Introduction : Data Structures and Algorithms

성균관대학교 컴퓨터공학과
데이터베이스 연구실
김응모

Algorithm

- Definition of Algorithm

  Algorithm is a step-by-step procedure for solving a problem in a finite amount of time. It consists of :
  - Instructions
  - Input data
  - Output data

- Classes of Algorithms : Computer Science
  - Searching algorithms
  - Sorting algorithms
  - Tree algorithms
  - Graph algorithms
  - Hashing algorithms
  - Parsing algorithms
  - . . . .

Example : Algorithm

- Problem : Compute GCD
- Algorithm : Euclidean Algorithm
  - Input : Integers ($L \geq S$)
  - Output : GCD of L, S

```
GCD (int L, S)
    int  R;
    while (S > 0)
     {
        R = L % S;
        L = S;
        S = R;
     }
    return (L)
```

Example : Algorithm

- Problem : Compute X^N
- Algorithm : Power Algorithm
  - Input : Integers X, N
  - Output : X^N

```
POWER (int X, int N)
    if (N == 0) return 1;
    else {
       factor = POWER(X, N/2)
       if  N%2 == 0  return factor*factor
       else  return factor*factor*X
          }
```

Example : Algorithm

- Problem : Sorting integers
- Algorithm : Selection Sort
  - Input : n unsorted integers
  - Output : n sorted integers

```
Selection Sort (int list[n])
  for (i = 0; i < n; i++)
   {
    1) Examine list[i] to list[n-1]
    2) Find the smallest integer
    3) Let it store list[min];
    4) Swap list[i] and list[min]
    }
```

Performance Analysis

- **Performance Analysis**

  - Space Complexity
    - the amount of memory space used by the algorithm

  - Time Complexity
    - the amount of computing time used by the algorithm

- Typically, the more (less) space, the less (more) time.

  Thus, sometimes we need to trade off space vs. time.

Space Complexity

- Find a total sum of n numbers. Space = ?

```
SUM (float list[ ], int n)
  sum = 0;
   int i;
   for (i = 0; i < n; i++)
       sum = sum + list[i];
   return sum;
```

- Addition of two n x n matrices. Space = ?

- Representing an n x n sparse matrix. Space = ?

Time Complexity

- Time Complexity Criteria?
  - Theoretical Speed
    - <u>number of operations</u> by performed by the algorithm.
  - Practical Speed
    - the <u>execution time</u> performed by the algorithm.

```
sum = 0;
 for (i = 0;  i < 1000000; i++)
        sum = sum + i ;
```

- What is time complexity?
  - Theoretical Speed : 10^6 (additions)
  - Practical Speed : 10 msec. (Assume: Pentium III, 256M memory)

- Which criteria is <u>more reasonable</u>?
  - "<u>Theoretical</u>" speed gives <u>better</u> criteria. Why?

Time Complexity

- Linear

```
for (i=1; i<=n; i++)
    { application code }
```

Time = ?

```
for (i=1; i<=n; i+=2)
    { application code }
```

Time = ?

- Logarithmic

```
for (i=1; i<=n; i*= 2)
 { application code }
```

Time = ?

```
for (i=n; i>=1; i/=2)
{ application code }
```

Time = ?

- Quadratic

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
     { application code }
```

Time = ?

- Dependent quadratic

```
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
     { application code }
```

Time = ?

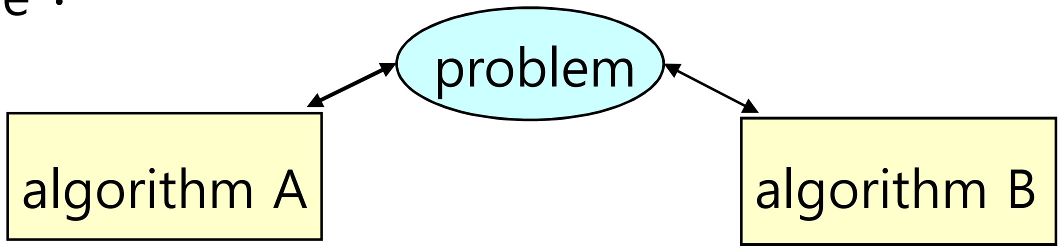
- Linear logarithmic

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j*=2)
     { application code }
```

Time = ?

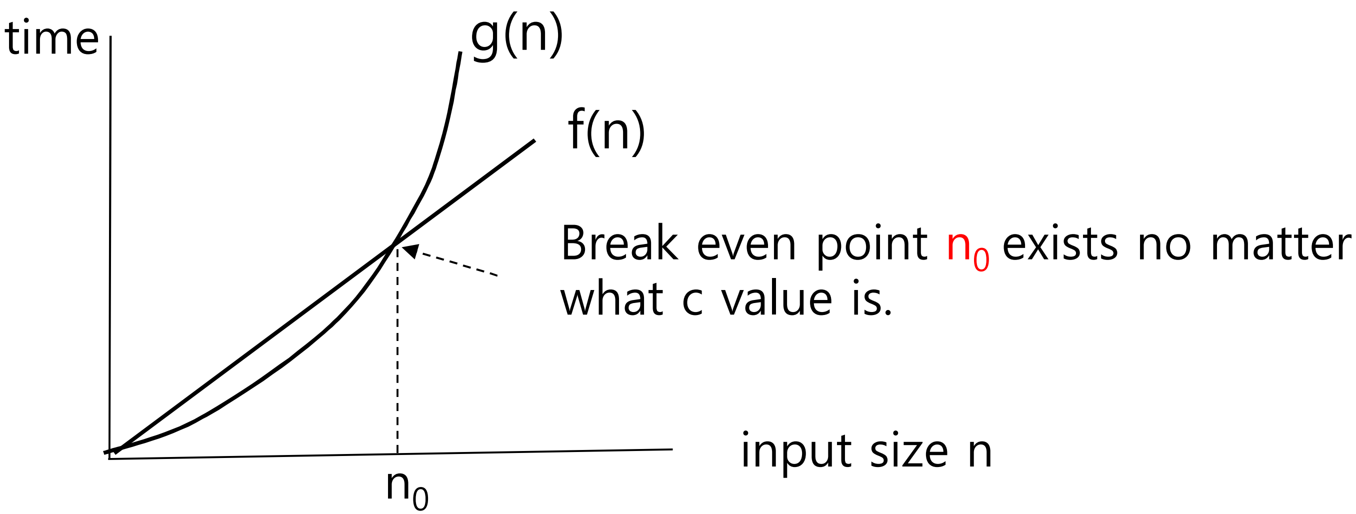
Time Performances : Big Oh(O)

- Which one is faster?
  Example :



$f(n) = 10n$ $g(n) = 1/2\, n^2$

- Given f(n) and g(n), we say that f(n) = **O**(g(n)) if there are positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.



- Note : c is implementation factor depending on H/W and S/W environmental variants. If $f(n) = a_k n^k + \dots + a_1 n + a_0$, then $f(n) = O(n^k)$.

Class of Time Complexities

- Polynomial Time
  - $O(1)$: Constant
  - $O(\log_2 n)$
  - $O(n)$
  - $O(n \cdot \log_2 n)$
  - $O(n^2)$
  - $O(n^3)$

  . . . . . .

  - $O(n^k)$

- Exponential Time
  - $O(2^n)$
  - $O(n!)$
  - $O(n^n)$

Class of Time Complexities

- Which one is bigger?
  - $O(n^k)$ vs $O(2^n)$
  - $O(n^k)$: Easy, Reasonable, Mostly solved within by $O(n^3)$
  - $O(2^n)$: Hard, Cannot be solved in practice.

- Ordering of complexities
  - $O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

- Which are meaning of these comparisons?
  - $O(n)$ vs $O(1)$
  - $O(n)$ vs $O(\log_2 n)$
  - $O(n^2)$ vs $O(n \cdot \log_2 n)$
  - $O(n^3)$ vs $O(n^2)$

Growth of Function Values

Seconds	Equivalent
10^2	1.7 mins
10^3	17 mins
10^4	2.8 hrs
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades

Powers of 2

$2^{10} = 10^3$

$2^{20} = 10^6$

$2^{30} = 10^9$

. . . .

Logarithmic

$\log_2 10^3 = 10$

$\log_2 10^6 = 20$

$\log_2 10^9 = 30$

. . . .

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^7$yr	$3.17*10^{43}$yr	

Example : Sorting

(35 38 70 75 12 25 18 54 65 90 86)

$\Downarrow$ sorting

(12 18 25 35 38 54 65 70 75 86 90) : sorted

- Classic Problem in Computer Science : Still many researches!
- Sorting is essential for solving many problems efficiently.
- 25% ~ 50 of total time for solving problem is spent for sorting.
- Performance Criteria : Number of Comparisons
- Selection, Bubble, Insertion, Heap, Shell, Quick, Merge
- $O(n^2)$ or $O(n\log_2 n)$

Sorting

list :

0	1	2	3	4	·	·	·	
26	5	37	1	61	11	59	15	48

- How many comparison operations? (Input size n)
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort

Comparison: Sorting Methods

Method	Average	Worst	Extra Space
Selection	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n^2)$	$O(n^2)$	$O(1)$
Quick	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$
Merge	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$

- Insertion sort is the best for small n.
- Quick sort is the best in average case.
- Merge sort is the best in worst case, but we need extra space.
- We usually combine Insertion, Quick, and Merge.

Sorting : Performance

- Algorithms by Sedgewick
  - ✓ PC : 10^8 comparisons/sec
  - ✓ Super : 10^{12} comparisons/sec

Insertion Sort ($O(n^2)$)

	$n = 10^3$	$n = 10^6$	$n = 10^9$
PC	instant	2.8hrs	317yrs
Super	instant	1sec	1.7wks

Merge Sort ($O(n\log_2 n)$)

	$n = 10^3$	$n = 10^6$	$n = 10^9$
PC	instant	1sec	18min
Super	instant	instant	instant

Quick Sort ($O(n\log_2 n)$)

	$n = 10^3$	$n = 10^6$	$n = 10^9$
PC	instant	0.3sec	6min
Super	instant	instant	instant

- Good algorithms are better than supercomputers.
- Good algorithms are better than good ones.

Practical Complexities

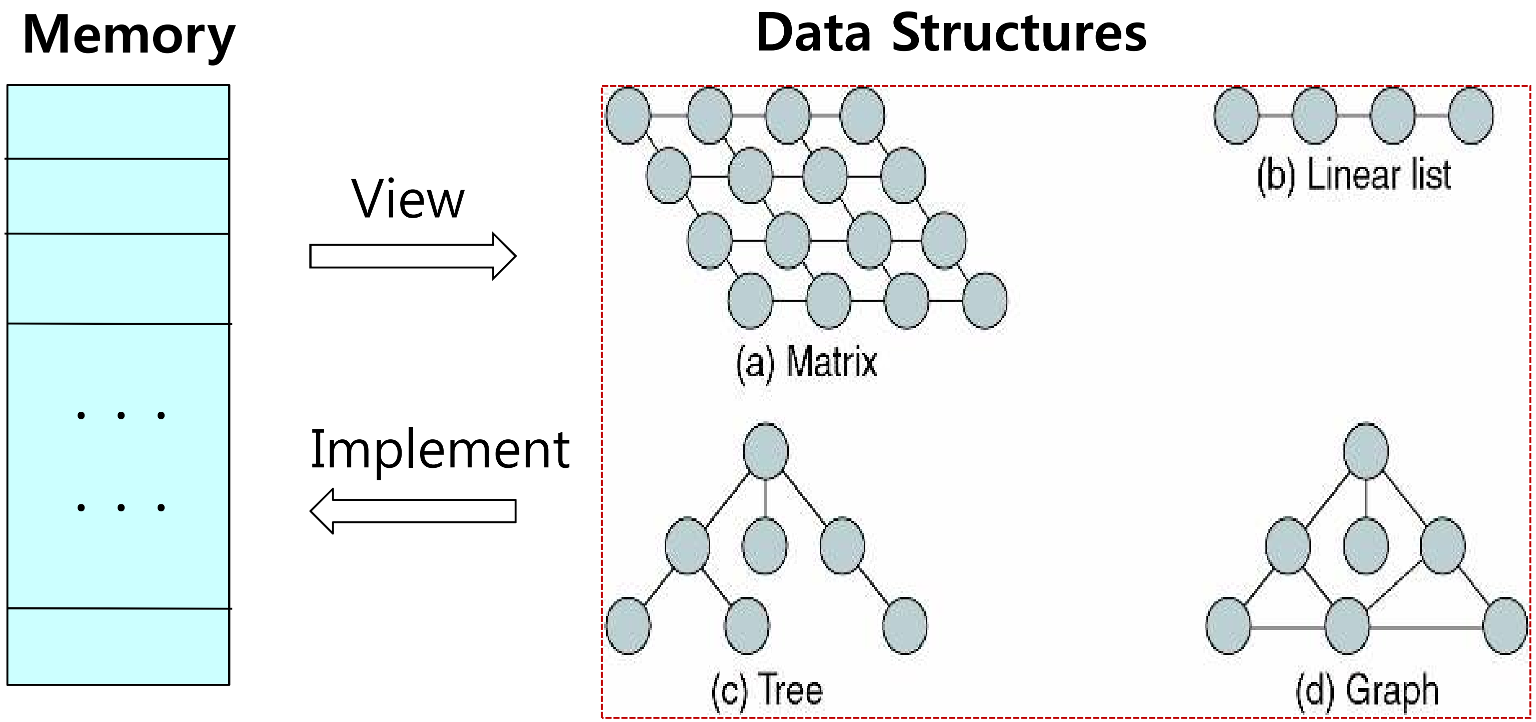
- Sequential Search : $O(n)$
- Binary Search : $O(\log_2 n)$
- External (B-Tree) Search : $O(\log_f n)$, $f \approx 133$
- Selection, Bubble, Insertion Sort : $O(n^2)$
- Quick, Heap, Merge Sort : $O(n \cdot \log_2 n)$
- Euler Cycle : $O(n^2)$
- Minimal Spanning Tree : $O(n \cdot \log_2 n)$
- Shortest Paths : $O(n^2)$
- Matrix Addition : $O(n^2)$
- Matrix Multiplication : $O(n^3)$ or $O(n^{2.81})$
- Satisfiability Problem : $O(2^n)$
- Hamiltonian Cycle : $O(n!)$
- Graph Coloring : $O(n^n)$
- . . . .

Data Structures

- How do we store the following data in memory efficiently?
  - Matrix Operations
  - Mazing Problem
  - Bank Customers Service
  - UNIX File Directory
  - Baseball Tournament
  - Airline Flights Connection
  - Given n integers, find an arbitrary number?
  - Given n integers, find a maximum number?
  - Courses Road Map
  - . . . .

Data Structures

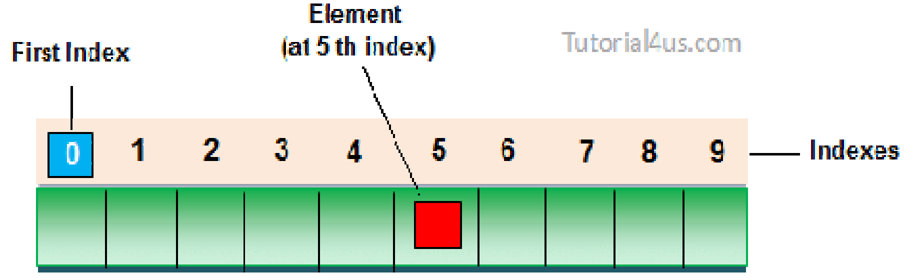
- Data Structure
  - How do we store data in a (mostly) memory?
  - We need to specify data structure to organize them.
  - Choice of different data structures gives us different algorithms.
  - Good data structures are essential for efficient algorithms.

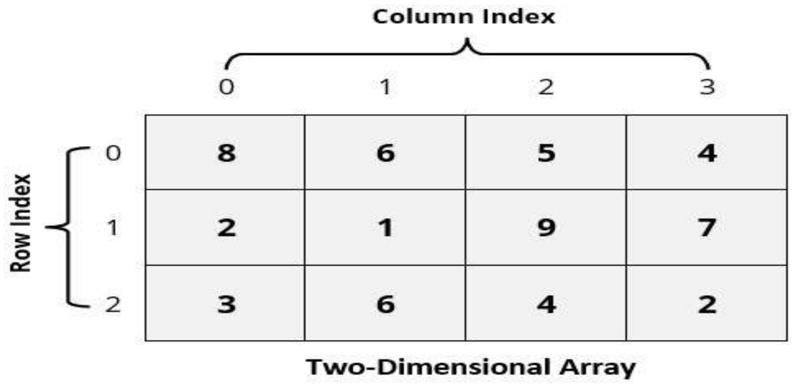


Array/Linked List

Array

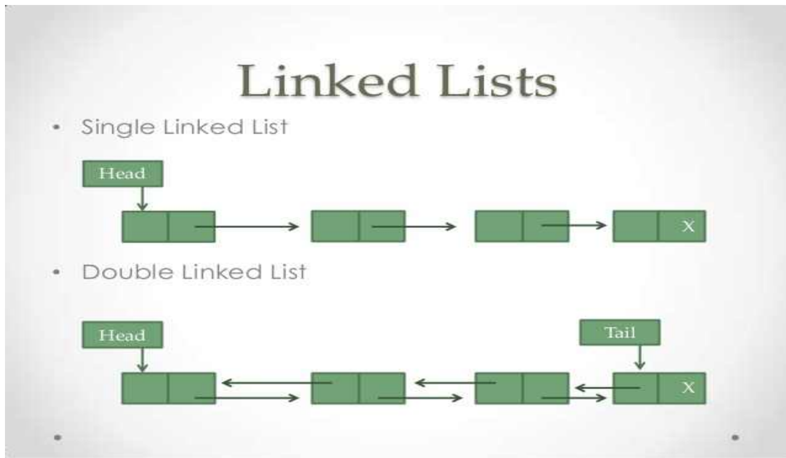
- A linear list with (index, value)
- Consecutive memory locations
- Static Allocation : Compile Time
- Reads/writes : O(1)
- Insert/deletes : O(n)





Linked List

- A linear list with pointers(links)
- Non-Consecutive memory locations
- Dynamic Allocation : Run Time
- Reads/writes : O(1)
- Insert/deletes : O(1)



Two Approaches : Arrays vs Linked Lists

- Lists (1 dimension) : Searching, Sorting, . . .
- Matrix Operations
- Binary Trees : Especially, Complete binary tree
- Trees
- Heaps
- Graphs : Roads, Maps, SNS Networks, . . .

Array : Sparse Matrix

- Sparse Matrix : Most elements are 0's; Real values are rare.

  Examples : Airline Flights, Web Pages Matrix, . .

$$
\begin{array}{c}
\begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\
\begin{array}{c} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array}
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

- (Conventional) 2-D array
  - A[m, n] (m : #rows, n : #columns)
  - Memory Usage : t / (m * n) (t : #non-zeros)
  - Very inefficient!

Array : Sparse Matrix

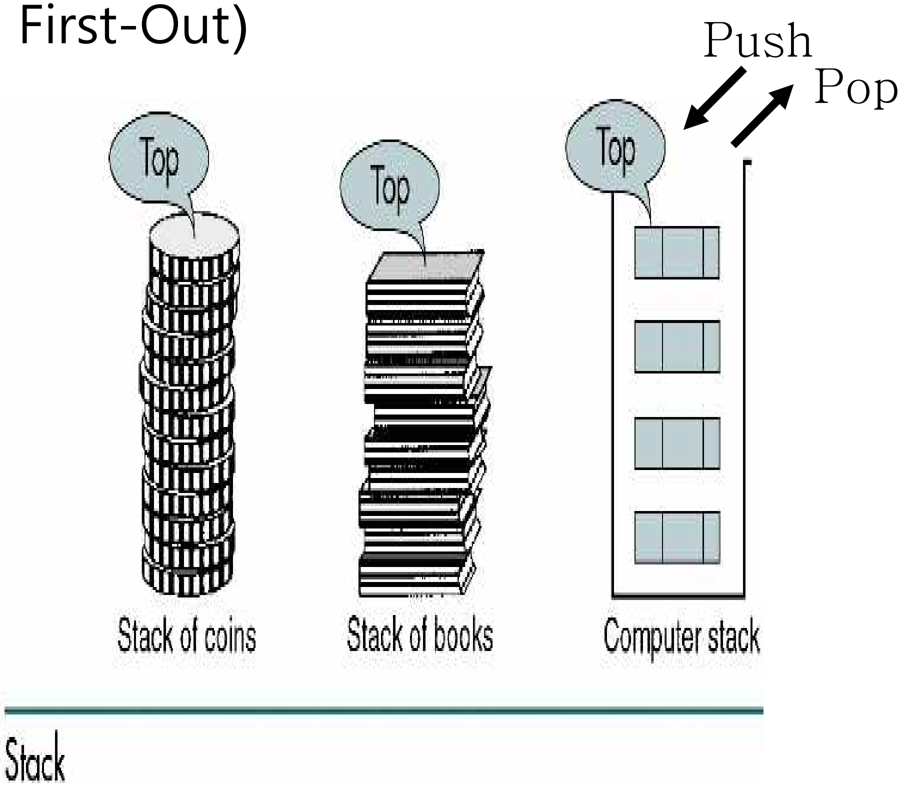
	row	col	value
[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

- Compressed 2-D array
  - Stores only non-zero values; By raw-major order;
  - <row position, column position, non-zero value>
  - Memory Usage : $\propto t$ (independent of matrix size)
  - Efficient!

Stack

Stack

- A linear List with top and bottom.
- All insertions and deletions occur at top.
- Push(insert) and Pop(delete)
- Top values grow and shrink.
- All items except top are invisible.
- LIFO (Last-In First-Out)



Implementing Stack

- Array vs Linked Lists
  - Create-Stack
  - Push
  - Pop
  - Stack-Full
  - Stack-Empty
- Implementation is easy, Very efficient : O(1)
- What about multiple stacks?

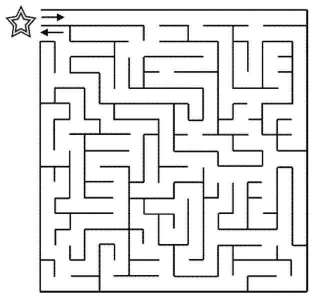
Applications : Stack

- Evaluation of Arithmetic Expressions
  - 3+2, 3+5*2, 6/2-3+4*2, (2/(8%4+(3*5))*(7-3)), . . .

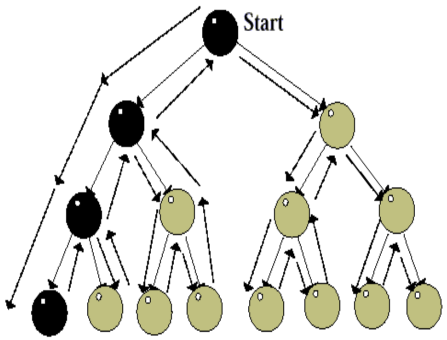
- Parsing (Pattern Matching)
  - $a^n b^n$, $a^{2n} b^n$, palindromes, . .

- Function Calls/Returns
  - Call function A, call B, Call C; How return?

- Maze Problem



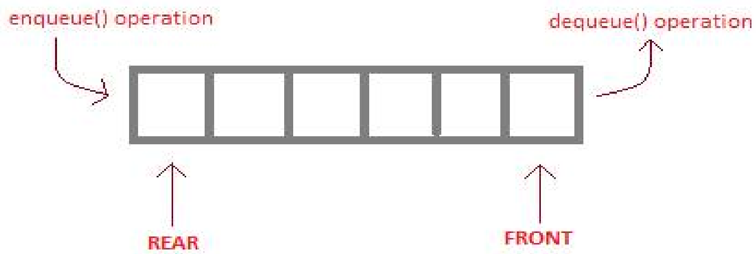
- Depth First Search



Queue

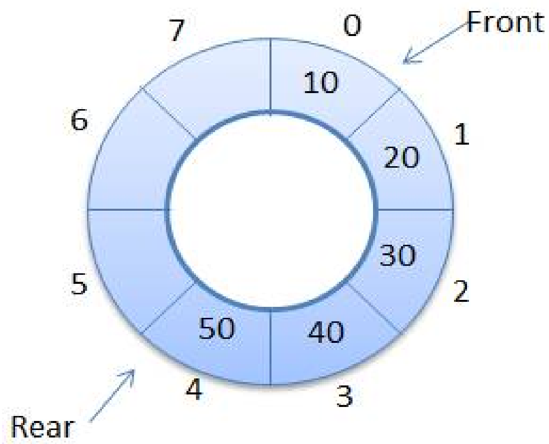
Queue

- A linear List with front and rear.
- All insertions (enqueue) : rear,
  All deletions (dequeue) : front
- All items except front and rear are invisible.
- FIFO (First-In First-Out)



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE



Implementing Queue

- Array vs Linked Lists
  - Create-Queue
  - Insert
  - Delete
  - Queue-Full
  - Queue-Empty
- For array, implementation is not so easy : O(n)

  → Use Circular Queue : O(1)
- What about multiple queues?

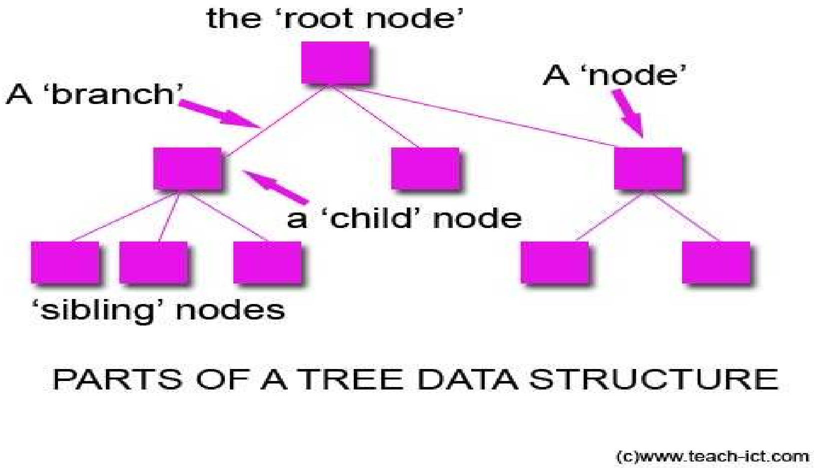
Applications : Queue

- Key board Data Buffers
- Job Processing (printer, CPU processor) : FCFS
- Breadth First Search
- Categorizing data into groups
- Waiting times of customers at call center
- Deciding # of cashiers at super market
- Traffic Analysis

Trees

Tree

- A non-linear list with nodes
- A special node : Root
- Parent : Child = 1 : m relationship
- Leaf node : Node with no child
- Connected
- Acyclic Graph



Binary Tree

- Every node has at most 2 children.
  (0, 1, or 2)
- Order of children is important.
- Connected
- Acyclic Graph

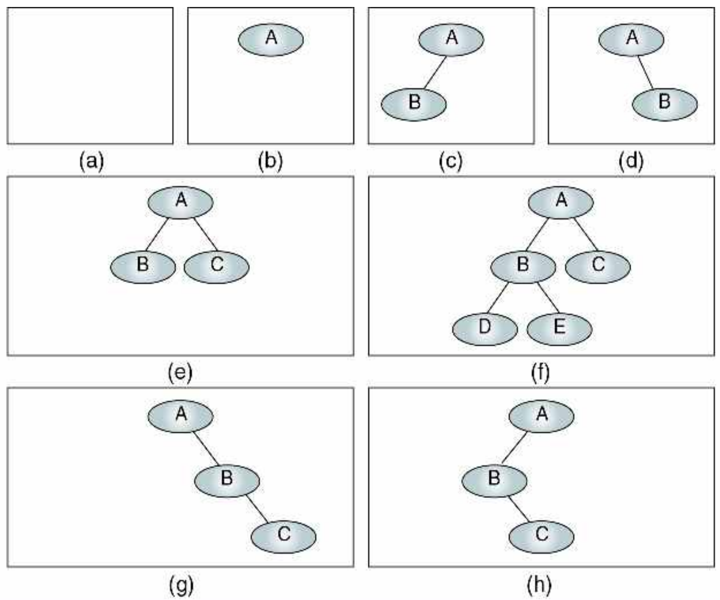
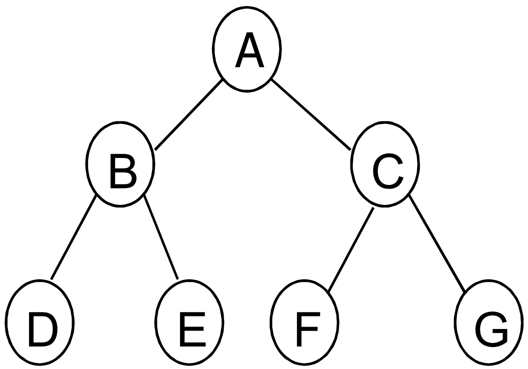


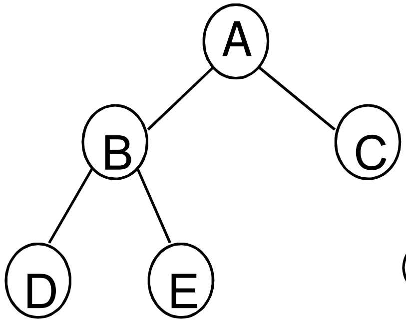
FIGURE 6-6 Collection of Binary Trees

Types of Binary Trees

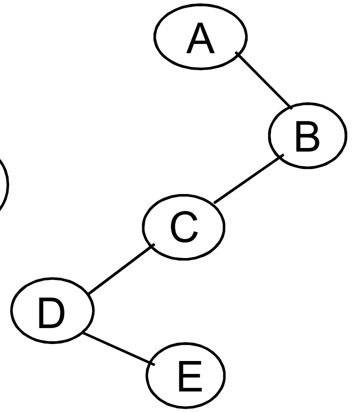
Full



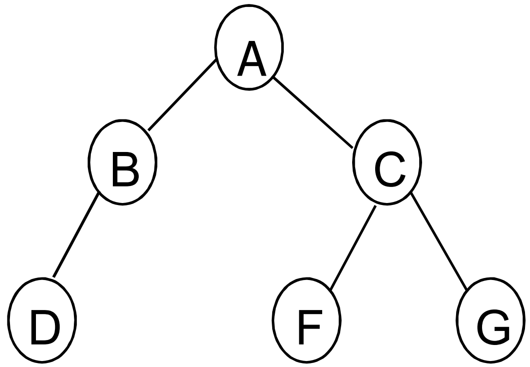
Complete



Skewed



General

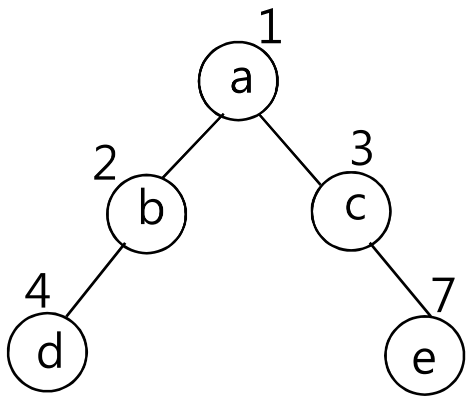


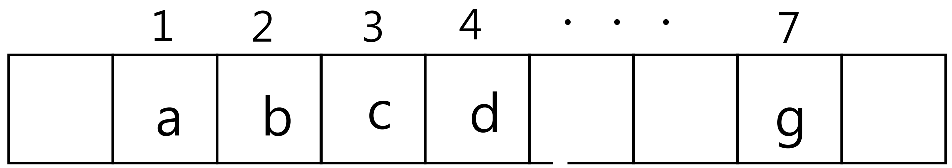
- What is height (h) of a binary tree
  - n : #nodes of a binary tree;
  - $h \leq n \leq 2^h - 1$
  - Thus, $\log_2(n+1) \leq h \leq n$
  - n =1,000? n =1,000,000?

- Question : What kind of trees do you prefer?

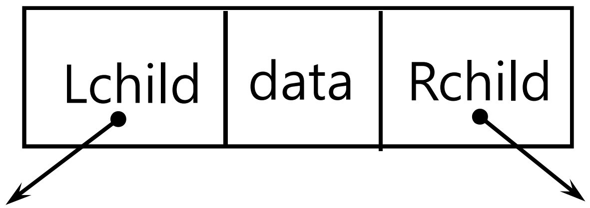
Implementing Binary Trees

- **Arrays**
  - **1-D** array : A[ ]
  - Parent[i] = i/2, Lchild[i] = 2i, Rchild[i] = 2i+1





- **Linked Lists**
  - Two links for each node
  - Lchild, RChild



- How many memories needed?

- How about trees? Array vs. LL?

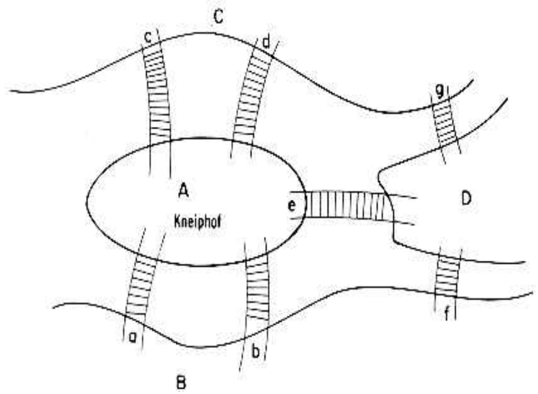
Applications : Trees

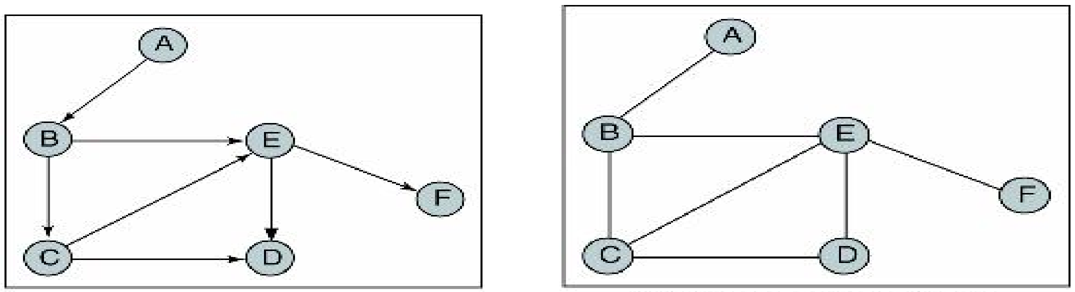
- Hierarchical Information
- Tree Traversals : INORDER, PREORDER, POSTORDER
- Internal Searching : BST, AVL Tree, Red/Black Tree, 2-3 Tree, . .
- External Searching : B Tree, B+ Tree, . .
- Decision Trees : Classifications
- Min/Max Heaps

Graphs

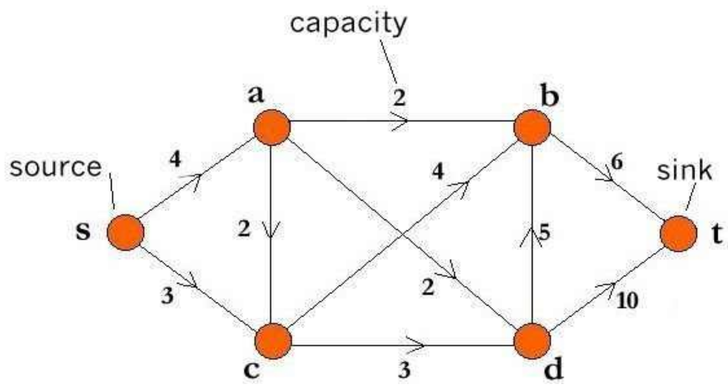
Graph : G = (V, E)

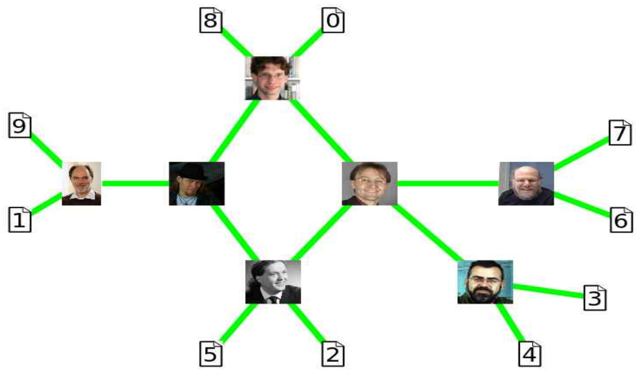
- V : a (non-empty) set of vertices
- G : a set of edges $\subseteq$ (V $\times$ V)
- Undirected : (u, v) = (v, u)
- Directed : (u, v) $\neq$ (u, v)





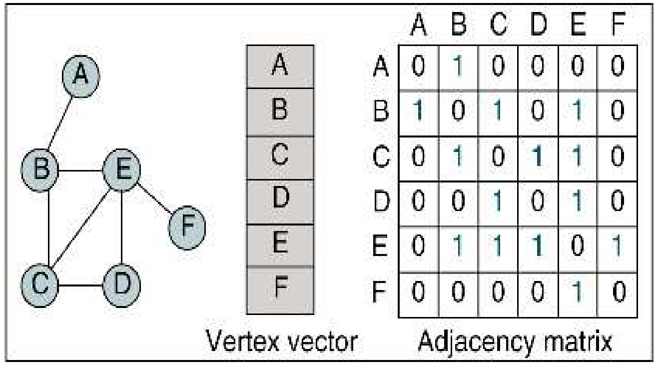
(a) Directed graph (b) Undirected graph



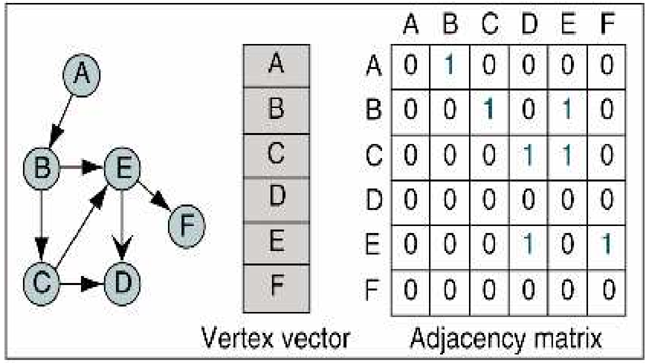


Implementing Graph

- **Adjacency Matrix** : $O(n^2)$
  - **2-D** array : A[n, n] (n : #vertices)
  - A(i, j) = 1 if vertex i and j are adjacent
    = 0 otherwise

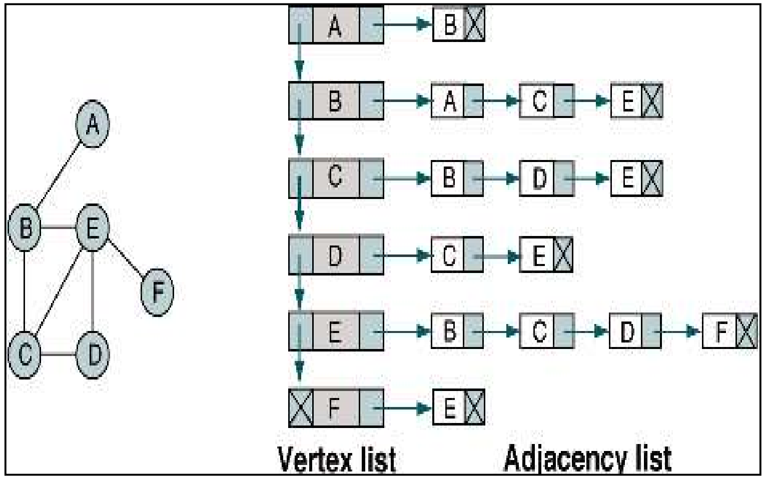


(a) Adjacency matrix for nondirected graph



(b) Adjacency matrix for directed graph

- **Adjacency List** : O(n +e)
  - Each node consists vertex and link.
  - For each linked list i, it contains vertices adjacent from vertex i



Applications : Graphs

- Depth First Search, Breadth First Search
- Connectivity
- Minimal Spanning Trees
- Articulation Points
- Topological Sorting
- Activity On Vertex(AOV) Networks
- Activity On Edge(AOE) Networks

Exercise : Searching

- Given n numbers, find an arbitrary number X;
  Design an efficient data structure; Search, Insert, Delete;
  - Array : Unordered
  - Array : Ordered (Too much burden!!)
  - Binary Search Tree
  - AVL Tree
  - Quad, Octal, . . Tree
  - B-Tree (External Searching)

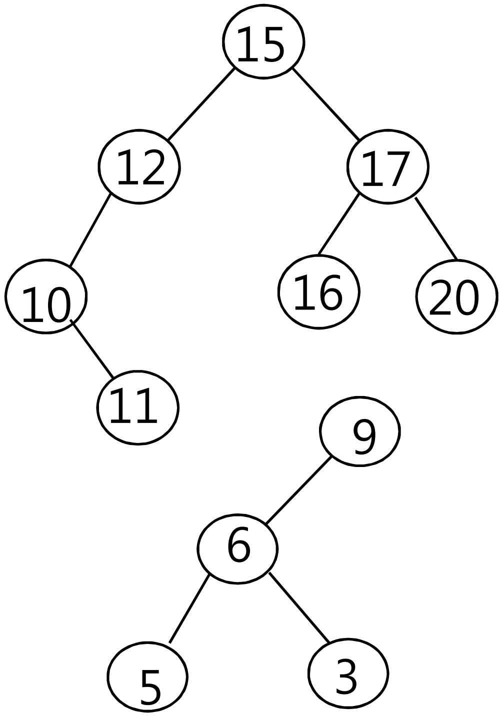
Binary Search Tree(BST)

BST :

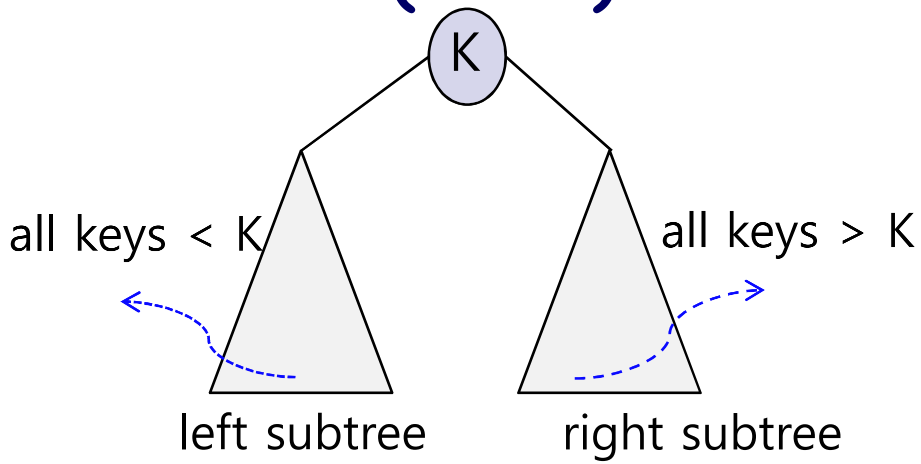
(1) Binary Tree

(2) Every node's key K is

① larger than all keys in its left subtree ② smaller than all keys in its right subtree.



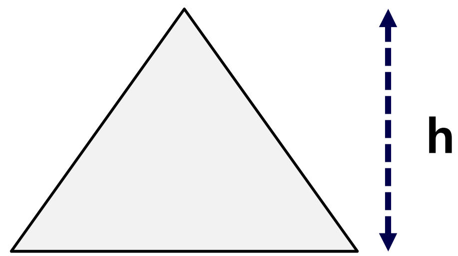
BSTs



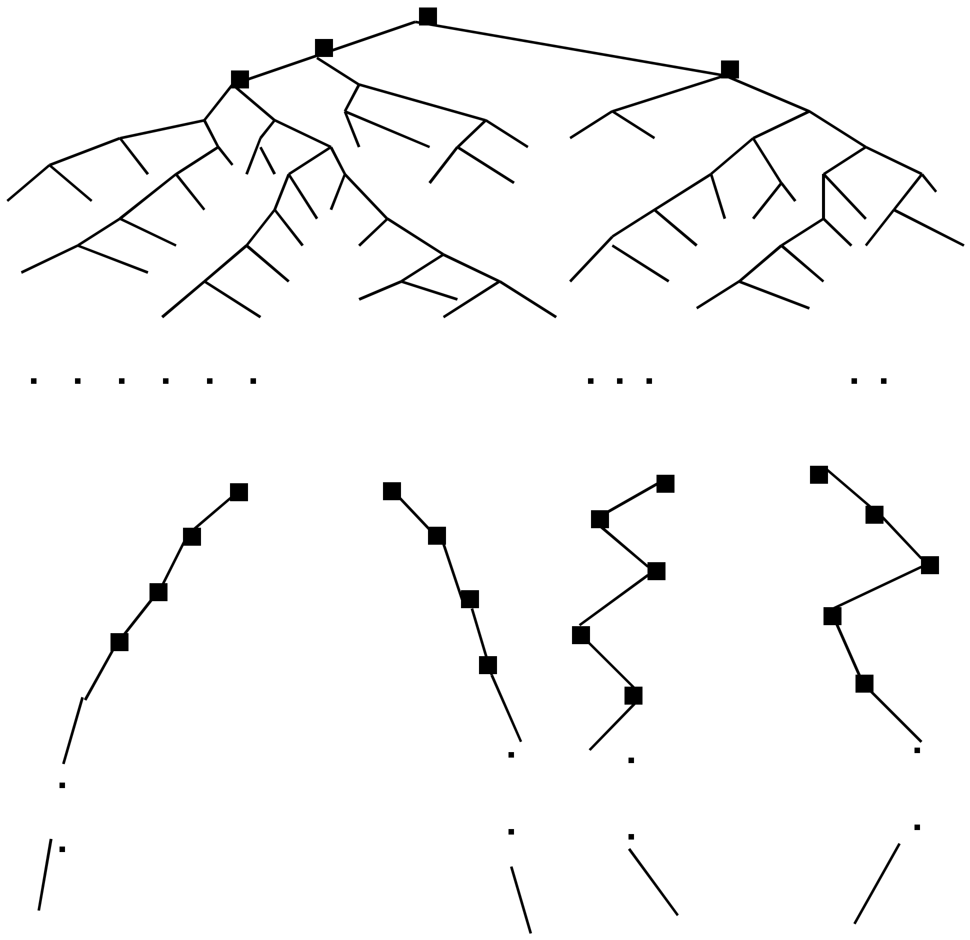
Search/Insert/Delete : Time

Ex: Search(11), Search(18), Insert(18), . .

Height(h) of a BST :



Performance : BST



Average Case :

$O(\log_2 n)$

Worst Case :

$O(n)$

Improving Worst Case

- Basic Idea : **Balanced + Many Children**
  - Binary Search Tree
  - AVL Tree
  - 2-3-4 Tree
  - Quad, Octal Tree
  - B-Tree (External Searching)

Performance Comparison : Searching

Data Structures	Worst	Average
Unordered Array	O(n)	O(n)
Ordered Array	$O(\log_2 n)$	$O(\log_2 n)$
Binary Search Tree	O(n)	$O(\log_2 n)$
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$
2-3-4 Tree	$O(\log_{2\sim4} n)$	$O(\log_{2\sim4} n)$
B Tree (External)	$O(\log_{133} n)$	$O(\log_{133} n)$

Exercise : Searching Maximum Value

- Given n numbers, find a maximum number X;
  Application : Priority Queue

  Design an efficient data structure; Search, Insert, Delete;

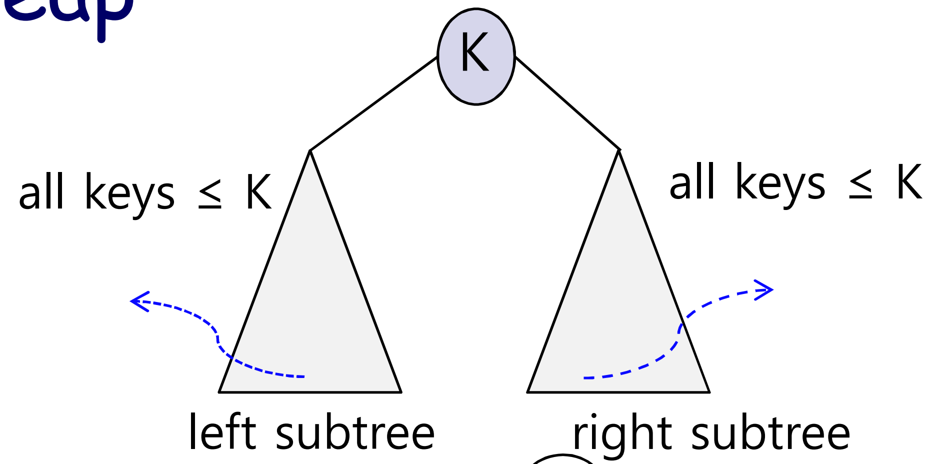
  - Array : Unordered
  - Array : Ordered
  - Binary Search Tree
  - Max Heap

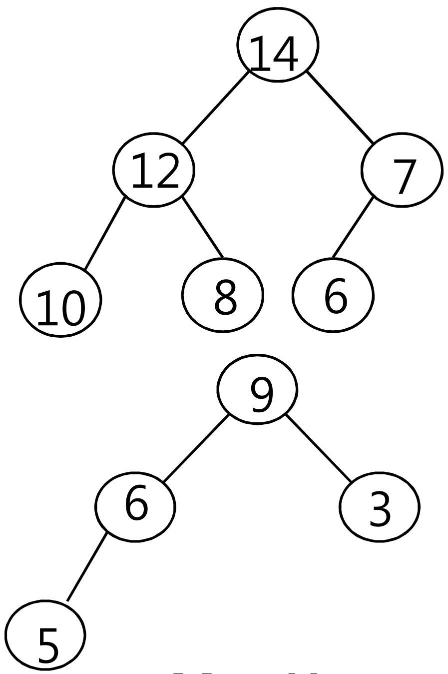
Max Heap

Max Heap :

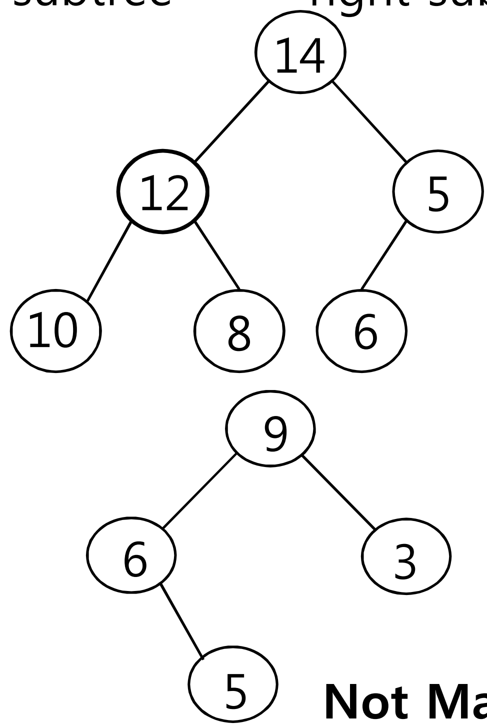
(1) Complete binary tree

(2) Value of each node is no smaller than its children's values.





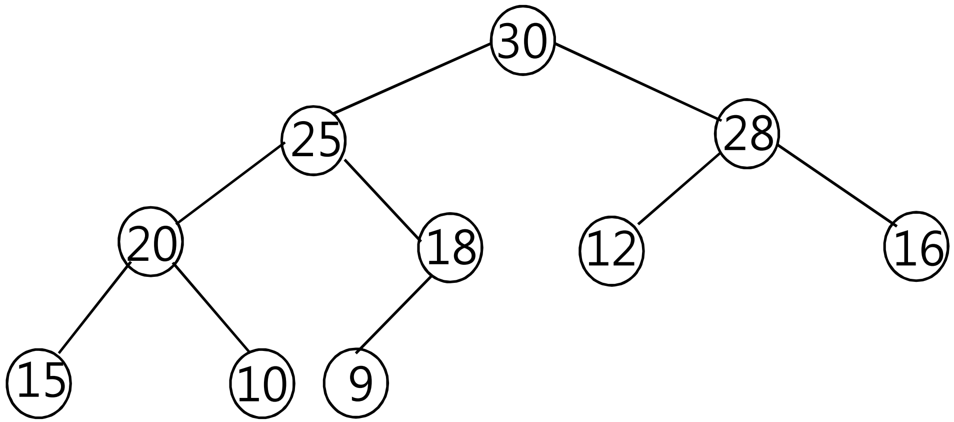
Max Heaps



Not Max Heaps

- Note : Root of a max heap always has the largest value.

Performance : Max Heap



- Insert 40, 45, . .
- Delete, Delete, . .

- **Insert :** $O(\log_2 n)$
- **Delete** : $O(\log_2 n)$

Performance Comparison : Find Max

Data Structures	Insertion	Deletion
Unordered Array	O(1)	O(n)
Unordered Linked list	O(1)	O(n)
Ordered Array	O(n)	O(1)
Ordered Linked list	O(n)	O(1)
Binary Search Tree	O(n)	O(n)
Max Heap	$O(\log_2 n)$	$O(\log_2 n)$

Constructing Algorithms

- Constructing Algorithm : Two Methods

  (1) **Iteration**
  - while-loop, for-loop, repeat-until, . . .
  - Conventional Methods

  (2) **Recursion**
  - Defined by calling itself.
  - Mostly based on divide and conquer
  - Simple, concise, high readability

- For every iterative algorithm, there exists an equivalently recursive algorithm; The reverse also is true.

Example : Factorial Number

Iteration

- Mathematical

$$Factorial(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{bmatrix}$$

- Algorithmic

```
int factorial (int n)
   i = 1; result = 1;
   while (i <= n)
     { result = result * i;
        i++;
     }
   return (result);
```

Recursion

$$Factorial(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (Factorial(n-1)) & \text{if } n > 0 \end{bmatrix}$$

```
int Factorial (int n)
    if (n==0 ) return(1);
    else return (n*Factorial(n-1));
```

Designing Recursion

- Rules for designing a recursion

1. Base case
   - Trivial case
   - Usually, n = 0 or n = 1
   - For Termination

2. General case (= Recursive step)
   - Break down the problem into sub-problems which are the same, but smaller size.
   - Usually, n > 0 or n > 1

3. Combine base case and general case.

Binary Search

- Find an integer X among n (> 1) integers; list[n]
  (All integers are stored by increasing order: Sorted)

- Construct Binary Search algorithms by recursion;
  (Use 3 variables : mid, left, right)

  1. Base Case : Termination Condition
     (1) X is found : ?
     (2) X is not found : ?

  2. General Case : Break a list into small size
     (1) X is in the first half (list[mid] > X) : ?
     (2) X is in the second half (list [mid] < X) : ?

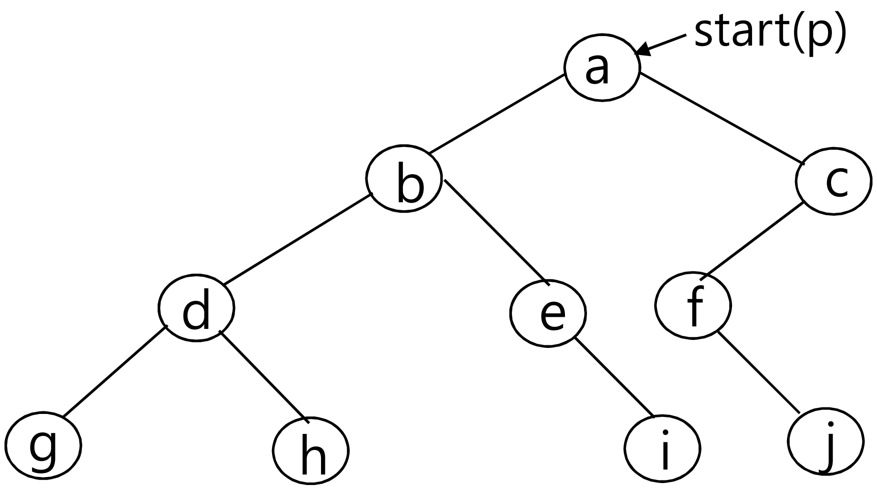
Binary Search

```
Bin-Search (list[], X, left, right)
  int mid;
  if (left <= right)  {
    mid = (left + right)/2;
    if  X < list[mid], Bin-Search(list [], X, left, mid–1);
    else if  X == list[mid], return(mid);
    else,  Bin-Search(list[], X, mid+1, right);    }
```

- What is time complexity? ($T(n) = T(n/2) + 1$)

Binary Tree Traversal

- We want visit every node in a binary tree.
  - INORDER : Left, Visit, Right (LVR)
  - PREORDER : Visit, Left, Right (VLR)
  - POSTORDER : Left, Right, Visit (LRV)




```
INORDER(p)
 if (p != NULL)
  INORDER(p->Lchild)
  print(ptr->data)
  INORDER(p->Rchild)
```

```
POSTORDER(p)
 if (p != NULL)
   PREORDER(p->Lchild)
   PREORDER(p->Rchild)
   print(ptr->data)
```

Computing X^N

```
POWER (int X, int N)
    if (N == 0) return 1;
    else {
      factor = POWER(X, N/2)
      if  N%2 == 0  return  factor*factor
      else  factor*factor*X
        }
```

$X^N = (X^{N/2} * X^{N/2})$ if N : even N = 8; $2^8 = 2^4 * 2^4$, $2^4 = 2^2 * 2^2$, $2^2 = 2^1 * 2^1$

$X^N = (X^{N/2} * X^{N/2}) * X$ if N : odd N = 9; $2^9 = 2^4 * 2^4 * 2^1$, $2^4 = 2^2 * 2^2$ $2^2 = 2^1 * 2^1$

Towers of Hanoi

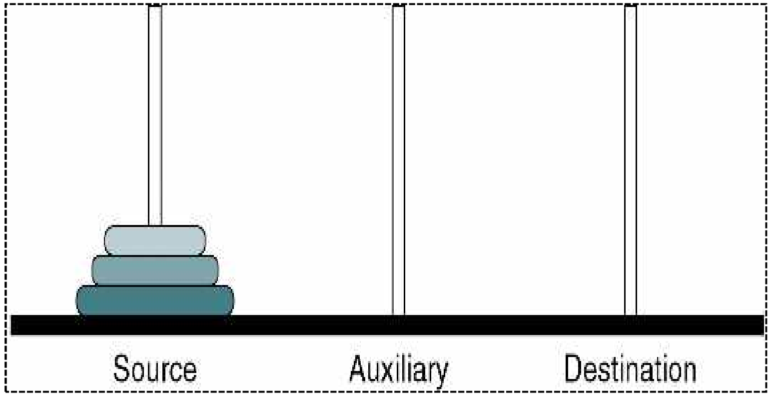
- Base case : n = 1

  : Move 1 disk from source to dest

- General case : n > 1

  (1) Move (n - 1) disks from source to aux : (Use des as aux)

  (2) Move (n - 1) disks from aux to des : (Use source as aux)




```
towers (int n, source, dest, aux)
    if (n == 1)          // base case
        print (Move from to, source, dest);
    else {               // general case
        towers (n - 1, source, aux, dest);
        towers (1, source, dest, aux);
        towers (n - 1, aux, dest, source); }
```

Recursion is Inefficient . . .

- Which algorithm is more efficient?

Iterative version

```
int factorial (int n)
{  i = 1;
   result = 1;
   while (i <= n)
     { result = result * i;
        i++;
     }
   return (result);}
```

Recursive version

```
int Factorial (int n)
{
     if ( n == 0 ) return(1);
     else return (n * Factorial (n - 1));
}
```

Pros/Conse : Recursion

- Pros/Cons

  (+) Coding is simple, concise, clear.

  (+) Implementation is hidden;

  (+) High understandability, readability.

  (-) Space Overhead

  (-) Time Overhead

- When do we need a recursion?

  Do <u>not</u> use a recursion if the answer of the the questions is 'no':

  1. Is the algorithm naturally suited to recursion?
  2. Is the recursive solution shorter and more understandable?
  3. Does the recursive solution run within acceptable time and space?

Algorithm Design Techniques

- Brute Force
- Greedy method
- Divide and Conquer
- Dynamic Programming
- Backtracking

Brute Force

- A straightforward approach to; It tries to find all possible searching spaces.

- Easiest approach and useful for solving small size of a problem.

- Exhaustive search: May be exponential!

- Examples :
  - Computing a^n (by multiplying a*a*…*a)
  - Selection Sort, Bubble Sort
  - Shortest Paths
  - Sequential search

Greedy Method

- At each solving step, choose the choice what it looks best; The choice must be locally optimal. Can't see the global solution.

- Making the locally optimal choice at each stage with the hope of finding a global optimum. For example, road driving, card playing, . .

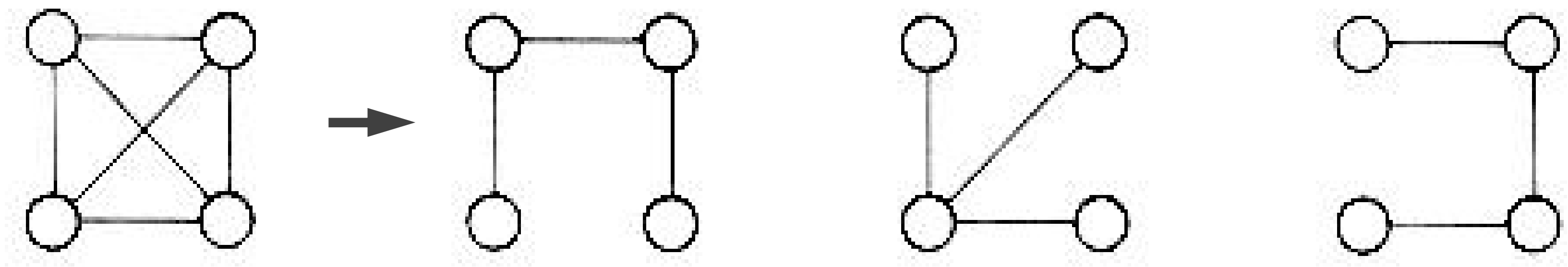
- This method always does not give optimal solution, but it works for many problems in a reasonable time.

- Examples :
  - Minimal Spanning Tree
  - Shortest Paths
  - Fractional Knapsack
  - Huffman Coding

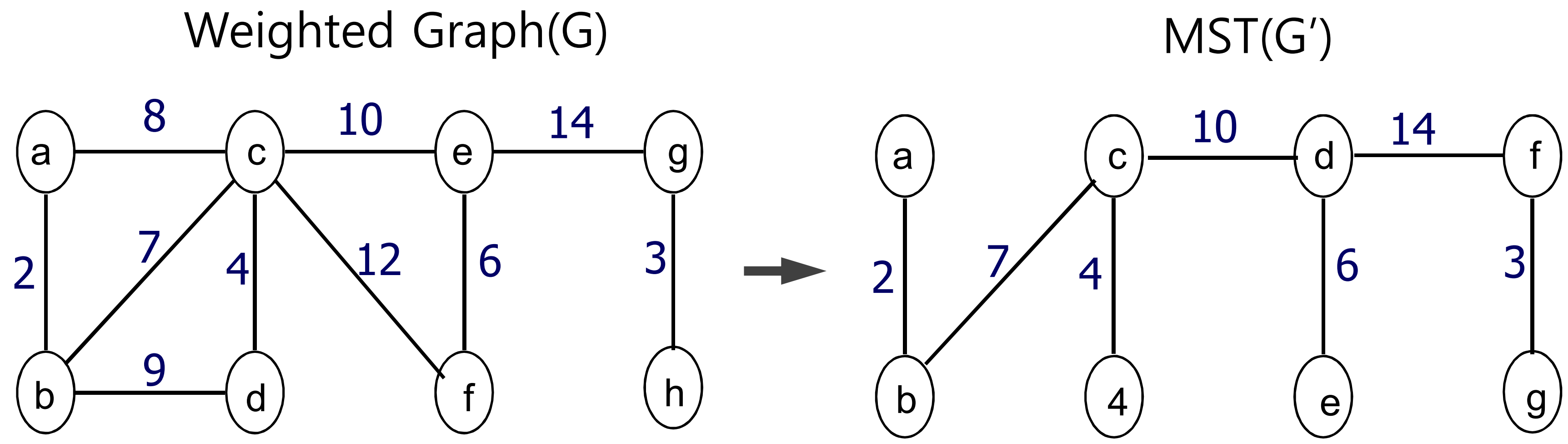
Spanning Tree

- **Spanning tree** G′ is a subgraph of a graph G such that
  - (1) V(G′) = V(G) = n (n : # vertices)
  - (2) G′ is connected.
  - (3) G′ has (n – 1) edges.
  - (4) If we add an edge into G′, then a cycle is generated.
  - (5) If we delete an edge from G′, then disconnected.

Graph G　　　　**Some Spanning Trees G′ of G**



Minimal Spanning Tree (MST)



- **MST** is a spanning tree with minimum total weight.

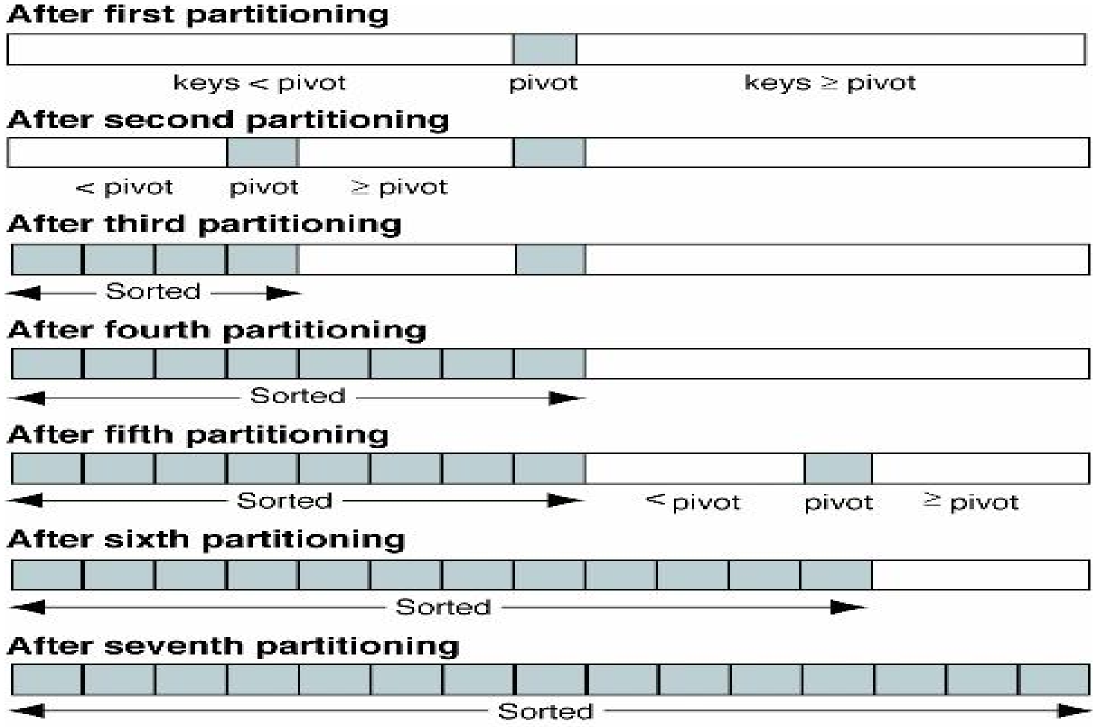
- Greedy Method : (Kruskals's algorithm : $O(e\log_2 e)$)
  - (1) At each step, choose an edge with smallest weight.
  - (2) If the selected edge creates a cycle, then discard it.
  - (3) Repeat (1), (2); If sum of total edges are (n – 1), then done!

Divide and Conquer

- **Divide** a problem into many **smaller** sized sub-problems.
- Independently solve each sub-problem and then **combine** the sub-instance solutions to yield a solution for the original problem.
- The size of the problem is usually reduced by a factor (e.g., half the input size).
- Examples :
  - Binary Search
  - Quick Sort
  - Merge Sort
  - Strassen's Matrix Multiplication
  - Computing a^n

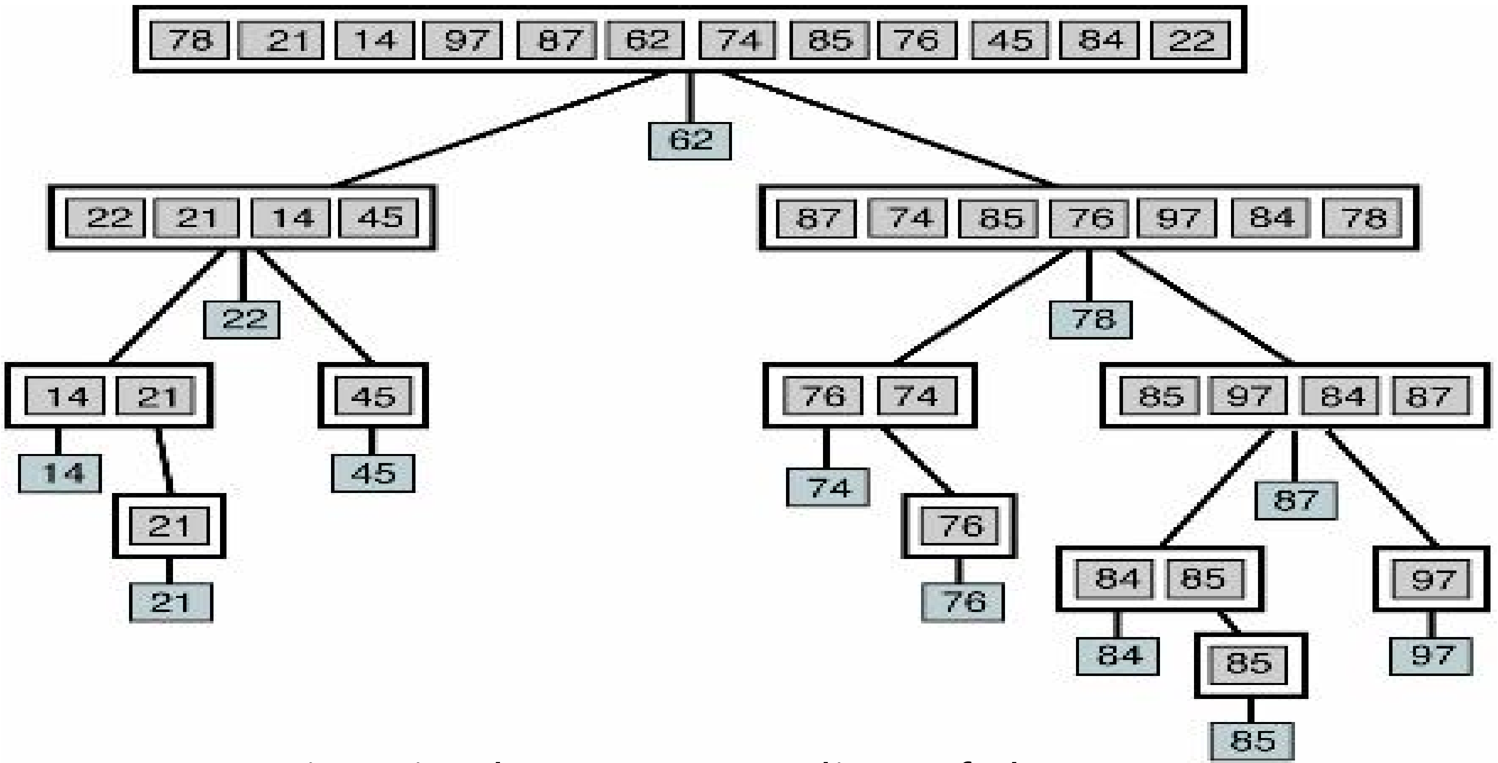
Quick Sort (Top 10 algorithms in 20th Century)

- Given a list of *n* elements (e.g., integers):
  - Pick one element to use as *pivot.*
  - Partition elements into two sub-lists:
    - Left sub-lists ***L*** : Elements less than or equal to pivot
    - Right sub-lists ***R*** : Elements greater than pivot
  - Recursively sort sub-list ***L*** and ***R***
  - Combine the results



Quick Sort

```
Quicksort (list[ ], int left, right)
    Partition; (list[ ], pivot)
    Quicksort (list, left, j-1);
    Quicksort (list, j+1, right);
```



- Assume : Pivot is chosen as median of three.

Quick Sort : Time Complexity

- Worst Case
  - When the sub-lists are completely biased
  - Pivot is chosen as a smallest (largest) key for each split
  - $T(n) = T(n-1) + c \cdot n$
  - $O(n^2)$
  - Rarely happens

- Average Case
  - When the sub-lists are likely balanced
  - Pivot is chosen as a random or median of three
  - $T(n) = 2 \cdot T(n/2) + c \cdot n$
  - $O(n \cdot \log_2 n)$
  - Fastest known sorting algorithm in practice

Dynamic Programming

- One drawback of “Divide and Conquer” is that the same computations repeatedly for identical sub-problems may arise.
- Dynamic Programming can avoid this drawback by defining the recurrence relation.
- Solve small sized sub-problems and store its result for later.
- The intermediate result can be reused for bigger problem.
- Examples :
  - Fibonacci Number
  - Warshall Algorithm
  - All Pairs Shortest Paths
  - 0/1 Knapsack
  - Matrix Chain Products

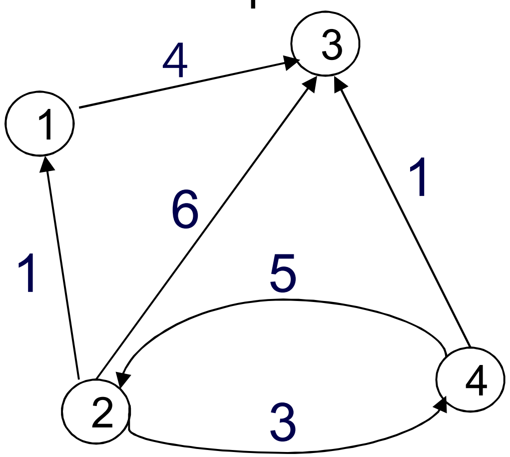
All pairs shortest paths

- Given a directed graph G with n vertices, find the shortest paths between every pairs of vertices

- Brute Force Approach :

- Dynamic Approach : Construct solution through series of matrices using increasing subsets of vertices allowed as intermediate.

Graph



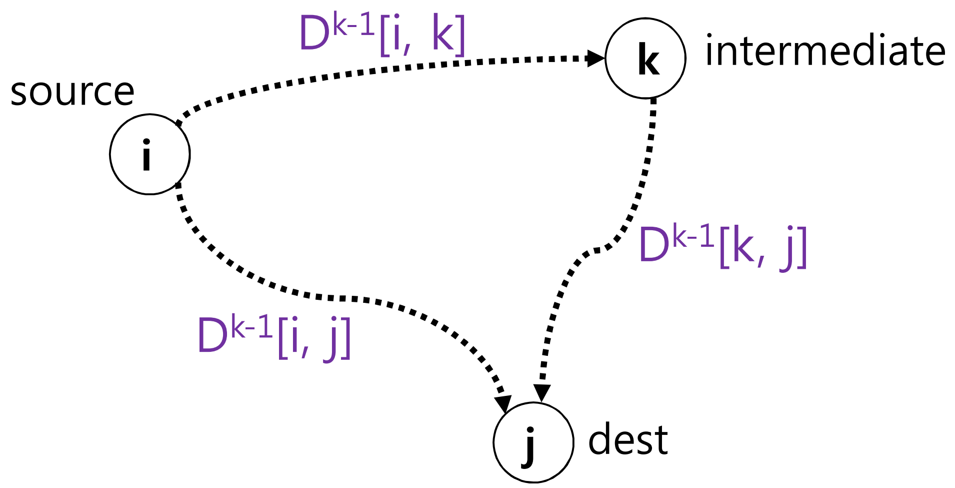
Adjacency Matrix

	1	2	3	4
1	0	∞	4	∞
2	1	0	4	3
3	∞	∞	0	∞
4	6	5	1	0

All pairs shortest paths

- We define as $D^k[i,j]$ as : length of the shortest path from **i** to **j** without going through any vertex greater than **k**.
  - Without going through k : $D^{k-1}[i,j]$
  - Going through k : $D^{k-1}[i,k]+D^{k-1}[k,j]$

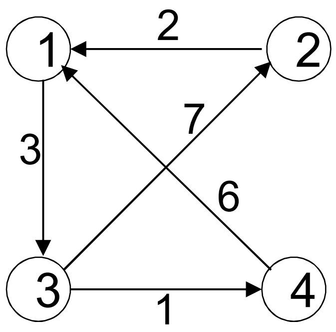
$$D^k[i,j] = \min \{D^{k-1}[i,j],\ D^{k-1}[i,k]+D^{k-1}[k,j]\}$$



- Our goal : k = n; Compute $D^n[i, j]$ for every pair of vertices i, j where i, j, k in [1, . . . n]

All pairs shortest paths

- Compute $D^4[i, j]$ for every pair of vertices i, j;



D^0 :

	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

D^1 :

	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0

D^2 :

	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0

D^3 :

	1	2	3	4
1	0	**10**	3	4
2	2	0	5	**6**
3	9	7	0	1
4	6	**16**	9	0

D^4 :

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	**7**	7	0	1
4	6	16	9	0

- For example, $D^1[2, 3] = \min \{D^0[2, 3], D^0[2, 1]+D^0[1, 3]\}$
  $= \min \{\infty, 2+3\} = 5$

All pairs shortest paths

- Floyd Algorithm

```
for (k=1; k<=n; i++)
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            D^k[i, j] = min {D^(k-1)[i, j], D^(k-1)[i, k] + D^(k-1)[k, j]}
```

- Time Complexity : $O(n^3)$

- Space Complexity : $O(n^2)$

- Note : Works on graphs with negative edges but without negative cycles.

Backtracking

- A sort of brute force approach, but additional condition that only the possible candidate solutions are considered.

- A systematic searching method by pruning searching spaces; This is to avoid unnecessary efforts as early as possible.

- Upon failure, we can go back to the previous choice simply by returning a failure node.

- Backtracking vs. DFS

- Examples :
  - Maze Problem
  - N-Queens Problem
  - Graph Coloring
  - Hamiltonian Cycle
  - Data Mining : Apriori Algorithm

Backtracking

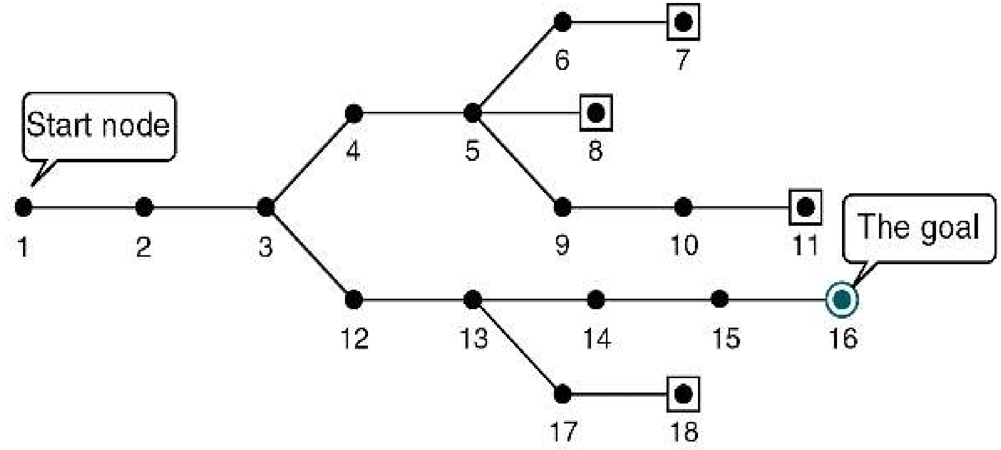
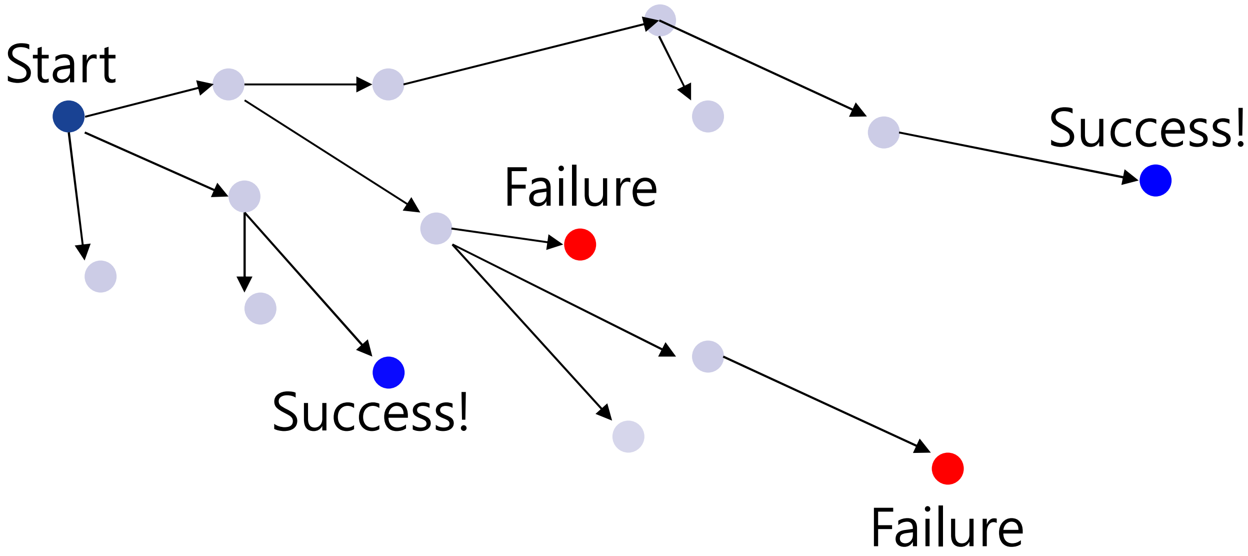


FIGURE 3-17 Backtracking Example

At 4	At 6	At 1st end	At 2nd end	At 3rd end	At goal
		end			
		7		end	goal
		6	end	11	16
	B8	B8	8	10	15
	B9	B9	B9	9	14
	5	5	5	5	B17
	4	4	4	4	13
B12	B12	B12	B12	B12	12
3	3	3	3	3	3
2	2	2	2	2	2
1	1	1	1	1	1

Backtracking

- In backtracking, we explore each node, as follows:
- To explore node N:
    1. If N is a goal node, return "success"
    2. If N is a leaf node, return "failure"
    3. For each child C of N,
        3.1. Explore C
            3.1.1. If C was successful, return "success"
    4. Return "failure"



Hard Problems

- So far, many problems can be solved by efficient algorithms.

- In other respect, for many problems, any efficient algorithms have not been found; What's worse, for such problems, we can't even tell whether or not an efficient solution might exist.

- Programmers : Why can not find such efficient algorithms?
  Theoreticians : Why can not find any reason why these problems
  should be difficult?

- Consider the following problems;
  - Easy : Is there a path from x to y with weight $\leq$ M
    - Shortest Path : $O(n)$
  - Hard(?) : Is there a path from x to y with weight $\geq$ M
    - Longest Path : $O(2^n)$

Hard Problems

- ***P*** Problems
  - Can be solved by deterministic algorithms in polynomial time.
  - Can be solved with efficient amount of time.
  - Searching, Soring, . . .

- ***NP*** Problems
  - Can be solved by non-deterministic algorithms in polynomial time.
  - For many problems, only exponential time algorithms are known. (Deterministic polynomial time algorithms are not known (so far).)
  - Can not be solved with efficient amount of time.
  - Satisfiability, Graph Coloring, . . .

- Relationship between *P* and *NP*
  - Clearly, $\boldsymbol{P} \subseteq \boldsymbol{NP}$ (Any problem in ***P*** is in ***NP***)
  - The biggest open problem in Computer Science;

    - Is $\boldsymbol{P} \subset \boldsymbol{NP}$ or $\boldsymbol{P} = \boldsymbol{NP}$?

Unsolvable (Undecidable) Problems

- Is every problem is solvable?
  - The number algorithms is countably infinite.
  - The number of problems is un-countably infinite.
  - There exist some problems not solvable by any algorithms.
  - There exist infinite number of problems not solvable by computers.
  - Turing-Undecidable

- Examples
  - Post Correspondence Problem(PCP)
  - Halting Problem
  - Ambiguity Problem
  - . . . .

Conclusions To Remember

- Lesson 1 :
  Good algorithms are better than super computers.

- Lesson 2 :
  Good algorithms are better than good algorithms.

- Lesson 3 :
  Good data structures are essential for good algorithms.

- Lesson 4 :
  Try to remember a few well known algorithms.

- Lesson 5 :
  Try to learn programming languages and exercise coding.