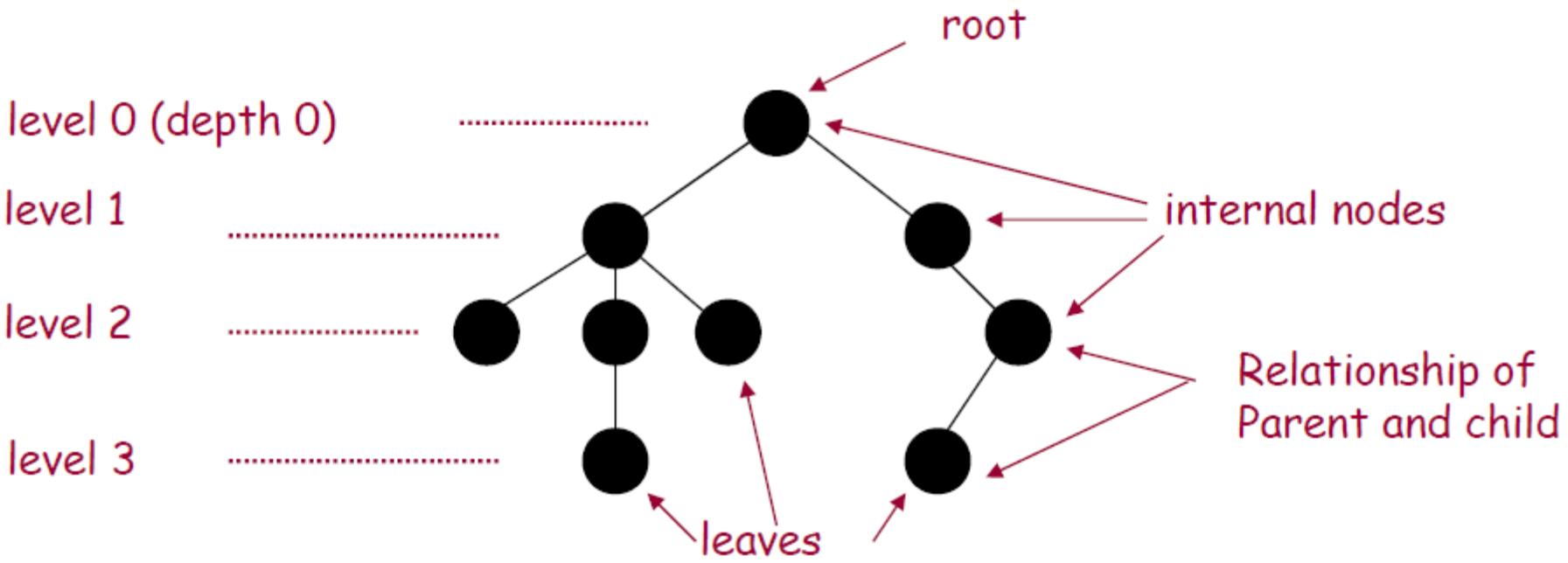
1 트리 Tree

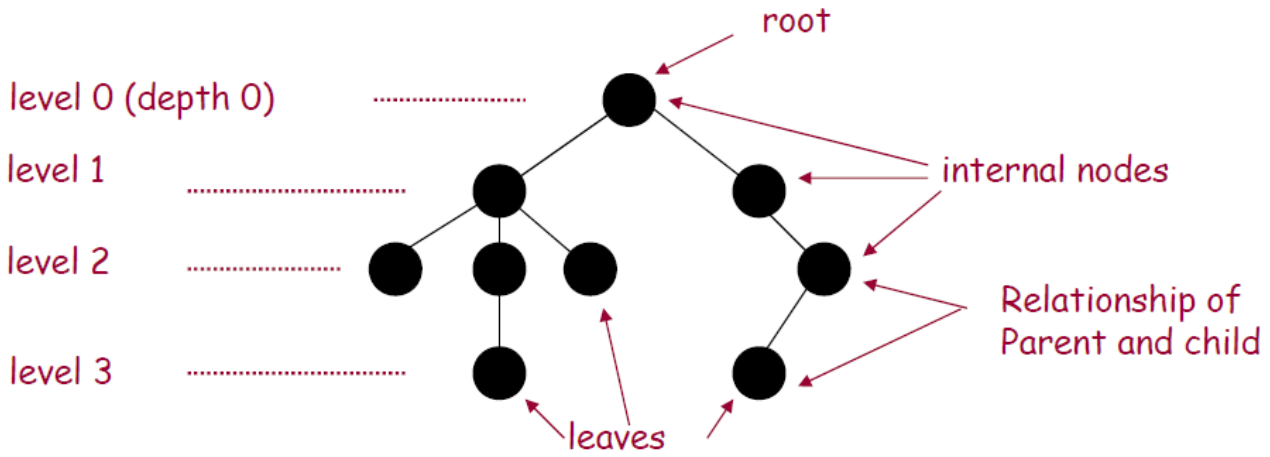
다양한 분야에서 널리 사용되고 있는 자료구조인 트리(Tree)에 대해 살펴봅니다.

기본개념

트리는 그 모양이 뒤집어 놓은 나무와 같다고 해서 이런 이름이 붙었습니다.



1 트리 Tree



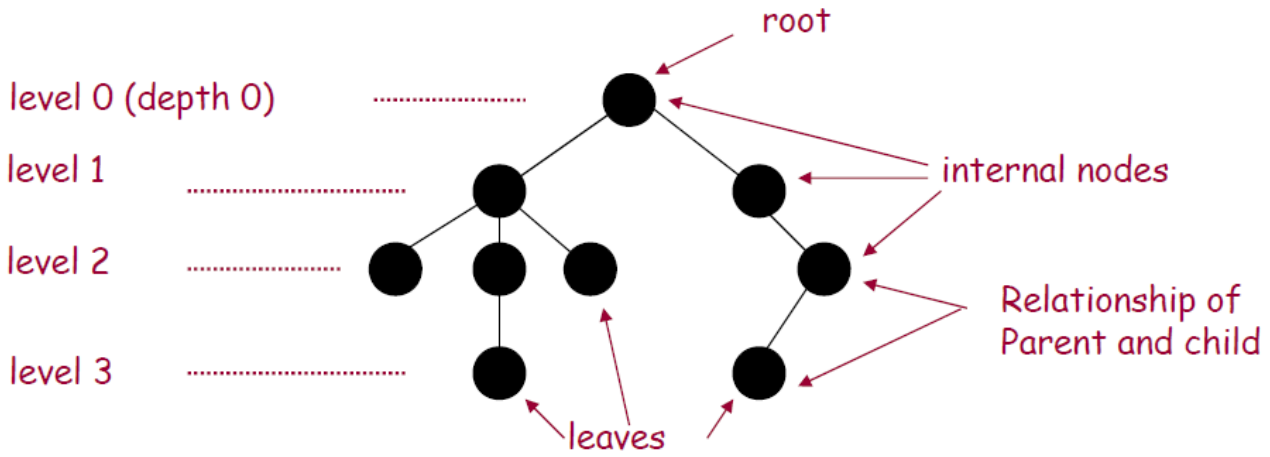
검정색 동그라미를 **노드**(Node)라고 합니다. 보통 데이터가 여기에 담깁니다.
노드와 노드사이를 이어주는 선을 **엣지**(Edge)라고 합니다. 노드와의 관계를 표시합니다.

경로(path)란 엣지로 연결된, 즉 인접한 노드들로 이뤄진 시퀀스(sequence)를 가르킵니다.
경로의 길이(length)는 경로에 속한 엣지의 수를 나타냅니다.

트리의 높이(height)는 루트노드에서 말단노드에 이르는 가장 긴 경로의 엣지 수를 가르킵니다.
트리의 특정 깊이를 가지는 **노드의 집합을 레벨**(level)이라 부릅니다.

잎새노드(leaf node)란 자식노드가 없는 노드입니다. **내부노드**(Internal node)란 잎새 노드를 제외한 노드를 나타냅니다. **루트노드**(root node)란 부모 노드가 없는 노드를 가르킵니다.

1 트리 Tree

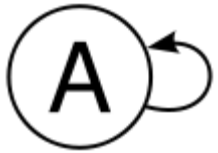


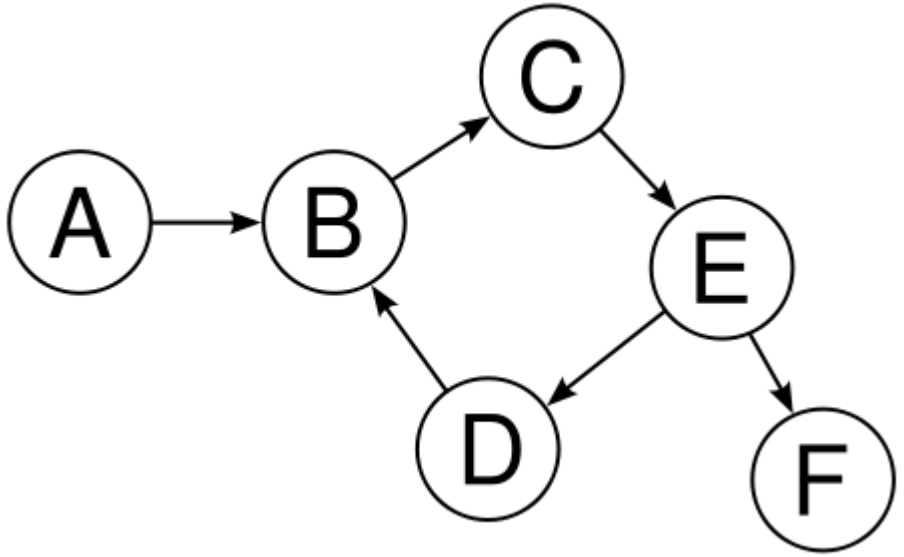
트리의 속성 중 가장 중요한 것이 '**루트노드를 제외한 모든 노드는 단 하나의 부모노드만을 가진다**'는 것입니다. 이 속성 때문에 트리는 다음 성질을 만족합니다.

- 임의의 노드에서 다른 노드로 가는 경로(path)는 유일하다.
- 회로(cycle)가 존재하지 않는다.
- 모든 노드는 서로 연결되어 있다.
- 엣지(edge)를 하나 자르면 트리가 두 개로 분리된다.
- 엣지(edge)의 수 $|E|$는 노드의수 $|V|$에서 1을 뺀것과 같다.

1 트리 Tree

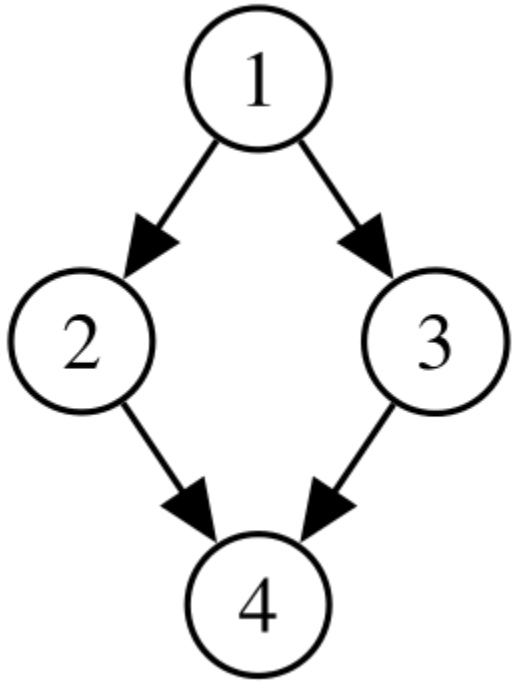
다음 두 예시는 트리가 아닙니다. 회로가 존재하기 때문입니다.





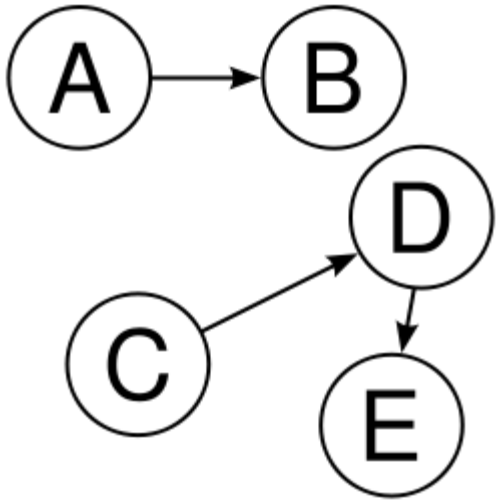
1 트리 Tree

다음 예시는 트리가 아닙니다.
회로가 존재하지 않지만 1에서 4로 가는 경로가 유일하지 않아서입니다.



1 트리 Tree

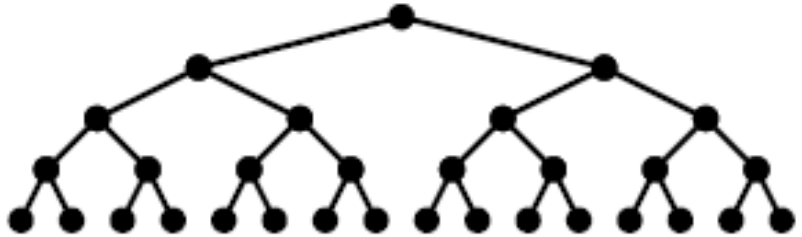
다음 예시는 트리가 아닙니다.
연결되지 않은 노드가 존재하기 때문입니다.



2 이진트리 Binary Tree

이진트리란 자식노드가 최대 두개인 노드들로 구성된 트리입니다. 이진트리에는 **정이진트리(full binary tree), 완전 이진트리(complete binary tree), 균형이진트리(balanced binary tree)**등이 있습니다.

정이진트리는 다음 그림과 같습니다. 모든 레벨에서 노드들이 꽉 채워진(=잎새노드를 제외한 모든노드가 자식노드는 2개 가짐) 이진트리입니다.



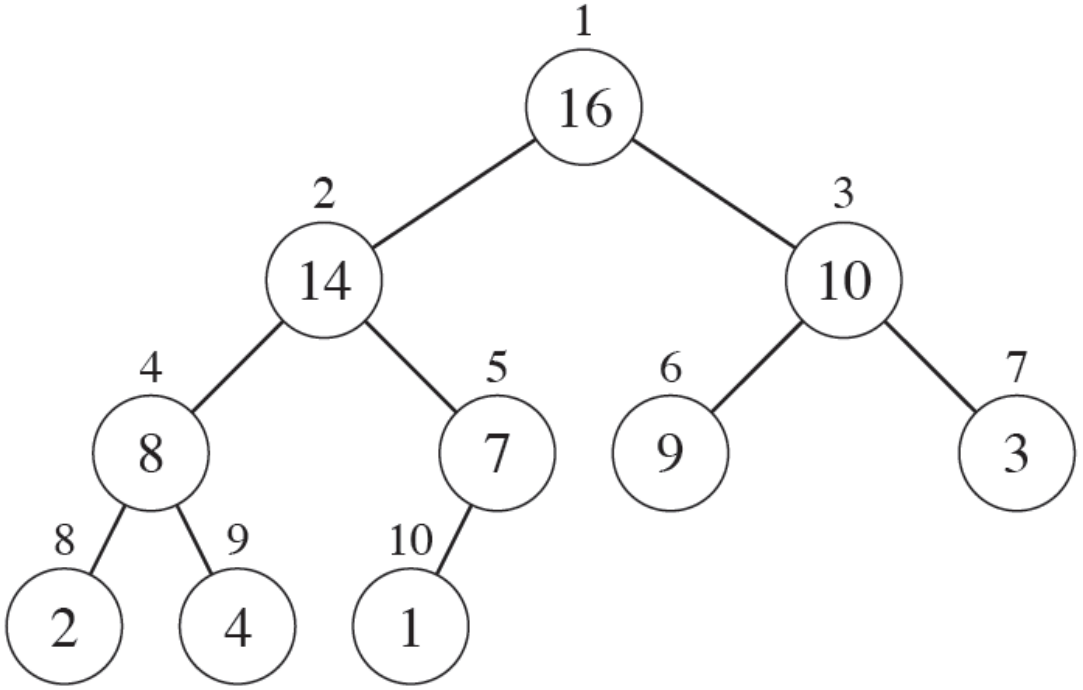
위 정이진트리에서 레벨에 따른 노드의 숫자는 다음 표와 같습니다.

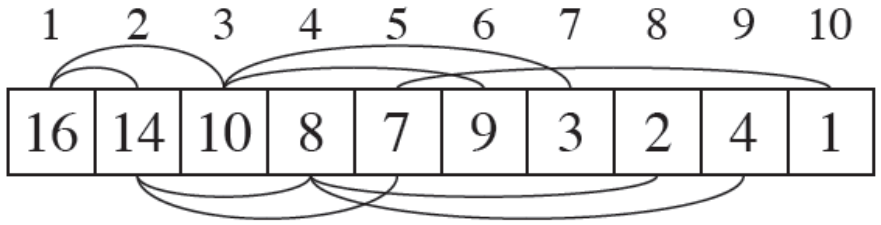
레벨	노드수
0	2^0
1	2^1
2	2^2
…	…
kk	2^k
total	$2^{k+1} - 1$

2 이진트리 Binary Tree

완전이진트리는 다음 그림과 같습니다. 마지막 레벨을 제외한 모든 레벨에서 노드들이 꽉 채워진 이진트리입니다.

정이진트리와 완전이진트리는 다음처럼 1차원 배열(array)로도 표현이 가능합니다.





2 이진트리 Binary Tree

어떤 노드의 인덱스를 index, 왼쪽 자식노드의 인덱스를 left_index, 오른쪽 자식노드의 인덱스를 right_index로 선언하면 다음과 같은 관계를 지닙니다. 이를 코드로 구현하면 다음과 같습니다.

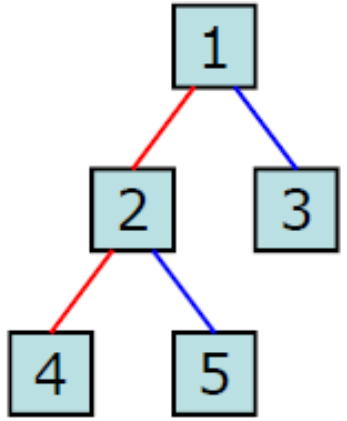
```
left_index = 2 * index + 1;
right_index = 2 * index + 2;
```

2 이진트리 Binary Tree

균형이진트리는 다음 그림과 같습니다. 모든 잎새노드의 깊이 차이가 많아야 1인 트리를 가리킵니다.
균형이진트리는 예측 가능한 깊이(predictable depth)를 가지며, 노드가 n개인 균형이진트리의 깊이는 $\log n$을 내림한 값이 됩니다.

3 트리순회 Tree traversal

트리순회(tree traversal)란 트리의 각 노드를 체계적인 방법으로 방문하는 과정을 말합니다. 하나도 빠뜨리지 않고, 정확히 한번만 중복없이 방문해야 합니다. 노드를 방문하는 순서에 따라 전위순회(preorder), 중위순회(inorder), 후위순회(postorder) 세 가지로 나뉩니다. 아래 트리를 예시로 각 방법 간 차이를 비교해 보겠습니다.

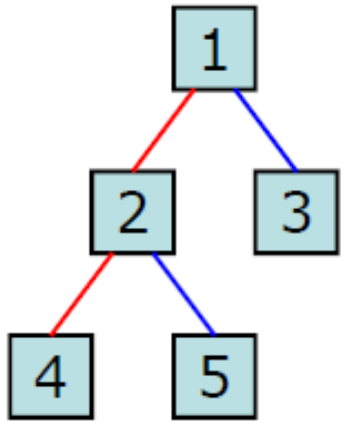


- 전위순회(preorder)

  루트 노드에서 시작해서 **노드-왼쪽 서브트리-오른쪽 서브트리** 순으로 순회하는 방식입니다.
  깊이우선순회(depth-first traversal)라고도 합니다. 위 예시 트리에 전위순회 방식을 적용하면 다음과 같습니다.

  1, 2, 4, 5, 3

3 트리순회 Tree traversal



- 중위순회(inorder)

  루트 노드에서 시작해서 **왼쪽 서브트리-노드-오른쪽 서브트리** 순으로 순회하는 방식입니다.
  대칭순회(symmetric traversal)라고도 합니다. 위 예시 트리를 중위순회 방식을 적용하면 다음과 같습니다.

  4, 2, 5, 1, 3

- 후위순회(postorder)

  루트 노드에서 시작해서 **왼쪽 서브트리-오른쪽 서브트리-노드** 순으로 순회하는 방식입니다. 위 예시 트리를 후위순회 방식을 적용하면 다음과 같습니다.

  4, 5, 2, 3, 1

3 트리순회 Tree traversal

숫자와 숫자 사이에 연산자를 넣어 표기하는 방법을 **중위표기법**(infix notation)이라 합니다. 예컨대 아래의 표기는 2와 3을 '더한다'는 뜻이 됩니다.

$$2 + 3$$

중위표기법은 괄호 연산자가 필요없는 전위표기법(+ 2 3)이나 후위표기법(2 3 +)과는 다르게 괄호가 매우 중요합니다. 연산 수행 순서를 명시적으로 나타내야 할 때가 발생하기 때문이죠. 예컨대 아래의 표기에서는 2와 3을 더하는 연산이 먼저 수행됩니다.

$$(2 + 3) \times 4$$

그런데 후위표기법에서는 위와 같은 식을 아래와 같이 쓰게 돼 괄호가 필요 없습니다.

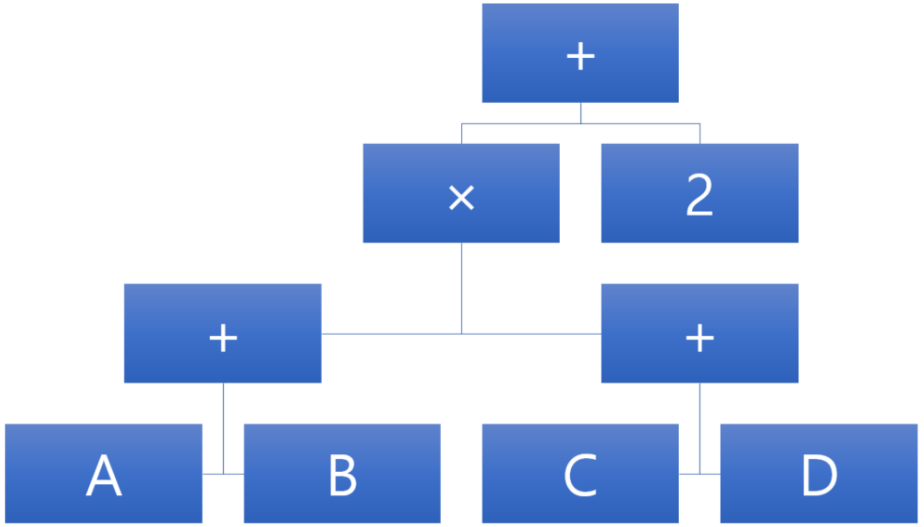
$$2\ 3 + 4 \times$$

연산의 우선순위(the priority of operands, precednece rule)은 모호하게 해석할 수 있는 수식에서 어느 연산을 먼저 계산할 것인가를 결정하는 명시적인 규칙입니다. 중위표기법에서는 다음과 같은 순위가 표준적으로 쓰입니다.

$$(,) > \times, / > +, -$$

3 트리순회 Tree traversal

$$(A + B) \times (C + D) + 2$$



이를 후위표기법으로 바꿔서 다시 쓰면 다음과 같습니다.

$$AB + CD + \times 2 +$$

위 이진트리를 후위순회 방식으로 읽어 보겠습니다. 이는 정확히 후위표기법과 일치합니다.

A, B, +, C, D, +, ×, 2, +

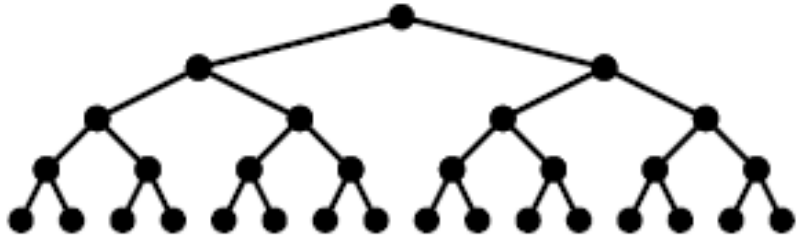
위 이진트리를 전위순회 방식으로 읽어 보겠습니다. 다음과 같습니다.

+, ×, +, A, B, +, C, D, 2

4 이진트리 Binary Tree

이진트리란 자식노드가 최대 두개인 노드들로 구성된 트리입니다. 이진트리에는 **정이진트리(full binary tree), 완전 이진트리(complete binary tree), 균형이진트리(balanced binary tree)**등이 있습니다.

정이진트리는 다음 그림과 같습니다. 모든 레벨에서 노드들이 꽉 채워진(=잎새노드를 제외한 모든노드가 자식노드는 2개 가짐) 이진트리입니다.



위 정이진트리에서 레벨에 따른 노드의 숫자는 다음 표와 같습니다.

레벨	노드수
0	2^0
1	2^1
2	2^2
…	…
kk	2^k
total	$2^{k+1} - 1$

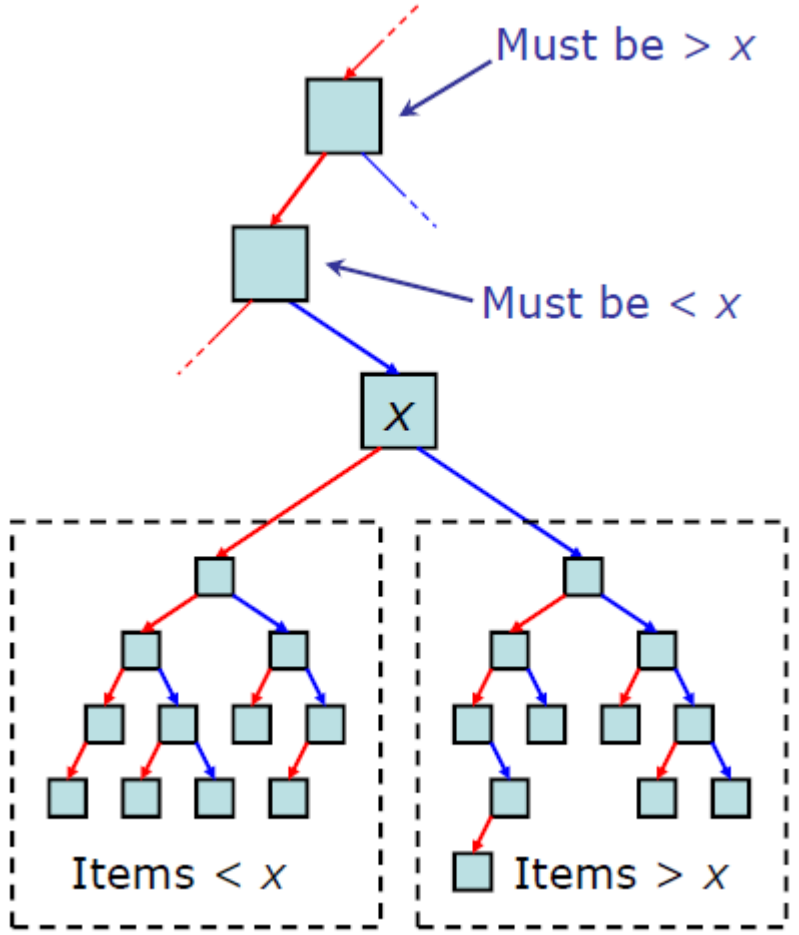
4 이진탐색트리 Binary Search Tree

기본개념

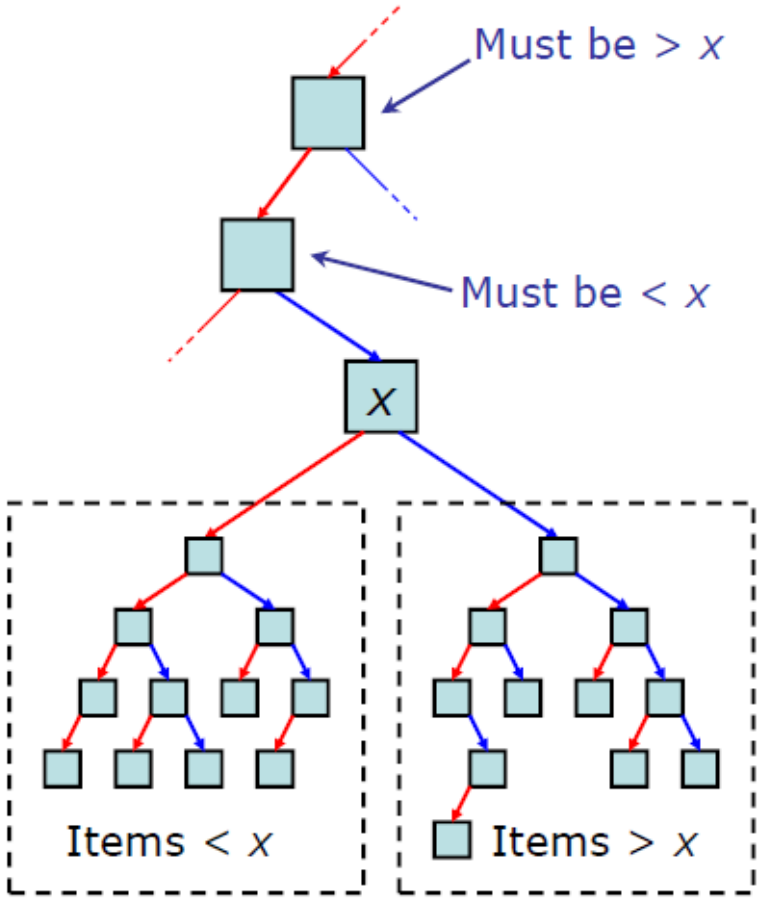
이진탐색트리란 이진탐색(binary search)과 연결리스트(linked list)를 결합한 자료구조의 일종입니다. 이진탐색의 효율적인 탐색 능력을 유지하면서도, 빈번한 자료 입력과 삭제를 가능하게끔 고안됐습니다.

예컨대 이진탐색의 경우 탐색에 소요되는 계산복잡성은 O(logn)으로 빠르지만 자료 입력, 삭제가 불가능합니다.
연결리스트의 경우 자료 입력, 삭제에 필요한 계산복잡성은 O(1)로 효율적이지만 탐색하는 데에는 O(n)의 계산복잡성이 발생합니다. 두 마리 토끼를 잡아보자는 것이 이진탐색트리의 목적입니다.

이진탐색트리는 다음과 같은 속성을 지니며 아래 그림과 같습니다.



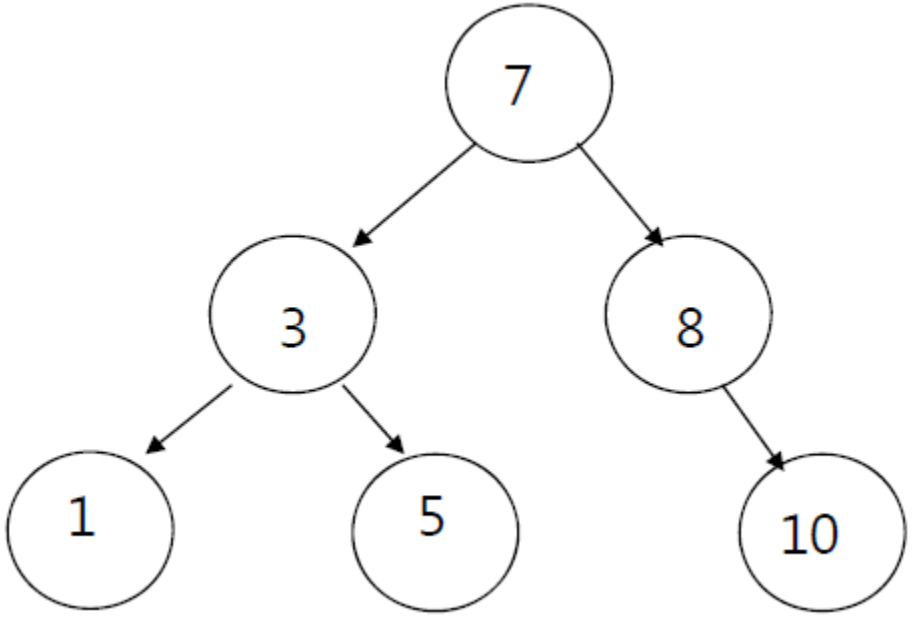
4 이진탐색트리 Binary Search Tree



- 각 노드의 왼쪽 서브트리에는 해당 노드의 값보다 작은 값을 지닌 노드들로 이루어져 있다.
- 각 노드의 오른쪽 서브트리에는 해당 노드의 값보다 큰 값을 지닌 노드들로 이루어져 있다.
- 중복된 노드가 없어야 한다.
- 왼쪽 서브트리, 오른쪽 서브트리 또한 이진탐색트리이다.

4 이진탐색트리 Binary Search Tree

이진탐색트리를 순회할 때는 중위순회(inorder) 방식을 씁니다. (왼쪽 서브트리-노드-오른쪽 서브트리 순으로 순회) 이렇게 하면 이진탐색트리 내에 있는 모든 값들을 정렬된 순서대로 읽을 수 있습니다. 다음 예시와 같습니다.



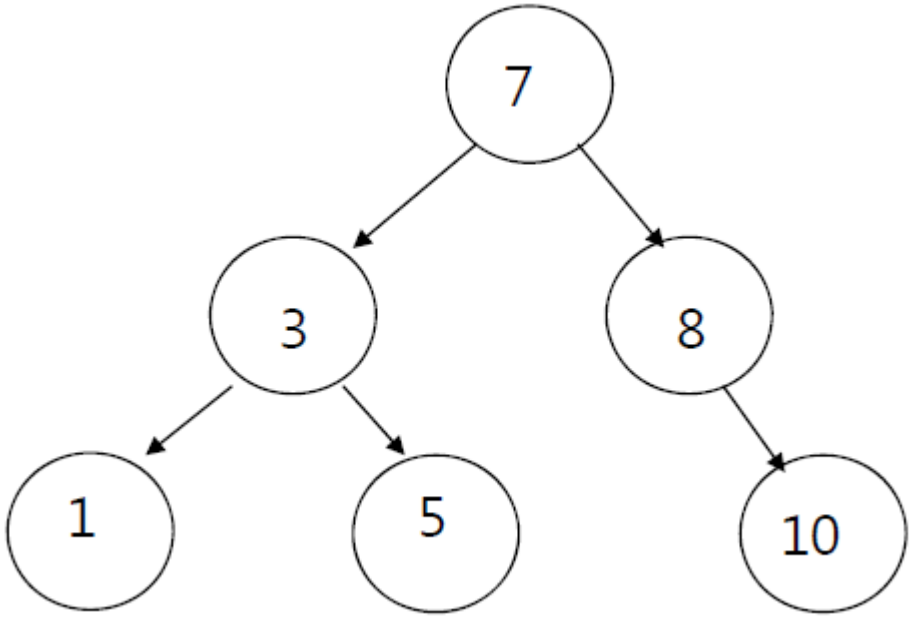
중위순회 : 1, 3, 5, 7, 8, 10

이진탐색트리의 핵심 연산은 검색(retreive), 삽입(insert), 삭제(delete) 세 가지입니다. 이밖에 이진탐색트리 생성(create), 이진탐색트리 삭제(destroy), 해당 이진탐색트리가 비어 있는지 확인(isEmpty), 트리순회(tree traverse) 등의 연산이 있습니다.

4 이진탐색트리 Binary Search Tree

검색(retrieve/find)

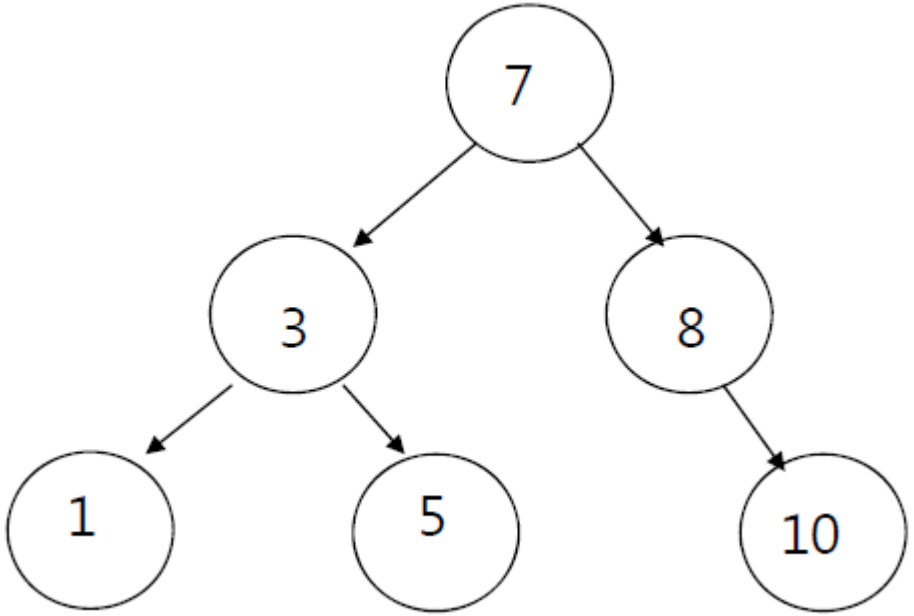
아래 이진탐색트리에서 10을 탐색(retreive, search)한다고 가정해 봅시다. 이진탐색트리는 부모노드가 왼쪽 자식노드보다 크거나 같고, 오른쪽 자식노드보다 작거나 같다는 점에 착안하면 효율적인 탐색이 가능합니다.



우선 루트노드(7)와 10을 비교합니다. 10은 7보다 큽니다. 따라서 왼쪽 서브트리(1, 3, 5)는 고려할 필요가 없습니다. 탐색공간이 대폭 줄어든다는 얘기입니다. 이번엔 오른쪽 서브트리의 루트노드(8)과 10을 비교합니다. 10은 8보다 큽니다. 따라서 오른쪽 서브트리의 루트노드(10)과 10을 비교합니다. 원하는 값을 찾았습니다.

4 이진탐색트리 Binary Search Tree

이번엔 4를 탐색해보겠습니다. 위와 같은 방식으로 4를 탐색할 때 비교하는 값은 다음과 같습니다.



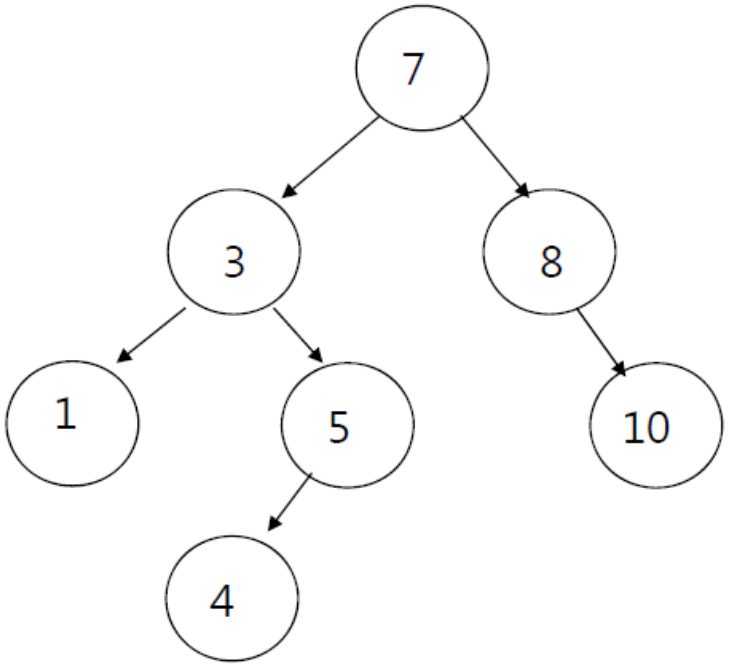
하지만 5까지 비교했는데도 원하는 값(4)을 찾지 못했습니다. 그 다음은 5의 왼쪽 서브트리를 비교할 차례인데 5는 트리의 잎새노드(leaf node)여서 서브트리가 존재하지 않습니다. 이 경우 '값을 찾지 못했다'고 반환하고 탐색을 종료합니다.

이진탐색트리의 탐색 연산에 소요되는 계산복잡성은 트리의 높이(루트노드-잎새노드에 이르는 엣지 수의 최대값)가 h일 때 O(h)가 됩니다. 최악의 경우 잎새노드까지 탐색해야 하기 때문입니다. 이 때 h번 비교 연산을 수행합니다.

4 이진탐색트리 Binary Search Tree

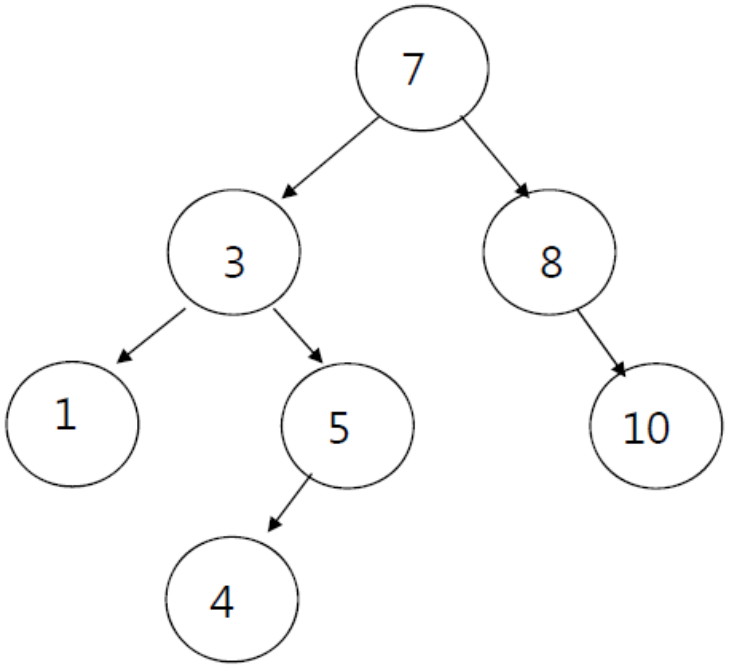
삽입(insert)

이번엔 삽입 연산을 살펴 보겠습니다. 새로운 데이터는 트리의 잎새노드에 붙입니다. 예컨대 탐색 예시에서 제시한 트리에 4를 추가한다고 가정해 봅시다. 아래와 같습니다.



그런데 위 트리에서 7과 3사이에 4를 추가해도 이진탐색트리의 속성이 깨지지 않음을 확인할 수 있습니다. 하지만 이진탐색트리가 커질 경우 이렇게 트리의 중간에 새 데이터를 삽입하게 되면 서브트리의 속성이 깨질 수 있기 때문에 삽입 연산은 반드시 잎새노드에서 이뤄지게 됩니다.

4 이진탐색트리 Binary Search Tree



이진탐색트리의 가장 왼쪽 잎새노드는 해당 트리의 최소값, 제일 오른쪽 잎새노드는 최대값이 됩니다. 만약 위 트리에서 100을 추가하려고 한다면 제일 오른쪽 잎새노드의 오른쪽 자식노드를 만들고 여기에 붙이면 됩니다.

이진탐색트리의 삽입 연산에 소요되는 계산복잡성은 트리의 높이(루트노드-잎새노드에 이르는 엣지 수의 최대값)가 h 일 때 가 됩니다. 삽입할 위치의 잎새노드까지 찾아 내려가는 데 h번 비교를 해야 하기 때문입니다. 물론 탐색 연산과 비교해 삽입이라는 계산이 추가되긴 하나, 연결리스트 삽입의 계산복잡성은 O(1)이므로 무시할 만한 수준입니다.

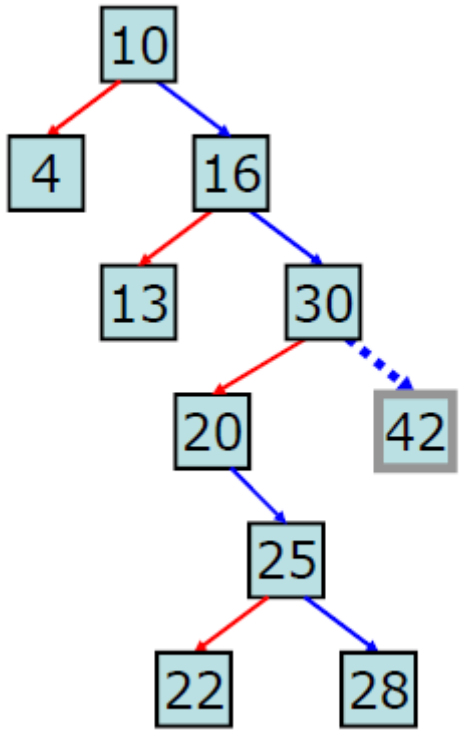
4 이진탐색트리 Binary Search Tree

삭제(delete)

삭제 연산은 탐색, 삽입보다는 약간 복잡합니다. 삭제 결과로 자칫 이진탐색트리의 속성이 깨질 수 있기 때문입니다. 가능한 세 가지 경우의 수를 모두 따져보겠습니다.

Case1:

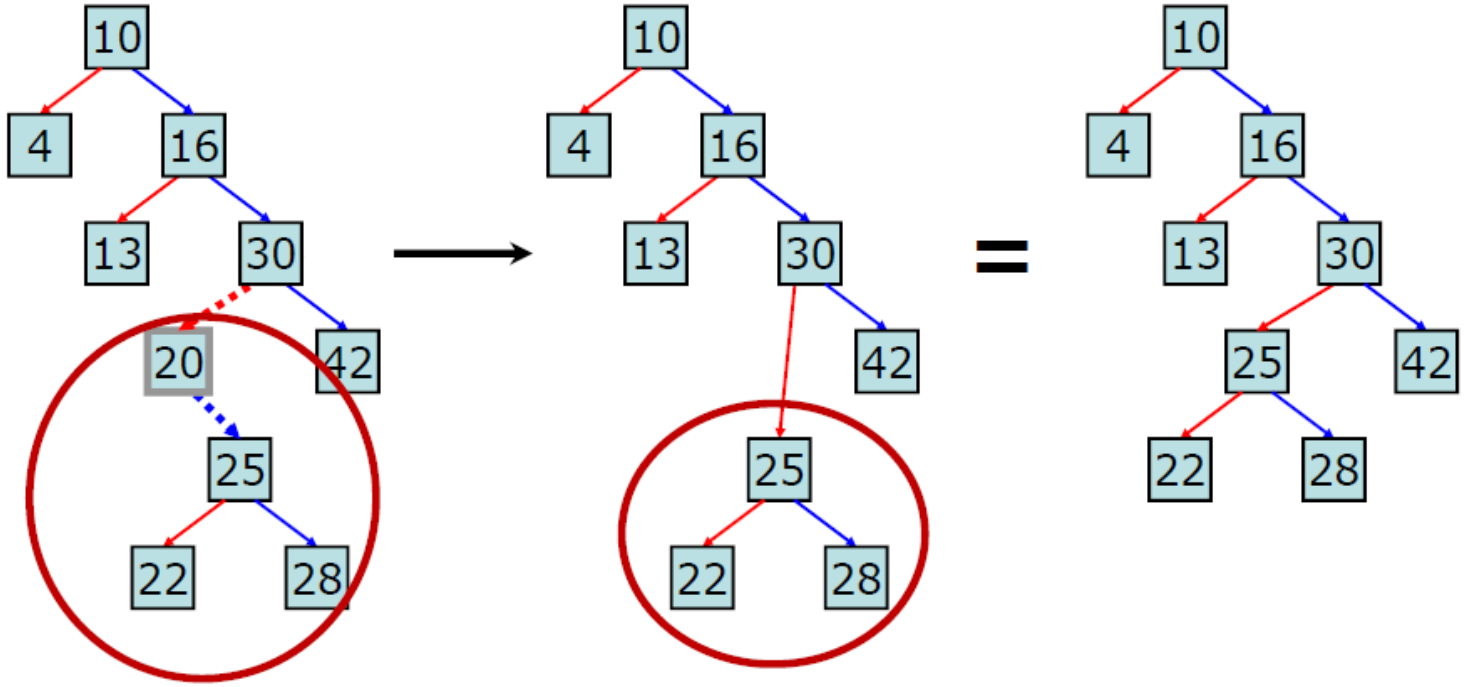
먼저 삭제할 노드에 자식노드가 없는 경우입니다. 이 케이스라면 해당 노드(아래 예시에서 42)를 그냥 없애기만 하면 됩니다. 다음과 같습니다.



4 이진탐색트리 Binary Search Tree

Case2:

이번엔 삭제할 노드에 자식노드가 하나 있는 경우입니다. 이 케이스라면 해당 노드를 지우고, 해당 노드의 자식노드와 부모노드를 연결해주면 됩니다. 아래 트리에서 20을 삭제한다고 칩시다.

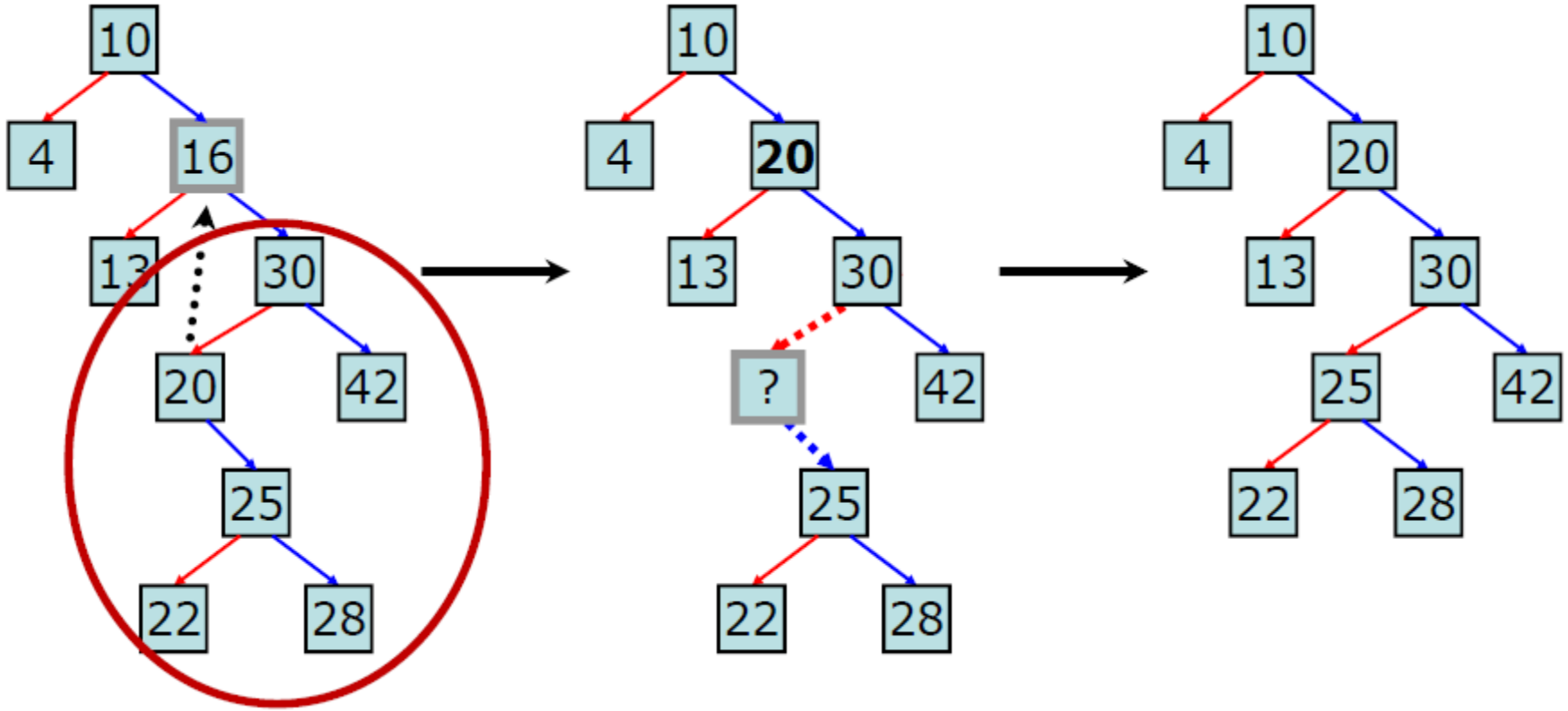


20을 루트노드로 하는 서브트리의 모든 값은 20의 부모노드인 30보다 작거나 같습니다.
이진탐색트리의 속성 때문입니다. 따라서 20을 지우고, 20의 하나뿐인 자식노드(25)와 부모노드(30)를 연결해도 이진탐색트리의 속성이 깨지지 않는 걸 확인할 수 있습니다.

4 이진탐색트리 Binary Search Tree

Case3:

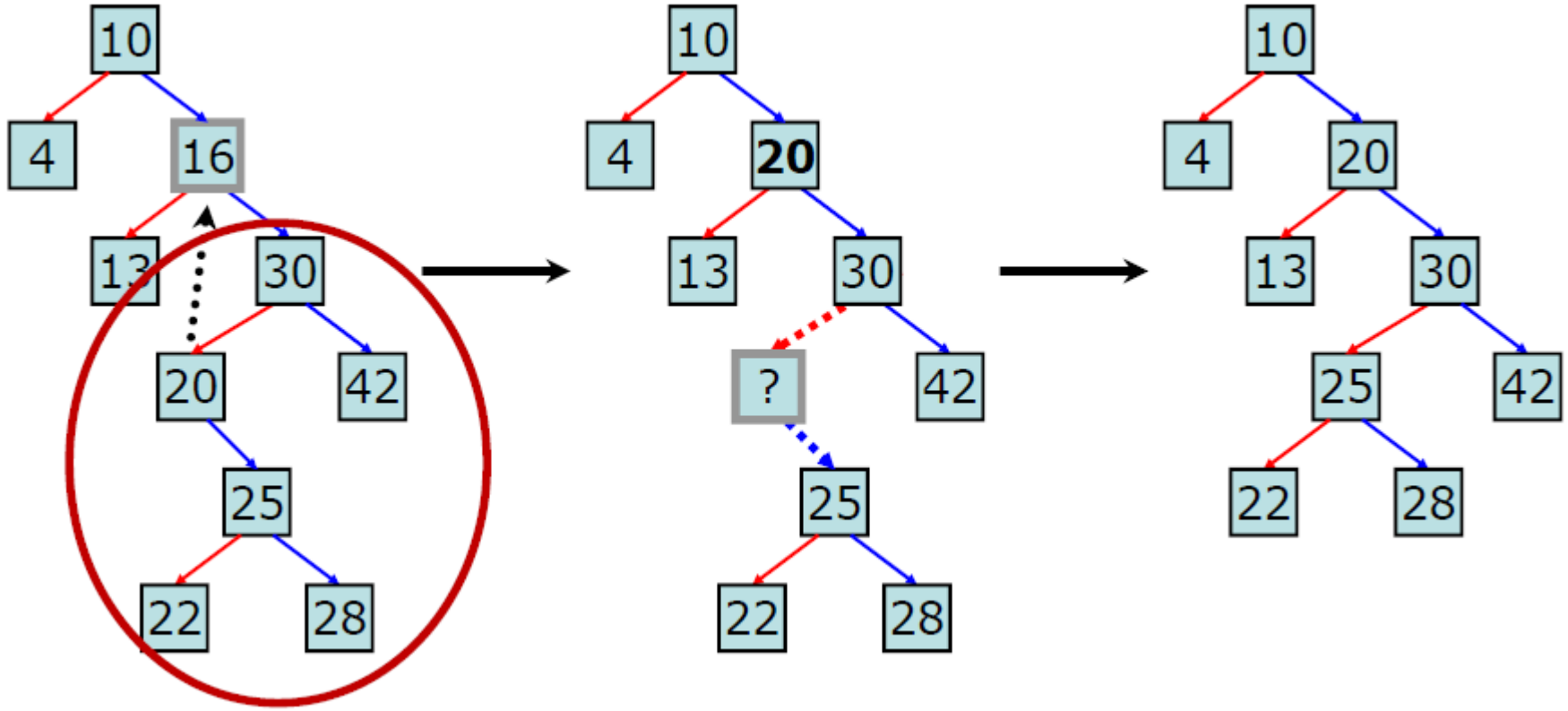
마지막으로 삭제할 노드에 자식노드가 두 개 있는 경우를 살펴보겠습니다. 아래 트리에서 16을 삭제해야 한다고 칩시다. 그런데 기존처럼 16을 무작정 지우게 되면 13의 위치가 애매해집니다. 계산복잡성을 줄이기 위해서는 트리의 요소값들을 크게 바꾸지 않고 원하는 값(16)만 삭제할 수록 좋은데, 아무래도 새로운 방법을 고민해 봐야 할 것 같습니다.



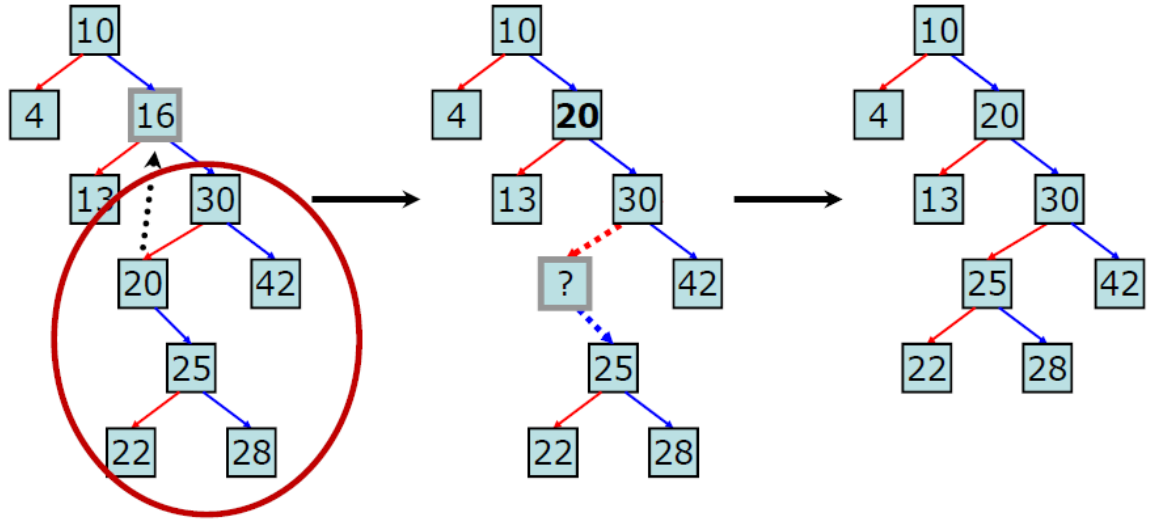
4 이진탐색트리 Binary Search Tree

이해를 돕기 위해 16을 삭제하기 전 위 트리 각 요소를 중위순회 방식(왼쪽 서브트리-노드-오른쪽 서브트리 순으로 순회)으로 읽어보겠습니다. 다음과 같이 정렬된 순으로 나타나 이진탐색트리 속성을 만족하고 있음을 확인할 수 있습니다.

4, 10, 13, 16, 20, 22, 25, 28, 30, 42



4 이진탐색트리 Binary Search Tree

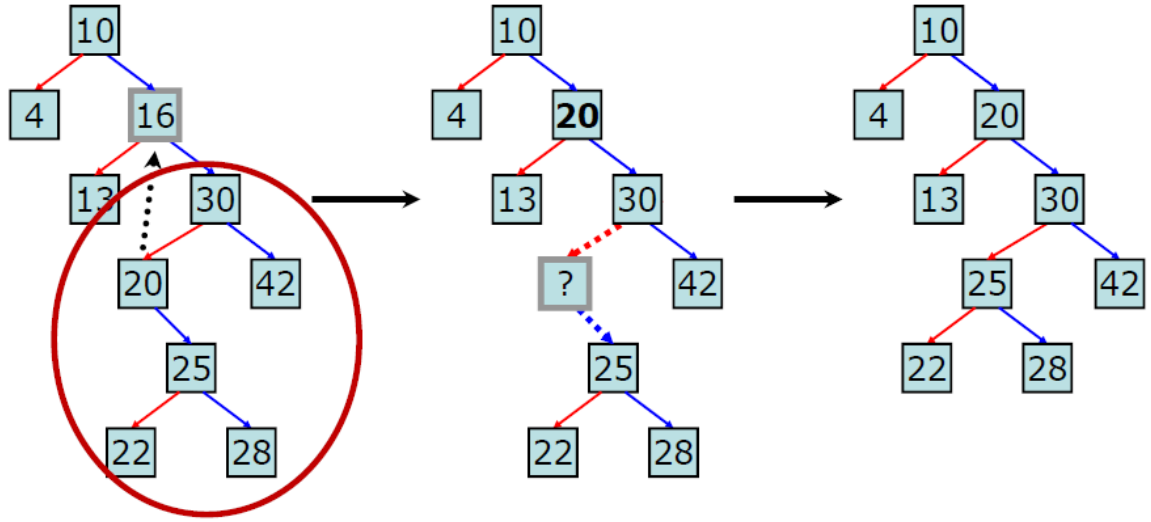


이 리스트와 예시 그림을 보면 16의 왼쪽 서브트리에 속한 모든 값은 16보다 작고, 오른쪽 서브트리에 속한 모든 값은 16보다 큰 것을 확인할 수 있습니다. 특히 13을 predecessor(삭제 대상 노드의 왼쪽 서브트리 가운데 최대값), 20을 successor(삭제 대상 노드의 오른쪽 서브트리 가운데 최소값)라고 합니다. 트리를 중위순회 방식으로 늘여뜨려 표시하면 16 바로 앞에 13이 있고, 바로 뒤에 20이 있기 때문에 각각 이런 이름이 붙은 것 같습니다.

따라서 아래와 같이 삭제할 노드인 16 위치에 20을 복사해 놓고, 기존 20 위치에 있던 노드를 삭제하게 되면 정렬된 순서를 유지(=이진탐색트리 속성을 만족)하면서도 원하는 결과를 얻을 수 있게 됩니다. 이는 위 그림에서도 확인할 수 있습니다. (물론 16 위치에 predecessor인 13을 놓고, 기존 13 위치에 있던 노드를 삭제해도 원하는 결과를 얻을 수 있습니다)

4, 10, 13, ~~16~~, **20**, ~~22~~, 25, 28, 30, 42

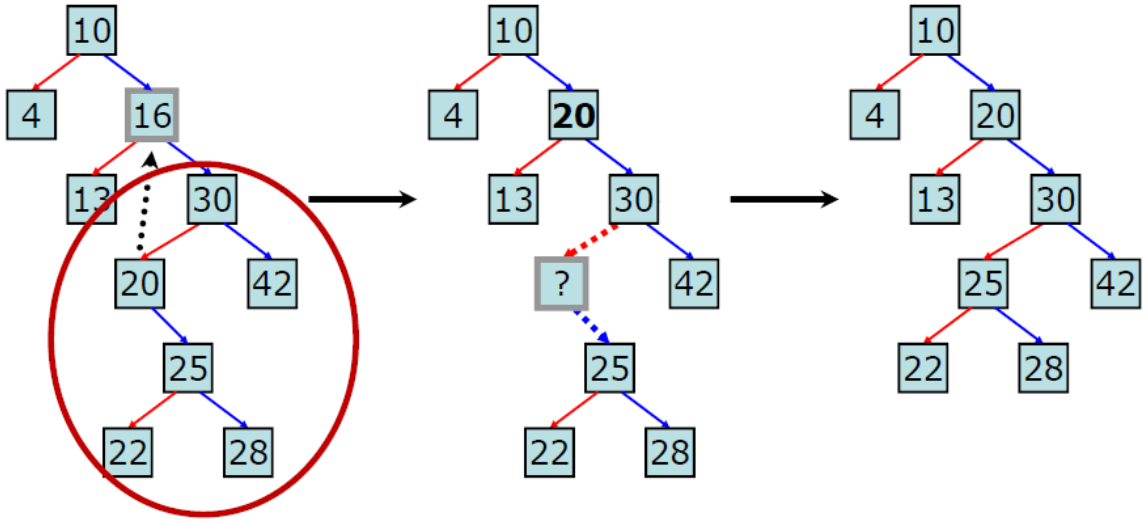
4 이진탐색트리 Binary Search Tree

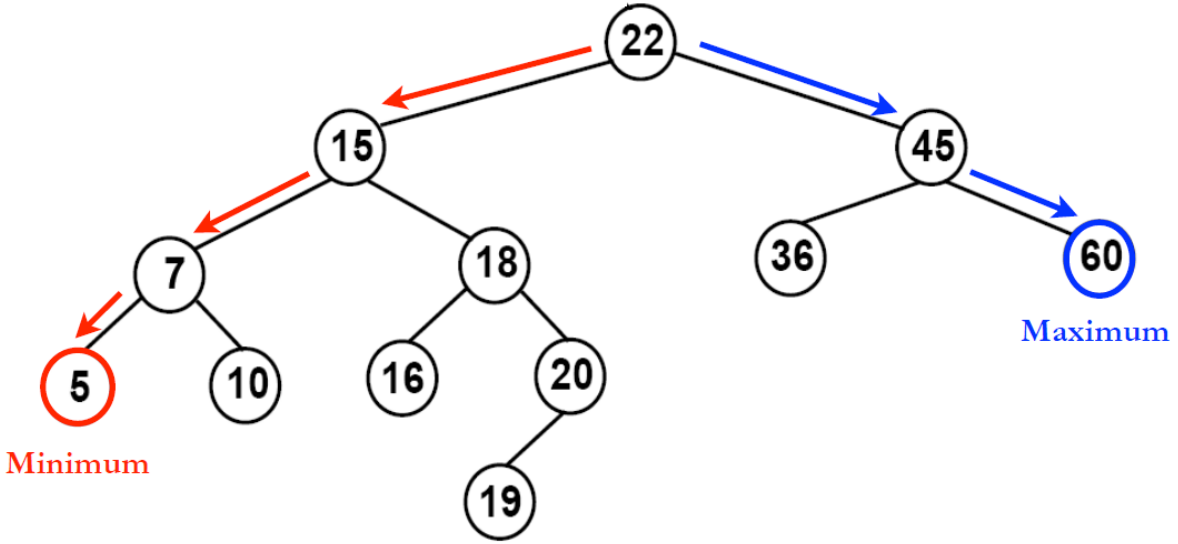


이진탐색트리 구조상 successor(삭제 대상 노드의 오른쪽 서브트리의 최소값)는 자식노드가 하나이거나, 하나도 존재하지 않습니다. 각각 살펴보면 다음과 같습니다.

- successor의 자식노드가 하나인 케이스 : 위 예시 그림과 같습니다. 삭제 대상 노드의 오른쪽 서브트리가 30을 루트노드로 하는 트리일 때, 이 트리의 맨 왼쪽 노드인 20은 하나의 자식노드(25)를 갖습니다.
- successor의 자식노드가 존재하지 않는 케이스 : 삭제 대상 노드의 오른쪽 서브트리가 아래 그림과 같을 때에는 successor는 자식노드를 가지지 않습니다.

4 이진탐색트리 Binary Search Tree

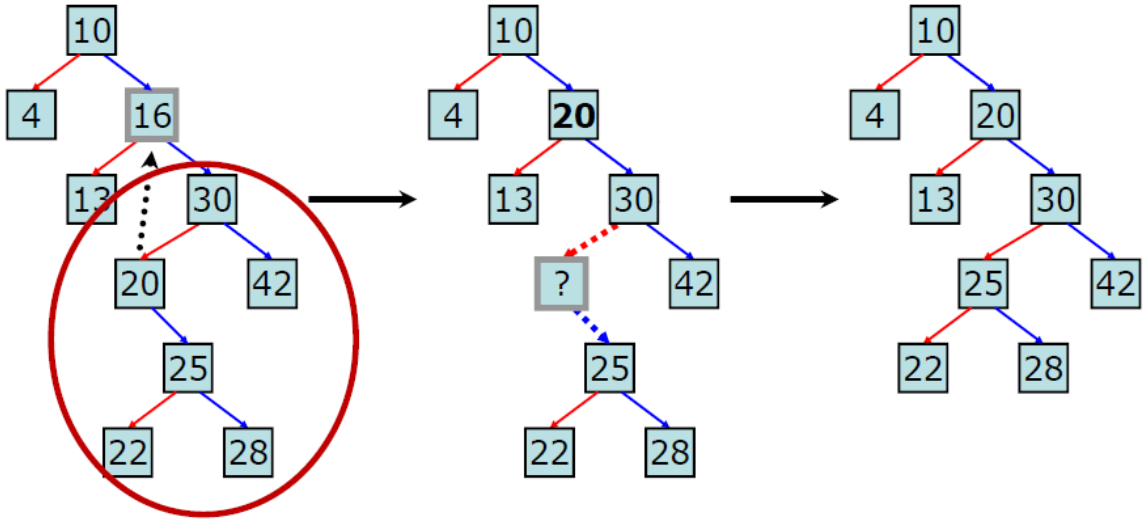


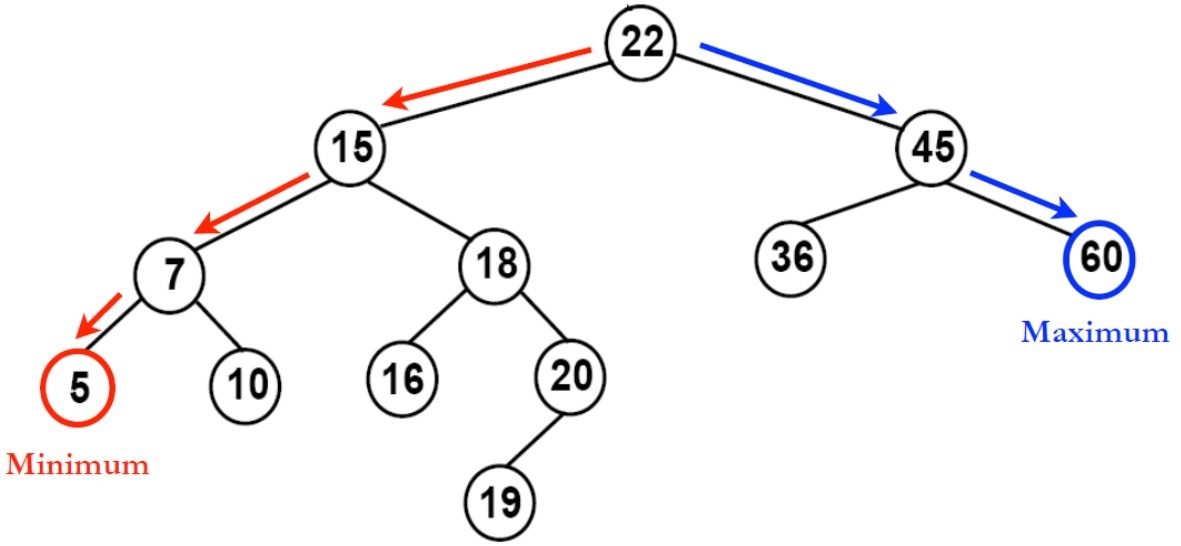


마찬가지로 왼쪽 서브트리의 맨 오른쪽 노드인 predecessor 또한 자식노드가 하나이거나, 하나도 존재하지 않습니다. 따라서 자식노드가 두 개인 경우에는 다음과 같이 삽입 연산을 수행하면 됩니다(successor 기준).

1. 삭제 대상 노드의 오른쪽 서브트리를 찾는다.
2. successor(1에서 찾은 서브트리의 최소값) 노드를 찾는다.
3. 2에서 찾은 successor의 값을 삭제 대상 노드에 복사한다.
4. successor 노드를 삭제한다.

4 이진탐색트리 Binary Search Tree





4번 successor 노드를 삭제하는 과정은 case 1나 case2에 해당합니다. 이미 언급했듯이 successor는 자식노드가 하나이거나, 하나도 존재하지 않기 때문입니다.

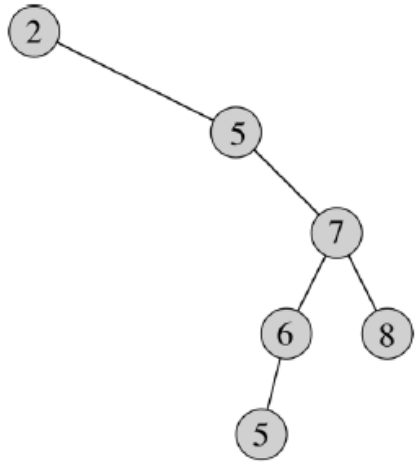
이번엔 삽입연산의 계산복잡성을 따져 보겠습니다. Big-O notation으로는 최악의 케이스를 고려해야 하므로 가장 연산량이 많은 case 3(삭제 대상 노드의 자식노드가 두 개인 경우)이 분석 대상입니다.

트리의 높이가 h이고 삭제대상 노드의 레벨이 d 라고 가정해 보겠습니다. 1번 과정에서는 d 번의 비교 연산이 필요합니다. 2번 successor 노드를 찾기 위해서는 삭제 대상 노드의 서브트리 높이$(h-d)$에 해당하는 비교 연산을 수행해야 합니다. 3번 4번은 복사하고 삭제하는 과정으로 상수시간이 걸려 무시할 만 합니다. 종합적으로 따지면 $O(d+h-d)$, 즉 $O(h)$가 됩니다.

4 이진탐색트리 Binary Search Tree

한계점

이진탐색트리 핵심 연산인 탐색, 삽입, 삭제의 계산복잡성은 모두 $O(h)$입니다. 트리의 높이에 의해 수행시간이 결정되는 구조입니다. 그러나 트리가 다음과 같은 경우 문제가 됩니다.



위 그림의 경우 노드 수는 적은 편인데 높이가 4나 됩니다. 균형이 안 맞기 때문입니다. 극단적으로는 n개의 노드가 크기 순으로 일렬로 늘어뜨려져 높이 또한 n 이 되는 경우도 이진트리탐색에 해당합니다. 결과적으로 이진탐색트리의 계산복잡성은 $O(n)$이라는 얘기입니다.

이래가지고서는 탐색 속도가 $O(\log n)$으로 빠른 이진탐색을 계승했다고 보기 어렵습니다. 이 때문에 트리의 입력, 삭제 단계에 트리 전체의 균형을 맞추는 이진탐색트리의 일종인 AVL Tree가 제안되었습니다.

1 트리 Tree

자료 원본 링크 :

https://ratsgo.github.io/data%20structure&algorithm/2017/10/21/tree/