HASHING

a way to quickly "compress" the "full key" to a "numerically small ID" while "avoiding" collisions

- hash function : full key를 numerically small ID로 compress 하는 function
- hash value : hash function의 리턴값
- hash table : hash value에 의해 indexed된 direct-address table.

Hash functions

Division : 나눗셈 결과 나머지를 hash value로 가짐. 나누는 수는 소수로 하는게 좋음.

Mid-square : 제곱한 후 적절한 크기의 중간값 hash value로 사용.

Folding
- shift folding : 12345 ··· ⇒ 
- folding at the boundaries : 12345 ··· ⇒ 

Digit analysis. : 처음에 전체 key를 알때 각 자리수를 examine 해서 key를 구분할 수 있는 hash value 정해.

universal hashing : hash table의 크기를 m이라 할 때 $P(h(x)=h(y)) = 1/m$이다.

Hash tables

If necessary, Full ID needs to be stored.

Collision : different ID, same index의 값이 여러개 있는 경우

→ 2-dimensional array 선언으로 해결 (overflow : bucket 수용가능 수 이상 들어온경우)

hash function의 목적
- ① fast
- ② small hash value
- ③ avoid collision

①,③ : full ID를 hash value로 설정 ①,② : using last 2 digit ②,③ : 2-dimensional array

⇒ 모두 해결을 위해

Open addressing : entry가 차있으면 다른 entry를 prob

① linear probing : prob하는 다음 entry는 다음칸. // hash table 크기가 저장할 data 수보다 커야함.

- primary clustering : cluster가 생김. 한쪽에 너무 값이 몰려.

② quadratic probing : $(i+1)^{st}$ prob하는 index는 $(h(k)+C_1 i+C_2 i^2) \bmod m$. ($k$는 key, $C_1, C_2 \neq 0$)

- secondary clustering : 같은 hash value의 sequence가 꽉찬경우 존재.

③ double hashing : $(i+1)^{st}$ probe of index is $(h_1(k)+ih_2(k)) \bmod m$ (h_1, h_2는 다른 hash function)

- $h_2(k)=500$, $m=1000$인 경우 sequence 2개뿐.

⇒ hash table 크기가 prime number여야.

$\{ia \bmod b \mid i \in \mathbb{N}\} = \{0, \dots, b-1\}$ 이려면 a.b가 relatively prime 이어야 함.

☆ deletion 연산

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	···	[99]
		90002 data	89003 data	72003 data	73005 data			···	

89003을 지울땐 문제가 없지만 값을 지워버리면 72003을 지울 때 값이 없다고 생각할 수 있음.

해결 ① : shifting (한개씩 앞으로) ⟶ 원래 제대로 된 hash value에 넣어진 값 (73005) 문제생김.

해결 ② : "deleted" flag
- new item insert시 : deleted == true 이면 empty cell취급.
- looking for data : deleted == true이면 occupied 정도로 생각하고 넘어가.

Chaining : entry를 linked list로 만듦. (#bin < #item)

Load factor (# elements / hash table size)

linear probing에서는 data가 많을수록 cluster가 일어날 확률이 크기 때문에 prob할 개수도 증가.

RED-BLACK TREE

date : . .

Red - black trees

* leaf node를 null로 표시하여 사용한다. 즉, leaf에 가서 null을 만나면 그것을 leaf로 봄.

Tree property

P1) Every node is colored either red or black

P2) The root is black

P3) Every leaf/null pointer is black

P4) If a node is red, both its children are black

P5) Every simple path from a node to a descendant leaf contains same # black nodes. (bh(x))

⇒ Lemma : A red-black tree with n(internal) nodes has height $\leq 2\log_2(n+1) - 2$

claim : (#internal nodes in a subtree rooted at x) $\geq 2^{bh(x)} - 1$

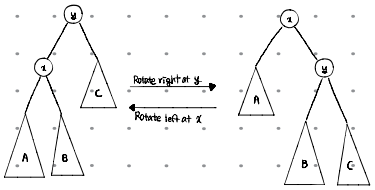
proof) n=0일때 $0 \geq 2^0 - 1 = 0$ 성립

n+0일때 각 child block-height $\geq bh(x) - 1$ ∴ (#nodes) $\geq 2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ 성립

→ height = h일때 $bh(x) \geq h/2 + 1$ (단, root)

$n \geq 2^{h/2+1} - 1$ → $h \leq 2\log_2(n+1) - 2$

Rotations



Insertion

Binary Search Tree에서 insert연산 이후에 insert한 node를 red로 색칠 이후 property 맞춰주기

<property 맞추기>

newly inserted node가 root가 아니고, parent가 red인동안 (while문)

i) uncle이 red일때 : parent uncle black으로 만들고 grandfather red로 바꾼 후 x → grandfather

ii) 아닐때

x가 right child일때 : left rotate at parent x→parent

parent black으로 , grandparent red로

right rotate at grandparent

x→ root node

root black으로

(p2) initially empty 였을때 red node 색칠로 발생

(p4) parent가 red인데 red를 색칠한 경우

⇒ running time : O(logn)

Deletion

Binary Search Tree의 deletion을 진행한 후 property 맞추기

disappearing node의 대체 노드를 x라고 하면

- disappearing node가 red일때 : no problem
- disappearing node가 black일때 : 모두 extra black 처리를 하고 만든수행

x가 root node가 아니고, black일 때 (while문)

1) sibling이 red일때 : sibling을 black으로, parent는 red로, left rotate at the parent

2) sibling이 black일때

i) nephew 둘 다 black일때 : sibling은 red로, x→parent

ii) else

1) right nephew는 black일때 : left nephew를 black으로, sibling은 red로, right rotate at sibling

Sibling과 parent의 색 SWAP

right nephew black으로

left rotate at the parent

x→root

x를 black으로

⇒ running time : O(logn)

FIBONACCI HEAPS

date : . .

Fibonacci Heaps

각노드의 key는 children보다 작음.

각 노드는 child포인터, parent, prev, next를 가지고 있음.

Sibling은 circular doubly linked list로 연결.

root도 circular doubly linked list로 연결되어있고, 그 중 minimum을 가리키는 포인터가 존재한다.

child를 잃으면 mark가 되고, parent가 바뀌지 않으면 child를 더이상 잃을 수 없다.

account (balance) : 남은시간을 저장하고 필요할 때 쓰는용도

potential : lower bound depending only on current state.

(#roots) + 2 · (#marked nodes)

Operations

① creating a new heap : root의 empty list initialize

- running time : O(1)

② insert a new value : new root로 만들어서 minimum check.

- running time : O(1) , potential increase : O(1)

③ Querying the minimum

- running time : O(1)

④ uniting two heaps : root를 하나의 list로만들고 min update.

- running time : O(1) potential = (1st potential) + (2nd potential)

⑤ deleting the minimum : minimum root를 지우고 children은 new root로만듬. → consolidating

- consolidating : 같은 degree를 가진 root이 없도록
- 1) delete → root만들기 running time : O(D(n)), potential increase : O(D(n))
  2) consolidation running time : potential decrease로 paid.
  3) build new list, min update : running time O(D(n))

  ⇒ O(D(n))

⑥ decrease Key : 값을 decrease 하고 property 만족안하면 잘라서 root로 만듬. 자른노드의 parent가 marked라면 parent도 자름.

- 1) decrease Value running time O(1) potential increase : O(1)
  2) parent losing 2nd child running time O(1) potential decrease O(1) { 마지막 node는 marking만 해주고 marked node숫자 체크유의 }

  ⇒ O(1)

⑦ delete : -∞로 decreaseKey 하고 delete Min.

SEGMENT TREE

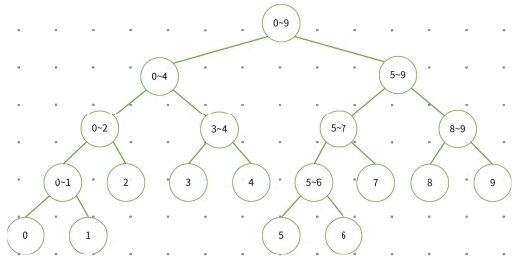
Curve of forgetting study planner

1HOUR	1DAY	2DAY	1WEEK	2WEEK	1MONTH	3MONTH

date : . .

Segment Tree

배열을 저장하는 도구로서 배열의 값이 바뀔 때 구간합을 저장함으로서 구간합을 빠르게 구할 수 있게 하는 자료구조.



- leaf node : 배열의 수 저장
- internal node : left child + right child 값 저장

구현

left child node 2*n

right child node 2*n + 1

→ 배열 구현 (vector)

✡ initialization : 배열을 모두 받은 후 tree init

저장할 배열 a, 트리 tree, 노드 번호(vector index) node, 담당하는 합의 범위 start & end)
↓
배열의 index

- 재귀

```
// a: 배열 a
// tree: 세그먼트 트리
// node: 세그먼트 트리 노드 번호
// node가 담당하는 합의 범위가 start ~ end
long long init(vector<long long> &a, vector<long long> &tree, int node, int start, int
end) {
    if (start == end) {
        return tree[node] = a[start];
    } else {
        return tree[node] = init(a, tree, node*2, start, (start+end)/2) + init(a, tree,
node*2+1, (start+end)/2+1, end);
    }
}
```

✡ calculating sum

- node가 담당하고 있는 구간 [start, end], 합을 구하는 구간이 [left, right]

① [left,right]와 [start, end]가 겹치지 않는 경우

② [left, right]가 [start, end]를 완전히 포함하는 경우

③ [start, end]가 [left, right]를 완전히 포함하는 경우

④ [left, right]와 [start, end]가 겹쳐져 있는 경우 (1,2,3 제외 나머지 경우)

```
// node가 담당하는 구간이 start~end이고, 구해야하는 합의 범위는 left~right
long long sum(vector<long long> &tree, int node, int start, int end, int left, int
right) {
    if (left > end || right < start) {
        return 0;
    }
    if (left <= start && end <= right) {
        return tree[node];
    }
    return sum(tree, node*2, start, (start+end)/2, left, right) + sum(tree, node*2+1, (
start+end)/2+1, end, left, right);
}
```

✡ 값 change

```
void update(vector<long long> &tree, int node, int start, int end, int index, long long
diff) {
    if (index < start || index > end) return;
    tree[node] = tree[node] + diff;
    if (start != end) {
        update(tree,node*2, start, (start+end)/2, index, diff);
        update(tree,node*2+1, (start+end)/2+1, end, index, diff);
    }
}
```

STANDARD LIBRARY

date : . .

1. vector
```
#include <vector>
vector<int> v;
v.push_back(10);
v.push_front(10);
v.size();
v.clear();
v.empty();
v.at(1); v[1];
v.pop_back();
v.pop_front();
v.insert(v.end(), 5); // end가 가리키게 되는 곳에 5의 값을 넣음

v.capacity();
v.reserve(100); //100개의 용량을 만들어줌
v.shrink_to_fit(19);
v.resize(10);

vector<int> v1(5); // 0을 5개 초기화
vector<int> v2(5, 2); // 2를 5개 초기화
```

2. iterator
```
vector<int>::iterator p;
p = v.begin();
p != v.end();
*p // iterator의 내부값
++, --로 iterator가 가리키는 칸을 옮김

const_iterator // const인 iterator
    v.cbegin();
    v.cend();
reverse_iterator // ++을 하면 반대방향으로 움직이는 iterator
    v.rbegin();
    v.rend();
const_reverse_iterator
    v.crbegin();
    v.crend();
```

3. list
```
#include <list> // doubly linked list만 있음 (singly linked list X)
list<int> List;
List.push_back(3);
List.push_front(3);
List.size();

list<int>::iterator iter;
```

4. deque : list + vector / 랜덤 엑세스, push_front, pop_front, vector의 기능 모두 가능
```
#include <deque>
deque<int> dq;
```

5. stack
```
#include <stack>
stack<int> Stack;
Stack.push(10);
Stack.pop();
Stack.size();
Stack.empty();
Stack.top();
```

6. queue
```
#include <queue>
queue<int> Queue;
Queue.front();
Queue.back();
Queue.push(4);
Queue.pop();
Queue.empty();

priority_queue
priority_queue<int> PQueue;
push하고 pop을 하면 큰 숫자부터 나옴 -> heapSort
// 작은숫자부터 나오게 하려면?
#include <functional>
priority_queue<int, vector<int>, greater<int> > PQueue; // 어렵네
// top에 있는 값만 정렬되어 있음
```

7. set
```
#include <set>
set<int> s;
s.insert(10); // 중복된 값은 2번들어가지 않음
s.find(10); // find할 대상을 찾은 경우 그 값을 가리키는 iterator가, 없으면 end()위치의 iterator를 리턴
s.end()
s.erase(s.find(1)); // 1이 없는 값일 경우 end()를 지우려고 하므로 문제가 될 수있음
s.erase(1);
// 정렬되어 저장되어있음

#include <cctype>
s.count(1); // 1이 존재하면 true, 아니면 false리턴

// iterator대신에 순회를 쉽게 하는 방법
for(int current:s){cout<<current<<endl;}
for(auto current:s){cout<<current<<endl;}
```

8. multiset
```
#include <set>
multiset<int> s;
s.erase(3); // 3의 중복된 값 모두 삭제
s.erase(find(3)); // 해당 iterator한개만 삭제 (숫자 하나만 삭제됨)
s.count(3); // 3의 갯수 출력
```