DB Index

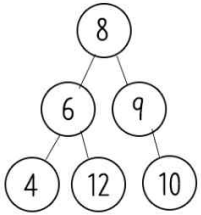
백설기 5차 스터디

AI6/1반 조해정

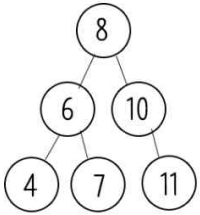
지난 스터디...Binary Search Tree

이진 탐색 트리(Binary Search Tree)

- 모든 원소는 **서로 다른 유일한 값**을 갖는다. (검색 목적 자료구조이므로)
- **모든 왼쪽 자식**의 값이 루트나 부모**보다 작고**, **모든 오른쪽 자식**의 값이 루트나 부모**보다 큰** 값을 가진다.
- 이진 탐색 트리를 통해 찾고자 하는 **원소를 보다 빠르고 효율적으로 찾거나 추가할 수 있다는 장점**이 있다.
- 이진 탐색 트리는 균형 잡힌 트리가 아닐 때(=편향 트리), 입력되는 값의 순서에 따라 한쪽으로 노드들이 몰리게 될 수 있다.
- 이진 탐색 트리를 순회할 땐 **중위순회** 방법을 사용한다.



Binary tree



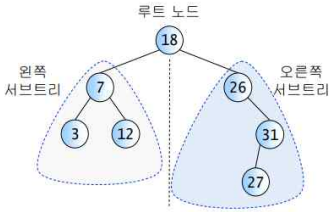
Binary search tree

글, 그림 참조:
https://it-and-life.tistory.com/164
https://richard25.tistory.com/73

지난 스터디...Binary Search Tree

이진 탐색 트리의 연산 시간 복잡도

이진 탐색 트리의 세 개 연산(검색, 삽입, 삭제)은 결국 트리를 순회하며
타겟 데이터의 위치를 찾는 연산이 공통적으로 필요하므로,
트리의 높이에 비례하여 시간 복잡도가 증가한다.



포화 이진 탐색 트리 :
모든 노드가 빈 자리 없이 꽉꽉 채워져 있는 이진 탐색 트리
가장 최적의 상황으로 구성된 트리

포화 이진 탐색 트리를 구성하고 있는 노드의 개수를 N, 높이를 H라고 했을 때,

$$N = 2^{(H+1)} - 1$$

노드의 개수를 통해 트리의 높이를 계산하면,

$$H = \log_2(N+1) - 1$$

따라서, 균형적으로 생성되어 있는 트리의 높이를 H라고 한다면, H에 비례하여 O(H) 시간 복잡도를 가지게 되는데,
이를 Big-O 시간 복잡도로 표현하면 **O(logN)**으로 표기할 수 있다.

글, 그림 참조:
https://velog.io/@haero_kim/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%AC-Binary-Search-Tree

지난 스터디...Binary Search Tree

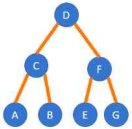
이진 탐색 트리의 연산 시간 복잡도

이진 탐색 트리는 **트리의 높이에 비례하여 시간 복잡도가 증가**한다.

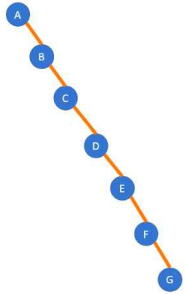
- **균형** 잡힌 이진 탐색 트리라면 시간 복잡도가 **O(logN)**이지만,
- **편향** 이진 탐색 트리라면 시간 복잡도가 **O(N)**이 된다.
- 이진 탐색 트리의 균형 여부는 검색 성능에 지대한 영향을 끼친다.

>> 해결 방법:

AVL 트리, 2-3 트리, 2-3-4 트리, Red-Black 트리, **B 트리**



완전 이진 탐색 트리



편향된 이진 탐색 트리

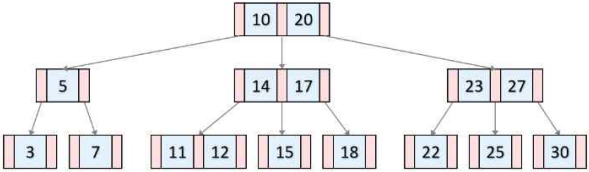
글, 그림 참조:
https://velog.io/@haero_kim/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%AC-Binary-Search-Tree

CONTENTS

B-Tree

B-Tree는 **탐색 성능을 높이기 위해** 균형 있게 높이를 유지하는 Balanced Tree의 일종이다.
이진 트리와 다르게 **하나의 노드에 많은 수의 정보**를 가지고 있을 수 있다.
최대 M개의 자식을 가질 수 있는 B트리를 **M차 B트리**라고 한다.

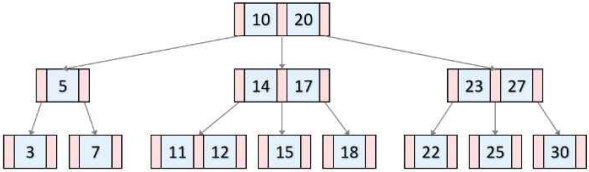
아래의 그림은 3차 B트리이다.
파란색 부분은 각 노드의 key이고, 빨간색 부분은 자식 노드들을 가리키는 포인터이다.



글, 그림 참조:
https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%8C%EC%95%84%EB%B3%B4%EB%8A%94-B-Tree

B-Tree

- **특징**:
- 노드는 최대 M개부터 (M/2)개까지의 자식을 가질 수 있다.
- 노드에는 최대 (M-1)개부터 (M/2-1)개의 키가 포함될 수 있다.
- 노드의 키가 x개라면 자식의 수는 $(x+1)$개이다.
- 최소차수는 자식수의 하한값을 의미하며, 최소차수가 t이면 $M = 2t - 1$ 을 만족한다.
- key들은 노드 안에서 항상 정렬된 값을 가지며, 이진 탐색 트리처럼 각 key들의 왼쪽 자식들은 항상 key보다 작은 값을, 오른쪽은 큰 값을 가진다.

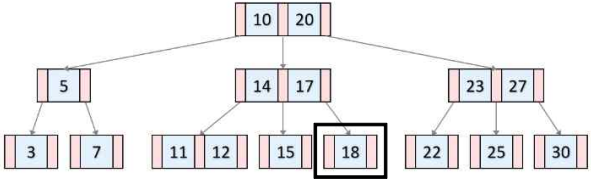


글, 그림 참조:
https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%8C%EC%95%84%EB%B3%B4%EB%8A%94-B-Tree

B-Tree Key 검색 과정

루트 노드에서 시작하여 하향식으로 검색을 수행한다.

검색하고자 하는 key의 값을 K라고 가정했을 때,
1. 루트 노드에서 시작하여 key들을 순회하면서 검사한다.
2. 만일 K와 같은 key를 찾았다면 검색을 종료한다.
3. 같지 않다면 K와 key들의 대소관계를 비교하여, 어떤 key들 사이에 K가 들어간다면 그 자식 노드로 내려간다.
4. 1-3과정을 리프노드에 도달할 때까지 반복하고, 리포 노드에도 K와 같은 key가 없다면 검색을 실패한다.



글, 그림 참조:
https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%8C%EC%95%84%EB%B3%B4%EB%8A%94-B-Tree

B-Tree Key 삽입 과정

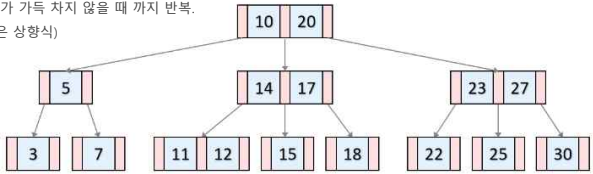
B-Tree Visualization :
https://www.cs.usfca.edu/~galles/visualization/BTree.html

삽입하고자 하는 key의 값을 K라고 가정했을 때,

1. 트리가 비어있으면 루트 노드를 할당하고 K를 삽입한다.
   만일 루트 노드가 가득 찼다면, 노드를 분할하고 리프노드가 생성된다.
2. 이후부터는 삽입하기에 적절한 리프 노드를 찾아 K를 삽입한다.
   삽입 위치는 노드의 key값과 K값을 검색 연산과 동일한 방법으로 비교해서 찾는다. (>> 검색은 하향식)

3-1. 리프 노드가 가득차지 않았다면 오름차순으로 K를 삽입한다.

3-2. 리프 노드에 key 노드가 가득 찬 경우(최대 key 개수 초과), 중앙값에서 분할을 수행한다.
   중앙값은 부모 노드로 병합하거나 새로 생성, 왼쪽 키들은 왼쪽 자식으로, 오른쪽 키들은 오른쪽 자식으로 분할.
   부모 노드가 가득 차지 않을 때 까지 반복.
   (>> 분할은 상향식)



글, 그림 참조:
https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%8C%EC%95%84%EB%B3%B4%EB%8A%94-B-Tree

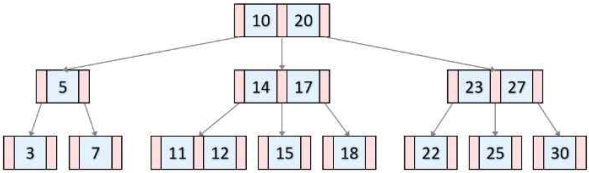
B-Tree Key 삽입 과정

B-Tree Visualization :
https://www.cs.usfca.edu/~galles/visualization/BTree.html

삭제하고자 하는 key의 값을 K라고 가정했을 때,

1. 삭제할 key(=K)가 있는 노드를 검색한다.
2. key(=K)를 삭제한다.
3. 필요한 경우, 트리의 균형을 조정한다.

최소 key 개수 여부에 따라 삭제한 K를 부모 노드로 대체하거나 병합하고, 왼쪽 형제 노드의 가장 큰 값 또는 오른쪽 형제 노드의 가장 작은 값을 부모 노드로 대체하는 등의 과정이 일어난다.



글, 그림 참조:
https://velog.io/@emplam27/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EA%B7%B8%EB%A6%BC%EC%9C%BC%EB%A1%9C-%EC%95%BC%EC%95%84%EB%B3%B4%EB%8A%94-B-Tree

Index

인덱스(Index)는 **데이터베이스의 테이블에 대한 검색 속도를 향상시켜주는 자료구조**이다.
테이블의 특정 컬럼(Column)에 인덱스를 생성하면, 해당 컬럼의 데이터를 정렬한 후 별도의 메모리 공간에 데이터의 물리적 주소와 함께 저장된다.**컬럼의 값과 물리적 주소**를 **(key, value)의 한 쌍으로 저장**한다.

인덱스는 책에서의 **목차** 혹은 **색인**이라 생각할 수 있다.
: 데이터=책의 내용, 인덱스=책의 목차, 물리적 주소=책의 페이지 번호.
인덱스를 이용하면 테이블에서 원하는 데이터를 빠르게 찾을 수 있다.
정보 검색에 있어 성능을 최적화시켜줄 수 있는 유용한 자료구조이다.

인덱스는 여러 자료구조를 이용해서 구현할 수 있는데, 대표적으로 **해시 테이블(Hash Table)**과 **B+Tree**가 있다.
B+Tree는 모든 데이터를 한 번 순회하는 데에 트리의 모든 노드를 방문해야 하는 단점을 보이는 B-Tree를 개선시킨 자료구조이다.

인덱스는 데이터베이스 메모리를 사용하여 테이블 형태로 저장되므로 개수와 저장공간은 비례한다.
따라서, 조회 시 자주 사용하고 고유한 값 위주로 인덱스를 설정하는 것이 좋다.

글, 그림 참조:
https://rebro.kr/167

Index

- **장점**:
- 테이블을 검색하는 속도와 성능이 향상된다.
- 시스템의 전반적인 부하를 줄일 수 있다.
- Where문, ORDER BY문, MIN/MAX 등에 이미 정렬이 되어 있어 빠르게 수행할 수 있다.

- **단점**:
- 인덱스를 관리하기 위한 추가 작업이 필요하다.
  (인덱스는 항상 정렬된 상태로 유지되기 때문에 인덱스가 적용된 컬럼에 삽입, 삭제, 수정 작업을 수행하면 추가 작업이 필요하다.)
- 추가 저장 공간이 필요하다.
- 잘 못 사용하는 경우 오히려 검색 성능이 저하된다.

글, 그림 참조:
https://rebro.kr/167

Index

- 인덱스를 **사용하면 좋은 경우**:

데이터의 range가 넓고 중복이 적을수록, 조회가 많거나 정렬된 상태가 유용한 컬럼에 사용하는 것이 좋다.

- **규모가 큰 테이블**
- **삽입(INSERT), 수정(UPDATE), 삭제(DELETE) 작업이 자주 발생하지 않는 컬럼**
- **WHERE나 ORDER BY, JOIN 등이 자주 사용되는 컬럼**
- **데이터의 중복도가 낮은 컬럼**

- 인덱스를 **사용하면 안 좋은 경우**:

- **데이터 range가 적은 컬럼**
  (나이나 성별 같은 경우,인덱스를 읽고 나서 다시 많은 데이터를 조회해야 하기 때문에 비효율적)
- **수정(UPDATE) 작업이 자주 발생하는 컬럼**
  (기존의 인덱스를 사용하지 않음으로 처리하고 갱신된 데이터에 대한 인덱스를 추가함으로, 실제 데이터에 비해 인덱스가 과도하게 커지고, 추가 저장 공간이 많이 필요하게 된다)

- 적용하면 **좋은 사례** : **배달의 민족** (사용자가 지역별 배달 음식점을 조회하고 검색하는 기능이 서비스의 주요 기능)
- 적용하면 **안 되는 사례** : **페이스북** (소셜 미디어는 사용자들이 새로운 게시글들을 끊임없이 생성한다)

글, 그림 참조:
https://rebro.kr/167

감사합니다