	780	2715	64899	257102
Scan Sort	72	1986	7362	190464	762081
Bubble Sort	72	1986	7362	190464	762081
Selection	27	147	297	1497	2997
Shell Sort	26	319	717	6208	14413
Merge / contiguous	40	300	800	5000	10000
Quicksort	60	489	1467	8655	18684
Hybrid 1	38	342	1096	7668	15664
Hybrid 2	39	343	1097	7669	15665
Heapsort	26	246	580	4039	9071

Number of assignments made by various contiguous sorting methods

(c) *the total running time.*

Answer

Method	50	100	500	1000
Insertion	0.02	0.06	1.59	6.37
Linked Insertion	0.01	0.03	0.36	1.52
Insertion / sentinel	0.02	0.06	1.39	5.58
Binary Insertion	0.02	0.06	1.12	4.34
Scan Sort	0.03	0.11	2.66	10.61
Bubble Sort	0.03	0.12	3.01	12.04
Selection	0.02	0.08	1.92	7.77
Linked Selection	0.01	0.04	0.95	4.18
Shell Sort	0.01	0.02	0.15	0.33
Distribution	0.01	0.01	0.04	0.08
Linked Merge	0.01	0.02	0.10	0.22
Merge / counter	0.01	0.02	0.10	0.20
Natural Merge 1	0.01	0.03	0.45	1.77
Natural Merge 2	0.01	0.02	0.11	0.24
Merge / contiguous	0.02	0.04	0.21	0.42
Quicksort	0.02	0.04	0.22	0.46
Linked Quicksort	0.01	0.02	0.11	0.23
Hybrid 1	0.01	0.02	0.13	0.26
Hybrid 2	0.01	0.02	0.14	0.27
Heapsort	0.01	0.03	0.20	0.45

Comparison of running times on various sorting methods

(d) *the working storage requirements of the program.*
(e) *the length of the program.*
(f) *the amount of programming time required to write and debug the program.*

Answer

Method	Storage	Length	Effort
Insertion	$O(1)$	24	1
Linked Insertion	$O(1)$	38	2
Insertion / sentinel	$O(1)$	21	1
Binary Insertion	$O(1)$	30	2
Scan Sort	$O(1)$	22	1
Bubble Sort	$O(1)$	12	1
Selection	$O(1)$	34	1
Linked Selection	$O(1)$	39	2
Shell Sort	$O(1)$	40	2
Distribution	$O(n)$	59	4
Linked Merge	$O(\log n)$	64	4
Merge / counter	$O(\log n)$	80	5
Natural Merge 1	$O(1)$	74	4
Natural Merge 2	$O(\log n)$	128	5
Merge / Contiguous	$O(n)$	75	5
Quicksort	$O(\log n)$	33	4
Linked Quicksort	$O(\log n)$	68	4
Hybrid 1	$O(\log n)$	46	4
Hybrid 2	$O(\log n)$	47	4
Heapsort	$O(1)$	40	4

This last table compares the tested methods in terms of running storage requirements for n items in a list, program length in lines, and programming effort rated on a scale from 1 for the easiest to 5 for the highest degree of effort.

E7. *Write a one-page guide to help a user of your computer system select one of our sorting algorithms according to the desired application.*

Answer The methods we shall consider are insertion sort, selection sort, Shell sort, mergesort, quicksort, and heapsort.

The first two of these are the easiest to write and are very efficient for short lists but usually not for long lists. The amount of work they do goes up like n^2, where n is the length of the list, whereas the more sophisticated methods do work that is $O(n \log n)$. If a sorting method is to be used only a few times, then one of these simple methods should usually be chosen. They are also better for lists of length about 20 or less, even if such lists must be sorted many times. Insertion sort does fewer comparisons, and selection sort does fewer movements of items. Insertion sort checks that a list is in correct order as quickly as can be done, and it is an excellent choice for lists that are already nearly sorted, even if they are very long. Selection sort achieves the smallest possible number of movements of items, and it is therefore an excellent choice, even for longer lists, if movement of items is extremely slow or expensive.

Shell sort is a good general-purpose sorting method for longer lists. It is based on insertion sort and is perhaps easier to write than the other efficient methods. It is, however, somewhat slower than the remaining methods and is not suitable for linked lists.

Mergesort comes very close to achieving the smallest possible number of comparisons of keys. It is extremely efficient in regard to time and is an excellent choice, especially for linked lists and for external sorting. The most obvious and straightforward implementation of mergesort for contiguous lists, however, requires auxiliary space equal to that needed for the original list. This need for extra space often limits its usefulness for sorting contiguous lists.

Quicksort is often the best choice for sorting contiguous lists. It does, on average, about 39 percent more comparisons of keys than mergesort, but it has minimal requirements for extra space. In its worst case, quicksort degenerates to become slower than insertion sort and selection sort, but a careful choice of the pivot makes this worst case extremely unlikely. The construction and traversal of a binary search tree yields a sorting method that shares the same underlying method with quicksort.

Heapsort is something of an insurance policy. It usually takes longer than quicksort, but avoids the slight possibility of a catastrophic failure by sorting a list of length n in time $O(n \log n)$, even in its worst case. Heapsort is usually implemented only for contiguous lists.

E8. *A sorting function is called* ***stable*** *if, whenever two entries have equal keys, then on completion of the sorting function, the two entries will be in the same order in the list as before sorting. Stability is important if a list has already been sorted by one key and is now being sorted by another key, and it is desired to keep as much of the original ordering as the new one allows. Determine which of the sorting methods of this chapter are stable and which are not. For those that are not, produce a list (as short as possible) containing some entries with equal keys whose orders are not preserved. In addition, see if you can discover simple modifications to the algorithm that will make it stable.*

stable sorting methods

Answer Shell sort is not stable and cannot easily be made stable. Adjacent items will appear on different sublists in the early stages of Shell sort, and may be moved in different directions in early passes. Suppose, for example, that the initial list has keys 3 1 1 and the increment is 2. Then 3 is swapped with the second 1, and the next pass makes no change, so the order of the two 1's is reversed.

Heapsort is also not stable. A list consisting of two equal keys already forms a heap, and function build_heap will make no change, but heap_sort will then swap the two keys.

Insertion sort, selection sort, and mergesort are stable, and quicksort is stable if it is given a stable partition algorithm. The partition algorithm developed in the text, however, is not stable. If the keys in the list are initially 1 2 1 (so the pivot is 2) then the pivot is first swapped into the first position and then at the end it is swapped with the second 1, so the order of the two 1's is reversed.

REVIEW QUESTIONS

1. *How many comparisons of keys are required to verify that a list of n entries is in order?*

   Verifying a that list of n entries is in the correct order requires $n - 1$ comparisons of keys.

2. *Explain in twenty words or less how insertion sort works.*

   Starting with the second key, each entry is extracted and placed in relative order with the entries preceding it.

3. *Explain in twenty words or less how selection sort works.*

   Starting with the last position, the largest unsorted key is swapped into the last unsorted position.

4. *On average, about how many more comparisons does selection sort do than insertion sort on a list of 20 entries?*

   On average, selection sort does about 100 more comparisons than insertion sort on a list of 20 entries.

5. *What is the advantage of selection sort over all the other methods we studied?*

   Selection does not move as many keys as the other methods.

6. *What disadvantage of insertion sort does Shell sort overcome?*

   Insertion sort moves entries only one position at a time; Shell sort quickly moves them closer to their final position.

7. *What is the lower bound on the number of key comparisons that any sorting method must make to put n keys into order, if the method uses key comparisons to make its decisions? Give both the average- and worst-case bounds.*

   Any algorithm that sorts a list of n entries by use of key comparisons must, in its worst case, perform at least $\lceil \lg n! \rceil$ comparisons of keys, and, in the average case, it must perform at least $\lg n!$ comparisons of keys.

8. *What is the lower bound if the requirement of using comparisons to make decisions is dropped?*

   The lower bound on time used for sorting without using comparison of keys is $O(n)$.

9. *Define the term divide and conquer.*

   Divide-and-conquer is the idea of dividing a problem into smaller but similar subproblems that are easier to solve.

10. *Explain in twenty words or less how mergesort works.*

    The list is divided into sublists, each sublist is sorted, then the sublists are merged together in proper order.

11. *Explain in twenty words or less how quicksort works.*

    The list is repeatedly partitioned into sublists with all the keys in the first sublist preceding those in the second.

12. *Explain why mergesort is better for linked lists than for contiguous lists.*

Mergesort is better for linked lists as merging two linked lists is easier and requires less storage than merging two contiguous lists.

13. *In quicksort, why did we choose the pivot from the center of the list rather than from one of the ends?*

Choosing the pivot from the center will lessen the chance of quicksort happening into its worst case, as would occur if the pivot is selected from one of the ends in a sorted or nearly-sorted list.

14. *On average, about how many more comparisons of keys does quicksort make than the optimum? About how many comparisons does it make in the worst case?*

On average, quicksort does about 39 percent more comparisons of keys than the optimum. In its worst case, quicksort does about $O(\frac{1}{2}n^2)$ comparisons of keys.

15. *What is a heap?*

A heap is a list with a key in each entry, such that the key in entry k is at least as large as those in entries $2k + 1$ and $2k + 2$, provided those entries exist.

16. *How does heapsort work?*

The entries in the list to be sorted are interpreted as a 2-tree in contiguous implementation. This representation of a 2-tree is then converted into a heap. Then the root, the largest key in the heap, is repeatedly swapped with the entry at the end of the list and the heap property is restored with the shortened list. When this process is completed, the list is sorted.

17. *Compare the worst-case performance of heapsort with the worst-case performance of quicksort, and compare it also with the average-case performance of quicksort.*

The worst-case performance of heapsort is $O(n \lg n)$ as opposed to the worst-case performance of $O(n^2)$ for quicksort. The average-case time needed for heapsort is less than twice that of quicksort.

18. *When are simple sorting algorithms better than sophisticated ones?*

Simple sorting methods are generally better than more sophisticated methods when the lists used are fairly small or the algorithm will not be used repeatedly.

Tables and Information Retrieval

9

9.2 RECTANGULAR TABLES

Exercises 9.2

E1. *What is the index function for a two-dimensional rectangular table whose rows are indexed from 0 to $m - 1$ and whose columns are indexed from 0 ot $n - 1$, inclusive, under column-major ordering?*

Answer Entry (i, j) goes to position $i + mj$.

E2. *Give the index function, with row-major ordering, for a two-dimensional table with arbitrary bounds r to s, inclusive, for the row indices, and t to u, inclusive, for the column indices.*

Answer Entry (i, j) goes to entry $(u - t + 1)(i - r) + (j - t)$.

E3. *Find the index function, with the generalization of row-major ordering, for a table with d dimensions and arbitrary bounds for each dimension.*

Answer Assume the array declaration

$$\text{array}(a_1..b_1, \ldots, a_d..b_d).$$

Let r_k be the size of the array in the k^{th} dimension, so $r_k = b_k - a_k + 1$. Then, if we begin the sequential representation at position 0, entry $(i_1, \ldots, i_d)$ goes to the position

$$(\ldots(((i_1 - a_1)r_2 + i_2 - a_2)r_3 + i_3 - a_3)\ldots)r_d + i_d - a_d.$$

9.3 TABLES OF VARIOUS SHAPES

Exercises 9.3

E1. *The **main diagonal** of a square matrix consists of the entries for which the row and column indices are equal. A **diagonal matrix** is a square matrix in which all entries not on the main diagonal are 0. Describe a way to store a diagonal matrix without using space for entries that are necessarily 0, and give the corresponding index function.*

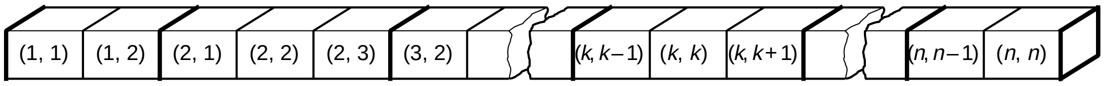
Answer Assuming that the diagonal matrix is of size (0 . . n, 0 . . n), all main diagonal entries can be stored in a one-dimensional array with an index of (0 . . n). Entry (i, i) goes to position i.

E2. *A **tri-diagonal matrix** is a square matrix in which all entries are 0 except possibly those on the main diagonal and on the diagonals immediately above and below it. That is, T is a tri-diagonal matrix means that $T(i, j) = 0$ unless $|i - j| \le 1$.*

(a) *Devise a space-efficient storage scheme for tri-diagonal matrices, and give the corresponding index function.*

Answer Each row of the tri-diagonal matrix contains three entries, with the exception of the first and last rows which contain only two entries.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & 0 & 0 & \dots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & \dots & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & a_{n,n} \end{pmatrix}$$



Stored in an array indexed from 0 to $3n-3$, with row-major ordering, the tri-diagonal matrix becomes:

$$a_{1,1} \quad a_{1,2} \quad a_{2,1} \quad a_{2,2} \quad a_{2,3} \quad a_{3,2} \quad a_{3,3} \quad a_{3,4} \quad \dots \quad a_{n,n-1} \quad a_{n,n}$$

Given the index (i, j), if $|i - j| \le 1$ then the value of $a_{i,j}$ is stored in position $2i + j - 3$ of the array, otherwise the value of $a_{i,j}$ is zero.

(b) *The **transpose** of a matrix is the matrix obtained by interchanging its rows with the corresponding columns. That is, matrix B is the transpose of matrix A means that $B(j, i) = A(i, j)$ for all indices i and j corresponding to positions in the matrix. Design an algorithm that transposes a tri-diagonal matrix using the storage scheme devised in the previous part of the exercise.*

Answer Transposing involves swapping all entries with the index (i, j) with entries (j, i). In a tri-diagonal matrix, only those entries above and below the main diagonal need be swapped. The following algorithm can be used:

```
x = 1;
while (x < 3n − 3) {
   swap position[x] with position[x + 1];
   x = x + 3;
}
```

E3. *An **upper triangular matrix** is a square matrix in which all entries below the main diagonal are 0. Describe the modifications necessary to use the access array method to store an upper triangular matrix.*

Answer An upper triangular matrix has the following form:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n} \\ 0 & a_{1,1} & \dots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{pmatrix}$$

This can be stored in a contiguous array, indexed from 0 to $\frac{1}{2}n(n+1)+n$, using column major ordering, as follows:

$$a_{0,0} \quad a_{0,1} \quad a_{1,1} \quad \dots \quad a_{0,n} \quad a_{1,n} \quad \dots \quad a_{n,n}$$

The access array is set up by the index function $\frac{1}{2}j(j+1)+i$. Access to the beginning of each column, provided $j \geq i$, is as follows:

$$0, \quad 1, \quad 3, \quad 6, \quad \dots, \quad \tfrac{1}{2}n(n+1).$$

E4. *Consider a table of the triangular shape shown in Figure 9.7, where the columns are indexed from $-n$ to n and the rows from 0 to n.*

(a) *Devise an index function that maps a table of this shape into a sequential array.*

Answer Entry (i, j) is stored in position $i(i+1)+j$ of a sequential array that is indexed from 0 to $n(n+2)$.

(b) *Write a function that will generate an access array for finding the first entry of each row of a table of this shape within the contiguous array.*

Answer The following function maps the position of the first entry in each row, not the position of the entry in column 0, to the access array.

```
void make_access_table(int access_table[ ], int number_of_rows)
{
  for (int i = 0; i < number_of_rows; i++)
    access_table[i] = i * i;
}
```

(c) *Write a function that will reflect the table from left to right. The entries in column 0 (the central column) remain unchanged, those in columns −1 and 1 are swapped, and so on.*

Answer

```
void reflect(int triangle[ ], int number_of_rows)
{
  for (int i = 0; i < number_of_rows; i++) {
    int mid_col = i * (i + 1);
    for (int j = 0; j < i; j++)
      swap(triangle[mid_col − j], triangle[mid_col + j]);
  }
}
```

Programming Projects 9.3

Implement the method described in the text that uses an access array to store a lower triangular table, as applied in the following projects.

P1. *Write a function that will read the entries of a lower triangular table from the terminal. The entries should be of type* **double**.

Answer The implementation of a class `Lower_triangle` with methods giving the functions for Projects P1, P2, P3 follows:

```
class Lower_triangle {
public:
    Lower_triangle(int side = 0);
    int get(int row, int col);
    void put(int row, int col, int value);
    void print();
    void read();
    bool test();
private:
    int side;
    int entry[(max_side * max_side + max_side) / 2];
    int access[max_side];
    void set_access(int s);
};
```

Implementation:

```
Lower_triangle::Lower_triangle(int s)
{
   if (s > max_side) s = max_side;
   if (s < 0) s = 0;
   side = s;
   set_access(s);
}

void Lower_triangle::set_access(int s)
{
   for (int i = 0; i < s; i++)
     access[i] = (i * i + i) / 2;
}

int Lower_triangle::get(int i, int j)
{
   if (j > i) {
      int temp = j;
      j = i;
      i = temp;
   }
   if (i >= side || j < 0) return 0;
   return entry[access[i] + j];
}

void Lower_triangle::put(int i, int j, int value)
{
   if (j > i) {
      int temp = j;
      j = i;
      i = temp;
   }
   if (i >= side || j < 0) return;
   entry[access[i] + j] = value;
}

void Lower_triangle::print()
/*
Post: display a lower triangular table
*/
```

```
{
    int i, j;
    for (i = 0; i < side; i++) {
        for (j = 0; j <= i; j++)
           cout << get(i,j) << " ";
        for (j = i + 1; j < side; j++)
           cout << "0 ";                  // print zeros above diagonal
        cout << endl;
    }
}

void Lower_triangle::read()
/*
Post: the entries of a lower triangular table are read
      from standard input
*/

{
    int i, j;
    do {
        cout << "How may rows? " << flush;
        cin  >> side;
        set_access(side);
    } while (side > max_side || side < 0);
    for (i = 0; i < side; i++) {
        cout << "\nEnter row " << i << " :" << flush;
        for (j = 0; j <= i; j++) {
           int value;
           cin >> value;
           put(i, j, value);
        }
    }
}

bool Lower_triangle::test()
/*
Post: tests the triangle rule
*/

{
    bool okay = true;
    int a, b, c;
    int ab, ac, bc;

    a = 0;
    while (okay && a < side) {
        c = 0;
        while (okay && c <= a) {
            b = 0;
            ac = get(a, c);
            while (okay && b < side) {   // test with intermediate cities
                ab = get(a, b);
                bc = get(b, c);
                okay = (ac <= ab + bc); // test triangle rule
                b++;
            }
            c++;
        }
        a++;
    }
```

```
        if (okay) {
           cout << "The triangle inequality holds." << endl;
           return true;
        }
        cout << "The Triangle inequality fails for entries "
             << a - 1 << " " << c - 1 << " with intermediate location "
             << b - 1 << endl;
        return false;
    }
```

P2. *Write a function that will print a lower triangular table at the terminal.*

Answer See the solution to Project P1.

P3. *Suppose that a lower triangular table is a table of distances between cities, as often appears on a road map. Write a function that will check the triangle rule: The distance from city A to city C is never more than the distance from A to city B, plus the distance from B to C.*

Answer See the solution to Project P1.

P4. *Embed the functions of the previous projects into a complete program for demonstrating lower triangular tables.*

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

const int max_side= 10;
#include "lower.h"
#include "lower.cpp"

int main()
{
    Lower_triangle t;
    cout << "Program to read a lower triangular table and \n"
         << "      check the triangle inequality." << endl;

    t.read();
    t.print();
    t.test();
}
```

9.5 APPLICATION: RADIX SORT

Exercises 9.5

E1. *Trace the action of radix sort on the list of 14 names used to trace other sorting methods:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

Unsorted:	Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan							
Letter 3:	Eva	Tim Tom Kim Jim	Jon Ann Ron Jan	Dot	Roy Guy Amy Kay			
Letter 2:	Jan Kay	Tim Kim Jim	Amy	Ann	Tom Jon Ron Dot Roy	Guy	Eva	
Letter 1:	Amy Ann	Dot	Eva	Guy	Jan Jim Jon	Kay Kim	Ron Roy	Tim Tom

E2. *Trace the action of radix sort on the following list of seven numbers considered as two-digit integers:*

26 33 35 29 19 12 22

Answer

Unsorted: 26 33 35 29 19 12 22

Digit 2: $\boxed{12\quad 22}$ $\boxed{33}$ $\boxed{35}$ $\boxed{26}$ $\boxed{29\quad 19}$

Digit 1: $\boxed{12\quad 19}$ $\boxed{22\quad 26\quad 29}$ $\boxed{33\quad 35}$

E3. *Trace the action of radix sort on the preceding list of seven numbers considered as six-digit binary integers.*

Answer

Unsorted: 011010 100001 100011 011101 010011 001100 010110

Digit 6: $\boxed{011010\ 001100\ 010110}$ $\boxed{100001\ 100011\ 011101\ 010011}$

Digit 5: $\boxed{001100\ 100001\ 011101}$ $\boxed{011010\ 010110\ 100011\ 010011}$

Digit 4: $\boxed{100001\ 011010\ 100011\ 010011}$ $\boxed{001100\ 011101\ 010110}$

Digit 3: $\boxed{100001\ 100011\ 010011\ 010110}$ $\boxed{011010\ 001100\ 011101}$

Digit 2: $\boxed{100001\ 100011\ 001100}$ $\boxed{010011\ 010110\ 011010\ 011101}$

Digit 1: $\boxed{001100\ 010011\ 010110\ 011010\ 011101}$ $\boxed{100001\ 100011}$

Programming Projects 9.5

P1. *Design, program, and test a version of radix sort that is implementation independent, with alphabetic keys.*

Answer Class for keys:

```
class Key: public String{
public:
    char key_letter(int position) const;
    void make_blank();
    Key (const char * copy);
    Key ();
};
```

Implementation:

```
void Key::make_blank()
{
   if (entries != NULL) delete []entries;
   length = 0;
   entries = new char[length + 1];
   strcpy(entries, "");
}

Key::Key()
{
   make_blank();
}

char Key::key_letter(int position) const
{
   if (position >= 0 && position < length) return entries[position];
   else return ' ';
}

Key::Key(const char *copy): String(copy)
{
}
```

TOC
Index
Help

Class for records:

```
class Record {
public:
   char key_letter(int position) const;
   Record();                       //  default constructor
   operator Key() const;           //  cast to Key
   Record (const Key &a_name);     //  conversion to Record
private:
   Key name;
};
```

Implementation:

```
Record::Record() : name()
{
}

char Record::key_letter(int position) const
{
   return name.key_letter(position);
}

Record::operator Key() const
{
   return name;
}

Record::Record (const Key &a_name)
{
   name = a_name;
}

int alphabetic_order(char c)
/*
Post: The function returns the alphabetic position of character
      c, or it returns 0 if the character is blank.
*/

{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c - 'a' + 1;
   if ('A' <= c && c <= 'Z') return c - 'A' + 1;
   return 27;
}
```

Class for radix sorting:

```
template <class Record>
class Sortable_list: public List<Record> {
public:   //   sorting methods
   void radix_sort();
   //   Specify any other sorting methods here.
private:  //   auxiliary functions
   void rethread(Queue queues[]);
};
```

Implementation:

```
const int max_chars = 28;
```

```
template <class Record>
void Sortable_list<Record>::rethread(Queue queues[])
/*
Post: All the queues are combined back to the Sortable_list, leaving
      all the queues empty.
Uses: Methods of classes List, and Queue.
*/

{
   Record data;
   for (int i = 0; i < max_chars; i++)
      while (!queues[i].empty()) {
         queues[i].retrieve(data);
         insert(size(), data);
         queues[i].serve();
      }
}

template <class Record>
void Sortable_list<Record>::radix_sort()
/*
Post: The entries of the Sortable_list have been sorted
      so all their keys are in alphabetical order.
Uses: Methods from classes List, Queue, and Record;
      functions position and rethread.
*/

{
   Record data;
   Queue queues[max_chars];
   for (int position = key_size - 1; position >= 0; position--) {
      //   Loop from the least to the most significant position.
      while (remove(0, data) == success) {
         int queue_number = alphabetic_order(data.key_letter(position));
         queues[queue_number].append(data);    //   Queue operation.
      }
      rethread(queues);                        //   Reassemble the list.
   }
}
```

Driver program:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include <string.h>
#include   "../../6/strings/string.h"
#include   "../../6/strings/string.cpp"
#include "key.h"
#include "key.cpp"
#include "record.h"
#include "record.cpp"
typedef Record Queue_entry;

#include "../../3/queue/queue.h"
#include "../../3/queue/queue.cpp"
```

```
#include "radix.h"
const int key_size = 8;
#include "radix.cpp"

void write_entry(Record &c)
{
   for (int i = 0; i < 8; i++)
      cout << c.key_letter(i);
   cout << "\n";
}

main()
{
   List<Record> s; List<Record> t = s;  //  Help the compiler.
   int list_size = 250;
   Random dice;
   char word[9];
   word[8] = '\0';

   int i;
   Sortable_list<Record> the_list;
   for (i = 0; i < list_size; i++) {
      for (int j = 0; j < 8; j++)
         word[j] = dice.random_integer(0,26) + 'a';
         Key target_name = word;
         Record target = target_name;
      if (the_list.insert(i, target) != success) {
         cout << " Overflow in filling list." << endl;
      }
   }

   cout << "Unsorted list \n";
   the_list.traverse(write_entry);
   cout << "\n";
   the_list.radix_sort();
   cout << "\n Sorted list \n";
   the_list.traverse(write_entry);
   cout << "\n";
}
```

P2. *The radix-sort program presented in the book is very inefficient, since its implementation-independent features force a large amount of data movement. Design a project that is implementation dependent and saves all the data movement. In* rethread *you need only link the rear of one queue to the front of the next. This linking requires access to protected* Queue *data members; in other words we need a modified* Queue *class. A simple way to achieve this is to add a method*

Sortable_list :: concatenate(**const** Sortable_list &add_on);

to our derived linked list implementation and use lists instead of queues in the code for radix sort. Compare the performance of this version with that of other sorting methods for linked lists.

Answer A more efficient reimplementation of the radix sorting class is:

```
template <class Record>
class Sortable_list: public List<Record> {
public:   //   sorting methods
   void radix_sort();
   void concatenate(Sortable_list<Record> &add_on);
   Error_code append(const Record &data);
   //   Specify any other sorting methods here.
private:  //   auxiliary functions
   void rethread(Sortable_list<Record> queues[]);
};
```

Implementation:

```
const int max_chars = 28;

template <class Record>
Error_code Sortable_list<Record>::append(const Record &data)
/*
Post: data is appended to the end of the List
*/

{
   return insert(size(), data);
}

template <class Record>
void Sortable_list<Record>::concatenate(Sortable_list<Record> &add_on)
/*
Post: The list add_on is concatenated onto the Sortable_list.
      The list add_on is destroyed.
Uses: Methods of class List
*/

{
   if (head == NULL) head = add_on.head;
   else set_position(count - 1)->next = add_on.head;
   count += add_on.count;
   add_on.head = NULL;
   add_on.count = 0;
}

template <class Record>
void Sortable_list<Record>::rethread(Sortable_list<Record> queues[])
/*
Post: All the queues are combined back to the Sortable_list, leaving
      all the queues empty.
Uses: Methods of class List
*/

{
   Record data;
   for (int i = 0; i < max_chars; i++)
      concatenate(queues[i]);
}

template <class Record>
void Sortable_list<Record>::radix_sort()
/*

Post: The entries of the Sortable_list have been sorted
      so all their keys are in alphabetical order.
Uses: Methods from classes List and Record;
      functions position and rethread.
*/
```

```
{
   Record data;
   Sortable_list<Record> queues[max_chars];
   for (int position = key_size - 1; position >= 0; position--) {
      //   Loop from the least to the most significant position.
      while (remove(0, data) == success) {
         int queue_number = alphabetic_order(data.key_letter(position));
         queues[queue_number].append(data);
      }
      rethread(queues);                          //   Reassemble the list.
   }
}
```

Driver program for this improved radix sort is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include <string.h>
#include   "../../6/strings/string.h"
#include   "../../6/strings/string.cpp"
#include "../radix/key.h"
#include "../radix/key.cpp"
#include "../radix/record.h"
#include "../radix/record.cpp"

#include "radix.h"
const int key_size = 8;
#include "radix.cpp"

void write_entry(Record &c)
{
   for (int i = 0; i < 8; i++)
      cout << c.key_letter(i);
   cout << "\n";
}

main()
{
   List<Record> s; List<Record> t = s;  //  Help the compiler.
   int list_size = 10;
   Random dice;
   char word[9];
   word[8] = '\0';

   int i;
   Sortable_list<Record> the_list;
   for (i = 0; i < list_size; i++) {
      for (int j = 0; j < 8; j++)
         word[j] = dice.random_integer(0,26) + 'a';
         Key target_name = word;
         Record target = target_name;
      if (the_list.insert(i, target) != success) {
         cout << " Overflow in filling list." << endl;
      }
   }
```

```
    cout << "Program to demonstrate radix sort" << endl;
    cout << "Unsorted list \n";
    the_list.traverse(write_entry);
    cout << "\n";
    the_list.radix_sort();
    cout << "\n Sorted list \n";
    the_list.traverse(write_entry);
    cout << "\n";
}
```

Test results using this version of radix sort will be implementation dependent.

9.6 HASHING

Exercises 9.6

E1. *Prove by mathematical induction that* $1 + 3 + 5 + \cdots + (2i - 1) = i^2$ *for all integers* $i > 0$.

Answer For $i = 1$ both sides are equal to 1, so the base case holds. Assume the result is true for case $i - 1$:

$$1 + 3 + 5 + \cdots + (2i - 3) = (i - 1)^2.$$

Then we have

$$\begin{aligned} 1 + 3 + 5 + \cdots + (2i - 3) + (2i - 1) &= (i - 1)^2 + (2i - 1) \\ &= i^2 - 2i + 1 + (2i - 1) \\ &= i^2, \end{aligned}$$

which was to be proved.

E2. *Write a C++ function to insert an entry into a hash table with open addressing using linear probing.*

Answer
```
Error_code Hash_table :: insert(const Record &new_entry)
/* Post: If the Hash_table is full, a code of overflow is returned. If the table already contains an
         item with the key of new_entry a code of duplicate_error is returned. Otherwise: The
         Record new_entry is inserted into the Hash_table and success is returned.
   Uses: Methods for classes Key, and Record. The function hash. */
{
   Error_code result = success;
   int probe_count,                             //  Counter to be sure that table is not full.
      probe;                                    //  Position currently probed in the hash table.
   Key null;                                    //  Null key for comparison purposes.
   null.make_blank();
   probe = hash(new_entry);
   probe_count = 0;
   while (table[probe] != null                  //  Is the location empty?
          && table[probe] != new_entry          //  Duplicate key?
          && probe_count < hash_size) {         //  Has overflow occurred?
      probe_count++;
      probe = (probe + 1) % hash_size;
   }
   if (table[probe] == null) table[probe] = new_entry;   //  Insert new entry.
   else if (table[probe] == new_entry) result = duplicate_error;
   else result = overflow;                      //  The table is full.
   return result;
}
```

E3. *Write a C++ function to retrieve an entry from a hash table with open addressing using* **(a)** *linear probing;* **(b)** *quadratic probing.*

Answer

(a)
```
Error_code Hash_table :: retrieve(const Key &target, Record &found) const
{
   int probe_count,                          // counter to be sure that table is not full
      probe;                                 // position currently probed in the hash table
   Key null;                                 // null key for comparison purposes
   null.make_blank( );
   probe = hash(target);
   probe_count = 0;
   while (table[probe] != null &&            // Is the location empty?
     table[probe] != target                  // Search Successful?
     && probe_count < hash_size) {           // Full table?
      probe_count++;
      probe = (probe + 1) % hash_size;
   }
   if (table[probe] == target) {
      found = table[probe];
      return success;
   }
   else return not_present;
}
```

(b)
```
Error_code Hash_table :: retrieve(const Key &target, Record &found) const
{
   int probe_count,                          // counter to be sure that table is not full
      increment,                             // increment used for quadratic probing
      probe;                                 // position currently probed in the hash table
   Key null;                                 // null key for comparison purposes
   null.make_blank( );
probe = hash(target);
probe_count = 0;
increment = 1;
while (table[probe] != null &&               // Is the location empty?
  table[probe] != target                     // Search Successful?
  && probe_count < (hash_size + 1)/2) {      // Full table?
   probe_count++;
   probe = (probe + increment) % hash_size;
   increment += 2;                           // Prepare increment for next iteration
}
if (table[probe] == target) {
   found = table[probe];
   return success;
}
else return not_present;
}
```

E4. *In a student project for which the keys were integers, one student thought that he could mix the keys well by using a trigonometric function, which had to be converted to an integer index, so he defined his hash function as*

(**int**) sin(n).

What was wrong with this choice? He then decided to replace the function sin(n) *by* exp(n). *Criticize this choice.*

Answer Since $|\sin n| < 1$ for all integers n (in fact, for all real numbers n that are not odd multiples of $\pi/2$), the student's first hash function always yields the value 0 and hence is worthless for its intended purpose. The second function will also yield 0 for all numbers $n < 0$, and hence it is useless if the function is needed for negative integers. If n is restricted to positive integers, then the function may achieve an adequate spread of results (after reducing % hash_size), but it will be a slow calculation. The evaluation of e^n, first of all, requires calculating an approximation to an infinite series using floating point (**double**) arithmetic, and this will take significant time. The result must then be converted to an integer, and this will take even more time.

E5. *Devise a simple, easy to calculate hash function for mapping three-letter words to integers between 0 and $n - 1$, inclusive. Find the values of your function on the words*

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

for $n = 11, 13, 17, 19$. Try for as few collisions as possible.

Answer

```
const int hash_size = 101;
int hash(const String &key)
{
   const char *content = key.c_str();
   int h = 2 * content[0] + content[1] + 3 * content[2];
   return h % hash_size;
}
```

This function is easy to calculate and results in only one collision with $n = 11$, 17, and 19, and two collisions with $n = 13$.

E6. *Suppose that a hash table contains* hash_size = *13 entries indexed from 0 through 12 and that the following keys are to be mapped into the table:*

10 100 32 45 58 126 3 29 200 400 0

(a) *Determine the hash addresses and find how many collisions occur when these keys are reduced by applying the operation* % hash_size.

Answer

10	100	32	45	58	126	3	29	200	400	0
10	9	6	6	6	9	3	3	5	10	0

A total of five collisions occur.

(b) *Determine the hash addresses and find how many collisions occur when these keys are first folded by adding their digits together (in ordinary decimal representation) and then applying* % hash_size.

Answer

10	100	32	45	58	126	3	29	200	400	0
1	1	5	9	0	9	3	11	2	4	0

A total of three collisions occur.

(c) *Find a hash function that will produce no collisions for these keys. (A hash function that has no collisions for a fixed set of keys is called* ***perfect****.)*

perfect hash functions

Answer

```
const int hash_size = 13;
typedef int Key;
int hash(Key key)
{
    int h = key/11 + key/13 + key/97 + 2003 − key + key/100 % 2 + key/10;
    return h % hash_size;
}
```

(d) *Repeat the previous parts of this exercise for* hash_size = *11. (A hash function that produces no collision for a fixed set of keys that completely fill the hash table is called* ***minimal perfect****.)*

Answer The following is the hashed addresses for these keys when reduced % hash_size.

10	100	32	45	58	126	3	29	200	400	0
10	1	10	1	3	5	3	7	2	4	0

A total of three collisions occur. The following table gives the hash addresses when the keys are first folded then reduced by % hash_size.

10	100	32	45	58	126	3	29	200	400	0
1	1	5	9	2	9	3	0	2	4	0

Four collisions occur. The following function produces unique locations for each of the eleven keys.

```
const int hash_size = 11;
typedef int Key;
int hash(Key key)
{
   int h = 7 * (key/200) − (key/400) + 4 * (key/100) + key % 7 + key;
   return h % hash_size;
}
```

E7. *Another method for resolving collisions with open addressing is to keep a separate array called the* ***overflow table****, into which are put all entries that collide with an occupied location. They can either be inserted with another hash function or simply inserted in order, with sequential search used for retrieval. Discuss the advantages and disadvantages of this method.*

Answer A separate overflow table eliminates the problem of clustering. If a fast retrieval method (such as a second hash function) is used to access the overflow table, then this method may be significantly faster than other methods. If sequential search is used, however, the method will deteriorate quickly as the number of collisions increases. If insertions are very infrequent compared to retrievals, then the overflow table could be kept in sorted order and binary search used for retrieval; doing so would be quite efficient. The major disadvantages of using an overflow table are that the programming is more complicated and the available space is divided into smaller parts. Since the hash table itself must be smaller to provide room for the overflow table, collisions will become more likely, and so more entries will be forced into the overflow table, for which retrieval is slower. If the hash function and method for collision resolution can be chosen so that chains are rarely more than two or three entries long, then it is usually more efficient to keep all the space in a single hash table.

E8. *Write the following functions for processing a chained hash table, using the function* sequential_search( ) *of Section 7.2 and the list-processing operations of Section 6.1 to implement the operations.*

(a) Hash_table :: Hash_table( )

Answer We shall suppose that our methods are implemented for a **class** Hash_table with the following declaration:

```
const int hash_size = 997;                 //  a prime number of appropriate size
class Hash_table {
public:
  Hash_table( );
  void clear( );
  Error_code insert(const Record &new_entry);
  Error_code retrieve(const Key &target, Record &found) const;
  Error_code remove(const Key &target, Record &found);
private:
  List<Record> table[hash_size];
};
```

The Hash_table constructor will automatically call List constructors for the individual table elements. It has no other task to perform. Therefore it has an empty implementation.

```
Hash_table :: Hash_table( )
{
   //   The List constructor automatically initializes entry table with
   //   empty Lists. No other initialization is required.
}
```

(b) Hash_table :: clear( )

Answer

```
void Hash_table :: clear( )
{
   for (int i = 0; i < hash_size; i++)
      table[i].clear( );
}
```

(c) Hash_table :: insert(**const** Record &new_entry)

Answer

```
Error_code Hash_table :: insert(const Record &new_entry)
/* Post: If the Hash_table is full, a code of overflow is returned. If the table already contains an
         item with the key of new_entry a code of duplicate_error is returned. Otherwise: The
         Record new_entry is inserted into the Hash_table and success is returned. */
{
   Record old_copy;
   if (retrieve(new_entry, old_copy) == success)
      return duplicate_error;
   int probe = hash(new_entry);
   return table[probe].insert(0, new_entry);
}
```

(d) Hash_table :: retrieve(**const** Key &target, Record &found) **const**;

Answer

```
Error_code Hash_table :: retrieve(const Key &target, Record &found) const
{
   int position;
   Error_code outcome;
   int probe = hash(target);
   outcome = sequential_search(table[probe], target, position);
   if (outcome == success)
      table[probe].retrieve(position, found);
   return outcome;
}
```

(e) Hash_table :: remove(**const** Key &target, Record &x)

Answer

```
Error_code Hash_table :: remove(const Key &target, Record &found)
/* Post: If the Hash_table is empty, a code of underflow is returned. If the table contains no
         item with a key of target, a code of not_found is returned. Otherwise: The Record with
         this key is deleted from the Hash_table and success is returned. */
{
   int position;
   int probe = hash(target);
   if (sequential_search(table[probe], target, position) != success)
      return not_present;
   return table[probe].remove(position, found);
}
```

E9. *Write a deletion algorithm for a hash table with open addressing using linear probing, using a second special key to indicate a deleted entry (see Part 8 of Section 9.6.3 on page 405). Change the retrieval and insertion algorithms accordingly.*

Answer The algorithm has the following form.

```
remove(target)
  Search for the target using linear probing.
  if found
    Replace the corresponding entry in the hash table with the special key.
    return success;
  else return not_present
```

This algorithm translates into the following method that makes use of a constant global Key object called fill_word that is used as the special key to mark deleted entries.

```
Error_code Hash_table :: remove(const Key &target, Record &found)
/* Post: Any entry with Key target is recorded as Record found and is removed from the table.
   Uses: Methods for classes Key, and Record. The function hash. */
{
   Key null;
   null.make_blank( );
   int probe_count = 0,
       probe = hash(target);
   while (table[probe] != null
           && table[probe] != target
           && probe_count < hash_size) {
     probe_count++;
     probe = (probe + 1) % hash_size;
    }
    if (table[probe] == target) {
       found = table[probe];
       table[probe] = fill_word;             /* insert the special key                    */
       return success;
    }
    return not_present;                      /* key was not found                         */
}
```

The Key fill_word signals the methods to continue searching for a target. Therefore, the method retrieve remains unchanged. The method insert can be modified to refill deleted entries. The **while** loop of the method insert will change as follows:

```
  while ((table[probe] != null)
    && (table[probe] != new_entry)
    && (probe_count < hash_size)
    && (table[probe] != fill_word))          //   Fillable location?
```

The final **if** statement of the method would have the following form:

```
  if (table[probe] == null || table[probe] == fill_word)
    table[probe] = new_entry;
  else . . .
```

E10. *With linear probing, it is possible to delete an entry without using a second special key, as follows. Mark the deleted entry empty. Search until another empty position is found. If the search finds a key whose hash address is at or before the just-emptied position, then move it back there, make its previous position empty, and continue from the new empty position. Write an algorithm to implement this method. Do the retrieval and insertion algorithms need modification?*

Answer

```
remove (target)
  Search for the target key in the hash table.
  if it was found
    Set that position in the hash table to null.
    Set last_empty to that position.
    Increment position, reduce mod hash_size if necessary.
    while table[position] ! = null
       if the entry table locations:
            hash(table[position]), last_empty, position
          are arranged in this relative order in the circular table then:
            table[last_empty] = table[position]
            table[position] = null
            last_empty = position
       Increment position, reduce mod hash_size if necessary.
```

This translates into the following method:

```
Error_code Hash_table :: remove(const Key &target, Record &found)
/* Post: Any entry with Key target is recorded as Record found and is removed from the table.
   Uses: Methods for classes Key, and Record. The function hash. */
{
   Key null;
   null.make_blank();
   int probe_count = 0;
   int probe = hash(target);
   while (table[probe] != null
            && table[probe] != target
            && probe_count < hash_size) {
      probe_count++;
      probe = (probe + 1) % hash_size;
   }
   if (table[probe] == target) {
      found = table[probe];
      table[probe] = null;
      int last_empty = probe;
      probe = (probe + 1) % hash_size;
      while (table[probe] != null) {
          int ideal = hash(table[probe]);
          if ((ideal <= last_empty && last_empty < probe) ||
               (last_empty < probe && probe < ideal) ||
               (probe < ideal && ideal <= last_empty)) {
                  table[last_empty] = table[probe];
                  table[probe] = null;
                  last_empty = probe;
          }
          probe = (probe + 1) % hash_size;
      }
      return success;
   }
   return not_present;                    /* key was not found              */
}
```

The retrieval and insertion algorithms do not need modification.

Programming Projects 9.6

P1. *Consider the set of all C++ reserved words.*[2] *Consider these words as strings of 16 characters, where words less than 16 characters long are filled with blanks on the right.*

(a) *Devise an integer-valued function that will produce different values when applied to all the reserved words. [You may find it helpful to write a short program that reads the words from a file, applies the function you devise, and determines what collisions occur.]*

Answer A list of the C++ keywords that was used for this project is given in the data file `keywords`. Different implementations of C++ may have slightly different lists of keywords, and the programs in this directory would need modifications for other versions of the `keywords` file.

The following hash function is an integer valued hash function that takes distinct values at all keywords. It is a slight modification of the standard hash function that we use in the text.

```
int hash(const Key &target)
/*
Post: target has been hashed, returning a value between 0 and hash_size -1.
Uses: Methods for the class Key.
*/

{
   int value = 0;
   for (int position = 0; position < 16; position++) {
      value = (6 * value + target.key_letter(position)) % hash_size;
   }
   value = value % hash_size;
   if (value < 0) value += hash_size;
   return value;
}
```

We made this modification by a trial and error process, using the following hash table implementation to report the number of collisions that result from various hash functions:

```
const int hash_size = 997;   //  a prime number of appropriate size
class Hash_table {
public:
   Hash_table();
   void clear();
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
   void collision_mod();
private:
   Record table[hash_size];
};
```

Implementation:

```
int collisions = 0;

int gcd(int m, int n) {
   if (m < n) return gcd(n, m);
   if (n == 0) return m;
   return gcd(n, m%n);
}
```

[2] Any textbook on C++ will contain a list of the reserved words. Different versions of C++, however, support different sets of reserved words.

```
void Hash_table::collision_mod()
{
   Key null;
   null.make_blank();
   int real_size = hash_size;
   for (int i = 0; i < hash_size; i++)
      if ((Key) table[i] != null)  {
         real_size = i + 1;
   }
   int min = real_size - 1;
   int min_at = 0;
   int min_mult = 1;

   for (int l = 1; l < real_size; l++)
   if (gcd(l, real_size) == 1)
   for (int j = 1; j < real_size; j++) {
     int temp_min = 0;
     int i;
     for (i = 0; i < real_size; i++) {
        if ((Key) table[i] != null) {
          int k = (i * l + j) % real_size;
          if (k >= min) break;

          if (k >= temp_min) temp_min = k;
        }
     }
     if (i == real_size) {
       min = temp_min;
       min_at = j;
       min_mult = l;
     }
   }

   cout << "multiply by " <<  min_mult << " and "
        << "add on " <<  min_at << " then reduce mod " << real_size
        << endl;
   cout << "for better hashing " << endl;
}

Hash_table::Hash_table()
{
}

void Hash_table::clear()
{
   Key null;
   null.make_blank();

   for (int i = 0; i < hash_size; i++)
      table[i].initialize(null);
}

Error_code Hash_table::insert(const Record &new_entry)
/*
Post: If the Hash_table is full, a code of overflow is returned.
      If the table already contains an item with the key of
      new_entry a code of duplicate_error is returned.
Otherwise: The Record new_entry is inserted into the Hash_table
      and success is returned.
Uses: Methods for classes Key, and Record.
      The function hash.
```

```
*/
{
   Error_code result = success;
   int probe_count,      //  Counter to be sure that table is not full.
       increment,        //  Increment used for quadratic probing.
       probe;            //  Position currently probed in the hash table.
   Key null;             //  Null key for comparison purposes.
   null.make_blank();

   probe = hash(new_entry);
   probe_count = 0;
   increment = 1;

   while ((Key) table[probe] != null          // Is the location empty?
     && (Key) table[probe] != (Key) new_entry // Duplicate key?
     && probe_count < (hash_size + 1) / 2) {  // Has overflow occurred?
      collisions++;
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;    //     Prepare increment for next iteration.
   }
   if ((Key) table[probe] == null)
      table[probe] = new_entry; //   Insert new entry.
   else if ((Key) table[probe] == (Key) new_entry)
      result = duplicate_error;
   else result =  overflow;     //   The table is full.
   return result;
}

Error_code Hash_table::retrieve(const Key &target, Record &found) const
{
   int probe_count,       //   counter to be sure that table is not full
      increment,          //   increment used for quadratic probing
      probe;              //   position currently probed in the hash table
   Key null;              //   null key for comparison purposes
   null.make_blank();

   probe = hash(target);
   probe_count = 0;
   increment = 1;
   while ((Key) table[probe] != null &&        //   Is the location empty?
     (Key) table[probe] != target              //   Search Successful?
     && probe_count < (hash_size + 1) / 2) {   //   Full table?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;       //     Prepare increment for next iteration
   }
   if ((Key) table[probe] == target) {
      found = table[probe];
      return success;
   }
   else return not_present;
}
```

Driver:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
```

```
#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include <string.h>
#include   "../../6/strings/string.h"
#include   "../../6/strings/string.cpp"

#include "../radix/key.h"
#include "../radix/key.cpp"
#include "../hash/record.h"
#include "../hash/record.cpp"

void write_entry(Record c)
{
   for (int i = 0; i < 16; i++)
      cout << c.key_letter(i);
   cout << "\n";
}

#include "hash.h"
#include "hashfn.cpp"
#include "hash.cpp"

main()
{
   char word[16];
   Hash_table keywords;

   ifstream file_in("keywords");
   if (file_in == 0)
      cout << "Can't open input file " << "keywords" << endl;
   else {
      while (!file_in.eof()) {
         file_in >> word;
         if (strlen(word) > 0) {
           Record target;
           target.initialize(word);
           keywords.insert(target);
         }
      }
      cout << " total number of collisions is " << collisions
           << endl;
      if (collisions > 0)
         keywords.collision_mod();
   }
}
```

(b) *Find the smallest integer* hash_size *such that, when the values of your function are reduced by applying* % hash_size, *all the values remain distinct.*

Answer An extra hash table method called `collision_mod` in the preceding implementation gives information about how to adjust a hash function, to map to a smaller range of values.

(c) *[Challenging] Modify your function as necessary until you can achieve* hash_size *in the preceding part to be the same as the number of reserved words. (You will then have discovered a minimal perfect hash function for the C++ reserved words, mapping these words onto a table with no empty positions.)*

Answer Repeated use of the improvements in (a) and (b) results in a new hash function:

```
int hash(const Key &target)
/*
Post: target has been hashed, returning a value between 0 and hash_size -1.
Uses: Methods for the class Key.
*/

{
   int value = 0;
   for (int position = 0; position < 16; position++) {
      value = (6 * value + target.key_letter(position)) % hash_size;
   }
   value = value % hash_size;
   value = value + value % (98 + 1) + 98 % (value + 1);
   value = value % 148;
   value = (31 * value + 125) % 145;
   value = (62 * value + 62) % 127;
   value = (50 * value + 96) % 111;
   value = (value + 66) % 100;
   value = (11 * value + 78) % 91;
   value = (35 * value + 66) % 82;
   value = (value + 44) % 74;
   value = (31 * value + 9) % 69;
   value = (11 * value + 47) % 64;
   value = (11 * value + 19) % 60;
   value = (7 * value + 16) % 57;
   value = (value + 28) % 54;
   value = (3 * value + 7) % 52;
   value = (value + 34) % 50;
   value = (value + 9) % 49;
   return value;
}
```

This hash function gives a minimal perfect hash function for the file keywords.

molecular weight

P2. *Write a program that will read a molecular formula such as H_2SO_4 and will write out the molecular weight of the compound that it represents. Your program should be able to handle bracketed radicals such as in $Al_2(SO_4)_3$. [Hint: Use recursion to find the molecular weight of a bracketed radical. Simplifications: You may find it helpful to enclose the whole formula in parentheses (. . .). You will need to set up a hash table of atomic weights of elements, indexed by their abbreviations. For simplicity, the table may be restricted to the more common elements. Some elements have one-letter abbreviations, and some two. For uniformity you may add blanks to the one-letter abbreviations.]*

Answer We can use the Key type from from radix sort, where it is defined as:

```
class Key: public String{
public:
    char key_letter(int position) const;
    void make_blank();
    Key (const char * copy);
    Key ();
};
```

Implementation:

```
void Key::make_blank()
{
   if (entries != NULL) delete []entries;
   length = 0;
   entries = new char[length + 1];
   strcpy(entries, "");
}

Key::Key()
{
   make_blank();
}

char Key::key_letter(int position) const
{
   if (position >= 0 && position < length) return entries[position];
   else return ' ';
}

Key::Key(const char *copy): String(copy)
{
}
```

We define a Record class by:

```
class Record {
public:
   double get_weight();
   void set_weight(double w);
   char key_letter(int position) const;
   Record();                       //  default constructor
   operator Key() const;           //  cast to Key
   void initialize(const Key &a_name);  //  conversion to Record
private:
   Key name;
   double weight;
};
```

Implementation:

```
Record::Record() : name()
{
}

char Record::key_letter(int position) const
{
   return name.key_letter(position);
}

Record::operator Key() const
{
   return name;
}

void Record::initialize(const Key &a_name)
{
   name = a_name;
}
```

```
int alphabetic_order(char c)
/*
Post: The function returns the alphabetic position of character
      c, or it returns 0 if the character is blank.
*/

{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c - 'a' + 1;
   if ('A' <= c && c <= 'Z') return c - 'A' + 1;
   return 27;
}

double Record::get_weight()
{
   return weight;
}

void Record::set_weight(double w)
{
   weight = w;
}
```

We use the Hash_table class from the text, where it is set up as:

```
const int hash_size = 997;   //  a prime number of appropriate size
class Hash_table {
public:
   Hash_table();
   void clear();
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

Implementation:

```
Hash_table::Hash_table()
{
}

void Hash_table::clear()
{
   Key null;
   null.make_blank();

   for (int i = 0; i < hash_size; i++)
      table[i].initialize(null);
}

Error_code Hash_table::insert(const Record &new_entry)
/*
Post: If the Hash_table is full, a code of overflow is returned.
      If the table already contains an item with the key of
      new_entry a code of duplicate_error is returned.
Otherwise: The Record new_entry is inserted into the Hash_table
      and success is returned.
Uses: Methods for classes Key, and Record.
      The function hash.
*/
```

```
{
   Error_code result = success;
   int probe_count,        //  Counter to be sure that table is not full.
       increment,          //  Increment used for quadratic probing.
       probe;              //  Position currently probed in the hash table.
   Key null;               //  Null key for comparison purposes.
   null.make_blank();

   probe = hash(new_entry);
   probe_count = 0;
   increment = 1;

   while ((Key) table[probe] != null            //   Is the location empty?
     && (Key) table[probe] != (Key) new_entry   //   Duplicate key?
     && probe_count < (hash_size + 1) / 2) {    //   Has overflow occurred?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;          //     Prepare increment for next iteration.
   }
   if ((Key) table[probe] == null)
      table[probe] = new_entry; //   Insert new entry.
   else if ((Key) table[probe] == (Key) new_entry)
            result = duplicate_error;
   else result =  overflow;                    //   The table is full.
   return result;
}

Error_code Hash_table::retrieve(const Key &target, Record &found) const
{
   int probe_count,       //   counter to be sure that table is not full
      increment,          //   increment used for quadratic probing
      probe;              //   position currently probed in the hash table
   Key null;              //   null key for comparison purposes
   null.make_blank();

   probe = hash(target);
   probe_count = 0;
   increment = 1;
   while ((Key) table[probe] != null &&       //   Is the location empty?
     (Key) table[probe] != target             //   Search Successful?
     && probe_count < (hash_size + 1) / 2) { //   Full table?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;        //     Prepare increment for next iteration
   }
   if ((Key) table[probe] == target) {
      found = table[probe];
      return success;
   }
   else return not_present;
}
```

Hash function:

```
int hash(const Key &target)
/*
Post: target has been hashed, returning a value between 0 and hash_size -1.
Uses: Methods for the class Key.
*/
{
   int value = 0;
   for (int position = 0; position < 8; position++)
      value = 4 * value + target.key_letter(position);
   value %= hash_size;
   if (value < 0) value += hash_size;
   return value;
}
```

The molecular weight project is then implemented as:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include <string.h>
#include   "../../6/strings/string.h"
#include   "../../6/strings/string.cpp"

#include "../radix/key.h"
#include "../radix/key.cpp"
#include "record.h"
#include "record.cpp"

void write_entry(Record c)
{
   for (int i = 0; i < 8; i++)
      cout << c.key_letter(i);
   cout << "\n";
}

#include "../hash/hash.h"
#include "../hash/hashfn.cpp"
#include "../hash/hash.cpp"

#include "auxil.cpp"

int main()
{
    Hash_table atoms;
    char formula[100];      // molecular formula to calculate
    int position = 0;       // current position in formula
    double total;

    if (initialize(atoms)) {
       read_in(formula);
       total = calculate(formula, position, atoms);
       cout << "The molecular weight of " << formula
            << " is " << total << endl;
    }
}
```

Auxiliary functions:

```
bool initialize(Hash_table &atoms)
/*
Initialize: read the atomic weight data
*/
{
   char name[3], file_name[100];
   double wt;
   cout << "What is the atom data file name: " << endl;
   cin  >> file_name;
   while (cin.get() != '\n');
   ifstream file_in(file_name);
   if (file_in == 0) {
      cout << "Can't open input file " << file_name << endl;
      return false;
   }
   while (!file_in.eof()) {
      file_in >> name >> wt;
      Record atom;
      atom.initialize(name);
      atom.set_weight(wt);
      atoms.insert(atom);
   }
   return true;
}

void read_in(char * formula)
/*
Post: A chemical formula from the user has been read
*/
{
   char ch;
   int i = 0;
   bool valid = false;
   cout << "Enter formula: " << flush;
   do {
      while ((ch = cin.get()) != '\n' && i < 99) {
         if (isalnum(ch) || ch == '(' || ch == ')')
            formula[i++] = ch;
      }
      if (ch == '\n')
         valid = true;
      else
         cout << "That formula is too long.  Try again. " << endl;
  } while (!valid);
  formula[i] = '\0';
}

double calculate(char * formula, int &position, const Hash_table &atoms)
/*
Post: the molecular weight of the *(formula + position) is returned
*/
{
   double sum = 0.0;
   double partial_sum;
   int factor;
   char name[3];
```

TOC
Index
Help

```
   while (true) {
      switch(formula[position]) {
      case '\0':
      case ')':
         return sum;

      case '(':
         position++;      /* skip left parenthesis    */
         partial_sum = calculate(formula, position, atoms);
         factor = 0;
         position++;      /* skip right parenthesis   */
         while (isdigit(formula[position])) {
            factor = factor * 10 + (formula[position] - '0');
            position++;
         }

         if (factor != 0)
            sum += partial_sum * factor;
         else
            sum += partial_sum;
         break;

      default:
         name[0] = formula[position++];
         if (islower(formula[position])) {
            name[1] = formula[position++];
            name[2] = '\0';
         }
         else
            name[1] = '\0';

         Key target(name);
         Record atom;
         if (atoms.retrieve(target, atom) == success) {
            partial_sum = atom.get_weight();
            factor = 0;
            while (isdigit(formula[position])) {
               factor = factor * 10 + (formula[position] - '0');
               position++;
            }
            if (factor) sum += partial_sum * factor;
            else sum += partial_sum;
         }
         else {
            cout << "Element " << name << " not found.  Ignored.\n"
                 << endl;
            while (isdigit(formula[position])) position++;
         }
      }
   }
}
```

TOC
Index
Help

9.7 ANALYSIS OF HASHING

Exercises 9.7

E1. *Suppose that each entry in a hash table occupies s words of storage (exclusive of the pointer member needed if chaining is used), where we take one word as the amount of space needed for a pointer. Also suppose that there are n occupied entries in the hash table, and the hash table has a total of t possible positions (t is the same as* hash_size*), including occupied and empty positions.*

(a) *If open addressing is used, determine how many words of storage will be required for the hash table.*

Answer The amount of memory used by open addressing, regardless of the load factor, is ts words, where t is the same as hash_size.

(b) *If chaining is used, then each node will require $s + 1$ words, including the pointer member. How many words will be used altogether for the n nodes?*

Answer If the hash table using chaining contains n nodes, then the nodes themselves occupy $n(s + 1)$ words of memory.

(c) *If chaining is used, how many words will be used for the hash table itself? (Recall that with chaining the hash table itself contains only pointers requiring one word each.)*

Answer A hash table using chaining requires t words of memory to store the pointers to each chain of nodes.

(d) *Add your answers to the two previous parts to find the total storage requirement for chaining.*

Answer The total required memory for hash tables with chaining is $t + n(s + 1)$.

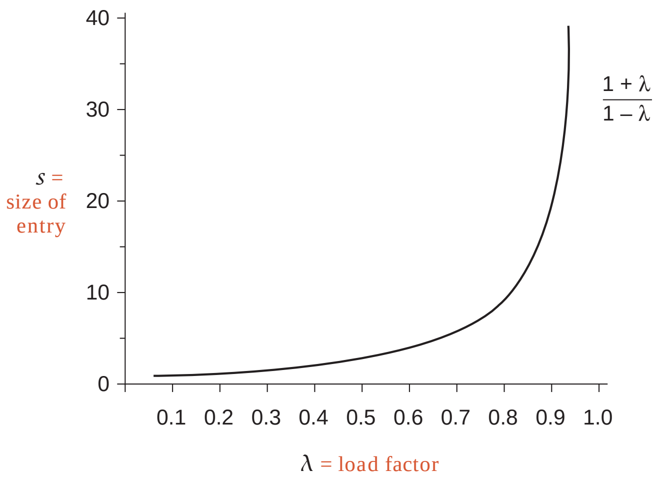
(e) *If s is small (that is, the entries have a small size), then open addressing requires less total memory for a given load factor $\lambda = n/t$, but for large s (large entries), chaining requires less space altogether. Find the break-even value for s, at which both methods use the same total storage. Your answer will be a formula for s that depends on the load factor λ, but it should not involve the numbers t or n directly.*

Answer To find the break-even size of an entry, for a given load factor λ, the memory used both by open addressing and by chaining must be equated as $ts = t + n(s + 1)$ so $s(t - n) = t + n$. Hence

$$s = \frac{t + n}{t - n} = \frac{1 + \lambda}{1 - \lambda}.$$

(f) *Evaluate and graph the results of your formula for values of λ ranging from 0.05 to 0.95.*

Answer As an example, a hash table with a load factor of 0.70 would be more efficiently implemented with chaining if each entry stored in the table occupies $\frac{1.70}{0.30} \approx 6$ words of memory or more.



E2. *One reason why the answer to the birthday problem is surprising is that it differs from the answers to apparently related questions. For the following, suppose that there are n people in the room, and disregard leap years.*

(a) *What is the probability that someone in the room will have a birthday on a random date drawn from a hat?*

Answer The probability of having a random date be the birthday of one of n people in a room is $1 - (\frac{364}{365})^n$.

(b) *What is the probability that at least two people in the room will have that same random birthday?*

Answer The following results involve the calculation of binomial probabilities.

$$p = \frac{1}{365} \qquad q = 1 - p = \frac{364}{365}$$

The probability of two of the n people having the same birthday is:

$$\begin{aligned} 1 - q^n - npq^{n-1} &= 1 - \left(\frac{364}{365}\right)^n - \left(\frac{n}{365}\right)\left(\frac{364}{365}\right)^{n-1} \\ &= 1 - \left(\frac{364}{365}\right)^n - \frac{n(364)^{n-1}}{365^n} \\ &= \frac{365^n - 364^n - n(364)^{n-1}}{365^n} \end{aligned}$$

(c) *If we choose one person and find that person's birthday, what is the probability that someone else in the room will share the birthday?*

Answer This problem is similar to part (a), only an essentially random date is compared with $n-1$ people. The probability of another person sharing the birthday of the chosen person is $1 - (\frac{364}{365})^{n-1}$.

E3. *In a chained hash table, suppose that it makes sense to speak of an order for the keys, and suppose that the nodes in each chain are kept in order by key. Then a search can be terminated as soon as it passes the place where the key should be, if present. How many fewer probes will be done, on average, in an unsuccessful search? In a successful search? How many probes are needed, on average, to insert a new node in the right place? Compare your answers with the corresponding numbers derived in the text for the case of unordered chains.*

ordered hash table

Answer Unsuccessful sequential searches on ordered lists can terminate, on average, after searching through half of the entries. Since the analogous action occurs with chained hash tables, the saving is also that of only searching through half of the entries in each chain. Therefore, the number of probes for an unsuccessful search would be, on average, $\frac{1}{2}\lambda$. The number of probes for a successful search would remain unchanged. An insertion would therefore require, on average, $\frac{1}{2}\lambda$ probes to find the proper place for each new entry.

TOC
Index
Help

E4. *In our discussion of chaining, the hash table itself contained only lists, one for each of the chains. One variant method is to place the first actual entry of each chain in the hash table itself. (An empty position is indicated by an impossible key, as with open addressing.) With a given load factor, calculate the effect on space of this method, as a function of the number of words (except links) in each entry. (A link takes one word.)*

Answer Let t be the total number of positions in the hash table and let n be the number of entries inserted, so the load factor is $\lambda = n/t$. Let s be the number of words in an entry, plus one more for the link.

We must first determine, on average, how many of the entries will be in the table itself and how many will be in dynamic storage. The number in dynamic storage is the same as the number of collisions that occurred as the n entries were inserted into the table. For $k = 1, \ldots, n$, let r_k be the number of entries in the table itself after entry k is inserted, so the number in dynamic storage is $k - r_k$. Then the probability of a collision as entry $k+1$ is inserted is r_k/t, so after the insertion of entry $k+1$ the expected number in the table itself is $r_{k+1} = r_k + 1 - (r_k/t)$, which we write as

$$r_{k+1} = \alpha r_k + 1$$

where $\alpha = 1 - (1/t)$. We know that $r_1 = 1$ since the first insertion into an empty table causes no collision. We solve the equation by substituting earlier cases of itself, finally obtaining

$$r_k = \alpha r_{k-1} + 1 = \alpha^2 r_{k-2} + \alpha + 1 = \cdots = \alpha^{k-1} + \alpha^{k-2} + \cdots + \alpha + 1 = \frac{1-\alpha^k}{1-\alpha}.$$

After all n entries are inserted, then, the average number in the table itself is

$$r_n = \frac{1-\alpha^n}{1-\alpha},$$

and the average number in dynamic storage is $n - r_n$.

We can now answer the exercise. When only pointers are in the hash table itself, then the total space used is $t + n(s+1)$ words. When the first entries are in the table itself, the total space used increases to $t(s+1)+(n-r_n)(s+1)$ words.

Programming Project 9.7

P1. *Produce a table like Figure 9.0 for your computer, by writing and running test programs to implement the various kinds of hash tables and load factors.*

Answer The tests used to make the following table were done using random real numbers, from 0 to 1, on a hash table of size 997.

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search						
Chaining	1.02	1.28	1.39	1.46	1.49	2.00
Open, quadratic probes	1.02	1.49	2.15	2.58	4.70	—
Open, linear probes	1.02	1.55	2.78	6.68	29.33	—
Unsuccessful search						
Chaining	0.15	0.58	0.82	0.88	0.99	2.00
Open, quadratic probes	1.15	2.28	5.19	10.87	92.08	—
Open, linear probes	1.15	2.65	10.47	48.47	424.27	—

TOC
Index
Help

9.9 APPLICATION: THE LIFE GAME REVISITED

Programming Projects 9.9

P1. *Write the* Life *methods* **(a)** neighbor_count, **(b)** retrieve, *and* **(c)** initialize.

Answer

```
class Life {
public:
   Life();
   void initialize();
   void print();
   void update();
   ~Life();

private:
   List<Cell *> *living;
   Hash_table *is_living;
   bool retrieve(int row, int col) const;
   Error_code insert(int row, int col);
   int neighbor_count(int row, int col) const;
};
```

Implementation:

```
Life::Life()
/*
Post: The members of a Life object are
      dynamically allocated and initialized.
Uses: The class Hash_table and the class List.
*/

{
   living = new List<Cell *>;
   is_living = new Hash_table;
}

Life::Łife()
/*
Post: The dynamically allocated members of a Life object
      and all ell objects that they reference are deleted.
Uses: The class Hash_table and the class List.
*/

{
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);
      delete old_cell;
   }
   delete is_living;          //  Calls the Hash_table destructor
   delete living;             //  Calls the List destructor
}

int Life::neighbor_count(int x, int y) const
{
   int count = 0;
   for (int x_add = -1; x_add < 2; x_add++)
     for (int y_add = -1; y_add < 2; y_add++)
        if (is_living->retrieve(x + x_add, y + y_add)) count++;
   if (is_living->retrieve(x, y)) count--;
   return count;
}

bool Life::retrieve(int x, int y) const
{
   return is_living->retrieve(x, y);
}

Error_code Life::insert(int row, int col)
/*
Pre:  The cell with coordinates row and col does not
      belong to the Life configuration.
Post: The cell has been added to the configuration.  If insertion into
      either the List or the Hash_table fails, an error code is returned.
Uses: The class List, the class Hash_table, and the struct Cell
*/

{
   Error_code outcome;
   Cell *new_cell = new Cell(row, col);
   int index = living->size();
   outcome = living->insert(index, new_cell);
   if (outcome == success)
      outcome = is_living->insert(new_cell);
   if (outcome != success)
      cout << " Warning: new Cell insertion failed" << endl;
   return outcome;
}

void Life::print()
/*
Post: A central window onto the Life object is displayed.
Uses: The auxiliary function Life::retrieve.
*/

{
   int row, col;
   cout << endl << "The current Life configuration is:" << endl;
   for (row = 0; row < 20; row++) {
      for (col = 0; col < 80; col++)
         if (retrieve(row, col)) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << "There are " << living->size() << " living cells."
        <<  endl;
}

void Life::update()
/*
Post: The Life object contains the next generation of configuration.
Uses: The class Hash_table and the class Life and its
      auxiliary functions.
*/

{
   Life new_configuration;
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);      //  Obtain a living cell.
      for (int row_add = -1; row_add < 2; row_add ++)
         for (int col_add = -1; col_add < 2; col_add++) {
            int new_row = old_cell->row + row_add,
                new_col = old_cell->col + col_add;
//  new_row, new_col is now a living cell or a neighbor of a living cell,

            if (!new_configuration.retrieve(new_row, new_col))
               switch (neighbor_count(new_row, new_col)) {
               case 3:  //  With neighbor count 3, the cell becomes alive.
                  new_configuration.insert(new_row, new_col);
                  break;

               case 2:  //  With count 2, cell keeps the same status.
                  if (retrieve(new_row, new_col))
                     new_configuration.insert(new_row, new_col);
                  break;
               default: //  Otherwise, the cell is dead.
                  break;
               }
         }
   }
// Exchange data of current configuration with data of new_configuration.
   List<Cell *> *temp_list = living;
   living = new_configuration.living;
   new_configuration.living = temp_list;
   Hash_table *temp_hash = is_living;
   is_living = new_configuration.is_living;
   new_configuration.is_living = temp_hash;
}

void Life::initialize()
/*
Post:
Uses:
*/

{
   Cell new_cell;
   int row, col;
   int read_rows, read_cols;
   char c;

   cout << "Please enter the number of rows and columns to be read in.\n";
   cout << "Note: The map for any cells that are not within the range:\n";
   cout << "0 <= row < 20 and 0 <= column < 80 will not be printed "
        << "to the terminal\n";
   cout << "Such cells will still be included within the calculations.\n";

   cout << "Please enter number of rows: " << flush;
   cin >> read_rows;
   cout << "Please enter number of columns: " << flush;
   cin >> read_cols;
   do {
      cin.get(c);
   } while (c != '\n');

   cout << "Please enter blanks to signify empty cells.\n";
   for (row = 0; row < read_rows; row++) {
      cout << row << " : ";
      for (col = 0; col < read_cols; col++) {
         cin.get(c);
         if (c == '\n') break;

         if (c != ' ') {
            insert(row, col);
         }
      }
      while (c != '\n') cin.get(c);
   }
}
```

P2. *Write the* Hash_table *methods* **(a)** insert *and* **(b)** retrieve *for the chained implementation that stores pointers to cells of a Life configuration.*

Answer

```
const int hash_size = 997;    //  a prime number of appropriate size
```

```
class Hash_table {
public:
   Error_code insert(Cell *new_entry);
   bool retrieve(int row, int col) const;
private:
   List<Cell *> table[hash_size];
};
```

Implementation:

```
Error_code Hash_table::insert(Cell *new_entry)
{
   int probe, location;
   Cell *found;
   probe = hash(new_entry->row, new_entry->col);
   if (sequential_search(table[probe], new_entry->row, new_entry->col,
                                       location, found) == not_present)
      return (table[probe].insert(0, new_entry));
   else
      return (duplicate_error);
}

bool Hash_table::retrieve(int row, int col) const
{
   int probe, location;
   Cell *found;
   probe = hash(row, col);
   if (sequential_search(table[probe], row, col, location, found) ==
                        success) return true;
   else return false;
}
```

The remaining code (mainly from the text) for the Life project is:

```
struct Cell {
   Cell() { row = col = 0; }   //  constructors
   Cell(int x, int y) { row = x;  col = y; }
   int row, col;               //   grid coordinates
};

#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

#include "cell.h"
#include "hash.h"
#include "hashfn.cpp"
#include "hash.cpp"
#include "life.h"
#include "life.cpp"

void instructions()
{
   cout << "           Life version 2   \n\n";
   cout << "Welcome to Conway's game of Life on an unbounded grid.\n\n";
   cout << "This implementation uses a hash table to locate cells.\n";
   cout << "The Life cells are supposed to be on an unbounded grid.\n";
   cout << "A data structure such as a hash table allows a virtually\n";
   cout << "unlimited size cell storage area thus allowing an unbounded\n";
   cout << "grid to be produced.\n\n";
}
int main()  //  Program to play Conway's game of Life.
/*
Pre:  The user supplies an initial configuration of living cells.
Post: The program prints a sequence of pictures showing the changes in
      the configuration of living cells according to the rules for
      the game of Life.
Uses: The class Life and its methods initialize(),
      print() and update().
      The functions  instructions(),  user_says_yes().
*/
{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}
```

P3. *Modify* update *so that it uses a second local* Life *object to store cells that have been considered for insertion, but rejected. Use this object to make sure that no cell is considered twice.*

Answer A modified update implementation is:

```
Life::Life()
/*
Post: The members of a Life object are
      dynamically allocated and initialized.
Uses: The class Hash_table and the class List.
*/
{
   living = new List<Cell *>;
   is_living = new Hash_table;
}
Life::~Life()
/*
Post: The dynamically allocated members of a Life object
      and all ell objects that they reference are deleted.
Uses: The class Hash_table and the class List.
*/
{
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);
      delete old_cell;
   }
   delete is_living;         //  Calls the Hash_table destructor
   delete living;            //  Calls the List destructor
}
```

```
int Life::neighbor_count(int x, int y) const
{
   int count = 0;
   for (int x_add = -1; x_add < 2; x_add++)
     for (int y_add = -1; y_add < 2; y_add++)
        if (is_living->retrieve(x + x_add, y + y_add)) count++;
   if (is_living->retrieve(x, y)) count--;
   return count;
}

bool Life::retrieve(int x, int y) const
{
   return is_living->retrieve(x, y);
}

Error_code Life::insert(int row, int col)
/*
Pre:  The cell with coordinates row and col does not
      belong to the Life configuration.
Post: The cell has been added to the configuration.  If insertion into
      either the List or the Hash_table fails, an error
      code is returned.
Uses: The class List, the class Hash_table, and the struct Cell
*/

{
   Error_code outcome;
   Cell *new_cell = new Cell(row, col);

   int index = living->size();
   outcome = living->insert(index, new_cell);
   if (outcome == success)
      outcome = is_living->insert(new_cell);
   if (outcome != success)
      cout << " Warning: new Cell insertion failed" << endl;
   return outcome;
}

void Life::print()
/*
Post: A central window onto the Life object is displayed.
Uses: The auxiliary function Life::retrieve.
*/

{
   int row, col;
   cout << endl << "The current Life configuration is:" << endl;
   for (row = 0; row < 20; row++) {
      for (col = 0; col < 80; col++)
         if (retrieve(row, col)) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

```
void Life::update()
/*
Post: The Life object contains the next generation of configuration.
Uses: The class Hash_table and the class Life and its
      auxiliary functions.
*/

{
   Life new_configuration;
   Life cells_considered;
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);      //  Obtain a living cell.
      for (int row_add = -1; row_add < 2; row_add ++)
         for (int col_add = -1; col_add < 2; col_add++) {
            int new_row = old_cell->row + row_add,
                new_col = old_cell->col + col_add;
//  new_row, new_col is now a living cell or a neighbor of a living cell,

            if (!cells_considered.retrieve(new_row, new_col)) {
               cells_considered.insert(new_row, new_col);
               switch (neighbor_count(new_row, new_col)) {
               case 3:  //  With neighbor count 3, the cell becomes alive.
                  new_configuration.insert(new_row, new_col);
                  break;

               case 2:  //  With count 2, cell keeps the same status.
                  if (retrieve(new_row, new_col))
                     new_configuration.insert(new_row, new_col);
                  break;

               default: //  Otherwise, the cell is dead.
                  break;
               }
            }
         }
   }

// Exchange data of current configuration with data of new_configuration.
   List<Cell *> *temp_list = living;
   living = new_configuration.living;
   new_configuration.living = temp_list;
   Hash_table *temp_hash = is_living;
   is_living = new_configuration.is_living;
   new_configuration.is_living = temp_hash;
}

void Life::initialize()
/*
Post:
Uses:
*/

{
   Cell new_cell;
   int row, col;
   int read_rows, read_cols;
   char c;
   cout << "Please enter the number of rows and columns to be read in.\n";
   cout << "Note: The map for any cells that are not within the range:\n";
   cout << "0 <= row < 20 and 0 <= column < 80 will not be printed "
        << "to the terminal\n";
   cout << "Such cells will still be included within the calculations.\n";

   cout << "Please enter number of rows: " << flush;
   cin >> read_rows;
   cout << "Please enter number of columns: " << flush;
   cin >> read_cols;
   do {
      cin.get(c);
   } while (c != '\n');

   cout << "Please enter blanks to signify empty cells.\n";
   for (row = 0; row < read_rows; row++) {
      cout << row << " : ";
      for (col = 0; col < read_cols; col++) {
         cin.get(c);
         if (c == '\n') break;

         if (c != ' ') {
            insert(row, col);
         }
      }
      while (c != '\n') cin.get(c);
   }
}
```

A driver for Life that uses this update method is:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

#include "../life4/cell.h"
#include "../life4/hash.h"
#include "../life4/hashfn.cpp"
#include "../life4/hash.cpp"
#include "../life4/life.h"
#include "life.cpp"

void instructions()
{
   cout << "            Life version 2   \n\n";
   cout << "Welcome to Conway's game of Life on an unbounded grid.\n\n";
   cout << "This implementation uses a hash table to locate cells.\n";
   cout << "The Life cells are supposed to be on an unbounded grid.\n";
   cout << "A data structure such as a hash table allows a virtually\n";
   cout << "unlimited size cell storage area thus allowing an unbounded\n";
   cout << "grid to be produced.\n\n";
}
```

```
int main()  //  Program to play Conway's game of Life.
/*
Pre:  The user supplies an initial configuration of living cells.
Post: The program prints a sequence of pictures showing the changes in
      the configuration of living cells according to the rules for
      the game of Life.
Uses: The class Life and its methods initialize(),
      print() and update().
      The functions  instructions(),  user_says_yes().
*/

{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}
```

REVIEW QUESTIONS

1. *In terms of the Θ and Ω notations, compare the difference in time required for table lookup and for list searching.*

Table lookup generally requires $\Theta(1)$ time compared to $\Omega(\log n)$ time required by list searching.

2. *What are row-major and column-major ordering?*

Both row- and column-major ordering are methods of storing a rectangular array in a one-dimensional array. Row-major ordering is the method of storing the rectangular array by row, that is the first row as read from left to right is stored, then the second row, and so on. Column-major ordering is storage by column, that is the first column as read from top to bottom is stored, then the second, and so on.

3. *Why do jagged tables require access arrays instead of index functions?*

A jagged table has no relation between the position of a row and its length, and therefore it is impossible to produce an index function. An access array is the only way to access specified entries in jagged tables.

4. *For what purpose are inverted tables used?*

Inverted tables are used for tables which may require access by more than just one key. The example given in the text involved a table of telephone subscribers which could be easily accessed by name, address, or telephone number.

5. *What is the difference in purpose, if any, between an index function and an access array?*

There is no difference in purpose; an access array and an index function both provide a method of accessing a table of entries.

6. *What operations are available for an abstract table?*

The operations available for an abstract table are *retrieval*, *assignment*, *insertion*, and *deletion*.

7. *What operations are usually easier for a list than for a table?*

It is usually easier to detect if a list is either empty or full than it is to check these conditions on a table. A table may provide no way to access its entries sequentially.

8. *In 20 words or less, describe how radix sort works.*

Distribute the keys into sublists according to the character in each position of the key, starting with the least significant.

9. *In radix sort, why are the keys usually partitioned first by the least significant position, not the most significant?*

The keys are partitioned first by the least significant position so that for each separate sublist we can simply recombine them and repeat the process. Otherwise, for character keys, for example, we would need 26 sublists for the first character and for each of these 26 sublists we would need 26 more and so on. If we start from the least significant position we can recombine the sublists into a single list at each stage and not lose the order of the positions already sorted.

10. *What is the difference in purpose, if any, between an index function and a hash function?*

Although an index function and a hash function are not intended for the same type of table, they do have the same purpose, that of allowing access to a data structure when the target position within the structure is not known.

11. *What objectives should be sought in the design of a hash function?*

A hash function should be quick and easy to calculate and spread the keys as evenly as possible over the hash table to avoid collisions.

12. *Name three techniques often built into hash functions.*

Three methods often employed in hash functions are truncation, folding, and modular arithmetic.

13. *What is clustering in a hash table?*

Clustering within a hash table occurs when records start to collect in long strings of adjacent positions.

14. *Describe two methods for minimizing clustering.*

Quadratic probing, rehashing, key-dependent increments, pseudorandom probing, and chaining are all designed to reduce or eliminate clustering.

15. *Name four advantages of a chained hash table over open addressing.*

A chained hash table has advantages over open addressing because of efficient use of space, an easy method of collision resolution, insurance against overflow, and an easy method of deleting keys.

16. *Name one advantage of open addressing over chaining.*

For hash tables storing records of small size, open addressing may provide a significant saving of memory used over chaining.

17. *If a hash function assigns 30 keys to random positions in a hash table of size 300, about how likely is it that there will be no collisions?*

We know from the Birthday Surprise that it is more than 50% likely that 2 out of 23 people share their birthday. Hence, if a collision of 23 keys out of 365 positions is likely, a collision of 30 keys out of 300 positions is more likely. The actual probability of no collisions is

$$\frac{299}{300} \times \frac{298}{300} \times \ldots \times \frac{271}{300} \approx 0.223.$$

TOC
Index
Help

Binary Trees

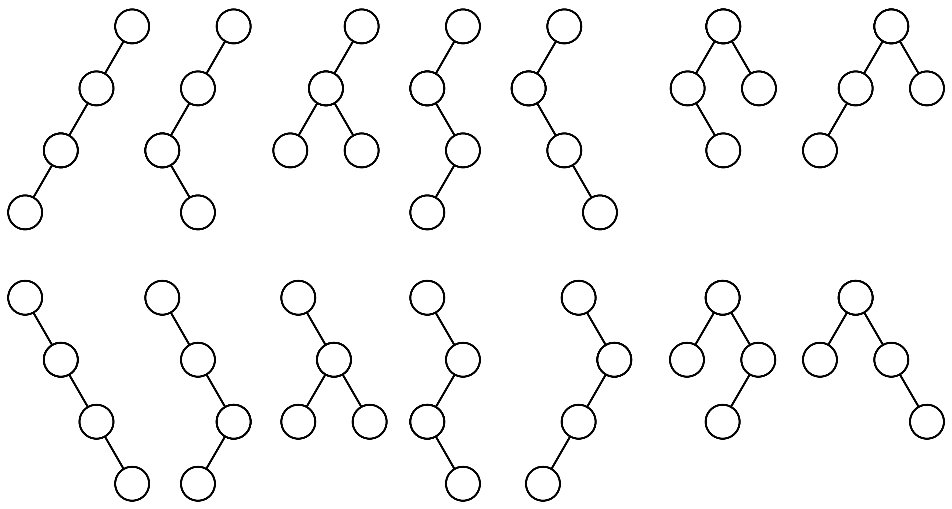
10

10.1 BINARY TREES

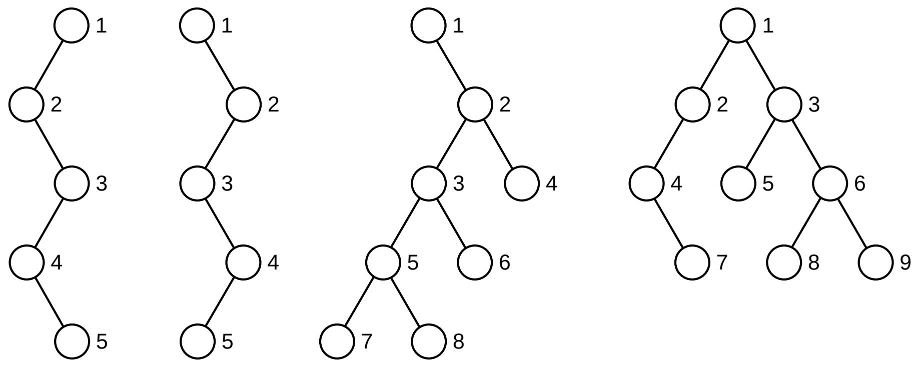
Exercises 10.1

E1. *Construct the 14 binary trees with four nodes.*

Answer



E2. *Determine the order in which the vertices of the following binary trees will be visited under* **(1)** *preorder,* **(2)** *inorder, and* **(3)** *postorder traversal.*



(a) (b) (c) (d)

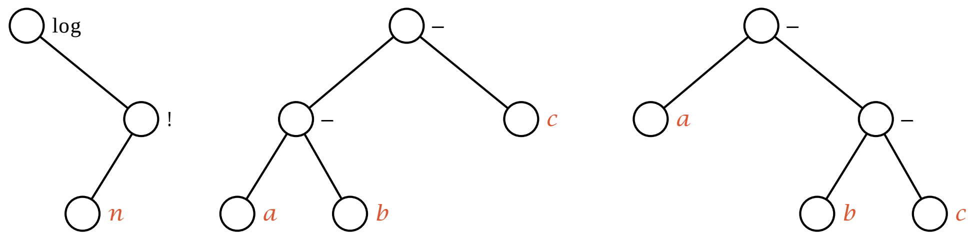
Answer	**(a)** **(1)** 1, 2, 3, 4, 5	**(2)** 2, 4, 5, 3, 1	**(3)** 5, 4, 3, 2, 1
	(b) **(1)** 1, 2, 3, 4, 5	**(2)** 1, 3, 5, 4, 2	**(3)** 5, 4, 3, 2, 1
	(c) **(1)** 1, 2, 3, 5, 7, 8, 6, 4	**(2)** 1, 7, 5, 8, 3, 6, 2, 4	**(3)** 7, 8, 5, 6, 3, 4, 2, 1
	(d) **(1)** 1, 2, 4, 7, 3, 5, 6, 8, 9	**(2)** 4, 7, 2, 1, 5, 3, 8, 6, 9	**(3)** 7, 4, 2, 5, 8, 9, 6, 3, 1

E3. *Draw expression trees for each of the following expressions, and show the order of visiting the vertices in* **(1)** *preorder,* **(2)** *inorder, and* **(3)** *postorder:*

(a) $\log n!$ **(c)** $a - (b - c)$

(b) $(a - b) - c$ **(d)** $(a < b)$ and $(b < c)$ and $(c < d)$

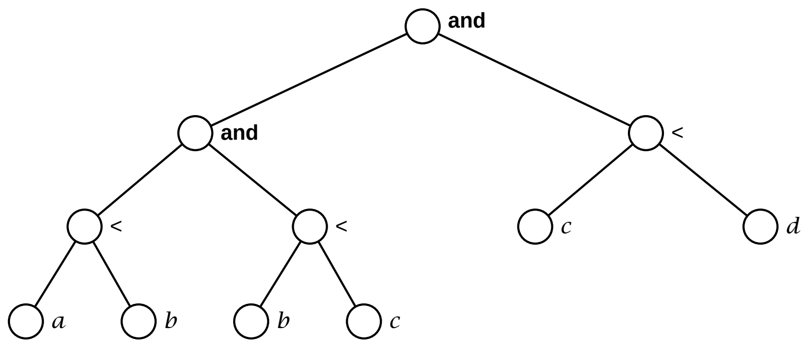
Answer **(a)** **(b)** **(c)**



$\log n!$ $(a - b) - c$ $a - (b - c)$

(1) preorder:	**(a)** log ! n	**(b)** – – a b c	**(c)** – a – b c
(2) inorder:	**(a)** log n !	**(b)** a – b – c	**(c)** a – b – c
(3) postorder:	**(a)** n ! log	**(b)** a b – c –	**(c)** a b c – –

(d)



$(a < b)$ **and** $(b < c)$ **and** $(c < d)$

preorder:	and and $<$ a b $<$ b c $<$ c d
inorder:	a $<$ b and b $<$ c and c $<$ d
postorder:	a b $<$ b c $<$ and c d $<$ and

E4. *Write a method and the corresponding recursive function to count all the nodes of a linked binary tree.*

Answer
```
template <class Entry>
int Binary_tree<Entry> :: size( ) const
/* Post: The number of entries in the binary tree is returned. */
{
  return recursive_size(root);
}

template <class Entry>
int Binary_tree<Entry> :: recursive_size(Binary_node<Entry> *sub_root) const
/* Post: The number of entries in the subtree rooted at sub_root is returned. */
{
  if (sub_root == NULL) return 0;
  return 1 + recursive_size(sub_root->left) + recursive_size(sub_root->right);
}
```

E5. *Write a method and the corresponding recursive function to count the leaves (i.e., the nodes with both subtrees empty) of a linked binary tree.*

Answer

```
template <class Entry>
int Binary_tree<Entry> :: recursive_leaf_count
                    (Binary_node<Entry> *sub_root) const
/* Post: The number of leaves in the subtree rooted at sub_root is returned. */
{
   if (sub_root == NULL) return 0;
   if (sub_root->left == NULL && sub_root->right == NULL) return 1;
   return recursive_leaf_count(sub_root->left)
               + recursive_leaf_count(sub_root->right);
}
template <class Entry>
int Binary_tree<Entry> :: leaf_count( ) const
/* Post: The number of leaves in the tree is returned. */
{
   return recursive_leaf_count(root);
}
```

E6. *Write a method and the corresponding recursive function to find the height of a linked binary tree, where an empty tree is considered to have height 0 and a tree with only one node has height 1.*

Answer

```
template <class Entry>
int Binary_tree<Entry> :: height( ) const
/* Post: The height of the binary tree is returned. */
{
   return recursive_height(root);
}

template <class Entry>
int Binary_tree<Entry> :: recursive_height(Binary_node<Entry> *sub_root) const
/* Post: The height of the subtree rooted at sub_root is returned. */
{
   if (sub_root == NULL) return 0;
   int l = recursive_height(sub_root->left);
   int r = recursive_height(sub_root->right);
   if (l > r) return 1 + l;
   else return 1 + r;
}
```

Binary_tree insert

E7. *Write a method and the corresponding recursive function to insert an* Entry, *passed as a parameter, into a linked binary tree. If the* root *is empty, the new entry should be inserted into the root, otherwise it should be inserted into the shorter of the two subtrees of the root (or into the left subtree if both subtrees have the same height).*

Answer

```
template <class Entry>
void Binary_tree<Entry> :: insert(const Entry &x)
/* Post: The Entry x is added to the binary tree. */
{
   recursive_insert(root, x);
}

template <class Entry>
void Binary_tree<Entry> :: recursive_insert(Binary_node<Entry> * &sub_root,
                                           const Entry &x)
/* Pre:  sub_root is either NULL or points to a subtree of the Binary_tree.
   Post: The Entry x has been inserted into the subtree in such a way that the properties of a
         binary search tree have been preserved.
   Uses: The functions recursive_insert, recursive_height */
```

```
{
   if (sub_root == NULL) sub_root = new Binary_node<Entry>(x);
   else
     if (recursive_height(sub_root->right) < recursive_height(sub_root->left))
        recursive_insert(sub_root->right, x);
   else
      recursive_insert(sub_root->left, x);
}
```

E8. *Write a method and the corresponding recursive function to traverse a binary tree (in whatever order you find convenient) and dispose of all its nodes. Use this method to implement a* Binary_tree *destructor.*

Answer Postorder traversal is the best choice, since a node is then not disposed until its left and right subtrees have both been cleared.

```
template <class Entry>
void Binary_tree<Entry> :: recursive_clear(Binary_node<Entry> * &sub_root)
/* Post: The subtree rooted at sub_root is cleared. */
{
   Binary_node<Entry> *temp = sub_root;
   if (sub_root == NULL) return;
   recursive_clear(sub_root->left);
   recursive_clear(sub_root->right);
   sub_root = NULL;
   delete temp;
}

template <class Entry>
void Binary_tree<Entry> :: clear( )
/* Post: All entries of the binary tree are removed. */
{
   recursive_clear(root);
}

template <class Entry>
Binary_tree<Entry> :: ~Binary_tree( )
/* Post: All entries of the binary tree are removed. All dynamically allocated memory in the
         structure is deleted. */
{
   clear( );
}
```

E9. *Write a copy constructor*

```
Binary_tree<Entry> :: Binary_tree(const Binary_tree<Entry> &original)
```

Binary_tree *copy constructor*

that will make a copy of a linked binary tree. The constructor should obtain the necessary new nodes from the system and copy the data from the nodes of the old tree to the new one.

Answer

```
template <class Entry>
Binary_tree<Entry> :: Binary_tree (const Binary_tree<Entry> &original)
/* Post: A new binary tree is initialized to copy original. */
{
   root = recursive_copy(original.root);
}
```

```
template <class Entry>
Binary_node<Entry> *Binary_tree<Entry> :: recursive_copy(
                                          Binary_node<Entry> *sub_root)
/* Post: The subtree rooted at sub_root is copied, and a pointer to the root of the new copy is
         returned. */
{
   if (sub_root == NULL) return NULL;
   Binary_node<Entry> *temp = new Binary_node<Entry>(sub_root->data);
   temp->left = recursive_copy(sub_root->left);
   temp->right = recursive_copy(sub_root->right);
   return temp;
}
```

E10. *Write an overloaded binary tree assignment operator*

```
Binary_tree<Entry> & Binary_tree<Entry> :: operator =
                                (const Binary_tree<Entry> &original)}.
```

Answer

```
template <class Entry>
Binary_tree<Entry> &Binary_tree<Entry> :: operator = (const
                                        Binary_tree<Entry> &original)
/* Post: The binary tree is reset to copy original. */
{
   Binary_tree<Entry> new_copy(original);
   clear( );
   root = new_copy.root;
   new_copy.root = NULL;
   return *this;
}
```

E11. *Write a function to perform a* ***double-order traversal*** *of a binary tree, meaning that at each node of the tree, the function first visits the node, then traverses its left subtree (in double order), then visits the node again, then traverses its right subtree (in double order).*

double-order traversal

Answer

```
template <class Entry>
void Binary_tree<Entry> :: recursive_double_traverse
       (Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/* Post: The subtree rooted at sub_root is doubly traversed. */
{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->right, visit);
   }
}
template <class Entry>
void Binary_tree<Entry> :: double_traverse(void (*visit)(Entry &))
/* Post: The tree is doubly traversed. */
{
   recursive_double_traverse(root, visit);
}
```

E12. *For each of the binary trees in Exercise E2, determine the order in which the nodes will be visited in the mixed order given by invoking method* A*:*

```
void Binary_tree<Entry> ::
  A(void (*visit)(Entry &))
{
  if (root != NULL) {
    (*visit)(root->data);
    root->left.B(visit);
    root->right.B(visit);
  }
}
```

```
void Binary_tree<Entry> ::
  B(void (*visit)(Entry &))
{
  if (root != NULL) {
    root->left.A(visit);
    (*visit)(root->data);
    root->right.A(visit);
  }
}
```

Answer **(a)** 1, 2, 3, 4, 5 **(b)** 1, 3, 5, 4, 2 **(c)** 1, 3, 7, 5, 8, 6, 2, 4 **(d)** 1, 4, 7, 2, 5, 3, 6, 8, 9

printing a binary tree

E13. (a) *Suppose that* Entry *is the type* **char**. *Write a function that will print all the entries from a binary tree in the* ***bracketed form*** (data: LT, RT) *where* data *is the* Entry *in the root,* LT *denotes the left subtree of the root printed in bracketed form, and* RT *denotes the right subtree in bracketed form. For example, the first tree in Figure 10.3 will be printed as*

(+ : (a: (: ,), (: ,)), (b: (: ,), (: ,)))

(b) *Modify the function so that it prints nothing instead of* (: ,) *for an empty subtree, and* x *instead of* (x: ,) *for a subtree consisting of only one node with the* Entry x. *Hence the preceding tree will now print as* (+ : a, b).

Answer Since both of the requested functions are minor variations of each other, we shall only list the solution to the second half of this exercise. In this implementation, we assume that tree entries can be printed with the stream output operator ≪ . For non-primitive entry types, this requires the implementation of an overloaded output operator.

```
template <class Entry>
void Binary_tree<Entry> :: recursive_bracketed
      (Binary_node<Entry> *sub_root)
/* Post: The subtree rooted at sub_root is printed with brackets. */
{
   if (sub_root != NULL) {
      if (sub_root->left ==  NULL && sub_root->right ==  NULL)
         cout << sub_root->data;
      else {
         cout << "(" << sub_root->data << ":";
         recursive_bracketed(sub_root->left);
         cout << ",";
         recursive_bracketed(sub_root->right);
         cout << ")";
      }
   }
}
template <class Entry>
void Binary_tree<Entry> :: bracketed_print()
/* Post: The tree is printed with brackets. */
{
   recursive_bracketed(root);
   cout << endl;
}
```

E14. *Write a function that will interchange all left and right subtrees in a linked binary tree. See the example in Figure 10.7.*

Answer
```
template <class Entry>
void Binary_tree<Entry> :: recursive_interchange
        (Binary_node<Entry> *sub_root)
/* Post: In the subtree rooted at sub_root all pairs of left and right links are swapped. */
{
   if (sub_root != NULL) {
      Binary_node<Entry> *tmp = sub_root->left;
      sub_root->left = sub_root->right;
      sub_root->right = tmp;
      recursive_interchange(sub_root->left);
      recursive_interchange(sub_root->right);
   }
}
template <class Entry>
void Binary_tree<Entry> :: interchange( )
/* Post: In the tree all pairs of left and right links are swapped round. */
{
   recursive_interchange(root);
}
```

E15. *Write a function that will traverse a binary tree level by level. That is, the root is visited first, then the immediate children of the root, then the grandchildren of the root, and so on. [Hint: Use a queue to keep track of the children of a node until it is time to visit them. The nodes in the first tree of Figure 10.7 are numbered in level-by-level ordering.]*

level-by-level traversal

Answer In this solution, we assume that an implementation of a **template class** Queue with our standard Queue methods is available.

```
template <class Entry>
void Binary_tree<Entry> :: level_traverse(void (*visit)(Entry &))
/* Post: The tree is traversed level by level, starting from the top. The operation *visit is applied
         to all entries. */
{
   Binary_node<Entry> *sub_root;
   if (root != NULL) {
      Queue < Binary_node<Entry> * > waiting_nodes;
      waiting_nodes.append(root);
      do {
         waiting_nodes.retrieve(sub_root);
         (*visit)(sub_root->data);
         if (sub_root->left) waiting_nodes.append(sub_root->left);
         if (sub_root->right) waiting_nodes.append(sub_root->right);
         waiting_nodes.serve( );
      } while (!waiting_nodes.empty( ));
   }
}
```

E16. *Write a function that will return the width of a linked binary tree, that is, the maximum number of nodes on the same level.*

Answer We perform a level by level traverse to measure the width. To perform the traversal, we assume that an implementation of a **template class** Queue with our standard Queue methods is available.

```
template <class Entry>
int Binary_tree<Entry> :: width( )
/* Post: The width of the tree is returned. */
```

```
{
  if (root == NULL) return 0;
  Extended_queue < Binary_node<Entry> * > current_level;
  current_level.append(root);
  int max_level = 0;
  while (current_level.size( ) != 0) {
    if (current_level.size( ) > max_level)
      max_level = current_level.size( );
    Extended_queue < Binary_node<Entry> * > next_level;
    do {
      Binary_node<Entry> *sub_root;
      current_level.retrieve(sub_root);
      if (sub_root->left) next_level.append(sub_root->left);
      if (sub_root->right) next_level.append(sub_root->right);
      current_level.serve( );
    } while (!current_level.empty( ));
    current_level = next_level;
  }
  return max_level;
}
```

traversal sequences *For the following exercises, it is assumed that the data stored in the nodes of the binary trees are all distinct, but it is not assumed that the trees are binary search trees. That is, there is no necessary connection between any ordering of the data and their location in the trees. If a tree is traversed in a particular order, and each key is printed when its node is visited, the resulting sequence is called the sequence corresponding to that traversal.*

E17. *Suppose that you are given two sequences that supposedly correspond to the preorder and inorder traversals of a binary tree. Prove that it is possible to reconstruct the binary tree uniquely.*

Answer Since preorder traversal shows the root of the tree first, it is easy to detect the root key. The inorder sequence then shows which of the remaining keys occur in the left and the right subtrees as those appearing on the left and on the right of the root. Proceeding by recursion with each of the subtrees, we can then reconstruct the binary tree uniquely from the preorder and inorder traversals.

E18. *Either prove or disprove (by finding a counterexample) the analogous result for inorder and postorder traversal.*

Answer Since postorder traversal shows the root of the tree last, it is easy to detect the root key. The inorder sequence then shows which of the remaining keys occur in the left and the right subtrees as those appearing on the left and on the right of the root. Proceeding by recursion with each of the subtrees, we can then reconstruct the binary tree uniquely from the postorder and inorder traversals.

E19. *Either prove or disprove the analogous result for preorder and postorder traversal.*

Answer It is not possible to reconstruct the binary tree uniquely from a preorder and postorder traversal. Consider the tree of two nodes whose root has key 1 and whose right child has key 2. This binary tree would have the same preorder and postorder traversal as the tree whose root has key 1 and whose left child has key 2.

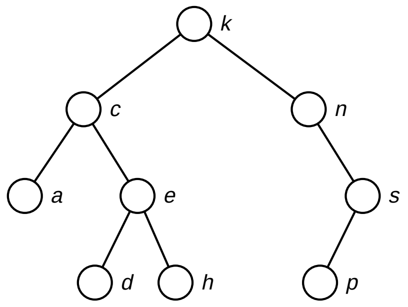
E20. *Find a pair of sequences of the same data that could not possibly correspond to the preorder and inorder traversals of the same binary tree. [Hint: Keep your sequences short; it is possible to solve this exercise with only three items of data in each sequence.]*

Answer The sequence a b c could not possibly represent the inorder traversal of a binary tree if the sequence b c a represented the preorder traversal. For the preorder sequence shows that b is the root, and then the inorder sequence shows that a is in the left subtree and c is in the right subtree. But then c cannot come before a as required for the preorder sequence.

10.2 BINARY SEARCH TREES

Exercises 10.2

The first several exercises are based on the following binary search tree. Answer each part of each exercise independently, using the original tree as the basis for each part.



E1. *Show the keys with which each of the following targets will be compared in a search of the preceding binary search tree.*

(a) c — *Answer* k, c

(d) a — k, c, a

(g) f — k, c, e, h (fails)

(b) s — *Answer* k, n, s

(e) d — k, c, e, d

(h) b — k, c, a (fails)

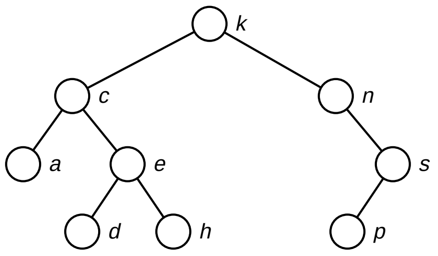
(c) k — *Answer* k

(f) m — k, n (fails)

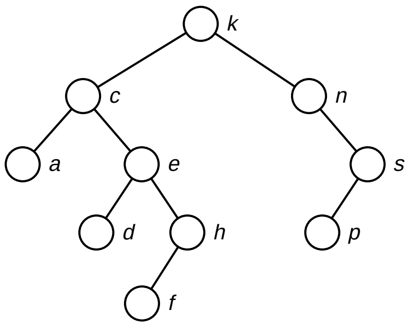
(i) t — k, n, s (fails)

E2. *Insert each of the following keys into the preceding binary search tree. Show the comparisons of keys that will be made in each case. Do each part independently, inserting the key into the original tree.*

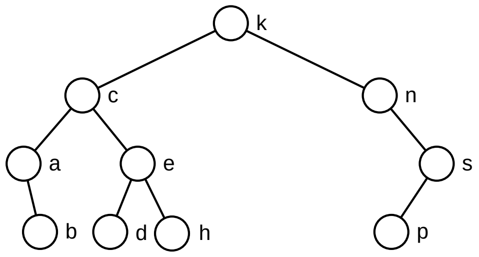
(a) m



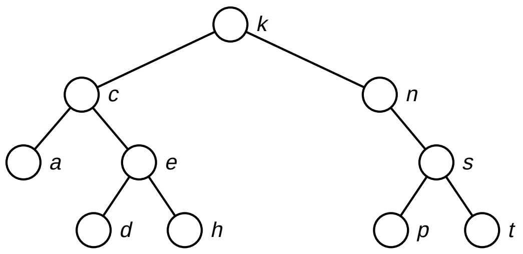
(b) f



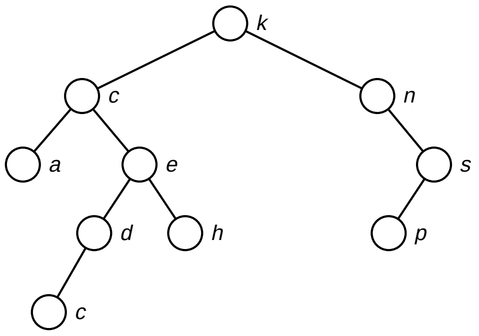
(c) b



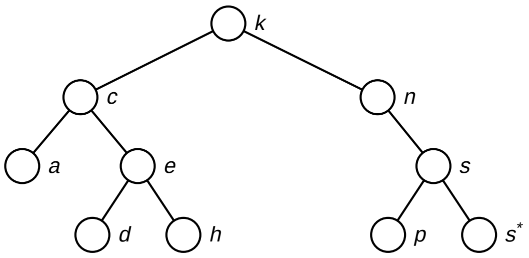
(d) t



(e) *c*

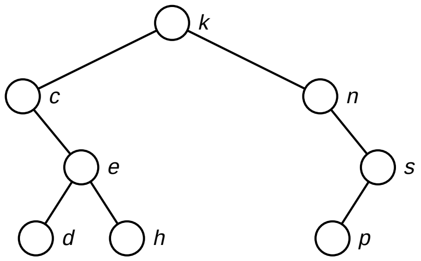


(f) *s*

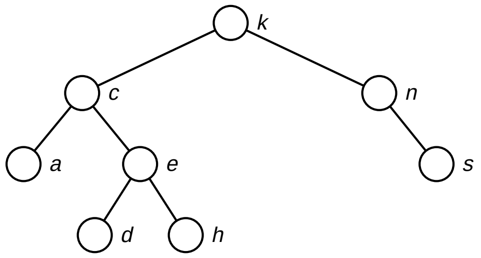


E3. *Delete each of the following keys from the preceding binary search tree, using the algorithm developed in this section. Do each part independently, deleting the key from the original tree.*

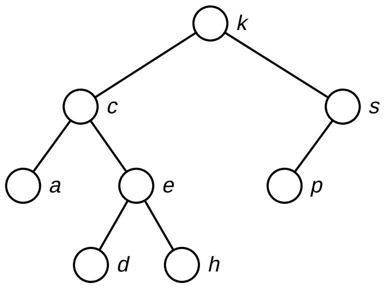
(a) *a*



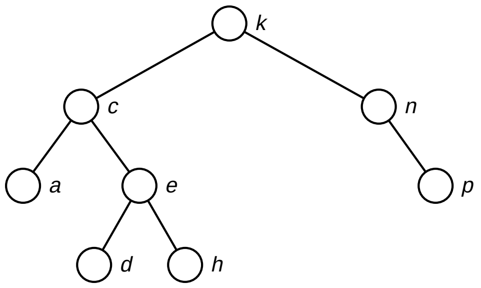
(b) *p*



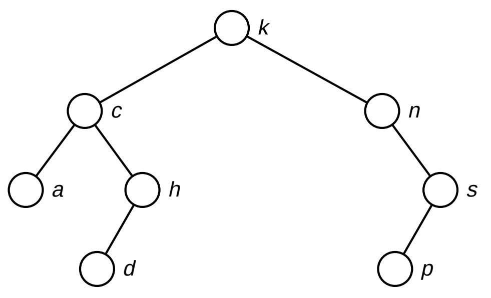
(c) *n*



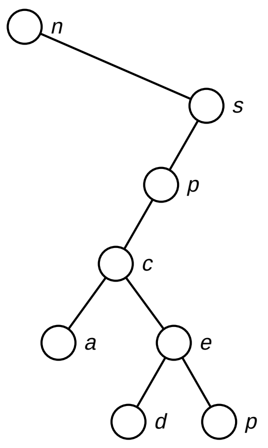
(d) *s*



(e) *e*



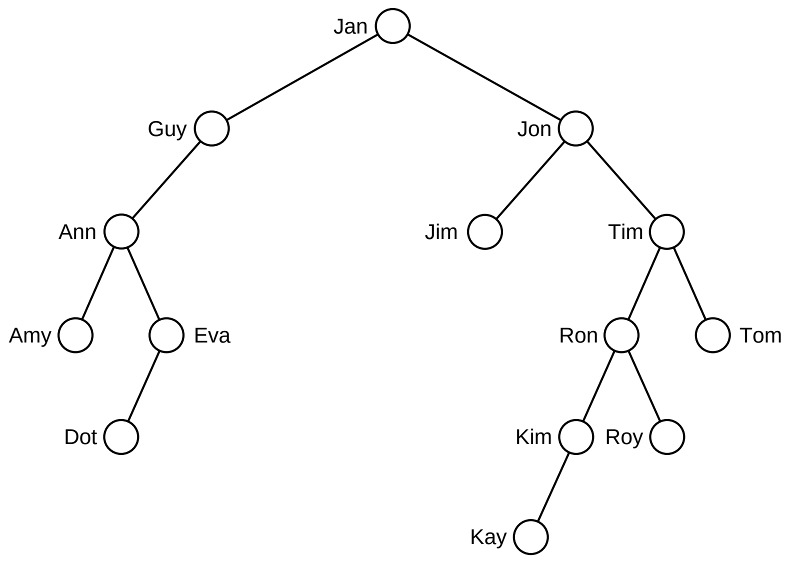
(f) *k*



E4. *Draw the binary search trees that function* insert *will construct for the list of 14 names presented in each of the following orders and inserted into a previously empty binary search tree.*

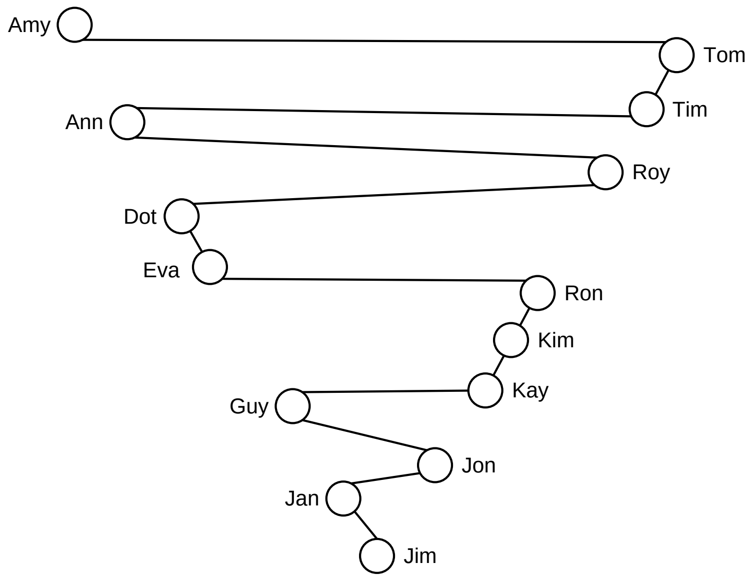
(a) Jan Guy Jon Ann Jim Eva Amy Tim Ron Kim Tom Roy Kay Dot

Answer



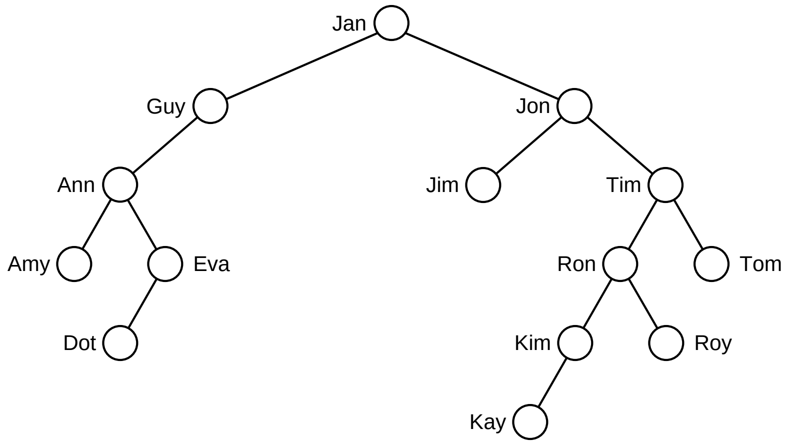
(b) Amy Tom Tim Ann Roy Dot Eva Ron Kim Kay Guy Jon Jan Jim

Answer



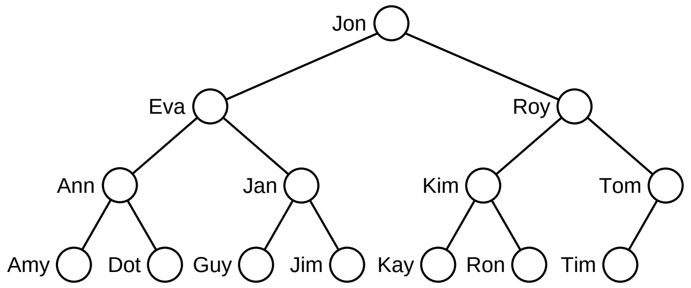
(c) Jan Jon Tim Ron Guy Ann Jim Tom Amy Eva Roy Kim Dot Kay

Answer



(d) Jon Roy Tom Eva Tim Kim Ann Ron Jan Amy Dot Guy Jim Kay

Answer



E5. *Consider building two binary search trees containing the integer keys 1 to 63, inclusive, received in the orders*

(a) *all the odd integers in order (1, 3, 5, . . . , 63), then 32, 16, 48, then the remaining even integers in order (2, 4, 6, . . .).*

(b) *32, 16, 48, then all the odd integers in order (1, 3, 5, . . . , 63), then the remaining even integers in order (2, 4, 6, . . .).*

Which of these trees will be quicker to build? Explain why. [Try to answer this question without actually drawing the trees.]

Answer The second tree will be quicker to build, because it is more balanced than the first. (For example, in the first tree all entries go to the right of the root entry of 1, whereas in the second tree equal numbers of entries are placed on the two sides of the root entry of 32.)

E6. *All parts of this exercise refer to the binary search trees shown in Figure 10.8 and concern the different orders in which the keys a, b, . . . , g can be inserted into an initially empty binary search tree.*

(a) *Give four different orders for inserting the keys, each of which will yield the binary search tree shown in part (a).*

Answer

1. d, b, f, a, c, e, g
2. d, b, a, c, f, g, e
3. d, f, g, e, b, c, a
4. d, f, b, g, e, c, a

There are many other possible orders.

(b) *Give four different orders for inserting the keys, each of which will yield the binary search tree shown in part (b).*

Answer

1. e, b, a, d, c, f, g
2. e, b, d, a, c, f, g
3. e, f, g, b, d, c, a
4. e, b, f, a, d, g, e

There are several other possible orders.

(c) *Give four different orders for inserting the keys, each of which will yield the binary search tree shown in part (c).*

Answer

1. a, g, e, b, d, c, f
2. a, g, e, b, d, f, c
3. a, g, e, b, f, d, c
4. a, g, e, f, b, d, c

These are the only possible orders.

(d) *Explain why there is only one order for inserting the keys that will produce a binary search tree that reduces to a given chain, such as the one shown in part (d) or in part (e).*

Answer The parent must be inserted before its children. In a chain, each node (except the last) is the parent of exactly one other node, and hence the nodes must be inserted in descending order, starting with the root.

E7. *The use of recursion in function* insert *is not essential, since it is tail recursion. Rewrite function* insert *in nonrecursive form. [You will need a local pointer variable to move through the tree.]*

Answer We introduce a local pointer variable sub_root that will move to the left or right subtree as we compare keys, searching for the place to insert the new node. If the new key is not a duplicate, then this search would normally terminate with sub_root == NULL. To make the insertion, however, we need the value of sub_root one step before it becomes NULL. We therefore introduce another pointer parent to record this information. We must be careful to treat the case of an empty tree specially, because in this case, no initial value can be supplied for the pointer parent.

```
template <class Record>
Error_code Search_tree<Record> :: insert(const Record &new_data)
{
  if (root == NULL) {
    root = new Binary_node<Record>(new_data);
    return success;
  }
  Binary_node<Record> *sub_root = root, *parent;
  do {
    parent = sub_root;
    if (new_data < sub_root->data) sub_root = sub_root->left;
    else if (new_data > sub_root->data) sub_root = sub_root->right;
    else return duplicate_error;
  } while (sub_root != NULL);
  sub_root = new Binary_node<Record>(new_data);
  if (new_data < parent->data) parent->left = sub_root;
  else parent->right = sub_root;
  return success;
}
```

Programming Projects 10.2

P1. *Prepare a package containing the declarations for a binary search tree and the functions developed in this section. The package should be suitable for inclusion in any application program.*

Answer Listings of most of the methods appear in the text. Here is the complete package. Binary node class:

```
enum Balance_factor { left_higher, equal_height, right_higher };

template <class Entry>
struct Binary_node {

//    data members:
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
//    constructors:
   Binary_node();
   Binary_node(const Entry &x);

//    virtual methods:
   virtual void set_balance(Balance_factor b);
   virtual Balance_factor get_balance() const;

};
```

Implementation:

```
template <class Entry>
Binary_node<Entry>::Binary_node()
{
   left = right = NULL;
}

template <class Entry>
Binary_node<Entry>::Binary_node(const Entry &x)
{
   data = x;
   left = right = NULL;
}
```

```
template <class Entry>
void Binary_node<Entry>::set_balance(Balance_factor b)
{
}

template <class Entry>
Balance_factor Binary_node<Entry>::get_balance() const
{
   return equal_height;
}

template <class Entry>
void Binary_tree<Entry>::recursive_inorder(Binary_node<Entry> *sub_root,
                                           void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points to a subtree of
      the Binary_tree.
Post: The subtree has been been traversed in inorder sequence.
Uses: The function recursive_inorder recursively
*/

{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}

template <class Entry>
void Binary_tree<Entry>::recursive_preorder(Binary_node<Entry> *sub_root,
                                            void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points to a subtree of
     the Binary_tree.
Post: The subtree has been been traversed in preorder sequence.
Uses: The function recursive_preorder recursively

*/

{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_preorder(sub_root->left, visit);
      recursive_preorder(sub_root->right, visit);
   }
}

template <class Entry>
void Binary_tree<Entry>::recursive_postorder(Binary_node<Entry> *sub_root,
                                             void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points to a subtree of
     the Binary_tree.
Post: The subtree has been been traversed in postorder sequence.
Uses: The function recursive_postorder recursively
*/
```

```
{
   if (sub_root != NULL) {
      recursive_postorder(sub_root->left, visit);
      recursive_postorder(sub_root->right, visit);
      (*visit)(sub_root->data);
   }
}

template <class Entry>
void Binary_tree<Entry>::recursive_insert(Binary_node<Entry> *&sub_root,
                                          const Entry &x)
/*
Pre:  sub_root is either NULL or points to a subtree of
     the Binary_tree.
Post: The Entry  has been inserted into the subtree in such a way
      that the properties of a binary search tree have been preserved.
Uses: The functions recursive_insert, recursive_height
*/

{
   if (sub_root == NULL) sub_root = new Binary_node<Entry>(x);
   else
     if (recursive_height(sub_root->right) <
                         recursive_height(sub_root->left))
        recursive_insert(sub_root->right, x);
   else
      recursive_insert(sub_root->left, x);
}

template <class Entry>
int Binary_tree<Entry>::recursive_size(Binary_node<Entry> *sub_root) const
/*
Post: The number of entries in the subtree rooted at
      sub_root is returned.
*/

{
   if (sub_root == NULL) return 0;
   return 1 + recursive_size(sub_root->left) +
              recursive_size(sub_root->right);
}

template <class Entry>
int Binary_tree<Entry>::
                recursive_height(Binary_node<Entry> *sub_root) const
/*
Post: The height of the subtree rooted at
      sub_root is returned.
*/

{
   if (sub_root == NULL) return 0;
   int l = recursive_height(sub_root->left);
   int r = recursive_height(sub_root->right);
   if (l > r) return 1 + l;
   else return 1 + r;
}
```

```
template <class Entry>
void Binary_tree<Entry>::recursive_clear(Binary_node<Entry> *&sub_root)
/*
Post: The subtree rooted at
      sub_root is cleared.
*/

{
   Binary_node<Entry> *temp = sub_root;
   if (sub_root == NULL) return;
   recursive_clear(sub_root->left);
   recursive_clear(sub_root->right);
   sub_root = NULL;
   delete temp;
}

template <class Entry>
Binary_node<Entry> *Binary_tree<Entry>::recursive_copy(
                                          Binary_node<Entry> *sub_root)
/*
Post: The subtree rooted at
      sub_root is copied, and a pointer to
      the root of the new copy is returned.
*/

{
   if (sub_root == NULL) return NULL;
   Binary_node<Entry> *temp = new Binary_node<Entry>(sub_root->data);
   temp->left = recursive_copy(sub_root->left);
   temp->right = recursive_copy(sub_root->right);
   return temp;
}

template <class Entry>
void Binary_tree<Entry>::recursive_swap(Binary_node<Entry> *sub_root)
/*
Post: In the subtree rooted at
      sub_root all pairs of left and right
      links are swapped.
*/

{
   if (sub_root == NULL) return;
   Binary_node<Entry> *temp = sub_root->left;
   sub_root->left = sub_root->right;
   sub_root->right = temp;
   recursive_swap(sub_root->left);
   recursive_swap(sub_root->right);
}

template <class Entry>
Binary_node<Entry> *&Binary_tree<Entry>::find_node(
                     Binary_node<Entry> *&sub_root, const Entry &x) const
/*
Post: If the subtree rooted at
      sub_root contains x as a entry, a pointer
      to the subtree node storing x is returned.
      Otherwise NULL is returned.
*/
```

```
{
   if (sub_root == NULL || sub_root->data == x) return sub_root;
   else {
      Binary_node<Entry> *&temp = find_node(sub_root->left, x);
      if (temp != NULL) return temp;
      else return find_node(sub_root->right, x);
   }
}

template <class Entry>
Error_code Binary_tree<Entry>::remove_root(Binary_node<Entry> *&sub_root)
/*
Pre:  sub_root is either NULL or points to a subtree of
     the Binary_tree
Post: If sub_root is NULL, a code of not_present is returned.
      Otherwise the root of the subtree is removed in such a way
      that the properties of a binary search tree are preserved.
      The parameter sub_root is reset as
      the root of the modified subtree and
      success is returned.
*/

{
   if (sub_root == NULL) return not_present;
   Binary_node<Entry> *to_delete = sub_root;
                                 // Remember node to delete at end.
   if (sub_root->right == NULL) sub_root = sub_root->left;
   else if (sub_root->left == NULL) sub_root = sub_root->right;

   else {                        //  Neither subtree is empty
      to_delete = sub_root->left;   //  Move left to find predecessor
      Binary_node<Entry> *parent = sub_root; //  parent of to_delete
      while (to_delete->right != NULL) { //to_delete is not the predecessor
         parent = to_delete;
         to_delete = to_delete->right;
      }
      sub_root->data = to_delete->data;  //  Move from to_delete to root
      if (parent == sub_root) sub_root->left = to_delete->left;
      else parent->right = to_delete->left;
   }

   delete to_delete;   //  Remove to_delete from tree
   return success;
}
```

Binary tree class:

```
template <class Entry>
class Binary_tree {
public:

//   Add methods here.

   void double_traverse(void (*visit)(Entry &));
   void bracketed_print();
   void interchange();
   void level_traverse(void (*visit)(Entry &));
   int width();
   int leaf_count() const;
   Binary_tree();
   bool empty() const;
   void preorder(void (*visit)(Entry &));
   void inorder(void (*visit)(Entry &));
   void postorder(void (*visit)(Entry &));

   int size() const;
   void clear();
   int height() const;
   void insert(const Entry &);

   Binary_tree (const Binary_tree<Entry> &original);
   Binary_tree & operator =(const Binary_tree<Entry> &original);
   ~Binary_tree();

   void recursive_double_traverse
       (Binary_node<Entry> *sub_root, void (*visit)(Entry &));

   void recursive_bracketed
       (Binary_node<Entry> *sub_root);

   void recursive_interchange
       (Binary_node<Entry> *sub_root);

   int recursive_leaf_count
                  (Binary_node<Entry> *sub_root) const;
   Error_code remove(Entry &);
   void swap();
   void recursive_inorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_preorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_postorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_insert(Binary_node<Entry> *&sub_root, const Entry &x);
   int recursive_size(Binary_node<Entry> *sub_root) const;
   int recursive_height(Binary_node<Entry> *sub_root) const;
   void recursive_clear(Binary_node<Entry> *&sub_root);
   Binary_node<Entry> *recursive_copy(Binary_node<Entry> *sub_root);
   void recursive_swap(Binary_node<Entry> *sub_root);
   Binary_node<Entry> *&find_node(Binary_node<Entry> *&,
                                  const Entry &) const;
   Error_code remove_root(Binary_node<Entry> *&sub_root);

protected:
   //   Add auxiliary function prototypes here.
   Binary_node<Entry> *root;
};
```

Implementation:

```
template <class Entry>
Binary_tree<Entry>::Binary_tree()
/*
Post: An empty binary tree has been created.
*/

{
   root = NULL;
}
```

TOC
Index
Help

```
template <class Entry>
bool Binary_tree<Entry>::empty() const
/*
Post: A result of true is returned if the binary tree is empty.
      Otherwise, false is returned.
*/
{
   return root == NULL;
}

template <class Entry>
void Binary_tree<Entry>::inorder(void (*visit)(Entry &))
/*
Post: The tree has been been traversed in infix
      order sequence.
Uses: The function recursive_inorder
*/
{
   recursive_inorder(root, visit);
}

template <class Entry>
Error_code Binary_tree<Entry>::remove(Entry &x)
/*
Post: An entry of the binary tree with Key matching x is
      removed.  If there is no such entry a code of not_present
      is returned.
*/
{
   Binary_node<Entry> *&found = find_node(root, x);
   return remove_root(found);
}

template <class Entry>
void Binary_tree<Entry>::insert(const Entry &x)
/*
Post: The Entry x is added to the binary tree.
*/
{
   recursive_insert(root, x);
}

template <class Entry>
void Binary_tree<Entry>::preorder(void (*visit)(Entry &))
/*
Post: The binary tree is traversed in prefix order,
      applying the operation *visit at each entry.
*/
{
   recursive_preorder(root, visit);
}

template <class Entry>
void Binary_tree<Entry>::postorder(void (*visit)(Entry &))
/*
Post: The binary tree is traversed in postfix order,
      applying the operation *visit at each entry.
*/
```

```
{
   recursive_postorder(root, visit);
}

template <class Entry>
int Binary_tree<Entry>::size() const
/*
Post: The number of entries in the binary tree is returned.
*/

{
   return recursive_size(root);
}

template <class Entry>
int Binary_tree<Entry>::height() const
/*
Post: The height of the binary tree is returned.
*/

{
   return recursive_height(root);
}

template <class Entry>
void Binary_tree<Entry>::clear()
/*
Post: All entries of the binary tree are removed.
*/

{
   recursive_clear(root);
}

template <class Entry>
Binary_tree<Entry>::~Binary_tree()
/*
Post: All entries of the binary tree are removed.
      All dynamically allocated memory in the structure
      is deleted.
*/

{
   clear();
}

template <class Entry>
Binary_tree<Entry>::Binary_tree (const Binary_tree<Entry> &original)
/*
Post: A new binary tree is initialized to copy original.
*/

{
   root = recursive_copy(original.root);
}

template <class Entry>
Binary_tree<Entry> &Binary_tree<Entry>::operator =(const
                                        Binary_tree<Entry> &original)
/*
Post: The binary tree is reset to copy original.
*/
```

```
{
   Binary_tree<Entry> new_copy(original);
   clear();
   root = new_copy.root;
   new_copy.root = NULL;
   return *this;
}

template <class Entry>
void Binary_tree<Entry>::swap()
/*
Post: All left and right subtrees are switched
      in the binary tree.
*/

{
   recursive_swap(root);
}

template <class Entry>
void Binary_tree<Entry>::recursive_double_traverse
       (Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/*
Post: The subtree rooted at sub_root is
      doubly traversed.
*/

{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->right, visit);
   }
}
template <class Entry>
void Binary_tree<Entry>::double_traverse(void (*visit)(Entry &))
/*
Post: The tree is doubly traversed.
*/

{
   recursive_double_traverse(root, visit);
}

template <class Entry>
void Binary_tree<Entry>::recursive_bracketed
       (Binary_node<Entry> *sub_root)
/*
Post: The subtree rooted at sub_root is
      printed with brackets.
*/
```

```
{
   if (sub_root != NULL) {
      if (sub_root->left == NULL && sub_root->right == NULL)
         cout << sub_root->data;
      else {
         cout << "(" << sub_root->data << ":";
         recursive_bracketed(sub_root->left);
         cout << ",";
         recursive_bracketed(sub_root->right);
         cout << ")";
      }
   }
}
template <class Entry>
void Binary_tree<Entry>::bracketed_print()
/*
Post: The tree is printed with brackets.
*/

{
   recursive_bracketed(root);
   cout << endl;
}

template <class Entry>
void Binary_tree<Entry>::recursive_interchange
       (Binary_node<Entry> *sub_root)
/*
Post: In the subtree rooted at
      sub_root all pairs of left and right
      links are swapped.
*/

{
   if (sub_root != NULL) {
      Binary_node<Entry> *tmp = sub_root->left;
      sub_root->left = sub_root->right;
      sub_root->right = tmp;
      recursive_interchange(sub_root->left);
      recursive_interchange(sub_root->right);
   }
}
template <class Entry>
void Binary_tree<Entry>::interchange()
/*
Post: In the tree all pairs of left and right
      links are swapped round.
*/

{
   recursive_interchange(root);
}

template <class Entry>
void Binary_tree<Entry>::level_traverse(void (*visit)(Entry &))
/*
Post: The tree is traversed level by level,
      starting from the top.  The operation
      *visit is applied to all entries.
*/
```

TOC
Index
Help

```
{
   Binary_node<Entry> *sub_root;
   if (root != NULL) {
      Queue<Binary_node<Entry> *> waiting_nodes;
      waiting_nodes.append(root);
      do {
         waiting_nodes.retrieve(sub_root);
         (*visit)(sub_root->data);
         if (sub_root->left) waiting_nodes.append(sub_root->left);
         if (sub_root->right) waiting_nodes.append(sub_root->right);
         waiting_nodes.serve();
      } while (!waiting_nodes.empty());
   }
}

template <class Entry>
int Binary_tree<Entry>::width()
/*
Post: The width of the tree is returned.
*/

{
   if (root == NULL) return 0;
   Extended_queue<Binary_node<Entry> *> current_level;
   current_level.append(root);
   int max_level = 0;
   while (current_level.size() != 0) {
      if (current_level.size() > max_level)
         max_level = current_level.size();
      Extended_queue<Binary_node<Entry> *> next_level;
      do {
         Binary_node<Entry> *sub_root;
         current_level.retrieve(sub_root);
         if (sub_root->left) next_level.append(sub_root->left);
         if (sub_root->right) next_level.append(sub_root->right);
         current_level.serve();
      } while (!current_level.empty());
      current_level = next_level;
   }
   return max_level;
}

template <class Entry>
int Binary_tree<Entry>::recursive_leaf_count
                  (Binary_node<Entry> *sub_root) const
/*
Post: The number of leaves in the subtree
      rooted at sub_root is returned.
*/

{
   if (sub_root == NULL) return 0;
   if (sub_root->left == NULL && sub_root->right == NULL) return 1;
   return recursive_leaf_count(sub_root->left)
            + recursive_leaf_count(sub_root->right);
}
template <class Entry>
int Binary_tree<Entry>::leaf_count() const
/*
```

```
Post: The number of leaves in the tree is returned.
*/

{
   return recursive_leaf_count(root);
}
```

Additional node functions for seach-tree nodes:

```
template <class Record>
Error_code Search_tree<Record>::search_and_insert(
           Binary_node<Record> *&sub_root, const Record &new_data)

{
   if (sub_root == NULL) {
      sub_root = new Binary_node<Record>(new_data);
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}

template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
   Binary_node<Record>* sub_root, const Record &target) const
{
   if (sub_root == NULL || sub_root->data == target) return sub_root;
   else if (sub_root->data < target)
      return search_for_node(sub_root->right, target);
   else return search_for_node(sub_root->left, target);
}

template <class Record>
Error_code Search_tree<Record>::search_and_destroy(
           Binary_node<Record>* &sub_root, const Record &target)

/*
Pre:  sub_root is either NULL or points to a subtree
      of the Search_tree.
Post: If the key of target is not in the subtree, a
      code of not_present is returned.
      Otherwise, a code of success is returned and the subtree node
      containing target has been removed in such a way that
      the properties of a binary search tree have been preserved.
Uses: Functions search_and_destroy recursively and remove_root
*/

{
   if (sub_root == NULL || sub_root->data == target)
      return remove_root(sub_root);
   else if (target < sub_root->data)
      return search_and_destroy(sub_root->left, target);
   else
      return search_and_destroy(sub_root->right, target);
}
```

A (derived) binary search tree class:

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
   Error_code tree_search(Record &target) const;

   Error_code search_and_insert(Binary_node<Record> *&, const Record &);
   Binary_node<Record>
              *search_for_node(Binary_node<Record>*, const Record&) const;
   Error_code search_and_destroy(Binary_node<Record> *&, const Record &);

private: //  Add auxiliary function prototypes here.
};
```

Implementation:

```
template <class Record>
Error_code Search_tree<Record>::insert(const Record &new_data)
{
   return search_and_insert(root, new_data);
}

template <class Record>
Error_code Search_tree<Record>::remove(const Record &target)
/*
Post: If a Record with a key matching that of target
      belongs to the Search_tree a code of
      success is returned and the corresponding node
      is removed from the tree.  Otherwise,
      a code of not_present is returned.
Uses: Function search_and_destroy
*/

{
   return search_and_destroy(root, target);
}

template <class Record>
Error_code Search_tree<Record>::tree_search(Record &target) const

/*

Post: If there is an entry in the tree whose key matches that in target,
      the parameter target is replaced by the corresponding record from
      the tree and a code of success is returned.  Otherwise
      a code of not_present is returned.
Uses: function search_for_node
*/

{
   Error_code result = success;
   Binary_node<Record> *found = search_for_node(root, target);
   if (found == NULL)
      result = not_present;
   else
      target = found->data;
   return result;
}
```

P2. *Produce a menu-driven demonstration program to illustrate the use of binary search trees. The entries may consist of keys alone, and the keys should be single characters. The minimum capabilities that the user should be able to demonstrate include constructing the tree, inserting and removing an entry with a given key, searching for a target key, and traversing the tree in the three standard orders. The project may be enhanced by the inclusion of additional capabilities written as exercises in this and the previous section. These include determining the size of the tree, printing out all entries arranged to show the shape of the tree, and traversing the tree in various ways. Keep the functions in your project as modular as possible, so that you can later replace the package of operations for a binary search tree by a functionally equivalent package for another kind of tree.*

demonstration program

Answer

```
#include "../../c/utility.h"

void help()
/* PRE:  None.
   POST: Instructions for the tree operations have been printed.
*/

{
     cout << "\n";
     cout << "\t[S]ize       [I]nsert    [D]elete \n"
             "\t[H]eight     [E]rase     [F]ind \n"
             "\t[W]idth      [C]ontents  [L]eafs     [B]readth first \n"
             "\t[P]reorder   P[O]storder I[N]order   [?]help \n" << endl;
}

#include <string.h>
char get_command()
/* PRE:  None.
   POST: A character command belonging to the set of legal commands for
         the tree demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command (? for help) and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("clwbe?sidfponqh",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}
```

```
char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include auxiliary data structures

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/sortable.h"
#include "../../8/linklist/merge.cpp"
#include "queue.h"
#include "queue.cpp"
#include "extqueue.h"
#include "extqueue.cpp"

// include binary search tree data structure

#include "node.h"
#include "tree.h"
#include "node.cpp"
#include "tree.cpp"
#include "stree.h"
#include "snode.cpp"
#include "stree.cpp"

int do_command(char c, Search_tree<char> &test_tree)
/* PRE:  The tree has been created and command is a valid tree
         operation.
   POST: The command has been executed.
   USES: All the functions that perform tree operations.
*/

{
   char x;
   switch (c) {
   case '?': help();
     break;

   case 's':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'i':
      cout << "Enter new character to insert:";
      x = get_char();
      test_tree.insert(x);
      break;

   case 'd':
      cout << "Enter character to remove:" << flush;
      x = get_char();
      if (test_tree.remove(x) != success) cout << "Not found!\n";
      break;
   case 'f':
      cout << "Enter character to look for:";
      x = get_char();
      if (test_tree.tree_search(x) != success)
          cout << "  The entry is not present";
      else
          cout << "  The entry is present";
      cout << endl;
      break;

   case 'h':
      cout << "The height of the tree is " << test_tree.height() << "\n";
      break;

   case 'e':
      test_tree.clear();
      cout << "Tree is cleared.\n";
      break;

   case 'w':
      cout << "The width of the tree is "
           << test_tree.width() << "\n";
      break;

   case 'c':
      test_tree.bracketed_print();
      break;

   case 'l':
      cout << "The leaf count of the tree is "
           << test_tree.leaf_count() << "\n";
      break;

   case 'b':
      test_tree.level_traverse(write_ent);
      cout << endl;
      break;

   case 'p':
      test_tree.preorder(write_ent);
      cout << endl;
      break;

   case 'o':
      test_tree.postorder(write_ent);
      cout << endl;
      break;

   case 'n':
      test_tree.inorder(write_ent);
      cout << endl;
      break;

   case 'q':
      cout << "Tree demonstration finished.\n";
      return 0;
   }
   return 1;
}
```

```
int main()
/* PRE:  None.
   POST: A binary tree demonstration has been performed.
   USES: get_command, do_command, Tree methods
*/

{
   Binary_tree<char> s; Binary_tree<char> t = s;

   Search_tree<char> test_tree;
   cout << "Menu driven demo program for a binary search tree of "
        << "char entries."
        << endl << endl;

   while (do_command(get_command(), test_tree));
}
```

P3. *Write a function for treesort that can be added to Project P1 of Section 8.2 (page 328). Determine whether it is necessary for the list structure to be contiguous or linked. Compare the results with those for the other sorting methods in Chapter 8.*

treesort

Answer This method of sorting does not require the initial list to be in either a linked or contiguous implementation. It however requires another structure in which to store and sort the entries of the list. Use either of the sorting demonstration programs developed in Project P1 of Section 8.2 (page 328) and include the additional declarations and functions needed for a binary search tree as developed in this section. The method is to remove all of the entries from the list and insert them into the tree. To insert all the entries in order into the list, an inorder traversal is all that is necessary.

P4. *Write a function for searching, using a binary search tree with* ***sentinel*** *as follows: Introduce a new sentinel node, and keep a pointer called* sentinel *to it. See Figure 10.11. Replace all the* NULL *links within the binary search tree with* sentinel *links (links to the sentinel). Then, for each search, store the target key into the sentinel node before starting the search. Delete the test for an unsuccessful search from* tree_search*, since it cannot now occur. Instead, a search that now finds the sentinel is actually an unsuccessful search. Run this function on the test data of the preceding project to compare the performance of this version with the original function* tree_search.

sentinel search

Answer The directory contains a package for binary search trees with a sentinel for Project P4 and a menu-driven demonstration program.

The files in the directory are:

Binary node class:

```
enum Balance_factor { left_higher, equal_height, right_higher };

template <class Entry>
struct Binary_node {

//    data members:
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
//    constructors:
   Binary_node();
   Binary_node(const Entry &x);

//    virtual methods:
   virtual void set_balance(Balance_factor b);
   virtual Balance_factor get_balance() const;

};
```

Implementation:

```
template <class Entry>
Binary_node<Entry>::Binary_node()
{
   left = right = NULL;
}

template <class Entry>
Binary_node<Entry>::Binary_node(const Entry &x)
{
   data = x;
   left = right = NULL;
}

template <class Entry>
void Binary_node<Entry>::set_balance(Balance_factor b)
{
}

template <class Entry>
Balance_factor Binary_node<Entry>::get_balance() const
{
   return equal_height;
}

template <class Entry>
void Binary_tree<Entry>::recursive_inorder(Binary_node<Entry> *sub_root,
                                           void (*visit)(Entry &))
/*
Pre:  sub_root either points to the sentinel,
      or points to a subtree of the Binary_tree.
Post: The subtree has been been traversed in inorder sequence.
Uses: The function recursive_inorder recursively
*/

{
   if (sub_root != sentinel) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}

template <class Entry>
void Binary_tree<Entry>::recursive_preorder(Binary_node<Entry> *sub_root,
                                            void (*visit)(Entry &))
/*
Pre:  sub_root either points to the sentinel,
      or points to a subtree of the Binary_tree.
Post: The subtree has been been traversed in preorder sequence.
Uses: The function recursive_preorder recursively
*/

{
   if (sub_root != sentinel) {
      (*visit)(sub_root->data);
      recursive_preorder(sub_root->left, visit);
      recursive_preorder(sub_root->right, visit);
   }
}
```

```
template <class Entry>
void Binary_tree<Entry>::recursive_postorder(Binary_node<Entry> *sub_root,
                                             void (*visit)(Entry &))
/*
Pre:  sub_root either points to the sentinel,
      or points to a subtree of the Binary_tree.
Post: The subtree has been been traversed in postorder sequence.
Uses: The function recursive_postorder recursively
*/

{
   if (sub_root != sentinel) {
      recursive_postorder(sub_root->left, visit);
      recursive_postorder(sub_root->right, visit);
      (*visit)(sub_root->data);
   }
}

template <class Entry>
void Binary_tree<Entry>::recursive_insert(Binary_node<Entry> *&sub_root,
                                          const Entry &x)
/*
Pre:  sub_root either points to the sentinel,
      or points to a subtree of the Binary_tree.
Post: The Entry  has been inserted into the subtree in such a way
      that the properties of a binary search tree have been preserved.
Uses: The functions recursive_insert, recursive_height
*/

{
   if (sub_root == sentinel) {
      sub_root = new Binary_node<Entry>(x);
      sub_root->left = sub_root->right = sentinel;
   }
   else
     if (recursive_height(sub_root->right) <
         recursive_height(sub_root->left))
        recursive_insert(sub_root->right, x);
   else
      recursive_insert(sub_root->left, x);
}

template <class Entry>
int Binary_tree<Entry>::recursive_size(Binary_node<Entry> *sub_root) const
/*
Post: The number of entries in the subtree rooted at
      sub_root is returned.
*/

{
   if (sub_root == sentinel) return 0;
   return 1 + recursive_size(sub_root->left) +
              recursive_size(sub_root->right);
}
```

```
template <class Entry>
int Binary_tree<Entry>::
           recursive_height(Binary_node<Entry> *sub_root) const
/*
Post: The height of the subtree rooted at
      sub_root is returned.
*/

{
   if (sub_root == sentinel) return 0;
   int l = recursive_height(sub_root->left);
   int r = recursive_height(sub_root->right);
   if (l > r) return 1 + l;
   else return 1 + r;
}

template <class Entry>
void Binary_tree<Entry>::recursive_clear(Binary_node<Entry> *&sub_root)
/*
Post: The subtree rooted at
      sub_root is cleared.
*/

{
   Binary_node<Entry> *temp = sub_root;
   if (sub_root == sentinel) return;
   recursive_clear(sub_root->left);
   recursive_clear(sub_root->right);
   sub_root = sentinel;
   delete temp;
}

template <class Entry>
Binary_node<Entry> *Binary_tree<Entry>::recursive_copy(
     Binary_node<Entry> *sub_root, Binary_node<Entry> *old_s,
                                   Binary_node<Entry> *new_s)
/*
Post: The subtree rooted at
      sub_root is copied, and a pointer to
      the root of the new copy is returned.
*/

{
   if (sub_root == old_s) return new_s;
   Binary_node<Entry> *temp = new Binary_node<Entry>(sub_root->data);
   temp->left = recursive_copy(sub_root->left, old_s, new_s);
   temp->right = recursive_copy(sub_root->right, old_s, new_s);
   return temp;
}

template <class Entry>
void Binary_tree<Entry>::recursive_swap(Binary_node<Entry> *sub_root)
/*
Post: In the subtree rooted at
      sub_root all pairs of left and right
      links are swapped.
*/
```

```
{
   if (sub_root == sentinel) return;
   Binary_node<Entry> *temp = sub_root->left;
   sub_root->left = sub_root->right;
   sub_root->right = temp;
   recursive_swap(sub_root->left);
   recursive_swap(sub_root->right);
}

template <class Entry>
Binary_node<Entry> *&Binary_tree<Entry>::find_node(
                     Binary_node<Entry> *&sub_root, const Entry &x) const
/*
Post: If the subtree rooted at
      sub_root contains x as a entry, a pointer
      to the subtree node storing x is returned.
      Otherwise the pointer sentinel is returned.
*/
{
   if (sub_root == sentinel || sub_root->data == x) return sub_root;
   else {
      Binary_node<Entry> *&temp = find_node(sub_root->left, x);
      if (temp != sentinel) return temp;
      else return find_node(sub_root->right, x);
   }
}

template <class Entry>
Error_code Binary_tree<Entry>::remove_root(Binary_node<Entry> *&sub_root)
/*
Pre:  sub_root either points to the sentinel,
      or points to a subtree of the Binary_tree.
Post: If sub_root is the sentinel, a code of not_present is returned.
      Otherwise the root of the subtree is removed in such a way
      that the properties of a binary search tree are preserved.
      The parameter sub_root is reset as
      the root of the modified subtree and
      success is returned.
*/
{
   if (sub_root == sentinel) return not_present;
   Binary_node<Entry> *to_delete = sub_root;
                                 //  Remember node to delete at end.
   if (sub_root->right == sentinel) sub_root = sub_root->left;
   else if (sub_root->left == sentinel) sub_root = sub_root->right;

   else {                        //  Neither subtree is empty
      to_delete = sub_root->left;   //  Move left to find predecessor
      Binary_node<Entry> *parent = sub_root; //  parent of to_delete
      while (to_delete->right != sentinel) {
                                   // to_delete is not the predecessor
         parent = to_delete;
         to_delete = to_delete->right;
      }
      sub_root->data = to_delete->data;  //  Move from to_delete to root
      if (parent == sub_root) sub_root->left = to_delete->left;
      else parent->right = to_delete->left;
   }
   delete to_delete;   //  Remove to_delete from tree
   return success;
}
```

Binary tree class:

```
template <class Entry>
class Binary_tree {
public:

//   Add methods here.

   void double_traverse(void (*visit)(Entry &));
   void bracketed_print();
   void interchange();
   void level_traverse(void (*visit)(Entry &));
   int width();
   int leaf_count() const;

   Binary_tree();
   bool empty() const;
   void preorder(void (*visit)(Entry &));
   void inorder(void (*visit)(Entry &));
   void postorder(void (*visit)(Entry &));

   int size() const;
   void clear();
   int height() const;
   void insert(const Entry &);

   Binary_tree (const Binary_tree<Entry> &original);
   Binary_tree & operator =(const Binary_tree<Entry> &original);
   ~Binary_tree();

   void recursive_double_traverse
       (Binary_node<Entry> *sub_root, void (*visit)(Entry &));

   void recursive_bracketed
       (Binary_node<Entry> *sub_root);

   void recursive_interchange
       (Binary_node<Entry> *sub_root);

   int recursive_leaf_count
                  (Binary_node<Entry> *sub_root) const;
   Error_code remove(Entry &);
   void swap();
   void recursive_inorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_preorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_postorder(Binary_node<Entry> *, void (*visit)(Entry &));
   void recursive_insert(Binary_node<Entry> *&sub_root, const Entry &x);
   int recursive_size(Binary_node<Entry> *sub_root) const;
   int recursive_height(Binary_node<Entry> *sub_root) const;
   void recursive_clear(Binary_node<Entry> *&sub_root);
   Binary_node<Entry> *recursive_copy(Binary_node<Entry> *sub_root,
     Binary_node<Entry> *old_s, Binary_node<Entry> *new_s);
   void recursive_swap(Binary_node<Entry> *sub_root);
   Binary_node<Entry> *&find_node(Binary_node<Entry> *&,
                               const Entry &) const;
   Error_code remove_root(Binary_node<Entry> *&sub_root);
protected:
   //   Add auxiliary function prototypes here.
   Binary_node<Entry> *root;
   Binary_node<Entry> *sentinel;
};
```

Implementation:

```
template <class Entry>
Binary_tree<Entry>::Binary_tree()
/*
Post: An empty binary tree has been created.
*/

{
   sentinel = new Binary_node<Entry>;
   sentinel->left = sentinel->right = sentinel;
   root = sentinel;
}

template <class Entry>
bool Binary_tree<Entry>::empty() const
/*
Post: A result of true is returned if the binary tree is empty.
      Otherwise, false is returned.
*/

{
   return root == sentinel;
}

template <class Entry>
void Binary_tree<Entry>::inorder(void (*visit)(Entry &))
/*
Post: The tree has been been traversed in infix
      order sequence.
Uses: The function recursive_inorder
*/

{
   recursive_inorder(root, visit);
}

template <class Entry>
Error_code Binary_tree<Entry>::remove(Entry &x)
/*
Post: An entry of the binary tree with Key matching x is
      removed.  If there is no such entry a code of not_present
      is returned.
*/

{
   Binary_node<Entry> *&found = find_node(root, x);
   return remove_root(found);
}

template <class Entry>
void Binary_tree<Entry>::insert(const Entry &x)
/*
Post: The Entry x is added to the binary tree.
*/
```

```
{
   recursive_insert(root, x);
}

template <class Entry>
void Binary_tree<Entry>::preorder(void (*visit)(Entry &))
/*
Post: The binary tree is traversed in prefix order,
      applying the operation *visit at each entry.
*/

{
   recursive_preorder(root, visit);
}

template <class Entry>
void Binary_tree<Entry>::postorder(void (*visit)(Entry &))
/*
Post: The binary tree is traversed in postfix order,
      applying the operation *visit at each entry.
*/

{
   recursive_postorder(root, visit);
}

template <class Entry>
int Binary_tree<Entry>::size() const
/*
Post: The number of entries in the binary tree is returned.
*/

{
   return recursive_size(root);
}

template <class Entry>
int Binary_tree<Entry>::height() const
/*
Post: The height of the binary tree is returned.
*/

{
   return recursive_height(root);
}

template <class Entry>
void Binary_tree<Entry>::clear()
/*
Post: All entries of the binary tree are removed.
*/

{
   recursive_clear(root);
}

template <class Entry>
Binary_tree<Entry>::~Binary_tree()
/*
Post: All entries of the binary tree are removed.
      All dynamically allocated memory in the structure
      is deleted.
*/
```

```
{
   clear();
   delete sentinel;
}

template <class Entry>
Binary_tree<Entry>::Binary_tree (const Binary_tree<Entry> &original)
/*
Post: A new binary tree is initialized to copy original.
*/
{
   sentinel = new Binary_node<Entry>;
   sentinel->left = sentinel->right = sentinel;
   root = recursive_copy(original.root, original.sentinel, sentinel);
}

template <class Entry>
Binary_tree<Entry> &Binary_tree<Entry>::operator =(const
                                        Binary_tree<Entry> &original)
/*
Post: The binary tree is reset to copy original.
*/
{
   Binary_tree<Entry> new_copy(original);
   clear();
   Binary_node<Entry> *temp = sentinel;
   sentinel = new_copy.sentinel;
   root = new_copy.root;
   new_copy.root = temp;
   new_copy.sentinel= temp;
   return *this;
}

template <class Entry>
void Binary_tree<Entry>::swap()
/*
Post: All left and right subtrees are switched
      in the binary tree.
*/
{
   recursive_swap(root);
}

template <class Entry>
void Binary_tree<Entry>::recursive_double_traverse
       (Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/*
Post: The subtree rooted at sub_root is
      doubly traversed.
*/
{
   if (sub_root != sentinel) {
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_double_traverse(sub_root->right, visit);
   }
}
```

```
template <class Entry>
void Binary_tree<Entry>::double_traverse(void (*visit)(Entry &))
/*
Post: The tree is doubly traversed.
*/

{
   recursive_double_traverse(root, visit);
}

template <class Entry>
void Binary_tree<Entry>::recursive_bracketed
       (Binary_node<Entry> *sub_root)
/*
Post: The subtree rooted at sub_root is
      printed with brackets.
*/

{
   if (sub_root != sentinel) {
      if (sub_root->left == sentinel && sub_root->right == sentinel)
         cout << sub_root->data;
      else {
         cout << "(" << sub_root->data << ":";
         recursive_bracketed(sub_root->left);
         cout << ",";
         recursive_bracketed(sub_root->right);
         cout << ")";
      }
   }
}

template <class Entry>
void Binary_tree<Entry>::bracketed_print()
/*
Post: The tree is printed with brackets.
*/

{
   recursive_bracketed(root);
   cout << endl;
}

template <class Entry>
void Binary_tree<Entry>::recursive_interchange
       (Binary_node<Entry> *sub_root)
/*
Post: In the subtree rooted at
      sub_root all pairs of left and right
      links are swapped.
*/

{
   if (sub_root != sentinel) {
      Binary_node<Entry> *tmp = sub_root->left;
      sub_root->left = sub_root->right;
      sub_root->right = tmp;
      recursive_interchange(sub_root->left);
      recursive_interchange(sub_root->right);
   }
}
```

```
template <class Entry>
void Binary_tree<Entry>::interchange()
/*
Post: In the tree all pairs of left and right
      links are swapped round.
*/

{
   recursive_interchange(root);
}

template <class Entry>
void Binary_tree<Entry>::level_traverse(void (*visit)(Entry &))
/*
Post: The tree is traversed level by level,
      starting from the top.  The operation
      *visit is applied to all entries.
*/

{
   Binary_node<Entry> *sub_root;
   if (root != sentinel) {
      Queue<Binary_node<Entry> *> waiting_nodes;
      waiting_nodes.append(root);

      do {
         waiting_nodes.retrieve(sub_root);
         (*visit)(sub_root->data);

         if (sub_root->left != sentinel)
            waiting_nodes.append(sub_root->left);

         if (sub_root->right != sentinel)
            waiting_nodes.append(sub_root->right);
         waiting_nodes.serve();
      } while (!waiting_nodes.empty());
   }
}

template <class Entry>
int Binary_tree<Entry>::width()
/*
Post: The width of the tree is returned.
*/

{
   if (root == sentinel) return 0;
   Extended_queue<Binary_node<Entry> *> current_level;
   current_level.append(root);
   int max_level = 0;

   while (current_level.size() != 0) {
      if (current_level.size() > max_level)
         max_level = current_level.size();

      Extended_queue<Binary_node<Entry> *> next_level;
      do {
         Binary_node<Entry> *sub_root;
         current_level.retrieve(sub_root);

         if (sub_root->left != sentinel)
            next_level.append(sub_root->left);
         if (sub_root->right != sentinel)
            next_level.append(sub_root->right);
         current_level.serve();
      } while (!current_level.empty());
      current_level = next_level;
   }
   return max_level;
}
```

```
template <class Entry>
int Binary_tree<Entry>::recursive_leaf_count
                  (Binary_node<Entry> *sub_root) const
/*
Post: The number of leaves in the subtree
      rooted at sub_root is returned.
*/
{
   if (sub_root == sentinel) return 0;
   if (sub_root->left == sentinel && sub_root->right == sentinel) return 1;
   return recursive_leaf_count(sub_root->left)
             + recursive_leaf_count(sub_root->right);
}
```

```
template <class Entry>
int Binary_tree<Entry>::leaf_count() const
/*
Post: The number of leaves in the tree is returned.
*/
{
   return recursive_leaf_count(root);
}
```

Additional node functions for seach-tree nodes:

```
template <class Record>
Error_code Search_tree<Record>::search_and_insert(
           Binary_node<Record> *&sub_root, const Record &new_data)
{
   if (sub_root == sentinel) {
      sub_root = new Binary_node<Record>(new_data);
      sub_root->left = sub_root->right = sentinel;
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}
```

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
   Binary_node<Record>* sub_root, const Record &target) const
{
   if (sub_root->data == target) return sub_root;
   else if (sub_root->data < target)
      return search_for_node(sub_root->right, target);
   else return search_for_node(sub_root->left, target);
}
```

```
template <class Record>
Error_code Search_tree<Record>::search_and_destroy(
           Binary_node<Record>* &sub_root, const Record &target)
/*
Pre:  sub_root is either sentinel or points to a subtree
      of the Search_tree.
Post: If the key of target is not in the subtree, a
      code of not_present is returned.
      Otherwise, a code of success is returned and the subtree node
      containing target has been removed in such a way that the properties
      of a binary search tree have been preserved.
Uses: Functions search_and_destroy recursively and remove_root
*/

{
   if (sub_root == sentinel || sub_root->data == target)
      return remove_root(sub_root);
   else if (target < sub_root->data)
      return search_and_destroy(sub_root->left, target);
   else
      return search_and_destroy(sub_root->right, target);
}
```

A (derived) binary search tree class:

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
   Error_code tree_search(Record &target) const;

   Error_code search_and_insert(Binary_node<Record> *&, const Record &);
   Binary_node<Record>
             *search_for_node(Binary_node<Record>*, const Record&) const;
   Error_code search_and_destroy(Binary_node<Record> *&, const Record &);

private: //  Add auxiliary function prototypes here.
};
```

Implementation:

```
template <class Record>
Error_code Search_tree<Record>::insert(const Record &new_data)
{
   return search_and_insert(root, new_data);
}

template <class Record>
Error_code Search_tree<Record>::remove(const Record &target)
/*
Post:
If a Record with a key matching that of target
belongs to the Search_tree a code of
success is returned and the corresponding node
is removed from the tree.  Otherwise,
a code of not_present is returned.
Uses: Function search_and_destroy
*/
```

```
{
   return search_and_destroy(root, target);
}

template <class Record>
Error_code Search_tree<Record>::tree_search(Record &target) const
/*
Post:
If there is an entry in the tree whose key matches that in target,
the parameter target is replaced by the corresponding record from
the tree and a code of success is returned.  Otherwise
a code of not_present is returned.
Uses: function search_for_node
*/

{
   Error_code result = success;
   sentinel->data = target;
   Binary_node<Record> *found = search_for_node(root, target);
   if (found == sentinel)
      result = not_present;
   else
      target = found->data;
   return result;
}
```

A menu driven demonstration program:

```
#include "../../c/utility.h"

void help()
/* PRE:  None.
   POST: Instructions for the tree operations have been printed.
*/

{
     cout << "\n";
     cout << "\t[S]ize       [I]nsert    [D]elete \n"
             "\t[H]eight     [E]rase     [F]ind \n"
             "\t[W]idth      [C]ontents  [L]eafs     [B]readth first \n"
             "\t[P]reorder   P[O]storder I[N]order   [?]help \n" << endl;
}

#include <string.h>
char get_command()
/* PRE:  None.
   POST: A character command belonging to the set of legal commands for the
         tree demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command (? for help) and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if(strchr("clwbe?sidfponqh",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}

char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include auxiliary data structures

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/sortable.h"
#include "../../8/linklist/merge.cpp"
#include "queue.h"
#include "queue.cpp"
#include "extqueue.h"
#include "extqueue.cpp"

// include binary search tree data structure
#include "node.h"
#include "tree.h"
#include "node.cpp"
#include "tree.cpp"
#include "stree.h"
#include "snode.cpp"
#include "stree.cpp"

int do_command(char c, Search_tree<char> &test_tree)
/* PRE:  The tree has been created and command is a valid tree
         operation.
   POST: The command has been executed.
   USES: All the functions that perform tree operations.
*/

{
   char x;
   switch (c) {
   case '?': help();
     break;

   case 's':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'i':
      cout << "Enter new character to insert:";
      x = get_char();
      test_tree.insert(x);
      break;
```

```
case 'd':
   cout << "Enter character to remove:" << flush;
   x = get_char();
   if (test_tree.remove(x) != success) cout << "Not found!\n";
   break;

case 'f':
   cout << "Enter character to look for:";
   x = get_char();
   if (test_tree.tree_search(x) != success)
       cout << "  The entry is not present";
   else
       cout << "  The entry is present";
   cout << endl;
   break;

case 'h':
   cout << "The height of the tree is " << test_tree.height() << "\n";
   break;

case 'e':
   test_tree.clear();
   cout << "Tree is cleared.\n";
   break;

case 'w':
   cout << "The width of the tree is "
        << test_tree.width() << "\n";
   break;

case 'c':
   test_tree.bracketed_print();
   break;

case 'l':
   cout << "The leaf count of the tree is "
        << test_tree.leaf_count() << "\n";
   break;

case 'b':
   test_tree.level_traverse(write_ent);
   cout << endl;
   break;

case 'p':
   test_tree.preorder(write_ent);
   cout << endl;
   break;

case 'o':
   test_tree.postorder(write_ent);
   cout << endl;
   break;

case 'n':
   test_tree.inorder(write_ent);
   cout << endl;
   break;
   case 'q':
      cout << "Tree demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/* PRE:  None.
   POST: A binary tree demonstration has been performed.
   USES: get_command, do_command, Tree methods
*/

{
   Binary_tree<char> s; Binary_tree<char> t = s;

   Search_tree<char> test_tree;
   cout << "Menu driven demo program for a binary search tree of "
        << "char entries."
        << endl << endl;
   while (do_command(get_command(), test_tree));
}
```

P5. *Different authors tend to use different vocabularies and to use common words with differing frequencies. Given an essay or other text, it is interesting to find what distinct words are used and how many times each is used. The purpose of this project is to compare several different kinds of binary search trees useful for this information retrieval problem. The current, first part of the project is to produce a driver program and the information-retrieval package using ordinary binary search trees. Here is an outline of the main driver program:*

information retrieval program

1. *Create the data structure (binary search tree).*
2. *Ask the user for the name of a text file and open it to read.*
3. *Read the file, split it apart into individual words, and insert the words into the data structure. With each word will be kept a frequency count (how many times the word appears in the input), and when duplicate words are encountered, the frequency count will be increased. The same word will not be inserted twice in the tree.*
4. *Print the number of comparisons done and the CPU time used in part 3.*
5. *If the user wishes, print out all the words in the data structure, in alphabetical order, with their frequency counts.*
6. *Put everything in parts 2–5 into a* **do** ... **while** *loop that will run as many times as the user wishes. Thus the user can build the data structure with more than one file if desired. By reading the same file twice, the user can compare time for retrieval with the time for the original insertion.*

Here are further specifications for the driver program:

- *The input to the driver will be a file. The program will be executed with several different files; the name of the file to be used should be requested from the user while the program is running.*
- *A word is defined as a sequence of letters, together with apostrophes (') and hyphens (-), provided that the apostrophe or hyphen is both immediately preceded and followed by a letter. Uppercase and lowercase letters should be regarded as the same (by translating all letters into either uppercase or lowercase, as you prefer). A word is to be truncated to its first 20 characters (that is, only 20 characters are to be stored in the data structure) but words longer than 20 characters may appear in the text. Nonalphabetic characters (such as digits, blanks, punctuation marks, control characters) may appear in the text file. The appearance of any of these terminates a word, and the next word begins only when a letter appears.*
- *Be sure to write your driver so that it will not be changed at all when you change implementation of data structures later.*

TOC
Index
Help
345
346

347

Here are specifications for the functions to be implemented first with binary search trees.

void update(**const** String &word,
Search_tree<Record> &structure, **int** &num_comps);

postcondition: If word was not already present in structure, then word has been inserted into structure and its frequency count is 1. If word was already present in structure, then its frequency count has been increased by 1. The variable parameter num_comps is set to the number of comparisons of words done.

void print(**const** Search_tree<Record> &structure);

postcondition: All words in structure are printed at the terminal in alphabetical order together with their frequency counts.

void write_method();

postcondition: The function has written a short string identifying the abstract data type used for structure.

Answer Main program:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/doubly/list.h"
#include "../../6/doubly/list.cpp"
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "../bt/node.h"
#include "../bt/tree.h"
#include "../bt/node.cpp"
#include "../bt/tree.cpp"

#include "../bst/stree.h"
#include "../bst/snode.cpp"
#include "../bst/stree.cpp"

#include "key.h"
#include "key.cpp"
#include "record.h"
#include "record.cpp"

#include "auxil.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Name of an input file can be given as a command-line argument.
Post: The word storing project p6 has been performed.
*/

{
   write_method();
   Binary_tree<Record> t;
   Search_tree<Record> the_words;
   char infile[1000];
   char in_string[1000];
   char *in_word;
   Timer clock;
   do {
      int initial_comparisons = Key::comparisons;
      clock.reset();
      if (argc < 2) {
         cout << "What is the input data file: " << flush;
         cin >> infile;
      }
      else strcpy(infile, argv[1]);

      ifstream file_in(infile);   //  Declare and open the input stream.
      if (file_in == 0) {
         cout << "Can't open input file " << infile << endl;
         exit (1);
      }

      while (!file_in.eof()) {
        file_in >> in_string;
        int position = 0;
        bool end_string = false;
        while (!end_string) {
           in_word = extract_word(in_string, position, end_string);
           if (strlen(in_word) > 0) {
              int counter;
              String s(in_word);
              update(s, the_words, counter);
           }
        }
      }
      cout << "Elapsed time: " << clock.elapsed_time() << endl;
      cout << "Comparisons performed = "
           << Key::comparisons - initial_comparisons << endl;
      cout << "Do you want to print frequencies? ";
      if (user_says_yes()) print(the_words);
      cout << "Do you want to add another file of input? ";
   } while (user_says_yes());
}
```

Auxiliary functions to carry out the required operations:

```
void write_method()
/*
Post:  A short string identifying the abstract
       data type used for the structure is written.
*/

{
   cout << " Word frequency program:  Binary Search Tree implementation";
   cout << endl;
}
```

```
void update(const String &word, Search_tree<Record> &structure,
               int &num_comps)
/*
Post:  If word was not already present in structure, then
       word has been inserted into structure and its frequency
       count is 1.  If word was already present in structure,
       then its frequency count has been increased by 1.  The
       variable parameter num_comps is set to the number of
       comparisons of words done.
*/

{
  int initial_comparisons = Key::comparisons;
  Record r((Key) word);
  if (structure.tree_search(r) == not_present)
     structure.insert(r);
  else {
     structure.remove(r);
     r.increment();
     structure.insert(r);
  }
  num_comps = Key::comparisons - initial_comparisons;
}

void wr_entry(Record &a_word)
{
   String s = ((Key) a_word).the_key();
   cout << s.c_str();
   for (int i = strlen(s.c_str()); i < 12; i++) cout << " ";
   cout << " : " << a_word.frequency() << endl;
}

void print(Search_tree<Record> &structure)
/*
Post:  All words in structure are printed at the terminal
       in alphabetical order together with their frequency counts.
*/

{
     structure.inorder(wr_entry);
}

char *extract_word(char *in_string, int &examine, bool &over)
{
   int ok = 0;

   while (in_string[examine] != '\0') {
     if ('a' <= in_string[examine] && in_string[examine] <= 'z')
       in_string[ok++] = in_string[examine++];

     else if ('A' <= in_string[examine] && in_string[examine] <= 'Z')
       in_string[ok++] = in_string[examine++] - 'A' + 'a';

     else if (('\'' == in_string[examine] || in_string[examine] == '-')
            && (
            ('a' <= in_string[examine + 1] &&
            in_string[examine + 1] <= 'z') ||
            ('A' <= in_string[examine + 1] &&
            in_string[examine + 1] <= 'Z')))
       in_string[ok++] = in_string[examine++];
     else break;
   }
   in_string[ok] = '\0';
   if (in_string[examine] == '\0') over = true;
   else examine++;
   return in_string;
}
```

Methods for keys and records to include into a binary search tree for the project.

```
class Key {
   String key;
public:
   static int comparisons;
   Key (String x = "");
   String the_key() const;
};

bool operator ==(const Key &x,const Key &y);
bool operator >(const Key &x,const Key &y);
bool operator <(const Key &x,const Key &y);
bool operator >=(const Key &x,const Key &y);
bool operator <=(const Key &x,const Key &y);
bool operator !=(const Key &x,const Key &y);
```

Implementation:

```
int Key::comparisons = 0;

String Key::the_key() const
{
   return key;
}

Key::Key (String x)
{
   key = x;
}

bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}

bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}

bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}

bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}
```

```
bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}
```

Record definitions:

```
class Record {
public:
   Record();                      //  default constructor
   operator Key() const;          //  cast to Key

   Record (const Key &a_name);    //  conversion to Record
   void increment();
   int frequency();
private:
   Key name;
   int count;
};
```

Implementation:

```
Record::Record() : name()
{
   count = 1;
}

Record::operator Key() const
{
   return name;
}

int Record::frequency()
{
   return count;
}

void Record::increment()
{
   count++;
}

Record::Record (const Key &a_name)
{
   name = a_name;
   count = 1;
}
```

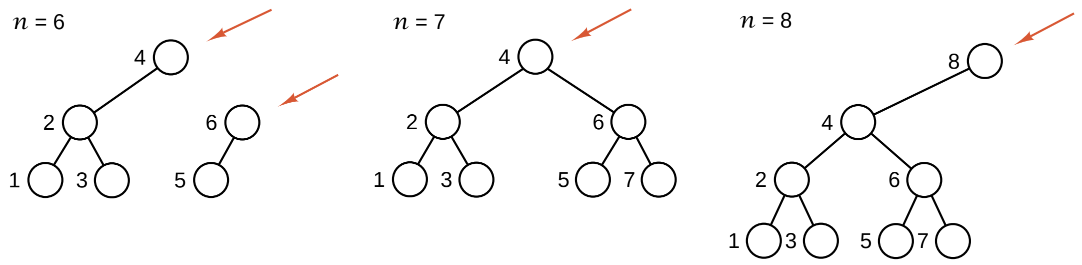
Two text files are included for testing: COWPER and RHYMES.

10.3 BUILDING A BINARY SEARCH TREE

Exercises 10.3

E1. *Draw the sequence of partial binary search trees (like Figure 10.13) that the method in this section will construct for the following values of n.* **(a)** *6, 7, 8;* **(b)** *15, 16, 17;* **(c)** *22, 23, 24;* **(d)** *31, 32, 33.*

Answer



E2. *Write function* build_tree *for the case when* supply *is a queue.*

Answer We assume that an implementation of a **template class** Queue with the standard Queue operations is available.

```
template <class Record>
Error_code Buildable_tree<Record> :: build_tree(const Queue<Record> &supply)
/* Post: If the entries of supply are in increasing order, a code of success is returned and the
         Search_tree is built out of these entries as a balanced tree. Otherwise, a code of fail
         is returned and a balanced tree is constructed from the longest increasing sequence of
         entries at the start of supply.
   Uses: The methods of class Queue and the functions build_insert, connect_subtrees, and
         find_root */
{
   Queue<Record> temporary(supply);
   Error_code ordered_data = success;        //  Set this to fail if keys do not increase.
   int count = 0;                            //  number of entries inserted so far
   Record x, last_x;
   List < Binary_node<Record> * > last_node;  //  pointers to last nodes on each level
   Binary_node<Record> *none = NULL;
   last_node.insert(0, none);                //  permanently NULL (for children of leaves)
   while (temporary.retrieve(x) == success) {
      temporary.serve( );
      if (count > 0 && x <= last_x) {
         ordered_data = fail;
         break;
      }
      build_insert(++count, x, last_node);
      last_x = x;
   }
   root = find_root(last_node);
   connect_trees(last_node);
   return ordered_data;                   //  Report any data-ordering problems back to client.
}
```

E3. *Write function* build_tree *for the case when the input structure is a binary search tree. [This version gives a function to rebalance a binary search tree.]*

Answer We first convert the input tree into a List and then apply the standard build_tree method from the text.

```
template <class Record>
Error_code Buildable_tree<Record> :: build_tree(const Search_tree<Record> &supply)
/* Post: If the entries of supply are in increasing order, a code of success is returned and the
         Buildable_tree is built out of these entries as a balanced tree. Otherwise, a code of fail
         is returned and a balanced tree is constructed from the longest increasing sequence of
         entries at the start of supply.
   Uses: The methods of class Search_tree and the functions build_insert, connect_subtrees, and
         find_root */
{
   List<Record> resupply;
   supply.tree_list(resupply);
   return build_tree(resupply);
}
```

An auxiliary recursive method to turn a binary search tree into an ordered list is coded as follows.

```
template <class Record>
void Search_tree<Record> :: recursive_tree_list
      (List<Record> &listed, Binary_node<Record> *sub_root) const
{
   if (sub_root != NULL) {
      recursive_tree_list(listed, sub_root->left);
      listed.insert(listed.size( ), sub_root->data);
      recursive_tree_list(listed, sub_root->right);
   }
}

template <class Record>
void Search_tree<Record> :: tree_list(List<Record> &listed) const
{
   recursive_tree_list(listed, root);
}
```

E4. *Write a version of function* build_tree *that will read keys from a file, one key per line. [This version gives a function that reads a binary search tree from an ordered file.]*

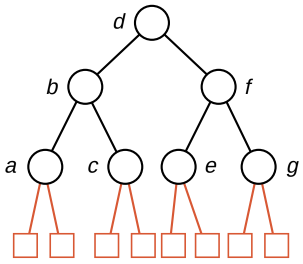
Answer We assume that the file has been opened as a readable **ifstream** object and that entries can be read from the stream with the input operator ≫ . (If necessary, a client would overload the operator ≫ for an instantiation of the **template** parameter **class** Record.) The implementation follows.

```
template <class Record>
Error_code Buildable_tree<Record> :: build_tree(ifstream &supply)
/* Post: If the entries of supply are in increasing order, a code of success is returned and the
         Search_tree is built out of these entries as a balanced tree. Otherwise, a code of fail
         is returned and a balanced tree is constructed from the longest increasing sequence of
         entries at the start of supply.
   Uses: The functions build_insert, connect_subtrees, and find_root */
{  Error_code ordered_data = success;         //  Set this to fail if keys do not increase.
   int count = 0;                             //  number of entries inserted so far
   Record x, last_x;
   List < Binary_node<Record> * > last_node;  //  pointers to last nodes on each level
   Binary_node<Record> *none = NULL;
   last_node.insert(0, none);                 //  permanently NULL (for children of leaves)
```

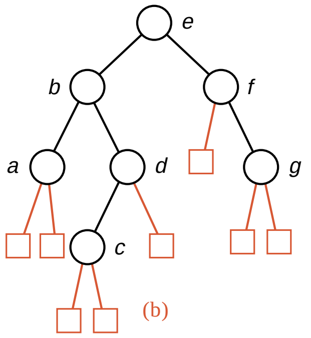
```
    while (1) {
       supply ≫ x;
       if (count > 0 && x <= last_x) {ordered_data = fail; break; }
       build_insert(++count, x, last_node);
       last_x = x;
    }
    root = find_root(last_node);
    connect_trees(last_node);
    return ordered_data;                    //   Report any data-ordering problems back to client.
}
```

E5. *Extend each of the binary search trees shown in Figure 10.8 into a 2-tree.*

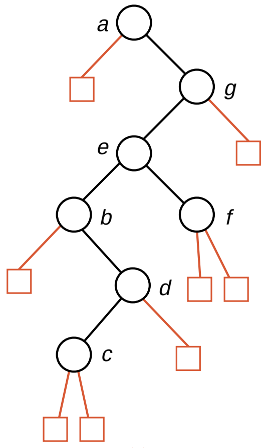
Answer



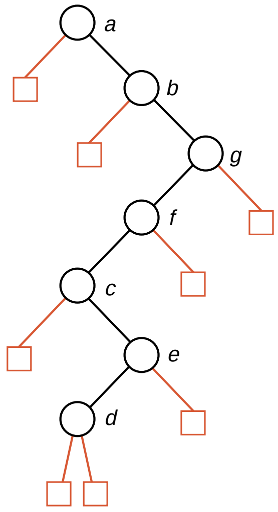
(a)



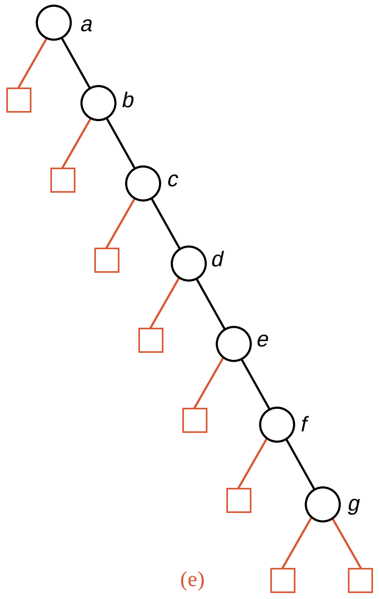
(b)



(c)



(d)



(e)

E6. *There are* 6 = 3! *possible ordered sequences of the three keys 1, 2, 3, but only 5 distinct binary trees with three nodes. Therefore, these binary trees are not equally likely to occur as search trees. Find which one of the five binary search trees corresponds to each of the six possible ordered sequences of 1, 2, 3. Thereby find the probability for building each of the binary search trees from randomly ordered input.*

Answer Refer to Figure 10.2, the binary trees with three nodes, in the text. The following ordered sequences of keys will result in a binary search tree of one the listed shapes.

TOC
Index
Help

Order of Keys	*Resulting tree*
3 2 1	*first*
3 1 2	*second*
2 1 3	*third*
2 3 1	*third*
1 2 3	*fourth*
1 3 2	*fifth*

The probability of getting each tree is $\frac{1}{6}$ except for the third tree, which is twice as likely, $\frac{1}{3}$ probability, to occur than the others, due to its symmetrical shape.

E7. *There are* $24 = 4!$ *possible ordered sequences of the four keys 1, 2, 3, 4, but only 14 distinct binary trees with four nodes. Therefore, these binary trees are not equally likely to occur as search trees. Find which one of the 14 binary search trees corresponds to each of the 24 possible ordered sequences of 1, 2, 3, 4. Thereby find the probability for building each of the binary search trees from randomly ordered input.*

Answer Of the 14 distinct binary trees, 8 are chains, each of which is obtained by only one input order (the preorder traversal of the tree). Hence each chain has probability $\frac{1}{24}$. There are 2 trees with a branch that is not at the root. The two nodes below this branch may be inserted in either order, so each such tree has probability $\frac{2}{24}$ or $\frac{1}{12}$. There are 4 trees with a branch at the root. Each of these has a one-node tree on one side and a two-node tree on the other side of the root. The node on a side by itself may be inserted at any of three times relative to the two nodes on the other side, so each of these trees has probability $\frac{3}{24}$ or $\frac{1}{8}$.

E8. *If* T *is an arbitrary binary search tree, let* $S(T)$ *denote the number of ordered sequences of the keys in* T *that correspond to* T *(that is, that will generate* T *if they are inserted, in the given order, into an initially-empty binary search tree). Find a formula for* $S(T)$ *that depends only on the sizes of* L *and* R *and on* $S(L)$ *and* $S(R)$, *where* L *and* R *are the left and right subtrees of the root of* T.

Answer If $T =$ is the empty binary search tree, then it is constructed in exactly one way, from the empty sequence, so $S() = 1$. Otherwise, let r be the key in the root of T, and let L and R be the left and right subtrees of the root. Take any sequence of keys that generates T. Its first entry must be r, since the root of T must be inserted before any other node. Remove r from the sequence and partition the remaining sequence into two subsequences, with all the keys from L in one subsequence and all the keys from R in the other. Then these two subsequences, separately, will generate L and R. On the other hand, we can obtain a sequence that generates T by starting with any sequences that generate L and R, merging these sequences together in any possible way (preserving their orders), and putting r at the start of the sequence. How many ways are there to merge two ordered sequences? If the two sequences have lengths n_l and n_r, then they will merge into a sequence of length $n_l + n_r$ in which n_l of the places are filled with entries from the first sequence (in the same order) and the remaining n_r places are filled with entries from the second sequence. Hence, the number of ways of merging the two sequences is the number of ways of choosing n_l positions out of $n_l + n_r$ positions, which is the binomial coefficient

$$\binom{n_l + n_r}{n_l}.$$

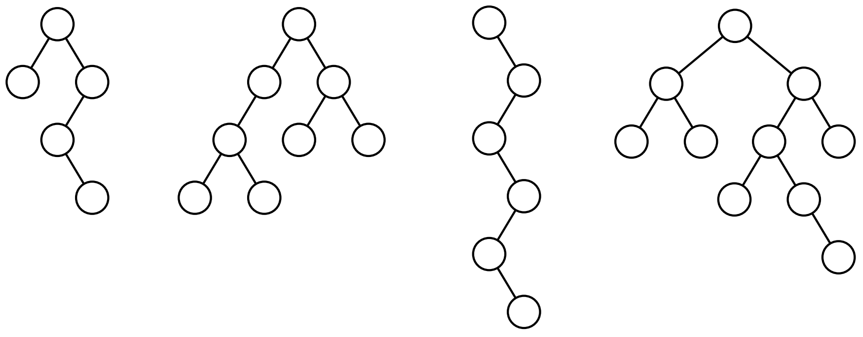
It follows that, if T is a nonempty binary search tree whose root has subtrees L and R of sizes n_l and n_r, then the number of ordered sequences of keys that will generate T is exactly

$$S(T) = \binom{n_l + n_r}{n_l} \cdot S(L) \cdot S(R).$$

10.4 HEIGHT BALANCE: AVL TREES

Exercises 10.4

E1. *Determine which of the following binary search trees are AVL trees. For those that are not, find all nodes at which the requirements are violated.*



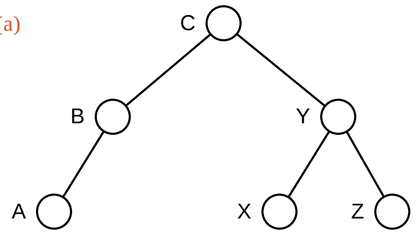
(a) (b) (c) (d)

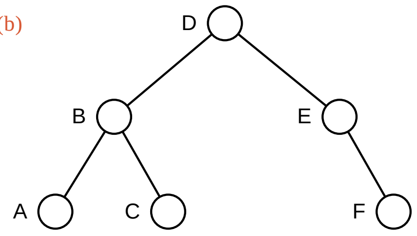
Answer **(a)** The tree is not an AVL tree because the right subtree is two levels taller than the left subtree.
(b) The left child of the root is too much unbalanced to the left.
(c) The top four nodes are all too unbalanced.
(d) The tree is not an AVL tree because the right subtree is two levels taller than the left subtree.

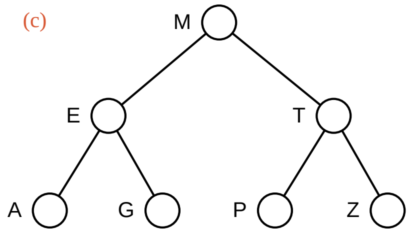
E2. *In each of the following, insert the keys, in the order shown, to build them into an AVL tree.*

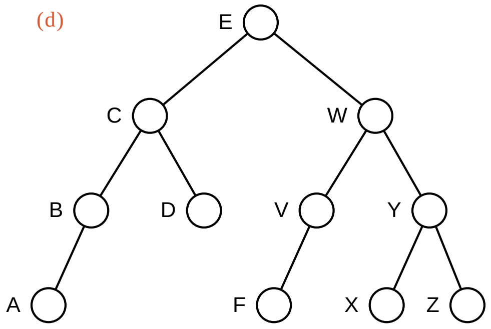
(a) A, Z, B, Y, C, X.
(b) A, B, C, D, E, F.
(c) M, T, E, A, Z, G, P.
(d) A, Z, B, Y, C, X, D, W, E, V, F.
(e) A, B, C, D, E, F, G, H, I, J, K, L.
(f) A, V, L, T, R, E, I, S, O, K.

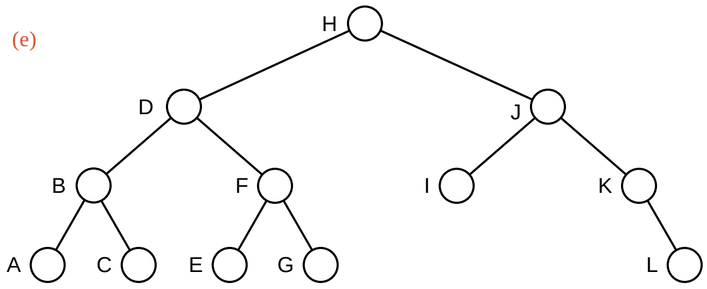
Answer

(a)


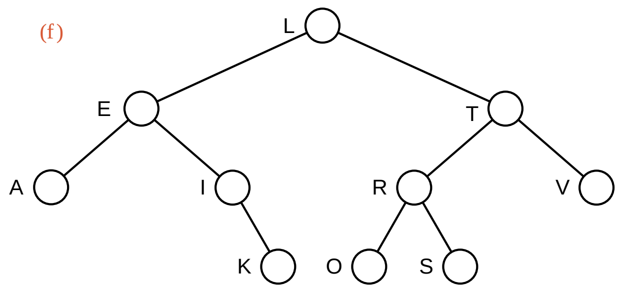
(b)


(c)


(d)


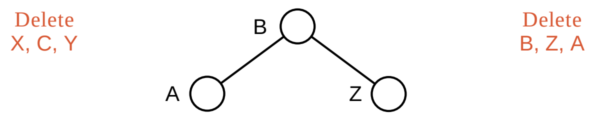
(e)


(f)

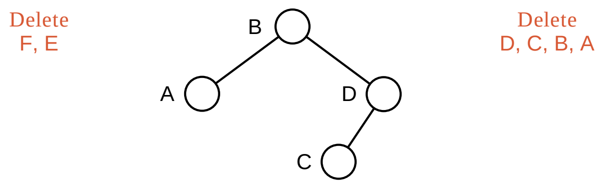


E3. *Delete each of the keys inserted in Exercise E2 from the AVL tree, in LIFO order (last key inserted is first removed).*

Answer **(a)**

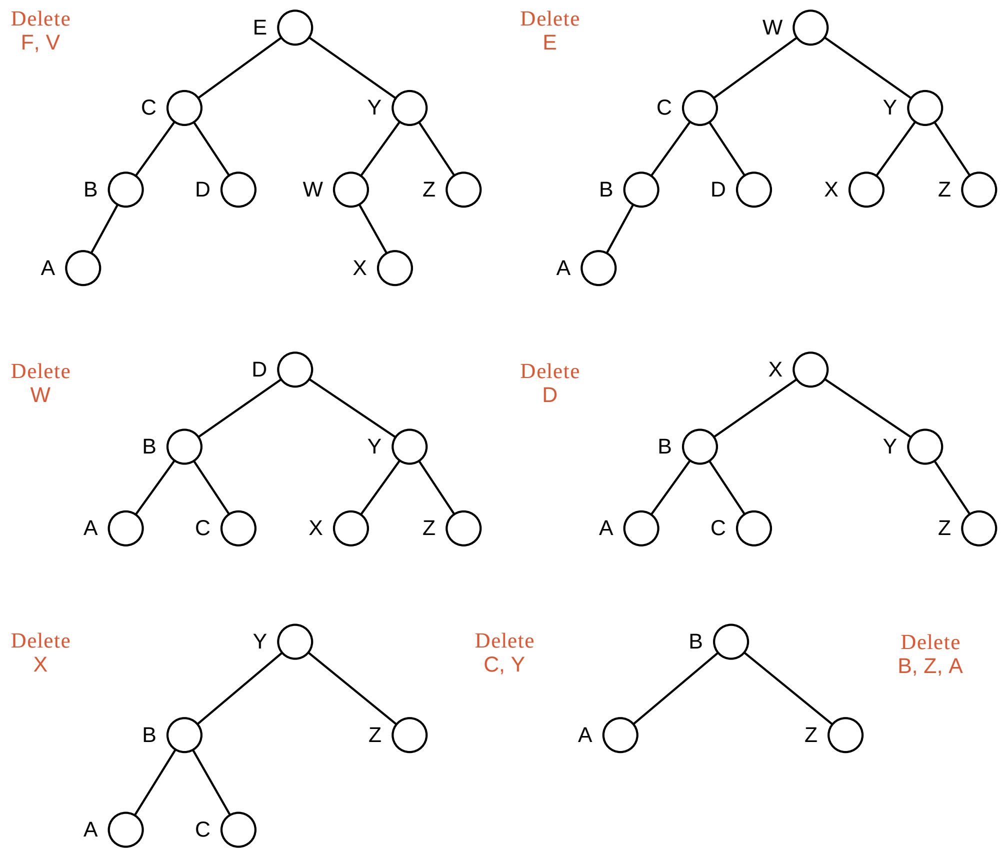


(b)



(c) No rotations are required in the process of deleting all nodes in the given order.

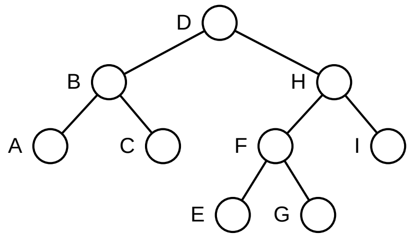
(d)



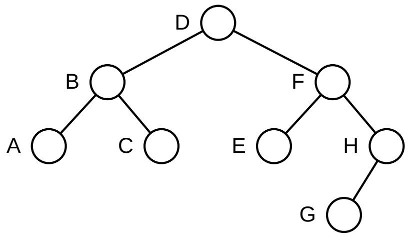
TOC
Index
Help

(e)

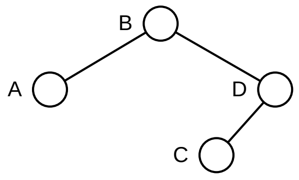
Delete
L, K, J



Delete
I



Delete
H, G, F, E,



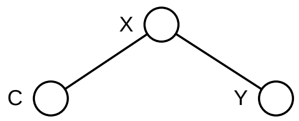
Delete
D, C, B, A

(f) No rotations are required in the process of deleting all nodes in the given order.

E4. *Delete each of the keys inserted in Exercise E2 from the AVL tree, in FIFO order (first key inserted is first removed).*

Answer **(a)**

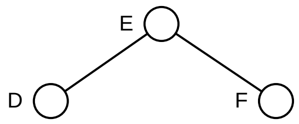
Delete
A, Z, B



Delete
Y, C, X

(b)

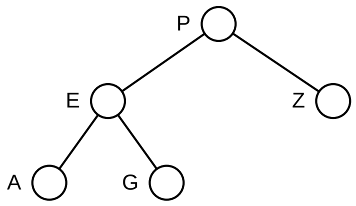
Delete
A, B, C



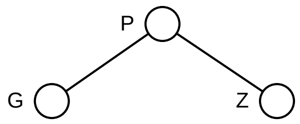
Delete
D, E, F

(c)

Delete
M, T



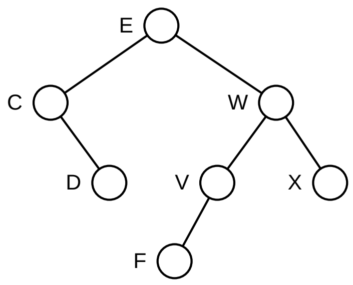
Delete
E, A



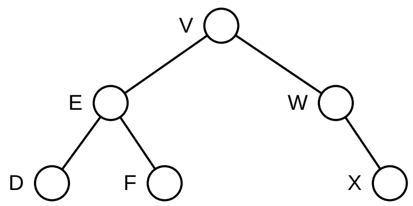
Delete
Z, G, P

(d)

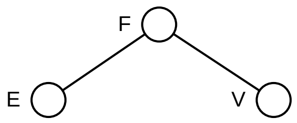
Delete
A, Z, B, Y



Delete
C

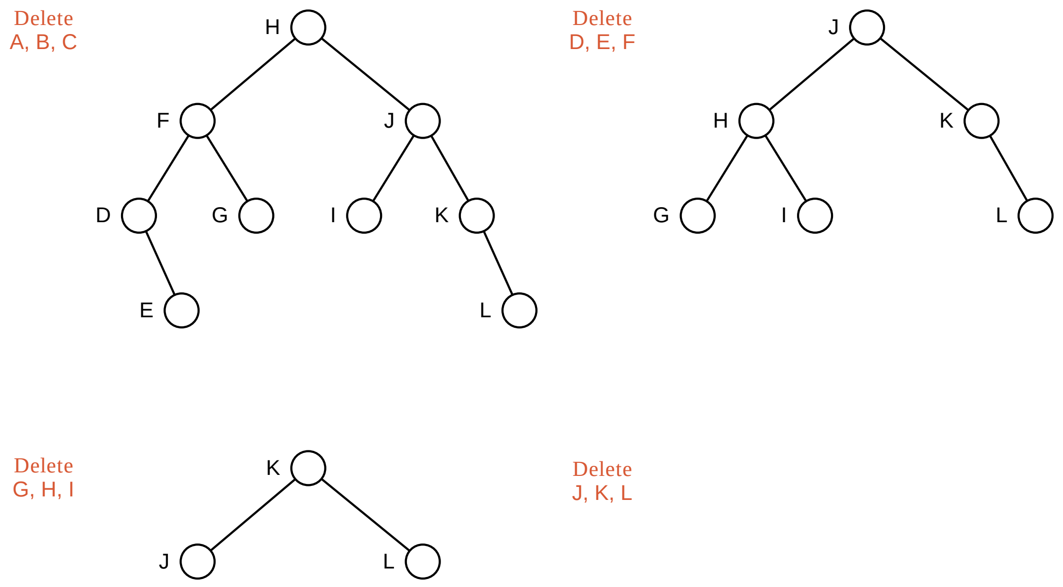


Delete
X, D, W

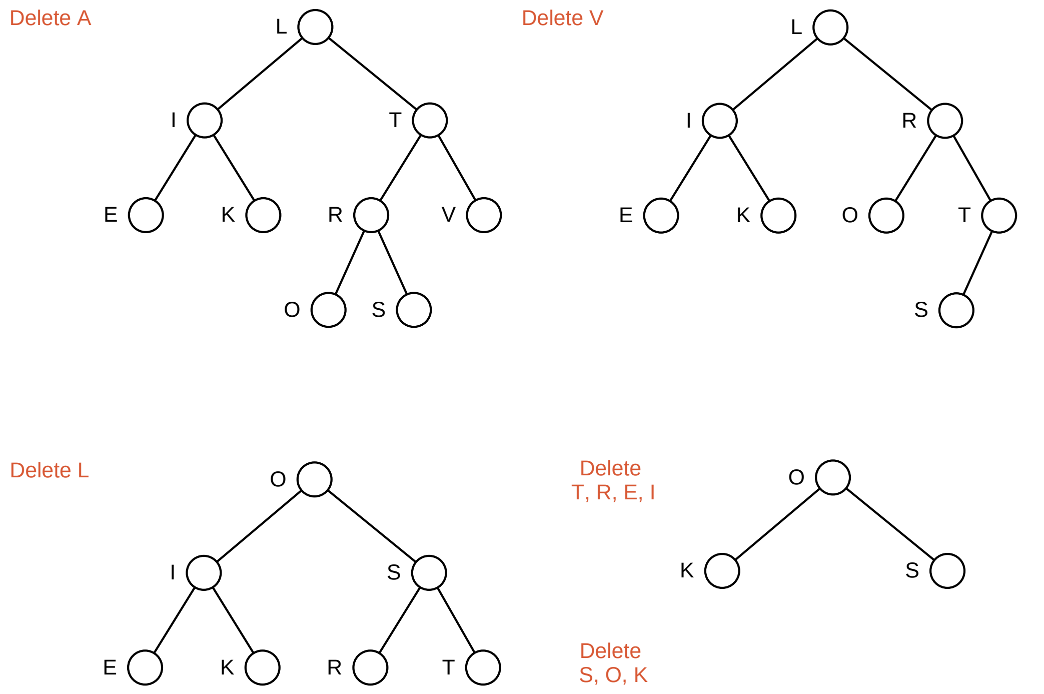


Delete
E, V, F

(e)



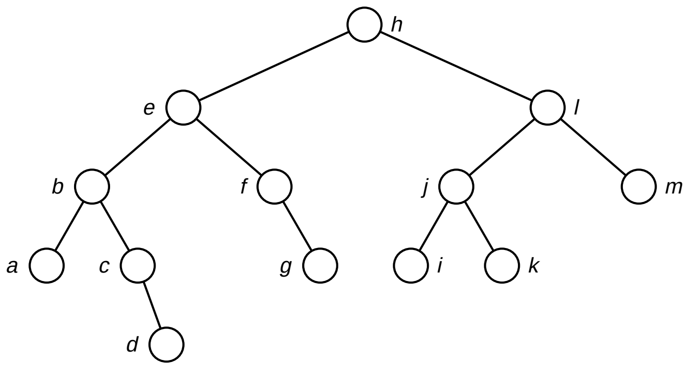
(f)



TOC
Index
Help

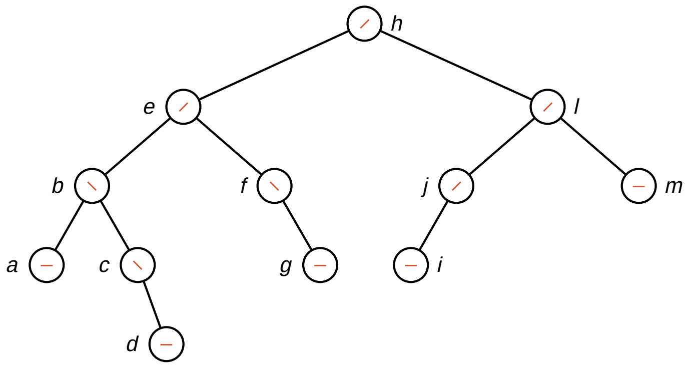
E5. *Start with the following AVL tree and remove each of the following keys. Do each removal independently, starting with the original tree each time.*

(a) k
(b) c
(c) j
(d) a
(e) g
(f) m
(g) h

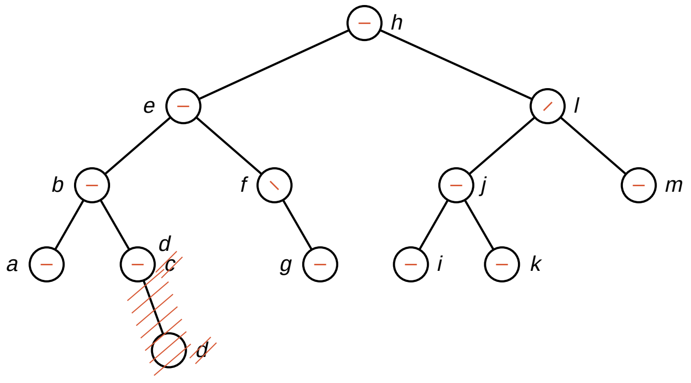

h
e
l
b
f
j
m
a
c
g
i
k
d

Answer

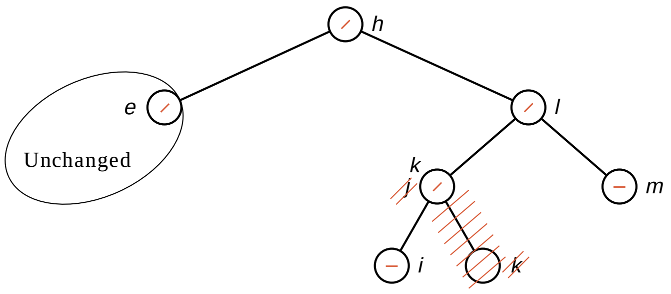
(a)


h
e
l
b
f
j
m
a
c
g
i
d

(b)

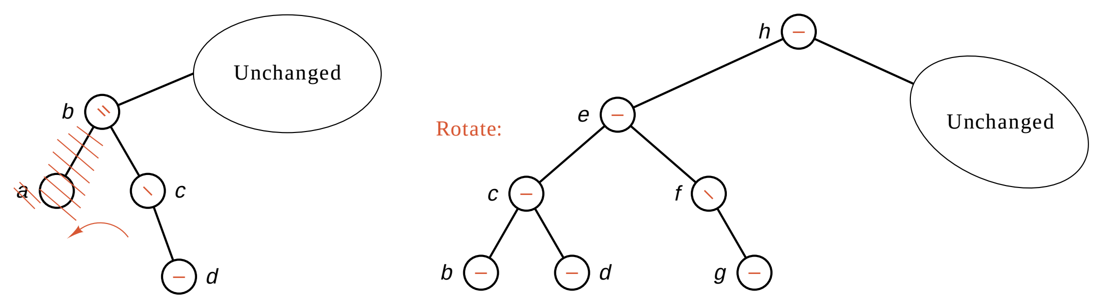

h
e
l
b
f
j
m
a
d
c
g
i
k
d

(c)

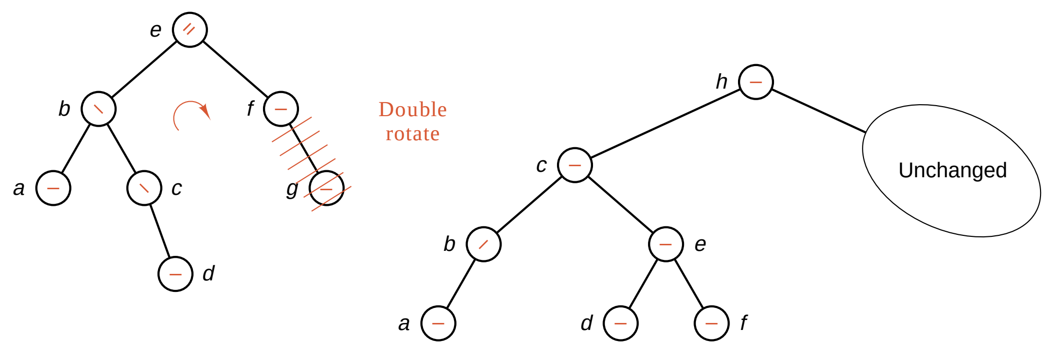

h
e
l
Unchanged
k
j
m
i
k

TOC
Index
Help

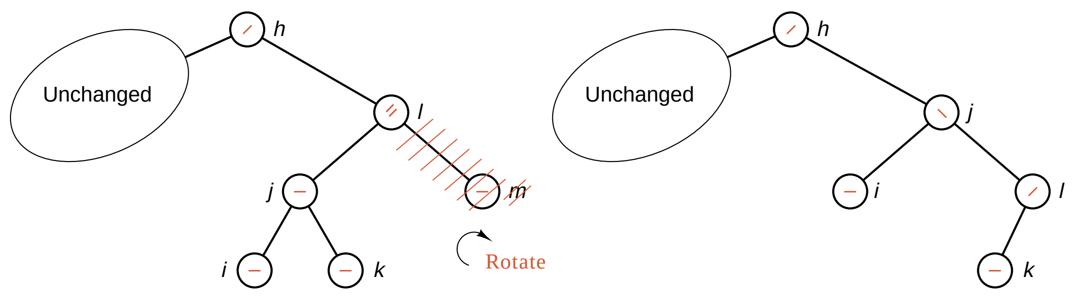
(d)



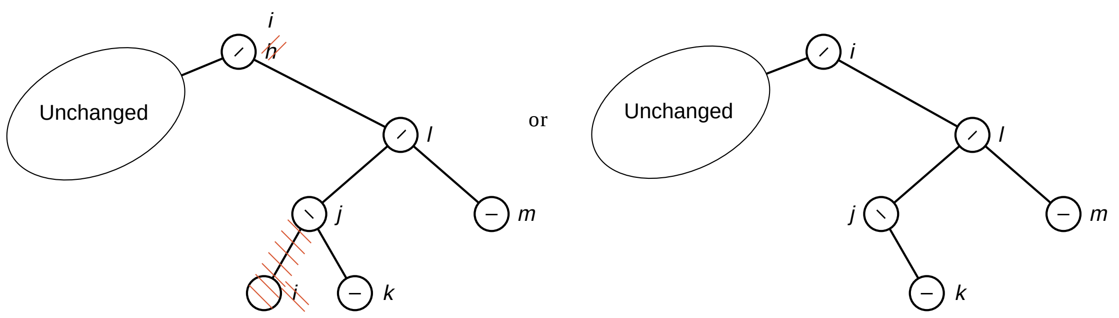
(e)



(f)



(g)



E6. *Write a method that returns the height of an AVL tree by tracing only one path to a leaf, not by investigating all the nodes in the tree.*

Answer

```
template <class Record>
int AVL_tree<Record> :: recursive_height(Binary_node<Record> *sub_root)
/* Post: Returns the height of the sub-AVLtree rooted at sub_root. */
```

```
{
   if (sub_root == NULL) return 0;
   if (sub_root->get_balance( ) == right_higher)
       return 1 + recursive_height(sub_root->right);
   return 1 + recursive_height(sub_root->left);
}

template <class Record>
int AVL_tree<Record> :: height( )
/* Post: Returns the height of an AVL tree. */
{
   return recursive_height(root);
}
```

E7. *Write a function that returns a pointer to the leftmost leaf closest to the root of a nonempty AVL tree.*

Answer The following implementation returns a pointer to a private node object from within a tree structure. It should therefore only be available as a private auxiliary member function of an AVL tree.

```
template <class Record>
Binary_node<Record> *AVL_tree<Record> :: recursive_near_leaf
                                      (Binary_node<Record> *sub_root, int &depth)
/* Post: Returns a pointer to the closest leaf of the subtree rooted at sub_root. If the subtree is
         empty NULL is returned. */
{
   depth = −1;
   if (sub_root == NULL) return NULL;
   int depth_l, depth_r;
   Binary_node<Record> *found_l, *found_r;
   found_l = recursive_near_leaf(sub_root->left, depth_l);
   found_r = recursive_near_leaf(sub_root->right, depth_r);
   if (depth_l == −1) {
      depth = depth_r + 1;
      if (depth_r == −1) return sub_root;
      return found_r;
   }
   if (depth_r == −1 || depth_l <= depth_r) {
      depth = depth_l + 1;
      return found_l;
   }
   depth = depth_r + 1;
   return found_r;
}

template <class Record>
Binary_node<Record> *AVL_tree<Record> :: near_leaf( )
/* Post: Returns a pointer to the leaf of the AVL tree that is closest to the root; if more than one
         such leaf exists, it points to the leftmost one. If the tree is empty NULL is returned. */
{
   int depth;
   return recursive_near_leaf(root, depth);
}
```

E8. *Prove that the number of (single or double) rotations done in deleting a key from an AVL tree cannot exceed half the height of the tree.*

Answer Let $x_0, x_1, \dots, x_n$ be the nodes encountered along the path from the removed node to the root of the tree, where x_0 is the removed node, x_1 is the parent of the removed node, and x_n is the root of the tree. Let T_i be the tree rooted at x_i, before the removal is made, where $i = 0, \dots, n$. For any tree T, let $\mathrm{ht}(T)$ denote the height of T. We then know that $\mathrm{ht}(T_i) \geq \mathrm{ht}(T_{i-1})+1$ for $i = 1, \dots, n$. If a single or double rotation occurs at x_i, then we see from Figure 9.20, Cases 3(a, b, c), that $\mathrm{ht}(T_i) = \mathrm{ht}(T_{i-1})+2$. Let R be the set of indices i from $\{1, \dots, n\}$ for which rotations occur at x_i, and let S be the remaining indices (for which no rotations occur). Let $|R|$ be the number of indices in R, so that the number in S is $|S| = n - |R|$. Since $\mathrm{ht}(T_0) \geq 0$, we have:

$$\begin{aligned}
\mathrm{ht}(T_n) &= \sum_{i=1}^{n}\Big(\mathrm{ht}(T_i)-\mathrm{ht}(T_{i-1})\Big) + \mathrm{ht}(T_0)\\
&= \sum_{R}\Big(\mathrm{ht}(T_i)-\mathrm{ht}(T_{i-1})\Big) + \sum_{S}\Big(\mathrm{ht}(T_i)-\mathrm{ht}(T_{i-1})\Big) + \mathrm{ht}(T_0)\\
&\geq 2\,|R| + |S| + 0 = 2\,|R| + n - |R| = |R| + n.
\end{aligned}$$

The number $|R|$ of nodes at which rotations occur surely cannot exceed the total number n of nodes on the path from x_1 to the root x_n, so $|R| \leq n$, and we therefore obtain $\mathrm{ht}(T_n) \geq |R| + n \geq 2\,|R|$, from which $|R| \leq \frac{1}{2}\mathrm{ht}(T_n)$ follows. Therefore, the removal of a node from an AVL tree requires a number of (single or double) rotations that is no more than half the height of the tree.

Programming Projects 10.4

P1. *Write a C++ method to remove a node from an AVL tree, following the steps outlined in the text.*

Answer

```
template <class Record>
Error_code AVL_tree<Record>::remove_avl(Binary_node<Record> *&sub_root,
           const Record &target, bool &shorter)

{
   Binary_node<Record> *temp;
   if (sub_root == NULL) return fail;
   else if (target < sub_root->data)
      return remove_avl_left(sub_root, target, shorter);
   else if (target > sub_root->data)
      return remove_avl_right(sub_root, target, shorter);

   else if (sub_root->left == NULL) { //  Found target: delete current node.
      temp = sub_root;               //  Move right subtree up to delete node.
      sub_root = sub_root->right;
      delete temp;
      shorter = true;
   }

   else if (sub_root->right == NULL) {
      temp = sub_root;               //  Move left subtree up to delete node.
      sub_root = sub_root->left;
      delete temp;
      shorter = true;
   }

   else if (sub_root->get_balance() == left_higher) {
                    //  Neither subtree is empty; delete from the taller.
      temp = sub_root->left;
              //  Find predecessor of target and delete it from left tree.
      while (temp->right != NULL) temp = temp->right;
      sub_root->data = temp->data;
      remove_avl_left(sub_root, temp->data, shorter);
   }
   else {               //  Find successor of target and delete from right.
      temp = sub_root->right;
      while (temp->left != NULL) temp = temp->left;
      sub_root->data = temp->data;
      remove_avl_right(sub_root, temp->data, shorter);
   }
   return success;
}

template <class Record>
Error_code AVL_tree<Record>::remove_avl_right(Binary_node<Record>
           *&sub_root, const Record &target, bool &shorter)

{
   Error_code result = remove_avl(sub_root->right, target, shorter);
   if (shorter == true) switch (sub_root->get_balance()) {
   case equal_height:                      //  case 1 in text
      sub_root->set_balance(left_higher);
      shorter = false;
      break;

   case right_higher:                      //  case 2 in text
      sub_root->set_balance(equal_height);
      break;

   case left_higher:
         //  case 3 in text: shortened shorter subtree; must rotate
      Binary_node<Record> *temp = sub_root->left;
      switch (temp->get_balance()) {
      case equal_height:       //  case 3a
         temp->set_balance(right_higher);
         rotate_right(sub_root);
         shorter = false;
         break;

      case left_higher:       //  case 3b
         sub_root->set_balance(equal_height);
         temp->set_balance(equal_height);
         rotate_right(sub_root);
         break;

      case right_higher:        //  case 3c; requires double rotation
         Binary_node<Record> *temp_right = temp->right;
         switch (temp_right->get_balance()) {
         case equal_height:
            sub_root->set_balance(equal_height);
            temp->set_balance(equal_height);
            break;

         case left_higher:
            sub_root->set_balance(right_higher);
            temp->set_balance(equal_height);
            break;
         case right_higher:
            sub_root->set_balance(equal_height);
            temp->set_balance(left_higher);
            break;
         }
         temp_right->set_balance(equal_height);
         rotate_left(sub_root->left);
         rotate_right(sub_root);
         break;
      }
   }
   return result;
}

template <class Record>
Error_code AVL_tree<Record>::remove_avl_left(Binary_node<Record>
           *&sub_root, const Record &target, bool &shorter)
{
   Error_code result = remove_avl(sub_root->left, target, shorter);
   if (shorter == true) switch (sub_root->get_balance()) {
   case equal_height:                         //  case 1 in text
      sub_root->set_balance(right_higher);
      shorter = false;
      break;

   case left_higher:                          //  case 2 in text
      sub_root->set_balance(equal_height);
      break;

   case right_higher:
           //  case 3 in text: shortened shorter subtree; must rotate
      Binary_node<Record> *temp = sub_root->right;
      switch (temp->get_balance()) {
      case equal_height:        //  case 3a
         temp->set_balance(left_higher);
         rotate_left(sub_root);
         shorter = false;
         break;

      case right_higher:        //  case 3b
         sub_root->set_balance(equal_height);
         temp->set_balance(equal_height);
         rotate_left(sub_root);
         break;

      case left_higher:         //  case 3c; requires double rotation
         Binary_node<Record> *temp_left = temp->left;
         switch (temp_left->get_balance()) {
         case equal_height:
            sub_root->set_balance(equal_height);
            temp->set_balance(equal_height);
            break;

         case right_higher:
            sub_root->set_balance(left_higher);
            temp->set_balance(equal_height);
            break;
         case left_higher:
            sub_root->set_balance(equal_height);
            temp->set_balance(right_higher);
            break;
         }
         temp_left->set_balance(equal_height);
         rotate_right(sub_root->right);
         rotate_left(sub_root);
         break;
      }
   }
   return result;
}
```

P2. *Substitute the AVL tree class into the menu-driven demonstration program for binary search trees in Section 10.2, Project P2 (page 460), thereby obtaining a demonstration program for AVL trees.*

Answer Main program:

```
#include "../../c/utility.h"

void help()
/* PRE:  None.
   POST: Instructions for the tree operations have been printed.     */

{
     cout << "\n";
     cout << "\t[S]ize       [I]nsert    [D]elete \n"
             "\t[H]eight     [E]rase     [F]ind \n"
             "\t[W]idth      [C]ontents  [L]eafs     [B]readth first \n"
             "\t[P]reorder   P[O]storder I[N]order   [?]help \n" << endl;
}

#include <string.h>
char get_command()
/* PRE:  None.
   POST: A character command belonging to the set of legal commands for the
         tree demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command (? for Help) and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("clwbe?sidfponqh",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions
```

```
void write_ent(char &x)
{
   cout << x;
}

char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include auxiliary data structures

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/sortable.h"
#include "../../8/linklist/merge.cpp"
#include "../2p1p2/queue.h"
#include "../2p1p2/queue.cpp"
#include "../2p1p2/extqueue.h"
#include "../2p1p2/extqueue.cpp"

// include binary search tree data structure

#include "../2p1p2/node.h"
#include "../2p1p2/tree.h"
#include "../2p1p2/node.cpp"
#include "../2p1p2/tree.cpp"
#include "../2p1p2/stree.h"
#include "../2p1p2/snode.cpp"
#include "../2p1p2/stree.cpp"

#include "../avlt/anode.h"
#include "../avlt/atree.h"
#include "../avlt/anode.cpp"
#include "../avlt/atree.cpp"

int do_command(char c, AVL_tree<char> &test_tree)
/* PRE:  The tree has been created and command is a valid tree
         operation.
   POST: The command has been executed.
   USES: All the functions that perform tree operations.
*/

{
   char x;
   switch (c) {
   case '?': help();
     break;

   case 's':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'i':
      cout << "Enter new character to insert:";
      x = get_char();
      test_tree.insert(x);
      break;
```

TOC
Index
Help

```
case 'd':
   cout << "Enter character to remove:" << flush;
   x = get_char();
   if (test_tree.remove(x) != success) cout << "Not found!\n";
   break;

case 'f':
   cout << "Enter character to look for:";
   x = get_char();
   if (test_tree.tree_search(x) != success)
       cout << "  The entry is not present";
   else
       cout << "  The entry is present";
   cout << endl;
   break;

case 'h':
   cout << "The height of the tree is " << test_tree.height() << "\n";
   break;

case 'e':
   test_tree.clear();
   cout << "Tree is cleared.\n";
   break;

case 'w':
   cout << "The width of the tree is "
        << test_tree.width() << "\n";
   break;

case 'c':
   test_tree.bracketed_print();
   break;

case 'l':
   cout << "The leaf count of the tree is "
        << test_tree.leaf_count() << "\n";
   break;

case 'b':
   test_tree.level_traverse(write_ent);
   cout << endl;
   break;

case 'p':
   test_tree.preorder(write_ent);
   cout << endl;
   break;

case 'o':
   test_tree.postorder(write_ent);
   cout << endl;
   break;

case 'n':
   test_tree.inorder(write_ent);
   cout << endl;
   break;
```

TOC
Index
Help

```
   case 'q':
      cout << "Tree demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/* PRE:  None.
   POST: A binary tree demonstration has been performed.
   USES: get_command, do_command, Tree methods
*/

{
   Binary_tree<char> s; Binary_tree<char> t = s;
   Search_tree<char> s1; Search_tree<char> t1 = s1;

   AVL_tree<char> test_tree;
   while (do_command(get_command(), test_tree));
}
```

AVL tree class:

```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);

   void prenode(void (*f)(Binary_node<Record> *&));
   Error_code avl_insert(Binary_node<Record> *&, const Record &, bool &);
   void right_balance(Binary_node<Record> *&);
   void left_balance(Binary_node<Record> *&);
   void rotate_left(Binary_node<Record> *&);
   void rotate_right(Binary_node<Record> *&);
   Error_code remove_avl(Binary_node<Record> *&, const Record &, bool &);
   Error_code remove_avl_right(Binary_node<Record> *&,
                               const Record &, bool &);
   Error_code remove_avl_left(Binary_node<Record> *&,
                              const Record &, bool &);

private: //  Add auxiliary function prototypes here.
};
```

Implementation:

```
template <class Record>
Error_code AVL_tree<Record>::insert(const Record &new_data)
/*
Post: If the key of new_data is already in the AVL_tree, a code
      of duplicate_error is returned.
      Otherwise, a code of success is returned and the Record new_data
      is inserted into the tree in such a way that the properties of
      an AVL tree are preserved.
Uses: avl_insert.
*/

{
   bool taller;
   return avl_insert(root, new_data, taller);
}
```

```
template <class Record>
Error_code AVL_tree<Record>::remove(const Record &target)
{
   bool shorter;
   return remove_avl(root, target, shorter);
}

template <class Record>
void AVL_tree<Record>::prenode(void (*f)(Binary_node<Record> *&))
{
   avl_prenode(root, f);
}

template <class Record>
void avl_prenode(Binary_node<Record> *root,
                 void (*f)(Binary_node<Record> *&))
{
   if (root != NULL) {
      (*f)(root);
      avl_prenode(root->left, f);
      avl_prenode(root->right, f);
   }
}
```

AVL node class:

```
template <class Record>
struct AVL_node: public Binary_node<Record> {

//    additional data member:
   Balance_factor balance;
//    constructors:
   AVL_node();
   AVL_node(const Record &x);
//    overridden virtual functions:
   void set_balance(Balance_factor b);
   Balance_factor get_balance() const;
};
```

Implementation:

```
template <class Record>
AVL_node<Record>::AVL_node()
{
   balance = equal_height;
   left = right = NULL;
}

template <class Record>
AVL_node<Record>::AVL_node(const Record &x)
{
   balance = equal_height;
   left = right = NULL;
   data = x;
}

template <class Record>
void AVL_node<Record>::set_balance(Balance_factor b)
{
   balance = b;
}
```

```
template <class Record>
Balance_factor AVL_node<Record>::get_balance() const
{
   return balance;
}

template <class Record>
Error_code AVL_tree<Record>::avl_insert(Binary_node<Record> *&sub_root,
           const Record &new_data, bool &taller)
/*
Pre:  sub_root is either NULL or points to a subtree of the AVL_tree.
Post: If the key of new_data is already in the subtree, a code of
      duplicate_error is returned.
      Otherwise, a code of success is returned and the Record new_data
      is inserted into the subtree in such a way that the properties of an
      AVL tree have been preserved.  If the subtree is increased in height,
      the parameter taller is set to true; otherwise it is set to false.
Uses: Methods of struct AVL_node; functions avl_insert recursively,
      left_balance, and right_balance.
*/
{
   Error_code result = success;
   if (sub_root == NULL) {
      sub_root = new AVL_node<Record>(new_data);
      taller = true;
   }

   else if (new_data == sub_root->data) {
      result = duplicate_error;
      taller = false;
   }

   else if (new_data < sub_root->data) {   //  Insert in left subtree.
      result = avl_insert(sub_root->left, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) {//  Change balance factors.
         case left_higher:
            left_balance(sub_root);
            taller = false;     // Rebalancing always shortens the tree.
            break;

         case equal_height:
            sub_root->set_balance(left_higher);
            break;

         case right_higher:
            sub_root->set_balance(equal_height);
            taller = false;
            break;
         }
   }

   else {                                      //  Insert in right subtree.
      result = avl_insert(sub_root->right, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) {
         case left_higher:
            sub_root->set_balance(equal_height);
            taller = false;
            break;
         case equal_height:
            sub_root->set_balance(right_higher);
            break;

         case right_higher:
            right_balance(sub_root);
            taller = false;        //  Rebalancing always shortens the tree.
            break;
         }
   }
   return result;
}

template <class Record>
void AVL_tree<Record>::right_balance(Binary_node<Record> *&sub_root)
/*
Pre:  sub_root points to a subtree of an AVL_tree that
      is doubly unbalanced on the right.
Post: The AVL properties have been restored to the subtree.
Uses: Methods of struct AVL_node;
      functions  rotate_right and rotate_left.
*/

{
   Binary_node<Record> *&right_tree = sub_root->right;
   switch (right_tree->get_balance()) {
   case right_higher:                            //  single rotation left
      sub_root->set_balance(equal_height);
      right_tree->set_balance(equal_height);
      rotate_left(sub_root);
      break;

   case equal_height:  //  impossible case
      cout << "WARNING: program error detected in right_balance" << endl;

   case left_higher:                            //  double rotation left
      Binary_node<Record> *sub_tree = right_tree->left;
      switch (sub_tree->get_balance()) {

      case equal_height:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(equal_height);
         break;

      case left_higher:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(right_higher);
         break;

      case right_higher:
         sub_root->set_balance(left_higher);
         right_tree->set_balance(equal_height);
         break;
      }

      sub_tree->set_balance(equal_height);
      rotate_right(right_tree);
      rotate_left(sub_root);
      break;
   }
}
```

```
template <class Record>
void AVL_tree<Record>::left_balance(Binary_node<Record> *&sub_root)
{
   Binary_node<Record> *&left_tree = sub_root->left;
   switch (left_tree->get_balance()) {
   case left_higher:
      sub_root->set_balance(equal_height);
      left_tree->set_balance(equal_height);
      rotate_right(sub_root);
      break;

   case equal_height:  //  impossible case
      cout << "WARNING: program error detected in left_balance" << endl;

   case right_higher:
      Binary_node<Record> *sub_tree = left_tree->right;
      switch (sub_tree->get_balance()) {

      case equal_height:
         sub_root->set_balance(equal_height);
         left_tree->set_balance(equal_height);
         break;

      case right_higher:
         sub_root->set_balance(equal_height);
         left_tree->set_balance(left_higher);
         break;

      case left_higher:
         sub_root->set_balance(right_higher);
         left_tree->set_balance(equal_height);
         break;
      }

      sub_tree->set_balance(equal_height);
      rotate_left(left_tree);
      rotate_right(sub_root);
      break;
   }
}

template <class Record>
void AVL_tree<Record>::rotate_left(Binary_node<Record> *&sub_root)
/*
Pre:  sub_root points to a subtree of the AVL_tree.
      This subtree has a nonempty right subtree.
Post: sub_root is reset to point to its former right child, and the former
      sub_root node is the left child of the new sub_root node.
*/

{
   if (sub_root == NULL || sub_root->right == NULL)     //  impossible cases
      cout << "WARNING: program error detected in rotate_left" << endl;
   else {
      Binary_node<Record> *right_tree = sub_root->right;
      sub_root->right = right_tree->left;
      right_tree->left = sub_root;
      sub_root = right_tree;
   }
}
```

```
template <class Record>
void AVL_tree<Record>::rotate_right(Binary_node<Record> *&sub_root)
{
   if (sub_root == NULL || sub_root->left == NULL) //  impossible cases
      cout << "WARNING: program error in detected in rotate_right" << endl;
   else {
      Binary_node<Record> *left_tree = sub_root->left;
      sub_root->left = left_tree->right;
      left_tree->right = sub_root;
      sub_root = left_tree;
   }
}
```

The removal method is not listed here, since it appears in the solution to Project P1.

P3. *Substitute the AVL tree class into the information-retrieval project of Project P5 of Section 10.2 ((page 461)). Compare the performance of AVL trees with ordinary binary search trees for various combinations of input text files.*

Answer

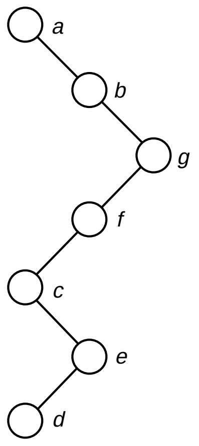
```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/doubly/list.h"
#include "../../6/doubly/list.cpp"
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "../bt/node.h"
#include "../bt/tree.h"
#include "../bt/node.cpp"
#include "../bt/tree.cpp"

#include "../bst/stree.h"
#include "../bst/snode.cpp"
#include "../bst/stree.cpp"

#include "../2p5/key.h"
#include "../2p5/key.cpp"
#include "../2p5/record.h"
#include "../2p5/record.cpp"

#include "../avlt/anode.h"
#include "../avlt/atree.h"
#include "../avlt/anode.cpp"
#include "../avlt/atree.cpp"

#include "auxil.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Name of an input file can be given as a command-line argument.
Post: The word storing project p6 has been performed.
*/

{
   write_method();
   Binary_tree<Record> t;
   Search_tree<Record> t1;
   AVL_tree<Record> the_words;

   char infile[1000];
   char in_string[1000];
   char *in_word;
   Timer clock;
   do {
      int initial_comparisons = Key::comparisons;
      clock.reset();
      if (argc < 2) {
         cout << "What is the input data file: " << flush;
         cin >> infile;
      }
      else strcpy(infile, argv[1]);

      ifstream file_in(infile);   //  Declare and open the input stream.
      if (file_in == 0) {
         cout << "Can't open input file " << infile << endl;
         exit (1);
      }

      while (!file_in.eof()) {
        file_in >> in_string;
        int position = 0;
        bool end_string = false;
        while (!end_string) {
           in_word = extract_word(in_string, position, end_string);
           if (strlen(in_word) > 0) {
              int counter;
              String s(in_word);
              update(s, the_words, counter);
           }
        }
      }
      cout << "Elapsed time: " << clock.elapsed_time() << endl;
      cout << "Comparisons performed = "
           << Key::comparisons - initial_comparisons << endl;
      cout << "Do you want to print frequencies? ";
      if (user_says_yes()) print(the_words);
      cout << "Do you want to add another file of input? ";
   } while (user_says_yes());
}

void write_method()
/*
Post:  A short string identifying the abstract
       data type used for the structure is written.
*/

{
   cout << " Word frequency program:  AVL-Tree implementation";
   cout << endl;
}

void update(const String &word, AVL_tree<Record> &structure,
               int &num_comps)
/*
```

```
Post:  If word was not already present in structure, then
       word has been inserted into structure and its frequency
       count is 1.  If word was already present in structure,
       then its frequency count has been increased by 1.  The
       variable parameter num_comps is set to the number of
       comparisons of words done.
*/
{
  int initial_comparisons = Key::comparisons;
  Record r((Key) word);
  if (structure.tree_search(r) == not_present)
     structure.insert(r);
  else {
     structure.remove(r);
     r.increment();
     structure.insert(r);
  }
  num_comps = Key::comparisons - initial_comparisons;
}

void wr_entry(Record &a_word)
{
   String s = ((Key) a_word).the_key();
   cout << s.c_str();
   for (int i = strlen(s.c_str()); i < 12; i++) cout << " ";
   cout << " : " << a_word.frequency() << endl;
}

void print(AVL_tree<Record> &structure)
/*
Post:  All words in structure are printed at the terminal
       in alphabetical order together with their frequency counts.
*/
{
     structure.inorder(wr_entry);
}

char *extract_word(char *in_string, int &examine, bool &over)
{
   int ok = 0;
   while (in_string[examine] != '\0') {
     if ('a' <= in_string[examine] && in_string[examine] <= 'z')
       in_string[ok++] = in_string[examine++];
     else if ('A' <= in_string[examine] && in_string[examine] <= 'Z')
       in_string[ok++] = in_string[examine++] - 'A' + 'a';
     else if (('\'' == in_string[examine] || in_string[examine] == '-')
       && (
       ('a' <= in_string[examine + 1] && in_string[examine + 1] <= 'z') ||
       ('A' <= in_string[examine + 1] && in_string[examine + 1] <= 'Z')))
       in_string[ok++] = in_string[examine++];
     else break;
   }
   in_string[ok] = '\0';
   if (in_string[examine] == '\0') over = true;
   else examine++;
   return in_string;
}
```

10.5 SPLAY TREES: A SELF-ADJUSTING DATA STRUCTURE

Exercises 10.5

E1. *Consider the following binary search tree:*

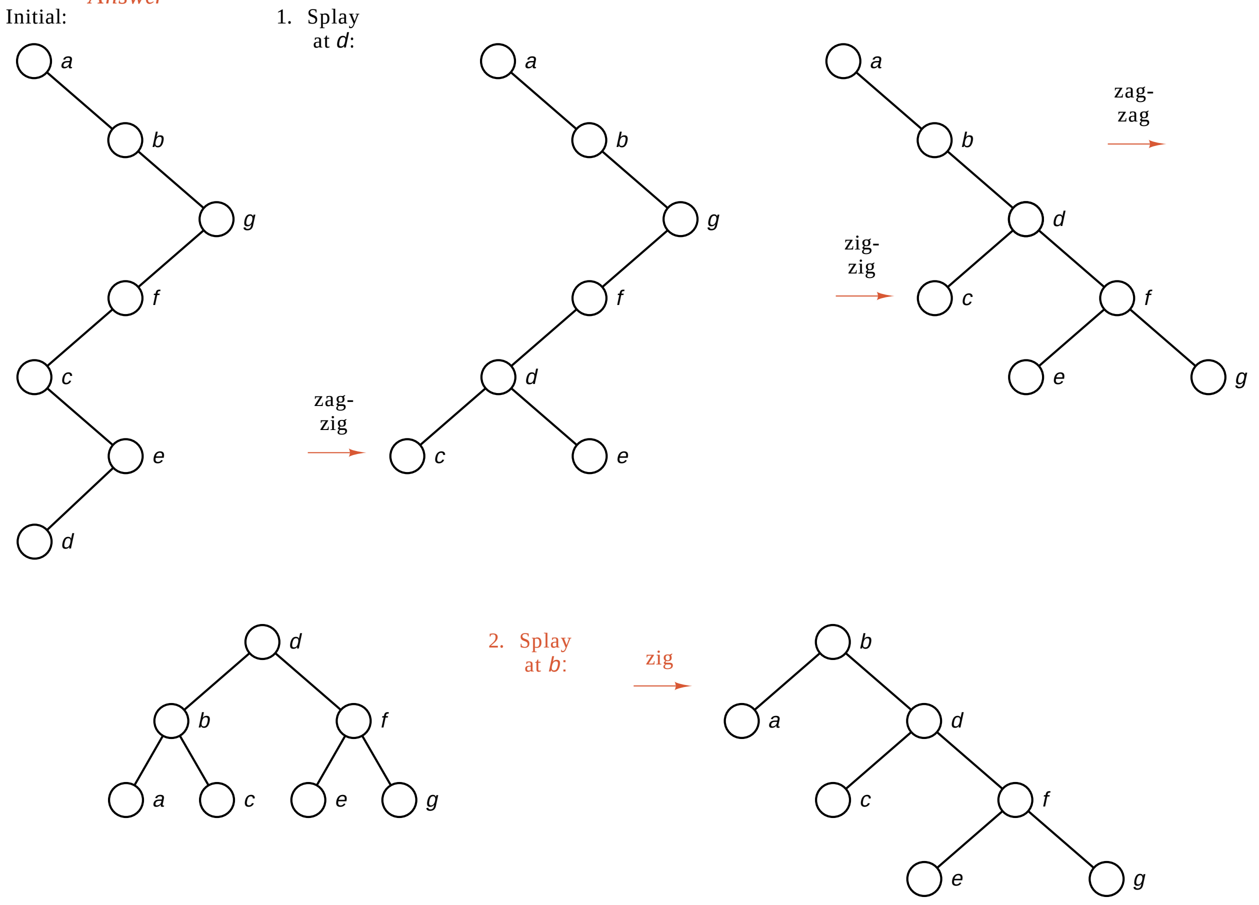


Splay this tree at each of the following keys in turn:

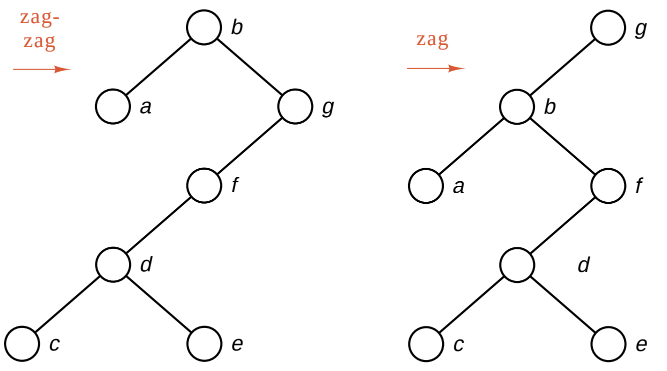
d b g f a d b d

Each part builds on the previous; that is, use the final tree of each solution as the starting tree for the next part. [Check: The tree should be completely balanced after the last part, as well as after one previous part.]

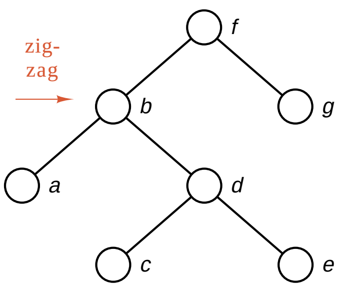
Answer



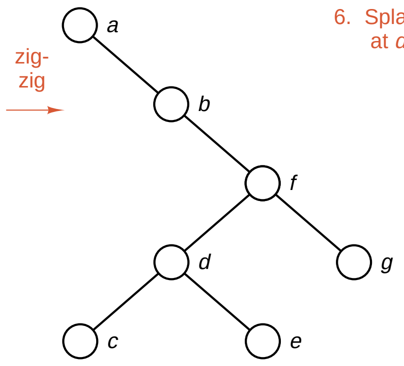
3. Splay at g:



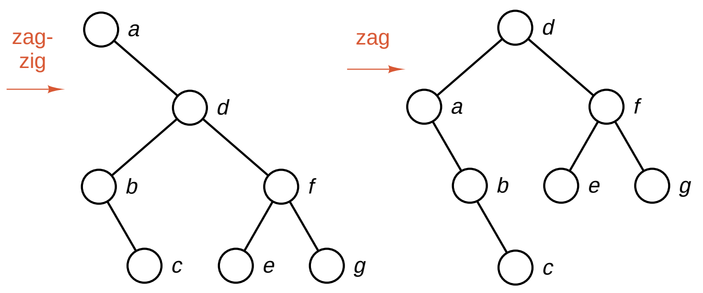
4. Splay at f:

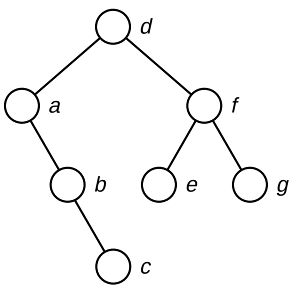


5. Splay at a:

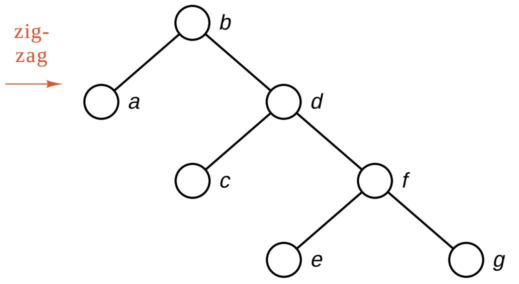


6. Splay at d:

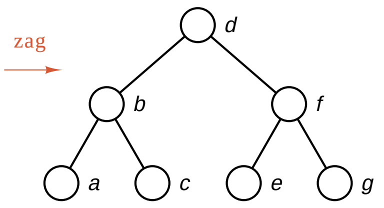




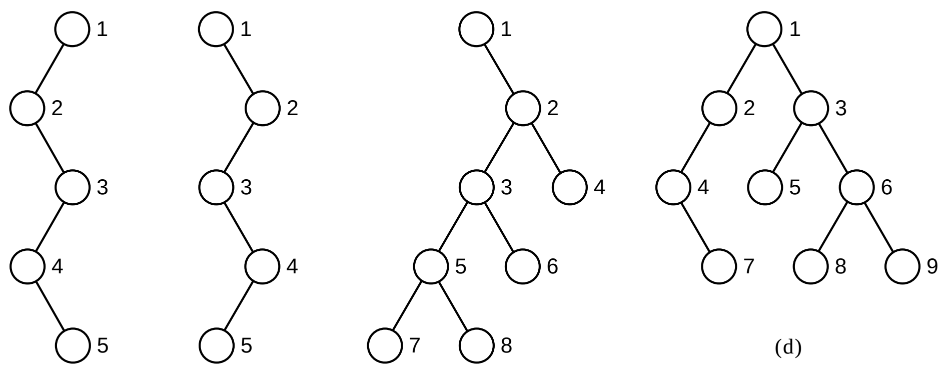
7. Splay at b:



8. Splay at d:



E2. *The **depth** of a node in a binary tree is the number of branches from the root to the node. (Thus the root has depth 0, its children depth 1, and so on.) Define the credit balance of a tree during preorder traversal to be the depth of the node being visited. Define the (actual) cost of visiting a vertex to be the number of branches traversed (either going down or up) from the previously visited node. For each of the following binary trees, make a table showing the nodes visited, the actual cost, the credit balance, and the amortized cost for a preorder traversal.*



(a) (b) (c) (d)

Answer In the following tables, the actual cost of visiting vertex i is denoted t_i, the credit balance (rank) by c_i, and the amortized cost by a_i. The vertices are arranged in order of preorder traversal.

(a)

i	t_i	c_i	a_i
1	0	0	0
2	1	1	2
3	1	2	2
4	1	3	2
5	1	4	2

(b)

i	t_i	c_i	a_i
1	0	0	0
2	1	1	2
3	1	2	2
4	1	3	2
5	1	4	2

(c)

i	t_i	c_i	a_i
1	0	0	0
2	1	1	2
3	1	2	2
5	1	3	2
7	1	4	2
8	2	4	2
6	3	3	2
4	3	2	2

(d)

i	t_i	c_i	a_i
1	0	0	0
2	1	1	2
4	1	2	2
7	1	3	2
3	4	1	2
5	1	2	2
6	2	2	2
8	1	3	2
9	2	3	2

E3. *Define a rank function $r(x)$ for the nodes of any binary tree as follows: If x is the root, then $r(x) = 0$. If x is the left child of a node y, then $r(x) = r(y) - 1$. If x is the right child of a node y, then $r(x) = r(y) + 1$. Define the credit balance of a tree during a traversal to be the rank of the node being visited. Define the (actual) cost of visiting a vertex to be the number of branches traversed (either going down or up) from the previously visited node. For each of the binary trees shown in Exercise E2, make a table showing the nodes visited, the actual cost, the credit balance, and the amortized cost for an inorder traversal.*

Answer In the following tables, the actual cost of visiting vertex i is denoted t_i, the credit balance (rank) by c_i, and the amortized cost by a_i. The vertices are arranged in order of inorder traversal.

(a)

i	t_i	c_i	a_i
2	1	−1	0
4	2	−1	2
5	1	0	2
3	2	0	2
1	2	0	2

(b)

i	t_i	c_i	a_i
1	0	0	0
3	2	0	2
5	2	0	2
4	1	1	2
2	2	1	2

(c)

i	t_i	c_i	a_i
1	0	0	0
7	4	−2	2
5	1	−1	2
8	1	0	2
3	2	0	2
6	1	1	2
2	2	1	2
4	1	2	2

(d)

i	t_i	c_i	a_i
4	2	−2	0
7	1	−1	2
2	2	−1	2
1	1	0	2
5	2	0	2
3	1	1	2
8	2	1	2
6	1	2	2
9	1	3	2

E4. *In analogy with Exercise E3, devise a rank function for binary trees that, under the same conditions as in Exercise E3, will make the amortized costs of a postorder traversal almost all the same. Illustrate your rank function by making a table for each of the binary trees shown in Exercise E2, showing the nodes visited, the actual cost, the credit balance, and the amortized cost for a postorder traversal.*

Answer Define the *credit balance* of a tree during preorder traversal to be the negative of the depth of the node being visited. In the following tables, corresponding to the binary trees shown in Exercise E2, the actual cost of visiting vertex i is denoted t_i, the credit balance (rank) by c_i, and the amortized cost by a_i. The vertices are arranged in order of postorder traversal.

TOC
Index
Help

(a)

i	t_i	c_i	a_i
5	4	−4	0
4	1	−3	2
3	1	−2	2
2	1	−1	2
1	1	0	2

(b)

i	t_i	c_i	a_i
5	4	−4	0
4	1	−3	2
3	1	−2	2
2	1	−1	2
1	1	0	2

(c)

i	t_i	c_i	a_i
7	4	−4	0
8	2	−4	2
5	1	−3	2
6	2	−3	2
3	1	−2	2
4	2	−2	2
2	1	−1	2
1	1	0	2

(d)

i	t_i	c_i	a_i
7	3	−3	0
4	1	−2	2
2	1	−1	2
5	3	−2	2
8	3	−3	2
9	2	−3	2
6	1	−2	2
3	1	−1	2
1	1	0	2

E5. *Generalize the amortized analysis given in the text for incrementing four-digit binary integers to n-digit binary integers.*

Answer Let k be the number of 1's at the far right of the integer before step i. (Then, for example, $k = 0$ if the rightmost digit is a 0.) Suppose, first, that we are not at the very last step, so the integer is not all 1's, which means $k < n$. Then the actual work done to increment the integer consists of changing these k 1's into 0 and changing the 0 immediately on their left into a 1. Hence $t_i = k+1$. The total number of 1's in the integer decreases by $k-1$, since k 1's are changed to 0 and one 0 is changed to 1; this means that the credit balance changes by $c_i - c_{i-1} = -(k-1)$. The amortized cost of step i is therefore

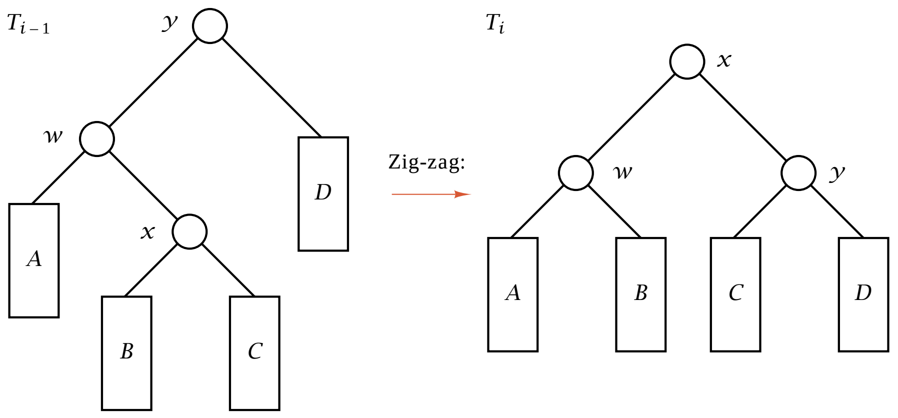
$$a_i = t_i + c_i - c_{i-1} = (k+1)-(k-1) = 2.$$

At the very last step (when $i = 2^n$), the integer changes from all 1's to all 0's. Hence the actual cost is $t_{2^n} = n$, the credit before the increment is $c_{2^n-1} = n$, and the credit after the increment is $c_{2^n} = 0$. Hence the amortized cost of the last step is

$$a_{2^n} = n + 0 - n = 0.$$

E6. *Prove Lemma 10.8. The method is similar to the proof of Lemma 10.7.*

Answer This case is illustrated as follows:



The actual complexity t_i of a zig-zag or a zag-zig step is 2 units, and only the sizes of the subtrees rooted at w, x, and y change in this step. Therefore, all terms in the summation defining c_i cancel against those for c_{i-1} except those indicated in the following equation:

$$\begin{aligned} a_i &= t_i + c_i - c_{i-1} \\ &= 2 + r_i(w)+r_i(x)+r_i(y)-r_{i-1}(w)-r_{i-1}(x)-r_{i-1}(y) \\ &= 2 + r_i(w)+r_i(y)-r_{i-1}(w)-r_{i-1}(x) \end{aligned}$$

We obtain the last line by taking the logarithm of $|T_i(x)| = |T_{i-1}(y)|$, which is the observation that the subtree rooted at y before the splaying step has the same size as that rooted at x after

TOC Index Help

the step. Lemma 10.6 can now be applied to cancel the 2 in this equation (the actual complexity). Let $\alpha = |T_i(w)|$, $\beta = |T_i(y)|$, and $\gamma = |T_i(x)|$. From the diagram for this case, we see that $T_i(w)$ contains components w, A, and B; $T_i(y)$ contains components y, C, and D; and $T_i(x)$ contains all these components (and x besides). Hence $\alpha + \beta < \gamma$, so Lemma 10.6 implies $r_i(w)+r_i(y) \le 2r_i(x)-2$, or $2r_i(x)-r_i(w)-r_i(y)-2 \ge 0$. Adding this nonnegative quantity to the right side of the last equation for a_i, we obtain

$$a_i \le 2r_i(x)-r_{i-1}(x)-r_{i-1}(w).$$

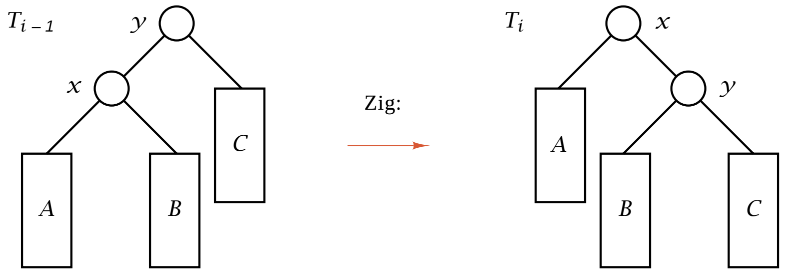
Before step i, w is the parent of x, so $|T_{i-1}(w)| > |T_{i-1}(x)|$. Taking logarithms, we have $r_{i-1}(w) > r_{i-1}(x)$. Hence we finally obtain

$$a_i < 2r_i(x)-2r_{i-1}(x),$$

which is the assertion in Lemma 10.8 that we wished to prove.

E7. *Prove Lemma 10.9. This proof does not require Lemma 10.6 or any intricate calculations.*

Answer This case is illustrated as follows:



The actual complexity t_i of a zig or a zag step is 1 unit, and only the sizes of the subtrees rooted at x and y change in this step. Therefore, all terms in the summation defining c_i cancel against those for c_{i-1} except those indicated in the following equation:

$$\begin{aligned} a_i &= t_i + c_i - c_{i-1} \\ &= 1 + r_i(x)+r_i(y)-r_{i-1}(x)-r_{i-1}(y) \\ &= 1 + r_i(y)-r_{i-1}(x) \end{aligned}$$

We obtain the last line by taking the logarithm of $|T_i(x)| = |T_{i-1}(y)|$, which is the observation that the subtree rooted at y before the splaying step has the same size as that rooted at x after the step. After step i, x is the parent of y, so $|T_i(x)| > |T_i(y)|$. Taking logarithms, we have $r_i(x) > r_i(y)$. Hence we finally obtain

$$a_i < 1 + r_i(x)-r_{i-1}(x),$$

which is the assertion in Lemma 10.9 that we wished to prove.

Programming Projects 10.5

P1. *Substitute the splay tree class into the menu-driven demonstration program for binary search trees in Section 10.2, Project P2 (page 460), thereby obtaining a demonstration program for splay trees.*

Answer Main program:

```
#include "../../c/utility.h"

void help()
/* PRE:  None.
   POST: Instructions for the tree operations have been printed.
*/
```

```
{
     cout << "\n";
     cout << "\t[S]ize        [I]nsert    [D]elete \n"
             "\t[H]eight      [E]rase     [F]ind     spla[Y] \n"
             "\t[W]idth       [C]ontents  [L]eafs    [B]readth first \n"
             "\t[P]reorder   P[O]storder I[N]order  [?]help \n" << endl;
}

#include <string.h>
char get_command()
/* PRE:  None.
   POST: A character command belonging to the set of legal commands for
         the tree demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command (? for Help) and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);

      if(strchr("clwbe?sidfponqhy",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

// auxiliary input/output functions

void write_ent(char &x)
{
   cout << x;
}

char get_char()
{
   char c;
   cin >>c;
   return c;
}

// include auxiliary data structures

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/sortable.h"
#include "../../8/linklist/merge.cpp"
#include "../2p1p2/queue.h"
#include "../2p1p2/queue.cpp"
#include "../2p1p2/extqueue.h"
#include "../2p1p2/extqueue.cpp"
```

```
// include binary search tree data structure
#include "../2p1p2/node.h"
#include "../2p1p2/tree.h"
#include "../2p1p2/node.cpp"
#include "../2p1p2/tree.cpp"
#include "../2p1p2/stree.h"
#include "../2p1p2/snode.cpp"
#include "../2p1p2/stree.cpp"

#include "../sst/splaytre.h"
#include "../sst/splaynod.cpp"
#include "../sst/splaytre.cpp"

int do_command(char c, Splay_tree<char> &test_tree)
/* PRE:  The tree has been created and command is a valid tree
         operation.
   POST: The command has been executed.
   USES: All the functions that perform tree operations.
*/

{
   char x;
   switch (c) {
   case '?': help();
     break;

   case 'y':
      cout << "Enter character to splay for:";
      x = get_char();
      if (test_tree.splay(x) == entry_inserted)
           cout << "Entry has been inserted" << endl;
      else cout << "Entry has been located" << endl;
      break;

   case 's':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'i':
      cout << "Enter new character to insert:";
      x = get_char();
      if (test_tree.splay(x) == entry_inserted)
           cout << "Entry has been inserted" << endl;
      else cout << "A duplicate entry has been located" << endl;
      break;

   case 'd':
      cout << "Enter character to remove:" << flush;
      x = get_char();
      if (test_tree.remove(x) != success) cout << "Not found!\n";
      break;

   case 'f':
      cout << "Enter character to look for:";
      x = get_char();
      if (test_tree.tree_search(x) != success)
          cout << "  The entry is not present";
      else
          cout << "  The entry is present";
      cout << endl;
      break;
   case 'h':
      cout << "The height of the tree is " << test_tree.height() << "\n";
      break;

   case 'e':
      test_tree.clear();
      cout << "Tree is cleared.\n";
      break;

   case 'w':
      cout << "The width of the tree is "
           << test_tree.width() << "\n";
      break;

   case 'c':
      test_tree.bracketed_print();
      break;

   case 'l':
      cout << "The leaf count of the tree is "
           << test_tree.leaf_count() << "\n";
      break;

   case 'b':
      test_tree.level_traverse(write_ent);
      cout << endl;
      break;

   case 'p':
      test_tree.preorder(write_ent);
      cout << endl;
      break;

   case 'o':
      test_tree.postorder(write_ent);
      cout << endl;
      break;

   case 'n':
      test_tree.inorder(write_ent);
      cout << endl;
      break;

   case 'q':
      cout << "Tree demonstration finished.\n";
      return 0;
   }
   return 1;
}

int main()
/* PRE:  None.
   POST: A binary tree demonstration has been performed.
   USES: get_command, do_command, Tree methods
*/

{
   Binary_tree<char> s; Binary_tree<char> t = s;
   Search_tree<char> s1; Search_tree<char> t1 = s1;

   Splay_tree<char> test_tree;
   while (do_command(get_command(), test_tree));
}
```

Splay tree class:

```
template <class Record>
class Splay_tree:public Search_tree<Record> {
public:
   Error_code splay(const Record &target);

   void prenode(void (*f)(Binary_node<Record> *&));
   void link_left(Binary_node<Record> *&, Binary_node<Record> *&);
   void link_right(Binary_node<Record> *&, Binary_node<Record> *&);
   void rotate_left(Binary_node<Record> *&);
   void rotate_right(Binary_node<Record> *&);

private: //  Add auxiliary function prototypes here.
};
```

Implementation:

```
template <class Record>
void Splay_tree<Record>::prenode(void (*f)(Binary_node<Record> *&))
{
   splay_prenode(root, f);
}

template <class Record>
void splay_prenode(Binary_node<Record> *root,
                   void (*f)(Binary_node<Record> *&))
{
   if (root != NULL) {
      (*f)(root);
      splay_prenode(root->left, f);
      splay_prenode(root->right, f);
   }
}
```

Splay tree node:

```
template <class Record>
void Splay_tree<Record>::link_right(Binary_node<Record> *&current,
                                    Binary_node<Record> *&first_large)

/*
Pre:  The pointer first_large points to an actual Binary_node
      (in particular, it is not NULL).  The three-way invariant holds.
Post: The node referenced by current (with its right subtree) is linked
      to the left of the node referenced by first_large.
      The pointer first_large is reset to current.
      The three-way invariant continues to hold.
*/

{
   first_large->left = current;
   first_large = current;
   current = current->left;
}

template <class Record>
void Splay_tree<Record>::link_left(Binary_node<Record> *&current,
                                   Binary_node<Record> *&last_small)
```

```
/*
Pre:   The pointer last_small points to an actual Binary_node
      (in particular, it is not NULL).  The three-way invariant holds.
Post: The node referenced by current (with its left subtree) is linked
      to the right of the node referenced by last_small.
      The pointer last_small is reset to current.
      The three-way invariant continues to hold.
*/

{
   last_small->right = current;
   last_small = current;
   current = current->right;
}

template <class Record>
void Splay_tree<Record>::rotate_left(Binary_node<Record> *&current)
/*
Pre:  current points to the root node of a subtree of a Binary_tree.
      This subtree has a nonempty right subtree.
Post: current is reset to point to its former right child, and the former
      current node is the left child of the new current node.
*/

{
   Binary_node<Record> *right_tree = current->right;
   current->right = right_tree->left;
   right_tree->left = current;
   current = right_tree;
}

template <class Record>
void Splay_tree<Record>::rotate_right(Binary_node<Record> *&current)
/*
Pre:   current points to the root node of a subtree of a Binary_tree.
      This subtree has a nonempty left subtree.
Post: current is reset to point to its former left child, and the former
      current node is the right child of the new current node.
*/

{
   Binary_node<Record> *left_tree = current->left;
   current->left = left_tree->right;
   left_tree->right = current;
   current = left_tree;
}

template <class Record>
Error_code Splay_tree<Record>::splay(const Record &target)
/*
Post: If a node of the splay tree has a key matching that of target,
      it has been moved by splay operations to be the root of the
      tree, and a code of entry_found is returned.  Otherwise,
      a new node containing a copy of target is inserted as the root
      of the tree, and a code of entry_inserted is returned.
*/
```

```
{
   Binary_node<Record> *dummy = new Binary_node<Record>;
   Binary_node<Record> *current = root,
                       *child,
                       *last_small = dummy,
                       *first_large = dummy;

// Search for target while splaying the tree.
   while (current != NULL && current->data != target)
      if (target < current->data) {
         child = current->left;
         if (child == NULL || target == child->data)    //  zig move
            link_right(current, first_large);
         else if (target < child->data) {            //  zig-zig move
            rotate_right(current);
            link_right(current, first_large);
         }
         else {                                      //  zig-zag move
            link_right(current, first_large);
            link_left(current, last_small);
         }
      }

      else {                              //  case: target > current->data
         child = current->right;
         if (child == NULL || target == child->data)
            link_left(current, last_small);             //  zag move
         else if (target > child->data) {            //  zag-zag move
            rotate_left(current);
            link_left(current, last_small);
         }
         else {                                      //  zag-zig move
            link_left(current, last_small);
            link_right(current, first_large);
         }
      }

//  Move root to the current node, which is created if necessary.
   Error_code result;
   if (current == NULL) {         //  Search unsuccessful: make a new root.
      current = new Binary_node<Record>(target);
      result = entry_inserted;
      last_small->right = first_large->left = NULL;
   }

   else {                         //  successful search
      result = entry_found;
      last_small->right = current->left;  //  Move remaining central nodes.
      first_large->left = current->right;
   }

   root = current;                   //  Define the new root.
   root->right = dummy->left;        //  root of larger-key subtree
   root->left = dummy->right;        //  root of smaller-key subtree
   delete dummy;
   return result;
}
```

P2. *Substitute the function for splay retrieval and insertion into the information-retrieval project of Project P5 of Section 10.2 (page 461). Compare the performance of splay trees with ordinary binary search trees for various combinations of input text files.*

information retrieval

Answer Main program:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/doubly/list.h"
#include "../../6/doubly/list.cpp"
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "../bt/node.h"
#include "../bt/tree.h"
#include "../bt/node.cpp"
#include "../bt/tree.cpp"

#include "../bst/stree.h"
#include "../bst/snode.cpp"
#include "../bst/stree.cpp"

#include "../2p5/key.h"
#include "../2p5/key.cpp"
#include "../2p5/record.h"
#include "../2p5/record.cpp"

#include "../sst/splaytre.h"
#include "../sst/splaynod.cpp"
#include "../sst/splaytre.cpp"

#include "auxil.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Name of an input file can be given as a command-line argument.
Post: The word storing project p6 has been performed.
*/

{
   write_method();
   Binary_tree<Record> t;
   Search_tree<Record> t1;

   Splay_tree<Record> the_words;

   char infile[1000];
   char in_string[1000];
   char *in_word;
   Timer clock;
   do {
      int initial_comparisons = Key::comparisons;
      clock.reset();
      if (argc < 2) {
         cout << "What is the input data file: " << flush;
         cin >> infile;
      }
      else strcpy(infile, argv[1]);
      ifstream file_in(infile);   //  Declare and open the input stream.
      if (file_in == 0) {
         cout << "Can't open input file " << infile << endl;
         exit (1);
      }

      while (!file_in.eof()) {
        file_in >> in_string;
        int position = 0;
        bool end_string = false;
        while (!end_string) {
           in_word = extract_word(in_string, position, end_string);
           if (strlen(in_word) > 0) {
              int counter;
              String s(in_word);
              update(s, the_words, counter);
           }
        }
      }
      cout << "Elapsed time: " << clock.elapsed_time() << endl;
      cout << "Comparisons performed = "
           << Key::comparisons - initial_comparisons << endl;
      cout << "Do you want to print frequencies? ";
      if (user_says_yes()) print(the_words);
      cout << "Do you want to add another file of input? ";
   } while (user_says_yes());
}
```

TOC
Index
Help

Auxiliary functions:

```
void write_method()
/*
Post:  A short string identifying the abstract
       data type used for the structure is written.
*/

{
   cout << " Word frequency program:  Splay-Tree implementation";
   cout << endl;
}

void update(const String &word, Splay_tree<Record> &structure,
               int &num_comps)
/*
Post:  If word was not already present in structure, then
       word has been inserted into structure and its frequency
       count is 1.  If word was already present in structure,
       then its frequency count has been increased by 1.  The
       variable parameter num_comps is set to the number of
       comparisons of words done.
*/
```

```
{
  int initial_comparisons = Key::comparisons;
  Record r((Key) word);
  if (structure.splay(r) != entry_inserted) {
     structure.tree_search(r);
     structure.remove(r);
     r.increment();
     structure.insert(r);
  }
  num_comps = Key::comparisons - initial_comparisons;
}

void wr_entry(Record &a_word)
{
   String s = ((Key) a_word).the_key();
   cout << s.c_str();
   for (int i = strlen(s.c_str()); i < 12; i++) cout << " ";
   cout << " : " << a_word.frequency() << endl;
}

void print(Splay_tree<Record> &structure)
/*
Post:  All words in structure are printed at the terminal
       in alphabetical order together with their frequency counts.
*/

{
     structure.inorder(wr_entry);
}

char *extract_word(char *in_string, int &examine, bool &over)
{
   int ok = 0;
   while (in_string[examine] != '\0') {
     if ('a' <= in_string[examine] && in_string[examine] <= 'z')
       in_string[ok++] = in_string[examine++];
     else if ('A' <= in_string[examine] && in_string[examine] <= 'Z')
       in_string[ok++] = in_string[examine++] - 'A' + 'a';
     else if (('\'' == in_string[examine] || in_string[examine] == '-')
       && (
       ('a' <= in_string[examine + 1] && in_string[examine + 1] <= 'z') ||
       ('A' <= in_string[examine + 1] && in_string[examine + 1] <= 'Z')))
       in_string[ok++] = in_string[examine++];
     else break;
   }
   in_string[ok] = '\0';
   if (in_string[examine] == '\0') over = true;
   else examine++;
   return in_string;
}
```

REVIEW QUESTIONS

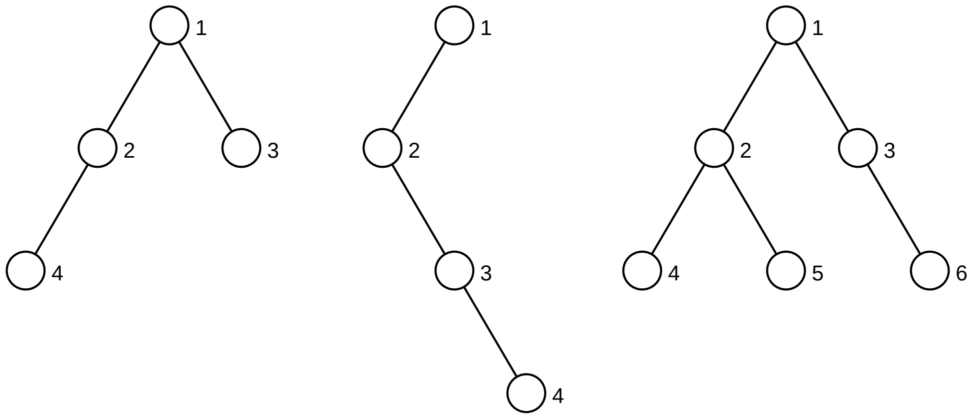
1. *Define the term binary tree.*

A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root.

2. *What is the difference between a binary tree and an ordinary tree in which each vertex has at most two branches?*

In a binary tree left and right are distinguished, but they are not in an ordinary tree. When a node has just one child, there are two cases for a binary tree but only one for an ordinary tree.

3. *Give the order of visiting the vertices of each of the following binary trees under* **(a)** *preorder,* **(b)** *inorder, and* **(c)** *postorder traversal.*



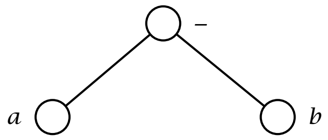
(a) (b) (c)

(1) 1, 2, 4, 3	1, 2, 3, 4	1, 2, 4, 5, 3, 6
(2) 4, 2, 1, 3	2, 3, 4, 1	4, 2, 5, 1, 3, 6
(3) 4, 2, 3, 1	4, 3, 2, 1	4, 5, 2, 6, 3, 1

4. *Draw the expression trees for each of the following expressions, and show the result of traversing the tree in* **(a)** *preorder,* **(b)** *inorder, and* **(c)** *postorder.*

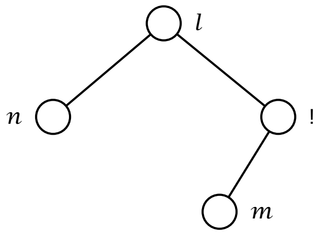
(a) $a - b$.

(1) $-\ a\ b$
(2) $a\ -\ b$
(3) $a\ b\ -$



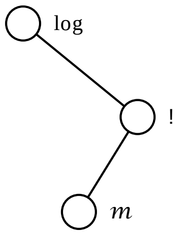
(b) $n/m!$.

(1) $/\ n\ !\ m$
(2) $n\ /\ m\ !$
(3) $n\ m\ !\ /$



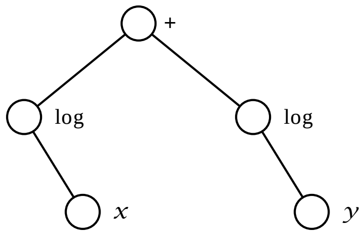
(c) $\log m!$.

(1) $\log\ !\ m$
(2) $\log\ m\ !$
(3) $m\ !\ \log$



(d) $(\log x) + (\log y)$.

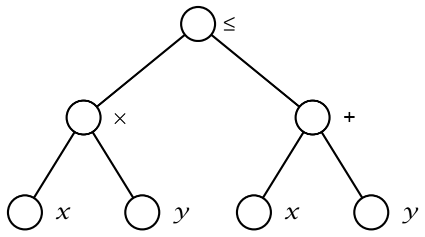
(1) $+\ \log\ x\ \log\ y$
(2) $\log\ x\ +\ \log\ y$
(3) $x\ \log\ y\ \log\ +$



TOC
Index
Help

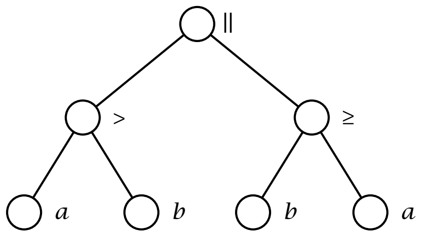
(e) $x \times y \leq x + y$.

(1) $\leq \times x\ y + x\ y$
(2) $x \times y \leq x + y$
(3) $x\ y \times x\ y + \leq$



(f) (a > b) || (b >= a)

(1) or $>$ a b $\geq$ b a
(2) $a > b$ or $b \geq a$
(3) a b $>$ b a $\geq$ or



5. *Define the term binary search tree.*

A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the condition: (1) all keys (if any) in the left subtree of the root precede the key in the root; (2) the key if the root precedes all keys (if any) in its right subtree; and (3) the left and right subtrees are binary search trees.

6. *If a binary search tree with n nodes is well balanced, what is the approximate number of comparisons of keys needed to find a target? What is the number if the tree degenerates to a chain?*

The approximate number of comparisons of keys needed to find a target in a well balanced binary search tree is $2 \lg n$ (with two comparisons per node) as opposed to approximately n comparisons (halfway, on average, through a chain of length n) if the tree degenerates to a chain.

7. *In twenty words or less, explain how treesort works.*

The items are built into a binary search tree which is then traversed in inorder yielding a sorted list.

8. *What is the relationship between treesort and quicksort?*

In quicksort the keys are partitioned into left and right sublists according to their size relative to the pivot. In treesort the keys are partitioned into the left and right subtrees according to their sizes relative to the root key. Both methods then proceed by recursion. Hence treesort proceeds by making exactly the same comparisons as quicksort does when the first key in every list is chosen as the pivot.

9. *What causes removal from a search tree to be more difficult than insertion into a search tree?*

Since it is possible to make all insertions at the leaves of the tree, insertions are very easy to perform without destroying the binary search tree property. Deletions, however, must occur at the designated node, a node which may have children thus complicating the function of easily removing it without destroying the binary search tree property.

10. *When is the algorithm for building a binary search tree developed in Section 10.3 useful, and why is it preferable to simply using the function for inserting an item into a search tree for each item in the input?*

The algorithm is designed for applications in which the input is already sorted by key. The algorithm will then build the input into a nearly balanced search tree, while the simple insertion algorithm will construct a tree that degenerates into a single long chain, for which retrieval time will not be logarithmic.

TOC
Index
Help

11. *How much slower, on average, is searching a random binary search tree than is searching a completely balanced binary search tree?*

The average binary search tree requires approximately $2 \ln 2 \approx 1.39$ times as many comparisons as a completely balanced tree.

12. *What is the purpose of AVL trees?*

The purpose of AVL trees is to have a binary search tree that remains balanced as insertions and removals take place.

13. *What condition defines an AVL tree among all binary search trees?*

An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

14. *Suppose that* A *is a base class and* B *is a derived class, and that we declare:* `A *pA; B *pB;` *Can* pA *reference an object of* **class** B*? Can* pB *reference an object of* **class** A*?*

A pointer to an object from a base class can also point to an object of a derived class. Accordingly, the pointer pA can point to an object of B. The pointer pB cannot point to an object of A.

15. *Explain how the virtual methods of a class differ from other class methods.*

The choice of implementation called for a virtual method is selected dynamically, at run time. This choice is made to correspond to the dynamically changing type of the object from which the method is called.

16. *Draw a picture explaining how balance is restored when an insertion into an AVL tree puts a node out of balance.*

Refer to Figures Figure 10.17 (page 474), Figure 10.18 (page 477), and Figure 10.19 on page 480 of the text.

17. *How does the worst-case performance of an AVL tree compare with the worst-case performance of a random binary search tree? How does it compare with its average-case performance? How does the average-case performance of an AVL tree compare with that of a random binary search tree?*

The worst-case performance of an AVL tree is approximately $1.44 \lg n$ as opposed to the worst-case performance of a random binary tree which is n. The average-case performance of an AVL tree is approximately $\lg n + 0.25$ and the average-case performance of a random binary tree is approximately $1.39 \lg n$.

18. *In twenty words or less, describe what splaying does.*

By doing certain rotations, splaying lifts each node when inserted or retrieved to become the root.

19. *What is the purpose of splaying?*

Splaying helps to balance the binary search tree and to keep more frequently accessed nodes near the root where they will be found more quickly.

20. *What is amortized algorithm analysis?*

We consider a *sequence* of operations on the data structure and determine the worst-case cost of the entire sequence. This may be much less than the worst-case cost of a single operation multiplied by the number of operations in the sequence.

21. *What is a credit-balance function, and how is it used?*

This function is a device added to the actual cost to help simplify the calculation of the amortized cost of operations. Its goal is to even out the differences between the costs of the operations. By telescoping series, it disappears from the final calculations.

22. *In the big-O notation, what is the cost of splaying amortized over a sequence of retrievals and insertions? Why is this surprising?*

If the tree never has more than n nodes, then insertion or retrieval with splaying can always be done in time $O(\log n)$, amortized over a long sequence of operations. This result is surprising because the tree may begin as, or degenerate to, a chain or other tree that is highly unbalanced, so an individual insertion or removal may require n operations.

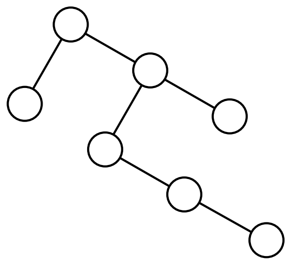
Multiway Trees 11

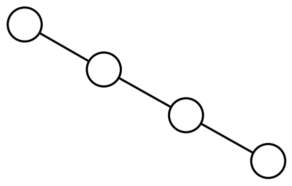
11.1 ORCHARDS, TREES, AND BINARY TREES

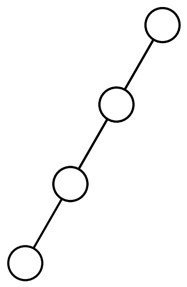
Exercises 11.1

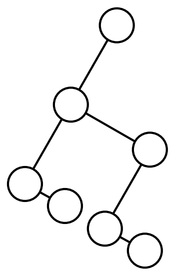
E1. *Convert each of the orchards shown in the text into a binary tree.*

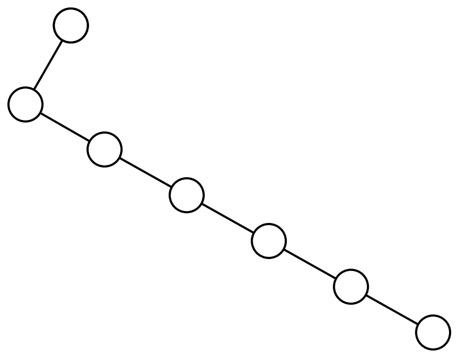
Answer

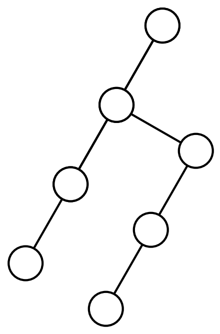

(a)

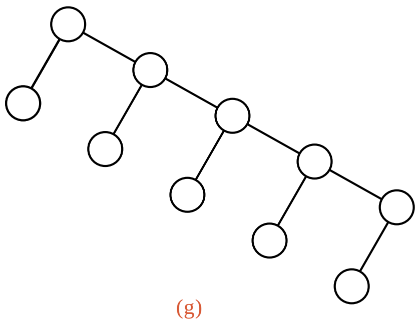

(b)

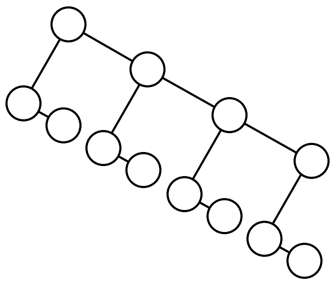

(c)


(d)


(e)


(f)

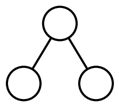

(g)

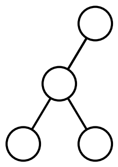

(h)

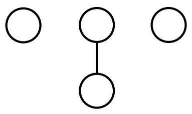
E2. *Convert each of the binary trees shown in the text into an orchard.*

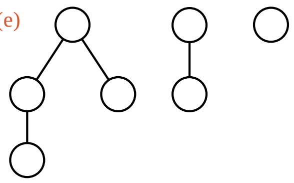
Answer

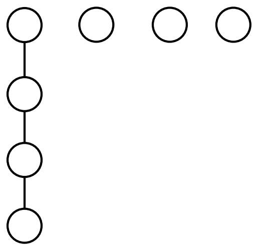
(a)

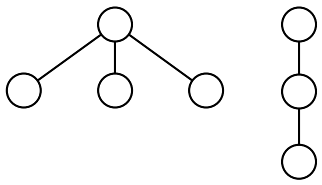

(b)

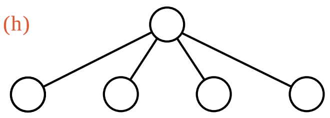

(c)

(d)


(e)


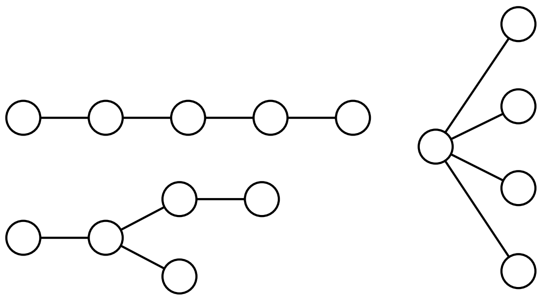
(f)


(g)


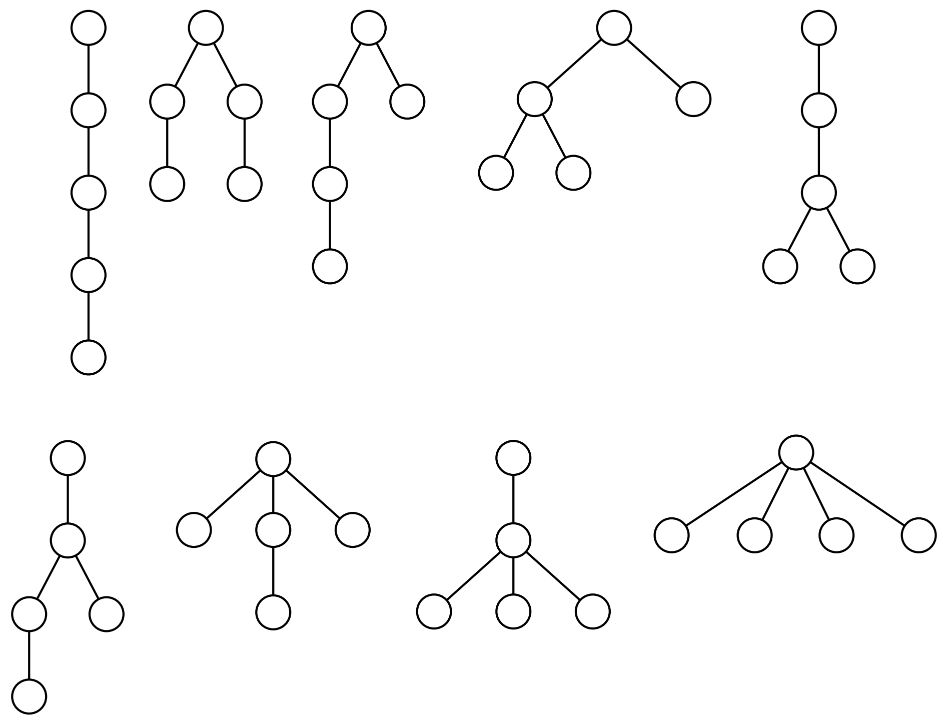
(h)


E3. *Draw all the* **(a)** *free trees,* **(b)** *rooted trees, and* **(c)** *ordered trees with five vertices.*

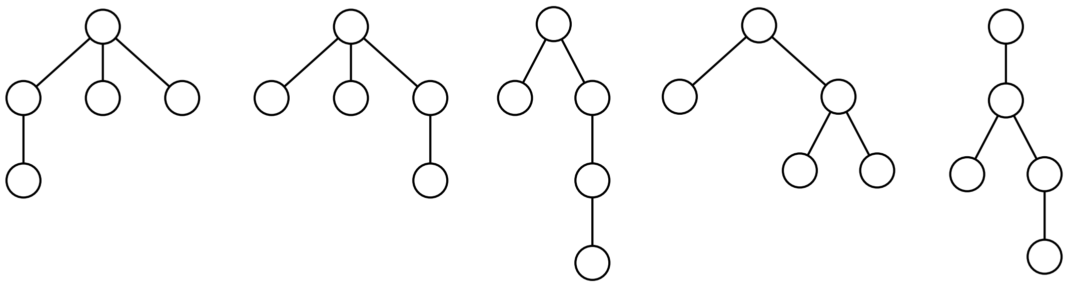
Answer **(a)** The free trees can be drawn with any orientation:



(b)



(c) The rooted trees shown in (b) can be regarded as ordered by distinguishing left from right. The additional ordered trees follow.



E4. *We can define the* ***preorder traversal*** *of an orchard as follows: If the orchard is empty, do nothing. Otherwise, first visit the root of the first tree, then traverse the orchard of subtrees of the first tree in preorder, and then traverse the orchard of remaining trees in preorder. Prove that preorder traversal of an orchard and preorder traversal of the corresponding binary tree will visit the vertices in the same order.*

Answer We shall use proof by induction on the number of vertices. If the orchard is empty, so is the corresponding binary tree, and both traversals do nothing. This is the initial case. A nonempty orchard has the form $O = (\{v, O_1\}, O_2)$ and corresponds to the binary tree $f(O) = [v, f(O_1), f(O_2)]$ where O_i corresponds to $f(O_i)$, $i = 1, 2$. By induction hypothesis the preorder traversals of O_i and $f(O_i)$, $i = 1, 2$, visit all vertices in the same order. Preorder traversal of O first visits v, then the vertices in O_1, then the vertices in O_2. Preorder traversal of the binary tree $f(O)$ first visits v then the vertices in $f(O_1)$ and then the vertices in $f(O_2)$, so altogether the vertices are visited in the same order.

E5. *We can define the* ***inorder traversal*** *of an orchard as follows: If the orchard is empty, do nothing. Otherwise, first traverse the orchard of subtrees of the first tree's root in inorder, then visit the root of the first tree, and then traverse the orchard of remaining subtrees in inorder. Prove that inorder traversal of an orchard and inorder traversal of the corresponding binary tree will visit the vertices in the same order.*

Answer The proof is similar to that for the previous exercise except that the vertices in O_1 (and $f(O_1)$, correspondingly) are visited first, then v, then the vertices in O_2 (and $f(O_2)$).

E6. *Describe a way of traversing an orchard that will visit the vertices in the same order as postorder traversal of the corresponding binary tree. Prove that your traversal method visits the vertices in the correct order.*

Answer Postorder traversal of an orchard first traverses the orchard of subtrees of the first tree in postorder, then traverses the orchard of remaining trees in postorder, and finally visits the root of the first tree. The proof is similar to that for Exercise E4 except that v is visited last.

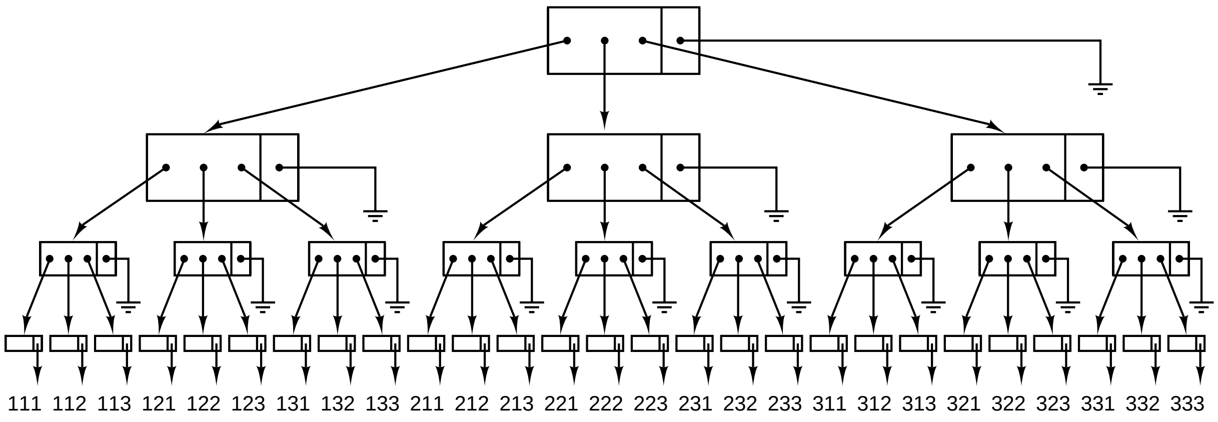
11.2 LEXICOGRAPHIC SEARCH TREES: TRIES

Exercises 11.2

E1. *Draw the tries constructed from each of the following sets of keys.*

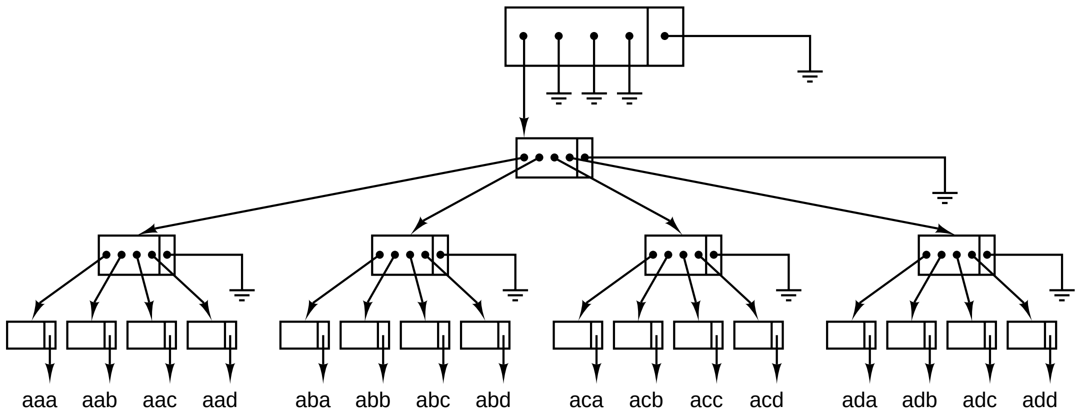
(a) *All three-digit integers containing only 1, 2, 3 (in decimal representation).*

Answer



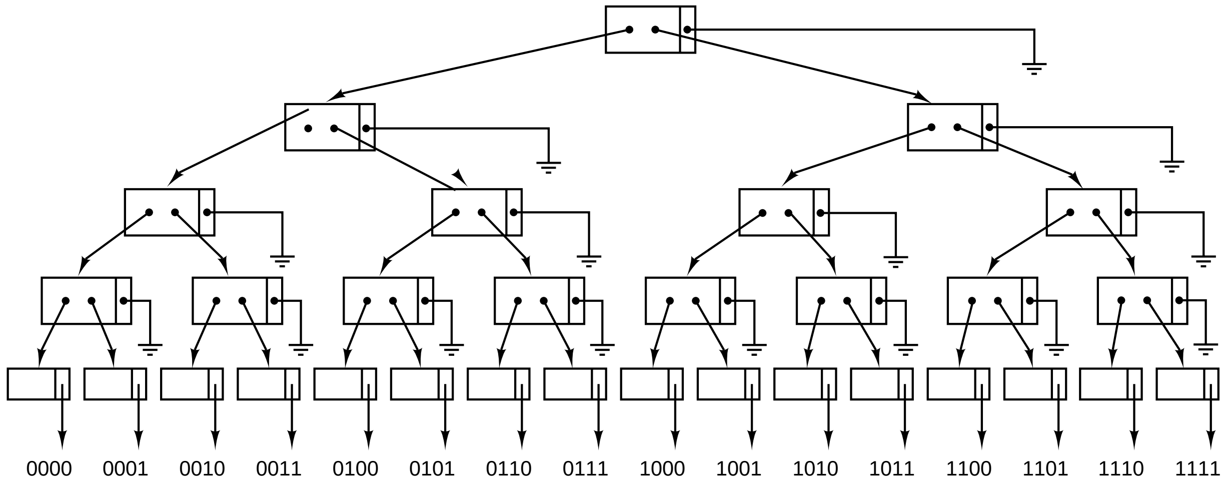
(b) *All three-letter sequences built from* a, b, c, d *where the first letter is* a.

Answer


aaa aab aac aad aba abb abc abd aca acb acc acd ada adb adc add

(c) *All four-digit binary integers (built from 0 and 1).*

Answer

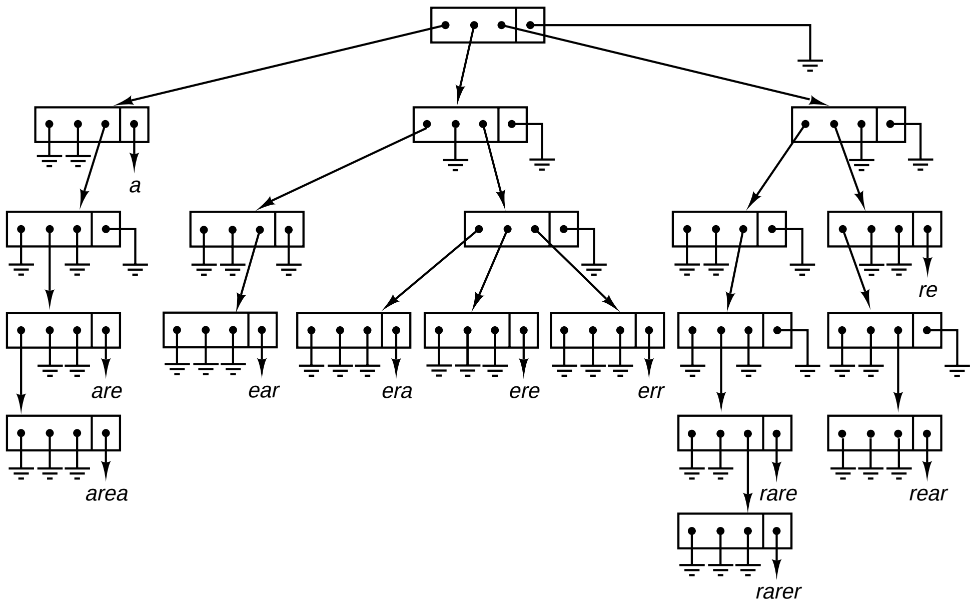


(d) *The words*

a ear re rare area are ere era rarer rear err

built from the letters a, e, r.

Answer

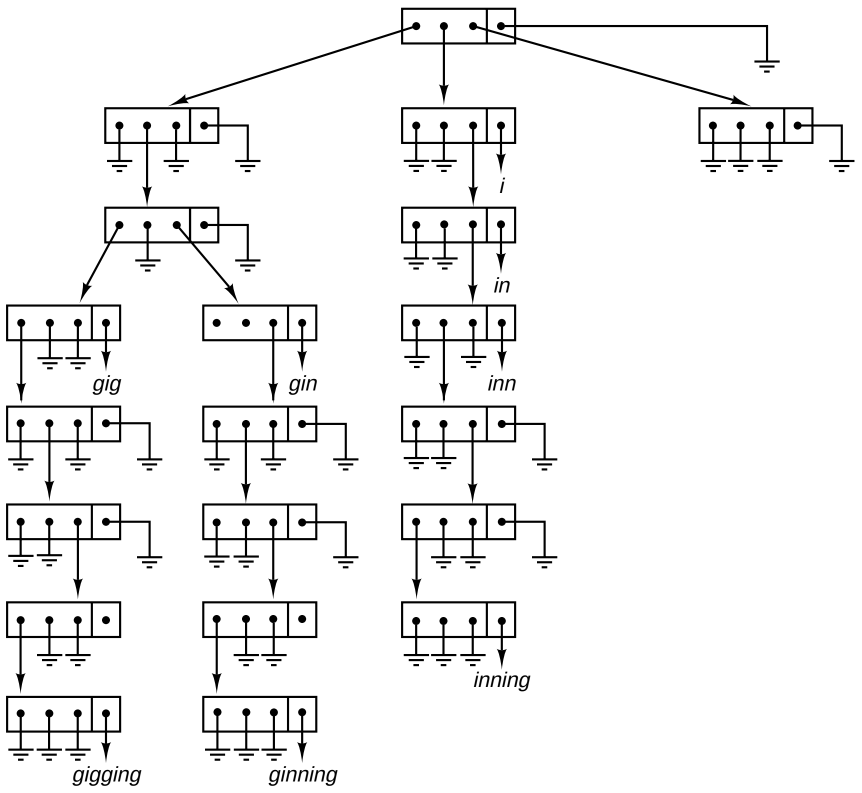


(e) *The words*

gig i inn gin in inning gigging ginning

built from the letters g, i, n.

Answer

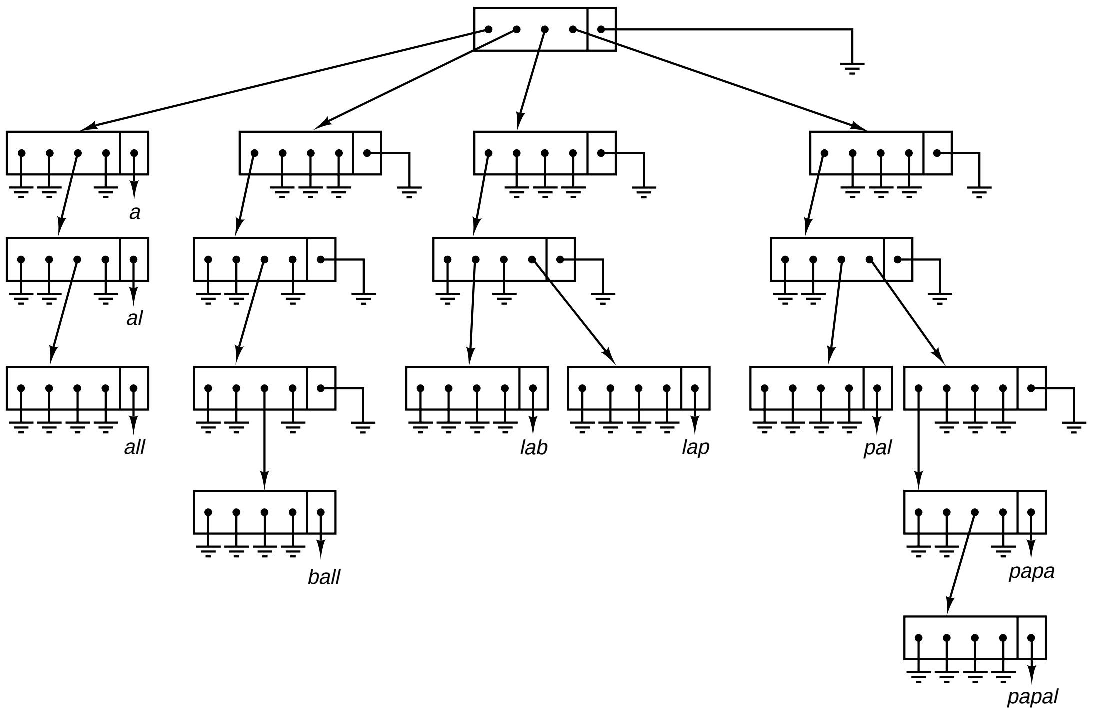


(f) *The words*

pal lap a papa al papal all ball lab

built from the letters a, b, l, p.

Answer



E2. *Write a method that will traverse a trie and print out all its words in alphabetical order.*

Answer We can implement a general inorder traversal method that will access its words in alphabetical order as follows.

```
void recursive_inorder(Trie_node *root, void (*visit)(Record *))
/* Post: The subtrie referenced by root is traversed, in order. The operation *visit is applied to
         all records of the trie. */
{
  if (root != NULL) {
    (*visit)(root->data);
    for (int i = 0; i < num_chars; i++) recursive_inorder(root->branch[i], visit);
  }
}

void Trie :: inorder(void (*visit)(Record *))
/* Post: The Trie is traversed, in order, and the operation *visit is applied to all records of the
         trie. */
{
  recursive_inorder(root, visit);
}
```

In order to use it to print entries, we pass a pointer to the following function as an argument for the inorder traversal.

```
void write_entry(Record *x)
{
  if (x != NULL) {
    for (int i = 0; x->key_letter(i) != ' '; i++) cout << x->key_letter(i);
    cout << "\n";
  }
}
```

E3. *Write a method that will traverse a trie and print out all its words, with the order determined first by the length of the word, with shorter words first, and, second, by alphabetical order for words of the same length.*

Answer We replace the traversal from the previous exercise with the following depth first traversal. We make use of a Queue structure, which we assume has been arranged to hold objects of type Trie_node *.

```
void Trie :: depth_traverse(void (*visit)(Record *))
/* Post: The Trie is traversed, in order of depth. The operation *visit is applied to all records of
         the trie. */
{
  if (root != NULL) {
    Queue waiting;
    waiting.append(root);
    do {
      Trie_node *sub_root;
      waiting.retrieve(sub_root);
      (*visit)(sub_root->data);
      for (int i = 0; i < num_chars; i++)
        if (sub_root->branch[i] != NULL)
         waiting.append(sub_root->branch[i]);
      waiting.serve();
    } while (!waiting.empty());
  }
}
```

E4. *Write a method that will delete a word from a trie.*

Answer

```
Error_code Trie :: remove(const Key &target)
/* Post: If an entry with Key target is in the Trie, it is removed. Otherwise, a code of not_present
         is returned.
   Uses: Methods of classes Record and Trie_node. */
```

```
{
  if (root == NULL) return not_present;
  int position = 0;                        // indexes letters of new_entry
  char next_char;
  Trie_node *location = root;              // moves through the Trie
  while (location != NULL &&
        (next_char = target.key_letter(position)) != ' ') {
    int next_position = alphabetic_order(next_char);
    if (location->branch[next_position] == NULL) return not_present;
    location = location->branch[next_position];
    position++;
  }
  if (location->data == NULL) return not_present;
  else {
     delete location->data;
     location->data = NULL;
  }
  return success;
}
```

Programming Project 11.2

P1. *Construct a menu-driven demonstration program for tries. The keys should be words constructed from the 26 lowercase letters, up to 8 characters long. The only information that should be kept in a record (apart from the key) is a serial number indicating when the word was inserted.*

Answer Main Trie demonstration program:

```
#include "../../c/utility.h"

void help()
/*
PRE:  None.
POST: Instructions for the trie operations have been printed.
*/
{
     cout << "Legal Commands are:\n";
     cout << "\t[#]size      [P]rint     [I]nsert  \n"
          << "\t[R]emove     [D]epth first traversal \n"
          << "\t[C]lear      [H]elp      [Q]uit " << endl;
}

#include   "../../6/linklist/list.h"
#include   "../../6/linklist/list.cpp"
#include <string.h>

#include   "../../6/strings/string.h"
#include   "../../6/strings/string.cpp"

#include "../../9/radix/key.h"
#include "../../9/radix/key.cpp"

#include "record.h"
#include "record.cpp"

#include "../trie/trienode.h"
#include "../trie/trienode.cpp"

#include "../2e3e4/trie.h"
#include "../trie/trie.cpp"
```

```
typedef Trie_node * Queue_entry;
#include "../../3/queue/queue.h"
#include "../../3/queue/queue.cpp"

#include "../2e3e4/e3.cpp"
#include "../2e3e4/e4.cpp"

void write_entry(Record *x)
/*
Post: The key and serial number are printed.
*/

{
   if (x != NULL) {
      cout << "Serial number: " << x->serial() << " ";
      if (x->serial() < 10) cout << " ";
      if (x->serial() < 100) cout << " ";
      if (x->serial() < 1000) cout << " ";
      for (int i = 0; x->key_letter(i) != ' '; i++)
          cout << x->key_letter(i);
      cout << endl;
   }
}

Key get_key()
/*
Post: returns a user specified key
*/

{
   char c;
   int position;
   int key_size = 10;
   char *k = new char[key_size];
   for (position = 0; position < key_size; position++) k[position] = ' ';
   position = 0;

   do {
      cin.get(c);
   } while (!(c >= 'a' && c <= 'z') && !(c >= 'A' && c <= 'Z'));

   while (c != '\n') {
      if (position < key_size) k[position] = c;
      position++;
      cin.get(c);
   }
   Key x(k);
   return x;
}

char get_command()
/*
Post: obtains a trie demo function from user
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if (c == '#' || c == 'i' || c == 'c' || c == 'r' ||
         c == 'h' || c == 'q' || c == 'p' || c == 'd') {
         return c;
      }
      cout << "Please enter a valid command or H for help:";
      help();
   }
}

int do_command(char c, Trie &test_tree)
/*
Post: runs the trie demo function specified by the parameter c
*/

{
   switch (c) {
   case 'h':
      help();
      break;

   case 'c':
      test_tree.clear();
      cout << "Tree is cleared.\n";
      break;

   case 'p':
      test_tree.inorder(write_entry);
      break;

   case '#':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'q':
      cout << "Tree demonstration finished.\n";
      return 0;

   case 'd':
      test_tree.depth_traverse(write_entry);
      break;

   case 'r':
   case 'i':
      cout << "Enter new (character string) key to remove or insert:";
      Key x(get_key());
      if (c == 'r') {
         test_tree.remove(x);
      }
      else {
         Record y(x);
         test_tree.insert(y);
      }
      break;
   }
   return 1;
}
```

```
main()
/*
Post: runs the trie demo program for P1 of Section 11.2
*/
{
   Trie test_tree;
   help();
   while (do_command(get_command(), test_tree));
}
```

Class definition for Records with serial numbers:

```
class Record {
public:
   char key_letter(int position) const;
   Record();                       //  default constructor
   operator Key() const;           //  cast to Key
   //  Add other methods and data members for the class.

   Record (const Key &a_name);   //  conversion to Record
   int serial();

private:
   Key name;
   int serial_number;
   static int record_counter;
};
```

Implementation:

```
int Record::record_counter = 0;

int Record::serial()
{
   return serial_number;
}

Record::Record() : name()
{
}

char Record::key_letter(int position) const
{
   return name.key_letter(position);
}

Record::operator Key() const
{
   return name;
}

Record::Record (const Key &a_name)
{
   name = a_name;
   serial_number = record_counter++;
}

int alphabetic_order(char c)
/*
Post: The function returns the alphabetic position of character
      c, or it returns 0 if the character is blank.
*/
```

```
{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c - 'a' + 1;
   if ('A' <= c && c <= 'Z') return c - 'A' + 1;
   return 27;
}
```

The Key class implementation is identical to that used in Radix sorting from Chapter 9.

11.3 EXTERNAL SEARCHING: B-TREES

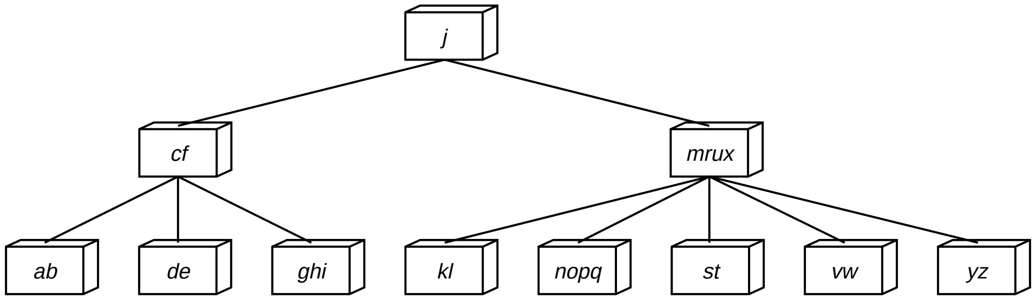
Exercises 11.3

E1. *Insert the six remaining letters of the alphabet in the order*

z, v, o, q, w, y

into the final B-tree of Figure 11.10 (page 539).

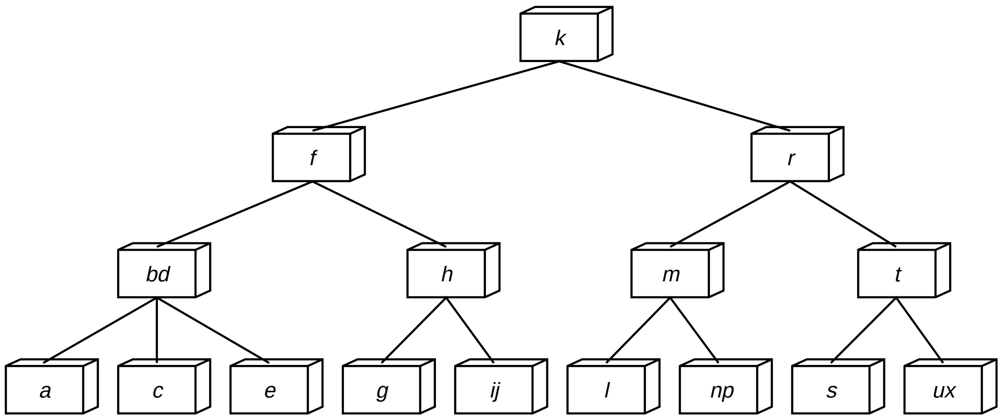
Answer



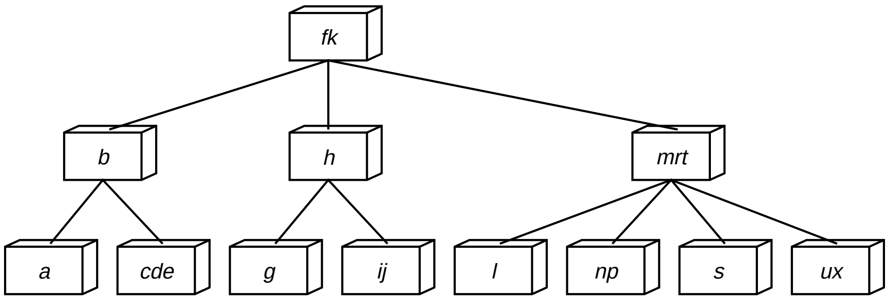
E2. *Insert the following entries, in the order stated, into an initially empty B-tree of order* **(a)** *3,* **(b)** *4,* **(c)** *7:*

a g f b k d h m j e s i r x c l n t u p

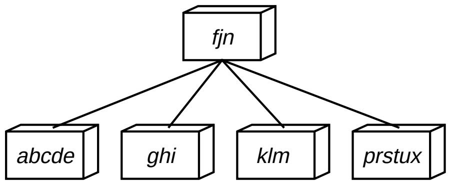
Answer



(a)



(b)



(c)

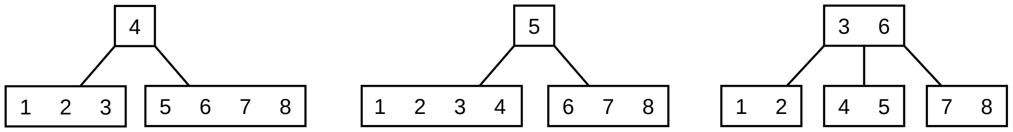
E3. *What is the smallest number of entries that, when inserted in an appropriate order, will force a B-tree of order 5 to have height 3 (that is, 3 levels)?*

Answer The sparsest possible B-tree of order 5 with three levels will have one entry in the root, two nodes on the next level down, each with two entries and each of which must have three children on the lowest level, each of which must have at least two entries. Hence the total number of entries is

$$1 \times 1 + 2 \times 2 + 6 \times 2 = 17.$$

E4. *Draw all the B-trees of order 5 (between 2 and 4 keys per node) that can be constructed from the keys 1, 2, 3, 4, 5, 6, 7, and 8.*

Answer There are too many keys to fit in one node (the root) and too few to make up a tree with three levels. Hence all the trees have two levels. They are:



E5. *If a key in a B-tree is not in a leaf, prove that both its immediate predecessor and immediate successor (under the natural order) are in leaves.*

Answer The immediate successor is found by taking the right child (assuming it is not in a leaf) and then taking the leftmost child of each node as long as it is not NULL. This process ends in a leaf, and the immediate successor of the given key is the first key in that leaf. Similarly, the immediate predecessor is found by taking the left child and then right children as long as possible, and the predecessor is then the rightmost key in a leaf.

E6. *Suppose that disk hardware allows us to choose the size of a disk record any way we wish, but that the time it takes to read a record from the disk is $a + bd$, where a and b are constants and d is the order of the B-tree. (One node in the B-tree is stored as one record on the disk.) Let n be the number of entries in the B-tree. Assume for simplicity that all the nodes in the B-tree are full (each node contains $d - 1$ entries).*

disk accesses

(a) *Explain why the time needed to do a typical B-tree operation (searching or insertion, for example) is approximately $(a + bd)\log_d n$.*

Answer The height of the B-tree is about $\log_d n$ and so the operation will require about this many disk reads, each of which takes time $a + bd$, giving the total indicated. The time required for computation in high-speed memory will likely be trivial compared to the time needed for disk reads. Hence the total time required is essentially that needed for the disk reads.

(b) *Show that the time needed is minimized when the value of d satisfies $d(\ln d - 1) = a/b$. (Note that the answer does not depend on the number n of entries in the B-tree.) [Hint: For fixed a, b, and n, the time is a function of d: $f(d) = (a + bd)\log_d n$. Note that $\log_d n = (\ln n)/(\ln d)$. To find the minimum, calculate the derivative $f'(d)$ and set it to 0.]*

Answer Since

$$f(d) = \frac{(a + bd)\ln n}{\ln d},$$

the quotient rule for derivatives gives

$$f'(d) = \frac{(b \ln n)(\ln d) - (a + bd)(\ln n)(1/d)}{(\ln d)^2}.$$

Setting this to 0 and clearing fractions gives $b(\ln n)(\ln d)d = (a + bd)(\ln n)$, which simplifies to $bd(\ln d - 1) = a$, or $d(\ln d - 1) = a/b$.

(c) *Suppose a is 20 milliseconds and b is 0.1 millisecond. (The records are very short.) Find the value of d (approximately) that minimizes the time.*

Answer Let $g(d) = d(\ln d - 1)$. Since $a/b = 200$, we wish to find d such that $g(d) = 200$. Trial and error on a hand calculator yields $g(63) \approx 198.0$ and $g(64) \approx 202.2$, so the answer is about 63 or 64.

(d) *Suppose a is 20 milliseconds and b is 10 milliseconds. (The records are longer.) Find the value of d (approximately) that minimizes the time.*

Answer Now $a/b = 2$. Since $g(4) \approx 1.54$ and $g(5) \approx 3.04$, we want a B-tree of about order 4 or 5.

E7. *Write a method that will traverse a linked B-tree, visiting all its entries in order of keys (smaller keys first).*

traversal

Answer
```
template <class Record, int order>
void B_tree<Record, order> :: recursive_inorder(
                B_node<Record, order> *sub_root, void (*visit)(Record &))
/* Post: The subtree referenced by sub_root is traversed, in order. The operation *visit is applied
         to all records of the tree. */
{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->branch[0], visit);
      for (int i = 0; i < sub_root->count; i++) {
         (*visit)(sub_root->data[i]);
         recursive_inorder(sub_root->branch[i + 1], visit);
      }
   }
}
template <class Record, int order>
void B_tree<Record, order> :: inorder(void (*visit)(Record &))
/* Post: The B_tree is traversed, in order, and the operation *visit is applied to all records of the
         tree. */
{
    recursive_inorder(root, visit);
}
```

E8. *Define **preorder** traversal of a B-tree recursively to mean visiting all the entries in the root node first, then traversing all the subtrees, from left to right, in preorder. Write a method that will traverse a B-tree in preorder.*

Answer
```
template <class Record, int order>
void B_tree<Record, order> :: recursive_preorder(
                B_node<Record, order> *sub_root, void (*visit)(Record &))
/* Post: The subtree referenced by sub_root is traversed, in prefix order. The operation *visit is
         applied to all records of the tree. */
{
   int i;
   if (sub_root != NULL) {
      for (i = 0; i < sub_root->count; i++)
         (*visit)(sub_root->data[i]);
      for (i = 0; i <= sub_root->count; i++)
         recursive_preorder(sub_root->branch[i], visit);
   }
}
```

TOC
Index
Help

```
template <class Record, int order>
void B_tree<Record, order> :: preorder(void (*visit)(Record &))
/* Post: The B_tree is traversed, in prefix order, and the operation *visit is applied to all records
         of the tree. */
{
   recursive_preorder(root, visit);
}
```

E9. *Define **postorder** traversal of a B-tree recursively to mean first traversing all the subtrees of the root, from left to right, in postorder, then visiting all the entries in the root. Write a method that will traverse a B-tree in postorder.*

Answer

```
template <class Record, int order>
void B_tree<Record, order> :: recursive_postorder(
                 B_node<Record, order> *sub_root, void (*visit)(Record &))
/* Post: The subtree referenced by sub_root is traversed, in postfix order. The operation *visit
         is applied to all records of the tree. */
{
   int i;
   if (sub_root != NULL) {
      for (i = 0; i <= sub_root->count; i++)
         recursive_postorder(sub_root->branch[i], visit);
      for (i = 0; i < sub_root->count; i++)
         (*visit)(sub_root->data[i]);
   }
}
template <class Record, int order>
void B_tree<Record, order> :: postorder(void (*visit)(Record &))
/* Post: The B_tree is traversed, in postfix order, and the operation *visit is applied to all records
         of the tree. */
{
   recursive_postorder(root, visit);
}
```

E10. *Remove the tail recursion from the function* recursive_search_tree *and integrate it into a nonrecursive version of* search_tree.

Answer

```
template <class Record, int order>
Error_code B_tree<Record, order> :: search_tree(Record &target)
/* Post: If there is an entry in the B-tree whose key matches that in target, the parameter target is
         replaced by the corresponding Record from the B-tree and a code of success is returned.
         Otherwise a code of not_present is returned.
   Uses: recursive_search_tree */
{
   B_node<Record, order> *sub_root = root;
   while (sub_root != NULL) {
      int position;
      if (search_node(sub_root, target, position) == not_present)
         sub_root = sub_root->branch[position];
      else {
         target = sub_root->data[position];
         return success;
      }
   }
   return not_present;
}
```

E11. *Rewrite the function* search_node *to use binary search.*

Answer

```
template <class Record, int order>
Error_code B_tree<Record, order> :: search_node(
   B_node<Record, order> *current, const Record &target, int &position)
/* Pre:   current points to a node of a B_tree.
   Post:  If the Key of target is found in *current, then a code of success is returned, the parameter
          position is set to the index of target, and the corresponding Record is copied to target.
          Otherwise, a code of not_present is returned, and position is set to the branch index on
          which to continue the search.
   Uses:  Methods of class Record. */
{
   if (target < current->data[0]) {
      position = 0;
      return not_present;
   }
   int low = 0, mid, high = current->count;
   while (high > low) {
      mid = (high + low)/2;
      if (current->data[mid] < target) low = mid + 1;
      else high = mid;
   }
   position = high;
   if (target == current->data[position]) return success;
   return not_present;
}
```

E12. *A* ***B*-tree*** *is a B-tree in which every node, except possibly the root, is at least two-thirds full, rather than half full. Insertion into a B*-tree moves entries between sibling nodes (as done during deletion) as needed, thereby delaying splitting a node until two sibling nodes are completely full. These two nodes can then be split into three, each of which will be at least two-thirds full.*

B-tree*

(a) *Specify the changes needed to the insertion algorithm so that it will maintain the properties of a B*-tree.*

Answer The constant min becomes $\lceil \frac{2}{3}m \rceil - 1$, where m is the order of the B*-tree. To permit moving entries between adjacent nodes, the function push_down requires an additional parameter parent, a pointer to the parent (if any) of the node *p. When push_down is called from insert the new parameter has value NULL to indicate that the root has no siblings. In the recursive call to push_down the new parameter has value p. Near the end of push_down, before a node is split, the function should check (if the parent pointer is not NULL) the sibling nodes on both sides of *p. If one of them is not full, move_left or move_right should be used to shift entries to allow the insertion. When splitting a node becomes necessary, function split needs to be given two full nodes and a pointer to the parent (along with its previous parameters). Function split must be completely rewritten to move entries among the two full nodes, the new third node, and the parent node (if any) so that all the nodes are no less than two thirds full.

(b) *Specify the changes needed to the deletion algorithm so that it will maintain the properties of a B*-tree.*

Answer Changes are required only in the methods restore and combine. The method restore should examine sibling nodes as far as two positions away from position, looking for a node with more than min entries, and then invoke move_left or move_right to move an entry into the node with too few entries. The method combine must be given three adjacent sibling nodes (or a convention as to which of such a sequence is its parameter) and then it must be rewritten to move entries as necessary to combine the three into two as well as removing one entry from and changing one entry in the parent node.

(c) *Discuss the relative advantages and disadvantages of B*-trees compared to ordinary B-trees.*

Answer The major advantage of B*-trees is the more efficient utilization of storage, since each node is at least two-thirds rather than half full. Hence, with the same number of entries, the tree will probably be smaller, its height may be less, and retrieval will usually be faster for a B*-tree. When insertion or deletion requires restructuring the tree, however, more work must be done for a B*-tree. Since, moreover, there is less flexibility in the number of keys per node, restructuring is likely to be required more often in a B*-tree. Hence B*-trees are likely to be slower than ordinary B-trees for insertion and deletion. The programming of B*-trees is also more complicated. The choice between B-trees and B*-trees therefore depends on the relative importance of fast retrieval, storage utilization, and the number of insertions and deletions, coupled with programming effort.

Programming Projects 11.3

P1. *Combine all the functions of this section into a menu-driven demonstration program for B-trees. If you have designed the demonstration program for binary search trees from Section 10.2, Project P2 (page 460) with sufficient care, you should be able to make a direct replacement of one package of operations by another.*

Answer The following program uses integer entries in a B-tree of order 4.

```
#include "../../c/utility.h"

void help()
/*
PRE:  None.
POST: Instructions for the B-tree operations have been printed.
*/

{
     cout << "Legal commands are: \n";
     cout << "\t[P]rint     [I]nsert    [F]ind\n"
          << "\tpr[E]order  i[N]order   po[S]torder\n"
          << "\t[R]emove    [H]elp      [Q]uit " << endl;
}

#include "../btree/bnode.h"
#include "../3e7__e11/btree.h"
#include "../3e7__e11/bnode.cpp"
#include "../3e7__e11/btree.cpp"
#include "../3e7__e11/e7.cpp"
#include "../3e7__e11/e8.cpp"
#include "../3e7__e11/e9.cpp"
#include "../3e7__e11/e10.cpp"
#include "../3e7__e11/e11.cpp"

void write_entry(int &x)
{
   cout << x << " ";
}

int get_int()
/*
POST: Returns a user specified integer
*/
```

```
{
   char c;
   int ans = 0, sign = 1;
   do {
      cin.get(c);
      if (c == '-') sign = -1;
   } while (c < '0' || c > '9');
   while (c >= '0' && c <= '9') {
      ans = 10 * ans + c - '0';
      cin.get(c);
   }
   return ans * sign;
}

char get_command()
/*
POST: Returns a user specified character command
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if (c == 'r' || c == 'i' || c == 'c' || c == 'f' || c == 'e' ||
          c == 'n' || c == 's' || c == 'h' || c == 'q' || c == 'p') {
         return c;
      }
      cout << "Please enter a valid command or H for help:" << endl;
      help();
   }
}

int do_command(char c, B_tree<int, 4> &test_tree)
/*
POST: Performs the command specified by the parameter c
*/

{
   int x;
   switch (c) {
   case 'h':
      help();
      break;
   case 'f':
      cout << "Enter new integer to find:";
      x = get_int();
      {
        Error_code e = test_tree.search_tree(x);
        if (e == not_present)
           cout <<"That entry is not in the B-tree." << endl;
        else cout <<"The entry is present in the B-tree." << endl;
      }
      break;

   case 'r':
      cout << "Enter integer to remove:" << flush;
      x = get_int();
      if (test_tree.remove(x) != success) cout << "Not found!\n";
      break;

   case 'i':
      cout << "Enter new integer to insert:";
      x = get_int();
      test_tree.insert(x);
      break;

   case 'p':
      test_tree.print();
      break;

   case 'e':
      test_tree.preorder(write_entry);
      cout << endl;
      break;

   case 'n':
      test_tree.inorder(write_entry);
      cout << endl;
      break;

   case 's':
      test_tree.postorder(write_entry);
      cout << endl;
      break;

   case 'q':
      cout << "B-tree demonstration finished.\n";
      return 0;
   }
   return 1;
}

main()
/*
Post: Runs the B-Tree demonstration program of
      Project P1 of Section 11.3
*/

{
   cout << "\n Demonstration Program for order 4 B-trees"
        << " with integer entries." << endl;
   B_tree<int, 4> test_tree;
   help();
   while (do_command(get_command(), test_tree));
}
```

B-tree class:

```
template <class Record, int order>
class B_tree {
public:  //  Add public methods.
   B_tree();
   Error_code search_tree(Record &target);
   Error_code insert(const Record &new_node);
   Error_code remove(const Record &target);
   ~B_tree();
   void print();

private:
   void recursive_print(B_node<Record, order> *, int);
   Error_code recursive_search_tree(B_node<Record, order> *, Record &);
   Error_code search_node(B_node<Record, order> *, const Record &, int &);
   Error_code push_down(B_node<Record, order> *, const Record &, Record &,
          B_node<Record, order> *&);

   void push_in(B_node<Record, order> *, const Record &,
             B_node<Record, order> *, int);

   void split_node(B_node<Record, order> *, const Record &extra_entry,
           B_node<Record, order> *, int,
           B_node<Record, order> *&, Record &);

   Error_code recursive_remove(B_node<Record, order> *, const Record &);

   void copy_in_predecessor(B_node<Record, order> *, int);
   void remove_data(B_node<Record, order> *, int);
   void restore(B_node<Record, order> *, int);
   void move_left(B_node<Record, order> *, int);
   void move_right(B_node<Record, order> *, int);
   void combine(B_node<Record, order> *, int);

private: //  data members
   B_node<Record, order> *root;
         //  Add private auxiliary functions here.
};
```

Implementation:

```
template <class Record, int order>
void B_tree<Record, order>::print()
{
   recursive_print(root, 0);
}

template <class Record, int order>
B_tree<Record, order>::B_tree()
{
   root = NULL;
}

template <class Record, int order>
B_tree<Record, order>::~B_tree()
{
//   This needs to be written some time!
}
```

TOC
Index
Help

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_tree(Record &target)
/*
Post: If there is an entry in the B-tree whose key matches that in target,
      the parameter target is replaced by the corresponding Record from
      the B-tree and a code of success is returned.  Otherwise
      a code of not_present is returned.
Uses: recursive_search_tree
*/

{
   return recursive_search_tree(root, target);
}

template <class Record, int order>
Error_code B_tree<Record, order>::insert(const Record &new_entry)
/*
Post: If the Key of new_entry is already in the B_tree,
      a code of duplicate_error is returned.
      Otherwise, a code of success is returned and the Record new_entry
      is inserted into the B-tree in such a way that the properties of a
      B-tree are preserved.
Uses: Methods of struct B_node and the auxiliary function push_down.
*/

{
   Record median;
   B_node<Record, order> *right_branch, *new_root;
   Error_code result = push_down(root, new_entry, median, right_branch);

   if (result == overflow) { // The whole tree grows in height.
                             // Make a brand new root for the whole B-tree.
      new_root = new B_node<Record, order>;
      new_root->count = 1;
      new_root->data[0] = median;
      new_root->branch[0] = root;
      new_root->branch[1] = right_branch;
      root = new_root;
      result = success;
   }
   return result;
}

template <class Record, int order>
Error_code B_tree<Record, order>::remove(const Record &target)
/*
Post: If a Record with Key matching that of target belongs to the
      B_tree, a code of success is returned and the corresponding node
      is removed from the B-tree.  Otherwise, a code of not_present
      is returned.
Uses: Function recursive_remove
*/
```

```
{
   Error_code result;
   result = recursive_remove(root, target);
   if (root != NULL && root->count == 0) {  //  root is now empty.
      B_node<Record, order> *old_root = root;
      root = root->branch[0];
      delete old_root;
   }
   return result;
}
```

Node definition:

```
template <class Record, int order>
struct B_node {
//  data members:
   int count;
   Record data[order - 1];
   B_node<Record, order> *branch[order];
//  constructor:
   B_node();
};
```

Implementation:

```
template <class Record, int order>
B_node<Record, order>::B_node()
{
   count = 0;
}

template <class Record, int order>
void B_tree<Record, order>::recursive_print(B_node<Record, order> *current,
                                             int indent)
{
   int i;
   if (current != NULL) {
      for (i = 0; i < indent; i++) cout << " ";
      cout << ": (";
      for (i = 0; i < current->count; i++) cout << current->data[i] << ",";
      cout << ")\n";
      for (i = 0; i <= current->count; i++)
         recursive_print(current->branch[i], indent + 1);
   }
}

template <class Record, int order>
Error_code B_tree<Record, order>::recursive_search_tree(
              B_node<Record, order> *current, Record &target)

/*
Pre:  current is either NULL or points to a subtree of the
      B_tree.
Post: If the Key of target is not in the subtree, a code of not_present
      is returned.
      Otherwise, a code of success is returned and
      target is set to the corresponding Record of
      the subtree.
Uses: recursive_search_tree recursively and search_node
*/
```

```
{
   Error_code result = not_present;
   int position;
   if (current != NULL) {
      result = search_node(current, target, position);
      if (result == not_present)
         result = recursive_search_tree(current->branch[position], target);
      else
         target = current->data[position];
   }
   return result;
}

template <class Record, int order>
Error_code B_tree<Record, order>::search_node(
   B_node<Record, order> *current, const Record &target, int &position)
/*
Pre:  current points to a node of a B_tree.
Post: If the Key of target is found in *current, then a code of
      success is returned, the parameter position is set to the index
      of target, and the corresponding Record is copied to
      target.  Otherwise, a code of not_present is returned, and
      position is set to the branch index on which to continue the search.
Uses: Methods of class Record.
*/

{
   position = 0;
   while (position < current->count && target > current->data[position])
      position++;         //  Perform a sequential search through the keys.
   if (position < current->count && target == current->data[position])
      return success;
   else
      return not_present;
}

template <class Record, int order>
Error_code B_tree<Record, order>::push_down(
                 B_node<Record, order> *current,
                 const Record &new_entry,
                 Record &median,
                 B_node<Record, order> *&right_branch)

/*
Pre:  current is either NULL or points to a node of a B_tree.
Post: If an entry with a Key matching that of new_entry is in the subtree
      to which current points, a code of duplicate_error is returned.
      Otherwise, new_entry is inserted into the subtree: If this causes the
      height of the subtree to grow, a code of overflow is returned, and the
      Record median is extracted to be reinserted higher in the B-tree,
      together with the subtree right_branch on its right.
      If the height does not grow, a code of success is returned.
Uses: Functions push_down (called recursively), search_node,
      split_node, and push_in.
*/
```

```
{
   Error_code result;
   int position;
   if (current == NULL) {
      //  Since we cannot insert in an empty tree, the recursion terminates.
      median = new_entry;
      right_branch = NULL;
      result = overflow;
   }

   else {   //   Search the current node.
      if (search_node(current, new_entry, position) == success)
         result = duplicate_error;

      else {
         Record extra_entry;
         B_node<Record, order> *extra_branch;
         result = push_down(current->branch[position], new_entry,
                            extra_entry, extra_branch);

         if (result == overflow) {
                   //  Record extra_entry now must be added to current
            if (current->count < order - 1) {
               result = success;
               push_in(current, extra_entry, extra_branch, position);
            }

            else split_node(current, extra_entry, extra_branch, position,
                            right_branch, median);
         // Record median and its right_branch will go up to a higher node.
         }
      }
   }
   return result;
}

template <class Record, int order>
void B_tree<Record, order>::push_in(B_node<Record, order> *current,
   const Record &entry, B_node<Record, order> *right_branch, int position)
/*
Pre:   current points to a node of a B_tree.
      The node *current is not full and
      entry belongs in *current at index position.
Post: entry has been inserted along with its right-hand branch
      right_branch into *current at index position.
*/

{
   for (int i = current->count; i > position; i--) {
                                    //  Shift all later data to the right.
      current->data[i] = current->data[i - 1];
      current->branch[i + 1] = current->branch[i];
   }
   current->data[position] = entry;
   current->branch[position + 1] = right_branch;
   current->count++;
}
```

```
template <class Record, int order>
void B_tree<Record, order>::split_node(
   B_node<Record, order> *current,    //  node to be split
   const Record &extra_entry,          //  new entry to insert
   B_node<Record, order> *extra_branch,//  subtree on right of extra_entry
   int position,                  //  index in node where extra_entry goes
   B_node<Record, order> *&right_half, // new node for right half of entries
   Record &median)                     //  median entry (in neither half)

/*
Pre:   current points to a node of a B_tree.
      The node *current is full, but if there were room, the record
      extra_entry with its right-hand pointer extra_branch would belong
      in *current at position position,
      0 <= position < order.
Post: The node *current with extra_entry and pointer extra_branch at
      position position are divided into nodes *current and *right_half
      separated by a Record median.
Uses: Methods of struct B_node, function push_in.
*/

{
   right_half = new B_node<Record, order>;
   int mid = order/2;  //  The entries from mid on will go to right_half.

   if (position <= mid) { // First case:  extra_entry belongs in left half.
      for (int i = mid; i < order - 1; i++) { // Move entries to right_half.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(current, extra_entry, extra_branch, position);
   }

   else {  //  Second case:  extra_entry belongs in right half.
      mid++;      //  Temporarily leave the median in left half.
      for (int i = mid; i < order - 1; i++) { // Move entries to right_half.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(right_half, extra_entry, extra_branch, position - mid);
   }

      median = current->data[current->count - 1];
                                          //  Remove median from left half.
      right_half->branch[0] = current->branch[current->count];
      current->count--;
}

template <class Record, int order>
Error_code B_tree<Record, order>::recursive_remove(
   B_node<Record, order> *current, const Record &target)
/*
Pre:   current is either NULL or
      points to the root node of a subtree of a B_tree.
```

```
Post: If a Record with Key matching that of target belongs to the subtree,
      a code of success is returned and the corresponding node is removed
      from the subtree so that the properties of a B-tree are maintained.
      Otherwise, a code of not_present is returned.
Uses: Functions search_node, copy_in_predecessor,
      recursive_remove (recursive-ly), remove_data, and restore.
*/

{
   Error_code result;
   int position;
   if (current == NULL) result = not_present;

   else {
      if (search_node(current, target, position) == success) {
                                          //  The target is in the current node.
         result = success;

         if (current->branch[position] != NULL) {     // not at a leaf node
            copy_in_predecessor(current, position);

            recursive_remove(current->branch[position],
                             current->data[position]);
         }
         else remove_data(current, position);    // Remove from a leaf node.
      }

      else result = recursive_remove(current->branch[position], target);
      if (current->branch[position] != NULL)
         if (current->branch[position]->count < (order - 1) / 2)
            restore(current, position);
   }
   return result;
}

template <class Record, int order>
void B_tree<Record, order>::copy_in_predecessor(
                       B_node<Record, order> *current, int position)

/*
Pre:   current points to a non-leaf node in a B-tree with an
      entry at position.
Post: This entry is replaced by its immediate predecessor
      under order of keys.
*/

{
   B_node<Record, order> *leaf = current->branch[position];
                                       //   First go left from the current entry.
   while (leaf->branch[leaf->count] != NULL)
      leaf = leaf->branch[leaf->count]; // Move as far right as possible.
   current->data[position] = leaf->data[leaf->count - 1];
}

template <class Record, int order>
void B_tree<Record, order>::remove_data(B_node<Record, order> *current,
                                        int position)

/*
Pre:   current points to a leaf node in a B-tree with an entry at position.
Post: This entry is removed from *current.

*/
```

TOC
Index
Help

```
{
   for (int i = position; i < current->count - 1; i++)
      current->data[i] = current->data[i + 1];
   current->count--;
}

template <class Record, int order>
void B_tree<Record, order>::restore(B_node<Record, order> *current,
                                    int position)
/*
Pre:   current points to a non-leaf node in a B-tree; the node to which
      current->branch[position] points has one too few entries.
Post: An entry is taken from elsewhere to restore the minimum number of
      entries in the node to which current->branch[position] points.
Uses: move_left, move_right, combine.
*/

{
   if (position == current->count)   //  case:  rightmost branch
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else
         combine(current, position);

   else if (position == 0)       //  case: leftmost branch
      if (current->branch[1]->count > (order - 1) / 2)
         move_left(current, 1);
      else
         combine(current, 1);

   else                        //  remaining cases: intermediate branches
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else if (current->branch[position + 1]->count > (order - 1) / 2)
         move_left(current, position + 1);
      else
         combine(current, position);
}

template <class Record, int order>
void B_tree<Record, order>::move_left(B_node<Record, order> *current,
                                      int position)
/*
Pre:  current points to a node in a B-tree with more than the minimum
      number of entries in branch position and one too few entries in branch
      position - 1.

Post: The leftmost entry from branch position has moved into
      current, which has sent an entry into the branch position - 1.
*/

{

   B_node<Record, order> *left_branch = current->branch[position - 1],
                         *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
                                            //  Take entry from the parent.
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   current->data[position - 1] = right_branch->data[0];
                          //   Add the right-hand entry to the parent.
   right_branch->count--;
   for (int i = 0; i < right_branch->count; i++) {
                          //  Move right-hand entries to fill the hole.
      right_branch->data[i] = right_branch->data[i + 1];
      right_branch->branch[i] = right_branch->branch[i + 1];
   }
   right_branch->branch[right_branch->count] =
      right_branch->branch[right_branch->count + 1];
}

template <class Record, int order>
void B_tree<Record, order>::move_right(B_node<Record, order> *current,
                                       int position)
/*
Pre:  current points to a node in a B-tree with more than the minimum
      number of entries in branch position and one too few entries
      in branch position + 1.

Post: The rightmost entry from branch position has moved into
      current, which has sent an entry into the branch position + 1.
*/

{

   B_node<Record, order> *right_branch = current->branch[position + 1],
                         *left_branch = current->branch[position];
   right_branch->branch[right_branch->count + 1] =
      right_branch->branch[right_branch->count];

   for (int i = right_branch->count ; i > 0; i--) {
                                          // Make room for new entry.
      right_branch->data[i] = right_branch->data[i - 1];
      right_branch->branch[i] = right_branch->branch[i - 1];
   }

   right_branch->count++;
   right_branch->data[0] = current->data[position];
                                         //  Take entry from parent.
   right_branch->branch[0] = left_branch->branch[left_branch->count--];
   current->data[position] = left_branch->data[left_branch->count];
}

template <class Record, int order>
void B_tree<Record, order>::combine(B_node<Record, order> *current,
                                    int position)
/*
Pre:  current points to a node in a B-tree with entries in the branches
      position and position - 1, with too few to move entries.
Post: The nodes at branches position - 1 and position have been combined
      into one node, which also includes the entry formerly in current
      at index  position - 1.
*/

{

   int i;
   B_node<Record, order> *left_branch = current->branch[position - 1],
                         *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   for (i = 0; i < right_branch->count; i++) {
      left_branch->data[left_branch->count] = right_branch->data[i];
      left_branch->branch[++left_branch->count] =
                                    right_branch->branch[i + 1];
   }

   current->count--;
   for (i = position - 1; i < current->count; i++) {
      current->data[i] = current->data[i + 1];
      current->branch[i + 1] = current->branch[i + 2];
   }
   delete right_branch;
}
```

P2. *Substitute the functions for B-tree retrieval and insertion into the information-retrieval project of Project P5 of Section 10.2 (page 461). Compare the performance of B-trees with binary search trees for various combinations of input text files and various orders of B-trees.*

Answer The tree used is a B-tree of order 4. Main program:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/doubly/list.h"
#include "../../6/doubly/list.cpp"
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "../btree/bnode.h"
#include "btree.h"
#include "bnode.cpp"
#include "btree.cpp"
#include "../3e7__e11/e7.cpp"
#include "../3e7__e11/e8.cpp"
#include "../3e7__e11/e9.cpp"
#include "../3e7__e11/e10.cpp"
#include "../3e7__e11/e11.cpp"

#include "../../10/2p5/key.h"
#include "../../10/2p5/key.cpp"
#include "../../10/2p5/record.h"
#include "../../10/2p5/record.cpp"

#include "auxil.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Name of an input file can be given as a command-line argument.
Post: The word storing project p6 has been performed.
*/

{
   write_method();
   B_tree<Record, 4> the_words;
```

TOC
Index
Help

```
   char infile[1000];
   char in_string[1000];
   char *in_word;
   Timer clock;
   do {
      int initial_comparisons = Key::comparisons;
      clock.reset();
      if (argc < 2) {
         cout << "What is the input data file: " << flush;
         cin >> infile;
      }
      else strcpy(infile, argv[1]);

      ifstream file_in(infile);   //  Declare and open the input stream.
      if (file_in == 0) {
         cout << "Can't open input file " << infile << endl;
         exit (1);
      }

      while (!file_in.eof()) {
        file_in >> in_string;
        int position = 0;
        bool end_string = false;
        while (!end_string) {
           in_word = extract_word(in_string, position, end_string);
           if (strlen(in_word) > 0) {
              int counter;
              String s(in_word);
              update(s, the_words, counter);
           }
        }
      }
      cout << "Elapsed time: " << clock.elapsed_time() << endl;
      cout << "Comparisons performed = "
           << Key::comparisons - initial_comparisons << endl;
      cout << "Do you want to print frequencies? ";
      if (user_says_yes()) print(the_words);
      cout << "Do you want to add another file of input? ";
   } while (user_says_yes());
}
```

Auxiliary functions to carry out the required operations:

```
void write_method()
/*
Post:  A short string identifying the abstract
       data type used for the structure is written.
*/

{
   cout << " Word frequency program:  order 4 B-Tree implementation";
   cout << endl;
}
```

```
void update(const String &word, B_tree<Record, 4> &structure,
               int &num_comps)
/*
Post:  If word was not already present in structure, then
       word has been inserted into structure and its frequency
       count is 1.  If word was already present in structure,
       then its frequency count has been increased by 1.  The
       variable parameter num_comps is set to the number of
       comparisons of words done.
*/

{
  int initial_comparisons = Key::comparisons;
  Record r((Key) word);

  if (structure.search_tree(r) == not_present)
     structure.insert(r);
  else {
     structure.remove(r);
     r.increment();
     structure.insert(r);
  }
  num_comps = Key::comparisons - initial_comparisons;
}

void wr_entry(Record &a_word)
{
   String s = ((Key) a_word).the_key();
   cout << s.c_str();
   for (int i = strlen(s.c_str()); i < 12; i++) cout << " ";
   cout << " : " << a_word.frequency() << endl;
}

void print(B_tree<Record, 4> &structure)
/*
Post:  All words in structure are printed at the terminal
       in alphabetical order together with their frequency counts.
*/
{
     structure.inorder(wr_entry);
}

char *extract_word(char *in_string, int &examine, bool &over)
{
   int ok = 0;
   while (in_string[examine] != '\0') {
     if ('a' <= in_string[examine] && in_string[examine] <= 'z')
       in_string[ok++] = in_string[examine++];

     else if ('A' <= in_string[examine] && in_string[examine] <= 'Z')
       in_string[ok++] = in_string[examine++] - 'A' + 'a';

     else if (('\'' == in_string[examine] || in_string[examine] == '-')
       && (
       ('a' <= in_string[examine + 1] && in_string[examine + 1] <= 'z') ||
       ('A' <= in_string[examine + 1] && in_string[examine + 1] <= 'Z')))
       in_string[ok++] = in_string[examine++];
     else break;
   }
   in_string[ok] = '\0';
   if (in_string[examine] == '\0') over = true;
   else examine++;
   return in_string;
}
```

A slightly modified B-tree implementation is provided by:

```
template <class Record, int order>
class B_tree {
public:  //   Add public methods.

   B_tree();
   Error_code search_tree(Record &target);
   Error_code insert(const Record &new_node);
   Error_code remove(const Record &target);
   ~B_tree();
void recursive_inorder(
    B_node<Record, order> *sub_root, void (*visit)(Record &));
void inorder(void (*visit)(Record &));
void recursive_preorder(
    B_node<Record, order> *sub_root, void (*visit)(Record &));
void preorder(void (*visit)(Record &));
void recursive_postorder(
    B_node<Record, order> *sub_root, void (*visit)(Record &));
void postorder(void (*visit)(Record &));

private:
   Error_code recursive_search_tree(B_node<Record, order> *, Record &);
   Error_code search_node(B_node<Record, order> *, const Record &, int &);
   Error_code push_down(B_node<Record, order> *, const Record &, Record &,
          B_node<Record, order> *&);
   void push_in(B_node<Record, order> *, const Record &,
             B_node<Record, order> *, int);
   void split_node(B_node<Record, order> *, const Record &extra_entry,
           B_node<Record, order> *, int,
           B_node<Record, order> *&, Record &);
   Error_code recursive_remove(B_node<Record, order> *, const Record &);
   void copy_in_predecessor(B_node<Record, order> *, int);
   void remove_data(B_node<Record, order> *, int);
   void restore(B_node<Record, order> *, int);
   void move_left(B_node<Record, order> *, int);
   void move_right(B_node<Record, order> *, int);
   void combine(B_node<Record, order> *, int);

private: //   data members
   B_node<Record, order> *root;
         //   Add private auxiliary functions here.
};


template <class Record, int order>
B_tree<Record, order>::B_tree()
{
   root = NULL;
}
```

TOC
Index
Help

```
template <class Record, int order>
B_tree<Record, order>::~B_tree()
{
//    This needs to be written some time!
}

template <class Record, int order>
Error_code B_tree<Record, order>::insert(const Record &new_entry)
/*
Post: If the Key of newentry is already in the Btree,
      a code of duplicateerror is returned.
      Otherwise, a code of success is returned and the Record newentry
      is inserted into the B-tree in such a way that the properties of a
      B-tree are preserved.
Uses: Methods of struct Bnode and the auxiliary function pushdown.
*/

{
   Record median;
   B_node<Record, order> *right_branch, *new_root;
   Error_code result = push_down(root, new_entry, median, right_branch);

   if (result == overflow) { // The whole tree grows in height.
                             // Make a brand new root for the whole B-tree.
      new_root = new B_node<Record, order>;
      new_root->count = 1;
      new_root->data[0] = median;
      new_root->branch[0] = root;
      new_root->branch[1] = right_branch;
      root = new_root;
      result = success;
   }
   return result;
}

template <class Record, int order>
Error_code B_tree<Record, order>::remove(const Record &target)
/*
Post: If a Record with Key matching that of target belongs to the
      Btree, a code of success is returned and the corresponding node
      is removed from the B-tree.  Otherwise, a code of notpresent
      is returned.
Uses: Function recursiveremove
*/

{
   Error_code result;
   result = recursive_remove(root, target);
   if (root != NULL && root->count == 0) {  //   root is now empty.
      B_node<Record, order> *old_root = root;
      root = root->branch[0];
      delete old_root;
   }
   return result;
}
```

```
template <class Record, int order>
B_node<Record, order>::B_node()
{
   count = 0;
}

template <class Record, int order>
Error_code B_tree<Record, order>::recursive_search_tree(
           B_node<Record, order> *current, Record &target)
/*
Pre:  current is either NULL or points to a subtree of the
      Btree.
Post: If the Key of target is not in the subtree, a code of notpresent
      is returned.
      Otherwise, a code of success is returned and
      target is set to the corresponding Record of
      the subtree.
Uses: recursivesearch_tree recursively and searchnode
*/

{
   Error_code result = not_present;
   int position;
   if (current != NULL) {
      result = search_node(current, target, position);
      if (result == not_present)
         result = recursive_search_tree(current->branch[position], target);
      else
         target = current->data[position];
   }
   return result;
}

template <class Record, int order>
Error_code B_tree<Record, order>::push_down(
                 B_node<Record, order> *current,
                 const Record &new_entry,
                 Record &median,
                 B_node<Record, order> *&right_branch)
/*
Pre:  current is either NULL or points to a node of a Btree.
Post: If an entry with a Key matching that of newentry is in the subtree
      to which current points, a code of duplicateerror is returned.
      Otherwise, newentry is inserted into the subtree: If this causes the
      height of the subtree to grow, a code of overflow is returned, and
      the Record median is extracted to be reinserted higher in the B-tree,
      together with the subtree rightbranch on its right.
      If the height does not grow, a code of success is returned.
Uses: Functions pushdown (called recursively), searchnode,
      splitnode, and pushin.
*/
```

```
{
   Error_code result;
   int position;
   if (current == NULL) {
       // Since we cannot insert in an empty tree, the recursion terminates.
      median = new_entry;
      right_branch = NULL;
      result = overflow;
   }

   else {   //    Search the current node.
      if (search_node(current, new_entry, position) == success)
         result = duplicate_error;

      else {
         Record extra_entry;
         B_node<Record, order> *extra_branch;
         result = push_down(current->branch[position], new_entry,
                                   extra_entry, extra_branch);

         if (result == overflow) {
               //  Record extraentry now must be added to current
            if (current->count < order - 1) {
               result = success;
               push_in(current, extra_entry, extra_branch, position);
            }

            else split_node(current, extra_entry, extra_branch, position,
                                right_branch, median);
         // Record median and its right_branch will go up to a higher node.
         }
      }
   }
   return result;
}

template <class Record, int order>
void B_tree<Record, order>::push_in(B_node<Record, order> *current,
   const Record &entry, B_node<Record, order> *right_branch, int position)
/*
Pre:   current points to a node of a Btree.
      The node *current is not full and
      entry belongs in *current at index position.
Post: entry has been inserted along with its right-hand branch
      rightbranch into *current at index position.
*/

{
   for (int i = current->count; i > position; i--) {
       //   Shift all later data to the right.
      current->data[i] = current->data[i - 1];
      current->branch[i + 1] = current->branch[i];
   }
   current->data[position] = entry;
   current->branch[position + 1] = right_branch;
   current->count++;
}
```

```
template <class Record, int order>
void B_tree<Record, order>::split_node(
   B_node<Record, order> *current,     //   node to be split
   const Record &extra_entry,          //   new entry to insert
   B_node<Record, order> *extra_branch,//   subtree on right of extraentry
   int position,                      //  index in node where extraentry goes
   B_node<Record, order> *&right_half, // new node for right half of entries
   Record &median)                     //   median entry (in neither half)

/*
Pre:  current points to a node of a B_tree.
      The node *current is full, but if there were room, the record
      extraentry with its right-hand pointer extrabranch would belong
      in *current at position position,
      0 <= position < order.
Post: The node *current with extra_entry and pointer extra_branch at
      position position are divided
      into nodes *current and *right_half
      separated by a Record median.
Uses: Methods of struct B_node, function push_in.
*/

{
   right_half = new B_node<Record, order>;
   int mid = order/2;  //   The entries from mid on will go to righthalf.

   if (position <= mid) {  // First case: extraentry belongs in left half.
      for (int i = mid; i < order - 1; i++) { // Move entries to righthalf.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(current, extra_entry, extra_branch, position);
   }

   else {  //   Second case:  extraentry belongs in right half.
      mid++;      //   Temporarily leave the median in left half.
      for (int i = mid; i < order - 1; i++) { // Move entries to righthalf.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }

      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(right_half, extra_entry, extra_branch, position - mid);
   }

      median = current->data[current->count - 1];
                                        //   Remove median from left half.
      right_half->branch[0] = current->branch[current->count];
      current->count--;
}

template <class Record, int order>
Error_code B_tree<Record, order>::recursive_remove(
   B_node<Record, order> *current, const Record &target)
/*
Pre:  current is either NULL or
      points to the root node of a subtree of a Btree.
```

```
Post: If a Record with Key matching that of target belongs to the subtree,
      a code of success is returned and the corresponding node is removed
      from the subtree so that the properties of a B-tree are maintained.
      Otherwise, a code of notpresent is returned.
Uses: Functions searchnode, copyin_predecessor,
      recursiveremove (recursive-ly), removedata, and restore.
*/

{
   Error_code result;
   int position;
   if (current == NULL) result = not_present;

   else {
      if (search_node(current, target, position) == success) {
                                    // The target is in the current node.
         result = success;

         if (current->branch[position] != NULL) {   //  not at a leaf node
            copy_in_predecessor(current, position);

            recursive_remove(current->branch[position],
                             current->data[position]);
         }
         else remove_data(current, position);  //  Remove from a leaf node.
      }

      else result = recursive_remove(current->branch[position], target);
      if (current->branch[position] != NULL)
         if (current->branch[position]->count < (order - 1) / 2)
            restore(current, position);
   }
   return result;
}

template <class Record, int order>
void B_tree<Record, order>::copy_in_predecessor(
                    B_node<Record, order> *current, int position)

/*
Pre:   current points to a non-leaf node in a B-tree with an
        entry at position.
Post: This entry is replaced by its immediate predecessor
        under order of keys.
*/

{
   B_node<Record, order> *leaf = current->branch[position];
                                 //    First go left from the current entry.
   while (leaf->branch[leaf->count] != NULL)
      leaf = leaf->branch[leaf->count]; // Move as far right as possible.
   current->data[position] = leaf->data[leaf->count - 1];
}

template <class Record, int order>
void B_tree<Record, order>::remove_data(B_node<Record, order> *current,
                                        int position)

/*
Pre:  current points to a leaf node in a B-tree with an entry at position.
Post: This entry is removed from *current.
*/
```

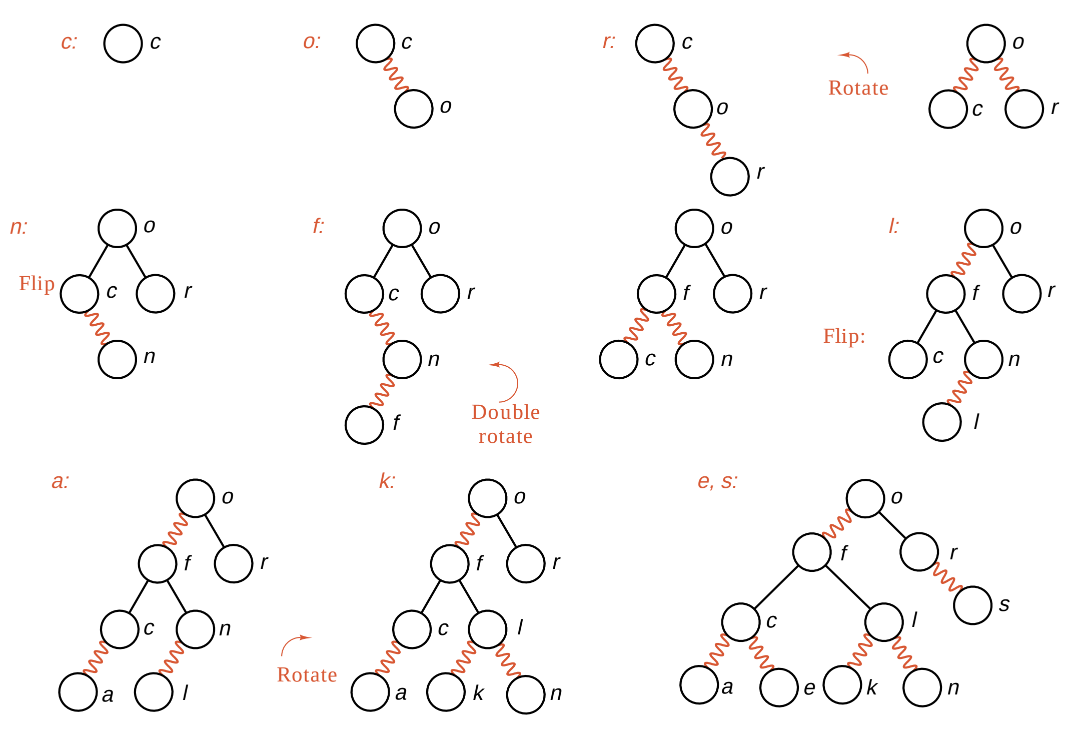
```
{
   for (int i = position; i < current->count - 1; i++)
      current->data[i] = current->data[i + 1];
   current->count--;
}

template <class Record, int order>
void B_tree<Record, order>::restore(B_node<Record, order> *current,
                                    int position)
/*
Pre:   current points to a non-leaf node in a B-tree; the node to which
      current->branch[position] points has one too few entries.
Post: An entry is taken from elsewhere to restore the minimum number of
      entries in the node to which current->branch[position] points.
Uses: move_left, move_right, combine.
*/

{
   if (position == current->count)   //   case:  rightmost branch
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else
         combine(current, position);

   else if (position == 0)       //   case: leftmost branch
      if (current->branch[1]->count > (order - 1) / 2)
         move_left(current, 1);
      else
         combine(current, 1);

   else                          //   remaining cases: intermediate branches
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else if (current->branch[position + 1]->count > (order - 1) / 2)
         move_left(current, position + 1);
      else
         combine(current, position);
}

template <class Record, int order>
void B_tree<Record, order>::move_left(B_node<Record, order> *current,
                                      int position)
/*
Pre:  current points to a node in a B-tree with more than the minimum
      number of entries in branch position and one too few entries in
      branch position - 1.

Post: The leftmost entry from branch position has moved into
      current, which has sent an entry into the branch position - 1.
*/

{
   B_node<Record, order> *left_branch = current->branch[position - 1],
                         *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
                                              //   Take entry from the parent.
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   current->data[position - 1] = right_branch->data[0];
                        //    Add the right-hand entry to the parent.
   right_branch->count--;
   for (int i = 0; i < right_branch->count; i++) {
                        //   Move right-hand entries to fill the hole.
      right_branch->data[i] = right_branch->data[i + 1];
      right_branch->branch[i] = right_branch->branch[i + 1];
   }
   right_branch->branch[right_branch->count] =
      right_branch->branch[right_branch->count + 1];
}

template <class Record, int order>
void B_tree<Record, order>::move_right(B_node<Record, order> *current,
                                       int position)
/*
Pre:   current points to a node in a B-tree with more than the minimum
      number of entries in branch position and one too few entries
      in branch position + 1.

Post: The rightmost entry from branch position has moved into
      current, which has sent an entry into the branch position + 1.
*/

{
   B_node<Record, order> *right_branch = current->branch[position + 1],
                         *left_branch = current->branch[position];
   right_branch->branch[right_branch->count + 1] =
      right_branch->branch[right_branch->count];
   for (int i = right_branch->count ; i > 0; i--) {
                                                //   Make room for new entry.
      right_branch->data[i] = right_branch->data[i - 1];
      right_branch->branch[i] = right_branch->branch[i - 1];
   }
   right_branch->count++;
   right_branch->data[0] = current->data[position];
                                                  // Take entry from parent.
   right_branch->branch[0] = left_branch->branch[left_branch->count--];
   current->data[position] = left_branch->data[left_branch->count];
}

template <class Record, int order>
void B_tree<Record, order>::combine(B_node<Record, order> *current,
                                    int position)
/*
Pre:   current points to a node in a B-tree with entries in the branches
      position and position - 1, with too few to move entries.
Post: The nodes at branches position - 1 and position have been combined
      into one node, which also includes the entry formerly in current at
      index  position - 1.
*/

{
   int i;
   B_node<Record, order> *left_branch = current->branch[position - 1],
                         *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   for (i = 0; i < right_branch->count; i++) {
      left_branch->data[left_branch->count] = right_branch->data[i];
      left_branch->branch[++left_branch->count] =
                                        right_branch->branch[i + 1];
   }
   current->count--;
   for (i = position - 1; i < current->count; i++) {
      current->data[i] = current->data[i + 1];
      current->branch[i + 1] = current->branch[i + 2];
   }
   delete right_branch;
}
```

11.4 RED-BLACK TREES

Exercises 11.4

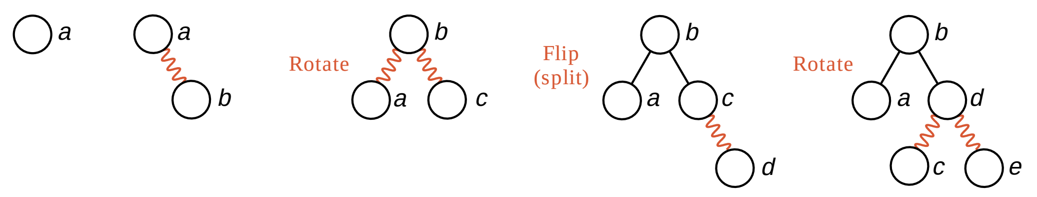
E1. *Insert the keys* c, o, r, n, f, l, a, k, e, s *into an initially empty red-black tree.*

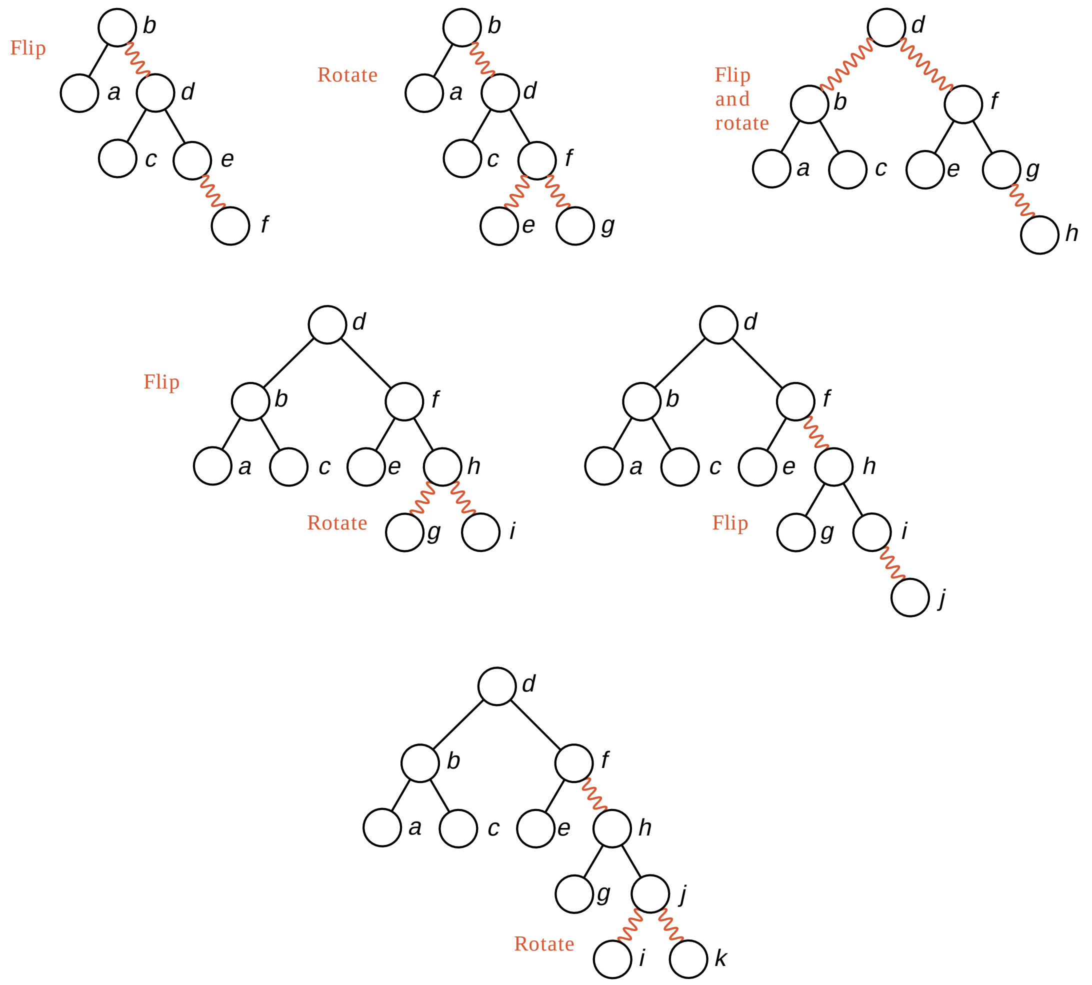
Answer



E2. *Insert the keys* a, b, c, d, e, f, g, h, i, j, k *into an initially empty red-black tree.*

Answer





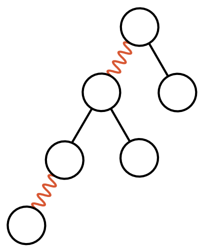
TOC
Index
Help

E3. *Find a binary search tree whose nodes cannot be colored so as to make it a red-black tree.*

Answer A chain of length three or more is such a tree.

E4. *Find a red-black tree that is not an AVL tree.*

Answer



E5. *Prove that any AVL tree can have its nodes colored so as to make it a red-black tree. You may find it easier to prove the following stronger statement: An AVL tree of height h can have its nodes colored as a red-black tree with exactly $\lceil h/2 \rceil$ black nodes on each path to an empty subtree, and, if h is odd, then both children of the root are black.*

Answer We prove the stronger statement by mathematical induction on the height h. Trees with $h = 0$ (empty tree) and with $h = 1$ (a single node) clearly satisfy the statement. Suppose that all AVL trees of height less than h satisfy the statement. Take T to be an AVL tree of height h. Then the left and right subtrees, which have height $h - 1$ or $h - 2$, satisfy the statement. If one (or both) of these has odd height, we change the color of its root from black to red. Since both children of its root are black, this change does not violate the red condition. Finally, we color the root of T black. It is now easy to check that, in every (even or odd height) case, the statement holds for T.

Programming Projects 11.4

P1. *Complete red-black insertion by writing the following missing functions:*

(a) modify_right
(b) flip_color
(c) rotate_left
(d) rotate_right
(e) double_rotate_left
(f) double_rotate_right

Be sure that, at the end of each function, the colors of affected nodes have been set properly, and the returned RB_code *correctly indicates the current condition. By including extensive error testing for illegal situations, you can simplify the process of correcting your work.*

Answer

```
template <class Record>
class RB_tree: public Search_tree<Record> {
public:
   Error_code insert(const Record & new_entry);

   void prenode(void (*visit)(Binary_node<Record> *&));
   RB_code rb_insert(Binary_node<Record> *&, const Record &);
   RB_code modify_left(Binary_node<Record> *&, RB_code &);
   RB_code modify_right(Binary_node<Record> *&, RB_code &);
   RB_code rotate_left(Binary_node<Record> *&current);
   RB_code rotate_right(Binary_node<Record> *&current);
   RB_code flip_color(Binary_node<Record> *current);
   RB_code double_rotate_right(Binary_node<Record> *&current);
   RB_code double_rotate_left(Binary_node<Record> *&current);

private:  //  Add prototypes for auxiliary functions here.
};

enum Color {red, black};

template <class Entry>
struct Binary_node {
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
   virtual Color get_color() const { return red; }
   virtual void set_color(Color c) { }
   Binary_node()                   { left = right = NULL; }
   Binary_node(const Entry &x)     { data = x; left = right = NULL; }
};

template <class Record>
struct RB_node: public Binary_node<Record> {
   Color color;
   RB_node(const Record &new_entry) { color = red; data = new_entry;
                                      left = right = NULL; }
   RB_node()                  { color = red; left = right = NULL; }
   void set_color(Color c)    { color = c; }
   Color get_color() const    { return color; }
};

template <class Record>
Error_code RB_tree<Record>::insert(const Record &new_entry)
/*
Post: If the key of new_entry is already in the RB_tree, a code of
      duplicate_error is returned.  Otherwise, a code of success is
      returned and the Record new_entry is inserted into the tree in such
      a way that the properties of an RB-tree have been preserved.
Uses: Methods of struct RB_node and recursive function rb_insert.
*/
```

```
{
   RB_code status = rb_insert(root, new_entry);
   switch (status) {   //  Convert private RB_code to public Error_code.
      case red_node:   //  Always split the root node to keep it black.
         root->set_color(black);
                  /*  Doing so prevents two red nodes at the top
                  of the  tree and a resulting attempt to rotate
                  using a parent node that does not exist. */

      case okay:
         return success;

      case duplicate:
         return duplicate_error;

      case right_red:
      case left_red:
         cout << "WARNING: Program error detected in RB_tree::insert"
              << endl;
         return internal_error;
   }
}

template <class Record>
void RB_tree<Record>::prenode(void (*f)(Binary_node<Record> *&))
{
   rb_prenode(root, f);
}

template <class Record>
void rb_prenode(Binary_node<Record> *current,
                void (*f)(Binary_node<Record> *&))
{
   if (current != NULL) {
      (*f)(current);
      rb_prenode(current->left, f);
      rb_prenode(current->right, f);
   }
}


template <class Entry> void Binary_tree<Entry>::
   recursive_inorder(Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points a subtree of
      a Binary_tree.
Post: The subtree has been been traversed in inorder sequence.
Uses: The function recursive_inorder recursively
*/

{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}
```

```
template <class Entry> void Binary_tree<Entry>::
   recursive_preorder(Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points to a subtree of
      a Binary_tree.
Post: The subtree has been been traversed in preorder sequence.
Uses: The function recursive_preorder recursively
*/
{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_preorder(sub_root->left, visit);
      recursive_preorder(sub_root->right, visit);
   }
}

template <class Entry> void Binary_tree<Entry>::
   recursive_postorder(Binary_node<Entry> *sub_root, void (*visit)(Entry &))
/*
Pre:  sub_root is either NULL or points to a subtree of
      a Binary_tree.
Post: The subtree has been been traversed in postorder sequence.
Uses: The function recursive_postorder recursively
*/
{
   if (sub_root != NULL) {
      recursive_postorder(sub_root->left, visit);
      recursive_postorder(sub_root->right, visit);
      (*visit)(sub_root->data);
   }
}

template <class Entry> void Binary_tree<Entry>::
   recursive_insert(Binary_node<Entry> *&sub_root, const Entry &x)
/*
Pre:  sub_root is either NULL or points to a subtree of
      a Binary_tree.
Post: The Entry  has been inserted into the subtree in such a way
      that the properties of a binary search tree have been preserved.
Uses: The functions recursive_insert, recursive_height
*/
{
   if (sub_root == NULL) sub_root = new Binary_node<Entry>(x);
   else
     if (recursive_height(sub_root->right) <
                                 recursive_height(sub_root->left))
        recursive_insert(sub_root->right, x);
   else
      recursive_insert(sub_root->left, x);
}

template <class Entry> int Binary_tree<Entry>::
   recursive_size(Binary_node<Entry> *sub_root) const
{
   if (sub_root == NULL) return 0;
   return 1 + recursive_size(sub_root->left) +
              recursive_size(sub_root->right);
}
```

```
template <class Entry> int Binary_tree<Entry>::
   recursive_height(Binary_node<Entry> *sub_root) const
{
   if (sub_root == NULL) return 0;
   int l = recursive_height(sub_root->left);
   int r = recursive_height(sub_root->right);
   if (l > r) return 1 + l;
   else return 1 + r;
}

template <class Entry> void Binary_tree<Entry>::
   recursive_clear(Binary_node<Entry> *&sub_root)
{
   Binary_node<Entry> *temp = sub_root;
   if (sub_root == NULL) return;
   recursive_clear(sub_root->left);
   recursive_clear(sub_root->right);
   sub_root = NULL;
   delete temp;
}

template <class Entry> Binary_node<Entry> *Binary_tree<Entry>::
   recursive_copy(Binary_node<Entry> *sub_root)
{
   if (sub_root == NULL) return NULL;
   Binary_node<Entry> *temp = new Binary_node<Entry>(sub_root->data);
   temp->left = recursive_copy(sub_root->left);
   temp->right = recursive_copy(sub_root->right);
   return temp;
}

template <class Entry> void Binary_tree<Entry>::
   recursive_swap(Binary_node<Entry> *sub_root)
{
   if (sub_root == NULL) return;
   Binary_node<Entry> *temp = sub_root->left;
   sub_root->left = sub_root->right;
   sub_root->right = temp;
   recursive_swap(sub_root->left);
   recursive_swap(sub_root->right);
}

template <class Entry> Binary_node<Entry> *&Binary_tree<Entry>::
   find_node(Binary_node<Entry> *&sub_root, const Entry &x) const
{
   if (sub_root == NULL || sub_root->data == x) return sub_root;
   else {
      Binary_node<Entry> *&temp = find_node(sub_root->left, x);
      if (temp != NULL) return temp;
      else return find_node(sub_root->right, x);
   }
}

template <class Entry> Error_code Binary_tree<Entry>::
   remove_root(Binary_node<Entry> *&sub_root)
/*
Pre:  sub_root is either NULL or points to a subtree of
      a Search_tree
```

```
Post: If sub_root is NULL, a code of not_present is returned.
      Otherwise the root of the subtree is removed in such a way
      that the properties of a binary search tree are preserved.
      The parameter sub_root is reset as the root of the modified subtree
      and success is returned.
*/

{
   if (sub_root == NULL) return not_present;
   Binary_node<Entry> *to_delete = sub_root;  //  Remember node to delete.
   if (sub_root->right == NULL) sub_root = sub_root->left;
   else if (sub_root->left == NULL) sub_root = sub_root->right;

   else {                          //  Neither subtree is empty.
      to_delete = sub_root->left; // Move left to start finding predecessor.
      Binary_node<Entry> *parent = sub_root; //  parent of to_delete.
      while (to_delete->right != NULL) { //to_delete is not the predecessor.
         parent = to_delete;
         to_delete = to_delete->right;
      }
      sub_root->data = to_delete->data;  //  Move from to_delete to root.
      if (parent == sub_root) sub_root->left = to_delete->left;
      else parent->right = to_delete->left;
   }

   delete to_delete;   //  Remove to_delete from tree.
   return success;
}


template <class Record>
RB_code RB_tree<Record>::rb_insert(Binary_node<Record> *&current,
                                   const Record &new_entry)
/*
Pre:  current is either NULL or points to the
      first node of a subtree of an RB_tree
Post: If the key of new_entry is already in the
      subtree, a code of duplicate
      is returned.  Otherwise, the Record new_entry is
      inserted into the subtree
      pointed to by current.  The properties of a red-black tree have been
      restored, except possibly at the root current and one of its children,
      whose status is given by the output RB_code.
Uses: Methods of class RB_node,
      rb_insert recursively, modify_left, and modify_right.
*/

{

   RB_code status,
           child_status;
   if (current == NULL) {
      current = new RB_node<Record>(new_entry);
      status = red_node;
   }
   else if (new_entry == current->data)
      return duplicate;
   else if (new_entry < current->data) {
      child_status = rb_insert(current->left, new_entry);
      status = modify_left(current, child_status);
   }
   else {
      child_status = rb_insert(current->right, new_entry);
      status = modify_right(current, child_status);
   }
   return status;
}

template <class Record>
RB_code RB_tree<Record>::modify_left(Binary_node<Record> *&current,
                                     RB_code &child_status)
/*
Pre:  An insertion has been made in the left subtree of *current that
      has returned the value of child_status  for this subtree.
Post: Any color change or rotation needed for the tree rooted at current
      has been made, and a status code is returned.
Uses: Methods of struct RB_node, with rotate_right,
      double_rotate_right, and flip_color.
*/
{
   RB_code status = okay;
   Binary_node<Record> *aunt = current->right;
   Color aunt_color = black;
   if (aunt != NULL) aunt_color = aunt->get_color();

   switch (child_status) {
   case okay:
      break;         //  No action needed, as tree is already OK.

   case red_node:
      if (current->get_color() == red)
         status = left_red;
      else
         status = okay;        //  current is black, left is red, so OK.
      break;

   case left_red:
      if (aunt_color == black) status = rotate_right(current);
      else                     status = flip_color(current);
      break;

   case right_red:
      if (aunt_color == black) status = double_rotate_right(current);
      else                     status = flip_color(current);
      break;
   }
   return status;
}

template <class Record>
RB_code RB_tree<Record>::modify_right(Binary_node<Record> *&current,
                                      RB_code &child_status)
{
   RB_code status = okay;
   Binary_node<Record> *left_child = current->left;
   switch (child_status) {
   case okay: break;

   case red_node:
      if (current->get_color() == red)
         status = right_red;
      else
         status = okay;
      break;

   case right_red:
      if (left_child == NULL)
         status = rotate_left(current);
      else if (left_child->get_color() == red)
         status = flip_color(current);
      else
         status = rotate_left(current);
      break;

   case left_red:
      if (left_child == NULL)
         status = double_rotate_left(current);
      else if (left_child->get_color() == red)
         status = flip_color(current);
      else
         status = double_rotate_left(current);
      break;
   }
   return status;
}

template <class Record>
RB_code RB_tree<Record>::rotate_left(Binary_node<Record> *&current)
/*
Pre:  current points to a subtree of an RB_tree. The subtree has a
      nonempty right subtree.
Post: current is reset to point to its former right child, and the former
      current node is the left child of the new current node.
*/

{
   if (current == NULL || current->right == NULL)     //  impossible cases
      cout << "WARNING: program error in detected in rotate_left\n\n";
   else {
      Binary_node<Record> *right_tree = current->right;
      current->set_color(red);
      right_tree->set_color(black);
      current->right = right_tree->left;
      right_tree->left = current;
      current = right_tree;
   }
   return okay;
}

template <class Record>
RB_code RB_tree<Record>::rotate_right(Binary_node<Record> *&current)
```

```
{
   if (current == NULL || current->left == NULL) //  impossible cases
      cout << "WARNING: program error in detected in rotate_right\n\n";
   else {
      Binary_node<Record> *left_tree = current->left;
      current->set_color(red);
      left_tree->set_color(black);
      current->left = left_tree->right;
      left_tree->right = current;
      current = left_tree;
   }
   return okay;
}

template <class Record>
RB_code RB_tree<Record>::flip_color(Binary_node<Record> *current)
{
   Binary_node<Record> *left_child = current->left,
      *right_child = current->right;
   current->set_color(red);
   left_child->set_color(black);
   right_child->set_color(black);
   return red_node;
}

template <class Record>
RB_code RB_tree<Record>::double_rotate_right(Binary_node<Record> *&current)
{
   rotate_left(current->left);
   rotate_right(current);
   return okay;
}

template <class Record>
RB_code RB_tree<Record>::double_rotate_left(Binary_node<Record> *&current)
{
   rotate_right(current->right);
   rotate_left(current);
   return okay;
}
```

P2. *Substitute the function for red-black insertion into the menu-driven demonstration program for binary search trees from Section 10.2, Project P2 (page 460), thereby obtaining a demonstration program for red-black trees. You may leave removal not implemented.*

Answer

```
#include "../../c/utility.h"

void help()
/*
PRE:  None.
POST: Instructions for the RB-tree operations have been printed.
*/

{
     cout << "Legal commands are: \n";
     cout << "\t[P]rint     [I]nsert    [#]size \n"
          << "\t[R]emove    [C]lear     [H]elp      [Q]uit " << endl;
}
```

```
#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/sortable.h"
#include "../../8/linklist/merge.cpp"

void write_ent(int &x)
{
   cout << x << " , ";
}

#include "../rb/node.h"
#include "../../10/bt/tree.h"
#include "../rb/node.cpp"
#include "../../10/bt/tree.cpp"
#include "../../10/bst/stree.h"
#include "../../10/bst/snode.cpp"
#include "../../10/bst/stree.cpp"
#include "../rb/rbcode.h"
#include "../rb/rbnode.h"

template <class Entry>
void pr_node(Binary_node<Entry> *&x)
/*
Post: A node of the RB-tree has been printed
*/

{
   cout << "( " << x->data << " :  -> " << "  ";
   switch (x->get_color()) {
   case red: cout << "  R"; break;

   case black: cout << "  B"; break;
   }
   cout << " )  ===> ";
   if (x->left != NULL) cout << (x->left)->data << " ";
   if (x->right != NULL) cout << (x->right)->data << " ";
   cout << " \n" << flush;
}

#include "../rb/rbtree.h"
#include "../rb/rbnode.cpp"
#include "../rb/rbtree.cpp"

int get_int()
/*
Post: A user specified integer is returned.
*/

{
   char c;
   int ans = 0, sign = 1;
   do {
      cin.get(c);
      if (c == '-') sign = -1;
   } while (c < '0' || c > '9');

   while (c >= '0' && c <= '9') {
      ans = 10 * ans + c - '0';
      cin.get(c);
   }
   return ans * sign;
}
```

```
char get_command()
/*
POST: Returns a user specified character command
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');

      do {
         cin.get(d);
      } while (d != '\n');

      c = tolower(c);
      if (c == 'r' || c == '#' || c == 'i' || c == 'c' ||
         c == 'h' || c == 'q' || c == 'p') {
         return c;
      }

      cout << "Please enter a valid command or H for help:";
      cout << "\n\t[R]emove  entry\t[P]rint tree\t[#] size of tree\n"
         << "\t[C]lear tree\t[I]nsert entry\n"
            << "\t[Q]uit.\n";
   }
}

int do_command(char c, RB_tree<int> &test_tree)
/*
POST: Performs the command specified by the parameter c
*/

{
   int x;
   int sz;
   switch (c) {
   case 'h':
      help();
      break;

   case 'r':
      cout << "Enter integer to remove:" << flush;
      x = get_int();
      if (test_tree.remove(x) != success) cout << "Not found!\n";
      break;

   case 'i':
      cout << "Enter new integer to insert:";
      x = get_int();
      test_tree.insert(x);
      break;

   case 'c':
      test_tree.clear();
      cout << "Tree is cleared.\n";
      break;
   case 'p':
      sz = test_tree.size();
      if (sz == 0) cout << "Tree is empty.\n";
      else test_tree.prenode(pr_node);
      break;

   case '#':
      cout << "The size of the tree is " << test_tree.size() << "\n";
      break;

   case 'q':
      cout << "RB-tree demonstration finished.\n";
      return 0;
   }
   return 1;
}

main()
/*
Post: Runs the RB-Tree demonstration program of
      Project P2 of Section 11.4
*/

{
   cout << "\n Demonstration Program for RB-trees"
        << " with integer entries." << endl;
   Binary_tree<int> t; Binary_tree<int> s = t;
   Search_tree<int> t1; Search_tree<int> s1 = t1;
   help();

   RB_tree<int> test_tree;
   while (do_command(get_command(), test_tree));
}

enum RB_code {okay, red_node, left_red, right_red, duplicate};

/*  These outcomes from a call to the recursive insertion function describe
the following results:

duplicate:
okay:      The color of the current root (of the subtree) has not changed;
           the tree now satisfies the conditions for a red-black tree.

red_node:  The current root has changed from black to red; modification
           may or may not be needed to restore the red-black properties.

right_red: The current root and its right child are now both red;
           a color flip or rotation is needed.

left_red:  The current root and its left child are now both red;
           a color flip or rotation is needed.

duplicate: The entry being inserted duplicates another entry; this is
           an error.
*/
```

P3. *Substitute the function for red-black insertion into the information-retrieval project of Project P5 of Section 10.2 (page 461). Compare the performance of red-black trees with other search trees for various combinations of input text files.*

Answer Main program:

```
#include <stdlib.h>
#include <string.h>
#include "../../c/utility.h"
#include "../../c/utility.cpp"

#include "../../6/doubly/list.h"
#include "../../6/doubly/list.cpp"
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "../rb/node.h"
#include "../../10/bt/tree.h"
#include "../rb/node.cpp"
#include "../../10/bt/tree.cpp"
#include "../../10/bst/stree.h"
#include "../../10/bst/snode.cpp"
#include "../../10/bst/stree.cpp"
#include "../rb/rbcode.h"
#include "../rb/rbnode.h"

#include "../rb/rbtree.h"
#include "../rb/rbnode.cpp"
#include "../rb/rbtree.cpp"

#include "../../10/2p5/key.h"
#include "../../10/2p5/key.cpp"
#include "../../10/2p5/record.h"
#include "../../10/2p5/record.cpp"

#include "auxil.cpp"
#include "../../c/timer.h"
#include "../../c/timer.cpp"

int main(int argc, char *argv[]) // count, values of command-line arguments
/*
Pre:  Name of an input file can be given as a command-line argument.
Post: The word storing project p6 has been performed.
*/

{
   write_method();
   Binary_tree<Record> t; Binary_tree<Record> t1 = t;
   Search_tree<Record> s; Search_tree<Record> s1 = s;

   RB_tree<Record> the_words;

   char infile[1000];
   char in_string[1000];
   char *in_word;
   Timer clock;
   do {
      int initial_comparisons = Key::comparisons;
      clock.reset();
      if (argc < 2) {
         cout << "What is the input data file: " << flush;
         cin >> infile;
      }
      else strcpy(infile, argv[1]);
      ifstream file_in(infile);   //  Declare and open the input stream.
      if (file_in == 0) {
         cout << "Can't open input file " << infile << endl;
         exit (1);
      }

      while (!file_in.eof()) {
        file_in >> in_string;
        int position = 0;
        bool end_string = false;
        while (!end_string) {
           in_word = extract_word(in_string, position, end_string);
           if (strlen(in_word) > 0) {
              int counter;
              String s(in_word);
              update(s, the_words, counter);
           }
        }
      }
      cout << "Elapsed time: " << clock.elapsed_time() << endl;
      cout << "Comparisons performed = "
           << Key::comparisons - initial_comparisons << endl;
      cout << "Do you want to print frequencies? ";
      if (user_says_yes()) print(the_words);
      cout << "Do you want to add another file of input? ";
   } while (user_says_yes());
}
```

Auxiliary functions to carry out the required operations:

```
void write_method()
/*
Post: A short string identifying the abstract
      data type used for the structure is written.
*/

{
   cout << " Word frequency program:  RB-Tree implementation";
   cout << endl;
}

void update(const String &word, RB_tree<Record> &structure,
               int &num_comps)
/*
Post: If word was not already present in structure, then
      word has been inserted into structure and its frequency
      count is 1.  If word was already present in structure,
      then its frequency count has been increased by 1.  The
      variable parameter num_comps is set to the number of
      comparisons of words done.
*/

{
  int initial_comparisons = Key::comparisons;
  Record r((Key) word);
  if (structure.tree_search(r) == not_present)
     structure.insert(r);
  else {
     structure.remove(r);
     r.increment();
     structure.insert(r);
  }
  num_comps = Key::comparisons - initial_comparisons;
}

void wr_entry(Record &a_word)
{
   String s = ((Key) a_word).the_key();
   cout << s.c_str();
   for (int i = strlen(s.c_str()); i < 12; i++) cout << " ";
   cout << " : " << a_word.frequency() << endl;
}

void print(RB_tree<Record> &structure)
/*
Post: All words in structure are printed at the terminal
      in alphabetical order together with their frequency counts.
*/

{
     structure.inorder(wr_entry);
}

char *extract_word(char *in_string, int &examine, bool &over)
{
   int ok = 0;
   while (in_string[examine] != '\0') {
     if ('a' <= in_string[examine] && in_string[examine] <= 'z')
       in_string[ok++] = in_string[examine++];

     else if ('A' <= in_string[examine] && in_string[examine] <= 'Z')
       in_string[ok++] = in_string[examine++] - 'A' + 'a';

     else if (('\'' == in_string[examine] || in_string[examine] == '-')
       && (
          ('a' <= in_string[examine + 1] && in_string[examine + 1] <= 'z')
       || ('A' <= in_string[examine + 1] && in_string[examine + 1] <= 'Z')))
       in_string[ok++] = in_string[examine++];
     else break;
   }
   in_string[ok] = '\0';
   if (in_string[examine] == '\0') over = true;
   else examine++;
   return in_string;
}
```

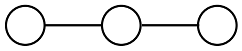
REVIEW QUESTIONS

1. *Define the terms* **(a)** *free tree,* **(b)** *rooted tree, and* **(c)** *ordered tree.*

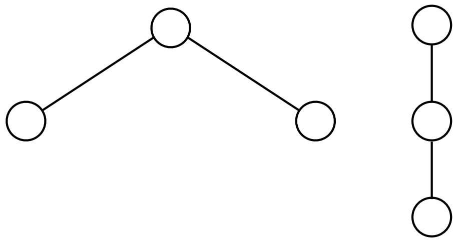
(a) A ***free tree*** is any set of points (called *vertices*) and any set of distinct pairs of these points (called *edges*) such that (1) there is a sequence of edges (a *path*) from any vertex to any other and (2) there is no path starting and ending at the same vertex (a *cycle*). Alternatively, a ***free tree*** can be defined as a connected undirected graph with no cycles. **(b)** A ***rooted tree*** is a free tree in which one vertex is distinguished and called the *root*. **(c)** An ***ordered tree*** is a rooted tree in which the children of each vertex are assigned an order.

2. *Draw all the different* **(a)** *free trees,* **(b)** *rooted trees, and* **(c)** *ordered trees with three vertices.*

There is only one free tree with three vertices:



This tree leads to two rooted trees, which are also the only ordered trees with three vertices:



3. *Name three ways describing the correspondence between orchards and binary trees, and indicate the primary purpose for each of these ways.*

(1) The first_child and next_sibling links provide the way the correspondence is usually implemented in computer programs. (2) The rotation of diagrams is usually the easiest for people to picture. (3) The formal notational equivalence is most useful for proving properties of binary trees and orchards.

4. *What is a trie?*

A trie is a multiway tree in which the branching at each level is determined by the appropriate character of the key, but a trie differs from an arbitrary multiway tree in that it is pruned to remove all branches and nodes that do not lead to any entry.

5. *How may a trie with six levels and a five-way branch in each node differ from the rooted tree with six levels and five children for every node except the leaves? Will the trie or the tree likely have fewer nodes, and why?*

Some of the branches and nodes may be missing from the trie, since there may be no corresponding entry. The trie will therefore likely have fewer nodes.

6. *Discuss the relative advantages in speed of retrieval of a trie and a binary search tree.*

The speed of retrieval from a trie depends on the *length* of the keys, while the speed of retrieval from a search tree depends on the *number* of keys. As the total number of keys increases, of course, the average length of a key must also increase, so the speed of a trie will decrease as will that of a search tree (and the length of keys will increase proportionally to $\lg n$). If comparison of entire keys can be made quickly or it is relatively hard to process a key character by character, then binary search trees should be superior. If comparisons of entire keys are relatively slow (depending, say, on a loop iterating character by character) then tries are likely to prove faster than binary search trees.

7. *How does a multiway search tree differ from a trie?*

In any search tree (binary or multiway), keys are considered as indivisible units and compared in their entirety. In tries, keys are decomposed and processed as sequences of characters. Hence the branching processes are entirely different.

8. *What is a B-tree?*

A ***B-tree*** is a multiway search tree which is balanced in that all the leaves are on the same level and all non-leaves except possibly the root are at least half full of entries.

9. *What happens when an attempt is made to insert a new entry into a full node of a B-tree?*

The node splits into two nodes that become siblings on the same level in the tree. The two nodes are now each half full, and their median entry is sent upward to be inserted into their parent node.

10. *Does a B-tree grow at its leaves or at its root? Why?*

When entries are inserted into leaves they may split as described in the previous question, but the tree does not then grow in height. Its height increases when a new entry is inserted into the root and it is already full. The root then splits into two nodes and the median entry goes by itself into a new root, whereby the tree grows in height.

11. *In deleting an entry from a B-tree, when is it necessary to combine nodes?*

If deleting an entry reduces the number of entries in a node to less than half its capacity and neither of the sibling nodes has an extra entry (above half capacity) that can be moved into the underfull node, then the node, one of its siblings, and the median entry from the parent are combined into a single node. If the parent becomes underfull, the process propagates upward through the tree.

12. *For what purposes are B-trees especially appropriate?*

B-trees are valuable in external information retrieval, where a single access brings in a large block or page of memory with room for several entries at once.

13. *What is the relationship between red-black trees and B-trees?*

A red-black tree *is* a B-tree of order 4 (1, 2, or 3 entries per node) where the B-tree nodes are implemented as miniature binary search trees. The red links describe links within one of these binary search trees, and the black links connect one B-tree node to another.

14. *State the black and the red conditions.*

A red-black tree has the same number of black nodes on any path from the root to an empty subtree. If a node is red, then its parent exists and is black.

15. *How is the height of a red-black tree related to its size?*

If a red-black tree has n nodes, its height is no more than $2 \lg n$.

Graphs

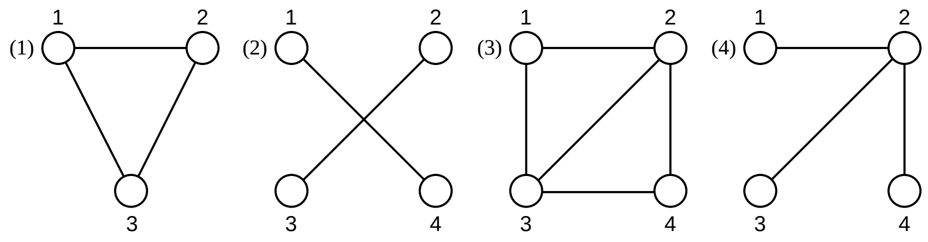
12

TOC
Index
Help

12.7 GRAPHS AS DATA STRUCTURES

Exercises 12.7

E1. **(a)** *Find all the cycles in each of the following graphs.* **(b)** *Which of these graphs are connected?* **(c)** *Which of these graphs are free trees?*



Answer

(a) Graph (1) consists of a single cycle. Graph (3) has the cycles 1 2 3, 2 4 3, and 1 2 4 3. Graphs (2) and (4) have no cycles.

(b) All graphs are connected except (2).

(c) Only (4) is a free tree.

E2. *For each of the graphs shown in Exercise E1, give the implementation of the graph as* **(a)** *an adjacency table,* **(b)** *a linked vertex list with linked adjacency lists,* **(c)** *a contiguous vertex list of contiguous adjacency lists.*

Answer

(a)

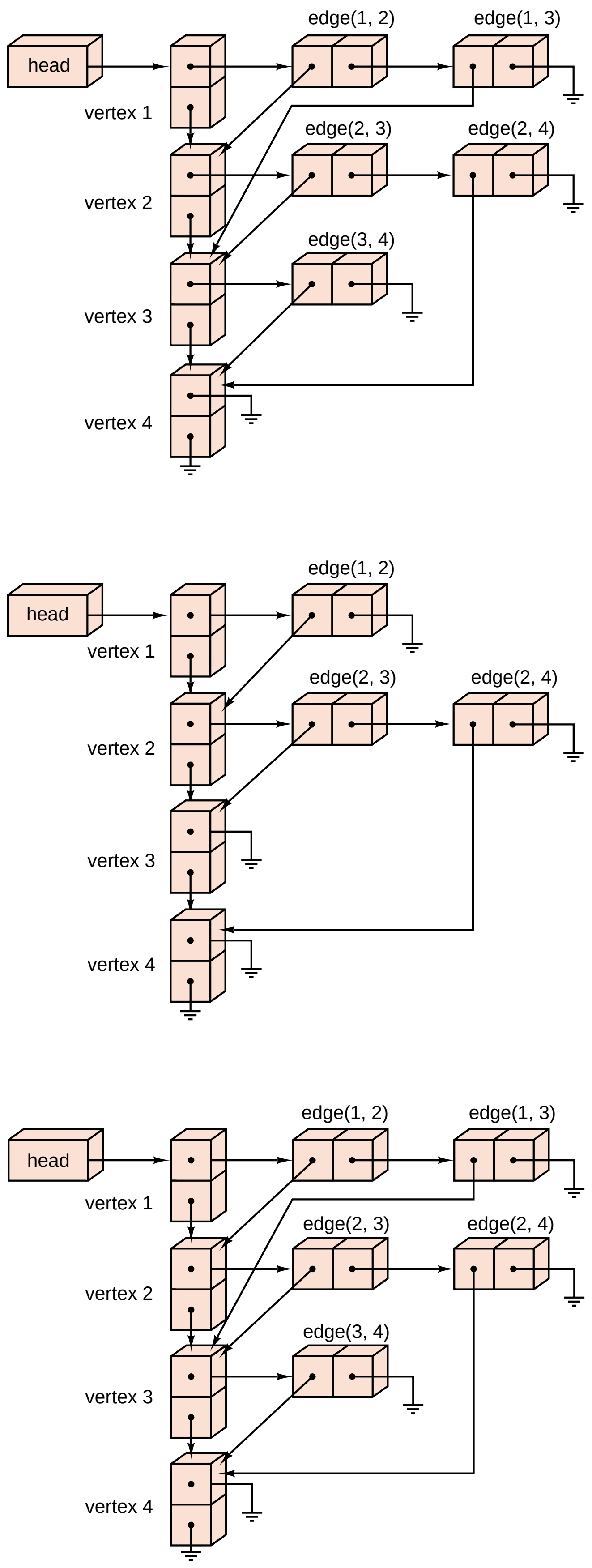
(1)	1	2	3
1	F	T	T
2	T	F	T
3	T	T	F

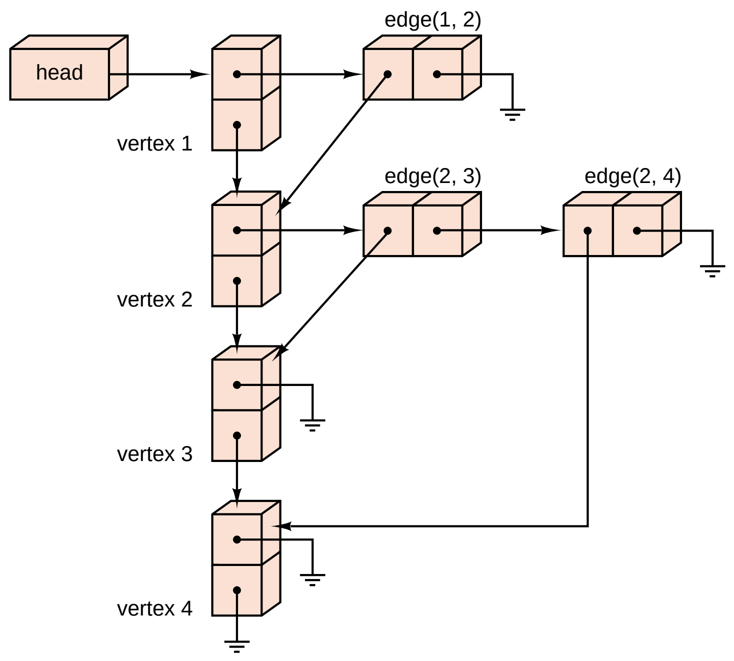
(2)	1	2	3	4
1	F	F	F	T
2	F	F	T	F
3	F	T	F	F
4	T	F	F	F

(3)	1	2	3	4
1	F	T	T	F
2	T	F	T	T
3	T	T	F	T
4	F	T	T	F

(4)	1	2	3	4
1	F	T	F	F
2	T	F	T	T
3	F	T	F	F
4	F	T	F	F

(b)


head
vertex 1
vertex 2
vertex 3
vertex 4
edge(1, 2)
edge(1, 3)
edge(2, 3)
edge(2, 4)
edge(3, 4)



(c) For each of the following four tables, the leftmost column gives the index of a vertex and the remaining columns the list of vertices to which it is adjacent.

(1)	1	2 3	(2)	1	4	(3)	1	2 3	(4)	1	2
	2	1 3		2	3		2	1 3 4		2	1 3 4
	3	1 2		3	2		3	1 2 4		3	2
				4	1		4	2 3		4	2

TOC
Index
Help

E3. *A graph is* ***regular*** *if every vertex has the same valence (that is, if it is adjacent to the same number of other vertices). For a regular graph, a good implementation is to keep the vertices in a linked list and the adjacency lists contiguous. The length of all the adjacency lists is called the* ***degree*** *of the graph. Write a C++ class specification for this implementation of regular graphs.*

Answer

```
template <int degree>
class Vertex {
  Vertex *next_vertex;
  Vertex *adjacent[degree];
};
```

E4. *The topological sorting functions as presented in the text are deficient in error checking. Modify the* **(a)** *depth-first and* **(b)** *breadth-first functions so that they will detect any (directed) cycles in the graph and indicate what vertices cannot be placed in any topological order because they lie on a cycle.*

Answer **(a)** One method for the depth-first algorithm is to add a second **bool** array completed (of the same type as visited). This second array should also be initialized to false for each vertex, and the entry for vertex v should be changed to true at the end of the sorting function. At both places where the algorithm checks that a vertex has not been visited it should also check for a vertex that has been visited but not completed. The appearance of such a vertex means that the algorithm has traversed a cycle including (at least) the current vertex and the vertex being checked.

(b) For the breadth-first algorithm, entries in the array predecessor_count will never reach 0 for vertices lying on cycles, and those vertices will never be put into the topological order. Hence the existence of cycles can be checked by comparing the number of vertices in the topological order with the total number of vertices at the end of the program, and the vertices lying on cycles can be found by a scan through predecessor_count at the end of the program.

E5. *How can we determine a maximal spanning tree in a network?*

Answer We first negate all of the edge weights to obtain a new network. A minimal spanning tree for the new network, which can be obtained with our implementation of Prim's algorithm, gives a maximal spanning tree for the old network.

E6. *Kruskal's algorithm to compute a minimal spanning tree in a network works by considering all edges in increasing order of weight. We select edges for a spanning tree, by adding edges to an initially empty set. An edge is selected if together with the previously selected edges it creates no cycle. Prove that the edges chosen by Kruskal's algorithm do form a minimal spanning tree of a connected network.*

Answer We write G for the graph of the original network, and K for the graph produced by Kruskal's algorithm. We begin by observing that Kruskal's algorithm selects the edge joining a pair of vertices, x and y, if and only if x and y can not be connected by a sequence of previously considered edges. Although this observation is not helpful in reformulating the algorithm, it is useful in proving that the algorithm works.

It is clear that K has no cycles, so to prove that K is a spanning tree for G, we just need to show that K connects together all vertices of G. Suppose that X_1 and X_2 are any pair of sets partitioning the vertices of K. Let e be the first edge of G (in the ordering of edges used by Kruskal's algorithm) that links X_1 to X_2. (There must be such an edge because G is connected). Our earlier observation shows that e is selected by the algorithm, and so X_1 and X_2 are linked by an edge of K. We conclude that K has just one connected component, and therefore it is connected.

Let us label the edges of K as $e_1, e_2, \dots, e_n$ in the order that they are selected by Kruskal's algorithm. To show that K is a minimal spanning tree, we prove, as in our analysis of Prim's algorithm, that if m is an integer with $0 \le m \le n$, then there is a minimal spanning tree that contains the edges e_i with $i \le m$. We can work by induction on m to prove this result. The base case, where $m = 0$, is certainly true, since any minimal spanning tree does contain the requisite empty set of edges. Moreover, once we have completed the induction, the final case with $m = n$ shows that there is a minimal spanning tree that contains all the edges of K, and therefore agrees with K. (Note that adding any edge to a spanning tree creates a cycle, so any spanning tree that does contain all the edges of K must be K itself). In other words, once we have completed our induction, we will have shown that K is a minimal spanning tree.

We must therefore establish the inductive step, by showing that if $m < n$ and T is a minimal spanning tree that contains the edges e_i with $i \le m$, then there is a minimal spanning tree U with these edges and e_{m+1}. If e_{m+1} already belongs to T, we can simply set $U = T$, so we shall also suppose that e_{m+1} is not an edge of T. The connected network T certainly contains a multi-edge path, P say, linking the endpoints of e_{m+1}. Our earlier observation shows that not all edges of P can be considered before e_{m+1} by Kruskal's algorithm (otherwise, the algorithm would have had to reject e_{m+1}). Let f be an edge of P that is considered after e_{m+1} by the algorithm. Then f must be at least as expensive as e_{m+1} (because Kruskal's algorithm considers cheaper edges before more expensive ones). The spanning tree $U = T - f + e_{m+1}$ is at least as cheap as T, and therefore it is also a minimal spanning tree with the requisite properties. This completes our induction.

E7. *Dijkstra's algorithm to compute a minimal spanning tree in a network works by considering all edges in any convenient order. As in Kruskal's algorithm, we select edges for a spanning tree, by adding edges to an initially empty set. However, each edge is now selected as it is considered, but if it creates a cycle together with the previously selected edges, the most expensive edge in this cycle is deselected. Prove that the edges chosen by Dijkstra's algorithm also form a minimal spanning tree of a connected network.*

Answer We begin with the following observation about a connected network G:

Observation *Suppose that the vertices of G are partitioned into two sets X and Y and that in G there is a unique cheapest edge, e say, that joins X to Y. Then e belongs to every minimal spanning tree of G.*

Proof If this is not the case, and T is a spanning tree that does not include e, then the graph S obtained by adding e to T includes a cycle through e. In this cycle e is cheaper than at least one other edge f (where f is another edge of the cycle that joins X to Y). We can now delete f from S to obtain a cheaper minimal spanning tree than T: a contradiction. ■ *end of proof*

We remark that for certain partitions of the vertices of G, there may be no unique cheapest edge linking the pieces of the partition. In this case, of course, our observation makes no claim about edges that must be included in minimal spanning trees. It is also conceivable that there could be minimal spanning trees that contain edges that are not covered by our observation. The following lemma shows that neither of these limitations need be considered in the case where all edges of the network have different weights.

Lemma *Suppose that all the edges of a network have distinct weights. Then the network has a unique minimal spanning tree that consists of the set of edges e for which e is the cheapest edge linking the two parts of some vertex partition of G.*

Proof By our earlier observation, every edge with the stated property does belong to every minimal spanning tree. It is enough to show that no other edge belongs to any minimal spanning tree. Equivalently, since no spanning tree can properly contain any other spanning tree, it is enough to show that there is one minimal spanning tree whose edges all have the stated property. It is easy to see that this holds for the minimal spanning tree produced by Prim's algorithm, because each edge picked out by Prim's algorithm is the (unique) cheapest edge linking the set of currently selected vertices to the remaining set of vertices in the original network. ■ *end of proof*

In the situation where the edges of a network have distinct weights, Dijkstra's algorithm can never reject an edge which is cheaper than any other edge linking the two parts of a partition of the vertices of G. Therefore, by our Lemma, Dijkstra's algorithm must end up with the edges of the only minimal spanning tree of the network.

In the situation where the edge weights of the network G are not necessarily distinct, we must extend the original argument of Dijkstra (which we have summarized above). We claim that even in the more general case, Dijkstra's algorithm does always produce a minimal spanning tree, for suppose on the contrary that G is a network with a minimal spanning tree S that is cheaper than a spanning tree T selected by some application of Dijkstra's algorithm.

We label the edges of G as $e_1, e_2, \ldots, e_r, e_{r+1}, e_{r+2}, \ldots, e_s$ in such a way that $e_{r+1}, e_{r+2}, \ldots, e_s$ are the edges of T and that the application of Dijkstra's algorithm that produces T rejects the other edges of G in the order $e_1, e_2, \ldots, e_r$. (This labelling serves to reflect the particular ordering of edges of G that has led to our hypothetical failure of Dijkstra's algorithm. Moreover, it describes the "random" decision that is occasionally required when the algorithm has to decide which of two equally bad edges of a cycle will be rejected.)

We select a very small positive quantity ϵ that is less than half the difference between the cost of S and the cost of T and is also less than the smallest positive difference between the weights of edges of G. We now construct a new network G' from G by using the same underlying graph, but by increasing the weight of each edge e_i by $\epsilon/2^i$, $1 \le i \le s$. We first observe that the edges of G' have distinct weights. (No two edges that had identical weights in G can have identical weights in G' since we have incremented the edge weights by distinct amounts. Moreover, since ϵ is smaller than the difference between distinct edge weights of G, no weight difference between other edges of G can be compensated for by the addition of fractional parts of ϵ to the edge weights.) A similar argument shows that the relative weighting of two edges can not be reversed in passing from G to G'. Therefore, in an application with the same edge ordering Dijkstra's algorithm would reject exactly the same edges from G' that it rejects from G. (A little care is needed, in checking cases where the algorithm must decide which of two or more equally bad edges of G should be rejected. We chose to increment the edge weights so that the most expensive of the edges in a cycle of G' is the edge of G that is actually rejected.)

We deduce from Dijkstra's original analysis that T is a minimal spanning tree of G'. However, the weight of the spanning tree S in G' exceeds its weight in G by less than $\epsilon = \epsilon\sum_1^\infty 1/2^i$. Hence, our choice of ϵ shows that the weight of S in G' is less than the weight of the minimal spanning tree T for G': a contradiction.

The original reference for Dijkstra's algorithm for minimal spanning trees is

E. W. Dijkstra, "Some theorems on spanning subtrees of a graph," *Indagationes Mathematicæ* 28 (1960), 196–199.

Programming Projects 12.7

P1. *Write* Digraph *methods called* read *that will read from the terminal the number of vertices in an undirected graph and lists of adjacent vertices. Be sure to include error checking. The graph is to be implemented with*

(a) *an adjacency table;*

Answer

```
template <int graph_size>
void Digraph<graph_size>::read()
/*
Post:  A user specified Digraph has been read from the terminal
*/

{
   cout << "How many vertices are in the digraph "
        << "(answer between 1 and " << graph_size << ")? " << flush;
   cin >> count;

   cout << "For each vertex, give the vertices to which it points."
        << endl;
   cout << "Type the numbers (terminated by a : for each vertex), "
        << "separate numbers by blanks." << endl;

   for (Vertex v = 0; v < count; v++) {
      for (Vertex w = 0; w < count; w++) adjacency[v][w] = false;
      char c;
      cout << "Vertex " << v << "  : " << flush;

      do {
         Vertex w;
         c = cin.get();
         if ('0' <= c && c <= '9') {
            w = c - '0';
            while ((c = cin.get()) >= '0' && c <= '9')
               w = 10 * w + c - '0';
            if (0 <= w && w < count) adjacency[v][w] = true;
         }
      } while (c != ':');
   }
}
```

(b) *a linked vertex list with linked adjacency lists;*

Answer

```
void Digraph::read()
/*
Post:  A user specified Digraph has been read from the terminal
*/

{
   cout << "How many vertices are in the digraph? " << flush;
   cin >> count;

   cout << "For each vertex, give the vertices to which it points."
        << endl;
   cout << "Type the numbers (terminated by a : for each vertex), "
        << "separate numbers by blanks." << endl;
```

```
      for (Vertex v = 0; v < count; v++) {
         List<int> points_to;
         int degree = 0;
         char c;
         cout << "Vertex " << v << "   : " << flush;

         do {
            c = cin.get();
            if ('0' <= c && c <= '9') {
               Vertex w = c - '0';
               while ((c = cin.get()) >= '0' && c <= '9')
                  w = 10 * w + c - '0';

               if (0 <= w && w < count)
                  points_to.insert(degree++, w);
               else
                  cout << "That is an illegal vertex" << endl;
            }
         } while (c != ':');
         neighbors.insert(v, points_to);
      }
   }
```

(c) *a contiguous vertex list of linked adjacency lists.*

Answer

```
template <int graph_size>
void Digraph<graph_size>::read()
/*
Post:  A user specified Digraph has been read from the terminal
*/

{
   cout << "How many vertices are in the digraph "
        << "(answer between 1 and " << graph_size << ")? " << flush;
   cin >> count;

   cout << "For each vertex, give the vertices to which it points."
        << endl;
   cout << "Type the numbers (terminated by a : for each vertex), "
        << "separate numbers by blanks." << endl;

   for (Vertex v = 0; v < count; v++) {
      int degree = 0;
      char c;
      cout << "Vertex " << v << "   : " << flush;

      do {
         c = cin.get();
         if ('0' <= c && c <= '9') {
            Vertex w = c - '0';

            while ((c = cin.get()) >= '0' && c <= '9')
               w = 10 * w + c - '0';

            if (0 <= w && w < count)
               neighbors[v].insert(degree++, w);
            else
               cout << "That is an illegal vertex" << endl;
         }
      } while (c != ':');
   }
}
```

P2. *Write* Digraph *methods called* write *that will write pertinent information specifying a graph to the terminal. The graph is to be implemented with*

(a) *an adjacency table;*

Answer

```
template <int graph_size>
void Digraph<graph_size>::write()
/*
Post:  The Digraph has been printed to the terminal
*/
{
   for (Vertex v = 0; v < count; v++) {
      cout << "Vertex " << v << " has succesors: ";
      for (Vertex w = 0; w < count; w++)
         if (adjacency[v][w])
            cout << w << " ";
      cout << "\n";
   }
}
```

A driver proram is:

```
#include "../../c/utility.h"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

#include "grapha.h"
#include "grapha.cpp"

main()
/*
Post: Tests Projects P1(a) and P2(a)
*/
{
   cout << "This program reads in, stores, and prints out a graph.\n"
        << "An adjacency table is used to store the graph." << endl;
   Digraph<12> g;
   cout << "\n\n Please enter a digraph: \n" << endl;
   g.read();
   cout << "\n\n The Digraph you entered is: \n" << endl;
   g.write();
}
```

(b) *a linked vertex list with linked adjacency lists;*

Answer

```
void Digraph::write()
/*
Post:  The Digraph has been printed to the terminal
*/
{
   for (Vertex v = 0; v < count; v++) {
      List<int> points_to;
      neighbors.retrieve(v, points_to);
      int degree = points_to.size();
      cout << "Vertex " << v << "  : ";

      if (degree == 0) cout << " has no successors" << endl;
      else cout << "has successors: ";
      for (int j = 0; j < degree; j++) {
         Vertex w;
         points_to.retrieve(j, w);
         cout << w;
         if (j < degree - 1) cout << ", ";
         else cout << "\n";
      }
   }
}
```

A driver proram is:

```
#include "../../c/utility.h"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

#include "graphb.h"
#include "graphb.cpp"

main()
/*
Post: Tests Projects P1(b) and P2(b)
*/
{
   cout << "This program reads in, stores, and prints out a graph.\n"
        << "Linked vertex and edge lists are used to store the graph."
        << endl;
   Digraph g;
   cout << "\n\n Please enter a digraph: \n" << endl;
   g.read();
   cout << "\n\n The Digraph you entered is: \n" << endl;
   g.write();
}
```

(c) *a contiguous vertex list of linked adjacency lists.*

Answer

```
template <int graph_size>
void Digraph<graph_size>::write()
/*
Post:  The Digraph has been printed to the terminal
*/
{
   for (Vertex v = 0; v < count; v++) {
      int degree = neighbors[v].size();
      cout << "Vertex " << v << "  : ";

      if (degree == 0) cout << " has no successors" << endl;
      else cout << "has successors: ";

      for (int j = 0; j < degree; j++) {
         Vertex w;
         neighbors[v].retrieve(j, w);
         cout << w;
         if (j < degree - 1) cout << ", ";
         else cout << "\n";
      }
   }
}
```

A driver proram is:

```
#include "../../c/utility.h"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

#include "graphc.h"
#include "graphc.cpp"

main()
/*
Post: Tests Projects P1(c) and P2(c)
*/

{
   cout << "This program reads in, stores, and prints out a graph.\n"
        << "A contiguous vertex list of linked adjacency lists \n"
        << "             is used to store the graph." << endl;
   Digraph<10> g;
   cout << "\n\n Please enter a digraph: \n" << endl;
   g.read();
   cout << "\n\n The Digraph you entered is: \n" << endl;
   g.write();
}
```

P3. *Use the methods* read *and* write *to implement and test the topological sorting functions developed in this section for*

(a) *depth-first order and*
(b) *breadth-first order.*

Answer Main program:

```
#include "../../c/utility.h"
#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
#include "graph.h"
#include "graph.cpp"

int main()
/*
Post: Tests the sorting methods for Project P3 of 12.7
*/

{
   Digraph<10> g;
   List<int> top_order;
   cout << "Testing topological sorting methods. " << endl << endl;
   cout << "Enter a graph as input for topological sorting: " << endl;
   g.read();

   cout << "The graph is: " << endl;
   g.write();

   cout << "Applying a depth-first sort.  With outcome:" << endl;
   g.depth_sort(top_order);
   int i;
   for (i = 0; i < top_order.size(); i++) {
      int j;
      top_order.retrieve(i, j);
      cout << j << " ";
   }
   cout << endl;
   cout << "Applying a breadth-first sort.  With outcome:" << endl;
   g.breadth_sort(top_order);
   cout << endl;
   for (i = 0; i < top_order.size(); i++) {
      int j;
      top_order.retrieve(i, j);
      cout << j << " ";
   }
   cout << endl;
}
```

The class Digraph and its sorting methods are implemented in:

```
typedef int Vertex;

template <int graph_size>
class Digraph {
public:
   Digraph();
   void read();
   void write();

//  methods to do a topological sort
   void depth_sort(List<Vertex> &topological_order);
   void breadth_sort(List<Vertex> &topological_order);

private:
   int count;
   List <Vertex> neighbors[graph_size];
   void recursive_depth_sort(Vertex v, bool visited[],
                             List<Vertex> &topological_order);
};


template <int graph_size>
Digraph<graph_size>::Digraph()
{
   count = 0;
}

template <int graph_size>
void Digraph<graph_size>::read()
{
   cout << "How many vertices in the digraph (between 1 and "
        << graph_size << ")? " << flush;
   cin >> count;

   cout << "For each vertex, give the vertices to which it points." << endl;
   cout << "Type the numbers (terminated by a : for each vertex),"
        << " separated by blanks." << endl;
   for (Vertex v = 0; v < count; v++) {
      int degree = 0;
      char c;
      cout << "Vertex " << v << "  : " << flush;
      do {
         c = cin.get();
         if ('0' <= c && c <= '9') {
            Vertex w = c - '0';
            while ((c = cin.get()) >= '0' && c <= '9')
               w = 10 * w + c - '0';
            if (0 <= w  && w < count)
               neighbors[v].insert(degree++, w);
         }
      } while (c != ':');
   }
}

template <int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool *visited,
                          List<Vertex> &topological_order)
/*
Pre:  Vertex v of the Digraph does not belong to
     the partially completed List topological_order.
Post: All the successors of v and finally v itself are added to
      topological_order with a depth-first search.
Uses: Methods of class List and the function recursive_depth_sort.
*/

{
   visited[v] = true;
   int degree = neighbors[v].size();

   for (int i = 0; i < degree; i++) {
      Vertex w;                        //  A (neighboring) successor of v
      neighbors[v].retrieve(i, w);
      if (!visited[w])                 //  Order the successors of w.
         recursive_depth_sort(w, visited, topological_order);
   }
   topological_order.insert(0, v);    //  Put v into topological_order.
}

template <int graph_size>
void Digraph<graph_size>::depth_sort(List<Vertex> &topological_order)
/*
Post: The vertices of the Digraph are placed into
      List topological_order with a depth-first traversal
      of those vertices that do not belong to a cycle.
Uses: Methods of class List, and function recursive_depth_sort
      to perform depth-first traversal.
*/

{
   bool visited[graph_size];
   Vertex v;
   for (v = 0; v < count; v++) visited[v] = false;
   topological_order.clear();
   for (v = 0; v < count; v++)
      if (!visited[v])  //  Add v and its successors into topological order.
         recursive_depth_sort(v, visited, topological_order);
}
```

```
template <int graph_size>
void Digraph<graph_size>::write()
{
   for (Vertex v = 0; v < count; v++) {
      int degree = neighbors[v].size();
      cout << "Vertex " << v << "  : ";
      if (degree == 0) cout << " has no successors" << endl;
      else cout << "has successors: ";
      for (int j = 0; j < degree; j++) {
         Vertex w;
         neighbors[v].retrieve(j, w);
         cout << w;
         if (j < degree - 1) cout << ", ";
         else cout << "\n";
      }
   }
}

typedef Vertex Queue_entry;
#include "../../3/queue/queue.h"
#include "../../3/queue/queue.cpp"

template <int graph_size>
void Digraph<graph_size>::breadth_sort(List<Vertex> &topological_order)
/*
Post: The vertices of the Digraph are arranged into the List
      topological_order which is found with a breadth-first
      traversal of those vertices that do not belong to a cycle.
Uses: Methods of classes Queue and List.
*/

{
   topological_order.clear();
   Vertex v, w;
   int predecessor_count[graph_size];

   for (v = 0; v < count; v++) predecessor_count[v] = 0;
   for (v = 0; v < count; v++)
      for (int i = 0; i < neighbors[v].size(); i++) {
                                    //  Loop over all edges v -- w.
         neighbors[v].retrieve(i, w);
         predecessor_count[w]++;
      }

   Queue ready_to_process;
   for (v = 0; v < count; v++)
      if (predecessor_count[v] == 0)
         ready_to_process.append(v);

   while (!ready_to_process.empty()) {
      ready_to_process.retrieve(v);
      topological_order.insert(topological_order.size(), v);
      for (int j = 0; j < neighbors[v].size(); j++) {
                                   //  Traverse successors of v.
         neighbors[v].retrieve(j, w);
         predecessor_count[w]--;
         if (predecessor_count[w] == 0)
            ready_to_process.append(w);
      }
      ready_to_process.serve();
   }
}
```

P4. *Write* Digraph *methods called* read *and* write *that will perform input and output for the implementation of Section 12.5. Make sure that the method* write() *also applies to the derived* **class** Network *of Section 12.6.*

Answer See the solution to Project P5 following.

P5. *Implement and test the method for determining shortest distances in directed graphs with weights.*

Answer Driver program:

```
#include "../../c/utility.h"
const int infinity = 1000000;
#include "graph.h"
#include "graph.cpp"

main()
{
   int d[10];
   Digraph<int, 10> g;
   cout << "Testing the method for shortest paths." << endl << endl;
   cout << "Enter a network.  The minimal weights of paths "
        << "from vertex 0 to other vertices \n  will be found."
        << endl;
   g.read();
   cout << "\nYou entered the graph:\n\n";
   g.write();

   cout << "The minimal distances from vertex 0 to the other vertices "
        << "are as follows:" << endl;

   g.set_distances(0, d);
   for (int i = 0; i < g.get_count(); i++)
     if (d[i] < infinity)
       cout << i << "@" << d[i] << ", ";
     else
       cout << i << " X, ";
   cout << endl;
}
```

A Digraph implementation with the required methods is:

```
typedef int Vertex;

template <class Weight, int graph_size>
class Digraph {
public:
   Digraph();
   void read();
   void write();
   int get_count();
   void set_distances(Vertex source, Weight distance[]) const;

protected:
   int count;
   Weight adjacency[graph_size][graph_size];
};
```

```
template <class Weight, int graph_size>
int Digraph<Weight, graph_size>::get_count()
{
   return count;
}

template <class Weight, int graph_size>
Digraph<Weight, graph_size>::Digraph()
{
   count = 0;
}

template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::read()
{
   cout << "How many vertices in the digraph (between 1 and "
        << graph_size << ")? " << flush;
   cin >> count;

   cout << "For each vertex, give the vertices to which it points." << endl;
   cout << " and the corresponding directed edge weight." << endl;

   cout << "Type the pairs of numbers (terminated by a : for each vertex),"
        << "  separated by blanks." << endl;

   for (Vertex v = 0; v < count; v++) {
      for (Vertex w = 0; w < count; w++)
         adjacency[v][w] = infinity;
      char c;
      cout << "Vertex " << v << "  : " << flush;
      do {
         Vertex w;
         c = cin.get();
         if ('0' <= c && c <= '9') {
            w = c - '0';
            while ((c = cin.get()) >= '0' && c <= '9')
               w = 10 * w + c - '0';
            while (c < '0' || c > '9') c = cin.get();
            Weight wt = c - '0';
            while ((c = cin.get()) >= '0' && c <= '9')
               wt = 10 * wt + c - '0';
            if (0 <= w && w < count) adjacency[v][w] = wt;
         }
      } while (c != ':');
   }
}

template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::write()
{
   for (Vertex v = 0; v < count; v++) {
      cout << "Vertex " << v << "  : ";
      for (Vertex w = 0; w < count; w++)
         if (adjacency[v][w] != 0 && adjacency[v][w] < infinity)
            cout << "(" << w << "^" << adjacency[v][w] << ") ";
      cout << "\n";
   }
}
```

```
template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::set_distances(Vertex source,
                                                Weight distance[]) const
/*
Post: The array distance gives the minimal path weight from vertex source
      to each vertex of the Digraph.
*/

{
   Vertex v, w;
   bool found[graph_size];   //  Vertices found in S
   for (v = 0; v < count; v++) {
      found[v] = false;
      distance[v] = adjacency[source][v];
   }

   found[source] = true; // Initialize with vertex source alone in set S.
   distance[source] = 0;

   for (int i = 0; i < count; i++) { //  Add one vertex v to S on each pass.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!found[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }

      found[v] = true;
      for (w = 0; w < count; w++) if (!found[w])
         if (min + adjacency[v][w] < distance[w])
             distance[w] = min + adjacency[v][w];
   }
}
```

P6. *Implement and test the methods of Prim, Kruskal, and Dijkstra for determining minimal spanning trees of a connected network.*

Answer The implementation of a Network class including Prim's algorithm is:

```
template <class Weight>
struct Edge {
   Vertex v, w;
   Weight wt;
   Edge(Vertex a = 0, Vertex b = 0, Weight x = infinity);
};


template <class Weight, int graph_size>
class Network: public Digraph<Weight, graph_size> {
public:
   Network();
   void read();   //  overridden method to enter a Network
   void make_empty(int size = 0);
   void add_edge(Vertex v, Vertex w, Weight x);
   void add_edge(Edge<Weight> e);
   void prim(Vertex source, Network<Weight, graph_size> &tree) const;
   void kruskal(Network<Weight, graph_size> &tree) const;
   void dijkstra(Network<Weight, graph_size> &tree) const;
private:
   void list_edges(Sortable_list<Edge<Weight> > &the_edges) const;
};

template <class Weight>
Edge<Weight>::Edge(Vertex a, Vertex b, Weight x) {
   v = a;
   w = b;
   wt = x;
}

template <class Weight>
bool operator == (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt == x.wt;
}

template <class Weight>
bool operator != (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt != x.wt;
}

template <class Weight>
bool operator >= (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt >= x.wt;
}

template <class Weight>
bool operator <= (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt <= x.wt;
}

template <class Weight>
bool operator >  (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt >  x.wt;
}

template <class Weight>
bool operator <  (const Edge<Weight> &y, const Edge<Weight> &x)
{
   return y.wt <  x.wt;
}

template <class Weight, int graph_size>
void Network<Weight, graph_size>
     ::add_edge(Edge<Weight> e)
{
   add_edge(e.v, e.w, e.wt);
}

template <class Weight, int graph_size>
Network<Weight, graph_size>::Network()
{
}
```

```
template <class Weight, int graph_size>
void Network<Weight, graph_size>::prim(Vertex source,
                     Network<Weight, graph_size> &tree) const
/*
Post: The Network tree contains a minimal spanning tree for the
      connected component of the original Network
      that contains vertex source.
*/

{
   tree.make_empty(count);
   bool component[graph_size];   //  Vertices in set X
   Weight distance[graph_size];  //  Distances of vertices adjacent to X
   Vertex neighbor[graph_size];  //  Nearest neighbor in set X
   Vertex w;

   for (w = 0; w < count; w++) {
      component[w] = false;
      distance[w] = adjacency[source][w];
      neighbor[w] = source;
   }

   component[source] = true;     //  source alone is in the set X.
   for (int i = 1; i < count; i++) {
      Vertex v;                  //  Add one vertex v to X on each pass.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!component[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }

      if (min < infinity) {
         component[v] = true;
         tree.add_edge(v, neighbor[v], distance[v]);
         for (w = 0; w < count; w++) if (!component[w])
            if (adjacency[v][w] < distance[w]) {
               distance[w] = adjacency[v][w];
               neighbor[w] = v;
            }
      }
      else break;            //  finished a component in disconnected graph
   }
}

template <class Weight, int graph_size>
void Network<Weight, graph_size>::read()
{
   int size;
   cout << "How many vertices in the Network (between 1 and "
        << graph_size << ")? " << flush;
   cin >> size;
   make_empty(size);

   cout << "For each vertex, give the later adjacent vertices." << endl;
   cout << " and the corresponding edge weight." << endl;

   cout << "Type the pairs of numbers (terminated by a : for each vertex),"
        << "  separated by blanks." << endl;
   for (Vertex v = 0; v < count; v++) {
      char c;
      cout << "Vertex " << v << "  : " << flush;
      do {
         Vertex w;
         c = cin.get();
         if (’0’ <= c && c <= ’9’) {
            w = c - ’0’;
            while ((c = cin.get()) >= ’0’ && c <= ’9’)
               w = 10 * w + c - ’0’;

            while (c < ’0’ || c > ’9’) c = cin.get();
            Weight wt = c - ’0’;

            while ((c = cin.get()) >= ’0’ && c <= ’9’)
               wt = 10 * wt + c - ’0’;
            if (0 <= w && w < size) add_edge(v, w, wt);
         }
      } while (c != ’:’);
   }
}

template <class Weight, int graph_size>
void Network<Weight, graph_size>::make_empty(int size = 0)
{
   count = size;
   for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++)
         adjacency[i][j] = infinity;
}

template <class Weight, int graph_size>
void Network<Weight, graph_size>::add_edge(Vertex v, Vertex w, Weight x)
{
   adjacency[v][w] = adjacency[w][v] = x;
}
```

Dijkstra’s algorithm is implemented in:

```
template <class Weight, int graph_size>
void Network<Weight, graph_size>::dijkstra(
     Network<Weight, graph_size> &tree) const
/*
Post: The Network tree contains minimal spanning trees for the
      connected components of the original Network.
*/

{
   Sortable_list<Edge<Weight> > the_edges;
   list_edges(the_edges);
   tree.make_empty(count);

   Vertex component[graph_size];
   Edge<Weight> path[graph_size];
   Vertex x, y;
   for (x = 0; x < count; x++) component[x] = x;
```

TOC
Index
Help

```
while (!the_edges.empty()) {
   Edge<Weight> e;
   the_edges.retrieve(0, e);
   if (component[e.v] != component[e.w]) {
      x = e.w;
      Vertex fusing = component[x];

      List<Edge<Weight> > new_path;
      Edge<Weight> reverse_e(e.w, e.v, e.wt);
      new_path.insert(0, reverse_e);
      while (x != fusing) {
         Edge<Weight> f = path[x];
         Edge<Weight> reverse_f(f.w, f.v, f.wt);
         x = f.w;
         new_path.insert(0, reverse_f);
      }
      while (!new_path.empty()) {
         Edge<Weight> ff;
         new_path.retrieve(0, ff);
         path[ff.v] = ff;
         new_path.remove(0, ff);
      }

      for (x = 0; x < count; x++) if (component[x] == fusing)
         component[x] = component[e.v];
   }
   else {
      x = e.v;
      y = e.w;
      Vertex center = component[x];
      List<Edge<Weight> > x_path, y_path;
      while (x != center) {
         x_path.insert(0, path[x]);
         x = (path[x]).w;
      }

      while (y != center) {
         y_path.insert(0, path[y]);
         y = (path[y]).w;
      }

      if (x_path.size() > 0 && y_path.size() > 0) do {
         Edge<Weight> e_x, e_y;
         x_path.retrieve(0, e_x);
         y_path.retrieve(0, e_y);
         x = e_x.w;
         y = e_y.w;
         if (x == y) {
            x_path.remove(0, e_x);
            y_path.remove(0, e_y);
         }
      } while (x == y);
```

```
         Edge<Weight> worst = e, temp;
         int position = -1;
         char path_name = ' ';
         for (int i = 0; i < x_path.size(); i++) {
            x_path.retrieve(i, temp);
            if (temp > worst) {
               worst = temp;
               position = i;
               path_name = 'x';
            }
         }

         for (int j = 0; j < y_path.size(); j++) {
            y_path.retrieve(j, temp);
            if (temp > worst) {
               worst = temp;
               position = j;
               path_name = 'y';
            }
         }

         Edge<Weight> temp_reverse;
         if (path_name == 'x') {
            path[e.v] = e;
            for (int k = x_path.size() - 1; k > position; k--) {
               x_path.retrieve(k, temp);
               temp_reverse.v = temp.w;
               temp_reverse.w = temp.v;
               temp_reverse.wt = temp.wt;
               path[temp.w] = temp_reverse;
            }
         }

         if (path_name == 'y') {
            temp_reverse.v = e.w;
            temp_reverse.w = e.v;
            temp_reverse.wt = e.wt;
            path[e.w] = temp_reverse;

            for (int k = y_path.size() - 1; k > position; k--) {
               y_path.retrieve(k, temp);
               temp_reverse.v = temp.w;
               temp_reverse.w = temp.v;
               temp_reverse.wt = temp.wt;
               path[temp.w] = temp_reverse;
            }
         }
      }
      the_edges.remove(0, e);
   }
   for (x = 0; x < count; x++) {
      Edge<Weight> e = path[x];
      if (e.wt > 0 && e.wt < infinity)
        tree.add_edge(e);
   }
}
```

Kruskal's algorithm is implemented in:

```
#include "../../6/linklist/list.cpp"
#include "../../8/linklist/insert.cpp"
#include "../../8/linklist/merge.cpp"

template <class Weight, int graph_size>
void Network<Weight, graph_size>::kruskal(
     Network<Weight, graph_size> &tree) const
/*
Post: The Network tree contains minimal spanning tree for the
      connected components of the original Network.
*/

{
   Sortable_list<Edge<Weight> > the_edges;
   list_edges(the_edges);
   the_edges.merge_sort();
   tree.make_empty(count);

   Vertex component[graph_size];
   Vertex x;
   for (x = 0; x < count; x++) component[x] = x;

   while (!the_edges.empty()) {
      Edge<Weight> e;
      the_edges.retrieve(0, e);

      if (component[e.v] != component[e.w]) {
         tree.add_edge(e);
         Vertex fusing = component[e.w];
         for (x = 0; x < count; x++) if (component[x] == fusing)
            component[x] = component[e.v];
      }
      the_edges.remove(0, e);
   }
}

template <class Weight, int graph_size>
void Network<Weight, graph_size>
     ::list_edges(Sortable_list<Edge<Weight> > &the_edges) const
{
   for (Vertex v = 0; v < count; v++)
      for (Vertex w = v; w < count; w++)
         if (adjacency[v][w] != 0 && adjacency[v][w] < infinity) {
            Edge<Weight> e(v, w, adjacency[v][w]);
            the_edges.insert(0, e);
         }
}
```

A driver for the project is:

```
#include "../../c/utility.h"
const int infinity = 1000000;
#include "../7p4p5/graph.h"
#include "../7p4p5/graph.cpp"

#include "../../6/linklist/list.h"
#include "../../8/linklist/sortable.h"

#include "edge.h"
#include "graph.h"
#include "edge.cpp"
#include "graph.cpp"
```

```
#include "kruskal.cpp"
#include "dijkstra.cpp"

main()
{
   Network<int, 10> g;
   Network<int, 10> prim_tree;
   Network<int, 10> kruskal_tree;
   Network<int, 10> dijkstra_tree;
   List<Edge<int> > l;

   cout << "Testing the minimal spanning tree methods." << endl << endl;
   cout << "Enter a network. "
        << "Minimal spanning trees will be determined."
        << endl;
   g.read();
   cout << "\nYou entered the graph:\n\n";
   g.write();

   cout << "The minimal spanning tree from Prim's algorithm is: " << endl;
   g.prim(0, prim_tree);
   prim_tree.write();

   cout << "\n\nThe minimal spanning tree from Kruskal's algorithm is: "
        << endl;
   g.kruskal(kruskal_tree);
   kruskal_tree.write();

   cout << "\n\nThe minimal spanning tree from Dijkstra's algorithm is: "
        << endl;
   g.dijkstra(dijkstra_tree);
   dijkstra_tree.write();
}
```

Data files with various edge weightings for the Petersen graph are in the directories of executables.

REVIEW QUESTIONS

1. *In the sense of this chapter, what is a graph? What are edges and vertices?*

A ***graph*** is a set whose members are called ***vertices*** of the graph, together with a set of (unordered or ordered) pairs of vertices, called the ***edges*** of the graph.

2. *What is the difference between an undirected and a directed graph?*

If the pairs are unordered, the graph is ***undirected***; if they are ordered, it is ***directed***.

3. *Define the terms adjacent, path, cycle, and connected.*

Two vertices in a graph are ***adjacent*** if there is an edge from the first vertex to the second. A ***path*** is a sequence of distinct vertices, each adjacent to the next. A ***cycle*** is a path containing at least three vertices in which the last vertex is adjacent to the first. A graph is ***connected*** if there is a path from any vertex to any other vertex in the graph.

4. *What does it mean for a directed graph to be strongly connected? Weakly connected?*

It is strongly connected if there is a path from any vertex to any other vertex which always follows the direction given by the ordering of each edge. It is weakly connected if there is a path between any two vertices when the ordering of the edges is ignored.

5. *Describe three ways to implement graphs in computer memory.*

Graphs are often implemented as adjacency tables, as linked structures, and as contiguous or linked adjacency lists.

6. *Explain the difference between depth-first and breadth-first traversal of a graph.*

Depth-first traversal continues to visit the first unvisited successor of each node as long as possible and then moves to the next unvisited successor of the original node. Breadth-first traversal first visits all the immediate successors of a node before moving to their successors.

7. *What data structures are needed to keep track of the waiting vertices during* **(a)** *depth-first and* **(b)** *breadth-first traversal?*

A stack or recursion is used for depth-first traversal; a queue is used for breadth-first traversal.

8. *For what kind of graphs is topological sorting defined?*

It is defined for directed graphs in which there are no directed cycles.

9. *What is a topological order for such a graph?*

A topological order is a (sequential) listing of all the vertices of the directed graph such that, if there is a (directed) edge from vertex v to vertex w, then v comes before w in the listing.

10. *Why is the algorithm for finding shortest distances called greedy?*

The algorithm tries to reach its goal by making the choice that gives it the greatest immediate gain, without regard to long-term benefits.

11. *Explain how Prim's algorithm for minimal spanning trees differs from Kruskal's algorithm.*

Both algorithms both work by adding edges to an initially empty set until the set of edges gives a minimal spanning tree. Both algorithms use a greedy condition to select the edge to add at any given stage. However, Prim's algorithm uses a greedy condition on vertices, and adds the cheapest edge to link a new vertex to the set of edges under construction, whereas Kruskal's algorithm uses a greedy condition on edges and adds the cheapest edge that does not create a cycle.

TOC
Index
Help

Case Study: The Polish Notation

13

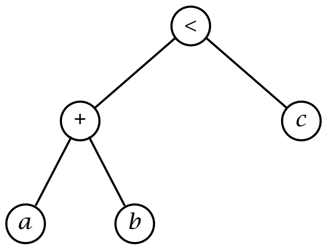
13.2 THE IDEA

Exercises 13.2

(a) *Draw the expression tree for each of the following expressions. Using the tree, convert the expression into* **(b)** *prefix and* **(c)** *postfix form. Use the table of priorities developed in this section, not those in C++.*

E1. $a + b < c$

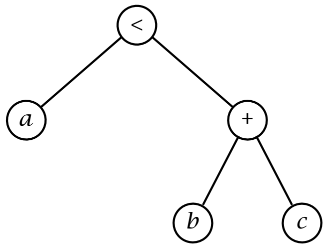
Answer **(a)**



(b) Prefix: $< \; + \; a \; b \; c$
(c) Postfix: $a \; b \; + \; c \; <$

E2. $a < b + c$

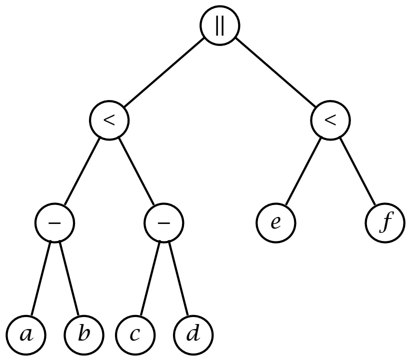
Answer **(a)**



(b) Prefix: $< \; a \; + \; b \; c$
(c) Postfix: $a \; b \; c \; + \; <$

E3. $a - b < c - d \;||\; e < f$

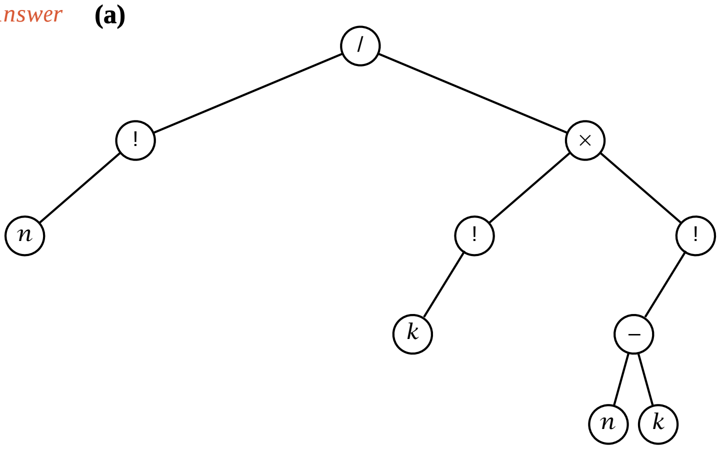
Answer **(a)**



(b) Prefix: $|| \; < \; - \; a \; b \; - \; c \; d \; < \; e \; f$
(c) Postfix: $a \; b \; - \; c \; d \; - \; < \; e \; f \; < \; ||$

E4. $n!\,/\,(k! \times (n-k)!)$ *(formula for binomial coefficients)*

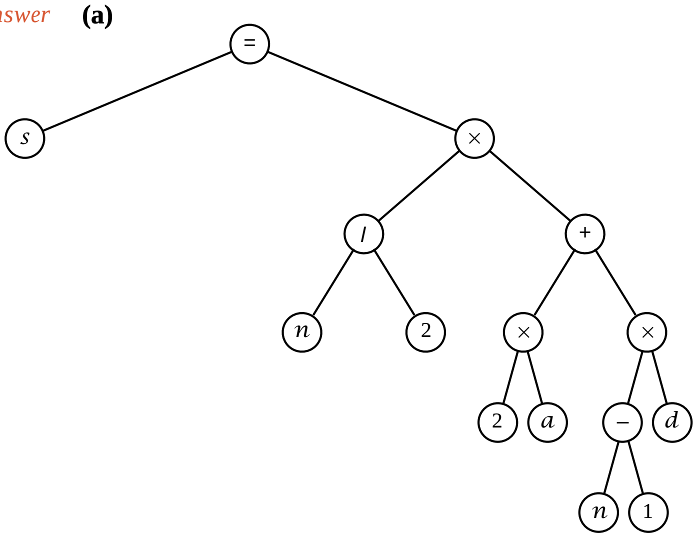
Answer **(a)**



(b) Prefix: $/ \; ! \; n \times \; ! \; k \; ! \; - \; n \; k$
(c) Postfix: $n \; ! \; k \; ! \; n \; k \; - \; ! \; * \; /$

E5. $s = (n/2)\times(2 \times a + (n-1)\times d)$ *(This is the sum of the first n terms of an arithmetic progression.)*

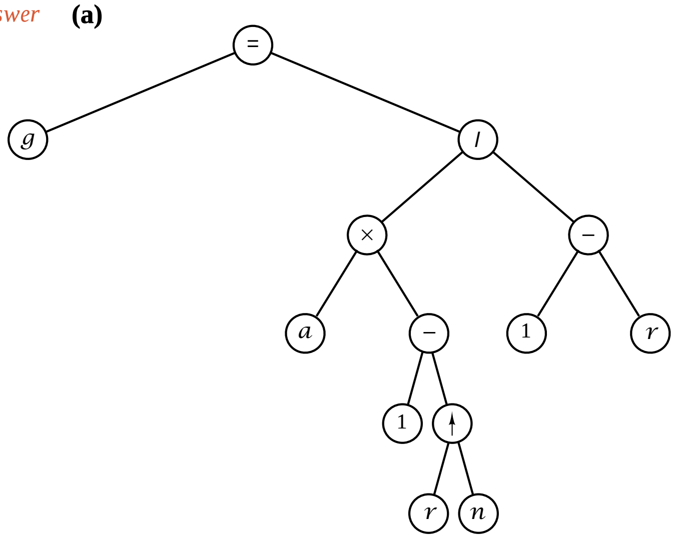
Answer **(a)**



(b) Prefix: $= s \times / \, n \, 2 + \times 2 \, a \times - n \, 1 \, d$
(c) Postfix: $s \, n \, 2 \, / \, 2 \, a \times n \, 1 - d \times + \times =$

E6. $g = a \times (1 - r^n)/(1-r)$ *(sum of first n terms of a geometric progression)*

Answer **(a)**



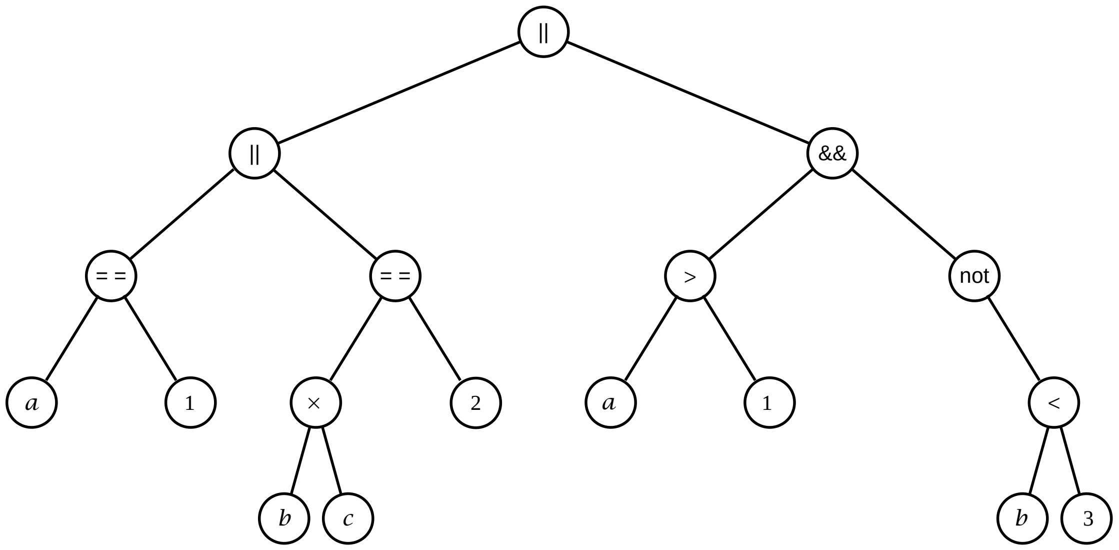
(b) Prefix: $= g \, / \times a - 1 \uparrow r \, n - 1 \, r$
(c) Postfix: $g \, a \, 1 \, r \, n \uparrow - \times 1 \, r - / =$

Note: The symbol ↑ represents exponentiation.

E7. a == 1 || $b \times c$ == 2 || ($a > 1$ && not $b < 3$)

Answer **(a)**

(b) Prefix: || || = a 1 = $\times$ b c 2 && > a 1 not < b 3
(c) Postfix: a 1 = b c $\times$ 2 = || a 1 > b 3 < not && ||



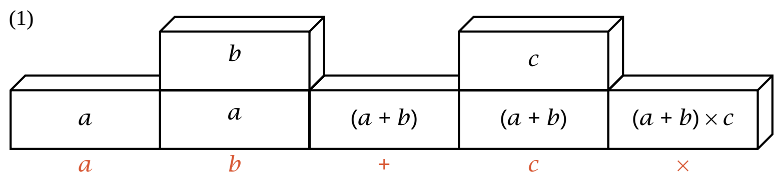
13.3 EVALUATION OF POLISH EXPRESSIONS

Exercises 13.3

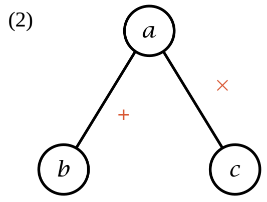
E1. *Trace the action on each of the following expressions by the function* evaluate_postfix *in* **(1)** *nonrecursive and* **(2)** *recursive versions. For the recursive function, draw the tree of recursive calls, indicating at each node which tokens are being processed. For the nonrecursive function, draw a sequence of stack frames showing which tokens are processed at each stage.*

(a) $a \quad b \quad + \quad c \quad \times$

Answer

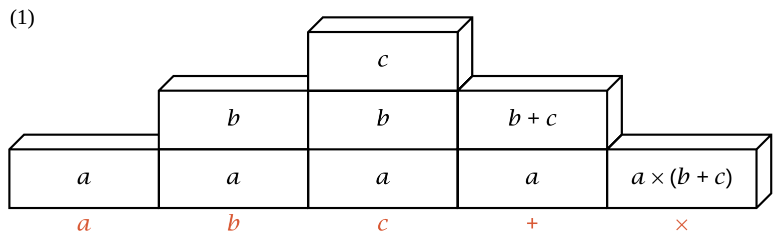


(a)

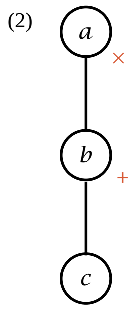


(b) $a \quad b \quad c + \quad \times$

Answer



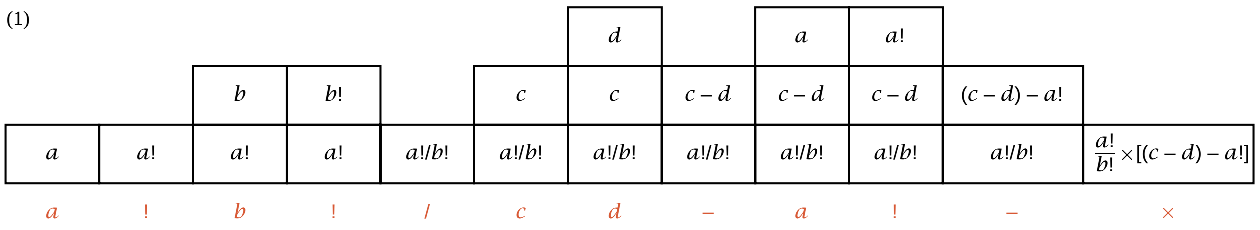
(b)



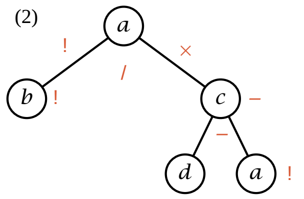
(c) $a \quad ! \quad b \quad ! \quad / \quad c \quad d \quad - \quad a \quad ! \quad - \quad \times$

Answer

(1)



(2)

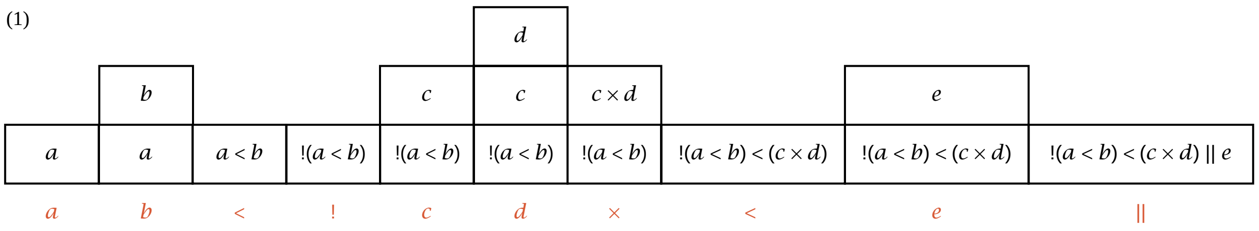


(c)

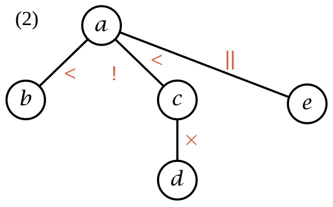
(d) $a \quad b \quad < \quad ! \quad c \quad d \quad \times \quad < \quad e \quad ||$

Answer

(1)



(2)

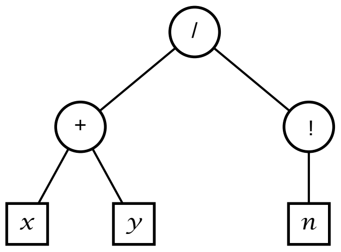


(d)

E2. *Trace the action of the function* evaluate_prefix *on each of the following expressions by drawing a tree of recursive calls showing which tokens are processed at each stage.*

(a) $/ \quad + \quad x \quad y \quad ! \quad n$

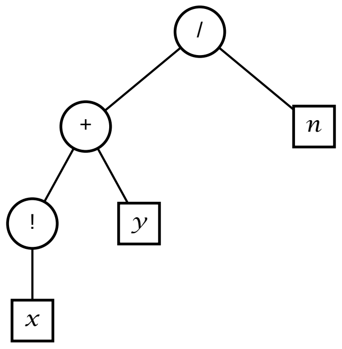
Answer



(a)

(b) $/ \quad + \quad ! \quad x \quad y \quad n$

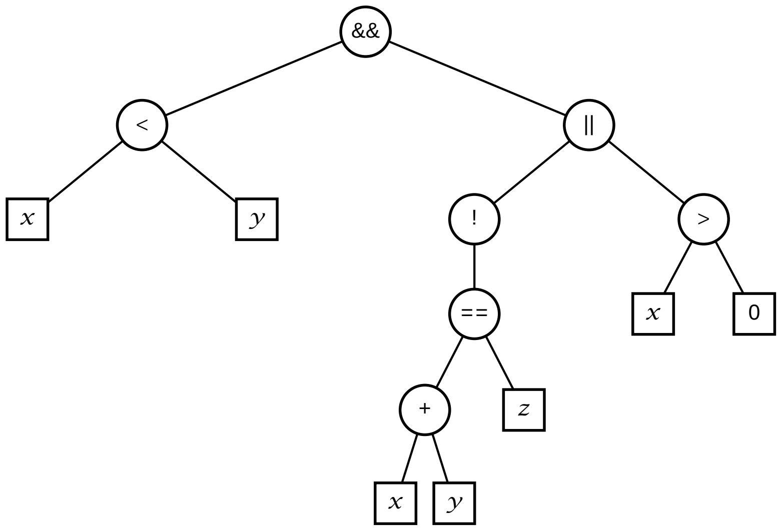
Answer



(b)

(c) $\&\& \quad < \quad x \quad y \quad || \quad ! \quad = \quad + \quad x \quad y \quad z \quad > \quad x \quad 0$

Answer



(c)

E3. *Which of the following are syntactically correct postfix expressions? Show the error in each incorrect expression. Translate each correct expression into infix form, using parentheses as necessary to avoid ambiguities.*

(a) $a \quad b \quad c \quad + \quad \times \quad a \quad / \quad c \quad b \quad + \quad d \quad / \quad -$

Answer $((a \times (b + c)) / a) - ((c + b) / d)$

(b) $a \quad b \quad + \quad c \quad a \quad \times \quad b \quad c \quad / \quad d \quad -$

Answer The expression is not in valid postfix form; it lacks two additional binary operators.

(c) $a \quad b \quad + \quad c \quad a \quad \times \quad - \quad c \quad \times \quad + \quad b \quad c \quad -$

Answer The expression is not in valid postfix form; the second $+$ does not have two operands.

(d) $a \quad \sim \quad b \quad \times$

Answer $(\sim a) \times b$

TOC
Index
Help

(e) $a \quad \times \quad b \quad \sim$

Answer The expression is not in valid postfix form; $\times$ does not have two operands.

(f) $a \quad b \quad \times \quad \sim$

Answer $\sim \quad (a \quad \times \quad b)$

(g) $a \quad b \quad \sim \quad \times$

Answer $a \quad \times \quad (\sim \quad b)$

E4. *Translate each of the following expressions from prefix form into postfix form.*

(a) $/ \quad + \quad x \quad y \quad ! \quad n$

Answer $x \quad y \quad + \quad n \quad ! \quad /$

(b) $/ \quad + \quad ! \quad x \quad y \quad n$

Answer $x \quad ! \quad y \quad + \quad n \quad /$

(c) $\&\& \quad < \quad x \quad y \quad || \quad ! \quad = \quad + \quad x \quad y \quad z \quad > \quad x \quad 0$

Answer $x \quad y \quad < \quad x \quad y \quad + \quad z \quad = \quad ! \quad x \quad 0 \quad > \quad || \quad \&\&$

E5. *Translate each of the following expressions from postfix form into prefix form.*

(a) $a \quad b \quad + \quad c \quad *$

Answer $* \quad + \quad a \quad b \quad c$

(b) $a \quad b \quad c \quad + \quad \times$

Answer $* \quad a \quad + \quad b \quad c$

(c) $a \quad ! \quad b \quad ! \quad / \quad c \quad d \quad - \quad a \quad ! \quad - \quad \times$

Answer $\times \quad / \quad ! \quad a \quad ! \quad b \quad - \quad - \quad c \quad d \quad ! \quad a$

(d) $a \quad b \quad < \quad ! \quad c \quad d \quad \times \quad < \quad e \quad ||$

Answer or $\quad < \quad ! \quad < \quad a \quad b \quad \times \quad c \quad d \quad e$

E6. *Formulate and prove theorems analogous to Theorems* **(a)** *13.1,* **(b)** *13.3, and* **(c)** *13.4 for the prefix form of expressions.*

Answer The running sum for prefix expressions becomes:

For a sequence E of operands, unary operators, and binary operators, form a running sum by starting at the left end of E and counting $+1$ for each operand, 0 for each unary operator, and -1 for each binary operator. E satisfies the ***prefix running-sum condition*** *provided that this running sum never goes above 0 except at the right-hand end of E, where it becomes $+1$.*

(a)

Theorem *If E is a properly formed expression in prefix form, then E must satisfy the prefix running-sum condition.*

TOC
Index
Help

Proof We use mathematical induction on the length of the expression E being evaluated. The starting point for the induction is the case that E is a single operand alone, with length 1. This operand contributes $+1$, and the running-sum condition is satisfied. For the induction hypothesis we now assume that E is a proper prefix expression of length more than 1 and that all shorter expressions satisfy the running-sum condition. Since the length of E is more than 1, E is constructed at its first step either as $op\ \ F$, where op is a unary operator and F a prefix expression, or as $op\ \ F\ \ G$, where op is a binary operator and F and G are prefix expressions.

In either case the lengths of F and G are less than that of E, so by induction hypothesis both of them satisfy the running-sum condition. First, take the case when op is a unary operator. Since F satisfies the running-sum condition, the sum at its end is exactly $+1$. As a unary operator, op contributes 0 to the sum, so the full expression E satisfies the running-sum condition. If, finally, op is binary, then the running sum begins at -1. On F alone the running sum would never go above 0 and would end at $+1$, so, combined with the -1 from op, the running sum never goes above -1 except to reach 0 at the end of F. By the induction hypothesis the running sum for G will never go above 0 except to end at $+1$. Hence the combined running sum for all of E never goes above 0 except to end at $+1$. ■

end of proof

(b)

Theorem *If E is any sequence of operands and operators that satisfies the prefix running-sum condition, then E is a properly formed expression in prefix form.*

Proof We again use mathematical induction. The starting point is an expression containing only one token. Since the running sum (same as final sum) for a sequence of length 1 will be 1, this one token must be a simple operand. One simple operand alone is indeed a syntactically correct expression. For the inductive step, suppose that the theorem has been verified for all expressions strictly shorter than E, and E has length greater than 1. If the first token of E were an operand, then the running sum would begin at $+1$, contrary to the condition. Thus the first token of E must be an operator. If the operator is unary, then it can be omitted and the remaining sequence still satisfies the condition on running sums, so by induction hypothesis is a syntactically correct expression, and all of E then also is. Finally suppose that the first token is a binary operator op. To show that E is syntactically correct, we must find where in the sequence the first operand of op ends and the second one starts, by using the running sum. This place will be the first place in sequence where the running sum reaches 0. Let F begin with the second token and end at this place, and let G be the remainder of the sequence. Since the running sums in $op\ \ F$ never go above -1 except to end at 0, F alone satisfies the prefix running-sum condition and so by induction hypothesis is a properly formed prefix expression. Since the running sum is 0 just before G starts, it too satisfies the condition and so is a properly formed prefix expression. Hence all of E is a correct prefix expression of the form $op\ \ F\ \ G$. ■

end of proof

(c)

Theorem *An expression in prefix form that satisfies the prefix running-sum condition corresponds to exactly one fully bracketed expression in infix form. Hence no parentheses are needed to achieve the unique representation of an expression in prefix form.*

Proof To establish this theorem we need only show that, in the proof of the preceding theorem, the first place where the running sum reaches 0 is the *only* place where the expression can be broken as $op\ \ F\ \ G$ with F and G also satisfying the prefix running-sum condition. For suppose it was split elsewhere. If at the end of $op\ \ F$ the running sum is not 0, then the running sum for F alone would not end at $+1$, so F is not a syntactically correct expression. If the running sum is 0 at the end of $op\ \ F$, but it had reached 0 previously, then at that point the running sum for F alone would be $+1$, which is not allowed before the end of F. We have now shown that there is only one way to recover the two operands of a binary operator. Clearly there is only one way to recover the single operand for a unary operator. Hence we can recover the infix form of an expression from its prefix form, together with the order in which the operations are done, which we can denote by bracketing the result of every operation in the infix form with another pair of parentheses. ■

end of proof

13.4 TRANSLATION FROM INFIX FORM TO POLISH FORM

Exercises 13.4

E1. *Devise a method to translate an expression from prefix form into postfix form. Use the C++ conventions of this chapter.*

Answer

```
Expression Expression :: prefix_to_postfix( )
/* Pre:  The Expression stores a valid prefix expression.
   Post: A postfix expression that translates the prefix expression is returned. */
{
   Expression answer;
   rec_prefix_to_postfix(answer);
   answer.put_token(";");
   return answer;
}
```

A recursive translation function is used to process prefix subexpressions from the input. The recursive function is coded as follows:

```
void Expression :: rec_prefix_to_postfix(Expression &answer)
/* Pre:  The Expression stores an initial segment giving a valid prefix expression (possibly empty).
   Post: A postfix expression that translates the prefix expression is appended to the output
         parameter answer. */
{
   Token current;
   if (get_token(current) != fail) {
      switch (current.kind( )) {
      case operand:
         answer.put_token(current);
         break;
      case unaryop:
         rec_prefix_to_postfix(answer);
         answer.put_token(current);
      case binaryop:
         rec_prefix_to_postfix(answer);
         rec_prefix_to_postfix(answer);
         answer.put_token(current);
      }
   }
}
```

E2. *Write a method to translate an expression from postfix form into prefix form. Use the C++ conventions of this chapter.*

Answer The method can use a Stack of Token data to reverse the input postfix expression. A recursive auxiliary function to convert the Stack into a prefix Expression is applied to complete the work.

```
Expression Expression :: postfix_to_prefix( )
/* Pre:  The Expression stores a valid postfix expression.
   Post: A prefix expression that translates the postfix expression is returned. */
{
   Expression answer;
   Token current;
   Stack reverse_input;                    //  Reverse of the input expression
   while (true) {
      get_token(current);
      if (current.kind( ) == end_expression) break;
      reverse_input.push(current);
   }
   stack_to_prefix(reverse_input, answer);
   return answer;
}
```

The auxiliary recursive function is implemented as follows:

```
void Expression :: stack_to_prefix(Stack &s, Expression &answer)
/* Pre:  The Stack s stores the reversed sequence of tokens from a valid postfix expression.
   Post: A prefix expression that translates the postfix expression is returned through the output
         parameter answer. */
{
   if (s.empty( )) return;
   Token current;
   s.top(current);
   s.pop( );
   switch (current.kind( )) {
      case operand:
         answer.put_token(current);
         break;
      case unaryop:
         answer.put_token(current);
         stack_to_prefix(s, answer);
      case binaryop:
         answer.put_token(current);
         Expression second_arg;
         stack_to_prefix(s, second_arg);
         stack_to_prefix(s, answer);
         while (second_arg.get_token(current) != fail)
            answer.put_token(current);
   }
}
```

E3. *A **fully bracketed** expression is one of the following forms:*

i. *a simple operand;*

ii. *$(op\ \ E)$ where op is a unary operator and E is a fully bracketed expression;*

iii. *$(E\ \ op\ \ F)$ where op is a binary operator and E and F are fully bracketed expressions.*

Hence, in a fully bracketed expression, the results of every operation are enclosed in parentheses. Examples of fully bracketed expressions are $((a+b)-c)$, $(-a)$, $(a+b)$, $(a+(b+c))$. Write methods that will translate expressions from (a) prefix and (b) postfix form into fully bracketed form.

Answer **(a)**
```
void Expression :: prefix_to_full(Expression &answer)
/* Pre:  The Expression begins with a valid prefix expression.
   Post: A fully bracketed expression that translates the prefix expression is appended to
         the output parameter answer. */
{
   Token current;
   if (get_token(current) != fail) {
      switch (current.kind( )) {
      case operand:
         answer.put_token(current);
         break;
```

```
        case unaryop:
          answer.put_token("(");
          answer.put_token(current);
          prefix_to_full(answer);
          answer.put_token(")");
        case binaryop:
          answer.put_token("(");
          prefix_to_full(answer);
          answer.put_token(current);
          prefix_to_full(answer);
          answer.put_token(")");
        }
      }
    }
```

(b)

```
void Expression :: postfix_to_full(Expression &answer)
/* Pre:  The Expression stores a valid postfix expression.
   Post: A fully bracketed expression that translates the prefix expression is returned through
         the output parameter answer. */
{
  Expression temp = postfix_to_prefix();
  temp.prefix_to_full(answer);
}
```

E4. *Rewrite the method* infix_to_postfix *as a recursive function that uses no stack or other array.*

Answer The recursive function is called by:

```
Expression Expression :: run_infix_to_postfix()
/* Pre:  The Expression stores a valid infix expression.
   Post: A postfix expression that translates the infix expression is returned. */
{
  Expression answer;
  Token first_token, last_token;
  if (get_token(first_token) != fail)
    rec_infix_to_postfix(answer, −1, first_token, last_token);
  answer.put_token(";");
  return answer;
}
```

The recursive function is coded as follows:

```
void Expression :: rec_infix_to_postfix(Expression &answer, int stop_priority,
        Token current, Token &stop_at)
/* Pre:  When appended to the Token current, the Expression begins with a valid infix expres-
         sion.
   Post: A postfix expression that translates the valid infix expression is appended to the output
         parameter answer. */
{
  bool done = false;
  Token next;
  do {
    if (empty()) done = true;                //  no more tokens to convert
    else
     switch (current.kind()) {
     case operand:
       answer.put_token(current);
       get_token(current);
       break;
```

```
        case rightparen:
            done = true;
            break;
        case leftparen:
            get_token(current);
            rec_infix_to_postfix(answer, −1, current, stop_at);
            if (stop_at.kind( ) != rightparen)
               cout << "Warning bracketing error detected" << endl;
            get_token(current);
            break;
        case unaryop:
        case binaryop:
            int p = current.priority( );
            if (p <= stop_priority) done = true;
            else {
              get_token(next);
              if (p == 6)
                rec_infix_to_postfix(answer, 5, next, stop_at);
              else
                if (empty( )) answer.put_token(next);
                else rec_infix_to_postfix(answer, p, next, stop_at);
              answer.put_token(current);
              current = stop_at;
            }
            break;
        }
    } while (!done);
    stop_at = current;
}
```

TOC
Index
Help

Programming Project 13.4

P1. *Construct a menu-driven demonstration program for Polish expressions. The input to the program should be an expression in any of infix, prefix, or postfix form. The program should then, at the user's request, translate the expression into any of fully bracketed infix, prefix, or postfix, and print the result. The operators should be single letters or digits only. The operators allowed are:*

binary:	+	−	∗	/	%	∧	:	<	>	&	|	=
left unary:	#	∼										
right unary:	!	′	"									

Use the priorities given in the table on page 601, not those of C++. In addition, the program should allow parentheses '(' and ')' in infix expressions only. The meanings of some of the special symbols used in this project are:

&	*Boolean* and *(same as* && *in C++)*	|	*Boolean* or *(same as* || *in C++)*
:	*Assign (same as = in C++)*	%	*Modulus (binary operator)*
∧	*Exponentiation (same as ↑)*	!	*Factorial (on right)*
′	*Derivative (on right)*	"	*Second derivative (on right)*
∼	*Unary negation*	#	*Boolean* not *(same as ! in C++)*

Answer Note that the problem is ambiguous about the priority of !, which is given as 6 since it is unary and 2 according to the table on page 600. We take ! to have priority 6. We take = to correspond to the C++ operator ==, and accordingly, we give it a priority of 3.

Main program:

```
#include "../../c/utility.h"
```

```
void help()
/*
PRE:  None.
POST: Instructions for the expression operations have been printed.
*/

{
     cout << "\n";
     cout << "Terminate expression input with a $, or newline, or ;\n";
     cout << "\tread [I]nfix Expression\n"
          << "\tread [F]ully bracketed infix Expression\n"
          << "\tread p[R]efix Expression\n"
          << "\tread p[O]stfix Expression\n"

          << "\tconvert to i[N]fix Expression\n"
          << "\tconvert to f[U]lly bracketed infix Expression\n"
          << "\tconvert to pr[E]fix Expression\n"
          << "\tconvert to po[S]tfix Expression\n"

          << "\t[W]rite expression [Q]uit  [?]help\n" << endl;
}

#include <string.h>

#include "polish.h"
#include "polish.cpp"
#include "valid.cpp"
#include "e1.cpp"
#include "e2.cpp"
#include "e3a.cpp"
#include "../4e1__e4/e3b.cpp"

char get_command()
/*
PRE:  None.
POST: A character command belonging to the set of legal commands for the
      expression demonstration has been returned.
*/

{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (1) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if(strchr("ironesq?wfu",c) != NULL)
         return c;
      cout << "Please enter a valid command or ? for help:" << endl;
      help();
   }
}

enum E_type{none, e_infix, e_fullfix, e_prefix, e_postfix} current_type;
```

```
int do_command(char c, Expression &current)
/*
PRE:  The Expression has been created and command is
      a valid expression operation.
POST: The command has been executed.
USES: All the functions that perform Expression operations.
*/

{
   Expression post, pre, full, temp;
   if (current_type != none) {
      if (current_type == e_infix || current_type == e_fullfix)
         post = current.infix_to_postfix();
      else if (current_type == e_prefix)
         post = current.prefix_to_postfix();
      else post = current;

      current.rewind();
      pre  = post.postfix_to_prefix();
      post.rewind();
      pre.prefix_to_full(full);
      pre.rewind();
      if (!post.valid_postfix() || !pre.valid_prefix()
                                || !full.valid_infix())
        cout << "Warning, Program Error Detected: \nAn invalid expression "
             << "has been computed." << endl;
   }
   switch (c) {
   case '?': help();
     break;

   case 'i':
     temp.read();
     if (!temp.valid_infix())
        cout << "Invalid input ignored." << endl;
     else {
        current = temp;
        current_type = e_infix;
        temp.clear();
     }
     break;

   case 'f':
     temp.read();
     if (!temp.valid_infix())
        cout << "Invalid input ignored." << endl;
     else {
        current = temp;
        current_type = e_fullfix;
        temp.clear();
     }
     break;
```

```
      case 'r':
        temp.read();
        if (!temp.valid_prefix())
           cout << "Invalid input ignored." << endl;
        else {
           current = temp;
           current_type = e_prefix;
           temp.clear();
        }
        break;

      case 'o':
        temp.read();
        if (!temp.valid_postfix())
           cout << "Invalid input ignored." << endl;
        else {
           current = temp;
           current_type = e_postfix;
           temp.clear();
        }
        break;

      case 'n':
        current = full;
        current_type = e_infix;
        current.dump();
        break;

      case 'u':
        current = full;
        current_type = e_fullfix;
        current.dump();
        break;

      case 'e':
        current = pre;
        current_type = e_prefix;
        current.dump();
        break;

      case 's':
        current = post;
        current_type = e_postfix;
        current.dump();
        break;

      case 'w':
        current.dump();
        break;

      case 'q':
        cout << "Expression conversion demonstration finished.\n";
        return 0;
      }
      return 1;
   }
```

```
int main()
/*
PRE:  None.
POST: An expression conversion demonstration has been performed.
USES: get_command, do_command, Expression methods
*/

{
   cout << "Running an interactive expression converter." << endl << endl;
   help();

   Expression test;
   current_type = none;
   while (do_command(get_command(), test));
}
```

The implementation for expressions is:

```
typedef int Value;

enum Token_type {operand, unaryop, binaryop, end_expression,
                          leftparen, rightparen, rightunaryop};

struct Token {
   Token_type kind();
   int priority();
   char value[20];
};

typedef Token Node_entry;
typedef Token Stack_entry;
#include "../../4/nodes/node.h"
#include "../../4/linkstac/stack.h"

class Expression {
public:
   Error_code evaluate_prefix(Value &result);
   Error_code evaluate_postfix(Value &result);
   Error_code recursive_evaluate(Value &result, Token &final);
   Error_code get_token(Token &result);

   Expression infix_to_postfix();
   Expression run_infix_to_postfix();

   void rec_infix_to_postfix(Expression &, int, Token, Token &);
   Expression prefix_to_postfix();
   void rec_prefix_to_postfix(Expression &);
   Expression postfix_to_prefix();

   void stack_to_prefix(Stack &s, Expression &answer);
   void prefix_to_full(Expression &answer);
   void postfix_to_full(Expression &answer);
   void put_token(Token &result);
   void put_token(char *);
   void read();
   void dump() { cout << value << endl; }
   Expression() { value[0] = '\0'; start = 0; count = 0; }
   void clear() { value[0] = '\0'; start = count = 0; }
   ~Expression() { clear(); }
   bool empty();
   void rewind(){start = 0;}
   bool valid_infix();
   bool valid_prefix();
   bool valid_postfix();

private:  //   data to store an expression
   char value[500];
   int start, count;
};


int Token::priority()
{
   if (kind() == operand) return -1;
   if (kind() == leftparen) return -1;
   if (kind() == rightparen) return -1;
   if (kind() == end_expression) return -1;

   switch (value[0]) {
   case ':': return 1;
   case '|': return 1;
   case '&': return 1;
   case '=': return 3;
   case '<': return 3;
   case '>': return 3;
   case '+': return 4;
   case '-': return 4;
   case '*': return 5;
   case '/': return 5;
   case '%': return 5;
   case '~': return 6;
   case '#': return 6;
   case '^': return 6;
   case '!': return 6;
   case '\'': return 6;
   case '"': return 6;
   }
   return -1;
}

Value get_value(Token &t)
{
   Value answer = 0;
   char *p = t.value;
   while (*p != 0) answer = 10 * answer + *(p++) - '0';
   return answer;
}

Value do_binary(Token &t, Value &x, Value &y)
```

```
{
   switch (t.value[0]) {
   case '+': return x + y;
   case '-': return x - y;
   case '*': return x * y;
   case '/': return x / y;
   case '%': return x % y;
   }
   return x;
}

Value do_unary(Token &t, Value &x)
{
   if (t.value[0] == '-') return -x;
   return x;
}

#include "../../4/nodes/node.cpp"
#include "../../4/linkstac/stack.cpp"

Expression Expression::infix_to_postfix()
/*
Pre:  The Expression stores a valid infix expression.
Post: A postfix expression translating the infix expression is returned.
*/

{
   Expression answer;
   Token current, prior;
   Stack delayed_operations;

   while (get_token(current) != fail) {
      switch (current.kind()) {
      case rightunaryop:
      case operand:
         answer.put_token(current);
         break;

      case leftparen:
         delayed_operations.push(current);
         break;

      case rightparen:
         delayed_operations.top(prior);
         while (prior.kind() != leftparen) {
            answer.put_token(prior);
            delayed_operations.pop();
            delayed_operations.top(prior);
         }

         delayed_operations.pop();
         break;

      case unaryop:
      case binaryop:                 //   Treat all operators together.
         bool end_right = false;     //   End of right operand reached?
```

```
            do {
               if (delayed_operations.empty()) end_right = true;
               else {
                  delayed_operations.top(prior);
                  if (prior.kind() == leftparen) end_right = true;
                  else if (prior.priority() < current.priority())
                      end_right = true;
                  else if (current.priority() == 6) end_right = true;
                  else answer.put_token(prior);
                  if (!end_right) delayed_operations.pop();
               }
            } while (!end_right);

            delayed_operations.push(current);
            break;
         }
      }
      while (!delayed_operations.empty()) {
         delayed_operations.top(prior);
         answer.put_token(prior);
         delayed_operations.pop();
      }
      answer.put_token(";");
      return answer;
   }

   Error_code Expression::recursive_evaluate(Value &result, Token &final)
   {
      while (true) {
         if (get_token(final) == fail) return fail;
         switch (final.kind()) {
         case binaryop:
         case end_expression:
            return success;

         case rightunaryop:
         case unaryop:
            result = do_unary(final, result);
            break;

         case operand:
            Value second_arg = get_value(final);
            if (recursive_evaluate(second_arg, final) == fail) return fail;
            if (final.kind() != binaryop) return fail;
            result = do_binary(final, result, second_arg);
            break;
         }
      }
   }

   Error_code Expression::evaluate_postfix(Value &result)
```

```
{
   Token t;
   Error_code outcome;
   if (get_token(t) == fail || t.kind() != operand) outcome = fail;
   else {
      result = get_value(t);
      outcome = recursive_evaluate(result, t);
      if (outcome == success && t.kind() != end_expression)
         outcome = fail;
   }
   return outcome;
}

Error_code Expression::evaluate_prefix(Value &result)
{
   Token t;
   Value x, y;
   if (get_token(t) == fail) return fail;
   switch (t.kind()) {
   case unaryop:
      if (evaluate_prefix(x) == fail) return fail;
      else result = do_unary(t, x);
      break;

   case binaryop:
      if (evaluate_prefix(x) == fail) return fail;
      if (evaluate_prefix(y) == fail) return fail;
      else result = do_binary(t, x, y);
      break;

   case operand:
      result = get_value(t);
      break;
   }
   return success;
}

Token_type Token::kind()
```

TOC
Index
Help

```
{
   if ('0' <= value[0] && value[0] <= '9') return operand;
   if (value[0] == '-' && value[1] != 0) return operand;
   if (value[0] == '~') return unaryop;
   if (value[0] == '#') return unaryop;
   if (value[0] == '!') return rightunaryop;
   if (value[0] == '"') return rightunaryop;
   if (value[0] == '\'') return rightunaryop;
   if (value[0] == '^') return binaryop;
   if (value[0] == '+') return binaryop;
   if (value[0] == '-') return binaryop;
   if (value[0] == '*') return binaryop;
   if (value[0] == '/') return binaryop;
   if (value[0] == '%') return binaryop;
   if (value[0] == '>') return binaryop;
   if (value[0] == '<') return binaryop;
   if (value[0] == '=') return binaryop;
   if (value[0] == '&') return binaryop;
   if (value[0] == '|') return binaryop;
   if (value[0] == ':') return binaryop;
   if (value[0] == '(') return leftparen;
   if (value[0] == ')') return rightparen;
   if (value[0] == ';') return end_expression;
   else return operand;
}

void Expression::put_token(char *result) {
   strcat(value, result);
   strcat(value, " ");
   count += 1 + strlen(result);
}

void Expression::put_token(Token &result) {
   strcat(value, result.value);
   strcat(value, " ");
   count += 1 + strlen(result.value);
}

bool Expression::empty() {
   return start >= count;
}

Error_code Expression::get_token(Token &result) {
   while (start < count) {
      char c = value[start];
      if ('0' <= c && c <= '9') {
         int i = 0;
         char *p = result.value;
         p[i] = c;
         while ('0' <= (c = value[++start]) && c <= '9')
            if (i < 18) p[++i] = c;
         if (i < 19) p[++i] = '\0';
         else p[19] = '\0';
         return success;
      }
```

TOC
Index
Help

```
      else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%'
            || c == '#' || c == '!' || c == '"' || c == '^' || c == '\''
            || c == '>' || c == '<' || c == '=' || c == '&' || c == '|'
            || c == ':' || c == '~' || c == ';' || c == '(' || c == ')'
            || ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')) {
         start++;
         char *p = result.value;
         p[0] = c; p[1] = '\0';
         return success;
      }
      else start++;
   }
   return fail;
}

void Expression::read()
{
   clear();
   Expression temp;
   while (1) {
      char c;
      c = temp.value[temp.count++] = cin.get();
      if (c == '$' || c == '\n' || c== EOF) break;
   }
   temp.value[temp.count] = '\0';
   temp.start = 0;
   Token current;
   value[0] = '\0';
   while (temp.get_token(current) != fail) put_token(current);
}
```

Certain auxiliary functions and methods:

```
Expression Expression::prefix_to_postfix()
/*
Pre:  The Expression stores a valid prefix expression.
Post: A postfix expression that translates the prefix expression is
      returned.
*/

{
   Expression answer;
   rec_prefix_to_postfix(answer);
   answer.put_token(";");
   return answer;
}

void Expression::rec_prefix_to_postfix(Expression &answer)
/*
Pre:  The Expression stores an initial segment giving a
      valid prefix expression (possibly empty).
Post: A postfix expression that translates the prefix expression is
      appended to the output parameter answer.
*/
```

```
{
   Token current;
   if (get_token(current) != fail) {
      switch (current.kind()) {
      case operand:
         answer.put_token(current);
         break;

      case rightunaryop:
      case unaryop:
         rec_prefix_to_postfix(answer);
         answer.put_token(current);
         break;

      case binaryop:
         rec_prefix_to_postfix(answer);
         rec_prefix_to_postfix(answer);
         answer.put_token(current);
      }
   }
}


Expression Expression::postfix_to_prefix()
/*
Pre:  The Expression stores a valid postfix expression.
Post: A prefix expression translating the postfix expression is returned.
*/

{
   Expression answer;
   Token current;
   Stack reverse_input;  //  Reverse of the input expression

   while (true) {
      get_token(current);
      if (current.kind() == end_expression) break;
      reverse_input.push(current);
   }
   stack_to_prefix(reverse_input, answer);
   return answer;
}

void Expression::stack_to_prefix(Stack &s, Expression &answer)
/*
Pre:  The Stack s stores the reversed sequence of tokens
      from a valid postfix expression.
Post: A prefix expression that translates the postfix expression is
      returned through the output parameter answer.
*/

{
   if (s.empty()) return;
   Token current;
   s.top(current);
   s.pop();
   switch (current.kind()) {
      case operand:
         answer.put_token(current);
         break;
      case rightunaryop:
      case unaryop:
         answer.put_token(current);
         stack_to_prefix(s, answer);
         break;

      case binaryop:
         answer.put_token(current);
         Expression second_arg;
         stack_to_prefix(s, second_arg);
         stack_to_prefix(s, answer);
         while (second_arg.get_token(current) != fail)
             answer.put_token(current);
   }
}

void Expression::prefix_to_full(Expression &answer)
/*
Pre:  The Expression begins with a valid prefix expression.
Post: A fully bracketed expression that translates
      the prefix expression is appended to the output parameter answer.
*/

{
   Token current;
   if (get_token(current) != fail) {
      switch (current.kind()) {
      case operand:
         answer.put_token(current);
         break;

      case rightunaryop:
         answer.put_token("(");
         prefix_to_full(answer);
         answer.put_token(current);
         answer.put_token(")");
         break;

      case unaryop:
         answer.put_token("(");
         answer.put_token(current);
         prefix_to_full(answer);
         answer.put_token(")");
         break;

      case binaryop:
         answer.put_token("(");
         prefix_to_full(answer);
         answer.put_token(current);
         prefix_to_full(answer);
         answer.put_token(")");
      }
   }
}

bool Expression::valid_infix()
/*
Post:   Returns true if and only if the Expression stores a
        valid infix expression.
*/
```

TOC
Index
Help

```
{
   Token current;
   bool leading = true;
   int paren_count = 0;

   while (get_token(current) != fail) {
      Token_type type = current.kind();

      if (type == rightparen || type == binaryop || type == rightunaryop) {
          if (leading) return false;
      }
      else if (!leading) return false;

      if (type == leftparen) paren_count++;
      else if (type == rightparen)
         if (--paren_count < 0) return false;

      if (type == leftparen || type == binaryop || type == unaryop)
         leading = true;
      else leading = false;
   }
   if (leading) return false;
   if(paren_count > 0) return false;
   rewind();
   return true;
}

bool Expression::valid_postfix()
/*
Post:   Returns true if and only if the Expression stores a
        valid postfix expression.
*/

{
    Expression temp, temp_copy;
    postfix_to_full(temp);
    rewind();
    if (!temp.valid_infix()) return false;
    temp_copy = temp.infix_to_postfix();
    if (strcmp(value, temp_copy.value) != 0) {
       cout << value << " " << temp_copy.value <<":Post" << endl;
       return false;
    }
    return true;
}

bool Expression::valid_prefix()
/*
Post: Returns true if and only if the Expression stores a
      valid prefix expression.
*/
```

```
{
    Expression temp, temp_copy;
    temp = prefix_to_postfix();
    rewind();
    if (!temp.valid_postfix()) return false;
    temp_copy = temp.postfix_to_prefix();
    if (strcmp(value, temp_copy.value) != 0) {
       cout << value << ":" << temp_copy.value <<":Pre" << endl;
       return false;
    }
    return true;
}
```

13.5 AN INTERACTIVE EXPRESSION EVALUATOR

Exercises 13.5

E1. *State precisely what changes in the program are needed to add the base 10 logarithm function log( ) as an additional unary operator.*

Answer A new unary operator comlog (for common logarithm) should be placed in the lexicon of predefined tokens. This requires a change to the String of predefined tokens in the Lexicon method set_standard_tokens. It also requires the addition of a statement to the attributes function so that comlog is recorded as a unary operator. Moreover, if the indexing of tokens has been disturbed, other tokens will have to be renumbered in set_standard_tokens and in the functions do_binary and do_unary. Code to call an auxiliary common logartihm function must be inserted into the function do_unary and the auxiliary function can be implemented as follows:

```
double common_log(double x)
/* Post: The base 10 logarithm of x is returned. */
{
  if (x > 0)
    return (ln(x)/ln(10.0));
  else
    return 0.0;
}
```

E2. *Naïve users of this program might (if graphing a function involving money) write a dollar sign '$' within the expression. What will the present program do if this happens? What changes are needed so that the program will ignore a '$'?*

Answer The Expression method read could be modified to ignore all occurrences of dollar signs by treating them as special symbols that would not be copied into the expression. One easy way to achieve this would be to make the function get_word (which splits a String of input into tokens) treat a dollar sign at the start of a numerical token as a space.

E3. *C++ programmers might accidentally type a semicolon ';' at the end of the expression. What changes are needed so that a semicolon will be ignored at the end of the expression but will be an error elsewhere?*

Answer The Expression method read could be modified to ignore terminal occurrences of semicolons by treating them as special symbols that would not be copied into the expression. One easy way to achieve this would be to make the function get_word (which splits a String of input into tokens) treat a semicolon followed by an end of String symbol as an end of String symbol.

E4. *Explain what changes are needed to allow the program to accept either square brackets* [. . .] *or curly brackets* {. . . } *as well as round brackets* (. . .). *The nesting must be done with the same kind of brackets; that is, an expression of the form* (. . . [. . .). . .] *is illegal, but forms like* [. . . (. . .). . . {. . .} . . .] *are permissible.*

Answer The program would have to be modified to recognize all of (, [, and { as being valid tokens of kind leftparen and),], and } as valid tokens of kind rightparen. This would allow the program to operate as before. The nesting can be checked in the Expression method valid_infix. A stack of characters is needed to keep track of the brackets. When a leftparen is encountered, its type, one of (, [, or {, should be pushed onto this stack. When a rightparen is encountered, the stack is popped and the left and right brackets can then be compared to make sure they are of the same shape.

Programming Project 13.5

P1. *Provide the missing functions and methods and implement the graphing program on your computer.*

Answer

```
#include "../../c/utility.h"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"

class List_int: public List<int> {
   public: List_int() {}
};

#include <string.h>
#include <stdlib.h>
#include "../../6/strings/string.h"
#include "../../6/strings/string.cpp"

#include "stack.h"
#include "stack.cpp"

#include "auxil.cpp"
#include "token_r.h"
#include "lexicon.h"
#include "token.h"
#include "token_r.cpp"
#include "lexicon.cpp"
#include "token.cpp"

typedef double Value;
#include "values.cpp"
#include "expr.h"
#include "expr.cpp"

#include "point.h"
#include "point.cpp"
typedef Point Key;
#include "../../8/linklist/sortable.h"
#include "plot.h"
#include "../../8/linklist/merge.cpp"
#include "plot.cpp"
```

```
char get_command()
{
   char c, d;
   cout << "Select command and press <Enter>:";
   while (true) {
      do {
         cin.get(c);
      } while (c == '\n');
      do {
         cin.get(d);
      } while (d != '\n');
      c = tolower(c);
      if (c == 'r' || c == 'w' || c == 'g' || c == 'l' ||
          c == 'p' || c == 'n' || c == 'h' || c == 'q') {
         return c;
      }
      cout << "Please enter a valid command or type H for help:" << endl;
   }
}

void help()
{
   cout << "Valid commands are:" << endl;
   cout << "\n\t[R]ead new function\t[W]rite function\t[G]raph function\n"
        << "\tset plot [L]imits\twrite [P]arameters\tset [N]ew parameters\n"
        << "\t[H]elp\t[Q]uit.\n";
}

void do_command(char c, Expression &infix, Expression &postfix, Plot &graph)
/*
Pre:  None
Post: Performs the user command represented by char c on the
      Expression infix, the Expression postfix, and the Plot graph.
Uses: Classes Token, Expression and Plot.
*/

{
   switch (c) {
   case 'r':      //  Read an infix expression from the user.
      infix.clear();
      infix.read();
      if (infix.valid_infix() == success)
         postfix = infix.infix_to_postfix();
      else cout << "Warning: Bad expression ignored. " << endl;
      break;

   case 'w':      //  Write the current expression.
      infix.write();
      postfix.write();
      break;

   case 'g':      //  Graph the current postfix expression.
      if (postfix.size() <= 0)
         cout << "Enter a valid expression before graphing!" << endl;
      else {
         graph.clear();
         graph.find_points(postfix);
         graph.draw();
      }
      break;
   case 'l':    //  Set the graph limits.
      if (graph.set_limits() != success)
         cout << "Warning: Invalid limits" << endl;
      break;

   case 'p':    //  Print the graph parameters.
      Token::print_parameters();
      break;

   case 'n':    //  Set new graph parameters.
      Token::set_parameters();
      break;

   case 'h':    //  Give help to user.
      help();
      break;
   }
}

void introduction()
{
   List<Point> t; List<Point> s = t; //  To help certain compilers.
   cout << "Running the menu driven graphing program."
        << endl << endl;
   help();
   cout << endl;
   cout << "Functions should be specified using a variable called: x"
        << endl;
}

int main()
/*
Pre:  None
Post: Acts as a menu-driven graphing program.
Uses: Classes Expression and Plot,
      and functions introduction, get_command, and do_command.
*/

{
   introduction();
   Expression infix;      //  Infix expression from user
   Expression postfix;    //  Postfix translation
   Plot graph;

   char ch;
   while ((ch = get_command()) != 'q')
      do_command(ch, infix, postfix, graph);
}
```

Class definitions:

```
class Expression {
public:

//  Add method prototypes.
```

```
   Expression();
   Expression(const Expression &original);
   Error_code get_token(Token &next);
   void put_token(const Token &next);
   Expression infix_to_postfix();
   Error_code evaluate_postfix(Value &result);
   void read();
   void clear();
   void write();
   Error_code valid_infix();
   int size();
   void rewind();

private:
   List<Token> terms;
   int current_term;

//  Add auxiliary function prototypes.

   Error_code recursive_evaluate(const Token &first_token,
                                 Value &result, Token &final_token);

};


const int hash_size = 997;

struct Lexicon {
   Lexicon();
   int hash(const String &x) const;
   void set_standard_tokens();  //  Set up the predefined tokens.

   int count;                   //  Number of records in the Lexicon
   int index_code[hash_size];   //  Declare the hash table.
   Token_record token_data[hash_size];
};


class Plot {
public:
   Plot();
   Error_code set_limits();
   void find_points(Expression &postfix);
   void draw();
   void clear();
   int get_print_row(double y_value);
   int get_print_col(double x_value);

private:
   Sortable_list<Point> points;   //  records of points to be plotted
   double x_low, x_high;          //  x limits
   double y_low, y_high;          //  y limits
   double x_increment;            //  increment for plotting
   int max_row, max_col;          //  screen size
};


struct Point {
   int row;
   int col;

   Point();
   Point(int a, int b);
   bool operator == (const Point &p);
   bool operator != (const Point &p);
   bool operator >= (const Point &p);
   bool operator <= (const Point &p);
   bool operator >  (const Point &p);
   bool operator <  (const Point &p);
};

template <class Stack_entry>
struct Stack_node {

   Stack_entry entry;
   Stack_node<Stack_entry> *next;

   Stack_node();
   Stack_node ( Stack_entry item, Stack_node<Stack_entry> *add_on = NULL );
};

template <class Stack_entry>
class Stack {
public:
   Stack();
   bool empty() const;
   Error_code push(const Stack_entry &item);
   Error_code pop();
   Error_code top(Stack_entry &item) const;
   ~Stack();
   Stack (const Stack<Stack_entry> &original);
   void operator =(const Stack<Stack_entry> &original);

protected:
   Stack_node<Stack_entry> *top_node;
};

class Token {
public:

//  Add methods here.

   Token() {}
   Token (const String &x);
   Token_type kind() const;
   int priority() const;
   double value() const;
   String name() const;
   int code_number() const;
   static void set_parameters();
   static void print_parameters();
   static void set_x(double x_val);

private:
   int code;
   static Lexicon symbol_table;
   static List<int> parameters;
};

List<int> Token::parameters; // Allocate storage for static Token members.
Lexicon Token::symbol_table;

enum Token_type {operand, unaryop, binaryop, end_expression,
      leftparen, rightparen, rightunaryop};
```

```
struct Token_record {
   String name;
   double value;
   int priority;
   Token_type kind;
};
```

Class implementations:

```
Expression::Expression()
{
   current_term = 0;
}

Expression::Expression(const Expression &original)
{
   terms = original.terms;
   current_term = original.current_term;
}

int Expression::size()
{
   return terms.size();
}

void Expression::write()
{
   Token t;
   for (int i = 0; i < terms.size(); i++) {
      terms.retrieve(i, t);
      cout << (t.name()).c_str() << " ";
   }
   cout << endl;
}

void Expression::rewind()
{
   current_term = 0;
}

Error_code Expression::get_token(Token &next)
/*
Post: The Token next records the current_term of the Expression,
      current_term is incremented, and an error code of success is returned,
      or if there is no such term a code of fail is returned.
Uses: Class List.
*/

{
   if (terms.retrieve(current_term, next) != success) return fail;
   current_term++;
   return success;
}

void Expression::put_token(const Token &next)
{
   terms.insert(terms.size(), next);
}
```

```
void Expression::read()
/*
Post: A line of text, entered by the user, is split up into
      tokens and stored in the Expression.
Uses: Classes String, Token, and List.
*/

{
   String input, word;
   int term_count = 0;
   int x;
   cout << "Enter an infix expression giving a function of x:"
        << endl;
   input = read_in(cin, x);
   add_spaces(input);                //  Tokens are now words of input.

   bool leading = true;
   for (int i = 0; get_word(input, i, word) != fail; i++) {
                                     //  Process next token
      if (leading)
         if (word == "+") continue;                  //  unary +
         else if (word == "-") word = "~";          //  unary -

      Token current = word;                     //  Cast word to Token.
      terms.insert(term_count++, current);
      Token_type type = current.kind();
      if (type == leftparen || type == unaryop || type == binaryop)
         leading = true;
      else
         leading = false;
   }
}

void Expression::clear()
{
   terms.clear();
   current_term = 0;
}

Expression Expression::infix_to_postfix()
{
   Expression answer;
   Token t;
   Stack<Token> s;

   while (get_token(t) != fail) {
      switch (t.kind()) {
      case rightunaryop:
      case operand:
         answer.put_token(t);
         break;

      case leftparen:
         s.push(t);
         break;
```

```
            case rightparen:
               s.top(t);
               while (t.kind() != leftparen) {
                  answer.put_token(t);
                  s.pop();
                  s.top(t);
               }
               s.pop();
               break;

            case unaryop:
            case binaryop:
               Token stored;
               bool end_right = false;
               do {
                  if (s.empty()) end_right = true;
                  else {
                     s.top(stored);
                     if (stored.kind() == leftparen) end_right = true;
                     else if (stored.priority() < t.priority()) end_right = true;
                     else if (t.priority() == 6 && stored.priority() == 6)
                        end_right = true;
                     else answer.put_token(stored);
                     if (!end_right) s.pop();
                  }
               } while (!end_right);

               s.push(t);
               break;
            }
         }
         while (!s.empty()) {
            s.top(t);
            answer.put_token(t);
            s.pop();
         }
         answer.put_token(( Token ) (( String ) ";"));
         return answer;
      }

      Error_code Expression::recursive_evaluate(const Token &first_token,
                                          Value &result, Token &final_token)
      /*
      Pre:  Token first_token is an operand.
      Post: If the first_token can be combined with initial tokens of
            the Expression to yield a legal postfix expression followed
            by either an end_expression symbol or a binary operator,
            a code of success is returned, the legal postfix subexpresssion
            is evaluated, recorded in result, and the terminating Token is
            recorded as final_token.  Otherwise a code of fail is returned.
            The initial tokens of Expression are removed.
      Uses: Methods of classes Token, and Expression including
            recursive_evaluate and functions do_unary, do_binary,
            and get_value.
      */

      {
```

```
   Value first_segment = get_value(first_token), next_segment;
   Error_code outcome;
   Token current_token;
   Token_type current_type;

   do {
      outcome = get_token(current_token);
      if (outcome != fail) {
         switch (current_type = current_token.kind()) {
         case binaryop:      // Binary operations terminate subexpressions.
         case end_expression:  // Treat subexpression terminators together.
            result = first_segment;
            final_token = current_token;
            break;

         case rightunaryop:
         case unaryop:
            first_segment = do_unary(current_token, first_segment);
            break;

         case operand:
            outcome = recursive_evaluate(current_token,
                                             next_segment, final_token);
            if (outcome == success && final_token.kind() != binaryop)
               outcome = fail;
            else
               first_segment = do_binary(final_token, first_segment,
                                                     next_segment);
            break;
         }
      }
   } while (outcome == success && current_type != end_expression &&
                                  current_type != binaryop);

   return outcome;
}

Error_code Expression::evaluate_postfix(Value &result)
/*
Post: The tokens in Expression up to the first end_expression symbol are
      removed.  If these tokens do not represent a legal postfix expression,
      a code of fail is returned.   Otherwise a code of success is returned,
      and the removed sequence of tokens is evaluated to give Value result.
*/

{
   Token first_token, final_token;
   Error_code outcome;

   if (get_token(first_token) == fail || first_token.kind() != operand)
      outcome = fail;

   else {
      outcome = recursive_evaluate(first_token, result, final_token);
      if (outcome == success && final_token.kind() != end_expression)
         outcome = fail;
   }
   return outcome;
}
```

```
Error_code Expression::valid_infix()
/*
Post: A code of success or fail is returned according
      to whether the Expression is a valid or invalid infix sequence.
Uses: Class Token.
*/

{
   Token current;
   bool leading = true;
   int paren_count = 0;

   while (get_token(current) != fail) {
      Token_type type = current.kind();
      if (type == rightparen || type == binaryop || type == rightunaryop) {
         if (leading) return fail;
      }
      else if (!leading) return fail;

      if (type == leftparen) paren_count++;
      else if (type == rightparen) {
         paren_count--;
         if (paren_count < 0) return fail;
      }

      if (type == binaryop || type == unaryop || type == leftparen)
         leading = true;
      else leading = false;
   }

   if (leading)  return fail;   //  An expected final operand is missing.
   if (paren_count > 0)  return fail;  //  Right parentheses are missing.
   rewind();
   return success;
}


void Lexicon::set_standard_tokens()
{
   int i = 0;
   String symbols = ( String )
"; ( )  aãabs sqr sqrt exp ln lg sin cos arctan round trunc ! % + - * / ^ x pi e"
   String word;
   while (get_word(symbols, i++, word) != fail) {
      Token t = word;
   }
   token_data[23].value = 3.14159;
   token_data[24].value = 2.71828;
}

int Lexicon::hash(const String &identifier) const
/*
Post: Returns the location in table Lexicon::index_code that
      corresponds to the String identifier.
      If the hash table is full and does not contain a record
      for identifier, the exit function is called to terminate the
      program.
Uses: The class String, the function exit.
*/
```

```
{
   int location;
   const char *convert = identifier.c_str();
   char first = convert[0], second;  // First two characters of identifier
   if (strlen(convert) >= 2) second = convert[1];
   else second = first;

   location = first % hash_size;
   int probes = 0;
   while (index_code[location] >= 0 &&
          identifier != token_data[index_code[location]].name) {
      if (++probes >= hash_size) {
         cout << "Fatal Error: Hash Table overflow. Increase table size\n";
         exit(1);
      }
      location += second;
      location %= hash_size;
   }
   return location;
}

Lexicon::Lexicon()
/*
Post: The Lexicon is initialized with
      the standard tokens.
Uses: set_standard_tokens
*/

{
   count = 0;
   for (int i = 0; i < hash_size; i++)
      index_code[i] = -1;     //  code for an empty hash slot
   set_standard_tokens();
}
```

The following class for plotting can be adjusted for improved, but system dependent, plotting capabilities.

```
int Plot::get_print_col(double x_value)
{
   double interp_x = ((double) max_col) * (x_value - x_low)
                                        / (x_high - x_low) + 0.5;
   int answer = (int) interp_x;
   if (answer < 0 || answer > max_col) answer = -1;
   return answer;
}

int Plot::get_print_row(double y_value)
/*
Post: Returns the row of the screen at which a point with
      y-coordinate y_value should be plotted, or
      returns -1 if the point would fall off the edge of
      the screen.
*/
```

```
{
   double interp_y =
      ((double) max_row) * (y_high - y_value) / (y_high - y_low) + 0.5;
   int answer = (int) interp_y;
   if (answer < 0 || answer > max_row) answer = -1;
   return answer;
}

Error_code Plot::set_limits()
{
   double x, y;
   double new_increment = x_increment,
      new_x_low = x_low,
         new_x_high = x_high,
            new_y_low = y_low,
               new_y_high = y_high;
   cout << "Give a new value for x increment. Previous value = "
        << x_increment;

   cout << "\n Enter a value or a new line to keep the old value: "
        << flush;

   if (read_num(x) == success) new_increment = x;
   cout << "Give a new value for lower x limit. Previous value = " << x_low;
   cout << "\n Enter a value or a new line to keep the old value: "
        << flush;
   if (read_num(x) == success) new_x_low = x;

   cout << "Give a new value for upper x limit. Previous value = "
        << x_high;

   cout << "\n Enter a value or a new line to keep the old value: "
        << flush;
   if (read_num(x) == success) new_x_high = x;

   cout << "Give a new value for lower y limit. Previous value = " << y_low;
   cout << "\n Enter a value or a new line to keep the old value: "
        << flush;
   if (read_num(y) == success) new_y_low = y;

   cout << "Give a new value for upper y limit. Previous value = "
        << y_high;
   cout << "\n Enter a value or a new line to keep the old value: "
        << flush;
   if (read_num(y) == success) new_y_high = y;

   if (new_increment <= 0 || new_x_high <= new_x_low
                          || new_y_high <= new_y_low)
      return fail;
   x_increment = new_increment;
   x_low = new_x_low;
   x_high = new_x_high;
   y_low = new_y_low;
   y_high = new_y_high;
   return success;
}

void Plot::clear()
{
   points.clear();
}
```

```
void Plot::find_points(Expression &postfix)
/*
Post: The Expression postfix is evaluated for values of
      x that range from x_low to x_high in steps of
      x_increment.  For each evaluation we add a Point
      to the Sortable_list points.
Uses: Classes Token, Expression, Point, and List.
*/

{
   double x_val = x_low;
   double y_val;
   while (x_val <= x_high) {
      Token::set_x(x_val);
      postfix.evaluate_postfix(y_val);
      postfix.rewind();
      Point p(get_print_row(y_val), get_print_col(x_val));
      points.insert(0, p);
      x_val += x_increment;
   }
}

void Plot::draw()
/*
Post: All screen locations represented in points have
      been marked with the character '#' to produce a
      picture of the stored graph.
Uses: Classes Point and Sortable_list and its method merge_sort.
*/

{
   points.merge_sort();
   int at_row = 0, at_col = 0;         //  cursor coordinates on screen

   for (int i = 0; i < points.size(); i++) {
      Point q;
      points.retrieve(i, q);
      if (q.row < 0 || q.col < 0) continue;  //  off the scale of the graph
      if (q.row < at_row || (q.row == at_row && q.col < at_col)) continue;

      if (q.row > at_row) {              //  Move cursor down the screen.
         at_col = 0;
         while (q.row > at_row) {
            cout << endl;
            at_row++;
         }
      }

      if (q.col > at_col)                 //  Advance cursor horizontally.
         while (q.col > at_col) {
            cout << " ";
            at_col++;
         }
      cout << "#";
      at_col++;
   }
   while (at_row++ <= max_row) cout << endl;
}
```

```
Plot::Plot()
{
   x_low = -10;
   x_high = 10;
   y_low = -1000;
   y_high = 1000;
   x_increment = 0.01;
   max_row = 19;
   max_col = 79;
}

Point::Point()
{
   row = 0;
   col = 0;
}

Point::Point(int a, int b)
{
   row = a;
   col = b;
}

bool Point::operator == (const Point &p)
{
   return col == p.col && row == p.row;
}

bool Point::operator != (const Point &p)
{
   return col != p.col || row != p.row;
}

bool Point::operator > (const Point &p)
{
   return (row > p.row) || (col > p.col && row == p.row);
}

bool Point::operator >= (const Point &p)
{
   return (row > p.row) || (col >= p.col && row == p.row);
}

bool Point::operator < (const Point &p)
{
   return (row < p.row) || (col < p.col && row == p.row);
}

bool Point::operator <= (const Point &p)
{
   return (row < p.row) || (col <= p.col && row == p.row);
}

template <class Stack_entry>
Error_code Stack<Stack_entry>::push(const Stack_entry &item)
{
   Stack_node<Stack_entry>
             *new_top = new Stack_node<Stack_entry>(item, top_node);
   if (new_top == NULL) return fail;
   top_node = new_top;
   return success;
}
```

```
template <class Stack_entry>
Stack<Stack_entry>::~Stack()
{
   while (!empty())
      pop();
}

template <class Stack_entry>
Stack_node<Stack_entry>::Stack_node()
{
   next = NULL;
}

template <class Stack_entry>
Stack<Stack_entry>::Stack()
{
   top_node = NULL;
}

template <class Stack_entry>
Stack_node<Stack_entry>::
   Stack_node (Stack_entry item, Stack_node<Stack_entry> *add_on = NULL )
   {
      entry = item;
      next = add_on;
   }

template <class Stack_entry>
Stack<Stack_entry>::Stack (const Stack<Stack_entry> &copy)
{
   Stack_node<Stack_entry> *new_copy, *copy_node = copy.top_node;
   if (copy_node == NULL) top_node = NULL;
   else {
      new_copy = top_node = new Stack_node<Stack_entry>(copy_node->entry);
      while (copy_node->next != NULL) {
         copy_node = copy_node->next;
         new_copy->next = new Stack_node<Stack_entry>(copy_node->entry);
         new_copy = new_copy->next;
      }
   }
}

template <class Stack_entry>
void Stack<Stack_entry>:: operator =(const Stack<Stack_entry> &copy)
{
   Stack_node<Stack_entry> *new_top, *new_copy, *copy_node = copy.top_node;
   if (copy_node == NULL) new_top = NULL;
   else {
      new_copy = new_top = new Stack_node<Stack_entry>(copy_node->entry);
      while (copy_node->next != NULL) {
         copy_node = copy_node->next;
         new_copy->next = new Stack_node<Stack_entry>(copy_node->entry);
         new_copy = new_copy->next;
      }
   }
   Stack_entry temp;
   while (!empty()) pop();
   top_node = new_top;
}
```

```
template <class Stack_entry>
bool Stack<Stack_entry>::empty() const
/*
Post: Return true if the Stack is empty,
      otherwise return false.
*/

{
   return top_node == NULL;
}

template <class Stack_entry>
Error_code Stack<Stack_entry>::top(Stack_entry &item) const
/*
Post: The top of the Stack is reported in item. If the Stack
      is empty return underflow otherwise return success.
*/

{
   if (top_node == NULL) return underflow;
   item = top_node->entry;
   return success;
}

template <class Stack_entry>
Error_code Stack<Stack_entry>::pop()
/*
Post: The top of the Stack is removed.  If the Stack
is empty return underflow otherwise return success.
*/

{
   Stack_node<Stack_entry> *old_top = top_node;
   if (top_node == NULL) return underflow;
   top_node = old_top->next;
   delete old_top;
   return success;
}


Token_type Token::kind() const
{
   return symbol_table.token_data[code].kind;
}

int Token::priority() const
{
   return symbol_table.token_data[code].priority;
}

double Token::value() const
{
   return symbol_table.token_data[code].value;
}

String Token::name() const
{
   return symbol_table.token_data[code].name;
}
```

```
int Token::code_number() const
{
   return code;
}

void Token::set_parameters()
/*
Post: All parameter values are printed for the user, and any changes
      specified by the user are made to these values.
Uses: Classes List, Token_record, and String, and function
      read_num.
*/

{
   int n = parameters.size();
   int index_code;
   double x;

   for (int i = 0; i < n; i++) {
      parameters.retrieve(i, index_code);
      Token_record &r = symbol_table.token_data[index_code];

      cout << "Give a new value for parameter " << (r.name).c_str()
           << " with value " << r.value << endl;
      cout << "Enter a value or a new line to keep the old value: "
           << flush;
      if (read_num(x) == success) r.value = x;
   }
}

void Token::print_parameters()
{
   int n = parameters.size();
   int location;

   for (int i = 0; i < n; i++) {
      parameters.retrieve(i, location);
      Token_record &r = symbol_table.token_data[location];
      cout << (r.name).c_str() << " has value " << r.value << endl;
   }
}

void Token::set_x(double x_val)
{
   symbol_table.token_data[22].value = x_val;
}

Token::Token(const String &identifier)
/*
Post: A Token corresponding to String identifier
      is constructed.  It shares its code with any other Token object
      with this identifier.
Uses: The class Lexicon.
*/
```

```
{
   int location = symbol_table.hash(identifier);
   if (symbol_table.index_code[location] == -1) {  // Create a new record.
      code = symbol_table.count++;
      symbol_table.index_code[location] = code;
      symbol_table.token_data[code] = attributes(identifier);
      if (is_parameter(symbol_table.token_data[code]))
         parameters.insert(0, code);
   }
   else code = symbol_table.index_code[location];  // Code of an old record
}


bool is_parameter(const Token_record &r)
{
   const char *temp = (r.name).c_str();
   if (r.kind == operand && ('0' > temp[0] || temp[0] > '9'))
      return true;
   else
      return false;
}

Token_record attributes(const String &identifier)
{
   Token_record r;
   r.name = identifier;
   const char *temp = identifier.c_str();
   r.value = 0.0;

   if ('0' <= temp[0] && temp[0] <= '9') {
      r.kind = operand;
      r.value = get_num(temp);
   }

   else if (identifier == ";") r.kind = end_expression;

   else if (identifier == "(") r.kind = leftparen;

   else if (identifier == ")") r.kind = rightparen;

   else if (identifier == "~" || identifier == "abs" ||
            identifier == "sqr" || identifier == "lg" ||
            identifier == "sqrt" || identifier == "exp" ||
            identifier== "ln" || identifier == "round" ||
            identifier == "sin" || identifier == "cos" ||
            identifier == "arctan" || identifier == "trunc") {
      r.kind = unaryop;
      r.priority = 6;
   }

   else if (identifier == "!" || identifier == "%") {
      r.kind = rightunaryop;
      r.priority = 6;
   }

   else if (identifier == "+" || identifier == "-") {
      r.kind = binaryop;
      r.priority = 4;
   }
   else if (identifier == "*" || identifier == "/" ||
                identifier == "^" ) {
      r.kind = binaryop;
      r.priority = 5;
   }

   else if ( identifier == "^" ) {
      r.kind = binaryop;
      r.priority = 6;
   }

   else
      r.kind = operand;

   return r;
}
```

Random Numbers B

B.3 PROGRAM DEVELOPMENT

Programming Projects B.3

P1. *Write a driver program that will test the three random number functions developed in this appendix. For each function, calculate the average value it returns over a sequence of calls (the number of calls specified by the user). For* random_real, *the average should be about 0.5; for* random_integer(low, high) *the average should be about* (low + high)/2, *where* low *and* high *are given by the user; for* poisson(expected_value), *the average should be approximately the expected value specified by the user.*

Answer Driver program for Projects P1–P3:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void help()
{
   cout << "User options are:\n"
        << "[H]elp  [Q]uit \n"

        << "[R] Test random real    --- Project P1a\n"
        << "[I] Test random integer --- Project P1b\n"
        << "[P] Test random poisson --- Project P1c\n"
        << "[2] Project P2: linear frequency test\n"
        << "[3] Project P3: rectangular frequency test\n"
        << endl;
}

void intro()
{
   cout << "Testing program for random number methods." << endl;
   help ();
}
```

```
int main()
{
   intro();
   Random dice;
   int n_calls;
   char command;

   while (command != 'q' && command != 'Q') {
      cout << "Enter a command of H, Q, R, I, P, 2, 3:  "
           << flush;
      cin  >> command;
      if ('a' <= command && command <= 'z') command = command + 'Z' - 'z';
      if (command == 'R' || command == 'I' || command == 'P' ||
          command == '2' || command == '3') {
         cout << "How many numbers: " << flush;
         cin  >> n_calls;
         if (n_calls <= 0) n_calls = 1;
      }
      int i;
      switch (command) {
         case 'H':
           help();
         break;

         case 'R': {
           double sum = 0.0;
           for (i = 0; i < n_calls; i++) sum += dice.random_real();
           cout << "Average value is: " << sum / ((double) n_calls)
                << endl;
         }
         break;

         case 'P': {
           double sum = 0.0;
           double expect;
           cout << "Enter the expected Poisson value: " << flush;
           cin  >> expect;
           for (i = 0; i < n_calls; i++) sum += dice.poisson(expect);
           cout << "Average value is: " << sum / ((double) n_calls)
                << endl;
         }
         break;

         case 'I': {
           int low, high;
           double sum = 0.0;
           cout << "Enter the range for integers as a pair,  low high : "
                << flush;
           cin  >> low >> high;
           if (high <= low) {
              cout << "illegal range!" << endl;
              break;

           }
           for (i = 0; i < n_calls; i++)
              sum += dice.random_integer(low, high);
           cout << "Average value is: " << sum / ((double) n_calls)
                << endl;
         }
         break;
```

```
            case '2': {
              int high;
              cout << "Enter the number of random integers to generate: "
                   << flush;
              cin  >> high;
              if (high <= 0) high = 1;
              int *array = new int[high];
              for (i = 0; i < high; i++) array[i] = 0;
              for (i = 0; i < n_calls; i++)
                 array[dice.random_integer(0, high - 1)]++;
              cout << "Distribution of numbers is: " << endl;
              for (i = 0; i < high; i++) cout << array[i] << " ";
              cout << endl;
              delete []array;
            }
            break;

            case '3': {
              int x_dimension, y_dimension;
              cout << "Give the number of rows of random integers to generate:"
                   << flush;
              cin  >> x_dimension;

              if (x_dimension<= 0) x_dimension= 1;
              int **array = new int *[x_dimension];
              cout << "Enter the number of columns of random integers: "
                   << flush;
              cin  >> y_dimension;

              if (y_dimension<= 0) y_dimension= 1;
              for (i = 0; i < x_dimension; i++)
                 array[i] = new int[y_dimension];

              for (i = 0; i < x_dimension; i++)
               for (int j = 0; j < y_dimension; j++) array[i][j] = 0;

              for (i = 0; i < n_calls; i++) {
                 int x, y;
                 x = dice.random_integer(0, x_dimension - 1);
                 y = dice.random_integer(0, y_dimension - 1);
                 array[x][y]++;
              }

              cout << "Distribution of numbers is: " << endl;
              for (i = 0; i < x_dimension; i++) {
               for (int j = 0; j < y_dimension; j++)
                  cout << array[i][j] << " ";
               delete array[i];
               cout << endl;
              }
              delete []array;
            break; }
         }  // end of outer switch statement
      }     // end of outer while statement
   }
```

All programs in this appendix use the class definition from the text:

```
class Random {
public:
   Random(bool pseudo = true);
//    Declare random-number generation methods here.
   int random_integer(int low, int high);
   double random_real();
   int poisson(double mean);

private:
   int reseed(); //  Re-randomize the seed.
   int seed,
       multiplier, add_on; //  constants for use in arithmetic operations
};
```

Implementation:

```
#include <limits.h>
const int max_int = INT_MAX;
#include <math.h>
#include <time.h>

Random::Random(bool pseudo)
/*
Post: The values of seed, add_on, and multiplier
      are initialized.  The seed is initialized randomly only if
      pseudo == false.
*/
{
   if (pseudo) seed = 1;
   else seed = time(NULL) % max_int;
   multiplier = 2743;
   add_on = 5923;
}

double Random::random_real()
/*
Post: A random real number between 0 and 1 is returned.
*/
{
   double max = max_int + 1.0;
   double temp = reseed();
   if (temp < 0) temp = temp + max;
   return temp / max;
}

int Random::random_integer(int low, int high)
/*
Post: A random integer between low and high (inclusive)
      is returned.
*/
{
   if (low > high) return random_integer(high, low);
   else return ((int) ((high - low + 1) * random_real())) + low;
}

int Random::poisson(double mean)
/*
Post: A random integer, reflecting a Poisson distribution
      with parameter mean, is returned.
*/
```

```
{
   double limit = exp(-mean);
   double product = random_real();
   int count = 0;
   while (product > limit) {
      count++;
      product *= random_real();
   }
   return count;
}

int Random::reseed()
/*
Post: The seed is replaced by a psuedorandom successor.
*/

{
   seed = seed * multiplier + add_on;
   return seed;
}
```

P2. *One test for uniform distribution of random integers is to see if all possibilities occur about equally often. Set up an array of integer counters, obtain from the user the number of positions to use and the number of trials to make, and then repeatedly generate a random integer in the specified range and increment the appropriate cell of the array. At the end, the values stored in all cells should be about the same.*

Answer See the solution to Project P1.

P3. *Generalize the test in the previous project to use a rectangular array and two random integers to determine which cell to increment.*

Answer See the solution to Project P1.

scissors-paper-rock

P4. *In a certain children's game, each of two players simultaneously puts out a hand held in a fashion to denote one of scissors, paper, or rock. The rules are that scissors beats paper (since scissors cut paper), paper beats rock (since paper covers rock), and rock beats scissors (since rock breaks scissors). Write a program to simulate playing this game with a person who types in* S, P, *or* R *at each turn.*

Answer The following program is a (dishonest) driver for the game.

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void intro()
{
   cout << "Program plays Rock, Scissors, Paper."
        << endl;
   cout << "Legal user commands are R S P or Q to quit"
        << endl;
}

int main()
{
   intro();
   Random dice;
   int i_won = 0, you_won = 0;
   char command;
   while (command != ’Q’) {
     char my_command = ’Q’ + dice.random_integer(0, 2);
     if (my_command == ’Q’) my_command = ’P’;
     cout << "Enter P, Q, R or S: " << flush;
     cin  >> command;
     if (’p’ <= command && command <=’s’)
         command = command + ’Q’ - ’q’;
     switch (command) {
     case ’P’:
        cout << my_command << endl;
        if (my_command == ’S’) {
           cout << "Scissors cuts paper. I win." << endl;
           i_won++;
        }
        if (my_command == ’R’) {
           cout << "Paper covers rock. You win." << endl;
           you_won++;
        }
     break;
     case ’R’:
        cout << my_command << endl;
        if (my_command == ’P’) {
           cout << "Paper covers rock.  I win." << endl;
           i_won++;
        }
        if (my_command == ’S’) {
           cout << "Rock breaks scissors. You win." << endl;
           you_won++;
        }
     break;
     case ’S’:
        cout << my_command << endl;
        if (my_command == ’R’) {
           cout << "Rock breaks scissors.  I win." << endl;
           i_won++;
        }
        if (my_command == ’P’) {
           cout << "Scissors cuts paper. You win." << endl;
           you_won++;
        }
     break;
     }
   }
   cout << "Game over.  Final score: I won "
        << (i_won * 10) / 9                // The program cheats!
        << " You won "
        << you_won
        << endl;
}
```

P5. *In the game of Hamurabi you are the emperor of a small kingdom. You begin with 100 people, 100 bushels of grain, and 100 acres of land. You must make decisions to allocate these resources for each year. You may give the people as much grain to eat as you wish, you may plant as much as you wish, or you may buy or sell land for grain. The price of land (in bushels of grain per acre) is determined randomly (between 6 and 13). Rats will consume a random percentage of your grain, and there are other events that may occur at random during the year. These are:*

Hamurabi

TOC
Index
Help

- Plague
- Bumper Crop
- Population Explosion
- Flooding
- Crop Blight
- Neighboring Country Bankrupt! Land Selling for Two Bushels per Acre
- Seller's Market

At the end of each year, if too many (again this is random) people have starved for lack of grain, they will revolt and dethrone you. Write a program that will determine the random events and keep track of your kingdom as you make the decisions for each year.

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void intro()
{
   cout << "Program plays Hamurabi." << endl;
}

int main()
{
   intro();
   Random dice;

   int year = 0;
   bool living = true;
   int people = 100;
   int grain = 100;
   int land = 100;

   int planted = 100;
   int land_cost = 10;
   int harvest;
   int cost, feed, buy, plant;
   int rats;
   int event;
   int popularity = 10;

   while (living) {
     cout << "In the year " << year++ << " the harvest is "
          << (harvest = dice.random_integer(4, 10)) << " bushels per acre"
          << endl;
     harvest *= planted;
     rats = dice.random_integer(-2 * harvest, harvest - 1);

     grain = grain + harvest;
     if (rats > 0) {
        cout << "Rats ate " << rats << " bushels." << endl;
        grain -= rats;
     }
```

```
cout << "You rule over " << people << " people, "
     << land << " acres of land, and "
     << grain << " bushels of grain." << endl;
 land_cost = dice.random_integer(6, 13);
 event = dice.random_integer(0, 20);
 switch (event) {
 case 0:
    cout << "Plague! " << endl;
    if (rats > 0) {
       double deaths = ((double) rats) / ((double) ( 10 * grain));
       people -= (int) deaths;
    }
  break;

 case 1:
    cout << "Bumper Crop!" << endl;
    grain += harvest / 4;
    popularity += 1;
  break;

 case 2:
    cout << "Population Explosion!" << endl;
    people += people / 10;
    popularity += 1;
  break;

 case 3:
    cout << "Flooding!" << endl;
    people -= people / 50;
    grain -= grain / 20;
    popularity -= 1;
  break;

 case 4:
    cout << "Crop Blight!" << endl;
    grain -= grain / 20;
  break;

 case 5:
    cout << "Neighboring Country Bankrupt! Land selling cheap."
         << endl;
    land_cost = 2;
  break;

 case 6:
    cout << "Seller's market! Land expensive." << endl;
    land_cost += 2;
  break;
 }

 cout << "Land costs " << land_cost << " bushels per acre." << endl;
 bool ok = false;
 while (!ok) {
    cout << "How many bushels will you feed your people: " << flush;
    cin >> feed;
    cout << "How many acres will you buy: " << flush;
    cin >> buy;
    cout << "How many acres will you plant: " << flush;
    cin >> plant;
```

```
         cost = feed + buy * land_cost + plant;
         if (cost > grain) {
            cout << "That would cost " << cost << " bushels. You have "
                 << grain << " Resubmit your budget.";
         }
         else ok = true;
         if (plant + plant / 10 < land) {
            cout << "That would leave more than 10% of your land idle. "
                 << "Are you sure? ";
            ok = user_says_yes();
         }
         if (people - feed > people / 10) {
            cout << "Your people will revolt. "
                 << "Are you sure? ";
            ok = user_says_yes();
         }
      }

      grain -= cost;
      if (feed < people) {
         cout << (people - feed) << " people starved to death." << endl;
         popularity -= (100 * (people - feed)) / people;
         people = feed;
      }
      else popularity += (50 * (feed - people)) / people;
      land += buy;
      planted = plant;
      if (planted > land) planted = land;
      if (planted > people) planted = people;
      if (popularity < 0) {
         cout << "Your people revolt and kill you."
              << endl;
         living = false;
      }
      if (popularity > 0) {
         people += dice.random_integer(0, popularity);
      }
   }
}
```

TOC
Index
Help

P6. *After leaving a pub, a drunk tries to walk home, as shown in Figure B.1. The streets between the pub and the home form a rectangular grid. Each time the drunk reaches a corner, he decides at random what direction to walk next. He never, however, wanders outside the grid.*

random walk

(a) *Write a program to simulate this random walk. The number of rows and columns in the grid should be variable. Your program should calculate, over many random walks on the same grid, how long it takes the drunk to get home on average. Investigate how this number depends on the shape and size of the grid.*

(b) *To improve his chances, the drunk moves closer to the pub—to a room on the upper left corner of the grid. Modify the simulation to see how much faster he can now get home.*

(c) *Modify the original simulation so that, if the drunk happens to arrive back at the pub, then he goes in and the walk ends. Find out (depending on the size and shape of the grid) what percentage of the time the drunk makes it home successfully.*

(d) *Modify the original simulation so as to give the drunk some memory to help him, as follows. Each time he arrives at a corner, if he has been there before on the current walk, he remembers what streets he has already taken and tries a new one. If he has already tried all the streets from the corner, he decides at random which to take now. How much more quickly does he get home?*

Answer

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../../b/random.h"
#include "../../b/random.cpp"

void intro()
{
   cout << "Program simulates a drunkard's walk." << endl;
}

bool is_ok(int direction, int r, int c, int rows, int cols, bool ***memory)
/*
Post: Returns a success code for a direction choice in part (d)
*/

{
  if (r == rows - 1 && direction == 0) return false;
  if (r == 0 && direction == 1) return false;
  if (c == cols - 1 && direction == 2) return false;
  if (c == 0 && direction == 3) return false;
  if (!memory[r][c][direction]) {
      memory[r][c][direction] = true;
      return true;
  }

  if (memory[r][c][0] && memory[r][c][1] && memory[r][c][2]
                      && memory[r][c][3]) return true;

  if (r == rows - 1   && memory[r][c][1] && memory[r][c][2]
                      && memory[r][c][3]) return true;

  if (memory[r][c][0] && r == 0          && memory[r][c][2]
                      && memory[r][c][3]) return true;

  if (memory[r][c][0] && memory[r][c][1] && c == cols - 1
                      && memory[r][c][3]) return true;

  if (memory[r][c][0] && memory[r][c][1] && memory[r][c][2]
                      && c == 0)          return true;

  if (r == rows - 1   && memory[r][c][1] && c == cols - 1
                      && memory[r][c][3]) return true;

  if (r == rows - 1   && memory[r][c][1] && memory[r][c][2]
                      && c == 0)          return true;

  if (memory[r][c][0] && r == 0          && c == cols - 1
                      && memory[r][c][3]) return true;

  if (memory[r][c][0] && r == 0          && memory[r][c][2]
                      && c == 0)          return true;
  return false;
}

int main()
{
   intro();
   Random dice;
   int rows, cols;
   int time = 0;
   int r = 0, c = 0, i;
   int trials;
```

```
cout << "How many grid rows do you want? " << flush;
cin  >> rows;
cout << "How many grid columns do you want? " << flush;
cin  >> cols;
cout << "How many trials per part do you want? " << flush;
cin  >> trials;

cout << "Running Part(a)." << endl;
int total_time = 0;
for (i = 0; i < trials; i++) {
   time = r = c = 0;
   while (r < rows - 1 || c < cols - 1) {
      int direction = dice.random_integer(0 , 3);
      switch (direction) {
          case 0: if (r < rows - 1) { r++; time++;} break;
          case 1: if (r > 0)        { r--; time++;} break;
          case 2: if (c < cols - 1) { c++; time++;} break;
          case 3: if (c > 0)        { c--; time++;} break;
      }
   }
   total_time += time;
}
cout << "For part (a) the average time is: "
     << (((double) total_time) / ((double) trials)) << endl;

cout << "Running Part(b)." << endl;
total_time = 0;
for (i = 0; i < trials; i++) {
   time = r = c = 0;
   while (r > 0 || c < cols - 1) {
      int direction = dice.random_integer(0 , 3);
      switch (direction) {
          case 0: if (r < rows - 1) { r++; time++;} break;
          case 1: if (r > 0)        { r--; time++;} break;
          case 2: if (c < cols - 1) { c++; time++;} break;
          case 3: if (c > 0)        { c--; time++;} break;
      }
   }
   total_time += time;
}
cout << "For part (b) the average time is: "
     << (((double) total_time) / ((double) trials)) << endl;

cout << "Running Part(c)." << endl;
int successes = 0;

for (i = 0; i < trials; i++) {
   time = r = c = 0;
   while (r < rows - 1 || c < cols - 1) {
      int direction = dice.random_integer(0 , 3);
```

```
         switch (direction) {
             case 0: if (r < rows - 1) { r++; time++;} break;
             case 1: if (r > 0)        { r--; time++;} break;
             case 2: if (c < cols - 1) { c++; time++;} break;
             case 3: if (c > 0)        { c--; time++;} break;
         }
         if (r == 0 && c == 0 && time > 0) break;
      }
      if (r == rows - 1 && c == cols - 1) successes++;
   }

   cout << "For part (c) the average success rate of return home is: "
        << (((double) successes) / ((double) trials)) << endl;

   cout << "Running Part(d)." << endl;
   bool ***memory;
   memory = new bool**[rows];
   for (i = 0; i < rows; i++) {
      memory[i] = new bool*[cols];
      for (int j = 0; j < cols; j++) {
         memory[i][j] = new bool[4];
      }
   }

   total_time = 0;
   for (int ii = 0; ii < trials; ii++) {
      for (i = 0; i < rows; i++)
         for (int j = 0; j < cols; j++)
            for (int k = 0; k < 4; k++) memory[i][j][k] = false;
      time = r = c = 0;

      while (r < rows - 1 || c < cols - 1) {
         int direction = dice.random_integer(0 , 3);
         while (!is_ok(direction, r, c, rows, cols, memory))
             direction = dice.random_integer(0 , 3);
         switch (direction) {
             case 0: r++; time++; break;
             case 1: r--; time++; break;
             case 2: c++; time++; break;
             case 3: c--; time++; break;
         }
      }
      total_time += time;
   }

   for (i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         delete []memory[i][j];
      }
      delete [] memory[i];
   }
   delete []memory;
   cout << "For part (d) the average time is: "
        << (((double) total_time) / ((double) trials)) << endl;
}
```

P7. *Write a program that creates files of integers in forms suitable for testing and comparing the various searching, sorting, and information retrieval programs. A suitable approach to testing and comparing these programs is to use the program to create a small suite of files of varying size and various kinds of ordering, and then use the CPU timing unit to compare various programs operating on the same data file.*

generation of searching and sorting data

The program should generate files of any size up to 15,000 positive integers, arranged in any of several ways. These include:

(a) *Random order, allowing duplicate keys*
(b) *Random order with no duplicates*
(c) *Increasing order, with duplicates allowed*
(d) *Increasing order with no duplicates*
(e) *Decreasing order, with duplicates allowed*
(f) *Decreasing order with no duplicates*
(g) *Nearly, but not quite ordered, with duplicates allowed*
(h) *Nearly, but not quite ordered, with no duplicates*

The nearly ordered files can be altered by entering user-specified data. The program should use pseudo-random numbers to generate the files that are not ordered.

Answer The file-generation menu-driven program is implemented as:

```
#include "../../c/utility.h"
#include "../../c/utility.cpp"
#include "../random.h"
#include "../random.cpp"

#include "../../6/linklist/list.h"
#include "../../6/linklist/list.cpp"
typedef int Key;
typedef Key Record;
#include "../../7/3/ordlist.h"
#include "ordlist.cpp"    // changed to disallow duplicates

void help()
{
   cout << "User options are:\n"
        << "[?]Help  [Q]uit [#]set file size [\"]print to screen\n"
        << "----- and output commands as follows----------" << endl
        << "[a] random order, duplicates allowed " << endl
        << "[b] random order, no duplicates allowed " << endl
        << "[c] increasing order, duplicates allowed " << endl
        << "[d] increasing order, no duplicates allowed " << endl
        << "[e] decreasing order, duplicates allowed " << endl
        << "[f] decreasing order, no duplicates allowed " << endl
        << "[g] nearly ordered, duplicates allowed " << endl
        << "[h] nearly ordered, duplicates allowed " << endl
        << endl;
}

int main()
{
   Ordered_list l;
   List<int> rand;
   help();

   Random dice;
   char command = ' ';
```

```
while (command != 'q' && command != 'Q') {
   cout << "Enter a command of ?, Q, #, \", A, B, "
        << "C, D, E, F, G, H: " << flush;
   cin  >> command;
   if ('a' <= command && command <= 'h') command += 'A' - 'a';

   int i;
   switch (command) {
      case '"': {
        int d;
        for (i = 0; i < l.size(); i++) {
          l.retrieve(i, d);
          cout << d << " ";
        }
        cout << endl; }
      break;

      case '?':
        help();
      break;

      case '#': {
         int n;
         rand.clear();
         l.clear();
         cout << "What size of file are you interested. Give # entries: "
              << flush;
         cin  >> n;
         if (n > 15000) n = 15000;
         for (i = 0; i < n; i++) {
            int d = dice.random_integer(0, 1000000);
            if (l.insert(d) == duplicate_error) i--;
            else rand.insert(0, d);
         }
      }
      break;

      case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
      case 'G': case 'H': {
         cout << "Name your output file: " << flush;
         char name[1000];
         cin  >> name;
         ofstream ofile(name);
         int n = rand.size(), d;

         if (command == 'A')
          for (i = 0; i < n; i++) {
             int j = dice.random_integer(0, n - 1);
             rand.retrieve(j, d);
             ofile << d << " ";
             if (i % 10 == 9) ofile << endl;
         }

         if (command == 'B')
          for (i = 0; i < n; i++) {
             int d;  rand.retrieve(i, d);
             ofile << d << " ";
             if (i % 10 == 9) ofile << endl;
         }
```

TOC
Index
Help

```
if (command == 'C' || command == 'D')
 for (i = n - 1; i >= 0; i--) {
    int d;  l.retrieve(i, d);
    ofile << d << " ";
    if ((n - i) % 10 == 0) ofile << endl;
    int x = dice.random_integer(0, 9);
    if (command == 'C' && x == 9) {
      ofile << d << " ";
      i--;
      if ((n - i) % 10 == 0) ofile << endl;
    }
}

if (command == 'E' || command == 'F')
 for (i = 0; i < n; i++) {
    int d;  l.retrieve(i, d);
    ofile << d << " ";
    if (i % 10 == 9) ofile << endl;
    int x = dice.random_integer(0, 9);
    if (command == 'E' && x == 9) {
      ofile << d << " ";
      i++;
      if (i % 10 == 9) ofile << endl;
    }
}

if (command == 'G' || command == 'H') {
 rand.clear();
 for (i = 0; i < n; i++) {
    int d;  l.retrieve(i, d);
    rand.insert(0, d);
    int x = dice.random_integer(0, 9);
    if (command == 'G' && x == 9) {
      rand.insert(0, d);
      i++;
    }
 }

 while (true) {
    cout << "Do you still want to change entries? " << endl;
    if (!user_says_yes()) break;
    cout << "Which entry?" << flush;
    int d;
    cin >> i;
    cout << "New Value?" << flush;
    cin >> d;
    rand.replace(i, d);
 }
```

```
                for (i = 0; i < n; i++) {
                   int d;  rand.retrieve(i, d);
                   ofile << d << " ";
                   if (i % 10 == 9) ofile << endl;
                }
               }
               ofile << endl;
            }
            break;
         }
      }
}
```

The program uses a revised order list implementation:

```
Ordered_list::Ordered_list()
{
}

Error_code Ordered_list::insert(const Record &data)
/*
Post: If the Ordered_list is not full,
      the function succeeds: The Record data is
      inserted into the list, following the
      last entry of the list with a strictly lesser key
      (or in the first list position if no list
      element has a lesser key).
      Else: the function fails with the diagnostic Error_code overflow.
*/

{
   int s = size();
   int position;
   for (position = 0; position < s; position++) {
      Record list_data;
      retrieve(position, list_data);
      if (data > list_data) break;
      if (data == list_data) return duplicate_error;
   }
   return List<Record>::insert(position, data);
}

Error_code Ordered_list::insert(int position, const Record &data)
/*
Post: If the Ordered_list is not full,
      0 <= position <= n,
      where n is the number of entries in the list,
      and the Record ata can be inserted at
      position in the list, without disturbing
      the list order, then
      the function succeeds:
      Any entry formerly in
      position and all later entries have their
      position numbers increased by 1 and
      data is inserted at position of the List.
      Else: the function fails with a diagnostic Error_code.
*/
```

```
{
   Record list_data;
   if (position > 0) {
      retrieve(position - 1, list_data);
      if (data < list_data)
         return fail;
   }

   if (position < size()) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
      if (data == list_data)
         return duplicate_error;
   }
   return List<Record>::insert(position, data);
}

Error_code Ordered_list::replace(int position, const Record &data)
{
   Record list_data;
   if (position > 0) {
      retrieve(position - 1, list_data);
      if (data < list_data)
         return fail;
   }

   if (position < size()) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
   }
   return List<Record>::replace(position, data);
}
```