Data Structures:
Binary Search Trees

YoungWoon Cha
(Slide credits to Won Kim)

Spring 2022

Binary Search

Binary Search

- Search a key in $O(\log_2 n)$ time complexity
- Prerequisite
  - An ordered list
  - e.g.

7 9 2 5 15 3 14 1 8 11 12 4 13 6 10

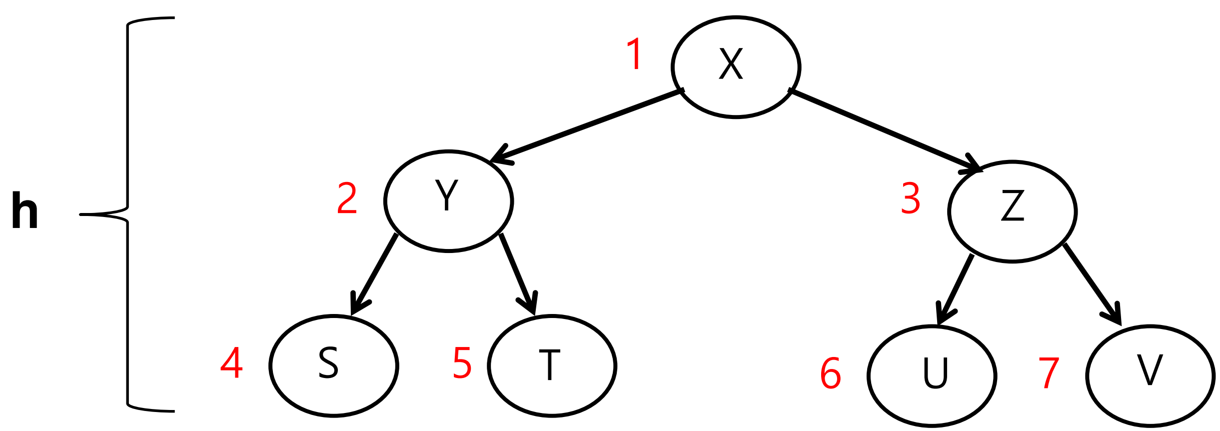
↓

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Binary Search

- Time Complexity: $O(\log_2 n)$

- In Tree Structures,
  - The maximum total number of nodes
    - $n = 2^h - 1$
  - $h \approx \log_2 n$



Binary Search: Example

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

search for 13

Algorithm

- Take the midpoint of the sorted list.
- If the search key is < the key at the midpoint, take the list to the left of the midpoint, and repeat from the start.
- If the search key is > the key at the midpoint, take the list to the right of the midpoint, and repeat from the start.
- If the search key matches the key at the midpoint, search ends in success.
- If the list contains only one key and the search key does not match it, search ends in failure.

Binary Search: Solution

take the midpoint of the list and compare with 13

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

take the list after 8

9 10 11 12 13 14 15

take the midpoint of the list and compare with 13

9 10 11 12 13 14 15

take the list after 12

13 14 15

Binary Search: Solution (cont.)

take the list after 12

13 14 15

take the midpoint of the list and compare with 13

13 14 15

take the list before 14

13

take the midpoint of the list and compare with 13

13

Binary Search: Examples

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

search for 5

1 3 4 5 7 8 9 10 12 13 14 16 20 21 25

search for 4
search for 22

Binary Search: Examples

1 2 3 4 5

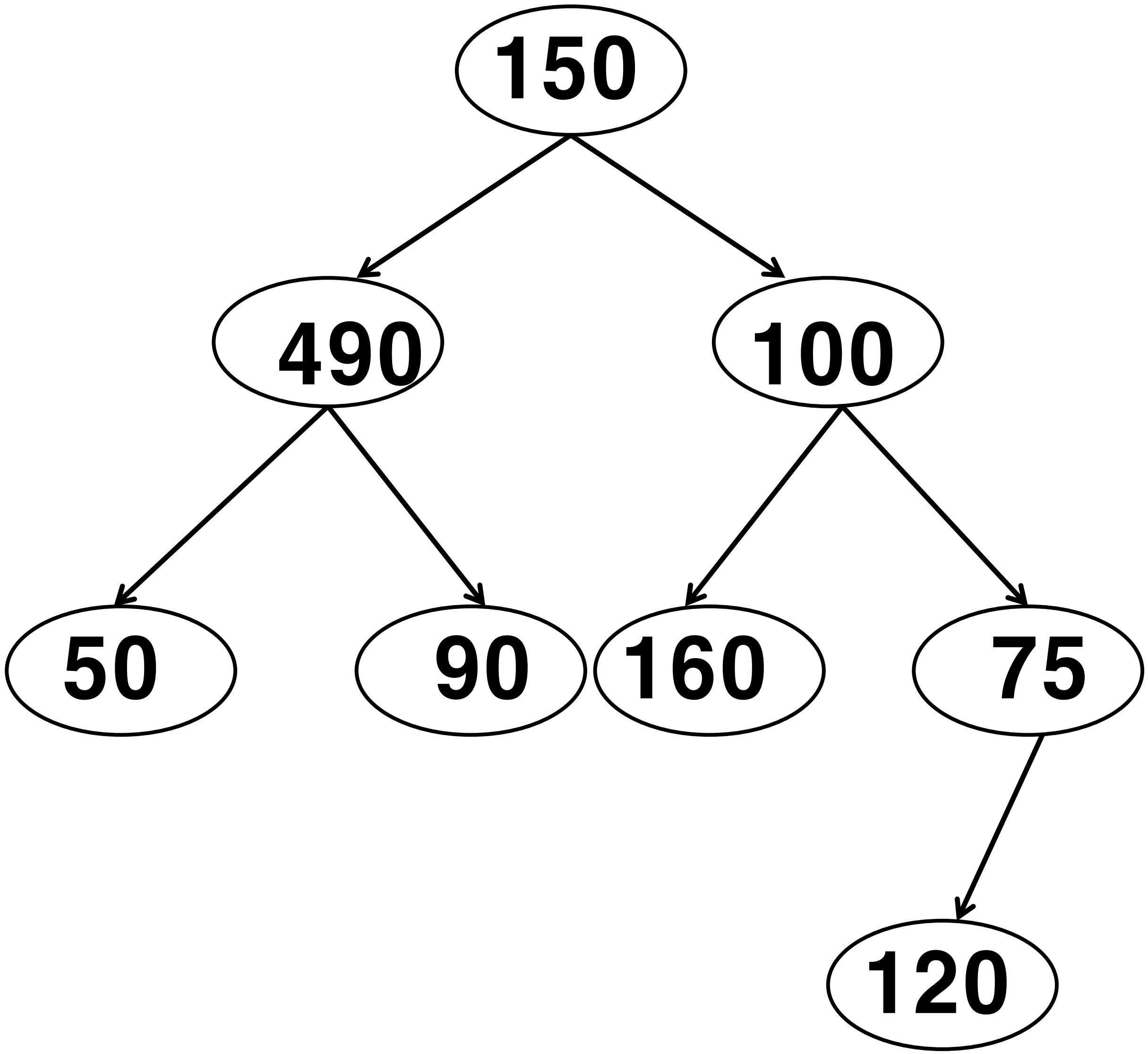
search for 3, 1, 5, 9

1 2 3 4

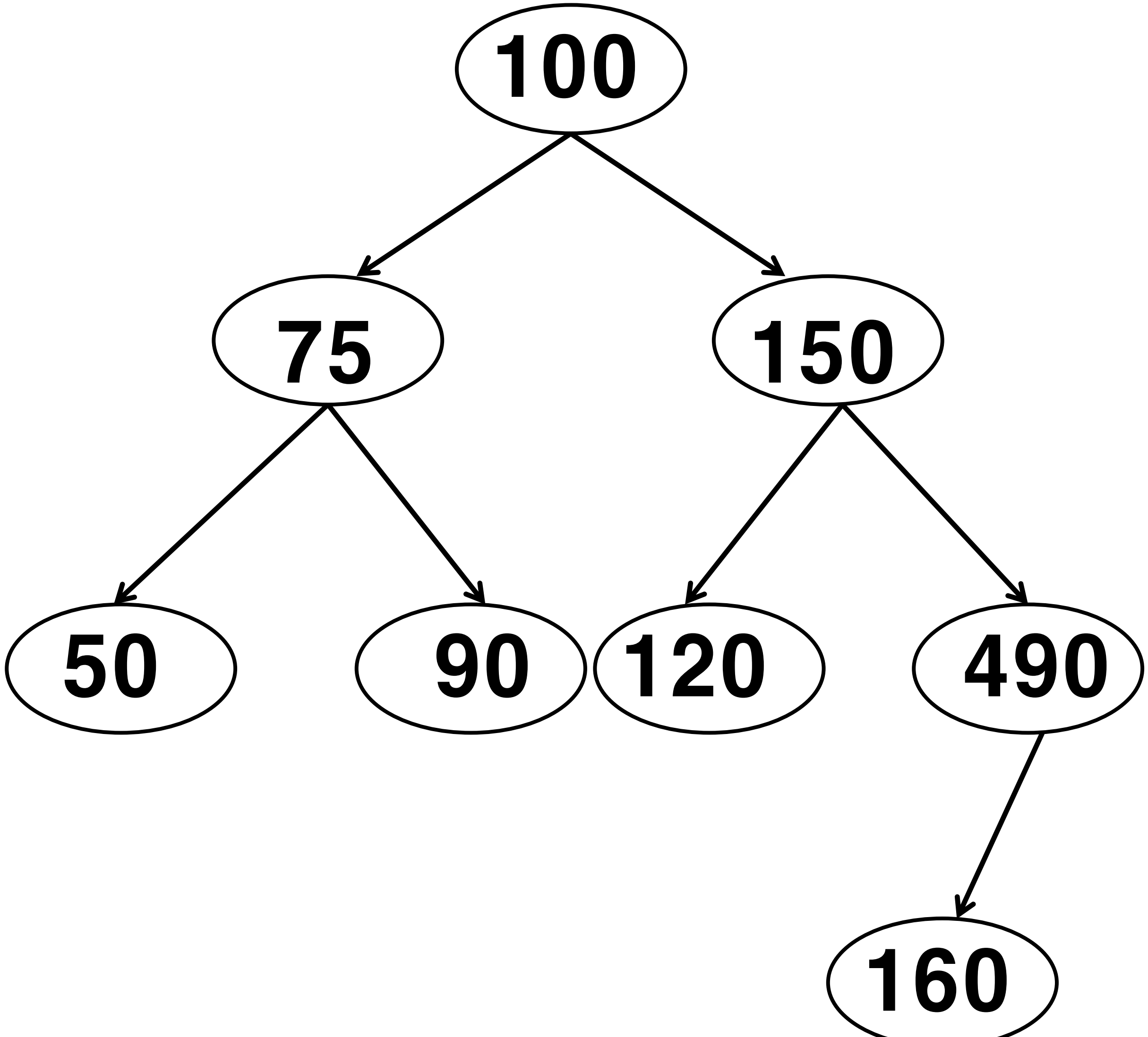
search for 1, 3, 2, 4, 9

Binary Search Trees

Binary Tree (no order)



Binary Search Tree (ordered)



Properties of (ordered) Binary Search Tree

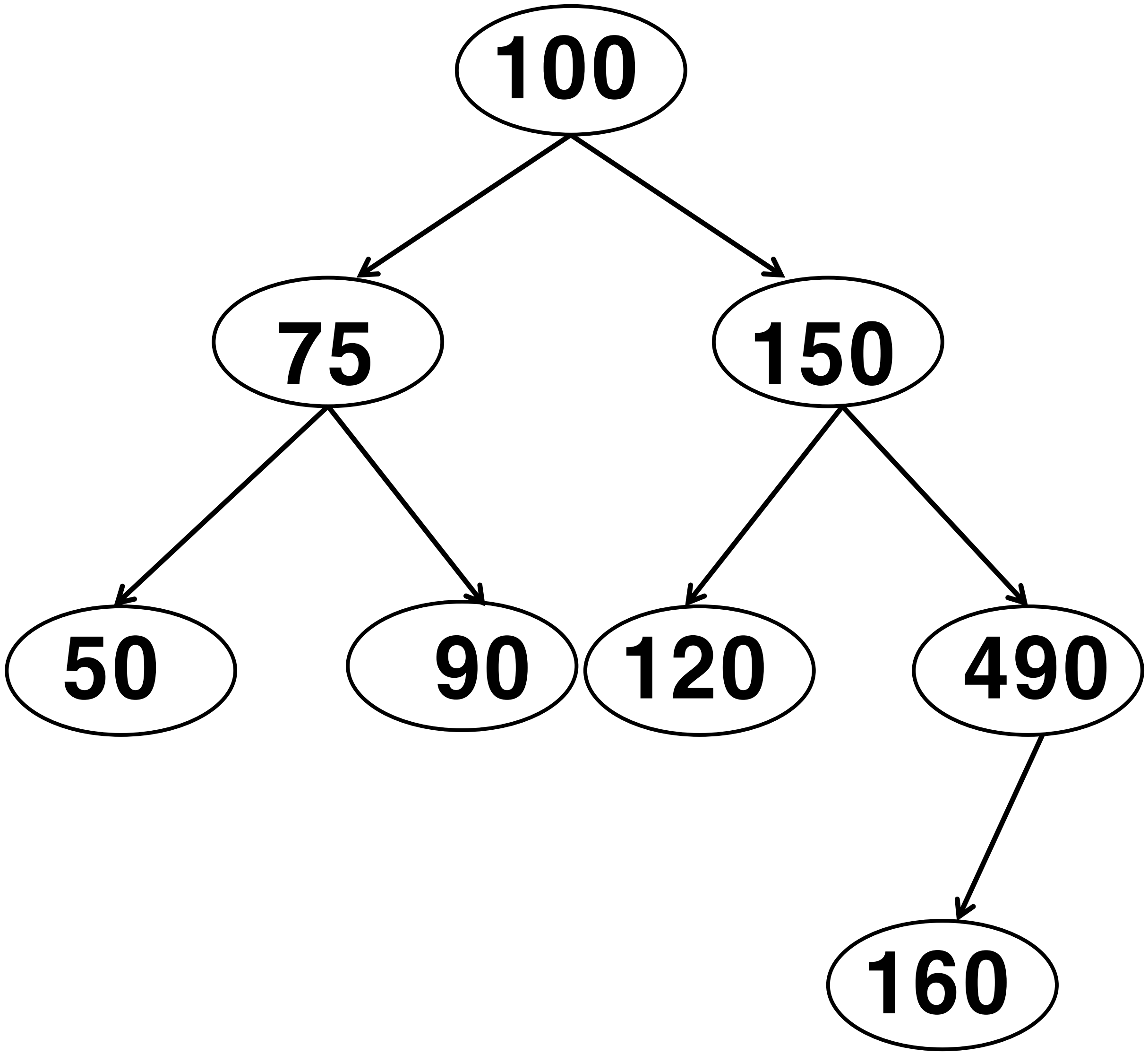
- Each Node Has a Unique Key (Data)
- The Key < The Key of Any Node in the Right Subtree
- The Key > The Key of Any Node in the Left Subtree

Smaller to the Left, Larger to the Right

Searching a Binary Search Tree

- Compare the Key to Be Searched with the Key of the Root Node.
- If the Search Key = the Root Key, Success.
- If the Search Key < the Root Key, Search the Left Subtree.
- If the Search Key > the Root Key, Search the Right Subtree.
- If No Match, Failure.

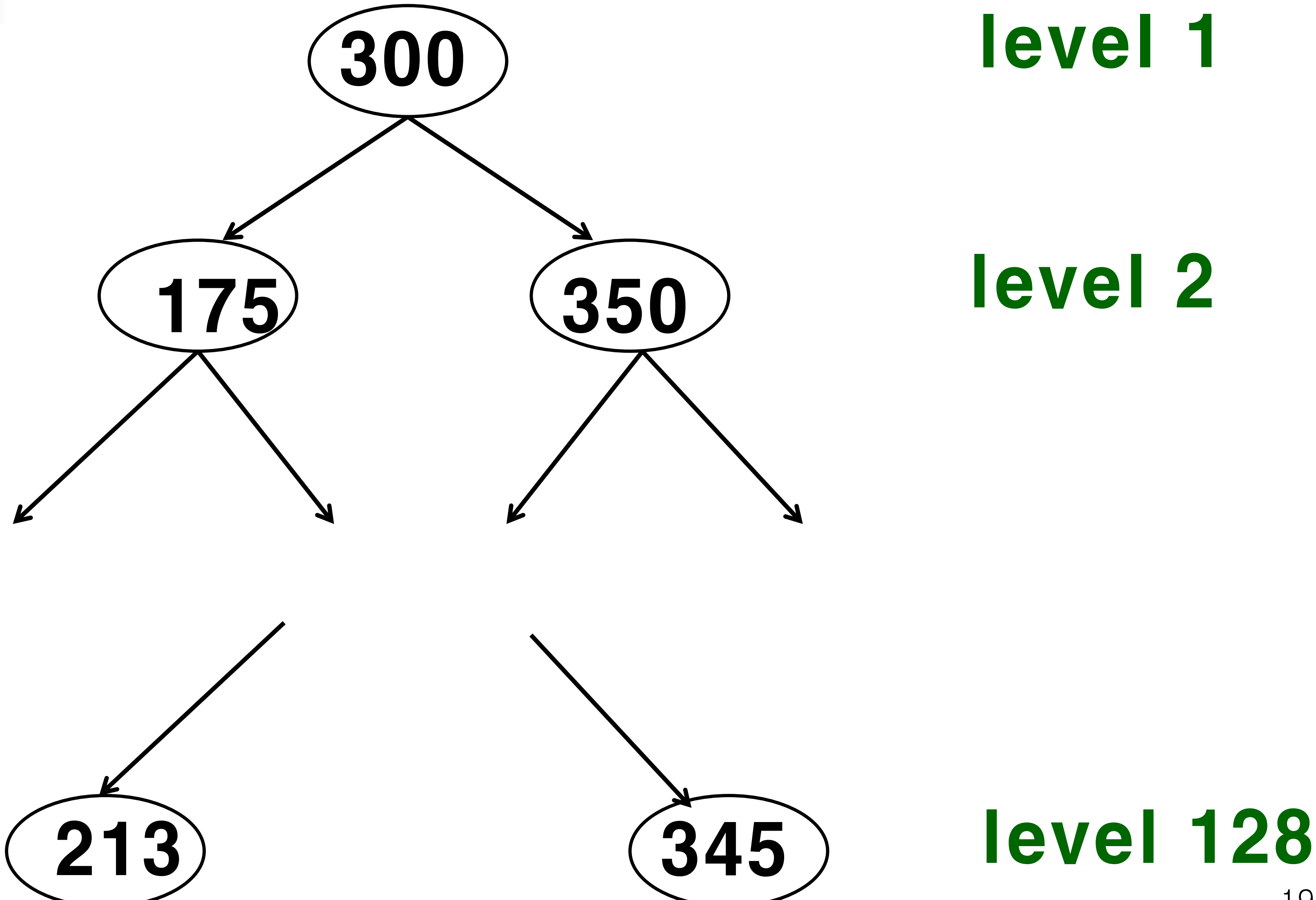
Example search 120, search 400



Caution (1/2): complex "key", large number of nodes

- In Textbooks
  - "key": a small number or short string
    - e.g., 250, "Hong Gil Dong"
  - "number of nodes": 5-15
  - "number of levels": 3-5
- In Real Software
  - "key": a structure, array, array of structures
    - e.g., [xxxxxxxxx, "Hong Gil Dong", 3xxxxx, "Suwon"]
  - "number of nodes": 100, 1000, 10000, 1000000,…

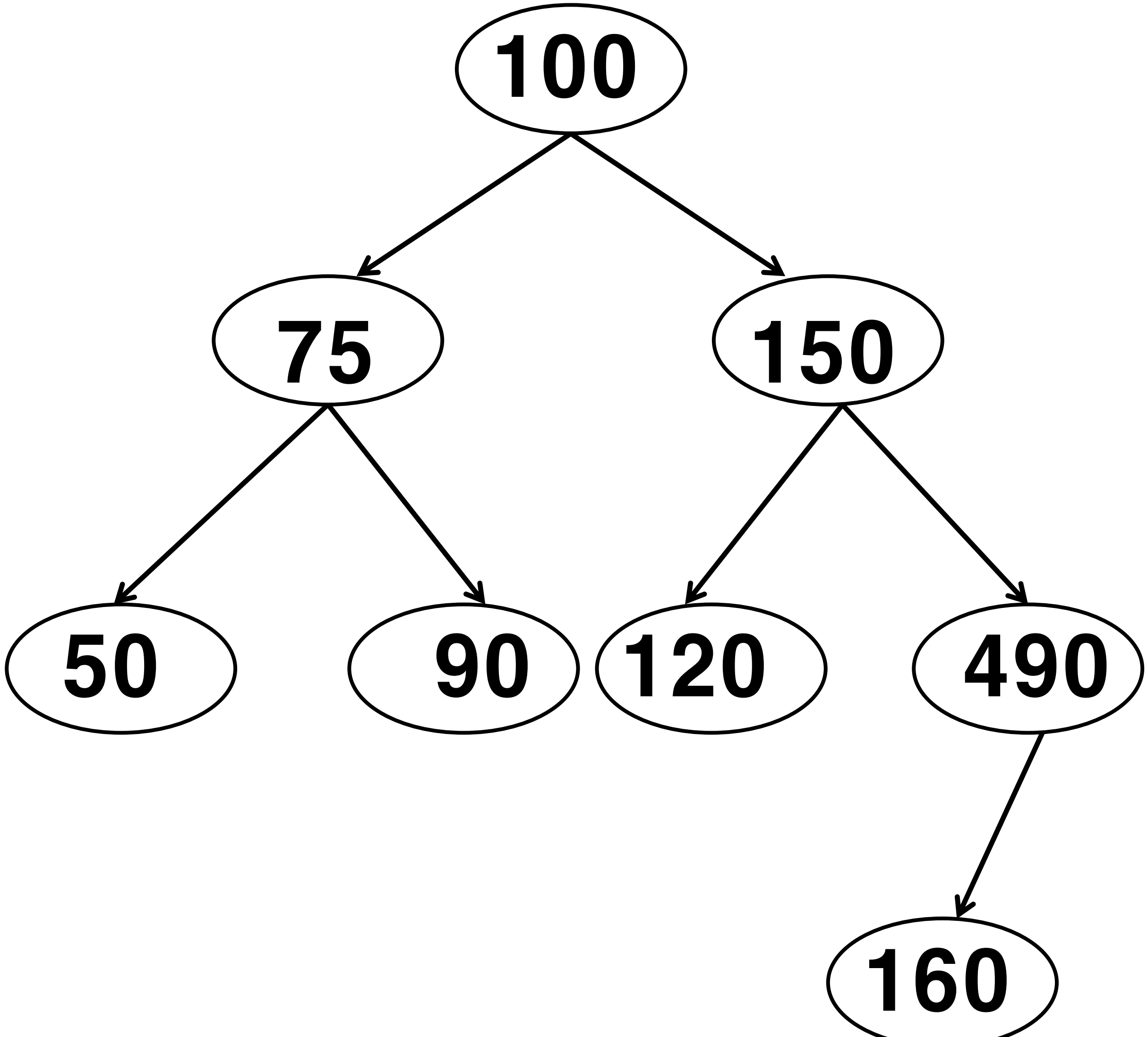
Caution (2/2): large number of levels



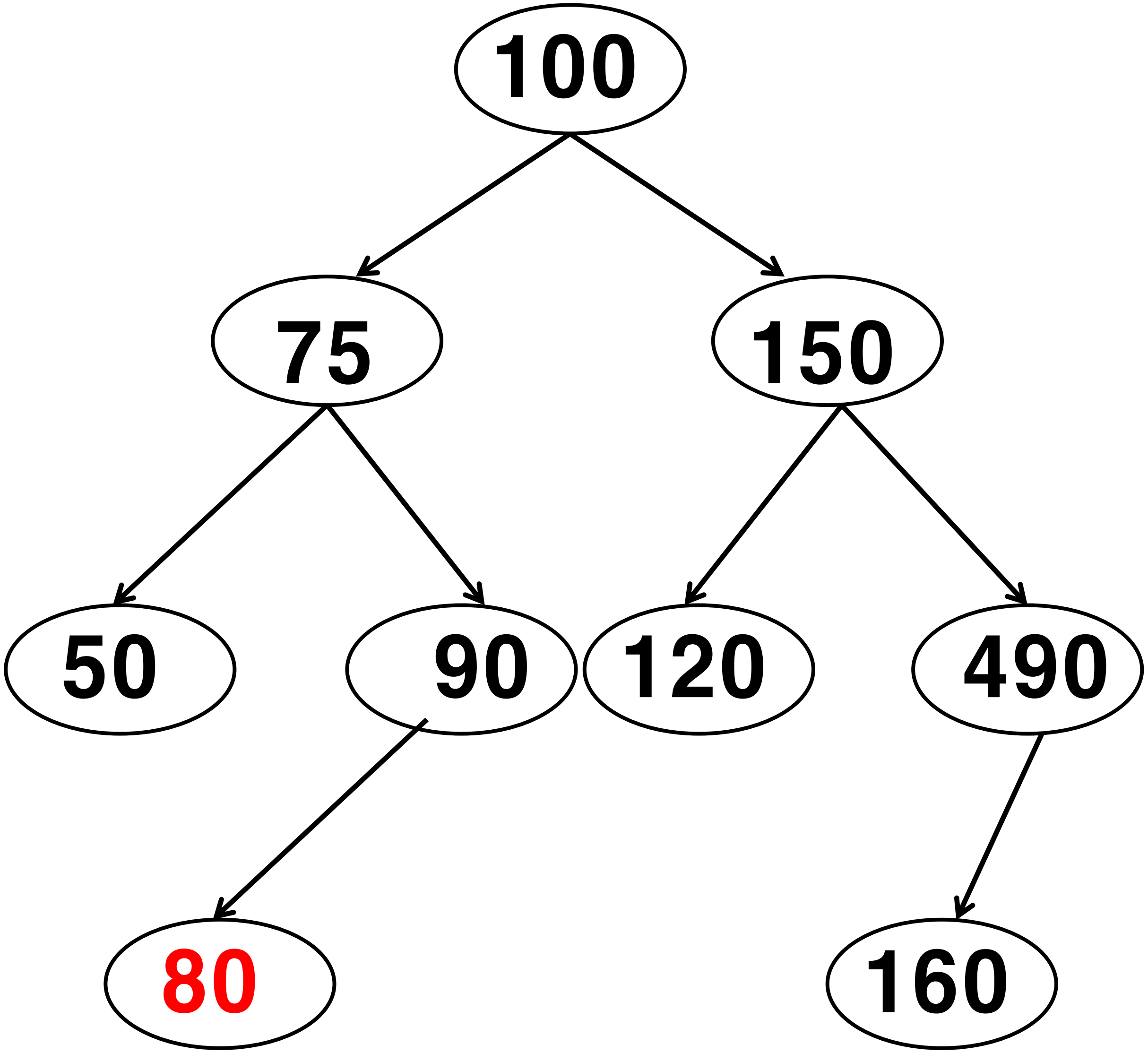
Inserting into a Binary Search Tree

- Search the Key to Be Inserted.
- If the Search Fails, Insert the Key Where the Search Failed.

Example insert 80



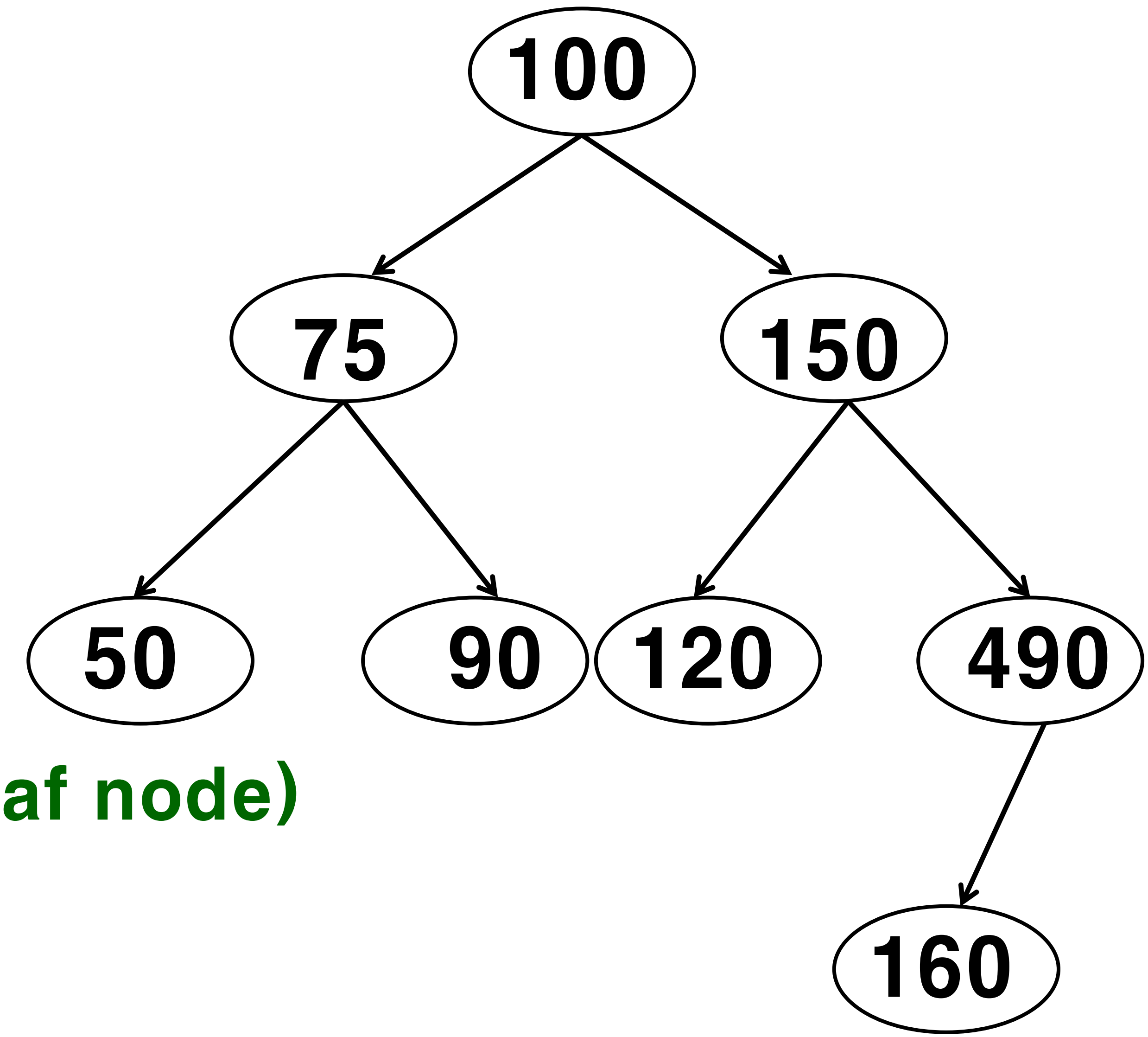
Example: Result



Deleting from a Binary Search Tree

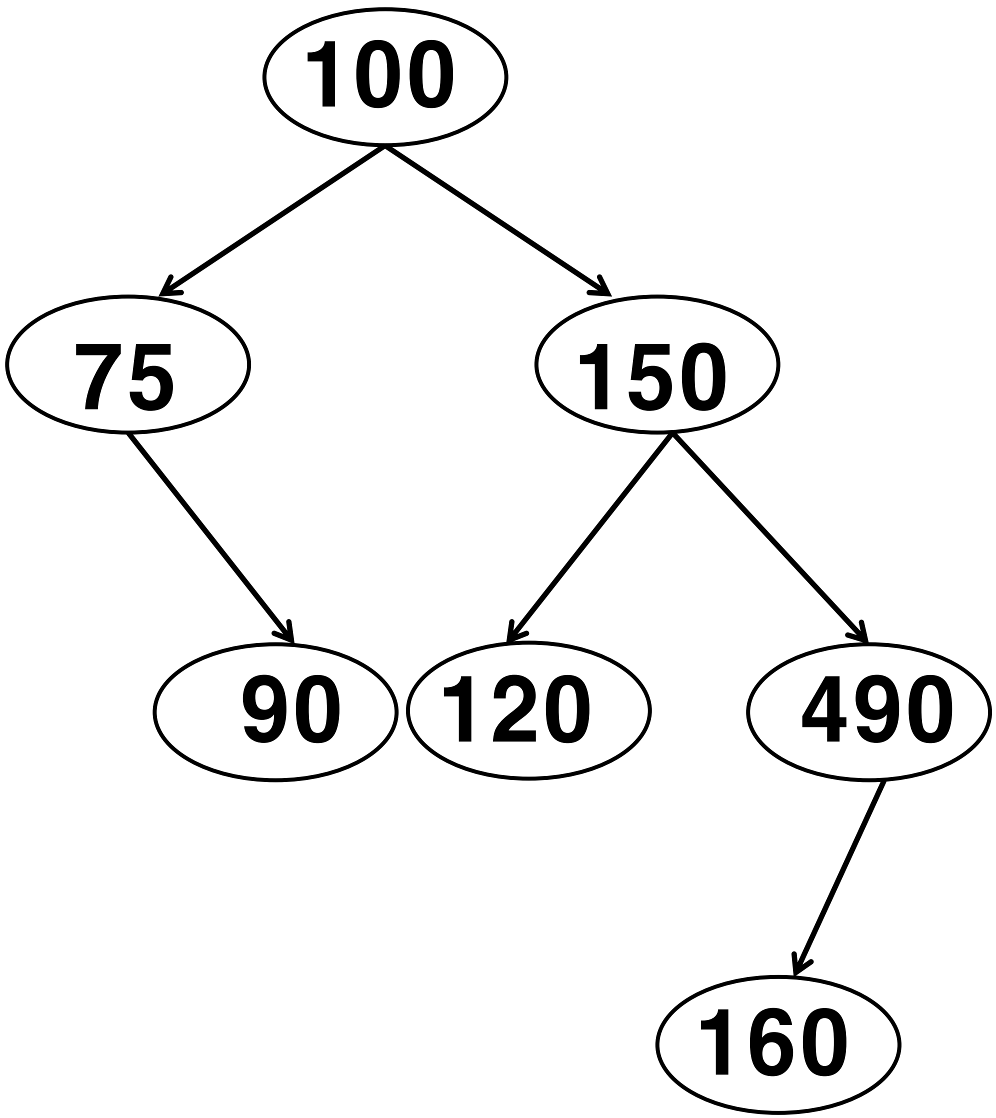
- Search the Key to Be Deleted.
- If the Search Fails, No Deletion.
- If the Search Succeeds
  - If the key is a leaf node, just delete it.
  - If the key is an interior node with one child node, delete the key, and simply promote the child node.
  - If the key is an interior node with two child nodes, delete the key, and
    - Promote the largest node in its left subtree or
    - Promote the smallest node in its right subtree.

Example (1/3) delete 50



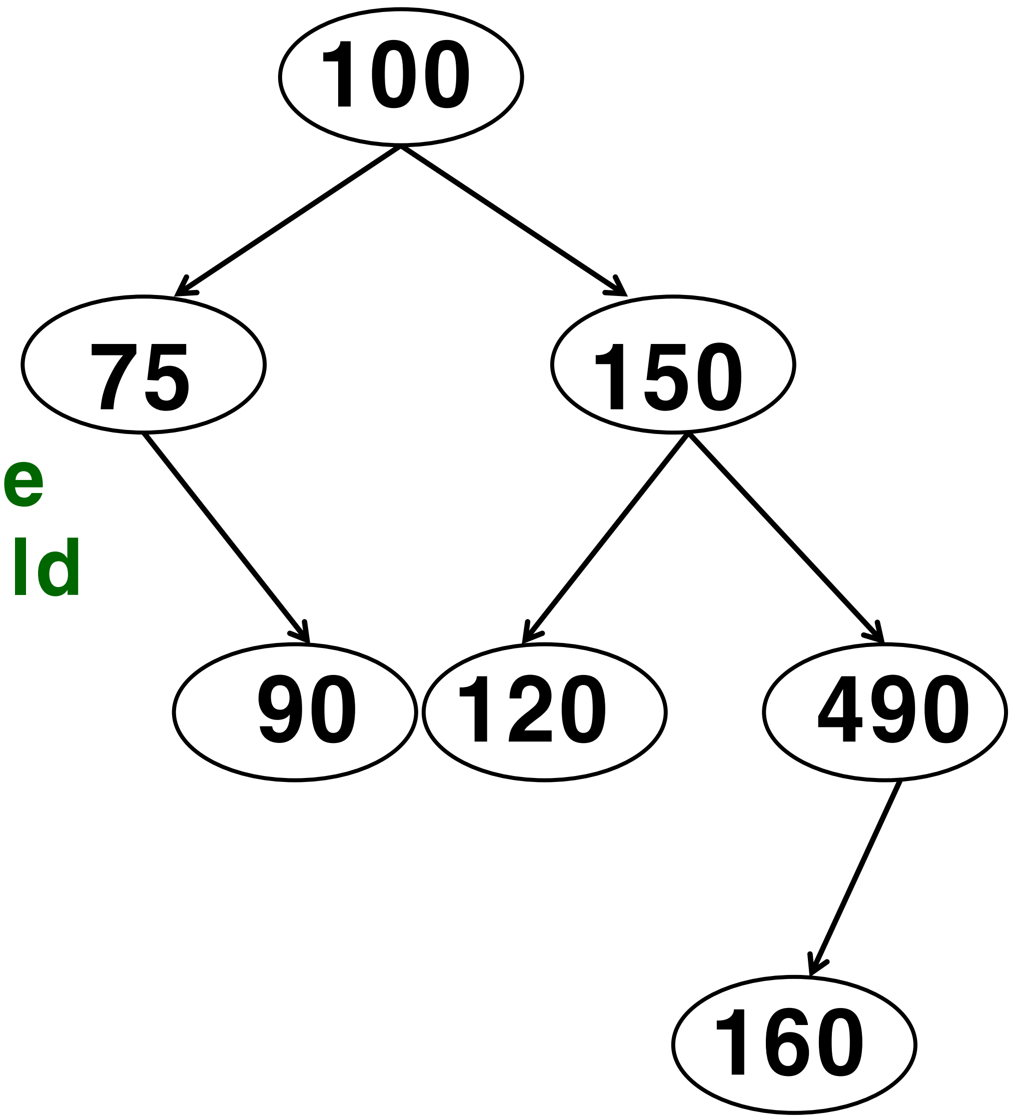
(leaf node)

Example (1/3): Result

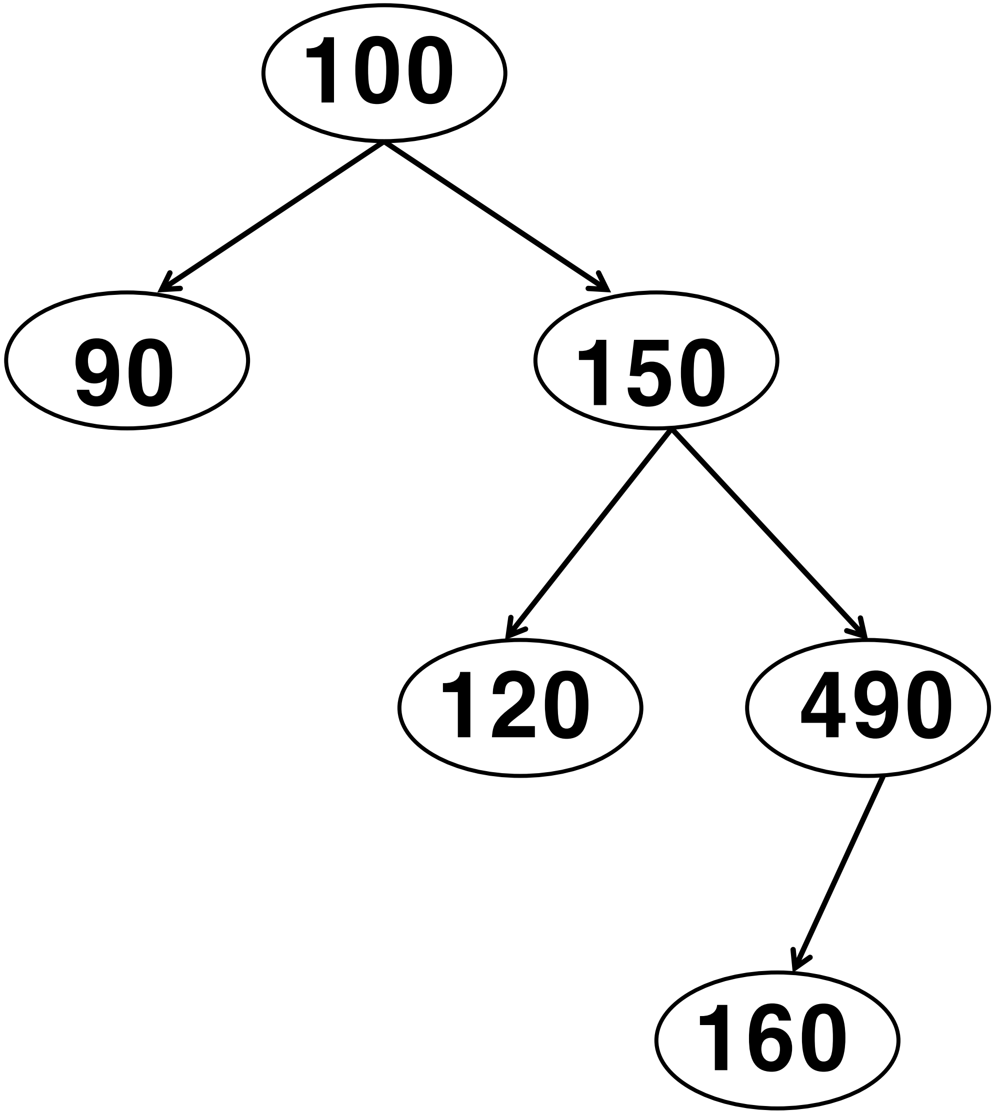


Example (2/3) delete 75

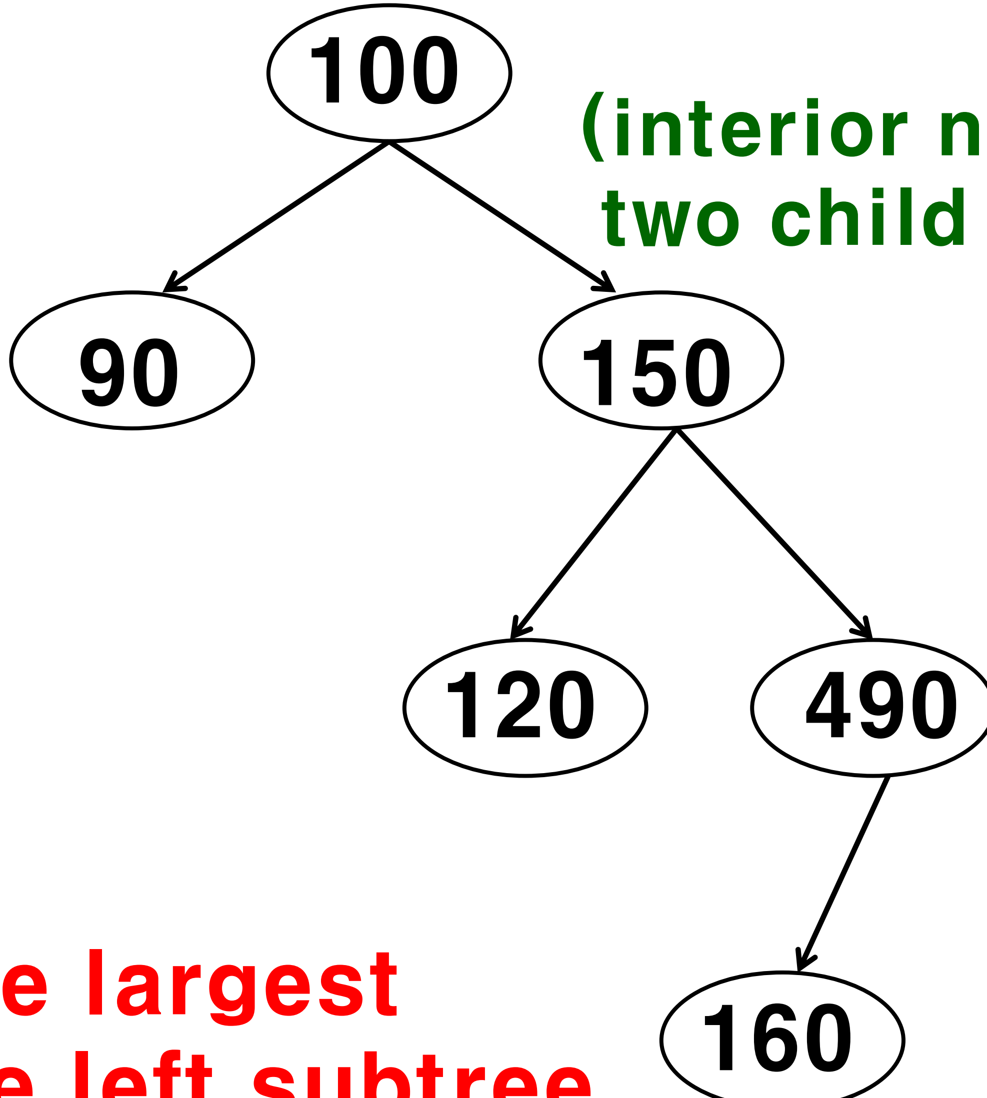
(interior node with one child node)



Example (2/3): Result



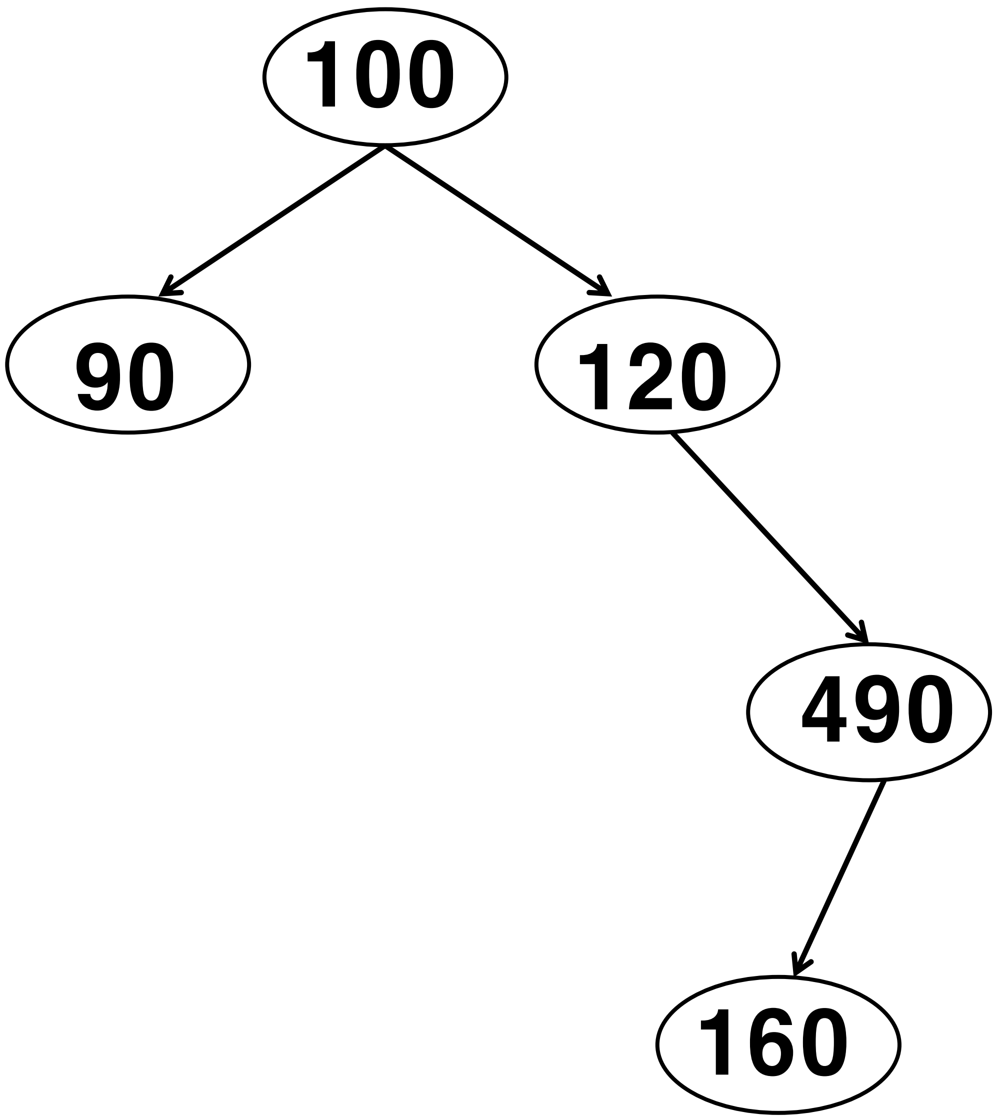
Example (3/3) delete 150



(interior node with two child nodes)

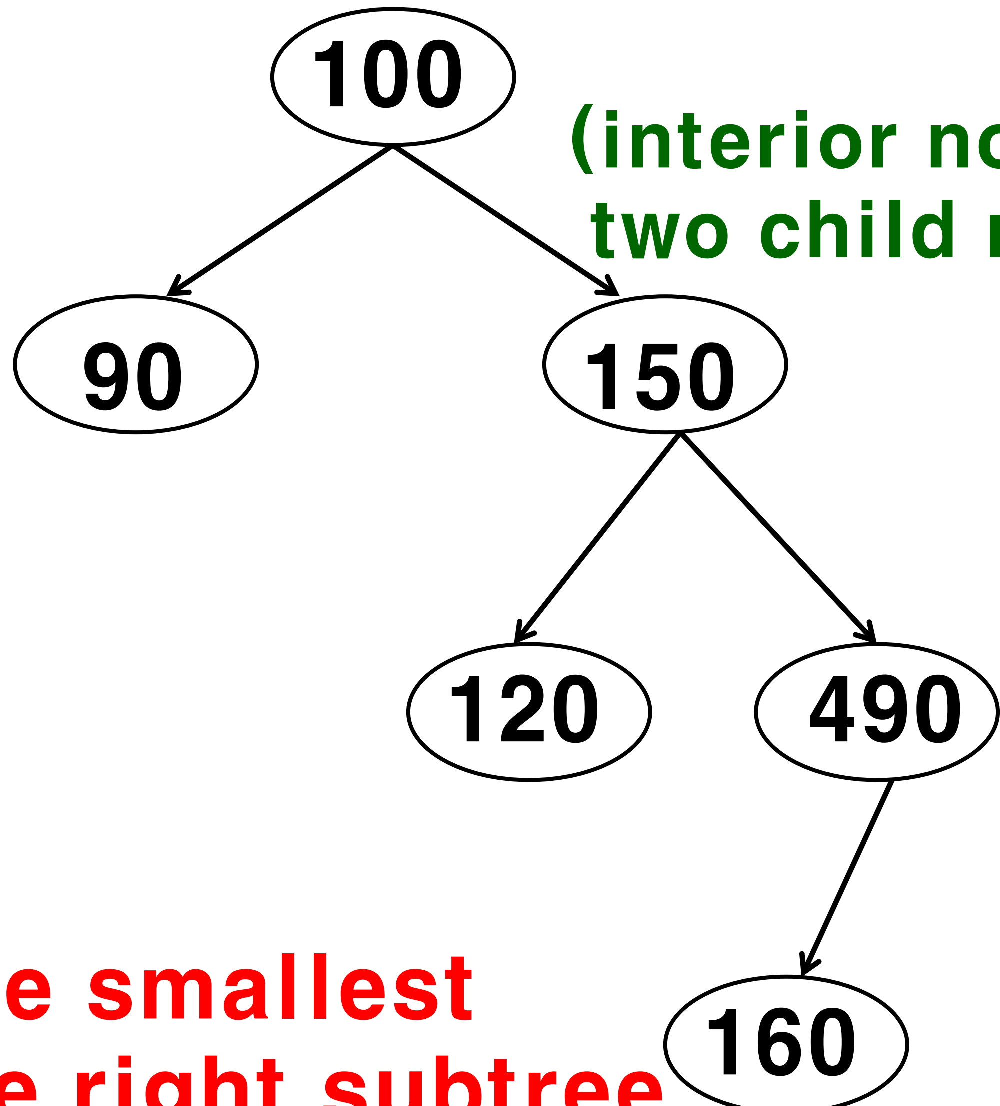
option 1:
promote the largest node on the left subtree

Example (3/3): Result (1/2)



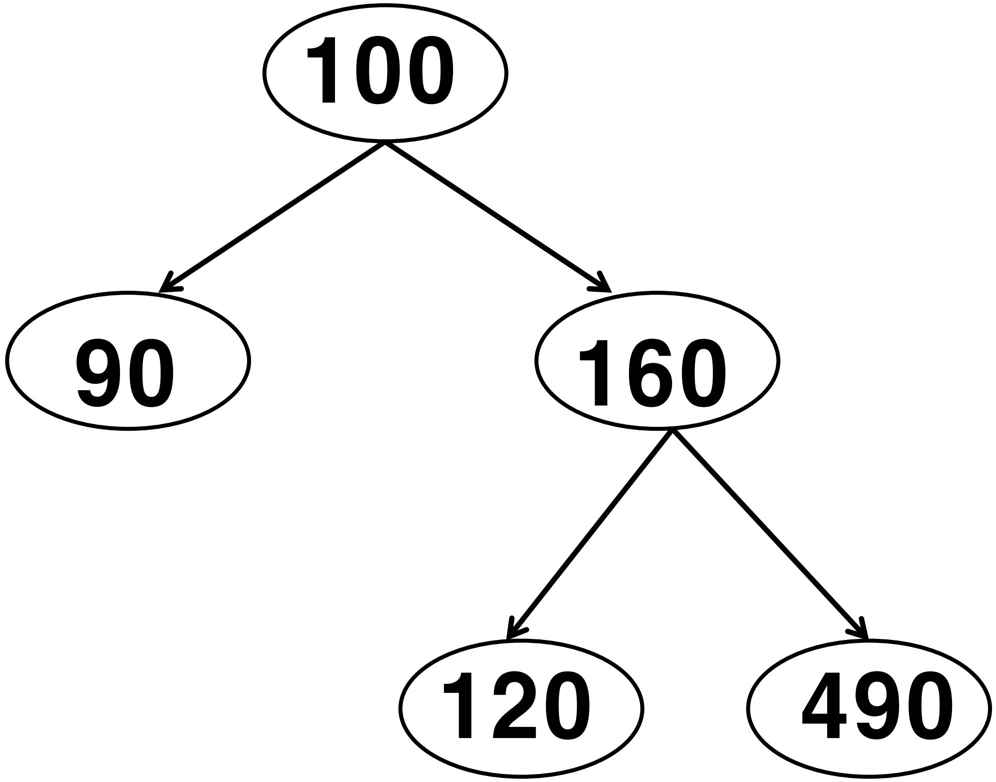
Example (3/3) delete 150

(interior node with two child nodes)

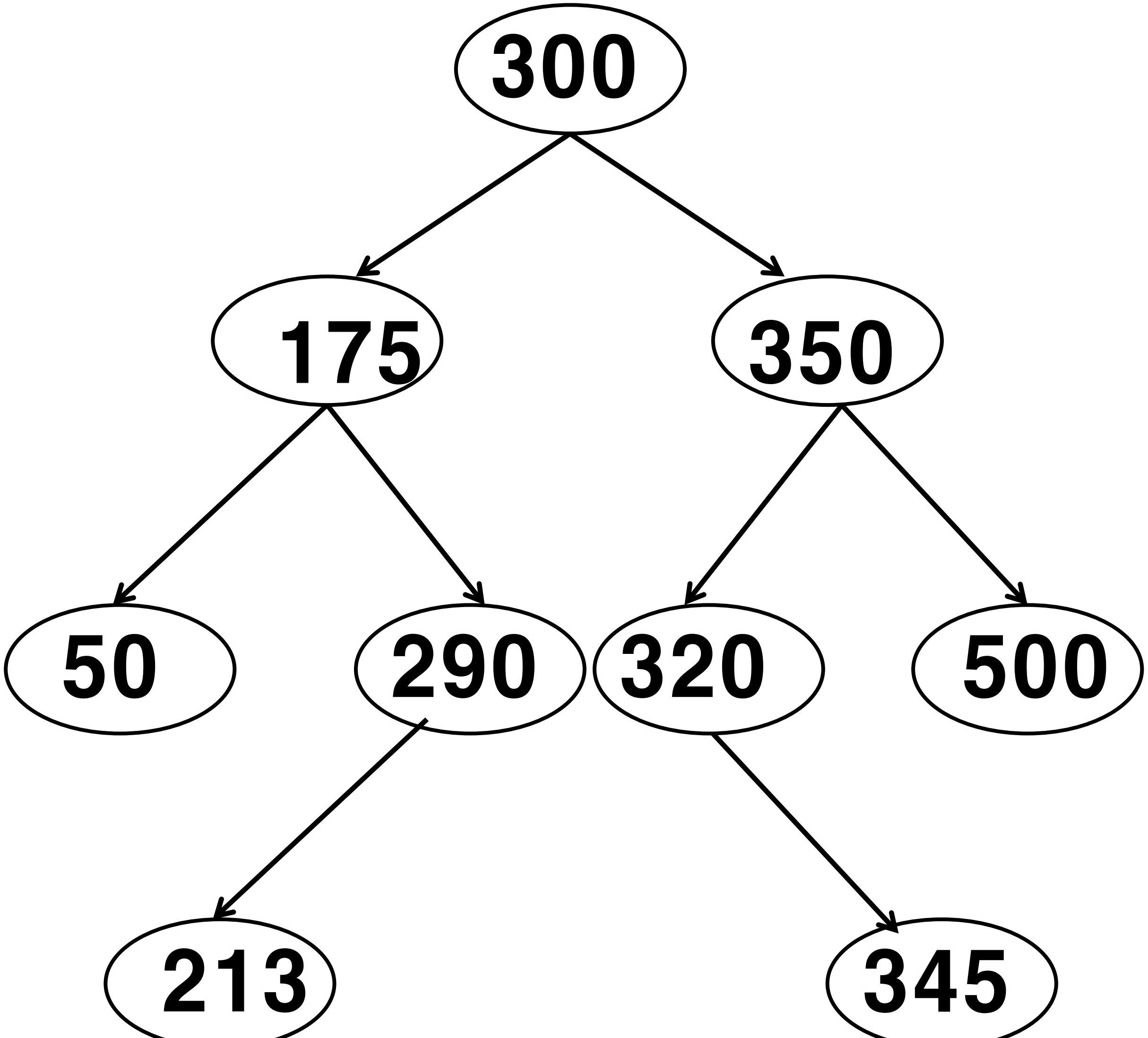


option 2:
promote the smallest node on the right subtree

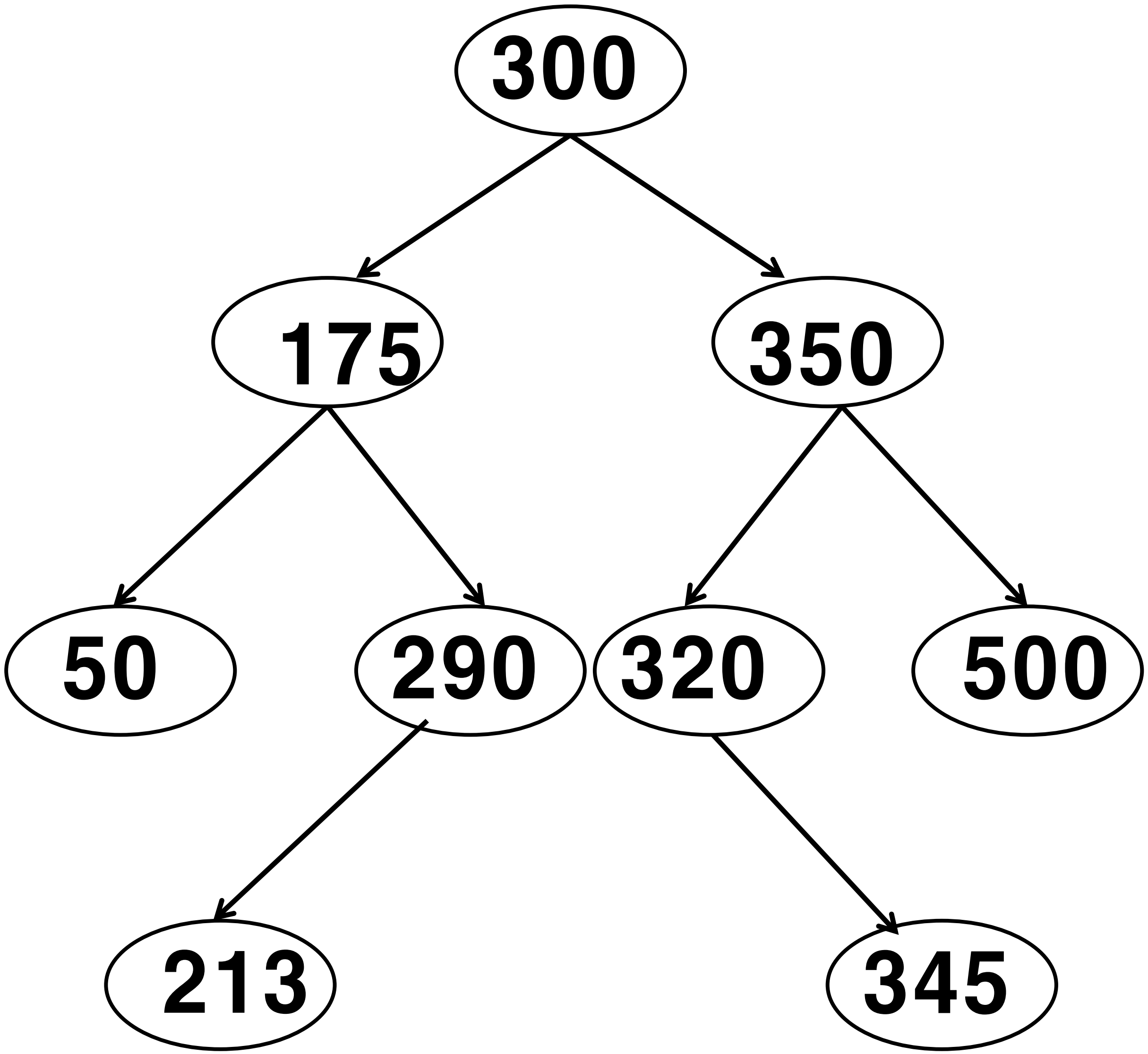
Example (3/3): Result (2/2)



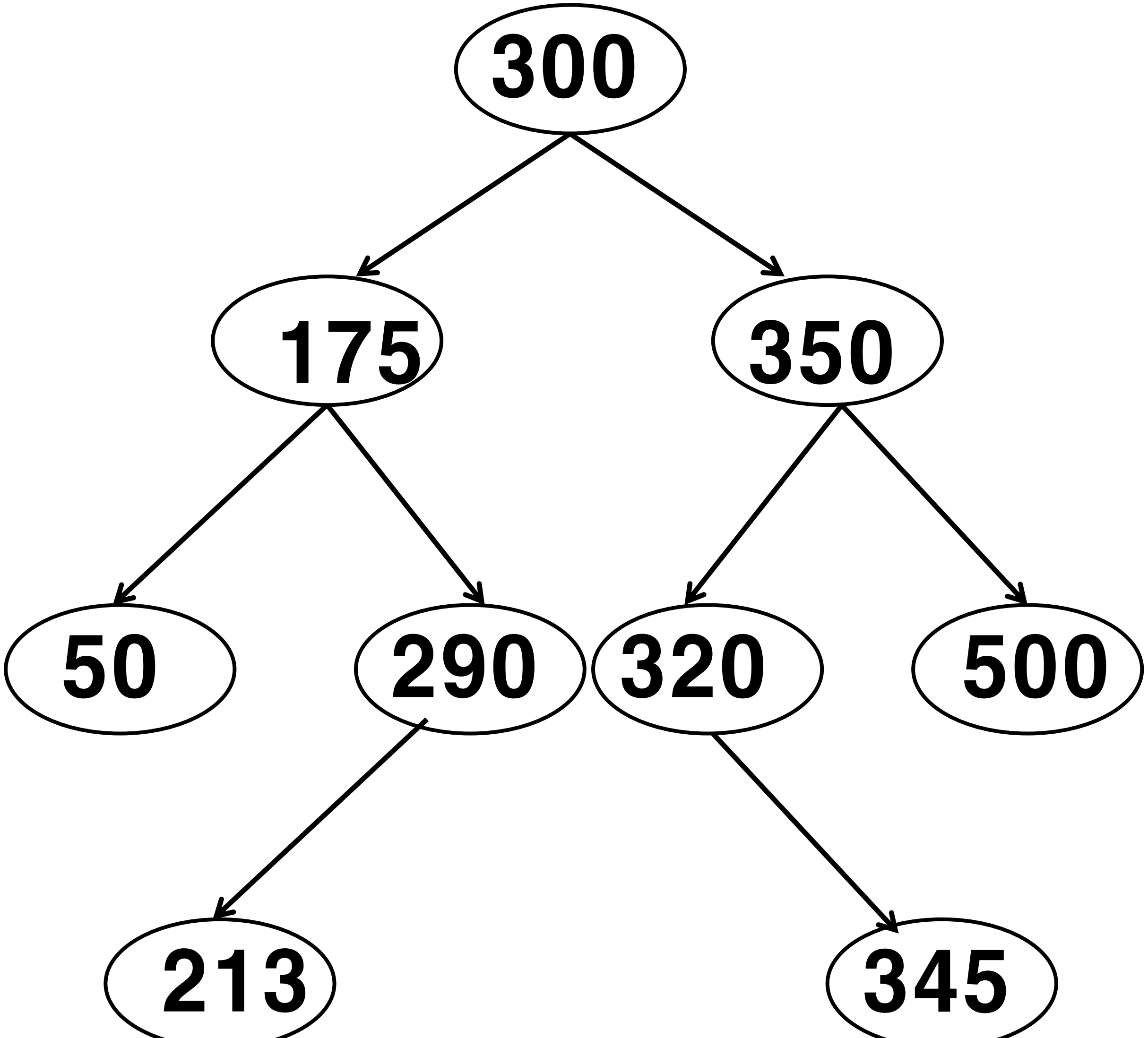
Exercise: Search for 330 – At Which Node Does the Search Fail?



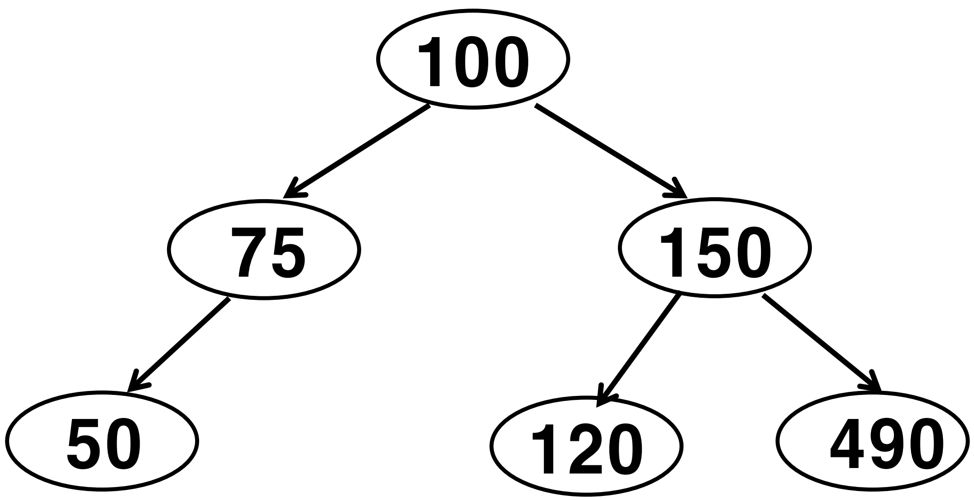
Exercise: Insert 330



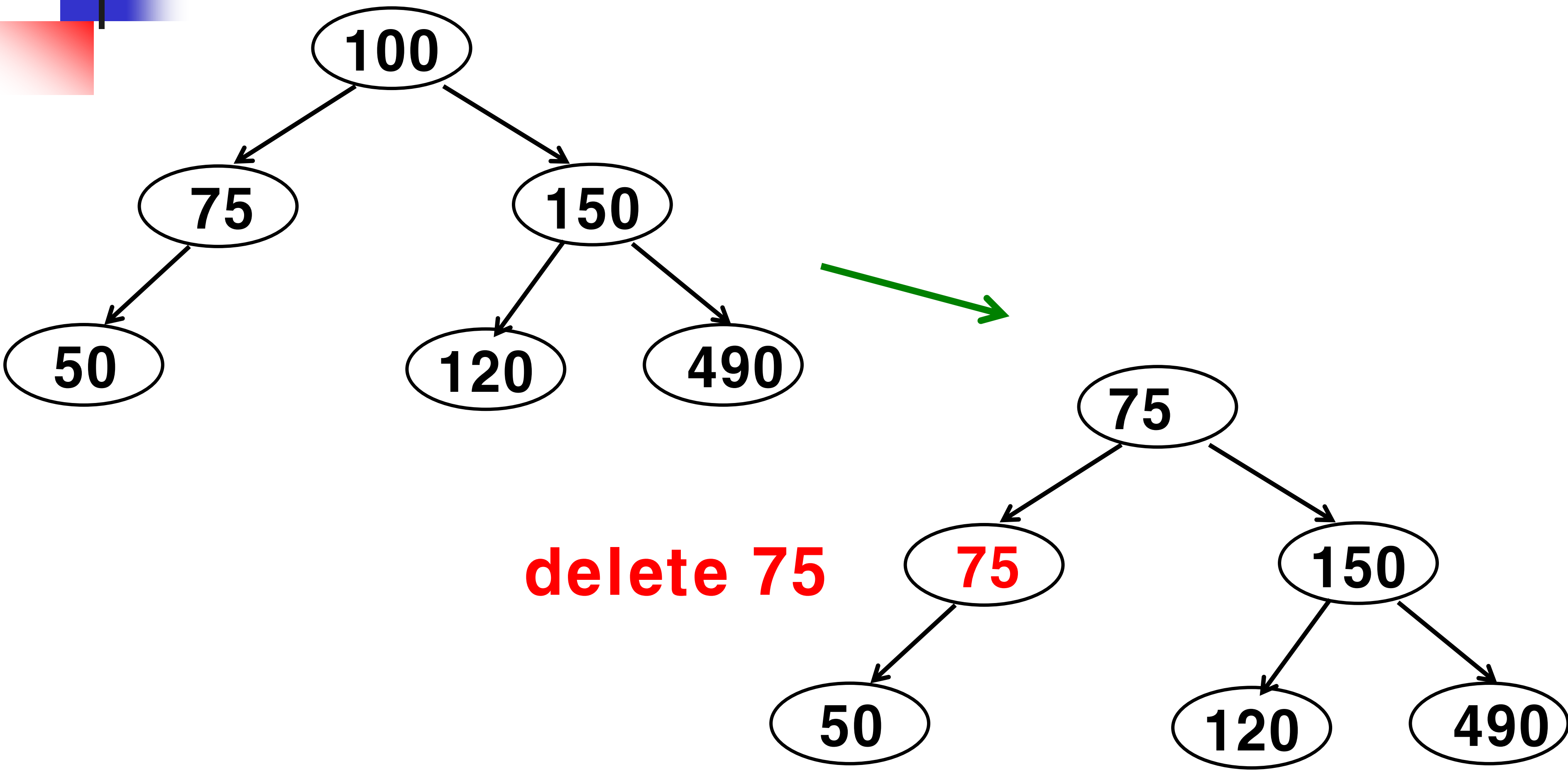
Exercise: Delete 300



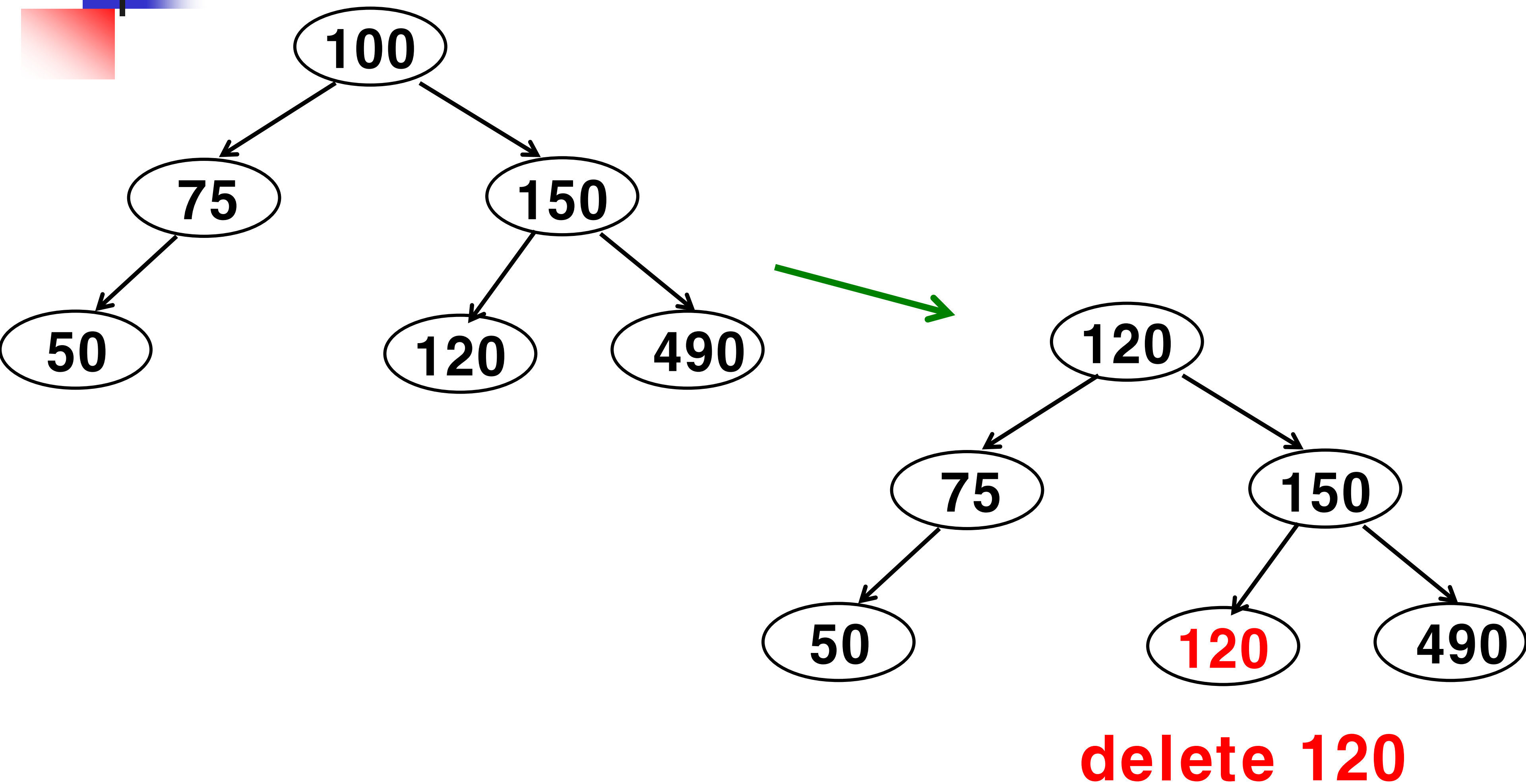
Exercise: Delete 100



Solution option 1: replace the key with the largest key on the left subtree



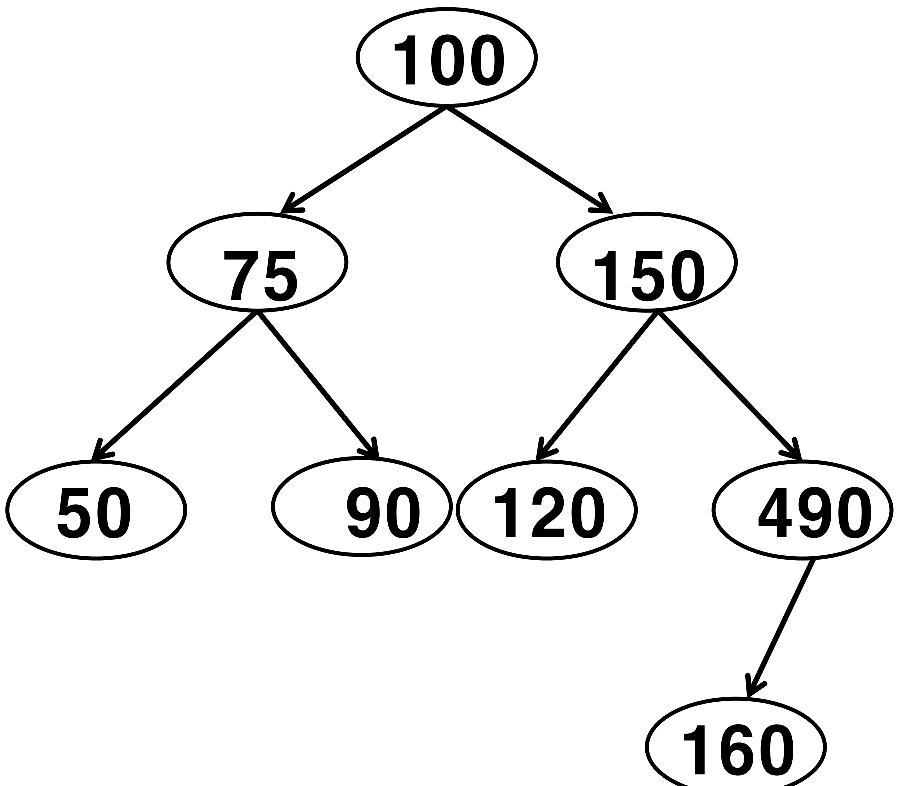
Solution option 2: replace the key with the smallest key on the right subtree



Assignment 5

HW 5-1: Implementing BST Search

- Implement BST Search function
  - Use the data in page 17.
  - Fill in the blank in the following pages.



1. search 120

2. search 400

HW 5-1: Implementing BST Search

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

struct NODE
{
int  key;
struct NODE* parent = NULL;
struct NODE* left = NULL;
struct NODE* right = NULL;
};

struct NODE* getNewNode(int val)
{
struct NODE* newNode = (struct NODE*)malloc(sizeof(str
uct NODE));
newNode->key = val;
newNode->parent = NULL;
newNode->left = NULL;
newNode->right = NULL;

return newNode;
}

struct TREE
{
struct NODE* root = NULL;
};
```

HW 5-1: Implementing BST Search

```
void set_left_child(NODE* parent, NODE* child)
{
child->parent = parent;
parent->left = child;
}
void set_right_child(NODE* parent, NODE* child)
{
child->parent = parent;
parent->right = child;
}

void Preorder_Traversal(struct NODE* node)
{
if (node)
{
printf("[%d] \n", node->key);
Preorder_Traversal(node->left);
Preorder_Traversal(node->right);
}
}

void print_tree(struct TREE* tree)
{
printf("--Print tree in preorder: \n");
Preorder_Traversal(tree->root);
printf("\n");
}
```

HW 5-1: Implementing BST Search

```
// --- Similar to Preorder Traversal !
bool search_key(const int key, struct NODE* node, int level)
{
// [Implement your code here] !!
}

int main()
{
// --- Constructing the (Ordered) Binary Search Tree in the slide.
TREE Tree;
Tree.root = getNewNode(100);
set_left_child(Tree.root, getNewNode(75));
set_right_child(Tree.root, getNewNode(150));

set_left_child(Tree.root->left, getNewNode(50));
set_right_child(Tree.root->left, getNewNode(90));

set_left_child(Tree.root->right, getNewNode(120));
set_right_child(Tree.root->right, getNewNode(490));

set_left_child(Tree.root->right->right, getNewNode(160));

print_tree(&Tree); // print tree structure

search_key(120, Tree.root, 1);
search_key(400, Tree.root, 1);

return 0;
}
```

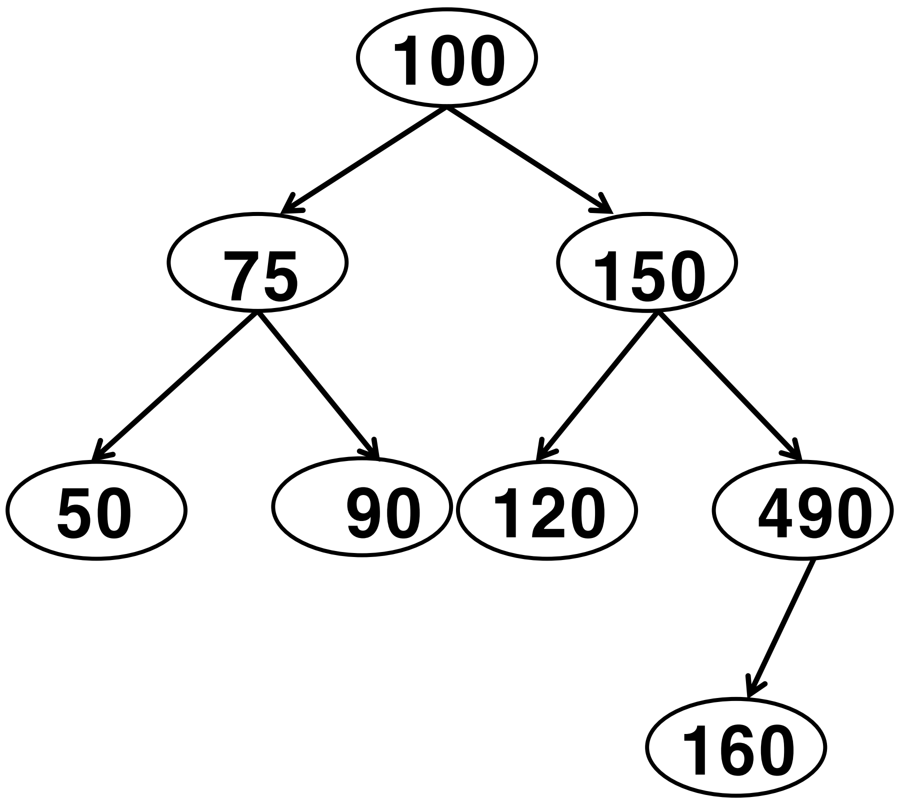
HW 5-1: Implementing BST Search

- You should get similar results as below:

```
C:\Windows\system32\cmd.exe
--Print tree in preorder:
[100]
[75]
[50]
[90]
[150]
[120]
[490]
[160]


//--- Search for the key(120) in the tree...
L(1): The key(120) > node(100) --> right node
L(2): The key(120) < node(150) --> left node
L(3): The key(120) is found in the tree!

//--- Search for the key(400) in the tree...
L(1): The key(400) > node(100) --> right node
L(2): The key(400) > node(150) --> right node
L(3): The key(400) < node(490) --> left node
L(4): The key(400) > node(160) --> right node
L(4): The key(400) does not exist in the tree!
Press any key to continue . . .
```

HW 5-2: Implementing BST Insertion

- Implement BST Insertion function
  - Use the data in page 21.
  - Fill in the blanks in the following pages.



1. insert 80

HW 5-2: Implementing BST Insertion

```
struct NODE* find_insert_loc(const int key, struct NODE* node)
{
// [Implement your code here] !!
}

void insert_key(const int key, struct TREE* tree)
{
if (search_key(key, tree->root, 1))
{
printf("(Insert Failed): The key(%d) already exists..\n", key);
return;
}

struct NODE* loc = find_insert_loc(key, tree->root);

// [Implement your code here] !!
}
```

HW 5-2: Implementing BST Insertion

```
int main()
{
// --- Constructing the (Ordered) Binary Search Tree in the slide.
TREE Tree;
insert_key(100, &Tree);
insert_key(75, &Tree);
insert_key(150, &Tree);
insert_key(50, &Tree);
insert_key(90, &Tree);
insert_key(120, &Tree);
insert_key(490, &Tree);
insert_key(160, &Tree);

print_tree(&Tree); // print tree structure

// you should get the same results with the previous version
search_key(120, Tree.root, 1);
search_key(400, Tree.root, 1);

// check the insertion result.
insert_key(80, &Tree);
print_tree(&Tree);

return 0;
}
```

HW 5-2: Implementing BST Insertion

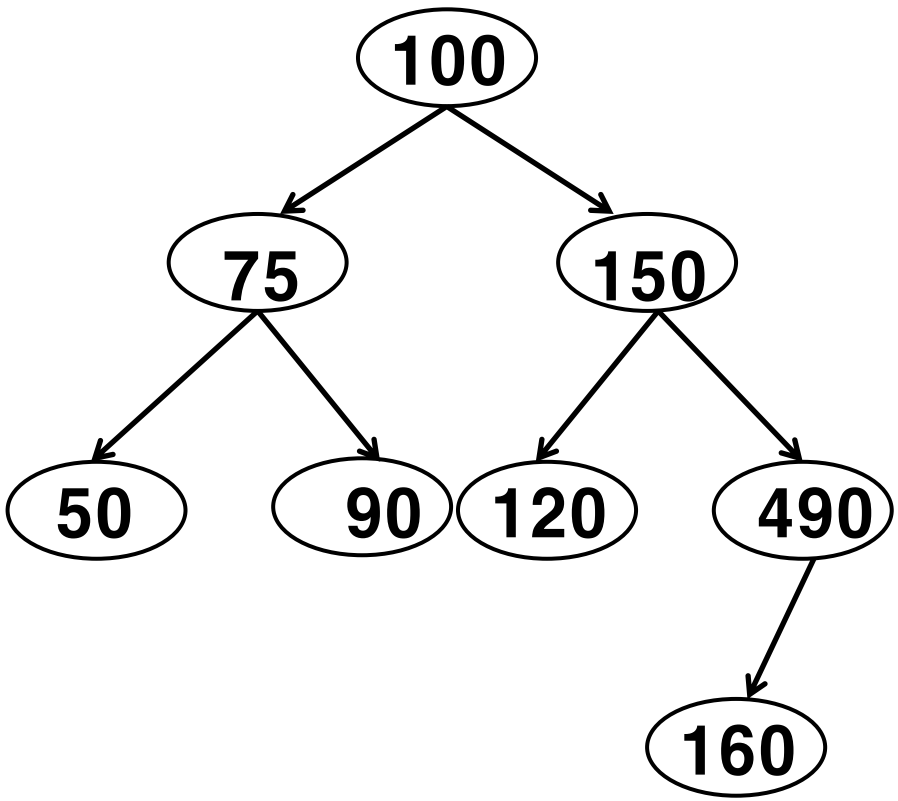
- You should get similar results as below:

```
//--- Search for the key(100) in the tree...
L(0): The key(100) does not exist in the tree!

//--- Search for the key(75) in the tree...
L(1): The key(75) < node(100) --> left node
L(1): The key(75) does not exist in the tree!
-- The key(75) is inserted as the [Left] child of node(100)

//--- Search for the key(150) in the tree...
L(1): The key(150) > node(100) --> right node
L(1): The key(150) does not exist in the tree!
-- The key(150) is inserted as the [Right] child of node(100)

//--- Search for the key(50) in the tree...
L(1): The key(50) < node(100) --> left node
L(2): The key(50) < node(75) --> left node
L(2): The key(50) does not exist in the tree!
-- The key(50) is inserted as the [Left] child of node(75)

//--- Search for the key(90) in the tree...
L(1): The key(90) < node(100) --> left node
L(2): The key(90) > node(75) --> right node
L(2): The key(90) does not exist in the tree!
-- The key(90) is inserted as the [Right] child of node(75)

//--- Search for the key(120) in the tree...
L(1): The key(120) > node(100) --> right node
L(2): The key(120) < node(150) --> left node
L(2): The key(120) does not exist in the tree!
-- The key(120) is inserted as the [Left] child of node(150)

//--- Search for the key(490) in the tree...
L(1): The key(490) > node(100) --> right node
L(2): The key(490) > node(150) --> right node
L(2): The key(490) does not exist in the tree!
-- The key(490) is inserted as the [Right] child of node(150)

//--- Search for the key(160) in the tree...
L(1): The key(160) > node(100) --> right node
L(2): The key(160) > node(150) --> right node
L(3): The key(160) < node(490) --> left node
L(3): The key(160) does not exist in the tree!
-- The key(160) is inserted as the [Left] child of node(490)
--Print tree in preorder:
[100]
[75]
[50]
[90]
[150]
[120]
[490]
[160]
```

```
//--- Search for the key(120) in the tree...
L(1): The key(120) > node(100) --> right node
L(2): The key(120) < node(150) --> left node
L(3): The key(120) is found in the tree!

//--- Search for the key(400) in the tree...
L(1): The key(400) > node(100) --> right node
L(2): The key(400) > node(150) --> right node
L(3): The key(400) < node(490) --> left node
L(4): The key(400) > node(160) --> right node
L(4): The key(400) does not exist in the tree!

//--- Search for the key(80) in the tree...
L(1): The key(80) < node(100) --> left node
L(2): The key(80) > node(75) --> right node
L(3): The key(80) < node(90) --> left node
L(3): The key(80) does not exist in the tree!
-- The key(80) is inserted as the [Left] child of node(90)
--Print tree in preorder:
[100]
[75]
[50]
[90]
[80]
[150]
[120]
[490]
[160]

Press any key to continue . . .
```

HW 5-3: Implementing BST Deletion

- Implement BST Deletion function
  - Use the data in pp. 24-31.
  - Fill in the blanks in the following pages.



1. delete 50
2. delete 75
3. delete 150

HW 5-3: Implementing BST Deletion

```
// You should call this function only if search_key() == true.
struct NODE* find_delete_node(const int key, struct NODE* node)
{
// node == NULL Never Happens!

if (key == node->key)
{
return node;
}
else if (key > node->key)
{
return find_delete_node(key, node->right);
}
else
{
return find_delete_node(key, node->left);
}
}
```

HW 5-3: Implementing BST Deletion

```
int num_child(struct NODE* node)
{
int count = 0;

if (node->left)
count++;

if (node->right)
count++;

return count;
}

struct NODE* find_smallest_node(struct NODE* node)
{
if (node->left == NULL)
return node;
else
return find_smallest_node(node->left);
}
```

HW 5-3: Implementing BST Deletion

```
void delete_key(const int key, struct TREE* tree)
{
if (!search_key(key, tree->root, 1))
{
printf("(Delete Failed): The key(%d) does not exist..\n", key);
return;
}

struct NODE* loc = find_delete_node(key, tree->root);
int num = num_child(loc);
NODE* one_child = NULL;

switch (num)
{
case 0:
// [Implement your code here] !!
break;

case 1:
// [Implement your code here] !!
break;

case 2:
// [Implement your code here] !!
break;

default:
break;
}
free(loc);
}
```

HW 5-3: Implementing BST Deletion

```
int main()
{
// --- Constructing the (Ordered) Binary Search Tree in the slide.
TREE Tree;
insert_key(100, &Tree);
insert_key(75, &Tree);
insert_key(150, &Tree);
insert_key(50, &Tree);
insert_key(90, &Tree);
insert_key(120, &Tree);
insert_key(490, &Tree);
insert_key(160, &Tree);

print_tree(&Tree); // print tree structure

// check the deletion results.
delete_key(50, &Tree); // leaf node
print_tree(&Tree);

delete_key(75, &Tree); // interior node with one child node
print_tree(&Tree);

delete_key(150, &Tree); // interior node with two child nodes
// Use option  2: promote the smallest node on the right subtree
print_tree(&Tree);
return 0;
}
```

HW 5-3: Implementing BST Deletion

- You should get similar results as below:

```
C:\Windows\system32\cmd.exe
-- The key(490) is inserted as the [Right] child of node(150)

//--- Search for the key(160) in the tree...
L(1): The key(160) > node(100) --> right node
L(2): The key(160) > node(150) --> right node
L(3): The key(160) < node(490) --> left node
L(3): The key(160) does not exist in the tree!
-- The key(160) is inserted as the [Left] child of node(490)
--Print tree in preorder:
[100]
[75]
[50]
[90]
[150]
[120]
[490]
[160]


//--- Search for the key(50) in the tree...
L(1): The key(50) < node(100) --> left node
L(2): The key(50) < node(75) --> left node
L(3): The key(50) is found in the tree!
--Print tree in preorder:
[100]
[75]
[90]
[150]
[120]
[490]
[160]


//--- Search for the key(75) in the tree...
L(1): The key(75) < node(100) --> left node
L(2): The key(75) is found in the tree!
--Print tree in preorder:
[100]
[90]
[150]
[120]
[490]
[160]


//--- Search for the key(150) in the tree...
L(1): The key(150) > node(100) --> right node
L(2): The key(150) is found in the tree!
--Print tree in preorder:
[100]
[90]
[160]
[120]
[490]

Press any key to continue . . .
```

End of Lecture