ALGORITHM
알고리즘

Lecture 06

탐색 (1)

컴퓨터과학과 | 이관용 교수

학습목차

01. 순차 탐색

탐색 관련 기본 개념

탐색?

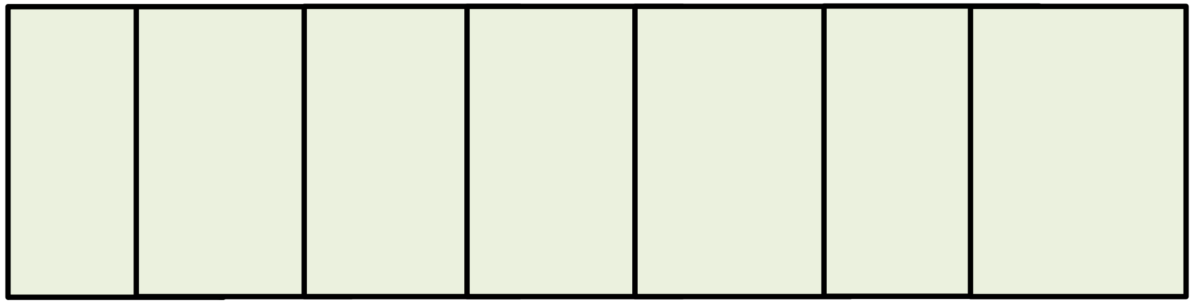
- 여러 개의 원소로 구성된 데이터에서 원하는 값을 갖는 원소를 찾는 것
  - ✓ 데이터의 형태 → 리스트, 트리, 그래프 등
  - ✓ 내부 탐색 vs 외부 탐색
  - ✓ 관련 연산 → 탐색 + (초기화, 삽입, 삭제)

탐색 방법

- 리스트 형태 → 순차 탐색, 이진 탐색
- 트리 형태 → 이진 탐색 트리, 2-3-4 트리, 레드-블랙 트리, B-트리
- 해시 테이블 → 해시 함수, 충돌 해결 방법

순차 탐색

섞어서 뒤집어 놓은 카드 중에서 10을 찾아라!



- 리스트 형태로 주어진 원소들을 처음부터 하나씩 차례로("순차") 비교하면서 원하는 값을 갖는 원소를 찾는 방법

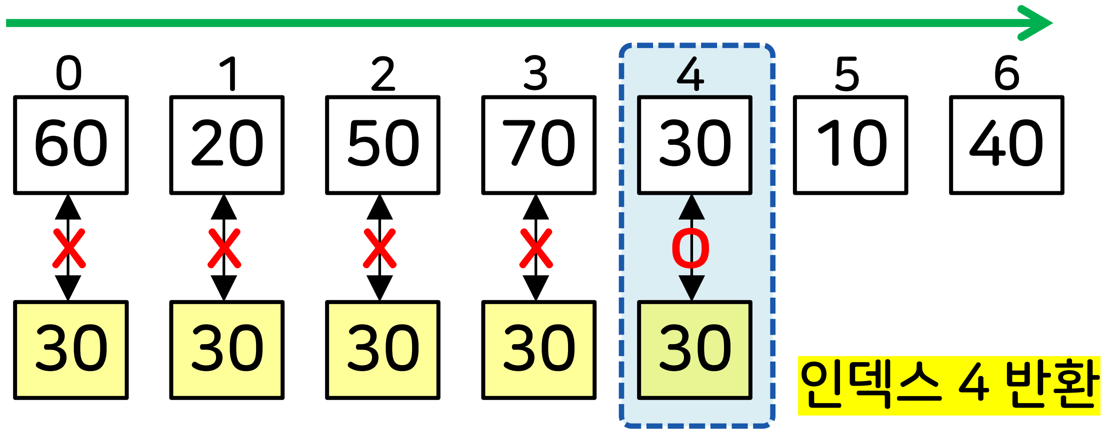
순차 탐색_탐색 연산

```
SequentialSearch (A[ ], n, x)
{
  i = 0;
  while (i < n && A[i] != x)
     i = i + 1;
  return (i);
}
```

입력: A[0..n-1] : 입력 배열
n : 입력 크기(탐색할 데이터의 개수)
x : 탐색 키

출력: x가 배열 내에 존재하면 인덱스, 아니면 n을 반환

배열 A[ ]에서 탐색 키 30의 탐색 과정



순차 탐색_삽입 연산


```
SequentialSearch_Insert (A[ ], n, x)
{
  A[n] = x;
  return (A, n+1);
}
```

입력: A[0..n-1] : 입력 배열
n : 입력 크기(탐색할 데이터의 개수)
x : 삽입할 원소
출력: A[0..n], n+1

배열 A[ ]에 원소 35의 삽입 과정

						n-1	n
0	1	2	3	4	5	6	7
60	20	50	70	30	10	40	

삽입할 원소 35

순차 탐색_삭제 연산

```
SequentialSearch_Delete (A[ ], n, x)
{
   Index = SequentialSearch (A, n, x);
   if (Index == -1) return (A, n);
   A[Index] = A[n-1];
   return (A, n-1);
}
```

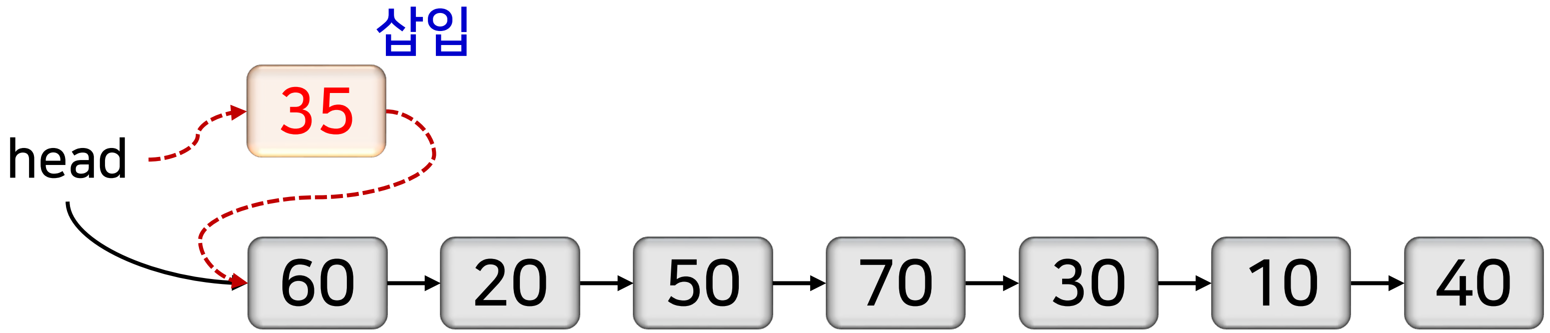
입력: A[0..n-1] : 입력 배열
n : 입력 크기(탐색할 데이터의 개수)
x : 삭제할 원소
출력: A[0..n-2], n-1

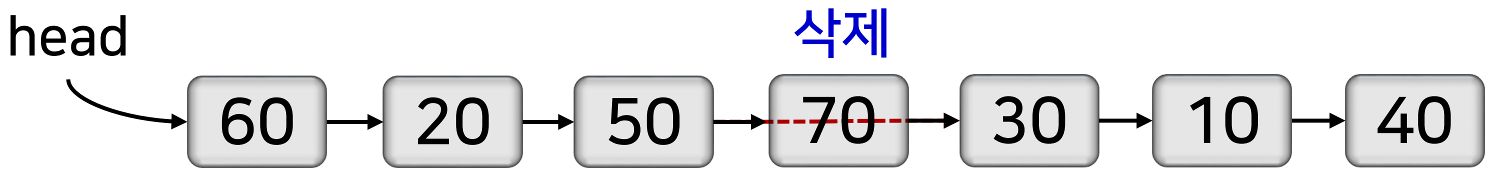
배열 A[ ]에 원소 70의 삭제 과정

					n-2	n-1
0	1	2	3	4	5	6
60	20	50	70	30	10	40

삭제할 원소 70

순차 탐색_연결 리스트로 구현한 경우





성능과 특징

탐색, 삭제 연산의 시간 복잡도 → O(n)

- 탐색 성공 → 1번 ∽ n번 비교 (평균 (n+1)/2번), 탐색 실패 → 항상 n번 비교
- 삭제 → 삭제할 원소의 순차 탐색 O(n) 후, 마지막 원소의 이동 O(1)

삽입 연산의 시간 복잡도 → O(1)

- 리스트의 마지막에 추가하는 데 상수 시간만 필요

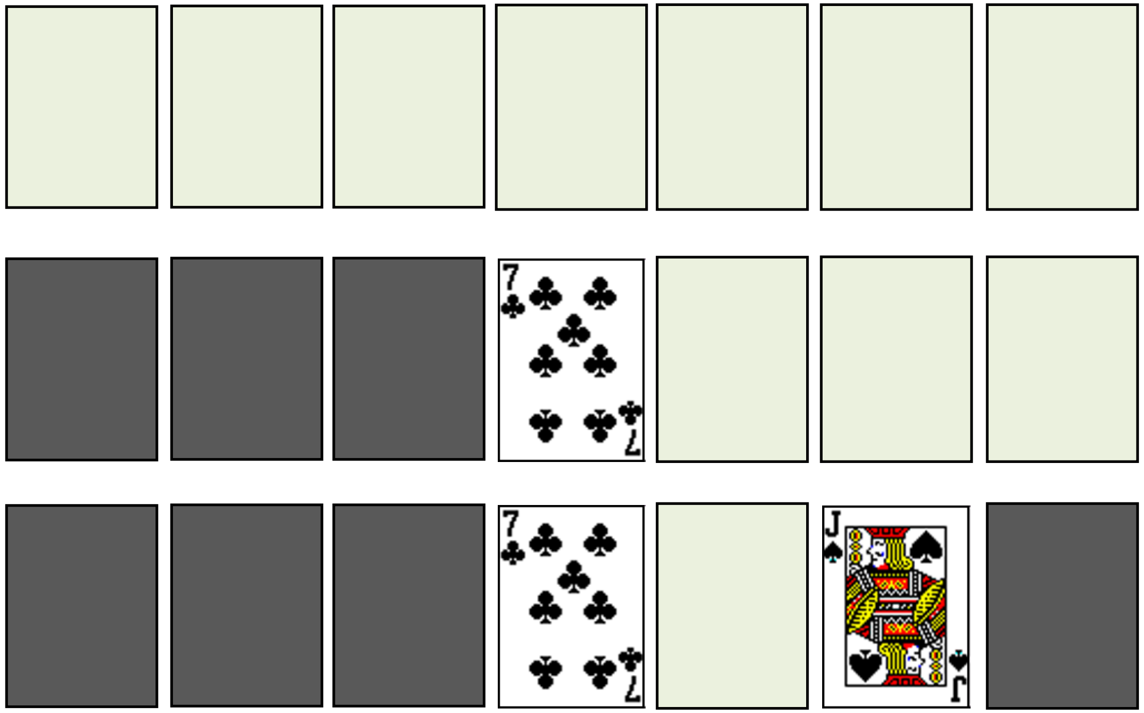
정렬되지 않고 크기가 작은 데이터에 적합

- 모든 리스트 형태의 입력에 적용 가능 → 비정렬 데이터 탐색에 적합
- 탐색과 삭제에 O(n) 시간이 필요 → 데이터가 큰 경우에는 부적합

02. 이진 탐색

이진 탐색

순서대로 나열해서 뒤집어 놓은 카드 중에서 10을 찾아라!



이진 탐색

정렬된 리스트 형태로 주어진 원소들을 절반씩 줄여 가면서 원하는 값을 가진 원소를 찾는 방법

- 분할정복 방법이 적용됨

탐색 방법

$$mid = \frac{(\text{시작 인덱스 } Left + \text{마지막 인덱스 } Right)}{2}$$

- 배열의 가운데 원소 A[mid]와 탐색 키 key를 비교

(1) A[mid] = key → 탐색 성공(인덱스 mid 반환 후 종료)

(2) key < A[mid] → '이진 탐색(원래 크기의 ½인 왼쪽 부분배열)' 순환 호출

(3) A[mid] < key → '이진 탐색(원래 크기의 ½인 오른쪽 부분배열)' 순환 호출

탐색을 반복할 때마다 대상 원소의 개수가 ½씩 감소

이진 탐색_탐색 연산


```
BinarySearch (A[ ], key, Left, Right)
{
   if (Left > Right) return (-1);
   mid = ⌊(Left + Right) /2⌋;
   if (A[Mid] == key) return (Mid);
   else if (key < A[Mid]) BinarySearch(A, key, Left, Mid-1)
         else BinarySearch(A, key, Mid+1, Right);
}
```

$T(n) = T(n/2) + O(1)$ $(n>1)$, $T(1)=1$

$T(n) = O(\log n)$

Left=0　　Mid=4　　Right=8

10	15	20	25	30	35	40	45	50

왼쪽 부분배열
A[Left .. Mid-1]

오른쪽 부분배열
A[Mid+1 .. Right]

이진 탐색_탐색 연산

Left=0 Right=8

10	15	20	25	30	35	40	45	50

탐색키 key=35

Mid=4

A[Mid] < key이므로 오른쪽 부분배열을 다시 탐색

Left=5 Right=8

10	15	20	25	30	35	40	45	50

Mid=6

key < A[Mid]이므로 왼쪽 부분배열을 다시 탐색

Left=Right=5

10	15	20	25	30	35	40	45	50

Mid=5

key = A[Mid]이므로 탐색 성공(Mid=5 반환)

이진 탐색_초기화 연산

주어진 배열이 정렬되어 있지 않으면 정렬 수행

```
BinarySearch_Initialize (A[ ], n)
{
  for (i= 0; i < n-1; i++)                O(n)
    if (A[i] > A[i+1]) {
       A = Sort (A, n);   // 가정: 오름차순으로 정렬
       break;                O(nlogn)
    }
  return (A);
}
```

⇒ O(nlogn)

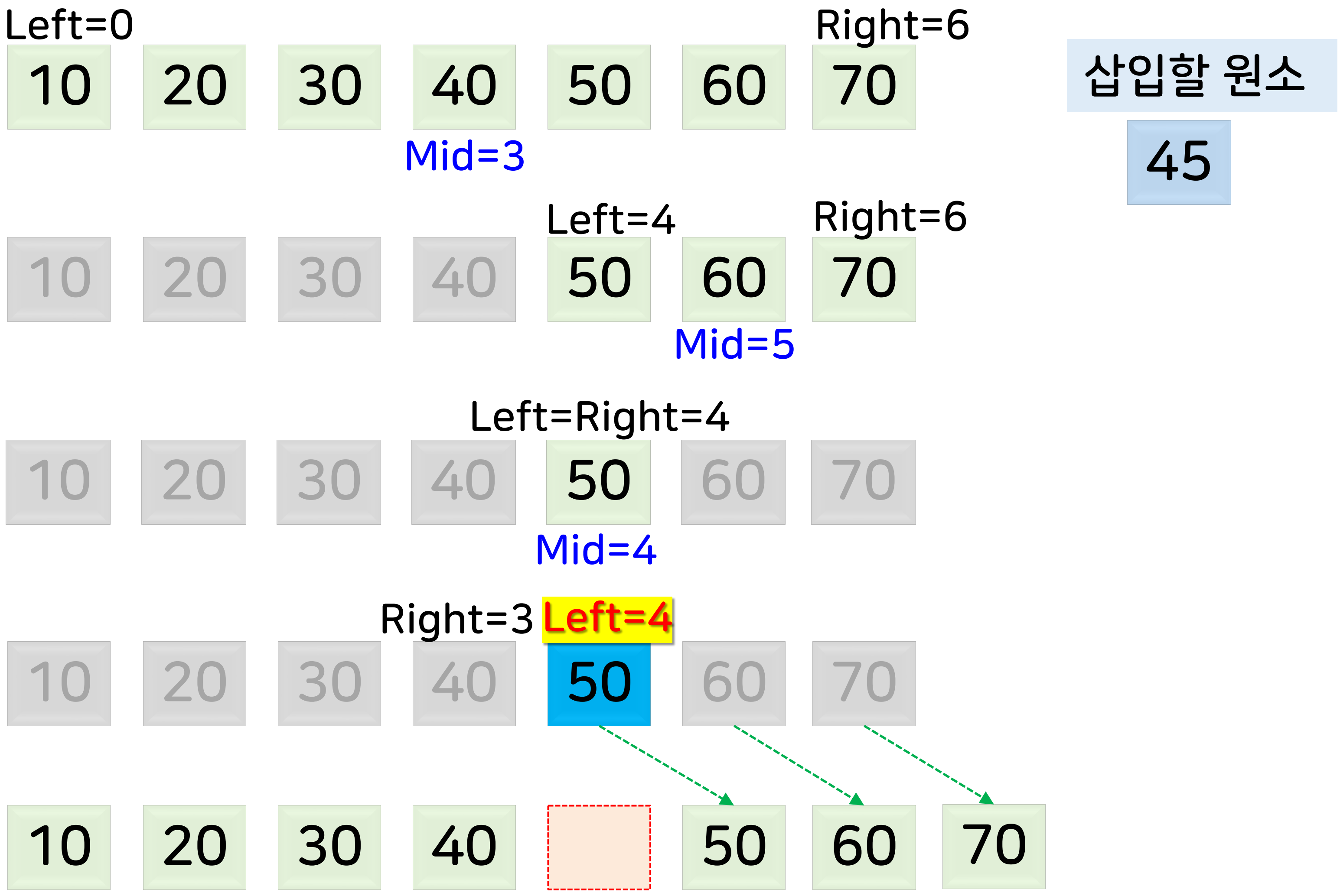
이진 탐색_삽입 연산


```
BinarySearch_Insert (A[ ], n, x)
{
  Left = 0; Right = n-1;                    O(logn)
  while (Left <= Right) {
    Mid = ⌊(Right – Left + 1) / 2 ⌋+ Left;
    if ( x == A[Mid] ) return (A, n);          // 삽입할 원소가 이미 존재
    else  if ( x < A[Mid] ) Right = Mid – 1;   // 왼쪽 부분배열 탐색
              else Left = Mid + 1;             // 오른쪽 부분배열 탐색
  }
  for (i=n; i > Left; i--)  A[i] = A[i-1];   // A[Left]부터 오른쪽으로 한 칸씩 이동
  A[Left] = x;                               // 원소 삽입
  return (A, n+1);                           O(n)
}
```

반복문으로 구현한
이진 탐색 알고리즘

O(n)

이진 탐색_삽입 연산


Left=0
Right=6
10 20 30 40 50 60 70
Mid=3
삽입할 원소
45
Left=4
Right=6
10 20 30 40 50 60 70
Mid=5
Left=Right=4
10 20 30 40 50 60 70
Mid=4
Right=3
Left=4
10 20 30 40 50 60 70
10 20 30 40 50 60 70

이진 탐색_삭제 연산

```
BinarySearch_Delete (A[ ], n, x)
{
  Index = BinarySearch (A, x, 0, n-1);
  if (Index == -1) return (A, n);    // 삭제할 원소가 존재하지 않음
  for (i=Index; i < n-1; i++)        // 삭제할 위치의 오른쪽 모든 원소를
     A[i] = A[i+1];                  // 왼쪽으로 한 칸씩 이동(원소 삭제)
  return (A, n-1);
}
```

O(logn)

O(n)

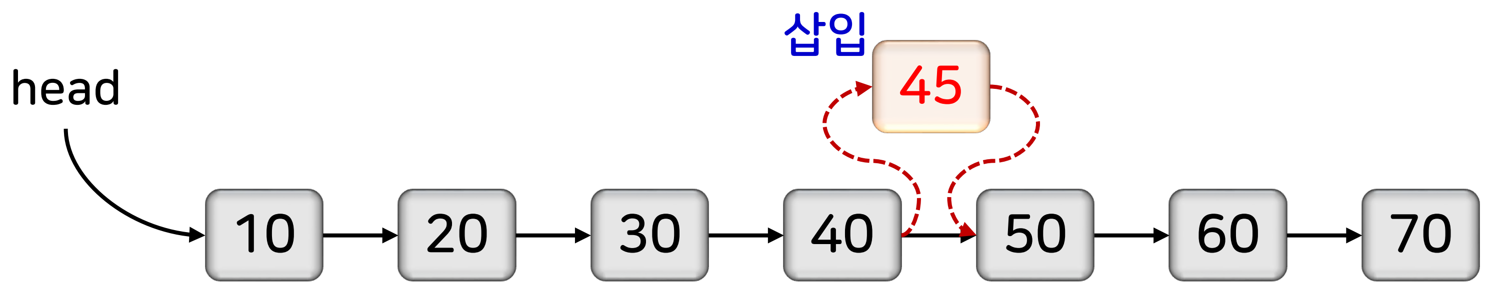
O(n)

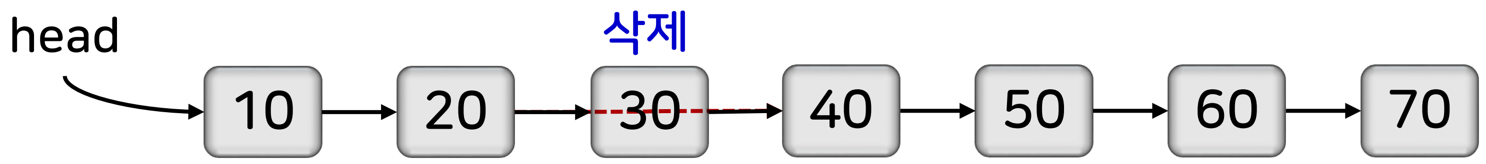
삭제할 원소	30

Index=2

10	20	30	40	45	50	60	70

이진 탐색을 연결 리스트로 구현하면





연결 리스트 구조에서는 이진 탐색 자체가 불가능

성능과 특징

성능

- 탐색 연산 → O(logn)
- 초기화 연산 → O(nlogn)
- 삽입/삭제 연산 → O(n)

정렬된 리스트에 대해서만 적용 가능

삽입과 삭제가 빈번한 경우에는 부적합

- 연산 후 리스트의 정렬 상태를 유지하기 위해서 O(n)의 데이터 이동이 필요

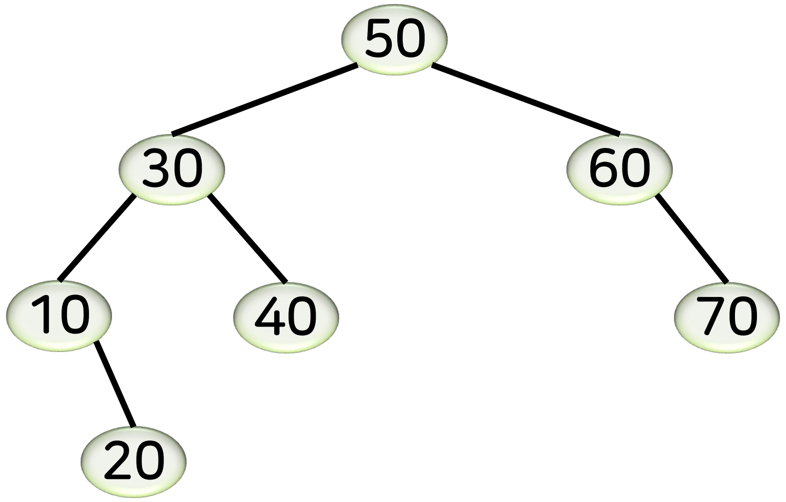
  → 데이터가 작은 경우에 적합

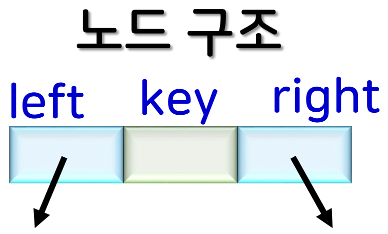
03. 이진 탐색 트리

이진 탐색 트리?

이진 트리

- 한 노드의 왼쪽 서브트리에 있는 모든 키 값은 그 노드의 키값보다 작다.
- 한 노드의 오른쪽 서브트리에 있는 모든 키 값은 그 노드의 키값보다 크다.

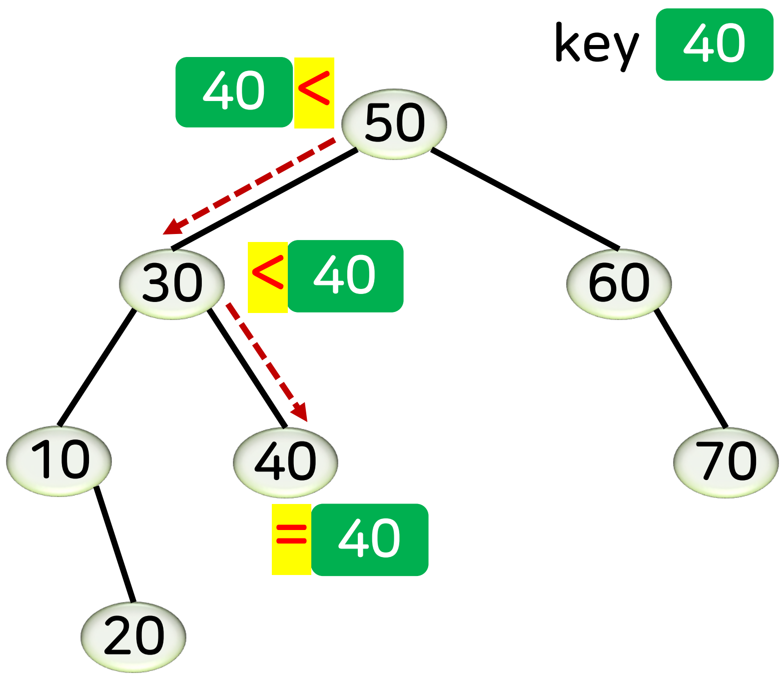





```
struct node {
  struct node *left;
  int key;
  struct node *right;
}
```

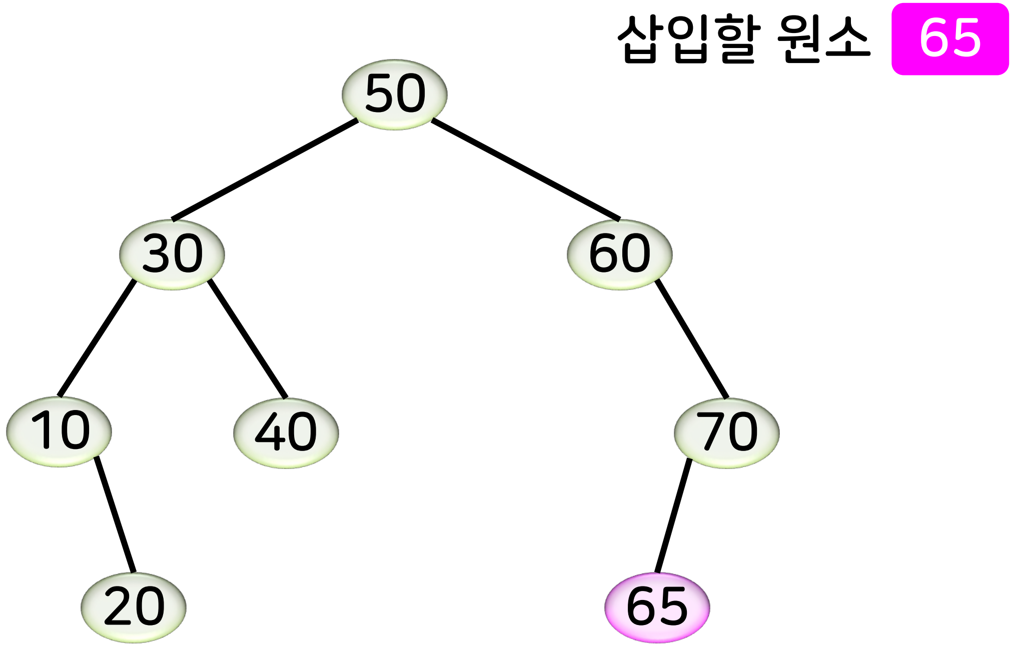
이진 탐색 트리_탐색 연산

루트 노드에서부터 시작해서 값의 크기 관계에 따라 트리의 경로를 따라 내려가면서 탐색 진행



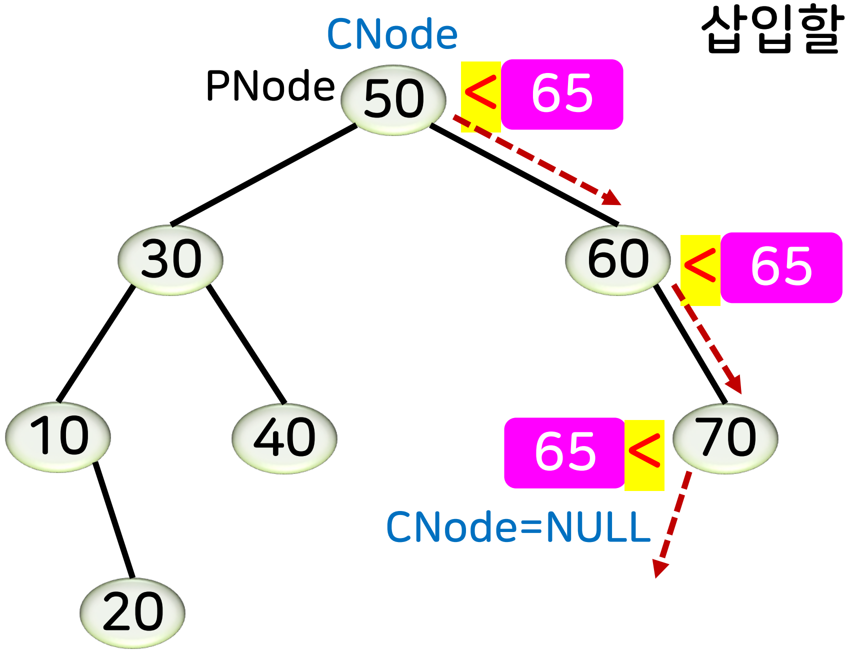
이진 탐색 트리_삽입 연산

삽입할 원소를 탐색한 후, 탐색이 실패하면 해당 위치에 자식 노드로서 새 노드를 추가



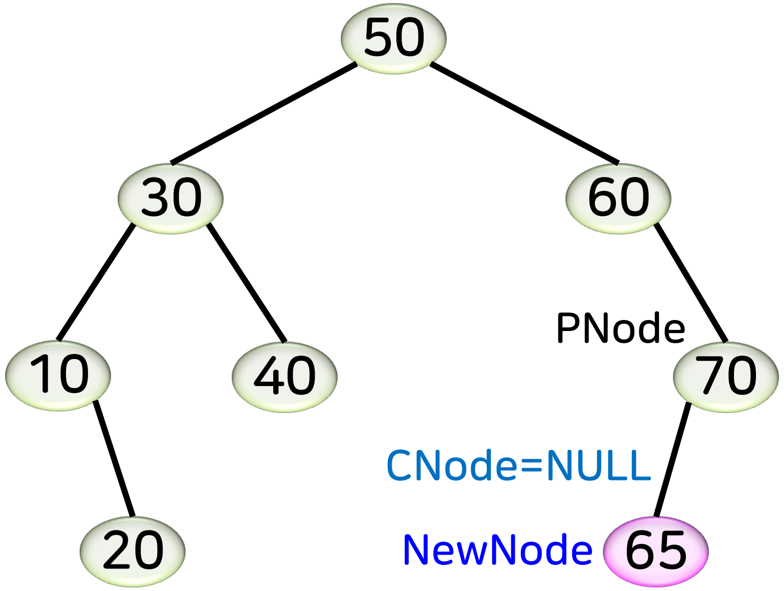
이진 탐색 트리_삽입 연산

삽입할 원소 65



이진 탐색 트리_삽입 연산

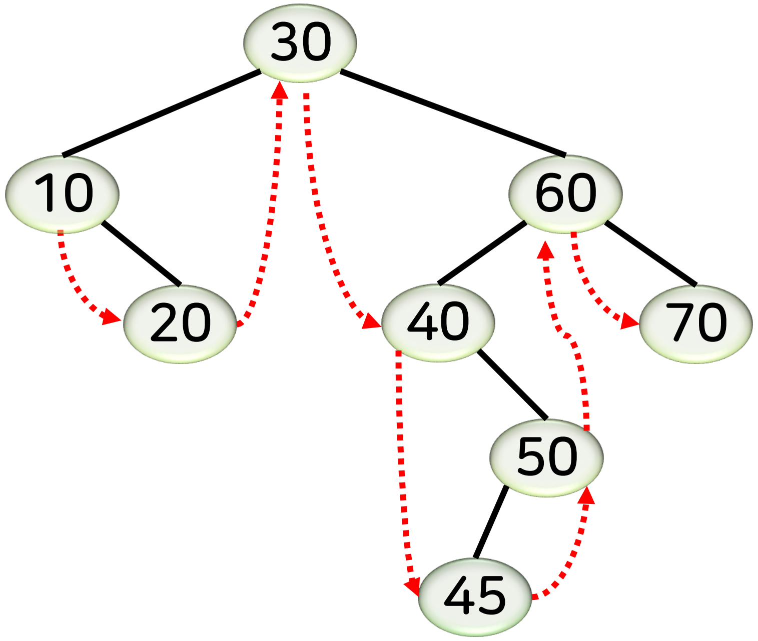
삽입할 원소 65



이진 탐색 트리_삭제 연산

후속자 successor, 계승자 노드

- 어떤 노드의 바로 다음 키값을 갖는 노드



10	20	30	40	45	50	60	70

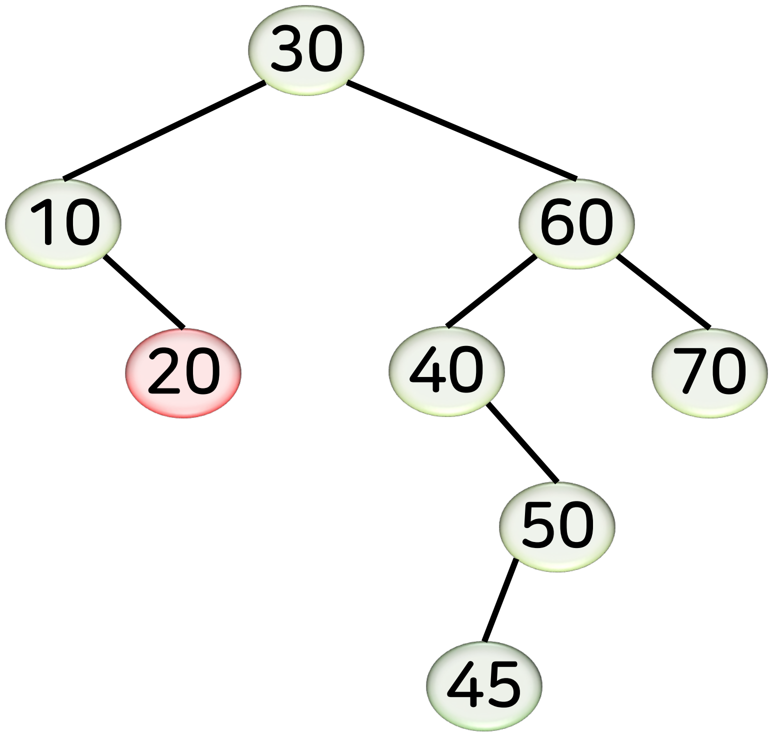
이진 탐색 트리_삭제 연산

삭제되는 노드의 자식 노드의 개수에 따라 구분해서 처리

1. 자식 노드가 없는 경우(리프 노드의 경우)
   - ✓ 남는 노드가 없어 위치 조절이 불필요
2. 자식 노드가 하나인 경우
   - ✓ 자식 노드를 삭제되는 노드의 위치로 올리면서 서브트리 전체도 따로 올림
3. 자식 노드가 2개인 경우
   - ✓ 삭제되는 노드의 후속자 노드를 삭제되는 노드의 위치로 올리고,
   - ✓ 후속자 노드를 삭제되는 노드로 취급하여 자식 노드의 개수에 따라 다시 처리

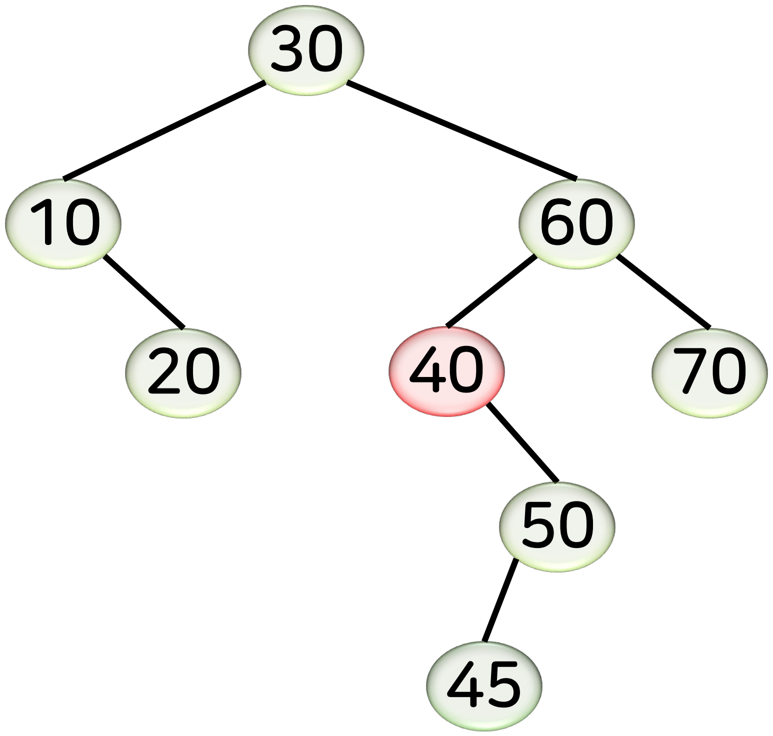
이진 탐색 트리_삭제 연산

<경우 1> 노드 20 삭제



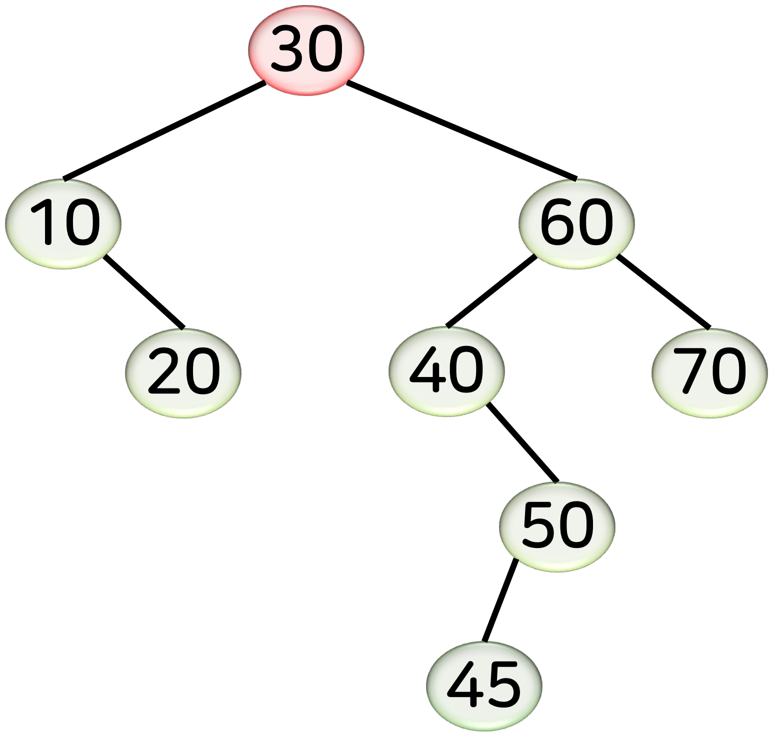
이진 탐색 트리_삭제 연산

<경우 2> 노드 40 삭제



이진 탐색 트리_삭제 연산

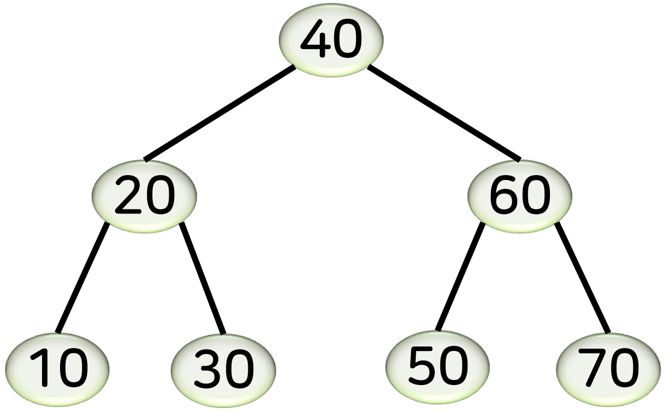
<경우 3> 노드 30 삭제



성능과 특징

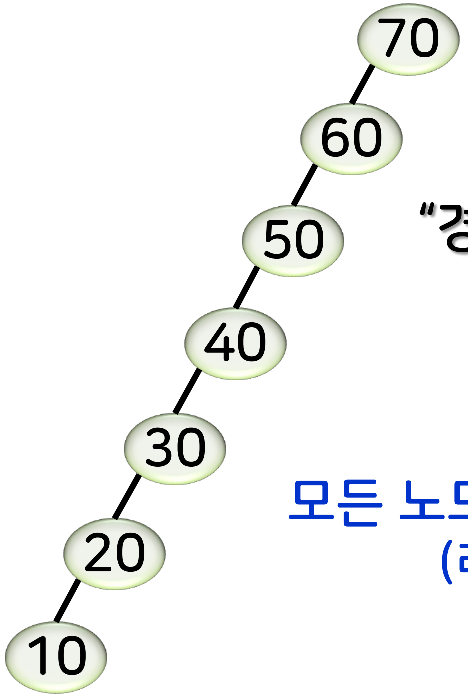
탐색, 삽입, 삭제 연산의 시간 복잡도

- 키값을 비교하는 횟수에 비례 → 이진 트리의 높이가 h라면 O(h)



모든 노드의 차수가 2인 경우
(리프 노드 제외)

평균 수행시간 O(logn)



"경사 이진 트리"

모든 노드의 차수가 1인 경우
(리프 노드 제외)

최악 수행시간 O(n)

성능과 특징

삽입/삭제 연산 시 기존 노드의 이동이 거의 발생하지 않음

- 삽입 연산 → 노드의 이동이 없음
- 삭제 연산 → 상수 번 이동(0, 1, 1 또는 2)

원소의 삽입/삭제에 따라 경사 트리 형태가 될 수 있음

- 최악의 수행시간 O(n)을 가짐
  - ✓ 경사 트리가 만들어지지 않도록 트리의 균형을 유지해서 O(logn)을 보장

    → 균형 탐색 트리(탐색 트리의 좌우 서브트리가 같은 높이를 유지하는 자료구조)

    → 2-3-4 트리, 레드-블랙 트리, B-트리

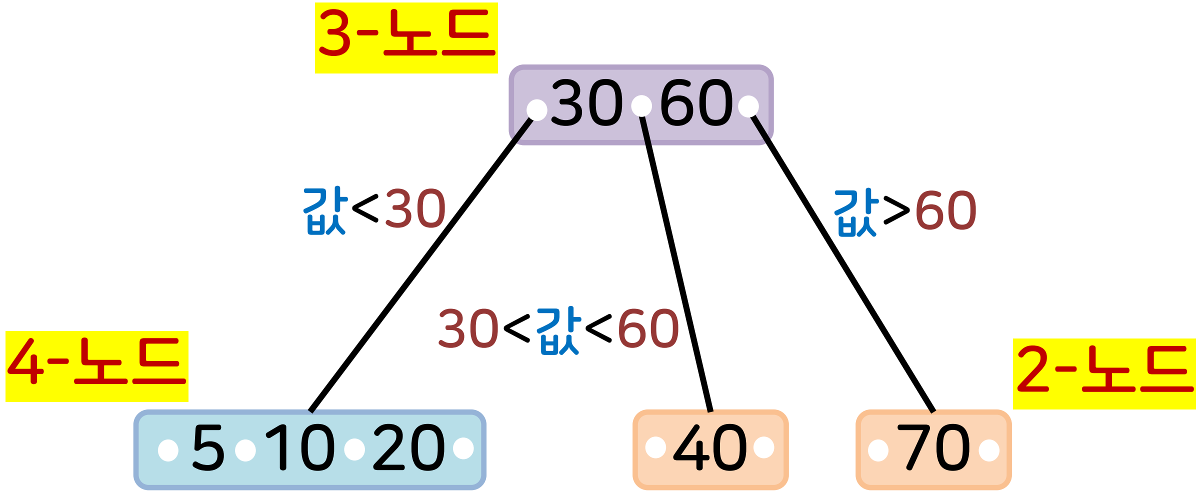
04. 2-3-4 트리

2-3-4 트리?

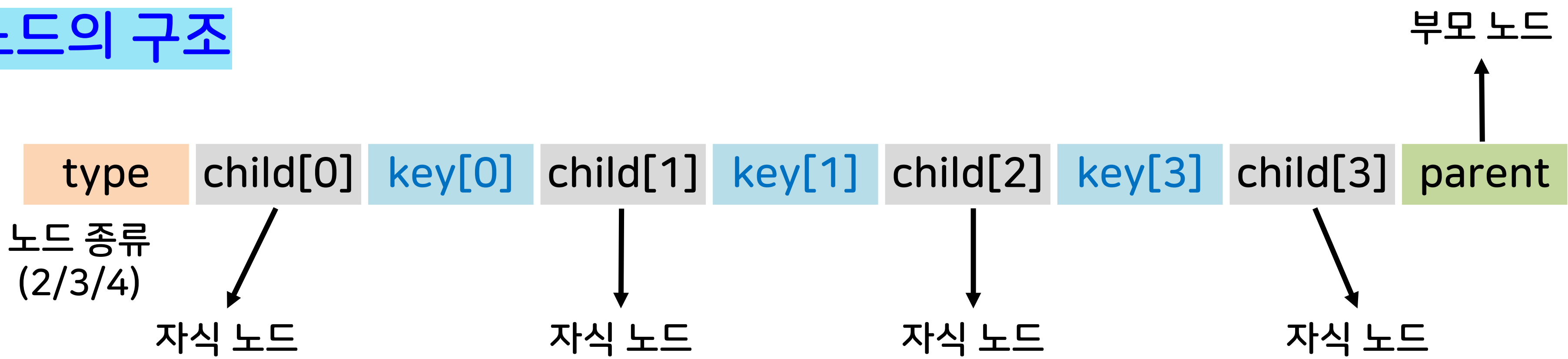
다음 성질을 만족하는 균형 탐색 트리

- 2-노드 → 1개의 키와 2개의 자식을 갖는 노드
- 3-노드 → 2개의 키와 3개의 자식을 갖는 노드
- 4-노드 → 3개의 키와 4개의 자식을 갖는 노드
- 각 노드의 한 키의 왼쪽 서브트리에 있는 모든 키값은 그 키값보다 작다.
- 각 노드의 한 키의 오른쪽 서브트리에 있는 모든 키값은 그 키값보다 크다.
- 모든 리프 노드의 레벨은 동일

2-3-4 트리

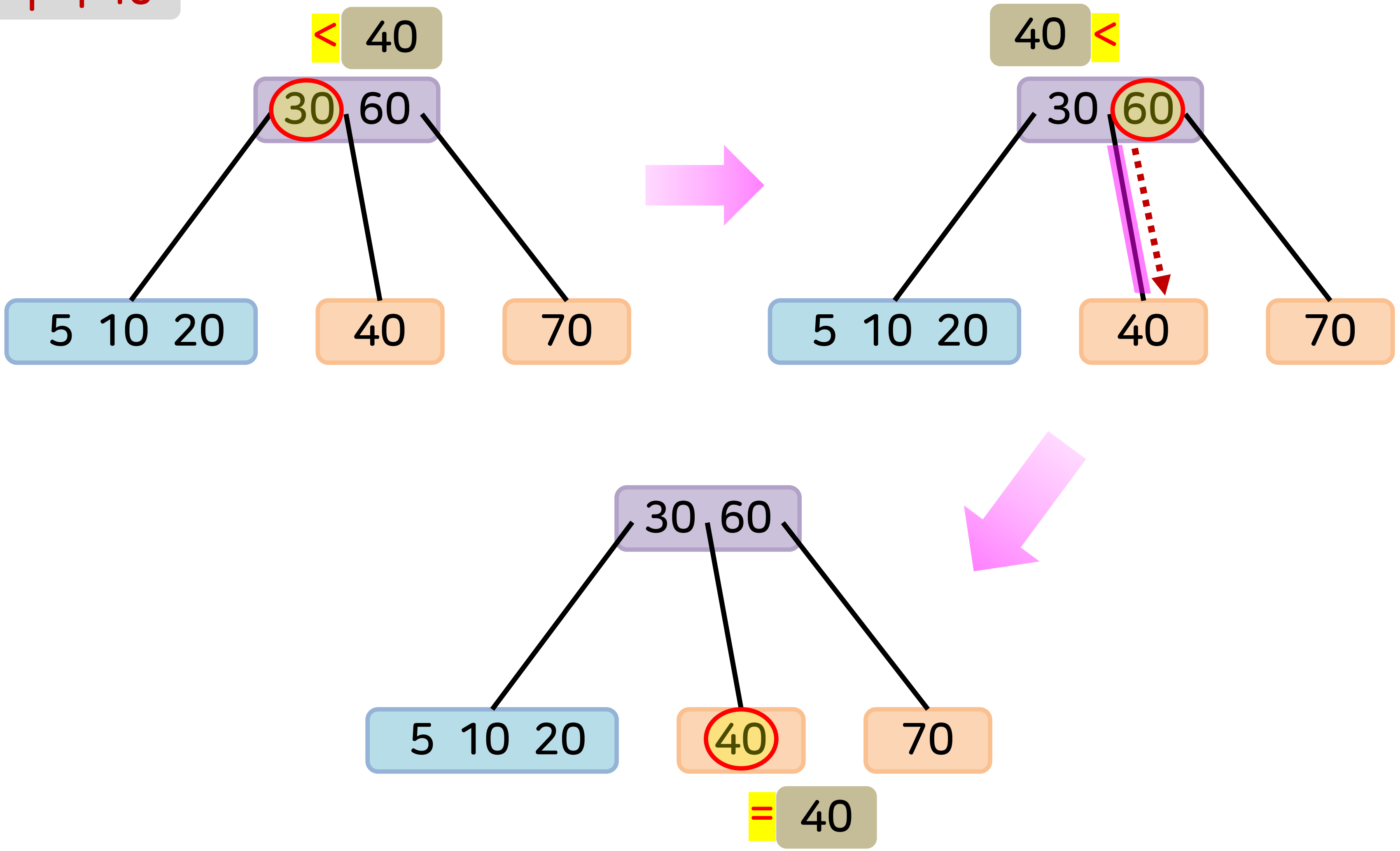

3-노드
30 60
값<30
30<값<60
값>60
4-노드
5 10 20
40
70
2-노드

노드의 구조


부모 노드
type
child[0]
key[0]
child[1]
key[1]
child[2]
key[3]
child[3]
parent
노드 종류
(2/3/4)
자식 노드
자식 노드
자식 노드
자식 노드

2-3-4 트리_탐색 연산

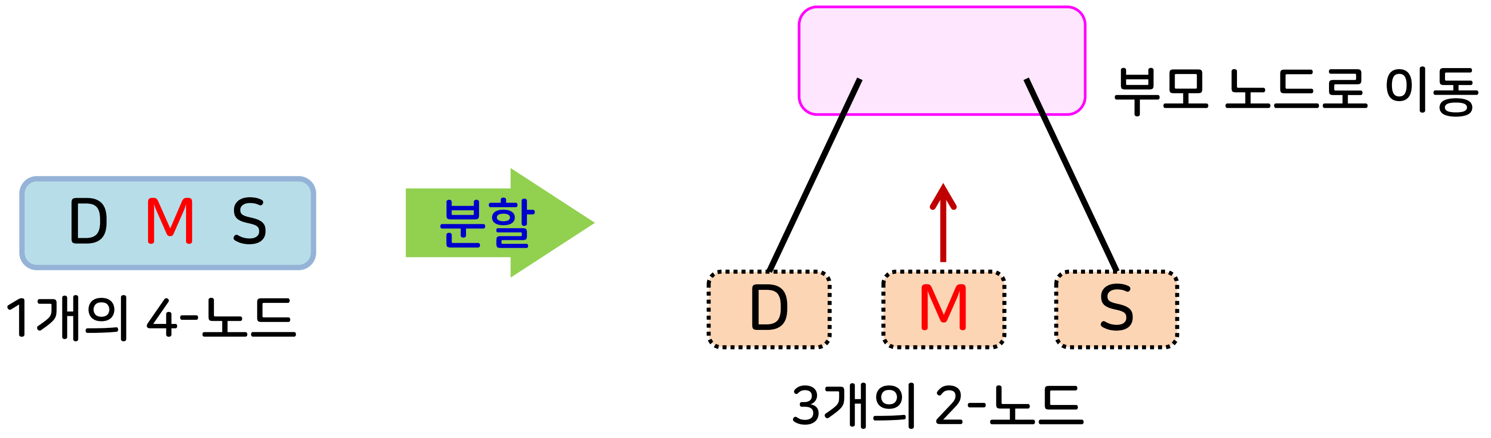
탐색 키 40



2-3-4 트리_삽입 연산

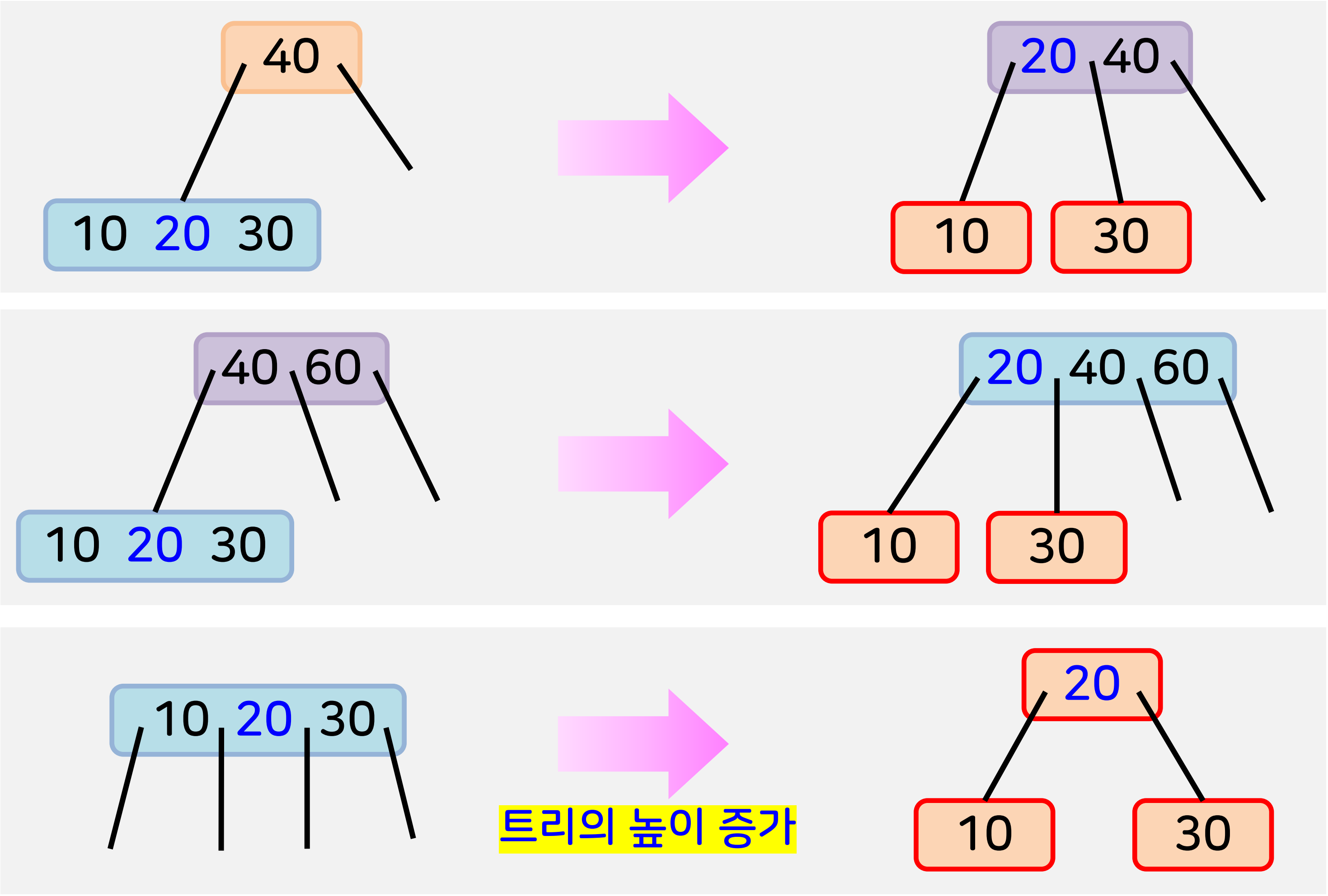
- 탐색 과정에서 4-노드를 만나면 항상 노드 분할을 우선 수행

- 노드 분할



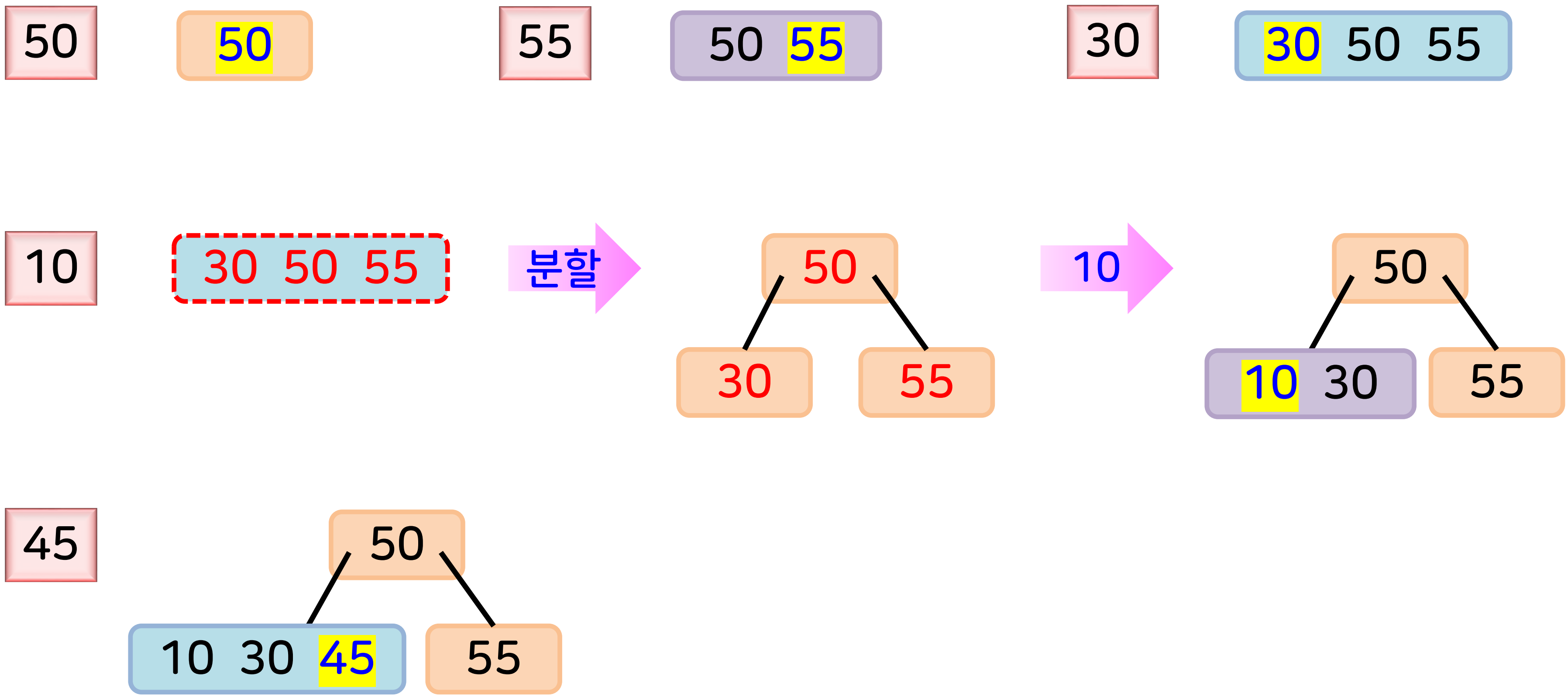
2-3-4 트리_삽입 연산

노드 분할의 유형

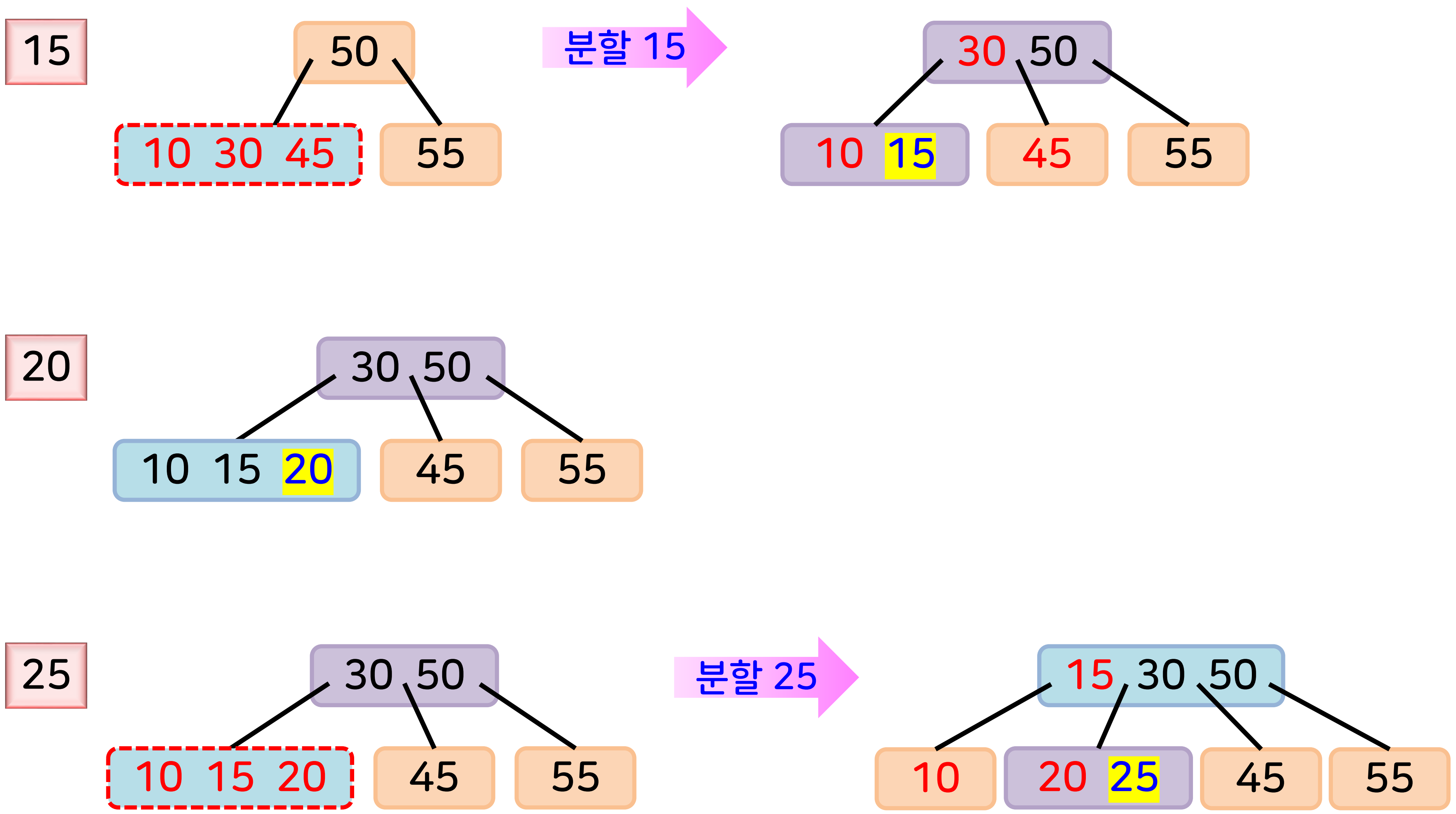


2-3-4 트리_삽입 연산의 예

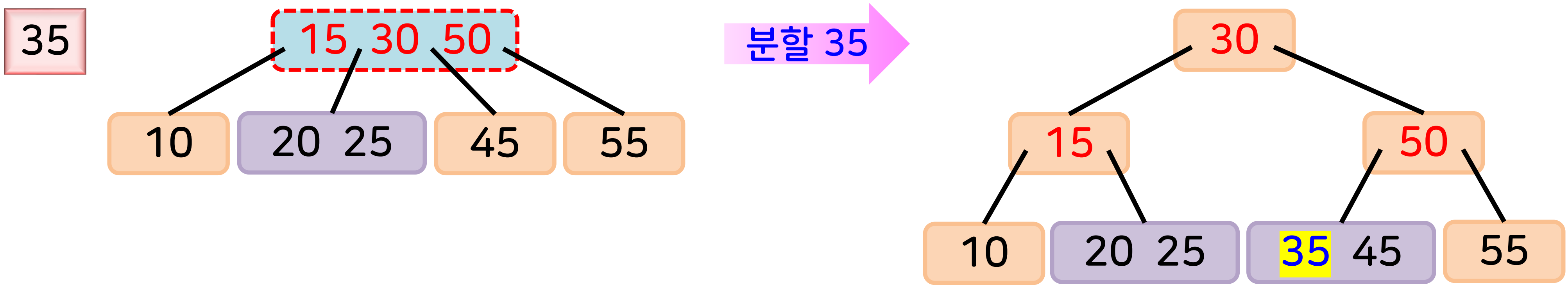
키값 → 50 55 30 10 45 15 20 25 35 40

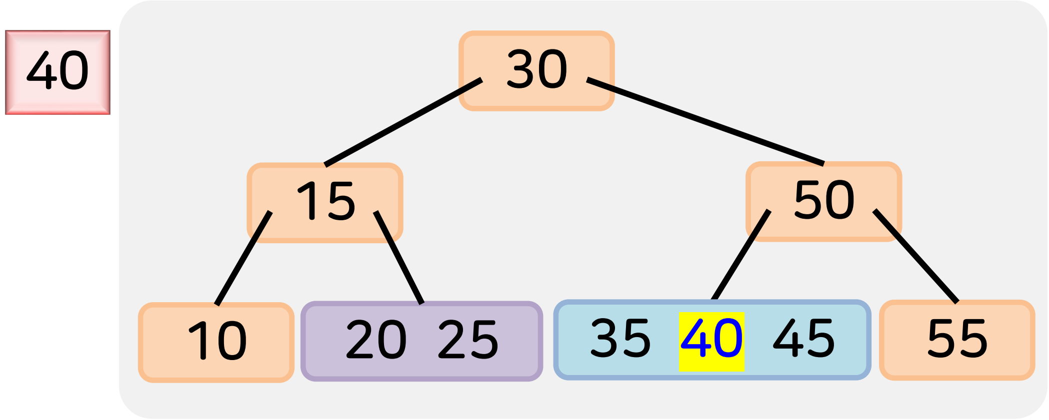


2-3-4 트리_삽입 연산의 예



2-3-4 트리_삽입 연산의 예





성능과 특징

탐색, 삽입, 삭제 연산의 시간 복잡도 → O(logn)

- 균형 탐색 트리 → 트리의 최대 높이 $\lfloor \log n \rfloor$

삽입/삭제가 일어나도 경사 트리가 되지 않음

- 루트 노드가 분할되는 경우에 한해서 모든 노드의 레벨이 동일하게 1씩 증가

2-3-4 트리를 그대로 구현하면 노드 구조가 복잡해서 이진 탐색 트리보다 더 느려질 가능성이 많음

정리하기

1. 순차 탐색

- O(n), 비정렬 데이터 탐색에 가장 적합, 데이터가 큰 경우 부적합

2. 이진 탐색

- 탐색 O(logn), 초기화 O(nlogn), 삽입/삭제 O(n)
- 정렬된 리스트에만 적용 가능, 삽입/삭제가 연산이 빈번한 응용에 부적합

3. 이진 탐색 트리

- 성질-2가지, 연산(탐색, 삽입, 삭제-3가지 경우)
- 평균 O(logn), 최악 O(n)
- 삽입/삭제가 진행됨에 따라 최악의 성능을 갖는 경사 트리가 될 수 있음

4. 2-3-4 트리

- 성질-6가지, 연산(탐색, 삽입), 4-노드 분할, O(logn)
- 그대로 구현하면 노드의 구조가 복잡해서 이진 탐색 트리보다 더 느려질 가능성 있음

ALGORITHM 알고리즘

다음시간에는

Lecture 07

탐색 (2)

컴퓨터과학과 | 이관용 교수