Introduction

Keun-Liang Sue
Information Management
NCU

Introduction

- What is an Algorithm?
  - Definition 1
    - An *algorithm* is a sequence of unambiguous instructions for solving a problem
  - Definition 2
    - for obtaining a required output for any legitimate input in a finite amount of time.

Notation of Algorithm

- Features of Algorithm
  - Each step must be non-ambiguous
  - Specified range of inputs
  - Same algorithm can be represented in different ways
  - Several algorithms may exist for one problem
  - Algorithms can resolve problem with different speed

Notation of Algorithm

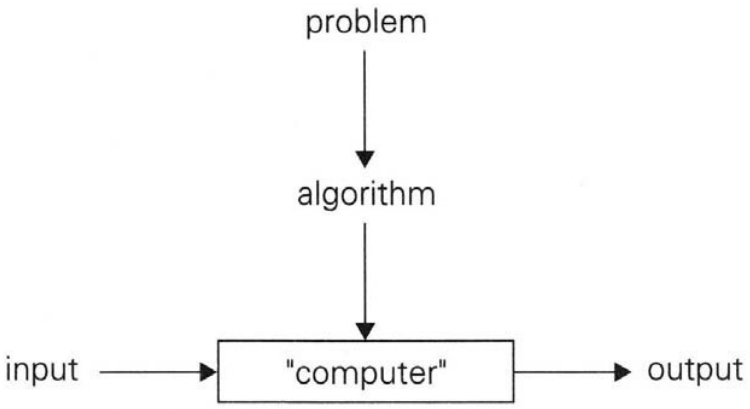


FIGURE 1.1 Notion of algorithm

Notation of Algorithm

- Problem: find gcd(m,n)
  - Solution 1:
    - Step 1 assign the value of min{m,n} to t
    - Step 2 divide m by t
    - If the remainder of this division is 0
    - goto step 3; otherwise, go to step4
    - Step 3 divide n by t
    - If the remainder of this division is 0
    - return t and stop; otherwise, go to step4
    - Step 4 Decrease t by 1. go to step2

Notation of Algorithm

- Solution 2:
- Euclid's algorithm

$$gcd(m,n)=gcd(n, m \bmod n)$$

e.g.

$$gcd(60,24)=gcd(24,12)=gcd(12,0)=12$$

Notation of Algorithm

ALGORITHM *Euclid*(m, n)

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```


Notation of Algorithm

- Solution 3:
  - Step 1 Find the prime factors of m
  - Step 2 Find the prime factors of n
  - Step 3 identify all common factors in step 1 & step2
  - Step 4 return the product of all common factors

e.g. 60=2*2*3*5
24=2*2*2*3
gcd(60,24)=2*2*3=12

Notation of Algorithm

- Sieve of Eratosthenes
  - Generate consecutive primes not exceeding n
    - n is a given integer
  - Idea
    - Initial a list of integers from 2 to n
    - 1 iteration: eliminate 2,4,6,8,and so on
    - 2 iteration: eliminate 3,6,9,12,and so on
    - Continues till no more numbers can be eliminated

Notation of Algorithm

- When to stop?
  - What is the largest number p whose multiples can still remain on the list?
    - Suppose integer p is being considered
    - We should first consider p*p
      - Because 2p,…., (p-1)p have been eliminated
    - So, if p*p>n, then p's multiples can not remain on the list
  - Hence,

$$p = \lfloor \sqrt{n} \rfloor$$

Notation of Algorithm

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
2	3		5		7		9		11		13		15		17		19		21		23		25	
2	3		5		7				11		13				17		19				23		25	
2	3		5		7				11		13				17		19				23			

An example for n=25

Notation of Algorithm

ALGORITHM *Sieve*(n)

```
//Implements the sieve of Eratosthenes
//Input: A positive integer n ≥ 2
//Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0    //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0    //mark an element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

Fundamentals of Algorithmic Problem Solving

- Steps to design & analyze algorithm
  - Understand the problem
  - Ascertain capability of computational device
    - Sequential algorithm
    - Parallel algorithm
  - Exact or approximate problem solving
    - Exact algorithm
    - Approximation algorithm
      - Reason 1: can not be resolved exactly, e.g. square root
      - Reason 2: exact algorithm is unacceptably slow

Fundamentals of Algorithmic Problem Solving

- Appropriate data structure
  - Algorithm+ data structures = Programming
- Algorithm design techniques
  - Check the book's table of contents
- Methods of specifying an algorithm
  - Pseudocode
    - A mixture of natural language and programming language
  - Flowchart

Fundamentals of Algorithmic Problem Solving

- Prove an algorithm's correctness
  - A common technique
    - Mathematical induction
  - Trace algorithm for some inputs
    - cannot prove correctness
  - Show algorithm's incorrectness
    - one instance of input, which fails the algorithm

Fundamentals of Algorithmic Problem Solving

- Analyze an algorithm
  - Time efficiency
    - How fast the algorithm runs
  - Space efficiency
    - How much memory the algorithm needs
  - Simplicity
    - Easy to understand
    - Simple algorithm maybe more efficient
      - but not always

Fundamentals of Algorithmic Problem Solving

- Generality
  - Two issues here
    - Generality of the problem the algorithm solves
    - The range of inputs it accepts
  - First issue
    - Sometimes, design a more general algorithm is easier
      - e.g. whether two integers are relative prime
      - sol: compute the greatest common divisor
    - Sometimes, it is unnecessary, difficult or impossible
      - Sort a list of number to find its median
      - Find the root of a quadratic equation by handling polynomials of arbitrary degrees

Fundamentals of Algorithmic Problem Solving

  - Second issue
    - Naturally, not exclude 1 as possible inputs for gcd algorithm
    - Normally, not accept complex coefficients for polynomials
- Three factors to evaluate algorithm
  - Efficiency, Simplicity and Generality
  - If not satisfied, redesign it
  - Even satisfied, worth searching for other solution
- Principles
  - Not expect to get the best algorithm on the first try
  - Try to fine-tune the algorithm you already have

Fundamentals of Algorithmic Problem Solving

- Keep in mind!!

> "*A designer knows he has arrived at perfection*
> *not when there is no longer anything to add,*
> *but when there is no longer anything to take away*"

Fundamentals of Algorithmic Problem Solving

- Coding an algorithm
  - Peril
    - Correct algorithm → incorrect or inefficient program
  - Valid of program
    - Program is validated by testing
  - Efficiency
    - Modern compiler provides code optimization
    - Computing loop's invariant outside the loop
    - Collect common sub-expressions
    - Replace expensive operation by cheap one

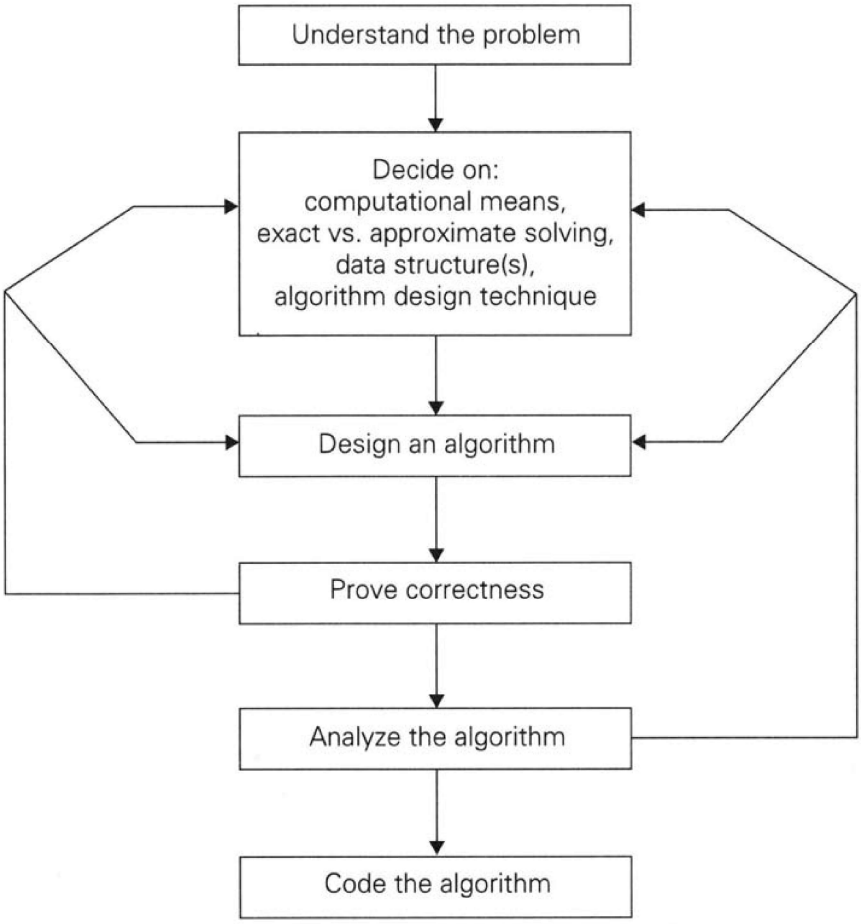


FIGURE 1.2 Algorithm design and analysis process

Important problem types

- The most important problem types
  - Sorting
  - Searching
  - String processing
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems

Important problem types

- Sorting
  - Rearrange items in ascending order
  - Why sorting?
    - Make many questions about the list easier
      e.g. searching
  - Dozens of sorting algorithms are developed
  - Two property of sorting algorithms
    - Stable ⇔ it preserves the relative order of any two equal items in the input
    - In place ⇔ it does not require extra memory

Important problem types

- Searching
  - Finding a given value in a given set
    - The given value is called **search key**
  - Plenty of searching algorithms
    - Sequential search
    - Binary search
  - No single algorithm can fit all situations best
    - Some are faster but applicable only to sorted array

Important problem types

- String processing
  - String
    - A sequence of characters
  - Type of string
    - Text string
      - comprise letters, numbers, and special characters
    - Bit string
      - Zeros and ones
  - String matching
    - Searching for a given word in a text

Important problem types

- Graph problems
  - Graph traversal algorithms
  - Shortest path algorithms
  - Topological sorting
  - The traveling salesman problem
    - Find shortest tour through n cities
    - Visit every city exactly once
  - Graph coloring problem
    - No two adjacent vertices are the same color

Important problem types

- Combinational problems
  - Find a combinational object
    - e.g. permutation, combination, or a subset
  - Satisfy certain constraints
    - e.g. maximize a value or minimize a cost
  - Such problem are difficult
    - Combinational objects grows fast with a problem's size
    - No known algorithm is good enough for most such problems
      - In an acceptable amount of time
    - Most computer scientist believe that such algorithm don't exist

Important problem types

- Geometric Problems
  - Deal with geometric objects
    - e.g. Points, lines, and polygons
  - Two classic problems
    - The closest-pair
      - Find the closest pair among n points in a plane
    - The convex hull problem
      - Find the smallest convex polygon
      - The polygon must include all points of a given set

Important problem types

- Numerical problems
  - Types
    - Solve equations
    - Solve system equations
    - Compute definite integrals
    - Evaluate functions
  - Difficulty
    - Typically require manipulating real numbers
    - Real number is represented in a computer only approximately

Fundamental Data Structures

- Linear data structure
  - Array
    - A sequence of n items of the same data type
    - Stored continuously in computer memory
    - Accessible by specifying a value of *index*

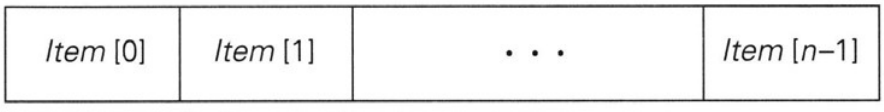


FIGURE 1.3 Array of n elements

Fundamental Data Structures

- Linked list
  - A sequence of zero or more nodes
  - Each node contains
    - Data
    - Pointer
- Singly linked list

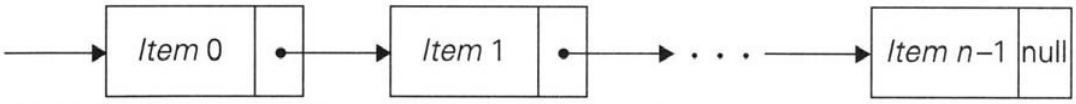


FIGURE 1.4 Singly linked list of n elements

Fundamental Data Structures

- Doubly linked list

Introduction

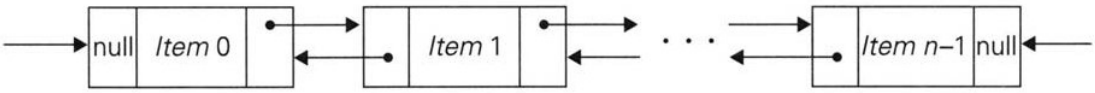


FIGURE 1.5 Doubly linked list of n elements

- Terms
  - Stack
    - **Top**
    - **Push and pop**
  - Queue
    - **front and rear**
    - **dequeue and enqueue**

Fundamental Data Structures

- Graph
  - Define a graph G=<V,E>
    - V: a collection of points called vertices
    - E: line segments
  - Types of graph
    - Undirected graph
    - Directed graph

Fundamental Data Structures

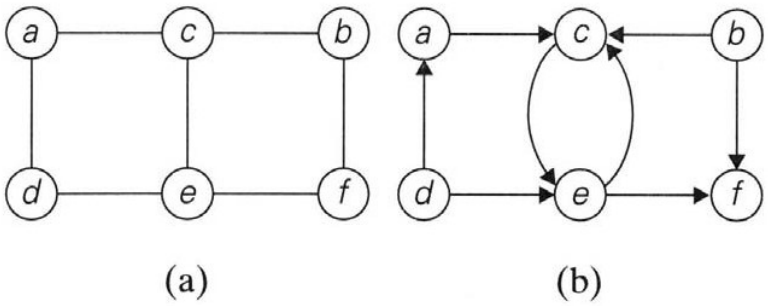


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

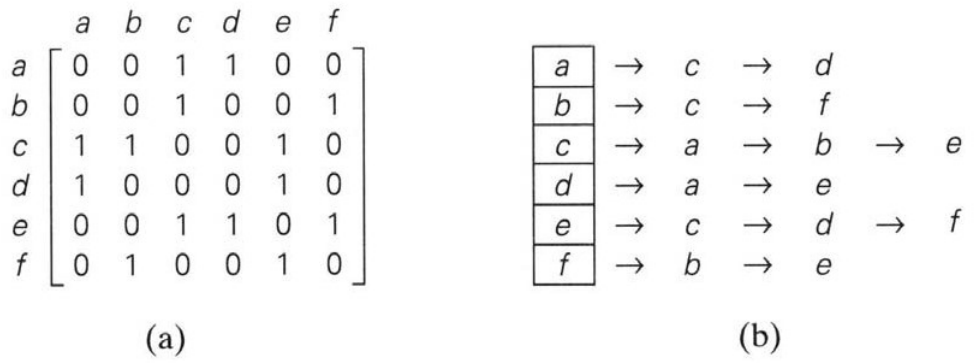
Fundamental Data Structures

- For Figure 1.6 (a)
  - V={a,b,c,d,e,f}
  - E={(a,c), (a,d), (b,c), (b,f), (c,e), (d,e), (e,f)}
- For Figure 1.6 (b)
  - V={a,b,c,d,e,f}
  - E={(a,c), (b,c), (b,f), (c,e), (d,a), (d,e), (e,c), (e,f)}

Fundamental Data Structures

- Properties
  - No loops ⇔ 0<= |E| <=|V|(|V|-1)/2
  - Complete ⇔ every pair of vertices connected by an edge
- Graph representation
  - Adjacency matrix
  - Adjacency linked lists

Fundamental Data Structures



	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

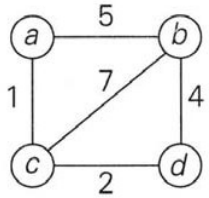
a	→ c	→ d			
b	→ c	→ f			
c	→ a	→ b	→ e		
d	→ a	→ e			
e	→ c	→ d	→ f		
f	→ b	→ e			

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency linked lists of the graph of Figure 1.6a

Fundamental Data Structures

- Weighted graphs
  - a graph with numbers assigned to its edges



(a)

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

(b)

a	→ b, 5 → c, 1
b	→ a, 5 → c, 7 → d, 4
c	→ a, 1 → b, 7 → d, 2
d	→ b, 4 → c, 2

(c)

FIGURE 1.8 (a) Weighted graph. (b) Its adjacency matrix. (c) Its adjacency linked lists.

Fundamental Data Structures

- Connected graph
  - Each pair of its vertices u and v has a path from u to v
- Connected component
  - Maximal subgraph of a given graph
    - A subgraph of a given graph G=<V,E> is a graph G'=<V',E'> such that V'⊆V and E' ⊆E
- Acyclic
  - A graph with no cycle
- Cycle
  - A simple path that starts and ends at the same vertex

Fundamental Data Structures

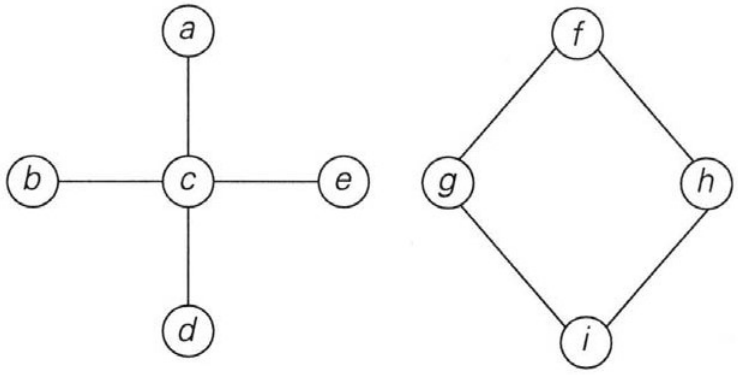


FIGURE 1.9 Graph that is not connected

Fundamental Data Structures

- Simple path
  - All edges of a path are distinct
- Path (from u to v)
  - A sequence of vertices that start from u and end with v
- Length of a path
  - The total number of vertices in a path minus one

Fundamental Data Structures

- Tree
  - Definition
    - A connected acyclic graph
  - Forest
    - An acyclic graph but is not necessarily connected
  - Important property
    - |E|=|V|-1

Fundamental Data Structures

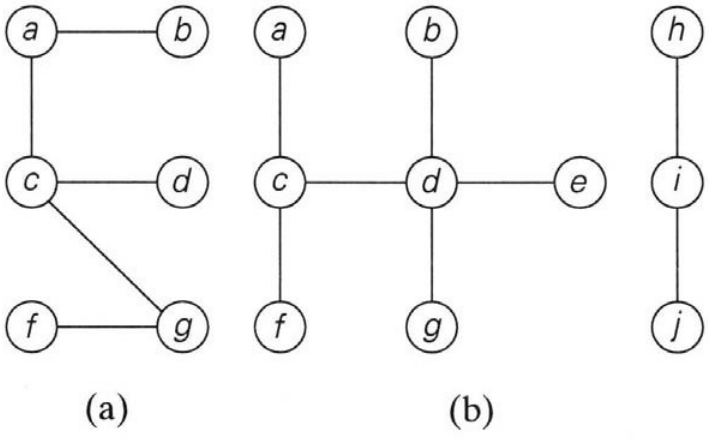


FIGURE 1.10 (a) Tree. (b) Forest.

Fundamental Data Structures

- Rooted tree
  - arbitrary vertex in a free tree can be as root

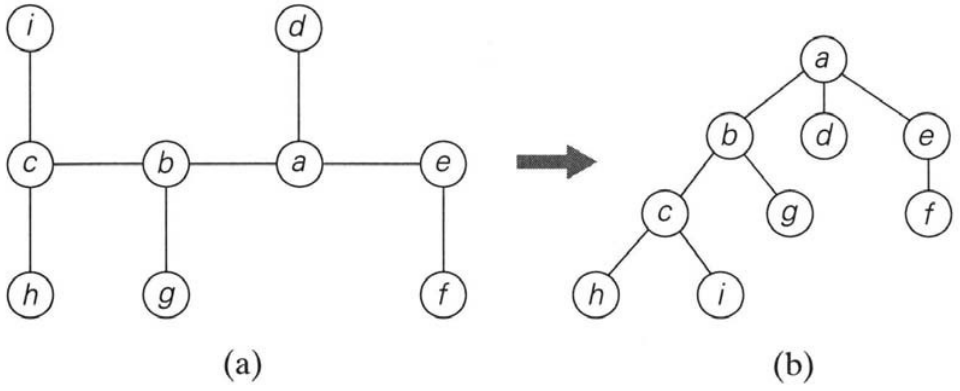


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Fundamental Data Structures

- Terms
  - vertex *v*'s **ancestor**
    - Vertices on the simple path from root to v
    - v is called **descendant** of the ancestor
  - *u* is *v*'s **parent**
    - (*u*,*v*) is the last edge of the path from root to vertex *v*
    - At the same time, *v* is *u*'s **child**
  - **Siblings**
    - Vertices that have the same parent
  - **Leaf**
    - A vertex with no child

Fundamental Data Structures

- **Subtree**
  - A vertex *v* with all its descendants
- **Depth** of vertex *v*
  - Length of the simple path from root to *v*
- **Height** of a tree
  - Length of the longest simple path from root to a leaf

Fundamental Data Structures

- **Ordered tree**
  - A rooted tree
  - All the children of each vertex are ordered
- **Binary tree**
  - A ordered tree
  - Every vertex has no more than two children
  - Each child is either **left child** or **right child** of its parent

Fundamental Data Structures

- Binary search tree
  - All vertices are assigned a number
  - Conditions
    - The number of each vertex is larger than all numbers in its left subtree
    - The number of each vertex is smaller than all numbers in its right subtree
  - Property

$$\lfloor \log_2 n \rfloor \le h \le n-1$$

Fundamental Data Structures

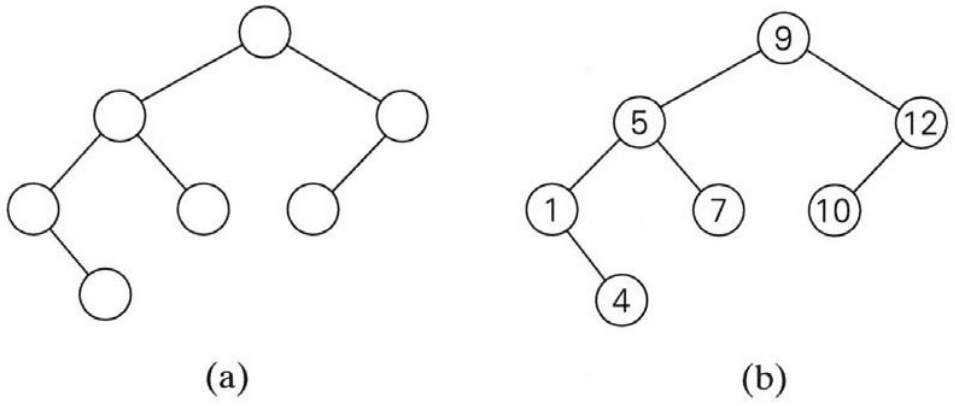


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

Fundamental Data Structures

- Implementation for binary search tree

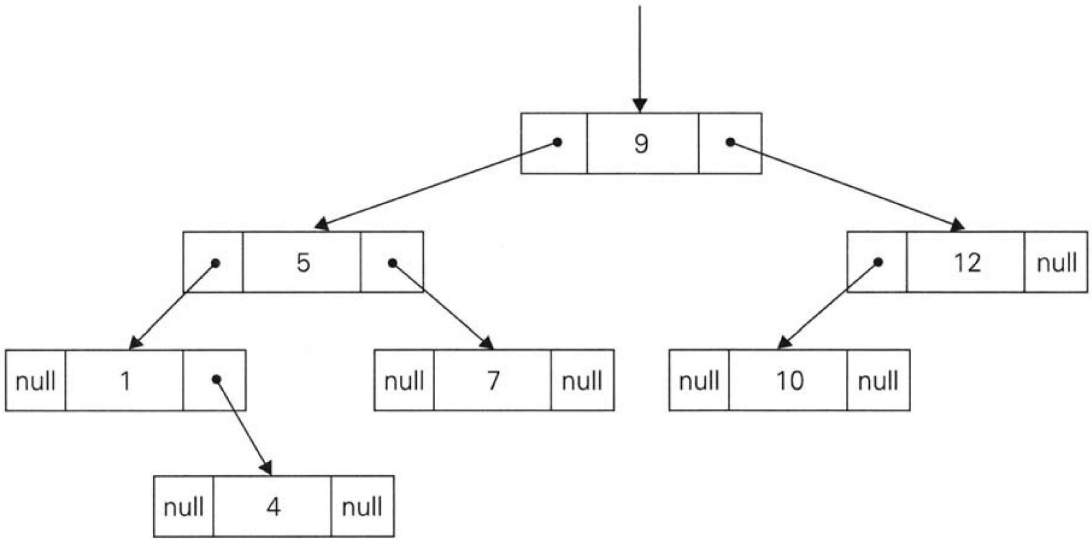


FIGURE 1.13 Standard implementation of the binary search tree of Figure 1.12b.

Fundamental Data Structures

- First child-next sibling representation
  - Purpose
    - To present a multi-way tree into binary tree
  - Definition
    - Each node with only two pointer
      - left pointer points to the first child
      - right pointer points to its next sibling
    - All siblings of a vertex are linked as a linked list
      - The first element is pointed by the left pointer of its parent

Fundamental Data Structures

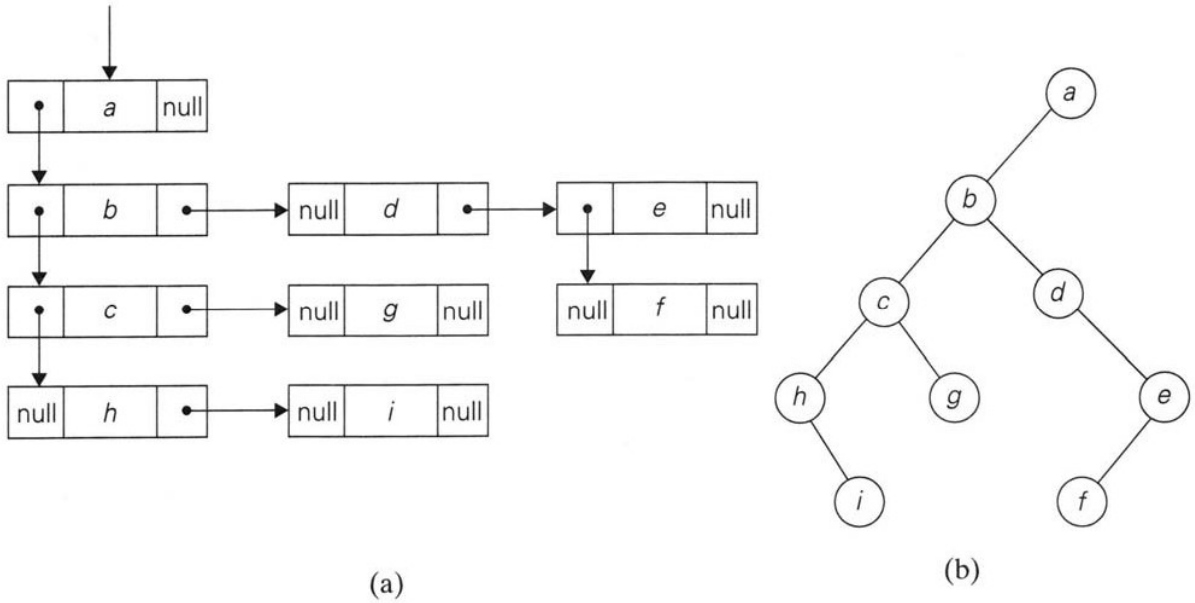


FIGURE 1.14 (a) First child–next sibling representation of the graph of Figure 1.11b. (b) Its binary tree representation.

Fundamental Data Structures

- Set
  - An unordered collection of distinct items
  - The items are called its elements
  - e.g. S={2,3,5,7}
  - e.g. S={n: n is a prime number and n<10}
  - Important operations
    - Union of two sets
    - Intersection of two sets

Fundamental Data Structures

- Set can be implemented in two ways
  - Bit vector
  - List structure
- Bit vector
  - Only for subsets of some set U, called universal set
    - Set U={1,2,3,4,5,6,7,8,9}
    - Subset S={2,3,5,7}
    - S can be represented by bit string 011010100
  - Merit
    - Implement the standard set operations fast

Fundamental Data Structures

- List structure
  - Feasible only for finite sets
  - Two distinction between sets and lists
    - A set cannot contain identical elements; a list can
    - A set is an unordered set; a list is exactly the opposite
  - Multiset
    - A set allows containing identical elements

Fundamental Data Structures

- Dictionary
  - Data structure implement 3 operations for a set
    - Searching a given items
    - Adding a new items
    - Deleting an item
  - Tradeoff
    - Efficiency of searching
    - Efficiency of the two other operations

Fundamental Data Structures

- Ways of dictionary implementation
  - Array
  - Linked list
  - Hashing
  - Balance searching tree
- Abstract data type (ADT)
  - Definition
    - A set of abstract objects representing data items
    - A collection of operations that can be performed on them
    - e.g. queue, stack, and dictionary