**Advanced Algorithms**

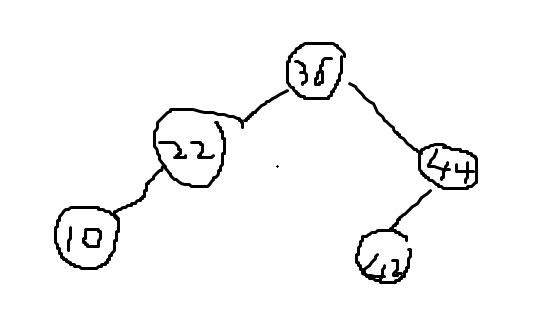
**Exercise for Lecture 5,6,7**

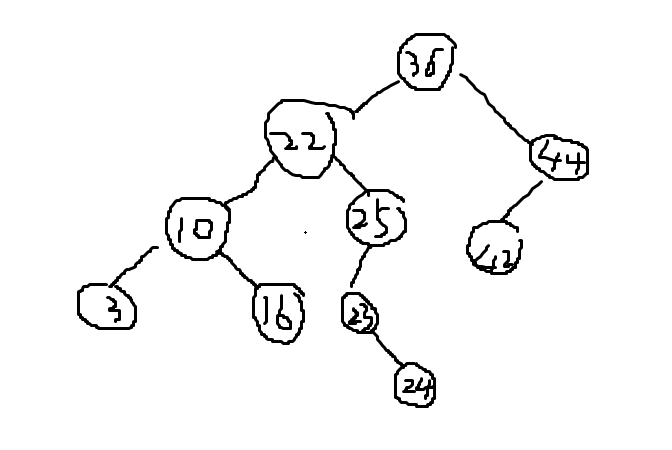
|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** | **周思源** | **Student ID** | 24S151092 |
| **Lecture 5** |  | | |
| **Lecture 6** |  | | |
| **Lecture 7** |  | | |
| **Total Score** |  | | |
| **Notes** | Deadline: **2024-10-10 24:00**  Submission Format: ‘**Lecture567\_Name\_ID.docx**’, and please send to: **algo1\_23fall@163.com**.  This assignment is meant to be an evaluation of your **individual** understanding coming into the course and should be completed **without collaboration** or outside help. | | |

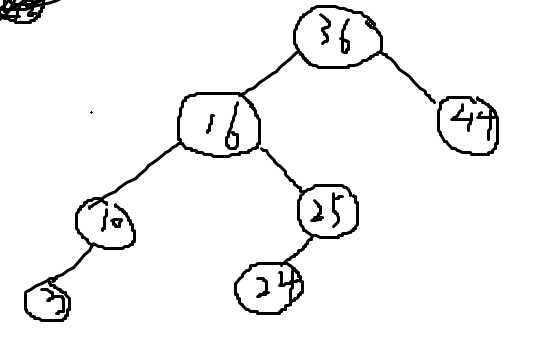
**Lecture 5**

**Problem 5.1[20 points]** Draw the BST where the data value at each node is an integer and the values are entered in the following order: 36, 22, 10, 44, 42.Then, add 16, 25, 3, 23, 24 in this order, and again draw the tree.Then draw the tree after deletions of 42, 23 and 22 in this order.

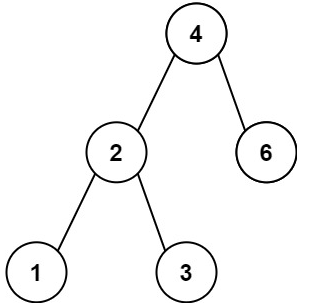
**Solution**:







**Problem 5.2[15 points]** Given the root of a binary search tree T, write an algorithm Minimum-Difference-BST(T) which returns the minimum difference between the values of any two different nodes in the tree.The difference is a positive number, equal to the absolute value of the difference between the two values. For example, the minimum difference of the following tree is 1.



**Solution**:

void dfs(struct TreeNode\* root, int\* pre, int\* ans){

if(root == NULL){

return;

}

dfs(root->left, pre, ans);

if(\*pre == -1){

\*pre = root->val;

}

else{

\*ans = fmin(\*ans, root->val - (\*pre));

\*pre = root->val;

}

dfs(root->right, pre, ans);

}

int minDiffInBST(struct TreeNode\* root) {

int ans = INT\_MAX;

int pre = -1;

dfs(root, &pre, &ans);

return ans;

}

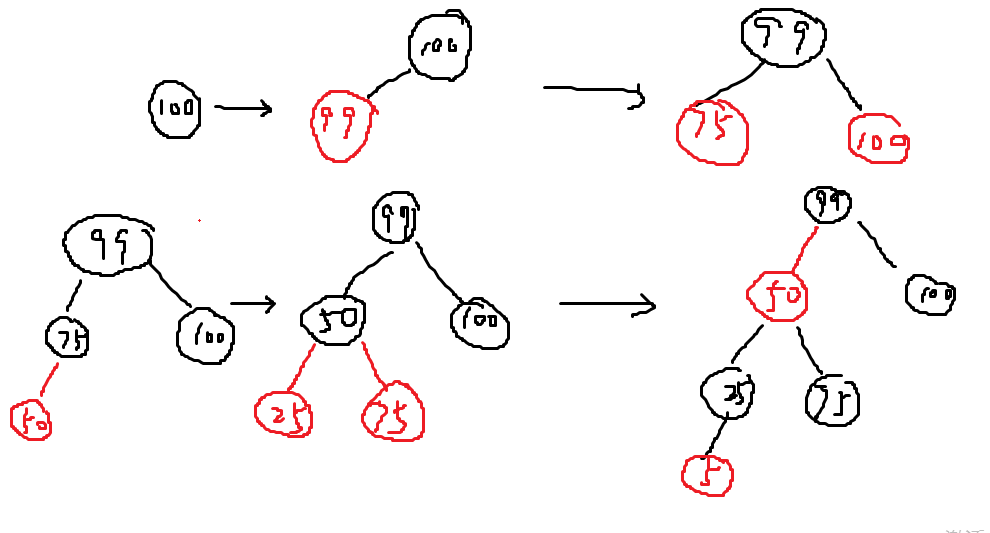
**Lecture 6**

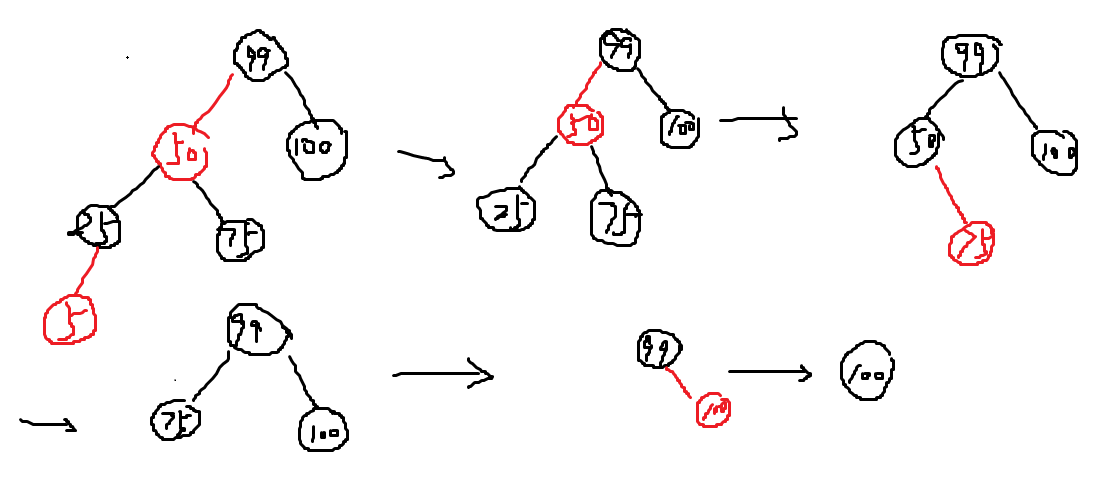
**Problem 6.1[20 points]**

(1) Please insert key value 100, 99, 75, 50, 25, 5 into a red black tree one by one. Please show each intermediate state of the red black tree.

(2) Please delete key value 5, 25, 50, 75, 99, from the red black tree one by one which you get in (1). Please show each intermediate state of the red black tree.

**Solution:**





**Problem 6.2[15 points]** Consider a red-black tree formed by inserting n nodes into an initially empty red-black tree. Argue that if n > 1, the tree has at least one red node. Your argument must be in the form of a proof by induction. There are two cases you will need to take into consideration when making your inductive step. The trivial case when Red Node N+1 is inserted as a child of a black node does not need to be solved.

**Solution**:

Case 1:z and z.p.p are red after the fix up, then the loop didn’t stop but the red-black tree has at least two red nodes.

Case 2: this case is same as case 3 after the rotation fix up.

Case 3: after the fix up, z become red but not the root, so the loop was broken at that time, the red-black tree has at least a red node.

The last of the insert operation suppose the root node black, but only need to fix up when it is red, so there is only case 1 has at least two red nodes, sub one node there is still a red node at least.

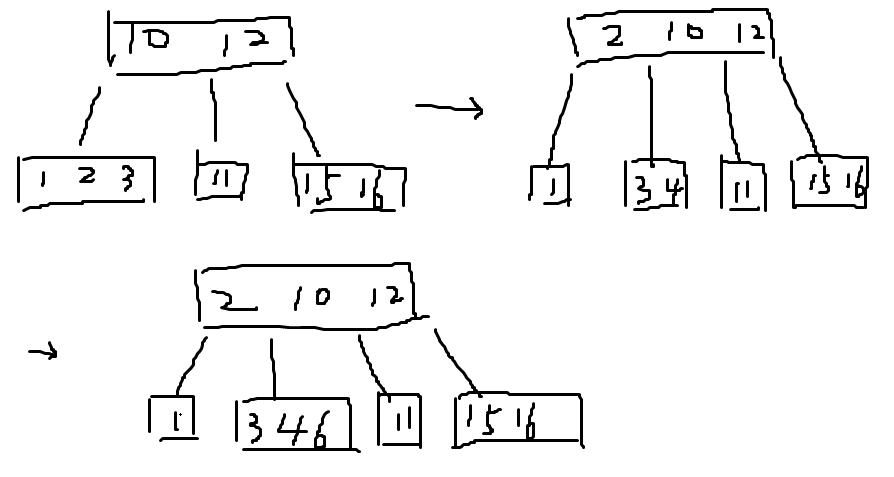
Above all, if n > 1, the red-balck tree has at least a red node.

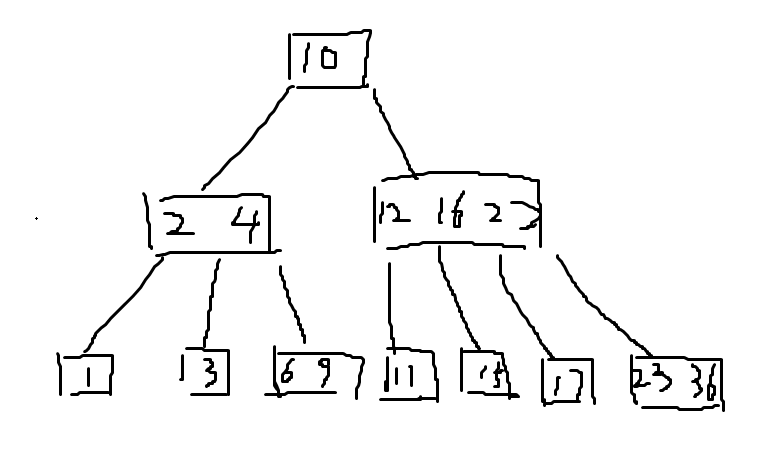
**Lecture 7**

**Problem 7.1[20 points]** Show the results of inserting the keys 10, 11, 1, 2, 12, 16, 15, 3, 4, 6, 22, 17, 36, 9, 23 in order into an empty B-tree with minimum degree 2. Draw the configurations of inserting 4,6 and the final configuration.

**Solution**:

If the minimum degree is 2, the maxmum degree should be 4, so every node can have at least one at most three key words.





**Problem 7.2[10 points]** Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree. Provide an explanation and include relevant pseudocode.

**Solution**:

To find the minimum key in a B-tree, we should start at the root and keep moving to the left child until reach the leaf node.

int find\_minimum(BTreeNode \*node) {

if (node->is\_leaf)

return node->keys[0];

return find\_minimum(node->children[0]);

}

To find the predecessor, w should find the predecessor whether the key has a left subtree or needs to be found by traversing upwards in the B-tree.

// find the maxmum key

int find\_maximum(BTreeNode \*node) {

if (node->is\_leaf)

return node->keys[node->num\_keys - 1];

return find\_maximum(node->children[node->num\_keys]);

}

//find the key’s index

int find\_key\_in\_node(BTreeNode \*node, int k) {

int i = 0;

while (i < node->num\_keys && k > node->keys[i])

i++;

return i;

}

// find the predecessor

int find\_predecessor(BTreeNode \*node, int k) {

int index = find\_key\_in\_node(node, k);

if (!node->is\_leaf && node->children[index]) {

return find\_maximum(node->children[index]);

}

BTreeNode \*parent = node;

while (parent != NULL && parent->keys[0] == k) {

k = parent->keys[0];

parent = parent->parent;

}

if (parent == NULL)

return -1;

return parent->keys[index - 1];

}