Top Tree로 Dynamic Tree 관리하기

이 글을 읽기 이전에 동적 계획법을 최적화하는 9가지 방법 (4/4) 의 "Dynamic Tree DP" 단원을 이해하는 것이 좋다. 이 글은 해당 내용을 잘 이해하고 있다고 가정하고 설명한다.

그래프의 특수한 경우인 트리는 PS를 포함한 알고리즘 전반에서 자주 활용되는 구조이다. 트리는 일반적인 그래프에 비해 여러 방면으로 효율적으로 관리할 수 있다. 이러한 방법은 그 자체로도 관심의 대상이 되며, 그래프 문제를 풀 때도 다양하게 응용할 수 있다. 너무나도 중요하다는 사실이 잘 알려져 있으니, 구태여 그 중요성에 대해서 긴 글을 쓸 필요는 없어 보인다.

2021년 기준으로, PS에서 다루는 수준의 트리는 대부분 정적이다: 즉, 한번 입력으로 받은 트리는 그 구조가 바뀌지 않는다. 이런 트리를 활용하는 방법은 크게 세 가지가 있다:

- **Euler Tour**는 트리 대신 트리의 오일러 경로 (euler tour) 를 관리한다. 트리의 rooted subtree가 구간으로 나타난다는 성질 때문에, 이것만으로도 서브트리 관련된 쿼리를 자료구조 문제로 바꿀 수 있다. Naive해 보이지만, 생각보다 매우 강력한 방법이다.
- **Heavy-Light Decomposition**은 트리를 여러 직선들의 체인으로 변환한다. 이 체인은, 모든 루트 - 리프 경로가 최대 $\log N$ 개의 체인과만 겹친다는 성질을 가진다. 고로 경로 관련된 쿼리는 여러 체인에 대한 구간 쿼리가 되어서, 역시 자료구조 문제로 바꿀 수 있다.
- **Dynamic Tree DP**는 Euler Tour의 구조가 너무 단순하기 때문에 서브트리 관련된 구조를 완벽하게 처리하지 못한다는 한계점에서 시작한다. Subtree 관련된 정보를 리프에서 루트로 Propagate시킬 수 있다는 것이 핵심 특징이다. 기본적으로 HLD에 기반해 있는데, 각 노드에 대해서 "해당 노드에서 Light edge로 연결된 자식들" 을 적절한 자료 구조 (부분 합, `std::set` 등등...) 로 관리해 준다. 리프에서 루트로 정보를 전파할 때, Light edges들의 자료 구조를 갱신하고, 이를 토대로 체인의 자료 구조를 갱신하다. 이러한 식으로 $\log N$ 번 반복하면 전체 정보가 관리된다. HLD에 기반해 있기 때문에, 경로 쿼리와 서브트리 쿼리가 둘 다 지원된다. *여담으로 널리 사용되는 이름이 아니고, 그렇게 깔끔한 이름도 아니기 때문에, 전세계적으로 통용되는 단어라고 오해하지 않는 것이 좋다.*

실제로는 트리를 *동적* 으로 다루고 싶은 경우가 많다: 즉, 위에서 지원하는 기능들에 더해 트리에서 간선이 추가되고 제거되는 연산을 지원하는 것이다. 위 테크닉들은, 이러한 경우에도 다음과 같이 일반화할 수 있다.

- **Euler Tour Tree (aka ET-tree)** 는 Euler tour를 BBST를 사용하여 관리하는 것을 뜻한다. BBST의 split / merge 연산을 사용하여 간선의 삽입과 삭제를 지원할 수 있으며, 똑같이 서브트리 연산을 지원한다.
- **Link Cut Tree (aka ST-tree)** 는 HLD의 Dynamic 버전이라고 생각할 수 있다. 정확히 Heavy path를 관리하지는 않지만, 각 Splay tree에서 관리하고 있는 path가 사실상 Heavy path의 역할을 한다. 이 역시 서브트리 연산을 지원한다.
- **Top Tree**는 Dynamic Tree DP의 Dynamic 버전이라고 생각할 수 있다. 서브트리 연산과 경로 연산을 모두 지원하므로, 두 트리의 상위 호환이라고 볼 수 있다 (LCT의 상위 호환인 건 명백하고, ET-tree에서도 내가 생각할 수 있는 연산은 모두 지원하나 장담은 못 하겠다). Top Tree는 원래 80년대 즈음 Frederickson이 개발한 *Topology Tree* 에서 시작한 것으로, Dynamic Tree DP와 명확한 연관 관계는 없다. 하지만 2006년 Werneck이 제안한 *rake-compress* 형태의 정의는, Top Tree를 Dynamic Tree DP의 개선된 형태로 볼 수 있는 관점을 제시하였다.

세 자료구조는 각 쿼리를 $O(\log N)$ 에 처리할 수 있다. Amortization이 붙는 것으로 알려진 LCT도, 이를 제거하는 것이 가능하다.

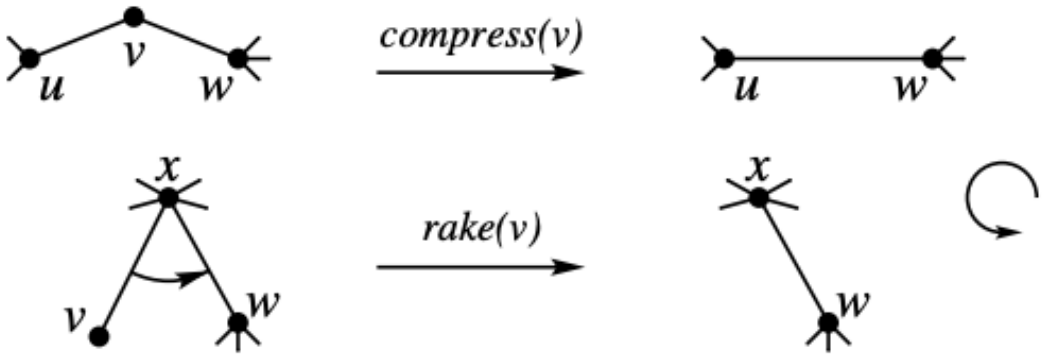
앞 두 트리만 해도 PS에서 잘 알려진 편이지만, 마지막 *Top Tree* 의 경우에는 아직 잘 알려진 자료가 없다. 이 글에서는 Top Tree가 무엇인지를 소개하고, PS에서 사용할 수 있게끔 구현하는 방법을 알려줌으로써 트리 자료구조에 대한 이해를 높이고자 한다. 글의 내용은 대부분 Self-Adjusting Top Trees 논문에 기반해 있다.

정의

이 글에서는 트리 상의 간선의 배열 순서를 중요하게 여긴다. 어떠한 정점에 대해서, 이 정점에 붙은 간선들은 모두 순서가 있다고 생각한다. 달리 말해, 트리가 평면 상에 있다고 생각하고 각 정점에 대해서 인접한 정점의 리스트를 Cyclic하게 가지고 있는 것이다. 일반적으로는, 어떠한 정점에 붙어 있는 간선의 순서를 중요하게 생각하지 않음을 기억하자.

트리 T 가 있을 때, 다음 두 연산은 정점의 개수가 1 감소한 새로운 트리를 만든다.

- `compress(v)` : v 의 차수가 2일 때, v 에 인접한 두 정점 u, w 를 잇고, v 를 지운다. (이 과정에서 Cyclic한 순서를 보존한다.)
- `rake(v)` : v 의 차수가 1일 때 (리프일 때), v 의 sibling (부모의 자식) 중 cyclic한 순서 상에서 인접한 정점을 골라, 해당 정점에 붙인다.



이제 포레스트 F 가 주어졌을 때, F 에 대한 Top Tree를 정의한다. Top Tree는 **클러스터 (cluster)** 로 구성된 Rooted binary forest로, 각 클러스터는

- 두 자식 클러스터를 **Compress**한 것.
- 두 자식 클러스터를 **Rake**한 것.
- 자식이 없고, 실제 포레스트 F 의 간선에 대응되는 리프 클러스터 (**Base Cluster**)

리프가 아닌 자식 클러스터는, Compress / Rake 과정에서 생긴 새로운 간선에 대응된다고 볼 수 있다. 즉, 각 클러스터는 위 Compress / Rake 연산으로 줄여나가는 과정에서 등장하는 간선에 대응된다. 클러스터를 compress, rake한다는 것은 이 맥락에서 해석할 수 있다.

사용자는 Top Tree에 대해서 다음과 같은 public function을 호출할 수 있다. 이 함수는 $O(\log N)$ 시간에 작동한다. 이 글에서는 Amortized bound를 맞추는 방법에 대해서만 논의하며, Worst-case bound는 가능하지만, 논의하지 않는다.

- `link(v, w)` : v, w 가 주어졌을 때, 둘이 연결되어 있지 않다면 (v, w) 를 잇는 간선 추가.
- `cut(v, w)` : v, w 가 주어졌을 때, 둘을 잇는 간선이 있다면 (v, w) 를 잇는 간선 제거.
- `expose(v, w)` : v, w 가 주어졌을 때, 둘이 연결되어 있다면 루트 클러스터가 (v, w) 를 직접 잇는 간선에 대응되게끔 수정.

Expose 함수는, 위 Compress / Rake의 최종 결과가 특정 간선이 되게끔 강제하는 효과를 가진다.

여기까지 봤을 때, 우리는 Top Tree를 포레스트에 대한 이진 탐색 트리 (BST) 라고 생각할 수 있다. 예를 들어, 관리하는 트리가 직선의 형태라면, Link / Cut 연산은 Merge / Split 연산에 대응되며, Expose 연산은 특정 구간을 자르는 연산에 대응된다.

BST의 이점을 가져가기 위해서는, Top Tree의 깊이가 작아야 한다. 임의의 트리에 대해서, 항상 $4 \log N$ 의 깊이를 가지는 Top Tree를 구성할 수 있다. 하지만 이 글에서는 이에 대해서 다루지 않고, 대신 Splay tree / Link cut tree와 동일한 느낌의 *self-balancing* 전략을 사용한다. LCT가 $O(\log N)$ amortized bound인 것을 증명하지 않듯이, 이 글에서도 이 전략을 사용할 때 $O(\log N)$ amortized bound가 나오는 것을 엄밀히 증명하지 않는다. 대단히 귀찮고, Splay / LCT에 익숙한 독자들에게는 거의 직관에 어필할 수 있는 내용이기 때문이다.

초기화

이 단락에서는 트리 T 가 주어졌을 때, T 에 대한 Top Tree를 어떻게 만드는지를 소개한다. 이 부분은 실제 구현에 필요한 것이 아니라 단순히 개념을 설명하기 위함으로, 실제로는 단순히 `link` 연산을 $n-1$ 번 수행하면 되기 때문에 굳이 시간을 들여 초기화할 필요가 없다.

트리 T 에서 아무 리프 z 를 잡아서 루트로 만든 후, 트리의 간선 집합 $E(T)$ 를 루트 방향으로 가는 path들의 집합으로 분할하자. 각 path는 리프에서 출발하여, 다른 path의 끝에서 끝나거나, 아니면 루트 (z) 에서 끝난다. 루트에서 끝나는 케이스는 단 하나이며 이 경우를 root path라고 부른다. Root path를 제외한 모든 path에 대해서, 해당 path에 대한 "parent path" 가 있다고 생각할 수 있다: 이는 해당 path의 끝이 닿아 있는 다른 path를 뜻한다. 이렇게 생각할 경우, 트리의 path들 역시 트리를 이룬다.

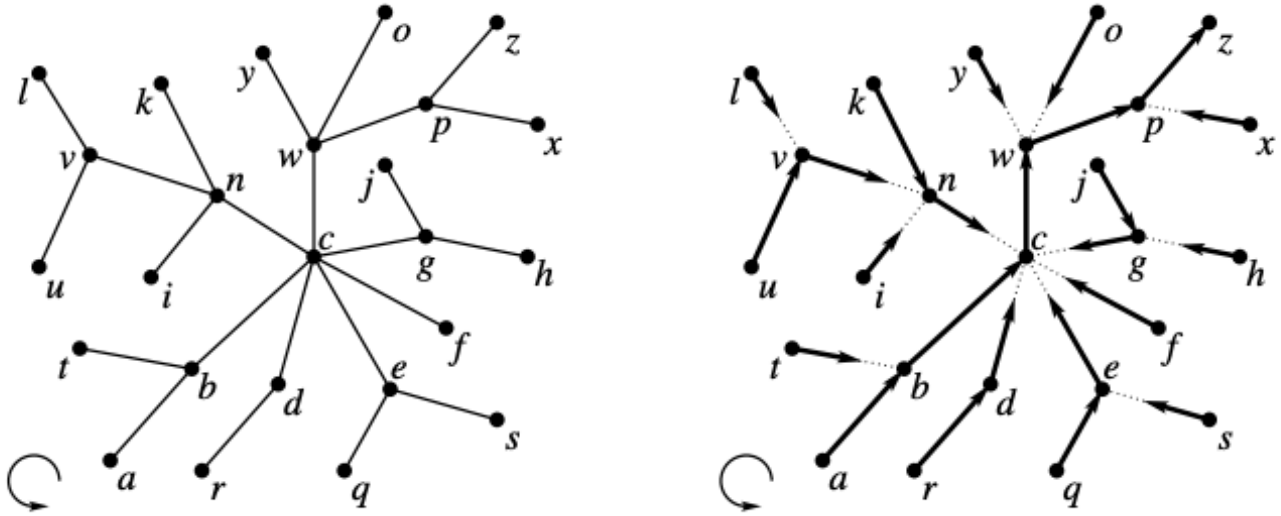


Figure 11: Example: Original tree and directed version (rooted at z and partitioned).

이제 Root path에서부터 재귀적으로 내려가면서 Top tree를 초기화한다. 초기화는, 트리 전체를 하나의 Root cluster로 묶는 것이다.

위에서 Path들 역시 일종의 트리를 이룬다고 설명했다. 현재 처리하고 있는 경로를 p 라고 하자. p 에 대해서, p 의 정점에서 끝이 나는 다른 경로들이 있을 것이다. 이들은 path들로 만든 트리에서 자식 노드에 대응된다. 각 자식들에 대해서 재귀적으로 초기화를 진행한다. 초기화를 할 경우, 자식 서브트리는 하나의 간선으로 압축되어 있을 것이다.

이제 p 에 어떠한 클러스터들이 달려 있다. 이 클러스터들 역시 자식 간의 *순서* 를 지키기 때문에, 경로 왼쪽에 있는 클러스터와 오른쪽에 있는 클러스터는 분리되어 있다고 생각할 수 있다. p 의 왼쪽에 달린 클러스터들을 **rake** 연산으로 합쳐주고, 오른쪽에 달린 클러스터들을 **rake** 연산으로 합쳐주자. (Self-balancing할 것이니 상관 없지만, 결국 rake 연산으로 합친 것도 최대한 밸런스가 유지되도록 하는 것이 좋다.) 만약 p 의 길이가 k 라면, 이후 최대 $2k-2$ 개의 클러스터가 경로의 양 옆에 붙어 있다.

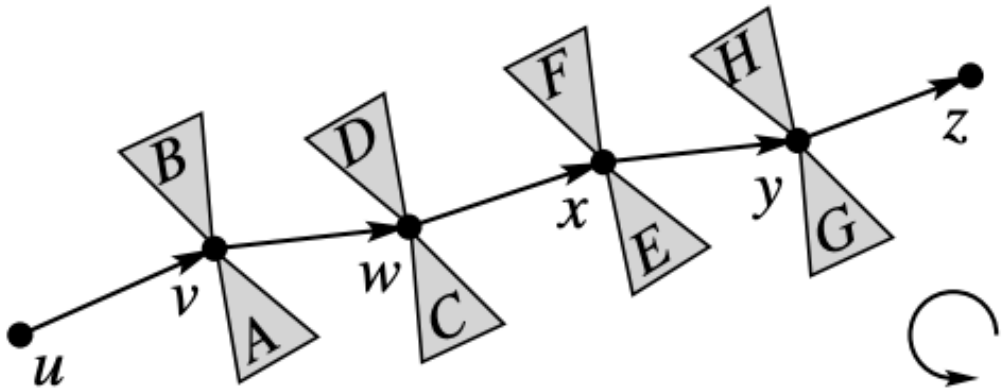
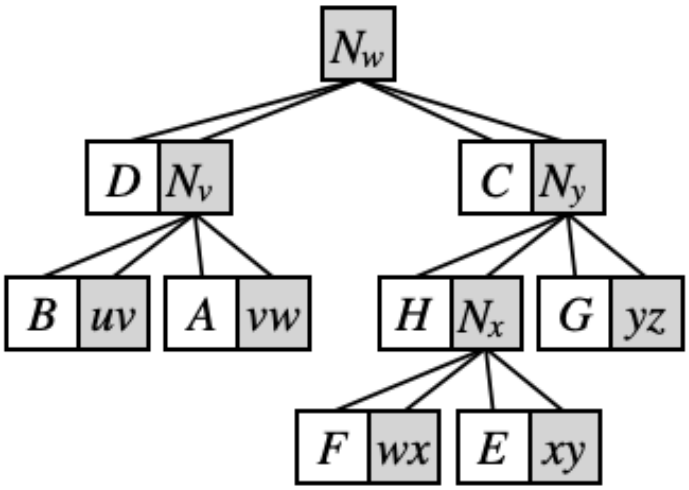


Figure 2: A unit tree rooted at z.

p 와 클러스터들을 하나의 클러스터로 합치기 전에, p 에 달린 클러스터가 없고 경로만이 존재하는 케이스를 생각해 보자. 이 경우, 경로 상 홀수 번째 정점들을 반복적으로 **compress**하는 것으로, p 를 쉽게 하나의 클러스터로 만들어 줄 수 있다.

p 에 달린 클러스터가 존재할 경우, **rake** 연산을 통해서 이 클러스터들을 지워줘야 한다. 이들을 지울 때는, 어떠한 정점 v 가 Compress되기 직전에 v 에 붙은 두 클러스터를 rake하는 식으로 처리한다. 예를 들어, p 위에 경로 $u-v-w$ 가 있고, v 의 왼쪽과 오른쪽에 클러스터 A, B 가 있다고 하자. v 를 Compress하기 직전에, A 를 $v-w$ 에 rake하고, B 를 $u-v$ 에 rake한 후, v 를 compress한다. 이 경우 경로 상에서의 Compress 연산 하나는 총 4개의 클러스터를 표현한다고 할 수 있다. 이를 표현하는 것이 아래의 그림인데, N_v 라는 노드의 자식이 4개가 있고, 이 중 Compress 직전에 Rake한 클러스터들끼리는 붙어 있음을 볼 수 있다. 흰색으로 표기된 노드들은, 맨 처음에 압축한 Rake tree들이다.



이렇게, Top tree의 구성에 대해서 살펴봤다. Dynamic Tree DP의 관점에서 살펴봤을 때, Compress tree는 각 Heavy chain에 대해서 관리한 자료구조이고, Rake tree는 각 정점마다 Light edge들을 관리하는 데 사용되는 자료구조에 대응됨을 알 수 있다. Top tree는 Compress tree의 각 노드마다 Rake tree를 적절히 붙이는 방식으로, 이러한 두 다른 역할의 자료구조를 최대한 일관성있게 관리하려고 한다.

업데이트

이제 실제 구현에 사용되는 업데이트들을 어떻게 구현하는지에 대해서 살펴본다.

먼저 정점에 대한 표현을 지원하기 위해 각 정점의 **Handle** 을 정의한다. 기본적으로, 리프가 아닌 노드 v 에 대해서 Handle N_v 는, $compress(v)$ 가 진행 된 노드를 뜻한다. 리프의 경우에는, v 를 끝점으로 가지는 top-most non-rake 노드이다. 즉, v 에서 시작하는 Path의 루트 노드라고 생각하면 좋다. 이 글에서는 정점을 표현할 때 Handle 이라는 개념을 사용하지만, 실제로 구현에서는 다양한 방법이 있을 수 있다. 예를 들어서, 각 정점마다 Dummy node를 만들어서, 해당 노드가 항상 Top tree의 리프에 와 있다는 Invariant를 지키는 선에서 구현하는 방법이 그 중 하나이다.

Splay 연산은 표현되는 트리를 바꾸지 않으면서 어떠한 노드를 루트로 바꾸는 과정이다. Splay 연산을 할 때 중요한 점은, 각 Rake tree / Compress tree의 Scope를 벗어나지 않는다는 점이다. 다른 말로, splay 연산을 통해서 어떠한 노드를 Top tree의 루트로 바꾸는 것이 아니고, 각 Rake/compress tree의 루트로 바꾸는 것이다. 이러한 형태의 Splay를 **Guarded splay** 라고 부른다. 약간 Naive해 보이지만, 이 정도로도 Top tree의 시간 복잡도를 amortized $O(\log N)$ 으로 보장할 수 있다. 이것을 제외하고 Splay 연산에 대해 특이한 점은 없다. 일반적인 Splay tree와 같이 rotate 연산을 반복 적용하면 된다.

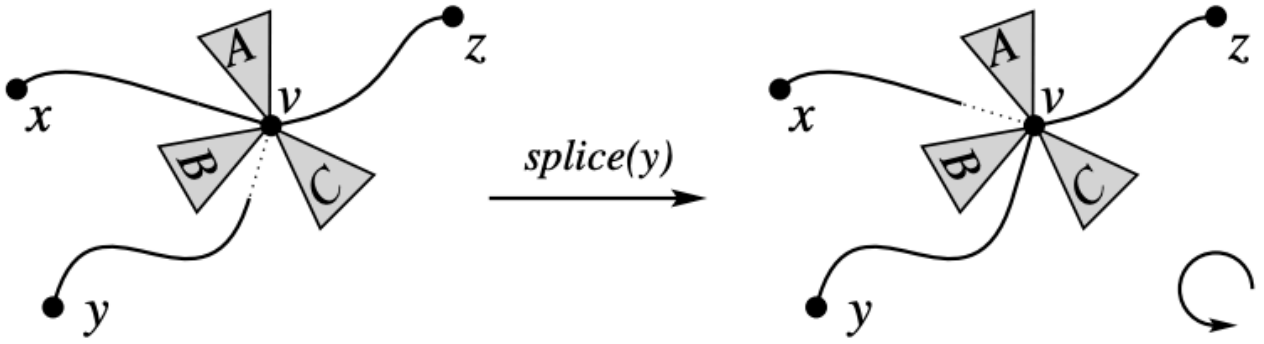


Figure 6: Splice: $y \cdots v \cdots z$ replaces $x \cdots v \cdots z$ as the exposed path.

Splice 연산은 Link-cut tree의 **Access** 연산을 atomic하게 적용한 버전이라고 생각하면 좋다. $p_1, p_2, \ldots, p_k$ 를 루트에서 v 로 가는 internal path들의 수열이라고 하자. Splice 연산은 p_{k-1} 을 p_k 에 인접한 지점을 기준으로 두 조각으로 자른 후, 루트에 가까운 조각을 p_k 에 붙인다. 이렇게 되면 internal path의 수열의 길이가 하나 줄어든다.

Expose 연산은 $v - w$ 간의 경로를 추출하는 방법이다. **Soft Expose** 는 경로를 루트 바로 근처 (루트의 오른쪽-> 왼쪽, 왼쪽 -> 오른쪽 자식) 에 두는 것이고, **Hard Expose**는 잠시 연결을 끊으면서까지 경로를 루트로 만드는 것이다. 이 글에서는 Soft Expose만 설명한다 (Hard expose를 굳이 구현할 필요가 별로 없다). 일단 N_v 를 Splice하고, 중간 경로를 뒤집는 식으로 v 가 루트가 되도록 rerooting하자. 이렇게 되면, N_w 를 Splice하면 경로를 추출할 수 있다.

Cut 연산은, 자를 간선 $v - w$ 를 Soft expose함으로써 쉽게 구현할 수 있다. 정확히는, 자를 간선을 Soft expose한 후, 간선의 부모 (둘 다 Vertex handle이다) 와 간선 사이의 연결을 끊는다. 이제 두 handle에 대해서 한 쪽 Child가 비어있게 되는데, 지운 간선과 Cyclic하게 인접한 두 간선을 해당 Child에 붙여주면 된다. **Link** 연산에서는 이를 반대로 하면 된다.

Top Tree를 사용하여 해결할 수 있는 문제들

이 단락에서는 Top Tree를 사용하여 간단한 문제들을 해결해 봄으로써, 이 자료구조가 어떻게 문제 해결에 도움이 되는지를 알아본다. Top Tree를 사용하면, 간선이 삽입 / 삭제 되는 트리에 대해서 다음과 같은 문제들을 해결할 수 있다.

문제 1. 모든 간선의 가중치 합을 계산하라.

증명. 각 노드에 대해서, $sum(v)$ 라는 값을 관리하자. 이는 해당 노드의 서브트리에 있는 가중치의 합을 뜻한다. (이 문제는 물론 자명한 문제이며, 서브트리 값을 가져올 때 클러스터 타입도 중요하지 않다.)

문제 2. 경로 $v - w$ 상에 속하는 간선의 최대 가중치를 계산하라.

증명. 경로 $v - w$ 를 Expose하면, 해당 Path 안에 있는 간선들의 가중치 최대를 계산하면 된다. Compress cluster에서 온 가중치의 최댓값 $pathMax$ 를 bottom-up으로 모아주면 된다. Rake cluster에서 온 가중치는 무시해 주자.

문제 3. 경로 $v - w$ 상에 **속하지 않는** 간선의 최대 가중치를 계산하라.

증명. 문제 2의 풀이에 $ignoredMax$ 라는 값을 추가로 두어서, Rake cluster에서 버려진 값 ($pathMax, ignoredMax$) 들을 모아준다. 값을 무시할 때마다 모아준다고 생각하면 된다.

Remark. 문제 2/3의 풀이는 최댓값이 아닌 합에 대해서도 당연히 적용된다.

문제 4. 경로 $v - w$ 상의 간선에 특정 상수 c 를 더하거나, 최댓값을 계산하는 쿼리를 해결하라.

증명. 이제 Lazy propagation이 추가되지만, 원리는 동일하다. Compress cluster 자식에만 값을 전파해 주고, Rake cluster 자식에는 하지 말자.

문제 5. 트리의 지름을 계산하라.

증명. 각 노드에 대해서 1) 지름, 2) 경로의 길이, 3) 경로의 각 끝점에서 가장 먼 점과의 거리 를 관리하자. 두 경우 모두, 두 클러스터가 공유하는 정점이 정확히 하나 있다. 클러스터 안에서 이 정점을 지나는 가장 긴 경로는 2) 정보를 통해서 알 수 있고, 고로 지름을 구해줄 수 있다. 지름 외 정보는 쉽게 계산할 수 있다.

문제 6. 각 정점을 mark / unmark 하는 쿼리, 그리고 각 정점에 대해서 해당 정점과 가장 가까운 marked 정점과의 거리를 찾을 수 있다.

증명. 각 정점에 대해서 더미 간선을 만들어 주자: 그냥 해당 정점에서 리프 하나를 더 뻗어주면 된다. 각 간선에 대해서, 양 끝점에 가장 가까운 마킹된 정점과의 거리, 그리고 경로 길이를 계산해 주자. 이 둘은 각 클러스터 종류에 따라 쉽게 합쳐줄 수 있다. 쿼리가 들어올 때는, 각 정점에 연계된 더미 간선을 expose하면 된다. Mark는 양 끝점의 거리를 0으로, Unmark는 ∞ 로 설정하자.

문제 6까지는 모두 *local* 한 성질에 대해서만 다뤄봤다. 어떠한 성질이 *local* 하다는 것은, 해당 성질이 서브트리의 값만 가지고 유추 가능하다는 것을 뜻한다. 예를 들어서, 서브트리 합은 Local한 성질이다. 두 자식의 서브트리 합만 알면 구할 수 있기 때문이다.

일반적인 BST로 구하고자 하는 모든 정보가 *local* 한 것은 아니다. 예를 들어서, 배열의 k 번째 원소를 접근하는 것은 Local하지 않다. 자식의 k 번째 원소랑 부모의 k 번째 원소와는 연관이 없으며, 연관을 가지기 위해서는 훨씬 더 큰 context가 필요하기 때문이다. 일반적인 BST에서는, 이를 Top-down으로 루트에서부터 트리를 타고 내려가면서 해결할 수 있다. 만약 Top-tree에서도 이것을 할 수 있다면, 흥미로운 문제들을 많이 해결할 수 있다. 예를 들어서, Level ancestor를 계산하여 LCA를 찾거나, 트리의 Centroid를 단순 이진 탐색으로 구할 수 있다.

일단 Level ancestor를 계산하는 것은 단순 Expose 후 정말 BST에서 하는 것처럼 클러스터를 타고 내려가면 된다. 이 부분에 대해서는 설명을 생략한다. 다른 케이스들의 경우, Top Tree에서는 *choice* 라는 함수를 통해서 이러한 성질을 Abstraction한다. 이에 관해, 아래에 몇 가지 Lemma를 소개한다.

- **Lemma (Center)**. 트리 T 와 두 인접한 클러스터 A, B가 $A \cap B = \{c\}$, $A \cup B = T$ 를 만족하면, 다음이 성립한다. $max\_dist(A, w)$ 가 임의의 점 $v \in A$ 와 w 간의 최대 거리라고 했을 때, $max\_dist(A, c) \geq max\_dist(B, c)$ 일 경우, A 는 모든 Center를 포함한다.
- **Lemma (Centroid)**. 위와 같이 정의하자. 만약 $|A| \geq |B|$ 이면 A 가 T 의 Centroid를 포함한다.

이제 루트 클러스터가 주어졌을 때, 우리는 루트 클러스터의 자식 A, B 에 대해서 어느 쪽이 우리가 원하는 정점을 가지고 있는지를 알 수 있다. 즉, 해집합의 크기를 반으로 줄일 수 있다. 이 함수를 $choice$ 라고 부르자. 여기서 착각하면 안 되는 것이, *루트 클러스터* 에 대해서만 위 정보를 알 수 있기 때문에, 반복적으로 해집합의 크기를 반으로 줄일 수 있는 것이 아니다.

해집합을 반복적으로 반으로 줄이기 위해서는 Top Tree를 수정할 수 있어야 한다. 해집합을 한 클러스터에 관리하고, 이를 C 라고 부르자. 초반에는 $C = T$ 이며, 이를 반으로 줄이는 것은 $choice$ 함수를 사용하면 된다. 이후에도 반복적으로 해집합을 줄이기 위해서는, C 클러스터의 자식 (A, B) 를 변경해서, 이들이 루트 클러스터 $R = (A', B')$ 의 서로 다른 자식으로 들어가게끔 변경해 주면 된다. 이렇게 하면, $choice(R)$ 의 반환값을 토대로 $C \leftarrow A$ 인지 $C \leftarrow B$ 인지를 판별할 수 있고 전체 문제가 해결된다.

연습 문제

- BOJ 17936. 트리와 쿼리 13
- BOJ 20148. 트리와 쿼리 18
- BOJ 1921. 트리와 쿼리 20
- Yosupo Judge. Dynamic Tree Subtree Add Subtree Sum
- Codeforces 1172E. Nauuo and ODT