6 트리(Tree)

컴퓨터AI공학부
천세진

1

목차

2

미리보기

트리를 왜 배워야 하는가?

- 트리는 '특별한 형태의 한 관계(relation)' 라고 간단히 정의할 수 있다.
- 트리는 그래프에서 가장 중요한 클래스이다.
- 가계도와 회사의 조직도, 컴퓨터 폴더 혹은 디렉토리 구조, 의사결정 트리 구조 등을 나타낼 때 유용하게 사용되는 계층적 자료구조이다.
- 자료 저장, 데이터베이스 구성이나 언어 번역기 등에 대단히 유용하게 쓰이는 개념이다.

미리보기

트리의 응용 분야

- 트리는 독일의 물리학자 키르히호프(Gustav R. Kirchhoff)에 의해 1847년에 회로이론에 처음 사용되었다.
- 영국의 수학자 케일리(Arthur Cayley)는 1857년에 탄화수소 계열의 물질들을 표현하기 위해 트리를 사용하였다.
- 최적화 문제, 알고리즘, 자료의 탐색(searching) 및 정렬(sorting), 문법의 파싱, 결정 트리, 게임 트리 등의 IT 분야에 많이 활용되고 있다.

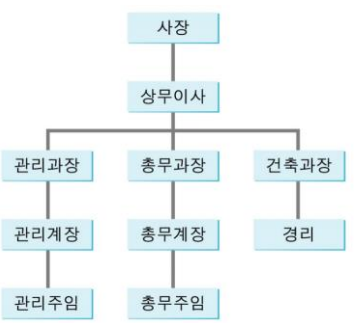
미리보기

이 장에서 배우는 내용

❖ 트리의 개념, 레이블을 갖는 트리, 최소 스패닝 트리, 트리 탐방 알고리즘에 대해 알아본다.

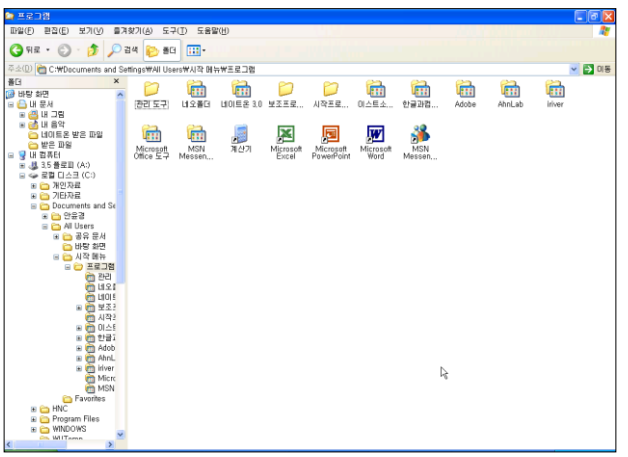
미리보기

트리 구조 예 – 회사의 조직도

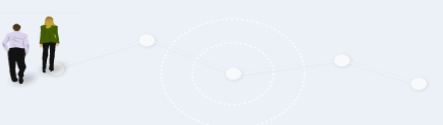


미리보기

트리 구조 예 – 컴퓨터 폴더 구조



8.1 트리의 기본 개념

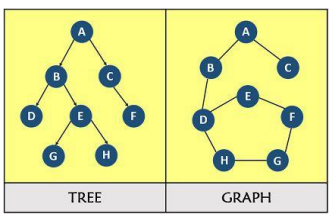


트리의 기본 개념

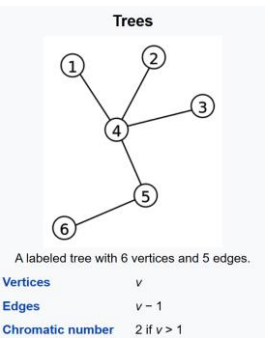
정의 8-1 트리

A를 유한집합, T를 A 위에서의 한 관계라고 하자. 이때 A 안에 적당한 정점 v_0이 존재해서 v_0에서 $A-\{v_0\}$의 모든 정점으로의 유일한 경로가 존재하고 v_0에서 v_0으로의 경로는 존재하지 않을 때, T를 A 위에서의 **트리**[tree] 혹은 **수형도**라고 한다.

- A tree is a connected graph without cycles
- A tree is a connected graph on n vertices with n-1 edges
- A graph is a tree if and only if there is a unique simple path but between any pair of its vertices
- 이때의 v_0를 그 트리의 루트(root) 노드라 하며, 트리를 간단히 (T, v_0)로 표현한다.



그림출처 : https://techdifferences.com/difference-between-tree-and-graph.html



트리의 기본 개념

정리 8-1 트리의 성질

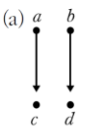
(T, v_0)을 트리라 할 때 다음이 성립한다.

(1) T에는 어떤 순환도 존재하지 않는다.
(2) v_0은 유일한 루트 노드이다.
(3) v_0의 내차수는 0이고 나머지 정점들의 내차수는 1이다.

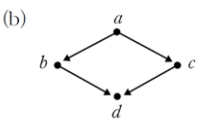
진입 차수(in-degree) : 방향 그래프에서 외부에서 오는 간선의 수 (내차수라고도 부름)
진출 차수(out-degree) : 방향 그래프에서 외부로 향하는 간선의 수 (외차수라고도 부름) 방향 그래프에 있는 정점의 진입 차수 또는 진출 차수의 합 = 방향 그래프의 간선의 수(내차수 + 외차수)

예제 8-1 트리 판단하기

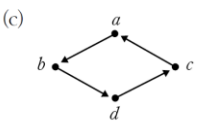
다음 유향 그래프가 트리인지 판단하라.



내차수 0인 2개의 정점을 가지므로 트리가 아님



내차수 2인 정점(d) 있으므로 트리가 아님



하나의 순환을 가지므로 트리가 아님

트리의 기본 개념

예제 8-2 트리의 루트 노드

다음의 같이 주어졌을 때 T는 트리가 된다. 이때 트리의 루트 노드를 구하라.

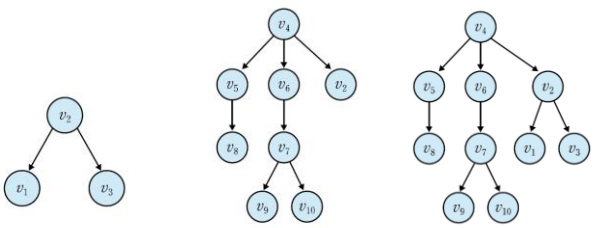
$A = \{v_1, v_2, v_3, \cdots, v_{10}\}$

$T = \{(v_2, v_3), (v_2, v_1), (v_4, v_5), (v_4, v_6), (v_5, v_8), (v_6, v_7), (v_4, v_2), (v_7, v_9), (v_7, v_{10})\}$

풀이

내차수를 따져가며 정의를 이용해서 루트 노드를 구하는 것이 옳은 방법이겠지만, 여기서는 유향 그래프를 만들어 가면서 루트 노드를 구해보자.

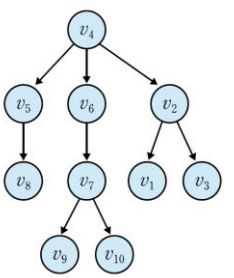
T의 순서대로 유향 그래프를 그려보면 그림 (a)와 그림 (b)를 얻을 수 있다.



이를 종합해보면 다음과 같이 된다. 따라서 루트 노드는 v_4가 된다.

트리의 기본 개념

트리와 관계된 용어



❖ **루트(root) 노드**

- 주어진 그래프의 시작 노드
- [그림 8-1]의 v_4가 루트 노드이다.

[그림 8-1] 유향 그래프

❖ **부모 노드(parent node)와 자식 노드(child node)**

- 정점 v와 정점 v_1을 $\{v \in A | (v_1, v) \in T\}$로 정의하면
  정점 v를 정점 v_1의 **자식 노드**라, 정점 v_1을 정점 v의 **부모 노드**라 한다.
- 한 부모에 여러 명의 자식 노드가 존재하면 그 자식 노드들을 **자식들(children)**이라 한다. 자식들 사이의 관계를 **형제 노드(sibling node)** 라 한다.
- [그림 8-1]에서 v_2는 v_1과 v_3의 부모 노드이고, v_1과 v_3는 v_2의 자식들이며 v_1과 v_3은 형제 노드라 한다.

트리의 기본 개념

❖ **Terminal node or Leaf node**

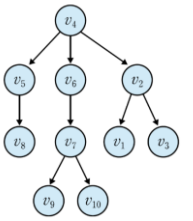
- 어떤 자식도 가지고 있지 않은 정점들
- [그림 8-1]의 $v_8, v_9, v_{10}, v_1, v_3$은 터미널 노드이다.

❖ **Level**

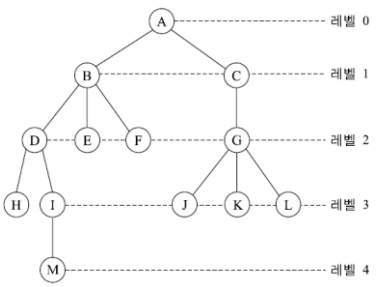
- 루트 노드의 레벨은 0으로 정의한다.
- 어떤 정점의 레벨이 i라면 그 정점의 자식들 레벨은 $i+1$이다.
- [그림 8-1]에서 v_4는 레벨 0, v_1과 v_3는 레벨 2, v_9와 v_{10}은 레벨 3이다.

❖ **Height**

- 높이는 정점들 중 최고의 레벨로 정의한다.
- [그림 8-1]의 높이는 3이다.



[그림 8-1] 유향 그래프



[그림 10-3] [그림 10-1]의 트리 T의 레벨

13/72

13

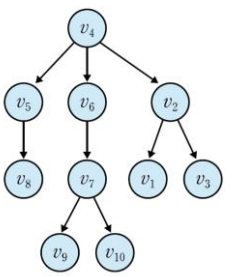
트리의 기본 개념

❖ **정점의 차수(degree)**

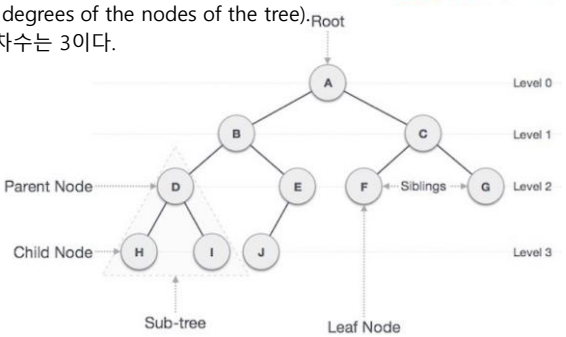
- 한 노드에 포함된 자식 노드의 개수
- 트리는 유향 그래프이든지 무향 그래프이든지 상관없다.
- [그림 8-1]의 v_4의 차수는 3이다.

❖ **트리의 차수**

- 모든 정점들의 차수 중 최대값 (Degree of the tree is the maximum of the degrees of the nodes of the tree).
- [그림 8-1]의 트리의 차수는 3이다.



[그림 8-1] 유향 그래프



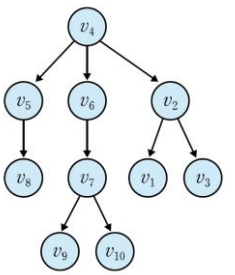
14/72

14

트리의 기본 개념

❖ n-트리

- 모든 정점들이 기껏해야 n개의 자식들을 가진 트리
- 터미널 노드들을 제외한 모든 정점들의 자식들이 정확히 n개일 때, **완전 n-트리**라고 한다.
- $n = 2$일 때 **2진 트리** 혹은 **완전 2진 트리**라고 한다.
- [그림 8-1]은 3-트리이다.



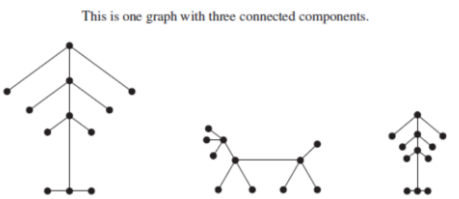
[그림 8-1] 유향 그래프

❖ 순서 트리(ordered tree)

- 트리의 유향 그래프에서 같은 레벨에서 어떤 정점의 자식이 많을 경우, 왼쪽부터 오른쪽으로 순서가 되어 있는 트리

❖ Forest

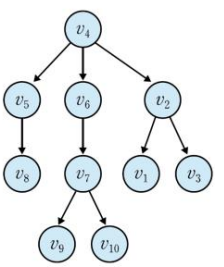
- A undirected graph in which any two vertices are connected by at most one path. Equivalently, a forest is an undirected acyclic graph



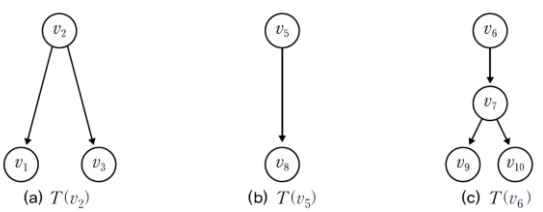
트리의 기본 개념

❖ 부분 트리(sub-tree)

- (T, v_0)를 v_0를 근으로 하는 A 위에서의 트리라 하고 $v \in A$인 v가 존재해서 $T(v)$를 v를 근으로 하는 트리라 하면 $T(v)$를 (T, v_0)의 부분 트리라 한다.
- [그림 8-1]에서 부분 트리 $T(v_2), T(v_5), T(v_6)$을 구해보면 [그림 8-2]와 같다.



[그림 8-1] 유향 그래프



[그림 8-2] [그림 8-1] 트리의 부분 트리

트리의 기본 개념

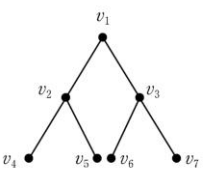
트리의 표현 방법

❖ 관례상 상단부에는 루트 노드를 그린다.

❖ 하단부 쪽으로 갈수록 레벨 순서에 따라 자식들을 그린다.

예제 8-3 정점의 레벨과 높이

다음의 트리를 보고 정점 v_1, v_2, v_3, v_4, v_5, v_6, v_7의 레벨과 높이를 구하라.



풀이

v_1, v_2, v_3, v_4, v_5, v_6, v_7의 레벨은 각각 0, 1, 1, 2, 2, 2, 2이다. 그리고 트리의 높이는 2이다.

트리의 기본 개념

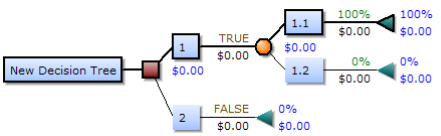
트리의 활용 방법

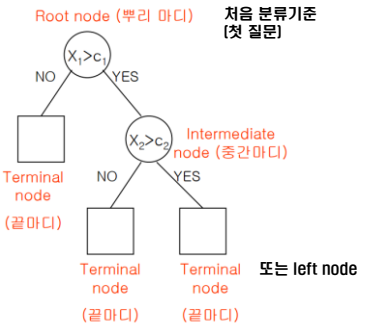
- 트리는 parsing할 때도 그대로 사용한다.
- 결정 트리(decision tree)를 사용하는 데 결정 트리는 알고리즘을 시각적으로 표현하여 의사를 결정하거나 시간 복잡도를 증명할 때 사용하는 트리이다.
- 가능성 있는 경우의 수를 효과적으로 표현하는 수단이다.

트리의 기본 개념

Decision tree

- A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called a decision tree.
- A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.





트리의 기본 개념

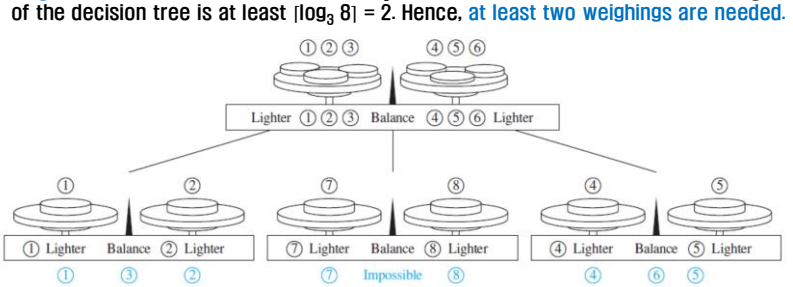
Example

Problem : Suppose there are seven coins, all with the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one?

Solution :

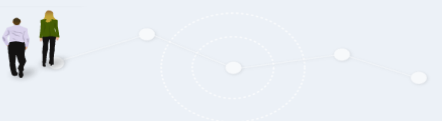
the decision tree for the sequence of weighings is a ***3-ary tree***. There are at least *eight leaves* in the decision tree because there are eight possible outcomes (because each of the eight coins can be the counterfeit lighter coin), and each possible outcome must be represented by at least one leaf.

The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree. From Corollary 1 of Section 11.1 it follows that the height of the decision tree is at least $\lceil \log_3 8 \rceil = 2$. Hence, at least two weighings are needed.



A decision tree for locating a counterfeit coin. The counterfeit coin is shown in color below each final weighing.

8.2 레이블을 갖는 트리



21

레이블을 갖는 트리

레이블을 갖는 트리

- A *labeled tree* is a tree in which each vertex is given a unique label. The vertices of a labeled tree on n vertices are typically given the labels 1, 2, ..., n. A *recursive tree* is a labeled rooted tree where the vertex labels respect the tree order (i.e., if u < v for two vertices u and v, then the label of u is smaller than the label of v).

- [알고리즘 8-1]은 정수 M, N, 그리고 양의 정수 MAX가 주어졌을 때 M^N 값을 계산하는 알고리즘이다.

- 단, subroutine PRINT가 존재한다고 가정했으며, M^N 값이 MAX보다 크면 에러(error)를 출력하게 되어 있다.

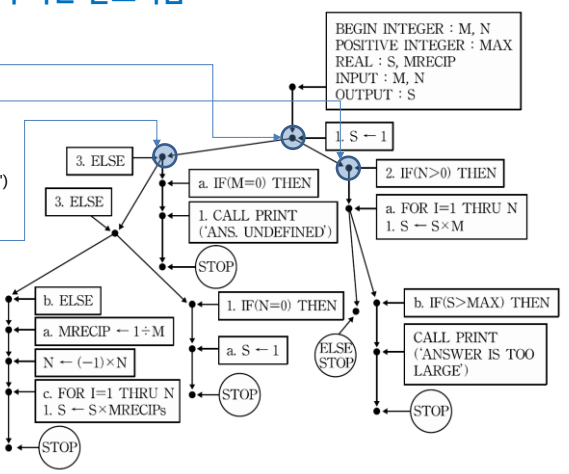
22/72

22

레이블을 갖는 트리

[알고리즘 8-1] M^N의 값을 구하는 알고리즘

```
1. S←1
2. IF (N>0) THEN
   a. FOR I=1 THRU N
      1. S←S×M
   b. IF (S>MAX) THEN
      1. CALL PRINT ('ANSWER IS TOO LAGER')
3. ELSE
   a. IF (M=0) THEN
      1. CALL PRINT ('ANS. UNDEFINED')
   b. ELSE
      1. IF (N=0) THEN
         a. S←1
      2. ELSE
         a. MRECIP←1/M
         b. N←(-1)×N
         c. FOR I=1 THRU N
            1. S←S×MRECIP
```



[그림 8-3] M^N의 값을 구하는 알고리즘을 레이블을 갖는 트리로 표현

레이블을 갖는 트리

- 트리에 자료를 저장하기 위해서는 선형의 산술식을 저장해야 한다.
- 선형의 산술식이 있을 때 이를 트리로 저장하기 위해서는 산술식이 어떻게 계산되는지를 알아야 한다.
- 이때 가장 필요한 요소가 2장에서 설명한 연산자 우선순위와 결합법칙이다.
- 여기서는 괄호를 이용하여 연산자 우선순위가 정해져 있을 때 이를 트리로 저장하는 방법을 설명한다.

- 괄호로 묶여진 산술식 (3 - (2*x)) + ((x - 2) - (3 + x))를 생각해보자.
- 이 산술식을 트리에 저장하기 위해서는 중심 연산자가 필요하다.
- 산술식에서 마지막으로 연산이 되는 연산자(이 식에서는 +)를 그 산술식의 중심 연산자(central operator)라고 한다.
- 이러한 중심 연산자를 알면 [알고리즘 8-2]와 같은 방법으로 산술식을 트리로 표현할 수 있다.

레이블을 갖는 트리

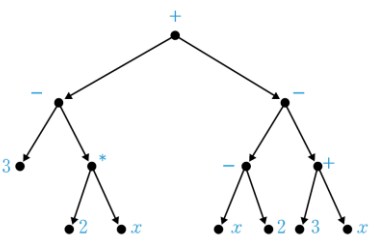
[알고리즘 8-2] 산술식으로부터 트리로 변환

① 트리의 루트 노드를 전체 산술식의 중심 연산자로 한다.

② ①의 루트 노드에 대해서 중심 연산자가 이항 연산자라고 하면 루트 노드의 왼쪽 부분식과 오른쪽 부분식이 존재한다. 그러면 루트 노드에 대해서 왼쪽 부분식은 왼쪽 부분 트리에 오른쪽 부분식은 오른쪽 부분 트리로 구성한다. 다시 왼쪽 부분식과 오른쪽 부분식에 대해서 재귀적으로 ①과 ②를 재귀적으로 행한다.

③ 이런 방법으로 상수나 변수 하나로 된 터미널 노드가 나올 때까지 계속한다.

- 이 방법을 이용하여 트리로 표현하면 [그림 8-4]와 같다.

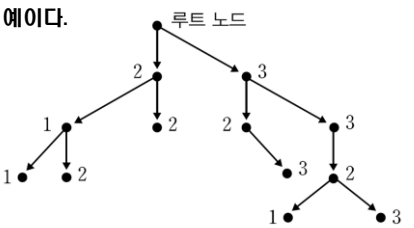


[그림 8-4] 산술식 $(3-(2*x))+((x-2)-(3+x))$를 트리로 표현

레이블을 갖는 트리

n-순서 트리(ordered tree)의 유향 그래프 그리기

- 이때 각 정점의 자식들은 n개 모두 있다고 생각하고 각 자식들의 순서에 대한 위치를 고정시켜보자.
- 그리고는 집합 $\{1, 2, 3, \ldots, n\}$의 원소들로 레이블을 붙이되 고정된 위치에 맞는 원소를 골라 붙여보자.
- 이런 유향 그래프를 위치와 레이블을 갖는 유향 그래프(positional-labeled digraph)라고 한다.
- 이것은 자료 구조 중 트리 구조로 컴퓨터에 기억시킬 때 매우 중요하다.
- [그림 8-5]는 3-순서 트리의 유향 그래프의 예이다.



[그림 8-5] 3-순서 트리의 유향 그래프의 예

Spanning Tree

Spanning tree

- 7장에서 가중치 그래프의 응용에 있어서, 그래프 G 안에 있는 모든 정점을 다 포함하면서 트리가 되는 연결 부분 그래프가 필요하다는 것을 살펴보았다.
- 이런 트리를 spanning tree라고 하고, 컴퓨터 인터넷 네트워크와 자동차 내비게이션 등에 많이 사용된다.

정의 8-2 스패닝 트리

그래프 G의 부분 그래프 T가 트리이고, T가 G의 모든 정점들을 포함한다면 T를 그래프 G의 **스패닝 트리**spanning tree라고 한다.

- 그래프 G의 spanning tree는 유일하지 않고 구성하는 방법도 다양하지만, 대표적인 방법으로 너비우선 탐색 방법과 깊이우선 탐색 방법을 사용해서 구성하는 방법이 있다.
- 이렇게 구한 spanning tree를 각각 너비우선 spanning tree와 깊이우선 spanning tree라고 한다.

Spanning Tree

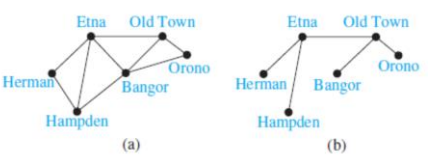


Figure 1. (a) A road system and (b) a set of roads to plow

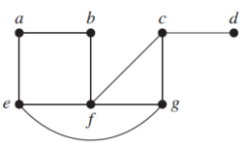


Figure 2. The simple graph G

❖ Find a spanning tree of the simple graph G shown in Figure 2.

Solution: The graph G is connected, but it is not a tree because it contains simple circuits. *Remove the edge {a, e}*. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of G. Next *remove the edge {e, f }* to eliminate a second simple circuit. Finally, remove edge {c, g} to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of G.

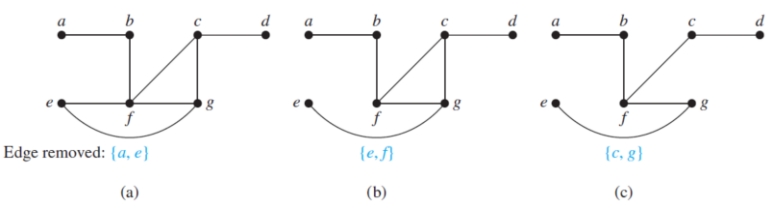


Figure 3. Producing a spanning tree for G by removing edges that form simple circuits.

It is not only spanning tree of graph G

Spanning Tree

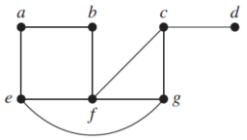


Figure 2. The simple graph G

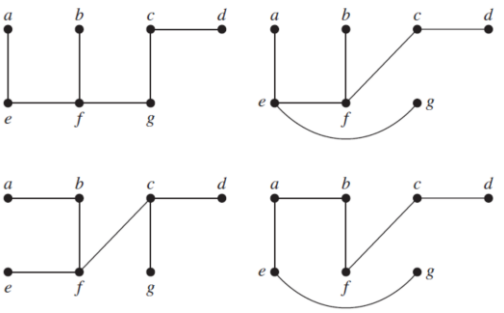


Figure 4. Spanning trees of graph G

Spanning Tree

- Depth-First Search

ALGORITHM 1 Depth-First Search.

procedure *DFS*(*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(*v*: vertex of *G*)
for each vertex *w* adjacent to *v* and not yet in *T*
  add vertex *w* and edge {*v*, *w*} to *T*
  visit(*w*)

- Breadth-First Search

ALGORITHM 2 Breadth-First Search.

procedure *BFS*(*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex v_1
L := empty list
put v_1 in the list L of unprocessed vertices
while L is not empty
  remove the first vertex, *v*, from L
  for each neighbor *w* of *v*
    if *w* is not in L and not in T **then**
      add *w* to the end of the list L
      add *w* and edge {*v*, *w*} to T

Spanning Tree

Example : Use the Depth-First Search to find a spanning tree of the Graph G

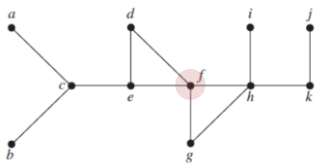


Figure 6. Graph G

Solve:

- Arbitrarily start with the vertex f.
- A path is built by successively adding edges incident with vertices not already in the path.
- This produces a path f, g, h, k, j (note that other paths could have been built).
- Next, backtrack to k. There is no path beginning at k containing vertices not already visited.
- So we backtrack to h. Form the path h, i. Then backtrack to h, and then to f. From f build the path f, d, e, c, a. Then backtrack to c and form the path c, b. This produces the spanning tree.

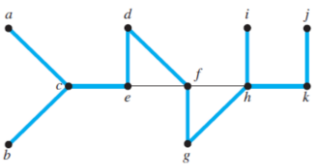


Figure 8. The tree edges and back edges (e-f, f-h) of the depth-first search

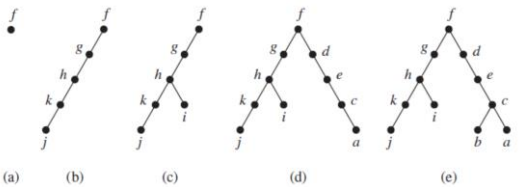


Figure 7. Depth-First Search of Graph G

Spanning Tree

Example : Use the Breadth-First Search to find a spanning tree of the Graph G

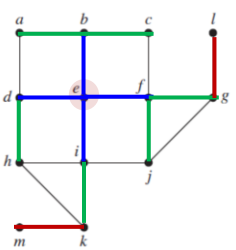


Figure 9. Graph G

Solve:

- Arbitrarily choose the vertex e to be the root.
- Then we add edges incident with all vertices adjacent to e, so edges from e to b, d, f, and i are added. These four vertices are at level 1 in the tree.
- Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence, the edges from b to a and c are added, as are edges from d to h, from f to j and g, and from i to k. The new vertices a, c, h, j, g, and k are at level 2.
- Next, add edges from these vertices to adjacent vertices not already in the graph. This adds edges from g to l and from k to m.

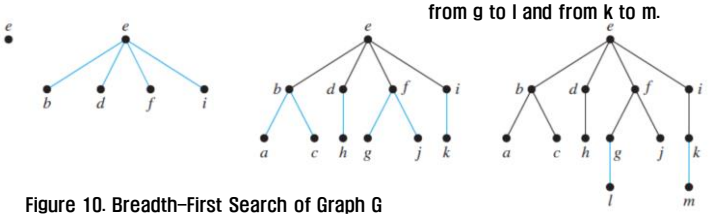


Figure 10. Breadth-First Search of Graph G

MST (minimal spanning tree)

정의 8-3 최소 스패닝 트리

임의의 그래프 G가 가중치 그래프라고 할 때, G로부터 생성된 스패닝 트리 가운데 각 간선의 가중치 합이 최소가 되는 것을 **최소 스패닝 트리**minimal spanning tree; MST라고 한다.

m개의 정점을 갖는 가중치 그래프 $G = (V, E)$에서 MST를 찾기 위한 두 가지 방법

- **Prim's algorithm**
- **Kruskal's algorithm**

MST (minimal spanning tree)

❖ **Prim's algorithm**

- **그래프 $G = (V, E)$에서 최소 가중치를 가지는 간선 $\{u, v\}$를 선택하고, 이 간선을 MST에 포함시킨다.**
- **그런 다음 정점 u또는 v에 연결된 간선 중에서 최소 가중치를 가지는 간선을 선택하여 MST에 포함시킨다.**
- **물론 이때 순회가 되지 않아야 한다.**

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

MST (minimal spanning tree)

Example : Use Prim's algorithm to find a minimum spanning tree in the graph

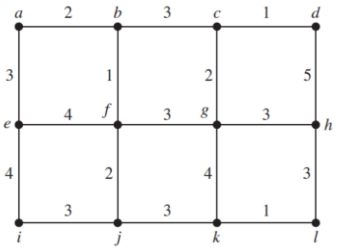
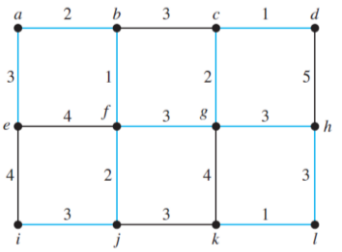


Figure 1. Graph G



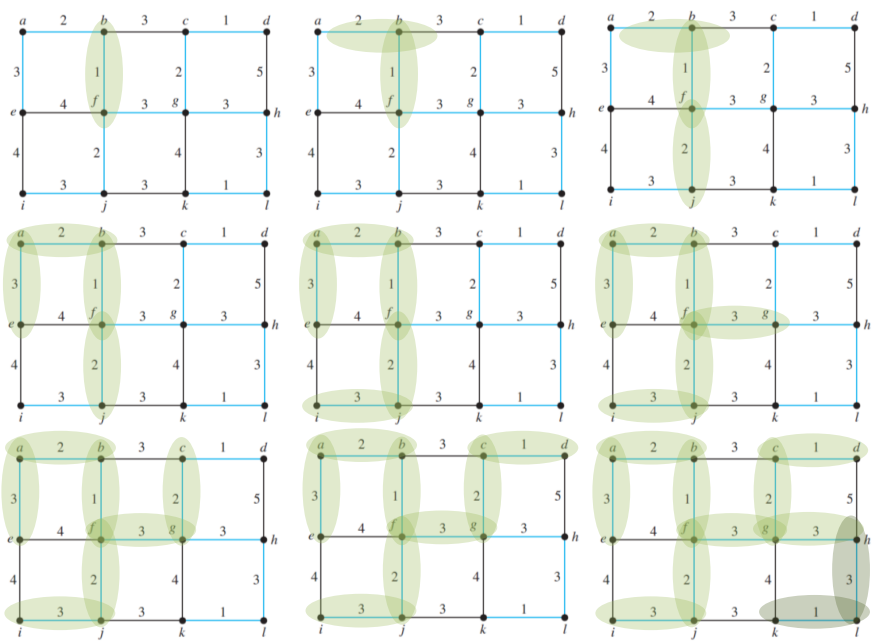
Choice	Edge	Weight
1	{b, f}	1
2	{a, b}	2
3	{f, j}	2
4	{a, e}	3
5	{i, j}	3
6	{f, g}	3
7	{c, g}	2
8	{c, d}	1
9	{g, h}	3
10	{h, l}	3
11	{k, l}	1
	Total:	24

Figure 2. A minimum spanning tree produced using Prim's algorithm

35/72

35

MST (minimal spanning tree)



36/72

36

MST (minimal spanning tree)

❖ **Kruskal's algorithm**

- 무조건 최소 가중치 순으로 순서를 정한 뒤에 순환을 형성하지 않으면 그 간선을 선택하는 방법이다.

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
  e := any edge in G with smallest weight that does not form a simple circuit when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

MST (minimal spanning tree)

Example : Use Kruskal's algorithm to find a minimum spanning tree in the graph

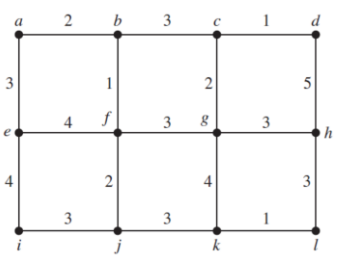


Figure 1. Graph G

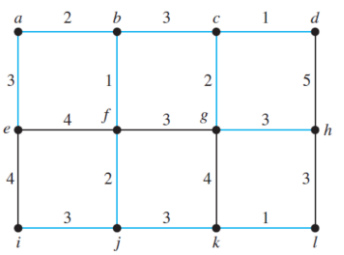
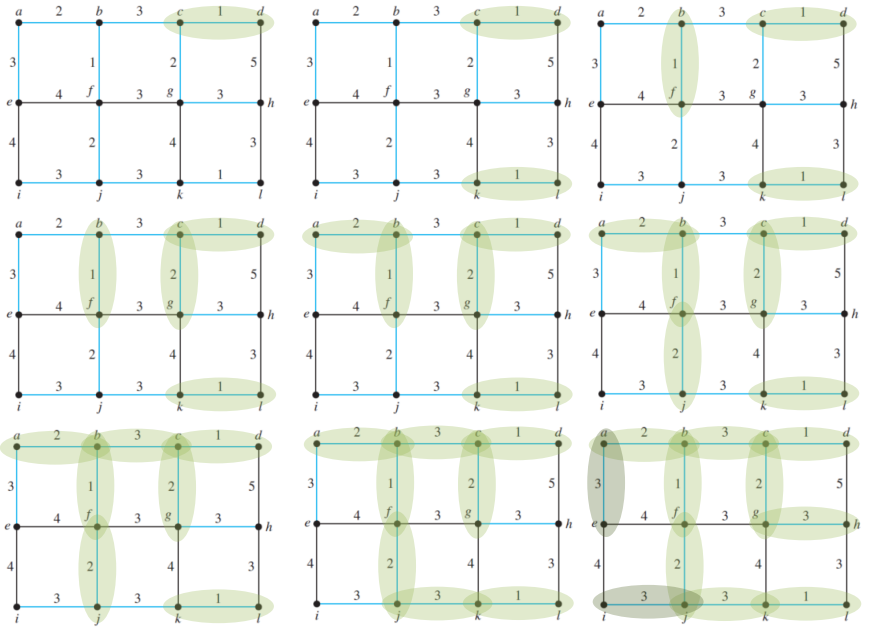


Figure 2. A minimum spanning tree produced using Prim's algorithm

Choice	Edge	Weight
1	{c, d}	1
2	{k, l}	1
3	{b, f}	1
4	{c, g}	2
5	{a, b}	2
6	{f, j}	2
7	{b, c}	3
8	{j, k}	3
9	{g, h}	3
10	{i, j}	3
11	{a, e}	3
	Total:	24

MST (minimal spanning tree)

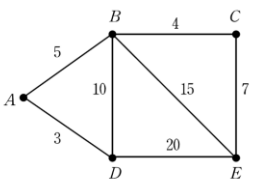


39

MST (minimal spanning tree)

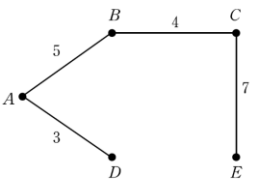
예제 8-5 최소 스패닝 트리(1)

다음의 가중치 그래프에서 프림의 방법으로 최소 스패닝 트리를 찾아라.



풀이

첫 번째 정점을 A로 선택한다. 여기서 간선 $\{A, B\}$와 $\{A, D\}$ 중 최소 가중치를 가지는 간선은 $\{A, D\}$이므로 이를 선택한다. 다음으로 간선 $\{A, B\}$, $\{D, B\}$, $\{D, E\}$ 중 최소 가중치를 가지는 간선은 $\{A, B\}$이므로 이를 선택한다. 동일한 방법으로 $\{B, C\}$, $\{C, E\}$를 선택하면 $U = V$이므로 종결된다. 따라서 최소 스패닝 트리는 다음 그림과 같다.

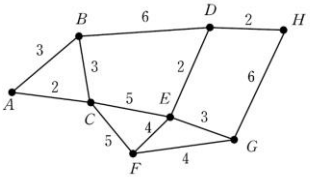


40

MST (minimal spanning tree)

예제 8-6 최소 스패닝 트리(2)

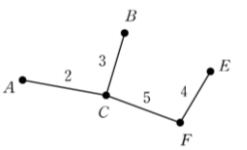
다음의 가중치 그래프에서 프림의 방법으로 A로부터의 최소 스패닝 트리와 E로부터의 최소 스패닝 트리를 찾아라.



풀이

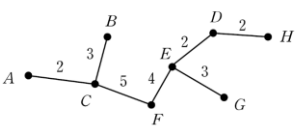
- A로부터 최소 스패닝 트리 찾기

  첫 번째 정점을 A로 선택한다. 여기서 간선 $\{A, B\}$와 $\{A, C\}$ 중 최소 가중치를 가지는 간선은 $\{A, C\}$이므로 이를 선택한다. 다음으로 $\{A, C\}$로부터 연결된 간선들 $\{A, B\}$, $\{C, B\}$, $\{C, E\}$, $\{C, F\}$ 중 최소 가중치를 가지는 간선은 $\{A, B\}$나 $\{C, B\}$이다. 먼저 $\{C, B\}$를 선택하자. 다음으로는 $\{A, B\}$, $\{B, D\}$, $\{C, E\}$, $\{C, F\}$ 중 최소 가중치는 $\{A, B\}$이지만 순환이 되므로 선택하면 안 된다. 그러면 $\{C, E\}$나 $\{C, F\}$가 가능하므로 둘 중 아무거나 선택할 수 있다. 이번에는 $\{C, F\}$를 선택하자. 그 다음으로는 $\{B, D\}$, $\{C, E\}$, $\{F, E\}$, $\{F, G\}$가 가능하다. 그 중 $\{F, E\}$나 $\{F, G\}$가 최소 가중치를 가진다. 이번에는 $\{F, E\}$를 선택하자. 현재까지의 최소 스패닝 트리는 다음과 같다.



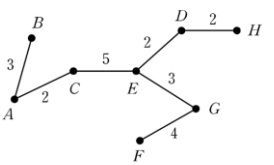
MST (minimal spanning tree)

같은 방법으로, $\{B, D\}$, $\{C, E\}$, $\{E, D\}$, $\{E, G\}$, $\{F, G\}$가 가능하다. 최소 가중치는 $\{E, D\}$이다. 다음으로는 $\{B, D\}$, $\{C, E\}$, $\{D, H\}$, $\{E, G\}$, $\{F, G\}$가 가능하다. 최소 가중치는 $\{D, H\}$이다. 다음으로는 $\{B, D\}$, $\{C, E\}$, $\{E, G\}$, $\{F, G\}$, $\{H, G\}$가 가능하다. 최소 가중치는 $\{E, G\}$이다. 그러므로 A로부터의 최소 스패닝 트리는 다음과 같다.



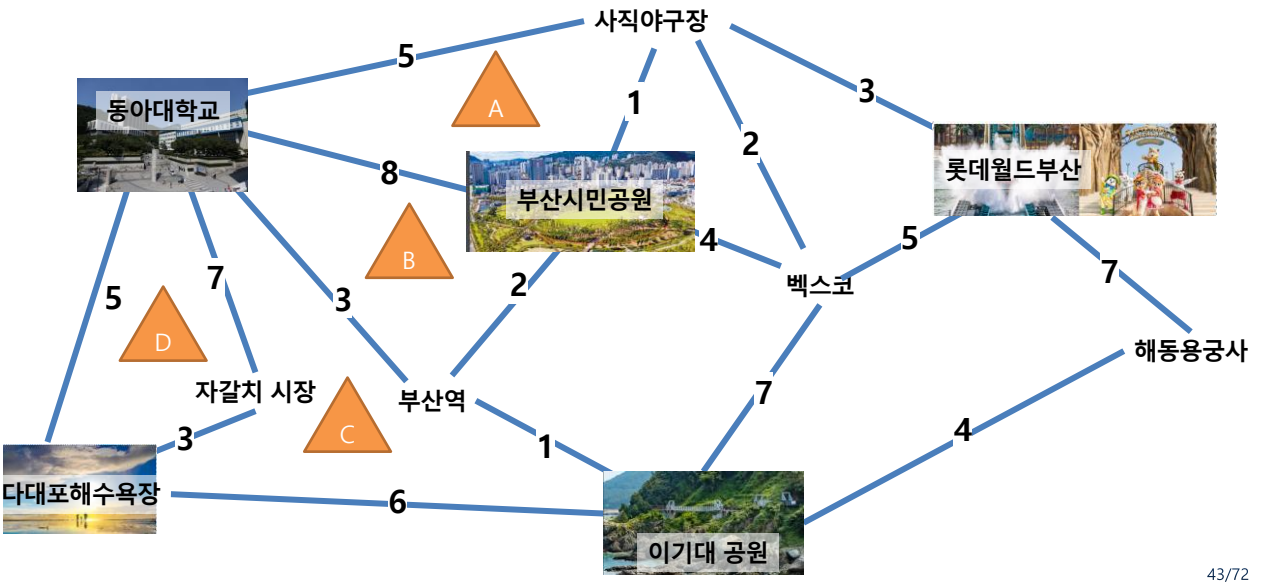
- E로부터 최소 스패닝 트리 찾기

  같은 방법으로, 다음과 같은 최소 스패닝 트리를 얻을 수 있다.



나의 원룸 입지 구하기

A or B or C or D 에 집을 구하는 경우, 둘러싸인 그래프



43/72

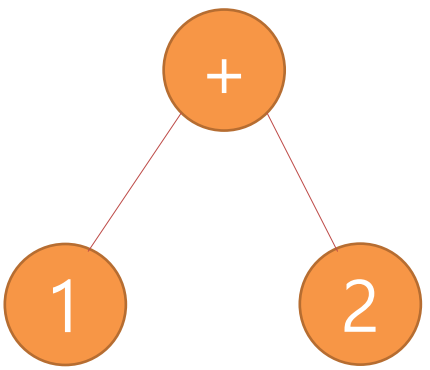
43

Chapter 8 트리

8.3 Tree Traversal Algorithm

44

Tree Traversal Algorithm



- Preorder traversal: +12
  - 전위순회
- Inorder traversal: 1+2
  - 중위순회
- Postorder traversal: 12+
  - 후위순회

연산자 트리
(Leaf node는 숫자, Root/중간 노드들은 연산자)

Tree Traversal Algorithm

Tree traversal algorithm

- 트리에서 필요한 자료를 찾고자 할 때 사용되는 알고리즘

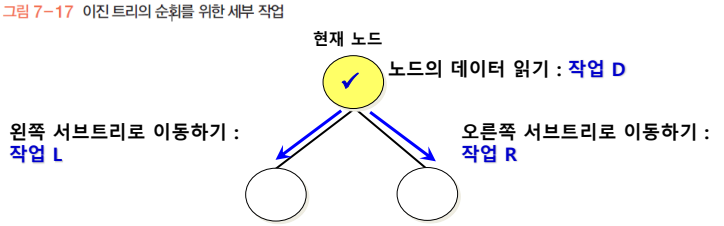
Binary Tree traversal

- 이진 트리에서 모든 원소를 (중복하지 않고) 처리하는 연산

- 작업 D : 현재 노드를 방문하여 처리한다.
- 작업 L : 현재 노드의 왼쪽 서브 트리로 이동한다.
- 작업 R : 현재 노드의 오른쪽 서브 트리로 이동한다.

- 전위 순회(preorder traversal),
- 중위 순회(inorder traversal),
- 후위 순회(postorder traversal)

그림 7-17 이진 트리의 순회를 위한 세부 작업



Tree Traversal Algorithm

전위 순회 (Preorder Traversal)

- D → L → R 순서로, 현재 노드를 방문하여 처리하는 작업 D를 가장 먼저 수행

❶ 작업 D : 현재 노드 n을 처리한다.
❷ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.
❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

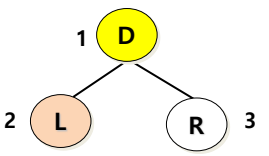


그림 7-18 이진 트리의 전위 순회 작업 순서

알고리즘 7-1 이진 트리의 전위 순회

```
preorder(T)
   if (T≠NULL) then {
      visit T.data;
      preorder(T.left);
      preorder(T.right);
   }
end preorder()
```

ALGORITHM 1 Preorder Traversal.

procedure *preorder*(*T*: ordered rooted tree)
r := root of *T*
list *r*
for each child *c* of *r* from left to right
  T(*c*) := subtree with *c* as its root
  preorder(*T*(*c*))

Tree Traversal Algorithm

- Preorder Traversal Example

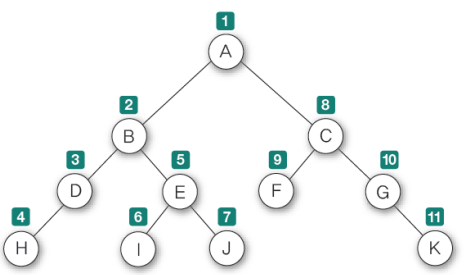


그림 7-19 이진 트리의 전위 순회 경로 : A-B-D-H-E-I-J-C-F-G-K

- 수식 A*B-C/D

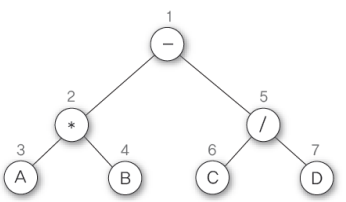


그림 7-20 수식에 대한 이진 트리의 전위 순회 경로 : −*AB/CD

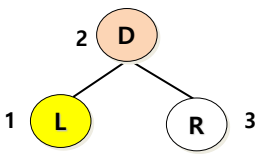
Tree Traversal Algorithm

중위 순회 (Inorder Traversal)

- L → D → R 순서로, 현재 노드를 방문하는 작업 D를 작업 L과 작업 R의 중간에 수행

❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.
❷ 작업 D : 현재 노드 n을 처리한다.
❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

그림 7-21 이진 트리의 중위 순회 작업 순서



알고리즘 7-2 이진 트리의 중위 순회

```
inorder(T)
   if (T≠NULL) then {
      inorder(T.left);
      visit T.data;
      inorder(T.right);
   }
end inorder()
```

ALGORITHM 2 Inorder Traversal.

```
procedure inorder(T: ordered rooted tree)
r := root of T
if r is a leaf then list r
else
   l := first child of r from left to right
   T(l) := subtree with l as its root
   inorder(T(l))
   list r
   for each child c of r except for l from left to right
      T(c) := subtree with c as its root
      inorder(T(c))
```


Tree Traversal Algorithm

- Inorder Traversal Example

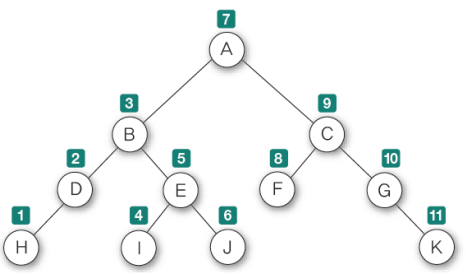


그림 7-22 이진 트리의 중위 순회 경로 : H-D-B-I-E-J-A-F-C-G-K

- 수식 A*B-C/D

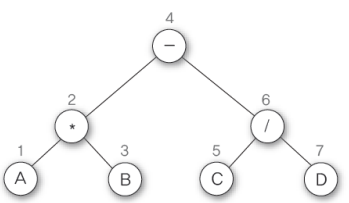
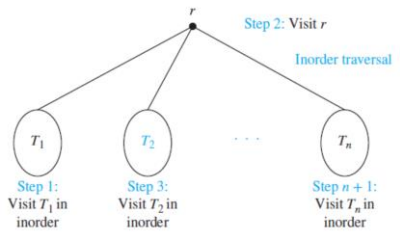
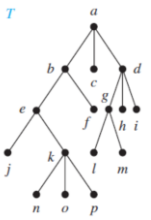


그림 7-23 수식에 대한 이진 트리의 중위 순회 경로 : A*B-C/D

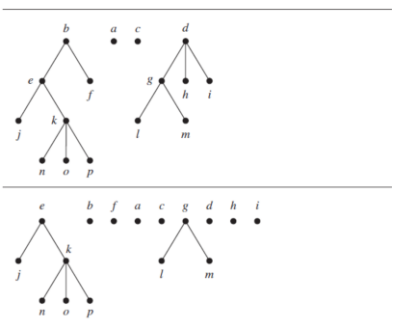
Tree Traversal Algorithm

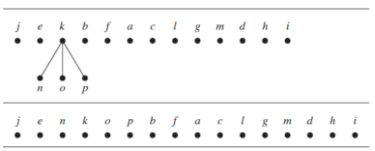
- Inorder traversal





Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right



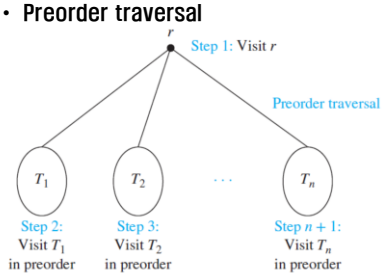


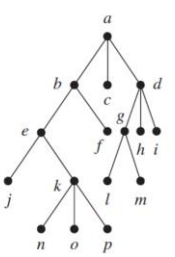
51/72

51

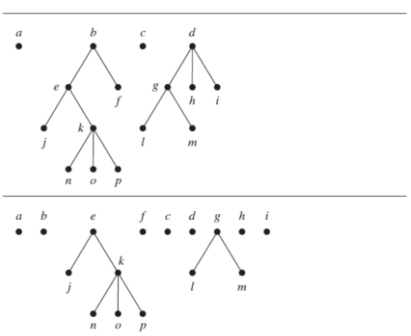
Tree Traversal Algorithm

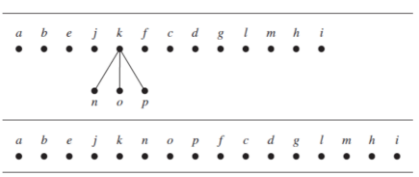
- Preorder traversal





Preorder traversal: Visit root, visit subtrees left to right





52/72

52

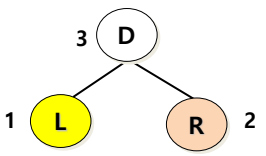
Tree Traversal Algorithm

후위 순회 (Postorder Traversal)

- L-R-D 순서로 현재 노드를 방문하는 D 작업을 가장 나중에 수행

❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.
❷ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.
❸ 작업 D : 현재 노드 n을 처리한다.

그림 7-24 이진 트리의 후위 순회 작업 순서



알고리즘 7-3 이진 트리의 후위 순회

```
postorder(T)
   if (T≠NULL) then {
      postorder(T.left);
      postorder(T.right);
      visit T.data;
   }
end postorder()
```

ALGORITHM 3 Postorder Traversal.

procedure *postorder*(*T*: ordered rooted tree)
r := root of *T*
for each child *c* of *r* from left to right
  T(*c*) := subtree with *c* as its root
  postorder(*T*(*c*))
list *r*

Tree Traversal Algorithm

- Postorder Traversal Example

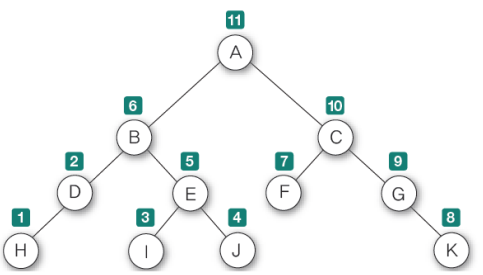


그림 7-25 이진 트리의 후위 순회 경로 : H-D-I-J-E-B-F-K-G-C-A

- 수식 A*B-C/D

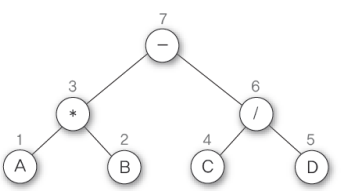
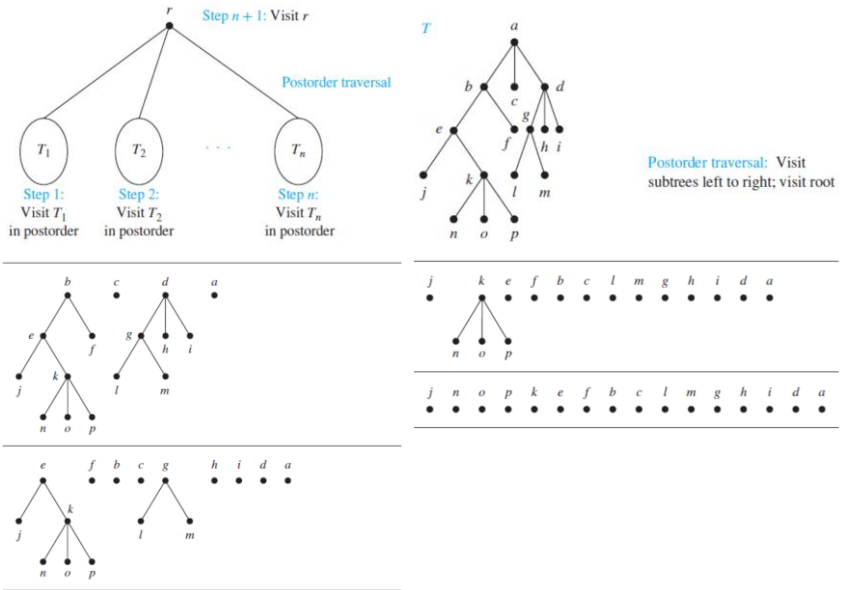


그림 7-26 수식에 대한 이진 트리의 후위 순회 경로 : AB*CD/-

Tree Traversal Algorithm

- Postorder traversal



55/72

55

Tree Traversal Algorithm

트리로 표현된 자료를 선형 순서로 표현하는 방법

❖ 전위 표기법(prefix)

- 트리로 표현된 자료를 전위 탐방하여 찾은 자료들을 선형으로 나열

❖ 후위 표기법(postfix)

- 후위 탐방하여 찾은 자료들을 선형으로 나열

❖ 중위 표기법(infix)

- 전위 탐방하여 찾은 자료들을 선형으로 나열

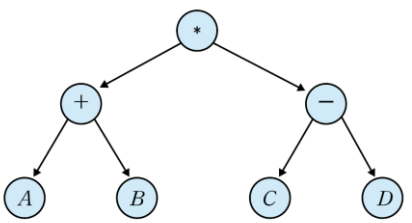
56/72

56

전위 표기법

전위 표기법(prefix)

- 일반 전위 표기법(ordinary prefix), 케임브리지 폴란드식 표기법(Cambridge polish), 폴란드식 표기법(polish)이라고도 하며, 이 표현들 사이에 약간의 차이가 있다.
- 예를 들어, 산술식에 대해 전위 표기법으로 표현한다면, 산술식에서 연산자(operator)를 먼저 쓰고 다음에 피연산자(operand)를 쓰는 표현법이다.
- [그림 8-8]과 같이 이진 트리로 산술식이 표현되었을 때 이를 일반 전위 표기법, 케임브리지 폴란드식 표기법, 폴란드식 표기법으로 표현해보자.



[그림 8-8] 산술식 $(A+B)*(C-D)$에 대한 이진 트리

전위 표기법

❖ 일반 전위 표기법

- 연산자를 먼저 쓰고, 그 연산에 따른 피연산자를 괄호로 묶어 표현하는 방법이다.
- [그림 8-8]을 이 방법으로 표현하면 ***(+(A, B), -(C, D))**

❖ 케임브리지 폴란드식 표기법

- 일반 전위 표기법에서 콤마(comma)를 제거하고 연산 기호를 괄호 안으로 묶는 방법으로서 프로그래밍 언어 LISP에서는 이 방법을 사용해서 언어를 표현한다.
- [그림 8-8]을 이 방법으로 표현하면 **(*(+AB)(-CD))**

❖ 폴란드식 표기법

- 케임브리지 폴란드식 표기법에서 괄호를 제거한 방법
- [그림 8-8]을 이 방법으로 표현하면 ***+AB-CD**
  - ✓ 일반적으로 전위 표기법을 표현하는 방법은 폴란드식 표기법 이다.

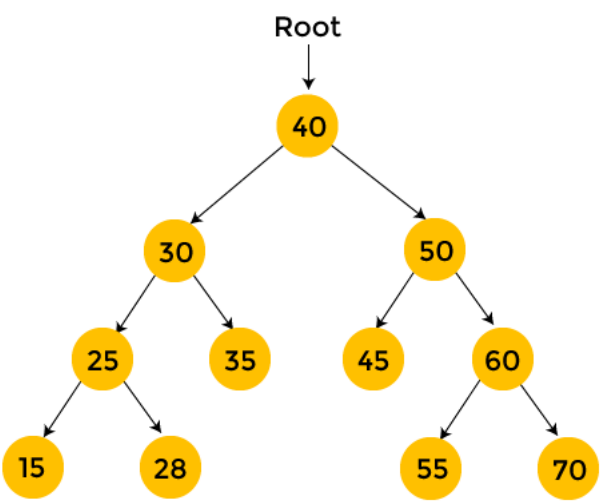
후위 표기법

후위 표기법(postfix)

- 역 폴란드식 표기법(reverse polish)이라고 하며, 피연산자를 먼저 쓰고 다음에 연산자를 쓰는 방법
- [그림 8-8]을 이 방법으로 표현하면 A B + C D - *

중위 표기법(infix)

- 이진 연산에서만 적당한 표현 방법으로 피연산자와 피연산자 사이에 연산자를 표현하는 방법
- [그림 8-8]을 이 방법으로 표현하면 A + B * C - D



Preorder
Inorder
Postorder

부 록

트리의 기본 개념

예제 8-4 트리 활용

8개의 코인 문제[8-coin problem]를 살펴보자. 8개의 코인은 모두 똑같은 색깔에 똑같은 모양이다. 그 중 1개만 다른 7개와 무게가 다르다. 무게가 같거나 무겁거나 가벼움만을 판정할 수 있는 천칭을 이용하여 단 3회만에 8개 중 어느 코인이 정상인 코인보다 무게가 무거운지 또는 가벼운지를 알아내는 결정 트리를 만들라.

풀이

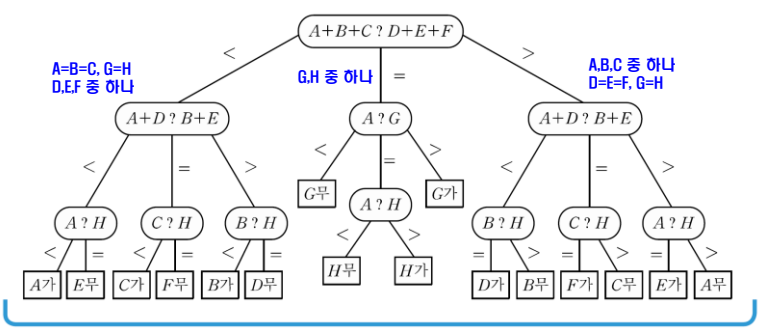
8개의 코인 중 무게가 다른 코인 하나를 천칭을 이용해서 판별하는 방법이다. 코인 8개를 A, B, C, D, E, F, G, H라 하자. 우선 코인을 3개씩(A, B, C와 D, E, F) 선택한다. 각각 3개의 코인을 천칭에 올려놓는다. 즉, $A+B+C$와 $D+E+F$이다. 그러면 세 가지 경우가 발생한다. 첫 번째는 1] $A+B+C < D+E+F$인 경우이다. 이 경우에는 A, B, C, D, E, F 중에 무게가 다른 코인이 있다는 것이다. 그래서 이번에는 $A+D$와 $B+E$를 천칭에 올려놓는다. 그러면 또 세 경우가 발생한다. 첫 번째로 1-1] $A+D < B+E$인 경우이다. 이 경우에는 A와 E 중에 무게가 다른 코인이 있으며 A라면 가볍고 E라면 무겁다.

트리의 기본 개념

그래서 세 번째 천칭에는 A와 이미 무게가 정상인 H를 비교한다. 이 경우에 $A < H$라면 A는 가벼운 코인이 되고, $A = H$라면 E는 무거운 코인이 되며, $A > H$인 경우는 발생하지 않는다. 같은 방법으로 $A+D = B+E$인 경우라면 C나 F 중에 무게가 다른 코인이 있으며 C라면 가볍고 F라면 무겁다. 그래서 이번에는 정상인 H 코인과 C 코인을 비교한다. $C < H$라면 C는 가벼운 코인이다. $C = H$라면 F가 무거운 코인이 된다. $C > H$인 경우는 발생하지 않는다. 같은 방법으로 $A+D > B+E$인 경우에도 B나 D 중에 무게가 다른 코인이 있으며 B라면 무겁고 D라면 가볍다. 그래서 이번에는 정상인 H 코인과 B 코인을 비교한다. $B < H$라면 B가 가벼운 코인이다. $B = H$라면 D가 무거운 코인이 되며, $B > H$인 경우는 발생하지 않는다.

트리의 기본 개념

이와 같은 방법으로 계속하면 다음과 같은 결정 트리를 만들 수 있다.



트리 탐방 알고리즘

트리 탐방(tree traversal) 알고리즘

- 앞에서 설명한 선형 표현 방법과 같이 연관해서 설명한다면,
  전위로 탐방해서 자료를 찾으면 그것은 전위 표기법이 된다.
- 중위로 탐방해서 자료를 찾으면 중위 표기법이 된다.
- 후위로 탐방해서 자료를 찾으면 후위 표기법이 된다.

트리 탐방 알고리즘

[알고리즘 8-4] 트리 탐방 알고리즘

① 전위 탐방 방법

1. 해당 트리의 루트 노드를 찾는다.

2. 왼쪽 부분 트리를 찾는다.
만약 왼쪽 부분 트리가 존재하면 왼쪽 부분 트리를 전위 탐방으로
다시 재귀적으로 탐방한다.

3. 오른쪽 부분 트리를 찾는다.
만약 오른쪽 부분 트리가 존재하면
오른쪽 부분 트리를 전위 탐방으로 다시 재귀적으로 탐방한다.

트리 탐방 알고리즘

[알고리즘 8-4] 트리 탐방 알고리즘

② 중위 탐방 방법

1. **왼쪽 부분 트리를 찾는다.**
   만약 왼쪽 부분 트리가 존재하면
   왼쪽 부분 트리를 중위 탐방으로 다시 재귀적으로 탐방한다.
2. **해당 트리의 루트 노드를 찾는다.**
3. **오른쪽 부분 트리를 찾는다.**
   만약 오른쪽 부분 트리가 존재하면
   오른쪽 부분 트리를 중위 탐방으로 다시 재귀적으로 탐방한다.

트리 탐방 알고리즘

[알고리즘 8-4] 트리 탐방 알고리즘

③ 후위 탐방 방법

1. **왼쪽 부분 트리를 찾는다.**
   만약 왼쪽 부분 트리가 존재하면
   왼쪽 부분 트리를 후위 탐방으로 다시 재귀적으로 탐방한다.
2. **오른쪽 부분 트리를 찾는다.**
   만약 오른쪽 부분 트리가 존재하면
   오른쪽 부분 트리를 후위 탐방으로 다시 재귀적으로 탐방한다.
3. **해당 트리의 루트 노드를 찾는다.**

트리 탐방 알고리즘

예제 8-7 트리 탐방 알고리즘(1)

[그림 8-8]에 있는 트리를 전위 탐방, 중위 탐방, 후위 탐방 방법으로 탐방해서 결과를 전위 표기법, 중위 표기법, 후위 표기법으로 표현하라.

풀이

• **전위 탐방 방법**

먼저 루트 노드를 탐방한다. 그래서 *를 출력한다. 두 번째는 왼쪽 부분 트리를 탐방한다. 왼쪽이 부분 트리로 구성되어 있으므로 다시 재귀적으로 왼쪽 부분 트리의 루트 노드를 방문한다. 그래서 +를 출력한다. 다시 왼쪽 부분 트리의 왼쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 A를 출력한다. 이번에는 왼쪽 부분 트리의 오른쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 B를 출력한다. 이번에는 원래 트리의 오른쪽 부분 트리를 탐방한다. 오른쪽이 부분

트리 탐방 알고리즘

트리로 구성되어 있으므로 다시 오른쪽 부분 트리의 루트 노드를 탐방해서 −를 출력한다. 오른쪽 부분 트리의 왼쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 C를 출력한다. 오른쪽 부분 트리의 오른쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 D를 출력한다. 출력된 결과를 순서대로 나타내면 $* + A\ B - C\ D$이다. 즉, 전위 표기법으로 표기하면 $* + A\ B - C\ D$이다.

• **중위 탐방 방법**

먼저 왼쪽 부분 트리를 탐방한다. 왼쪽이 부분 트리로 구성되어 있으므로 다시 재귀적으로 왼쪽 부분 트리의 왼쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 A를 출력한다. 이번에는 왼쪽 부분 트리의 루트 노드를 탐방해서 +를 출력한다. 다음으로 왼쪽 부분 트리의 오른쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 B를 출력한다. 이번에는 원래 트리의 루트 노드를 탐방해서

트리 탐방 알고리즘

$*$를 출력한다. 원래 트리의 오른쪽 부분 트리를 탐방한다. 오른쪽이 부분 트리로 구성되어 있으므로 다시 오른쪽 부분 트리의 왼쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 C를 출력한다. 오른쪽 부분 트리의 루트 노드를 탐방해서 $-$를 출력한다. 오른쪽 부분 트리의 오른쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 D를 출력한다. 출력된 결과를 순서대로 나타내면 $A + B * C - D$이다. 즉, 중위 표기법으로 표기하면 $A + B * C - D$이다.

- **후위 탐방 방법**

먼저 왼쪽 부분 트리를 탐방한다. 왼쪽이 부분 트리로 구성되어 있으므로 다시 재귀적으로 왼쪽 부분 트리의 왼쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 A를 출력한다. 이번에는 왼쪽 부분 트리의 오른쪽 부분 트리를 탐방한다. 터미널 노드로 구성되어 있으므로 탐방해서 B를 출력한다. 이번에는 왼쪽 부분 트리의 루트 노드를 탐방하여 $+$를 출력한다. 다음으로 원래 트리의 오른쪽 부분 트리를

트리 탐방 알고리즘

탐방한다. 오른쪽이 부분 트리로 구성되어 있으므로 다시 오른쪽 부분 트리의 왼쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 C를 출력한다. 오른쪽 부분 트리의 오른쪽 부분 트리를 탐방하는데 터미널 노드로 구성되어 있으므로 D를 출력한다. 이번에는 오른쪽 부분 트리의 루트 노드를 탐방해서 $-$를 출력한다. 마지막으로 원래 트리의 루트 노드를 탐방해서 $*$를 출력한다. 출력된 결과를 순서대로 나타내면 $A\ B + C\ D - *$이다. 즉, 후위 표기법으로 표기하면 $A\ B + C\ D - *$이다.

트리 탐방 알고리즘

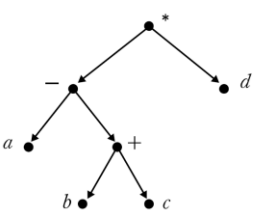
예제 8-8 트리 탐방 알고리즘(2)

대수식 $(a-(b+c))*d$를 트리로 표현하라. 트리에서 전위 탐방, 중위 탐방, 후위 탐방을 해서 전위 표기법, 중위 표기법, 후위 표기법으로 표현하라.

풀이

8.2절에서 살펴본 것처럼 먼저 트리로 표현하기 위해서 대수식에서 중심 연산자를 찾는다. * 연산자가 중심 연산자이므로 * 가 루트 노드에 $(a-(b+c))$가 왼쪽 부분 트리에 d가 오른쪽 부분 트리이다. 다시 왼쪽 부분 트리에서 중심 연산자를 찾으면 −가 된다. 왼쪽 부분 트리의 루트 노드는 −가 되고 − 연산자의 왼쪽 부분 트리는 a가 되고 오른쪽 부분 트리는 $b+c$가 된다. 그래서 오른쪽 부분 트리에 대해서 중심 연산자를 찾으면 +가 되고 + 연산자의 왼쪽 부분 트리는 b, 오른쪽 부분 트리는 c가 된다. 이를 전체적으로 트리로 표시하면 다음과 같다.

트리 탐방 알고리즘



이제 트리 탐방을 해보자. 전위 탐방을 하면 트리의 루트를 탐방한다. 그래서 * 를 출력한다. 다음으로는 왼쪽 부분 트리를 탐방한다. 왼쪽 부분 트리가 트리이므로 다시 왼쪽 부분 트리의 루트 노드를 탐방한다. −를 출력한다. 이번에는 왼쪽 부분 트리를 탐방한다. 터미널 노드이므로 a를 출력한다. 왼쪽 부분 트리의 오른쪽 부분 트리를 탐방한다. 트리이므로 루트 노드를 탐방한다. +를 출력한다. 다음으로는 왼쪽 부분 트리를 탐방하는데 터미널 노드이므로 b를 출력한다. 다시 오른쪽 부분 트리를 탐방하는데 터미널 노드이므로 c를 출력한다. 이제 원래 트리의 오른쪽 부분 트리를 탐방하는데 터미널 노드이므로 d를 출력한다. 그러면 $*-a+b\,c\,d$가 출력되고 이 표현이 전위 표기법이다.

트리 탐방 알고리즘

중위 탐방과 후위 탐방도 같은 방법으로 하면 다음과 같이 정리할 수 있다.

전위 표기법 : $* - a + b\ c\ d$
후위 표기법 : $a\ b\ c + - d *$
중위 표기법 : $a - b + c * d$