Fundamentals of Abstract Data Types (ADTs)

UNIT 2

CPE 1202L: Data Structures and Algorithms

COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

Unit 2: Fundamentals of Abstract Data Type (ADTs)

- Pseudocode
- The Abstract Data Type
- Model for an ADT
- ADT Implementation
  - Array
  - Linked List
- Algorithm Efficiency

Algorithm

- An **algorithm** is a finite set of well-defined instructions for accomplishing some task which, given an initial state, will terminate in a defined end-state.
- To develop any algorithm, it is necessary to know, how the corresponding task can be solved.

Types of Algorithms

- Linear algorithm consists of a sequence of unconditional straightforward steps.
- Loop is a group of steps that are repeated until some condition will not be satisfied.
- Nested Loop is a loop containing another loop (loops).
- Branching algorithm consists of a number of subsequences that can be taken depending on some condition (conditions).

Algorithms and Programming Languages

- To utilize any algorithm using a computer, we have to develop a program using a programming language.
- Low-level language (assembly language) is a language of machine instructions.
- High-level language is a language, which is closer to our natural language.
- A program is implementation of the algorithm in a form acceptable for a computer.

Data Structures

- The simplest data structures are: a simple variable and a constant.
- Other data structures are: arrays, records, lists, trees, stacks, queues, etc..
- Data structures are organized similarly in all the programming languages.
- The latter means that data structures can be studied independently of a particular programming language.
- Knowing data structures, it is easier to learn different programming languages.

Atomic Data

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

Data Structure

- A **Data Structure** is an aggregation of atomic and composite data into a set with defined relationships.
- Structure means a set of rules that holds the data together.
- Taking a combination of data and fit them into such a structure that we can define its relating rules, we create a **data structure**.

Composite Data Structures

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

Data Structure is:

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

Data Structures: Properties

- **Most** of the modern programming languages **support** a number of data structures.
- In addition, modern programming languages **allow** programmers **to create new** data structures for an application.
- Data structures can be nested. A data structure may contain other data structures (array of arrays, array of records, record of records, record of arrays, etc.)

Some Data Structures

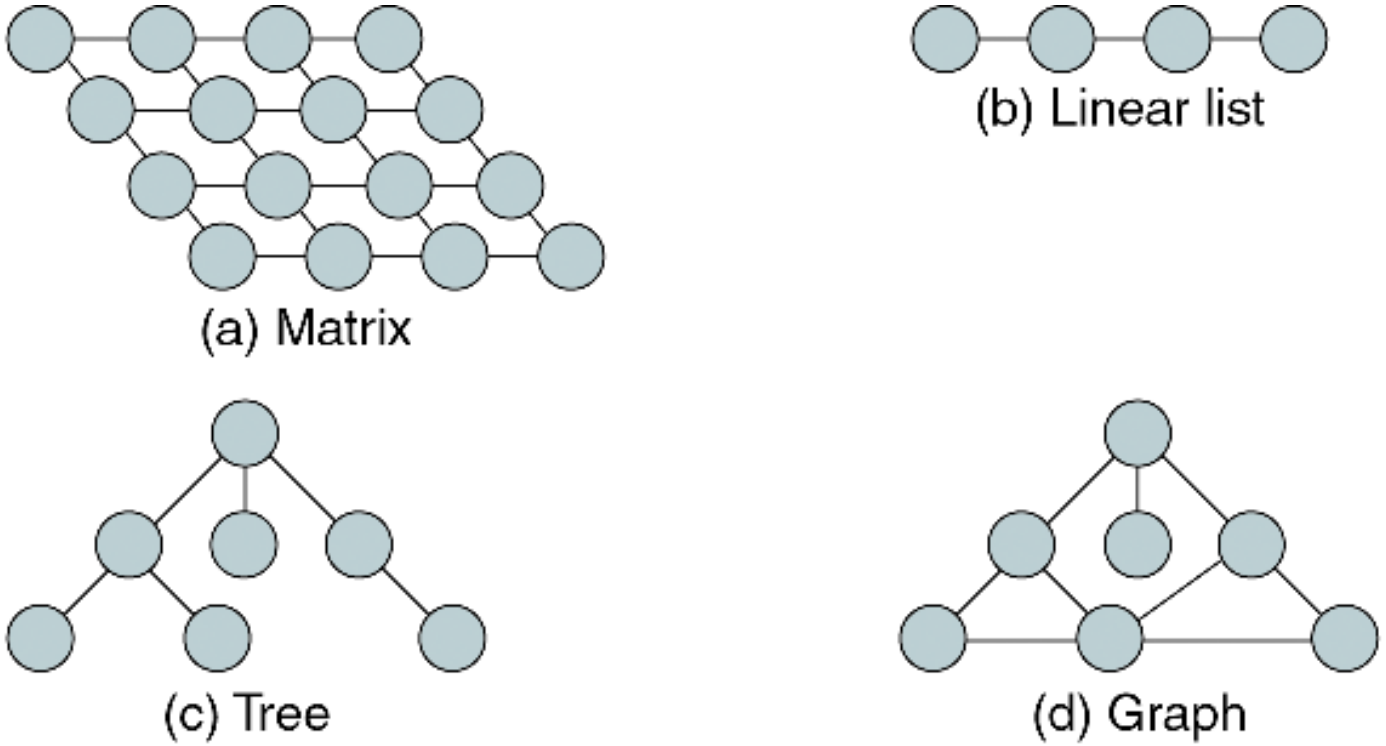


FIGURE 1-1 Some Data Structures

Pseudocode

Pseudocode

- Pseudocode is a pseudo programming language, which is commonly used to define algorithms
- Pseudocode is a natural language-like representation of the algorithm logic
- Its structure is close to the structure of the most high level programming languages, but it is free from many unnecessary details

Pseudocode: Example

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
```

The Abstract Data Type (ADT)

The Abstract Data Type (**ADT**)

The concept of abstraction means:

- We know what a data type can do
- How it is done is hidden for the user

➢With an ADT users are not concerned with how the task is done but rather with what it can do.

ADT: Example

- The program code to read/write some data is ADT. It has a data structure (character, array of characters, array of integers, array of floating-point numbers, etc.) and a set of operations that can be used to read/write that data structure.

The Abstract Data Type (**ADT**)

The Abstract Data Type (ADT) is:

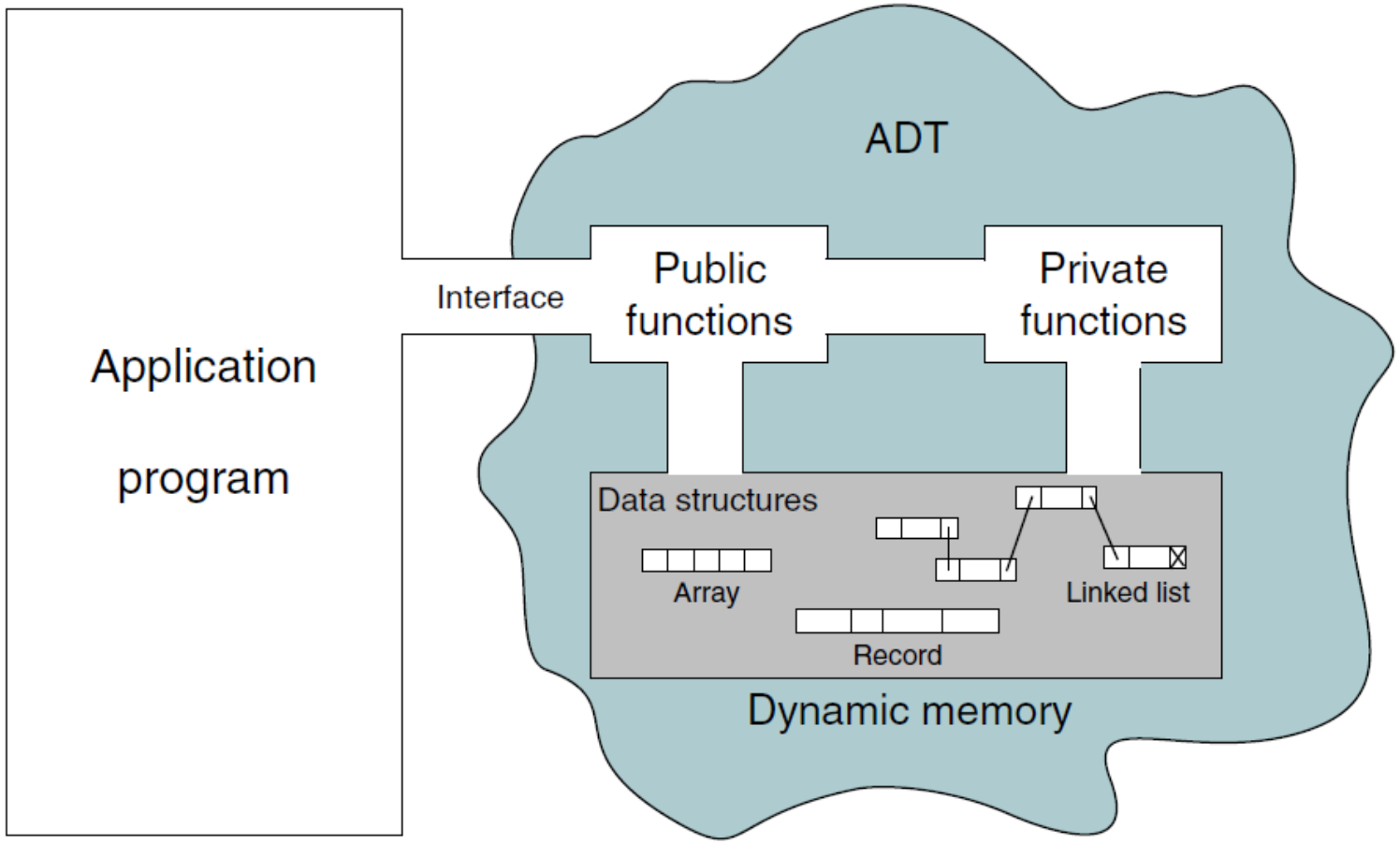
- A data declaration packaged together with the operations that are meaningful for the data type.
- In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

- ➢Declaration of data
- ➢Declaration of operations
- ➢Encapsulation of data and operations

The Abstract Data Type (**ADT**)

- All references to and manipulation of the data in a structure must be handled through defined interfaces to the structure.
- Allowing the application program to directly reference the data structure is a common fault in many implementations.
- It is necessary for multiply versions of the structure to be able to coexist.
- We must hide the implementation from the user while being able to store different data.

Abstract Data Type Model



ADT Operations

- Data are entered, accessed, modified and deleted through the external interface, which is a "passageway" located partially "in" and partially out of the ADT.
- Only the public functions are accessible through this interface.
- For each ADT operation there is an algorithm that performs its specific task.

Typical ADTs:

- Lists
- Stacks
- Queues
- Trees
- Heaps
- Graphs

ADT List Implementations

Array Implementations

- In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes).
- Although searching an array for an individual element can be very efficient, insertion and deletion of elements are complex and inefficient processes.

Linked Lists

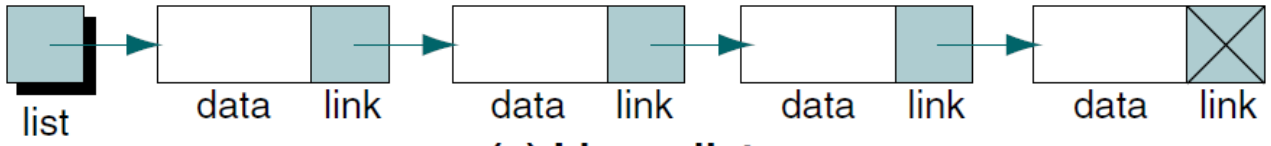
- A **Linked List** is an ordered collection of data in which each element contains the location of the next element or elements.
- In a linked list, each element contains two parts: data and one or more links.
- The data part holds the application data – the data to be processed.
- Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

Linear and non-linear Linked Lists

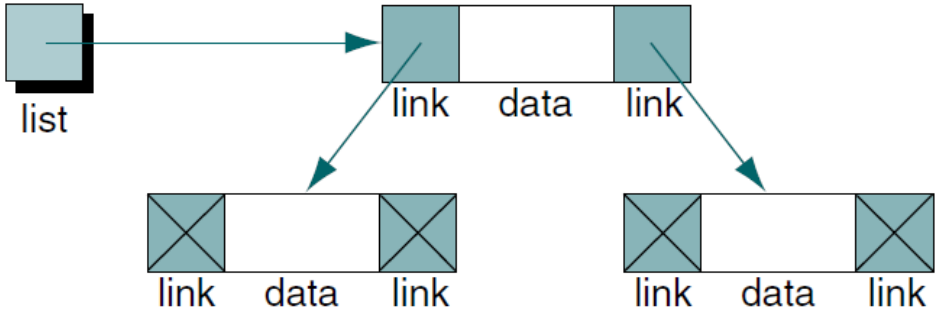
- In linear linked lists, each element has only zero or one successor.

- In non-linear linked lists, each element can have zero, one or more successors.

Linked Lists



(a) Linear list

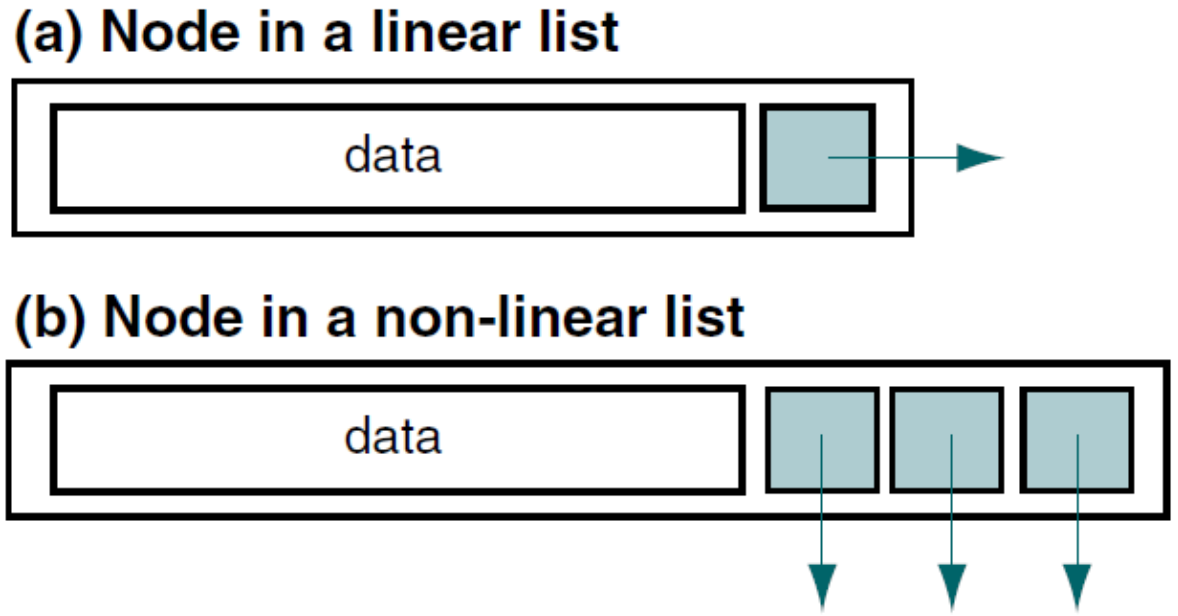


(b) Non-linear list

(c) Empty list

Nodes

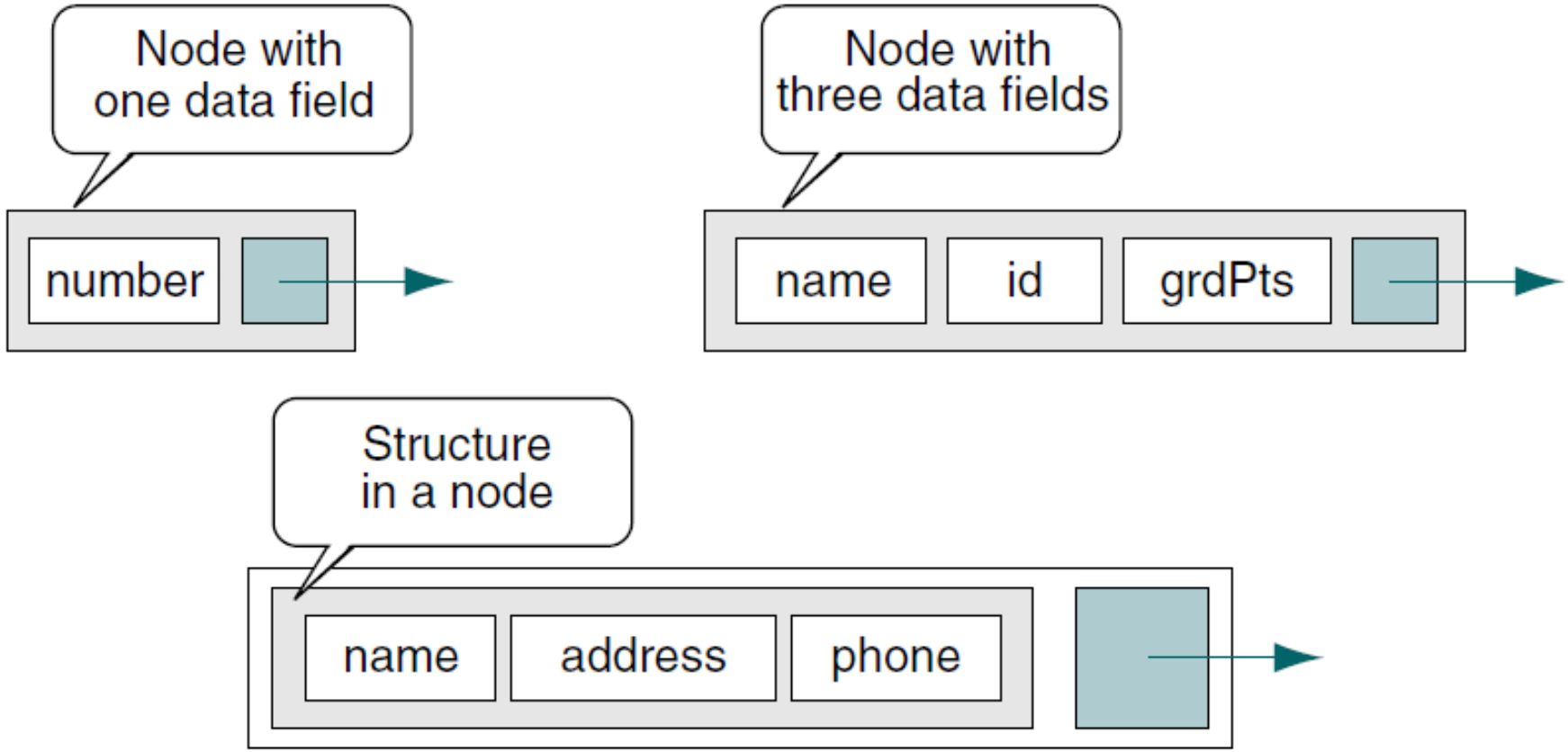
- A **node** is a structure that has two parts: the data and one or more links.
- The nodes in a linked list are called self-referential structures. In such a structure, each instance of the structure contains one or more pointers to other instances of the same structural type.



Nodes

- The data part in a node can be a single field, multiple fields, or a structure that contains several fields, but it always acts as a single field.

Linked List Node Structures



Linked List

- A chain of complex data structures “linked” together by pointers
- A sequential dynamic allocation that allocates memory when needed.
- It is sequential because the node can be accessed using the previous nodes.

- **Common Operations on Linked List**
  - Searching
  - Adding/Deleting Nodes
  - Inserting a new Element
  - Others

List Operations

- where L is a List and assuming that position p is defined
- **makenull(L)**
  - causes L become empty list
- **first(L)**
  - returns the first position of list L. If L is empty, the position returned is end(L).
- **end(L)**
  - returns the position following position n in an n-element list

List Operations

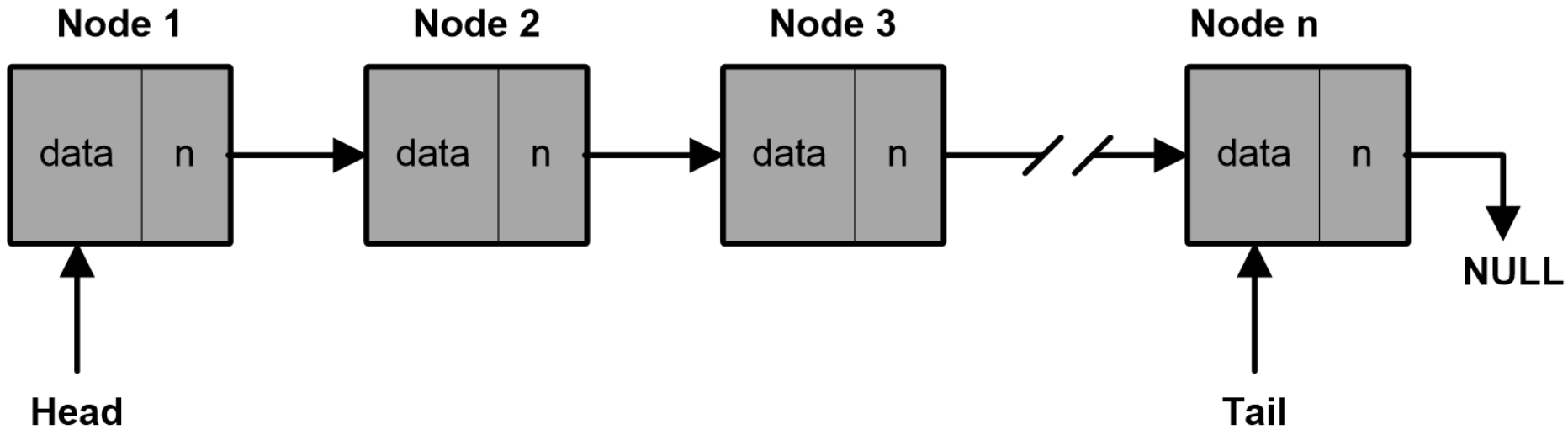
- **insert(x,p,L)**
  - inserts/adds x at position p in L
- **retrieve(p,L)**
  - retrieves the element at position p in L
- **delete(p,L)**
  - deletes the element at position p in L
- **next(p,L)**
  - returns the position following p in L

List Operations

- **previous(p,L)**
  - returns the position preceding position p in L
- **locate(x,L)**
  - returns the position of the element x in L. If more than one occurrences of x, the first position is returned
- **empty(L)**
  - returns true if L is empty, false otherwise
- **printList(L)**
  - prints the elements of L in the order of occurrence

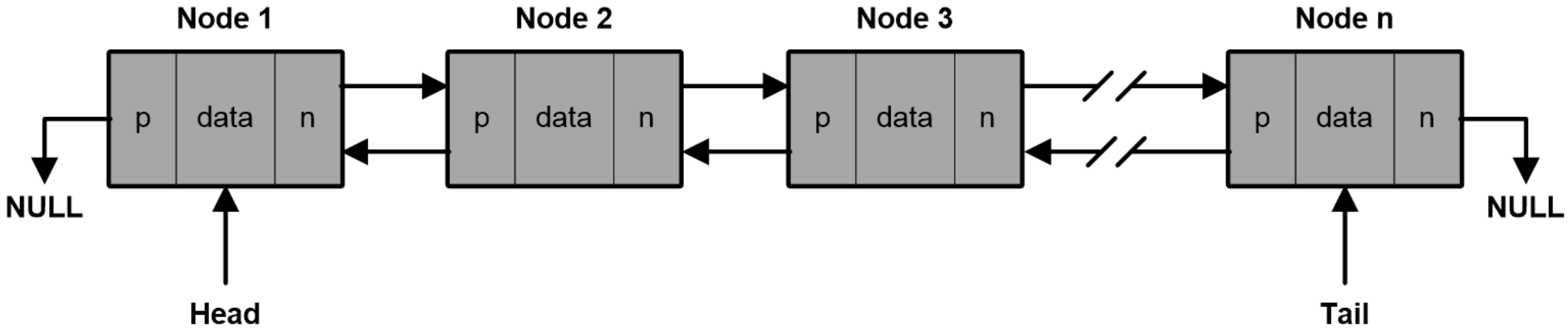
Types of Linked List

- **Singly Linked List**
  - A chain that requires each item of information to contain a link to the next item.



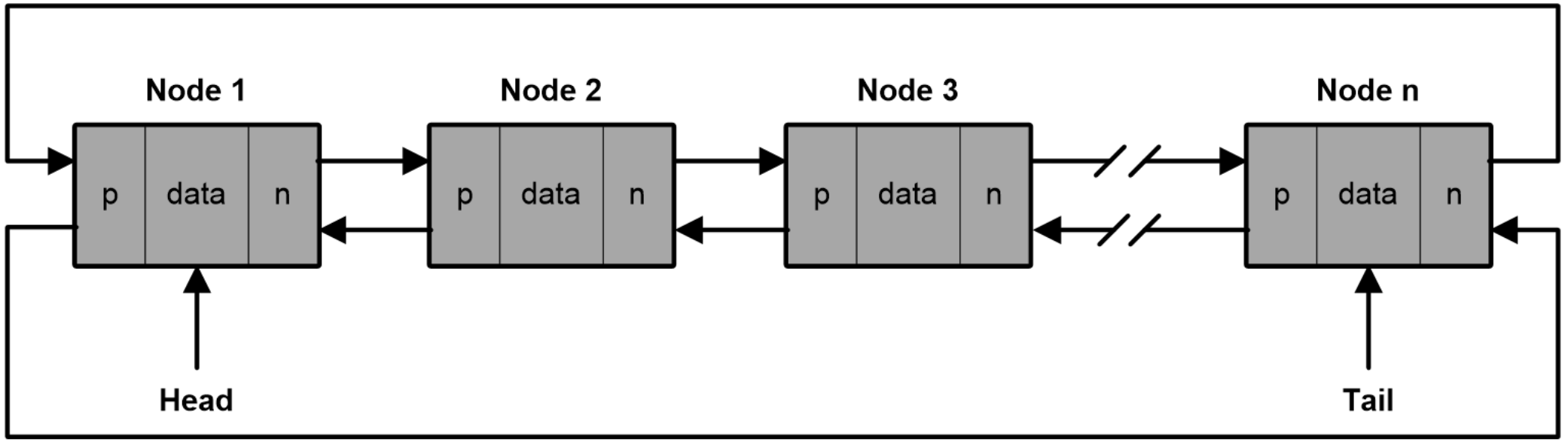
Types of Linked List

- **Doubly Linked List**
  - A chain that requires each item of information to contain links to the previous and next items.



Types of Linked List

- **Circular Linked List**
  - the last node points to the first node.



SINGLY (insert end)

```
typedef struct node
{
        int x;
        struct node *n;
}NODE;
```

```
typedef struct list
{
        struct node *head;
        struct node *tail;
        int count;
}LIST;
LIST *L;
```

```
/*first data*/
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 10;
newNode->n = NULL;
L->head = newNode;
L->tail = L->head;
(L->count)++;
```

SINGLY (insert end)

```
/*succeeding data*/
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 20;
newNode->n = NULL;
(L->tail)->n = newNode;
L->tail = newNode;
(L->count)++;
```

```
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 30;
newNode->n = NULL;
(L->tail)->n = newNode;
L->tail = newNode;
(L->count)++;
```

SINGLY (insert front)

```
typedef struct node
{
        int x;
        struct node *n;
}NODE;
```

```
typedef struct list
{
        struct node *head;
        struct node *tail;
        int count;
}LIST;
LIST *L;
```

```
/*first data*/
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 10;
newNode->n = NULL;
L->head = newNode;
L->tail = L->head;
(L->count)++;
```

SINGLY (insert in between nodes)

- Position = 0, insertFront
- Position = count, insertEnd
- Otherwise or between 0-count,

```
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 40;
newNode->n = NULL;
left->n = newNode;
left->n->n = right
(L->count)++;
```

DOUBLY (insert end)

```
typedef struct node
{
        int x;
        struct node *p;
        struct node *n;
}NODE;
```

```
typedef struct list
{
        struct node *head;
        struct node *tail;
        int count;
}LIST;
LIST *L;
```

```
/*first data*/
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 10;
newNode->p = NULL;
newNode->n = NULL;
L->head = newNode;
L->tail = L->head;
(L->count)++;
```

DOUBLY (insert end)

```
/*succeeding data*/
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 20;
newNode->p = NULL;
newNode->n = NULL;
(L->tail)->n = newNode;
(L->tail)->n->p = L->tail;
(L->tail)->n->n = NULL;
L->tail = (L->tail)->n;
(L->count)++;
```

```
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 30;
newNode->p = NULL;
newNode->n = NULL;
(L->tail)->n = newNode;
(L->tail)->n->p = L->tail;
(L->tail)->n->n = NULL;
L->tail = (L->tail)->n;
(L->count)++;
```

DOUBLY (insert front)

```
typedef struct node
{
        int x;
        struct node *p;
        struct node *n;
}NODE;
```

```
typedef struct list
{
        struct node *head;
        struct node *tail;
        int count;
}LIST;
LIST *L;
```

```
/*first data*/
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 10;
newNode->p = NULL;
newNode->n = NULL;
L->head = newNode;
L->tail = L->head;
(L->count)++;
```

DOUBLY (insert end)

```
/*succeeding data*/
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 20;
newNode->p = NULL;
newNode->n = NULL;
(L->head)->p = newNode;
(L->head)->p->n = (L->head);
(L->head) = newNode;
(L->count)++;
```

```
NODE *newNode =
(NODE*)malloc(sizeof(NODE));
newNode->x = 30;
newNode->p = NULL;
newNode->n = NULL;
(L->head)->p = newNode;
(L->head)->p->n = (L->head);
(L->head) = newNode;
(L->count)++;
```

DOUBLY (insert in between nodes)

- Position = 0, insertFront
- Position = count, insertEnd
- Otherwise or between 0-count,

```
NODE *newNode = (NODE*)malloc(sizeof(NODE));
newNode->x = 40;
newNode->p = NULL;
newNode->n = NULL;
left->n = newNode;
left->n->p = left;
left->n->n = right;
right->p = left->n;
(L->count)++;
```

Sample Program

- Singly Linked List
  - [1] Insert End
  - [2] Delete End
  - [3] Display List
  - [4] Quit

Algorithm Efficiency

Algorithm Efficiency

- The algorithm's efficiency is a function of the number of elements to be processed. The general format is

$$f(n) = \text{efficiency}$$

where n is the number of elements to be processed.

The Basic Concept

- When comparing two different algorithms that solve the same problem, we often find that one algorithm is an order of magnitude more efficient than the other.
- A typical example is a famous Fast Fourier Transform algorithm. It requires NlogN multiplications and additions, while a direct Fourier Transform algorithm requires N^2 multiplications and additions.

The Basic Concept

- If the efficiency function is linear then this means that the algorithm is linear and it contains no loops or recursions. In this case, the algorithm's efficiency depends only on the speed of the computer.

- If the algorithm contains loops or recursions (any recursion may always be converted to a loop), it is called nonlinear. In this case the efficiency function strongly and informally depends on the number of elements to be processed.

Algorithm Efficiency

- Linear Loops

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

$$f(n) = n$$

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

$$f(n) = n / 2$$

Algorithm Efficiency

- Logarithmic Loops

Multiply Loops

```
for (i = 0; i < 1000; i *= 2)
    application code
```

Divide Loops

```
for (i = 0; i < 1000; i /= 2)
    application code
```

```
multiply   2^Iterations < 1000
divide     1000 / 2^Iterations >= 1
```

$$f(n) = \log n$$

Algorithm Efficiency

- Nested Loops

```
Iterations = outer loop iterations x inner loop iterations
```

- Linear Logarithmic Loops

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

$$f(n) = n \log n$$

Algorithm Efficiency

- Quadratic Loop

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

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

- Dependent Quadratic

```
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 Θ

- Don't need to determine the complete measure of efficiency, only the factor that determines the magnitude. This factor is the big-O, as in "on the order of," and is expressed as O(n)—that is, on the order of n.

- This simplification of efficiency is known as big-O analysis.

Big-O Notation Θ

- **Big-O notation**: a measure of the efficiency of an algorithm in which only the dominant factor is considered.

- The seven standard measures of efficiencies are: **O (logn), O (n), O(n(logn)), O (n^2), O (n^k), O (c^n)** and **O (n!)**.

Big-O Notation Θ

- The big-O notation can be derived from *f* (*n*) using the following steps:

1. In each term, set the coefficient of the term to 1.
2. Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest as shown below.

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

Example

- Calculate the big-O notation:
  - $f(n) = n\frac{(n+1)}{2}$
  - $f(n) = a_j n^k + a_{j-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_o$

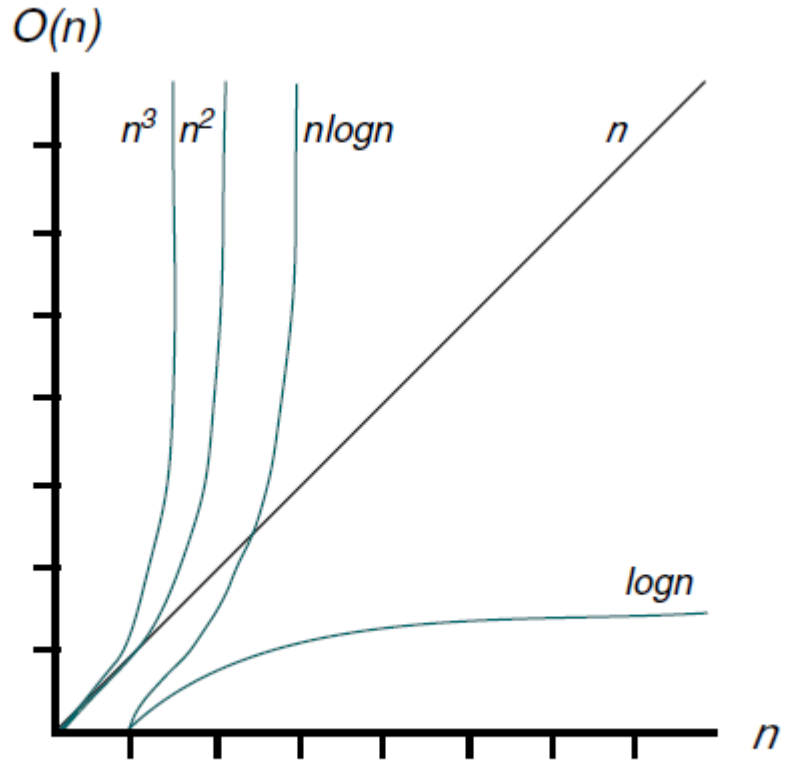
Big-O Notation Θ

- Standard Measures of Efficiency

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

Measures of Efficiency for n = 10,000

Big-O Notation Θ



Plot of Effeciency Measures

References

- Richard F. Gilberg, Behrouz A. Forouzan, **Computer Science: A Structured Programming Approach Using C 3rd edition**, Cengage Learning Course Technology © 2006

COMPUTER ENGINEERING
UNIVERSITY *of* SAN CARLOS