Chapter 6. Tree

- 1. Introduction

    1.1 Tree Terminology

- 2. Spanning Trees

    2.1 Tree search

    2.2 Minimal Spanning Trees (MST)

- 3. Binary Tree

    3.1 Properties

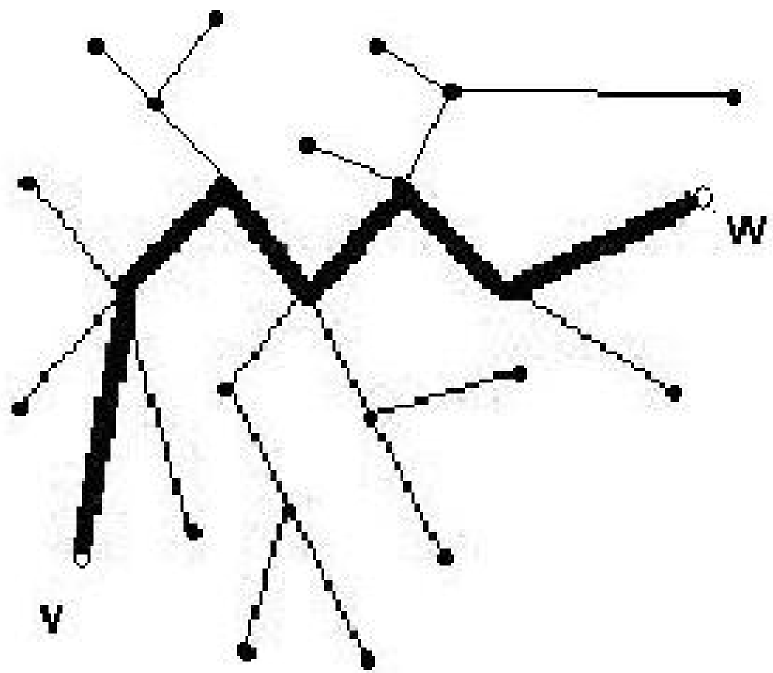
    3.2 Tree Traversal

    3.3 Binary Search Tree (BST)

    3.4 Other Properties of Trees

1. Introduction

A Graph is a TREE if graph is connected and Acyclic

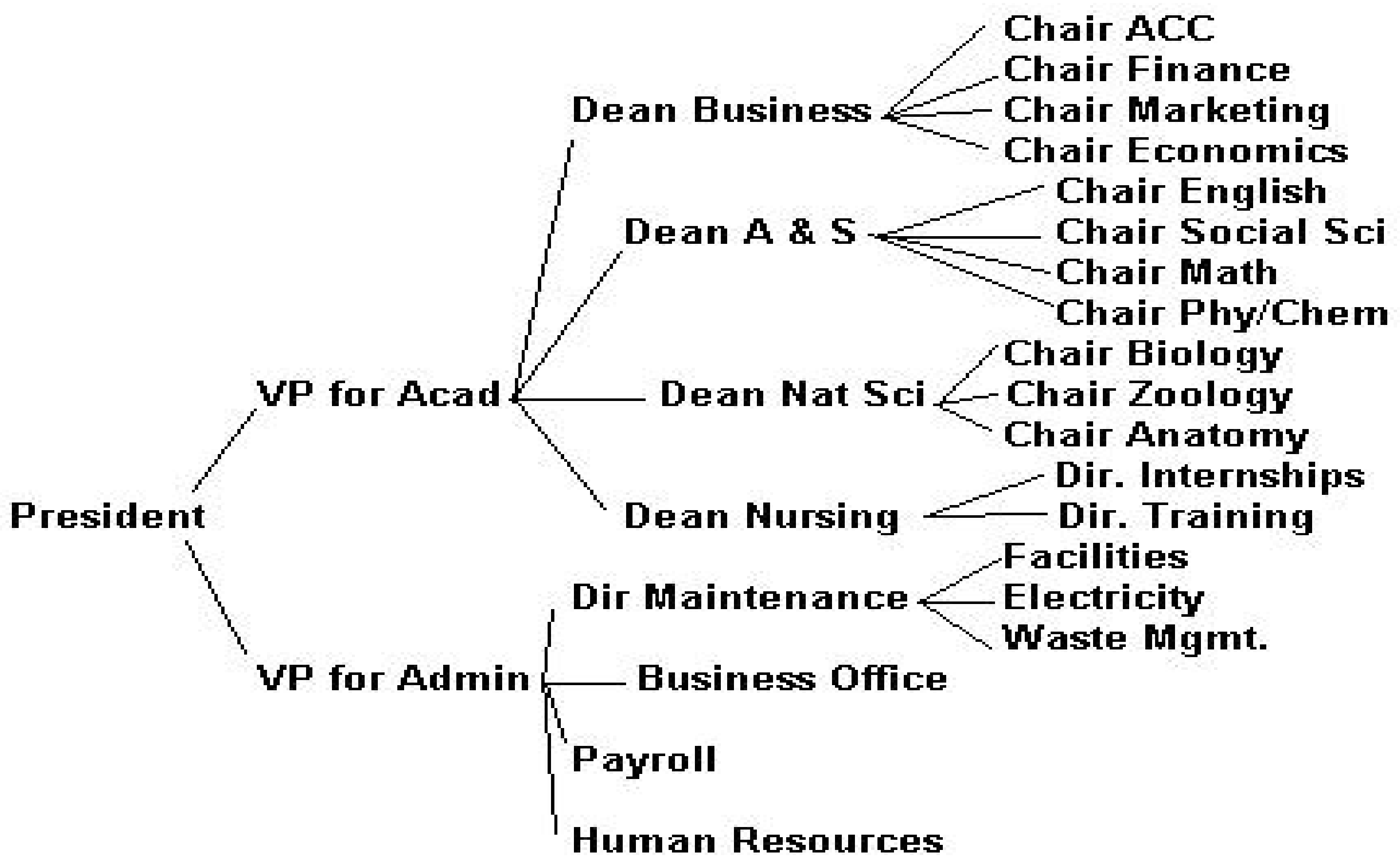


A *(free) tree* T is

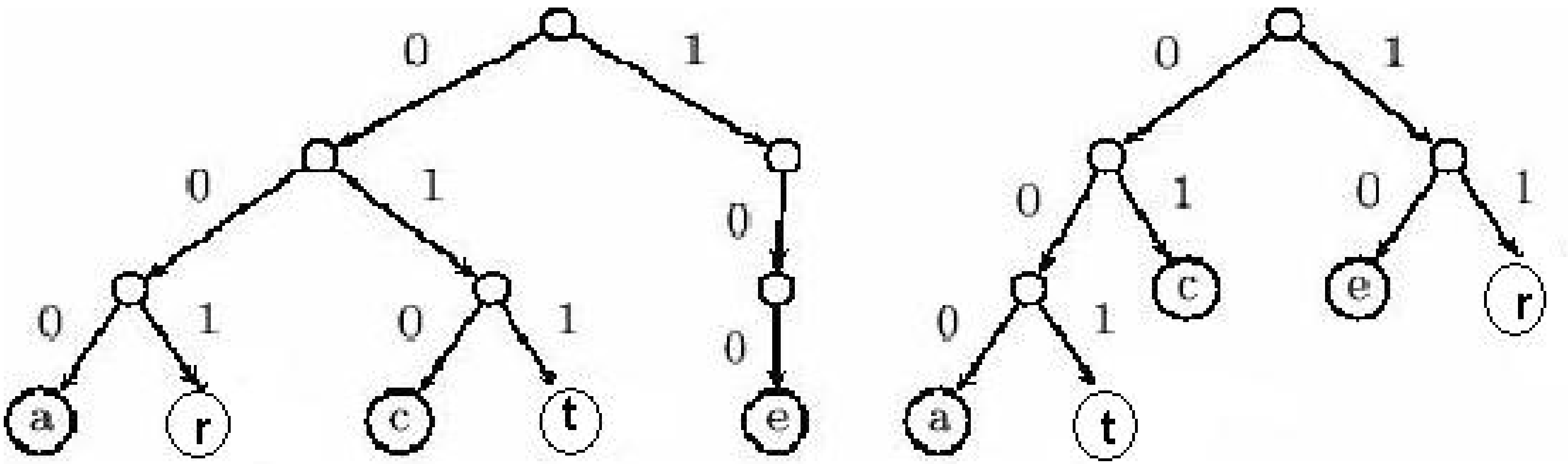
- A simple graph such that for every pair of vertices v and w, there is a unique path from v to w

- *Representation vehicle* for expression evaluation in sorting and searching
- *Hierarchical Representation*

Application - Organizational charts



Application - Huffman codes



- On the left tree the word **rate** is encoded

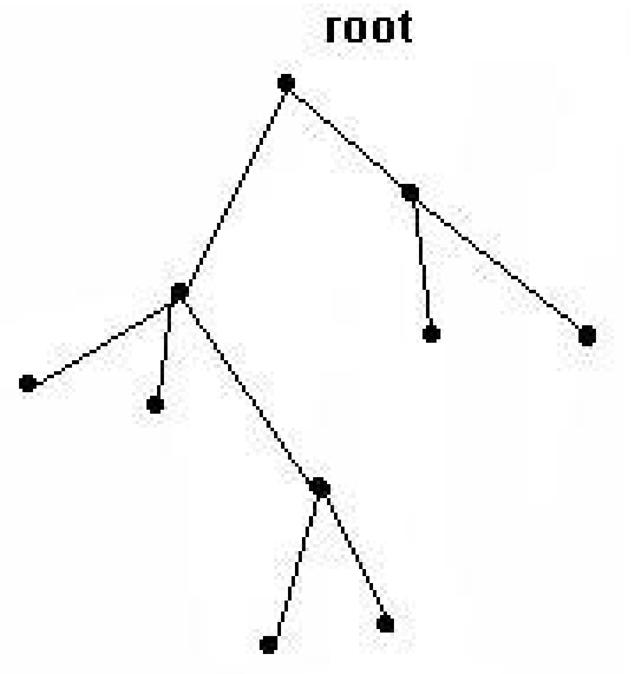
  001 000 011 100

- On the right tree, the same word **rate** is encoded

  11 000 001 10

1.1 Terminology

* **Rooted Tree**: one of its vertices is designated as the ***root***



Let T be a rooted tree:

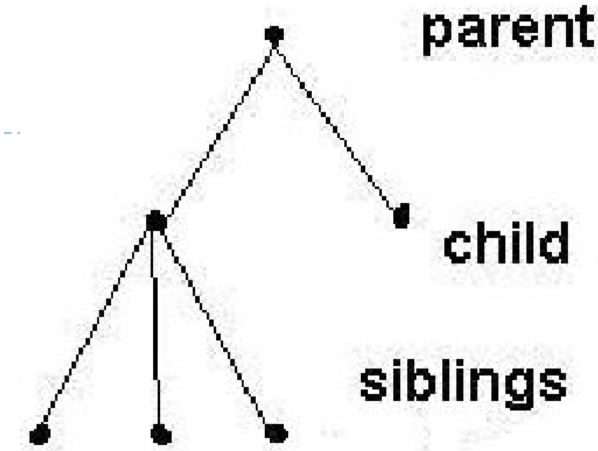
- The ***level l(v)*** of a vertex **v** is the length of the simple path from **v** to the root of the tree
- The ***height h*** of a rooted tree T is the maximum of all level numbers of its vertices:

$$h = \max_{v \in V(T)} \{ l(v) \}$$

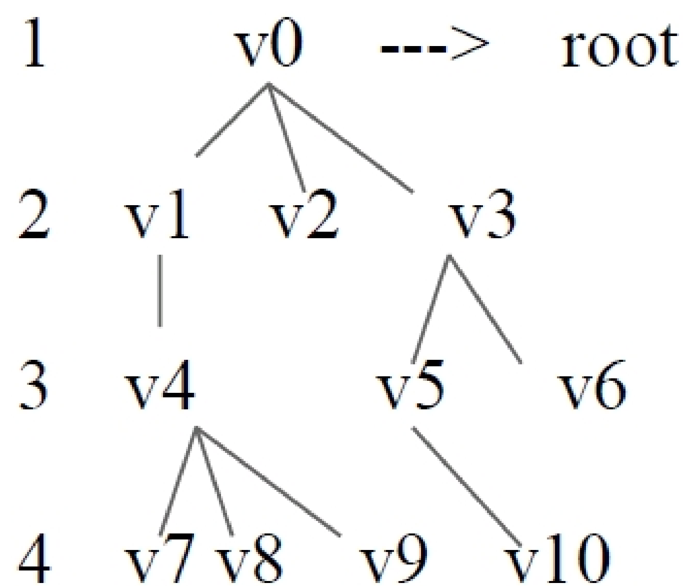
Ex) the tree on the right has height 3

Terminology

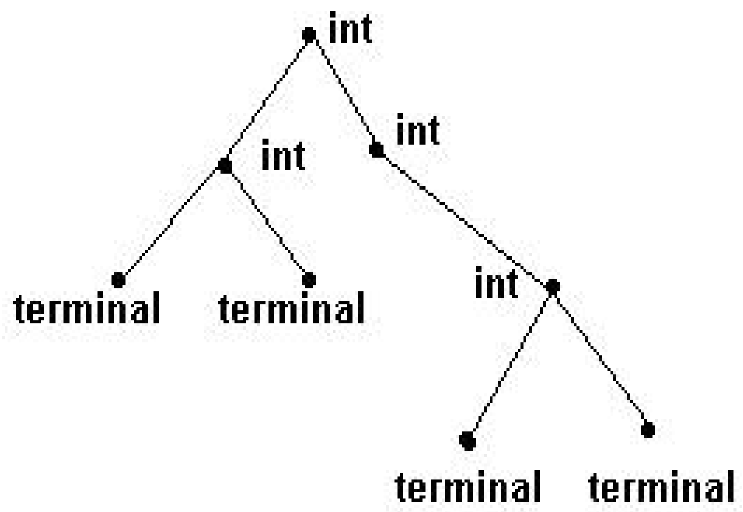
- Parent of V2 = ?
- Child of V3= ?
- Siblings(same parent) of V8 = ?



- Ancestor
- Descendant
- Terminal vertices
- Internal vertices



Internal and external vertices

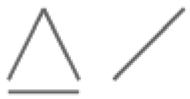


- An ***internal vertex*** is a vertex that has at least one child **(non leaf node, non-terminal)**

- A ***terminal* vertex** is a vertex that has no children **(leaf node)**

- The tree in the example has 4 internal vertices and 4 terminal vertices

Ex) If a graph has N vertices, N-1 edges.
Is this graph a TREE?



Def: If Graph G is connected graph with N vertices, and has N-1 edges, then it is a TREE

Characterization of trees

Theorem

If T=(V,E) is a graph with n vertices, the following are equivalent:

a) Graph T is a tree

b) T is connected and acyclic ("acyclic"= having no cycles)

c) T is connected and has n-1 edges

d) T is acyclic and has n-1 edges

e) There exists one path between any pair of vertices in T

f) If remove any edge -> disconnects T

g) Acyclic and adding any edge -> creates a cycle

Proof of the Theorem (1 to 2)

- **if Graph T is tree then T is connected and acyclic**

(proof)

Let T be a tree. Then T is connected since there is a path from any vertex to any other vertex. (by tree definition)

Suppose T contains a cycle C.

Then T contains a simple cycle C,   C = (v0,...vn), v0=vn

Since T is a simple graph, C cannot be a loop;

=> so C contains at least two distinct vertices vi and vj, i<j.

Now (vi, vi+1,...vj), (vi, vi-1,...v0, vn-1,...vj) are two distinct simple paths from vi to vj, which contradicts the definition of tree. => Therefore a tree cannot contain a cycle => a tree is acyclic

Proof of the Theorem (2 to 3)

- **if T is connected & acyclic, then T is connected & has n-1 edges** (Math Induction Proof)

- Basis : if n=1, then T consists of 1 vertex and 0 edges, so it is true

- Hypothesis : Suppose T is connected and acyclic graph with n vertices, so T has n-1 edges.

- Induction : Let T be a connected, acyclic graph with n+1 vertices

Choose simple path P of maximum length. Since T is acyclic, P is not a cycle. Therefore P contains a vertex v of degree 1.

Let T* be T with v and the edge incident on v removed

Then T* is connected and acyclic, because T* contains n vertices, by HYP T* contains n-1 edges. Therefore T contains n edges.

Proof of the Theorem (3 to 4)

- **If T is connected & n-1 edges, then T is acyclic & n-1 edges**

(Contradiction proof)

Suppose T is connected and has n-1 edges, then we need to show that T is acyclic

(not q:) Suppose T contains at least one cycle.

(prove) T* is a resulting graph from removing edges until graph is connected and acyclic

Now T* is acyclic, connected graph with n vertices.

From (2->3), we conclude that T* has n-1 edges.

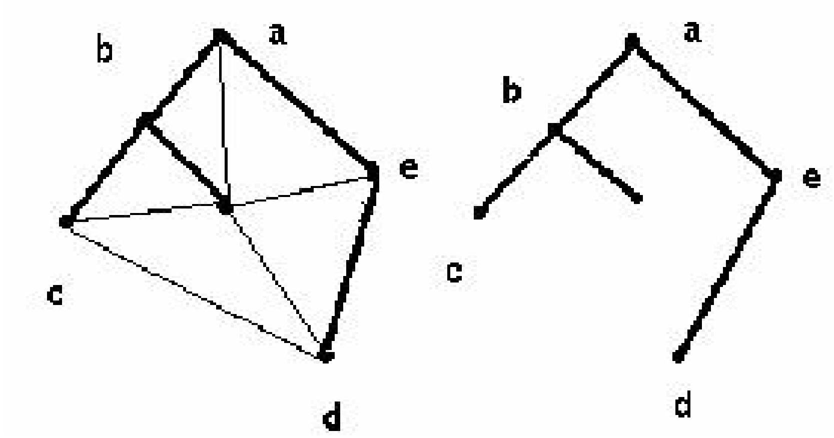
But T has more than n-1 edges, which is contradiction.

2. Spanning trees

Def: Given a graph G, a tree T is a *spanning tree* of G iff:

- T is a subgraph of G and T contains all the vertices of G

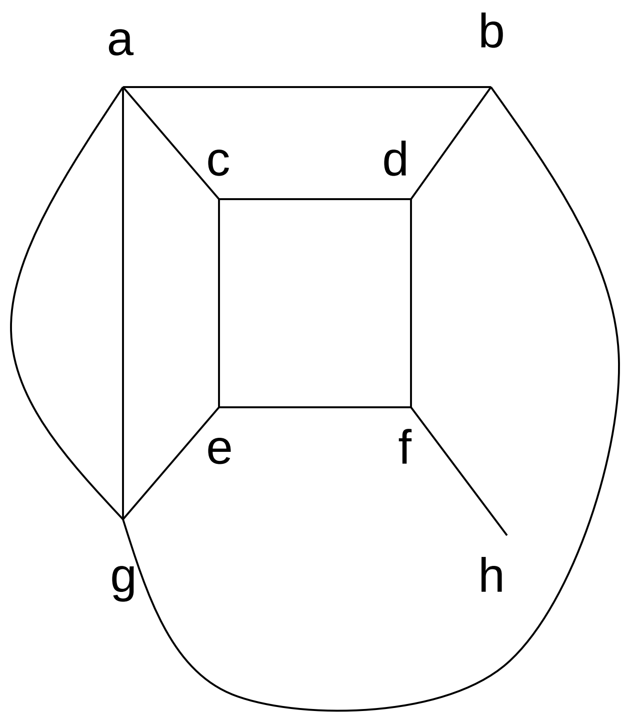
- Let G=(V, E), Let G'=(V',E'), if V'= V and E'$\subseteq$E, then G' is Spanning Tree of G)



Spanning tree search

- Breadth-first search method

- Depth-first search method (backtracking)



Algorithm 9.3.6: Breadth-First Search for a Spanning Tree

Input: A connected graph G with vertices ordered $v_1, v_2, \ldots v_n$

Output: A spanning tree T

```
bfs(V, E) {
   // V = vertices ordered v_1, ..., v_n; E = edges
   // V' = vertices of spanning tree T;
   // E' = edges of spanning tree T
   // v_1 is the root of the spanning tree
   // S is an ordered list
   S = (v_1)
   V' = {v_1}
   E' = ∅
   while (true) {
      for each x ∈ S, in order,
         for each y ∈ V − V', in order,
            if ((x, y) is an edge)
               add edge (x, y) to E' and y to V'
      if (no edges were added)
         return T
      S = children of S ordered consistently with the
            original vertex ordering
   }
}
```

Algorithm 9.3.7: Depth-First Search for a Spanning Tree

Input: A connected graph G with vertices ordered $v_1, v_2, \ldots v_n$

Output: A spanning tree T

```
dfs(V, E) {
  // V' = vertices of spanning tree T;
  // E' = edges of spanning tree T
  // v1 is the root of the spanning tree
  V' = {v1}
  E' = ∅
  w = v1
  while (true) {
    while (there is an edge (w, v) that when added to T
           does not create a cycle in T) {
      choose the edge (w, vk) with minimum k that when
        added to T does not create a cycle in T
      add (w, vk) to E'
      add vk to V'
      w = vk
    }
    if (w == v1)
      return T
    w = parent of w in T // backtrack
  }
}
```

2.1 Tree Search - BFS

1) Let's start with 'a'

2) Add all edges (a,x), x = b to h, which does not create a cycle

   x = g,c,b at level 1 (a,g), (a,c), (a,b)

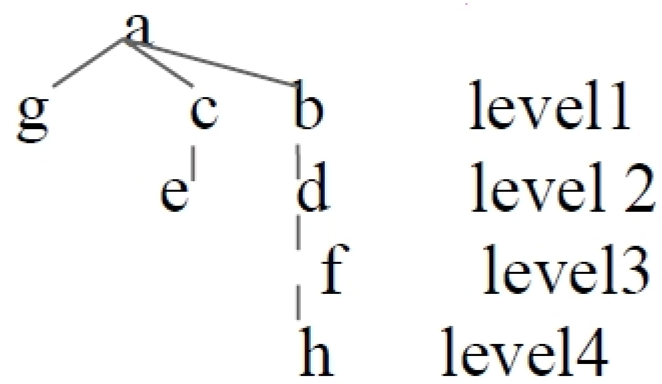
3) repeat this process at level 1

   b: include (b,d), c: include (c,e), g: none

4) repeat this process at level2 repeat this process at level3

   d: include (d,f), e: none

   f: include (f,h)



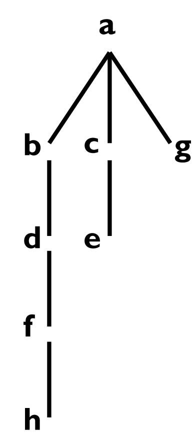
Tree Search - DFS

- 정점 a에서 시작, 자식 노드들: {(a,b),(a,c)(a,g)} 이다

- 다음 탐색 정점인 b를 탐색하면, (a, b)를 E'에 추가
  b의 child, (b, d) E'에 추가-> d 의 child, (d, f)추가
  -> f 의 child h, (f, h) 를 E'에 추가
- H다음-> deadlock 발생 -> backtrack -> c로 내려감

- 노드 c의 child 는 e, (c, e)를 E' 에 추가

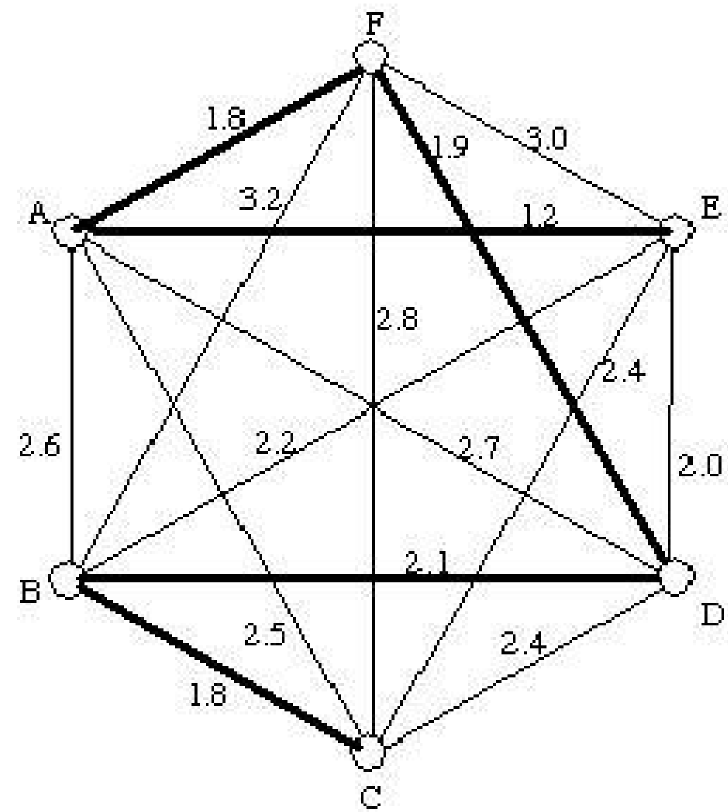
- 모든 노드를 탐색 하였기 때문에 종료함.



2.2 최소비용트리
Minimal spanning trees (MST)

Def: Given a weighted graph G, a *minimum spanning tree* is

- a spanning tree of G
- that has minimum “weight”



- 알고리즘

1) Kruskal

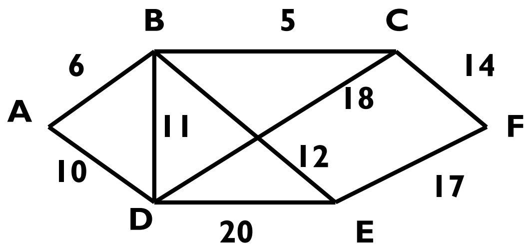
2) Prim

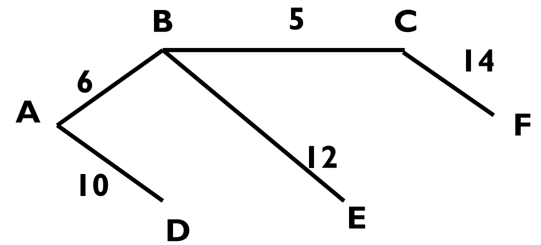
Kruskal’s algorithm

```
begin          // Assume edges are sorted already
 T <- {0}
  m <- 0

  while (m <n-1)
    find smallest e
    delete e from E
    If addition of e to T does not  produce cycle
    then add e to T
           and set m = m+1
end while
```

Kruskal's algorithm exercise





(MST) Cost = 47

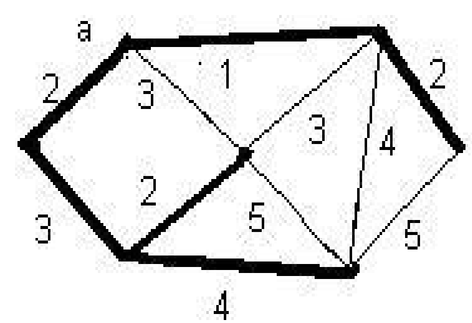
edges	Action	cost
5	accept edge 5	cost=5
6	accept edge 6	cost= 5+6=11
10	accept edge 10	cost= 11+10=21
11	edge 11추가, cycle발생,	reject edge 11
12	accept edge 12	cost= 21+12=33
14	accept edge 14	cost= 33+14 = 47
17	counter= n-1, 여기에서 stop. Edge 17, 18, 20은 사용 못함	

Prim's algorithm

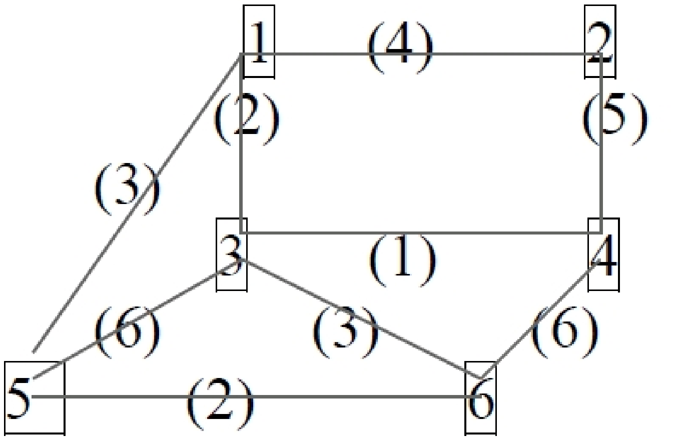
- Step 0: Pick any vertex as a starting vertex (call it *a*). T = {a}.
- Step 1:
  - Find the edge with smallest weight incident to *a*.
  - Add it to T Also include in T the next vertex and call it *b*.
- Step 2: Find the edge of smallest weight incident to either *a* or *b.*

Include in T that edge and the next incident vertex. Call that vertex *c*.

- Step 3: Repeat Step 2, choosing the edge of smallest weight that does not form a cycle until all vertices are in T. The resulting subgraph T is a minimum spanning tree.



Prime Algorithm exercise



1) Start at vertex 1: produce edges with vertex1

(1,2)-4, (1,3)-2, (1,5)-3

2) select **(1,3) weight = 2**

produce edges with vertex3 and the remaining

(1,2) -4, (1,5)-3, (3,4)-1, (3,5)-6, (3,6)-3

3) select **(3,4) weight=2+1=3**

produce edges with vertex4 and the remainings

(1,2)-4, (1,5)-3,(2,4)-5, (3,5)-6, (3,6)-3,(4,6)-6

4) select (1,5) or (3,6), => we will select **(1,5) w= 3+3**

produce edges with vertex 5 and remaining

(1,2)-4, (2,4)-5, (3,6)-3,(4,6)-6, (5,6)-2

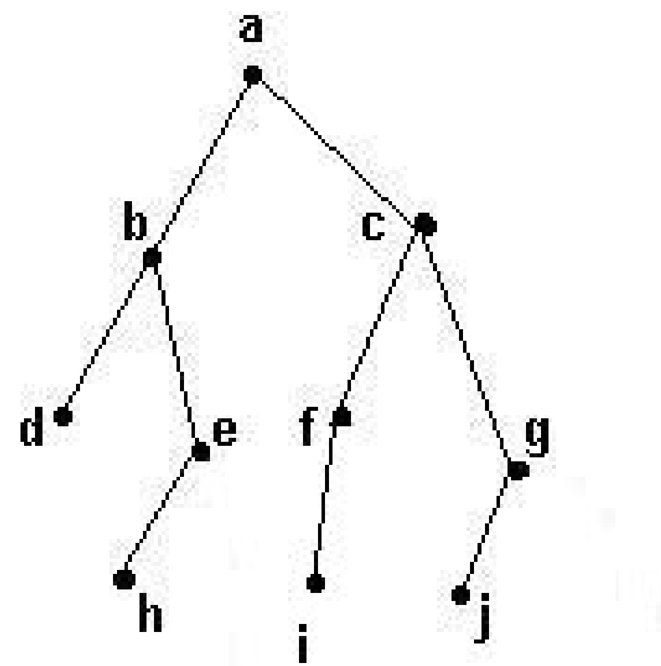
5) select (5,6) w = 6 + 2 = 8 produce edges (1,2)-4, (2,4)- 5

6) select (1,2) w= 8+4=12, which is least total w.

3. Binary trees

A ***binary tree*** is a **rooted tree** where each vertex has zero, one or two children.

And each child is designated as **left child** or **right child**.



Types of Binary Tree (BT)

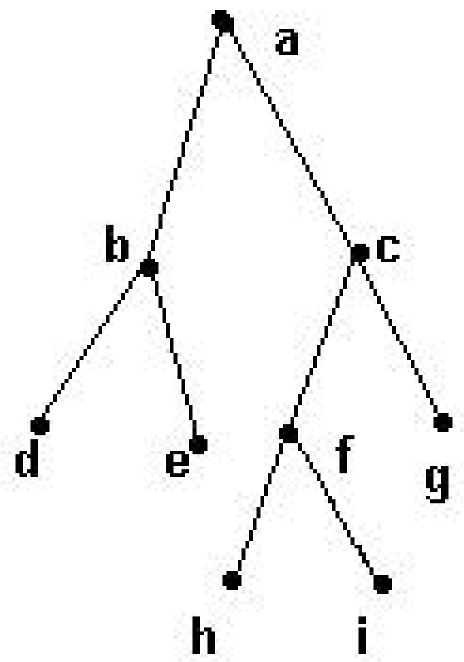
1) Skewed BT
2) Full BT
3) Complete BT

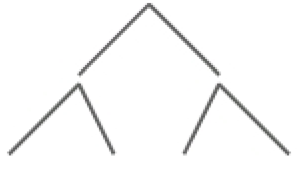
3.1 Properties of binary tree

- Full BT: A *full* binary tree is a binary tree in which each vertex has two or no children.

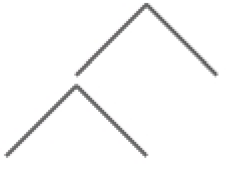
- Complete BT: BT is either full or full through

the next to last level, with the leaves on the last level as far as to the left as possible





Full and complete



complete

Properties of BT

Thm: If T is a full binary tree with n internal vertices, then T has n+1 terminal vertices and 2n+1 total vertices

(proof)

The vertices of T consists of children and non-children.

There is 1 non-child node →root.

There are **n** internal vertices, if each having two children, there are **2n** children.

Thus, the number of vertices of T is **2n+1** (including Root).

And the number of terminal vertices is **(2n+1) - n = n+1**

Properties of BT

We like to know maximum number of nodes in a BT of depth k

1) The maximum number of nodes on level I of a BT is => 2^{I-1},

2) The maximum number of nodes in a BT of depth k is => 2^{k}- 1,

(proof for 1)

. base: root is only node on level ***i=1, maximum nodes on level i is $2^{i-1}= 2^{0}=1$***

. hyp: For all j, 1<= j <i, maximum nodes on level ***j is => 2^{j-1},***

. induction: maximum nodes on level ***i-1 is => 2^{i-2} by HYP***

since every node in BT has a maximum degree of 2, the maximum nodes on level ***i is two times the maximum nodes on level i-1 or 2^{i-1}***

(Proof for 2)

$\sum_{i=1}^{k}$ (maximum number of nodes on level ***i) = $\sum 2^{I-1} = 2^{k}-1$***

➔ full binary tree of depth k has => 2^{k}-1 nodes

Properties of BT

- **Relation between number of leaf nodes and nodes of degree 2**

"For any BT, T, if n_0 is the number of leaf nodes and, n_2 the number of nodes of degree 2, then $n_0 = n_2+1$"

(sol)

Let n_1 be the number of nodes of degree 1.

N be the total number of nodes.

Since all the nodes in T are of degree at most 2,

we have: $n = n_0 + n_1 + n_2$

If B is the branches, then $n = B+1$

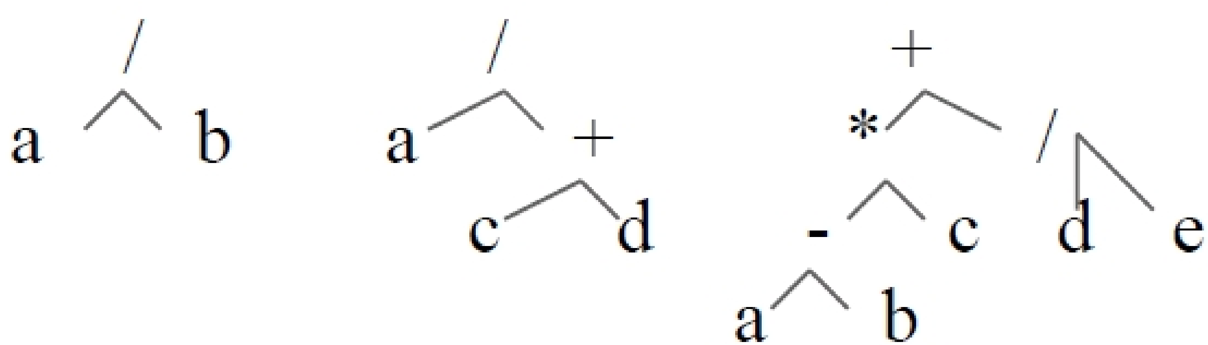
And all nodes stem from a node of degree one or two, then $B=n_1+2n_2$, So, we obtain $n = (n_1+2n_2)+1$,

➔ $n_0+n_1+n_2=n_1+2n_2+1$, Therefore, $n_0 = n_2+1$

Binary tree Representation

- Algebraic Expression involving Binary operation can be represented by an ordered rooted tree

ex) a / b   a / (c+d)   ((a-b) * c) + (d / e)



3.2 Tree Traversals

- Def: way to traverse a tree in a systematic way so that each vertex is visited exactly once.
- Methods: BFS, DFS, and Preorder, Postorder, Inorder methods
- Result - prefix(polish), postfix(reverse polish), infix notation
  - Preorder: Find Root(data), Find Leftchild, FindRightChild (DLR)
  - Inorder: Find Leftchild, Find Root, Find RightChild (LDR)
  - Postorder: Find Leftchild, Find RightChild, Find Root (LRD)

Algorithm 9.6.1: Preorder Traversal

Input: PT, the root of a binary tree

Output: Dependent on how "process" is interpreted in line 3

```
   preorder(PT) {
1.   if (PT is empty)
2.     return
3.   process PT
4.   l = left child of PT
5.   preorder(l)
6.   r = right child of PT
7.   preorder(r)
   }
```

Algorithm 9.6.3: Inorder Traversal

Input: PT, the root of a binary tree

Output: Dependent on how "process" is interpreted in line 5

```
   inorder(PT) {
1.     if (PT is empty)
2.         return
3.     l = left child of PT
4.     inorder(l)
5.     process PT
6.     r = right child of PT
7.     inorder(r)
   }
```

Algorithm 9.6.5: Postorder Traversal

Input: PT, the root of a binary tree

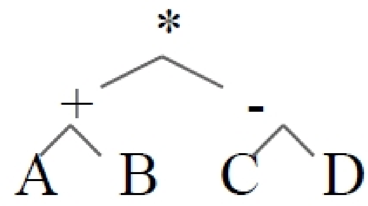
Output: Dependent on how "process" is interpreted in line 7

```
   postorder(PT) {
1.     if (PT is empty)
2.         return
3.     l = left child of PT
4.     postorder(l)
5.     r = right child of PT
6.     postorder(r)
7.     process PT
   }
```

Tree Traverse

1) Preorder

ex)

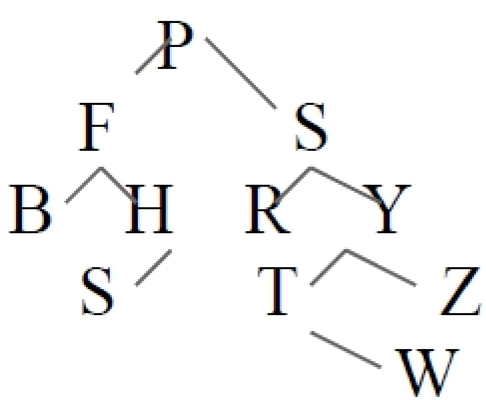


order of processing: * + A B - C D

2) **Inorder** of processing: (A+B) * (C-D)

3) **Postorder** of processing: AB+CD-*

Ex)



IN :

PRE :

POST:

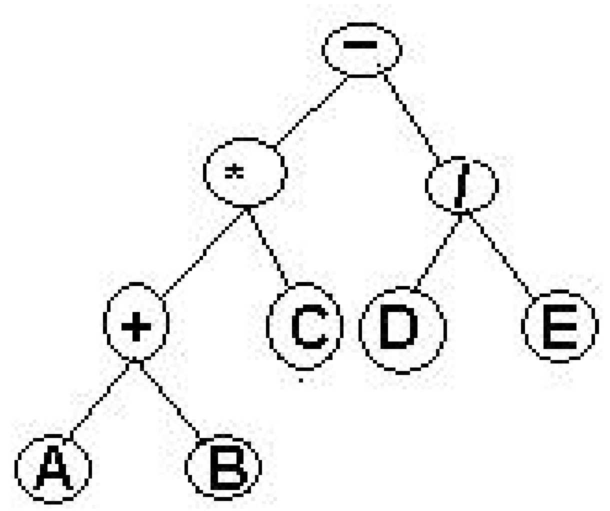
Arithmetic expressions

- Standard: *infix* form

  (A+B) ∗ C – D/ E

- Fully parenthesized form (in-order & parenthesis):

  (((A + B) ∗ C) – (D / E))



- *Postfix* form (reverse Polish notation): A B + C ∗ D E / -

- *Prefix* form (Polish notation):

  - ∗ + A B C / D E

Tree Traverse Exercise

ex) if A=1, B=2, C=3, what is the result of the postfix expression?

1) AB+C-

2) AB+CD*AA/--B*

3) ABAB*+*D*

4) ADBCD*-+*

ex) Draw the postfix form as a BT, and transform as prefix form, and represent as fully parenthesized infix form. [Postfix form: ABC+-]

1) BT 2) prefix form: 3) infix form:

→ ABCD+*/E-, ABC**CDE+/-

ex) preorder of a tree is 'ABCEFD', and inorder is ACFEBD.

Draw the tree.

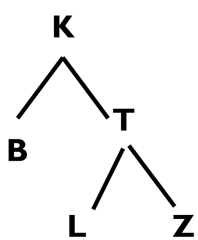
3.3 Binary Search Tree(BST)

- BST is the most effective way of searching data, and has the following characteristics

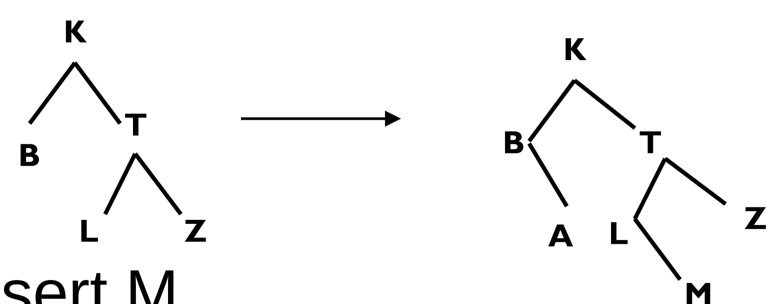
  . Left-subtree of Tree T : Store data if < root

  . Right-subtree of Tree T : Store data if > root

  It works same as the rest of the subtrees of Tree T



Search

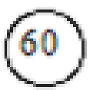


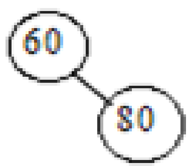
Insert M

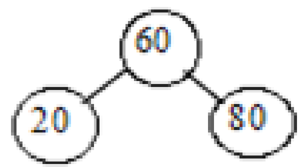
Binary Search Tree (BST)

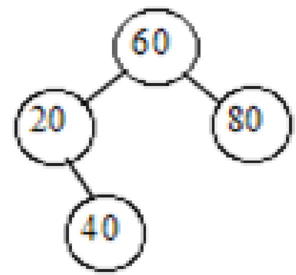
[example] Insert 60 80 20 40 90 10 70 in order

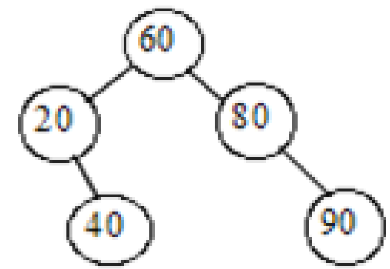
(1) insert 60 (2) insert 80 (3) insert 20 (4) insert 40 (5) insert 90



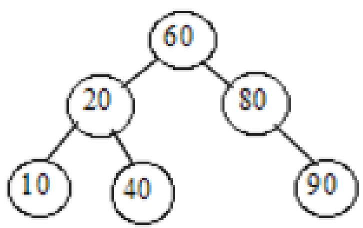




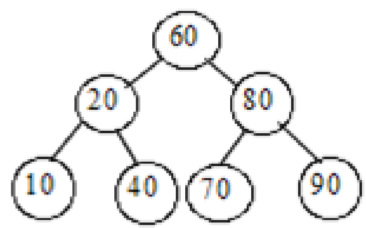




(6) insert 10



(7) insert 70



- Representation: same as BT
- Operation: same as tree traversal +additional (insert, delete, search)

BST (searching)

```
search (tree-ptr ptr,  int key)
{
  if (ptr = NULL)   return NULL;  //search unsuccessful
  else {
        if (key == p->data)  return ptr;
  else if (key < ptr->data)
          ptr = search (ptr->left, key);  //search leftsubtree
  else if (key > ptr->data)
          ptr= search(ptr->right, key);   //search rightsubtree
  }
  return ptr;
}
```

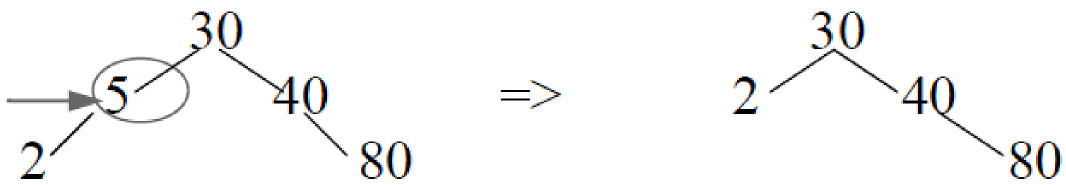
BST (inserting)

```
INSERT (ptr, key) //recursive version
{
  if (ptr=NULL) {        // create a new node with data
        create new_node(ptr);
        ptr->data = key;       ptr->left =  NULL;  ptr->right = NULL;
   }
   else if  (key < ptr->data)
         ptr->left =  INSERT(ptr->left, key);
   else if  (key >  ptr->data)
         ptr->right = INSERT(ptr->right, key);
   return ptr;
}
```

BST (deleting)

1) leaf node : set the child field of the node's parent pointer to NULL & free node

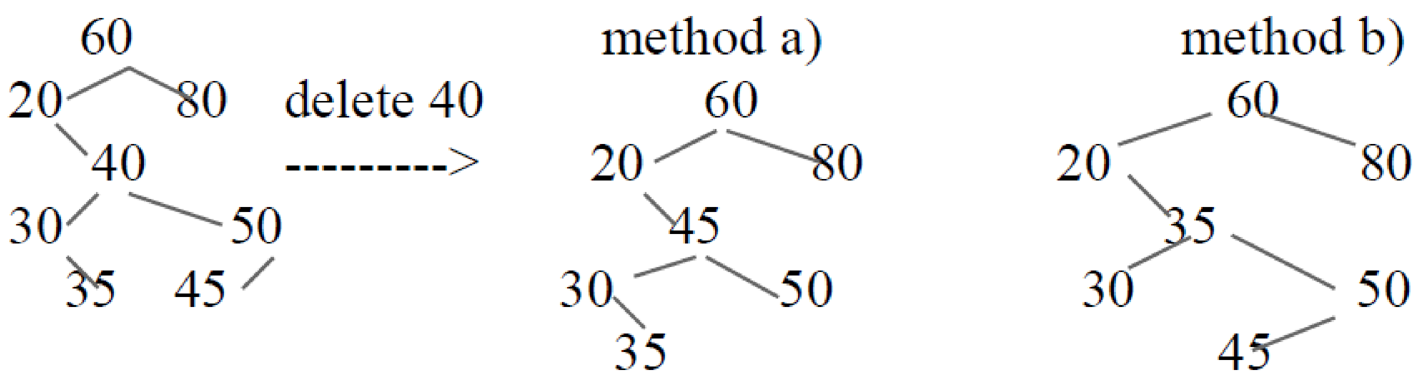
2) nonleaf node with one child: change pointer from parent to single child



3) Nonleaf node with two children

a. replace with smallest element in rightsubtree

b. replace with largest element in leftsubtree

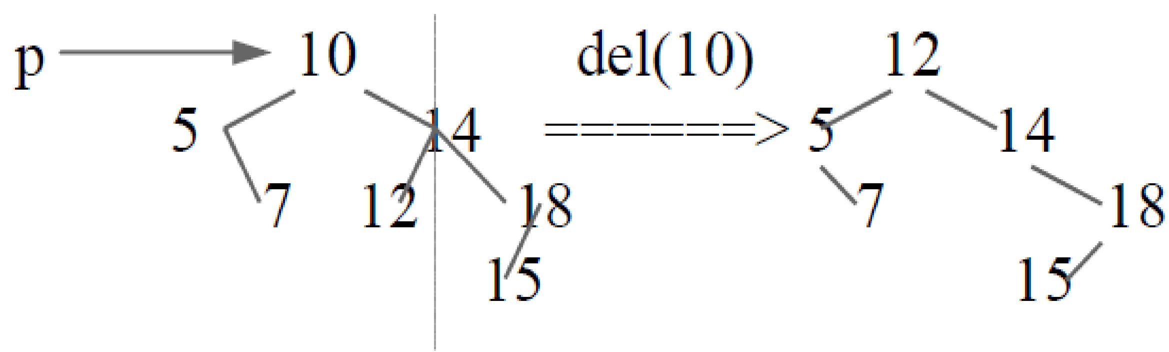


BST (deleting)

```
delete (key, ptr)  {
  if (key < ptr->data)
        ptr->left = delete(key, ptr->left)    /* move to the node  */
  else if (key > ptr->data)
        ptr->right = delete (key, ptr->right)  /* arrived at the node*/
  else if ((ptr->left == NULL) && (ptr->right==NULL))
        ptr=NULL                        /*leaf*/
  else if (ptr->left == NULL) {
         p = ptr;  ptr=ptr->right;  delete(p);  /rightchild only*/  }
  elseif  (ptr->right == NULL) {
         p = ptr;   ptr=ptr->left;  delete(p);  /*left child only */  }
  else  ptr->data = find_min(ptr->right) /*or find_max(ptr->left), both child exists */
}
```

BST (find-min algorithm)

```
int find_min(ptr)    /*right subtree에서 가장 작은것 선택 */
{
    if (ptr->left ==NULL)
    {
      temp = ptr->data;   ptr = ptr->right;
    }
    else  temp = find_min (ptr->left);
    return temp;
}
```

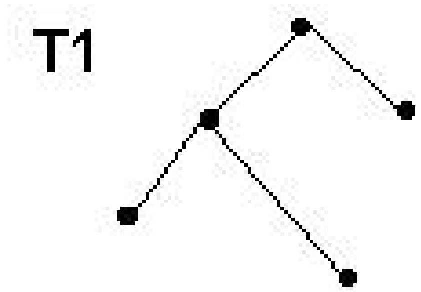


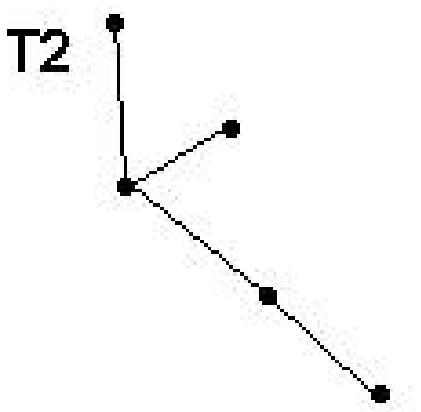
3.4 Isomorphism of trees

Given two trees T_1 and T_2

- T_1 is *isomorphic* to T_2

  - if we can find a one-to-one and onto function $f : T_1 \rightarrow T_2$
  - that preserves the adjacency relation
    - if $v, w \in V(T_1)$ and $e = (v, w)$ is an edge in T_1, then $e' = (f(v), f(w))$ is an edge in T_2.
    - Then we call the function "f" an Isomorphism



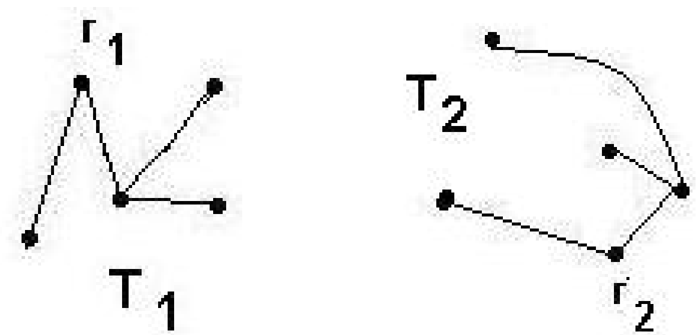


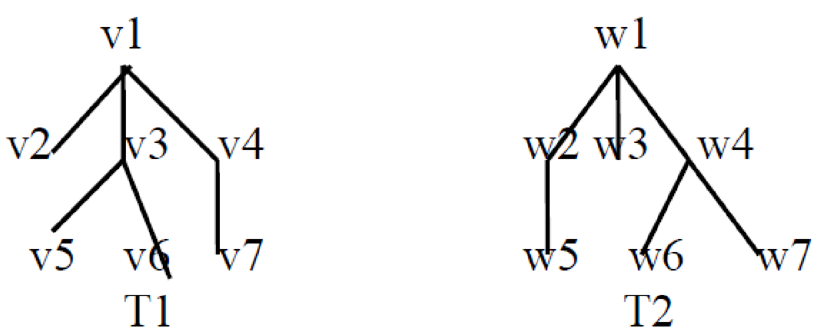
Isomorphism of rooted trees

Let T_1 and T_2 be rooted trees with roots r_1 and r_2, respectively. T_1 and T_2 are *isomorphic as rooted trees* if

- there is a one-to-one function $f: V(T_1) \rightarrow V(T_2)$ such that vertices v and w are adjacent in T_1 if and only if f(v) and f(w) are adjacent in T_2
- $f(r_1) = r_2$

Ex: T_1 and T_2 are isomorphic as rooted trees



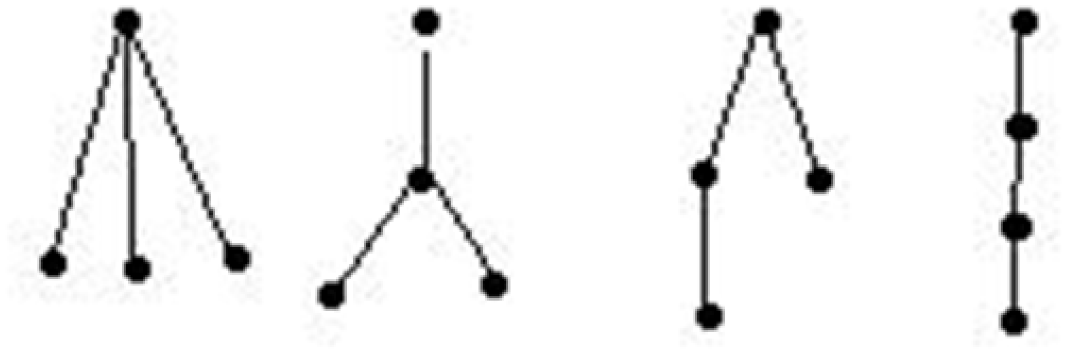


The rooted tree T1 and T2 are isomorphic.
An Isomorphism is f(v1) = w1, f(v2)= w3
f(v3)= w4 f(v4)= w2 f(v5)= w7, f(v6)= w6
f(v7)=w5

Non-isomorphism of rooted trees

Theorem: There are four non-isomorphic rooted trees with four vertices.

- The root is the top vertex in each tree.



Isomorphism of binary trees

Let T_1 and T_2 be binary trees with roots r_1 and r_2, respectively. T_1 and T_2 are <u>isomorphic as binary trees</u> if

a) T_1 and T_2 are isomorphic <u>as rooted trees</u> through an isomorphism f, and

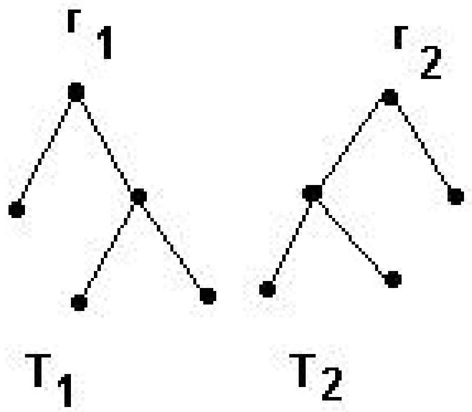
b) v is a left (right) child in T_1 if and only if f(v) is a left (right) child in T_2

- Note: Left children must be mapped onto left children and right children must be mapped onto right children.

Binary tree isomorphism

Example: the following two trees are

- isomorphic <u>as rooted trees</u>, but
- <u>not</u> isomorphic<u> as binary trees</u>



•Counting non-isomorphic BT with given N vertices can be done by Using CATALAN Numbers

$$C(2n, n)/(n+1) = 1, 2, 5, 14, 42, \ldots$$

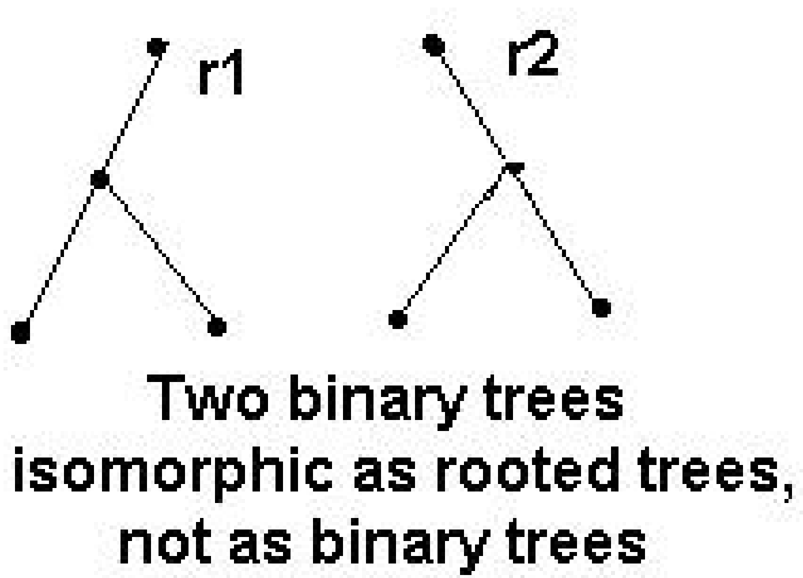
ex) if n=1, non-isomorphic BT = 1
if n=2, non-isomorphic BT = 2
if n=3, non-isomorphic BT = 5 …

Summary of tree isomorphism

There are 3 kinds of tree isomorphism

- Isomorphism of trees
- Isomorphism of rooted trees (root goes to root)
- Isomorphism of binary trees

(left children goes to left children, right children goes to right children)



Non-isomorphism of trees

- Many times it may be easier to determine when two trees are <u>not</u> isomorphic rather than to show their isomorphism.

- **<u>A tree isomorphism</u>** must respect certain properties, such as
  - the number of vertices
  - the number of edges
  - the degrees of corresponding vertices
  - roots must go to roots
  - position of children, etc.

Game trees

Trees can be used to analyze all possible move sequences in a game:

- Vertices are positions. An edge represents a move
- A path represents a sequence of moves

Ex)Tic-Tac-Toe