Data Structure Assignment 3

201911013 곽현우

A7-1 (2 points)

Give an example input list that requires merge-sort and heap-sort to take O(nlogn) time to sort, but insertion-sort runs in O(n) time. What if you reverse this list?

Answer)

우선 Merge sort 와 heap sort, insertion sort 가 무엇인지 알아보고 조건을 만족하는 적절한 input list 를 찾아보자.

1) merge sort 란 주어진 Input List 를 절반으로 재귀적으로 나눈 후 merge 과정에서 sort 해주는 것을 말한다. 이때 merge-sort 과정을 Binary tree 로 나타내었을 때 한 층에서 merge 하는데 걸리는 시간은 O(N)이다. 이때 Binary Tree 의 높이가 logN 이기 때문에 전체적인 Merge-sort 과정은 O(NlogN)이 걸린다.

2) heap-sort 란 Heap 을 기반으로 한 정렬방식을 말한다. 이때 Heap 에서 add 와 remove_min 이 O(logN) 만큼 시간이 걸리기 때문에 Heap sort 는 거기에 원소의 개수 N 을 곱해준 O(NlogN)만큼의 시간이 걸린다.

3) Insertion – sort 란 Input list 의 원소중 가장 큰 원소들을 탐색하여 Priority Queue 에 차례대로 삽입하고 난 후 remove_min operation 을 통해 원래 Input list 에 삽입하는 분류방법을 말한다. 따라서 P.Q 에 삽입할 때 Input list 에서 가장 큰 Item 을 탐색하는 과정이 O(N)의 시간이 걸린다. 또한 P.Q 에서 원소의 개수만큼 remove_min 을 수행하기 때문에 O(N)의 시간이 걸린다. 즉 최종적인 Insertion-sort 를 하는데 걸리는 시간은 $O(N^2)$이다.

이때 Heap-sort 와 merge – sort 는 어떤 list 간에 O(NlogN)이 걸린다, 그러나 Insertion sort 의 경우 이미 input list 가 정렬되어 있는 list 라면 모든 원소들을 탐색할 필요가 없으므로 O(1)이 되어 전체적인 Insertion sort 는 O(N)이다.

만약 이러한 List 가 reverse 된다면 Insertion sort 의 경우 일반적인 Input List 와 마찬가지로 Input list 에 대해 탐색을 해야하므로 $O(N^2)$의 시간이 걸릴 것이고 Heap sort, Merge sort 는 동일하게 O(NlogN)이 될 것이다.

A7-2 (3 points)

What is the best algorithm for sorting these items?

- General comparable objects
- Long-character strings
- 32-bit intergers
- double-precision floating-point numbers (64-bit, check here: https://en.wikipedia.org/wiki/Double-precision_floating-point_format)
- bytes

Select three items (at your favor) from the list, provide sorting algorithms for each, and justify your answer.

Answer)
1) General Comparable Objects
일반적으로 비교가능한 객체들의 경우 Quick Sort 가 Running time 이 O(NlogN)으로 가장 효율적인 분류방식이 될 것이다. 왜냐하면 다른 Merge sort 나 Heap sort 도 O(NlogN)의 running time 을 가지지만 Qiuck Sort 는 Pivot 을 하나씩 설정한다는 점에서 단계를 거칠수록 크기가 줄어든다. 따라서 Quick Sort 가 가장 효율적이다.

2) Long – character strings
긴 문자열은 기본적으로 한 문자가 가질 수 있는 문자의 범위가 매우 넓다. 알파벳의 경우 총 26 가지의 경우가 있으며 한글의 경우 자음과 모음의 조합에 따라 영어보다 더 다양한 경우를 갖는다. 따라서 긴 문자열들을 효율적으로 분류하기 위해서는 사전식 정렬, 즉 Lexicographin sort 중 Radix sort 를 하면 된다. 이때 Radix sort 의 실행시간은 O(d(N+n))(*N: 표현할 수 있는 문자의 개수, n: 비교하려는 대상의 개수, d: 문자열의 길이)즉 선형적이므로 가장 효율적인 sort 이다.

3) 32 – Bits Integers
32 – Bits Integers 의 경우 32 자리에 각각 0 또는 1 이 위치한 정수들을 말한다. 이러한 자료를 효율적으로 비교하려면 2)의 경우와 비슷하게 각각의 자리에 대하여 사전식 정렬, 즉 Radix sort 를 해야 한다. 이때 비교해야 할 정수들의 개수를 N 이라 하면 각 자릿수에 대해 정렬을 하는데 걸리는 시간이 O(N)이고 0 과 1 만 비교하면 되기 때문에 O(N + 1)만큼 시간이 걸린다. 32 자릿수를 모두 비교하려면 O(32(N+1)), 즉 O(N)걸린다.

A7-3 (3 points)

Bob has a set *A* of *n* nuts and a set *B* of *n* bolts, such that each nut in *A* has a unique matching bolt in *B*.

Unfortunately, the nuts in *A* all look the same, and the bolts in *B* all look the same as well. The only kind of a comparison that Bob can make is to take a nut-bolt pair (*a*, *b*), such that *a* is in *A* and *b* is in *B*, and test it to see if the threads of a are **larger, smaller, or a perfect match** with the threads of *b*.

Describe and analyze an efficient algorithm for Bob to match up all of his nuts and bolts.
Answer)

임의로 Nut set 에서 하나의 Nut(a 라 하자)를 선택한다. 이후 Bolt set 에서 하나씩 Bolt 를 꺼내어 a 와 맞는 지 비교한다. 이때 모든 Bolt 에 대해 a 보다 큰 bolt 의 그룹(L set 라 하자)과 작은 bolt 의 그룹(S set 라 하자)을 나눈다. 나누는 과정에서 맞는 bolt(b 라 하자)가 반드시 나온다.
이후 다른 Nut(a1 이라 하자)를 선택한다. 이때 a1 이 이전 a Nut 와 맞았던 b bolt 와 비교를 하여 b 보다 작으면 S set 에, 크면 L set 에 있는 bolt 와 비교를 한다.
a1 이 b bolt 보다 크다고 가정하면 L set(a1 이 b 보다 작으면 S set)의 모든 bolt 와 a1 을 다시 비교하여 Phase1 과 같이 L set 를 a1 보다 큰 bolt 그룹, a1 보다 작은 bolt 그룹으로 나눈다. 이러한 과정들을 다른 nut 에 대해서도 실시한다. 즉 정리를 해보면 다음과 같다.

- 임의의 Nut 를 선택하여 이전의 Nut 들과 맞았던 Bolt 들과 비교하여 비교할 Bolt 들이 있는 Bolt set 를 탐색한다.
- Bolt set 를 탐색 후 set 내에 있는 Bolt 와 임의로 선택한 Nut 를 비교하여 다시 Larger set 와 Smaller set 로 나눈다.
-위의 두가지 과정을 모든 Nut 들에 대해 반복한다.

이러한 알고리즘은 Quick Sort 가 수행되는 과정과 같다. 따라서 이 알고리즘에서 임의로 선택하는 Nut 들이 각각의 하위 list 들의 Pivot 이 되는 것이다. 이때 이 알고리즘의 과정을 Binary Tree 로 나타내었다고 생각하자. 이 Binary tree 의 높이를 i 라 할 때 좋은 Pivot 을 선택할 확률이 1/2 이므로 확률적으로 i/2 의 자손 노드들이 좋은 pivot 을 선택할 것이다. 따라서 각각의 과정들이 수행될 때 그 List 들의 기대되는 최대 크기는 $3/4^{i/2}$N 이다. 이때 맨 아래에 있는 list 의 크기($3/4^{i/2}$N)가 1 이 되어야 하므로 i 는 $2\log_{4/3} N$이다. 즉 높이는 O(logN)임을 알 수 있다. 이때 같은 높이에 대하여 그 높이에 해당하는 모든 원소들을 탐색해야 하므로 O(N)의 시간이 걸림을 알 수 있다, 따라서 전체적으로 기대되는 이 알고리즘의 수행시간은 O(NlogN)이다.

A8 - Search Trees (12 + 3 points)

In following questions, when removing a node *p* with two children in a BST, choose the largest predecessor (= before(*p*)) to replace with.

A8-1 (1 point)

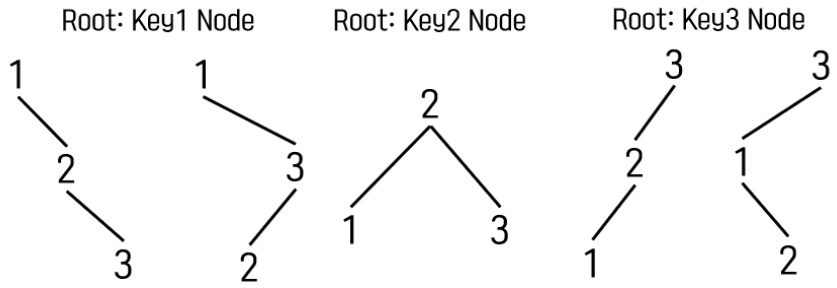
How many different binary search trees can store the keys {1, 2, 3}?
Answer)

Key : 1 인 노드를 root 로 가지는 BST: 2 개

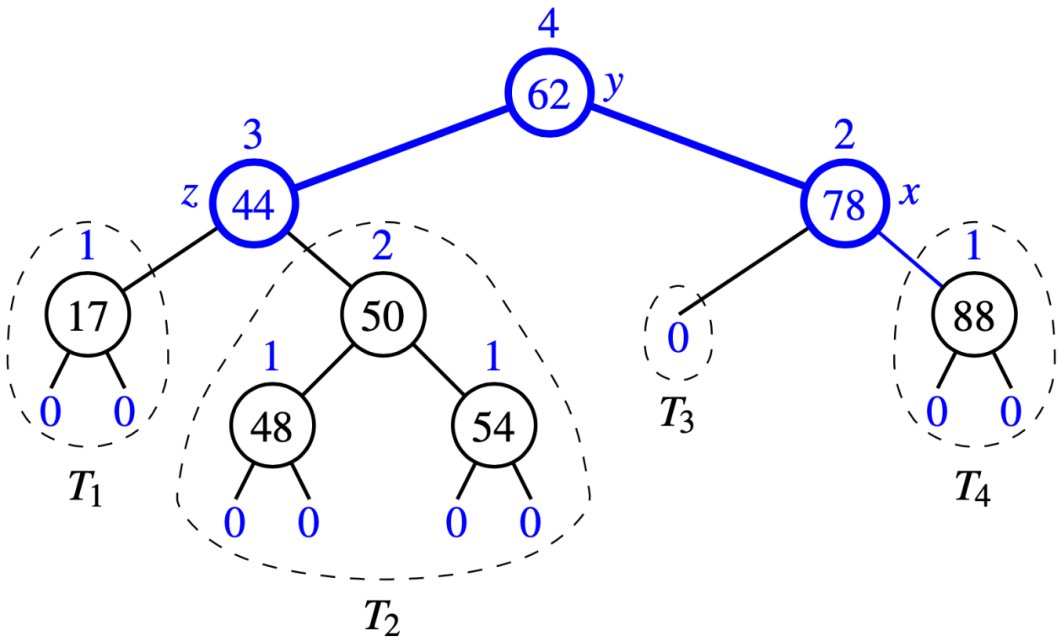
Key : 2 인 노드를 root 로 가지는 BST: 1 개

Key : 3 인 노드를 root 로 가지는 BST: 2 개

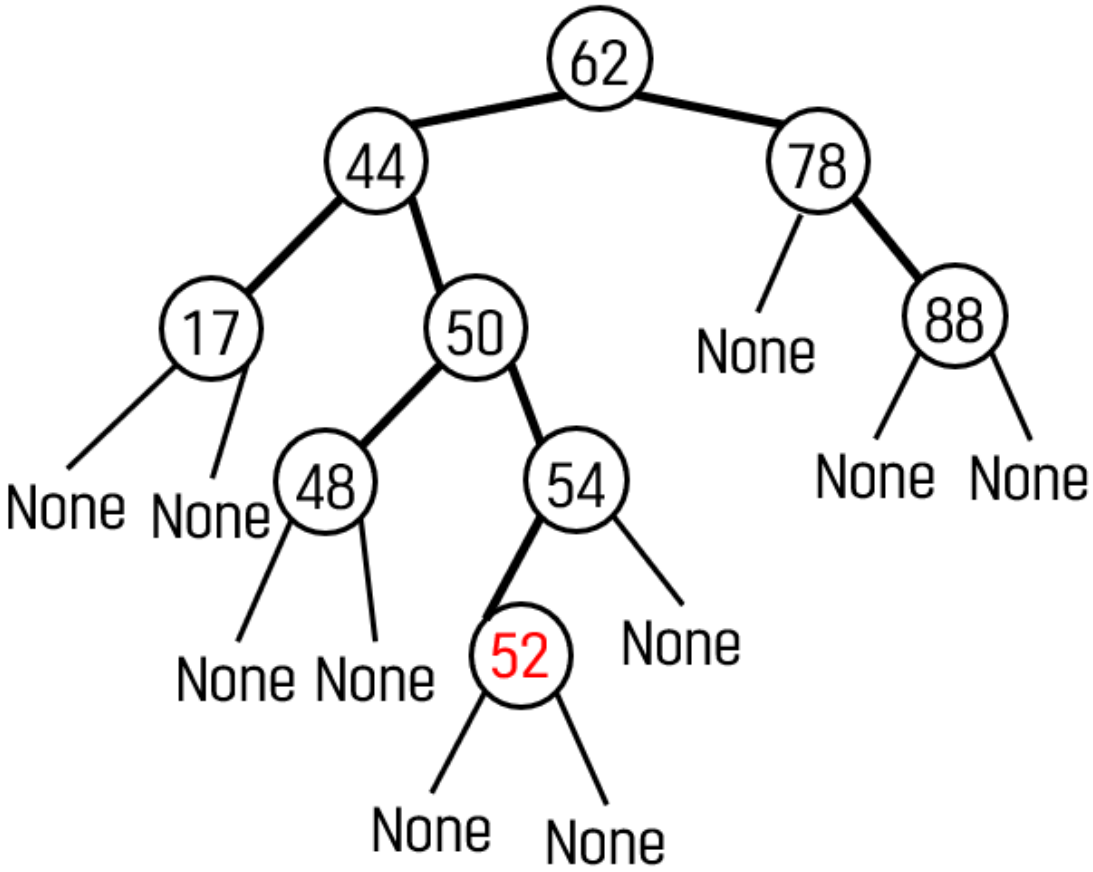
총 5 개이다.



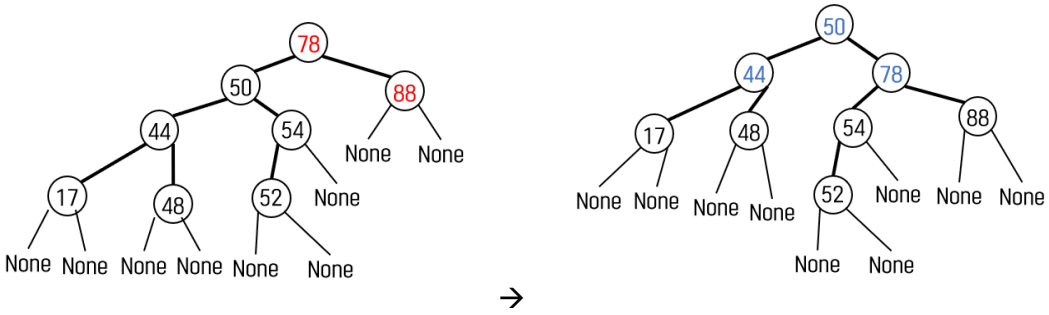
A8-2 (2 points)



1) Draw the AVL tree resulting from the insertion of an entry with key 52 into the given figure.



2) After 1, draw the AVL tree resulting from the deletion of an entry with key 62.



→

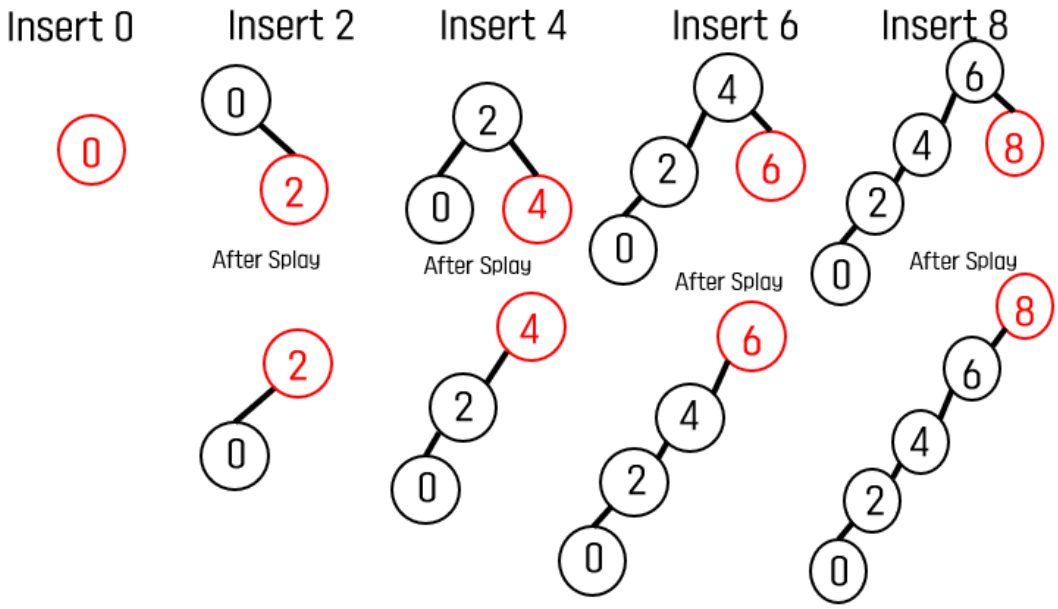
rebalancing

A8-3 (3 points)

Perform the followng sequence of operations in an initially empty splay tree and draw the trees after each set of operations.

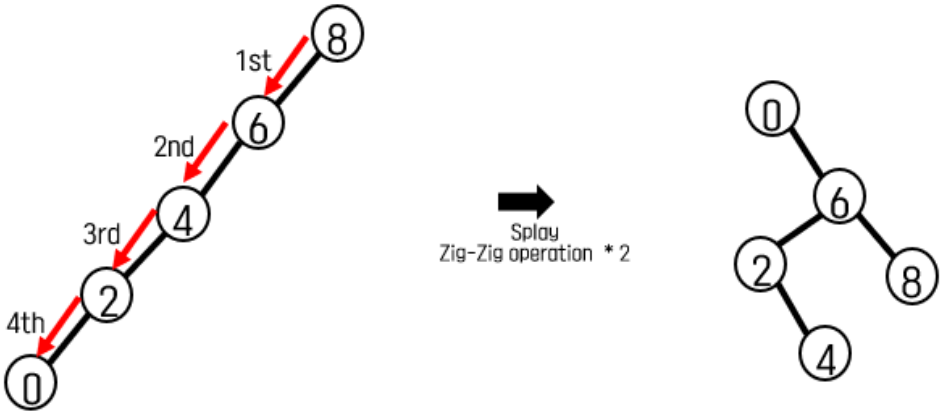
1) Insert keys 0, 2, 4, 6, 8, in this order.

이해하기 쉽도록 External node 들은 생략하여 Splay Tree 를 나타냈다.

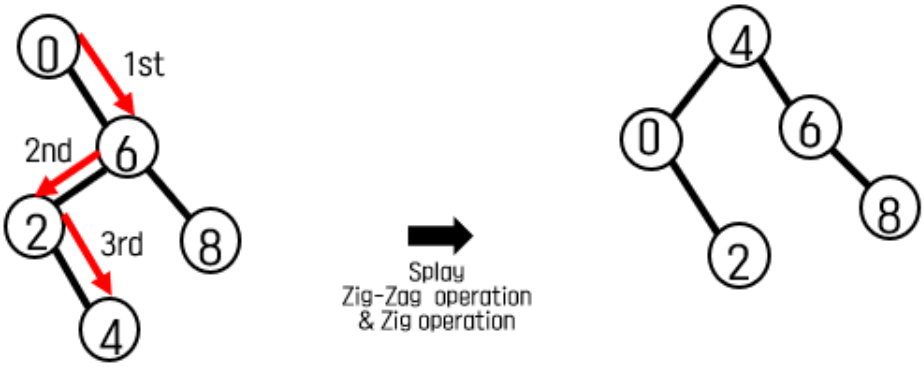


2) Search for keys 1, 3, 5, 7, in this order. .(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)

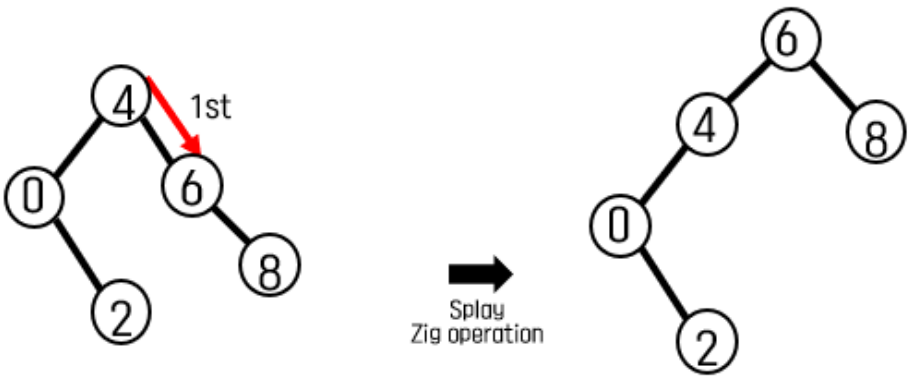
Search 1

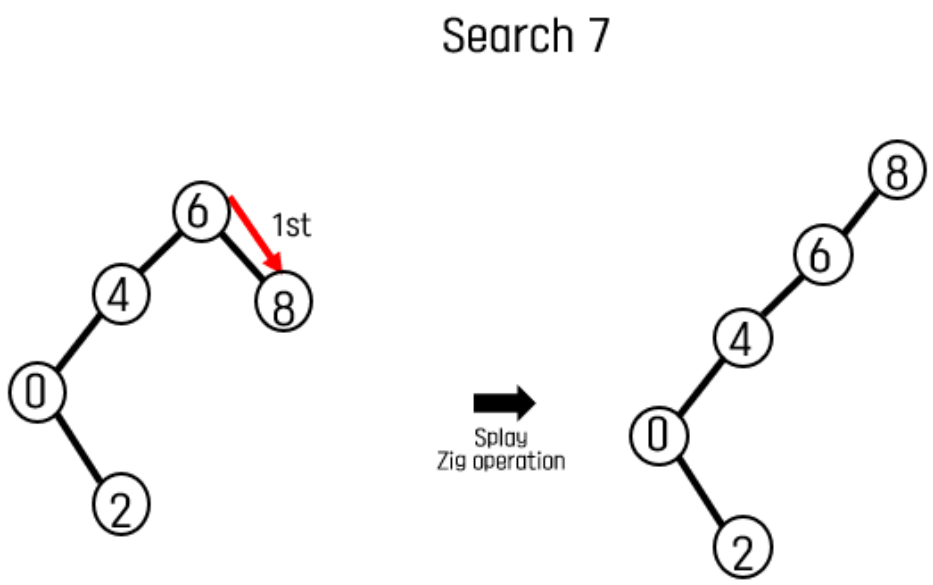


Search 3



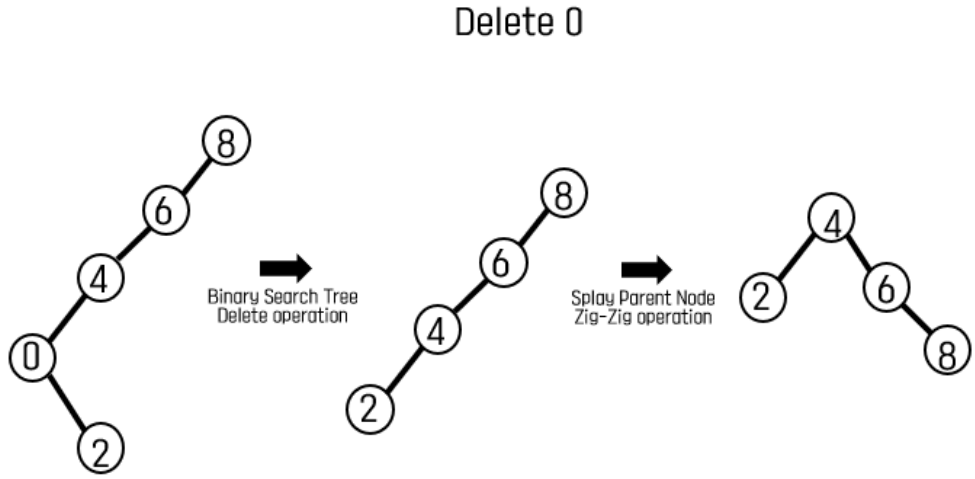
Search 5

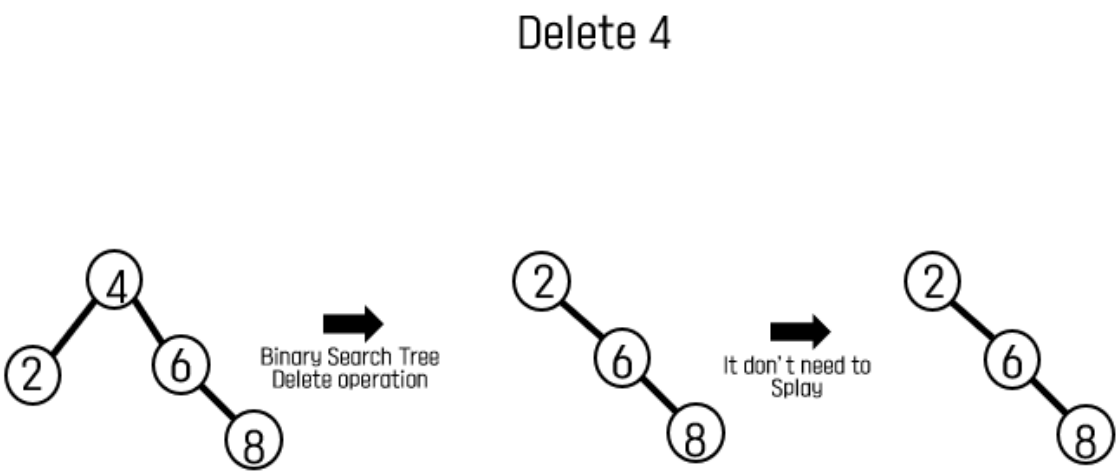


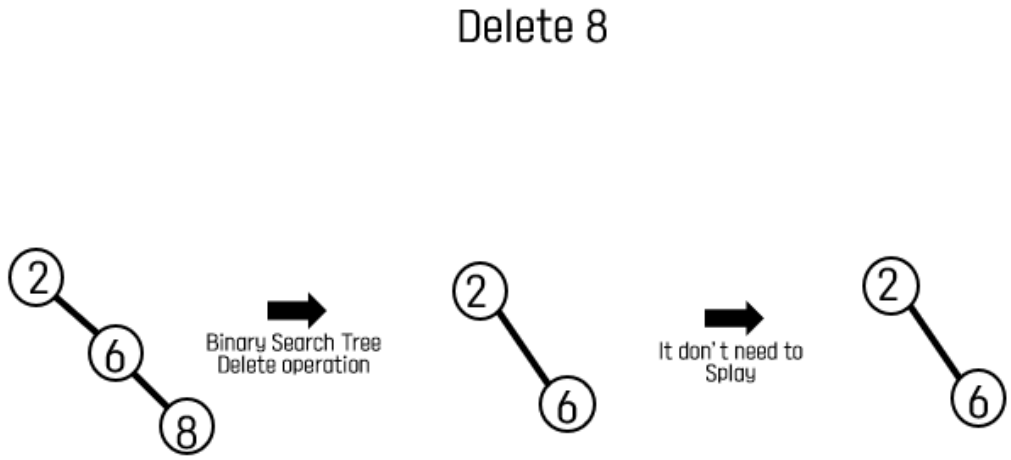


3) Delete keys 0, 4, 8, in this order. .(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)

-Binary Search Tree 의 delete operation 을 통해 노드를 지운 다음 지운 노드의 parent 노드를 splay 해준다.

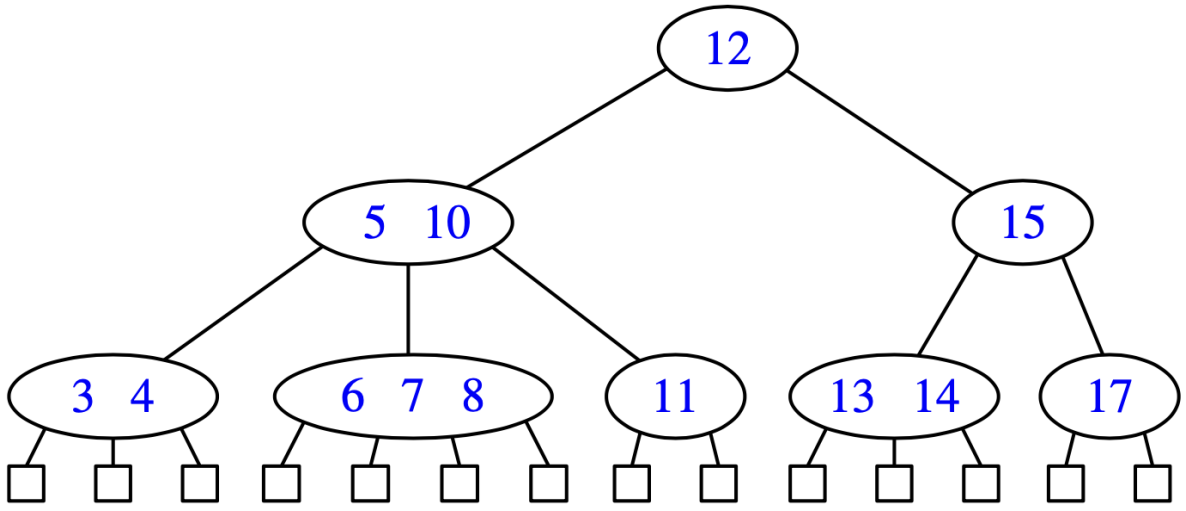






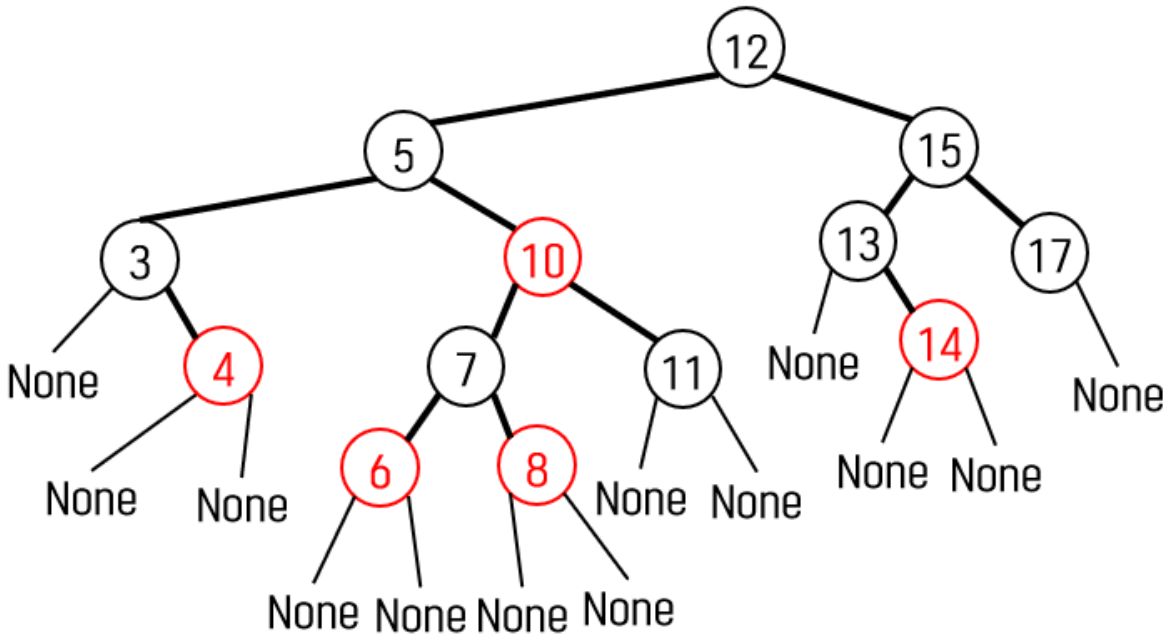
A8-4 (1 point)

Draw the red-black tree that correspond to a following (2,4) tree.



Answer)

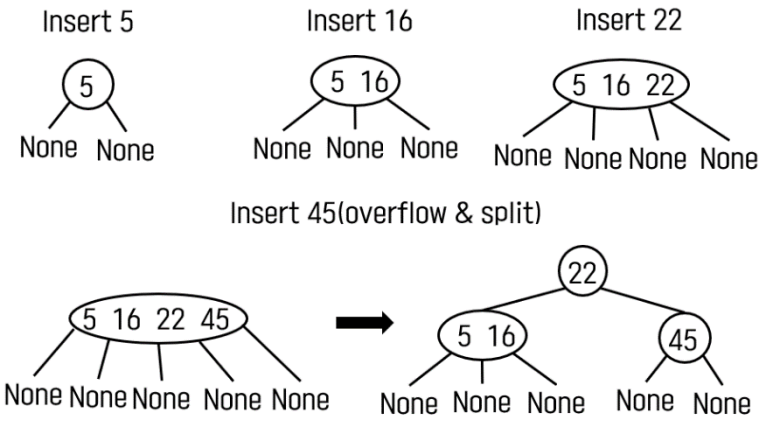
Red-Black Tree(Key 값이 2 개 있을 때 가장 작은 값을 Black Node 로 설정하였고 3 개 일때는 중앙값을 Black 으로 설정하였습니다.)

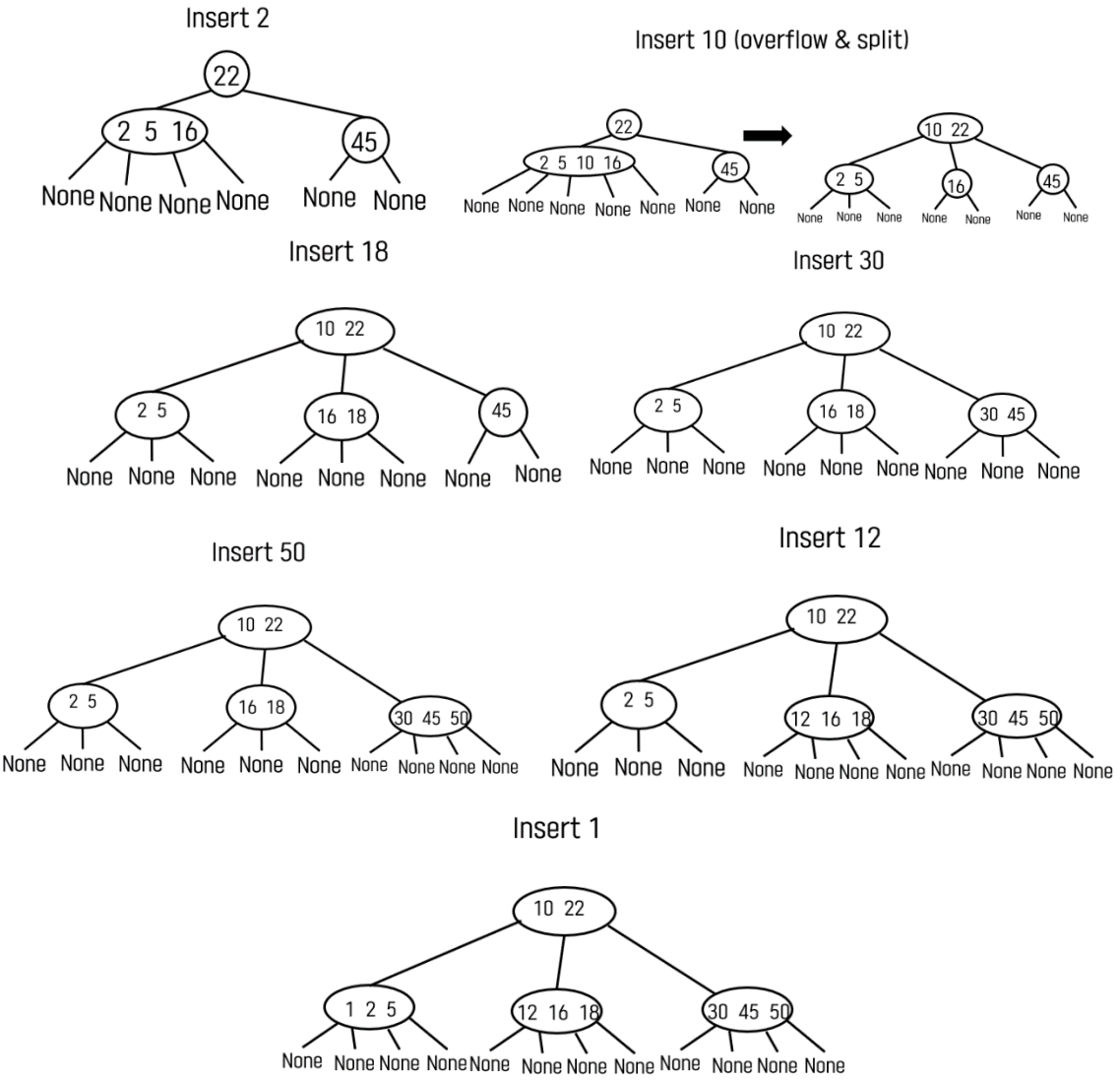


A8-5 (5 points)

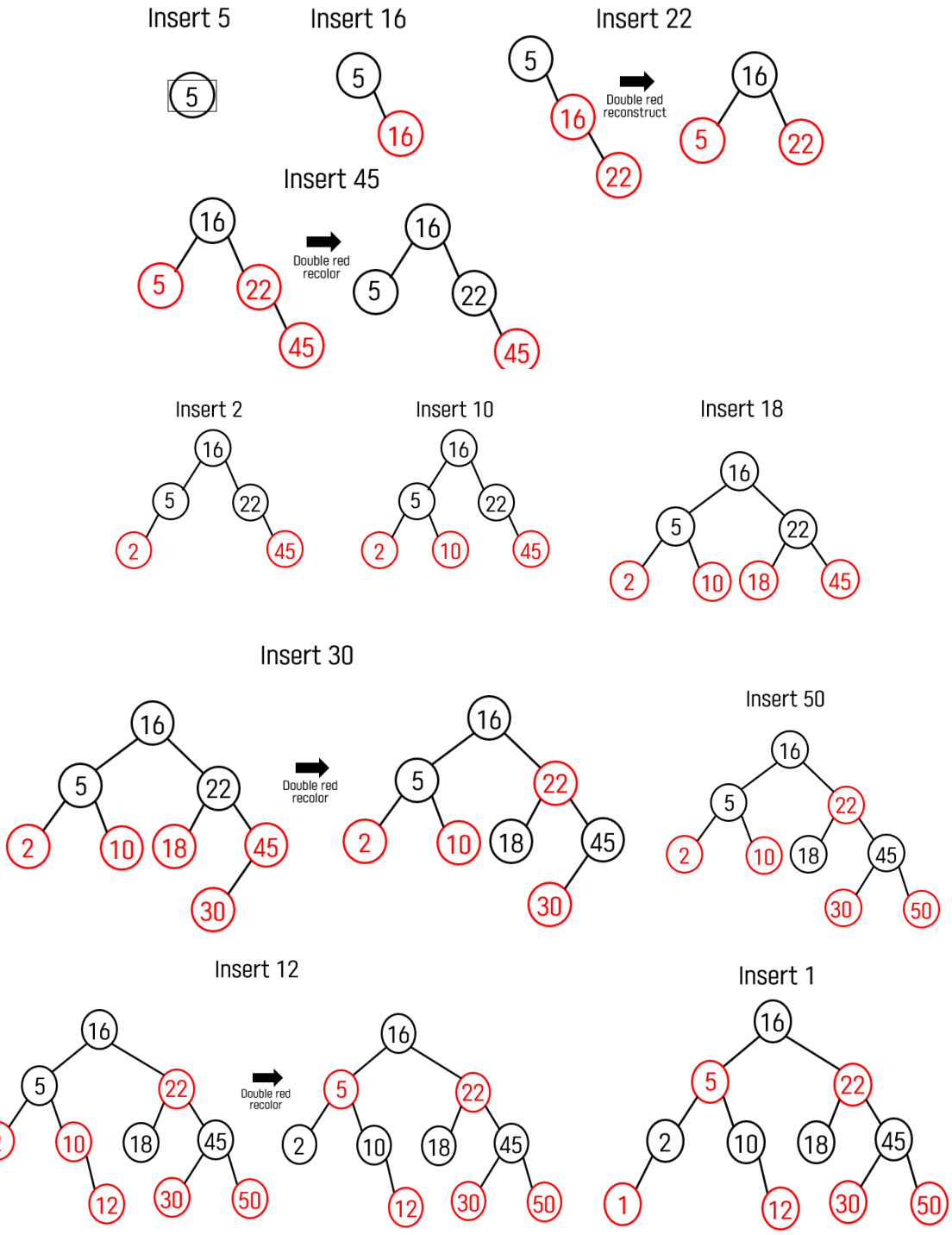
Consider the sequence of keys (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1). Draw the result of inserting entries with these keys (in the given order) into

1) An initially empty (2,4) tree.





2) An initially empty red-black tree.(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)



From the result of 2), draw the red-black trees after following deletions.

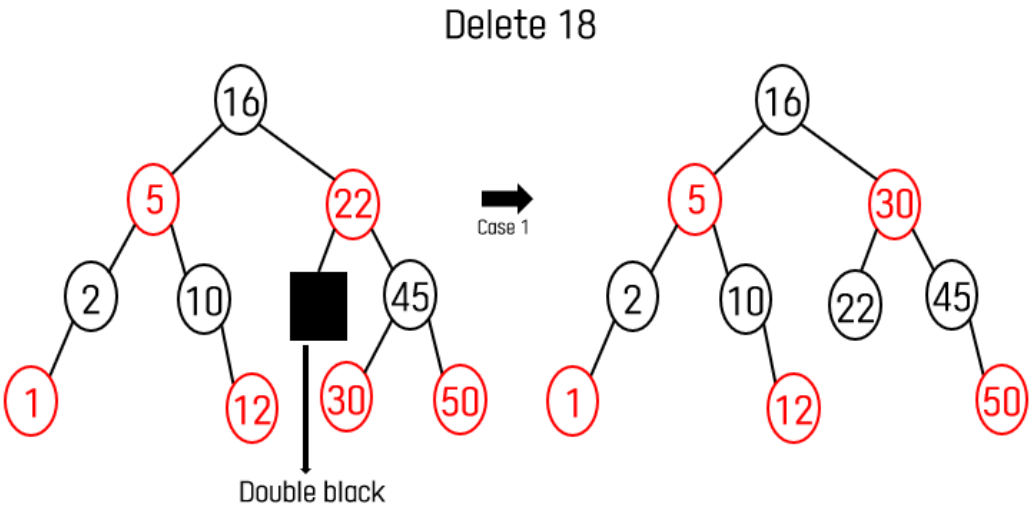
우선 지우려는 노드를 x 라 하고 x 의 sibiling node 를 y, x 와 y 의 parent node 를 z 라 하자. 만약 x 가 red 이면 red-black tree 의 4 가지 특징에 위반되지 않는다. 그러나 x 가 black 이라면 black depth 가 모든 노드에 대하여 동일하지 않기 때문에 double black 의 상태가 나타난다. 이때 y 의 조건에 따라 지우고 난 후 tree 를 수정해줄 수 있는 경우가 3 가지 있다.

첫번째, y 가 black 이고 red child 를 가질 경우, x, y, z 의 세 노드를 reconstruct 하고 원래 z 의 색깔에 따라 reconstruct 후 전의 z 위치에 위치한 노드의 색깔을 맞춰준다.

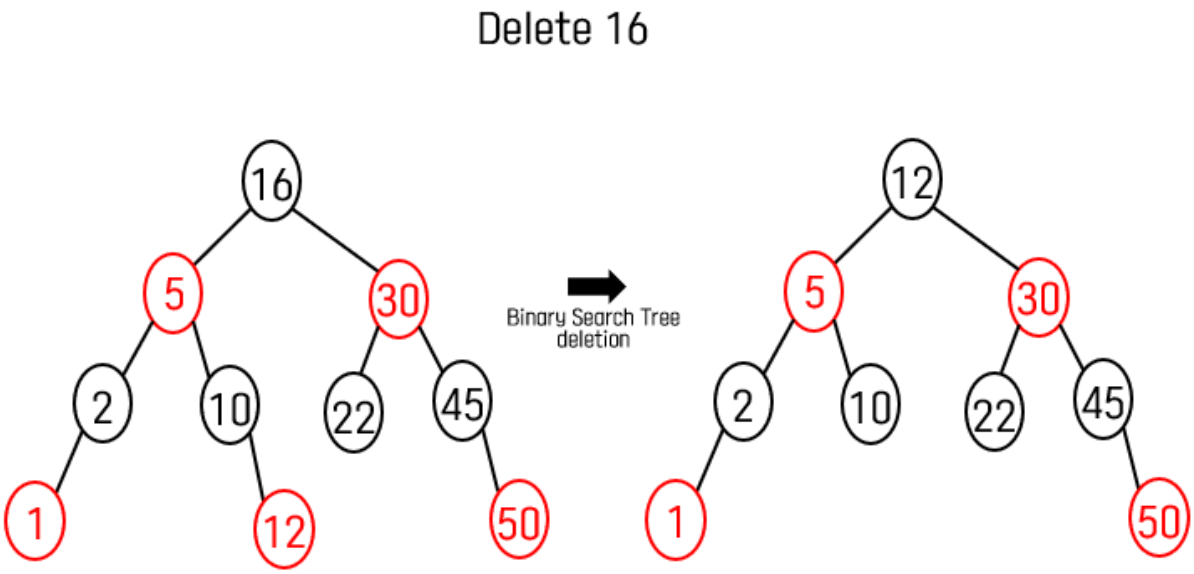
두번째, y 가 black 이고 그것의 children 들이 모두 black 인 경우,y 와 z 를 recoloring 을 한다.

세번째, y 가 red 인 경우, y 와 z 를 reconstruct 하면 case 1 또는 case 2 의 상황이 다시 나타나는데 case 에 따라 다시 수정한다. 따라서 이 세가지 경우에 따라 수정하는 방법이 달라진다.

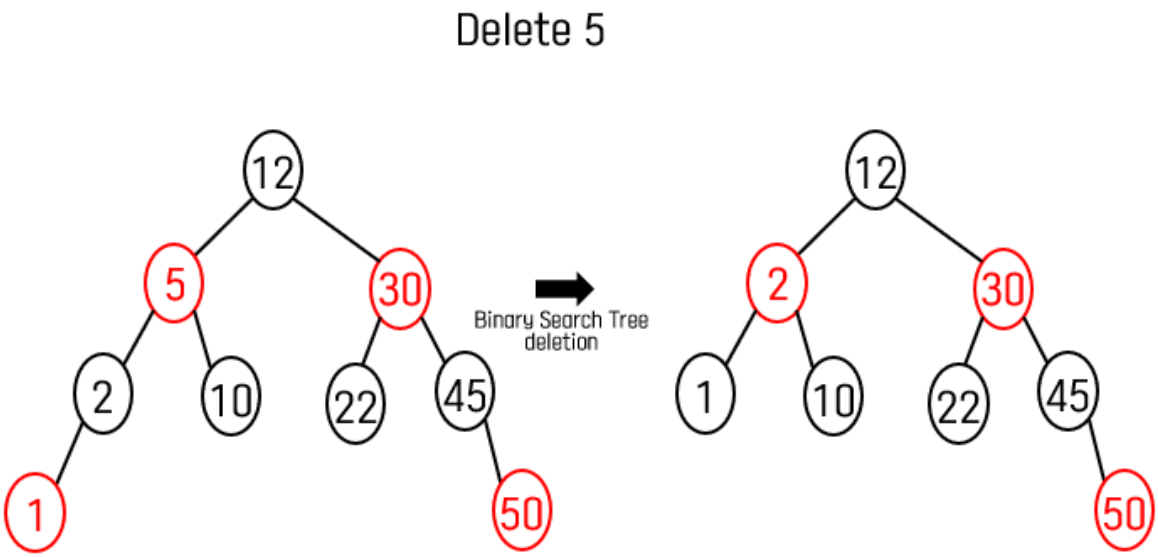
3) Delete 18(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)



4) After 3), Delete 16(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)



5) After 4), Delete 5(이해를 돕기 위해 External node 들은 그림에서 생략하였습니다.)



A8-Bonus (3 points)

Let *T* be a red-black tree and let *p* be the position of the parent of the original node that is deleted by the standard search tree deletion algorithm.(이 풀이에서 Leaf 는 Null 인 External Nodes 를 고려하지 않고 생각한 것이다.)

1) Prove that if *p* has zero children, the removed node was a red leaf.

P 가 zero child 가 되기 위해서는 delete operation 전 P 의 child 가 removed node 하나만 있어야하고 이때 removed node 는 leaf 여야 한다. 이때 지워진 Child 가 Black 일 경우 모든 Node 의 Black Depth 가 일치하지 않는다. 따라서 반드시 red 여야 한다.

2) Prove that if *p* has one child, the deletion has caused a black deficit at p, except for the case when the one remaining child is a red leaf.

P 의 Child 가 하나만 있을 경우, delete operation 이전에 P 의 child 가 2 개 있어야 하며 removed node 가 leaf 여야한다. 이때 남은 child 가 red leaf 인 경우는 제외하므로 남은 child 는 black 이 될 것이고 지워진 node 는 반드시 black 이 된다. 따라서 delete operation 수행이후에 p 에 black deficit 가 된다.

3) Prove that if *p* has two children, the removed node was black and had one red child.

P 가 2 개의 Child 를 가지기 위해서는 지우려는 Node(R 이라 하자)가 Child 를 가지고 있어야 하며 이때 R 이 P 의 Left child 면 R 은 Leaf 인 right Node 를 가져야 하고 R 이 P 의 Right child 면 R 은 Leaf 인 left child 를 가져야 한다. 이때 R 이 red 인 경우 R 의 child 는 Internal Property 에 의해 반드시 Black 이 되어야 하는데 이러면 R 의 Null Node 의 Black Depth 와 R 의 Child 의 Null Node 의 Black Depth 값이 같아지지 않는다. 따라서 R 은 반드시 Black 이 되어야 한다.

이 상태에서 R 의 Child 가 Black 인 경우 마찬가지로 R 의 Null Node 의 Black Depth 보다 R 의 Child 의 Null Node 의 Black Depth 값이 1 만큼 더 커서 같아지지 않는다. 따라서 반드시 R 의 child 는 Red 가 되어야 한다.

즉 P 의 Child 가 2 개일 때 지워진 노드는 하나의 Red Child 를 가진 Black node 이다.