Chapter 7 정렬과 탐색

p.329, 330

정렬

- 오름차순 if 기준 > 비교대상 then 교환;
- 내림차순 if 기준 < 비교대상 then 교환;

정렬 알고리즘 비교

정렬 종류	평균	최악	
버블 정렬	$O(n^2)$	$O(n^2)$	최선 $O(n)$
선택 정렬	//	//	
삽입 정렬	//	//	
퀵 정렬	$O(n \log n)$	$O(n^2)$	→ 정렬되서 들어올 때
2-way merge 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	→ 공간 그냥 사용 (시간효율성↑, 공간효율성↓)
힙 정렬	//	//	→ 힙트리 이용
기수 정렬	$O(n)$	$O(n)$	→ 분배 이용, 비교x, 공간 복잡도↑

정렬의 종류

1. 버블 정렬

① 인접 데이터 비교

② n개 → n-1 회전

③ 순서가 맞으면 정렬 도중 멈출 수 있다.

ex)

50	20	30	10	40

5개 → 4회전

기준

50 1회전: 20→30→10→40→50

20→30→40 2회전: 20→10→30→40

20→30→40 3회전: 10→20→30

10 4회전: 10→20

전체 비교 횟수 (시간)

4번 비교 (n-1)

3번 비교 (n-2)

2번 비교 ⋮

1번 비교 1

$= \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$

$= O(n^2)$

1회전에 교환이 이루어지지 않을 때 최선 $O(n)$

P.330, 331

2. 선택 정렬 → 최대값(내림차순)

① 최소 값을 선택하여 첫 레코드와 교환, 이후 반복

② 이중 반복문이 적다. → 이름 1번씩 (회전 당)

ex)

50	20	30	10	40

현재 기준	회전	오름차순	비교횟수
10	1회전	10 20 30 50 40	n-1
20	2회전	20 30 50 40	n-2
30	3회전	30 50 40	⋮
40	4회전	40 50	1

$\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$

$\Rightarrow O(n^2)$

3. 삽입 정렬

① 첫 번째 레코드를 정의된 것으로 보고, 두 번째 레코드부터 키의 순서에 맞는 위치로 삽입시켜가며 정렬

② 최소 비교횟수 $O(n)$, 최대 비교횟수 $O(n^2)$

ex)

50	20	30	10	40

기준		고려 X	비교
50	1회전	20 50 | 30 10 40	20
30	2회전	20 30 50 | 10 40	20, 50
10	3회전	10 20 30 50 |	20, 30, 50
40	4회전	10 20 30 40 50	30, 50

$\Rightarrow$ 정렬되어 있을 때 효과적 $O(n)$

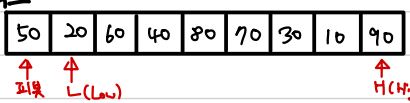
4. 퀵 정렬

① 비교에 의한 평균 수행시간이 제일 짧다.

② 순환 알고리즘 사용으로 스택 필요

③ 인덱스

ex)



H≥L { L이 가리키는 값 < 피봇이 가리키는 값 ⇒ L 1 증가 else 멈춤
H가 가리키는 값 > 피봇이 가리키는 값 ⇒ H 1 감소 else 멈춤 } 이후 교환

H<L ⇒ L, H 포인터가 만났을 때, 피봇을 H 포인터 값과 교환

(교환)

1회전 : [30 20 10 40] 50 [70 80 60 90]

피봇1 L1 H1 피봇2 L2 H2

2회전 : [[10 20] 30 [40]] 50 [[60] 70 [80 90]]

⇒ 정렬되어 있는 경우 매우 비효율적

p.331, 332

5. 힙정렬

① 최대 힙
- root에 최대값이 있으며, 모든 부노드는 자노드의 값보다 작지 않은 상태
- 우선순위가 가장 높은 데이터의 삭제 효율적

② 완전 이진트리에 값을 삽입하고 부노드가 자노드보다 작게 않도록 값을 교환

③ root의 최대값을 마지막 노드와 교환

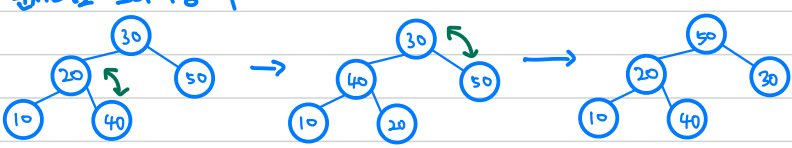
④ 나머지 값들로 최대 힙 구성

ex)

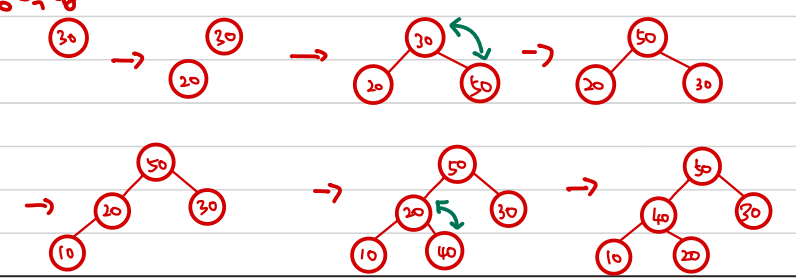
30	20	50	10	40

우선 순위 기준 - 완전 이진트리

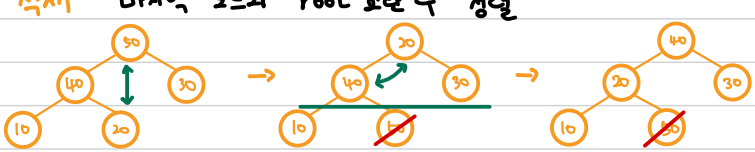
순서대로 트리 구성 ↓



공백힙



삭제 마지막 노드와 root 교환 후 정렬

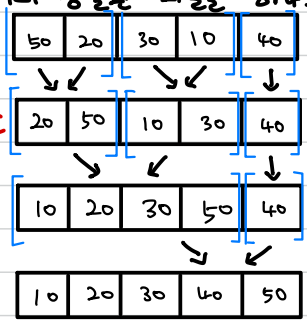


p.332

6. 2-way merge

① 2개의 정렬된 파일을 하나의 정렬된 파일로 병합

ex)



7. 기수 정렬

① 비교가 아닌 분배에 의한 정렬

② 레코드 내의 키들을 한 번에 하나씩 조사하여 그 값에 따라 큐에 분배하였다가 큐 결합, 다음 위치의 숫자로 다시 큐에 분배

=> 과정 반복

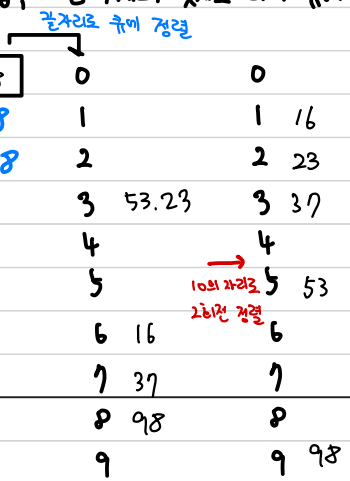
ex)

53	23	37	16	98

1회전 53 23 16 37 98

2회전 16 23 37 53 98

=> 비교를 하지 않는다.



p.335, 336

탐색

n보다 낮을 수 없다.
임의의 값 찾기

탐색 방식	시간 복잡도	
순차 탐색	O(n)	정렬되지 안아도됨
제미 탐색-이진 탐색	$O(\log_2 n)$	정렬 필수, 배열
이진 탐색 트리	균형적 : $O(\log_2 n)$, 불균형적 : O(n) (→ 사향)	
AVL 트리	$O(\log_2 n)$	이진트리 변형
3차 B-트리	$O(\log_3 n) \sim O(\log_2 n)$	차수 무조건 3
해싱	완전 해싱 : O(1), 충돌발생 : O(n)	

1. 순차탐색 O(n)
① 하나의 레코드씩 차례로 비교
② 정렬되지 않은 파일도 탐색 가능

2. 이진탐색
① 파일이 반드시 정렬, 배열, 고정 데이터에 적합
② mid = $\frac{low+high}{2}$ => 한 번의 탐색으로 탐색 대상이 반으로 감소
③ 최악일 시 평균보다 1회만 더 수행하면 되므로 레코드 AP 효과적

ex)

10	20	40	50	60	70	90	100
0	1	2	3	4	5	6	7

정렬, 배열

※ 70찾기
① 중간 찾기 : {0+7} / 2 = 3.5
(버리기준 : 내림 => [3] = 50 → 큰 범위 선정
② 큰 범위 중간 : {4+7} / 2 = 5.5 => 5
[5] = 70

3	2	3	1	3	2	3	4
0	1	2	3	4	5	6	7

탐색 성공 횟수

$\frac{(\text{첫범위}+\text{끝범위})}{2}$ = 탐색

p.336,337

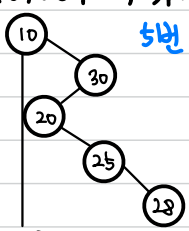
3. 이진 탐색 트리 정렬되어 있지 않음

★ ① 모든 원소는 동일하지 않은 키를 가진다. 오직 다른 수

② 왼쪽 서브트리에 있는 키들은 루트의 키보다 작다.

③ 오른쪽 // // 크다.

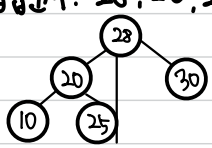
④ 삽입순서 : 10, 30, 20, 25, 28



5번 이내

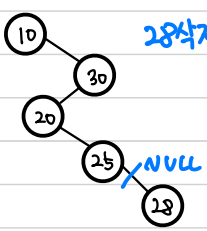
불균형 ⇒ AVL, B트리

삽입순서: 28, 20, 25, 10, 30

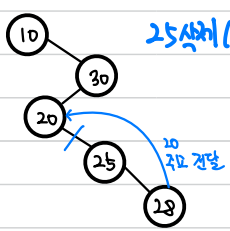


3번 이내

⑤ 삭제



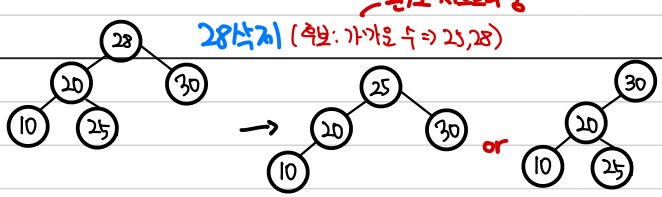
28삭제 (단말노드)



25삭제 (비단말)

왼/오 서브트리 중

28삭제 (주의: 가까운 수 ⇒ 25, 28)



⑥ 탐색 연산

- 주어진 탐색 키 값과 현재의 루트 노드의 키 값 비교
- 작으면 왼쪽 서브트리로, 크면 오른쪽 서브트리로 이동

p.338

4. AVL 트리

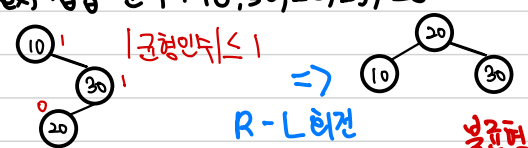
① 항상 균형을 유지하는 이진 탐색 트리

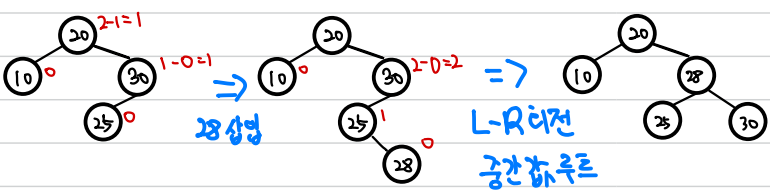
② 삽입, 삭제가 일어날 때마다 트리의 균형 상태를 점검하고 균형이 깨지면 트리 모습을 변형함으로써 항상 균형 유지

③ 균형인수 (왼쪽 서브트리 높이 − 오른쪽 서브트리 높이)가 항상 0, ±1을 유지한다. ⇒ 단말은 항상 0

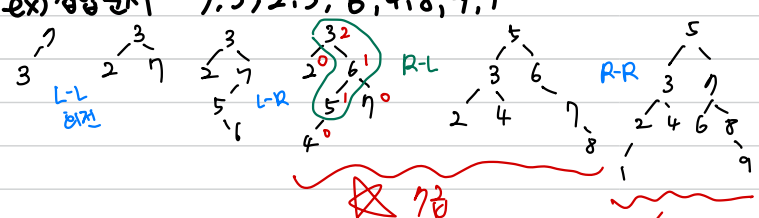
④ AVL 트리의 탐색 시간 복잡도는 항상 $O(\log_2 n)$이다.

ex) 삽입 순서 : 10, 30, 20, 25, 28





ex) 삽입 순서 7, 3, 2, 5, 6, 4, 8, 9, 1

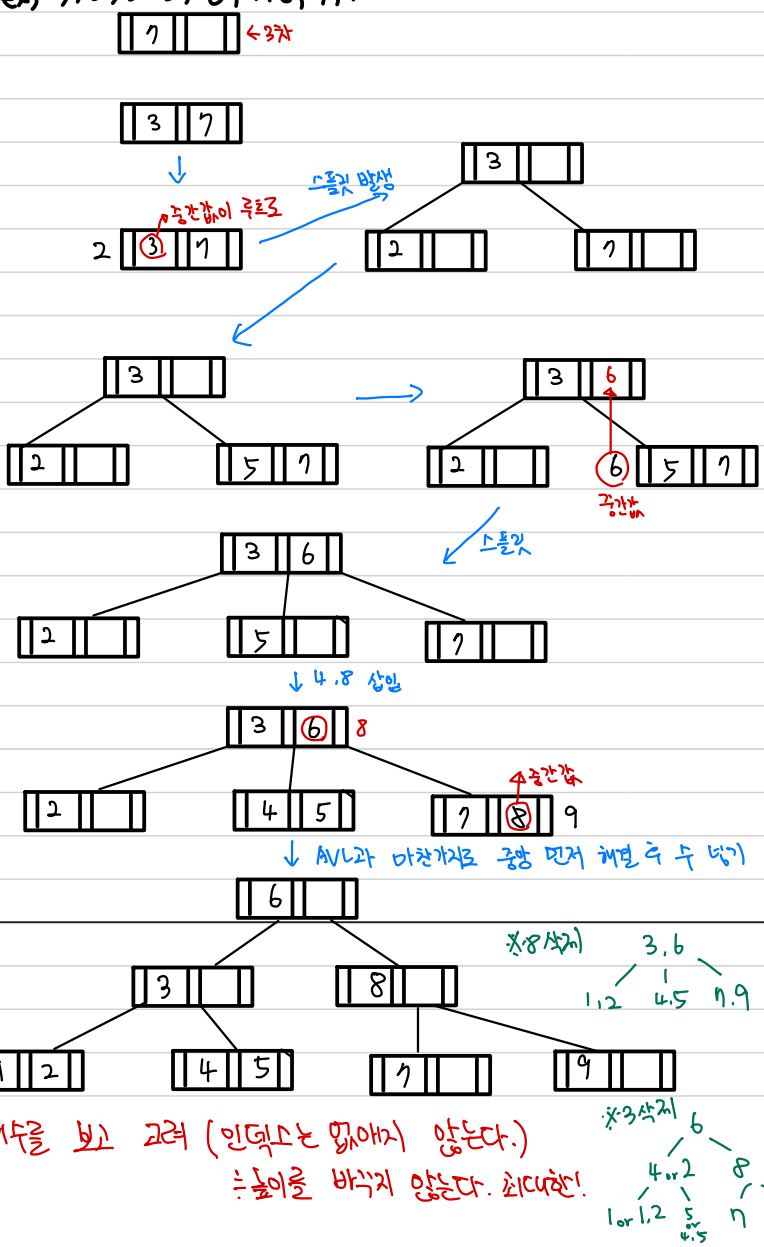


p.338

5. 차수 3인 B-트리 모든차수

ex) 7, 3, 2, 5, 6, 4, 8, 9, 1

동적 (삽입, 삭제 빈번히)인덱스



삭제 키의 개수를 보고 고려 (인덱스는 없애지 않는다.)

∴ 높이를 바꾸지 않는다. 최대한!

p.338

★ 차수가 m인 B-트리

공백이거나 다음 성질들을 만족하는 m원 탐색트리

① 루트 노드는 적어도 2개의 자식을 갖는다.

② 루트 노드와 외부노드를 제외한 모든 노드는 적어도 $\lceil m/2 \rceil$ 개의 자식을 갖는다. (올림)

③ 모든 외부 노드들은 같은 레벨에 있다.

(외부 노드: 단말노드 밑에 그리는 노드 (cf. 외부경로 길이 구하기))

P.340, 341

6. 해싱

① 탐색 목표가 아닌 다른 레코드의 키 값과 비교할 필요가 없는 탐색방법

② 시간 복잡도 : O(1)

③ 파일 내의 각 키에 대응하는 인덱스로 작성된 해시 표에서 단 한 번의 접근으로 레코드 탐색 방법

④ 충돌 발생 시 시간 복잡도 : O(n)

키 → 해시함수 → 해시 주소

제산법

서로다른 키 → 같은주소 (충돌) ⇒ 오버플로우 발생

골든박스

				해시테이블
9 →	%7	→	2	0
16 →	%7	→	2	1
2 →	%7	→	2	2
6 →	%7	→	6	3
20 →	%7	→	6	4
				5
				6

⇒ 덧장 해결

- 제산법 : % 나머지 연산을 통해 해시주소로 이용
- 제곱법 : 키값을 제곱하여 중간부분 추출하여 이용
- 폴딩법 : 키 값이 길기 때문에 일정부분 분할하여 그 값을 더한 것으로 이용
- 자리수 분석법 : 키 특성이나 분포가 잘 알려져 있을 때

p.342

오버플로우 해결방법

1. 개방 주소법

⇒ 해시 표 안에서 비어 있는 키의 해시 주소 버켓을 이미 다른 키가 차지하고 있을 때, 가까운 다른 해시 버킷의 버켓에 들어가는 것을 허용

① 선형 조사법 : overflow 발생 위치 +1, +2 …

↳ 제 1 덜집 현상 : 키 값들이 비는 한 곳에 모이는 현상

② 이차 조사법 : overflow 발생 위치 $+1^2, +2^2$ …

③ 이중 해싱 : overflow 발생 위치 + 다른 해시함수 적용 반복

…

2. 폐쇄 주소법

⇒ 오버플로가 발생해도 새로운 해시주소를 구하지 않는다.

① 연쇄방법 : 포인터를 이용하여 같은 해시 주소를 갖는 레코드들을 연결리스트로 만든다.

키 → 해시함수 → 해시 주소 (계산법)

서로다른 키 ⇒ 같은주소 (충돌) ⇒ 오버플로우 발생

해시테이블 (꽉찬 버킷)

오버플로우 해결

새로운 해시함수 ⇒ $h_2() = 5 - (k \bmod 5)$ (5보다 작은 수)

키	해시함수	해시 주소	해시테이블	① 선형조사	② 이차조사	③ 이중해싱
9	%7	2	0			
16	%7	2	1			
2	%7	2	2	2	2	2
6	%7	6	3	(2+1)%7=3	$(2+1^2)$%7=3	2+(5-(16%5))=6 ⇒ %7=6
20	%7	6	4	(2+2)%7=4	$(2+2^2)$%7=6	2+(5-(2%5))=5 ⇒ %7=5
			5	6	$(6+1^2)$%7=0	6+(5-(6%5))=10 ⇒ %7=3
			6	(6+1)%7=0	$(6+2^2)$%7=~~2~~ $(6+3^2)$%7=1	6+(5-(20%5))=11 ⇒ %7=4

④ 연쇄방법