UNIT I
INTRODUCTION TO DATA STRUCTURES

Session	Description of Topic	Contact hours	C-D-I-O	IOs	Reference
	UNIT I: INTRODUCTION TO DATA STRUCTURES	6			
1.	Introduction – Basic terminology – Data structures – Data structure operations	1	C	1	1
2.	ADT – Algorithms: Complexity, Time – Space trade off	1	C	1	1
3.	Mathematical notations and functions	1	C	1	1
4.	Asymptotic notations – Linear and Binary search	1	C,D,I	1	1
5.	Asymptotic notations – Bubble sort	1	C,D,I	1	1
6.	Asymptotic notations -Insertion sort	1	C,D,I	1	1

- **Data Structure** is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

Basic terminology

- **Data** are values or a set of values
- **Data item** refers to single unit of values ⍰ Data item
- **Group item** : Data item that can be subdivided into sub item. Ex Name : First Name, Middle initial and Last Name
- **Elementary item**: Data item that can not be sub divided into sub item Ex : PAN card number / Bank Pass Book Number is treated as single item
- Collection of data are frequently organized into a hierarchy of fields, records and files

- **Entity** with similar attributes (e.g all employees of an organization) form an entity set
- Each attribute of an entity set has a **range of values** [the set of possible values that could be assigned to the particular attribute]
- **Information:** processed data, Data with given attribute
- **Field** is a single elementary unit of information representing an attribute of an entity
- **Record** is the collection of field values of a given entity
- **File** is the collection of records of the entities in a given entity set

Nam	Age	Sex	Roll Number	Branch
A	17	M	109cs0132	CSE
B	18	M	109ee1234	EE

- Record

  1. **Fixed Length Record**: all records contain the same data items with the same amount of space assigned to each data items

  2. **Variable Length Record**: file records may contain different lengths

Study of Data Structure includes the following three steps

1. Logical or Mathematical description of the structure
2. Implementation of the structure on a computer
3. Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure

Classification of Data structures

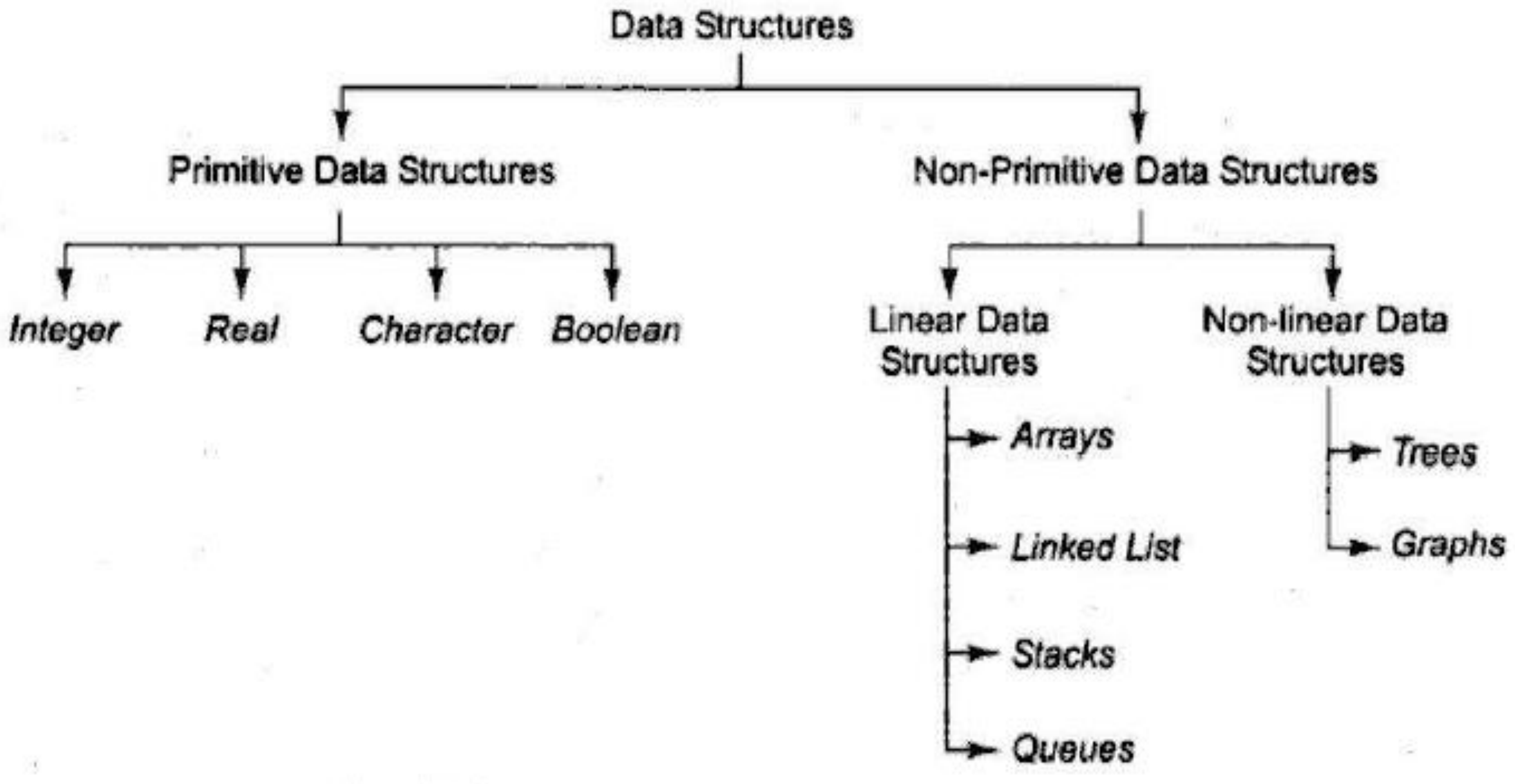


Fig. 1.1 Classification of Data Structures

PRIMITIVE DATATYPE

The primitive data types are the basic data types that are available in most of the programming languages. The primitive data types are used to represent single values.

NON-PRIMITIVE DATATYPES

The data types that are derived from primary data types are known as non-Primitive data types. These datatypes are used to store group of values.

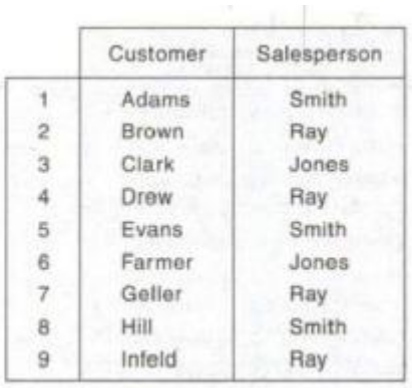
Arrays

- Arrays are statically implemented data structures by some programming languages like C and C++;
- hence the **size** of this data structure must be known at compile time and cannot be altered at run time.
- But modern programming languages, for example, Java implements **arrays as objects** and give the programmer a way to alter the size of them at run time.
- Arrays are the most common data structure used to store data.

Linked List

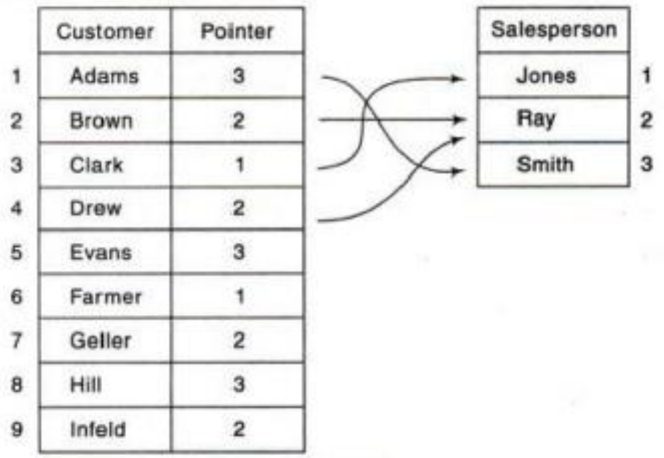
- Linked list data structure provides better memory management than arrays.
- Because linked list is allocated memory at run time, so, there is **no waste of memory**.
- Performance wise linked list is **slower than** array because there is no direct access to linked list elements.
- Linked list is proved to be a useful data structure when the number of elements to be stored is not known ahead of time.

Linked list



	Customer	Salesperson
1	Adams	Smith
2	Brown	Ray
3	Clark	Jones
4	Drew	Ray
5	Evans	Smith
6	Farmer	Jones
7	Geller	Ray
8	Hill	Smith
9	Infeld	Ray

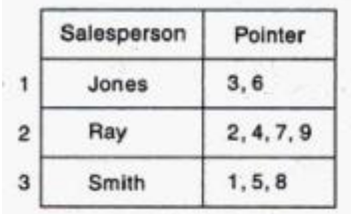
Fig. 1.4



	Customer	Pointer
1	Adams	3
2	Brown	2
3	Clark	1
4	Drew	2
5	Evans	3
6	Farmer	1
7	Geller	2
8	Hill	3
9	Infeld	2

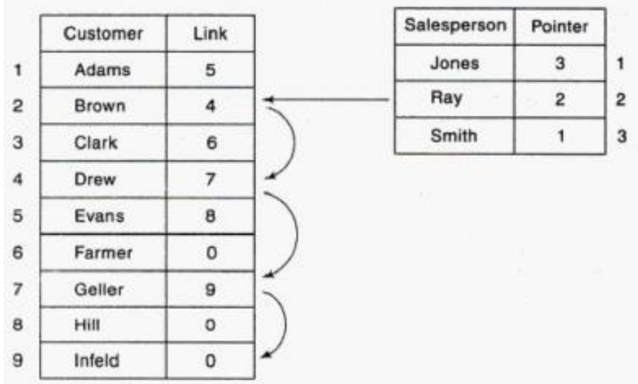
Salesperson	
Jones	1
Ray	2
Smith	3

Fig. 1.5



	Salesperson	Pointer
1	Jones	3, 6
2	Ray	2, 4, 7, 9
3	Smith	1, 5, 8

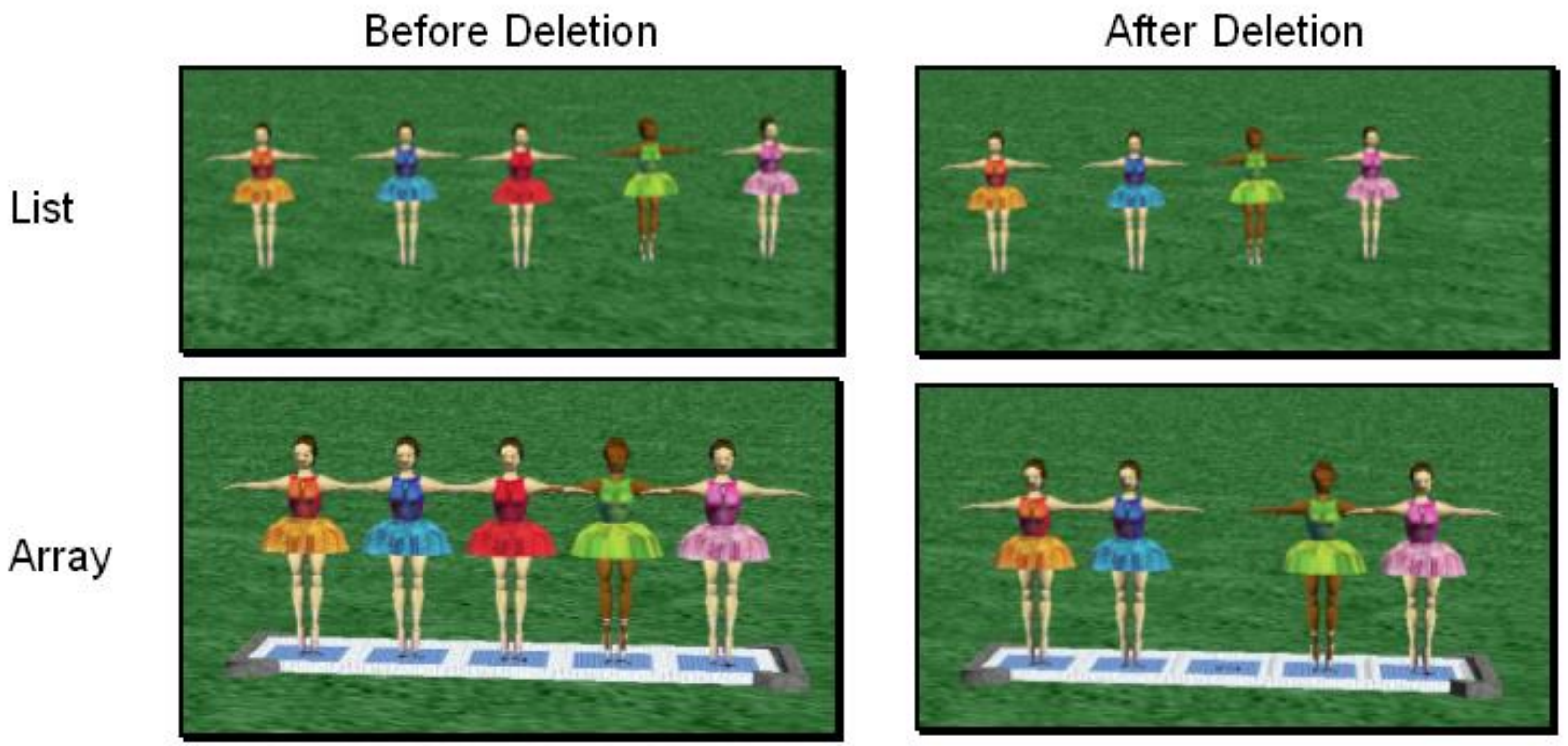
Fig. 1.6



	Customer	Link
1	Adams	5
2	Brown	4
3	Clark	6
4	Drew	7
5	Evans	8
6	Farmer	0
7	Geller	9
8	Hill	0
9	Infeld	0

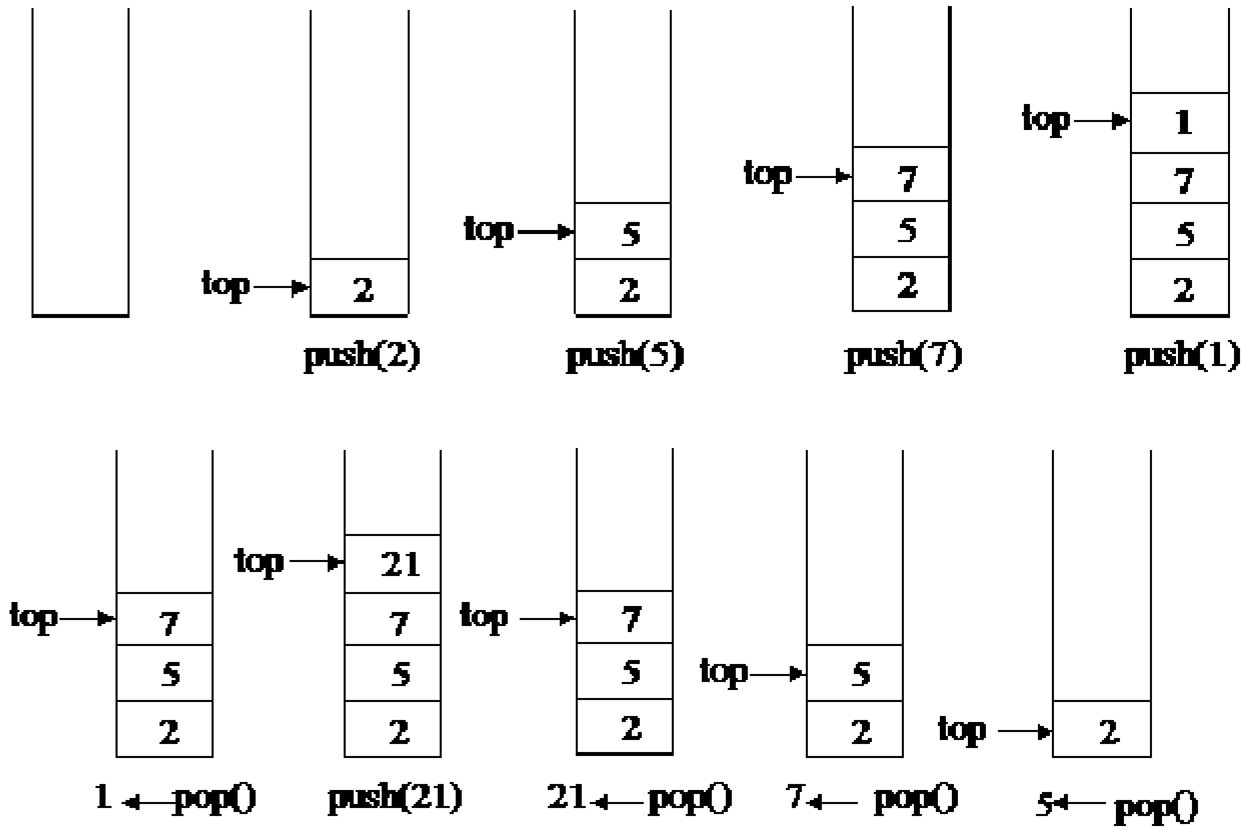
Salesperson	Pointer	
Jones	3	1
Ray	2	2
Smith	1	3

Fig. 1.7


Before Deletion
After Deletion
List
Array

Stack

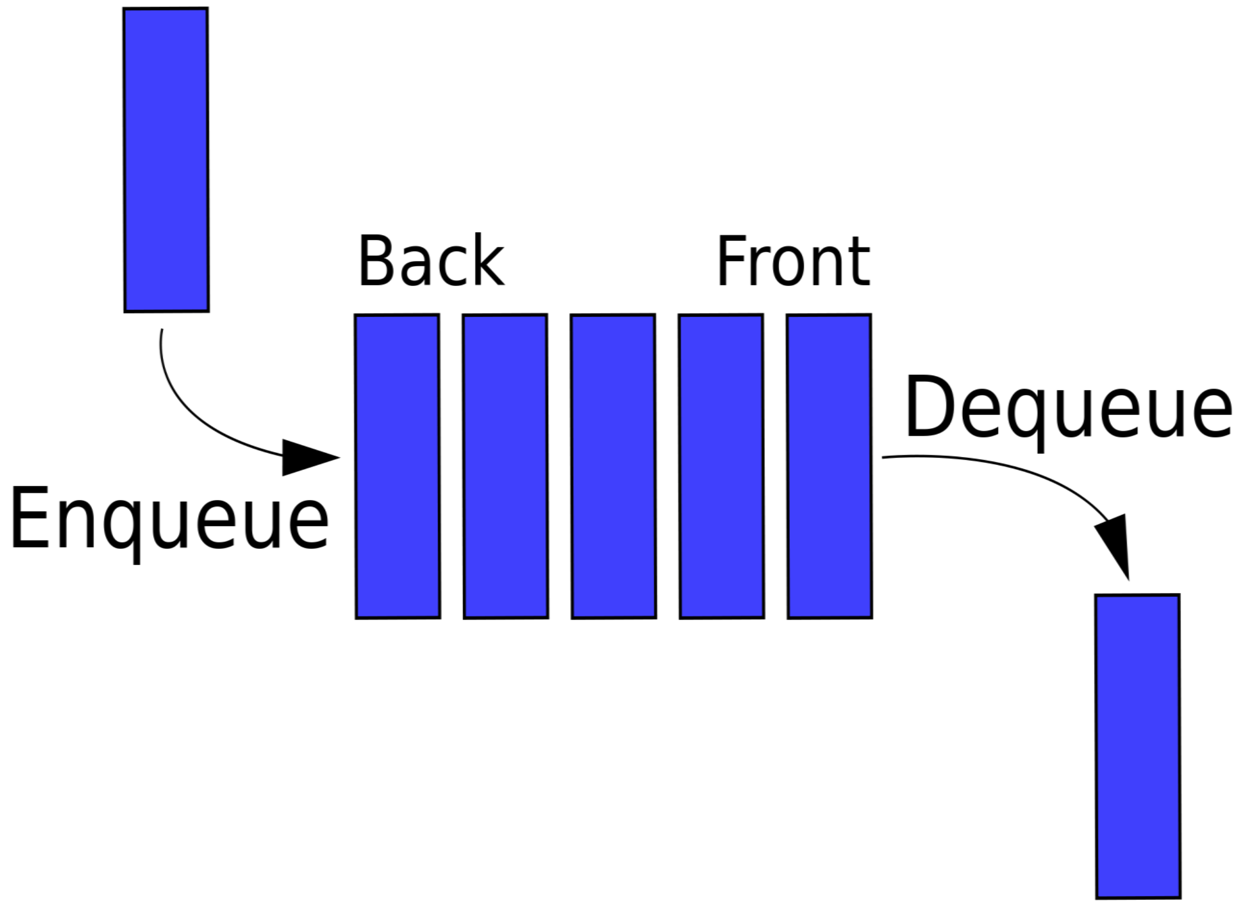
- Stack is a last-in-first-out strategy data structure; this means that the element stored in last will be removed first.
- Stack has specific but very useful applications; some of them are as follows:
    - Solving Recursion - recursive calls are placed onto a stack, and removed from there once they are processed.
    - Evaluating post-fix expressions
    - Solving Towers of Hanoi
    - Backtracking
    - Depth-first search
    - Converting a decimal number into a binary number



top
2
push(2)
top
5
2
push(5)
top
7
5
2
push(7)
top
1
7
5
2
push(1)
top
7
5
2
1 ← pop()
top
21
7
5
2
push(21)
top
7
5
2
21 ← pop()
top
5
2
7 ← pop()
top
2
5 ← pop()

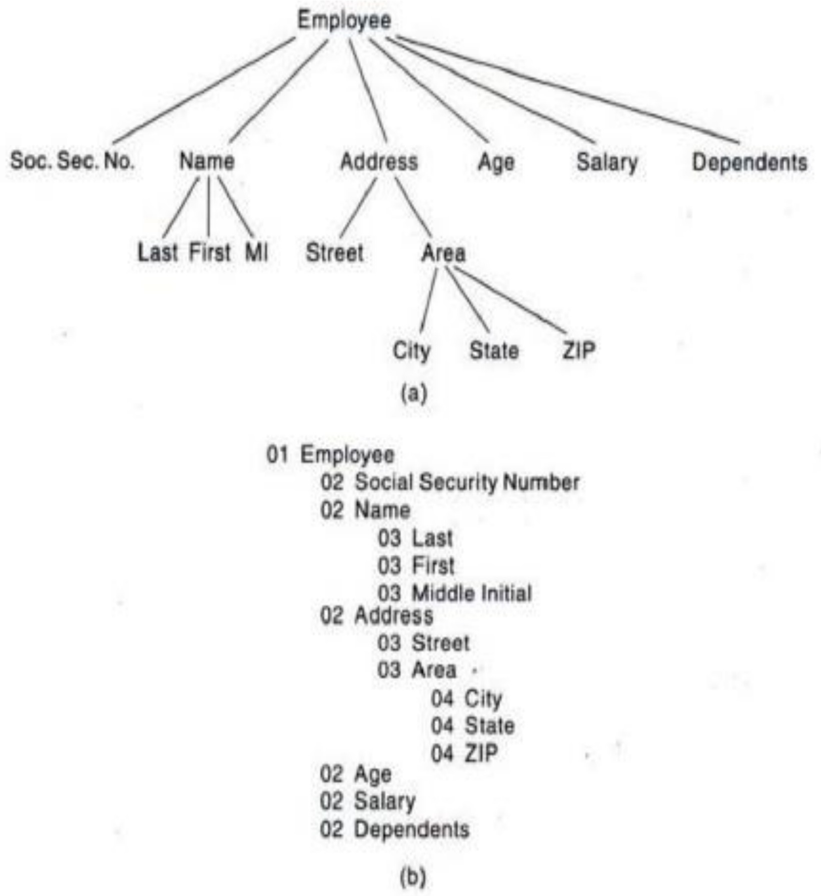
Queue

- Queue is a first-in-first-out data structure.
- The element that is added to the queue data structure first, will be removed from the queue first.
- Dequeue, priority queue, and circular queue are the variants of queue data structure. Queue has the following application uses:
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - Message queue


Back
Front
Enqueue
Dequeue

Trees

- Tree is a hierarchical data structure. The very top element of a tree is called the root of the tree.
- Except the root element every element in a tree has a parent element, and zero or more children elements.
- All elements in the left sub-tree come before the root in sorting order, and all those in the right sub-tree come after the root.
- Tree is the most useful data structure when you have hierarchical information to store.
- For example, directory structure of a file system; there are many variants of tree you will come across. Some of them are Red-black tree, threaded binary tree, AVL tree, etc.

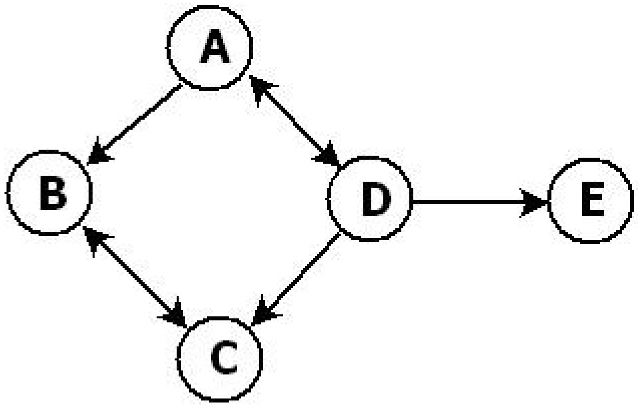


```
01 Employee
     02 Social Security Number
     02 Name
          03 Last
          03 First
          03 Middle Initial
     02 Address
          03 Street
          03 Area
               04 City
               04 State
               04 ZIP
     02 Age
     02 Salary
     02 Dependents
```

Fig. 1.8

Graph

- Graph is a networked data structure that connects a collection of nodes called vertices, by connections, called edges.
- An edge can be seen as a path or communication link between two nodes.
- These edges can be either directed or undirected.
- If a path is directed then you can move in one direction only, while in an undirected path the movement is possible in both directions.


A
B
C
D
E

Data Structure operations

- The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.
    - Traversing
    - Searching
    - Insertion
    - Deletion
    - Sorting
    - Merging

(1) *Traversing:* Accessing each records exactly once so that certain items in the record may be processed.

(2) *Searching:* Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.

(3) *Inserting:* Adding a new record to the structure.

(4) *Deleting:* Removing the record from the structure.

(5) *Sorting:* Managing the data or record in some logical order(Ascending or descending order).

(6) *Merging:* Combining the record in two different sorted files into a single sorted file.

ADT

- An abstract data type (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementaion .
- With an ADT, we know what a specific data type can do, but **how it actually does it is hidden.**
- Simply hiding the implementation

Example 1.7 List Representation

Consider a list *L* consisting of data items—1, 2, 3, 4, 5, 6, 7, 8, 9 as shown in Fig. 1.11(a). We can use any of four data structures to support *L*—a linear list, a matrix, a tree, or a graph, as given in Fig. 1.11(b), (c) and (d) respectively.

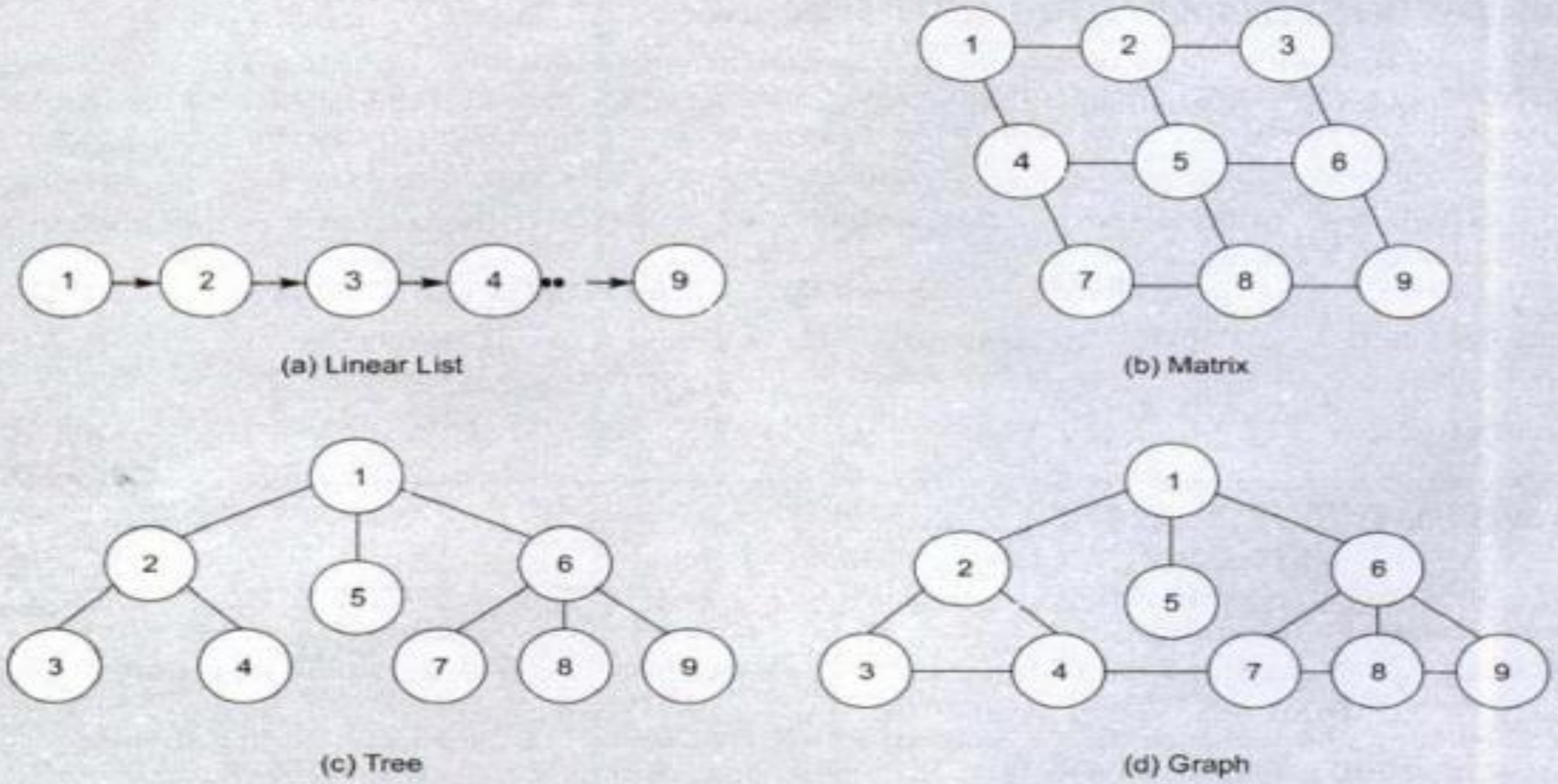


Fig. 1.11 Data structures which support a list

Assume that we place the list on an ADT. The users should not be aware of the structure that we use, i.e., whether it is a tree, or graph or something else. As long as they are able to insert and retrieve data, it does not make a difference as to how we store the data.

Example 1.8

A shop maintains the list of customers, sales assistants and the average transactions on a day as given in Fig. 1.12.

Suppose we need to write a program, which will help determine the number of sales assistants required to serve customers efficiently. Here, we will need to simulate the waiting line in the shop. This analysis will require the simulation of a queue. However, queues are not generally available in programming languages. Therefore, even if the queue type is available, we need some basic queue operations such as enqueuing and dequeuing, which are basically insertion and deletion operations, for the simulation.

Here is what we can do in this situation:

1. Write a program that simulates the queue, or
2. Write a queue ADT that can solve any queue problem.

If we choose the second option, we still need to write a program to simulate the shop application. However, doing that will actually be simpler and faster because we can concentrate on the application rather than the queue.

	Customers
1	Customers
2	Customers
3	Customers

	Sales Assistant
1	Ray
2	Reed
3	Kelly

Fig. 1.12

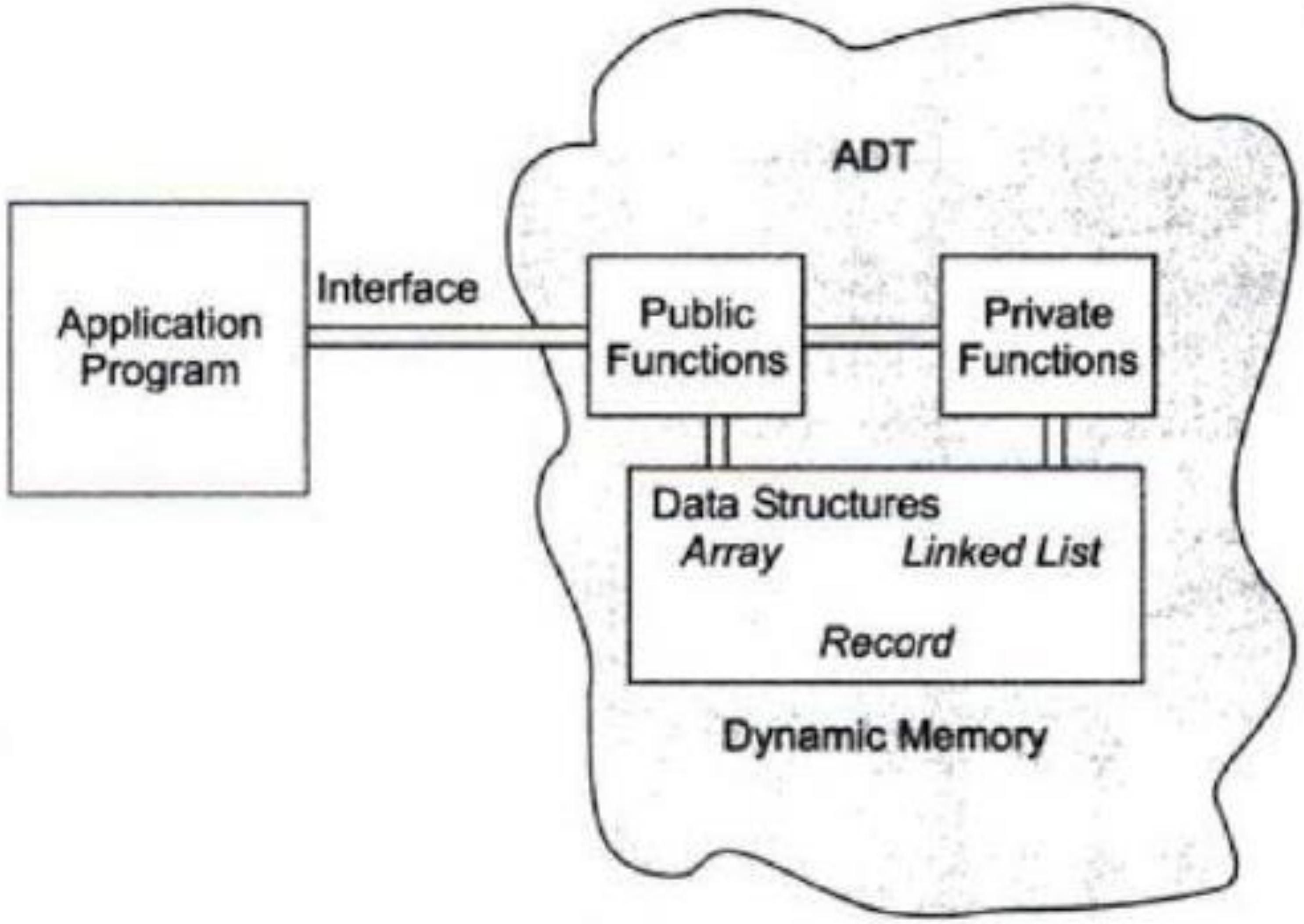


Fig. 1.13 ADT Model

Algorithms: Complexity, time-space tradeoff

- Algorithm is a well defined list of steps for solving a particular problem.
- One major purpose of this text is to develop efficient algorithms for the processing of our data.
- The time and space it uses are two major measures of the efficiency of an algorithm.
- The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size.

Algorithm Analysis

- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below
- ***A priori* analysis** − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
- ***A posterior* analysis** − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

space complexity of an algorithm represents the **amount of memory space** required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

- A **fixed part** that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
- A **variable part** is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

- Space complexity **S(P)** of any algorithm **P** is **S(P) = C + SP(I)** Where **C** is the fixed part and **S(I)** is the variable part of the algorithm which depends on instance characteristic **I**.

- Algorithm: SUM(A, B)
- Step 1 - START
- Step 2 - C ← A + B + 10
- Step 3 – Stop

- Here we have three variables A, B and C and one constant. Hence **S(P) = 1+3**.
- Now space depends on **data types** of given variables and constant types and it will be multiplied accordingly.

```
Algorithm sum( a[], n)
{
sum =0;
n;
for(i=0;i<=n;i++)
{
sum = sum + a[i];
return sum;
}
```

Space complexity is calculated as follows

One computer word is required to store **n**.

n space is required to store **array a[]**.

One space for variable **sum** and **one** for **i** is required.

Total Space(S) =1 +n +1 +1=n+3

Time Complexity

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.

Ways to measure time complexity

- Use a **stop watch** and time is obtained in **seconds or milliseconds**.
- **Step Count** - Count no of program steps.
  - **Comments** are not evaluated so they are **not** considered as program **step**.
  - In **while** loop **steps** are equal to the **number** of times loop gets **executed**.
  - In **for** loop,steps are equal to number of times an **expression** is checked for **condition**.
  - A single expression is considered as a **single** step.Example **a+f+r+w+w/q-d-f** is **one step**.

- for (i:=0;i<n;i++)

  //i=0 → 1time

  //i<n → n+1 times

  //i++ →n times

- x:=x+1; // n times

- i := 1;
- while (i < n)
- i = i * 2;

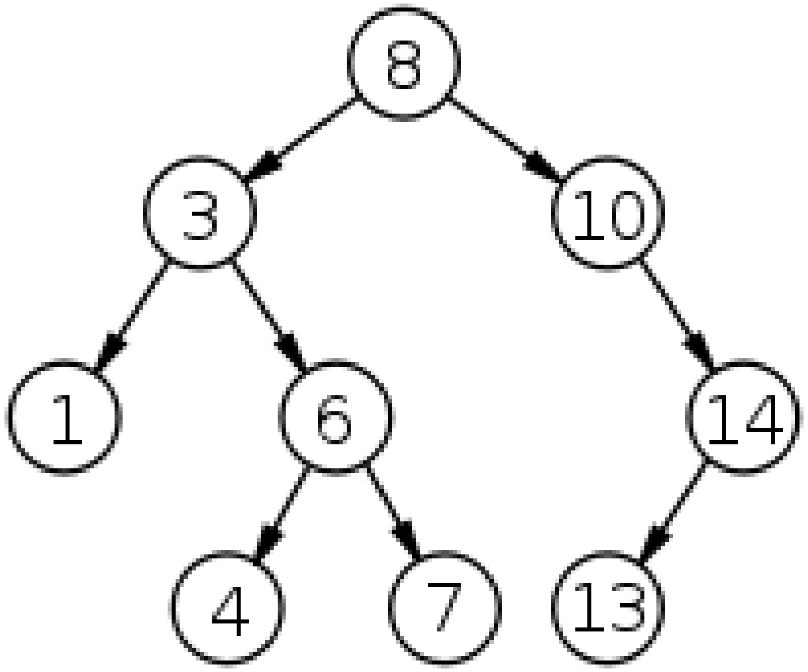
Size of list	Time to find a number
3	3
7	4
10	5
15	5
31	6
n	$2^{t-1} - 1$

- $2^{t-1} - 1 = n$
- $2^{t-1} = n+1$
- $\log 2^{t-1} = \log(n+1)$
- $t = \log(n+1) + 1$
- $t = \log n$

- Double test(int n)
- {
- int sum=0;
- int i;
- for(i=0; i<=n;i++)
- { scanf("%d",&a);
- sum=sum+a;
- }
- return sum; }

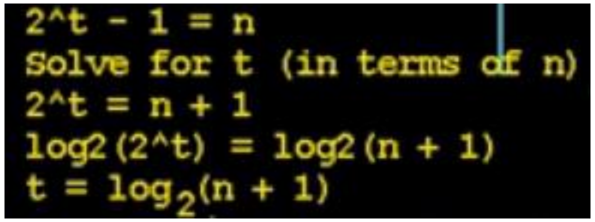
- Double test(int n)
- {
- int sum=0; -> 1 time
- int i; -> 0 time
- for(i=0; i<=n;i++) -> n+2 times
- { scanf("%d",&a); -> n+1 times
- sum=sum+a; -> n+1 times
- }
- return sum; -> 1 time }

Binary search tree



Size of list	Time to find a number
1	1
3	2
7	3
10	4
15	4
31	5

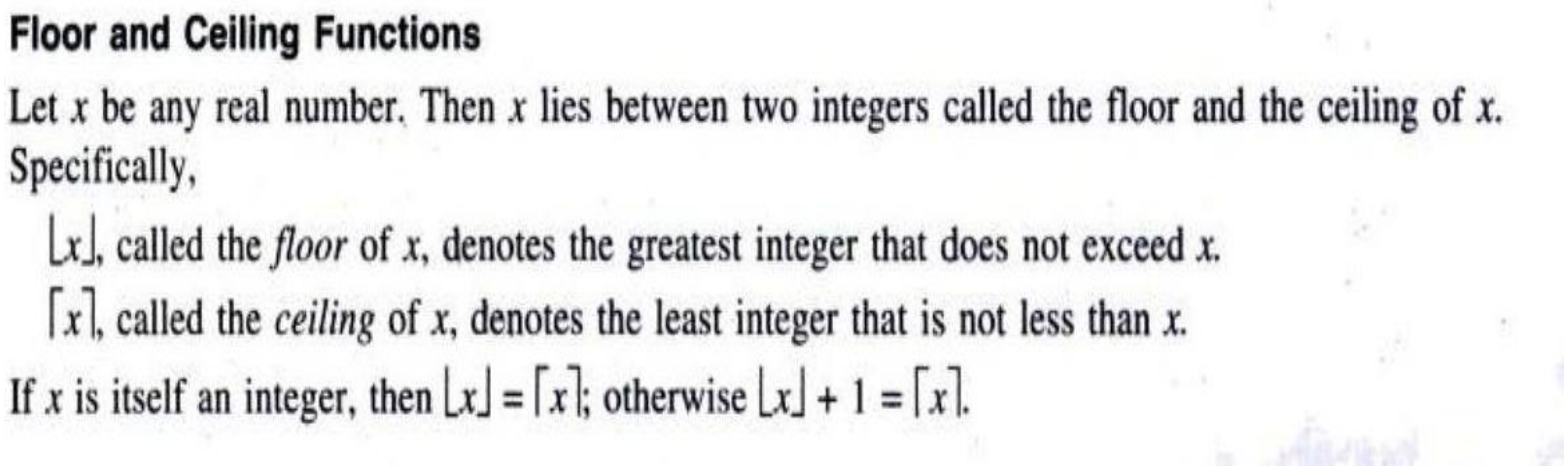
Size of list	Time to find a number
1	1
3	2
7	3
10	4
15	4
31	5
$2^t - 1$	n
	?



```
2^t - 1 = n
Solve for t (in terms of n)
2^t = n + 1
log2(2^t) = log2(n + 1)
t = log_2(n + 1)
```

log n

Mathematical Notation and Function



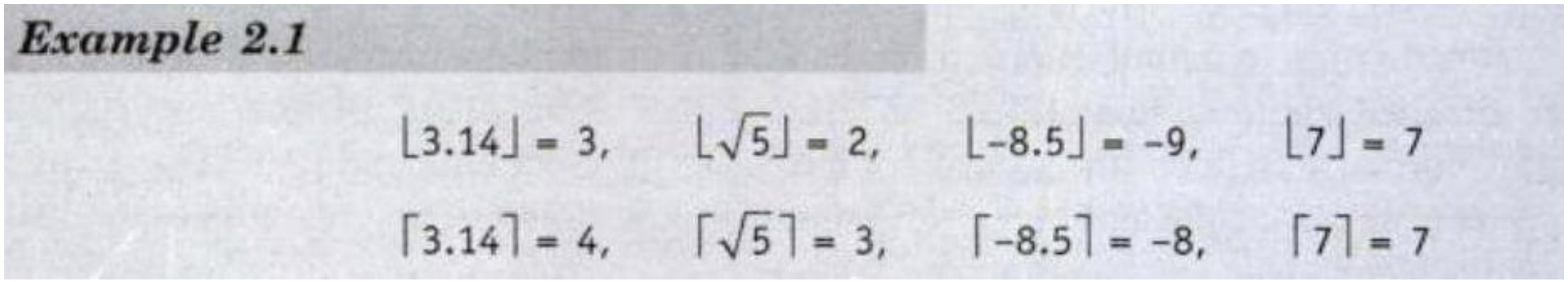
Floor and Ceiling Functions

Let x be any real number. Then x lies between two integers called the floor and the ceiling of x. Specifically,

$\lfloor x \rfloor$, called the *floor* of x, denotes the greatest integer that does not exceed x.

$\lceil x \rceil$, called the *ceiling* of x, denotes the least integer that is not less than x.

If x is itself an integer, then $\lfloor x \rfloor = \lceil x \rceil$; otherwise $\lfloor x \rfloor + 1 = \lceil x \rceil$.



Example 2.1

$\lfloor 3.14 \rfloor = 3, \quad \lfloor \sqrt{5} \rfloor = 2, \quad \lfloor -8.5 \rfloor = -9, \quad \lfloor 7 \rfloor = 7$

$\lceil 3.14 \rceil = 4, \quad \lceil \sqrt{5} \rceil = 3, \quad \lceil -8.5 \rceil = -8, \quad \lceil 7 \rceil = 7$

Remainder Function; Modular Arithmetic

Let k be any integer and let M be a positive integer. Then

$$k \pmod{M}$$

(read *k modulo M*) will denote the integer remainder when k is divided by M. More exactly, $k \pmod{M}$ is the unique integer r such that

Integer and Absolute Value Functions

Let x be any real number. The *integer value* of x, written INT(x), converts x into an integer by deleting (truncating) the fractional part of the number. Thus

$$\text{INT}(3.14) = 3, \quad \text{INT}(\sqrt{5}) = 2, \quad \text{INT}(-8.5) = -8, \quad \text{INT}(7) = 7$$

Observe that $\text{INT}(x) = \lfloor x \rfloor$ or $\text{INT}(x) = \lceil x \rceil$ according to whether x is positive or negative.

The *absolute value* of the real number x, written ABS(x) or $|x|$, is defined as the greater of x or $-x$. Hence ABS(0) = 0, and, for $x \neq 0$, $\text{ABS}(x) = x$ or $\text{ABS}(x) = -x$, depending on whether x is positive or negative. Thus

$$|-15| = 15, \quad |7| = 7, \quad |-3.33| = 3.33, \quad |4.44| = 4.44, \quad |-0.075| = 0.075$$

Summation Symbol; Sums

Here we introduce the summation symbol Σ (the Greek letter sigma). Consider a sequence a_1, a_2, a_3, ... Then the sums

$$a_1 + a_2 + \cdots + a_n \quad \text{and} \quad a_m + a_{m+1} + \cdots + a_n$$

will be denoted, respectively, by

$$\sum_{j=1}^{n} a_j \quad \text{and} \quad \sum_{j=m}^{n} a_j$$

Factorial Function

The product of the positive integers from 1 to n, inclusive, is denoted by $n!$ (read "n factorial"). That is,

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1)n$$

It is also convenient to define $0! = 1$.

Example 2.3

(a) $2! = 1 \cdot 2 = 2$; $3! = 1 \cdot 2 \cdot 3 = 6$; $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$

(b) For $n > 1$, we have $n! = n \cdot (n-1)!$ Hence

$$5! = 5 \cdot 4! = 5 \cdot 24 = 120; \qquad 6! = 6 \cdot 5! = 6 \cdot 120 = 720$$

Permutations

A *permutation* of a set of n elements is an arrangement of the elements in a given order. For example, the permutations of the set consisting of the elements a, b, c are as follows:

$$abc, \quad acb, \quad bac, \quad bca, \quad cab, \quad cba$$

One can prove: *There are $n!$ permutations of a set of n elements.* Accordingly, there are $4! = 24$ permutations of a set with 4 elements, $5! = 120$ permutations of a set with 5 elements, and so on.

Exponents and Logarithms

Recall the following definitions for integer exponents (where m is a positive integer):

$$a^m = a \cdot a \cdots a \ (m \text{ times}), \quad a^0 = 1, \quad a^{-m} = \frac{1}{a^m}$$

Exponents are extended to include all rational numbers by defining, for any rational number m/n,

$$a^{m/n} = \sqrt[n]{a^m} = \left(\sqrt[n]{a}\right)^m$$

For example,

$$2^4 = 16, \qquad 2^{-4} = \frac{1}{2^4} = \frac{1}{16}, \ 125^{2/3} = 5^2 = 25$$

In fact, exponents are extended to include all real numbers by defining, for any real number x,

$$a^x = \lim_{r \to x} a^r \qquad \text{where } r \text{ is a rational number}$$

Accordingly, the exponential function $f(x) = a^x$ is defined for all real numbers.

Logarithms are related to exponents as follows. Let b be a positive number. The logarithm of any positive number x to the base b, written

$$\log_b x$$

represents the exponent to which b must be raised to obtain x. That is,

$$y = \log_b x \quad \text{and} \quad b^y = x$$

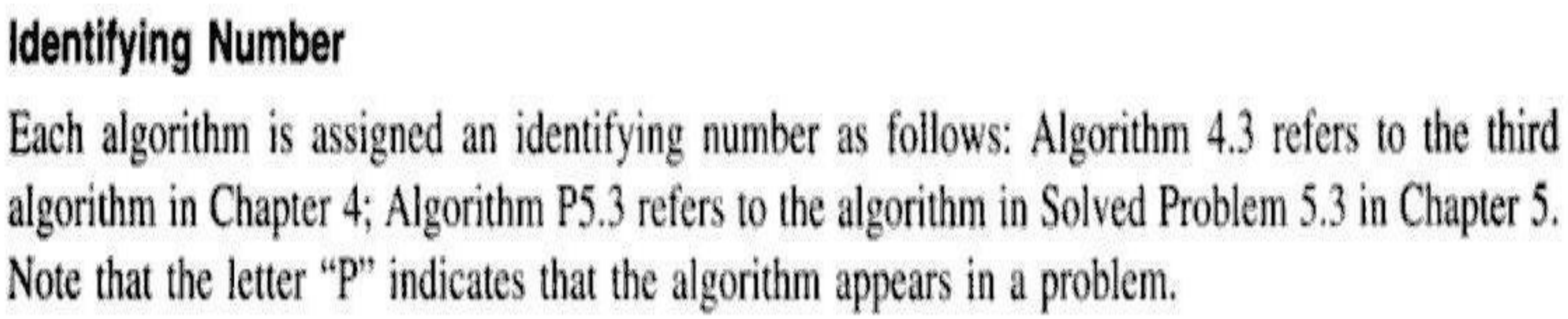
are equivalent statements. Accordingly,

$$\begin{array}{llllll} \log_2 8 = 3 & \text{since} & 2^3 = 8; & \log_{10} 100 = 2 & \text{since} & 10^2 = 100 \\ \log_2 64 = 6 & \text{since} & 2^6 = 64; & \log_{10} 0.001 = -3 & \text{since} & 10^{-3} = 0.001 \end{array}$$

Furthermore, for any base b,

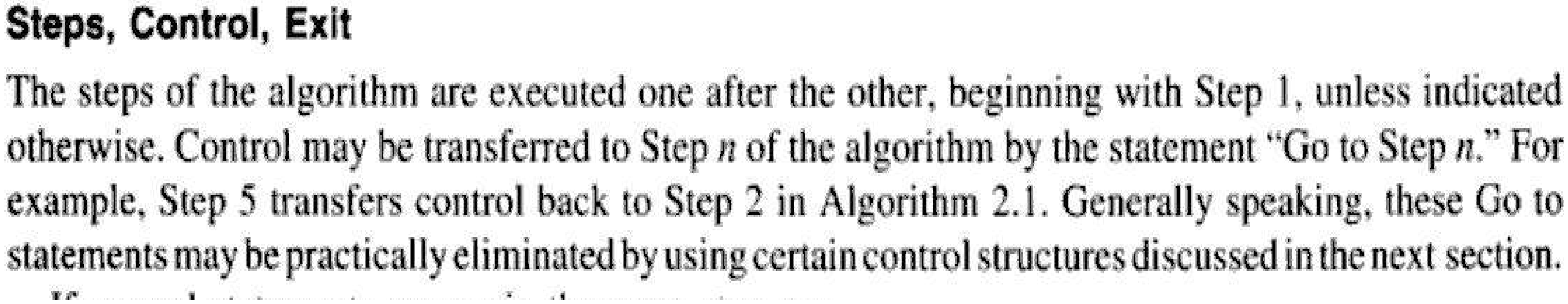
$$\begin{array}{lll} \log_b 1 = 0 & \text{since} & b^0 = 1 \\ \log_b b = 1 & \text{since} & b^1 = b \end{array}$$

Mathematical notations



Identifying Number

Each algorithm is assigned an identifying number as follows: Algorithm 4.3 refers to the third algorithm in Chapter 4; Algorithm P5.3 refers to the algorithm in Solved Problem 5.3 in Chapter 5. Note that the letter "P" indicates that the algorithm appears in a problem.



Steps, Control, Exit

The steps of the algorithm are executed one after the other, beginning with Step 1, unless indicated otherwise. Control may be transferred to Step *n* of the algorithm by the statement "Go to Step *n*." For example, Step 5 transfers control back to Step 2 in Algorithm 2.1. Generally speaking, these Go to statements may be practically eliminated by using certain control structures discussed in the next section.

Example 2.4

An array DATA of numerical values is in memory. We want to find the location LOC and the value MAX of the largest element of DATA. Given no other information about DATA, one way to solve the problem is as follows:

Initially begin with LOC = 1 and MAX = DATA[1]. Then compare MAX with each successive element DATA[K] of DATA. If DATA[K] exceeds MAX, then update LOC and MAX so that LOC = K and MAX = DATA[K]. The final values appearing in LOC and MAX give the location and value of the largest element of DATA.

A formal presentation of this algorithm, whose flow chart appears in Fig. 2.2, follows.

Algorithm 2.1: (Largest Element in Array) A nonempty array DATA with N numerical values is given. This algorithm finds the location LOC and the value MAX of the largest element of DATA. The variable K is used as a counter.

```
Step 1.  [Initialize.] Set K : = 1, LOC : = 1 and MAX : = DATA[1].
Step 2.  [Increment counter.] Set K : = K + 1.
Step 3.  [Test counter.] If K > N, then:
         Write: LOC, MAX, and Exit.
Step 4.  [Compare and update.] If MAX < DATA[K], then:
         Set LOC : = K and MAX : = DATA[K].
Step 5.  [Repeat loop.] Go to Step 2.
```

Comments

Each step may contain a comment in brackets which indicates the main purpose of the step. The comment will usually appear at the beginning or the end of the step.

Variable Names

Variable names will use capital letters, as in MAX and DATA. Single-letter names of variables used as counters or subscripts will also be capitalized in the algorithms (K and N, for example), even though lowercase may be used for these same variables (k and n) in the accompanying mathematical description and analysis. (Recall the discussion of italic and lowercase symbols in Sec. 1.3 of Chapter 1, under "Arrays.")

Assignment Statement

Our assignment statements will use the dots-equal notation := that is used in Pascal. For example,

Max := DATA[1]

assigns the value in DATA[1] to MAX. In C language, we use the equal sign = for this operation.

Input and Output

Data may be input and assigned to variables by means of a Read stateme.. with the following form:

Read: Variables names.

Similarly, messages, placed in quotation marks, and data in variables may be output by means of a Write or Print statement with the following form:

Write: Messages and/or variable names.

Procedures

The term "procedure" will be used for an independent algorithmic module which solves a particular problem. The use of the word "procedure" or "module" rather than "algorithm" for a given problem is simply a matter of taste. Generally speaking, the word "algorithm" will be reserved for the solution of general problems. The term "procedure" will also be used to describe a certain type of subalgorithm which is discussed in Sec. 2.6.

Asymptotic notation

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Usually, time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- O Notation- It measures the **worst case time** complexity or longest amount of time an algorithm can possibly take to complete.

- Ω Notation- It measures the **best case time** complexity or best amount of time an algorithm can possibly take to complete.

- θ Notation- The θ(n) is the formal way to express both the **lower bound and upper bound** of an algorithm's running time.

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

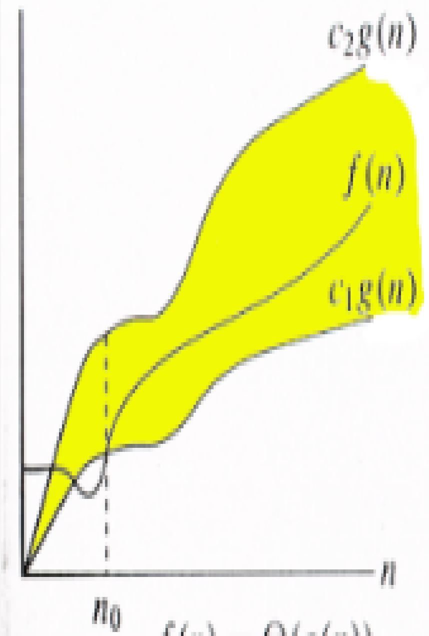
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta n^2)$ irrespective of the constants involved.

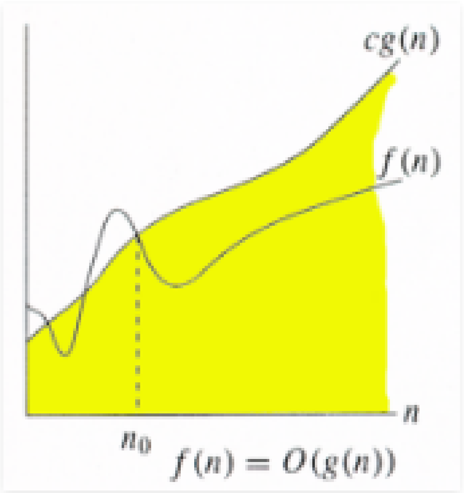
For a given function g(n), we denote Θ(g(n)) is following set of functions.

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
                 that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= 
```



$f(n) = \Theta(g(n))$

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.



If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

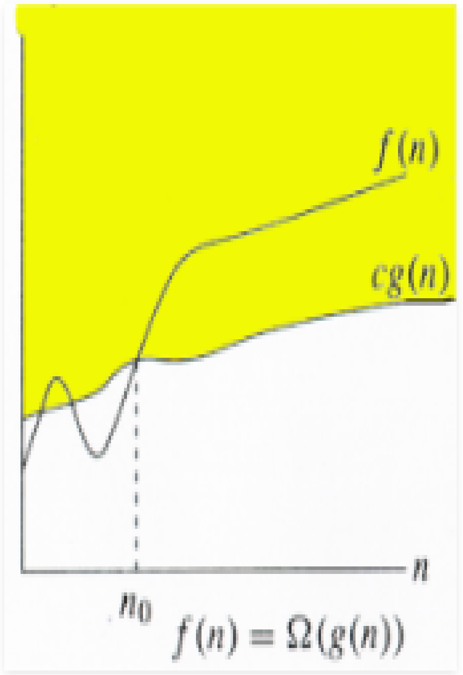
```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= cg(n) for
                  all n >= n0}
```

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
Ω (g(n)) = {f(n): there exist positive constants c and
                  n0 such that 0 <= cg(n) <= f(n) for
                  all n >= n0}.
```



Example 2.8 Linear Search

Suppose a linear array DATA contains *n* elements, and suppose a specific ITEM of information is given. We want either to find the location LOC of ITEM in the array DATA, or to send some message, such as LOC = 0, to indicate that ITEM does not appear in DATA. The linear search algorithm solves this problem by comparing ITEM, one by one, with each element in DATA. That is, we compare ITEM with DATA[1], then DATA[2], and so on, until we find LOC such that ITEM = DATA[LOC]. A formal presentation of this algorithm follows.

Algorithm 2.4: (Linear Search) A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC = 0.

```
1.  [Initialize] Set K := 1 and LOC := 0.
2.  Repeat Steps 3 and 4 while LOC = 0 and K ≤ N.
3.      If ITEM = DATA[K], then: Set LOC: = K.
4.      Set K := K + 1. [Increments counter.]
    [End of Step 2 loop.]
5.  [Successful?]
    If LOC = 0, then:
        Write: ITEM is not in the array DATA.
    Else:
        Write: LOC is the location of ITEM.
    [End of If structure.]
6.  Exit.
```

-
  ```
  int SequentialSearch(int A[], int x, int n)
  {
     int i;

     for(i = 0; i <= n-1; i++)
        if(x == A[i])  return i;

     return -1;
  }
  ```

  Example: Use the linear search algorithm to search for key 25 in the array containing: -9, 11, 0, 22, 14, 3, 8, 4.

- **Best Case** Find at first place - **one comparison**
- **Worst Case** Find at *n*th place or not at all - ***n* comparisons**
- **Average Case** that this case takes - **(*n*+1)/2** comparisons

Binary search

-
  ```
  int binarySearch(int A[], int n, int x)
  {
     int left = 0; // left endpoint of interval being searched
     int right = n-1; // right endpoint of interval being searched
     int mid;

     while(left <= right)
     {
        mid = (left+right)/2;
        if(x == A[mid])
           return mid;
        else if(x < A[mid])
           right = mid-1;  // search 1st half of interval
        else left = mid+1;  // search 2nd half of interval
     }

     return -1;
  }
  ```

Worst case analysis: The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of 2, ie n = 2^k.

Then T(n) = 2 + T(n/2)

= 2 + 2+T(n/2^2)

= 2 + 2 + 2 + T(n/2^3)

...

= i*2 + T(n/2^i)

...

= k*2 + T(1)

Note that k = logn, and that T(1) = 2.

So T(n) = 2logn + 2 = O(logn) (prove that 2logn + 2 = O(logn)).

- Best case is O(1). You could get lucky and the element you want is in the middle of the list.
- Worst case is O(log2(n)) as the number of times you can divide the list up in 2 is the maximum times you'll have to compare elements in a binary search.
- Average case is also O(log2(n)). The average number of times you would compare elements in a binary search is halfway between 1 and log2(n)

Bubble sort

```
LET N = LEN(X)

FOR I = 1 TO N
    FOR J = 1 TO N
      IF X[I] > X[J] THEN
        LET T = X[I]
        LET X[I] = X[J]
        LET X[J] = T
      END IF
    NEXT J
NEXT I
```

1. Executed once
2. (blank)
3. Executed once (sets I = 1 at the beginning)
4. Executed N times (sets J = 1 once per value of I)
5. Executed N^2 times
6. Executed between 0 and N^2 times.
7. Executed between 0 and N^2 times.

8. Executed between 0 and N^2 times.

9. (doesn't actually get executed)
10. Executed N^2 times (increments J and jumps to line 5)
11. Executed N times (increments I and jumps to line 4)

So if we count the number of times a line gets executed, we have something between

$5N^2 + 2N + 2$ (worst case), and $2N^2 + 2N + 2$ (best case).

$O(N^2)$.

Insertion Sort

```
input array $A$
for $j \leftarrow 2$ to $length[A]$
    do $key \leftarrow A[j]$
       $i \leftarrow j-1$
       while $i > 0$ and $A[i] > key$
           do $A[i+1] \leftarrow A[i]$
              $i \leftarrow i-1$
       $A[i+1] \leftarrow key$
```

key

$5, \overset{\leftarrow}{2}, 4, 6, 1, 3$

$\overset{\leftarrow}{2}, 5, 4, 6, 1, 3$

$2, 5, \overset{\leftarrow}{4}, 6, 1, 3$

$2, \overset{\leftarrow}{4}, 5, 6, 1, 3$

$2, 4, 5, \overset{\leftarrow}{6}, 1, 3$

$2, 4, 5, 6, \overset{\leftarrow}{1}, 3$

$2, 4, 5, \overset{\leftarrow}{1}, 6, 3$

$2, 4, \overset{\leftarrow}{1}, 5, 6, 3$

$2, \overset{\leftarrow}{1}, 4, 5, 6, 3$

$\overset{\leftarrow}{1}, 2, 4, 5, 6, 3$

$1, 2, 4, 5, 6, \overset{\leftarrow}{3}$

$1, 2, 4, 5, \overset{\leftarrow}{3}, 6$

$1, 2, 4, \overset{\leftarrow}{3}, 5, 6$

$1, 2, \overset{\leftarrow}{3}, 4, 5, 6$

Insertion Sort

	Cost	times
input array A		
for $j \leftarrow 2$ **to** $length[A]$	C_1	n
do $key \leftarrow A[j]$	C_2	n-1
$i \leftarrow j-1$	C_3	n-1
while $i > 0$ **and** $A[i] > key$	C_4	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	C_5	$\sum_{j=2}^{n} t_j - 1$
$i \leftarrow i-1$	C_6	$\sum_{j=2}^{n} t_j - 1$
$A[i+1] \leftarrow key$	C_7	n-1

- t_j no of shifts / comparison
- For best case $t_j = 1$ so, O(n)
- Worst case $O(n^2)$