탐색 알고리즘_1

2020년도 2학기 최 희석

목차

- 탐색의 개념
- 순차 탐색
- 이진 탐색
- 이진 탐색 트리

탐색의 개념

탐색의 개념

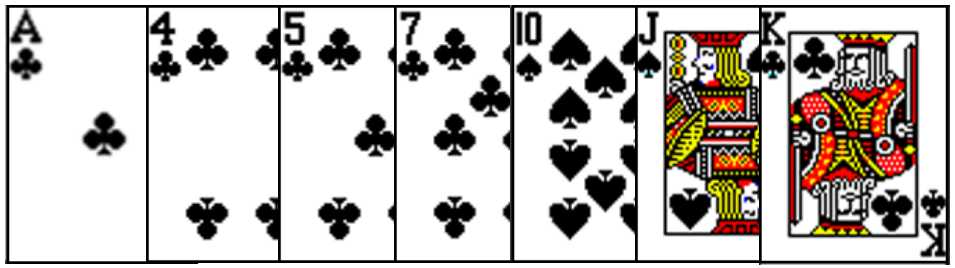
- 여러 원소로 구성된 데이터에서 원하는 값을 가진 원소를 찾는 것
  - 데이터의 형태 → 리스트, 트리, 그래프 등
  - 내부 탐색 vs 외부 탐색
  - 탐색 연산 + 초기화(정렬), 삽입, 삭제 등의 연산도 함께 고려해야 함

  - 순차 탐색
  - 이진 탐색
  - 탐색 트리 → 이진 탐색 트리, 흑적 트리, B-트리
  - 해싱
  - 해시 함수, 충돌 해결 방법

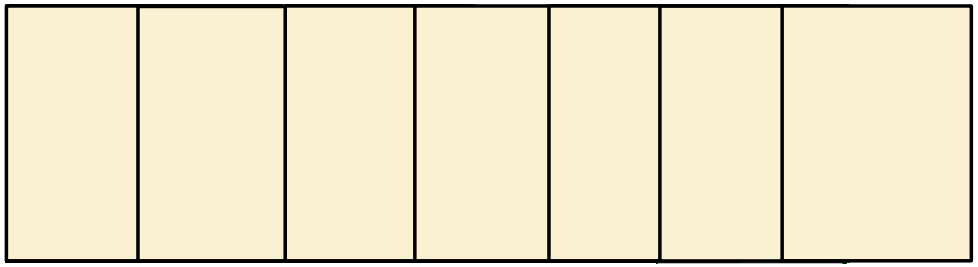
순차 탐색

순차 탐색

미 정렬된 카드 중에서 원하는 카드 찾기



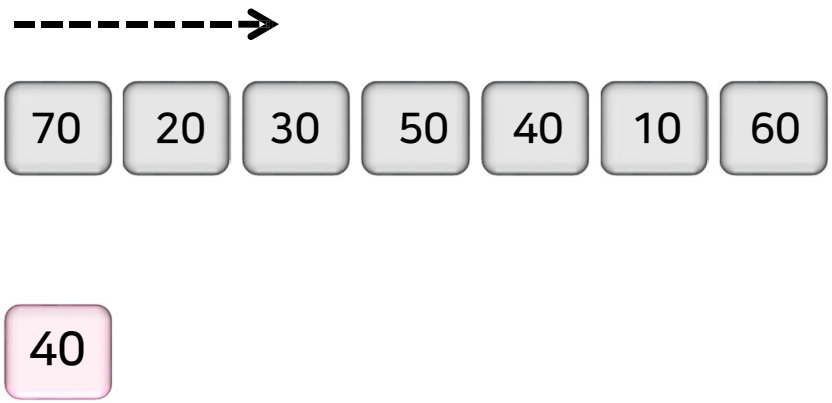
카드를 섞어서 뒤집어 놓았다. 이 카드 중에서 K를 찾아라!



순차 탐색

순차 탐색의 개념과 원리

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



순차 탐색 알고리즘

```
SequentialSearch (A[ ], n, x)
{
  for (i=0; i < n; i++) {
    if (A[i] == x) return i;
  }
  return -1;    //탐색키 x가 존재하지 않는 경우
}
```

입력: A[0..n-1] : 입력 배열
n : 입력 크기(탐색 대상 원소의 개수)
x : 탐색키
출력: 배열 A에서 x의 위치(인덱스)

순차 탐색에서의 삽입 연산

삽입할 원소 45


```
SequentialSearch_Insert (A[0..n-1], n, x)
{
  A[n] = x;
  return A, n+1;    //배열의 크기가 1 증가
}
```

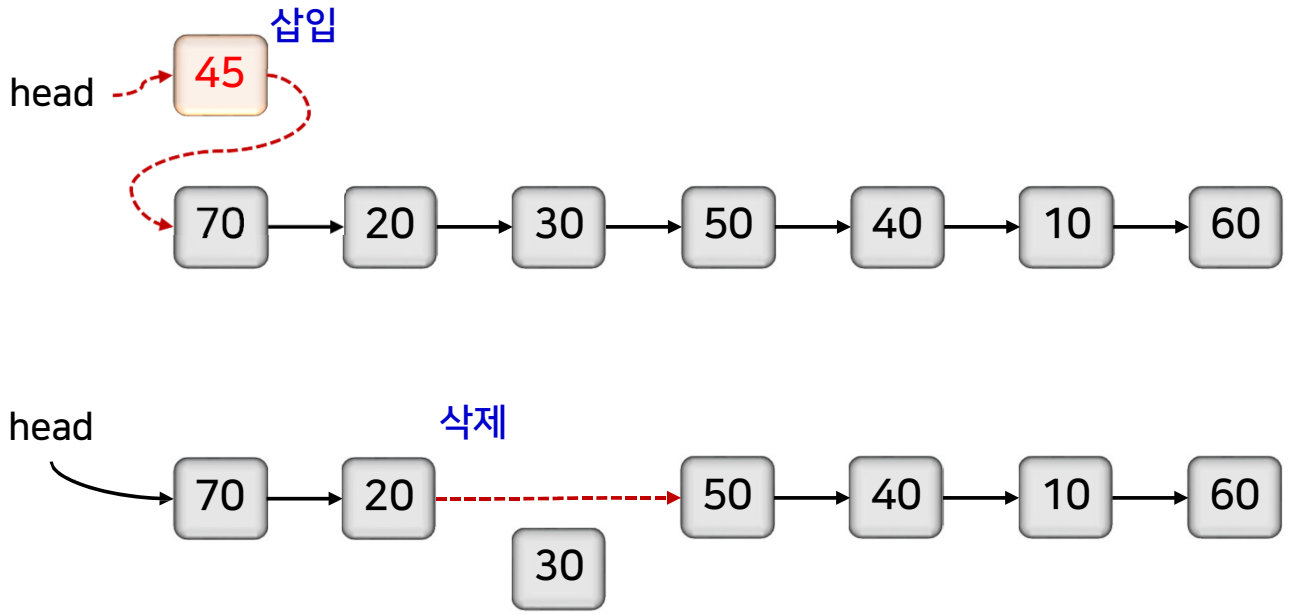
순차 탐색에서의 삭제 연산

삭제할 원소 30

Index = 2

```
SequentialSearch_Delete (A[0..n-1], n, x)
{
  Index = SequentialSearch(A,n,x);
  if (Index == -1) return A, n;
  A[Index] = A[n-1];
  return A, n-1;    //배열의 크기가 1 감소
}
```

삽입/삭제연산_연결리스트로 구현된 경우



순차 탐색의 성능 분석과 특징

- 성능 분석
  - 탐색 실패의 경우 → 항상 n번 비교
  - 탐색 성공의 경우 → 최소 1번, 평균 (n+1)/2번, 최대 n번 비교
  - 삽입 → O(1), 삭제 → O(n)

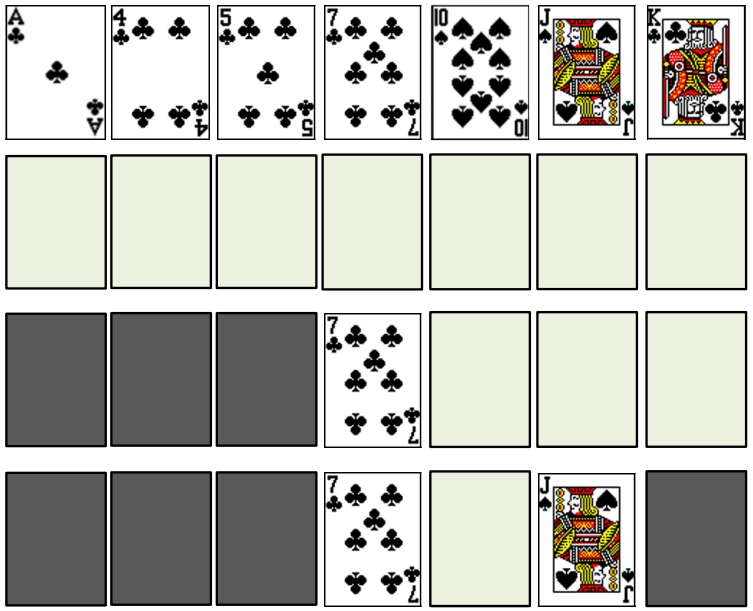
→ O(n)

- 특징
  - 모든 리스트에 적용 가능
    - 원소가 무순서로 연속해서 저장된 비정렬 데이터 탐색에 적합
  - 데이터가 큰 경우에는 부적합
    - 탐색과 삭제에 O(n) 시간을 요구

이진 탐색

이진 탐색의 개념과 원리

- 순서대로 정렬된 카드에서 원하는 카드 찾기



이진 탐색

이진 탐색

- 정렬된 리스트 형태로 주어진 원소들을 절반씩 줄여가면서 원하는 값을 가진 원소를 찾는 방법
  - 분할정복 방법
- 탐색 방법
  - 배열의 가운데 원소와 탐색키 x를 비교
    - 탐색키 = 가운데 원소 -> 탐색 성공
    - 탐색키 < 가운데 원소 -> '이진 탐색(크기 ½의 왼쪽 부분배열)' 순환 호출
    - 탐색키 > 가운데 원소 -> '이진 탐색(크기 ½의 오른쪽 부분배열)' 순환 호출

탐색을 반복할 때마다 대상 원소의 개수가 ½씩 감소

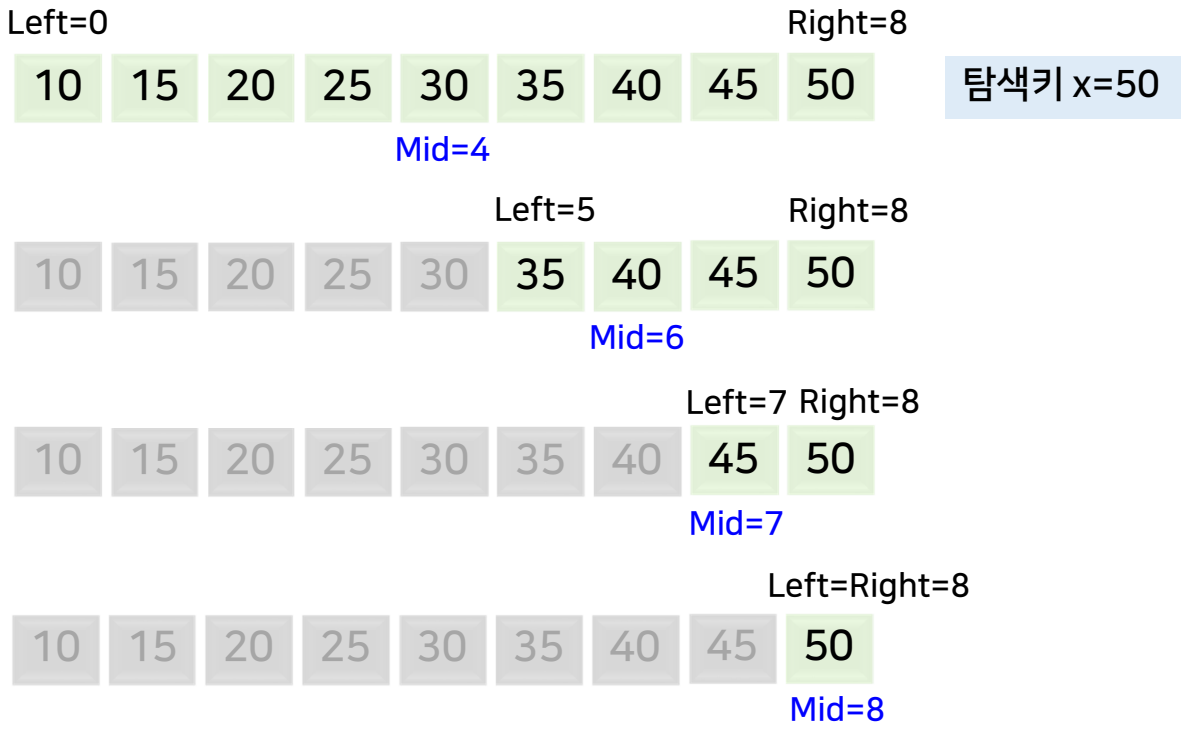
이진 탐색 알고리즘

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

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

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

이진 탐색 알고리즘의 적용 예



이진 탐색의 초기화 알고리즘

```
BinarySearch_Initialize ( A[], n)
{
   for (i=0; i<n-1; i++)
     if (A[i]>A[i+1]) {
        A = Sort(A, n);
        break;
     }
   return A;
}
```

O(n)

O(nlogn)

O(nlogn)

삽입 알고리즘

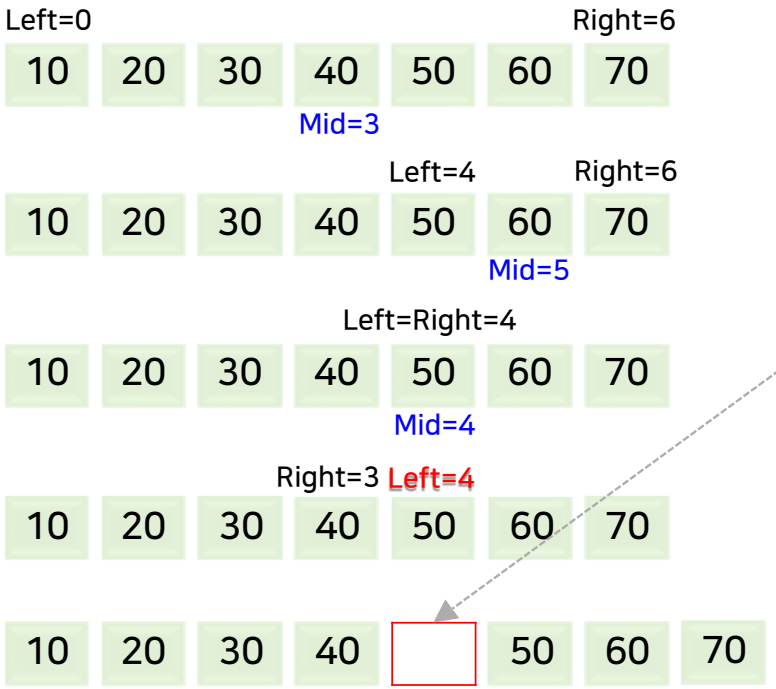
```
BinarySearch_Insert ( A[], n, x)
{
   Left = 0; Right = n-1;
   while (Left <= Right) {
      Mid = ⌊(Left+Right)/2⌋;
      if (x==A[Mid]) return A, n;
      else if (x<A[Mid]) Right = Mid-1;
            else Left = Mid+1;
   }
   for (i=n; i>Left; i--)
      A[i] = A[i-1];
   A[Left] = x;
   return A, n+1;
}
```

O(logn)

O(n)

O(n)

삽입 알고리즘의 적용 예


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

삭제 알고리즘

```
BinarySearch_Delete ( A[], n, x)
{
   Index = BinarySearch_Iteration(A, n, x);
   if (Index==-1) return A, n;
   for (i=Index; i<n-1; i++)
      A[i] = A[i+1];
   return A, n-1;
}
```

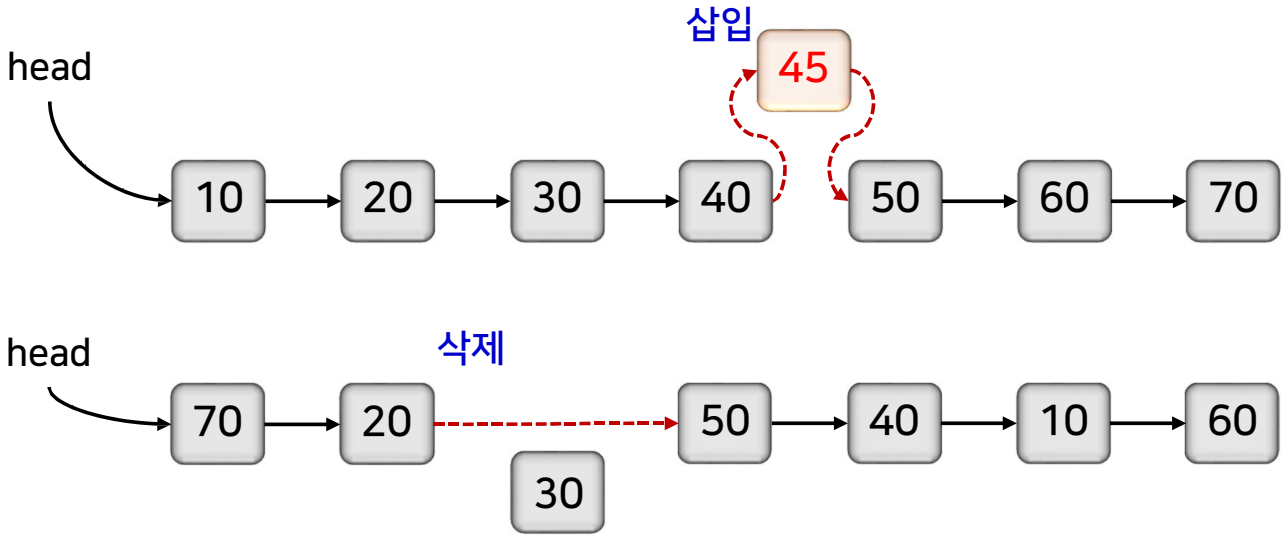
O(logn)

O(n)

O(n)

삭제할 원소 30

삽입/삭제연산_연결리스트로 구현된 경우



연결 리스트 구조에는 이진 탐색 자체가 불가능

이진 탐색의 성능 분석 및 특징

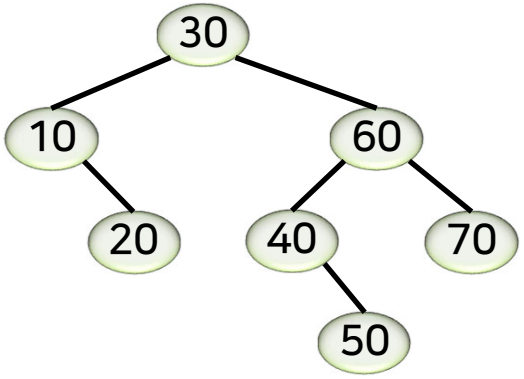
- 성능 분석
  - 탐색 → O(logn)
  - 초기화 → O(nlogn)
  - 삽입, 삭제 → O(n)

- 특징
  - 정렬된 리스트에 대해서만 적용 가능
  - 삽입과 삭제가 빈번한 경우에는 부적합
    - 삽입/삭제 후 입력 리스트의 정렬 상태를 유지하기 위해서 O(n)의 자료 이동이 필요

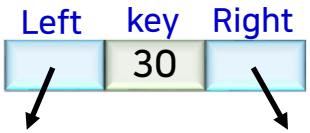
이진 탐색 트리

이진 탐색 트리의 개념과 원리

- 이진 탐색 트리 Binary Search Tree?
  - 이진 트리
  - 각 노드의 왼쪽 서브트리에 있는 모든 키 값은 그 노드의 키 값보다 작다.
  - 각 노드의 오른쪽 서브트리에 있는 모든 키 값은 그 노드의 키 값보다 크다.



노드의 구조



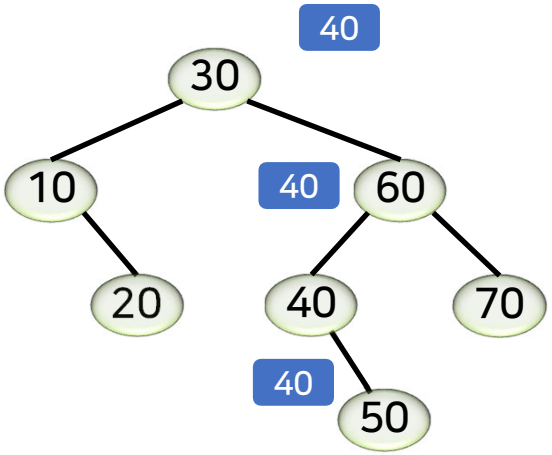

```
struct node {
  struct node *Left;
  int key;
  struct node *Right;
}
```

탐색 알고리즘

```
BST_Search (T, x)
{
   CNode = T의 루트 노드;
   while (CNode != Null)
      if (x==CNode.key) return CNode;
      else if (x<CNode.key) CNode = CNode.Left;
            else CNode = CNode.Right;
   return Null;
}
```

탐색 알고리즘의 적용 예

- 루트 노드로부터 시작해서 크기 관계에 따라 트리의 경로를 따라 내려가면서 탐색을 진행

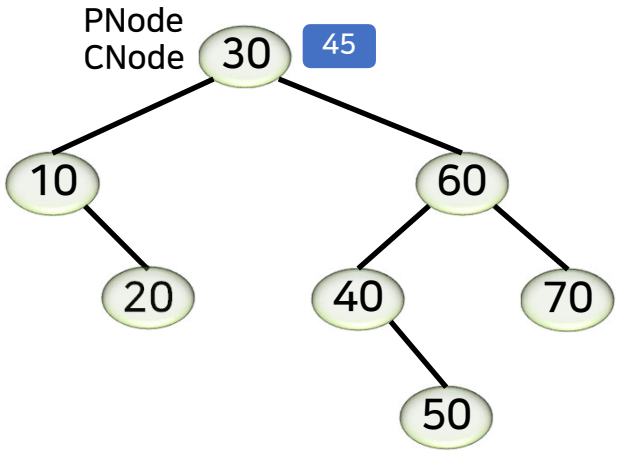


삽입 알고리즘

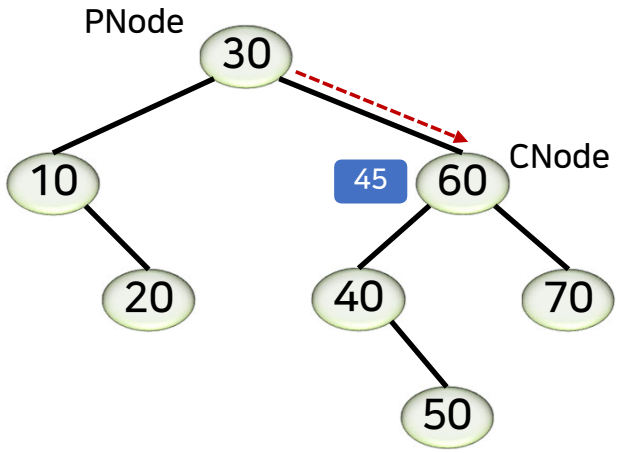
```
BST_Insert (T, x)
{
   PNode = CNode = T의 루트 노드;
   while (CNode != Null)
     if (x==CNode.key) return T;
     else {
         PNode = CNode;
         if (x<CNode.key) CNode = CNode.Left;
         else CNode = CNode.Right;
      }
   NewNode.key = x;
   if (x<PNode.key) PNode.Left = NewNode;
   else PNode.Right = NewNode;
   return T;
}
```

삽입 알고리즘의 적용 예

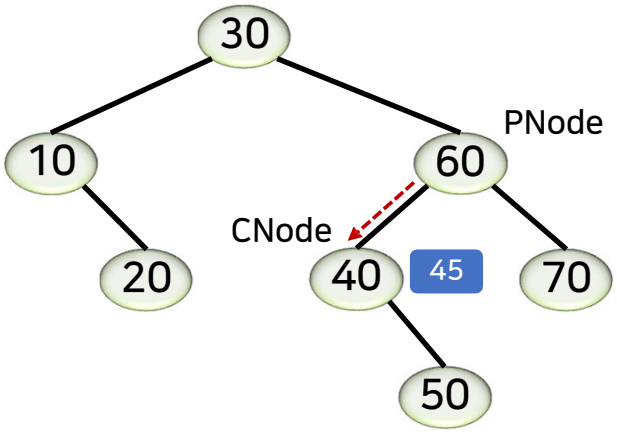
- 삽입할 원소를 탐색한 후, 탐색이 실패하면 해당 위치에 자식 노드로 추가



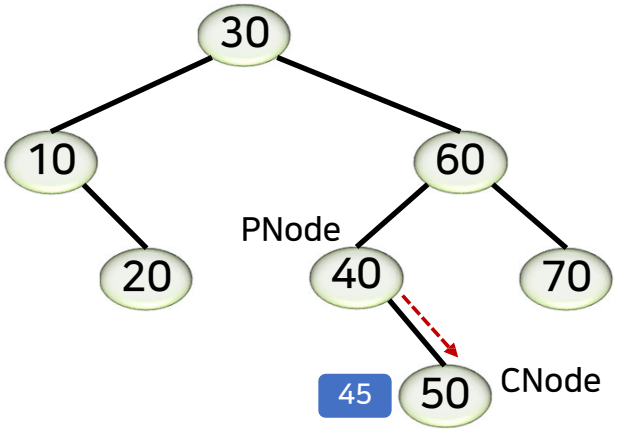
삽입 알고리즘의 적용 예



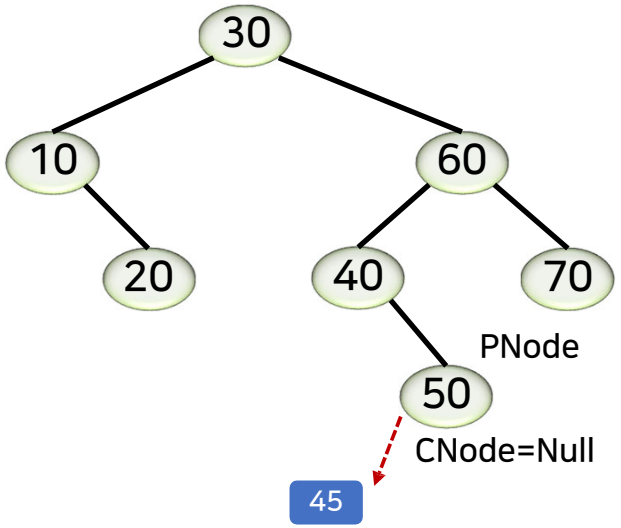
삽입 알고리즘의 적용 예



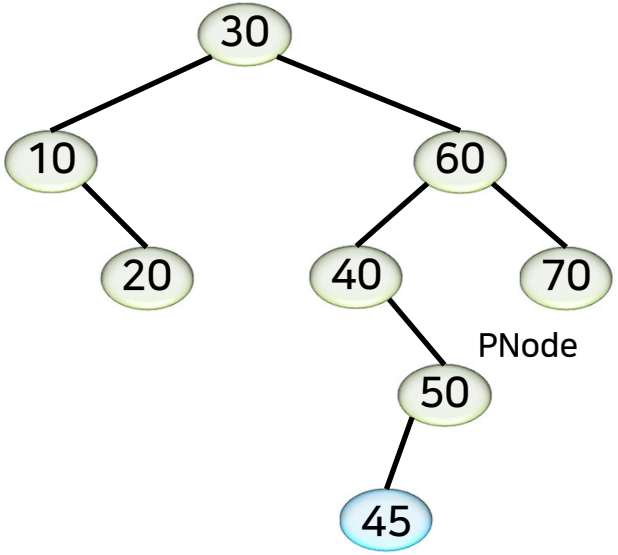
삽입 알고리즘의 적용 예



삽입 알고리즘의 적용 예

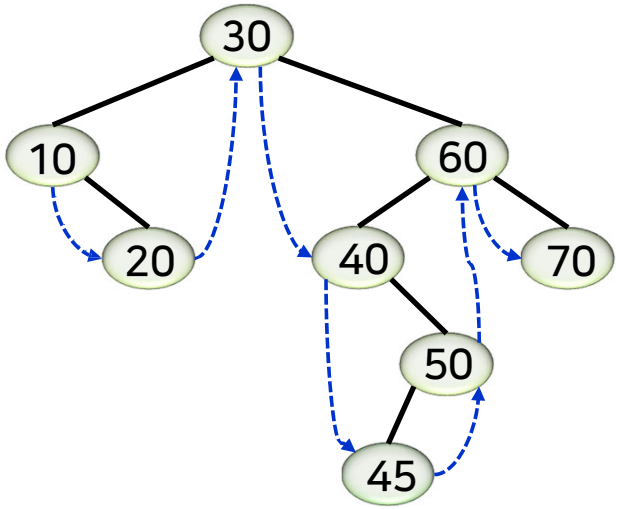


삽입 알고리즘의 적용 예



삭제 연산

- 후속자(successor, 계승자) 노드
  - 어떤 노드의 바로 다음 키 값을 갖는 노드



삭제 연산

- 삭제되는 노드의 자식 노드의 개수에 따라 3가지 경우로 구분

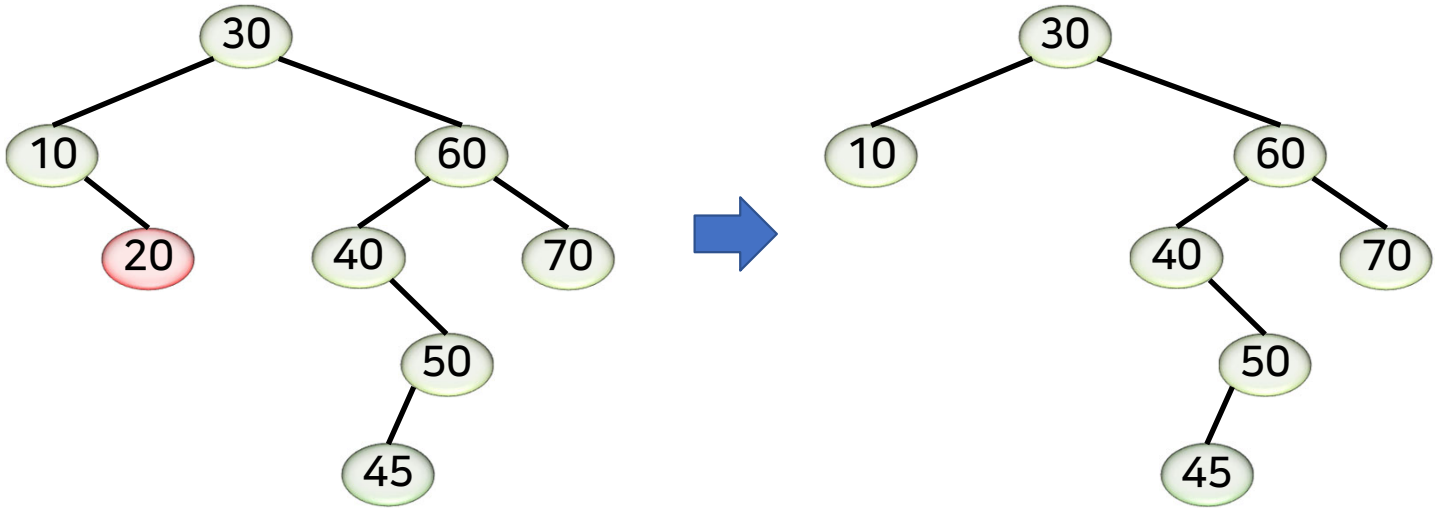
- 1. 자식 노드가 없는 경우(리프 노드의 경우)
  - 남은 노드의 위치 조절이 불필요

- 2. 자식 노드가 하나인 경우
  - 자식 노드를 삭제되는 노드의 위치로 올리면서 서브트리 전체도 따라 올린다.

- 3. 자식 노드가 두 개인 경우
  - 삭제되는 노드의 후속자 노드를 삭제되는 노드의 위치로 올리고,
  - 후속자 노드가 삭제되는 노드가 되어, 자식 노드의 개수(0, 1)에 따라 처리

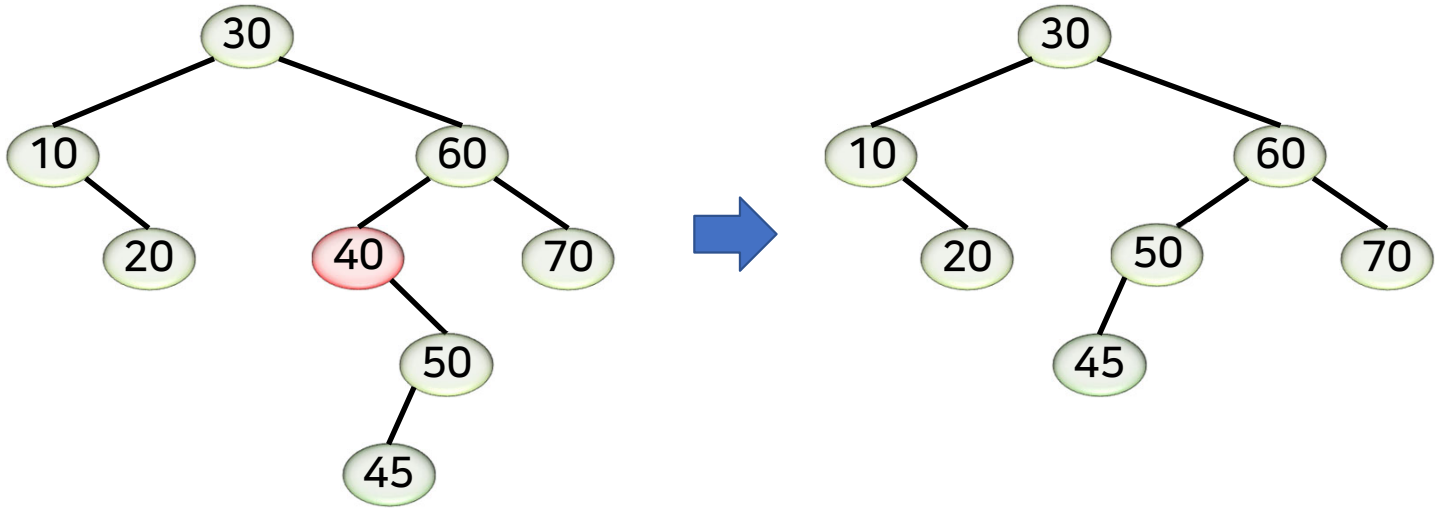
삭제 연산의 적용 예

- 노드 20 삭제



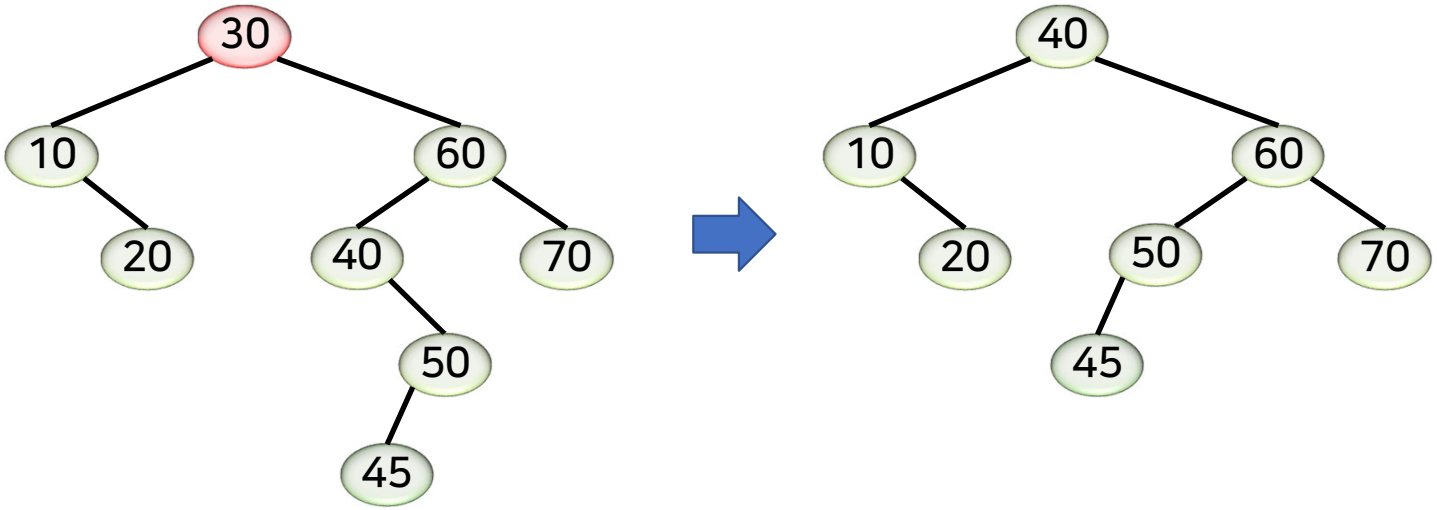
삭제 연산의 적용 예

- 노드 40 삭제



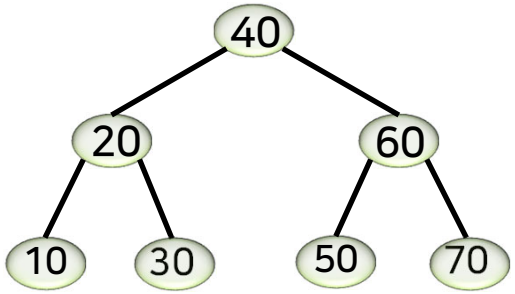
삭제 연산의 적용 예

- 노드 30 삭제



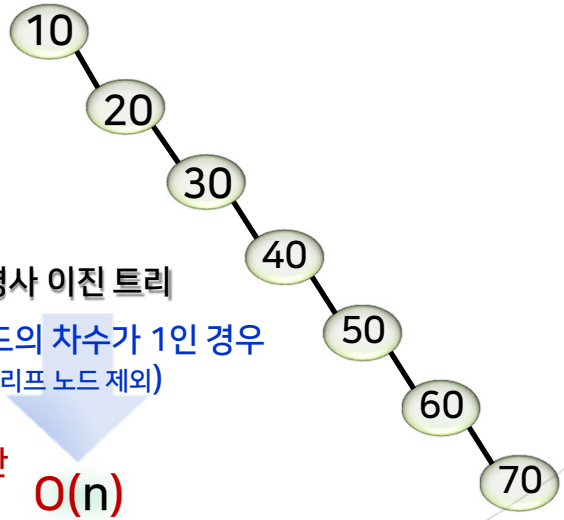
성능 분석

- 탐색, 삽입, 삭제의 시간 복잡도
  - 키 값을 비교하는 횟수에 비례 → 이진 트리의 높이가 h라면 O(h)



모든 노드의 차수가 2인 경우
(리프 노드 제외)

평균 수행 시간 O(logn)



경사 이진 트리

모든 노드의 차수가 1인 경우
(리프 노드 제외)

최악 수행 시간 O(n)

특징

- 삽입/삭제 연산 시 기존 노드의 이동이 거의 발생하지 않음
- 원소의 삽입, 삭제에 따라 경사 트리 형태가 될 수 있음
- 최악의 수행 시간 O(n)을 가짐
  - 경사 트리가 되지 않도록 균형을 유지해서 O(logn)을 보장

    → 균형 탐색 트리(흑적 트리, B-트리)

과제 안내

과제

- 이진 탐색 트리 구현하기
  - e-Class 업로드
- 양식 (한글, 워드,PDF -> 자유)
- 파일명 (이름_학번_전공)
  - 예) 최희석_2014182009_게임공학

- 질의 응답은 e-Class 질의응답 게시판에 남겨 주시길 바랍니다.