[[자료구조] ProrityQueue(우선순위큐) - Java Collections Framework](https://dheldh77.tistory.com/entry/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-ProrityQueue%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%ED%81%90-Java-Collections-Framework)

**PriorityQueue(우선순위 큐)**

일반적으로 Queue(큐)는 선입선출(FIFO, First In First Out)의 성격을 가지고 있는 자료구조이지만, 우선순위 큐는 들어오는 순서에 상관없이 우선순위가 높은 원소부터 빠져나가는 자료구조이다.

 운영체제의 CPU 스케줄링 중 우선순위 스케줄링이 우선순위 큐를 이용한 예이다.

 java Collections Framework에서 우선순위 큐의 자세한 사용법은 아래 코드를 참조하자

package algorithm;

import java.util.\*;

public class priorityQueue {

public static void main(String[] args) {

PriorityQueue<Integer> q = new PriorityQueue<>();

// 데이터 저장

q.add(4);

q.add(3);

q.add(7);

q.add(9);

/ 데이터 조회

System.out.println(q.peek());

//데이터 출력

while(!q.isEmpty()) {

System.out.println(q.poll());

}

}

}

저장된 순서와 상관없이 데이터가 오름차순으로 출력된 것을 확인할 수 있다.

package algorithm;

import java.util.\*;

public class priorityQueue {

public static void main(String[] args) {

PriorityQueue<Integer> q = new PriorityQueue<>(Collections.reverseOrder());

// 데이터 저장

q.add(4);

q.add(3);

q.add(7);

q.add(9);

// 데이터 조회

System.out.println(q.peek());

//데이터 출력

while(!q.isEmpty()) {

System.out.println(q.poll());

}

}

}

 우선순위를 반전시키는 방법은 객체 생성 시 Collections.reverseOrder()를 사용해 정렬 기준을 반전시킨다.

 아래는 객체의 정렬기준에 따라 우선순위 큐에 저장하고 출력하는 예제이다.

package algorithm;

import java.util.\*;

public class priorityQueue {

public static class Process implements Comparable<Process>{

int time;

int priority;

Process(int time, int priority){

this.time = time;

this.priority = priority;

}

@Override

public int compareTo(Process p) {

if(this.priority < p.priority) return 1;

else return -1;

}

}

public static void main(String[] args) {

PriorityQueue<Process> q = new PriorityQueue<>(Collections.reverseOrder());

// 데이터 저장

q.add(new Process(1, 4));

q.add(new Process(2, 2));

q.add(new Process(3, 1));

q.add(new Process(4, 6));

//데이터 출력

while(!q.isEmpty()) {

Process p = q.poll();

System.out.println("priority : " + p.priority + " time " + p.time);

}

}

}

[[자료구조] HashMap(해쉬맵) - Java Collections Framework](https://dheldh77.tistory.com/entry/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-HashMap-java)

**HashMap(해쉬맵)**

HashMap이란Map인터페이스 중 하나로써, key와 Value값으로 묶어 데이터를 저장하는 자료구조이다.

 Hashing을 사용함으로써 많은 양의 데이터를 검색하는데 뛰어난 성능을 가지고 있다.

 java Collections Framework에서 해쉬맵의 자세한 사용법은 아래 코드를 참조하자

package algorithm;

import java.util.\*;

public class hashmap {

public static void main(String[] args) {

HashMap<String, Integer> map = new HashMap<>();

// 데이터 삽입

map.put("apple", 2018);

map.put("banana", 2011);

map.put("melon", 2013);

map.put("kiwi", 2015);

// 중복된 키값을 저장할 경우 이전에 저장된 key - vlaue 쌍 위에 새로 저장

map.put("kiwi", 2017);

// 데이터 삭제

map.remove("apple");

// 데이터 출력

// 해쉬맵은 데이터를 저장한 순서대로 출력하지 않는다.

for(String s : map.keySet()) {

System.out.println(s + " " + map.get(s));

}

}

}

해쉬맵일 사용함에 있어서 주의할 점은

키 값의 중복을 허용하지 않는다는 점이다. value는 중복이 허용되지만  key값을 중복 저장할 경우, 이전에 저장된 key값에 덮어씌워지게 된다.

 또한 해쉬맵은 데이터를 저장한 순서대로 출력되지 않는다.

 해쉬맵을 정렬은 key, value에 따라 정렬이 가능하고, value값을 기준으로 정렬하기 위해서 아래와 같이 별도의 메소드를 정의해 주어야 한다.

**Key값에 의한 정렬**

package algorithm;

import java.util.\*;

public class hashmap {

public static void main(String[] args) {

HashMap<Integer, String> map = new HashMap<>();

// 데이터 삽입

map.put(1, "apple");

map.put(2, "banana");

map.put(3, "melon");

map.put(4, "kiwi");

Iterator<Integer> it = map.keySet().iterator();

while(it.hasNext()) {

int key = it.next();

System.out.println(key + " " + map.get(key));

}

}

}

**Value값에 의한 정렬**

package algorithm;

import java.util.\*;

public class hashmap {

// 정렬 메소드 정의

public static List sortByValue(final Map map) {

List<String> list = new ArrayList<>();

list.addAll(map.keySet());

Collections.sort(list, new Comparator() {

public int compare(Object o1, Object o2) {

Object v1 = map.get(o1);

Object v2 = map.get(o2);

return((Comparable)v2).compareTo(v1);

}

});

Collections.reverse(list);

return list;

}

public static void main(String[] args) {

HashMap<String, Integer> map = new HashMap<>();

// 데이터 삽입

map.put("apple", 2018);

map.put("banana", 2011);

map.put("melon", 2013);

map.put("kiwi", 2015);

Iterator it = sortByValue(map).iterator();

while(it.hasNext()) {

String s = (String)it.next();

System.out.println(s + " " + map.get(s));

}

}

}

[이진 탐색 트리(BST, Binary Search Tree)](https://dheldh77.tistory.com/entry/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%ACBST-Binary-Search-Tree)

2020. 5. 23. 06:28

[BST](https://dheldh77.tistory.com/tag/BST)

트리에 대해서 설명해보라는 질문을 들었을 때,

생각나는 말은 Collections Framework의 트리맵, 정렬된 상태로 값이 저장, 중복을 허용하지 않는 것 밖에 기억나지 않았다.

특정 자료구조를 사용했을 때의 시간 복잡도나 내부적으로 어떻게 동작하는 지도 정확하게 말할 수 있게끔 공부해야겠다

이진 탐색 트리는 이진 트리로 만들어진 탐색을 위한 자료구조이다. 4가지 조건을 가진다.

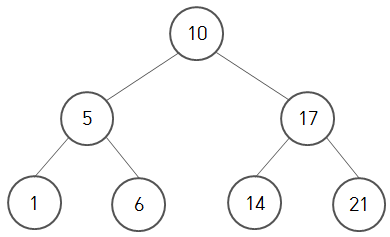
**1. 이진 탐색 트리의 4가지 조건**

**1) 모든 노드의 키는 유일하다.**

**2) 왼쪽 서브트리의 모든 키는 루트 노드의 키보다 작다**

**3) 오른쪽 서브트리의 모든 키는 루트 노드의 키보다 크다**

**4) 서브트리도 이진탐색트리이다.**



**2. 이진 탐색트리의 동작 방식**

**1) 탐색(Search)**

위의 이진 탐색 트리에서 14라는 키 값을 찾는다고 생각하면

루트노드 10과 비교했을 때, 14가 더 큼으로 오른쪽 서브 트리로 이동

오른쪽 서브 트리의 루트노드 17과 비교했을 때, 14가 더 작음으로 왼쪽 서브 트리로 이동

**2) 삽입(Inseart)**

삽입은 BST의 조건을 깨지 않기 위해서, 단말노드에 값을 추가해주는 방식이다.

위의 이진 탐색 트리에서 12를 추가한다고 하면,

루트노드 10과 비교했을 때, 12가 더 큼으로 오른쪽 서브 트리로 이동

오른쪽 서브 트리의 루트노드 17과 비교했을 때, 12가 더 작음으로 왼쪽 서브 트리로 이동

왼쪽 서브 트리의 루트노드 14와 비교했을 때, 12가 더 작음으로 왼쪽 서브트리의 루트 노드로 연결

**3) 삭제(Delete)**

삭제는 3가지 케이스로 나눌 수 있다

a. 삭제할 노드의 자식 노드가 없을 때,

단순히 부모 노드와의 링크만 끊어 주면 된다

자바는 GC가 알아서 메모리 회수해주므로 링크만 끊으면 되지만

C/C++은 free() 사용해서 메모리 회수해줘야한다

b. 삭제할 노드의 자식 노드가 하나 있을 때,

삭제되는 노드의 자식 노드와 부모 노드를 연결하도록 링크만 조절하면 된다

c. 삭제할 노드의 자식 노드가 두 개 있을 때,

삭제할 노드의 값을 삭제하고 매꿔줘야하는데,

중위 순회했을 때 삭제할 노드의 바로 앞의 값이나 뒤의 값으로 연결해야 BST의 조건을 위배하지 않는다

바로 앞의 값은 왼쪽 서브 트리에서 가장 큰 값(서브트리에서 계속 오른쪽 서브트리를 탐색했을 때 단말 노드)

바로 다음 값은 오른쪽 서브트리에서 가장 작은 값(서브트리에서 계속 왼쪽 서브트리를 탐색했을 때 단말 노드)

값을 매꾸고 나서는 중복을 허용하지 않기 때문에 이전 위치의 단말 노드는 지워준다

package algorithm;

import java.util.\*;

class BinarySearchTree {

Node root;

public static class Node{

int data;

Node left, right;

public Node(int data) {

this.data = data;

}

}

// 탐색

public Node search(Node root, int key) {

// 단말 노드이거나 찾는 노드를 찾았을 때

if(root == null || root.data == key) return root;

// 찾는 값이 더 작을 때 -> 왼쪽 서브 트리 탐색

if(root.data > key) return search(root.left, key);

// 찾는 값이 더 큰 경우 -> 오른쪽 서브 트리 탐색

return search(root.right, key);

}

// 삽입

public void insert(int data) {

root = insert(root, data);

}

private Node insert(Node root, int data) {

// 저장할 위치를 찾았을 때

if(root == null) {

root = new Node(data);

return root;

}

// 저장할 값이 루트 노드보다 작다면 왼쪽 서브 트리로 NULL이 나올 때까지 재귀

if(data < root.data)

root.left = insert(root.left, data);

// 저장할 값이 루트 노드보다 크다면 오른쪽 서브 트리로 NULL이 나올 때까지 재귀

else if(data > root.data)

root.right = insert(root.right, data);

return root;

}

// 삭제

public void delete(int data) {

root = delete(root, data);

}

private Node delete(Node root, int data) {

// 삭제할 노드를 못 찾았을 때

if(root == null) return root;

// 삭제할 데이터가 작으면 왼쪽 서브트리로 이동

if(data < root.data) root.left = delete(root.left, data);

// 삭제할 데이터가 크면 오른쪽 서브트리로 이동

else if(data > root.data) root.right = delete(root.right, data);

// 삭제할 데이터를 찾았을 때

else {

// 삭제할 데이터의 자식 노드가 없을 때

if(root.left == null && root.right == null) return null;

// 삭제할 데이터의 자식 노드가 하나만 있을 때

else if(root.left == null) return root.right;

else if(root.right == null) return root.left;

// 삭제할 데이터의 자식 노드가 두개 있을 때

root.data = findMin(root.right); // 오른쪽 서브 트리에서 가장 작은 값으로 대체

// 대체한 값의 원래 위치에 있는 노드 삭제

root.right = delete(root.right, root.data);

}

return root;

}

// 오른쪽 서브트리에서 가장 작은 값 삭제

int findMin(Node root) {

int min = root.data;

while(root.left != null) {

min = root.left.data;

root = root.left;

}

return min;

}

public void inorder() {

inorder(root);

System.out.println("");

}

private void inorder(Node root) {

if(root != null) {

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

}

}

public class BST{

public static void main(String[] args) {

BinarySearchTree tree= new BinarySearchTree();

tree.insert(4);

tree.insert(2);

tree.insert(1);

tree.insert(3);

tree.insert(6);

tree.insert(5);

tree.insert(7);

tree.inorder();

tree.delete(5);

tree.inorder();

}

}

**3. 한계점**

BST는 삽입, 삭제, 탐색과정에서 모두 트리의 높이만큼 탐색하기 때문에 O(logN)의 시간 복잡도를 가진다

 문제는 트리가 편향트리가 되어버렸을 때 결국 배열과 다름 없어지고 시간 복잡도는 O(N)이 된다