Answers to Selected Exercises

Ada Plus Data Structures: An Object Oriented Approach, 2nd Edition

Nell Dale & John McCormick
John.McCormick@acm.org

Chapter 1

4. The knowledge software engineers have collected over time such as algorithms, and data structures. It also includes programming methodologies such as object-oriented design and software concepts such as information hiding, encapsulation, and abstraction. Finally, it can include tools such as CRC cards and UML.

8. Use meaningful identifier names. Document each logical block of code, each loop, and each subprogram. Include module reuse as a goal of your design process.

10. Understand the problem to be solved.

15c. An aspirin is seen by a patient as a means for controlling headache pain. A chemist sees it as a compound to be formulated.

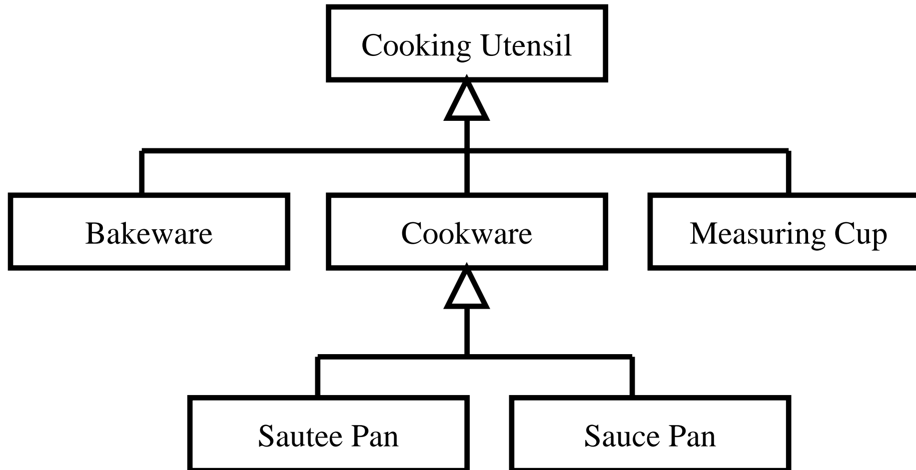
16. The details of all the steps necessary to actually display text on the screen as are the details of converting the binary representation of a floating point number to characters are hidden from the user of the I/O operations.

23. Some attributes not relevant to cooking are color, shipping weight, self-cleaning capability, and number of light bulbs.

25. Ada packages are written in two parts in order to separate *what* the package does from *how* it does it.

29. Inheritance provides a powerful reuse tool that allows programmers to create a new class (subclass) that is a specialization of an existing class (superclass). The subclass inherits the attributes and operations of its superclass. We usually make a subclass more specialized by adding more attributes and operations and changing (overriding) some of the superclass operations.

35d and 36d



44. A programmer might assume that a user of the program enters large numbers without the commas that are commonly used to make number easier to read. A comma may result in an unhandled Data_Error exception.

47a. Verifying a program without executing it.

b. Verifying a program by executing it with a set of test data.

c. A statement that is either true or false but not both. Also called a proposition.

50. The result of calling a subprogram when the precondition is False is not known. The program may crash with a run-time error or continue in an invalid state that may not be evident for some time.

55.

```
Sum   := 0;
Index := 1;
loop
   exit when Index > Max_List or else List(Index) = 0;
   Sum := Sum + List(Index);
end loop;
```

59. The loop displays all of the components of the array List. There are only 37 "good" values. We need to use another variable whose value is equal to the number of good values in the array List and use that variable in our loop rather than the maximum number of values List can hold.

63. Data coverage testing uses test cases based on the possible input values. Code coverage uses test cased based on the branches or paths in the code.

66a. The functional domain is the set of integers from 0 to 100.

b. Yes, it would only require 101 test cases.

c. Using data coverage, create a test case for the lowest, middle, and highest scores for each of the letter grade ranges.

69.

```
function Terminate_Loop (Factor_1 : in Integer;
                         Factor_2 : in Integer) return Boolean is
begin
   return Factor_1 = Factor_2;
end Terminate_Loop;
```

71a. The definition of Integer ensures that Value is a whole number.

b. The definition of Dozen ensures that Chicken is greater than zero.

Chapter 2

3. A data structure is collection of elements. A data type may be a single element or multiple elements. The logical organization of the elements in a data structure have a direct influence on the accessing operations used to store or retrieve individual elements. Even when a data type has multiple elements, their organization does not usually influence the storage and retrieval of individual elements.

7. At the application level, a list of student academic records might be used to determine scholarship recipients. At the abstract level, there may be operations to retrieve the record of the student with the highest grade point average, to print a list of students in order by grade point average, to modify a student's grade point average, etc. At the implementation level, the list might be implemented as an array of records.

10. Modular types are not signed so we can store larger values. (For example, on a 32 bit processor, the largest Integer is usually 2^{31}-1 while the largest modular type is usually 2^{32}-1.) Modular types use modular (clock) arithmetic. Finally, we can use the logical operators with modular types to manipulate the bits comprising them.

14. Equality testing, assignment, relational testing (for array with discrete components), indexing, slicing, and attributes ('First, 'Last, 'Length, 'Range).

17a.

```
type Decade_Weather_Array is array (Month_Type, Year_Range) of Weather_Rec;
Decade_Weather : Decade_Weather_Array;
```

c.

```
Decade_Weather(March, 1989).Avg_Lo_Temp := 26;
```

21. The type from which the operations and values (domain) for a subtype are taken

22. The package.

25. Classes are stored in packages. A subclass is stored in a child package of its superclass's package.

28. An operation for a type that is declared in the same package specification as the type and has a parameter or return value of the type.

30. So that the construction operations are not primitive operations. This ensures that a subclass does not inherit any constructor operations from its superclass. Each subclass must define its own constructors.

32a. The details of an ADO are encapsulated within the package body which is not accessible to the application programmer.

c. By defining a private type and encapsulating the details within the private section of the package specification. The private part of a package specification is not accessible to the application programmer.

d. We can create many objects of the type.

e. We can create just one object of the type.

35. Constructors are used to create new values of a class. We cannot create an instance of an abstract class and therefore we cannot make use of a constructor.

40a. It is easy to set and retrieve the real and imaginary parts of a complex number. We can construct complex values any way we like.

b. A constructor operation and operations to retrieve the real and imaginary parts.

Chapter 3

1 a.
```
type Month_Type is (Jan, Feb, Mar, Apr, May, Jun,
                    Jul, Aug, Sep, Oct, Nov, Dec);

package Month_Set is new Discrete_Set (Element_Type => Month_Type);
```

d.
```
Winter_Months := Month_Set.Empty_Set;
Winter_Months := Winter_Months + Nov + Dec + Jan;
```

4 a.
```
B := My_Set.Universal_Set - A;
```

c.
```
function Complement (Set : in Set_Type) return Set_Type is
begin
   return not Set;
end Complement;
```

7. It is likely that creating an array of Booleans indexed by Integer required more memory than was available.

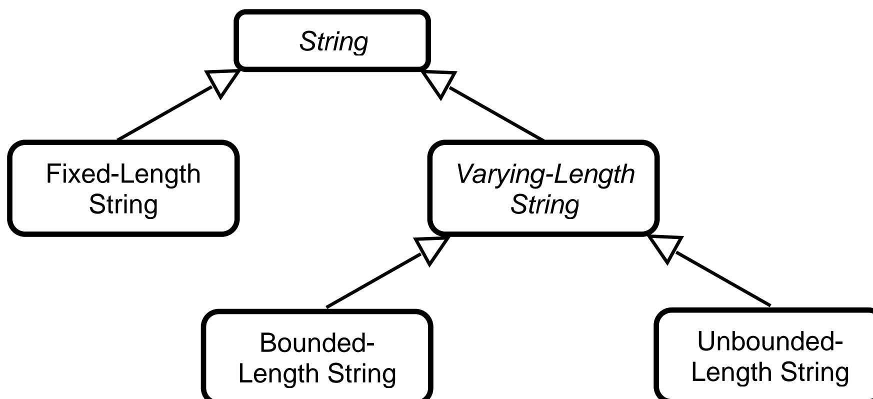
10.
```
package Precise_Ops is new
       Ada.Numerics.Generic_Elementary_Functions (Float_Type => High_Precision);
```

12.
```
procedure Sort is new Ada.Containers.Generic_Constrained_Array_Sort
                       (Index_Type   => Lowercase,
                        Element_Type => Natural,
                        Array_Type   => Natural_Array,
                        "<"          => Standard."<");
```

15 a
```
generic
   type Value_Type is (<>);
   with function ">=" (Left  : in Value_Type;
                       Right : in Value_Type) return Boolean;
function Largest (First : in Value_Type;
                  Second : in Value_Type;
                  Third  : in Value_Type) return Value_Type;


function Largest (First : in Value_Type;
                  Second : in Value_Type;
                  Third  : in Value_Type) return Value_Type is
begin
   if First >= Second then
      if First >= Third then
         return First;
      else
         return Third;
      end if;
   else
      if Second >= Third then
         return Second;
      else
         return Third;
      end if;
   end if;
end Largest;
```

Chapter 4

2.



5. `subtype My_String_Type is String (1..45);`

8 a. 6

c. 10

e. 10

10. Call the functions `Exists` and `Kind` in package `Ada.Directories`.

```
with Ada.Directories; use Ada.Directories;
with Ada.Text_IO;     use Ada.Text_IO;
procedure Demo is

   subtype File_Name_String is String (1..80);

   File_Name : File_Name_String;
   Length    : Natural;          -- Number of characters in File_Name

begin
   Put_Line ("Enter a file name");
   Get_Line (Item => File_Name, Last => Length);

   if Exists (File_Name(1..Length) ) then
      Put_Line ("The file " &
                 File_Name(1..Length)  &
                " exists and is type " &
                 File_Kind'Image (Kind (File_Name(1..Length))) );
   else
      Put_Line ("The file " &
                 File_Name(1..Length)  &
                " does not exist");
   end if;
end Demo;
```

12 a. `package File_Names is new Ada.Strings.Bounded.Generic_Bounded_Length(100);`

13 a.

```
procedure Get_Line (Item : out File_Names.Bounded_String) is
   Fixed_Item : String (1..File_Names.Max_Length);
   Length     : Natural;
begin
   Ada.Text_IO.Get_Line (Item => Fixed_Item,
                         Last => Length);
   Item := File_Names.To_Bounded_String (Fixed_Item (1..Length));
end Get_Line;
```

A better approach for reading a bounded length string is to instantiate a bounded-length string I/O package from `Ada.Text_IO.Bounded_IO`.

17.

```
function Index (Source  :  in Bounded_String;
                Pattern :  in Bounded_String) return Natural is
```

```
      First : Positive;       -- Pattern is compared to
      Last  : Positive;       --     Source (First..Last)

   begin  -- Search
      First := 1;             -- Begin search at the first position of Source
      Last  := Pattern.Length;
      Search_Loop:  -- Each iteration, the pattern is compared to one slice
      loop          -- of the source
         -- Exit when there are fewer characters remaining in Source to check
         -- than there are in the Pattern   or   when match is found
         exit when  Last > Source.Length  or else
                    Source.Data(First..Last) = Pattern.Data(1..Pattern.Length);
         First := First + 1;
         Last  := Last  + 1;
      end loop Search_Loop;
      if Last <= Source.Length then   -- Did we find the pattern?
         return First;
      else
         return 0;
      end if;
   end Index;
```

22. Convert the bounded-length string to a fixed-length string (using the operation from package `Strings`) and then to an unbounded-length string (using the operation from package `Ada.Strings.Unbounded`).

```
return To_Unbounded_String (To_String(Item));
```

26 a. 6 bytes

c. `Student_List.all(1).ID := 1000;`

e.
```
for Index in Student_List.all'Range loop
    Student_List.all(Index).Total_Hours :=  Student_List.all(Index).Total_Hours
                                         +  Student_List.all(Index).Current_Hours;
    Student_List.all(Index).Current_Hours := 0;
end loop
```

g. `Free (Student_List);`

After calling `Free`, `Student_List` contains the value `null`.

30 a. `Finalize` is called for `Oak`.
The value of `Maple` is copied into `Oak`.
`Adjust` is called for `Oak`.

31 c. The declaration of `Elm` specifies an initial value so `Initialize` is not called when `Elm` is elaborated.

Chapter 5

1. They are ordered so that the last item added to the stack is the first one removed (LIFO).

4 a.
```
package Name_Stack is new Stack(Element_Type => Name_Rec);
```

b.
```
Names : Name_Stack.Stack_Type (Max_Size => 250);
```

c.
```
package Name_Stack is new Unbounded_Stack(Element_Type => Name_Rec);
```

6 e.
```
Target.Clear;
Temp.Clear;

loop
   exit when Source.Empty;
   Source.Pop (Value);
   Temp.Push (Value);
end loop;

loop
   exit when Temp.Empty;
   Temp.Pop (Value);
   Source.Push (Value);
   Target.Push (Value);
end loop;
```

10.
```
procedure Replace_Element (Stack  : in out Stack_Type;
                           Old_El : in     Element_Type;
                           New_El : in     Element_Type) is

   Temp  : Stack_Type (Max_Size => Stack.Max_Size);
   Value : Element_Type;

begin
   loop
      exit when Stack.Empty;
      Stack.Pop (Value);
      if Value = Old_El then
         Value := New_El;
      end if;
      Temp.Push (Value);
   end loop;

   loop
      exit when Temp.Empty;
      Temp.Pop (Value);
      Stack.Push (Value);
   end loop;
end Replace_Element;
```

13 This program displays the numbers 1 through 5. The first loop of the program displays a random subset of these numbers in ascending order. The second loop of the program displays the remaining numbers, if any, in descending order.

b. This output sequence is possible. This output sequence occurs when all five of the calls to the random Boolean generator return True so no values are displayed by the first loop. All five values are pushed onto the stack and later displayed in reverse order.

c. This output sequence is not possible. The ascending order sequence 1, 3, 5 is followed by the ascending order sequence 2 and 4.

16 b.
```
type Stack_Array is array (Positive range <>) of Integer;

type Double_Stack_Type (Max_Size : Positive) is
   record
      Even_Top : Natural  := 0;
      Odd_Top  : Positive := Max_Size + 1;
      Elements : Stack_Array (1..Max_Size);
   end record;
```

19.
```
procedure Cut_Back (Stack : in out Stack_Type;
                    By    : in     Positive) is
begin
   if By > Stack.Top then
      raise UNDERFLOW;
   end if;
   Stack.Top := Stack.Top - By;
end Cut_Back;
```

21 a. Invalid. Assigning an integer to an access variable

b. Valid. Assigning an integer to the `Info` field in the node designated by `Stack`.

c. Valid. Leaving off the optional dereferencing operator `.all` is poor style.

22 a. Pointer

b. Record.

c. Array

24 a. The memory for all local variables is stored on the system stack (review the discussion on the *Organization of Memory* on page 255 of the textbook). The memory used by these local variables is recycled when control returns from the subprogram to the caller.

b. The memory for all local variables is recycled when control returns from the subprogram to the caller.

c. The maintenance programmer is worried about the memory used for the stack nodes. Memory for these nodes is allocated on the heap (review the discussion on the *Organization of Memory* on page 255 of the textbook). Since the unbounded stack type is a controlled type, procedure `Finalize` is called automatically when `First` and `Second` go out of scope (just before the memory for these local variables is removed from the system stack). Procedure `Finalize` is defined in the private part of the unbounded stack package specification as another name for the `Clear` operation. `Clear` recycles the memory used by all of the nodes in a stack.

25 a. Dynamically.

b. The space needed to store one access value.

28.

```
procedure Cut_Back (Stack : in out Stack_Type;
                    By    : in     Positive) is
   Current : Node_Ptr; -- A local pointer used to count and to recycle nodes
   Count   : Natural;  -- The number of nodes
begin
   -- To check for UNDERFLOW we need to count the nodes in the stack
   Count   := 0;
   Current := Stack.Top;
   -- Go through the linked list to see if there are enough nodes
   -- Each iteration, count one more node
   loop
      -- Exit when we reach the necessary count
      --        or we reach the end of the linked list
      exit when Count = By or Current = null;
      Count   := Count + 1;
      Current := Current.all.Next;
   end loop;
   if Count < By then
      raise UNDERFLOW;
   end if;

   -- Remove the first By nodes from the stack
   for Count in 1..By loop
      Current := Stack.Top;
      Stack.Top := Stack.Top.all.Next;
      Free (Current);
   end loop;
end Cut_Back;
```

31. Because both white box testing and McCabe's complexity are based on the number of possible paths through an algorithm.

34. The calculation of the square root is the "elephant" within the loop. The exit statement and the other two assignment statements are "goldfish".

36 a. $O(N^2)$

d. $O(N^2)$

f. $O(N^2)$

37 b. O(1)

d. $O(\log_2 N)$

40. All three of the statements are False.

Chapter 6

2a.
```
5
7
0
```

4 a.
```
package Name_Queue is new Stack(Element_Type => Name_Rec);
```

b.
```
Names : Name_Queue.Queue_Type (Max_Size => 250);
```

5. The call to `Dequeue` has a syntax error that would be caught by the compiler. The actual parameter for the formal out mode parameter `Item` must be a variable, not an expression.

7 a.
```
loop
   exit when My_Queue.Empty;
   My_Queue.Dequeue (Last);
end loop;
```

10 This program displays the numbers 1 through 5. The first loop of the program displays a random subset of these numbers in ascending order. The second loop of the program displays the remaining numbers, if any, in ascending order.

b. This output sequence is not possible because there is no way to partition this list into two lists each in ascending order

c. This output sequence is possible. The ascending order sequence 1, 3, 5 is followed by the ascending order sequence 2 and 4.

12 a. We have to first determine which queue in the array of ten queues the new job should be placed. The index of the correct queue may be calculated by dividing the ID by 100.

c.
```
procedure Clean_Up_Jobs (Jobs : in out Job_Array) is
   Job : Job_Token;
begin
   -- Clean up all the jobs
   -- Each iteration, clean up one queue of jobs
   for Index in Jobs'Range loop

      -- Clean up one queue of jobs
      -- Each iteration, clean up one job
      loop
         exit when Jobs(Index).Empty;
         Jobs(Index).Dequeue (Job);
         Notify (User => Job);
      end loop;
   end loop;
end Clean_Up_Jobs;
```

14.
```
51 / 14      =  3
51 rem 14    =  9
51 mod 14    =  9

-51 / 14     = -3
-51 rem 14   = -9
-51 mod 14   =  5

51 / -14     = -3
51 rem -14   =  9
51 mod -14   = -5

-51 / -14    =  3
-51 rem -14 = -9
-51 mod -14 = -9
```

16.
```
function "=" (Left  : in Queue_Type;
              Right : in Queue_Type) return Boolean is
   Left_Index  : Positive;
   Right_Index : Positive;
begin
   if Left.Count /= Right.Count then
      return False;
   else -- we need to compare the elements
      Left_Index  := Left.Front;
      Right_Index := Right.Front;
      -- Compare all the elements in the two queues
      -- Each iteration, compare one pair of elements
      for Count in 1..Left.Count loop
         if Left.Items(Left_Index) /= Right.Items(Right_Index) then
            return False;
         end if;
         Left_Index  := Left_Index  rem Left.Max_Size + 1;
         Right_Index := Right_Index rem Right.Max_Size + 1;
      end loop;
      return True;
   end if;
end "=";end Replace_Element;
```

18.
```
function Size (Queue : in Queue_Type) return Natural is
begin
   return Queue.Count;
end Size;
```

21 b. `5 16 27`

23.
```
function "=" (Left  : in Queue_Type;
              Right : in Queue_Type) return Boolean is
   Left_Ptr  : Node_Ptr;
   Right_Ptr : Node_Ptr;
begin
   Left_Ptr  := Left.Front;
   Right_Ptr := Right.Front;
   -- Compare all the elements in queues
   -- Each iteration, compare one pair of elements
   loop
      exit when Left_Ptr = null or Right_Ptr = null;
      if Left_Ptr.all.Info /= Right_Ptr.all.Info then
         return False;
      end if;
      Left_Ptr  := Left_Ptr.all.Next;
      Right_Ptr := Right_Ptr.all.Next;
   end loop;
   -- If we reach this point, all of the elements compared were equal.
   -- The queues are equal only if we have compared all the elements in both
   return Left_Ptr = null and Right_Ptr = null;
end "=";
```

26.
```
function Size (Queue : in Queue_Type) return Natural is
   Current : Node_Ptr;
   Count   : Natural;
begin
   Current := Queue.Front;
   Count   := 0;
   -- Count the nodes in the linked list
   -- Each iteration, count one node
   loop
      exit when Current = null;
      Count   := Count + 1;
      Current := Current.all.Next;
   end loop;
   return Count;
end Size;
```

Chapter 7

2 a. Name and address.

h. Name and year of award.

j. In the United States, each doctor has a Unique Physician Identification Number (UPIN). This number is being replaced by the National Provider Identifier (NPI).

5 a. They are all ordered sequences of elements. Lists are ordered by key while stacks and queues are ordered by the time of insertion.

b. The more general names Insert and Delete are used rather than the specific names Push, Pop, Enqueue, and Dequeue. These more general names reflect the ability to insert and delete any item in the collection rather than items in specific locations in the sequence.

c. The list is the more general class. Stacks and queues are more specialized lists.

8. By changing the mode from *in out* to *in*, we can eliminate the precondition that the Process procedure not change the key of an element. Also, most process procedures do not change elements in the list. Forcing the use of *in out* mode for such process procedures is poor style. The advantage of using *in out* mode is that process procedures can modify all the elements in the list.

10. Because many friends live in the same zip code, we cannot create a list with zip code as the key. Our approach is to create a unique key by combining the zip code and name.

```
type Zip_Rec is
   record
      Zip  : Zip_String;
      Name : Name_Rec;
   end record;


function Zip_Of (Friend : in Friend_Rec) return Zip_Rec is
   Result : Zip_Rec;
begin
   Result.Zip := Friend.Address.Zip;
   Result.Name := Friend.Name;
   Return Result;
end Zip_Of;

function "<" (Left : in Zip_Rec; Right : in Zip_Rec) return Boolean is
begin
   if Left.Zip = Right.Zip then
      return Left.Name < Right.Name;
   else
      return Left.Zip < Right.Zip;
   end if;
end "<";


package Zip_List is new Key_Ordered_List (Element_Type => Friend_Rec,
                                          Key_Type     => Zip_Rec,
                                          Key_Of       => Zip_Of,
                                          "="          => "=",
                                          "<"          => "<");

procedure Labels (File_Name : in String) is

   Label_File : Ada.Text_IO.File_Type;
   Friends    : Zip_List.List_Type(Max_Size => The_Book.Length);

   -- Local procedure for putting one friend into a zip ordered list
   procedure Insert_One_Friend (Friend : in out Friend_Rec) is
   begin
      Friends.Insert(Friend);
   end Insert_One_Friend;

   -- Local procedure for printing one label
   procedure Put_One_Label (Friend : in out Friend_Rec) is
   begin
      Put_Line (File => Label_File,
                Item => To_String(Friend.Name.First) & ' ' &
                        To_String(Friend.Name.Last));
      Put_Line (File => Label_File,
                Item => To_String(Friend.Address.Street));
      Put_Line (File => Label_File,
                Item => To_String(Friend.Address.City) & ", " &
                        Friend.Address.State & ' ');
      Put_Digits (File => Label_File,
                  Item => Friend.Address.Zip);
      New_Line(File => Label_File, Spacing => 2);
   end Put_One_Label;
```

```
begin
   -- Copy the book into a zip ordered list
   The_Book.Traverse (Insert_One_Friend'Access);
   Ada.Text_IO.Create (File => Label_File,
                       Name => File_Name);
   Friends.Traverse (Put_One_Label'Access);
   Ada.Text_IO.Close (Label_File);
end Labels;
```

12 a. In a sequential list, the elements the elements are physically ordered. So given the address of one element we can calculate the address of the next element in the list. With a linked list, the elements have no physical order. We have to use the link to locate the next element in the list.

b. We can search sequential lists faster than we can search linked lists. So any application where list observer operations are more common than adding and deleting elements would be better implemented with a sequential list.

c. Linked lists eliminate the need to move large numbers of elements required to insert or delete in a sequential list. So any application with a high frequency of adding and deleting elements would be better implemented with a linked list.

15 a. True (if you consider elaboration to be a form of dynamic memory allocation)
False (if you restrict dynamic memory allocation to memory obtained through the *new* operator)

b. True

c. False (all access variables have an initial value of null)

d. True

e. True

f. False (Ptr uses the memory of a single access value while One_Node uses the memory needed for an element plus an access value)

g. False (Node_Type is a record. We can use Node_Type as with any record type)

18 a. Valid

b. Invalid, assigning a record to an access variable

c. Invalid, assigning an access value to a record variable

d. Invalid, assigning an element type to an access variable

e. Valid

f. Valid

g. Valid syntax, however with the linked list shown on page 468 it dereferences a null pointer.

21. 20
10

24. 4
1
3
5
2
4

26 a.
```
procedure Traverse
      (List    : in out List_Type;
       Process : not null access procedure (Element : in out Element_Type));

-- Purpose        : To process all the elements in List in ascending order
-- Preconditions  : None
-- Postconditions : Every element in List is passed to a call of
--                  procedure Process
--                  Elements processed in ascending order
-- Exceptions     : KEY_ERROR raised when Process changes the key of Element
```

b.
```
procedure Traverse
      (List    : in out List_Type;
       Process : not null access procedure (Element : in out Element_Type)) is

   Current_Element : Element_Type;

begin
   for Index in 1..List.Length loop
      Current_Element := List.Items(Index);  -- Make a copy of the element
      Process (Current_Element);  -- Call user's procedure to process the copy
      if Key_Of (Current_Element) = Key_Of (List.Items(Index)) then
         List.Items(Index) := Current_Element; -- Put the element back into list
      else
         raise KEY_ERROR;  -- Process changed the key
      end if;
   end loop;
end Traverse;
```

c. Any list elements processed before KEY_ERROR is raised may be changed.

```
-- Exceptions     : KEY_ERROR raised when Process changes the key of Element
--                  Any elements processed before the exception is raised
--                  may be changed by this procedure
```

28. Function Size is identical to function Length which was already included in both packages

d. Size is O(1) for the array based key ordered list and O(*N*) for the linked-list based key ordered list.

31 a.

```
procedure Get_Element (List     : in  List_Type;
                       Position : in  Positive;
                       Element  : out Element_Type);
-- Purpose        : Retrieves the element at the given Position
-- Preconditions  : None
-- Postconditions : Element is a copy of the element at the given Position
-- Exceptions     : CONSTRAINT_ERROR raised if Position > List.Length
```

c.

```
procedure Get_Element (List     : in  List_Type;
                       Position : in  Positive;
                       Element  : out Element_Type) is
   Current : Node_Ptr;
begin
   Current := List.Head;            -- Set Current to designate first element
   for Count in 2..Position loop    -- Move Current to the Position element
      Current := Current.all.Next;
   end loop;
   Element := Current.all.Info;
end Get_Element;
```

When Position > List.Length, the above procedure raises CONSTRAINT_ERROR because Current runs off the end of the linked list and the procedure attempts to deference a null pointer.

d. Get_Element is O(1) for the array based key ordered list and O(*N*) for the linked-list based key ordered list.

32 c.
```
procedure Reverse_Traverse
          (List    : in out List_Type;
           Process : not null access procedure (Element : in out Element_Type));

   package Ptr_Stack is new Stack (Element_Type => Node_Ptr);

   Stack    : Ptr_Stack.Stack_Type(Max_Size => Length(List));
   Location : Node_Ptr;

begin

   Location := List.Head;

   -- Traverse the linked list pushing pointers to each
   --    of the nodes in the list onto the stack
   -- Each iteration, push one pointer onto the stack
   loop
      exit when Location = null;
      Stack.Push (Location);
      Location := Location.all.Next;
   end loop;

   -- Process all of the list elements in reverse order
   -- Each iteration, one list element is processed
   loop
      exit when Stack.Empty;
      Stack.Pop (Location);
      Process (Location.all.Info);
   end loop;

end Reverse_Traverse;
```

Chapter 8

2 a. O(*N*)

b. Twice

5. Insert at both the front and rear of the list is O(1).

9 a.
```
procedure Get_Element (List     : in  List_Type;
                       Position : in  Positive;
                       Element  : out Element_Type);
-- Purpose        : Retrieves the element at the given Position
-- Preconditions  : None
-- Postconditions : Element is a copy of the element at the given Position
-- Exceptions     : CONSTRAINT_ERROR raised if Position > List.Length
```

b.
```
procedure Get_Element (List     : in  List_Type;
                       Position : in  Positive;
                       Element  : out Element_Type) is
   Current : Node_Ptr;
begin
   Current := List.Tail.all.Next;  -- Set Current to designate first element
   for Count in 2..Position loop   -- Move Current to the Position element
      Current := Current.all.Next;
      if Current = List.Tail.all.Next then
         raise CONSTRAINT_ERROR;   -- we wrapped around
      end if;
   end loop;
   Element := Current.all.Info;
end Get_Element;
```

c. Get_Element is O(*N*).

12. More storage is used for pointers. Algorithms are more complex.

15. It cannot be done in O(1). We need to change the Next field of the predecessor of the last node. We cannot get directly to the predecessor of the last node. We must start at the first and work our way through the linked list to get to the predecessor of the last node.

18 a.
```
procedure Split_List (Main_List : in out List_Type;
                      Split_Key : in     Key_Type;
                      List_1    :    out List_Type;
                      List_2    :    out List_Type) is
   Found    : Boolean;
   Location : Node_Ptr;
begin
   Search_Circular_List (List     => Main_List,
                         Key      => Split_Key,
                         Found    => Found,
                         Location => Location);
   -- Change pointers to set up List_1
   List_1.Tail                   := Location.all.Back;
   List_1.Tail.all.Next          := Main_List.Tail.all.Next;
   List_1.Tail.all.Next.all.Back := List_1.Tail;
   -- Change pointers to set up List_2
   List_2.Tail          := Main_List.Tail;
   List_2.Tail.all.Next := Location;
   Location.all.Back    := Main_List.Tail;
   -- "Empty" Main_List
   Main_List.Tail := null;
end Split_List;
```

21 a. The special case of adding a node at the beginning of the list (where the external pointer is changed).

b. The condition checking for a null pointer in the exit statement of the search loop.

c. No. Push and pop both change the external pointer. Changing the external pointer is not a special case.

d. No. Enqueue and dequeue both change external pointes. Changing an external pointer is not a special case.

e. No. Enqueue and dequeue both change external pointes. Changing an external pointer is not a special case.

23 a. Horace and Mildred probably used a dummy node with a key less than Zzuan.

b. By using a name made up of the characters Character'Last.

26. In this data structure, an empty list consists of two nodes–a header node and a trailer node. We override procedure Initialize to create these two dummy nodes.

30 b.

```
procedure Get_Element (List     : in  List_Type;
                       Position : in  Positive;
                       Element  : out Element_Type) is
   Current : Node_Ptr;
begin
   Current := List.Head.all.Next;  -- Set Current to designate first element
   for Count in 2..Position loop   -- Move Current to the Position element
      Current := Current.all.Next;
   end loop;
   if Current.all.Info = Max_Element then
      raise CONSTRAINT_ERROR;
   end if;
   Element := Current.all.Info;
end Get_Element;
```

When Position > List.Length, the above procedure raises CONSTRAINT_ERROR because Current runs off the end of the linked list and the procedure attempts to deference a null pointer.

33 a. Doubly linked. We need to access the predecessors and successors of a node

b. Circular. Gives us O(1) insert at the end of the list.

c. Header and trailer nodes. Eliminates the special cases for adding to and deleting from an empty list.

36 a. John
Nell
Susan
Suzanne

b. David
Naomi
Robert

c. 8
1
9

d. Free is changed from 8 to 10. Here is what the array looks like.

Nodes	.Info	.Next
1	Miriam	9
2	Naomi	5
3	David	2
4	John	6
5	Robert	0
6	Susan	7
7	Suzanne	0
8	Joshua	1
9	Leah	0
10	Nell	8

e. Free is changed from 8 to 1. Here is what the array looks like.

Nodes	.Info	.Next
1	Miriam	9
2	Naomi	5
3	David	2
4	John	10
5	Robert	8
6	Susan	7
7	Suzanne	0
8	Tracy	0
9	Leah	0
10	Nell	6

39. O(1)

40. The Size function returns the number of elements in the file. Deleting an element from the file-based list does not reduce the size of the file. Our memory management logic only extends the size of the file when there are no unused locations in the file.

Chapter 9

1 a. A non-recursive solution for a trivial case of the problem being solved.

b. A combination of steps involving one or more recursive calls to solve smaller cases of the original problem.

c. A subprogram calling itself.

d. A subprogram that calls another subprogram that eventually calls the original subprogram to solve a smaller case of the original problem.

4 a. There are two base cases: `Base > Limit` and `Base = Limit`

b. `Result := Base * Puzzle (Base + 1, Limit);`

6 a. `type Num_Type is range Integer'First .. 0;`

9 a. A postfix expression contains two operands.
An operand may be a postfix expression

b. The operands of a postfix expression are integers.

12 a. The parameters must not be negative.

```
subtype Float_Type is Float range 0.0 .. Float'Last;
```

b.

```
function Sqrt (Num    : in Float_Type;
               Approx : in Float_Type;
               Tol    : in Float_Type) return Float_Type is
begin
   if abs (Approx ** 2 - Num) <= Tol then
      return Approx;
   else
      return Sqrt (Num    => Num,
                   Approx => (Approx ** 2 + Num) / (2.0 * Approx),
                   Tol    => Tol);
   end if;
end Sqrt;
```

15.
```
procedure Traverse (List : in List_Type) is
-- Traverses List in order.
begin
   if List = null then
      null; -- list is empty (do nothing)        -- Base case
   else
      Process (Element => List.all.Info);         -- General
      Traverse (List => List.all.Next);           -- Case
   end if;
end Traverse;
```

`Reverse_Traverse` is a much better use of recursion. `Traverse` could be done as easily with iteration.

18 a. A data structure that keeps track of activation records during the execution of a program.

b. The point in the compile-link-execution cycle when variable names are associated with addresses in memory.

c. The case where a recursive subprogram contains a single recursive call that is the last statement executed in the subprogram.

d. The association of a memory address with a variable name.

21. With dynamic storage allocation, variables are not bound to actual addresses in memory until run time. With recursion, parameters and local variables are assigned actual addresses (on the run-time stack) with each recursive call.

24. The base cases are

We have already escaped from the maze

The current position is the exit

The current position is a trap

The current position has already been tried

Chapter 10

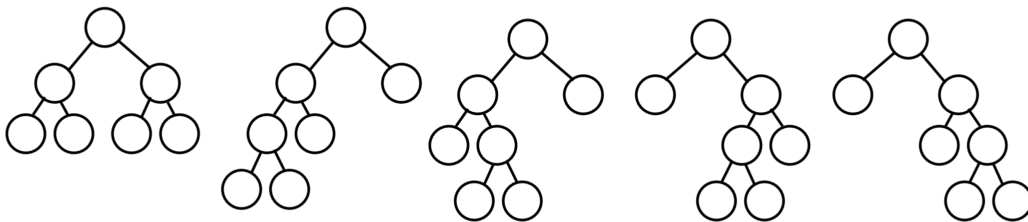
1 a. The maximum search time is proportional to the number of levels in the tree.

b. 100

c. 7

4. *N* ancestors

7.



10 a. False

b. True

c. False

13. Procedure Find changes the state of the binary search tree. It changes which node is designated as the current node.

17. Modify and Retrieve raise CURRENT_UNDEFINED when the current element is not defined.

20.

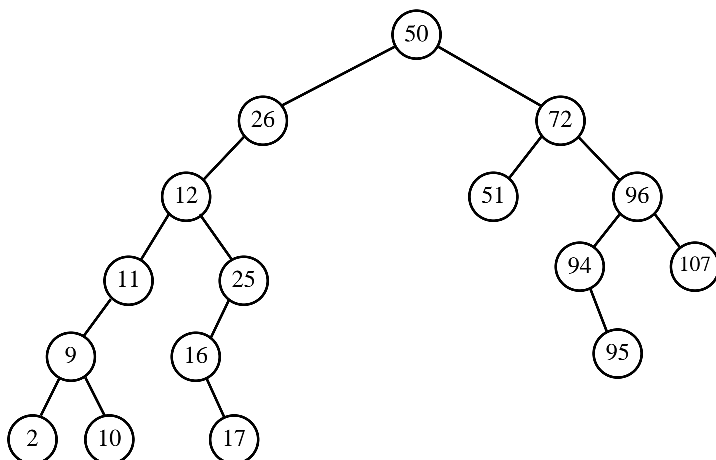
```
procedure Greatest_Frequency (Tree   : in out Freq_Tree.Tree_Type;
                              Result :    out Natural) is

   -- This local procedure uses the local global parameter Result
   procedure Check_One_Word (Word : in out Freq_Rec) is
   begin
      if Word.Freq > Result then
         Result := Word.Freq;
      end if;
   end Check_One_Word;

begin
   Result := 0;
   Tree.Traverse (Order   => Inorder,
                  Process => Check_One_Word'Access);
end Greatest_Frequency;
```

23 c. We need to create and display a list of women students ordered by GPA. We can use a binary search tree to implement such a list. Because there may be duplicate GPAs in the list, we cannot create a tree with GPA as the key. Our approach is to create a unique key by combining the GPA and ID number.

```
type GPA_Rec is
   record
      GPA  : GPA_Type;
      ID   : Positive;
   end record;

function GPA_Of (Student : in Student_Rec) return GPA_Rec is
   Result : GPA_Rec;
begin
   Result.GPA := Student.GPA;
   Result.ID  := Student.ID_Num;
   return Result;
end GPA_Of;

function ">" (Left : in GPA_Rec; Right : in GPA_Rec) return Boolean is
begin
   if Left.GPA = Right.GPA then
      return Left.ID > Right.ID;
   else
      return Left.GPA > Right.GPA;
   end if;
end ">";

package GPA_Tree is new Binary_Search_Tree
                        (Element_Type => Student_Rec,
                         Key_Type     => GPA_Rec,
                         Key_Of       => GPA_Of,
                         "<"          => ">",
                         "="          => "=");

procedure Women_By_GPA (Tree : in out Student_Tree.Tree_Type) is

   -- Contains two nested procedures

   Women : GPA_Tree.Tree_Type;

   procedure Insert_One_Woman (Student : in out Student_Rec) is
   begin
      if Student.Sex = Female then
         Women.Insert(Student);
      end if;
   end Insert_One_Woman;


   procedure Put_One_Student (Student : in out Student_Rec) is
   begin
      Put (Student.First_Name & ' ' & Student.Last_Name);
      Put (Item => Student.GPA, Fore => 2, Aft => 2, Exp =>0);
      Put (Item => Student.ID_Num, Width => 8);
      New_Line;
   end Put_One_Student;
```

```
begin -- procedure Women_By_GPA
   -- Insert all of the women into a tree ordered by GPA
   Tree.Traverse (Order   => Student_Tree.Preorder,
                  Process => Insert_One_Woman'Access);
   -- Display all of the women in order by GPA
   Women.Traverse (Order   => GPA_Tree.Inorder,
                   Process => Put_One_Student'access);
end Women_By_GPA;
```

26.



29. False

32 a. O($\log_2 N$) Mathematical note: The expected height of a binary search tree made by inserting keys in random order is 1.39 $\log_2 N$, about 40% taller than the minimum height tree.

b. O(N)

35 a.

```
procedure Ancestors
        (Tree    : in Tree_Type;
         Key   : in Key_Type;
         Process : not null access procedure (Element : in out Element_Type)) is
   Current : Node_Ptr;
begin
   Current := Tree.Root;
   loop
      exit when Current = null;
      Process (Current.all.Info);
      if Key < Key_Of (Current.all.Info) then
         Current := Current.all.Left;
      else
         Current := Current.all.Right;
      end if;
   end loop;
end Ancestors;
```

b. Instead of calling the procedure Process each time through the loop, push a copy of Current onto a stack. Add a second loop that pops the pointers off of the stack and processes the nodes. See the solution to Exercise 32c in Chapter 7 for code that you can modify to implement this algorithm.

c.
```
procedure Ancestors
        (Tree    : in Tree_Type;
         Key  : in Key_Type;
         Process : not null access procedure (Element : in out Element_Type)) is

   procedure Recursive_Ancestors (Root : Node_Ptr) is
   begin
      if Root /= null then
         -- Process all of the descendants before the root
         if Key < Key_Of (Root.all.Info) then
            Recursive_Ancestors (Root => Root.all.Left);
         else
            Recursive_Ancestors (Root => Root.all.Right);
         end if;
         Process (Root.all.Info);
      end if;
   end Recursive_Ancestors;
begin
   Recursive_Ancestors (Root => Tree.Root);
end Ancestors;
```

38.
```
procedure Copy (Source : in  Tree_Type;
                Target : out Tree_Type) is

   function Copy (Root: in Node_Ptr) return Node_Ptr is
   -- Recursive local function
   begin
      if Root = null then
         return null;
      else
         return new Node_Type'(Info  => Root.all.Info,
                               Left  => Copy (Root => Root.all.Left),
                               Right => Copy (Root => Root.all.Right));
      end if;
   end Copy;

begin
   Target.Root    := Copy (Source.Root);
   Target.Current := null;
end Copy;
```

41 a. Only the tree in figure (e) satisfies the binary search tree property.

b. The trees in figures (b), (d), and (e) are complete binary trees.

c. The trees in figures (b) and (e) are complete binary trees.

44 a. True. The internal nodes are those stored in locations 1 through 42. 42 is the index of the last internal node as it is the parent of the last leaf node (index 85).

b. False. The children of the node at index 42 are at indices 84 and 85. Since the last index in the tree is 85, the node at index 42 has two children.

c. False. The right child of Tree(13) is Tree(27).

d. True. By drawing a picture the subtree rooted at Tree(8) we see that the leaf nodes have indices 64 to 71. All of these nodes are in the original tree.

e. False. Draw the leftmost and rightmost branches of the tree. The leaf at the end of the leftmost branch, Tree(64), is at the 7th level. The leaf at the end of the rightmost branch, Tree(63), is at the 6th level. Therefore the tree has 6 perfect levels and one additional level that contains some elements.

Chapter 11

2. Someone with many items may wait for a <u>very</u> long time as people with fewer items keep coming and getting in front of them.

4 b.
```
subtype Name_String is String (1..20);
type    GPA_Type    is digits 4 range 0.0..4.0;

type Student_Rec is
   record
      Name   : Name_String;
      GPA    : GPA_Type;
      Missed : Natural;
   end record;

function Missed_Days (Student : in Student_Rec) return Natural is
begin
   return Student.Missed;
end Missed_Days;

package Student_Queue is new Priority_Queue
                         (Element_Type  => Student_Rec,
                          Priority_Type => Natural,
                          Priority_Of   => Missed_Days,
                          ">"           => Standard."<");
```

6 a. The item is the most recent time stamp. This is the item having the largest time value.

b. The instantiation would go in the private part of the stack specification as follows

```
private

   type Time_Stamped_Element is
      record
         Element : Element_Type;          -- The stack element type
         Time    : Ada.Calendar.Time;
      end record;

   function Priority_Of (Element : in Time_Stamped_Element)
            return Ada.Calendar.Time;

   package Stack_Priority_Queue is new
                  Priority_Queue (Element_Type  => Time_Stamped_Element,
                                  Priority_Type => Ada.Calendar.Time,
                                  Priority_Of   => Priority_Of,
                                  ">"           => Ada.Calendar.">");

   type Stack_Type (Max_Size : Natural) is tagged limited
      record
         PQ : Stack_Priority_Queue.Queue_Type(Max_Size);
      end record;
end Stack;
```

c.

```
procedure Push (Stack       : in out Stack_Type;
                New_Element : in     Element_Type) is
   Stamped_Element : Time_Stamped_Element;
begin
   Stamped_Element.Element := New_Element;
   Stamped_Element.Time    := Ada.Calendar.Clock;
   Stack.PQ.Enqueue (Stamped_Element);
end Push;


procedure Pop (Stack          : in out Stack_Type;
               Popped_Element :    out Element_Type) is
   Stamped_Element : Time_Stamped_Element;
begin
   Stack.PQ.Dequeue (Stamped_Element);
   Popped_Element := Stamped_Element.Element;
end Pop;
```

d. Push and pop are both O(1) for the array based stack described in Chapter 5. As they simply call the enqueue and dequeue operations, push and pop are both $O(\log_2 N)$ for this priority queue based stack.

e. No. The priority queue implementation of the stack has a poorer Big-O. It may also produce incorrect results. If the clock ticks slower than the programs pushes items we may push multiple items onto the stack with the same time stamps.

9 a.
```
private

   type Node_Type;
   type Node_Ptr is access Node_Type;
   type Node_Type is
      record
         Info  : Element_Type;
         Left  : Node_Ptr;
         Right : Node_Ptr;
      end record;

   type Queue_Type is new Ada.Finalization.Limited_Controlled with
      record
         Root : Node_Ptr;     -- Designates first node in the tree
      end record;

   overriding procedure Finalize (Queue: in out Queue_Type) renames Clear;

end Priority_Tree;
```

b.
```
procedure Enqueue (Queue : in out Queue_Type;
                   Item  : in     Element_Type) is

   procedure Insert (Root : in out Node_Ptr) is
   begin
      if Root = null then
         Root := new Node_Type'(Item, null, null);
      elsif Priority_Of (Item) > Priority_Of (Root.all.Info) then
         Insert (Root.all.Right);
      else
         Insert (Root.all.Left);
      end if;
   end Insert;

begin
   Insert (Queue.Root);
exception
   when STORAGE_ERROR =>
      raise OVERFLOW;
end Enqueue;
```

c.
```
procedure Dequeue (Queue : in out Queue_Type;
                   Item  :    out Element_Type) is
   Parent : Node_Ptr;
   Child  : Node_Ptr;
begin
   if Queue.Root = null then
      raise UNDERFLOW;
   end if;

   -- The node with the highest priority is at the
   -- end of the tree's right branch
   Child := Queue.Root;
   loop
      exit when Child.all.Right = null;
      Parent := Child;
      Child  := Child.all.Right;
   end loop;
   Item := Child.all.Info;

   -- Remove the node from the tree
   if Parent = null then
      Queue.Root := Queue.Root.all.Left;
   else
      Parent.all.Right := Child.all.Left;
   end if;
   Free (Child);
end Dequeue;
```

d. The Big-O depends on the height of the binary tree. The operations are $O(\log_2 N)$ for complete trees and $O(N)$ for degenerate trees.

12. Any binary search tree in which there are no right children.

15. By supplying a *less than* operator as the actual generic parameter for the heap's generic formal parameter ">".

18. Position 8 can contain any letter less than or equal to 'E'.
Position 9 can contain any letter less than or equal to 'E'.
Position 10 can contain any letter less than or equal to 'B'.

21 a.
```
package Word_Strings is new Ada.Strings.Bounded.Generic_Bounded_Length (50);
```

b.
```
package Word_Queue is new Priority_Queue
                         (Element_Type  => Word_Strings.Bounded_String,
                          Priority_Type => Natural,
                          Priority_Of   => Word_Strings.Length,
                          ">"           => Standard.">");
```

24 a. V (State Graph) = {Oregon, Alaska, Texax, Hawaii, Vermont, New York, California}

E (State Graph_ = {(Alaska, Oregon), (Hawaii, Alaska), (Hawaii, Texas), (Hawaii, New York), (Hawaii, California), (Texas, Hawaii), (Texas, Vermont), (Vermont, Alaska), (Vermont, New York)}

b. No.

c. Yes

d. Texas

e.

(1)	Alaska
(2)	California
(3)	Hawaii
(4)	Oregon
(5)	New York
(6)	Texas
(7)	Vermont

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
(1)	**F**	**F**	**F**	**T**	**F**	**F**	**F**
(2)	**F**	**F**	**F**	**F**	**F**	**F**	**F**
(3)	**T**	**T**	**F**	**F**	**T**	**T**	**F**
(4)	**F**	**F**	**F**	**F**	**F**	**F**	**F**
(5)	**F**	**F**	**F**	**F**	**F**	**F**	**F**
(6)	**F**	**F**	**T**	**F**	**F**	**F**	**T**
(7)	**T**	**T**	**F**	**F**	**T**	**F**	**F**

27.

Washington	Atlanta	Denver	(Atlanta taking off)
Washington	Atlanta		
Washington			
queue empty			(Washington taking off)

29.

Operation	**Big-O**
Index_Of	O(*N*)
Clear	O(1)
Add_Vertex	O(*N*)
Add_Edge	O(*N*)
Retrieve	O(*N*)
Weight_Of	O(*N*)
Get_Adjacent_Vertices	O(*N*)
Clear_All_Marks	O(*N*)
Mark_Vertex	O(*N*)
Marked	O(*N*)

32.
```
procedure Delete_Edge (Graph : in out Graph_Type;
                       Edge  : in     Edge_Type) is
   From_Index : Positive;
   To_Index   : Positive;
begin
   From_Index := Index_Of (Graph, Edge.From);   -- Location of From vertex
   To_Index   := Index_Of (Graph, Edge.To);     -- Location of To vertex
   -- Do both vertices exist?
   if From_Index > Graph.Num_Vertices  or  To_Index > Graph.Num_Vertices  then
      raise VERTEX_ERROR;
   -- Is the edge defined in the graph?
   elsif not Graph.Edges(From_Index, To_Index).Defined then
      raise EDGE_ERROR;
   else
      -- Remove the edge from the adjacency matrix
      Graph.Edges(From_Index, To_Index).Defined := False;
   end if;
end Delete_Edge;

procedure Delete_Vertex (Graph : in out Graph_Type;
                         Key   : in     Key_Type) is
   Index : Positive;
begin
   Index := Index_Of (Graph, Key);
   if Index > Graph.Num_Vertices then   -- Is Key in the Graph?
      raise VERTEX_ERROR;
   end if;
   -- Remove the node from the vertex array and edge matrix
   for Row in Index .. Graph.Num_Vertices - 1 loop
      -- Shuffle one vertex up
      Graph.Vertices(Row) := Graph.Vertices(Row + 1);
      -- Shuffle one row of the edge matrix up
      for Column in 1..Graph.Num_Vertices loop
         Graph.Edges(Row, Column) := Graph.Edges(Row + 1, Column);
      end loop;
   end loop;
   Graph.Num_Vertices := Graph.Num_Vertices - 1;
end Delete_Vertex;
```

36.
```
private

   type Vertex_Node;
   type Edge_Node;

   type Vertex_Node_Ptr is access Vertex_Node;
   type Edge_Node_Ptr   is access Edge_Node;

   type Vertex_Node is
      record
         Info   : Vertex_Type;        -- Information declared by user
         Marked : Boolean := False;   -- Has the Vertex been visited?
         Edges  : Edge_Node_Ptr;      -- Pointer to first edge in list
         Next   : Vertex_Node_Ptr;    -- Link to next vertex
      end record;

   type Edge_Node is
      record
         Weight  : Weight_Type;        -- Weight assigned to the edge
         Defined : Boolean := False;   -- Is this edge in the graph?
         Next    : Edge_Node_Ptr;      -- Link to next edge
      end record;


   type Graph_Type is new Ada.Finalization.Limited_Controlled with
      record
         Head : Vertex_Node_Ptr;
      end record;

   overriding procedure Finalize (Graph : in out Graph_Type) renames Clear;

end Digraph;
```

Chapter 12

1 a.
```
type Float_Array is array (Positive range <>) of Float;
procedure Sort_Floats is new Sort (Element_Type => Float,
                                   Array_Type   => Float_Array,
                                   "<"          => Standard."<");
```

b.
```
type Float_Array is array (Positive range <>) of Float;
procedure Sort_Floats is new Sort (Element_Type => Float,
                                   Array_Type   => Float_Array,
                                   "<"          => Standard.">");
```

4. The selection sort is always $O(N^2)$

7. Additional time is required to set and check the sorted flag.

9 a. $O(N^2)$

b. $O(N)$

c. $O(N^2)$

d. $O(N^2)$

12. The number of iterations of the outer loop is independent of the number of integers in the array being sorted so it is O(1). The number of iterations of the inner loop is equal to the number of integers in the array so it is $O(N)$. The nesting is therefore $O(N)$.

15. Base case a list of zero or one element is already sorted.
Smaller caller each recursive call is made with a list half the size of the original.
General case when we merge two sorted lists, the result is a sorted list

18 a. False

b. True

c. False

d. False

21 a. So we don't compare the split value to itself

b. Left can exceed Values'Last which would raise CONSTRAINT_ERROR when using Left to index the array Values. Right will never be less that Value'First as the pivot element at that location.

24.
```
begin
   if Values'Length > Limit then
      Split (Values, Split_Index);
      Quick_Sort (Values(Values'First .. Split_Index - 1));
      Quick_Sort (Values(Split_Index + 1 .. Values'Last));
   else
      Insert_Sort (Values);
   end if;
end Quick_Sort;
```

27 a. $O(N\log_2 N)$

b. $O(N^2)$

c. $O(N\log_2 N)$

30 a. True Merge sort requires twice the storage as it uses two arrays each of whose length is equal to the number of elements being sorted. Heap sort uses a single array for the elements.

b. False Quick sort (using the first element as the split value) is $O(N^2)$ for nearly sorted data while heap sort is $O(N\log_2 N)$ for all data.

c. True The execution times of the for loops of the heap sort are independent of data order.

33 a. N = 3

b. N = 9

c. N = 5

37. An unstable sort is one in which the order of two equal keys is maintained. The reheap down algorithm may reverse the original order.

39 a. False

b. True

c. False

d. True

42 a & b.

```
Students : Student_List (Max_Size => Max_Students);

type Pointer_Array is array (Positive range <>) of Positive;

function "<" (Left : in Name_Type; Right : in Name_Type) return boolean is
begin
   if Left.Last_Name < Right.Last_Name then
      return True;
   elsif  Left.Last_Name > Right.Last_Name then
      return False;
   else
      return Left.First_Name < Right.First_Name;
   end if;
end "<";
```

```
function Less_Then (Left : in Positive; Right : in Positive) return boolean is
-- This function accesses the global variable Students
begin
   return Students.Students(Left).Student_Name <
     Students.Students(Right).Student_Name;
end Less_Then;

procedure Sort_Pointers is new Quick_Sort (Element_Type => Positive,
                                           Array_Type   => Pointer_Array,
                                           "<"          => Less_Then);

procedure Initialize (Item : out Pointer_Array) is
begin
   for Index in Item'Range loop
      Item (Index) := Index;
   end loop;
end Initialize;

By_Name : Pointer_Array (1..Max_Students);
```

c. `Sort_Pointers (Values => By_Name(1..Students.Num_Students));`

45. O(1)

48 a. 50

b. 50

c. 50

51 a.

285	153					177	140		66	47	87	126	145	90	393	395	467	
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)

b.

285	153	395				177	140		66		87	47	393	90	126		467	145
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)

c.

285	153					177	140		66	47	87	126	145	90	395	467		393
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)

d.

285	153	145	47			177	140	467	66		87	126	393	90	395			
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)

54 a. 1019

b. 1021

57 a. 1

b. 3

c. 3

d. 5

e. 5

f. 4

60. Hash table with 10 slots. Each slot is a bucket with 3 elements

	0	1	2	3	4	5	6	7	8	9
1	90			153		145	66	47	467	
2	140			393		285	126	87		
3						395		177		

62. Only room for the first 10 numbers. -1 used to indicate the end of a chain

	0	1	2	3	4	5	6	7	8	9
Value	87	90	126	140	153	145	66	47	177	285
Link	1	3	-1	4	8	9	2	0	-1	-1

65. With direct chaining we need two direct access files. One file is a file of external pointers equivalent to the array of pointers in Figure 12.28. The second file is a file of nodes. We need to manage the memory of this second file using the techniques similar to those of the file-based linked list discussed in Chapter 8.

68. Unless the integer keys being hashed are small, this technique does not make use of all of the digits in the key. It would be better to select the four middle digits of the cube rather than the rightmost four digits.

71. As long as a single bit in a group being `Or`'ed is one, the result is one. `Or`'ing a group of bits almost always yields one. The result of `Xor` depends on whether the number of ones is odd or even.