**자료구조 과제 7 – 트리 조사**

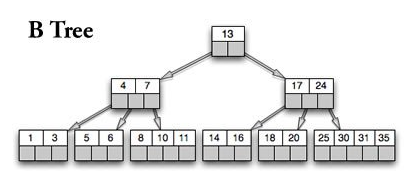
**201411802 응용통계학과 전민규**

**평형트리(Balanced Tree)**

-이진 탐색 트리는 값이 큰 것은 오른쪽에, 작은 것은 왼쪽에 몰리게 되므로 좌우 균형이 맞지가 않는다. 그에 대한 보완책으로 고안된 방법이 평형트리이다. 평형 트리 중에서 대표적인 예가 2-3-4 트리이다.(삽입과 삭제시 필요하면 스스로 균형을 유지함)

-AVL트리, 2-3-(-4)트리, Red\_Black Tree, B-Tree등이 있음

**B-트리**



\*개요

-대용량의 파일을 효율적으로 관리하기 위한 자료구조

-산업 분야에서 데이터베이스 시스템에 전반적으로 사용되고 있는 de facto표준

-어떻게 하면 메모리에 올릴 수 없을 정도로 큰 용량의 인덱스에 효율적으로 접근할 것인가?

->파일 전체를 이진 탐색하여 레코드를 찾는것보다 인덱스를 이용하는 것이 더 빠름

\*특징

-B트리는 하나의 노드에 여러 개의 데이터를 저장할 수 있으며, 하나의 노드에 최대로 저장할 수 있는 데이터의 수를 order라고 한다. 만약 어떠한 B트리의 order가 m일 때, B트리는 다음과 같은 속성을 갖는다.

1) 각 노드는 최대 m개의 자식을 가질 수 있다.

2) 루트 노드와 leaf node를 제외한 모든 노드는 반드시 [m/2]개의 자식 노드를 가져야 한다.

3) 높이가 1 이상인 B트리의 루트 노드는 반드시 두 개 ㅇㅣ상의 자식 노드를 가져야 한다

4) 모든 leaf node는 같은 레벨에 위치해야 한다. 즉, leaf node와 루트 노드의 거리는 모두 같아야 한다.

->순차검색과 트리검색의 이점을 갖는데 Balanced Tree이므로 최악의 경우가 없으며 퀵정렬에서 소구간에 대한 삽입정렬이 성능이좋다

-자식을 두개만 가질 수 있는 이진 트리를 확장하여 더 많은 수의 자식을 가질 수 있도록 일반화함

-같은 수의 자료라 할 지라도 하나의 레벨에 더 많은 자료가 저장될 수 있으므로 전체적으로 트리의 높이가 낮아지는 장점이 있음

-자식 수에 대한 일반화를 하며 트리의 균형을 자동으로 맞추는 로직까지 갖춤

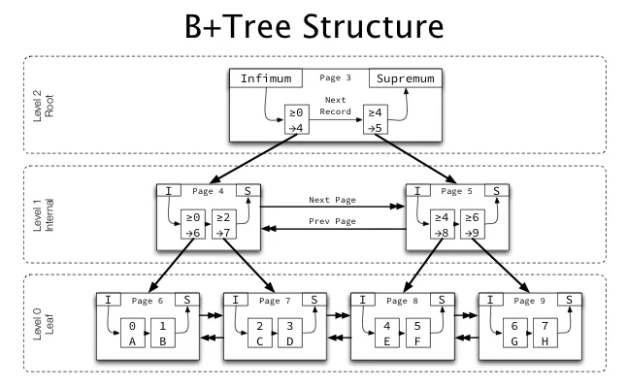
-단순하면서도 효율적

-> 레벨로만 따지면 완전히 균형을 맞춘 트리

-하나의 노드에 많은 데이터를 가질 수 있다는 점은 대량의 데이터를 처리해야 하는 검색 구조인 경우 큰 장점임.

Ex) 한 블록이 1024바이트라면, 2바이트를 읽으나 1024 바이트를 읽으나 입출력에 대한 비용은 동일함. 그렇다면 하나의 노드를 1024바이트가 되도록 조절한다면 입출력 aus에서 매우 효율적인 구성이된다. .게다가 균형까지 맞는다면 더욱 더 효율적인데 이것이 바로 B-트리의 장점이라 볼 수 있다.

**B+트리**



-B트리는 entry sequenced file을 이용하여 데이터 파일을 표현하지만 B+트리는 indexed sequential file을 이용함.

cf) Indexed sequenced

-인덱스를 통해 데이터 파일의 각 레코드에 접근할 수 있으며, 데이터 파일의 레코드가 정렬된 순서로 저장되어 있는 구조를 갖고 있다.

-인덱스를 통해 접근할 수 있다는 점이 entry sequenced file(B트리에서 사용되는)과 같지만, 데이터 파일의 레코드가 정렬되어 있다는 점이 다르다.

-그러나 데이터 파일을 정렬된 순서로 유지하는 것은 하드디스크 상에서 데이터가 삽입될 위치를 찾고, 그 뒤에 있는 데이터를 모두 shift해야 하기 때문에 막대한 비용이 소모된다. Indexed seqeuential file에서는 데이터 파일을 정렬하는 비용을 줄이기 위해 sequence set이라는 개념을 이용한다

cf) Sequence Set : key값의 순서에 따라 실제 물리적 위치가 정렬되어 있는 레코드들의 집합

\*특징

1) 삭제연산

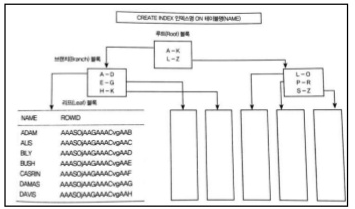
B+트리에서 삭제 연산은 해당하는 Block에서만 변화를 야기한다. 따라서 B+트리 전체에는 어떠한 변화도 발생하지 않으며 이런 단순함은 merge, redistribution등과 같은 복잡한 연산을 빈번하게 수행해야 했었던 B트리의 remove연산에 비해 매우 효율적이다.

-그러나 특정한 경우에는 B+트리의 삭제 연산에서도 underflow가 발생한다. 이 경우 B 트리와 같이 merge,redistribution 연산을 이용하여 트리를 재구성하면 된다.

2)삽입 연산

B+트리의 insert연산은 B트리와 마찬가지로 B트리의 leaf node에 해당하는 block에 데이터가 추가된다. 그 다음, overflow의 발생 여부를 확인하여 overflow가 발생하였다면, parent node로 이동하여 overflow에 대한 처리를 수행한다.

**B\*트리**



-B-트리의 문제점을 보완하기 위해 B-tree를 변형한 구조, 즉, B-tree에서는 제한 조건을 유지하기 위해 삽입과정에서의 분열과 삭제과정에서의 합병 등의 보조연산이 필요하다. B\*트리는 이런 보조연산을 감소시키려는 목적으로 구현되었다.

\*조건

-각 node는 2/3가량 채워지면 된다(B-tree는 1/2 이상)

-B\* tree는 node의 분열을 줄여서 보조연산을 줄이려고 한다. 따라서 node가 가득차면 분열하는 대신 이웃한 형제 node로 재배치를 한다.

-한 node가 가득차고 인접 node까지 모두 가득찰 때까지 분열을 지연한다.

\*삽입

-B-tree에서와 같은 방법으로 삽입을 한다

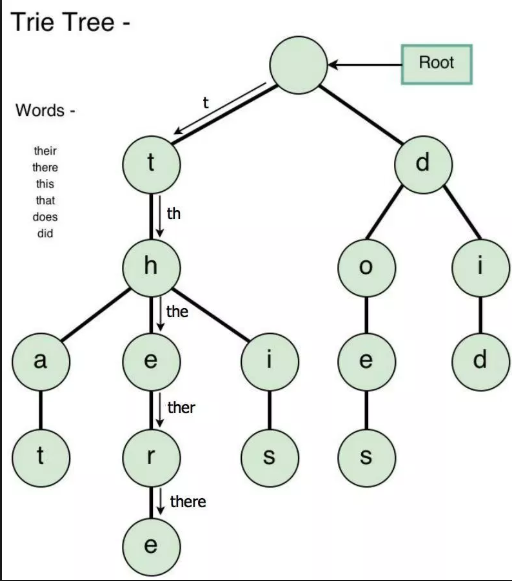
-node가 가득차면 이웃한 형제 node를 살펴보아 빈자리가 있으면 정렬하여 재배치한다.

-인접node에도 overflow가 일어나서 더 이상 빈자리가 없을 경우 가득찬 두 node를 분열하여 2/3정도 찬 3개의 node로 만든다.

\*삭제

-B-tree와 똑같이 삭제 후 key값의 개수가 모자라면 이웃한 형제 node로부터 재배치하고 재배치도 할 수 없는경우 합병한다. 합병할 때는 세 개의 node를 2개의 node로 합병한다.

**Trie**



-하나의 key값의 일부분만을 사용하여 분기하는 tree구조이다. 보통 하나의 node는 m개의 포인터로만 구성된 1차원 배열이며, 배열의 각 원소의 주소는 그 주소에 대응하는 숫자를 나타낸다.

\*성능

-탐색 시 B+tree와 같이 leaf node까지 가야만 key값을 찾을 수 있다. 하지만 찾는 key값이 없을 경우 어느 레벨에서든지 끝낼 수 있다. 따라서 key값이 없는 경우의 탐색 효율이 좋다.

\*삽입

-새로운 key값이 삽입이나 삭제될 leaf node를 찾기 위해서는 특정 key값을 직접 탐색해야한다.

-삽입 시 새로운 node가 들어갈 leaf node가 없는 경우 index부분에도 node를 삽입해야 한다.

-위치가 결정되면 해당 위치의 NULL을 고쳐 key값이 존재한다는 것을 나타내주거나 key값의 record주소를 저장해야 한다.

\*삭제

-해당 위치의 pointer를 NULL로 만들어 연결을 끊는다

->삽입이나 삭제 시 node의 분열이나 합병이 없다.

\*특징

1)가변길이의 key field 수용

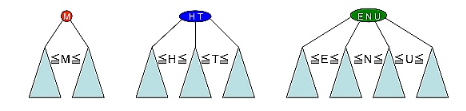
- 예를들어 영문자로만 구성된 이름을 key값으로 사용하며 최대 key값은 30으로 구성될 때 높이 30인 26-one-trie가 필요하다

-만약 key값의 대부분이 8문자 이내라면 각 node에 27개의 원소를 가지도록 node를 구성하고 마지막 원소는 특수문자를 사용하여 이름이 끝난다는 표시를 나타내게 한다. 그리고 trie를 8 level로 구성하여 8자 이상인 이름은 이름의 나머지를 가지고 있는 다른 장소를 가리키도록 하여 이름의 나머지를 찾도록 한다

2)기억 장소 감소

-영문자를 사용하는 key값에서 모든 알파벳 문자를 나타내지 않고 실제 사용하는 알파벳만을 나타내도록 하면 모든 node가 26개의 key값을 가질 필요가 없으므로 기억장소를 절약할 수 있음

**2-3-4트리**



\*특성

- 각각의 잎의 깊이는 모두 같다.

- 3 종류의 정점이 존재한다.(2노드, 3노드, 4노드)

- 2노드 : 한 개의 데이터와 2개의 자식 노드

- 3노드 : 두 개의 데이터와 3개의 자식 노드

- 4 노드 : 세 개의 키와 4개의 자식 노드

\*2-3-4트리에서의 삽입

1)삽입 위치를 탐색한다.(뿌리부터 잎의 방향으로 전개)

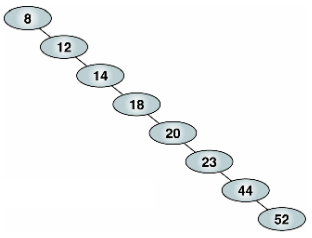
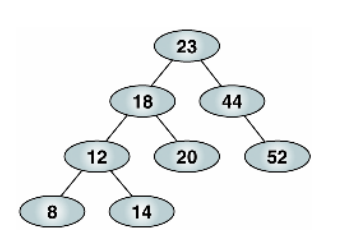
2)4노드를 발견

3)2개의 2노드로 분할

4)탐색 재개

5)찾은 위치에 데이터 삽입

**AVL트리**

 **=>** 

\*특성

-원래 이진트리는 왼쪽의 경우가 나올 수 있는데 이를 극복하여 오른쪽 트리로 만들기 위해 AVL 트리가 탄생

-서브트리의 높이를 적절하게 제어해 전체 트리가 어느 한쪽으로 늘어지지 않도록 한 이진탐색트리의 일종(균형된 트리를 만들기 위해!!)

-트리의 높이가 h일 때 이진탐색트리의 계산복잡성은 O(h)이기 때문에 균형된 트리를 만들어 h를 줄이고자 하는 발상에서 비롯됐다.

-Binary Search Tree에서 가장 초기에 Balanced를 제시한 트리.

-일반 Binary Search Tree는 탐색속도가 빠르다는 장점이 있지만, 편향 트리일 경우 단방향 연결리스트의 탐색과 같은 속도를 내기 때문에 Binrary Search Tree의 장점을 없앰.

-> AVL은 Balance를 맞춰서 BST의 장점이 사라지지 않도록함!

\*AVL 트리란

-각각의 노드마다 왼쪽 서브트리의 높이를 오른쪽 서브트리의 높이로 뺀 값인 균형치를 갖고 있으며 이 값은 +1 or -1이어야함. Height Balanced Tree(높이 균형 트리)라고도 함.

-삽입과 삭제를 할 때 트리의 높이가 달라지게 되는데 만약 어떠한 노드라도 균형치가 +2 or -2일 경우 RR,LL,RL,LR의 4가지 방식을 이용해 회전을 시킴.

\*AVL트리의 장점

-탐색속도가 빠르다는것과 전체를 재배열시키지 않아도 트리의 균형이 유지된다는 점

-이진 탐색 트리의 경우 삽입이 무작위로 될 경우 한쪽으로 편향트리가 될 가능성이 있다. 그럴경우 탐색 시 최대 비교횟수가 위에서 말했듯이 트리의 노드 개수로서 단방향 연결리스트와 다를 바 없다. 하지만 AVL트리는 이미 균형이 이루어져 있기 때문에 탐색이 O(logn)을 넘지 않는다.

-트리의 균형은 노드를 삽입이나 삭제할 때 깨질 수 있다. 삽입/삭제 연산을 할 경우 근접한 노드 뿐 아니라 루트 까지의 경로에 있는 노드에도 영향을 줄 수 있다. 만약 그 영향으로 균형치가 +2 or -2가 되는 노드가 생긴다면 그 노드의 서브트리들을 회전하여 트리의 균형을 잡는다.

\*동작 구조(rotation)

-거의 모든 동작 구조가 기본적인 이진 탐색트리와 비슷하지만, 노드의 삽입과 삭제는 노드들을 회전해야 하므로 노드를 재배열 시키지 않는 이진 탐색 트리와는 차이가 있다.

-LL회전 : 새로 삽입된 노드가 가장 가까우면서도 균형치가 맞는 노드의 왼쪽 서브트리의 왼쪽 서브트리로 삽입되는 경우

-RR회전 : 새로 삽입된 노드가 가장 가까우면서도 균형치가 맞는 노드의 오른쪽 서브트리의 오른쪽 서브트리로 삽입되는 경우

-LR회전 : 새로 삽입된 노드가 가장 가까우면서도 균형치가 맞는 노드의 왼쪽 서브트리의 오른쪽 서브트리로 삽입되는 경우

-RL회전 : 새로 삽입된 노드가 가장 가까우면서도 균형치가 맞는 노드의 오른쪽 서브트리의 왼쪽 서브트리로 삽입되는 경우

\*응용분야

-이진탐색트리를 사용할 수 있는 분야에선 모두 사용이 가능하며, 컴퓨터의 디렉토리 구조나 족보 같은 계층 구조를 나타내야 할 때 쉽게 표현 가능

\*문제점

-삽입, 삭제 시 원하는 노드를 찾기 위해 2개의 경로가 필요하기 때문에 RL(RED-BLACK TREE)만큼 효율이 좋지 않아 자주 쓰이지는 않는다.

**Red Block(RB)트리**

-이진트리의 범주를 벗어나지 않으면서 색상이라는 속성을 노드에 추가함으로써 자동으로 균형을 맞추는 알고리즘

아래는 RB트리의 조건들이다.

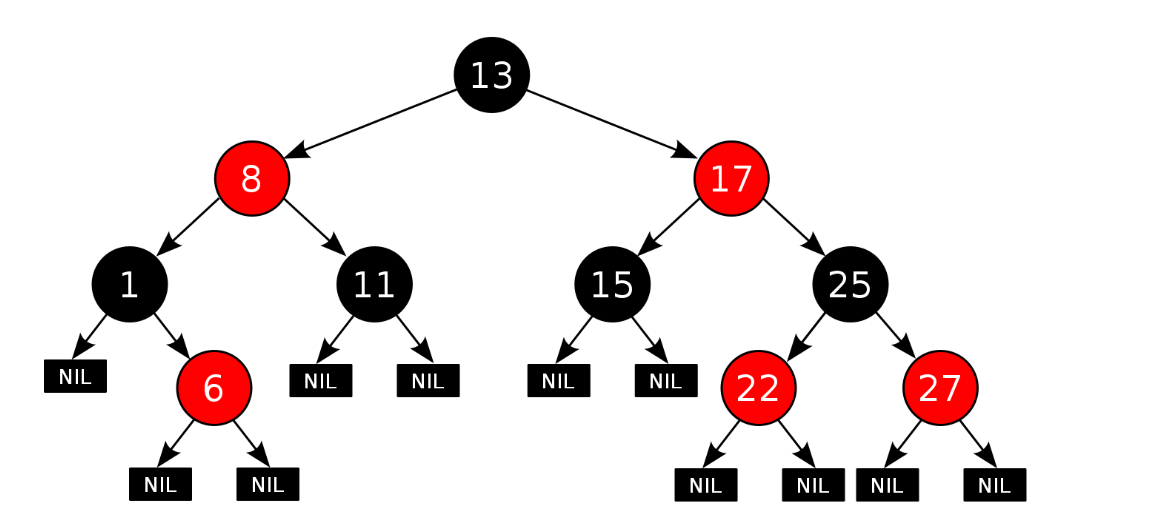
1. 모든 노드는 빨간색, 검은색 둘 중 하나다.

2. 루트노드는 검은색이다.

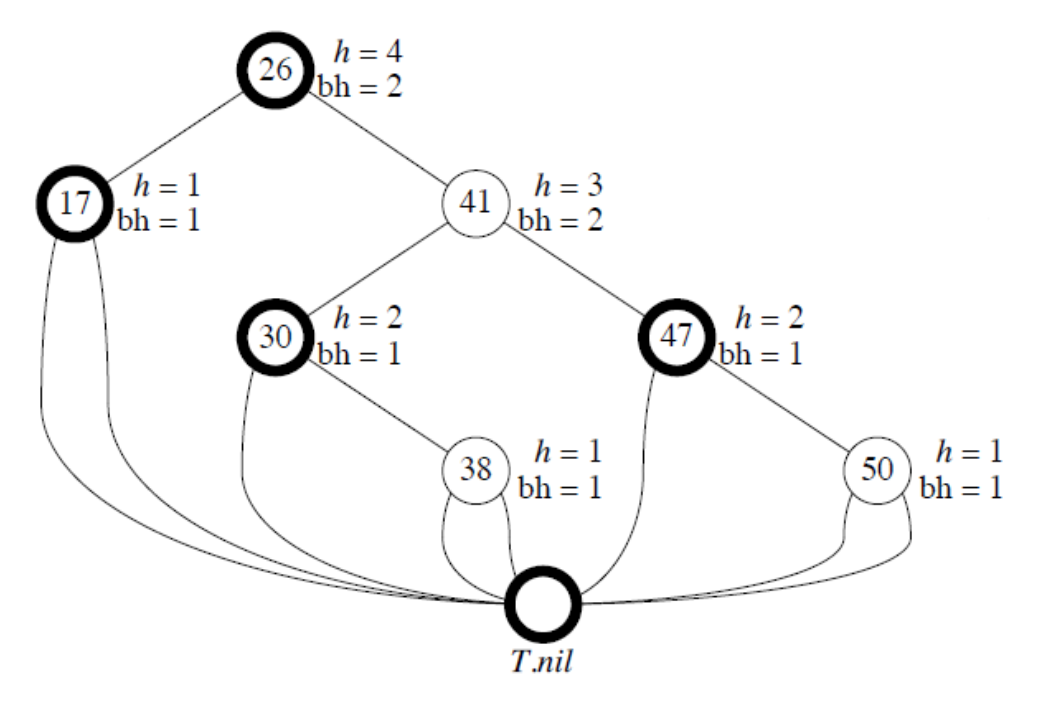
3. 모든 잎새노드(NIL)는 검은색이다

4. 어떤 노드가 빨간색이라면 두 개 자식노드는 모두 검은색이다.(따라서 빨간색 노드가 같은 경로상에 연이어 등장하지 않는다)

5.’각 노드~자손 잎새노드 사이에 모든 경로’에 대해 검은색 노드의 수가 같다.



임의의 노드 x의 Black-height는 x부터 잎새노드에 이르는 경로상에 있는 검은색 노드의 수로, bh(x)라고 표기함. X가 검은색 노드일 경우 1을 빼주며, 잎새노드(NIL)의 bh는 0이다.

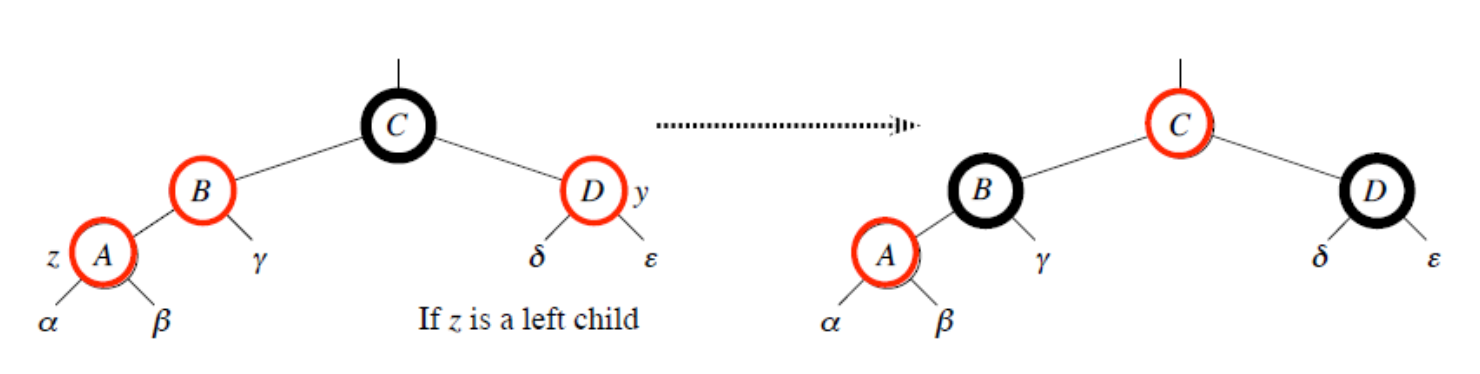


위 설명을 바탕으로 그림을 보면 맨 위 26의 경우에 높이 h는 4이고 bh는 x(=26)부터 잎새노드(T.nil)에 이르는 경로상에 있는 검은색 노드의 수 총 3개에서 x(=26)이 검은색 노드이므로 1을 빼준다. 그 결과 bh=2가 된다.

\*삽입과정

삽입할 새 요소 z를 빨간색이라 두고 z와 검은색 부모노드를 공유하는 형제노드를 y라고 두자.

1. y가 빨간색이면서 z가 left child인 경우 아래 그림에서 z가 삽입되면서 빨간색 노드가 연이어 등장하므로 RB트리의 조건에 위배됨. 이 경우 노드의 색을 바꿔서 다시 칠하는데, y가 빨간색이면서 z가 right child인 경우도 색을 바꿔서 다시 칠한다.



2. y가 검은색이면서 z가 right child인 경우에도 빨간색 노드가 연이어 등장하므로 RB트리의 조건에 위배됨. 이 경우 double rotation을 수행함.

3.y가 검은색이면서 z가 left child인 경우에 single rotation을 수행함

\*이해하기 어렵고 구현이 복잡하지만 B-트리에 비해 메모리 오버헤드가 적어 널리 이용됨.

**DFS(Depth First Search, 깊이 우선)탐색**

-트리와 그래프 같은 자료구조에서 스택을 이용하여 데이터를 탐색할 때 사용되는 알고리즘

-더 이상 나아갈 길이 보이지 않을만큼 깊이 찾아가면서 탐색함. 만약 나아갈 길이 존재하지 않는다면 이전의 위치로 돌아와 찾아가지 않은 다른 길로 뻗어 나가면서 탐색해 나감(스택!)

\*장점

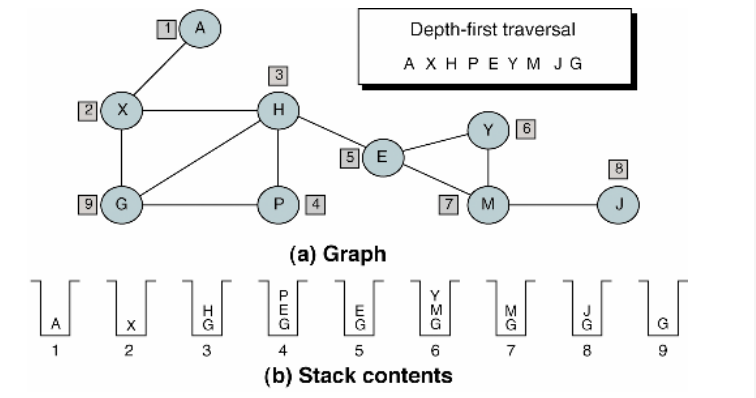
-단지 현 경로상의 노드들만을 기억하면 되므로 저장공간의 수요가 비교적 적다

-목표노드가 깊은 단계에 있을 경우 해를 빨리 구할 수 있다(말그대로 깊이 우선 탐색이므로!)

\*단점

-해가 없는 경로에 깊이 빠질 가능성이 있다. 따라서 실제의 경우 미리 지정한 임의의 깊이까지만 탐색하고 목표노드를 발견하지 못하면 다음의 경로를 따라 탐색하는 방법이 유용할 수 있다.

-얻어진 해가 최단 경로가 된다는 보장이 없다. 이는 목표에 이르는 경로가 다수인 문제에 대해 깊이 우선 탐색은 해에 다다르면 탐색을 끝내버리므로, 이때 얻어진 해는 최적이 아닐 수 있다는 의미이다.



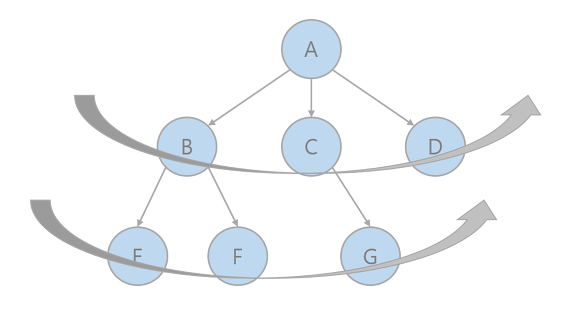
그림에서 보다시피 A-X-H-P-E-Y-M-J-G 순으로 탐색이 이루어지는데 과정은 다음과 같다

-vertex(트리에서의 노드) 옆의 숫자는 탐색 순서이다. A-X다음에 G,H 선택지가 두 개가 있는데 G와 H를 스택에 넣은 후, G는 내버려 두고 H의 자식들로 이동한다. 형제보다는 자식을 우선으로 탐색한다.

-그래프는 트리와는 다르게 arc(가중치)관계가 얽혀 있어서 누가 부모고 누가 자식인지 쉽게 구별하기가 힘들다. 이럴 때 시작점(A)과의 최단거리를 보면 된다. X는 A로부터 거리가 1이고 G와 H는 A로부터의 거리가 2이므로 G와 H는 X의 자식이 된다. 누구의 자식이 될 것인가는 아크 배치 순서에 따라 다르며, 이 상황에서는 P와 E 모두 H의 자식으로 두었다.

**BFS(Breath First Search, 너비 우선 탐색)**

-DFS의 반대개념이며 그래프 전체를 탐색하되, 인접한 노드들을 차례대로 방문하도록 구현



위 그림을 통해 진행과정을 설명하면 우선 노드 A에서 시작하여 BFS로 모든 노드를 방문하는 것을 나타냄.

-A에 인접한 노드는 B,C,D이므로 차례대로 해당 노드들을 방문해주고, 그 후 B,C,D에 인접한 노드들을 방문

-B에 인접한 노드 E,F와 노드 C에 인접한 노드 G를 방문해주면 그래프에 존재하는 모든 노드를 방문했기 때문에 전체 탐색을 끝마치게 됨

-> ‘그래프’를 ‘시작점을 루트로 하는 트리’ 라고 생각한다면, height가 작은 노드부터 차례대로 방문하는 전체 탐색 방식

\*queue를 이용한 구현

1)queue의 가장 앞에 있는 노드를 pop

2)현재 노드에 인접한 모든 노드들 중 아직 방문하지 않은 노드들을 queue에 push

3)queue가 비어있지 않다면, 1)번부터 다시 실행

Cf) push : 나중에 방문할 노드 저장 , pop : front에 있는 노드 방문

\*특징 및 DFS와의 비교

-DFS와의 가장 큰 차이로, 여러 갈래 중 무한한 길이를 가지는 경로가 존재하고 탐색 목표가 다른 경로에 존재하는 경우 DFS로 탐색할 시 무한한 길이의 경로에서 영원히 종료하지 못하지만, BFS의 경우는 모든 경로를 동시에 진행하기 때문에 탐색이 가능하다

-BFS는 한 갈림길에서 연결되는 모든 길을 한번씩 탐색하기 때문에 시작점에서 끝점까지의 최단경로를 알아낼 수 있다.

**다익스트라 알고리즘**

-너비우선탐색(BFS)를 기본으로함.

하나의 정점(vertex)에서 다른 모든 정점까지의 최단 경로를 구하는 알고리즘으로 간선들은 모두 양의 간선(arc)들을 가져야 한다. 기본적인 로직은 첫 vertex를 기준으로 연결되어 있는 arc node들을 추가해가며 최단 거리를 갱신하는 것이다. 정점을 잇기 전까지는 시작점을 제외한 모든 정점들은 모두 무한대의 값을 갖는다.

정점 A에서 정점 B로 이어지면 정점 B가 갖는 최단 거리는 시작점부터 정점 A까지의 최단거리 + 점A와 점B 간선(arc)의 가중치와, 기존에 있던 정점 B의 거리 중 작은 값이 B의 최단 거리가 된다.

-우선순위 큐를 이용

우선 순위큐와 일반 큐를 써도 결과는 같지만 속도에 더 큰 이점이 있다. 기존 일반 다익스트라에서는 가장 최단 거리를 가진 정점을 찾기위해 선형 탐색을 이용하지만 MinHeap을 이용하면 최솟값이 가장 위로 올라오도록 빠르게 정렬되기 때문이다.

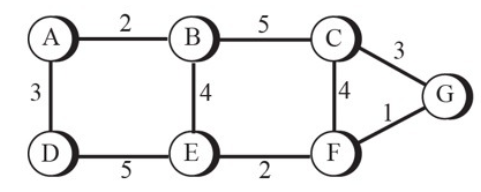
-O(E\*logV)의 시간 복잡도를 가짐(우선순위 큐를 사용할 때)

\*탐색 과정(Logic)

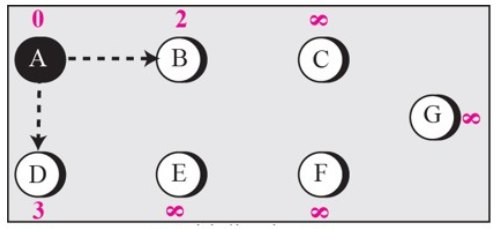
-한 정점(노드)를 선택한다

-아직 확인되지 않은 노드의 cost(거리)는 무한대로 초기화 시킨다

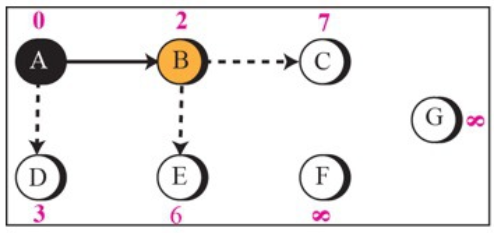
-cost가 적은 node들을 방문하며 모든 노드까지의 최단경로를 찾는다



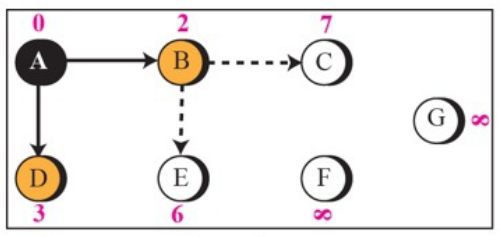
이 그래프에서 A가 선택된 노드라고 생각해보자



1) 자기 자신으로 가는 cost는 0으로 초기화해주고, 인접 노드들을 제외한 노드들을 모두 무한대값으로 초기화한다

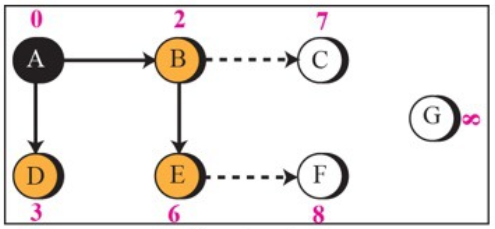


2) A->B까지 가는 경로가 2로 최소이기 때문에 B를 방문하고 B를 2로 기록한다. B->C는 A->B->C이므로 7, B->E는 A->B->E이므로 6으로 본다.

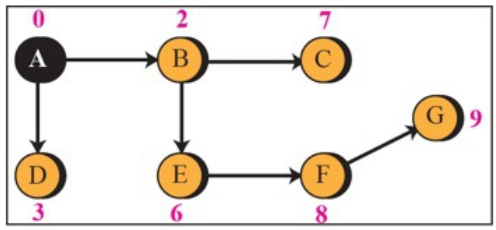


3)다음으로 CHLTHTRKQT이 A->D, 3이므로 D를 방문하고 기록한다

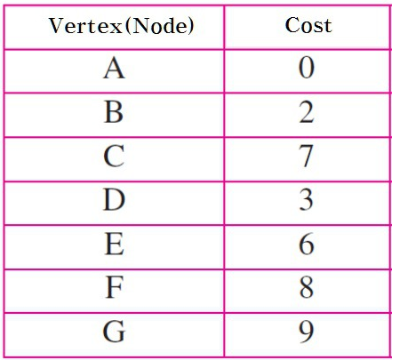
여기서 D->E도 갈 수 있지만 A->B->E의 경로보다 cost가 높기 때문에 생각하지 않는다.



4) 그 다음 최솟값인 E를 반문해준다. 이러한 방식으로 계속해서 진행하면서 모든 node를 방문해 A에서의 모든 노드까지의 최단경로를 알 수 있다.



5) 결국 위와같이 모든 노드를 방문하여 cost(거리)를 기록할 수 있다

 표로 나타낼 수 있다.

\*특징

-다익스트라 알고리즘을 사용하면 출발 노드(vertex) 하나로부터 각 모든 vertex에 대한 최단 경로를 구할 수 있음

-보통 시작점 하나에 대한 최단 경로만 알려주지만, 모든 점에서의 최단경로를 찾고싶다면 모든 점에서 다익스트라 알고리즘을 한 번씩 실행하면 됨

\*단점

- 가중치가 음수인 경우는 작동하지 않는다.

- 만일 음수인 가중치를 가진 edge가 포함된 그래프에 대해 다익스트라 알고리즘을 적용하려고 할 경우, 그래프 전체에서 가장 작은 가중치의 절댓값만큼을 모든 edge가중치에 더해준 뒤 다익스트라 알고리즘을 적용하고, 적용이 끝난 이후 해당 절댓값만큼을 다시 빼서 결과를 내는 방식으로 수행할 수도 있음.

**벨만 포드 알고리즘**

-그래프 상에 존재하는 두 노드간의 최단거리를 구하고 싶을 때 이용가능

-다익스트라에서 할 수 없었던 음의 가중치도 계산할 수 있도록 한 방식이지만 다익스트라보다 시간 복잡도가 높기에(더 느림) 어떤 상황에서 이용할지 잘 생각하여 써야한다.

\*전제조건

1) 최단 경로는 사이클을 포함할 수 없기 때문에, 최대 |V|-1개의 간선만 사용가능

-3개의 노드가 있을 때, 2개까지 간선만 허용한다는 의미

2) 최단 거리가 업데이트 되는 노드가 없어질 때 까지 계속해서 반복하여 구해주고, 음의 가중치로 인해 업데이트를 무한히 하게 되는 경우 탈출 시켜줘야함.(무한히 반복할 때는 최단거리가 없다고 함)

**A\* 알고리즘**

\*A스타를 사용하는 이유

-다익스트라의 현실 적용이 매우 어렵기 때문

-예를들어 네트워크 같은 디지털적인 공간이 아닌, 현실의, 사람이 사는 공간을 생각해보면 사람이 다닐 수 있는 거리는 아날로그이다. 이것들을 전부 노드화시키기에는 그 수가 엄청나게 많아질 수 있다. 그렇다면 탐색해야 하는 공간도 그만큼 커지게 되고, 시간 복잡도 또한 커진다.

-잘 노드화시켜서 다익스트라를 사용할 수 있는 상황을 만들어서 경로를 발견했다고 해도 그렇게 탐색한 경로가 자동차 정체 구간, 출근길 등 다양한 변수로 인해 오히려 더 느려질 수 있는 경우도 발생할 수 있다. 이러한 변수 때문에 A\* 알고리즘을 사용함

\*방법

- 갈 길 X에 대해 그 길을 통과하는 최상의 경로를 추정하는 순위값인 “휴리스틱 추정값” h(x)를 매기는 방법을 쓴다.

- 조사하지 않은 노드 중에서 가장 효율적이라고 판단되는 노드를 찾는다

- 찾아진 노드가 도착점이면 종료하고 아니면 인접한 다른 노드들에서 찾는다.

- 조사한 노드들은 Close List에 담고 조사하지 않은 노드들은 Open List에 담는다.

\*특징

-경로가 존재한다면 하나의 경로를 찾는다.

-경로가 존재하지 않는다면 모든 경로를 찾는다.

-최적의 경로를 찾는다.(비용 계산이 올바르다면)

-휴리스틱을 효율적으로 사용한다

-평균적으로 빠른 성능을 보인다

**Floyd warshall 알고리즘**

\*특징

-다익스트라, 벨만 포드 알고리즘은 하나의 시작점에 대해 최단 경로를 찾아주지만 플로이드 워셜 알고리즘은 모든 정점에 대해 모든 다른 정점에 대한 최단 경로를 다 구해준다

-이 때문에 시간이 많이 걸린다

-다익스트라 알고리즘과는 다르게 음수 가중치를 갖는 간선도 순환만 없다면 잘 처리가 가능

-구현 시 제일 바깥쪽 반복문은 거쳐가는 꼭짓점이고, 두 번째 반복문은 출발하는 꼭짓점, 세 번째 반복문은 도착하는 꼭짓점

-동적 계획법 접근으로, 그래프 상의 모든 두 정점을 잇는 경로의 최소 비용을 구한다. 여기에 경유지를 기록하면 최소 비용 경로까지 구할 수 있다.

Cf) 동적 계획법

-복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법을 말함. 이것은 부분 문제 반복과 최적 기본 구조를 가지고 있는 알고리즘을 일반적인 방법에 일반비해 더욱 적은 시간 내에 풀 때 사용한다.

-일반적으로 주어진 문제를 풀기 위해서, 문제를 여러 개의 하위 문제로 나누어 푼 다음, 그것을 결합하여 최종적인 목적에 도달하는 것이다.

-각 하위 문제의 해결을 계산한 뒤, 그 해결책을 저장하여 후에 같은 하위 문제가 나왔을 경우 그것을 간단하게 해결할 수 있다.

-위와 같은 방법으로 계산 횟수를 줄일 수 있다. 특히 하위 문제의 수가 기하급수적으로 증가할 때 유용하다.

**Kruskal’s 알고리즘**

\*개요

-Greedy Method를 기초로 한다

Cf)Greedy Method : 당장 눈앞의 최소 비용의 선택을 해서 결과적으로 최적의 Solution을 찾는다

-시작점을 정하지 않고 최소 비용의 간선을 차례대로 대입하면서 MST를 구성하므로, 그 과정에서 사이클을 이루는지 항상 확인해야 한다. 싸이클을 확인하는 방법으로는 Union-Find(Disjoint-Set)방법이 있다.

Cf)MST(최소 신장 트리) : 최소 비용으로 만들어진 신장 트리. 가중치의 합이 가장 작은 신장 트리

\*특성

-시간 복잡도는 Prim과 비슷하지만, 일반적으로 Dense하지 않은 그래프의 경우 kruskal이 유리함

-간선을 weight 기준으로 정렬하는 과정이 오래걸린다 (정렬에 대부분의 시간이 사용됨)

-알고리즘 순서는 정렬 – 연결 – cycle 체크 순이다

\*알고리즘 작동 순서

1) 그래프의 각 꼭짓점이 각각 하나의 tree가 되도록 하는 ‘숲(F)’을 만든다

2) 모든 변을 원소로 갖는 집합 S를 만든다

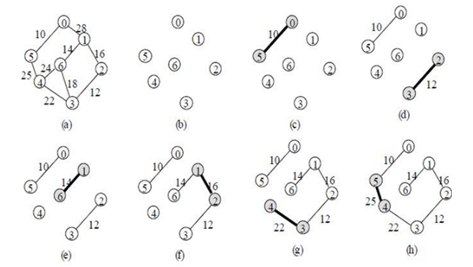
3) S가 비어있지 않는 동안

- 가장 작은 가중치의 변을 S에서 하나 빼낸다

- 그 변이 어떤 두 개의 나무를 연결한다면 두 나무를 연결하여 하나의 나무로 만든다

- 그렇지 않다면 그 변은 버린다

4) 알고리즘이 종료됐을 때 숲 F는 하나의 최소 비용 생성 나무만을 갖게 됨



이는 Kruskal 알고리즘의 각 단계들이다.

**Prim 알고리즘**

-가중치가 있는 연결된 방향이 없는 그래프의 모든 꼭짓점을 포함하면서 각 변의 비용의 합이 최소가 되는 부분 그래프인 트리(최소 비용 생성나무를 찾는 알고리즘)

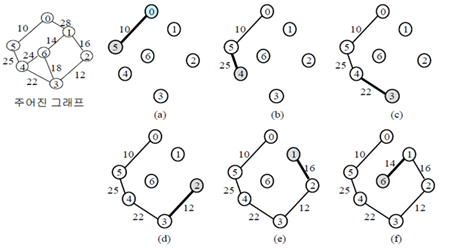
\*개요(작동 순서)

1) 그래프에서 하나의 꼭짓점을 선택하여 트리를 만든다

2) 그래프의 모든 변이 들어 있는 집합을 만든다.

3) 모든 꼭짓점이 트리에 포함되어 있지 않는 동안 트리와 연결된 변 가운데 트리 속의 두 꼭짓점을 연결하지 않는 가장 가중치가 작은 변을 트리에 추가한다.

4) 알고리즘이 종료됐을 때 만들어진 트리는 최소 비용 신장 트리가 된다.



-Prim 알고리즘을 사용하여 정점 0에서 시작하여 최소비용 신장트리를 구하는 과정

\*특성

-Greedy Method를 기초로 하고 있음

-정점 위주의 알고리즘(크루스칼은 간선 위주)

-시작점을 정하고, 시작점에서 가장 가까운 정점을 선택하면서 MST를 구성하므로 그 과정에서 사이클을 이루지 않는다

-시간 복잡도는 Kruskal과 비슷하지만 Dense한 그래프의 경우 프림이 유리

-최소 거리의 정점을 찾는 부분에서 자료구조의 성능에 영향을 받음