컴퓨터공학 All in One

C/C++ 문법, 자료구조 및 심화 프로젝트 (나동빈)
제 36강 - 이진 탐색 트리

학습 목표

이진 탐색 트리

1) 이진 탐색(Binary Search)이 항상 동작하도록 구현하여 탐색 속도를 극대화 시킨 자료구조를 이진 탐색 트리라고 합니다.

이진 탐색 트리

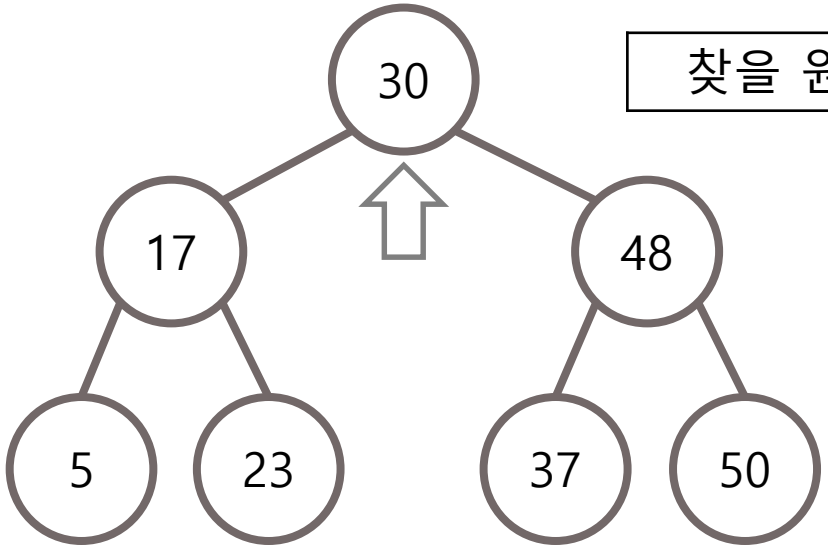
이진 탐색 트리

이진 탐색 트리에서는 항상 부모 노드가 왼쪽 자식보다는 크고, 오른쪽 자식보다는 작습니다.

이진 탐색 트리

이진 탐색 트리의 탐색

이진 탐색 트리에서는 항상 부모 노드가 왼쪽 자식보다는 크고, 오른쪽 자식보다는 작습니다.

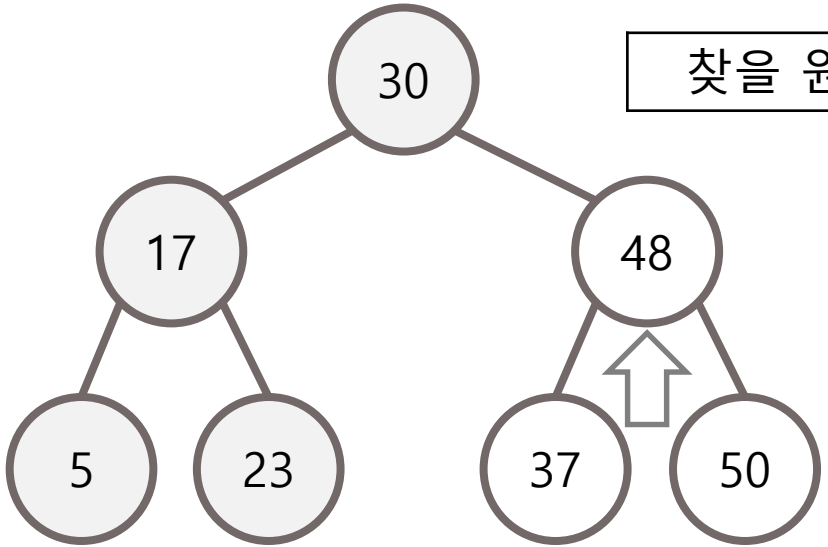


찾을 원소	37

이진 탐색 트리

이진 탐색 트리의 탐색

이진 탐색 트리에서는 항상 부모 노드가 왼쪽 자식보다는 크고, 오른쪽 자식보다는 작습니다.

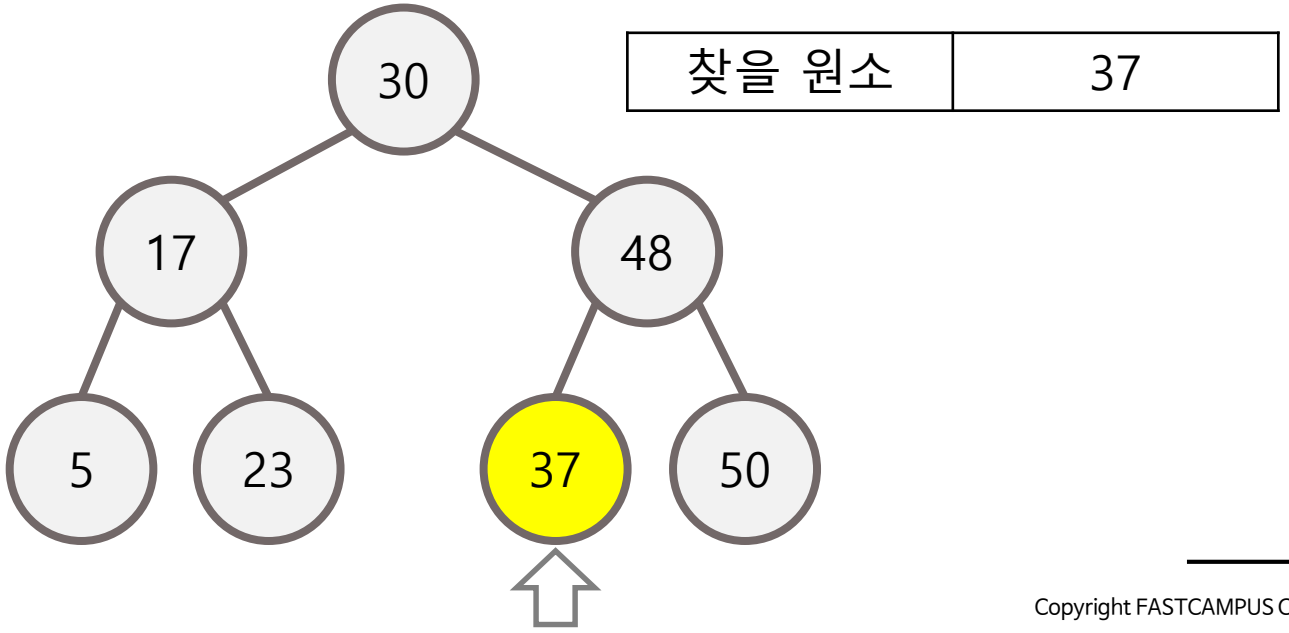


찾을 원소	37

이진 탐색 트리

이진 탐색 트리의 탐색

이진 탐색 트리에서는 항상 부모 노드가 왼쪽 자식보다는 크고, 오른쪽 자식보다는 작습니다.



이진 탐색 트리

이진 탐색 트리의 탐색

이진 탐색 트리(Binary Search Tree)에서는 한 번 확인할 때마다 보아야 하는 원소의 개수가 절반씩 줄어든다는 점에서 '완전 이진 트리'인 경우 탐색 시간에 $O(logN)$의 시간 복잡도를 가집니다.

이진 탐색 트리에서 탐색을 할 때는 찾고자 하는 값이 부모 노드보다 작을 경우 왼쪽 자식으로, 찾고자 하는 값이 부모 노드보다 클 경우 오른쪽 자식으로 이어 나가면서 방문합니다.

이진 탐색 트리

이진 탐색 트리의 정의

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int data;
  struct Node* leftChild;
  struct Node* rightChild;
} Node;
```

이진 탐색 트리

이진 탐색 트리의 삽입

```
Node* insertNode(Node* root, int data) {
  if (root == NULL) {
    root = (Node*)malloc(sizeof(Node));
    root->leftChild = root->rightChild = NULL;
    root->data = data;
    return root;
  }
  else {
    if (root->data > data) {
      root->leftChild = insertNode(root->leftChild, data);
    }
    else {
      root->rightChild = insertNode(root->rightChild, data);
    }
  }
  return root;
}
```

이진 탐색 트리

이진 탐색 트리의 탐색

```
Node* searchNode(Node* root, int data) {
  if (root == NULL) return NULL;
  if (root->data == data) return root;
  else if (root->data > data) return searchNode(root->leftChild, data);
  else return searchNode(root->rightChild, data);
}
```

Fast campus

이진 탐색 트리

이진 탐색 트리의 순회

```
void preorder(Node* root) {
  if (root == NULL) return;
  printf("%d ", root->data);
  preorder(root->leftChild);
  preorder(root->rightChild);
}
```

Fast campus

이진 탐색 트리

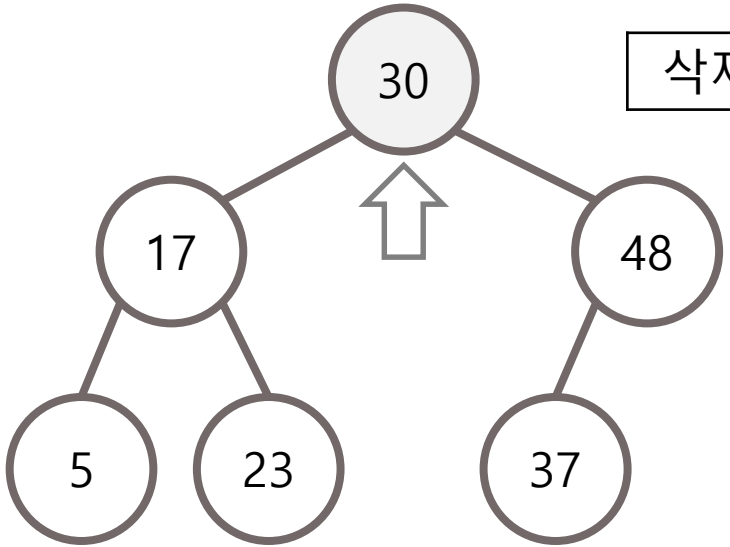
이진 탐색 트리 이용해보기

```
int main(void) {
  Node* root = NULL;
  root = insertNode(root, 30);
  root = insertNode(root, 17);
  root = insertNode(root, 48);
  root = insertNode(root, 5);
  root = insertNode(root, 23);
  root = insertNode(root, 37);
  root = insertNode(root, 50);
  preorder(root);
  system("pause");
}
```

이진 탐색 트리

이진 탐색 트리의 삭제 1) 자식이 없는 경우

삭제할 노드의 자식이 없는 경우 단순히 제거하면 됩니다.

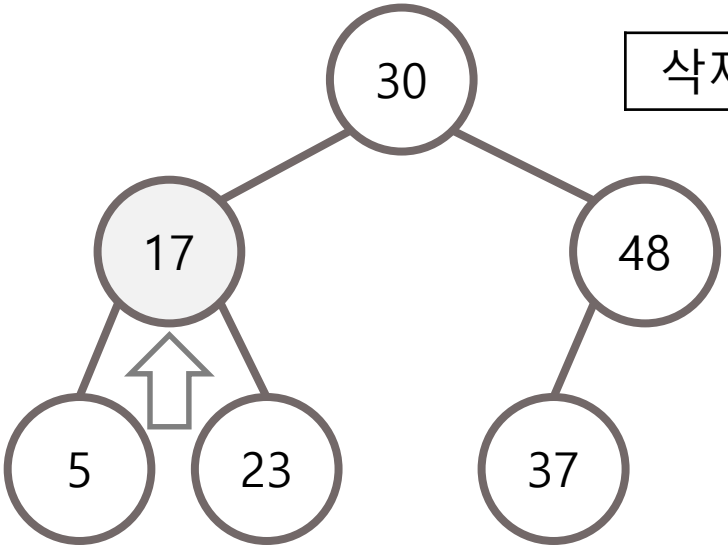


삭제할 원소	23

이진 탐색 트리

이진 탐색 트리의 삭제 1) 자식이 없는 경우

삭제할 노드의 자식이 없는 경우 단순히 제거하면 됩니다.

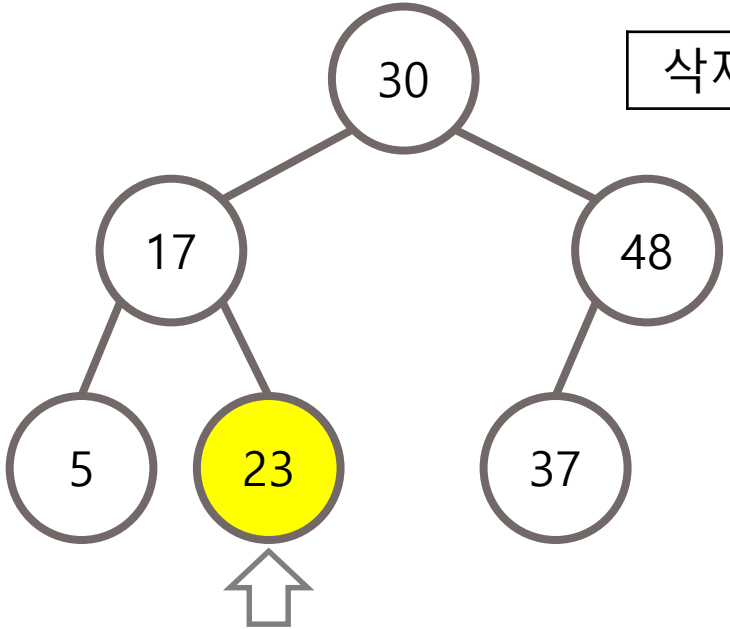


삭제할 원소	23

이진 탐색 트리

이진 탐색 트리의 삭제 1) 자식이 없는 경우

삭제할 노드의 자식이 없는 경우 단순히 제거하면 됩니다.



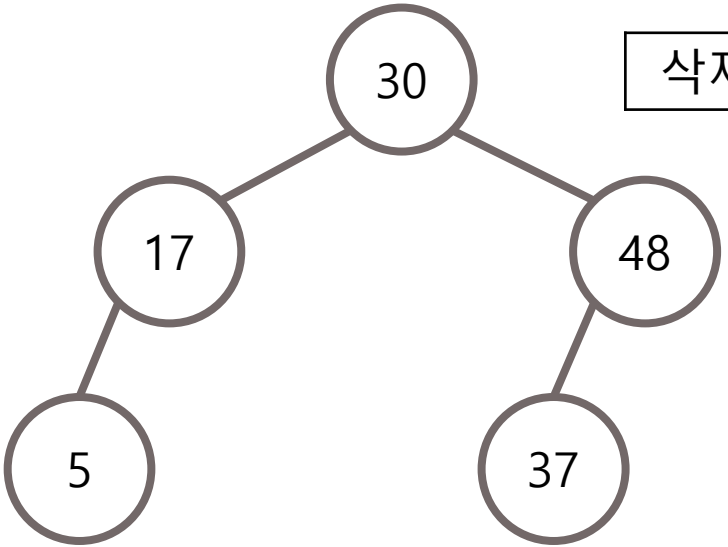
삭제할 원소	23

Fast campus

이진 탐색 트리

이진 탐색 트리의 삭제 1) 자식이 없는 경우

삭제할 노드의 자식이 없는 경우 단순히 제거하면 됩니다.

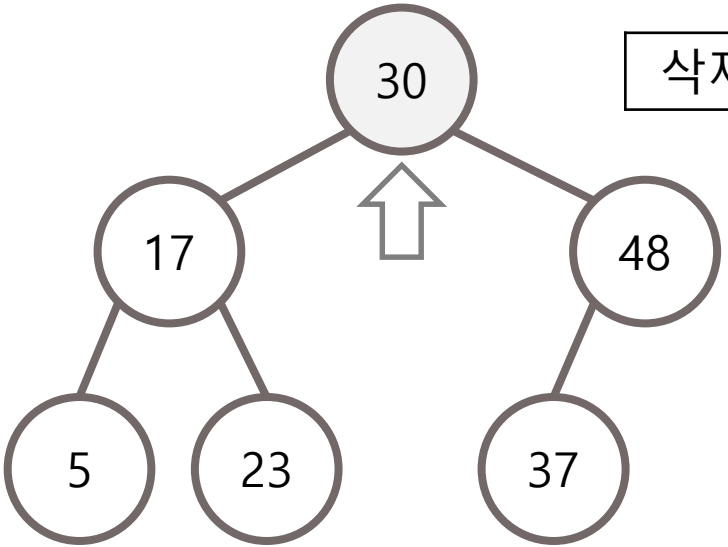


삭제할 원소	23

이진 탐색 트리

이진 탐색 트리의 삭제 2) 자식이 하나만 존재하는 경우

삭제할 노드의 자식이 하나만 존재하는 경우 삭제할 노드의 자리에 자식 노드를 넣습니다.

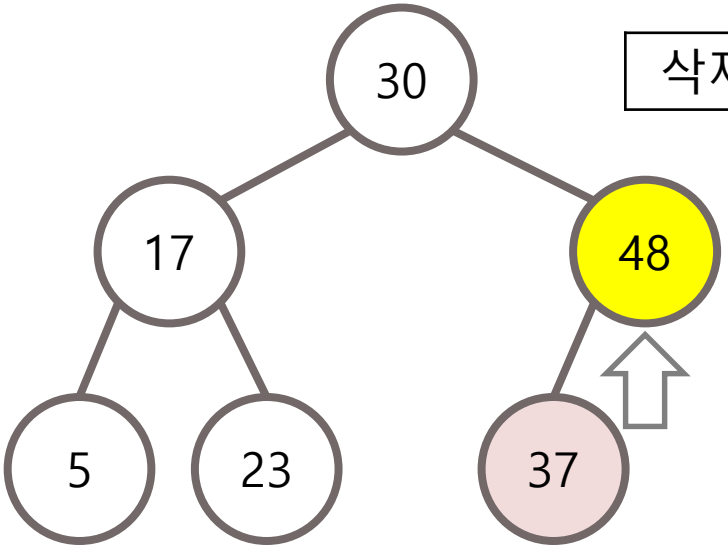


삭제할 원소	48

이진 탐색 트리

이진 탐색 트리의 삭제 2) 자식이 하나만 존재하는 경우

삭제할 노드의 자식이 하나만 존재하는 경우 삭제할 노드의 자리에 자식 노드를 넣습니다.

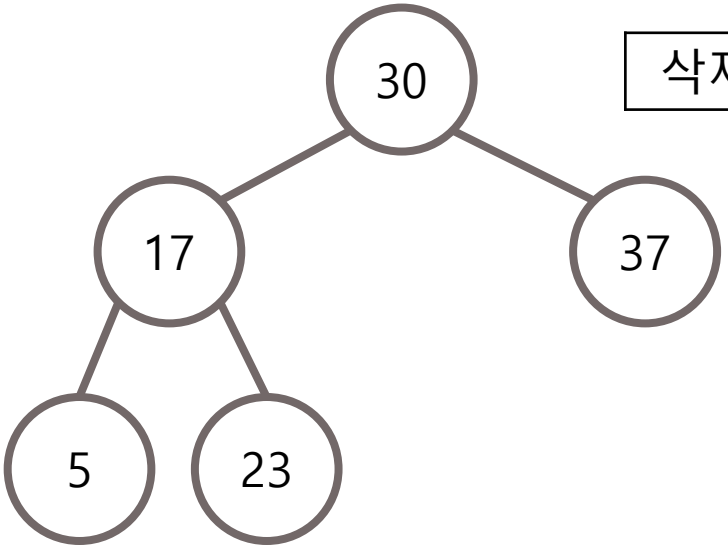


삭제할 원소	48

이진 탐색 트리

이진 탐색 트리의 삭제 2) 자식이 하나만 존재하는 경우

삭제할 노드의 자식이 하나만 존재하는 경우 삭제할 노드의 자리에 자식 노드를 넣습니다.

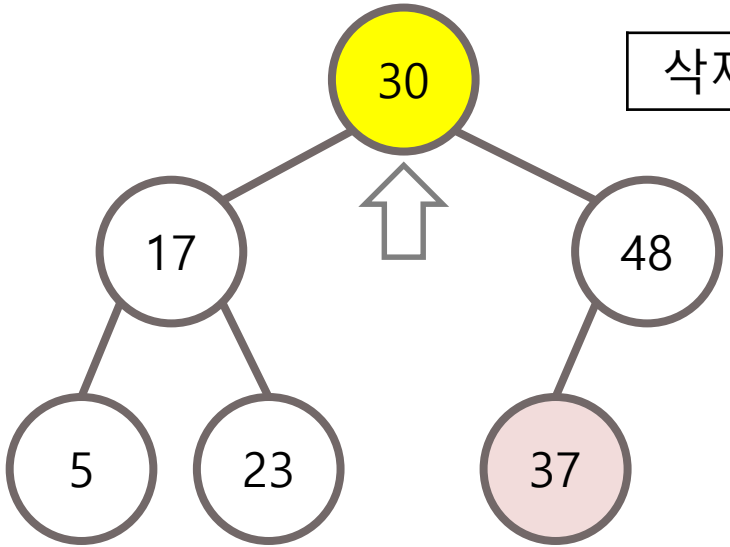


삭제할 원소	48

이진 탐색 트리

이진 탐색 트리의 삭제 3) 자식이 둘 다 존재하는 경우

자식이 둘 다 존재하는 경우 그 다음으로 삭제할 노드의 자리에 자기 다음으로 큰 노드를 넣습니다.

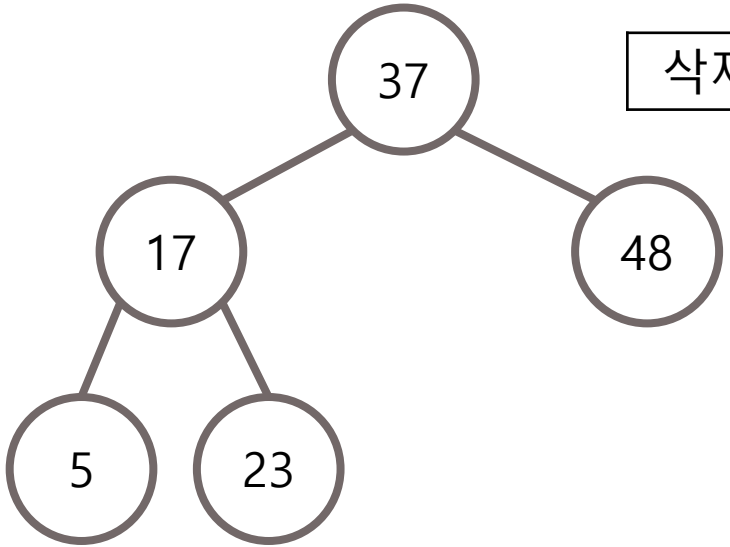


삭제할 원소	30

이진 탐색 트리

이진 탐색 트리의 삭제 3) 자식이 둘 다 존재하는 경우

자식이 둘 다 존재하는 경우 그 다음으로 삭제할 노드의 자리에 자기 다음으로 큰 노드를 넣습니다.



삭제할 원소	30

이진 탐색 트리

이진 탐색 트리의 가장 작은 원소 찾기 함수

```
Node* findMinNode(Node* root) {
  Node* node = root;
  while (node->leftChild != NULL) {
    node = node->leftChild;
  }
  return node;
}
```

이진 탐색 트리

이진 탐색 트리의 삭제 함수

```
Node* deleteNode(Node* root, int data) {
  Node* node = NULL;
  if (root == NULL) return NULL;
  if (root->data > data) root->leftChild = deleteNode(root->leftChild, data);
  else if (root->data < data) root->rightChild = deleteNode(root->rightChild, data);
  else {
    if (root->leftChild != NULL && root->rightChild != NULL) {
      node = findMinNode(root->rightChild);
      root->data = node->data;
      root->rightChild = deleteNode(root->rightChild, node->data);
    }
    else {
      node = (root->leftChild != NULL) ? root->leftChild : root->rightChild;
      free(root);
      return node;
    }
  }
  return root;
}
```

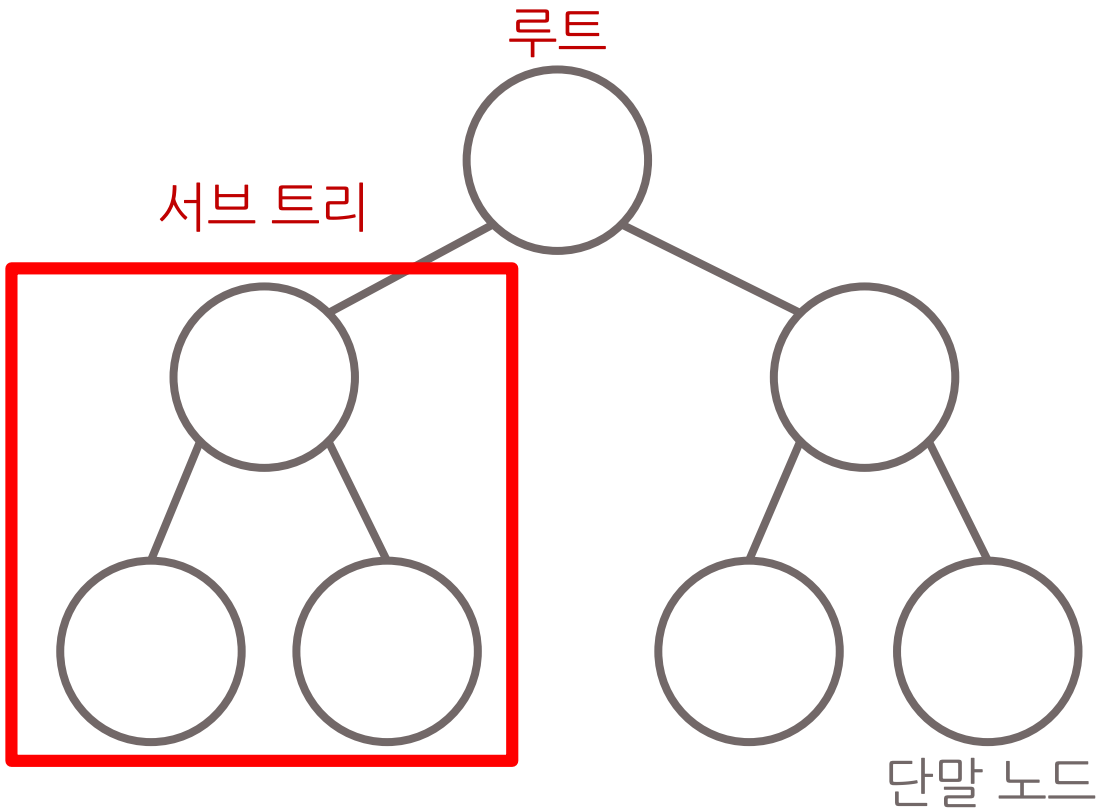
이진 탐색 트리

이진 탐색 트리 사용해보기

```
int main(void) {
  Node* root = NULL;
  root = insertNode(root, 30);
  root = insertNode(root, 17);
  root = insertNode(root, 48);
  root = insertNode(root, 5);
  root = insertNode(root, 23);
  root = insertNode(root, 37);
  root = insertNode(root, 50);
  root = deleteNode(root, 30);
  preorder(root);
  system("pause");
}
```

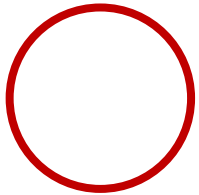
이진 탐색 트리

이진 탐색 트리의 성능을 최대로 끌어내기 위해서는 이진 탐색 트리가 완전 이진 트리에 가까워 질 수 있도록 설계해야 합니다.



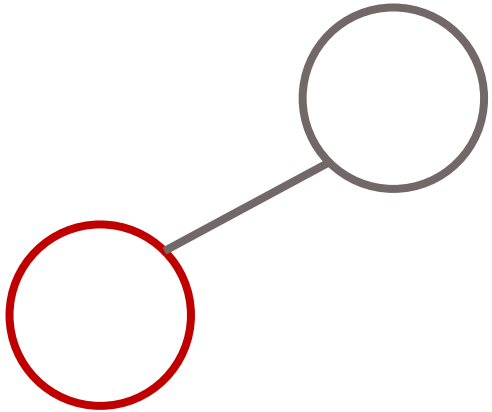
이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



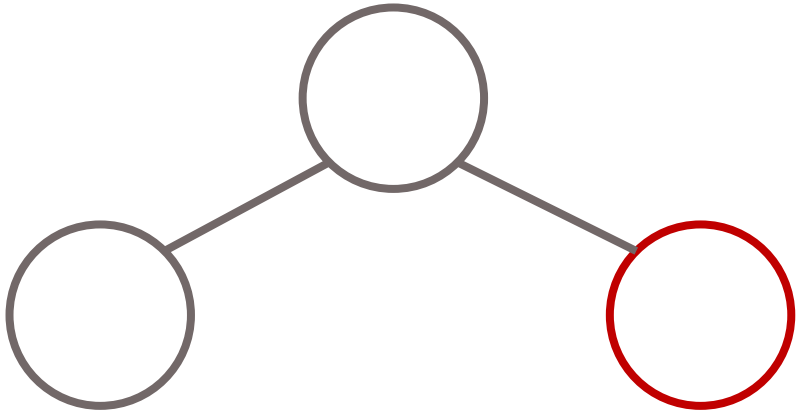
이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



이진 탐색 트리

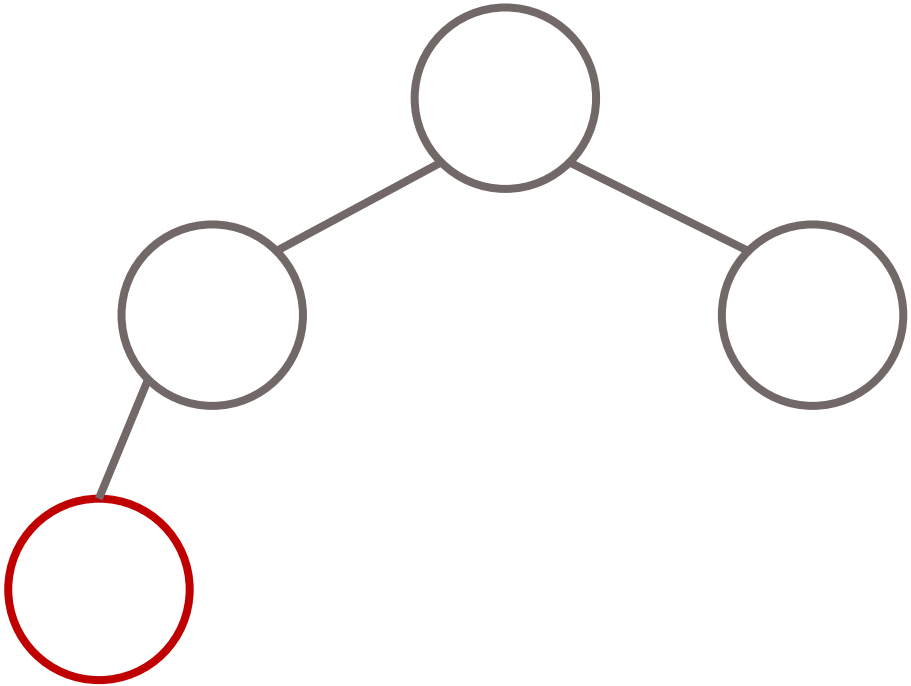
완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



Fast campus

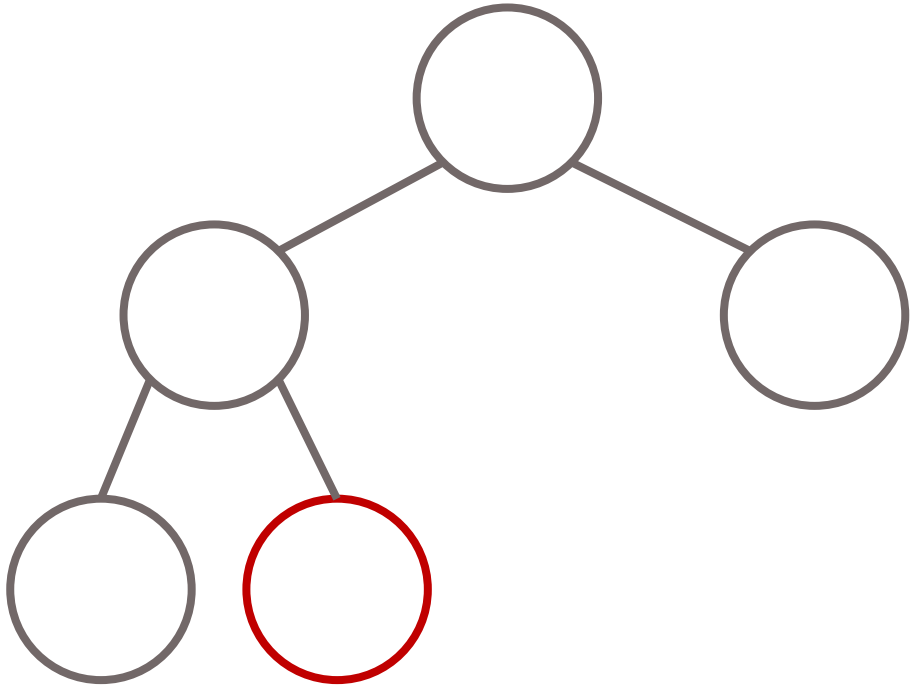
이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



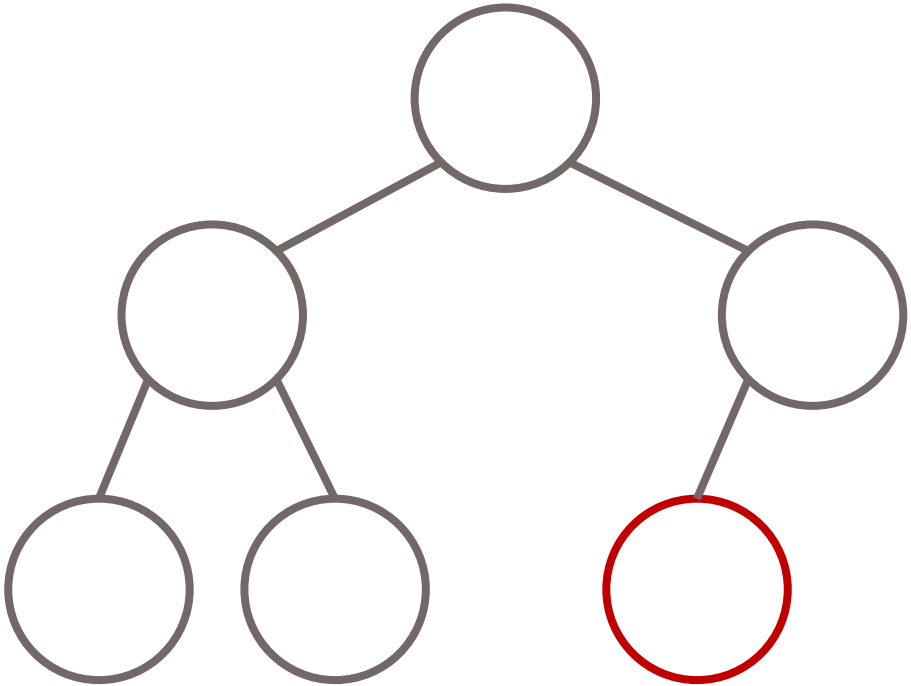
이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



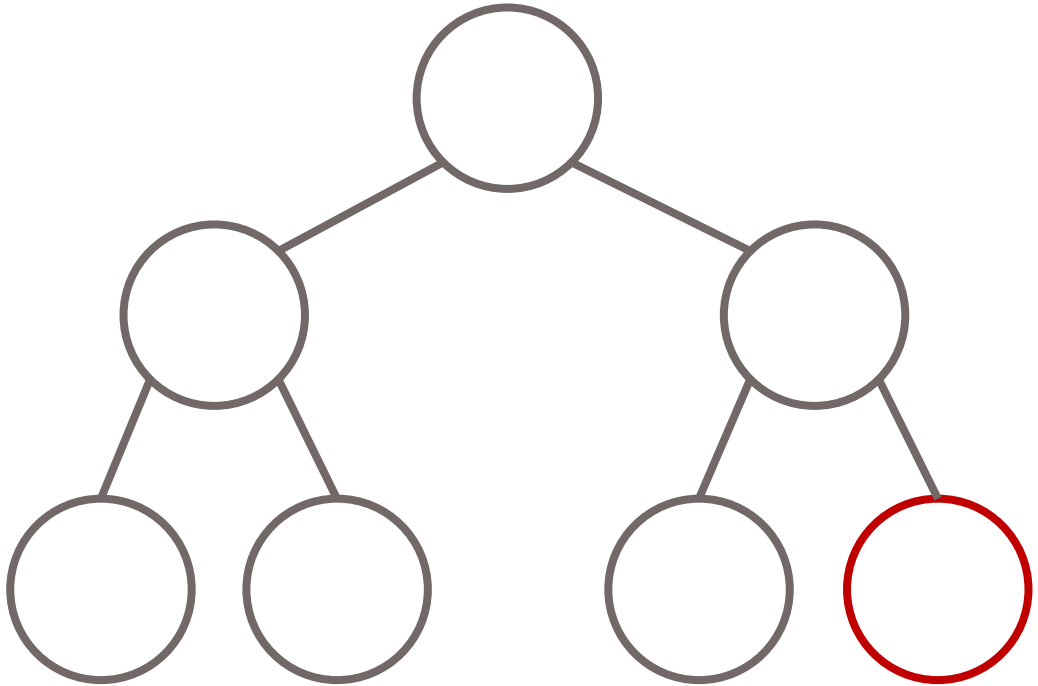
이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



이진 탐색 트리

완전 이진 트리(Complete Binary Tree)란 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차례대로 들어가는 구조의 이진 트리입니다.



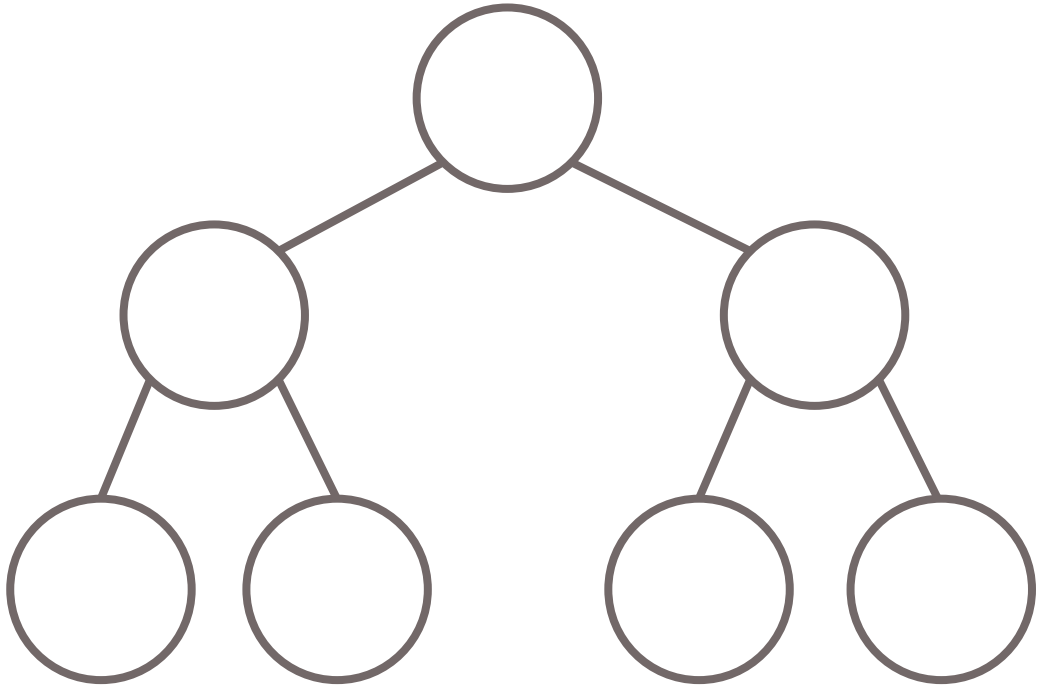
이진 탐색 트리

트리의 효율성

트리(Tree)를 사용하면 데이터를 처리함에 있어서 효율적입니다. 트리에서 데이터의 개수가 N개일 때 배열과 마찬가지로 O(N)의 공간만이 소요되며 삽입 및 삭제에 있어서 일반적인 경우 기존의 배열(Array)를 이용하는 방식보다 효율적입니다. 그래서 데이터베이스 등 대용량 저장 및 검색 자료구조로 많이 활용됩니다.

이진 탐색 트리

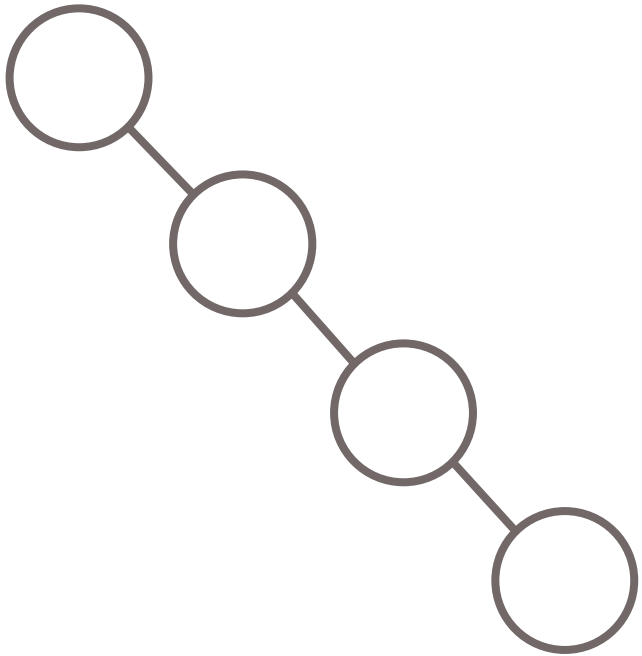
정상적으로 설계된 완전 이진 트리(Complete Binary Tree)에서는 어떠한 원소라도 탐색함에 있어서 $O(logN)$의 시간이 소요됩니다.



이진 탐색 트리

반면에 한쪽으로 치우친 편향 이진 트리(Skewed Binary Tree)의 경우 탐색에 있어 $O(N)$의 시간 복잡도가 형성되므로 기존의 배열(Array)을 사용하는 것보다 오히려 많은 공간과 시간이 낭비됩니다.

따라서 이진 트리(Binary Tree)를 만들 때는 트리의 균형이 맞도록 설정하는 것이 중요합니다.



배운 내용 정리하기

이진 탐색 트리

1) 편향 이진 트리(Skewed Binary Tree)의 경우 탐색에 있어 $O(N)$의 시간 복잡도를 가집니다.
2) 따라서 이진 탐색 트리를 최대한 완전 이진 트리의 형태를 유지할 수 있도록 해야 합니다.