Chapter 1

Basic Concepts

Objectives

Upon completion you will be able to:

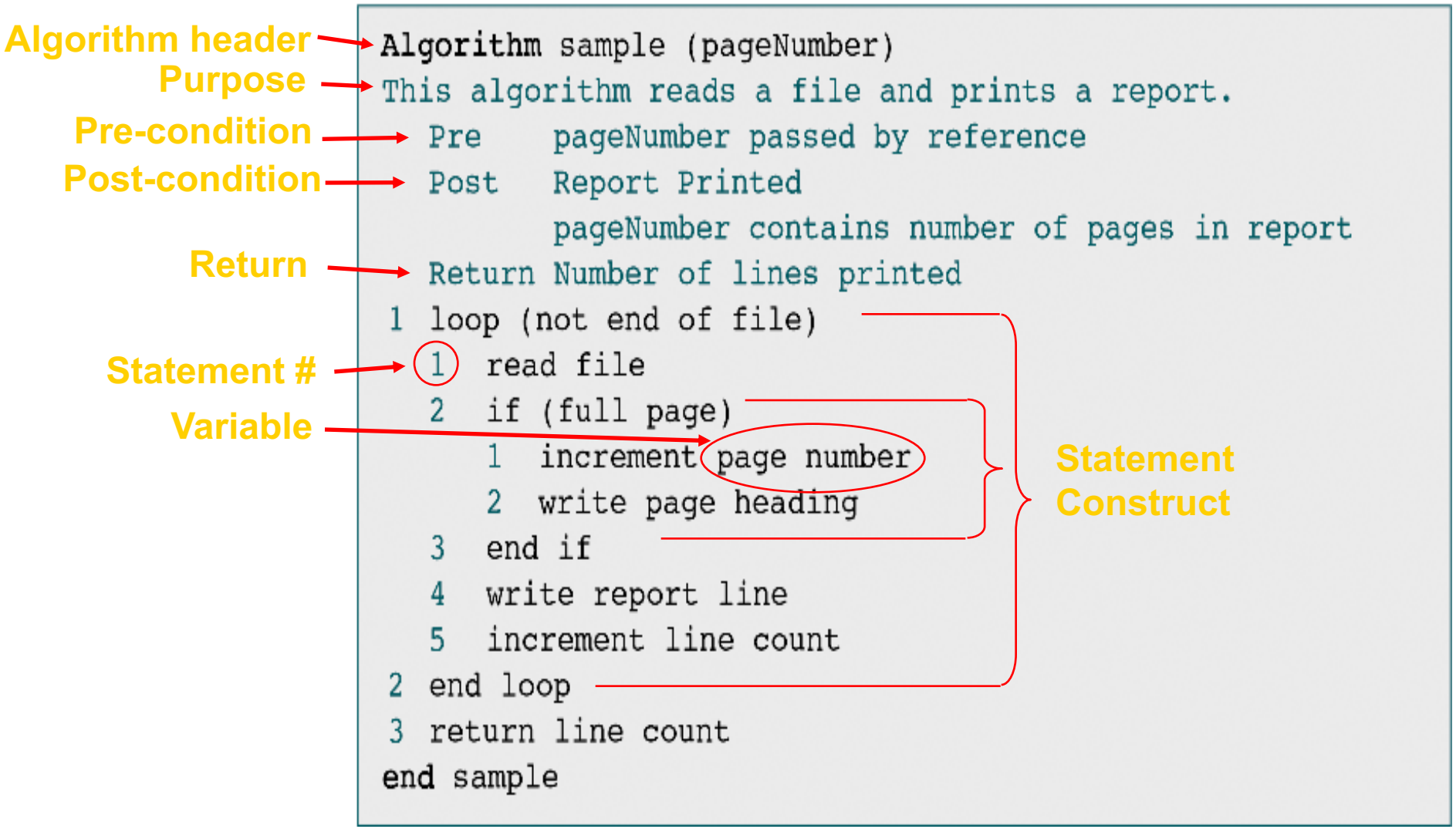
- **Use pseudocode in the development of algorithms**
- **Understand the need for Abstract Data Type (ADT)**
- **Understand the implementation of ADTs**
- **Use *void* pointers and pointer to functions**
- **Understand the role of Big-O notation**

1-1 Pseudocode

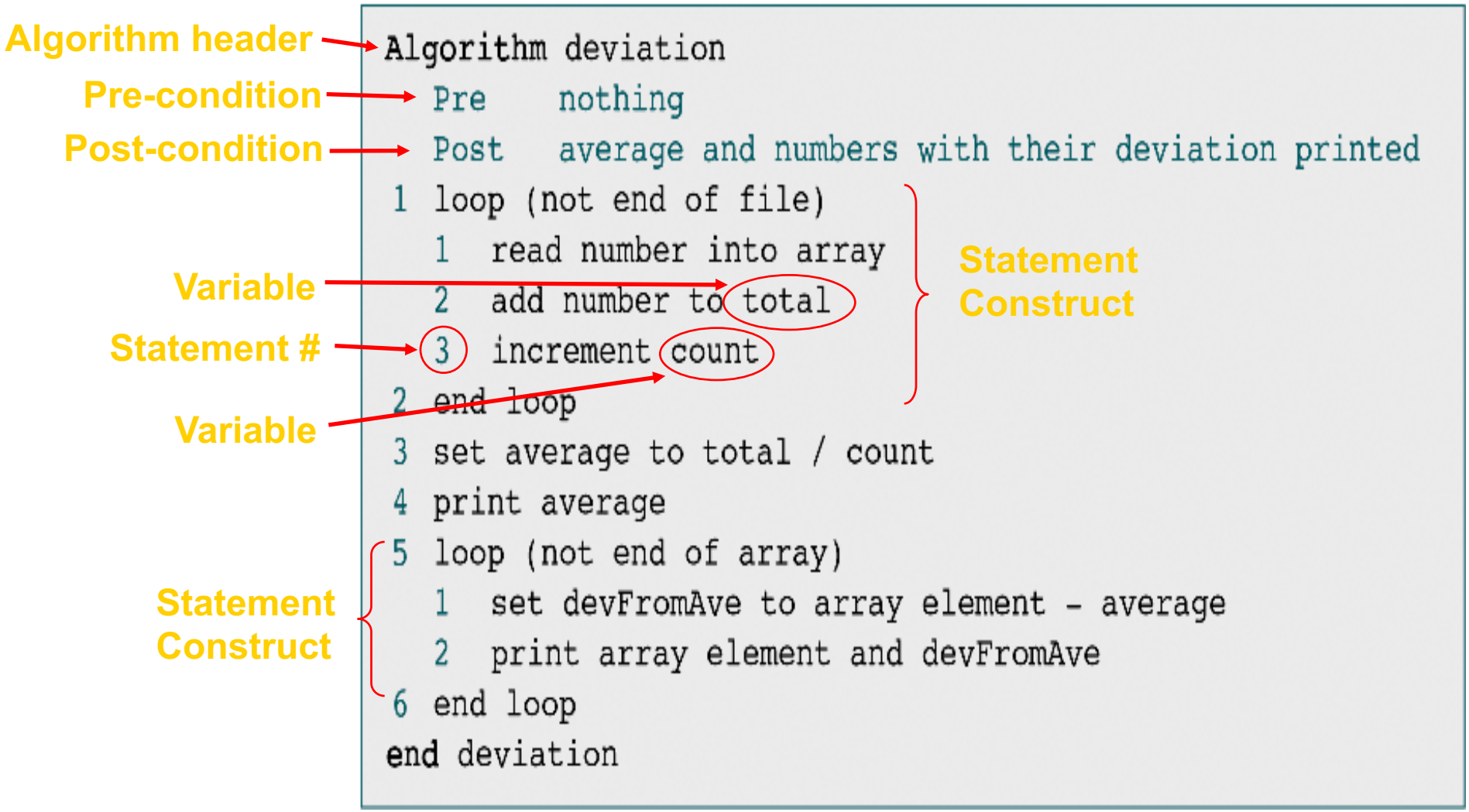
Pseudocode is an English-like representation of the algorithm logic. It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.

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

ALGORITHM 1-1 Example of Pseudocode



ALGORITHM 1-2 Print Deviation from Mean for Series



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

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

Atomic Data and Composite Data

- Atomic data: data that consist of a single piece of information
  - E.g., integer value, floating point value
- Composite Data : data that can be broken out into subfields that have meaning
  - E.g., telephone number (area code + exchange code + number within the exchange)

Data Type

- A data type consists of two parts
  - A set of values
  - A set of operations on values

For examples:

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

Data Structure

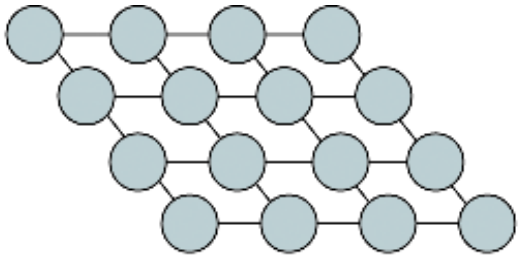
- A combination of elements in which each is either a data type or another data structure
- A set of associations or relationships (structure) involving the combined elements

For examples:

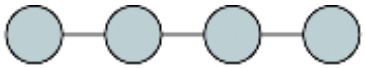
Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

Abstract Data Type (ADT)

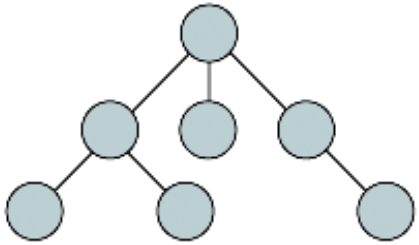
- A ADT consist of a set of definitions that allow programmers to use the functions while *hiding the implementation*
- With a ADT users are not concerned with *how* the task is done but rather with *what* it can do
- Example 1: Consider the concept of a list, there are at least 4 data structures can support a list



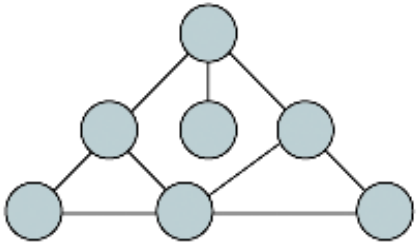
(a) Matrix



(b) Linear list



(c) Tree



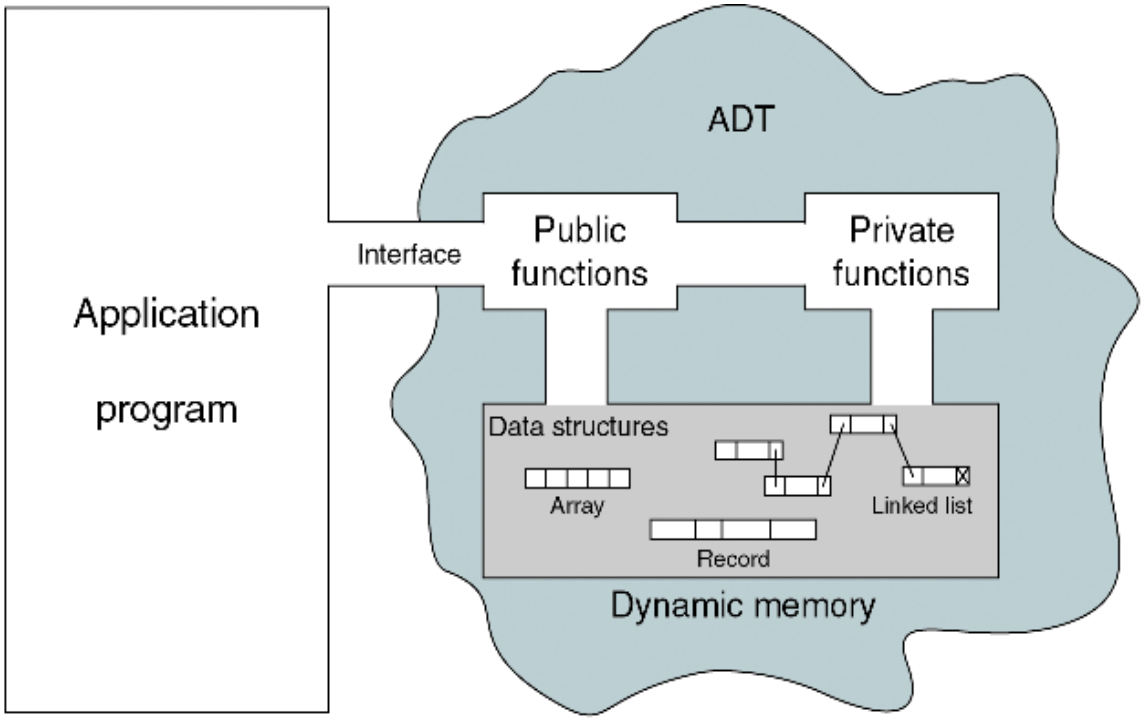
(d) Graph

Abstract Data Type (conti.)

- If we place a list in an ADT, user should not be aware of the structure we use while they can insert and retrieve data
- Example 2: consider the system analyst who need to simulate the waiting line of a bank
  - A simulation of queue is required
  - Basic queue operations (enqueue, dequeue) are required
- Two potential solutions to the above problem
  - Write a program that simulate the queue analyst needs
  - Write a queue ADT that can solve *any* queue problem
- A typical ADT should
  - Declaration of data
  - Declaration of operations
  - Encapsulation of data and operation on the data
  - Hide them from the user

1-3 Model for an Abstract Data Type

- ADT data structure: ADT for stacks, queues, lists, tree…….
- ADT operations
  - Data are entered , accessed, modified through the external interface
  - Only the public functions are accessible through the interface
  - For each ADT operation there is an algorithm that perform it's task
  - Only the operation name and parameters are available to the app.



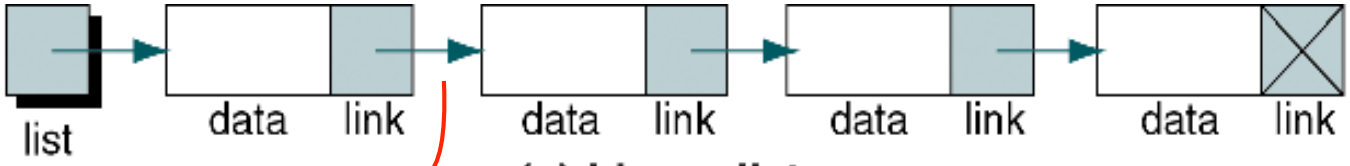
1-4 ADT Implementations

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.

主要是這個

- Array Implementation
- Linked List Implemenation

Linked List Implementation for List

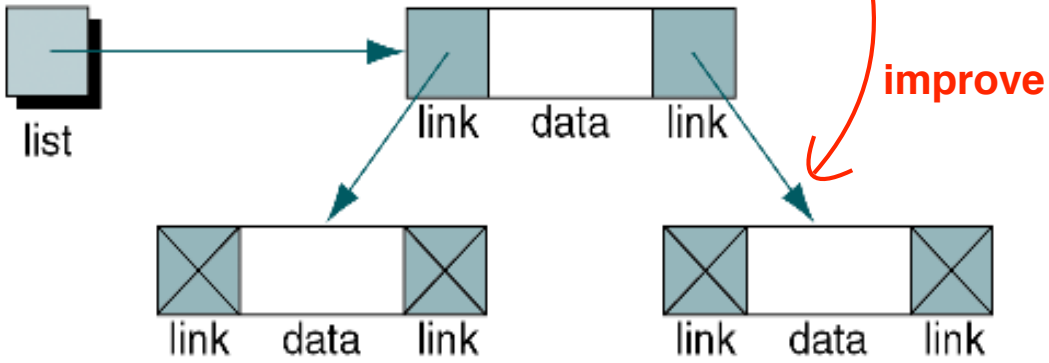


pointer

(a) Linear list

搜尋很花時間

improve

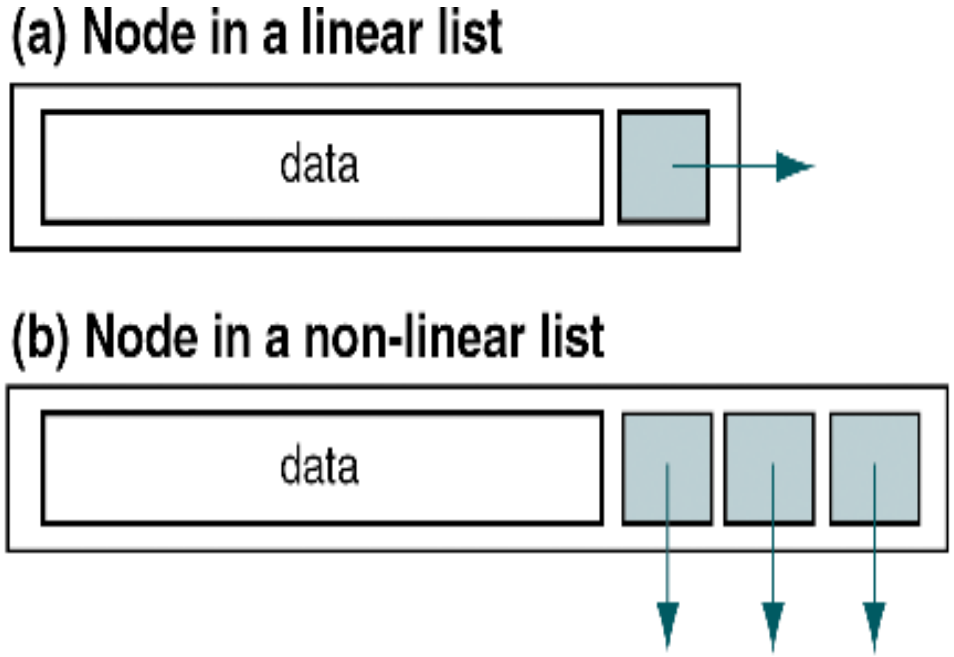


(b) Non-linear list

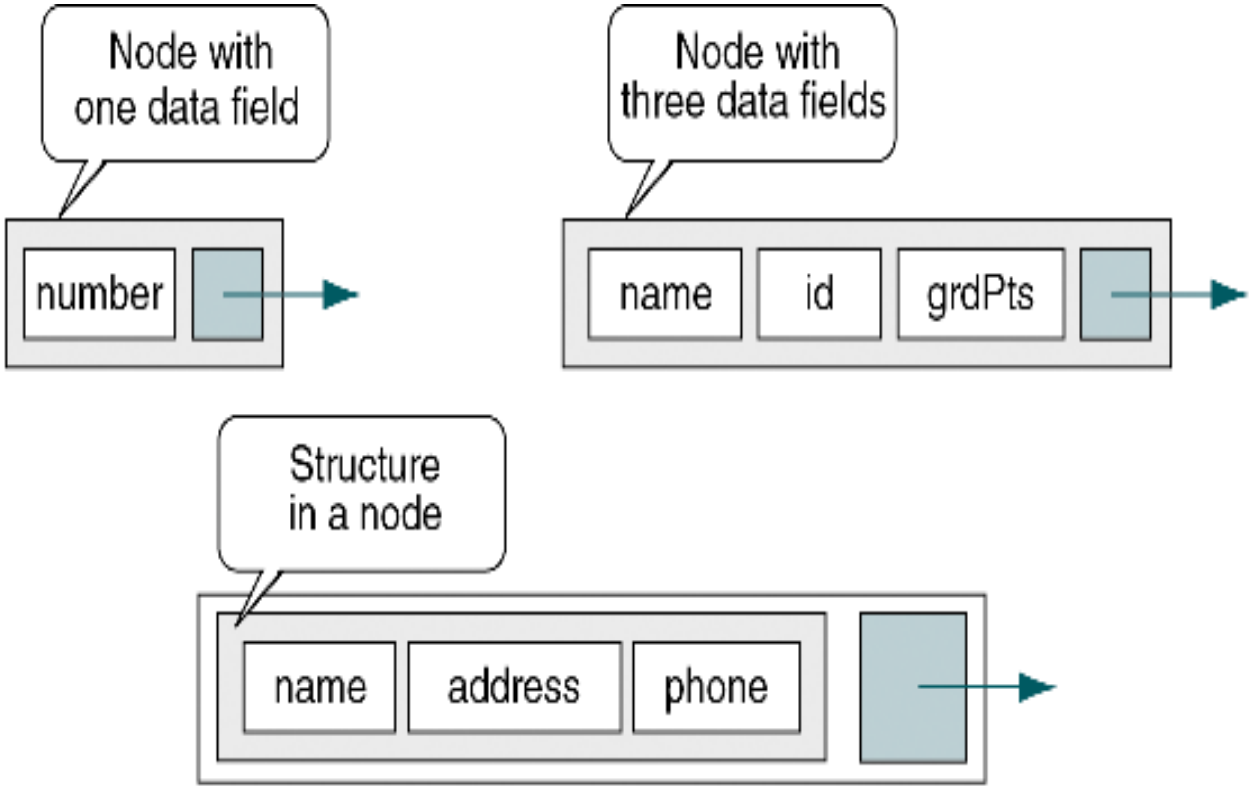
Array的缺點：
1.很難從中間插入資料
2.大小需要事先宣告

(c) Empty list

Linked List – Nodes



Linked List – Nodes



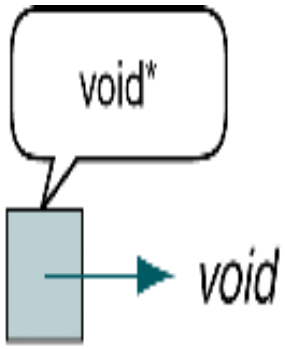
1-5 Generic Code for ADT

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

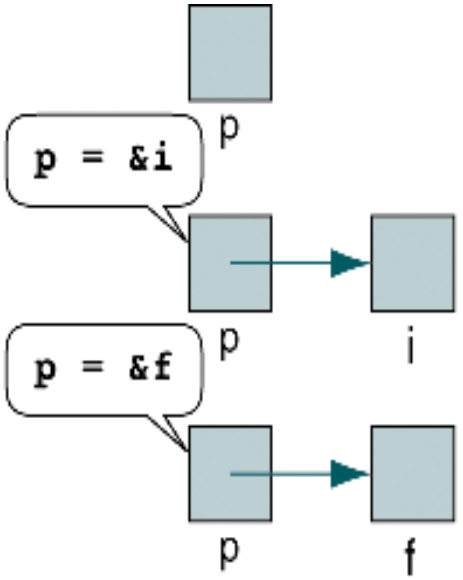
Pointer to "*void*"

- A pointer to *void* is a generic pointer that can be used to represent any data type during compilation or run time
- A pointer to *void* is not a null pointer; it is pointing to a generic data type




```
void* p;
int   i;
float f;

p = &i;
...
p = &f;
```



PROGRAM 1-1 Demonstrate Pointer to *void*

```
 1  /* Demonstrate pointer to void.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main ()
 8  {
 9  // Local Definitions
10     void* p;
11     int   i = 7 ;
12     float f = 23.5;
13
14  // Statements
15     p = &i;
16     printf ("i contains: %d\n", *((int*)p) );
17
18     p = &f;
19     printf ("f contains: %f\n", *((float*)p));
20
21     return 0;
22  }  // main
```

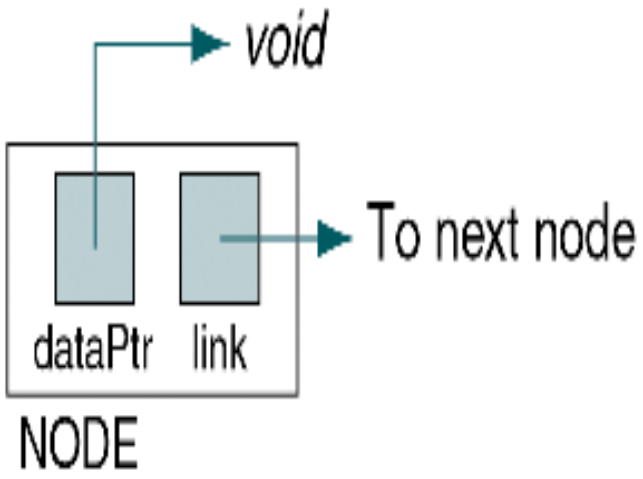
A pointer to *void* cannot be dereferenced

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

Pointer to Node

- The node structure has two fields
  - Link field is a pointer to the node structure (self-reference)
  - Data field can be ***any*** type: integer, floating point, string, or even another structure

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

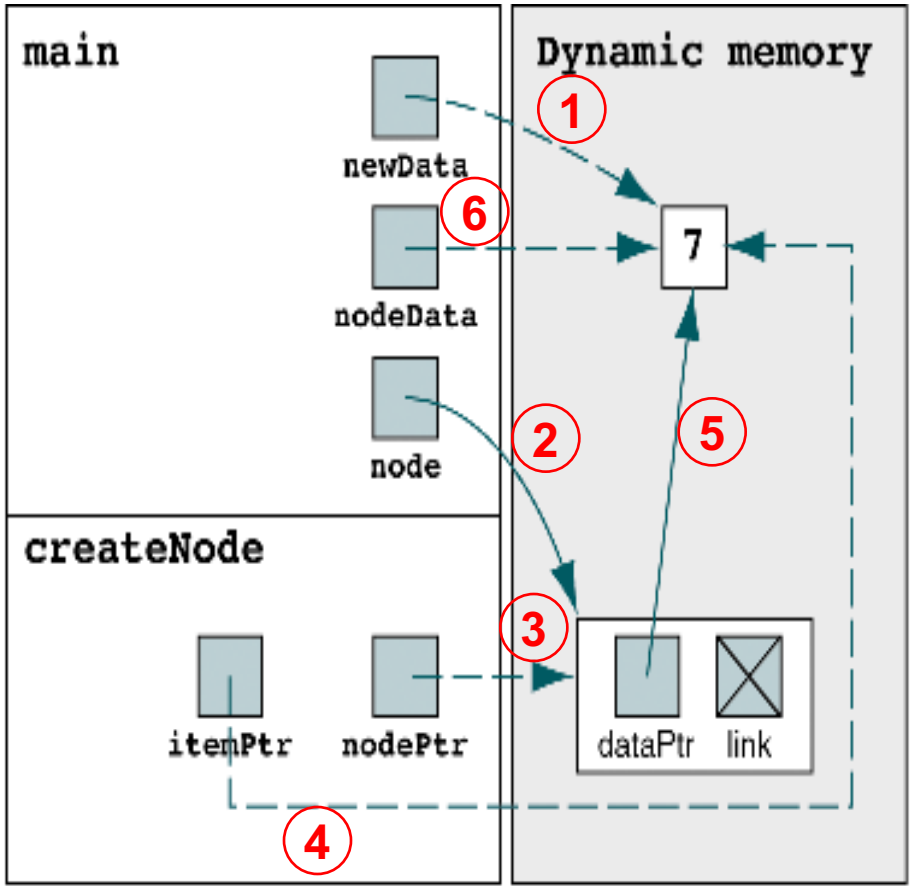


Example 1

- **Goal: Write a program that calls a function that accepts a pointer to data of any type and create a node that stores the data pointer and a link pointer**

User Application

Generic ADT Code



Create Node Function

```
5  typedef struct node
6  {
7          void* dataPtr;
8    struct node* link;
9  } NODE;

17 NODE* createNode (void* itemPtr)
18 {
19    NODE* nodePtr;
20    nodePtr = (NODE*) malloc (sizeof (NODE));
21    nodePtr->dataPtr = itemPtr;
22    nodePtr->link    = NULL;
23    return nodePtr;
24 }  // createNode
```

Call-by-reference

Dynamic memory allocation

Main Program

```
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include "P1-02.h"                          // Header file
 8
 9  int main (void)
10  {
11  // Local Definitions
12     int*  newData;
13     int*  nodeData;
14     NODE* node;
15
16  // Statements
17     newData  = (int*)malloc (sizeof (int));
18     *newData = 7;
```

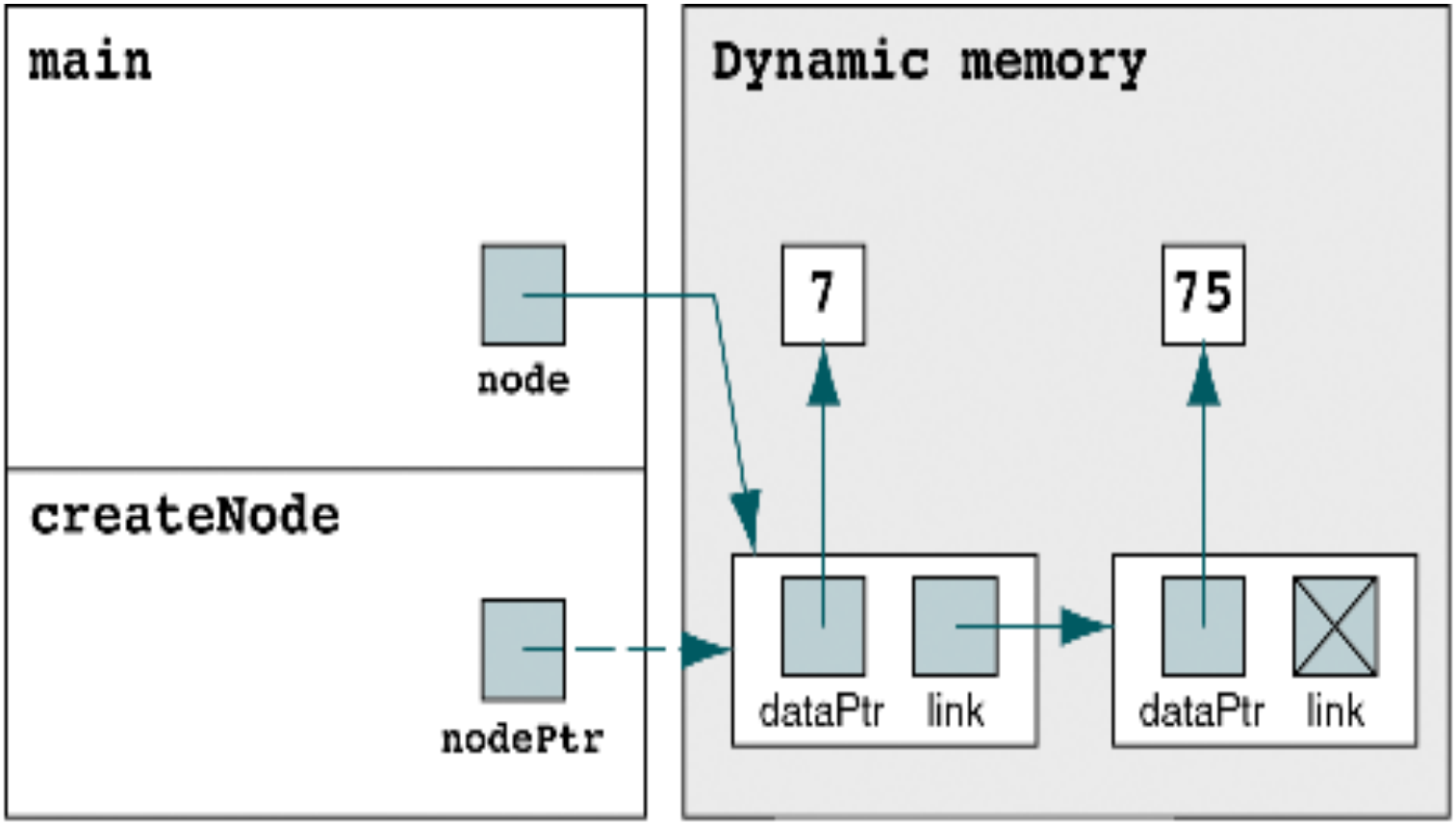
continued

Main Program (conti.)

```
19
20      node = createNode (newData);
21                Be careful !
22      nodeData = (int*)node->dataPtr;
23      printf ("Data from node: %d\n", *nodeData);
24      return 0;
25  }  // main
```

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

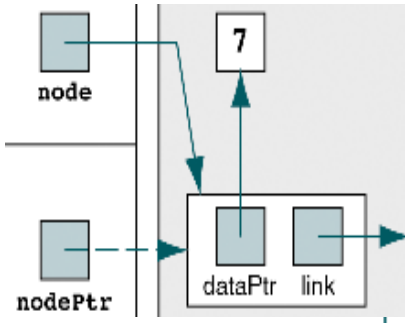
Example 2 – Two Linked Nodes

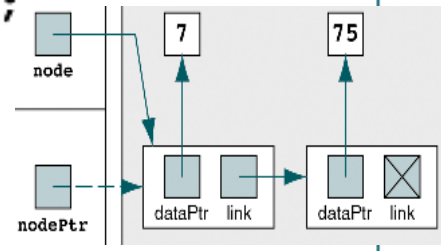


PROGRAM 1-4 Create List with Two Linked Nodes

```
1  /* Create a list with two linked nodes.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "P1-02.h"                          // Header file
8
9  int main (void)
10 {
11 // Local Definitions
12    int*  newData;
13    int*  nodeData;
14    NODE* node;
15
```

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



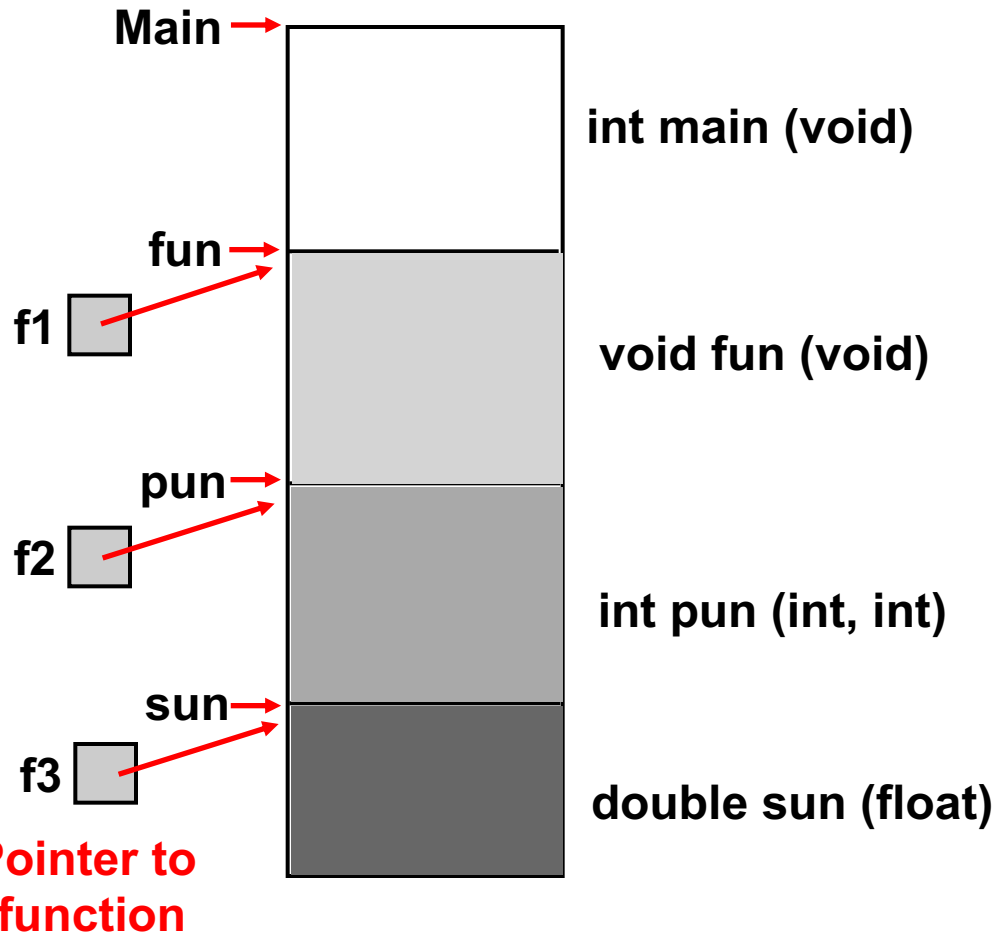



```
16  // Statements
17     newData  = (int*)malloc (sizeof (int));
18     *newData = 7;
19     node = createNode (newData);
20
21     newData     = (int*)malloc (sizeof (int));
22     *newData    = 75;
23     node->link = createNode (newData);
24
25     nodeData = (int*)node->dataPtr;
26     printf ("Data from node 1: %d\n", *nodeData);
27
28     nodeData = (int*)node->link->dataPtr;
29     printf ("Data from node 2: %d\n", *nodeData);
30     return 0;
31  }  // main
```

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

Pointer to Function

Note: Function name is a pointer to its code in memory

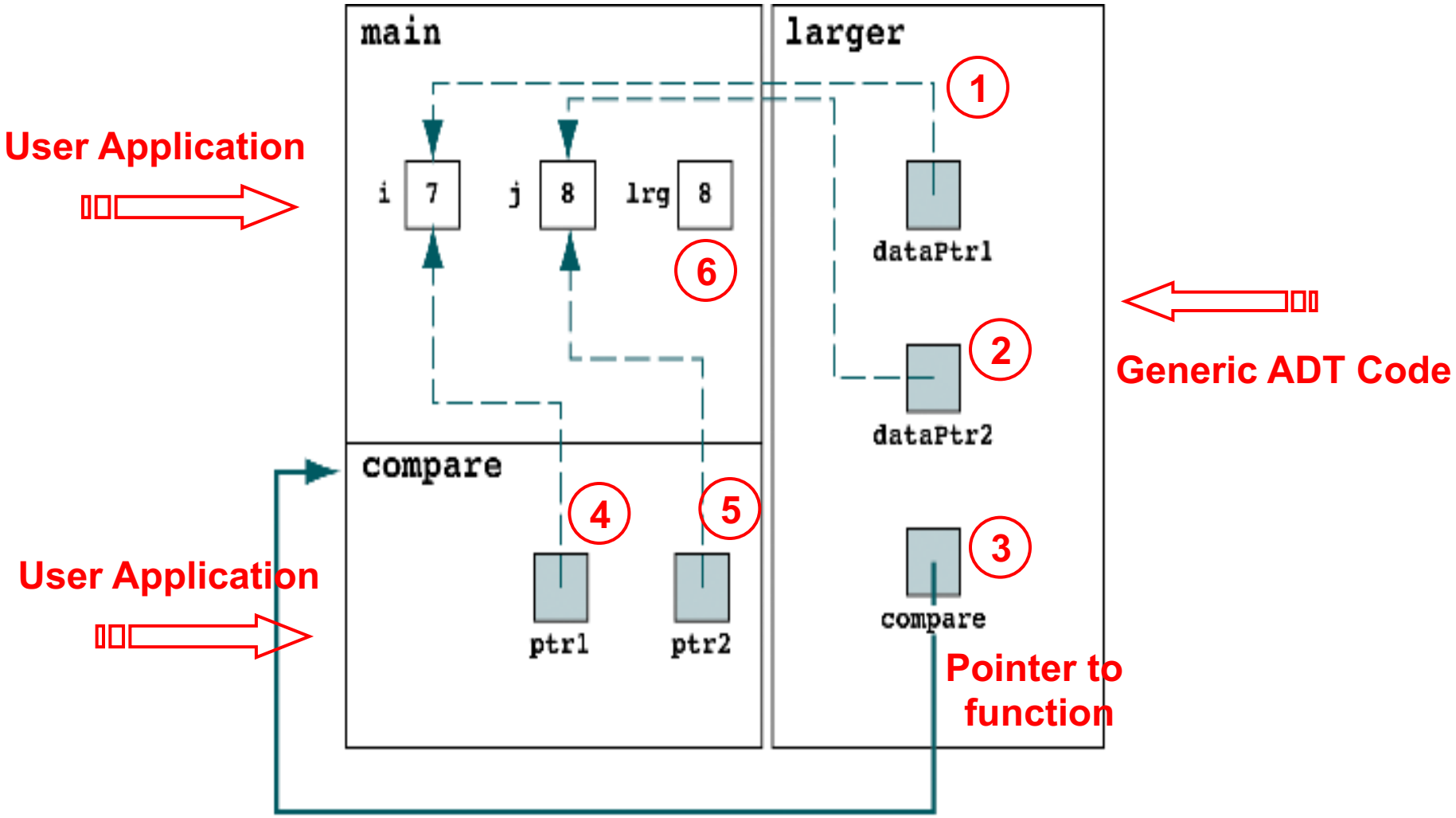


f1: Pointer to a function with no parameters; it returns *void*.

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

Example of Pointer to Function

- Goal: Compare two number (integer or floating point or character)



PROGRAM 1-6 Compare Two Integers

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                          // Header file
 9
10  int    compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     int i = 7 ;
17     int j = 8 ;
18     int lrg;
19
20  // Statements
21     lrg = (*(int*) larger (&i, &j, compare));
22
23     printf ("Larger value is: %d\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Integer specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
30             returns -1 if ptr1 <  ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
```

Notice !

continued

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

```
33  {
34    if (*(int*)ptr1 >=  *(int*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

Notice !

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

Larger Compare Function

```
 8  */
 9  void* larger (void* dataPtr1,    void* dataPtr2,
10                int (*ptrToCmpFun)(void*, void*))
11  {
12     if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
13           return dataPtr1;
14     else
15           return dataPtr2;
16  }  // larger
```

PROGRAM 1-7 Compare Two Floating-Point Values

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                          // Header file
 9
10  int   compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     float f1 = 73.4;
17     float f2 = 81.7;
18     float lrg;
19
20  // Statements
21     lrg = (*(float*) larger (&f1, &f2, compare));
22
23     printf ("Larger value is: %5.1f\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Float specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
```

continued

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

```
30                returns -1 if ptr1 <  ptr2
31   */
32   int compare (void* ptr1, void* ptr2)
33   {
34     if (*(float*)ptr1 >=  *(float*)ptr2)
35        return 1;
36     else
37        return -1;
38   }  // 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

Linear Loops

Example 1:

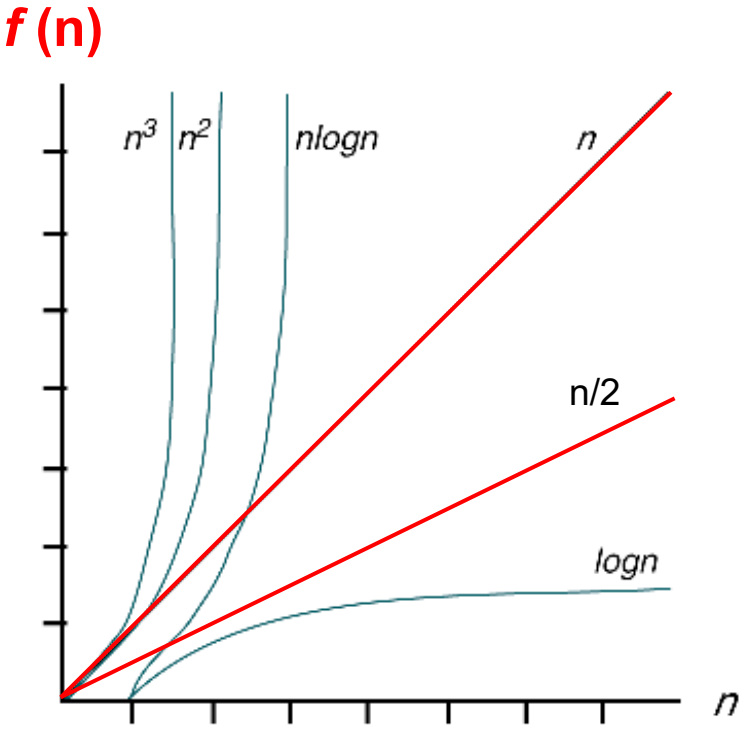
```
for (i=0; i < 1000; i++)
    Application code
```

If *f* (n) = efficiency, then *f* (n) = n

Example 2:

```
for (i=0; i < 1000; i += 2)
    Application code
```

***f* (n) = n/2**



Logarithmic Loops

for (i=1; i < 1000; i *= 2)

Application code

for (i=1000; i > 0; i /= 2)

Application code

$f(n) = log_2 n$

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

Linear Logarithmic & Square Loops

```
for (i=0; i<10; i++)
  for(j=0; j<10; j*=2)
    Application code
```

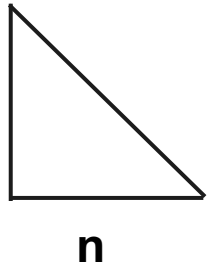
f **(n) = 10*log*$_2$10 => n*log*n**

```
for (i=0; i<10; i++)
  for(j=0; j<10; j++)
    Application code
```

$$f(n) = n^2$$

```
for (i=0; i<10; i++)
  for(j=0; j<i; j++)
    Application code
```

$$f(n) = n\left(\frac{n+1}{2}\right)$$



Big-O Notation

- Big-O of n (O(n)) = “on the order of n “
- The Big-O can be derived from *f* (n) using the following steps
  - Set the coefficient of each term to 1
  - Keep the largest term in the function and discard the others
  - Terms are ranked from lowest to highest as shown below

$$\log n \le n \le n\log n \le n^2 \le n^3 \ldots \le n^k \le 2^n \le n!$$

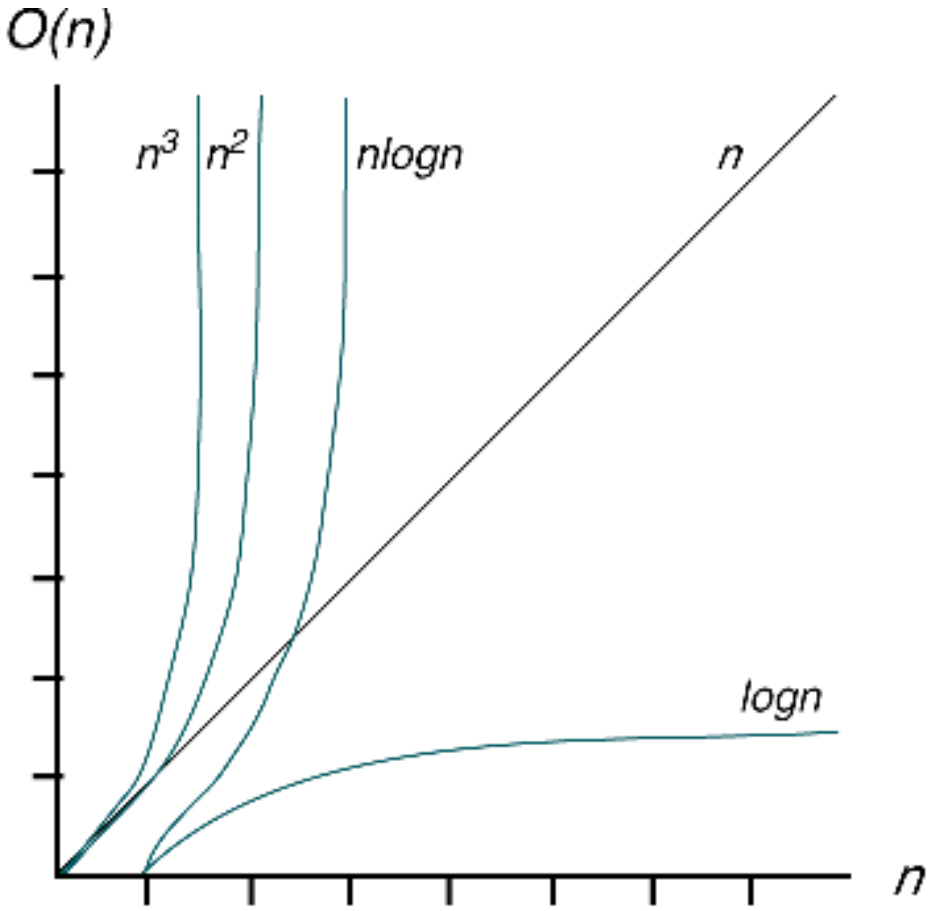
- For example: calculate the big-O for $f(n) = n\left(\frac{n+1}{2}\right) = \frac{1}{2}n^2 + \frac{1}{2}n$
  - Remove all coefficients => $n^2 + n$
  - Remove the smaller factors => n^2

$$\Rightarrow \quad O(f(n)) = O(n^2)$$

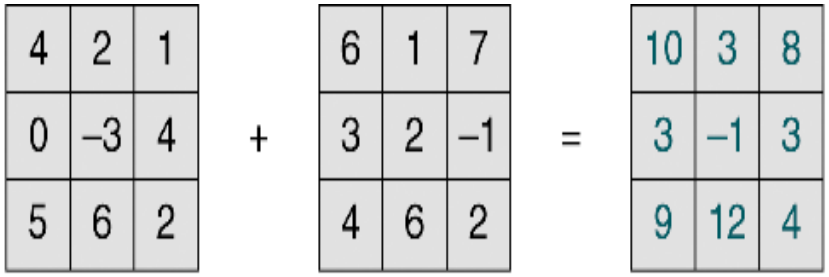
Measures of Efficiency for n=10,000

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

Plot of Efficiency Measures



Big-O Analysis Example-Add Matrix



4	2	1
0	−3	4
5	6	2

+

6	1	7
3	2	−1
4	6	2

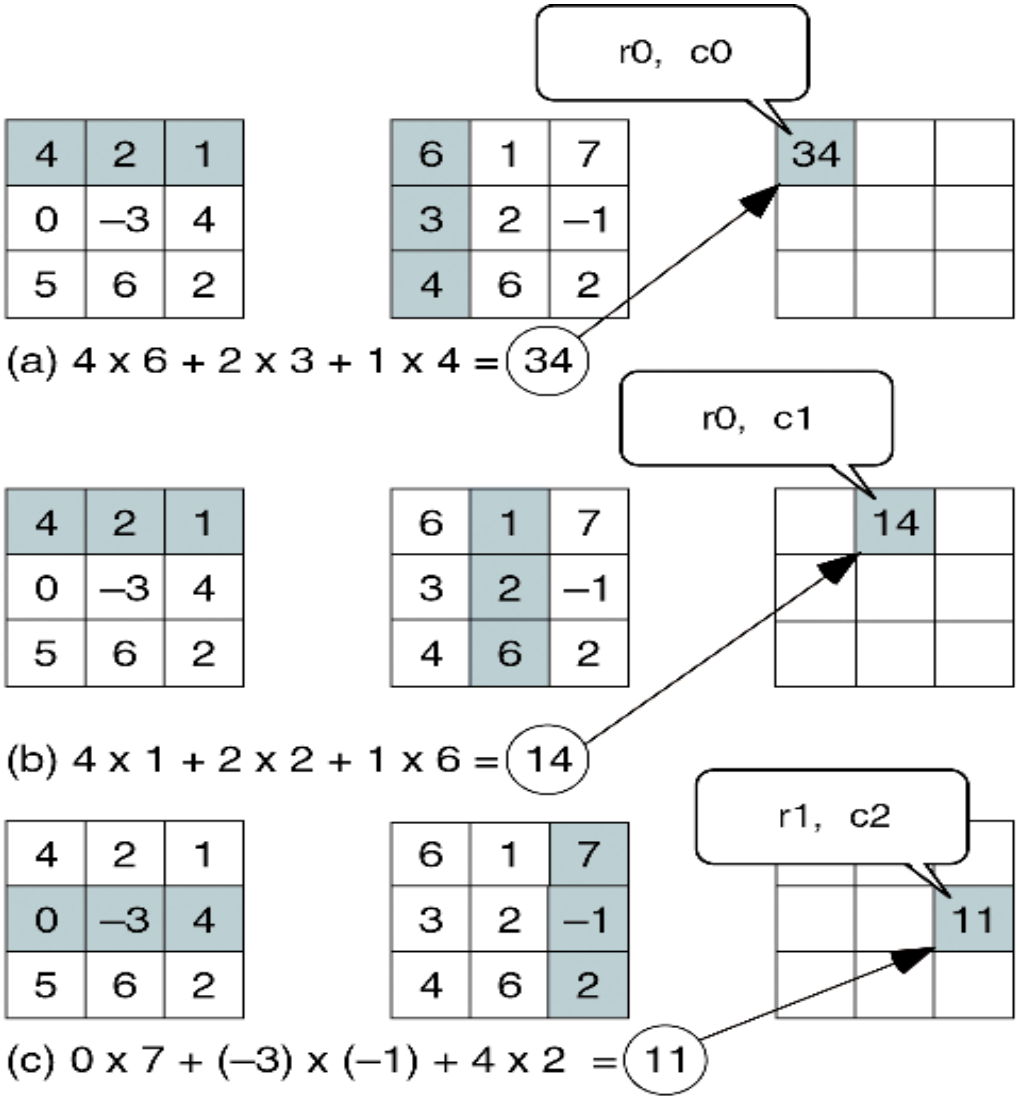
=

10	3	8
3	−1	3
9	12	4

$$O(\text{size}^2) = O(n^2)$$

```
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
```

Big-O Analysis Example - Multiply Matrix



Big-O Analysis - Multiply Matrix (conti.)

$$O(\text{size}^3) = O(n^3)$$

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