Basics (Chapter 1)

- Pseudocode
  - Express algorithm logic in English
  - Use some structured code
- Abstract Data Type (ADT)
  - Encapsulation of data and operations
  - Implementation of ADT
- Algorithm Efficiency
  - Big O notation
- ADT List Implementation
  - Array Implementation
  - Linked List Implementation
  - Void pointer usage
  - Pointer to functions

1-1 Pseudocode

Pseudocode is an English-like representation of the algorithm logic. It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.

- Algorithm Header
- Purpose, Condition, and Return
- Statement Numbers
- Variables
- Statement Constructs
- Algorithm Analysis

ALGORITHM 1-1 Example of Pseudocode

```
Algorithm sample (pageNumber)
This algorithm reads a file and prints a report.
   Pre    pageNumber passed by reference
   Post   Report Printed
          pageNumber contains number of pages in report
   Return Number of lines printed
 1 loop (not end of file)
   1  read file
   2  if (full page)
      1  increment page number
      2  write page heading
   3  end if
   4  write report line
   5  increment line count
 2 end loop
 3 return line count
end sample
```

ALGORITHM 1-2 Print Deviation from Mean for Series

```
Algorithm deviation
   Pre    nothing
   Post   average and numbers with their deviation printed
 1 loop (not end of file)
   1  read number into array
   2  add number to total
   3  increment count
 2 end loop
 3 set average to total / count
 4 print average
 5 loop (not end of array)
   1  set devFromAve to array element - average
   2  print array element and devFromAve
 6 end loop
end deviation
```

1-2 The Abstract Data Type

An ADT consists of a data declaration packaged together with the operations that are meaningful on the data while embodying the structured principles of encapsulation and data hiding. In this section we define the basic parts of an ADT.

- Atomic and Composite Data
- Data Type
- Data Structure
- Abstract Data Type

Type	Values	Operations
integer	-∞, ... , -2, -1, 0, 1, 2,..., ∞	*, +, -, %, /, ++, --, ...
floating point	-∞, ... , 0.0, ... , ∞	*, +, -, /, ...
character	\0, ..., 'A', 'B', ... , 'a', 'b', ... , ~	<, >, ...

TABLE 1-1 Three Data Types

Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

TABLE 1-2 Data Structure Examples

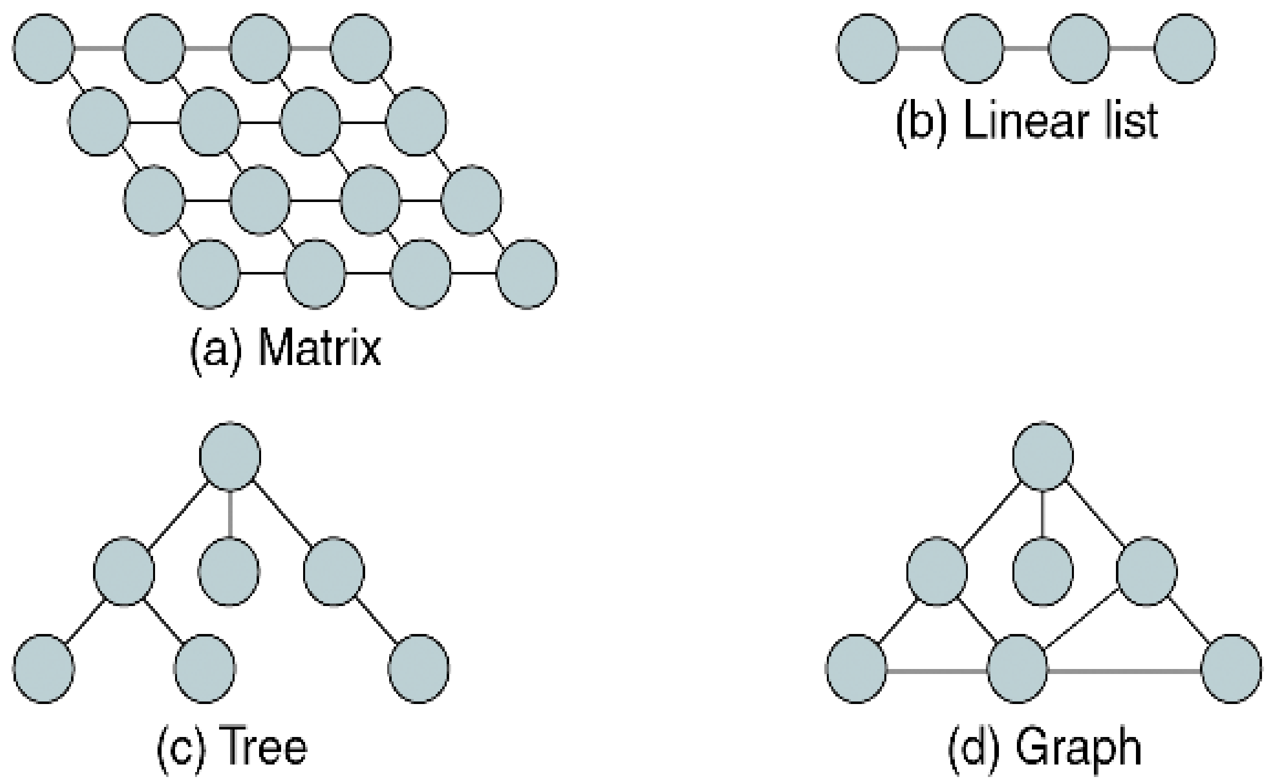

FIGURE 1-1 Some Data Structures

1-3 Model for an Abstract Data Type

In this section we provide a conceptual model for an Abstract Data Type (ADT).

- ADT Operation
- ADT Data Structure

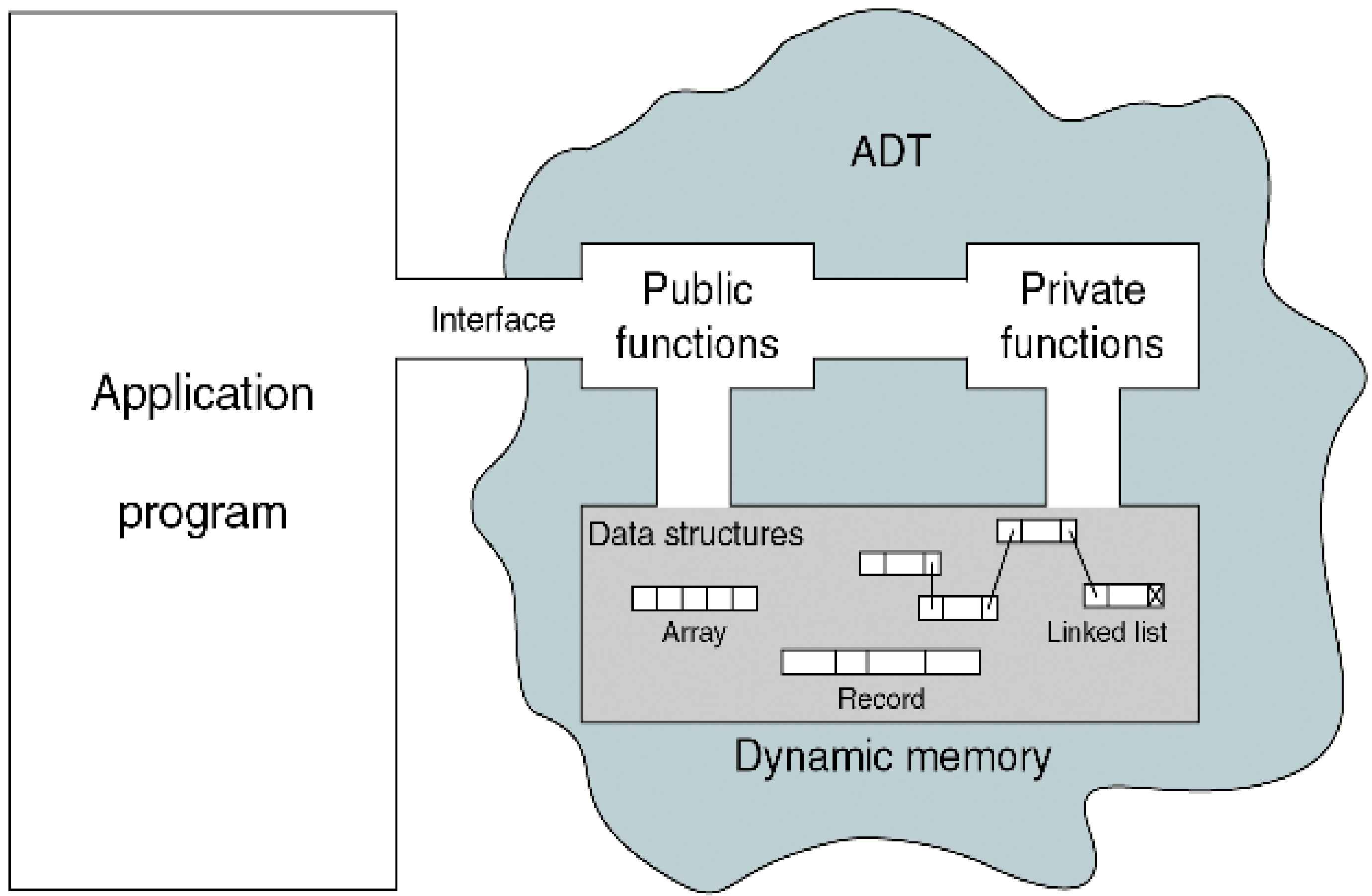

FIGURE 1-2 Abstract Data Type Model

1-4 ADT Implementations

There are two basic structures we can use to implement an ADT list: arrays and linked lists. In this section we discuss the basic linked-list implementation.

- Array Implementation
- Linked List Implemenation

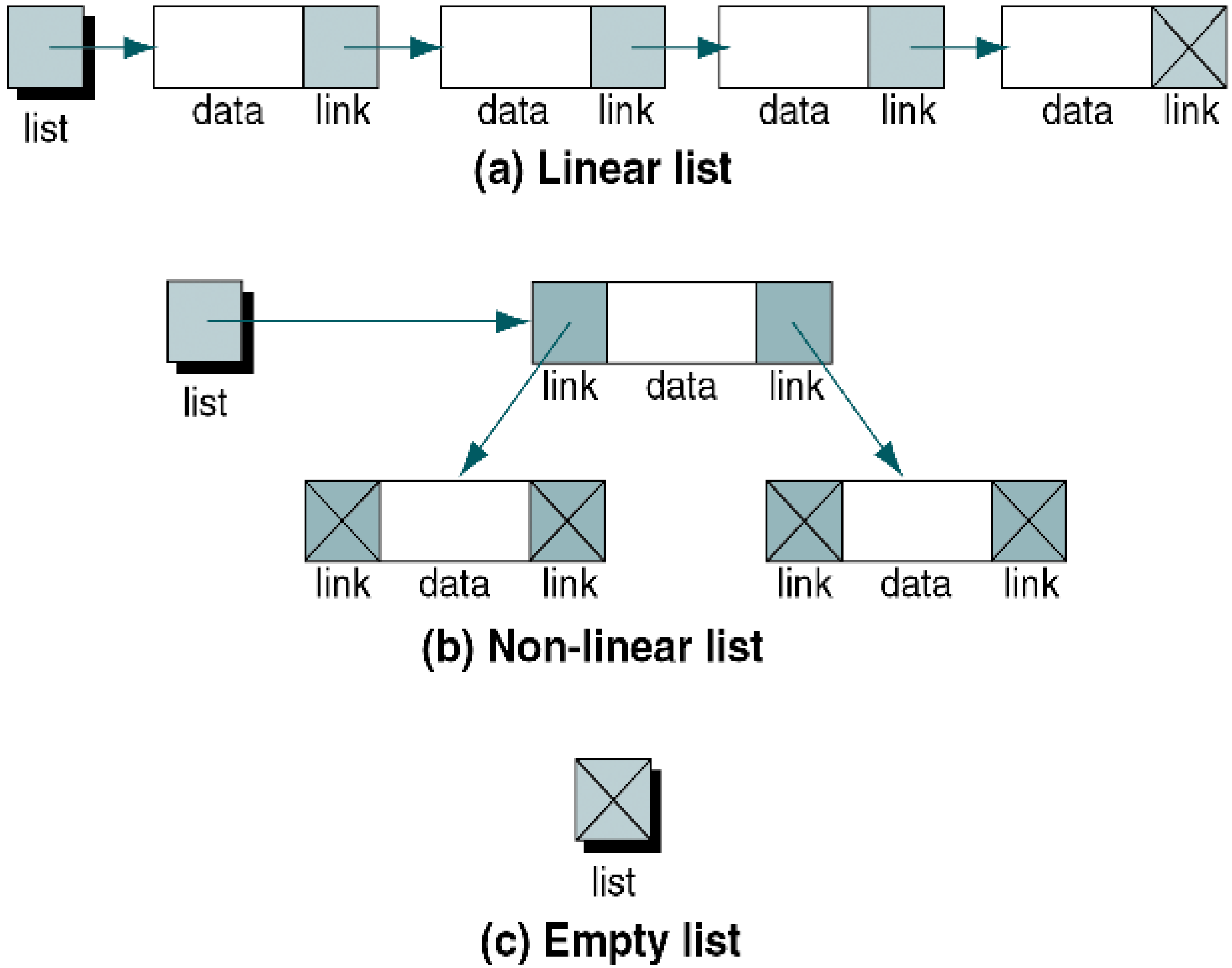

FIGURE 1-3 Linked Lists

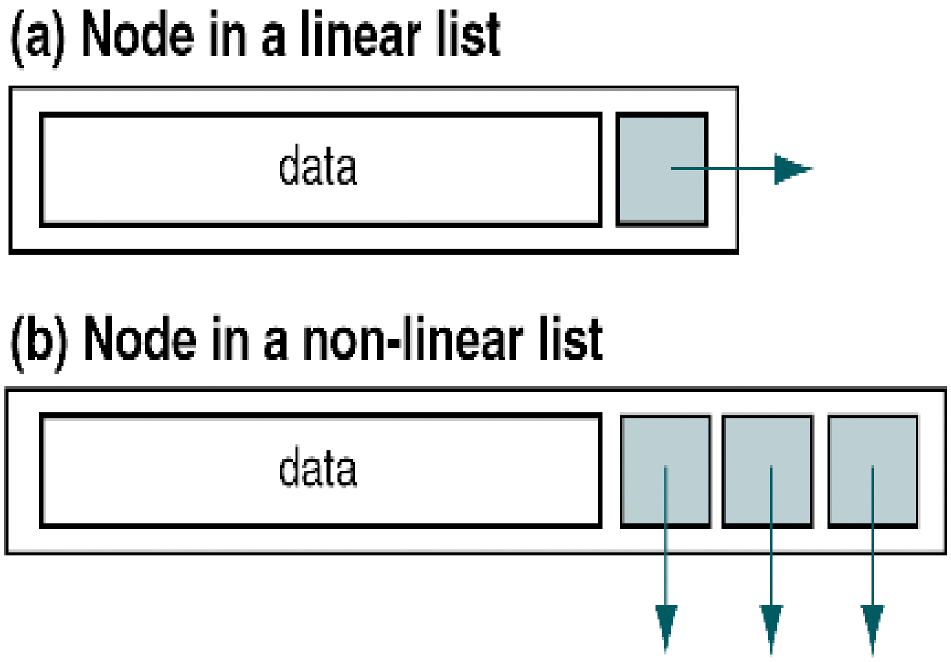

FIGURE 1-4 Nodes

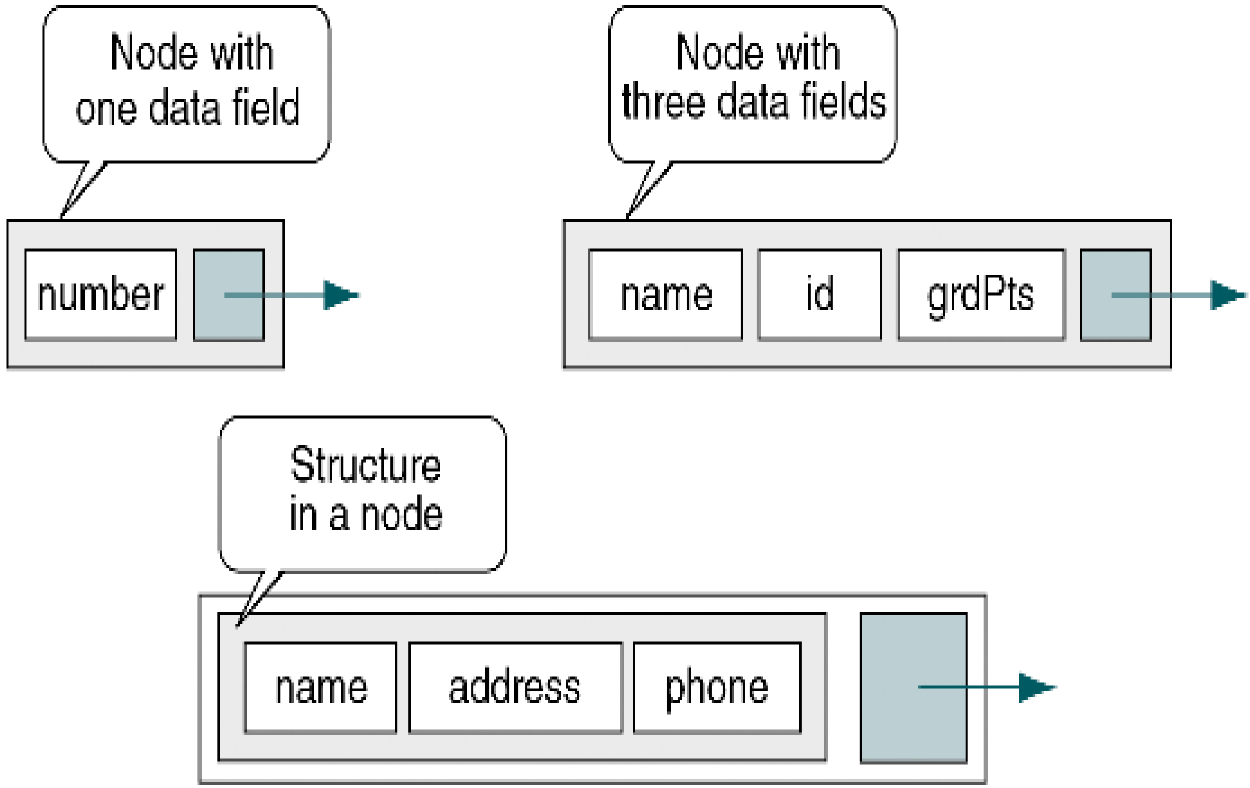

FIGURE 1-5 Linked List Node Structures

1-5 Generic Code for ADT

In this section we discuss and provide examples of two C tools that are required to implement an ADT.

- Pointer to Void
- Pointer to Function

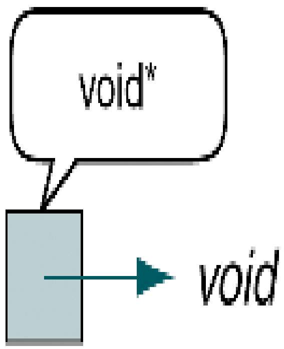

FIGURE 1-6 Pointer to *void*

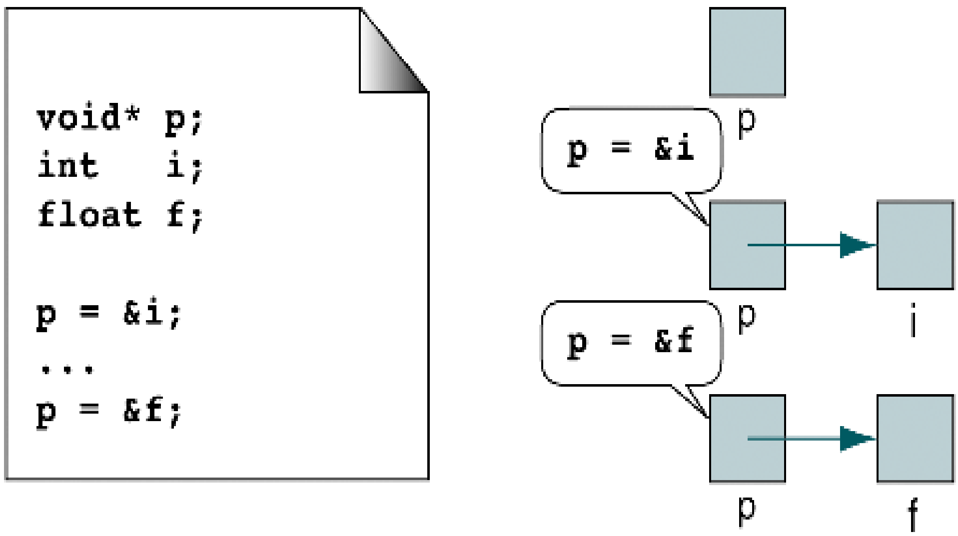

FIGURE 1-7 Pointers for Program 1-1

PROGRAM 1-1 Demonstrate Pointer to *void*

```
/* Demonstrate pointer to void.
      Written by:
      Date:
*/
#include <stdio.h>

int main ()
{
// Local Definitions
   void* p;
   int   i = 7 ;
   float f = 23.5;

// Statements
   p = &i;
   printf ("i contains: %d\n", *((int*)p) );

   p = &f;
   printf ("f contains: %f\n", *((float*)p));

   return 0;
}  // main
```

```
Results:
i contains 7
f contains 23.500000
```

Implementing a List of int/float/char .. etc

```
typedef struct node
{
         void* dataPtr;
  struct node* link;
} NODE;
```

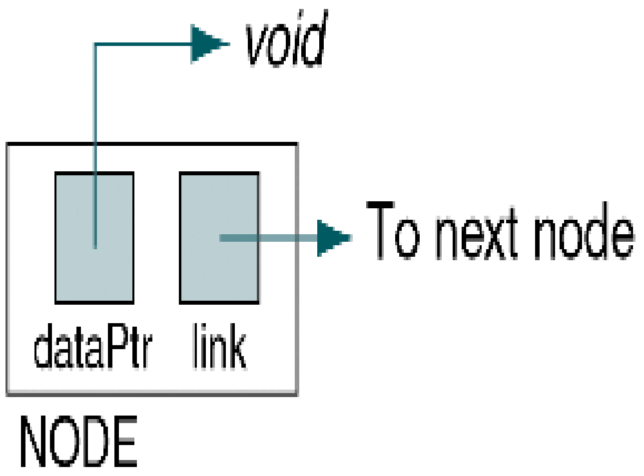

FIGURE 1-8 Pointer to Node

PROGRAM 1-2 Create Node Header File

```
/* Header file for create node structure.
      Written by:
      Date:
*/
typedef struct node
{
         void* dataPtr;
  struct node* link;
} NODE;
```

PROGRAM 1-2 Create Node Header File (Continued)

```
/* =================== createNode ====================
   Creates a node in dynamic memory and stores data
   pointer in it.
      Pre  itemPtr is pointer to data to be stored.
      Post node created and its address returned.
*/
NODE* createNode (void* itemPtr)
{
   NODE* nodePtr;
   nodePtr = (NODE*) malloc (sizeof (NODE));
   nodePtr->dataPtr = itemPtr;
   nodePtr->link    = NULL;
   return nodePtr;
} // createNode
```

PROGRAM 1-3 Demonstrate Node Creation Function

```
/* Demonstrate simple generic node creation function.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P1-02.h"                          // Header file

int main (void)
{
// Local Definitions
   int*  newData;
   int*  nodeData;
   NODE* node;

// Statements
   newData  = (int*)malloc (sizeof (int));
   *newData = 7;
```

continued

PROGRAM 1-3 Demonstrate Node Creation Function *(continued)*

```

   node = createNode (newData);

   nodeData = (int*)node->dataPtr;
   printf ("Data from node: %d\n", *nodeData);
   return 0;
} // main
```

```
Results:
Data from node: 7
```

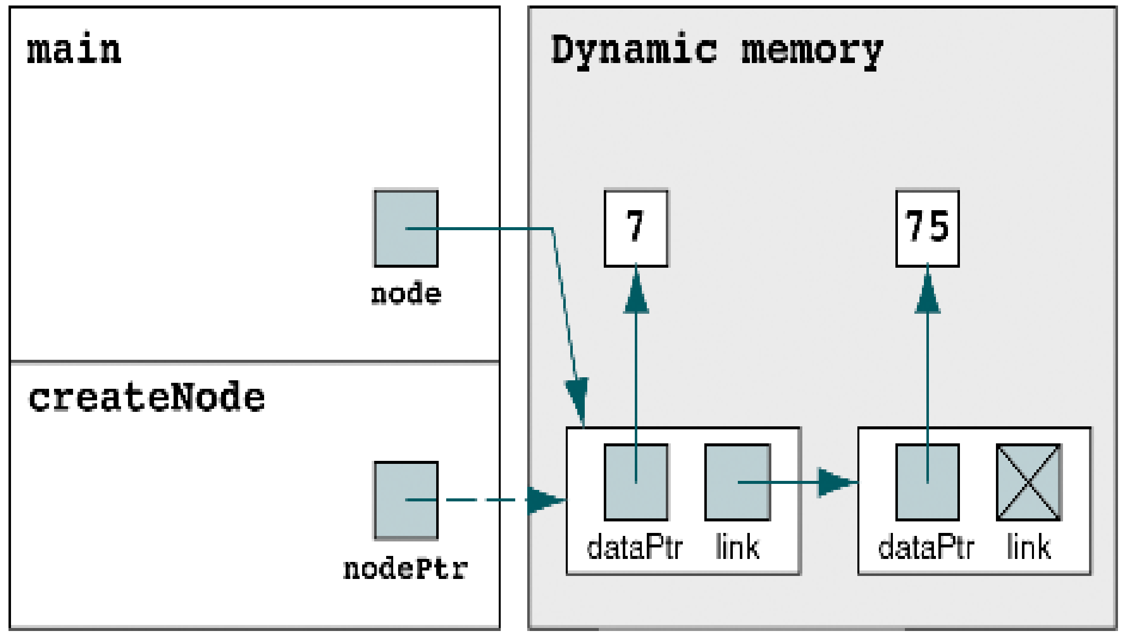

FIGURE 1-10 Structure for Two Linked Nodes

PROGRAM 1-4 Create List with Two Linked Nodes

```
/* Create a list with two linked nodes.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P1-02.h"                         // Header file

int main (void)
{
// Local Definitions
   int*   newData;
   int*   nodeData;
   NODE*  node;
```

PROGRAM 1-4 Create List with Two Linked Nodes (Continued)

```
// Statements
   newData  = (int*)malloc (sizeof (int));
   *newData = 7;
   node = createNode (newData);

   newData     = (int*)malloc (sizeof (int));
   *newData    = 75;
   node->link = createNode (newData);

   nodeData = (int*)node->dataPtr;
   printf ("Data from node 1: %d\n", *nodeData);

   nodeData = (int*)node->link->dataPtr;
   printf ("Data from node 2: %d\n", *nodeData);
   return 0;
}  // main
```

```
Results:
Data from node 1: 7
Data from node 2: 75
```

f1: Pointer to a function with no parameters; it returns *void*.

```
…
// Local Definitions
void   (*f1) (void);
int    (*f2) (int, int);
double (*f3) (float);
…
// Statements
…
f1  =  fun;
f2  =  pun;
f3  =  sun;
…
```

FIGURE 1-12 Pointers to Functions

PROGRAM 1-5 Larger Compare Function

```
/* Generic function to determine the larger of two
   values referenced as void pointers.
      Pre  dataPtr1 and dataPtr2 are pointers to values
                of an unknown type.
           ptrToCmpFun is address of a function that
                knows the data types
      Post data compared and larger value returned
```

continued

PROGRAM 1-5 Larger Compare Function *(continued)*

```
*/
void* larger (void* dataPtr1,    void* dataPtr2,
                 int (*ptrToCmpFun)(void*, void*))
{
   if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
         return dataPtr1;
   else
         return dataPtr2;
}  // larger
```

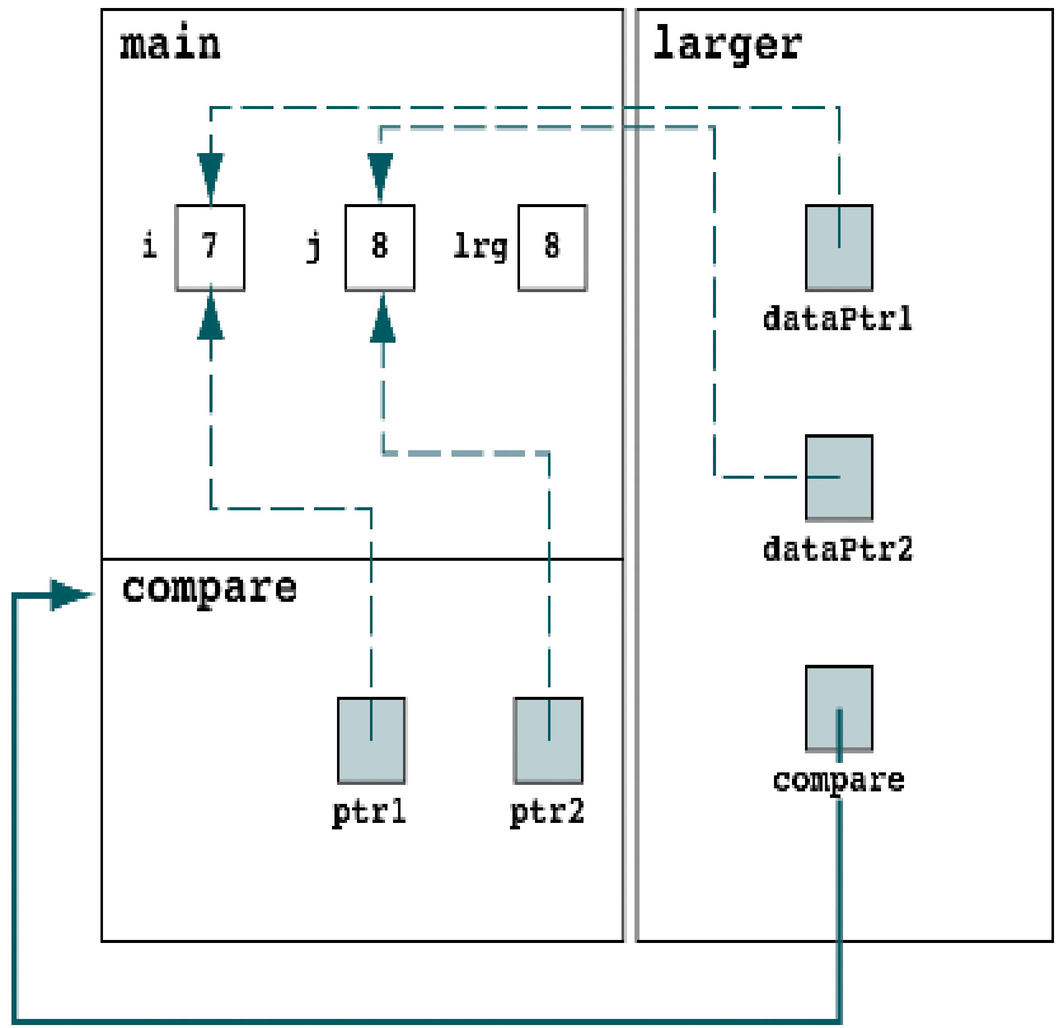

FIGURE 1-13 Design of Larger Function

PROGRAM 1-6 Compare Two Integers

```
/* Demonstrate generic compare functions and pointer to
   function.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P1-05.h"                               // Header file

int   compare (void* ptr1, void* ptr2);

int main (void)
{
// Local Definitions

   int i = 7 ;
   int j = 8 ;
   int lrg;

// Statements
   lrg = (*(int*) larger (&i, &j, compare));

   printf ("Larger value is: %d\n", lrg);
   return 0;
}  // main
/* ==================== compare ====================
   Integer specific compare function.
      Pre  ptr1 and ptr2 are pointers to integer values
      Post returns +1 if ptr1 >= ptr2
           returns -1 if ptr1 <  ptr2
*/
int compare (void* ptr1, void* ptr2)
```

continued

PROGRAM 1-6 Compare Two Integers *(continued)*

```
{
  if (*(int*)ptr1 >=  *(int*)ptr2)
     return 1;
  else
     return -1;
} // compare
```

```
Results:
Larger value is: 8
```

PROGRAM 1-7 Compare Two Floating-Point Values

```c
/* Demonstrate generic compare functions and pointer to
   function.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P1-05.h"                          // Header file

int   compare (void* ptr1, void* ptr2);

int main (void)
{
// Local Definitions

   float f1 = 73.4;
   float f2 = 81.7;
   float lrg;

// Statements
   lrg = (*(float*) larger (&f1, &f2, compare));

   printf ("Larger value is: %5.1f\n", lrg);
   return 0;
}  // main
/* ==================== compare ====================
   Float specific compare function.
      Pre  ptr1 and ptr2 are pointers to integer values
      Post returns +1 if ptr1 >= ptr2
```

continued

PROGRAM 1-7 Compare Two Floating-Point Values *(continued)*

```
                returns -1 if ptr1 <  ptr2
*/
int compare (void* ptr1, void* ptr2)
{
  if (*(float*)ptr1 >=  *(float*)ptr2)
     return 1;
  else
     return -1;
}  // compare
```

```
Results:
Larger value is: 81.7
```

1-6 Algorithm Efficiency

To design and implement algorithms, programmers must have a basic understanding of what constitutes good, efficient algorithms. In this section we discuss and develop several principles that are used to analyze algorithms.

- Linear Loops
- Logarithmic Loops
- Nested Loops
- Big-O Notation
- Standard Measurement of Efficiency

Multiply		Divide	
Iteration	Value of `i`	Iteration	Value of `i`
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

TABLE 1-3 Analysis of Multiply and Divide Loops

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

TABLE 1-4 Measures of Efficiency for $n = 10{,}000$

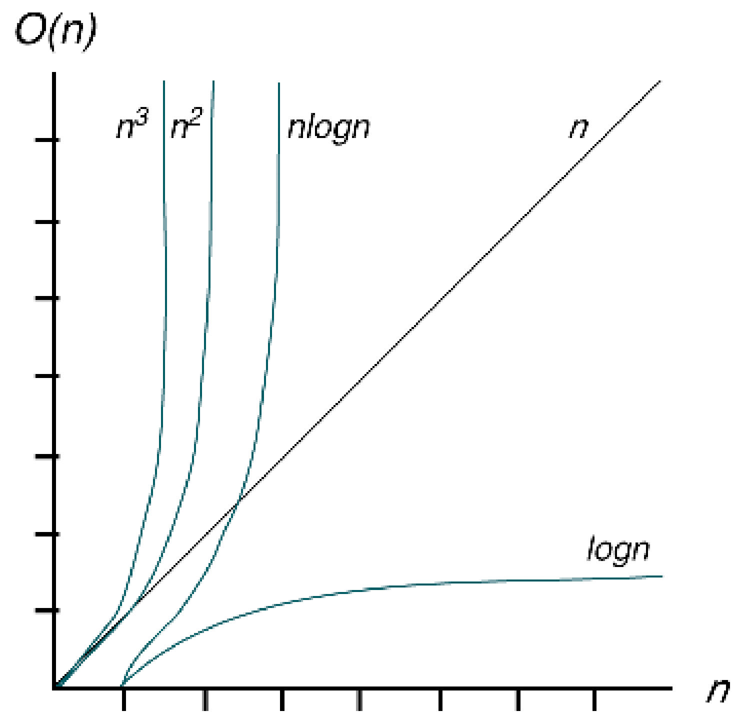

FIGURE 1-14 Plot of Effeciency Measures

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & -3 & 4 \\ 5 & 6 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 1 & 7 \\ 3 & 2 & -1 \\ 4 & 6 & 2 \end{bmatrix} = \begin{bmatrix} 10 & 3 & 8 \\ 3 & -1 & 3 \\ 9 & 12 & 4 \end{bmatrix}$$

FIGURE 1-15 Add Matrices

ALGORITHM 1-3 Add Two Matrices

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns or rows in matrix
   Post matrices added--result in matrix3
1 loop (not end of row)
   1  loop (not end of column)
      1  add matrix1 and matrix2 cells
      2  store sum in matrix3
   2  end loop
2 end loop
end addMatrix
```

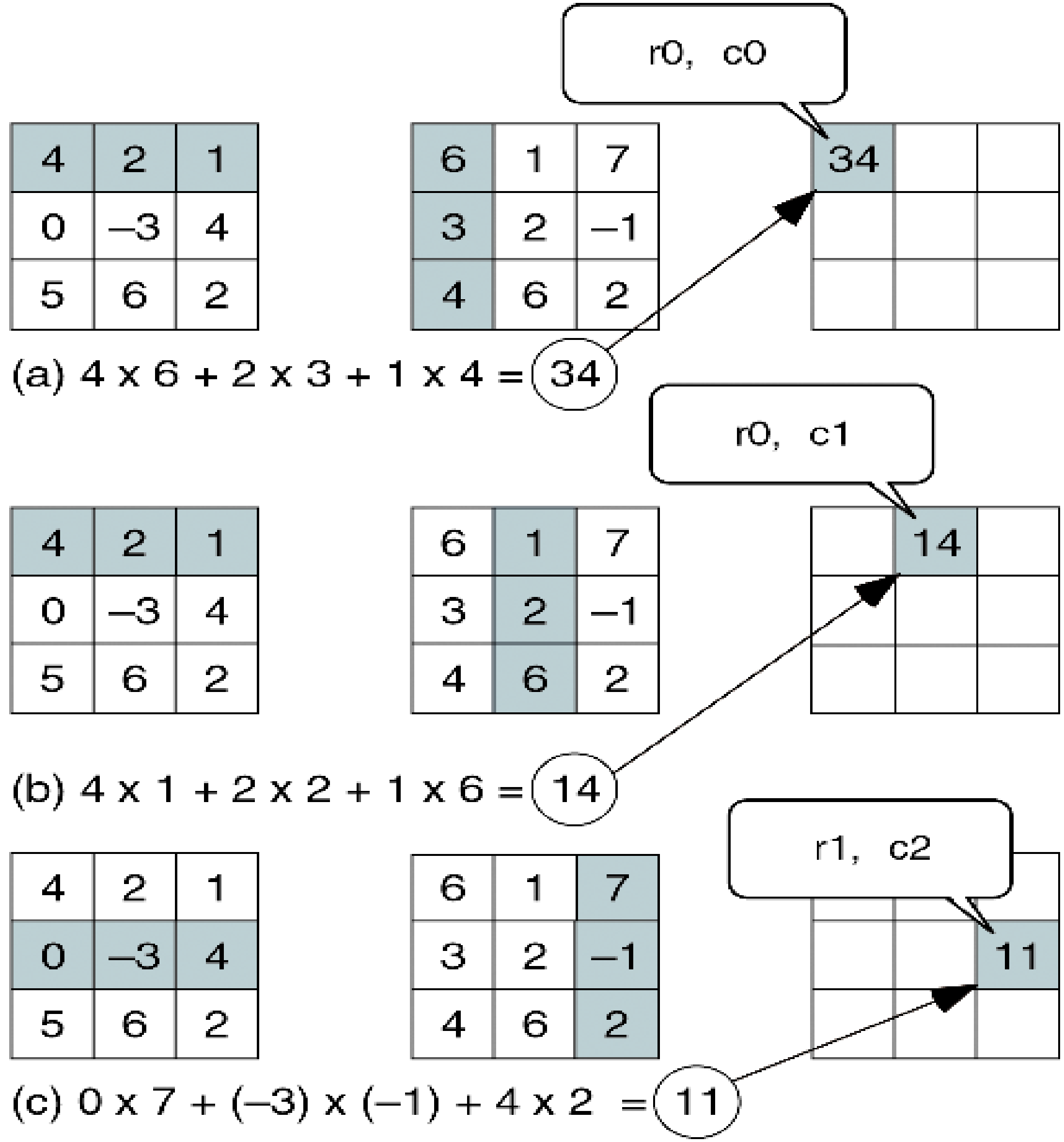

FIGURE 1-16 Multiply Matrices

ALGORITHM 1-4 Multiply Two Matrices

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns and rows in matrix
   Post matrices multiplied--result in matrix3
1 loop (not end of row)
   1 loop (not end of column)
      1 loop (size of row times)
         1 calculate sum of
              (all row cells) * (all column cells)
         2 store sum in matrix3
```

continued

ALGORITHM 1-4 Multiply Two Matrices *(continued)*

```
   2  end loop
2  end loop
3  return
end multiMatrix
```