● BST는 어떤 자료 구조인가?

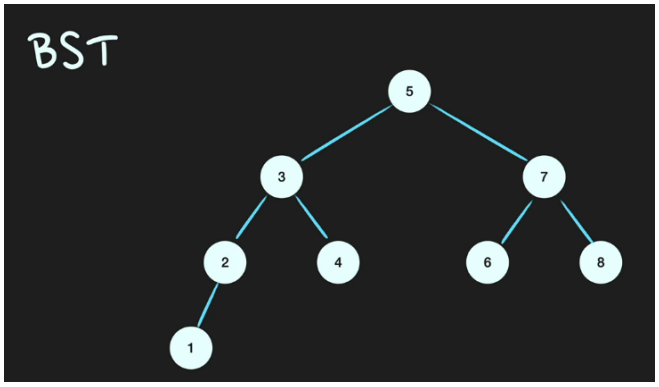
▶ 이진탐색트리(Binary Search Tree; BST)는 모든 노드의 자녀 노드 갯수가 2 이하인 이진 트리로, 저장과 동시에 정렬을 하는 자료구조입니다. 어느 노드를 선택하든 해당 노드의 left subtree에는 그 노드의 값보다 작은 값들을 지닌 노드들로만 이루어져 있고, 노드의 right subtree에는 그 노드의 값보다 큰 값들을 지닌 노드들로만 이루어져 있는 이진 트리입니다. 검색과 저장, 삭제의 시간복잡도는 모두 O(logn)이고, worst case는 한쪽으로 치우친 tree가 됐을 때 O(n)입니다.

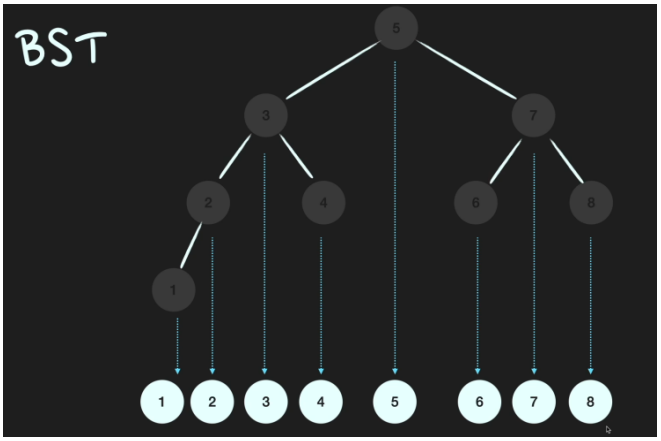
● 이진트리(Binary tree)는 어떤 자료구조인가?

▶ 모든 노드의 자녀 노드 갯수가 2 이하인 트리를 이진 트리라고 합니다.

● 이진탐색트리 BST(가 되기 위한) 조건

▶ 루트 노드(root node)의 값보다 작은 값은 left subtree에, 큰 값은 right subtree에 있어야 하고, 모든 subtree들도 그 조건을 만족해야 합니다(Recursive).





● 이진탐색트리 저장/구현

▶ BST는 저장과 동시에 정렬을 하는 자료구조입니다. 따라서, 새로운 데이터를 저장할 때 일정한 규칙에 따라 저장을 하게 됩니다. 정수 1부터 8까지가 정렬되어 있는 배열로 이진검색트리를 구현해 보겠습니다.
[1, 2, 3, 4, 5, 6, 7, 8] 먼저 중간 숫자인 5를 루트 노드로 적습니다. 5를 기준으로 더 작은 값들이 있는 왼쪽을 left subtree로, 더 큰 값들이 있는 오른쪽을 right subtree로 만들겠습니다. 먼저 왼쪽에서 중간 숫자인 3를 먼저 왼쪽 subtree의 루트 노드로 둡니다. 중간 숫자 3의 양옆으론 2, 4가 있는데, 2는 3보다 작기 때문에 3의 왼쪽 아래에, 4는 3보다 크기 때문에 3의 오른쪽 아래에 둡니다. 그러면 1이 남는데, 1은 2의 옆에 있으면서 2보다 작기에 2의 왼쪽 아래에 둡니다. 다음으로는 오른쪽 subtree 차례입니다. 중간 숫자인 7을 오른쪽 subtree의 루트 노드로 둡니다. 7

의 양 옆엔 6과 8이 있는데, 7보다 작은 6은 7의 왼쪽 아래에, 7보다 큰 8은 7의 오든쪽 아래에 둡니다. 이렇게 하면 저장과 동시에 정렬이 된 이진탐색트리가 완성이 됩니다.

● 이진탐색트리 검색(Search)과 삽입

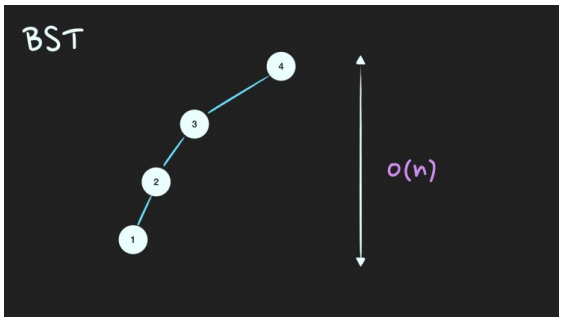
▶ 먼저 검색을 할 값을 루트 노드의 값과 비교하여 작으면 left subtree, 크면 right subtree로 갑니다. 그런 다음 다시 해당 subtree의 루트 노드와 비교하여 작으면 left subtree, 크면 right subtree로 갑니다. 이 과정을 검색할 값과 동일한 노드를 찾을 때까지 계속합니다. tree의 높이는 logn이므로, 시간복잡도는 최대 O(logn)입니다. 데이터의 삽입도 동일한 과정을 거쳐 자신의 자리를 찾아갑니다.

● 이진탐색트리 삭제

▶ 삭제할 노드가 하나의 자식 노드(서브 트리)를 갖는 경우, 자식 노드를 부모 노드와 연결하되, 삭제하려는 노드가 오든쪽이었다면 오든쪽에 연결하고 왼쪽이었다면 왼쪽에 연결합니다. 만약 삭제할 노드가 두 개의 자식 노드(서브트리)를 갖는 경우라면, 삭제하고자 하는 노드의 좌측 서브 트리의 가장 큰 값과 우측 서브 트리의 가장 작은 값을 찾은 다음에, 둘 중 하나의 값과 대체하면 됩니다 (https://1d1cblog.tistory.com/309).

● BST의 worst case 시간복잡도는 O(n)이다. 어떤 경우에 worst case가 발생하나?

▶ 균형이 많이 깨져서 한 쪽으로 치우친 BST의 경우에 worst case가 됩니다. 이렇게 되면 Linked list와 다를 게 없어집니다. 이때 tree의 높이는 O(n)이 되고, 따라서, 탐색시에 O(logn)이 아니라 O(n)의 시간복잡도를 가지게 됩니다.



● worst case 해결방법은 무엇인가

▶ 자가 균형 이진 탐색 트리(Self-Balancing BST)는 알고리즘으로 이진 트리의 균형이 잘 맞도록 유지하여 높이를 가능한 낮게 유지합니다. 대표적으로 AVL트리와 Red-black tree가 있습니다. 예를 들어 JAVA에서는 hashmap을 구현을 할 때, seperate chaning으로써 Linked list와 Red-black tree를 병행하여 저장합니다.
→ seperate chaining은 해시 충돌 (hash collision — 서로 다든 값을 해싱한 결과가 같을 수 있음)에 대한 회피 방법 중의 하나로 해시 충돌 시 충돌된 키-값 쌍의 데이터가 8개 미만이면 링크드 리스트로 연결하고, 8개 이상이면 레드 블랙 트리로 연결하여 해결합니다(6개가 되면 다시 링크드 리스트로 변환). 링크드 리스트는 충돌하는 데이터가 많아질수록 탐색 시간이 O(n)으로 늘어나기 때문에, 효율적이게 탐색하기 위해 레드 블랙 트리로(O(logn)) 변환하는 것입니다.

→ AVL트리: 발명자의 이름인 Adelson-Velsky and Landis에서 따온 이름—은 자가 균형 이진 탐색 트리입니다. 두 자식 서브트리의 높이는 항상 최대 1만큼 차이난다. 만약 어떤 시점에서 높이 차이가 1보다 커지면 회전(rotatio)을 통해 스스로 균형을 잡아 높이 차이를 줄입니다. AVL 트리는 높이를 logN으로 유지하기 때문에 검색, 삽입, 삭제는 모두 평균과 최악의 경우 O(log n)의 시간복잡도가 걸립니다. 삽입 삭제 시 불균형 상태(Balnace Factor이 -1 ,0, 1이 아닌 경우) 가 되면 AVL트리는 불균형 노드를 기준으로 서브트리의 위치를 변경하는 rotation 작업을 수행하여 트리의 균형을 맞추게 됩니다.
→ Red-black tree: 자가 균형 이진 탐색 트리(self-balancing binary search tree)로서, 대표적으로는 연관 배열(key 하나와 value 하나가 연관되어 있는 자료 구조) 구현하는 데 쓰이는 자료구조입니다. 레드-블랙 트리는 각각의 노드가 레드 나 블랙 인 색상 속성을 가지고 있는 이진 탐색 트리입니다. 이진 탐색 트리가 가지고 있는 일반적인 조건에 다음과 같은 추가적인 조건을 만족해야 합니다. 노드는 레드 혹은 블랙 중의 하나이고, 루트 노드는 블랙이고, 모든 리프 노드들은 블랙이고, 레드 노드의 자식노드 양쪽은 언제나 모두 블랙이어야 합니다. 즉, 레드 노드는 연달아 나타날 수 없으며, 블랙 노드만이 레드 노드의 부모 노드가 될 수 있습니다. 또한 어떤 노드로부터 시작되어 그에 속한 하위 리프 노드에 도달하는 모든 경로에는 리프 노드를 제외하면 모두 같은 개수의 블랙 노드가 있습니다. 위 조건

들을 만족하게 되면, 레드-블랙 트리는 가장 중요한 특성을 나타내게 됩니다. 루트 노드부터 가장 먼 잎노드 경로까지의 거리가, 가장 가까운 잎노드 경로까지의 거리의 두 배 보다 항상 작습니다. 다시 말해서 레드-블랙 트리는 개략적(roughly)으로 균형이 잡혀 있습니다(balanced). 따라서, 삽입, 삭제, 검색시 최악의 경우(worst-case)에서의 시간복잡도가 트리의 높이(또는 깊이)에 따라 결정되기 때문에 보통의 이진 탐색 트리에 비해 효율적이라고 할 수 있습니다. 왜 이런 특성을 가지는지 설명하기 위해서는, 네 번째 속성에 따라서, 어떤 경로에도 레드 노드가 연이어 나타날 수 없다는 것만 알고 있어도 충분합니다. 최단 경로는 모두 블랙 노드로만 구성되어 있다고 했을 때, 최장 경로는 블랙 노드와 레드 노드가 번갈아 나오는 것이 될 것입니다. 다섯 번째 속성에 따라서, 모든 경로에서 블랙 노드의 수가 같다고 했기 때문에 존재하는 모든 경로에 대해 최장 경로의 거리는, 최단 경로의 거리의 두 배 이상이 될 수 없습니다.

Resources: inflearn 개발남 노씨, 엔지니어대한민국, 1d1cblog