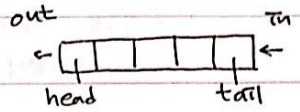
DATE. NO.

0122. (12강 11:16)

1. 배열 (Array) : 아주아주 기본적인 데이터구조

- 역할 : 데이터 나열, 이 때 데이터는 인덱스와 대응됨.
- int[], String[] (int[] list = {1, 2, 3};)
- 왜? ① 같은 타입의 데이터 효율적 관리
  ② " 데이터 순차적 저장 (→ 물리적으로 연결된 공간)
- 장점? ① 빠른 접근 가능 (인덱스 번호 활용) - 순차접근X
- 단점? ~~①~~ ① 미리 구조의 크기 설정이 필요 → 데이터 추가가 어렵다
  ② 데이터 삭제 후에도 빈공간 → 공간 낭비 or 당겨다야함.
- JAVA 2차원 배열 : int[][] list = new int[3][2];

2. 큐 (Queue)



- FIFO
- Index 같은 따로 존재하지 않는듯. ('꺼내라', '넣어라') Dequeue Enqueue
- 꺼낼 때 자동으로 큐에서 삭제, 한 칸씩 앞으로 당겨짐.
- Queue □ = new LinkedList(); - 링크드리스트로 구현함.
- □.offer('유') / □.peek() / □.poll()
  (넣기) (읽기) (꺼내기)
- 프로세스 스케줄링에 사용됨.

3. 스택 (Stack)

- FILO
- 프로세스 구조의 함수 동작 방식 (ex. 재귀함수)
- push(), pop(), peek(), isEmpty(), size(), search()
- 장점 : 구조가 단순하여 구현이 쉬움.
  데이터 저장 / 읽기 속도가 빠르다
- 단점 : 데이터 크기를 미리 정해놓아야함. → 낭비 발생 가능

4. 링크드리스트 (LinkedList)

- 순차적으로 연결된 구조 X , 화살표로 다음 데이터를 가리키는 구조
- (데이터값 & 포인터(주소)) → 하나의 노드
- 입력·출력 구현해보기 (node Class 활용하여 구현)
- 장점 : 미리 공간 할당 X , 삭제·추가가 쉬움(주소만 바꿔주면됨—이건 단점으로 보기도 함)
- 단점 : 별도 포인터 넣을 공간이 필요 (=낭비)
  순차접근 (→ 속도 느림)

✱ 아이디어 : while 반복문에 boolean type 사용하여 데이터 찾아 탈출 쉽게 구현

ex)
```
boolean search = true;
while (search) {
    if (list[i] == 1) search = false;   ↳ 다음 루프에서 탈출됨
    else i++;
}
```

✱ 삭제 구현 시 고려할것 ① 첫 node 삭제 시 ⇒ head 지우고 node 교체
② 마지막 〃 ⇒ 마지막 node 지우고 n-1의 node의 포인터를 null로.
③ 중간 〃 ⇒ pointer 가리키는 ➡ 위치 교체

5. 다양한 링크드리스트 구조

∨ 더블 링크드리스트 : 앞 node 를 가리키는 pointer 도 node에 존재함
(→ 검색의 양방향화)
⇒ head / tail 이 존재

주의할것 : (Insert / delete 시에 prev · tail 변경 꼭 해줄것!!!
순행·역행 함수 모두 구현할 것.

6. 알고리즘 복잡도 표현 방법 : 시간 복잡도 (✱계산)

- 어떤 알고리즘이 가장 효과적인지 판단 가능 (효과적 = 성능)
- 알고리즘 실행 속도 ⇒ 반복문이 결정
- Big O 표기법 : 최악의 실행시간을 표기

DATE. NO

※ 빅오 표기법

: 입력에 따라 결정되는 시간복잡도 함수

기본명제: $O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(n!)$

↳ 내가 어떤 알고리즘과 유사한지 판단하여 상대적으로 성능을 비교할 수 있음 (나: O(n) / 상대: O(1) ⇒ 상대의 성능이 좋음)

v 이때 n? 식에 가장 큰 영향을 미치는 변수 (= 입력)

v O(1) = 입력이 몇 개이든 실행(반복)은 2번(상수회) ex) if문

일차식 O(n) = n번, n+10번, 3n+10번 (변수값이 10번) ... ex) for(int i=0; i<=n; i++)

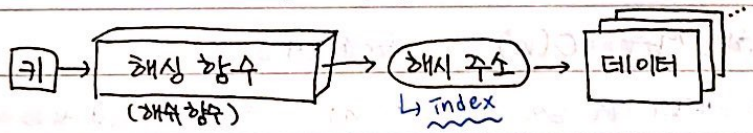
↳ 반복문 3번 도는거 써져 있고

이차식 $O(n^2)$ = n^2번, n^2+1000번 ...

cf. 함수로 생각할것: y=1, y=log x, y=n, y=x log x ...

7. 해쉬 테이블 (Hash table) (→ 블록체인에 활용)

- 파이썬의 딕셔너리 타입 생각할 것
- 주소와 데이터가 연결되어 있음 (키 → 주소 → 데이터)



- 결국에 원리는 배열, 인덱스가 자유로운 배열임... (키값을 index로 변환시킨후 저장)

cf. 아스키 코드 이용하기도함.

- 장점: 데이터 저장 / 읽기 속도 빠름

데이터가 있는지 쉽게 확인이 가능 (중복검사 역시)

- 단점: 저장공간(해쉬 테이블)이 많이 필요함.

주소의 충돌가능성 → 별도 구조 필요함 (중복 X → 테이블을 크게 잡아야 유리)

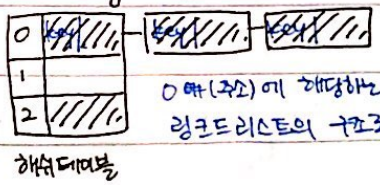
↳ 공간과 탐색 시간을 맞바꾼다

해결 방식

① Chaining: 해쉬 테이블 외의 저장공간 → 링크드 리스트 형태로 추가하기

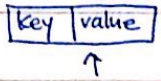
② Linear Probing: 해쉬 테이블 안의 빈 공간을 이용 (해당 주소 뒤 차음으로 등장하는 빈 공간에 데이터 입력)

I. Chaining 기법



해쉬테이블

0번(주소)에 해당하는 데이터 입력 시도 시 링크드리스트의 구조로 해당 데이터 뒤에 연결됨



이런 형태로..

↑

→ 역으로 찾을 때를 위하여 최초의 키 값 역시 저장 필요 (해당 데이터 주소의 중복 X)

Ⅱ. Linear Probing 기법 (close hashing 기법 中 하나)

- 저장 공간 활용도가 높음.
- 얘 역시 키 / value 형태로 저장이 필요 → key가 같다? 로 확인이 가능
  (∵ 주소에 직접 저장된 애인지, 밀려나서 저장된 애인지 확인해야함)
- 해싱함수 통과한 주소값 ~ 해쉬테이블의 마지막 주소까지 for loop 를 돌려서
  빈공간에 저장 or 키랑 맞는 데이터를 가져오기
  (Save) (read)
- Save 과정에서 Update 까지 고려해줄것. (만약에 키가 같으면 동일한 애구나 생각하기)

✡ 충돌 개선하기 → 단순히 해쉬 테이블의 크기를 늘릴것 (∞) 2배 이상~)

✡ 해쉬 함수 예시) SHA (Secure hash algorithm) -256 (⇒ 블록체인에 활용)

↳ Java.security.MessageDigest (여기서 getInstance("SHA-256"))

Java.security.NoSuchAlgorithmException (try-catch 문 → Catch에 사용)

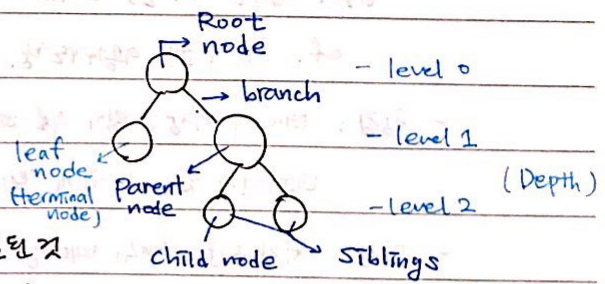
* 클래스기 확인할것.

✡ 해쉬 테이블의 시간 복잡도

- 일반적으로 O(1) … 고정 전환→ 입력 끝.
- 충돌이 발생하는 경우 O(n) … for문 돌아감
  (모든 데이터가)

* 배열의 경우 검색-저장 → O(n) : 일일이 다 확인 필요

8. 트리 구조



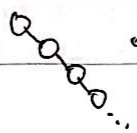
- node & branch 로 구성된 데이터 구조
  - 이진 트리 : branch가 2개씩
  - 이진 탐색 트리 : 이진트리인데, 크기대로 정렬된 것
    (오른쪽: 큰 값 / 왼쪽: 작은 값)

✡ LinkedList 이용하여 구현.

✡ 단말 node 삭제 / ChildNode가 한개인 node / 두개인 node 삭제의 세 경우를 나눠서 삭제를 구현해야함
  ↳ 왼쪽 child node의 가장 큰 값 / 오른쪽 " 작은 값을 Parent node로 이동시키기

① value 있는지 확인 → ② ✡탐색 → ③ 삭제

- 시간복잡도 (탐색) : O(h) cf. h = 트리의 차수
  O(logn) cf. n = 노드의 개수
- 단점 : 순서대로 들어가므로 오른쪽으로 (내림차순) 들어갈 때 어떤 꼴로 불균형하게 정렬됨



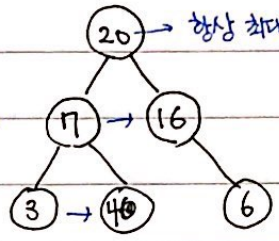
DATE. NO.

9. heap 구조 (tree 응용)

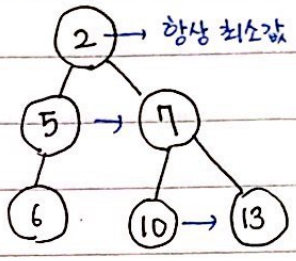
- 왜 사용? 최대값과 최소값을 찾는 데 유리하다. (시간복잡도가 배열 $O(n) \rightarrow O(\log n)$)

조건 ① 최대 heap : parent node가 무조건 child node보다 큼 (크거나 같음)
최소 heap : 〃 보다 작거나 같음

ex) 최대 heap



최소 heap



조건 ② - 완전 2진트리와 동일 (좌측부터 먼저 채워짐 ; 좌→우 node로 채워짐)

★ 2진 탐색 트리와의 공통점과 차이점

공 : 이진트리 꼴

차 : ✓ node 차수에 따라 크기대로 정렬됨 (heap구조)

✓ parent node 내의 child node 는 왼쪽 / 오른쪽 크기로 구분 X

⇒ 결국 2진 탐색 트리는 탐색, heap은 최대/최소값 검색 을 위한 용도

- 실제에서 주의할것

① 입력 : 입력 시 parent node가 교체되기도 하겠다.

(Max) 경우의 수 [입력이 현재 heap 노드보다 작을 때 → 왼쪽에 넣기 (채워졌으면 오른쪽)
〃 클 때 → 일단 넣고 위 node와 비교

② 삭제 : 일반적으로 삭제는 최상단 노드의 삭제를 의미한다. (∵ 최대/최소 검색이 목적임)

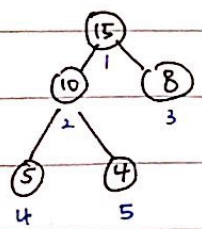
→ 따라서 삭제 후 가장 마지막에 넣은 값을 최상단으로 이동

→ child node와 비교하여 위치를 조정함

(나의 생각 : 그냥 child node 중 큰 값 올리기 반복하면 안되나?

⇒ 구현이 더 복잡하고, 시간복잡도로 더 클듯)

★ 힙 구현 시 일반적으로 배열 사용 (index 활용함) - 완전 이진 트리니까.



└ 부모 node 찾기 → 내 index / 2

└ 왼쪽 child node 찾기 → 부모 index * 2

└ 오른쪽 〃 → 부모 index * 2 + 1

⇒ index는 그냥 1번부터 쓰기

- 시간복잡도 : $O(\log n)$ (∵ 완전이진트리)