수학과 2020540023 이명규

1. AVL tree

1. 정의: AVL 트리는 (1) 모든 노드에 대해, 각각의 서브 트리가 최대 1만큼의 높이 차이를 가지고, (2) 모든 서브 트리가 AVL 트리인, 이진 검색 트리이다. [1]

2. 특징

(1) AVL tree에서, 임의의 노드의 서브 트리의 높이는 최대 1만큼의 차이를 가지며, 이를 높이균형성질이라고 한다. [2]

(2) 노드의 균형 인자는 '오른쪽 서브 트리의 높이 - 왼쪽 서브 트리의 높이'이며, 1, 0, -1의 값을 가지는 노드는 균형을 이룬다고 한다. [2]

3. 알고리즘(연산) 및 시간복잡도

(1) 탐색

① 트리의 루트 노드부터 탐색을 시작하며, 노드의 데이터가 찾고자 하는 키와 같은 지 비교한 후 같으면 데이터를 반환하고 탐색을 마치고, 작으면 왼쪽 포인터가 가리키는 노드로 이동한 후 다시 비교한다.

② 또한 클 경우에는 오른쪽 포인터가 가리키는 노드로 이동한 후 다시 비교한다. [3]

(2) 삽입

① 탐색에 의해 데이터가 삽입될 위치를 결정한 후, 데이터를 삽입한다.

② 데이터를 삽입한 후, 데이터가 삽입된 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.

③ 각각의 노드에 서브 트리의 깊이를 조정하는 과정에서 노드의 균형 인자의 절댓값이 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 트리의 균형을 맞춘다. [3]

(3) 삭제

① 탐색에 의해 삭제하려는 데이터를 갖는 노드 A를 결정하며, 노드 A가 리프이면 노드를 삭제한다.

② 노드 A가 리프가 아니면 삭제하려는 데이터의 바로 다음 크기의 데이터를 포함하는 노드 B와 맞바꾼 후 삭제한다. 이때 맞바꾸는 노드 B는 삭제될 노드 A의 오른쪽 서브 트리 중에서 가장 작은 데이터를 갖는 노드이다.

③ 데이터를 삭제한 후, 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.

④ 각각의 노드에 대해 서브 트리이 깊이를 조정하는 과정에서 노드의 균형 인자의 절댓값이 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 트리의 균형을 맞춘다. [3]

(4) 트리의 균형

① 트리에 노드가 추가되거나 삭제될 때마다 다음과 같이 트리의 균형을 맞추도록 한다.

② 리프에서 루트까지의 탐색 경로에 위치한 모든 노드에 대해 각각의 노드에 유지되는 균형 인자를 조사하여 절댓값이 2 이상 차이가 나면 방향 정보에 따라 적용할 회전 규칙의 종류를 결정한다.

③ 회전 규칙의 종류는 (그림 1)과 같이 RR, LL, RL, LR로 구분될 수 있으며, 트리의 불균형을 일으키는 노드의 서브 트리 방향을 이용하여

만들어지는데, LL은 회전이 적용될 노드의 Left Left 서브 트리의 깊이가 더 깊다는 것을 의미한다.

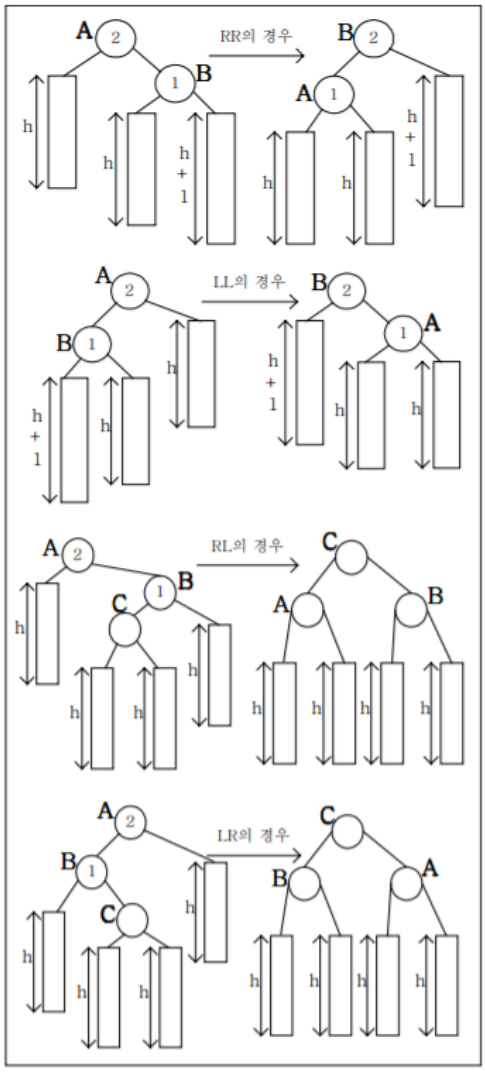

그림 1: AVL 트리의 회전 규칙 [3]

④ RR과 LL은 단순 회전에 의해 트리의 재균형을 유지할 수 있으나 RL과 LR의 경우에는 트리의 재균형을 유지하기 위해 회전이 두 번 적용된다. [3]

(5) 회전 규칙

① RR의 경우는 균형 인자가 2인 노드 A를 균형 인자가 1인 노드 B의 왼쪽 자식 노드가 되도록 하고, 노드 B의 왼쪽 서브 트리를 노드 A의 오른쪽 서브 트리가 되도록 한다.

② LL의 경우는 균형 인자가 2인 노드 A를 균형 인자가 1인 노드 B의 오른쪽 자식 노드가 되도록 하고, 노드 B의 오른쪽 서브 트리를 노드 A의 왼쪽 서브 트리가 되도록 한다.

③ RL의 경우는 균형 인자가 1인 노드 B에 대해 LL 회전 규칙을 적용한 후 균형 인자가 2인 노드 A에 대해 RR 규칙을 적용한다.

④ LR의 경우는 균형 인자가 1인 노드 B에 대해 RR 회전 규칙을 적용한 후 균형 인자가 2인 노드 A에 대해 LL 규칙을 적용한다. [3]

(6) 시간복잡도: 탐색, 삽입, 삭제의 경우 O(log n), 회전의 경우 O(1)의 시간이 걸린다. [4]

4. 장단점: RB tree 등에 비해 탐색 연산이 빠르고, 균형능력(balancing capabilities)이 우수하다. 하지만 삽입과 삭제는 회전이 일어나면서 수행시간이 느리기 때문에, 빠르게 업데이트 되는 시스템에서는 적절하지 않다. [5]

5. 응용 분야: 삽입과 삭제 연산보다는 탐색 연산이 더 필요한 경우에 많이 쓰인다. 예를 들어, 삽입 및 삭제 횟수는 적지만 필요한 데이터를 자주 검색하는 데이터베이스 응용 프로그램에서 자주 쓰인다. 이와는 별도로 향상된 검색이 필요한 응용 프로그램에서도 사용된다. [6] 또한 언어 사전이나 어셈블러, 인터프리터의 명령 코드(opcode) 등 프로그램 사전(program dictionaries) 같이 한번 만들어지면 재구성되지 않는 경우에 쓰인다. [7]

2. RB tree (Red-Black)

1. 정의: 레드-블랙 트리는 모든 노드에 대해 레드 또는 블랙이라는 색 정보를 가지는 자가

균형 이진 탐색 트리를 의미하며, 아래 조건을 만족해야 한다. [8]

(1) 모든 노드는 레드나 블랙의 색을 가진다.

(2) 루트 노드는 블랙이다.

(3) 모든 리프 노드(NULL)는 블랙이다.

(4) 레드 노드의 두 자식은 모두 블랙이다.

(5) 모든 노드에 대해, 하위의 리프 노드까지의 모든 경로는 같은 수의 블랙 노드를 포함한다.

2. 특징

(1) 루트 노드부터 가장 먼 리프 노드 경로까지의 거리가, 가장 가까운 리프 노드 경로까지의 거리의 두배보다 항상 작다. 즉 레드-블랙 트리는 개략적으로 균형이 잡혀 있다. [9]

(2) 자료의 삽입과 삭제, 검색에서 최악의 경우에도 일정한 실행 시간을 보장한다. 이는 실시간 처리와 같은 실행시간이 중요한 경우에 유용하게 쓰일 뿐만 아니라, 일정한 실행 시간을 보장하는 또 다른 자료구조를 만드는 데에도 쓸모가 있다. [9]

3. 알고리즘(연산) 및 시간 복잡도

(1) 탐색: 레드-블랙 트리는 이진 탐색 트리의 특수한 한 형태이기 때문에, 일반적인 이진 탐색 트리와 동일하게 동작한다. [9]

(2) 회전: 회전은 탐색 트리에서 중위 순회의 순서를 보존하는 지역적인(local) 연산이다. [10]

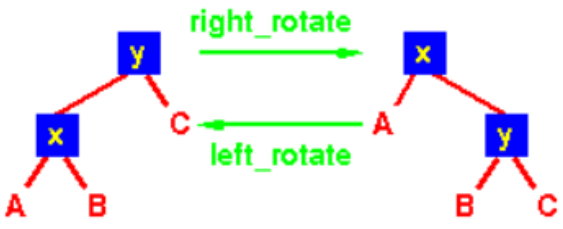

그림 2: 회전(Rotations) [10]

(3) 삽입: 다른 이진 탐색 트리와 동일하게 삽입하되, 색은 레드로 칠한다. 또한, N은 현재 노드, P는 N의 부모, G는 N의 조부모, U는 삼촌 노드라고 하자. [2]

① N이 루트라면, 정의 2를 만족하기 위해 블랙으로 칠한다.

② P가 블랙이라면, 정의 4와 5를 만족한다.

③ P, U 모두 레드라면, 둘을 블랙으로 칠한다.

④ P가 레드, U가 블랙이면, P에 left 회전을 하고, ⑤를 실행한다.

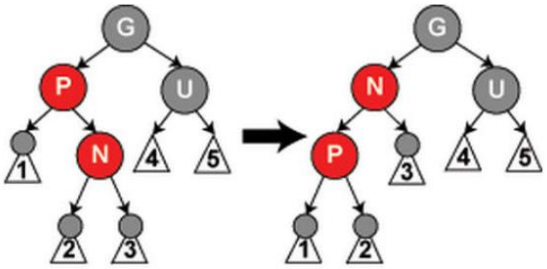

그림 3: ④번 경우 [2]

⑤ right 회전을 하고, P와 G를 다시 칠한다. [2]

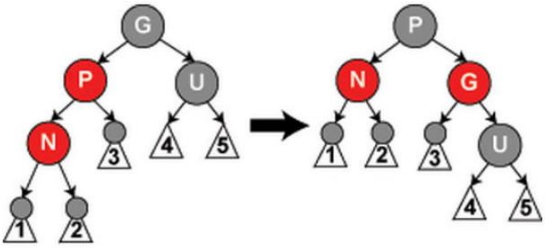

그림 4: ⑤번 경우 [2]

(4) 삭제:

① 삭제하려는 노드가 레드인 경우, 어떠한 추가 작업도 필요하지 않다.

② 삭제하려는 노드가 블랙인 경우, 그 자리를 대체하는 노드를 블랙으로 칠한다. 대체 노드가 레드라면 블랙으로 칠하는 것으로 충분하다.

③ 대체 노드가 블랙인 경우, 해당 대체 노드

를 이중흑색노드라고 한다. 이중흑색노드의 형제가 레드인 경우에는 형제를 블랙, 부모를 레드로 칠한 후, 부모 노드를 기준으로 left 회전함으로써 형제를 블랙으로 만들어준다. 이후 아래 경우 중 해당하는 것에 따라 처리한다.

④ 이중흑색노드의 형제가 블랙이고, 형제의 양쪽 자식 노드 모두 블랙인 경우 형제 노드만 레드로 만들고, 이중흑색노드의 검은색 1개를 부모에게 전달한다. 부모가 블랙이었다면, 마찬가지로 ④~⑥ 중 하나로 처리한다.

⑤ 이중흑색노드의 형제가 블랙이고 형제의 왼쪽 자식이 레드, 오른쪽 자식이 블랙인 경우, 형제를 레드로, 형제의 왼쪽 자식을 블랙으로 칠한 후에 형제 노드를 기준으로 우회전한다.

⑥ 이중흑색노드의 형제가 블랙이고 형제의 오른쪽 자식이 레드인 경우, 부모 노드의 색을 형제에게 넘긴다. 부모 노드와 형제 노드의 오른쪽 자식을 블랙으로 칠한다. 부모 노드 기준으로 left 회전한다. [11]

(5) 시간복잡도: 탐색, 삽입, 삭제의 경우 O(log n)의 시간이 걸린다. [8]

4. 장단점: 자가 균형적이기 때문에 연산들이 O(log n) 만에 수행됨이 보장되며, 특히 삽입이나 삭제가 빈번히 일어날 때 유용하다. 하지만 고려할 경우가 많아서 구현이 상대적으로 복잡하며, 한번 만들고 탐색 연산만을 이용하려는 경우에는 AVL 트리가 더 효과적이다. [12]

5. 응용 분야: Java의 TreeMap과 TreeSet, C++ STL: map, multimap, multiset, 그리고 Linux kernel: completely fair scheduler, linux/rbtree.h 등을 구현하는 데 쓰인다. [13]

3. Splay tree

1. 정의: 최근에 접근한 노드를 루트에 위치시켜서, 최근에 방문한 노드에 접근하는 데 걸리는 시간을 줄여주는 이진 탐색 트리이다. [8]

2. 특징: splay 트리는 회전을 통해, 마지막을 방문한 노드를 루트에 위치시키는 splaying이라는 과정을 거친다. [8]

3. 알고리즘(연산) 및 시간 복잡도

(1) splaying

① 노드 x에 접근했을 때, splay 연산을 통해 x를 루트 노드로 이동시킨다. [14]

① 노드 x를 splay한다고 하자. 이때 부모와 조부모를 (존재한다면) 각각 p와 g라고 하자. [8]

② x가 루트 노드의 자식이라면, x에서 p로의 간선에서 회전한다. (단순 회전, 혹은 zig) [8]

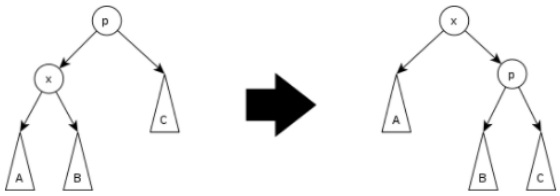

그림 5: Simple rotation or zig [14]

③ P가 루트 노드가 아니고, x와 p가 둘 다 오른쪽이나 왼쪽 자식일 때는, p에서 g로의 간선, 그리고 x에서 p로의 간선에서 회전한다. [8]

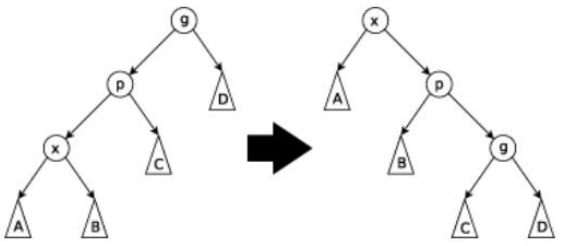

그림 6: Zig-Zig or Zag-Zag [14]

④ P가 루트 노드가 아니고, x와 p가 서로 다른 방향의 자식일 때는, p에서 x로의 간선, 그리고 g에서 x로의 간선에서 회전한다. [8]

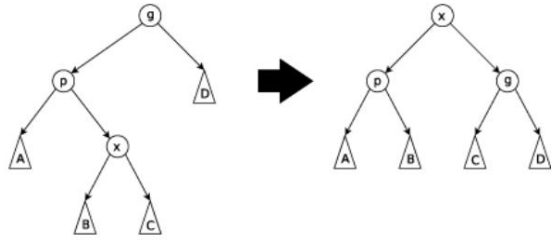

그림 7: Zig-Zag [14]

(2) 삽입 [14]

① 일반적인 이진 탐색 트리와 마찬가지로, 값 x를 삽입한다.

② 삽입 이후, splay가 수행된다.

③ 그 결과, 새롭게 삽입된 노드 x가 트리의 루트가 된다.

(3) 삭제 [14]

① x를 삭제하기 위해서, 이진 탐색 트리와 동일한 방법을 수행한다.

② x가 두 자식 노드를 가진다면, x의 값과 왼쪽 서브 트리의 가장 오른쪽 값(혹은 오른쪽 서브 트리의 가장 왼쪽 값)을 교체한다.

③ 대신 해당 노드를 삭제한다.

(4) 시간복잡도: 탐색, 삽입, 삭제를 amortized O(log n)에 처리한다. [14]

4. 장단점: 평균적인 경우에는 다른 트리들만큼 효율적이며, 자주 쓰이는 데이터를 루트 근처에 두는 것은 캐시, 가비지 커렉팅 알고리즘의 구현 등에 효과적이다. 하지만 트리의 높이가 선형적일 수 있으며, 접근 패턴이 랜덤일 경우에는 다른 덜 동적인 방법들보다 큰 상수 인자를 가질 수 있다. [14]

5. 응용 분야: Splay 트리는 최근에 방문한 노드에 빠르게 접근할 수 있다는 장점 덕분에, 캐시와 가비지 컬렉션에서 주로 쓰인다. [8]

4. 참고문헌

[1] The University of Auckland - Data Structures and Algorithms,

<https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html>

[2] Cal Poly Pomona – Lecture Notes: Self-Balancing Binary Search Tree,

<https://www.cpp.edu/~ftang/courses/CS241/notes/self%20balance%20bst.htm>

[3] 홍기채, 문병주. (2001). 메모리 기반의 인덱스 기법에 관한 연구. 전자통신동향분석, 16(6): 29-40

[4] 위키백과(AVL 트리),

<https://ko.wikipedia.org/wiki/AVL_%ED%8A%B8%EB%A6%AC>

[5] NYLN,

<https://nyln.org/avl-tree-pros-and-cons-list>

[6] Software Testing Help,

<https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/>

[7] Quora, <https://www.quora.com/What-are-some-practical-applications-of-AVL-trees-and-splay-trees>

[8] Azar, E., & Eguiluz Alebicto, M. (2016). Swift Data Structure and Algorithms. Packt Publishing.

[9] 위키백과(레드-블랙 트리),

<https://ko.wikipedia.org/wiki/%EB%A0%88%EB%93%9C-%EB%B8%94%EB%9E%99_%ED%8A

%B8%EB%A6%AC>

[10] The University of Auckland - Data Structures and Algorithms,

<https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html>

[11] 어소트락 게임아카데미,

< https://assortrock.com/88>

[12] Quora, <https://www.quora.com/What-is-the-efficiency-of-Red-Black-Trees-What-are-the-advantages-and-disadvantages-of-RB-Trees>

[13] Stack Overflow,

<https://stackoverflow.com/questions/3901182/applications-of-red-black-trees>

[14] Wikipedia,

<https://en.wikipedia.org/wiki/Splay_tree>