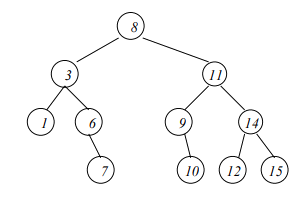
**트리(Trees)**

트리는 연결된 그래프에서 노드의 순서가 정해진 집합으로 구성되며, 각 노드는 최대 하나의 부모 노드와 특정 순서로 0개 이상의 자식 노드를 가집니다.

일반적인 트리의 사양(General specification of trees)

일반적으로 트리는 **노드(nodes, vertex 또는 point라고도 함)**와 **간선(edges, line 또는 arc라고도 함)**으로 구성된 트리 같은 구조로 지정할 수 있습니다.



트리는 빈 트리 또는 후속 트리의 목록을 가진 노드로 정의될 수 있습니다. 노드는 일반적으로 숫자나 검색 키와 같은 데이터 항목으로 레이블이 지정되지만 항상 그런 것은 아닙니다. 노드는 일반적으로 숫자나 검색 키와 같은 데이터 항목으로 레이블이 지정되지만 항상 그런 것은 아닙니다. 노드의 레이블을 **값(value)**이라고 부를 것입니다. 우리의 예에서는 일반적으로 정수로 레이블이 지정된 노드를 사용할 것이지만, 문자열과 같은 다른 것을 선택할 수도 있습니다.

항상 '최상위(top level)' 노드로 알려진 고유한 **루트(root)**가 있어야 합니다. 컴퓨터 과학에서 트리는 일반적으로 거꾸로 표시되며, 루트가 최상위를 형성한다는 점에 유의해야 합니다.

그림 에서 이것은 레이블이 8인 노드입니다.

그런 다음, 주어진 노드의 경우, 다음 레벨 '아래'의 모든 노드는 분기점을 통해 주어진 노드에 연결되어 있으며, 해당 노드의 **자식(child)**입니다.

그림 에서 노드 8의 자식은 노드 3과 11입니다.(the children of node 8 are nodes 3 and 11)

주어진 노드에 연결된 노드(최대 하나)는 위의 레벨에 있으며, **부모(parent)**입니다.

그림 에서 노드 11은 노드 9(및 노드 14)의 부모입니다.( node 11 is the parent of node 9 (and of node 14 as well)

같은 부모를 가진 노드는 **형제(siblings)**로 알려져 있습니다. 형제는 정의에 따라 항상 같은 레벨에 있습니다.

한 노드가 다른 노드의 자식의 자식의 ... 자식이라면 첫 번째 노드는 두 번째 노드의 **자손(descendent)** 이라고 합니다. 반대로, 두 번째 노드는 첫 번째 노드의 **조상(ancestor)**입니다. 자식이 없는 노드를 **잎(leaves)**이라고 합니다

그림에서 레이블이 1, 7, 10, 12, 15인 노드를 잎이라고 한다.

**경로(path)**는 한 노드에서 다른 노드로 연결된 간선의 시퀀스입니다. 트리는 모든 노드에 대해 루트와 연결하는 고유한 경로가 있다는 속성을 가지고 있습니다.

트리의 또 다른 가능한 정의로는, 노드의 **깊이(depth)** 또는 **레벨(level)**은 이 경로의 길이(length of this path)에 의해 주어집니다. 따라서 루트는 레벨 0이고, 자식은 레벨 1이고, 그렇게 계속됩니다. 트리에서 경로의 최대 길이를 **트리의 높이(height)**라고도 합니다. 최대 길이의 경로는 항상 루트에서 잎까지 이어집니다. 트리의 크기는 포함된 노드의 수로 주어집니다. 우리는 일반적으로 모든 트리가 유한하다고 가정하지만, 일반적으로 그럴 필요는 없습니다.

그림의 트리는 높이가 3이고 크기(size)가 11입니다. 단 하나의 노드로 구성된 트리는 높이가 0이고 크기가 1입니다.

**빈 트리는 크기가 0이고 편리하게도 높이 -1**을 갖도록 정의되어 있습니다.

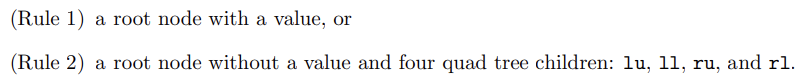
대부분의 데이터 구조와 마찬가지로, 트리를 구축하고 조작하기 위해 일련의 기본 연산자(생성자, 선택기, 조건)가 필요합니다.

**쿼드트리 (Quad-trees)**

쿼드트리는 각 잎 노드가 값으로 레이블이 지정되고 각 비-잎 노드가 정확히 네 개의 자식 노드를 갖는 특정 유형의 트리입니다. 이는 2차원 공간(예: 픽셀화된 이미지)을 4개의 사분면으로 재귀적으로 분할하여 가장 자주 사용됩니다.

쿼드트리는 숫자 또는 값(예: 0에서 255 사이)을 가진 단일 노드이거나 값이 없지만 네 개의 쿼드트리 자식 노드: lu, ll, ru, rl을 가진 노드로 정의될 수 있습니다. 따라서 다음 규칙에 의해 "유도적으로" (“inductively” )정의될 수 있습니다:

**Definition. A quad tree is either**



값을 가진 단일 노드 - 이 경우, 이 노드는 더 이상 나뉘지 않고 하나의 값을 포함하는 "단순한" 쿼드트리입니다.

값을 가지지 않고 네 개의 자식 노드를 가진 노드 - 이 노드 자체에는 값이 없지만, 4개의 하위 쿼드트리(lu, ll, ru, rl)를 포함합니다.

lu: 왼쪽 위(Left Upper) ll: 왼쪽 아래(Left Lower) ru: 오른쪽 위(Right Upper) rl: 오른쪽 아래(Right Lower)

#여기서 규칙 1은 "기본 사례(base case )"이고 규칙 2는 "유도 단계(induction step )"입니다.

쿼드트리가 단일 노드/숫자로 구성된 경우, 쿼드트리를 원시적이라고 합니다. 이는 다음 조건을 통해 테스트할 수 있습니다:

**조건(condition)**  
isValue(qt): 쿼드트리 qt가 단일 노드인 경우 true를 반환합니다.

**생산자(constructors)**  
baseQT(value): 레이블 값을 가진 단일 노드 쿼드트리를 반환합니다.  
makeQT(luqt, ruqt, llqt, rlqt): 네 개의 구성 쿼드트리 luqt, llqt, ruqt, rlqt로부터 쿼드트리를 생성합니다.

**선택자(selectors)**  
lu(qt): 왼쪽 위 쿼드트리를 반환합니다.

ru(qt): 오른쪽 위 쿼드트리를 반환합니다.

ll(qt): 왼쪽 아래 쿼드트리를 반환합니다.

rl(qt): 오른쪽 아래 쿼드트리를 반환합니다.

이는 isValue(qt)가 false일 때 적용할 수 있습니다. isValue(qt)가 true인 경우, 값을 반환하는 연산자 value(qt)를 정의할 수 있지만, 일반적으로 qt 자체를 필요한 값이라고 합니다.

#간단한 quad tree

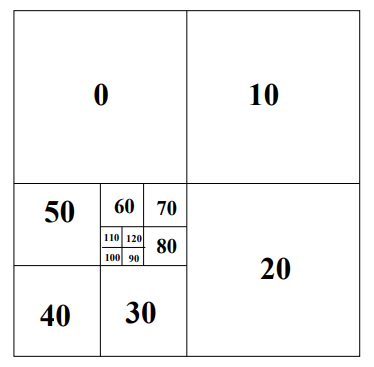
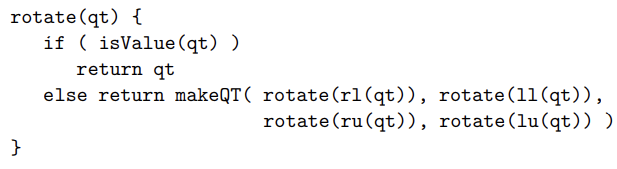
qt1: 값 = 5, value(qt1) = 5 (lu(qt1), ru(qt1), ll(qt1), rl(qt1)는 적용할 수 없습니다.)



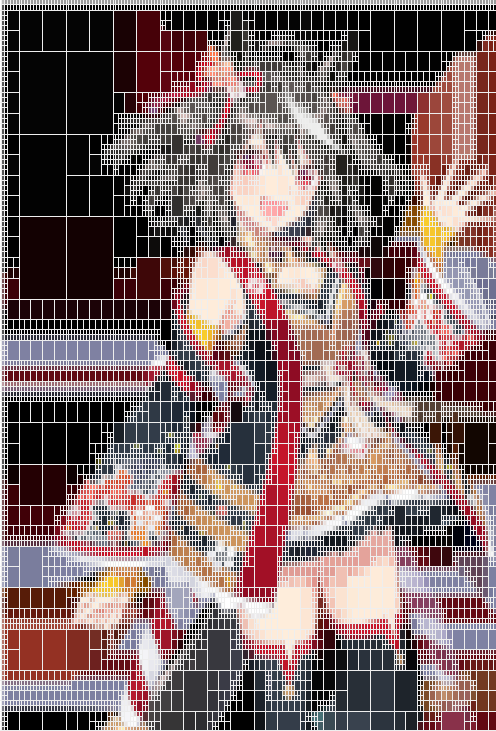
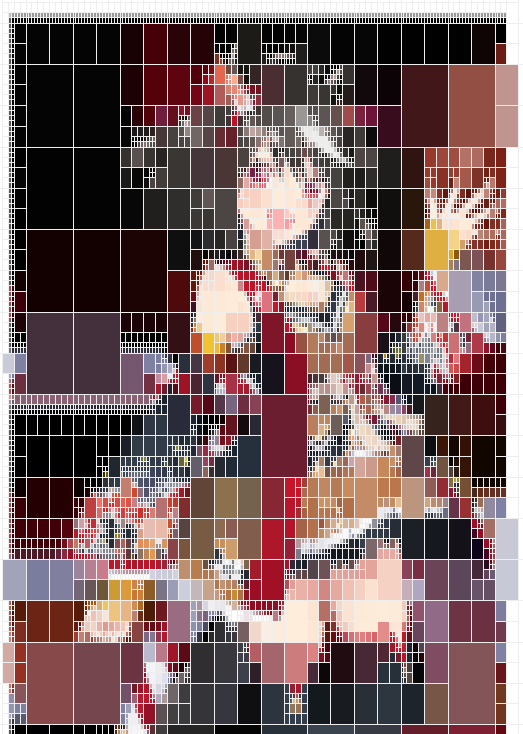
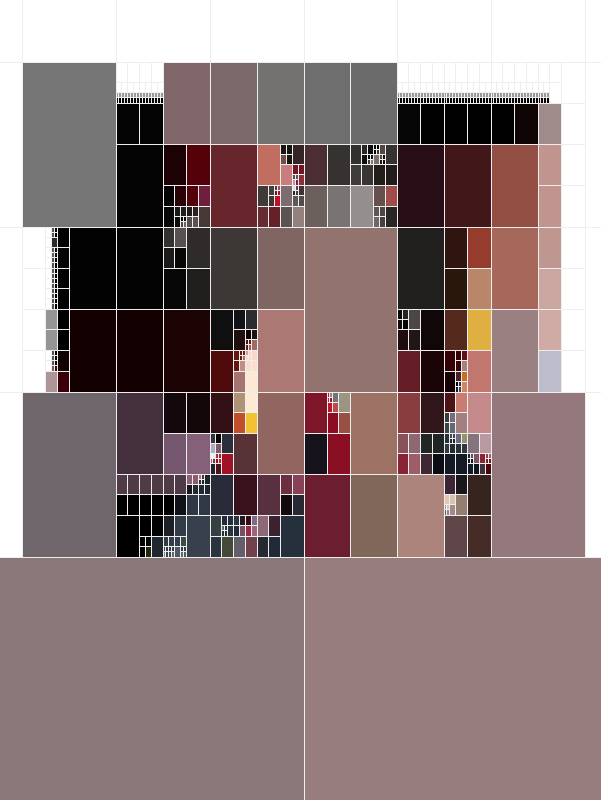
#복잡한 quad tree

isValue(qt2) = false, lu(qt2) = qt2.lu (값: 1) ru(qt2) = qt2.ru (값: 3) ll(qt2) = qt2.ll (값: 5) rl(qt2) = qt2.rl (값: 7)

isValue(lu(qt2)) = true라면 value(lu(qt2)) = 1

이러한 유형의 쿼드트리는 일반적으로 흑백 이미지(0은 검정, 255는 흰색)를 저장하는 데 가장 많이 사용됩니다.  
  
  
연산자를 이용한 활용( rotate a picture qt by 180◦)  




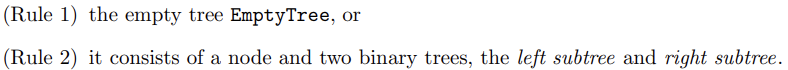


다양한 변형이 있습니다. 예를 들어, 색상 쿼드트리는 회색조 대신 색상을 나타내는 값-삼중항을 저장하고, 에지 쿼드트리는 선을 저장하여 임의의 정밀도로 곡선을 표현할 수 있습니다.

**이진 트리(binary tree)**

이진 트리는 컴퓨터 과학에서 가장 일반적인 트리 유형입니다. 이진 트리는 각 노드가 최대 두 개의 자식 노드를 갖는 트리로, 다음 규칙으로 "유도적으로" 정의될 수 있습니다

**Definition. A binary tree is either**

빈 트리 EmptyTree - 노드도 없는 "빈 트리(EmptyTree)", 더 이상 재귀적으로 분해할 수 없는 상태로, 트리 구조의 끝

노드와 서브트리로 구성된 트리 - 하나의 "노드"와 두 개의 하위 트리(왼쪽 서브트리와 오른쪽 서브트리)로 구성됩니다. 서브트리도 다시 이진 트리의 정의를 따릅니다(재귀적 구조).

#여기서 규칙 1은 "기본 사례(base case )"이고 규칙 2는 "유도 단계(induction step )"입니다.

순환적인 것처럼 보일 수 있지만, 실제로는 그렇지 않습니다. 왜냐하면 서브트리는 항상 원래 트리보다 간단하고, 결국 빈 트리에 도달하기 때문입니다. 결국, 모든 분해 과정에서 기본 사례(빈 트리)에 도달하게 됩니다. 따라서, 무한 반복이 일어나지 않습니다.

이진트리 생성 예시

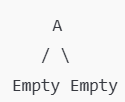
**Day 0**

*규칙 1*을 적용하여 "빈 트리(Empty Tree)"를 만듭니다. 이 단계에서는 트리에 아무 노드도 없습니다.

결과: 빈 트리 (높이 = 0)

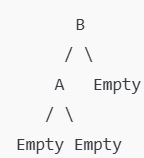
**Day 1**

*규칙 2*를 처음으로 적용합니다. 새 루트 노드를 만들고, 왼쪽과 오른쪽 서브트리로 Day 0에서 생성된 "빈 트리"를 사용합니다. 즉, **노드 1개를 가진 트리**를 생성합니다.

<-결과  
(높이 = 1)

**Day 2**

*규칙 2*를 다시 적용하여, 이전에 생성한 트리(예: Day 1에서 만든 트리)를 서브트리로 사용합니다. 이 과정에서 **노드가 2개 이상인 트리**를 생성할 수 있습니다.

Ex. 루트 노드를 만들고, 왼쪽 서브트리는 Day 1의 트리(단일 노드), 오른쪽 서브트리는 빈 트리를 사용.  
  
<- 결과 (높이 = 2)

**이진 트리의 기본 연산(Primitive operations on binary trees)**

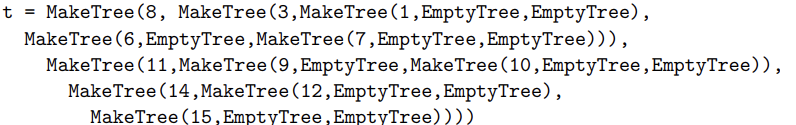
**생산자(constructors)**  
EmptyTree: 빈 트리를 반환합니다.  
MakeTree(v, l, r): 레이블이 v인 루트 노드와 두 개의 구성 이진 트리 l과 r로 이진 트리를 생성합니다.

**조건(condition)**  
isEmpty(t): 트리 t가 EmptyTree인 경우 true를 반환합니다.

**선택자(selectors)**  
root(t): 이진 트리 t의 루트 노드의 값을 반환합니다.  
left(t): 이진 트리 t의 왼쪽 서브트리를 반환합니다.  
right(t): 이진 트리 t의 오른쪽 서브트리를 반환합니다.

#편의를 위해 더 간단하고 읽기 쉬운 알고리즘 (**파생 생성자[derived operators]**)  
 Leaf(v) = MakeTree(v, EmptyTree, EmptyTree)  
레이블이 v인 단일 노드로 구성된 트리를 생성합니다. 이 노드는 동시에 트리의 루트이자 유일한 잎입니다.

ex. 파생 생성자  


ex.기본연산자  
선택자자는 비어 있지 않은 트리에만 적용할 수 있음을 유의하십시오. 예를 들어, 위에서 정의한 트리 t의 경우 다음이 성립합니다:

root(left(left(t))) = 1

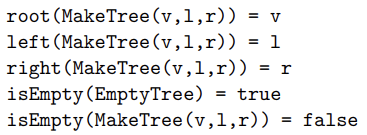
그러나 다음 식은 의미가 없습니다

root(left(left(left(t))))

이유는 left(left(left(t))) = EmptyTree 이고 빈트리는 루트를 가지지 않기 때문에 일반적으로 예외가 발생합니다. 그렇기 때문에 선택자를 사용하기 전에 항상 빈 트리인지 확인 해야 합니다.

**선택자 사용 전 체크 필요!**

다음 방정식은 기본 연산자 정의에서 명확해야 합니다.



**root(MakeTree(v, l, r)) = v**

트리에서 루트 값을 가져옵니다.

예: MakeTree(10, EmptyTree, EmptyTree) → root = 10.

**left(MakeTree(v, l, r)) = l**

왼쪽 서브트리를 반환합니다.

예: MakeTree(10, Leaf(5), EmptyTree) → left = Leaf(5).

**right(MakeTree(v, l, r)) = r**

오른쪽 서브트리를 반환합니다.

예: MakeTree(10, EmptyTree, Leaf(20)) → right = Leaf(20).

**isEmpty(EmptyTree) = true**

빈 트리인지 확인합니다.

**isEmpty(MakeTree(v, l, r)) = false**

노드가 있는 트리인지 확인합니다.

다음은 t가 비어 있지 않은 트리라는 가정 하에 의미가 있습니다



이는 비어 있지 않은 트리를 분해하고 조각을 사용하여 새로운 트리를 생성하면 동일한 트리를 다시 얻는다는 것을 의미합니다.

ADT는 이러한 구체적인 구현 방식과 독립적으로 설계와 알고리즘을 작성할 수 있도록 도와줍니다.

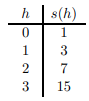
모든 것은 궁극적으로 컴퓨터에서 비트의 시퀀스로 표현되지만, 일반적으로 그렇게 낮은 수준의 용어로 생각할 필요는 없습니다.

**이진 트리의 높이(The height of a binary tree)**

이진 트리는 크기 n과 높이 h 사이에 단순한 관계를 가지지 않습니다. 노드가 n개인 이진 트리의 최대 높이는 (n−1)이며, 이는 모든 비-리프 노드가 정확히 하나의 자식 노드를 가지는 경우, 즉 트리가 체인처럼 형성될 때 발생합니다. 반대로, n개의 노드를 가지고 높이가 최소인 이진 트리를 생성하려고 한다면, 루트부터 시작하여 각 레벨을 순차적으로 '채우는' 방식으로 트리를 구성하면 됩니다. 즉, 이전 레벨이 모두 찬 뒤에야 다음 레벨에 추가할 수 있습니다.

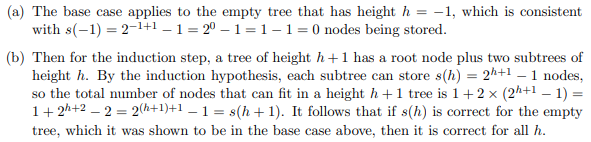
이러한 트리를 **완전 균형(perfectly balanced) 트리**(혹은 **높이 균형(height balanced) 트리**)라고 하며, 이는 다양한 목적에서 최적의 구조임을 확인할 수 있습니다. 기본적으로 이진 트리 기반의 많은 중요한 연산(예: 탐색)은 트리의 높이만큼의 단계가 필요하므로, 높이를 최소화하면 이러한 연산 시간을 줄일 수 있습니다.

**높이에 따른 최대 노드 수 계산**

높이 h인 이진 트리에 들어갈 수 있는 최대 노드 수를 s(h)라고 할 때 다음과 같은 관계식을 얻을 수 있습니다.

  
등비수열로 표현될 수 있습니다

**definition of a binary tree as follows**

****

수학적 귀납법을 통해 증명할 수 있습니다.   
기본 단계에서 빈 트리(높이 h=−1)일때 0 이므로 만족한다.  
귀납 단계에서 h+1인 트리는 s(h+1)=1+2×s(h)=1+2×(2^(h+1)−1)=2(h+2)−1 따라서 성립한다.

**높이 h와 노드 수 n의 관계**

트리의 높이와 노드 수 사이의 관계는 다음과 같이 나타낼 수 있습니다



완전 균형 트리는 n개의 노드가 있을 때 높이가 약 log2n로 매우 작습니다. 이는 트리가 균형을 유지하기 때문에 가능한 구조입니다.  
  
노드 수가 백만 개 이상이어도 높이는 겨우 20에 불과합니다. 이는 탐색 시간과 같은 연산 효율성에 큰 영향을 미칩니다.  
  
