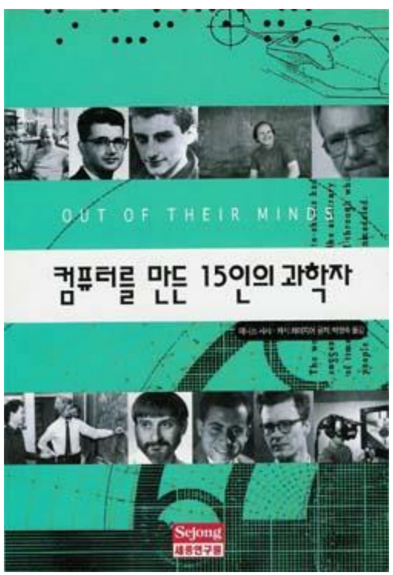
06장. 검색 트리

Youn-Hee Han
LINK@KOREATECH
http://link.koreatech.ac.kr

나는 좀 더 응용력 있는 유형의 수학이라는 이유 때문에
컴퓨터 과학을 하고 싶었다.

-로버트 타잔

이 책에서 다루고 있는 15인의 과학자는 다음과 같다.



1. 존 배커스 (포트란, FP)
2. 존 매커시 (LISP)
3. 앨런 C. 케이 (스몰토크, 객체지향)
4. 에드거 W. 다익스트라 (최단거리 알고리즘)
5. 마이클 O. 라빈 (무작위 알고리즘, 암호화)
6. 도널드 E. 크누스 (컴파일러, TEX, The Art of Programming)
7. 로버트 E. 타잔 (깊이 우선 탐색, 강한 연결)
8. 레즐리 램포트 (분산형 시스템)
9. 스티븐 쿡 & 레오나드 레빈 (NP-완전 문제)
10. 프레드릭 P. 브룩스 2세 (IBM 표준화)
11. 버튼 J. 스미스 (병렬 컴퓨터)
12. W. 대니얼 힐리스 (연결형 기계)
13. 에드워드 A. 파이겐바움 (전문가 시스템)
14. 더글러스 B. 레넛 (AM, 사이크cyc)

학습 목표

- ◈ 검색에서 레코드와 키의 역할을 구분한다.
- ◈ 이진 검색 트리에서 검색 · 삽입 · 삭제 작업의 원리를 이해한다.
- ◈ 이진 검색 트리의 균형이 작업의 효율성에 미치는 영향을 이해하고 ,
- ◈ 레드 블랙 트리의 삽입 · 삭제 작업의 원리를 이해한다.
- ◈ B-트리의 도입 동기를 이해하고 검색 · 삽입 · 삭제 작업의 원리를 이해한다.
- ◈ 검색 트리 관련 작업의 점근적 수행 시간을 이해한다.
- ◈ 일차원 검색의 기본 원리와 다차원 검색의 연관성을 이해한다.

01. 레코드, 키의 정의 및 검색 트리

레코드, 필드, 키

	학 번	소 속	성 명	본 적	전화번호
레코드1:	1001	전산	홍길동	서울	531-2351
레코드2:	1002	전산	이순신	대구	652-3323
		⋮	⋮	⋮	⋮
레코드n:	XXXX	전산	곽재우	대전	471-8203
	필드	필드	필드	필드	필드

◈ 레코드record

- 개체에 대해 수집된 모든 정보를 포함하고 있는 저장 단위
- 예: 사람의 레코드
  - 주민등록번호, 이름, 집주소, 집 전화번호, 직장 전화번호, 휴대폰 번호, 최종 학력, 연소득, 가족 상황 등의 정보 포함

◈ 필드field

- 레코드에서 각각의 정보를 나타내는 부분
- 예: 위 사람의 레코드에서 각각의 정보를 나타내는 부분

◈ 검색키search key 또는 키key

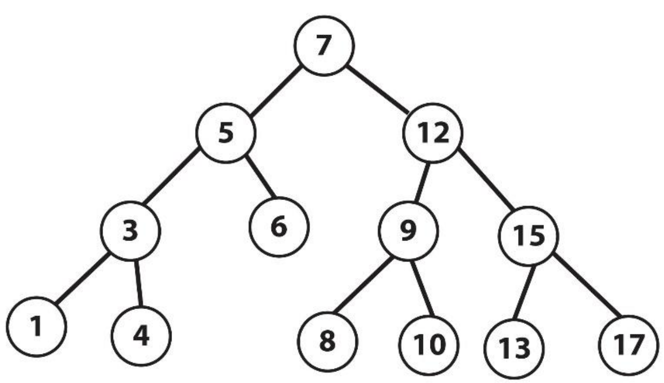
- 다른 레코드와 중복되지 않도록 **각 레코드를 대표하는 필드**
- 예: 주민등록번호

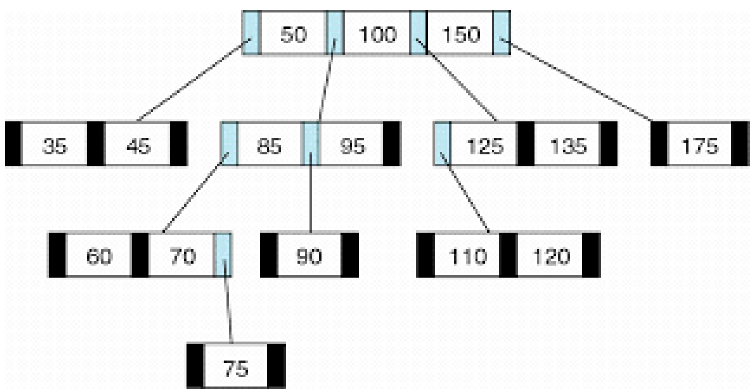
검색 트리

◈ **검색 트리**search tree

– 각 트리의 노드는 임의의 레코드와 대응
  - 일반적으로 각 노드는 대응되는 레코드로의 포인터 유지
– 각 트리의 노드는 해당 레코드의 검색키를 지님
– 각 노드의 검색키는 다음의 규칙을 만족

각각의 노드의 키 값을 K라고 할 때, 이 노드의 왼쪽 서브 트리내 존재하는 노드들의 키 값이 K 보다 작고, 그 노드의 오른쪽 서브 트리내 존재하는 노드들의 키 값이 K 보다 크다.

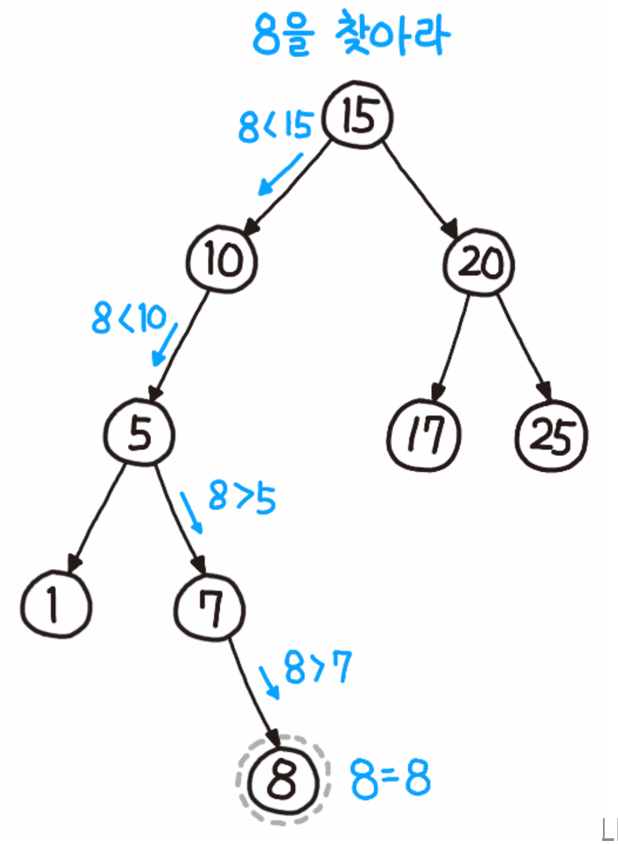


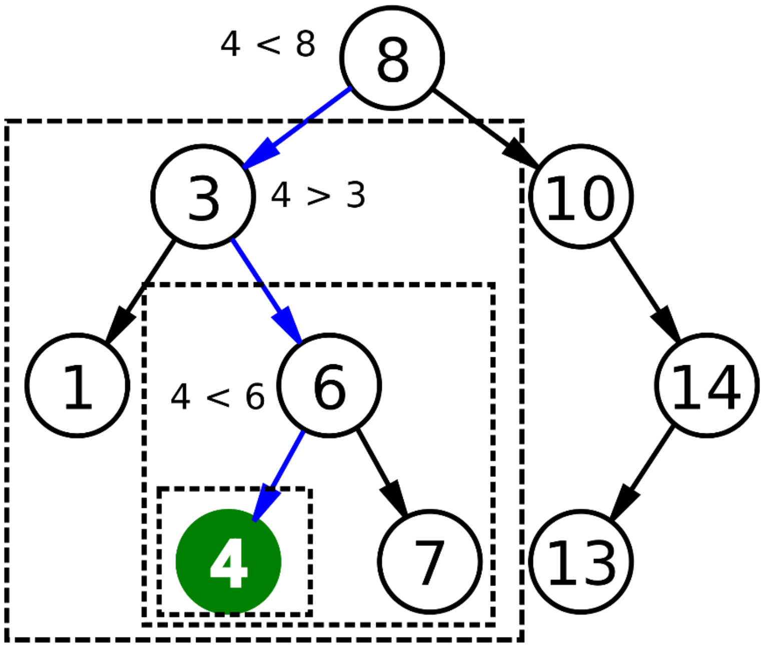


검색 트리

◈ **검색 트리**search tree

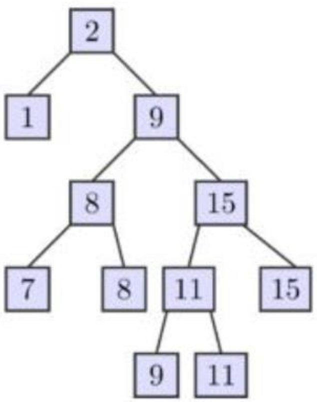
– **검색 트리의 장점:**
검색하고자 하는 레코드의 저장 위치를 매우 효율적으로 찾을 수 있음

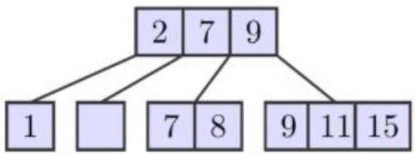


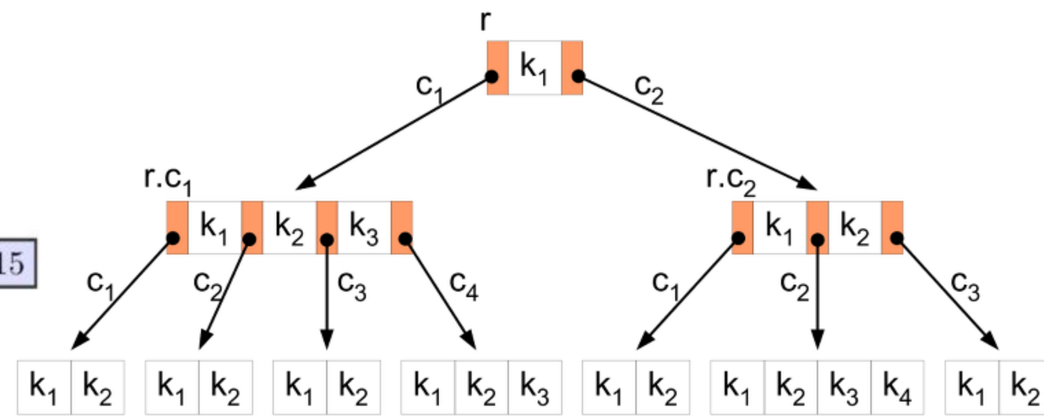


검색 트리의 종류

◈ 이진 검색 트리 vs. k진(다진) 검색 트리







◈ 내부 검색 트리 vs. 외부 검색 트리

- 내부 검색 트리
  - 메인 메모리에 모든 노드(모든 키)를 저장
- 외부 검색 트리
  - 외부 디스크 공간에 모든 노드 (모든 키)를 저장
  - 결국 디스크의 접근 시간이 검색의 효율을 좌우함

검색 트리의 종류

◈ 일차원 검색 트리 vs. 다차원 검색 트리

- 일차원 검색 트리
  - 검색키를 구성하는 필드가 1개
  - 예: 이진 검색 트리, AVL-트리, 레드 블랙 트리, B-트리
- 다차원 검색 트리
  - 검색키를 구성하는 필드가 2개 이상
  - 예: KD-트리, KDB-트리, R-트리

02. 이진 검색 트리

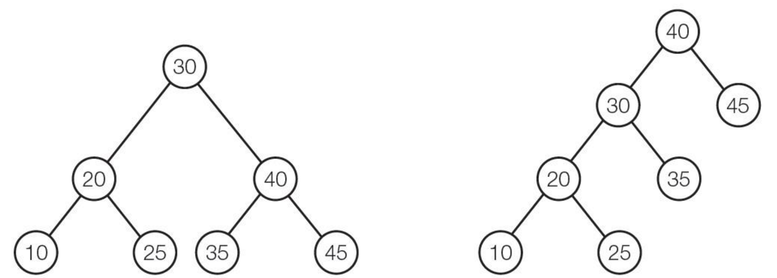
이진 검색 트리

◈ 이진 검색 트리의 특징

- 이진 검색 트리의 각 노드는 키 값을 하나씩 갖는다.
  - 각 노드의 키 값은 모두 달라야 한다.

- 최상위 레벨에 루트 노드가 있고, 각 노드는 최대 두 개의 자식을 갖는다.

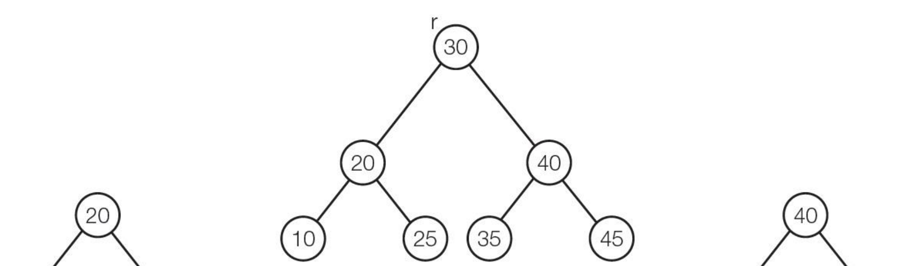
- 임의의 노드의 키 값은 자신의 왼쪽 자식 노드의 키 값보다 크고, 오른쪽 자식의 키 값보다 작다.



(a) 키 값이 30인 노드가 루트

(b) 키 값이 40인 노드가 루트

그림 6-1 이진 검색 트리의 예



(a) 이진 검색 트리의 한 예

(b) 노드 r의 왼쪽 서브 트리

(c) 노드 r의 오른쪽 서브 트리

그림 6-2 서브 트리의 예

이진 검색 트리에서 검색

◈ 이진 검색 트리에서 키가 x인 노드 검색

- 트리에 키가 x인 노드가 존재하면 해당 노드 리턴
- 존재하지 않으면 NIL을 리턴

알고리즘 6–1	이진 검색 트리에서 검색

```
treeSearch(t, x)
▷ t : 트리의 루트 노드
▷ x : 검색하고자 하는 키
{
  ❶ if (t = NIL or key[t] = x) then return t;
     if (x < key[t])
        ❷ then return treeSearch(left[t], x);
        ❸ else return treeSearch(right[t], x);
}
```

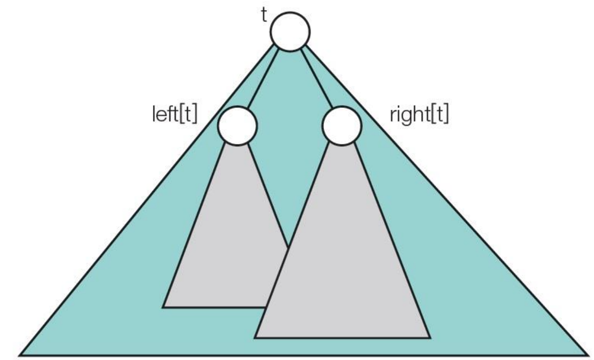
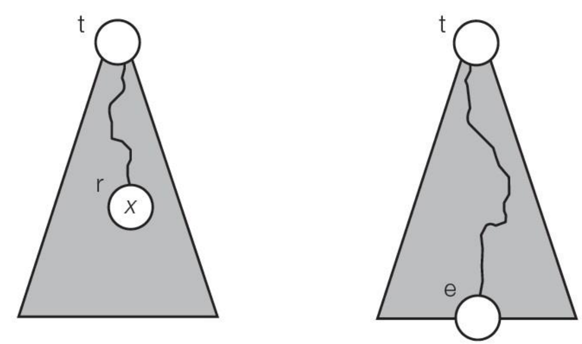


그림 6–3 이진 트리 검색에서 재귀적 관점



(a) 성공적인 검색　　(b) 실패한 검색

그림 6–4 성공적인 검색과 실패하는 검색

- 검색에 대한 점근적 분석
  - 트리의 높이에 따라 $O(logn)$~$O(n)$
  - 추후 더 자세히 살펴봅시다.

이진 검색 트리에서 삽입

◈ 이진 검색 트리에서 키가 x인 노드 삽입

알고리즘 6-3 이진 검색 트리에서 삽입

```
treeInsert(t, x)
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키
▷ 작업 완료 후 루트 노드의 포인터를 리턴한다.
{
    if (t = NIL) then {
        key[r] ← x; left[r] ← NIL; right[r] ← NIL;      ▷ r : 새 노드
        return r;
    }
    if (x < key[t])
      ❶ then {left[t] ← treeInsert(left[t], x); return t;}
      ❷ else {right[t] ← treeInsert(right[t], x); return t;}
}
```

이진 검색 트리에서 삽입

◈ 이진 검색 트리에서 키가 x인 노드 삽입

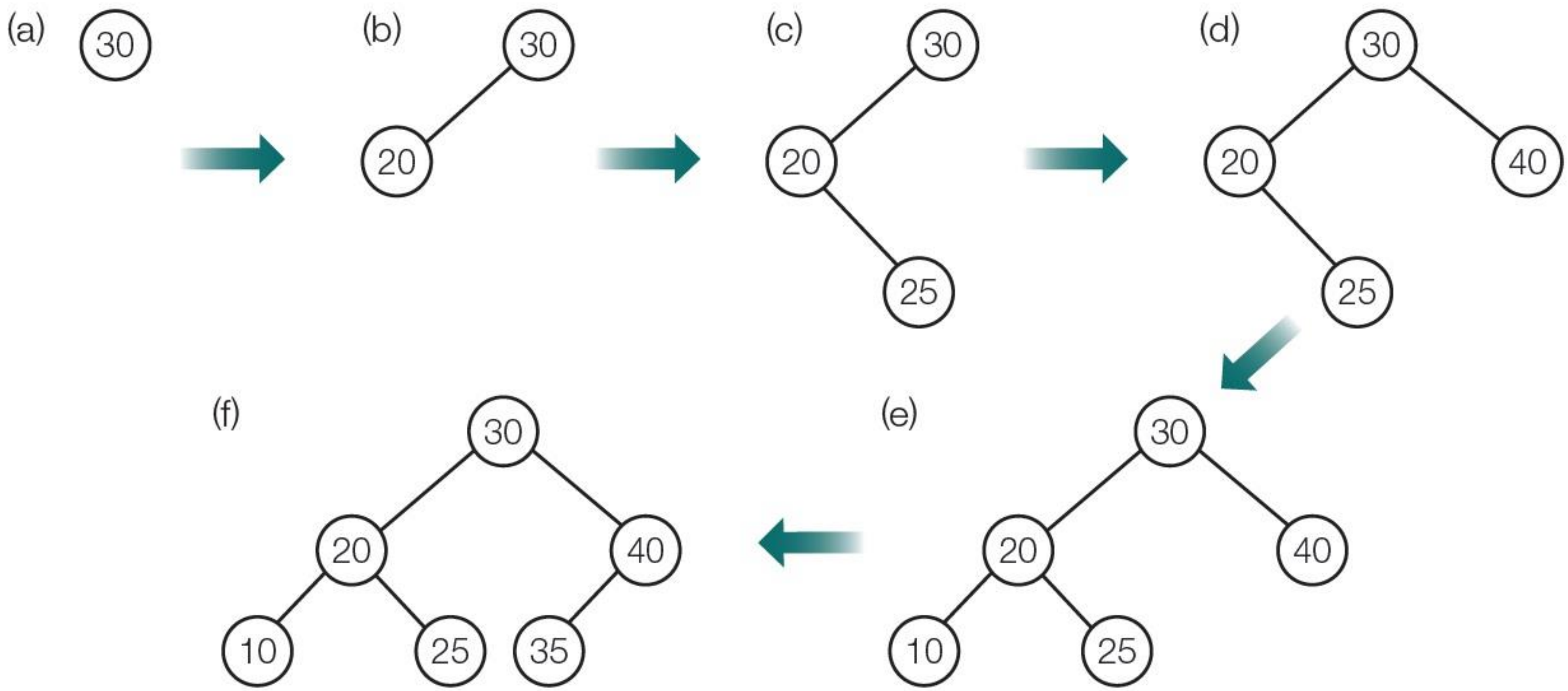


그림 6-5 이진 검색 트리의 삽입 과정을 보여주는 예

이진 검색 트리에서 삽입

◈ 이진 검색 트리에서 키가 x인 노드 삽입

알고리즘 6-3 이진 검색 트리에서 삽입

```
treeInsert(t, x)
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키
▷ 작업 완료 후 루트 노드의 포인터를 리턴한다.
{
    if (t = NIL) then {
        key[r] ← x; left[r] ← NIL; right[r] ← NIL;    ▷ r : 새 노드
        return r;
    }
    if (x < key[t])
     ❶ then {left[t] ← treeInsert(left[t], x); return t;}
     ❷ else {right[t] ← treeInsert(right[t], x); return t;}
}
```

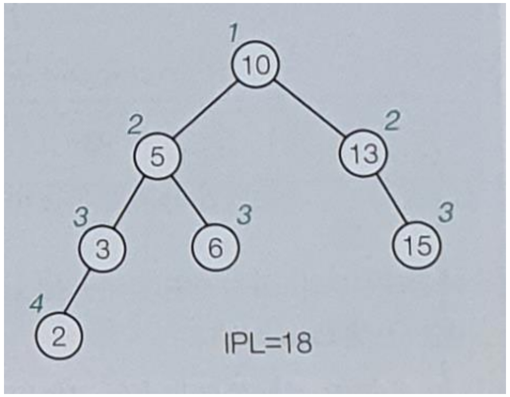
알고리즘 6-4 이진 검색 트리에서 삽입(비재귀 버전)

```
treeInsert(t, x)
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키
{
    key[r] ← x; left[r] ← NIL; right[r] ← NIL;    ▷ r : 새 노드
    if (t = NIL) then root ← r;
    else {
        p ← NIL; tmp ← t;
        while (tmp ≠ NIL) {
            p ← tmp;
            if (x < key[tmp]) then tmp ← left[tmp];
                              else tmp ← right[tmp];
        }
        if (x < key[p]) then left[p] ← r;
                        else right[p] ← r;
    }
}
```

이진 검색 트리에서 삽입

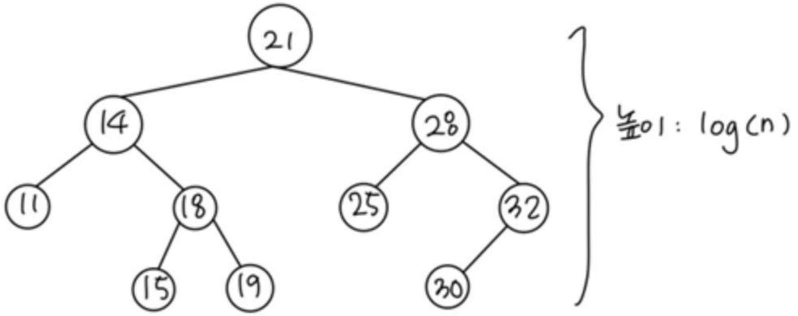
◈ IPL: Internal Path Length (내부 경로 길이)

- 오른쪽 이진 검색 트리의 IPL
  = 1 + 2 + 2 + 3 + 3 + 3 + 4 = 18



◈ [정리 6-1]

- 키의 총 수가 n개인 모든 이진 검색 트리의 평균 IPL은 $O(nlogn)$ 이다.
  - [증명 생략] – 교재 P.164 참조



◈ 이상적인 트리의 높이

- 평균 키 검색 수
- IPL / 키의 개수 → $O(nlogn)/n$

◈ 평균의 경우 삽입 알고리즘 점근적 복잡도: $\Theta(logn)$

이진 검색 트리에서 삽입

◈ **최악의 경우 삽입 알고리즘 점근적 복잡도: $\Theta(n)$**
  - 10, 20, 25, 30, 40, 45 의 순서로 원소가 삽입될 경우에 만들어지는 이진 검색 트리

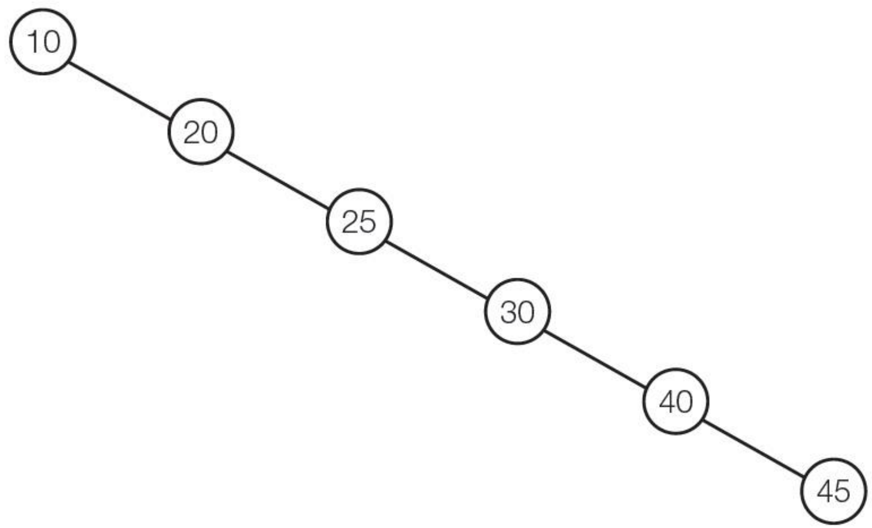


그림 6-6 균형이 맞지 않는 이진 검색 트리의 예

이진 검색 트리에서 삭제

- 이진 검색 트리에서 노드 r을 삭제할 때 고려해야 할 서로 다른 경우
  - Case I: 노드 r이 리프 노드인 경우
  - Case II: 노드 r의 자식 노드가 하나인 경우
  - Case III: 노드 r의 자식 노드가 두 개인 경우

알고리즘 6-5 이진 검색 트리에서 삭제 스케치

```
sketchDelete(t, r)
▷ t : 트리의 루트 노드
▷ r : 삭제하고자 하는 노드
{
    if (r이 리프 노드) then                          ▷ Case 1
        그냥 r을 버린다;
    else if (r의 자식이 하나만 있음) then            ▷ Case 2
        r의 부모가 r의 자식을 직접 가리키도록 한다;
    else {                                           ▷ Case 3
        r의 오른쪽 서브 트리의 최소 원소 노드 s를 삭제하고,
        s를 r 자리에 놓는다;
    }
}
```

이진 검색 트리에서 삭제

◈ 이진 검색 트리에서 노드 r을 삭제할 때 고려해야 할 서로 다른 경우

– Case I: 노드 r이 리프 노드인 경우

• 그냥 노드 r을 버린다.

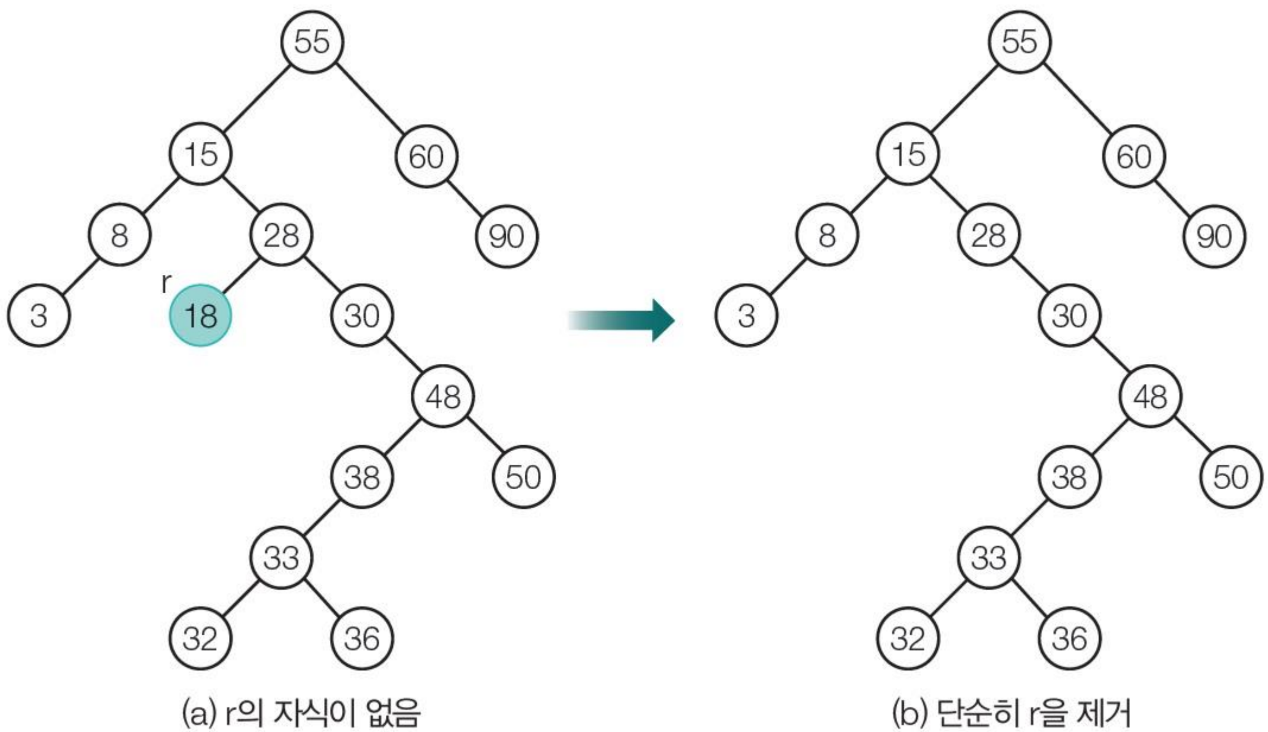


그림 6-7 이진 검색 트리에서 삭제 Case 1

이진 검색 트리에서 삭제

◈ 이진 검색 트리에서 노드 r을 삭제할 때 고려해야 할 서로 다른 경우

- Case II: 노드 r의 자식 노드가 하나인 경우
  - 노드 r의 부모가 노드 r의 자식을 직접 가리키도록 한다.

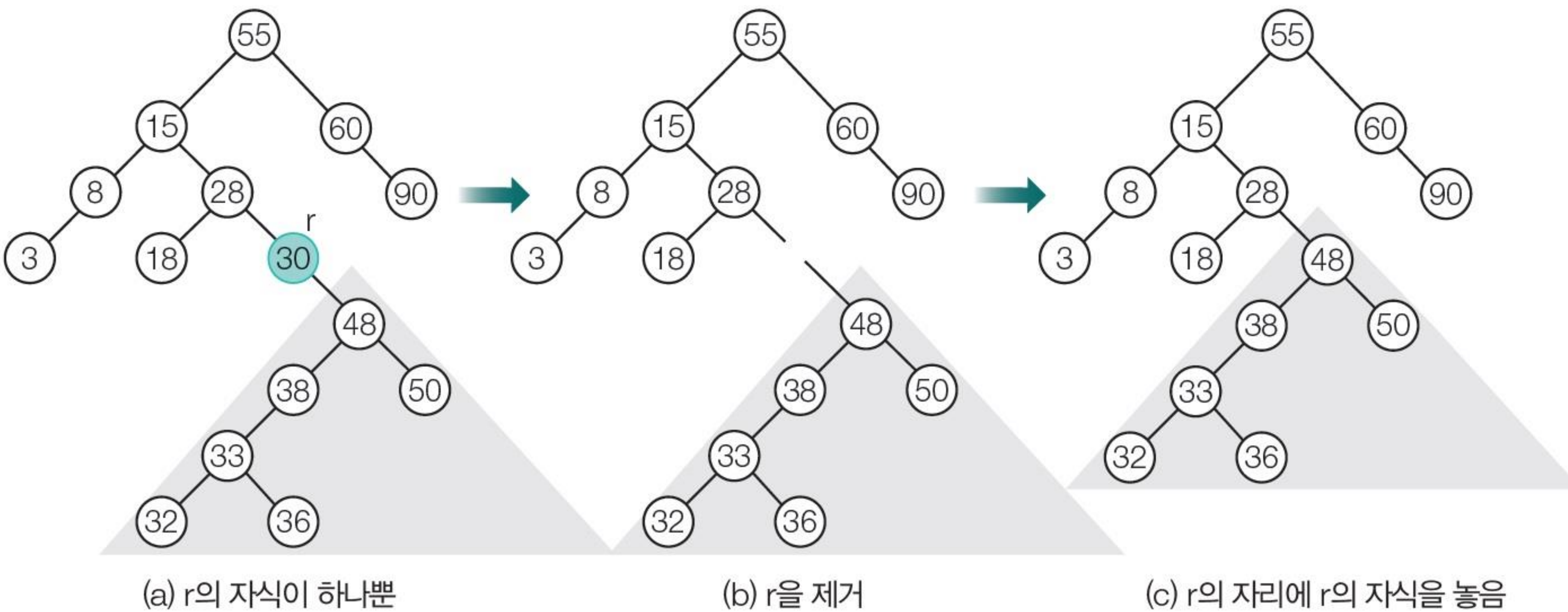
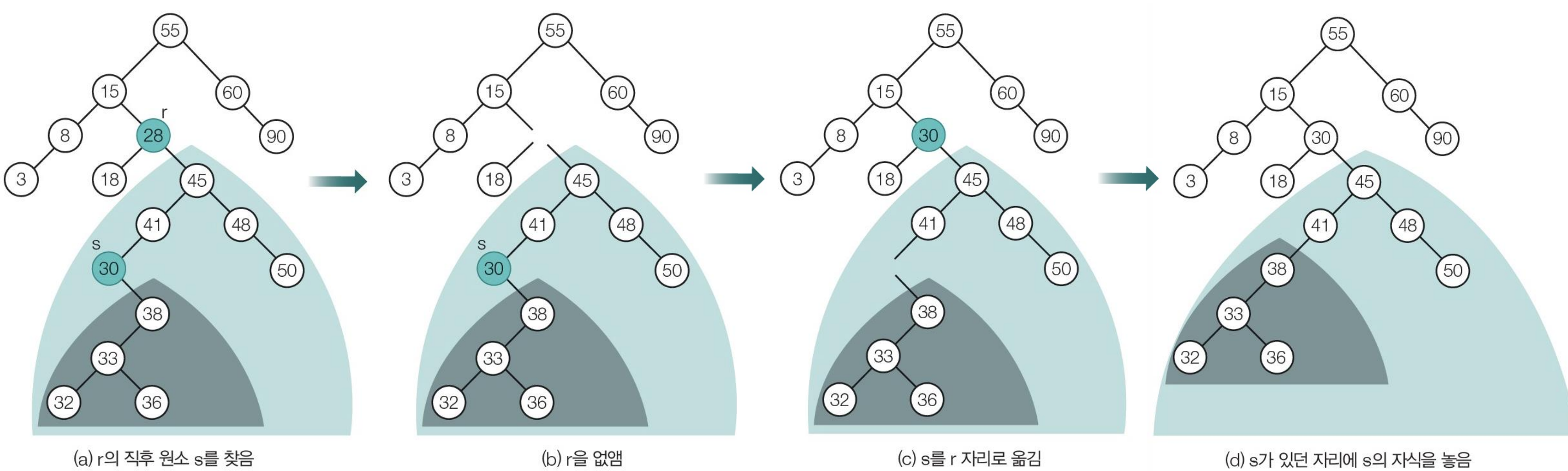


그림 6-8 이진 검색 트리에서 삭제 Case 2

이진 검색 트리에서 삭제

◈ 이진 검색 트리에서 노드 r을 삭제할 때 고려해야 할 서로 다른 경우

– Case III: 노드 r의 자식 노드가 두 개인 경우
  - 노드 r의 오른쪽 서브 트리의 최소 원소 노드 s를 삭제하고
  - 노드 s를 노드 r의 자리에 놓는다.

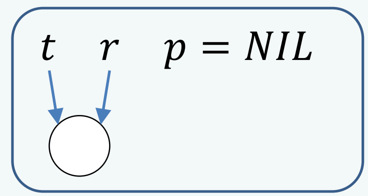


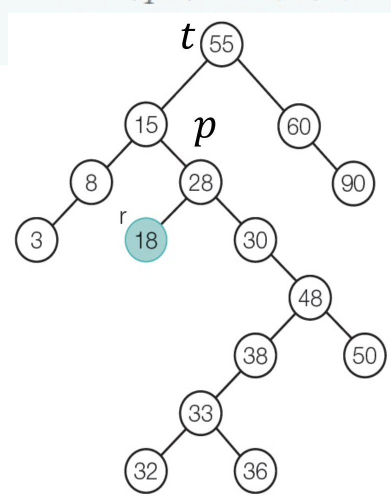
이진 검색 트리에서 삭제

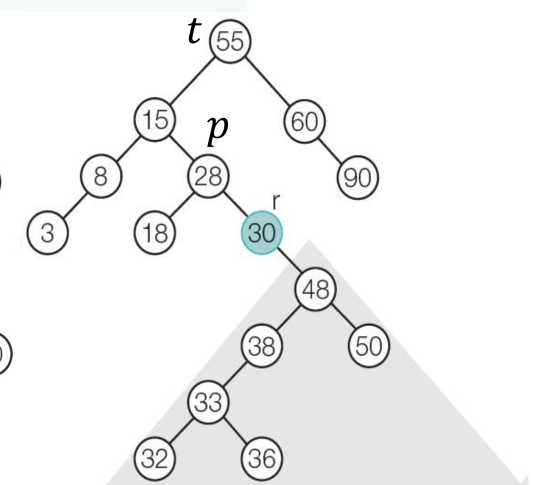
◈ 이진 검색 트리에서 노드 r을 삭제

알고리즘 6-6 | **이진 검색 트리에서 삭제**

treeDelete(t, r, p)
▷ t : 트리의 루트 노드
▷ r : 삭제하고자 하는 노드, p : r의 부모 노드
{
  ▷ r이 루트 노드인 경우
  if ($r = t$) then *root* ← deleteNode(t);
  ▷ r이 루트 노드가 아닌 경우
  else if ($r = left[p]$)
    then $left[p]$ ← deleteNode(r); ▷ r이 p의 왼쪽 자식
    else $right[p]$ ← deleteNode(r); ▷ r이 p의 오른쪽 자식
}



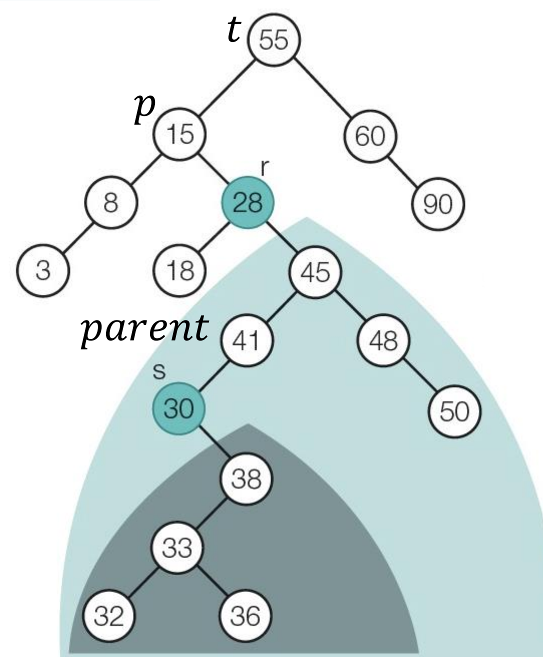




이진 검색 트리에서 삭제

◈ 이진 검색 트리에서 노드 r을 삭제

```
deleteNode(r)
{
    if (left[r] = right[r] = NIL) then return NIL;                        ▷ Case 1
    else if (left[r] = NIL and right[r] ≠ NIL) then return right[r];      ▷ Case 2-1
    else if (left[r] ≠ NIL and right[r] = NIL) then return left[r];       ▷ Case 2-2
    else {                                                                ▷ Case 3
            s ← right[r];
            while(left[s] ≠ NIL)
                    {parent ← s; s ← left[s];}
            key[r] ← key[s];
            if (s = right[r]) then right[r] ← right[s];
                               else left[parent] ← right[s];
            return r;
    }
}
```



이진 검색 트리에서 삭제

◈ 이진 검색 트리에서 노드 삭제의 점근적 분석

```
deleteNode(r)
{
    if (left[r] = right[r] = NIL) then return NIL;                          ▷ Case 1
    else if (left[r] = NIL and right[r] ≠ NIL) then return right[r];       ▷ Case 2-1
    else if (left[r] ≠ NIL and right[r] = NIL) then return left[r];        ▷ Case 2-2
    else {                                                                  ▷ Case 3
            s ← right[r];
            while (left[s] ≠ NIL)
                    {parent ← s; s ← left[s];}
            key[r] ← key[s];
            if (s = right[r]) then right[r] ← right[s];
                            else left[parent] ← right[s];
            return r;
    }
}
```

Case 1: $\Theta(1)$

Case 2: $\Theta(1)$

Case 3:
최악의 경우 트리 전체 높이에 대해 while루프 수행
따라서, 트리의 높이에 따라
$O(logn)$~$O(n)$

03. 레드 블랙 트리

균형 잡힌 이진 검색 트리의 필요성

◈ 이진 검색 트리의 단점

- 이진 검색 트리에서 저장, 검색, 삭제에 대한 점근적 복잡도
  - 평균의 경우: $O(logn)$
  - 최악의 경우: $O(n)$
- 즉, 트리의 균형이 깨지면 효율이 높지 않다.
- 그렇다면, 트리의 균형을 유지하는 형태로 검색 트리를 구성하면 어떨까?

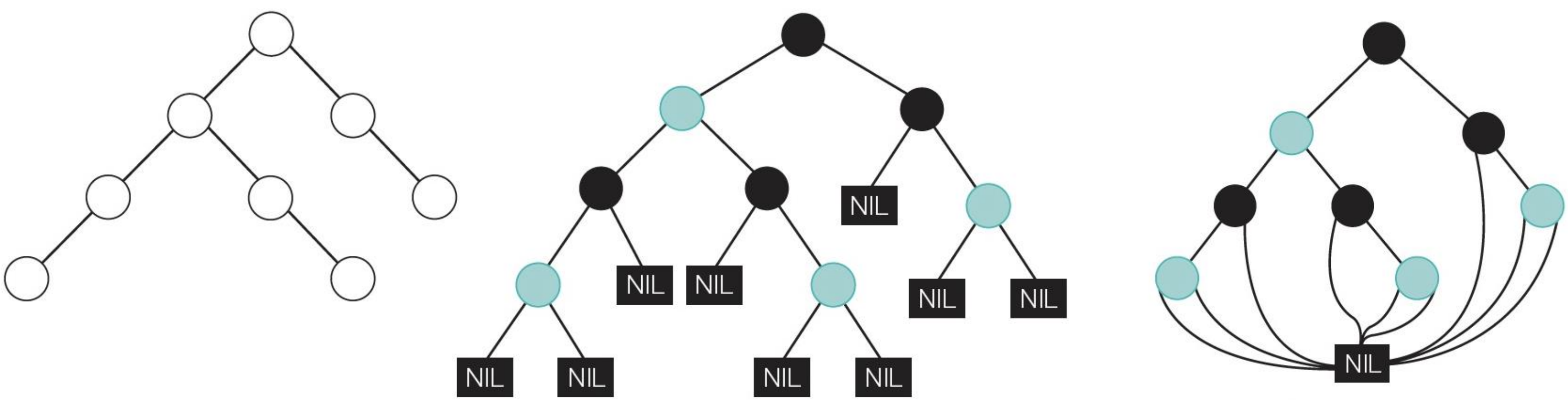
◈ 대표적인 균형 잡힌 이진 검색 트리 (self-balancing binary search tree)

- 레드 블랙 트리
- AVL 트리

레드 블랙 트리

◈ 레드 블랙 트리의 특징

- 이진 검색 트리의 임의의 노드에 자식 포인터 중 NIL이 있다면, 레드 블랙 트리에서는 해당 NIL 노드를 별도로 만들고 이 NIL 노드는 레드 블랙 트리의 리프 노드가 된다.
- 각 노드에 대해 다음 규칙으로 블랙 또는 레드의 색을 칠한다.
    - 루트는 블랙이다
    - 모든 리프(NIL노드)는 블랙이다
    - 임의의 노드가 레드이면 그 노드의 자식은 반드시 블랙이다
    - 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다



(a) 이진 검색 트리의 예 (b) (a)를 레드 블랙 트리로 만든 예 (c) 실제 구현할 때 NIL 노드 처리 방법

레드 블랙 트리의 연산

- ◈ 레드 블랙 트리에서 임의의 키 검색
  - 이진 검색 트리에서의 검색과 완전 동일
- ◈ 레드 블랙 트리에서 임의의 노드 삽입과 삭제
  - 이진 검색 트리에서의 삽입과 삭제와 비슷
  - 하지만, 노드 삽입 및 노드 삭제 후 레드 블랙 트리의 특징을 위반하는 경우에 대해 적절한 작업을 수행하여 레드 블랙 트리의 특징을 만족하도록 바로잡아 주어야 하는 추가 연산 필요

레드 블랙 트리 복잡도 분석

◈ [정리 6-2]

- 키의 총 수가 n개인 모든 레드 블랙 트리의 최대 트리의 깊이는 $O(logn)$ 이다.
  - [증명 생략] – 교재 P.180 참조

◈ 레드 블랙 트리의 장점

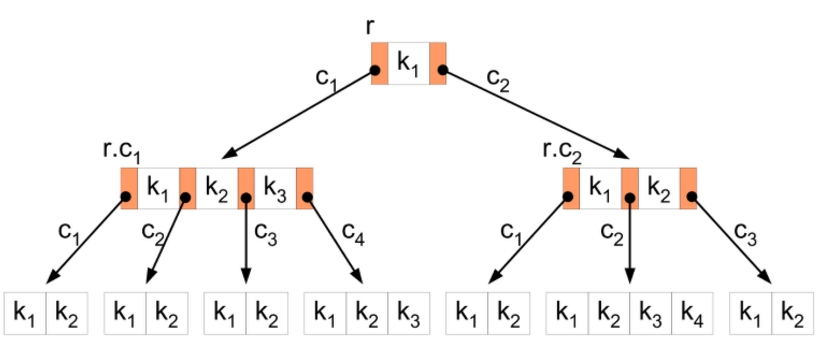
- 래드 블랙 트리에서 저장, 검색, 삭제에 대한 점근적 복잡도
  - 최악의 경우: $O(logn)$

04. B-트리

외부 검색 트리

◈ 외부 검색 트리
- 외부 디스크 공간에 모든 노드(모든 키)를 저장

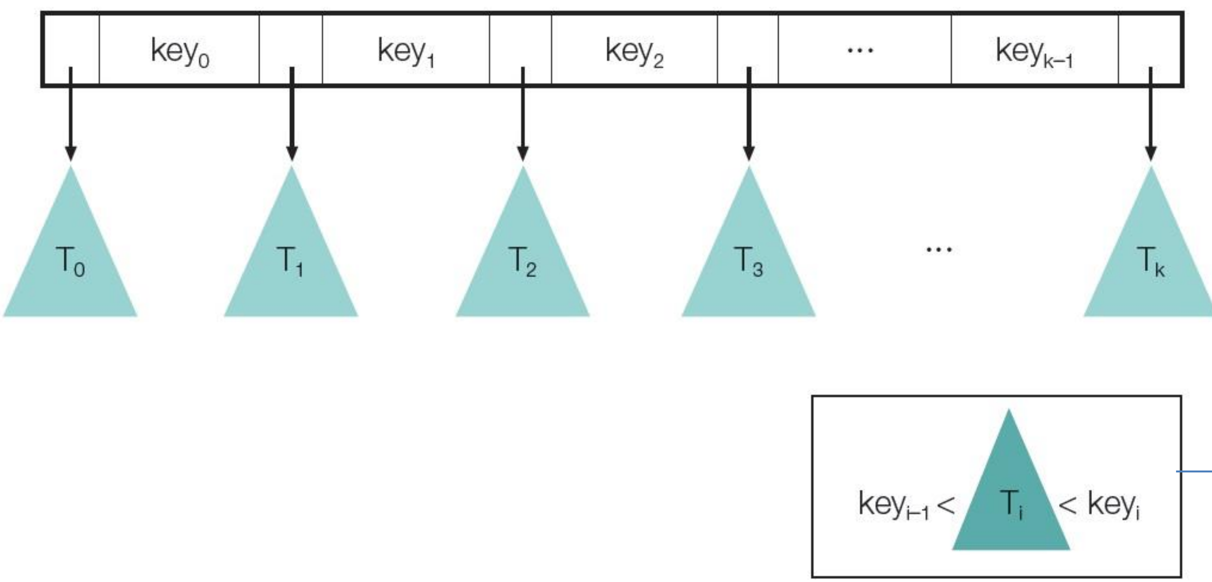
◈ 외부 검색 트리에서 고려해야 할 점
- 디스크의 접근 단위는 "블록(Block) or 페이지(Page)" ← 트리 노드
  - 사이즈 예: 8KB or 16KB
- 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다
  - 즉, 디스크 접근 횟수는 가급적 줄여야 함
- 외부 검색 트리에서 **한 노드의 분기 수(자식 수)를 늘려서 트리의 전체 높이를 최소화하는 것이 유리** → 다진 검색 트리
- 대표적인 다진 외부 검색 트리
  - **B-트리**



B-트리

◈ B-트리

- k진(다진) 검색 트리
- 트리의 균형을 유지하도록 하여 최악의 경우 디스크 접근 횟수를 줄인 것
  - 예를 들어, 10억개의 키
    - ➢ 이진 검색 트리로 자료를 구축할 때 트리의 높이: 약 30
    - ➢ B-트리 ($k = 256$)로 자료를 구축할 때 트리의 높이: 약 5



키의 개수: k

자식의 개수: $k + 1$

각 자식에 대응되는 서브 트리: $T_0, T_1, \ldots, T_k$

검색 트리(Search Tree)의 특징 보유

그림 6-17 다진 검색 트리에 키가 k개 있는 예

B-트리

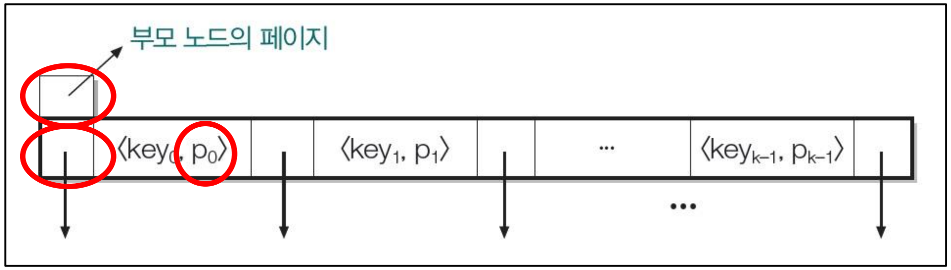
◈ B-트리

- B-트리는 균형 잡힌 다진 검색 트리로 다음의 성질을 만족한다.
  - 루트를 제외한 모든 노드는 $\left\lfloor \frac{k}{2} \right\rfloor$ 부터 k 개의 키를 갖는다.
  - 모든 리프 노드는 같은 깊이를 가진다

- 즉, B-트리는 균형을 맞추기 위해 각 노드가 채울 수 있는 최대 허용량의 절반 이상의 키는 채우고, 이에 따른 분기의 수를 맞추는 검색 트리

B-트리

◈ B-트리의 노드 구조



- ⟨Key, p 포인트⟩
  - Key : 키 값
  - p 포인트: 페이지 번호
    - 키가 검색키와 일치했을 때 해당 키에 대응되는 레코드를 가지고 올 수 있는 페이지 번호

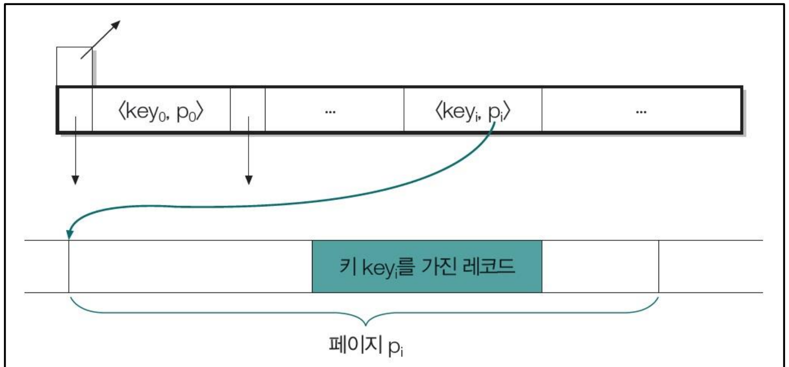


그림 6-18 레코드에 접근하는 절차를 보여주는 구조

- 부모 노드로의 포인트(페이지 번호)
- 자식 노드로의 포인트(페이지 번호)
- 모든 노드는 무조건 외부 디스크에 존재
- B-트리를 통해 먼저 트리 노드 페이지를 메모리로 가져오고, 해당 페이지에서 최종 레코드를 획득

B-트리

◈ B-트리의 노드 구조

- 한 노드(블록 또는 페이지)가 지닐 수 있는 최대 키 개수 산출
  - 블록(페이지) 크기 – 8,192바이트
  - 키의 크기 – 16바이트
  - 페이지 번호 크기 – 4 바이트

최대 341개의 키

$$4 + \left(4 + \underline{(16 + 4)}\right) * 341 + 4 = 8{,}192$$

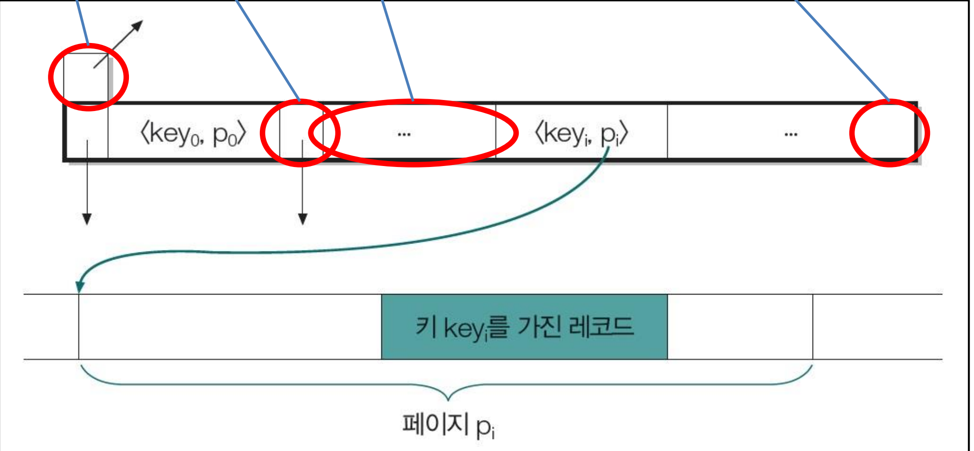


그림 6-18 레코드에 접근하는 절차를 보여주는 구조

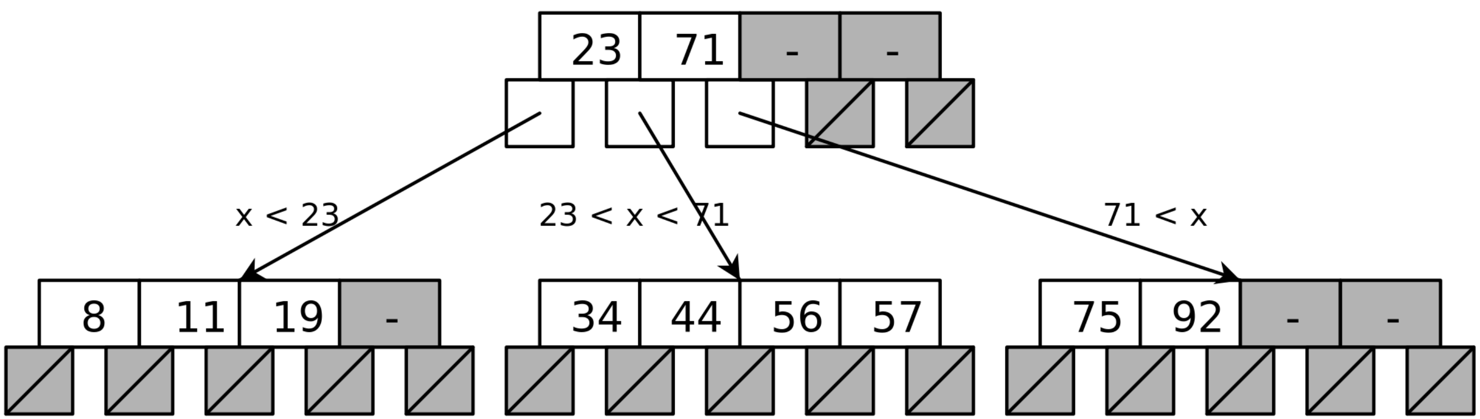
B-트리에서 검색

◈ B-트리에서 키 x에 대한 검색

- B-트리에는 최대 k개까지 키를 가질 수 있음
- 따라서, 최대 k개의 키 중 검색하고자 하는 키 x와 일치하는 지 확인 필요
  - 각 노드에서 반복 루프를 통한 다음 연산 수행

$key_{i-1} = x$ $\quad$ $key_{i-1} < x < key_i$ $\quad$ for $i = 1$ to $k - 1$

- 자식으로 분기하며 재귀 호출 필요



B-트리에서 삽입

◈B-트리에서 키 x에 대한 삽입 과정 의사코드

알고리즘 6-7 B-트리에서 삽입 스케치

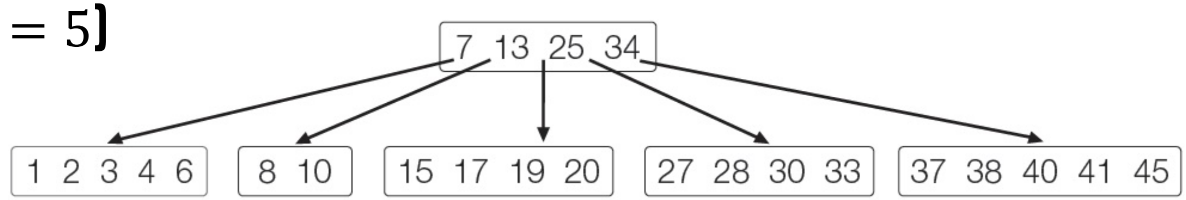
```
Sketch-BTreeInsert(t, x)
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키
{
    x를 삽입할 리프 노드 r을 찾는다;
    x를 r에 삽입한다;
    if (r에 오버플로 발생) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r의 형제 노드 중 공간 여유가 있는 노드가 있음) then {r의 남는 키를 넘긴다};
    else {
        r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
        if (부모 노드 p에 오버플로 발생) then clearOverflow(p);
    }
}
```

우선 실패하는 검색 시도

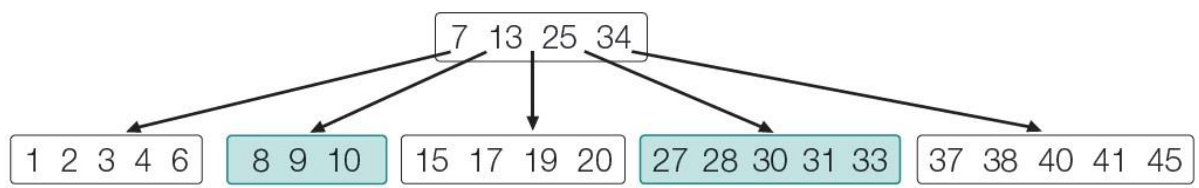
B-트리에서 삽입

◈ B-트리에서 삽입 과정 예 (1/4)

– 주어진 B-트리 ($k = 5$)



– 9, 31 삽입
- 우선 실패하는 검색 시도 및 삽입할 리프 노드 r을 찾음
- 해당 키를 리프 노드 r에 삽입



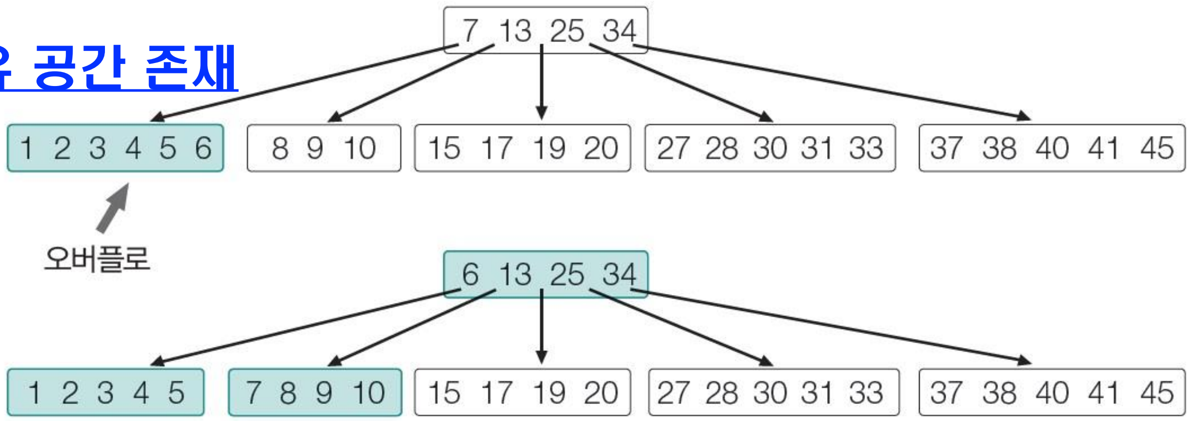
B-트리에서 삽입

◈ B-트리에서 삽입 과정 예 (2/4)

알고리즘 6-7 | B-트리에서 삽입 스케치

```
Sketch-BTreeInsert(t, x)
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키
{
    x를 삽입할 리프 노드 r을 찾는다;
    x를 r에 삽입한다;
    if (r에 오버플로 발생) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r의 형제 노드 중 공간 여유가 있는 노드가 있음) then {r의 남는 키를 넘긴다};
    else {
        r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
        if (부모 노드 p에 오버플로 발생) then clearOverflow(p);
    }
}
```

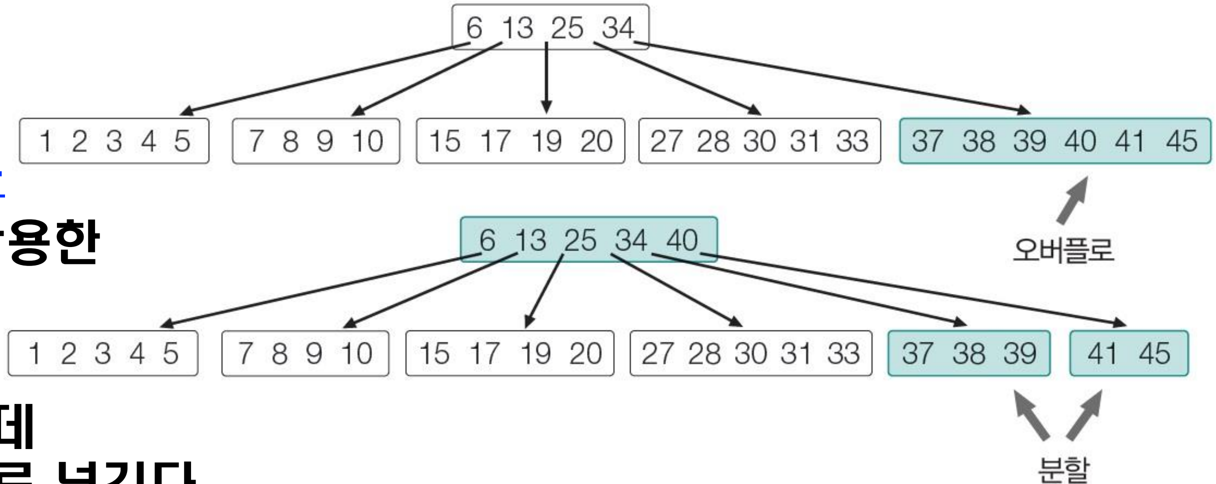
- 5 삽입
  - 삽입한 r에 오버플로 (Overflow) 발생
  - <u>형재 노드에 여유 공간 존재</u>
  - 부모 노드를 활용한 **재분배 (Redistribution)**
    - ➢ 남는 키를 넘김



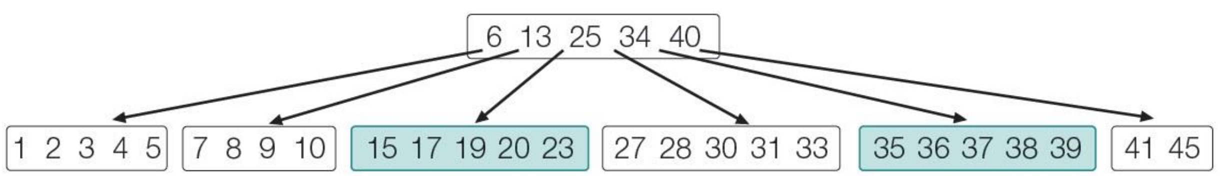
B-트리에서 삽입

◈ B-트리에서 삽입 과정 예 (3/4)

- 39 삽입
  - 오버플로 발생
  - 형재 노드에 여유 공간 없음
  - 부모 노드를 활용한 노드 분할
    - ➢ 둘로 분할하고 가운데 키를 부모로 넘긴다.



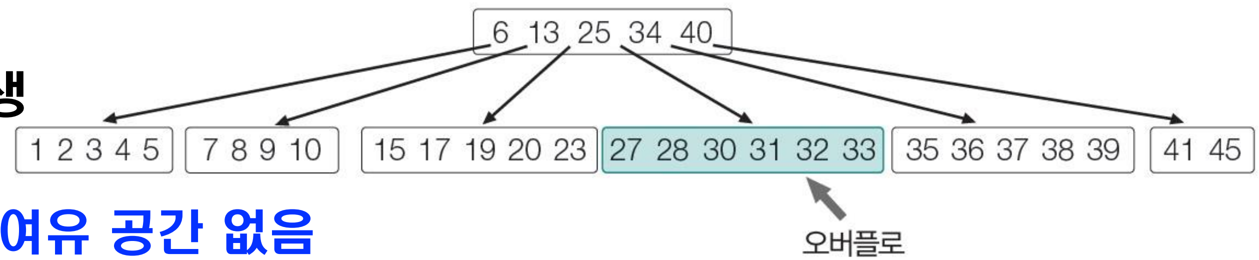
- 23, 35, 36 삽입



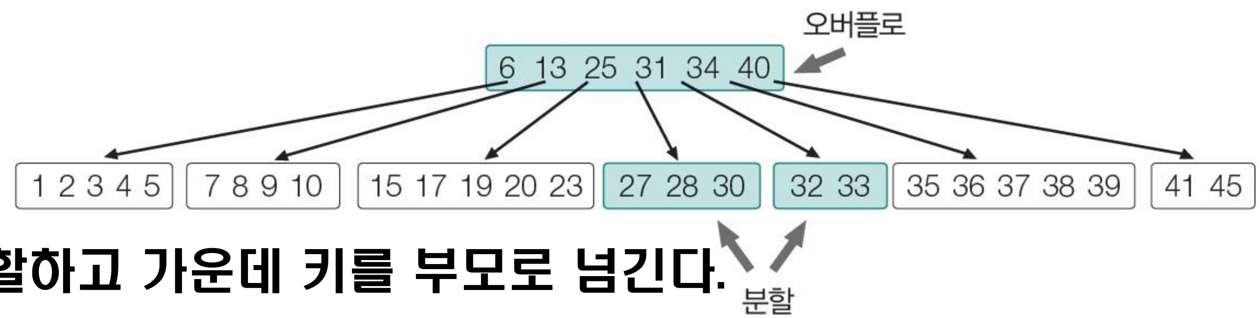
B-트리에서 삽입

◈ B-트리에서 삽입 과정 예 (4/4)

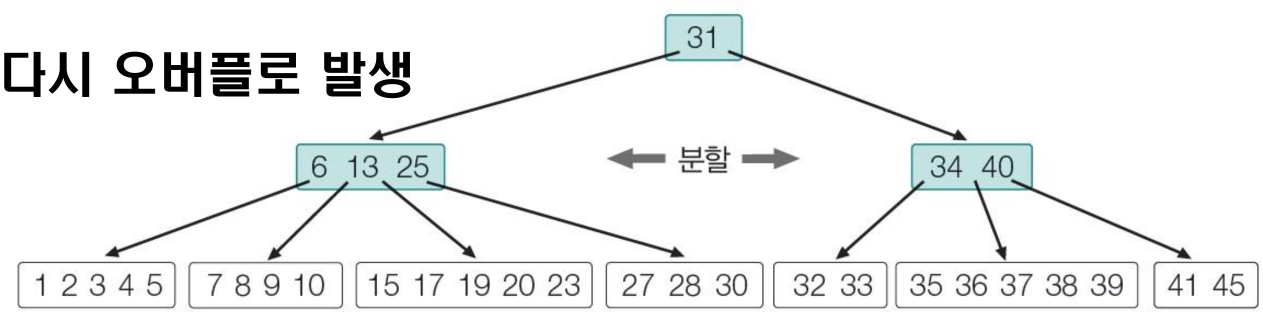
- 32 삽입
  - 오버플로 발생



  - 형재 노드에 여유 공간 없음
  - 부모 노드를 활용한 노드 분할
    - ➢ 둘로 분할하고 가운데 키를 부모로 넘긴다.



  - 부모 노드에 다시 오버플로 발생
  - 부모 노드를 둘로 나누고 가운데 키와 함께 새로운 루트 노드 생성



B-트리에서 삭제

◈ B-트리에서 삭제 과정 의사코드

본 알고리즘 내부에서 직접 검색 가능

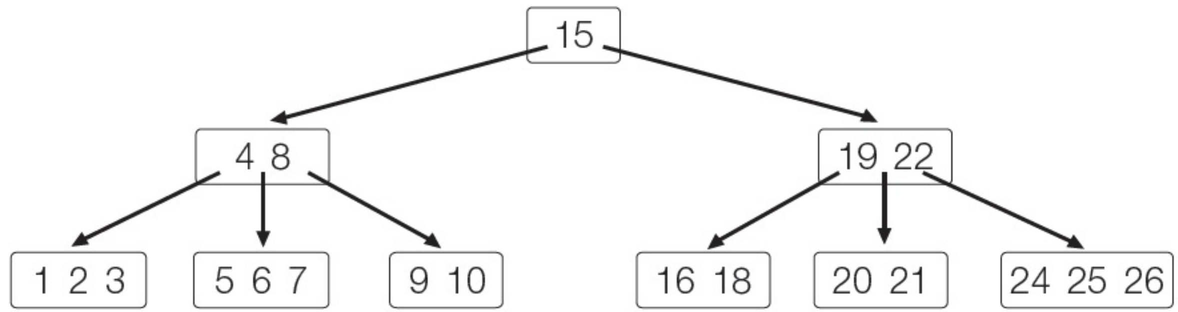
알고리즘 6-8	B-트리에서 삭제 스케치

```
Sketch-BTreeDelete(t, x, v)
▷ t : 트리의 루트 노드
▷ x : 삭제하고자 하는 키, v : x를 갖고 있는 노드
{
    if (v가 리프 노드 아님) then {
        x의 직후 원소 y를 가진 리프 노드를 찾는다;
        x와 y를 맞바꾼다;
    }
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 한다;
    if (r에서 언더플로 발생) then clearUnderflow(r);
}
clearUnderflow(r)
{
    if (r의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
    then {r이 키를 넘겨받는다;}
    else {
        r의 형제 노드와 r을 병합하고 부모 노드에서 키를 하나 받는다;
        if (부모 노드 p에 언더플로 발생) then clearUnderflow(p);
    }
}
```

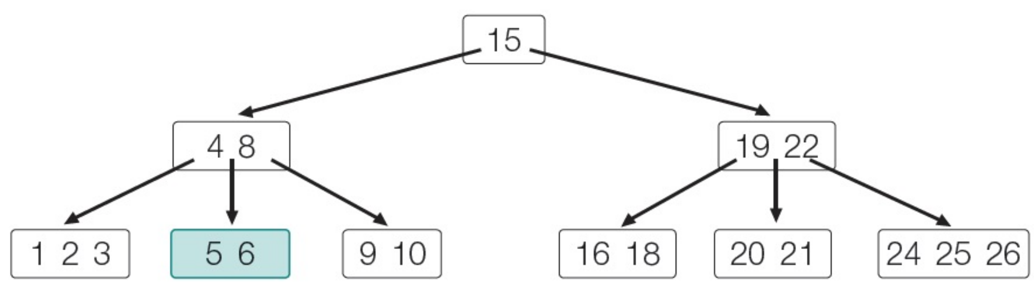
B-트리에서 삭제

◈ B-트리에서 삭제 과정 예 (1/3)

– 주어진 B-트리 ($k = 5$)



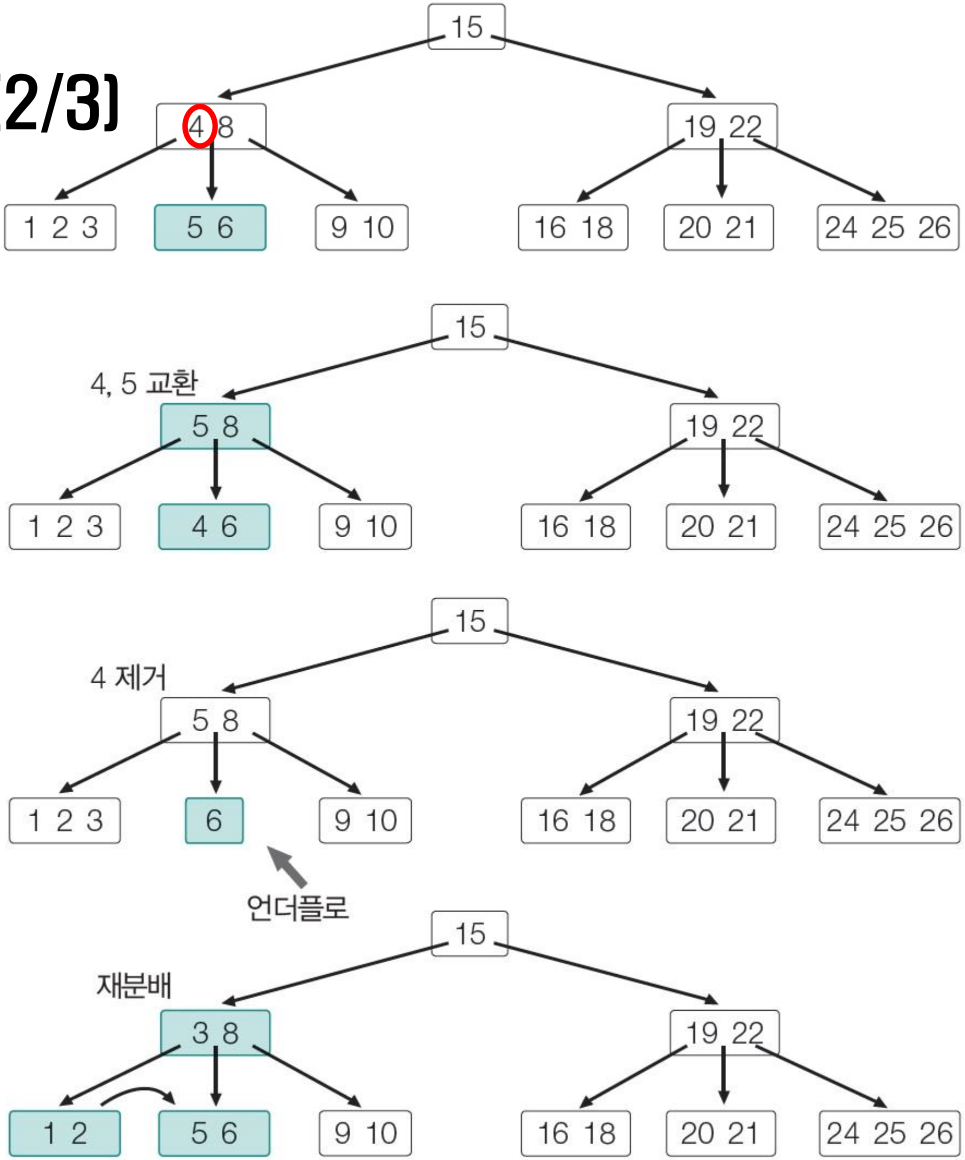
– 7 삭제
- 먼저 검색 알고리즘을 통해 키 7을 지닌 노드 v를 찾음
- 노드 v가 리프 노드라면 키 7 곧바로 삭제
- 언더프로(Underflow) 발생 하지 않음



B-트리에서 삭제

◈ B-트리에서 삭제 과정 예 (2/3)

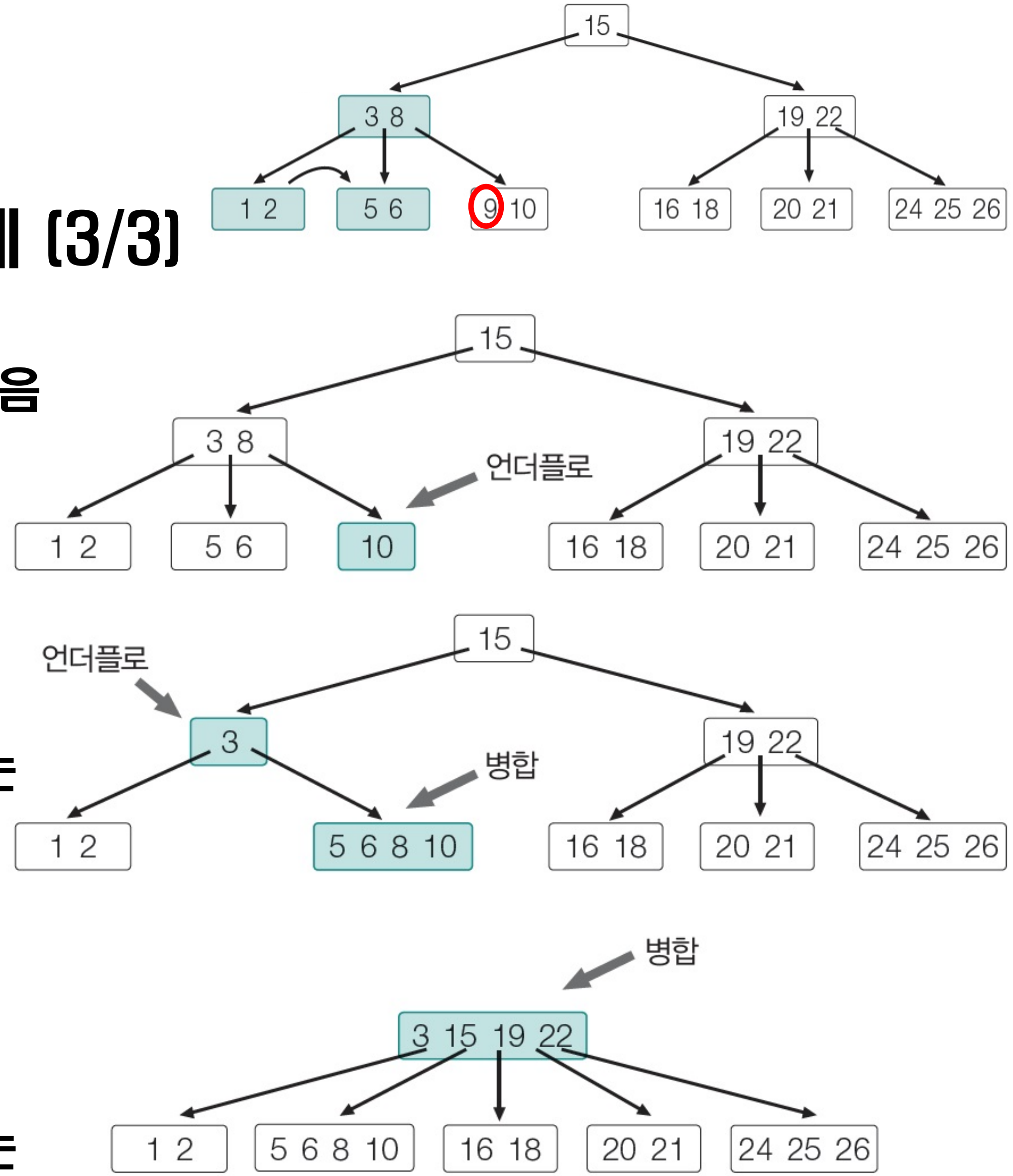
- 4 삭제
  - 키 4를 지닌 노드 v를 찾음
  - 노드 v가 리프 노드 아님
  - 4의 직후 키인 5를 지니고 있는 리프 노드 찾음
  - 키 4와 키 5를 교환
  - 리프 노드에 있는 키 4 제거
  - 해당 리프 노드에 언더플로우 발생
    - ➢ $\because \left\lfloor \frac{k}{2} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$
  - 형제 노드 중 여분이 있는 노드 존재
  - 부모 노드를 활용한 **재분배** (Redistribution)



B-트리에서 삭제

◈ B-트리에서 삭제 과정 예 (3/3)

– 9 삭제
  - 키 9를 지닌 노드 v를 찾음
  - 노드 v가 리프
  - 곧바로 9 삭제
  - 언더플로 발생
    - ➢ $\because \left\lfloor \frac{k}{2} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$
  - 형제 노드 중 여분이 있는 노드 존재하지 않음
  - 부모 노드에서 키 8을 받아오면서 병합 수행
  - 해당 부모 노드에 다시 언더플로 발생
  - 형제 노드 중 여분이 있는 노드 존재하지 않음
  - 부모 노드에서 키 15를 받아오면서 병합 수행



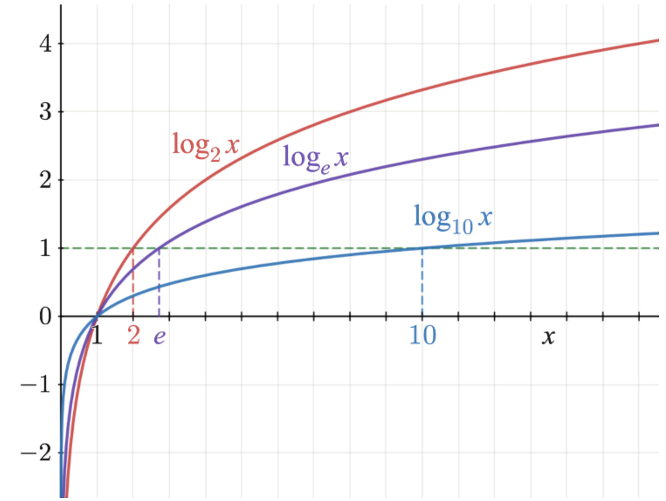
B-트리의 작업 성능 분석

◈ k진 B-트리의 최악의 경우 트리 높이

– $log_{\lfloor \frac{k}{2} \rfloor} n$

◈ k진 B-트리의 평균 트리 높이: h

– $log_k n \le h \le log_{\lfloor \frac{k}{2} \rfloor} n$ 범위 내 임의의 값



◈ B-트리의 검색, 삽입, 삭제 작업 수행 시간

– 시간 복잡도: $O(logn)$

– B-트리의 시간 복잡도는 디스크 접근 횟수와 동일

- 즉, 트리의 브랜치를 따라 다음 노드를 디스크에서 메모리로 가지고 오는 횟수

B-트리의 작업 성능 분석

◈B-트리의 삽입/삭제 작업 수행 시간에 대한 구체적 분석

- 삽입 시간 복잡도
  - 존재하는 않는 키에 대한 실패하는 검색: $O(logn)$
  - 오버플로 발생하지 않음: $O(1)$
  - 리프 노드에서 루트 노트까지 오버플로 발생: $O(logn)$
  - 따라서, 전체적인 삽입 시간 복잡도: $O(logn)$

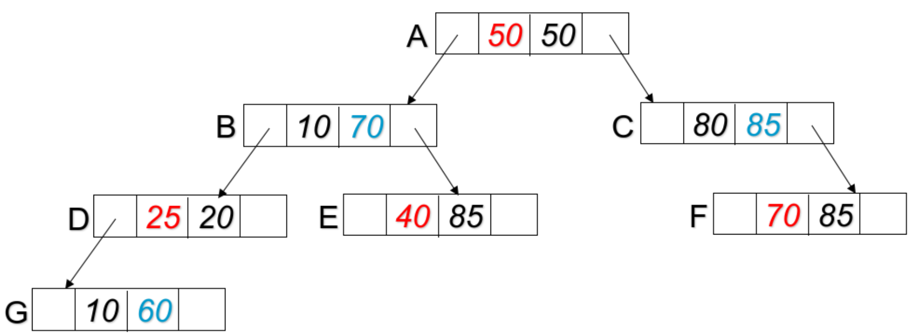
- 삭제 시간 복잡도
  - 삭제하려는 키 및 해당 키의 직후 키를 지닌 노드 검색: $O(logn)$
  - 삭제하려는 노드에서 언더플로 발생하지 않음: $O(1)$
  - 삭제하려는 노드에서부터 루트 노드까지 언더플로 발생: $O(logn)$
  - 따라서, 전체적인 삽입 시간 복잡도: $O(logn)$

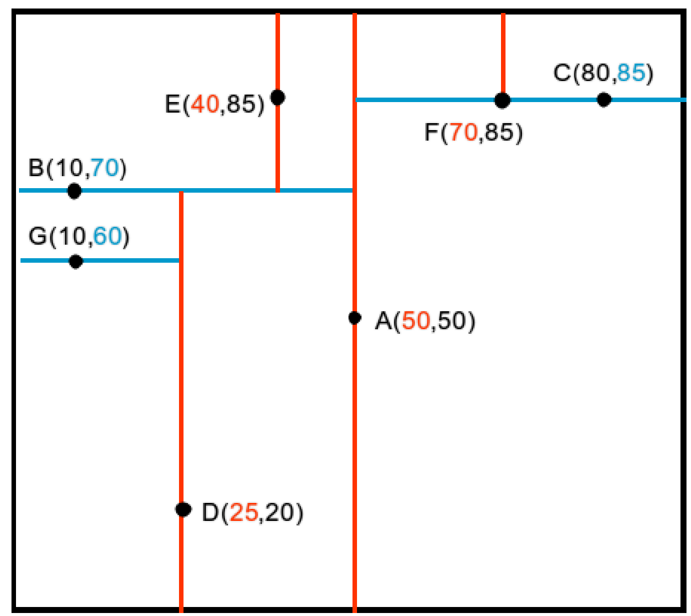
05. 다차원 검색 트리

다차원 검색 트리

◈ 일차원 검색 트리
- 각 노드에 존재하는 키는 하나의 필드로 구성

◈ 다차원 검색 트리
- 각 노드에 존재하는 키가 여러 개의 필드로 구성





Questions & Answers