고급 11회차
Splay Tree

서강대학교 전해성(seastar105)

Binary Search Tree

이진 검색 트리는 각 노드마다 key값을 가지고 있으며, 어떤 노드 x의 left child에는 x의 key보다 작은 노드들이 존재하며, right child에는 x의 key보다 큰 노드들이 존재하도록 하는 이진 트리입니다.

삽입, 읽기, 갱신, 삭제(CRUD)같은 기본적인 연산들을 지원합니다.
트리를 inorder로 순회하면 key들이 정렬된 상태로 출력이 가능합니다.

문제 : 트리가 Unbalance할 경우에 기본연산의 시간복잡도가 O(n)

Self-Balancing Binary Search Tree

트리가 unbalance한 상태가 되는 것을 막기 위해서 트리의 구조를 자동으로 바꾸는 이진 탐색 트리를 Self-Balancing Binary Search Tree라고 부른다.

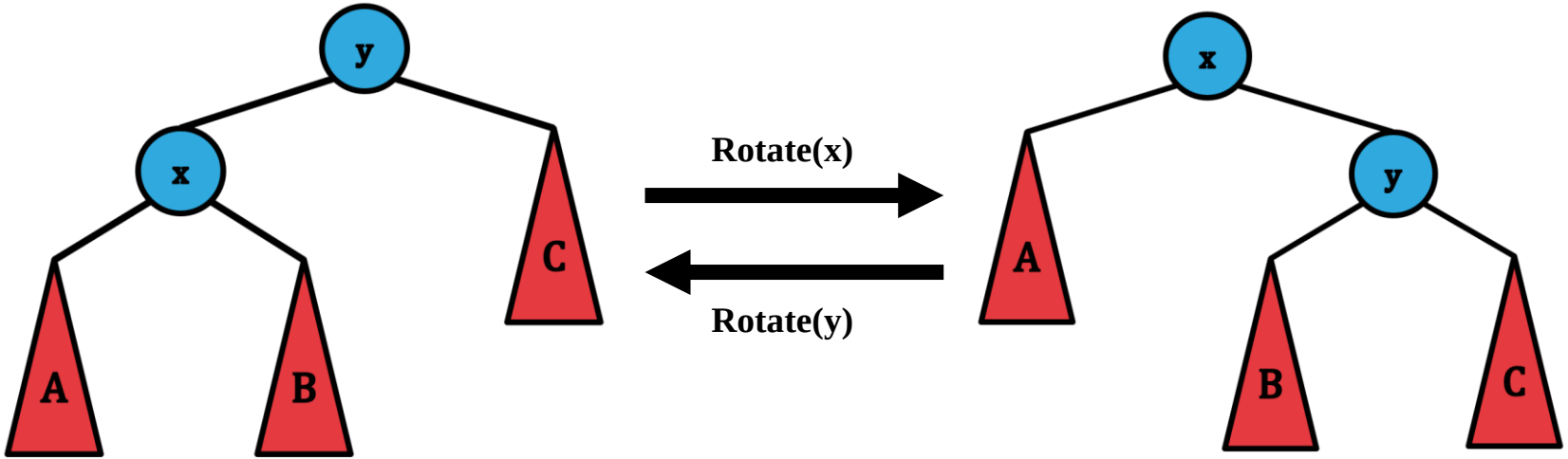
그러한 트리의 예시로 Red-Black Tree, AVL Tree, Treap, Splay Tree 등이 있다.

PS에서 주로 사용되는 것은 Treap과 Splay Tree이다.

Rotate Node in BST

BST에서 자주 사용되는 연산 중 하나인 Rotate를 알아봅시다.

어떤 노드에 rotate 연산을 취하면 그 노드의 높이가 하나 높아지되, BST의 특성은 유지하도록 하는 O(1) 연산입니다.



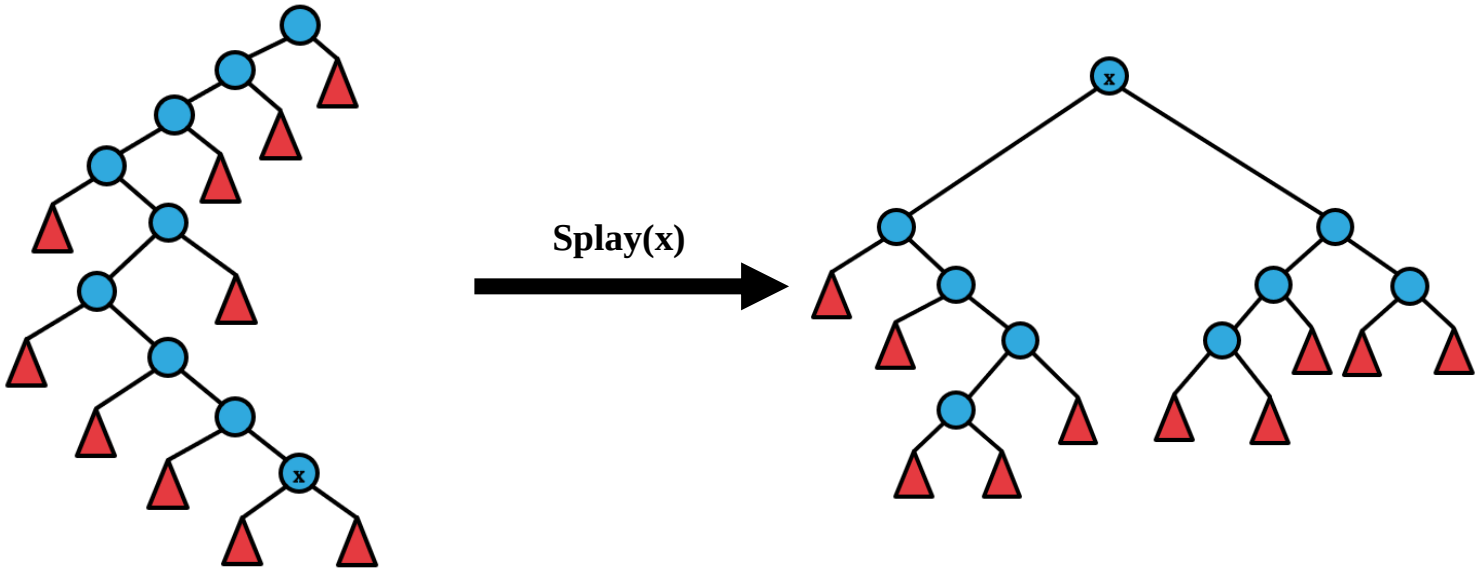
Splay Tree

Splay Tree는 Splay라는 연산을 통해서 트리를 Balance하게 유지하는 BST입니다.

삽입, 읽기, 삭제와 같은 기본적인 연산은 일반적인 BST와 동일합니다.
다만, 이러한 연산 후에 해당 노드에 Splay라는 연산을 수행함으로 트리의 구조를 바꾸게 됩니다.
정확히는 어떤 노드 x에 접근이 발생했다면 그 노드 x를 Splay합니다.

Splay Operation

Splay(x) 연산은 어떤 노드 x를 inorder의 순서를 유지하면서 그 노드가 속한 BST의 root로 만드는 연산을 말합니다.



Splay Operation

Splay 연산을 설명하기 위해서 먼저 사용할 Notation을 정리하고 가겠습니다.

x : Splay 대상 노드

p : x의 부모 노드

g : p의 부모노드(존재할 경우)

Splaying-step

Splay 연산은 x가 root가 될 때까지 x에 Splaying-step을 반복하게 됩니다.

Splaying-step은 세 가지 경우가 있습니다.

Case 1(Zig-step)

p가 root일 경우입니다. rotate(x)를 수행합니다.

Case 2(Zig-Zig step)

g가 존재하며, x와 p가 둘 다 left child거나, right child일 경우입니다.

rotate(p), rotate(x)를 순서대로 합니다.

```
def Splay(x):
    while x is not root:
        Splaying_step(x)
```

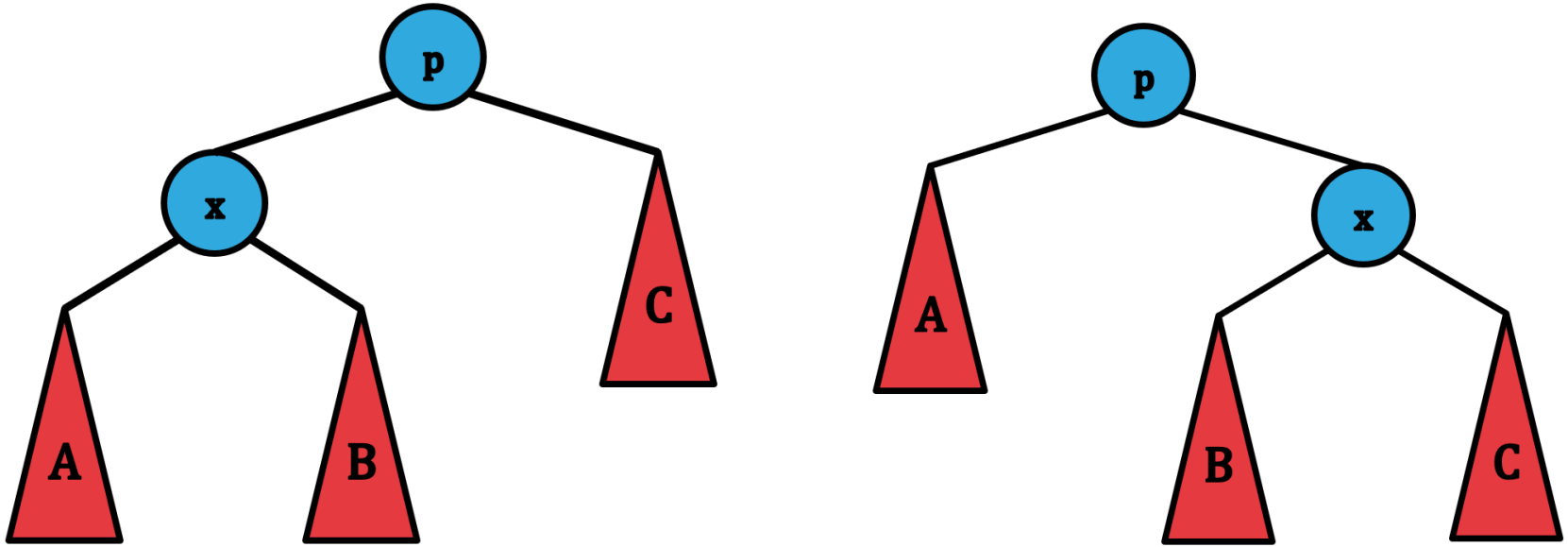
Case 3(Zig-Zag step)

g가 존재하며, x가 left child고 p가 right child이거나 그 반대일 경우 입니다.

rotate(x)를 두 번 수행합니다.

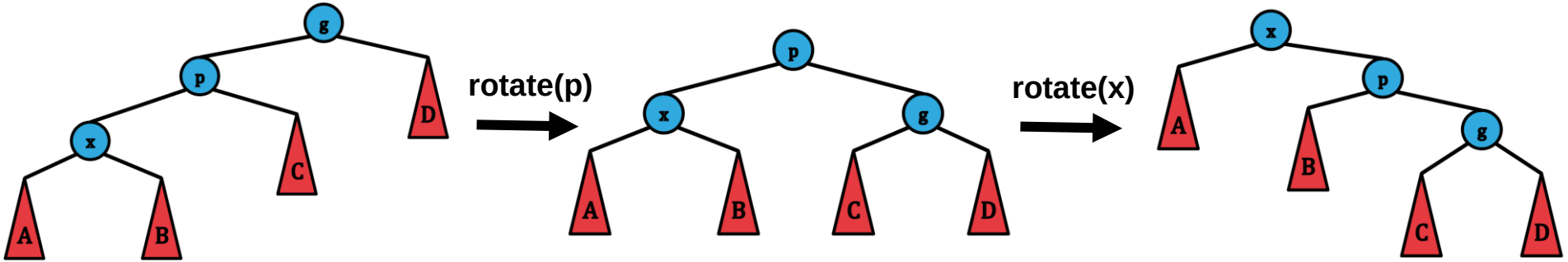
Splaying-step (Case 1 Zig step)

이 경우는 일반적인 rotate 상황과 동일합니다. rotate(x)를 수행하면 x가 root가 됩니다.



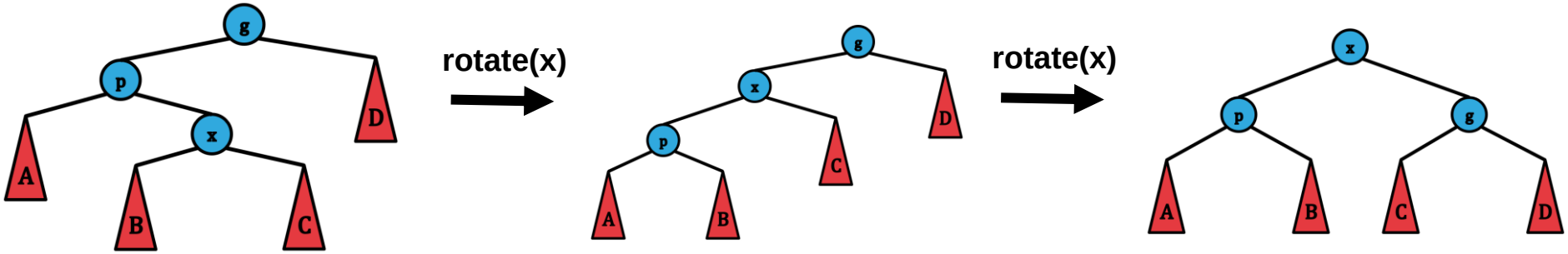
Splaying-step (Case 2 Zig-Zig step)

x와 p가 둘 다 같은 방향의 자식일 경우입니다. rotate(p)를 먼저 수행하고 rotate(x)를 수행합니다.



Splaying-step (Case 3 Zig-Zag step)

x와 p가 서로 다른 방향의 자식일 경우입니다. rotate(x)를 두 번 수행합니다.



Time Complexity

놀랍게도 각 접근마다 접근한 노드를 Splay하는 것으로 모든 BST 연산의 시간복잡도는 amortized O(log n)이 됩니다.

amortized라는 새로운 용어가 등장했습니다.
amortized time complexity라는 것은 어떤 연산이 있고 최악의 경우에 해당하는 연산 순서열을 수행했을 때, 각 연산의 평균적인 시간복잡도를 말합니다.
여기서 최악의 경우란 어떤 알고리즘에서 최악의 시간복잡도가 나올 수 있는 경우를 말합니다. Quick-sort의 최악의 경우가 $O(N^2)$인거처럼요

Amortized vs Average

평균 시간복잡도라고 말하면 Average 와 Amortized가 무엇이 다른 것인지 궁금하실 수도 있습니다.

Amortized Time Complexity는 길이 m의 operation sequence가 주어졌을 때, 이 operation sequence가 최악의 경우일 때 각 operation의 평균적인 시간복잡도 입니다.

Average Time Complexity는 가능한 모든 operation sequence에 대해서 평균적인 시간복잡도이기 때문에 Worst Case에 대한 시간복잡도가 보장이 안됩니다.

Amortized vs Average

예를 들자면 비어 있는 BST에 1부터 N까지 차례대로 삽입연산을 하고 N만 읽는 경우를 생각해볼 수 있습니다.

일반 BST는 위 순서대로 연산을 수행하면 각 연산이 O(N)만큼 걸리게 됩니다.
스플레이 트리는 amortized O(logN)을 보장해줍니다.

Potential Function

이제 amortized time complexity를 어떻게 계산하는지를 알아봅시다.

이를 위해서 Potential Function Φ를 정의합니다.

이 함수는 자료구조의 상태를 수치로 바꾸는 역할을 합니다.

이제 퍼텐셜 함수를 사용하여 amortized time을 정의합니다.

a를 amortized time, t를 실제 연산 시간이 t라고 하면 아래와 같이 정의됩니다.

$$a = t + \Phi' - \Phi$$

Φ'는 어떤 연산을 수행한 뒤의 퍼텐셜이고 Φ는 수행 전의 퍼텐셜입니다.

Potential Function

m개의 연산을 처리했을 때의 amortized time을 식으로 나타내면 아래와 같습니다.

$$\sum_{}^{m} a_i = \sum_{}^{m} (t_i + \Phi_i - \Phi_{i-1})$$
$$= \sum_{}^{m} t_i + \Phi_m - \Phi_0$$

이 때, Φ_0은 초기 퍼텐셜이고, a_i, t_i, Φ_i는 각각 i번째 연산을 했을 때 amortized time, 실제 시간, 연산을 한 뒤의 퍼텐셜입니다.

이 식으로 부터 만일 퍼텐셜 변화가 0보다 크거나 같으면 amortized time은 실제 시간의 upper bound가 되는 것을 알 수 있습니다.

Amortized Time Complexity of Splay

BST에서 기본 연산들의 시간 복잡도는 접근하고자 하는 노드의 깊이에 비례합니다.

Splay 연산 또한 노드의 깊이에 비례하는 연산입니다.

두 연산 모두 같은 비용이 드는 연산이라고 볼 수 있습니다.

따라서, Splay 연산의 amortized 시간 복잡도가 O(log n)임을 보이면 기본 연산들의 amortized 시간 복잡도 또한 O(log n)이 될 것으로 보입니다.

Amortized Time of Splay

Notation 정리

S(x) : x를 루트로 하는 서브트리의 크기

r(x) : $\log_2(S(x))$, 이를 노드 x의 rank라고 하겠습니다.

Φ : 모든 노드의 rank의 합, 이를 퍼텐셜로 정의하겠습니다.

Φ(T): 트리 T의 퍼텐셜

a(o): 연산 o의 amortized time

t(o) : 연산 o의 actual time

기호위에 ’가 찍혀 있으면 어떤 연산 뒤의 값이라고 하겠습니다.

Amortized Time of Splay

연산 시간을 계산하기 위해서는 기본 연산을 정해야 합니다.

rotate를 기본 연산으로 하겠습니다.

Splay는 여러 번의 Splaying-step으로 이루어집니다. Splaying-step은 3가지 경우가 있었으며, 각 경우에 대해서 **amortized time**을 계산해봅시다.

중요한 점이라 강조드립니다. 지금부터 구할 것은 **amortized time**입니다. **Amortized time complexity**가 아닙니다.

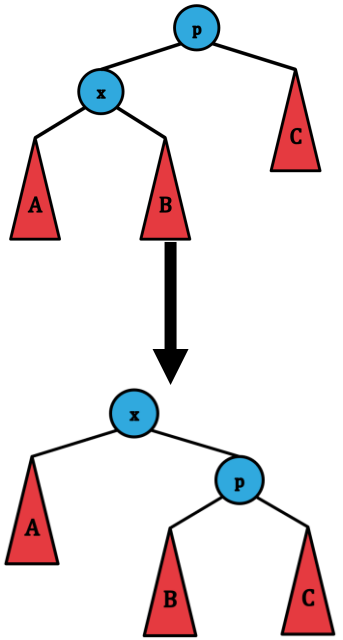
Amortized Time of Splaying-step (Case 1)

Case 1(Zig-step)

p가 root일 경우입니다. rotate(x)를 수행합니다.

연산 전후의 퍼텐셜 변화를 살펴봅시다.

$$\Phi = r(x) + r(p) + \Phi(A) + \Phi(B) + \Phi(C)$$
$$\Phi' = r'(x) + r'(p) + \Phi(A) + \Phi(B) + \Phi(C)$$



x와 p외에는 rank값이 변하지 않는 것을 확인할 수 있습니다.

rotate연산을 통해서 rank가 변하는 노드는 rotate에 실제 관여된 노드 말곤 없습니다.

Amortized Time of Splaying-step (Case 1)

이제 Case 1의 amortized time을 수식에 넣어서 구하면 아래와 같은 upper bound를 얻을 수 있습니다. t(Zig-step)은 rotate를 한 번 진행하기 때문에 1입니다.

$$
\begin{aligned}
a(Zig\text{-}step) &= t(Zig\text{-}step) + \Phi' - \Phi \\
&= 1 + r'(x) + r'(p) - r(x) - r(p) \\
&= 1 + r'(p) - r(x) \quad (\because r'(x) = r(p)) \\
&\leq 1 + (r'(x) - r(x)) \quad (\because r'(p) \leq r'(x)) \\
&\leq 1 + 3(r'(x) - r(x))
\end{aligned}
$$

굳이 3을 곱해서 upper bound를 잡은 이유는 후의 증명을 편하게 하기 위해서 입니다.

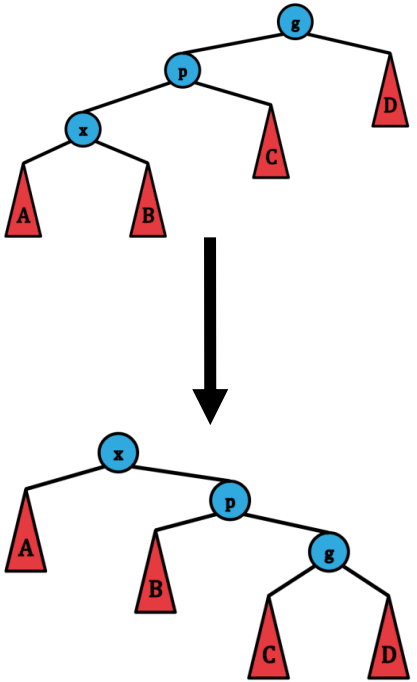
Amortized Time of Splaying-step (Case 2)

Case 2(Zig-Zig step)

g가 존재하며, x와 p가 둘 다 left child거나, right child일 경우입니다.

이번에도 연산 전후의 퍼텐셜을 봅시다.

$$\Phi = r(x) + r(p) + r(g) + \Phi(A) + \Phi(B) + \Phi(C) + \Phi(D)$$
$$\Phi' = r'(x) + r'(p) + r'(g) + \Phi(A) + \Phi(B) + \Phi(C) + \Phi(D)$$



Amortized Time of Splaying-step (Case 2)

a(Zig-Zig step)을 수식에 넣어서 구해봅시다. t(Zig-Zig step)은 rotate를 두 번 했기 때문에 2가 됩니다.

$$
\begin{aligned}
a(\textit{Zig-Zig step}) &= t(\textit{Zig-Zig step}) + \Phi' - \Phi \\
&= 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\
&= 2 + r'(p) + r'(g) - r(x) - r(p) \quad (\because r'(x) = r(g)) \\
&\leq 2 + r'(x) + r'(g) - 2r(x) \quad (\because r'(x) \geq r'(p),\ r(x) \leq r(p))
\end{aligned}
$$

현재 정리한 upper bound는 Case 1에서 구한 upper bound와 별로 관계가 없어 보입니다. 이전의 upper bound꼴을 구해봅시다.

Amortized Time of Splaying-step (Case 2)

이전 슬라이드에서 구한 upper bound를 Case 1의 것과 비슷한 꼴로 바꾸는 것이 목표입니다.

$$\text{Claim}: 2 + r'(x) + r'(g) - 2r(x) \leq 3(r'(x) - r(x))$$

$$2r'(x) - r(x) - r'(g) \geq 2$$

$$r'(x) - r(x) + r'(x) - r'(g) \geq 2$$

$$\log_2 \frac{S'(x)}{S(x)} + \log_2 \frac{S'(x)}{S'(g)} \geq 2$$

$$\log_2 \frac{(S(A) + S(B) + S(C) + S(D) + 3)^2}{(S(A) + S(B) + 1)(S(C) + S(D) + 1)} \geq 2$$

갑자기 수식이 막 나와서 당황하실 수도 있습니다.

r(x)와 S(x)에 로그를 취한 값입니다.

이를 대입해서 정리한 것이 제일 아래 부등식입니다.

Amortized Time of Splaying-step (Case 2)

마지막 부등식의 로그 안에 있는 식을 변형해서 증명이 가능합니다.

증명은 Lawali님이 도와주셨습니다.

$$\log_2 \frac{(S(A)+S(B)+S(C)+S(D)+3)^2}{(S(A)+S(B)+1)(S(C)+S(D)+1)} \geq 2$$

$$x = S(A)+S(B)+1, \quad x \geq 1$$
$$y = S(C)+S(D)+1, \quad y \geq 1$$

$$\frac{(S(A)+S(B)+S(C)+S(D)+3)^2}{(S(A)+S(B)+1)(S(C)+S(D)+1)} = \frac{(x+y+1)^2}{xy}$$
$$> \frac{(x+y)^2}{xy} = \frac{y}{x}+\frac{x}{y}+2$$
$$\geq 4$$

해당 식은 항상 4보다 큰 것을 알 수 있습니다.

따라서, 아래의 주장은 참입니다.

$$\text{Claim}: 2 + r'(x) + r'(g) - 2r(x) \leq 3(r'(x) - r(x))$$

이것으로 Zig-Zig step의 amortized time은 최대 3(r'(x)-r(x))라고 할 수 있습니다.

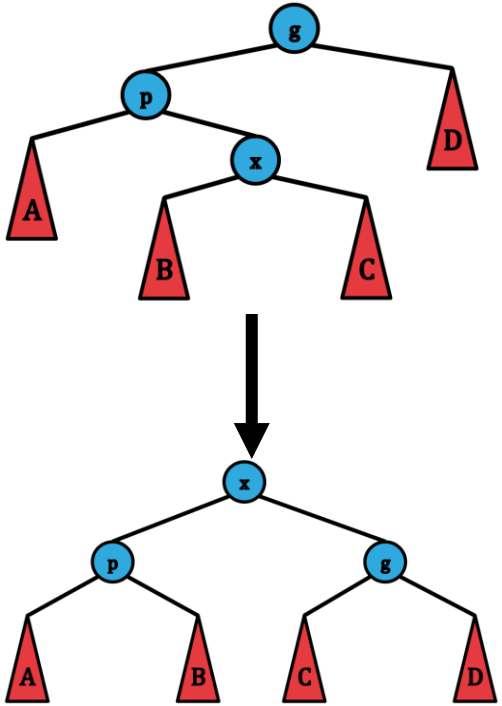
Amortized Time of Splaying-step (Case 3)

Case 3(Zig-Zag step)

g가 존재하며, x가 left child고 p가 right child이거나 그 반대일 경우입니다.

이번에도 연산 전후의 퍼텐셜을 봅시다.

$$\Phi = r(x) + r(p) + r(g) + \Phi(A) + \Phi(B) + \Phi(C) + \Phi(D)$$
$$\Phi' = r'(x) + r'(p) + r'(g) + \Phi(A) + \Phi(B) + \Phi(C) + \Phi(D)$$



Amortized Time of Splaying-step (Case 3)

a(Zig-Zag step)을 수식에 넣어서 구해봅시다. t(Zig-Zag step)도 rotate를 두 번 했기 때문에 2가 됩니다.

$$\begin{aligned} a(Zig\text{-}Zag\ step) &= t(Zig\text{-}Zag\ step) + \Phi' - \Phi \\ &= 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\ &\leq 2 + r'(p) + r'(g) - 2r(x) \quad (\because r'(x) = r(g),\ r(x) \leq r(p)) \end{aligned}$$

Zig-Zig step에서 했던 것과 마찬가지로 Case 1에서의 upper bound와 비슷한 꼴을 구해봅시다.

Amortized Time of Splaying-step (Case 3)

upper bound를 2(r'(x)-r(x))로 잡은 것 외에는 과정이 Case 2와 동일합니다.

$$\text{Claim}: 2 + r'(p) + r'(g) - 2r(x) \leq 2(r'(x) - r(x))$$

$$2r'(x) - r'(p) - r'(g) \geq 2$$

$$r'(x) - r'(p) + r'(x) - r'(g) \geq 2$$

$$\log_2 \frac{S'(x)}{S'(p)} + \log_2 \frac{S'(x)}{S'(g)} \geq 2$$

$$\log_2 \frac{(S(A) + S(B) + S(C) + S(D) + 3)^2}{(S(A) + S(B) + 1)(S(C) + S(D) + 1)} \geq 2$$

Amortized Time of Splaying-step

Splaying-step의 각 케이스의 amortized time의 upper bound를 계산하면 아래와 같습니다.

$$
\begin{aligned}
a(\textit{Zig-Zag step}(x)) &\leq 2(r'(x) - r(x)) \\
a(\textit{Zig-Zig step}(x)) &\leq 3(r'(x) - r(x)) \\
a(\textit{Zig}(x)) &\leq 1 + 3(r'(x) - r(x)) \\
a(\textit{Splaying-step}(x)) &\leq 1 + 3(r'(x) - r(x))
\end{aligned}
$$

Amortized Time of Splay

Splaying-step의 amortized time을 구했으니 이제 Splay의 amortized time을 봅시다.
Splay가 d개의 Splaying-step으로 이루어졌다고 합시다.

그러면 d-1개의 Zig-Zag step 혹은 Zig-Zig step과 1개의 Zig step이 있고, a(Splay)를 아래와 같이 계산할 수 있습니다.

$$a(Splay) = \sum^{d-1}(a(Zig\text{-}Zag\ step) \vee a(Zig\text{-}Zig\ step)) + a(Zig\ step)$$

Amortized Time of Splay

주목해야할 부분은 스텝은 모두 같은 노드 x에 대해서 이뤄진다는 점입니다.

Upper bound에 있는 r'(x)와 r(x)들은 마지막 Zig step에서의 r'(x)와 처음 step에서의 r(x) 외엔 모두 상쇄됩니다.

따라서, a(Splay)의 upper bound를 아래와 같이 구할 수 있으며, 트리의 크기를 n이라고 했을 때, O(log n)이 됩니다.

$$
\begin{aligned}
a(Splay) &= \sum^{d-1}(a(Zig\text{-}Zag\ step) \vee a(Zig\text{-}Zig\ step)) + a(Zig\ step) \\
&\leq 3(r(root) - r(x)) + 1 \\
&\leq 3\log_2 \frac{S(root)}{S(x)} + 1
\end{aligned}
$$

Actual Time of Splay

이제 Splay의 실제 연산시간을 알아봅시다. 처음에 연산 후의 퍼텐셜 변화량이 0보다 크거나 같으면 amortized time은 실제 연산시간의 upper bound가 된다고 했습니다.

퍼텐셜 변화량은 0보다 크거나 같습니다.

즉, 실제 연산시간은 amortized time보다 크지 않을 것입니다.

Time Complexity of Splay

이걸로 Splay의 시간 복잡도가 amortized O(log n)임을 증명했습니다.

왜 BST의 기본 연산들의 시간 복잡도가 Splay의 것과 같은 지에 대해서 엄밀하게 알고 싶은 분들은 원 논문을 보시는 것을 추천합니다.

D. Sleator and R. Tarjan. Self-adjusting binary search trees.JACM, 32(3):652–686, 1985

Implementation of Splay Tree

기본적인 토대는 일반적인 BST와 다를 것이 없습니다. 다만, 새롭게 사용되는 rotate와 splay 연산을 구현해줘야 합니다.

rotate는 x가 p의 left child일 때와 right child일 때로 구분해줘야 합니다.

```
struct Node {
    Node *l, *r, *p;
    Node() : l(nullptr), r(nullptr), p(nullptr) {};
};
```

```
void rotate(Node *x) {
    Node *parent = x->p;
    if(parent == nullptr) return ;  // x IS ROOT NODE
    Node *tmp;
    if(parent->l == x) {    // ROTATE RIGHT
        tmp = x->r;
        parent->l = x->r;
        x->r = parent;
    }
    else {      // ROTATE LEFT
        tmp = x->l;
        parent->r = x->l;
        x->l = parent;
    }
    // ADJUST EACH NODE'S PARENT
    Node *pp = parent->p;
    x->p = pp;
    parent->p = x;
    if(tmp) tmp->p = parent;
    if(pp) {
        if(pp->l == parent) pp->l = x;
        else if(pp->r == parent) pp->r = x;
    }
}
```

Implementation of Splay Tree

Splay는 Splaying-step을 반복하며 수행되는데 3가지 Case가 있었다.

아래는 Splay(x)의 구현이다.

```
void splay(Node *x) {
    while(x->p) {    // WHILE x IS NOT ROOT
        Node *p = x->p;
        Node *g = p->p;
        if(g) { // IF GRANDPARENT EXISTS
            if((g->l == p) == (p->l == x)) rotate(p);    // ZIG-ZIG STEP
            else rotate(x);      // ZIG-ZAG STEP
        }
        rotate(x);  // DEFAULT ZIG STEP
    }
}
```

Join and Split Operation

Splay Tree에서 유용한 연산인 Join과 Split이 있습니다.
아래와 같이 정의됩니다.

Join(T_1, T_2) : 두 개의 BST T_1, T_2를 하나의 BST로 합쳐줍니다. 이 때, T_1의 모든 key는 T_2의 모든 key보다 작아야 합니다.

Split(key, T) : BST인 T를 두 개의 BST T_1, T_2로 쪼갭니다. 이 때, T_1의 모든 key는 i보다 작거나 같고, T_2의 모든 key는 i보다 큽니다.

Join and Split Implementation

T_1에서 가장 큰 노드를 찾은 뒤 그 노드를 스플레이해줍니다.

T_1의 루트노드는 right subtree가 없는 상태입니다.

T_2를 right subtree로 만들어주면 됩니다.

```
// ASSUME THAT ALL KEYS IS left ARE LESS OR EQUAL TO THOSE OF right
Node* join(Node *left, Node *right) {
    if(left == nullptr) return right;
    if(right == nullptr) return left;
    Node *p = left;
    while(p->r) p = p->r;    // FIND LARGEST NODE OF left
    splay(p);                // SPLAY LARGEST NODE OF left
    p->r = right;
    right->p = p;            // MAKE right AS RIGHT SUBTREE OF left
    return p;
}
```

Join and Split Implementation

Split연산은 i의 lower_bound에 해당하는 노드를 스플레이 해주고 루트노드의 right subtree의 연결을 끊어주면 T_1, T_2를 만들 수 있습니다.

```
// SPLIT TREE INTO first <= x, second > x
pair<Node*, Node*> split(Node *x) {
    splay(x);
    Node *left = x;
    Node *right = x->r;
    if(left) left->p = nullptr;
    if(right) right->p = nullptr;
    return {left, right};
}
```

Insert and Delete

Insert(i,T) : T에 i를 삽입하는 연산입니다. 일반적인 BST에서와 마찬가지로 i가 삽입될 위치를 찾고 삽입해준 뒤에 새로 만든 노드를 스플레이해줍니다.

Delete(i,T) : T에서 i를 삭제하는 연산입니다. 이 연산도 일반적인 BST의 연산과 같게 i를 삭제 한 뒤에 삭제한 노드의 부모를 스플레이해줍니다.

Where to Use Splay Tree

지금까지 참 길었습니다. 이 스플레이 트리를 어디다 써야 좋을까요?

STL에 레드-블랙 트리 기반의 map, set이 존재합니다. 이 용도로는 굳이 직접 구현한 스플레이 트리를 쓸 이유가 없습니다.

map이나 set에서 K번째 원소를 찾는 것도 GNU의 policy based data structure를 가져다 쓰면 됩니다.

Dealing with linear array

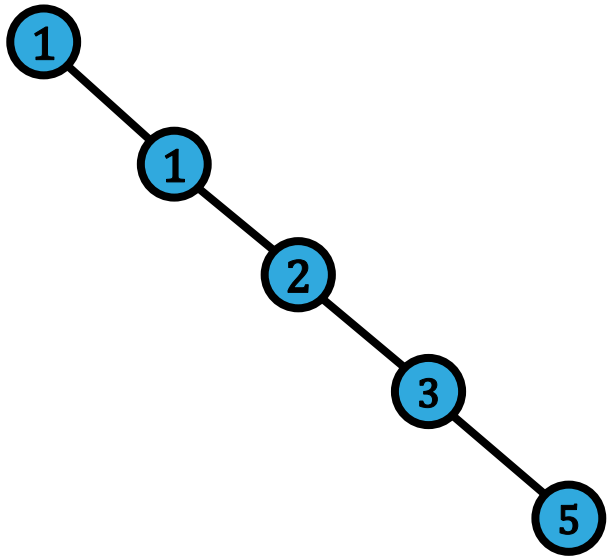
스플레이 트리는 항상 inorder 탐색 순서가 유지된다는 특징이 있습니다. 이를 이용해서 선형 배열의 원소들을 관리할 수 있습니다.

배열을 입력 받고 처음에 선형 트리를 만들고 inorder 순서대로 각 노드를 배열의 원소로 초기화 해줍니다. 그러면 이제 선형 배열의 순서를 inorder 순으로 가지는 스플레이 트리를 만들 수 있습니다.

Dealing with linear array

Example

1	1	2	3	5



Kth Operation

스플레이 트리로 선형 배열의 구간 쿼리를 처리하기 위해서 필수적인 연산이 있습니다.

Inorder 상으로 K번째 노드를 찾고 스플레이 해줍니다. 이를 위해서 각 노드는 자신을 루트로 하는 서브트리의 크기를 가지고 있습니다.

그리고 rotate 도중에 적절하게 본인의 자식 노드들을 확인하며 이를 관리해줘야 합니다. 이러한 연산은 pull이라고 부르겠습니다.

Kth Operation

pull을 rotate 끝에 자식 노드부터 수행함으로 서브트리의 크기를 유지합니다.

```
struct Node {
    Node *l, *r, *p;
    int sz;
    Node() : l(nullptr), r(nullptr), p(nullptr), sz(1) {};
};
```

```
void pull(Node *x) {
    x->sz = 1;
    if(x->l) x->sz += x->l->sz;
    if(x->r) x->sz += x->r->sz;
}
```

```
void rotate(Node *x) {
    Node *parent = x->p;
    if(parent == nullptr) return ;  // x IS ROOT NODE
    Node *tmp;
    if(parent->l == x) {    // ROTATE RIGHT
        tmp = x->r;
        parent->l = x->r;
        x->r = parent;
    }
    else {      // ROTATE LEFT
        tmp = x->l;
        parent->r = x->l;
        x->l = parent;
    }
    // ADJUST EACH NODE'S PARENT
    Node *pp = parent->p;
    x->p = pp;
    parent->p = x;
    if(tmp) tmp->p = parent;
    if(pp) {
        if(pp->l == parent) pp->l = x;
        else if(pp->r == parent) pp->r = x;
    }
    // PULL LOWER NODE FIRST
    pull(parent); pull(x);
}
```

Kth Operation

이제 노드 x에서 본인의 왼쪽 자식의 서브트리 크기와 오른쪽 자식의 서브트리 크기를 알기 때문에 inorder상에서 K번째 원소를 찾는 것이 가능해졌습니다. 루트 노드로부터 왼쪽 자식의 서브트리 크기와 K를 비교하면서 탐색을 진행하면 됩니다.

```
void kth(Node *root, int k) {
    Node *p = root;
    while(1) {
        while(p->l && p->l->sz > k) {    // WHILE LEFT SUBTREE IS BIGGER THAN K
            p = p->l;                    // GO LEFT
        }
        if(p->l) k -= p->l->sz;
        if(!k) break;
        else --k;                        // CURRENT NODE IS NOT kTH
        p = p->r;
    }
    splay(p);
    return p;
}
```

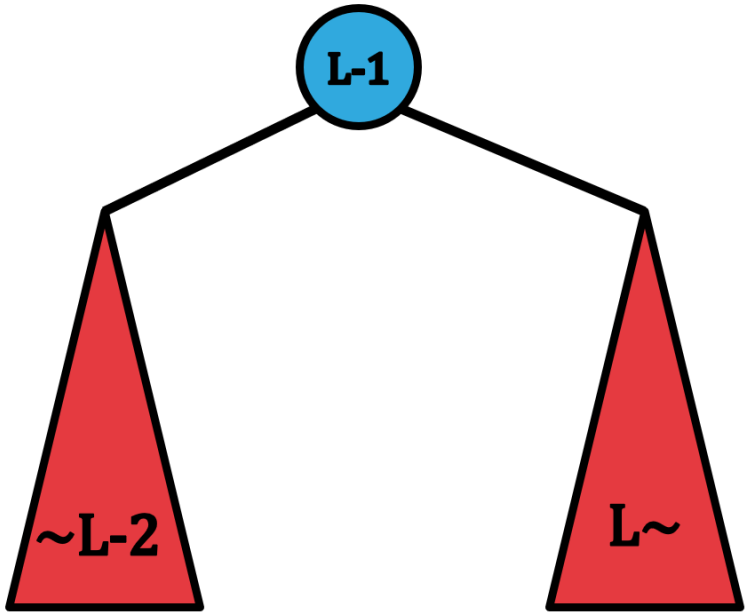
Now Segment Query

이제 구간을 다루기 위해서는 쿼리 구간에 해당하는 노드들을 모아줘야 합니다. Kth 연산을 이용해서 이것이 가능해집니다.

[L, R] 구간에 해당하는 노드들을 하나의 서브트리로 모으는 방법을 알아보겠습니다.

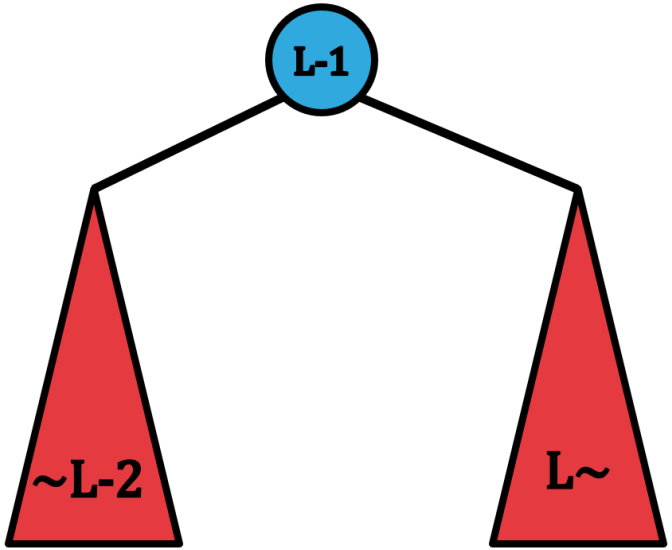
Now Segment Query

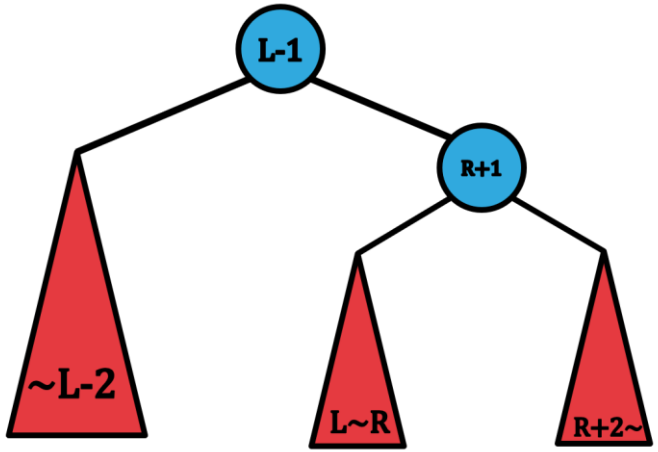
처음에 전체 트리에 대해서 Kth(L-1)을 수행합니다. 그러면 트리는 아래와 같아집니다.



Now Segment Query

오른쪽 서브트리에 대해서 Kth(R-L+1)을 수행하면 오른쪽과 같아져서 [L,R]에 해당하는 노드들을 하나의 서브트리로 모아줄 수 있습니다.





Segment Sum with Splay Tree

이제 구간쿼리를 처리할 수 있게 됐습니다.

대표적인 문제인 구간 합 구하기를 풀어봅시다.

트리의 노드와 pull을 새롭게 아래와 같이 정의합시다.

```
struct Node {
    Node *l, *r, *p;
    int sz;
    ll sum, val;
    Node() : l(nullptr), r(nullptr), p(nullptr), sz(1) {};
    Node(ll val_) {
        l = r = p = nullptr;
        sz = 1;
        // INITIALIZE ADDITIONAL VARIABLES
        val = val_;
    }
};
```

```
void pull(Node *x) {
    x->sz = 1;
    x->sum = x->val;
    if(x->l) {
        x->sz += x->l->sz;
        x->sum += x->l->sum;
    }
    if(x->r) {
        x->sz += x->r->sz;
        x->sum += x->r->sum;
    }
}
```

Segment Sum with Splay Tree

업데이트 쿼리는 업데이트 대상 노드를 Kth로 찾아준 뒤에 val을 변경해줍니다.

구간 합 쿼리는 말한 방식대로 쿼리 구간을 모아주고 구간에 해당하는 서브트리의 루트 노드 값을 출력해주면 됩니다.

Implement Issue

이제 배열이 주어졌을 때 트리를 초기화 하는 방법을 알아보겠습니다.

눈 여겨 볼 점은 실제 배열의 크기는 N이지만 앞뒤로 더미 노드를 추가한 점입니다.

이는 구간을 모아주는 과정을 편하게 하기 위해서 입니다.

```
int N;
Node *root;
int a[1000005];

void init_tree(int N) {
    Node *p;
    root = p = new Node(0);      // DUMMY NODE
    for(int i=1;i<=N;++i) {
        p->r = new Node(a[i]);
        p->r->p = p;
        p = p->r;
    }
    p->r = new Node(0);          // DUMMY NODE
    p->r->p = p;
    p = p->r;
    while(p) {
        pull(p);
        p = p->p;
    }
}
```

Implement Issue

아래는 [l, r]에 해당하는 서브트리를 모아주는 코드의 대략적인 구현입니다.

만약에 l이나 r이 끝에 걸쳐 있으면 Kth의 구현에 따라 문제가 생길 수도 있습니다.

이를 고려해서 구현한다고 치면 좀 골치 아프기 때문에 더미노드를 추가해서 해결합니다.

```
Node* segment(Node *x, int l, int r) {
    Node *p = kth(x, l-1);
    p->r->p = nullptr;
    p->r = kth(p->r, r-l+1);
    p->r->p = p;
    pull(p);
    return p;
}
```

```
void segment(int l, int r) {
    findKth(l-1);
    Node *p = root;
    root = p->right;
    root->p = nullptr;
    findKth(r-l+1);
    p->right = root;
    root->p = p;
    root = p;
}
```

더미노드 있는 버전

```
Node* segment(int l, int r) {
    if(l != 1 && r != N) {
        findKth(l-2);
        Node *p = root;
        root = p->right;
        root->p = nullptr;
        findKth(r-l+1);
        root->p = p;
        p->right = root;
        root = p;
        return root->right->left;
    }
    else if(l == 1 && r == N) {
        splay(ptr[(N+1)/2]);
        return root;
    }
    else if(l == 1) {
        findKth(r);
        return root->left;
    }
    else if(r == N) {
        if(l != r) findKth(l-2);
        else findKth(N-2);
        return root->right;
    }
}
```

더미노드 없는 버전

Lazy Propagation

스플레이트리는 세그먼트 트리가 하는 역할을 대체 가능합니다. 레이지를 뿌려주는 것도 가능합니다. 심지어 이를 이용해서 구간을 뒤집는 것이 가능합니다.

아래와 같이 노드에 lazy 변수를 추가하고 lazy를 뿌려주는 함수 push를 만들어서 노드에 접근할 때마다 push를 호출하면 레이지도 가능합니다.

```
struct Node {
    Node *l, *r, *p;
    int sz;
    ll sum, val, lazy;
    Node() : l(nullptr), r(nullptr), p(nullptr), sz(1) {};
    Node(ll val_) {
        l = r = p = nullptr;
        sz = 1;
        // INITIALIZE ADDITIONAL VARIABLES
        val = val_;
        lazy = 0;
    }
};
```

```
void push(Node *x) {
    if(x->lazy == 0) return ;        // NOTHING TO PROPAGATE
    x->val += x->lazy;
    if(x->l) {
        x->l->sum += x->lazy * x->l->sz;
        x->l->lazy += x->lazy;
    }
    if(x->r) {
        x->r->sum += x->lazy * x->r->sz;
        x->r->lazy += x->lazy;
    }
    x->lazy = 0;
}
```

Lazy Propagation

Kth 함수 내에서 push를 적절하게 호출해주면 레이지를 뿌려주면서 10999번 구간 합 구하기 2를 풀 수 있습니다.

```
Node* kth(Node *root, int k) {
    Node *p = root;
    push(p);
    while(1) {
        while(p->l && p->l->sz > k) {    // WHILE LEFT SUBTREE IS BIGGER THAN K
            p = p->l;                    // GO LEFT
            push(p);
        }
        if(p->l) k -= p->l->sz;
        if(!k) break;
        else --k;                        // CURRENT NODE IS NOT kTH
        p = p->r;
        push(p);
    }
    splay(p);
    return p;
}
```

Reverse Segment

특정 구간을 뒤집는 것 또한 Lazy Propagation으로 해결이 가능합니다. 어떤 구간을 뒤집는 것은 구간 내에 있는 특정 지점을 기준으로 왼쪽 오른쪽을 바꿔주면 가능해집니다. 이를 이용해서 노드에 그 노드가 담당하는 구간이 뒤집혀야 된다는 것을 뜻하는 변수 inv를 추가하고 push를 수정하면 뒤집는 것이 가능합니다.

```
struct Node {
    Node *l, *r, *p;
    int sz;
    ll sum, val;
    bool inv;
    Node() : l(nullptr), r(nullptr), p(nullptr), sz(1) {};
    Node(ll val_) {
        l = r = p = nullptr;
        sz = 1;
        // INITIALIZE ADDITIONAL VARIABLES
        val = val_;
        inv = false;
    }
};
```

```
void push(Node *x) {
    if(x->inv) {
        swap(x->l, x->r);
        if(x->l) x->l->inv = !x->l->inv;
        if(x->r) x->r->inv = !x->r->inv;
        x->inv = false;
    }
}
```

Shifting Segment

구간을 뒤집는 것이 가능해짐으로 특정 구간에 대해서 shift를 하는 것도 가능해집니다.

특정 구간 [l,r]을 오른쪽으로 k만큼 shift한다고 해보죠.

$$a_l a_{l+1} \cdots a_{r-1} a_r \text{ -> } a_{r-k+1} a_{r-k+2} \cdots a_r a_l a_{l+1} \cdots a_{r-k-1} a_{r-k}$$

$a_1 a_2 a_3 a_4 a_5$에서 [1, 5]를 2만큼 오른쪽으로 시프트하면 $a_4 a_5 a_1 a_2 a_3$

이러한 연산은 구간을 뒤집는 연산 세번으로 가능합니다.

Shifting Segment

[l, r]을 k만큼 오른쪽으로 k만큼 shift하려면 세 번 구간을 뒤집는 연산으로 가능해집니다.

[r-k+1, r]을 뒤집고, [l, r-k]를 뒤집고 [l, r]을 뒤집으면 이와 같아집니다.

[1, 5]를 2만큼 shift

1. [4, 5]를 뒤집는다. $a_1a_2a_3a_4a_5$ -> $a_1a_2a_3a_5a_4$
2. [1, 3]을 뒤집는다. $a_1a_2a_3a_5a_4$ -> $a_3a_2a_1a_5a_4$
3. [1, 5]를 뒤집는다. $a_3a_2a_1a_5a_4$ -> $a_4a_5a_1a_2a_3$

왼쪽으로 shift하는 것도 비슷한 방법으로 가능합니다.

Splay Tree in PS

이걸로 스플레이 트리에 관한 내용은 끝났습니다. 세그먼트 트리와 비슷하게 매우 강력한 자료구조입니다. 그리고 세그먼트 트리와 비슷하게 이 자료구조를 어떻게 쓸 지 고민하고 적용하는게 어렵습니다. (저도 못합니다)

여기서 다루진 않았지만 split, join연산으로 구간을 끊고 붙이고 하는 것도 가능합니다.

Link/Cut Tree라는 자료구조의 베이스가 되는 자료구조기도 합니다 .

BOJ 13159 배열

문제 요약

처음에 a_i=i(1-based)인 배열 a가 주어집니다. 해당 배열에 대해서 쿼리 4개를 처리해야 합니다.

1. 구간 [l, r]이 주어지면 해당 구간의 최솟값, 최댓값, 합을 구하고 뒤집는다.
2. l, r, k가 주어집니다. k가 양수라면 [l, r]을 오른쪽으로, 음수라면 왼쪽으로 k만큼 shift합니다.
3. i가 주어지면 현재 배열에서 a_i가 무엇인지를 출력합니다.
4. x가 주어지면 현재 배열에서 x가 어디에 위치했는지를 출력합니다.

BOJ 13159 배열

1번 쿼리와 2번 쿼리는 슬라이드에서 설명한 연산입니다.

3번 쿼리는 Kth(i)를 호출하면 a_i가 루트 노드에 위치하게 됩니다.

4번 쿼리는 초기에 1부터 N까지 각 숫자가 적힌 노드의 포인터를 들고 있습니다. 쿼리가 들어오면 x가 적힌 노드의 포인터가 있으니 이 노드를 스플레이 해주면 해당 노드의 인덱스를 구할 수 있습니다.

Problem Set

- Essential
  - 13159 - 배열
- Practice
  - 3444 - Robotic Sort
  - 13543 - 수열과 쿼리 2
  - 17607 - 수열과 쿼리 31
  - 19589 - 카드 셔플
  - 2844 - 자료 구조
  - 16586 - Linked List

고생 많으셨습니다. ☺