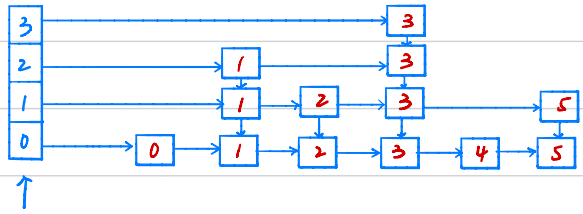
Skip List and Binary Search Tree.

①

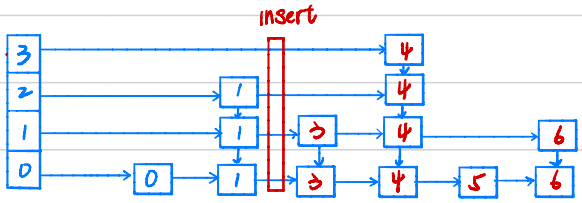
```
class Node{
    int key;
    int index;
    Node* down;
    Node* next;
}

class SkipList{
    int total_node;
    vector <Node*> head;
}
```

→ The index refers to that in bottom list!!

→ total number of node in bottom list.



* remember to update the index when insert/remove by sequential traverse.




```
Node* median(){
    int target = SkipList.total_index / 2;
    Node* current = SkipList.head.end()

    while(current->index != target){
        if(current->next != NULL && current->next->index <= target)
            current = current->next
        else
            current = current->down
    }

    while(current->index == target && current->down != NULL)
        current = current->down

    return current
}
```

← the head of most skipped list

← get to the rightmost

↑ get the current's base list position.

First add another member variable in node which records the index refering to the base list (S_0). Then use it as an identifier to find the $L\frac{n}{1}J$ node in base list. So check the top list first and if top list's index is largest then $L\frac{n}{2}J$, try the next layer. And if a

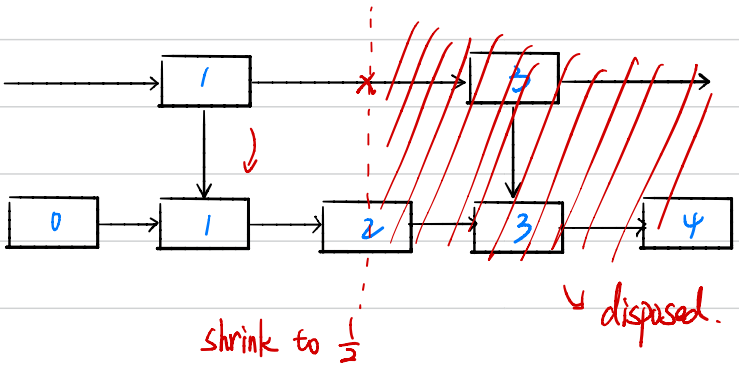
node's index equal $\lfloor \frac{n}{2} \rfloor$, find it in the base list and return. If its preceding (next) node is smaller or equal than $\lfloor \frac{n}{2} \rfloor$, move on to it ; otherwise, move to the next level (down).

And since each node is insert to the upper list with the probability of $1/2$ so from the point of normal distribution, the upper layer will have about $1/2$ numbers of node in the lower link. So, in average, skip lists will be like a binary tree with upper layer has nodes about $1/2$ of that of lower layer. So when we go through each node, we will split the lists to about $1/2$ of the remaining. (cut into $> \lfloor \frac{n}{2} \rfloor$ and $< \lfloor \frac{n}{2} \rfloor$)

Thus, for the k-th iteration, the lists is about $(\frac{1}{2})^k$ of the origin list.

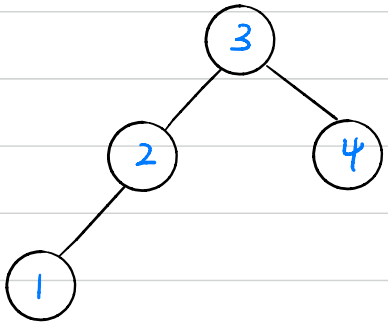
So at last, the lists remain 1 element $= n \cdot (\frac{1}{2})^k \Rightarrow k = \log_2 n = O(\log n)$

$\Rightarrow$ The complexity is $O(\log n)$. QED.

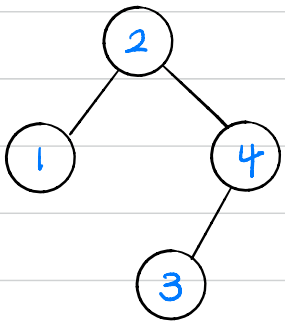


② If the root is different, the tree will definitely be different.

Consider the number set : {1, 2, 3, 4}



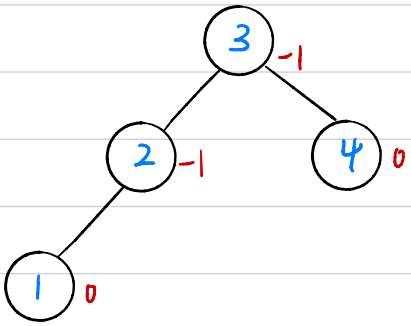
insert : 3, 4, 2, 1



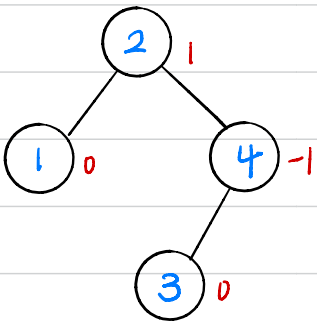
insert : 2, 1, 4, 3.

⇒ left ≤ root ≤ right

③ Same as the example above, and as long as △height ≤ 1, it is a legal AVL Tree.



insert : 3, 4, 2, 1



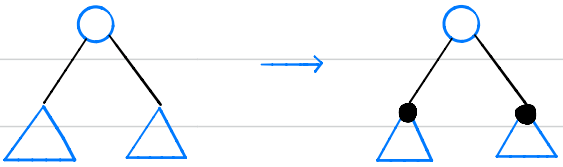
insert : 2, 1, 4, 3.

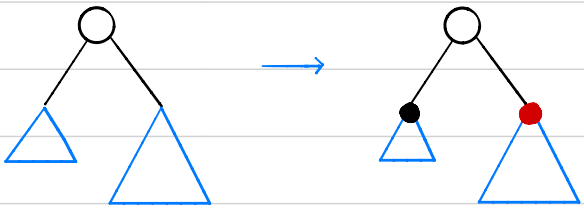
⇒ $|\triangle h| = |h(l) - h(r)| \leq 1$

⇒ left ≤ root ≤ right

④ Red - Black tree : ① root is black ② all node is red, black ③ red parent cannot have red children ④ number of black nodes counted from leaf → root are the same.

```
Function coloring(root)
    if root->left is NULL and root->right is NULL
        return
    if root->left is NULL
        if root.color == red
            root.color <- black
            root->right.color <- red
        else
            root->right .color <- red
        end if

    else if root->right is NULL
        if root.color == red
            root.color <- black
            root->left.color <- red
        else
            root->left .color <- red
        end if

    else
        if root->left.height == root->right.height
            root->left.color <- black
            root->right.color <- black
            coloring(root->left)
            coloring(root->right)

        else if root->left.height > root->right.height
            if root.color == black
                root->left.color <- red
                root->right.color <- black
            else
                root.color <- black
                root->left.color <- red
                root->right.color <- black
            end if
            coloring(root->left)
            coloring(root->right)

        else if root->left.height < root->right.height
            if root.color == black
                root->left.color <- black
                root->right.color <- red
            else
                root.color <- black
                root->left.color <- black
                root->right.color <- red
            end if
            coloring(root->left)
            coloring(root->right)

        end if
    end if
End function
```





<pf>

① color root to black at first (root.color ← black)

② all nodes are colored red, black.

③ if root is red and height of left subtree and right subtree are the same, both subtree root become black.; or we change root to black and change the root of higher subtree to red. So no two red nodes appear consecutively.

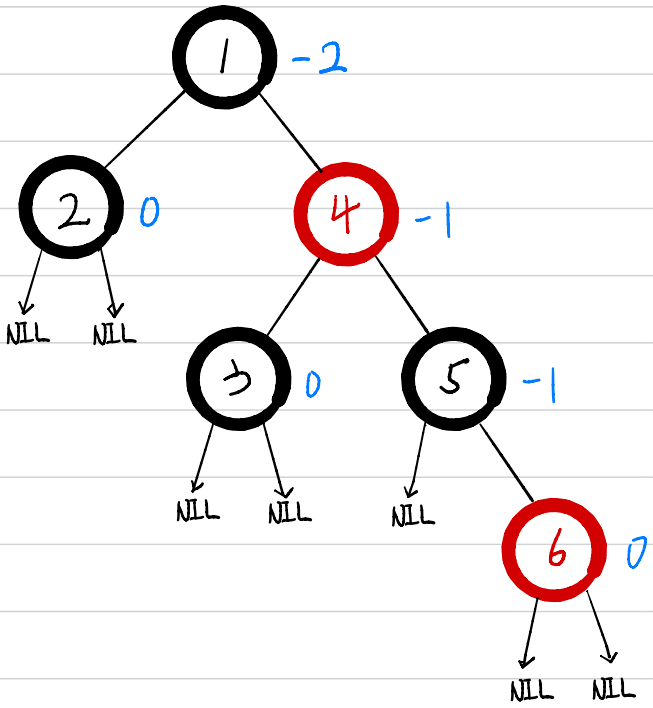
④ Consider a node in the tree with left and right subtree

1. if two subtrees has the same height = h, we make the root of two subtree into black and reolor the following to the same "black node height"
2. if two subtrees are different with left = right + 1, we turn the root of right subtree to red and rearrange the sequence of color. We can still make the "black node height" the same for left and right subtree.

So we can apply 1°, 2° to every node in AVL tree to turn it into RB tree.

Thus, AVL Tree can be colored into Red-Black tree.

⑤ Suppose the following red black tree, its root node has subtree height difference of 2. So it is a red black tree which is not an AVL Tree.



$\Delta h = h(\text{left}) - h(\text{right})$

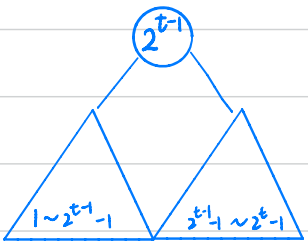
$\Rightarrow$ for AVL Tree, $|\Delta h| \le 1$.

⑥

Want to show: sequential insertion $1 \sim 2^k - 1$ makes AVL complete.

1° when $k = 1$, the AVL Tree contains one node. $\Rightarrow$ complete binary tree.

2° Suppose when $k = t$, the tree is a complete binary tree. So it will have the following structure.



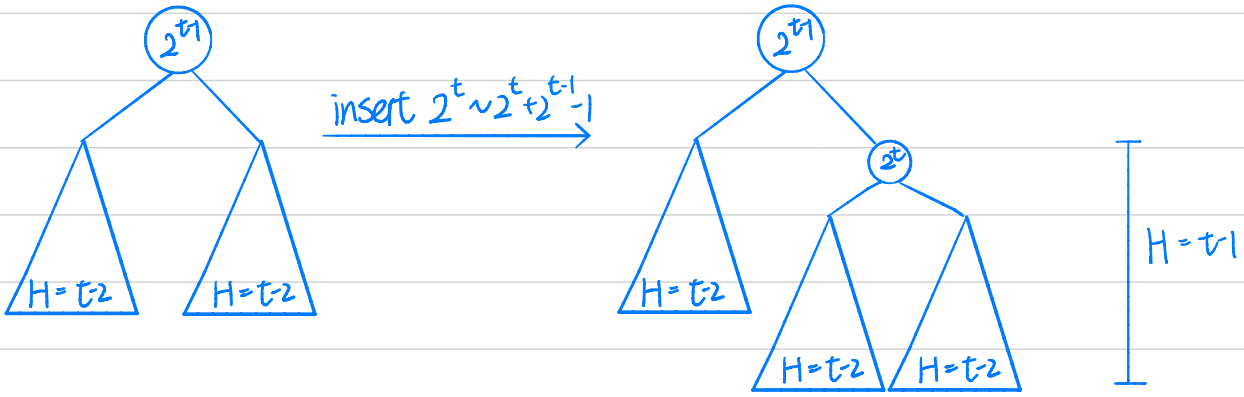
insert: $1 \sim 2^t - 1$

root: 2^{t-1}

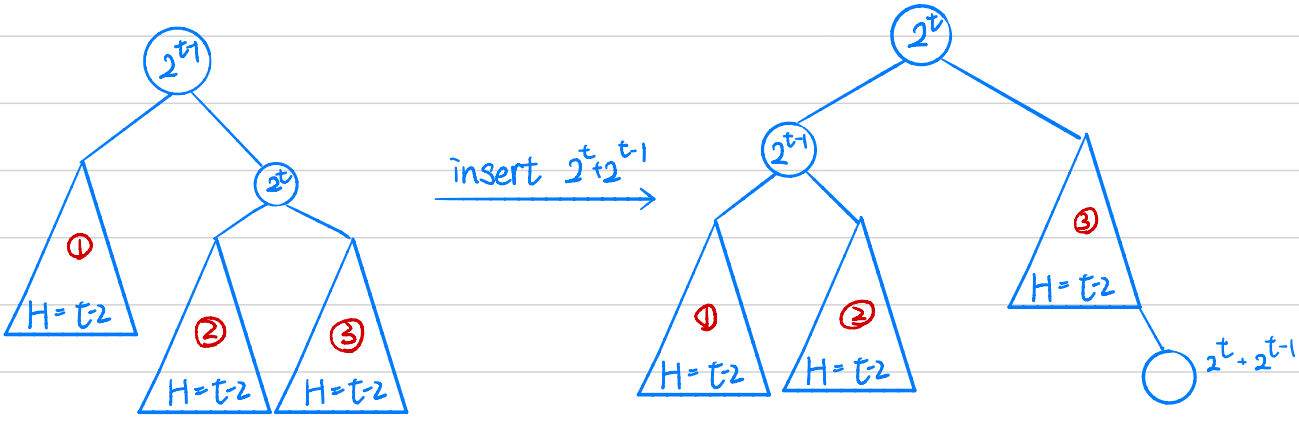
height: $t-1$ (root is 0)

3° Consider when $k = t+1$ from the hypothesis of $n = t$ is correct.

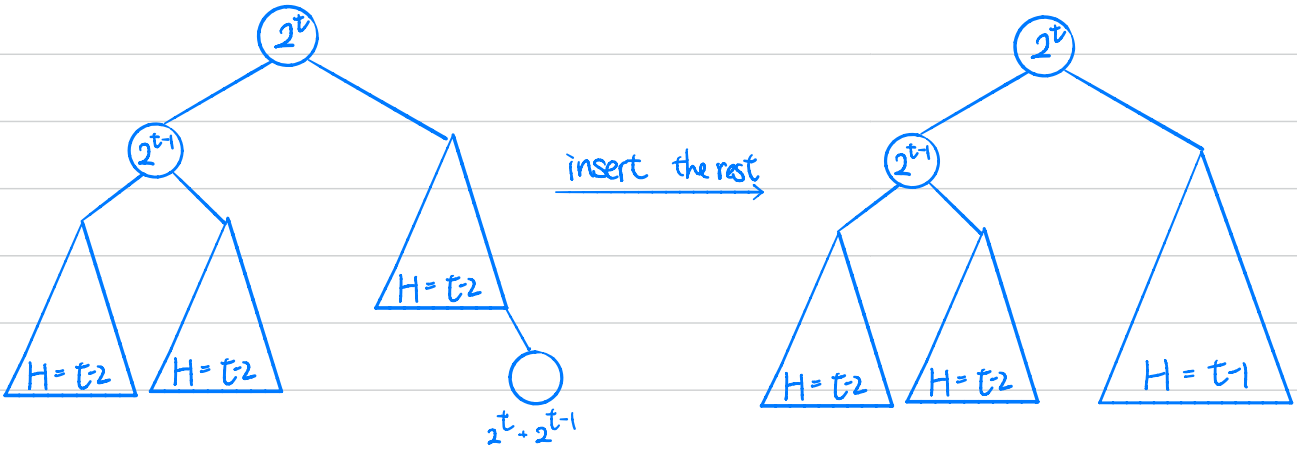
With the previous $2^t - 1$ insertion, the root is 2^{t-1} with height of $t-1$. Since the right subtree has the height of $t-2$, it can takes in the next 2^{t-1} insertion with the height of right subtree to be $t-1$ (height difference of root is 1). So we insert from 2^t to $2^t + 2^{t-1} - 1$ into the right subtree, making it contains $2^{t-1}+1$ to $2^t + 2^{t-1} - 1$. And to make the subtrees of right child to balance, the right child must be the median of the whole right subtree which is 2^t.



The we insert $2^t + 2^{t-1}$, which makes the height difference for root becomes 2, so we perform a RR rotate, making 2^t the root.



Then since the root is now 2^t, the following insertions from $2^t + 2^{t-1} + 1$ to $2^{t+1} - 1$ will be in the right subtree, and containg numbers from $2^t + 1$ to $2^{t-1} - 1$ making it a complete binary tree with height of $t-1$.



So at last, the height of left subtree and right subtree are both $t-1$, making it a complete binary tree. Thus, it also holds when $k = t+1$.

Thus from the Induction method ①, ②, ③, for sequential insertion $1 \sim 2^k - 1$ wher $k \in N$, the produced AVL Tree is a complete binary Tree. QED

2. Balanced Binary Search Tree

②

	+1~2048	-1~1024	+2049~4096
bst	2047	1023	3071
avl	11	10	11
rb	19	17	20

Since I didn't call libavl's traversal function, bst_balance is thus not called. And we inserted increasing numbers, making the bst tilted to the right. As a result, the height of bst tree is way higher than that of a rb tree and avl tree.

Conclusively, bst tree has the worst balance following rb tree. AVL tree has the best balance.

③

after first insertion

	MAX	min	avg.
bst	35	19	24
avl	13	12	12
rb	13	12	13

after removal

	MAX	min	avg.
bst	33	16	21
avl	12	10	11
rb	13	11	12

after 2nd insertion

	MAX	min	avg.
bst	36	20	25
avl	13	12	13
rb	14	13	13

For random input we can see that height : bst > rb > avl
However, different from the edge case of hw6_2-2, for a more generalized input, rb tree and avl tree are balancing good enough in contrast to bst. But AVL are still more height-bounded in usual.