TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Fundamentals

Chapter 2. Algorithmic paradigms

Chapter 3. Basic data structures

Chapter 4. Tree

Chapter 5. Sorting

Chapter 6. Searching

Chapter 7. Graph

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 6. Searching

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

Searching problem

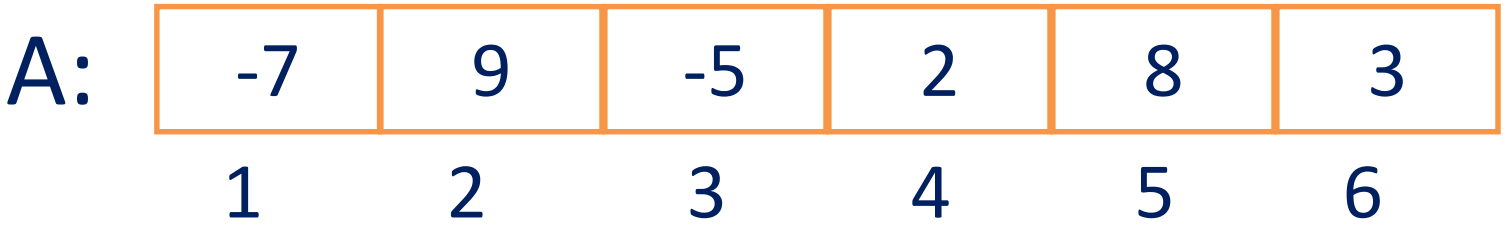
- Given a list A consists of n elements $a_1, a_2, .., a_n$ and a target number x.

Question: whether the target x is in the list ?

- If the target x is in the list, give the index of x (its location) in the list that means find the index i where $a_i = x$

6.1. Linear Search

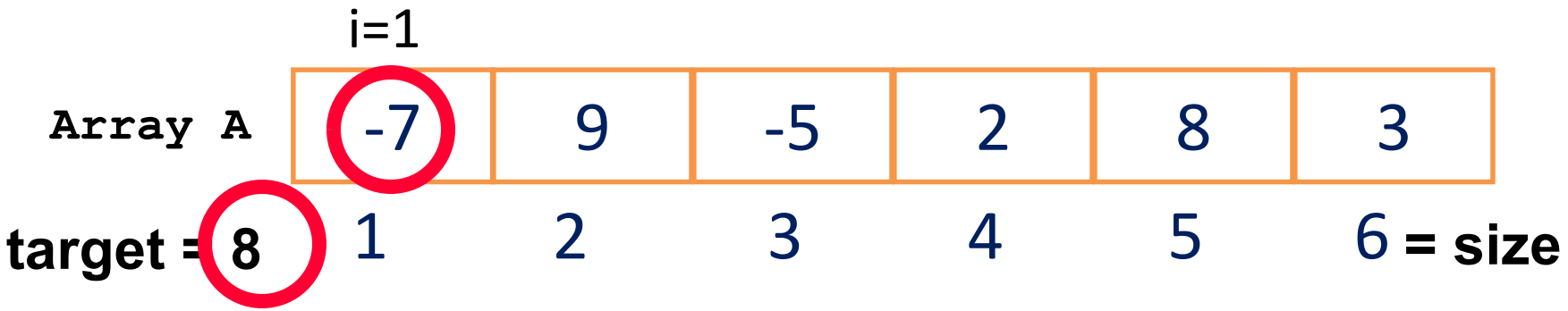
- Input:
  - the list A of n elements and the target item x.
  - The list A does not need to be sorted.
- Algorithm: Search *starts* at the *first element* in the list and *continues* until either the target *item x is found in the list - or entire list is searched*
- Complexity: $O(n)$



Target $x = 8$:

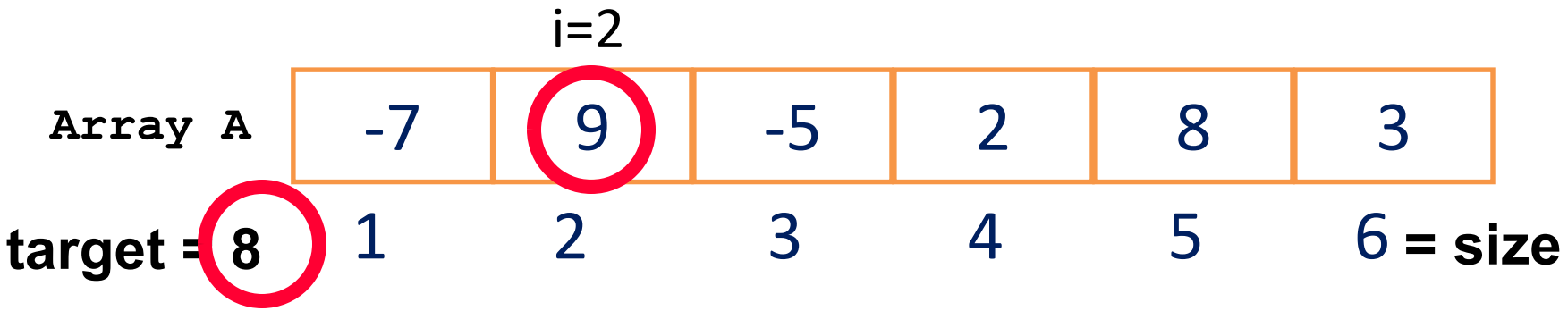
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



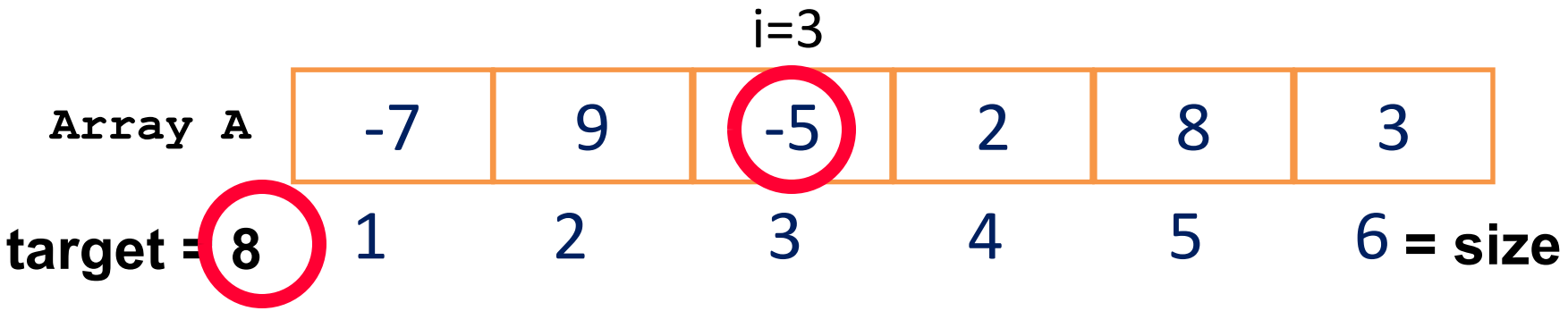
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



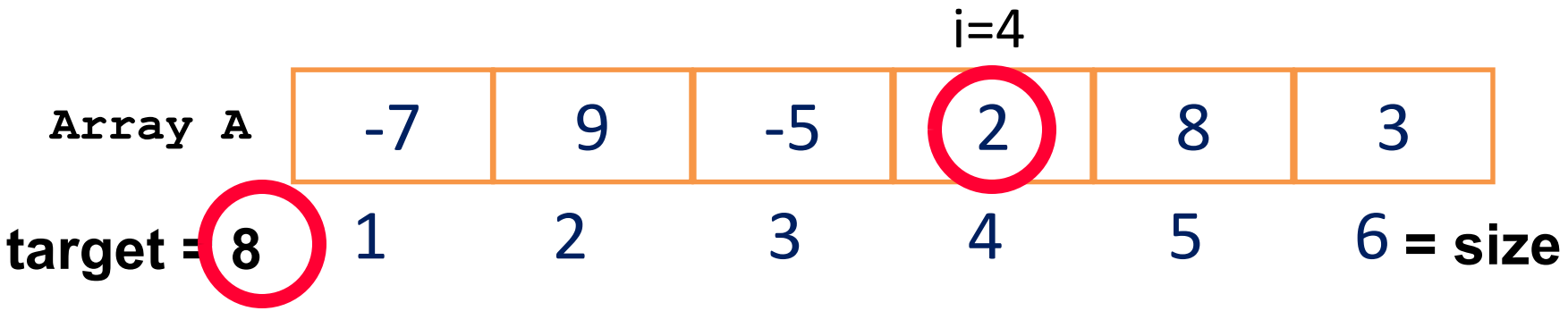
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



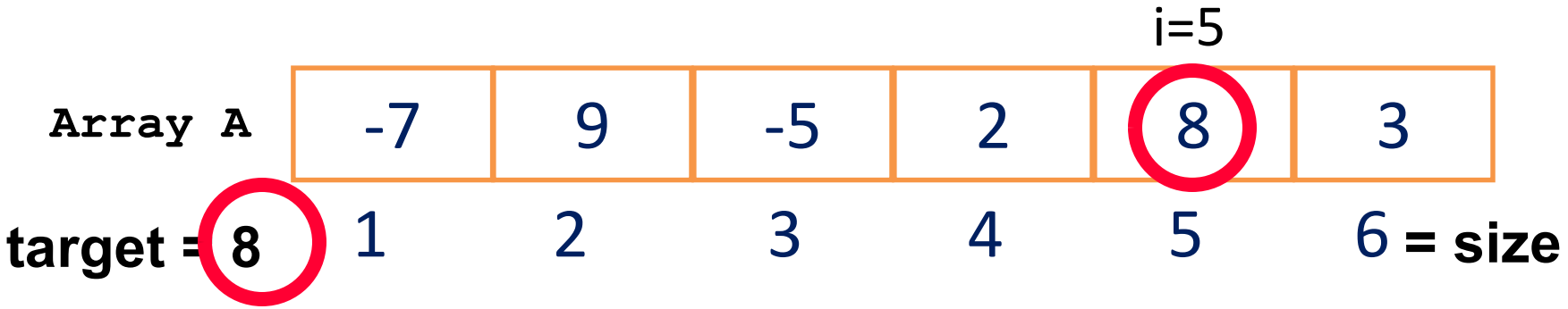
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



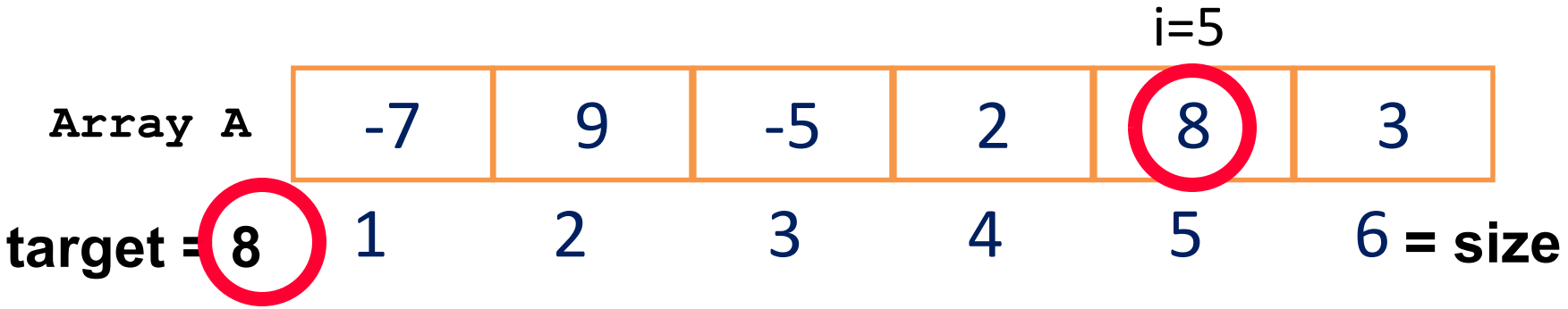
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
      printf("The target is in the list @ index = %d",i);
    else
      printf("The target is NOT in the list");
}
```



6.1. Linear Search

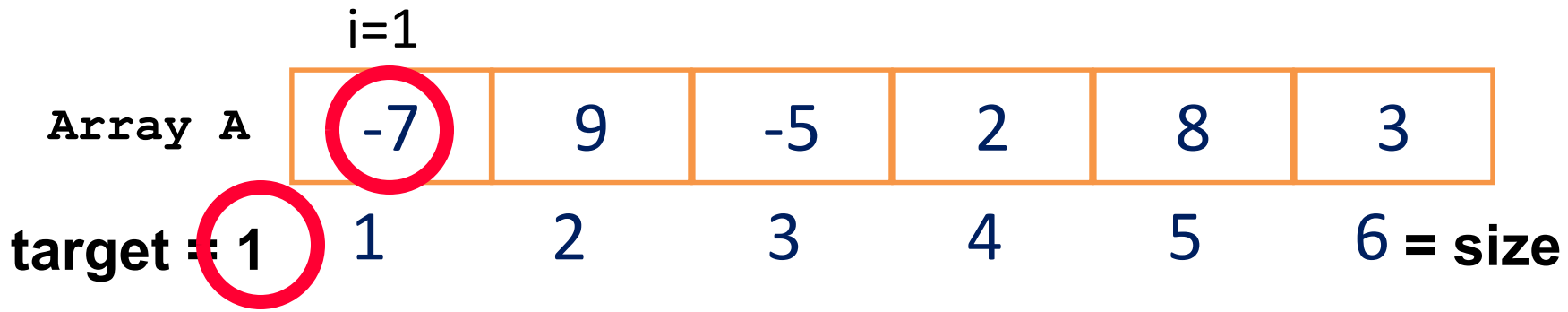
```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
      printf("The target is in the list @ index = %d",i);
    else
      printf("The target is NOT in the list");
}
```



The target is in the list @ index = 5

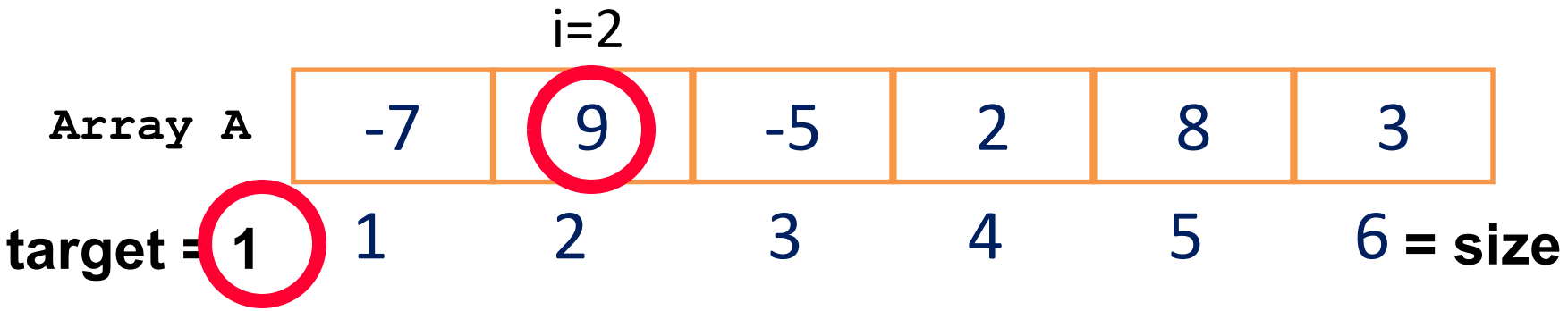
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



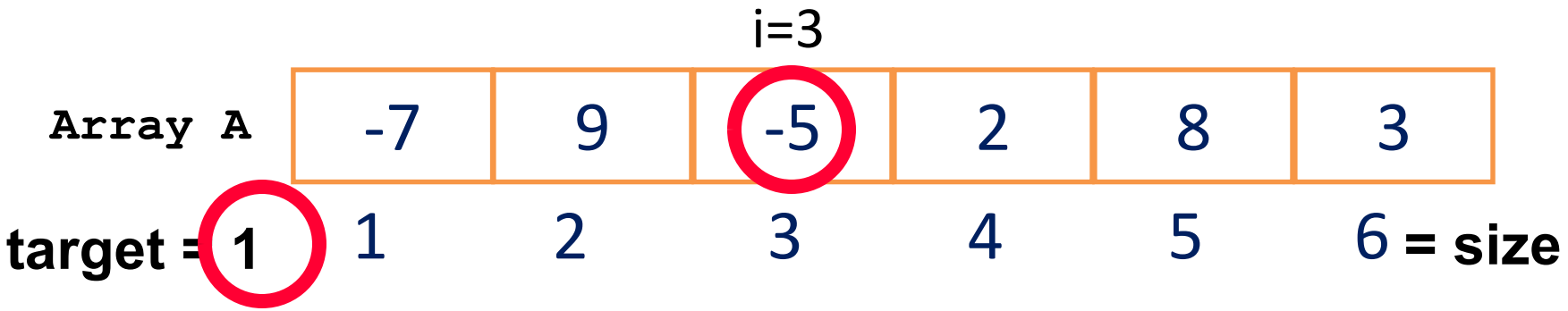
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



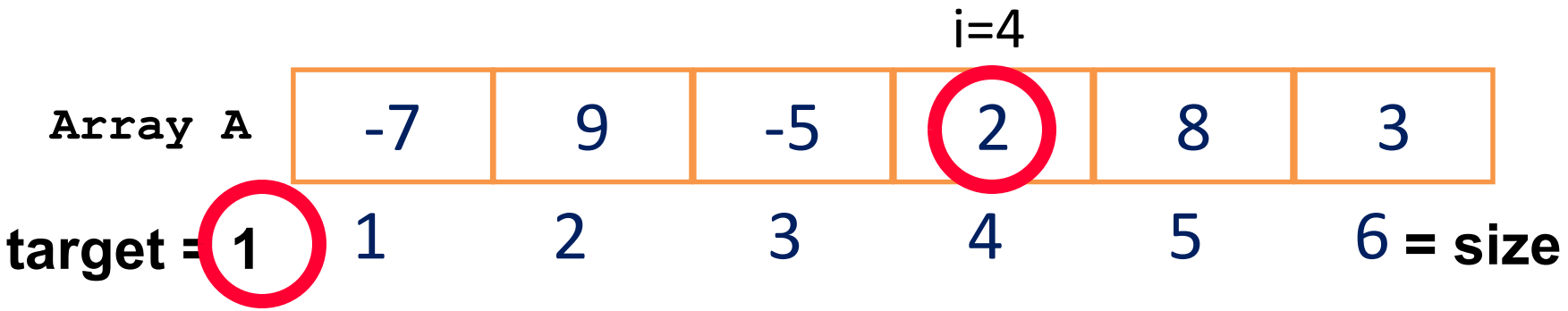
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



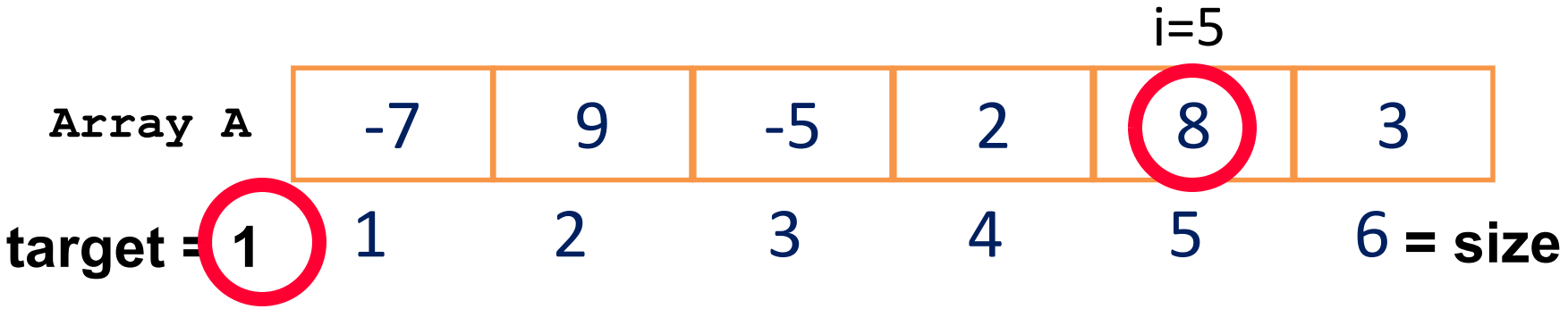
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



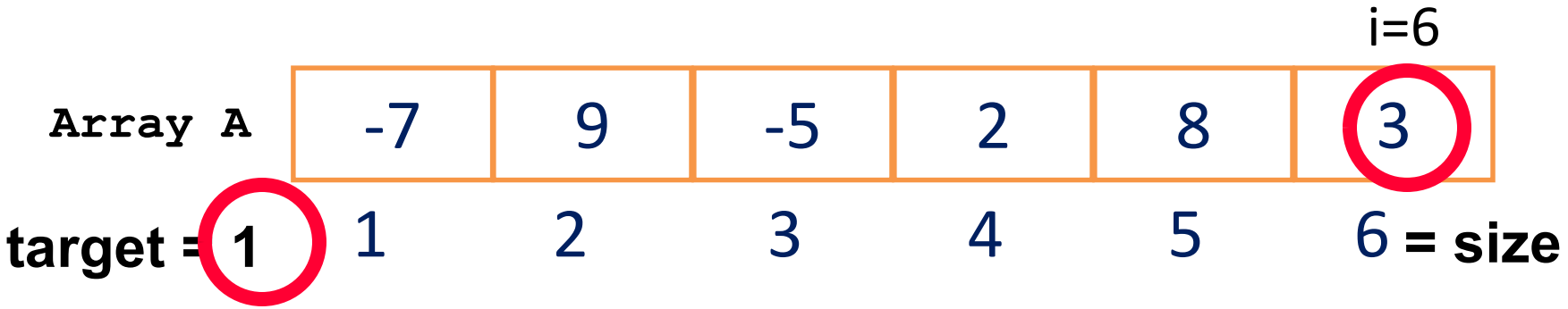
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{  int i;
   for (i = 1; i <= size; i++)
      if (a[i] == target) break;
   if (i <= size)
     printf("The target is in the list @ index = %d",i);
   else
     printf("The target is NOT in the list");
}
```



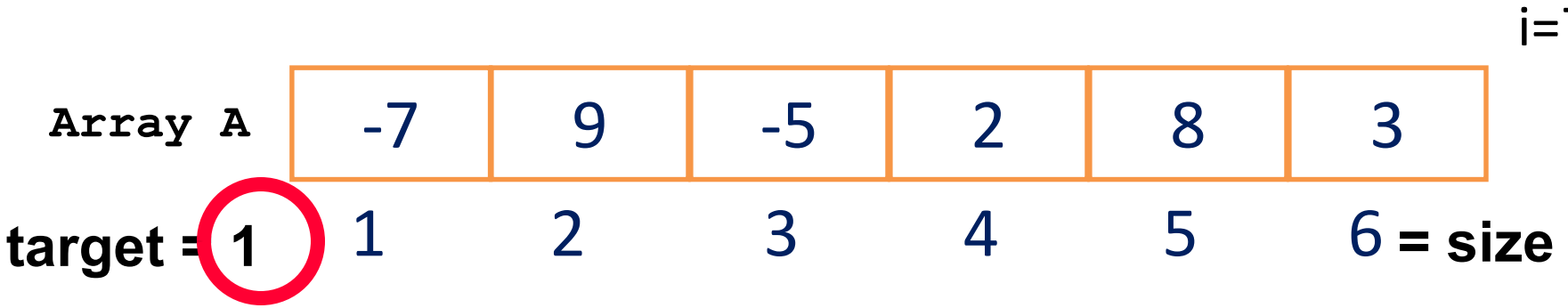
6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
      printf("The target is in the list @ index = %d",i);
    else
      printf("The target is NOT in the list");
}
```



6.1. Linear Search

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
      printf("The target is in the list @ index = %d",i);
    else
      printf("The target is NOT in the list");
}
```



The target is NOT in the list

Contents

6.2. Binary Search

- Input: An array *A* consists of *n* elements: A[0],…,A[*n*-1] in ascending order; Value ***target*** with the same data type as array *A*.
- Output: the index in array if ***target*** is found, -1 if ***target*** is not found
- Algorithm: looks at the middle element of the array
  - If the target < A[middle]:
    - The right half of the array can be ignored
    - This strategy is then applied to the left half of the array
  - Else If the target > A[middle]:
    - The left half of the array can be ignored
    - This strategy is then applied to the right half of the array
  - Else If the target == A[middle], then it has been found
  - If the entire array (or array segment) has been searched in this way without finding the target, then it is not in the array

6.2. Binary Search

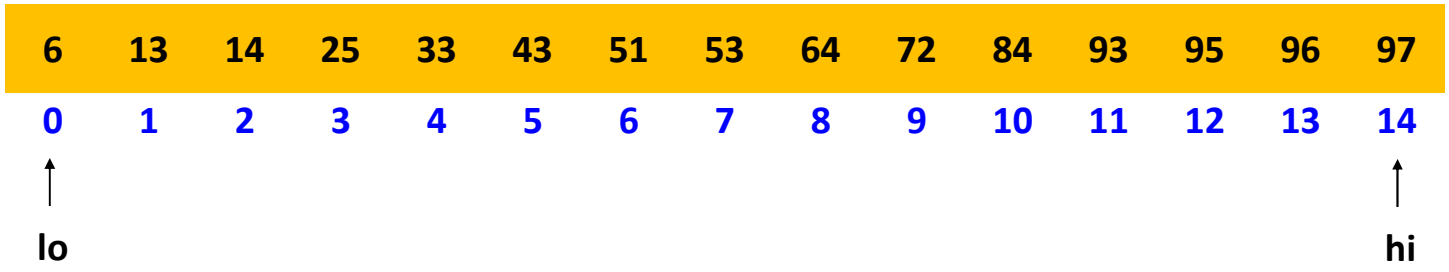
Input: An array *A* consists of *n* elements: A[0],…,A[*n*-1] in ascending order; Value ***target*** with the same data type as array *A*.

```
/*Search the array A for target. If target is not in the array,
then -1 is returned. Otherwise, returns an index such that
A[index]==target.
Precondition: A[low] ≤ A[low+1] ≤ …. ≤ A[last] */
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (A[mid]==target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
     }
     else return -1;
}
```

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, key);
       else
           return binsearch(mid+1, high, A, key);
    }
    else return -1;
}
```

target=33

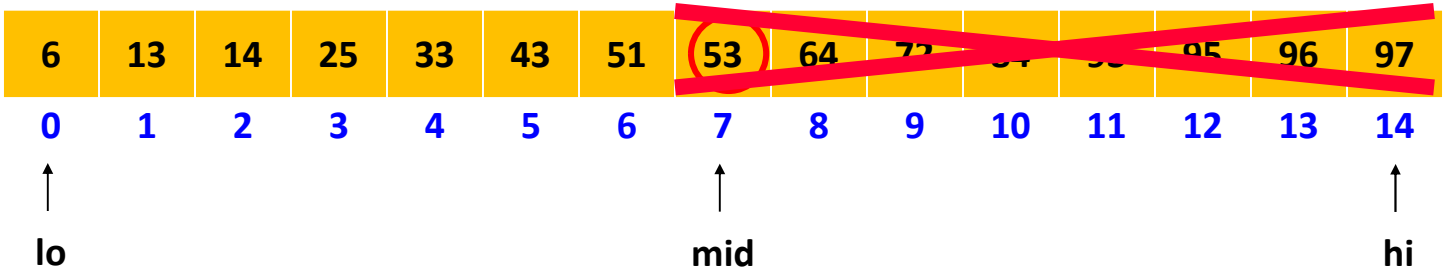


binsearch(0, 14, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



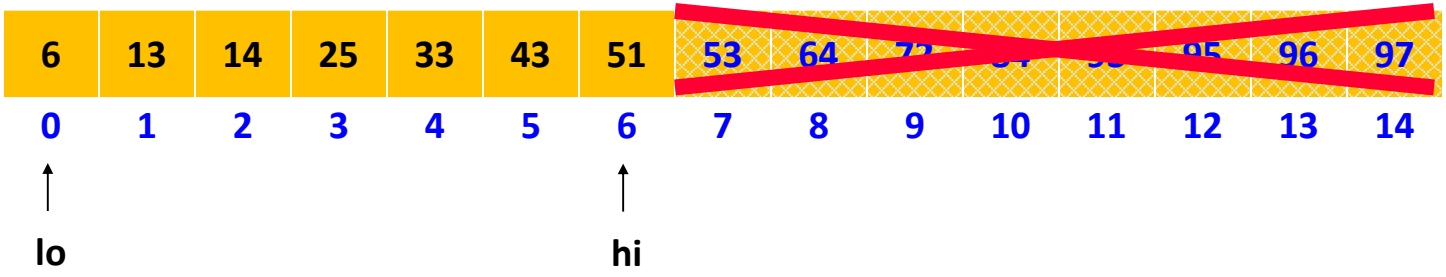
binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
     }
     else return -1;
}
```

target=33



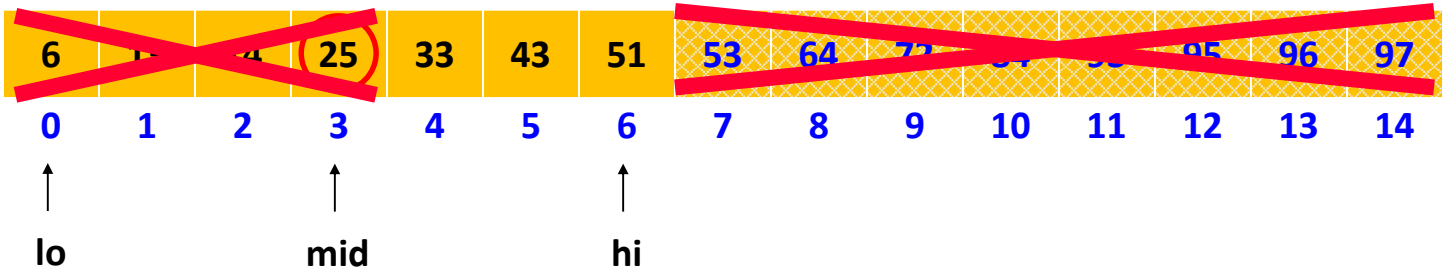
binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
        int mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

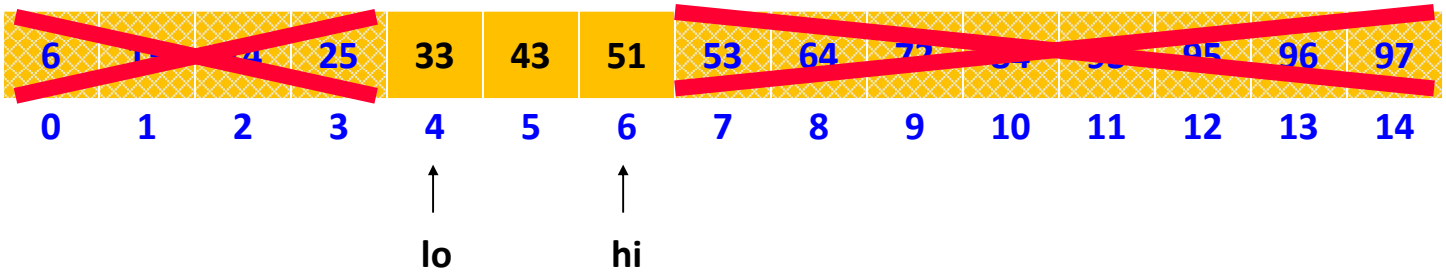
binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

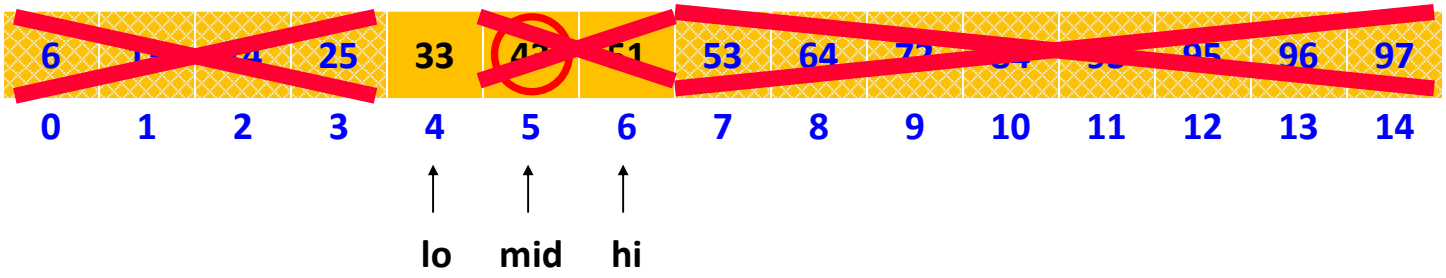
binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

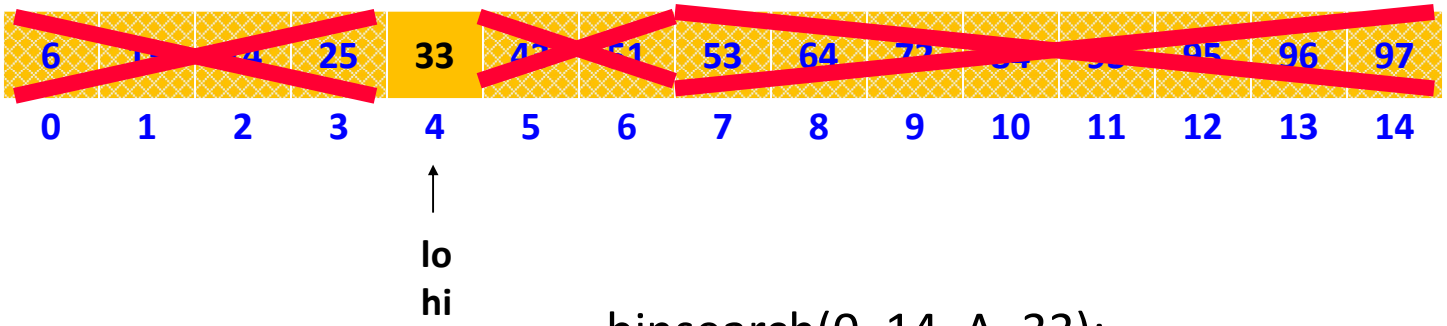
binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

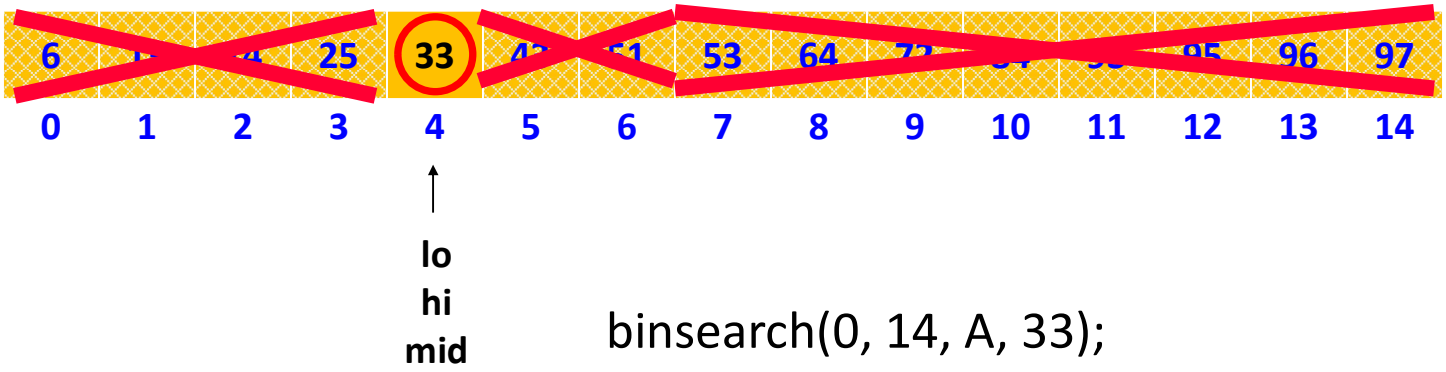
binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

6.2. Binary Search

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
     }
     else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

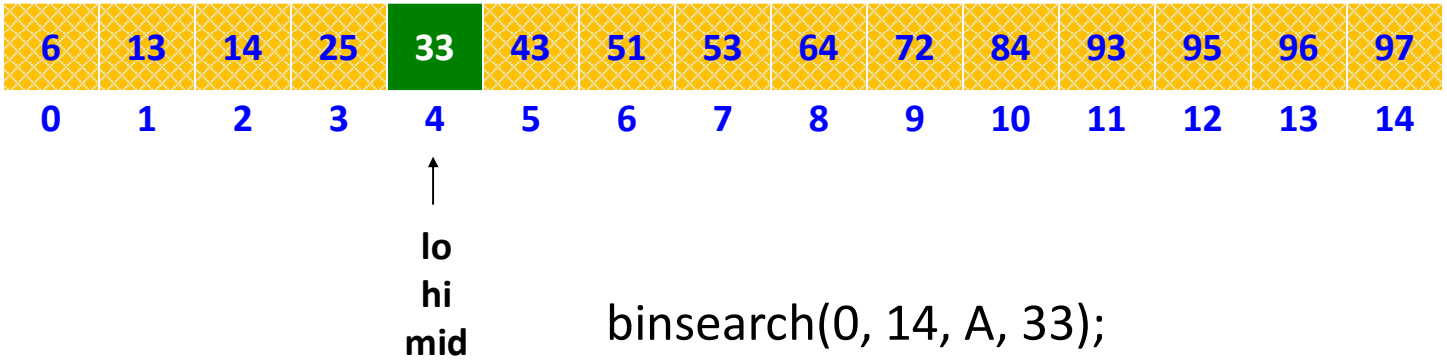
6.2. Binary Search

target=31??

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]== target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33

The target 33 is in the list @index=4



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

6.2. Binary Search

Input: An array *A* consists of *n* elements: A[0],…,A[*n*-1] in ascending order; Value ***target*** with the same data type as array *A*.

Output: the index in array if ***target*** is found, -1 if ***target*** is not found

```
int binsearch(int low, int high, int A[], int target)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (A[mid]==target) return mid;
       else if (target < A[mid])
           return binsearch(low, mid-1, A, target);
       else
           return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

➔ To find target in the array A, we call: binsearch(0, *n*-1, *A*, *target*);
Complexity: $T(n) = T(n/2) + C$ ➔ $T(n) = O(\log_2 n)$

Contents

6.3. Binary search tree

6.3.1. Definition

6.3.2. Representation of binary search tree

6.3.3. Operations on binary search tree

6.3. Binary search tree

6.3.1. Definition

6.3.2. Representation of binary search tree

6.3.3. Operations on binary search tree

Problem

We need to build a structure to represent *dynamic sets*

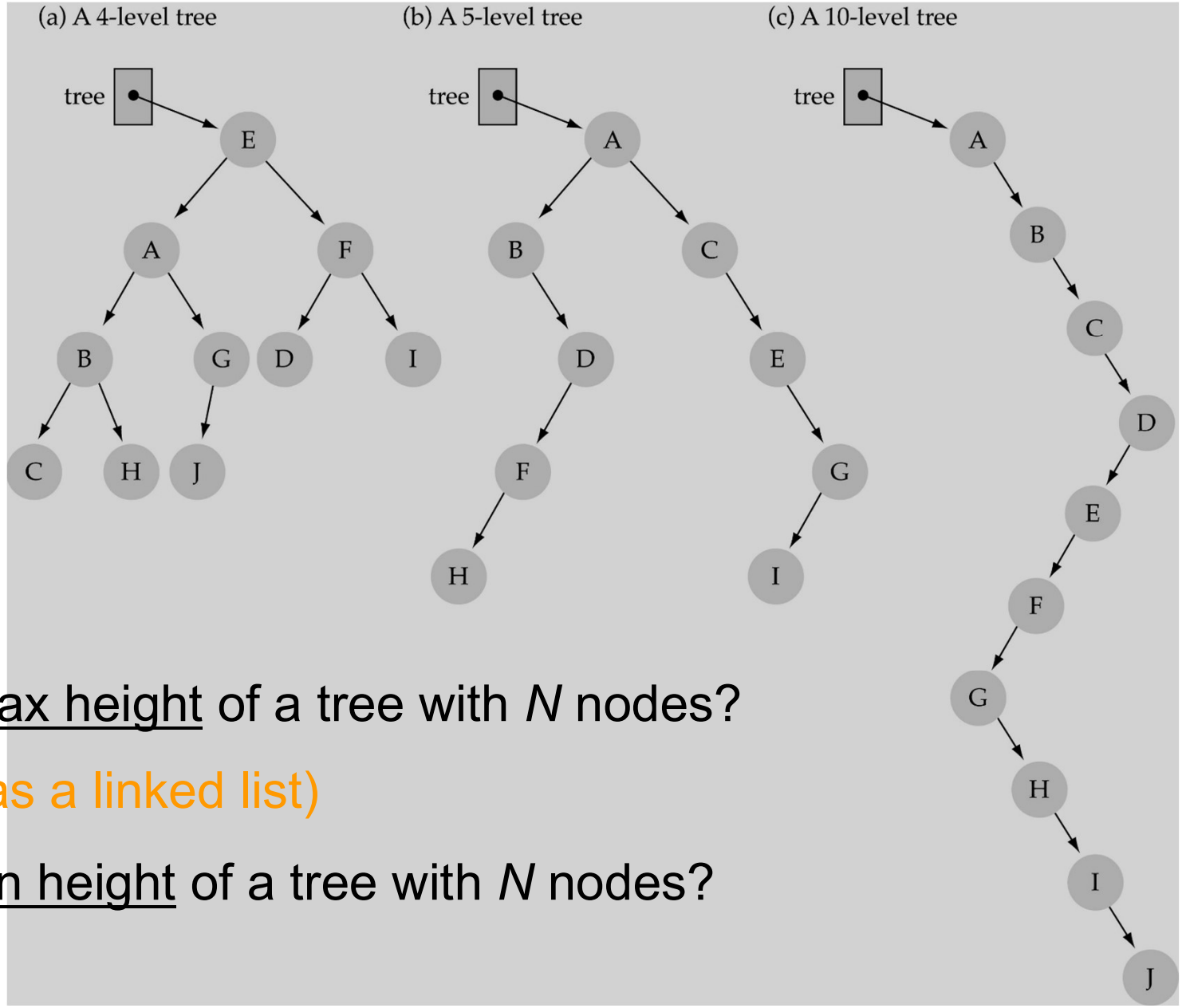
- Elements have *key* and associated information (*satellite data*)
- We need to do following queries on dynamic set S:
  - ***Search*(*S, k*)**: Find element with key *k* in the set *S*
  - ***Minimum*(*S*), *Maximum*(*S*)**: find element with smallest key, largest key in the set S
  - ***Predecessor*(*S, x*), *Successor*(*S, x*)**: find the preceding element (phần tử kế cận trước), succeeding element of *x* in the set *S*

And also do operations:

- ***Insert*(*S, x*)**: *insert x* into *S*
- ***Delete*(*S, x*)**: *delete x* from *S*

Binary search tree is a structure to represent dynamic set, in which all operations above could do with time complexity of O(h), where h is the height of tree.

Binary tree: N nodes



- What is the <u>max height</u> of a tree with *N* nodes?

  N (same as a linked list)

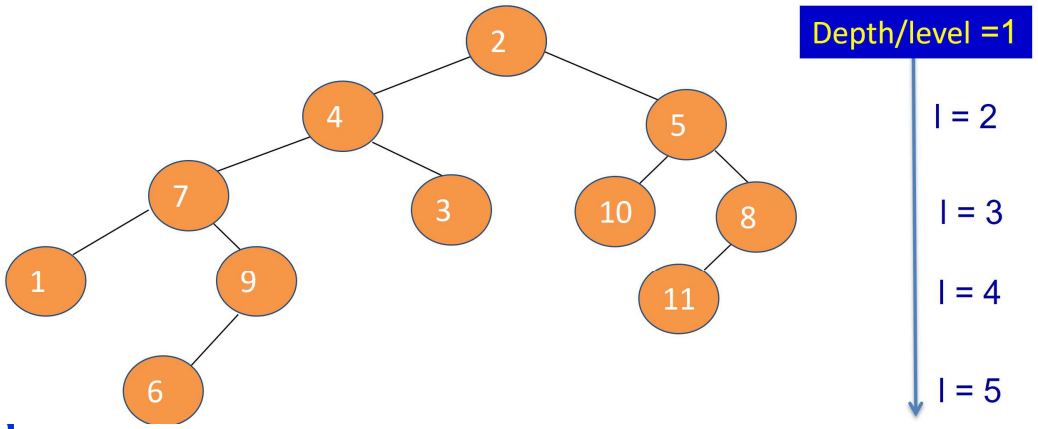
- What is the <u>min height</u> of a tree with *N* nodes?

  log(N+1)

How to search a value on a binary tree?

(1) Start at the root

(2) Search the tree level by level, until you find the element you are searching for or you reach a leaf.



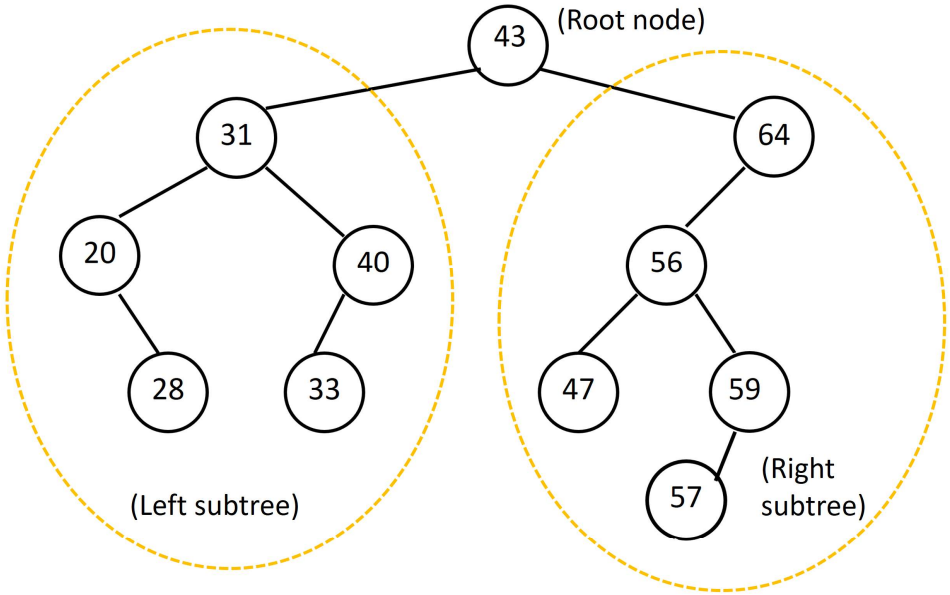
Is this better than searching a linked list?

No → O(N)
where N is the number of nodes

Binary **Search** Trees (BSTs)

- **Binary Search Tree Property**:

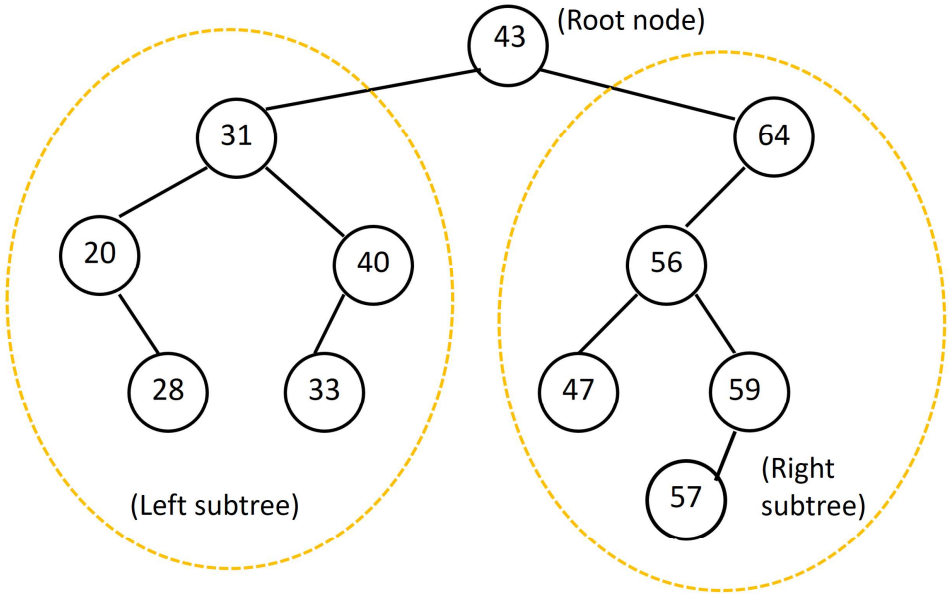
  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



All values on the left subtree are **less** than the value in the root node

All values on the right subtree are **greater** than the value in the root node

How to search a value on a binary search tree?



All values on the left subtree are **less** than the value in the root node

All values on the right subtree are **greater** than the value in the root node

Is this better than searching a linked list?

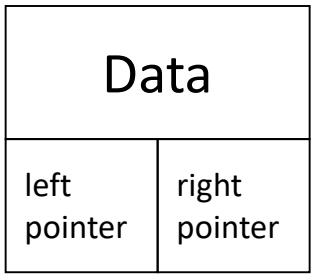
Yes !! ---> O(logN)

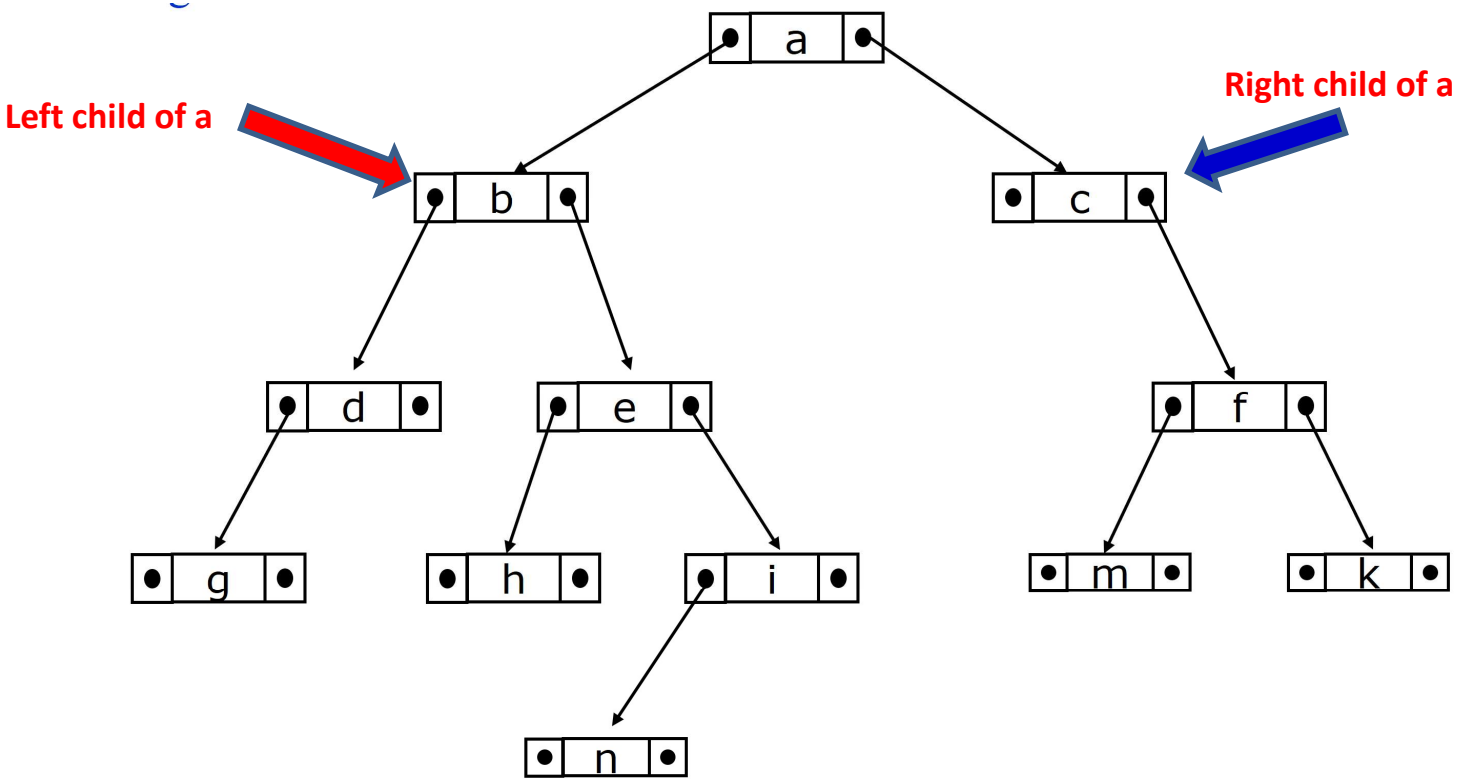
Binary tree

- A binary tree is a tree in which no node can have more than two children.

➔ Each node could either: (1) has no child, (2) has only left child, (3) has only right child, (4) has both left child and right child

- Each node has the data, a reference to a left child and a reference to a right child.

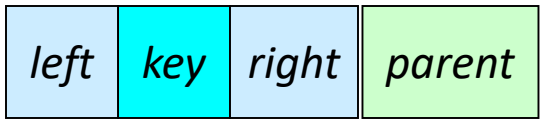




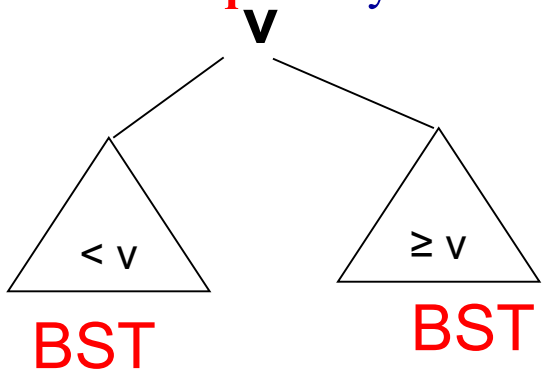
Binary search tree (cây nhị phân tìm kiếm)

Binary search tree (denote BST) has following properties:

- **Each node *x* (besides other information) has fields:**
  - *left* : pointer points to left child,
  - *right*: pointer points to right child,

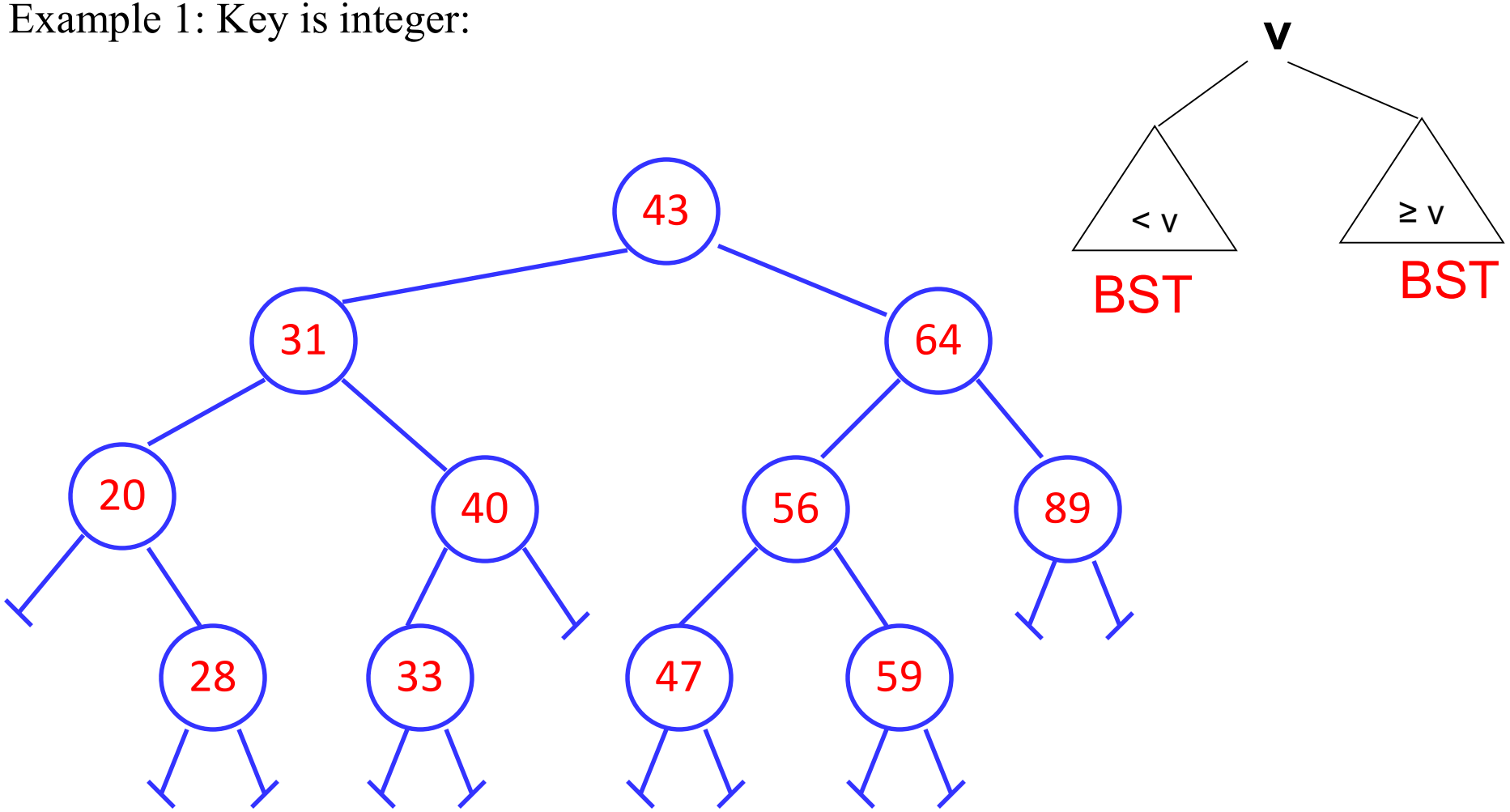


  - *parent*: pointer points to parent (this field is optional), and
  - *key*: **key (**it is often assumed that the keys of the nodes are different for each pair, whereas if there are identical keys, it is necessary to specify the order of the two identical keys).
- Each node has **a unique key**
- **All keys** in the **left subtree** of the node are **less than** key of the node
- **All keys** in the **right subtree** of the node are **greater than or equal** key of the node



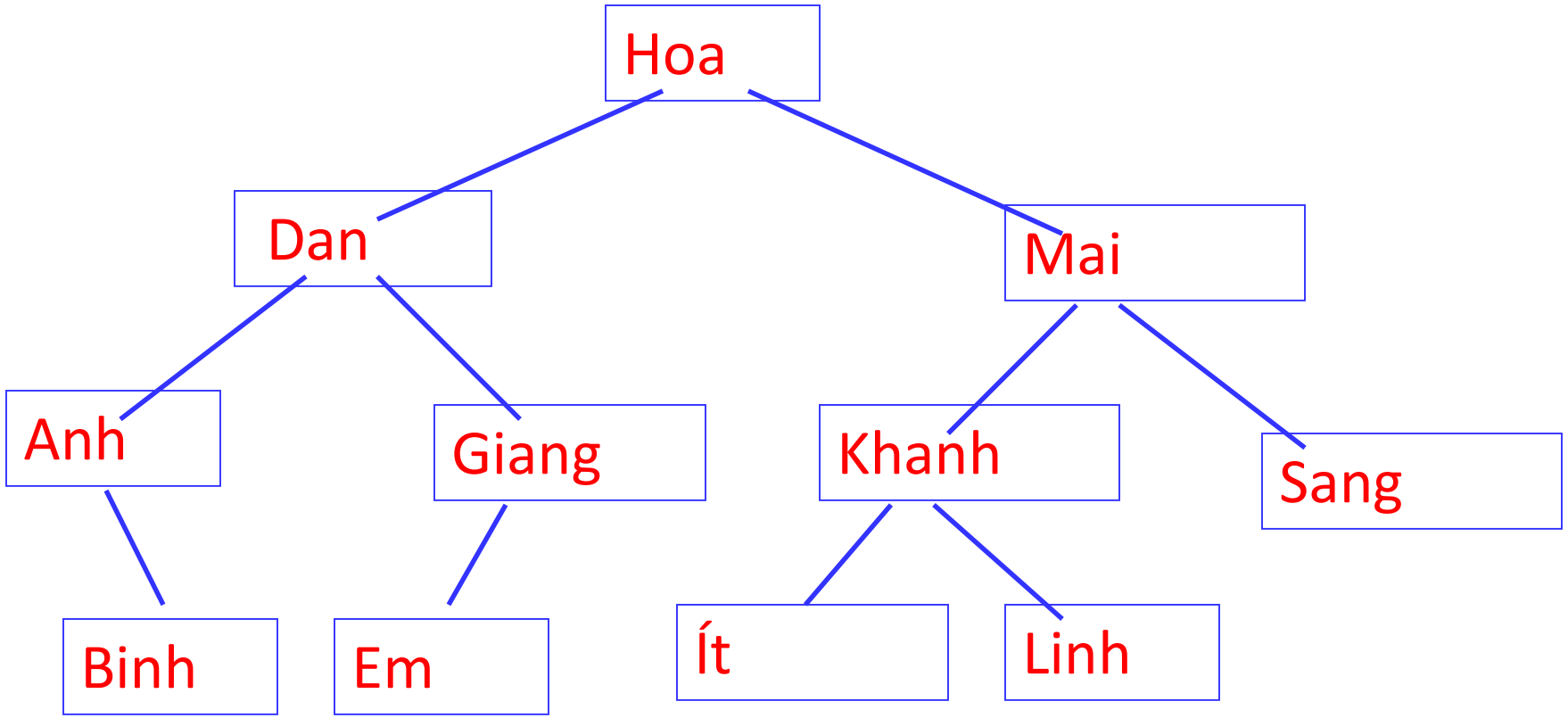
Binary search tree

Example 1: Key is integer:



Binary search tree

Example 2: Binary search tree with key is string of characters



6.3. Binary search tree

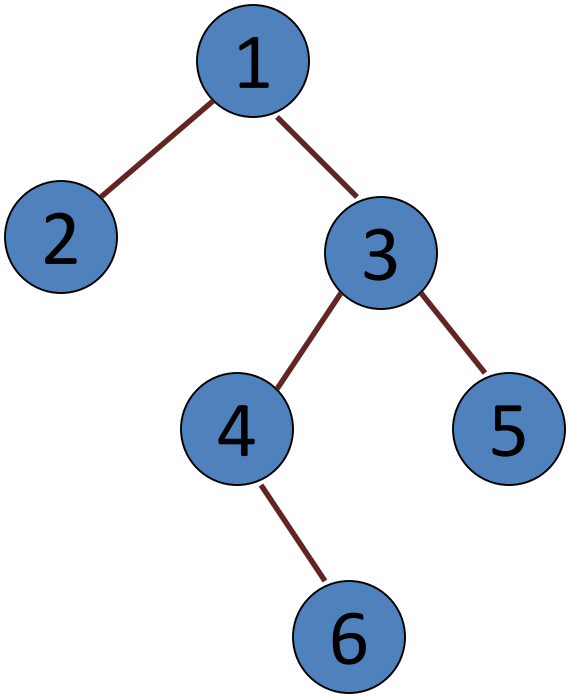
6.3.1. Definition

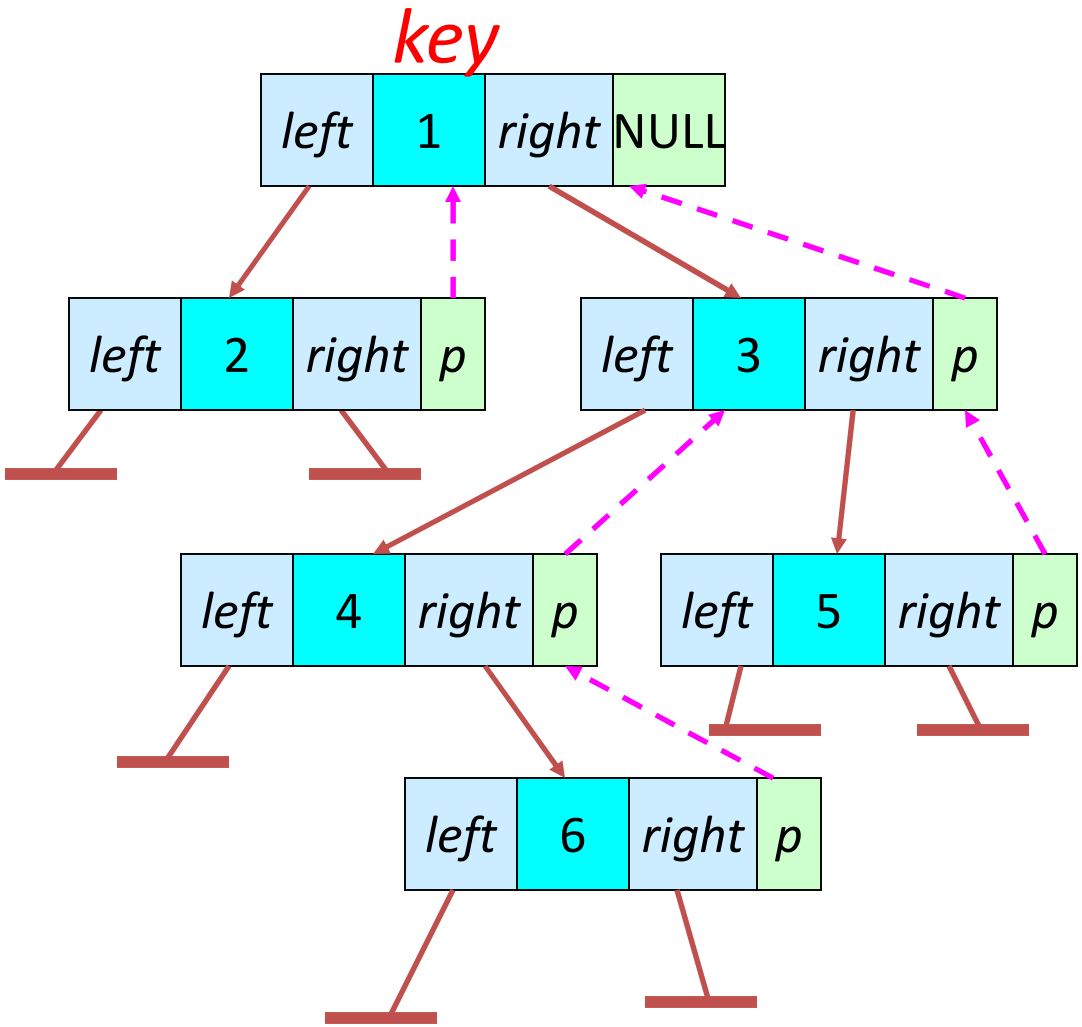
6.3.2. Representation of binary search tree

6.3.3. Operations on binary search tree

6.3.2. Representation of BST

Using Binary Tree Structure





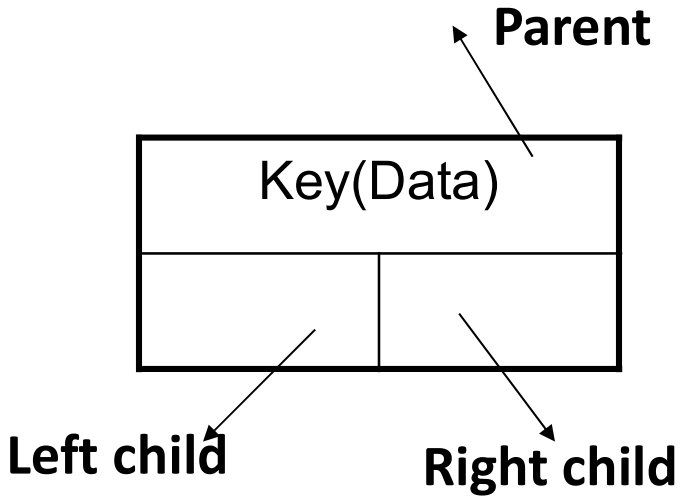
6.3.2. Representation of BST

Represent the links:

- Each node of tree is represented as an object of the same type.
- Memory space used to represent *n* nodes of BST
  = *n* * (memory space of one node)

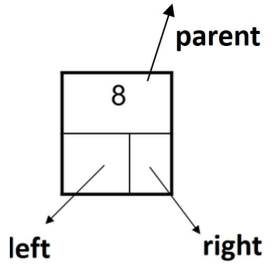
```
struct elementType {….};

struct TreeNode {
    elementType data;
    TreeNode *left, *right, *parent;
};
```



Example 1: if key (data) of node is integer

```
struct TreeNode {
    int data;
    TreeNode *left, *right, *parent;
};
```

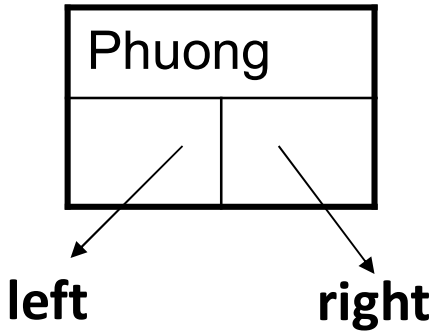


BST

```
typedef ... elementType;//bất cứ kiểu phần tử nào
typedef struct TreeNode {
   elementType data;
   struct TreeNode *left, *right;
};
```

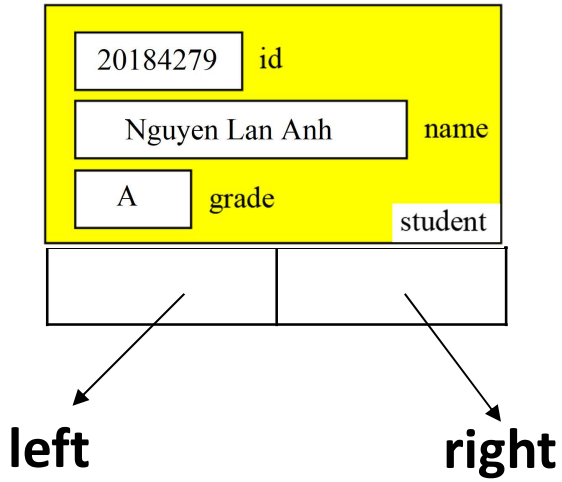
Example 2: key (also data) of each node is a string

```
struct TreeNode {
   char *data;
   TreeNode *left, *right;
};
```



Example 3: if data of each node is a record consisting of 3 information: studentID, full name of student, grade of DataStructures course (➔ studentID is key of node)

```
struct student  {
    char *id, *name;
    char grade;
};
struct TreeNode {
    student data;
    TreeNode *left, *right;
};
```



6.3. Binary search tree

6.3.1. Definition

6.3.2. Representation of binary search tree

6.3.3. Operations on binary search tree

6.3.3. Operations on BST

1. Search
2. Findmax, Findmin
3. Find Predecessor, Successor
4. Insert
5. Delete

To easy present how above operations work, we use a BST with integer key:

```
struct TreeNode
{
    int  key;
    TreeNode *left, *right, *parent;
};
```

Create a new node (create_node)

- **Input:** the element need to insert
- Steps:
  - Allocation memory for new node
  - Check the error of allocation
  - If success: insert data into new node; set left child and right child to NULL
- **Output:** pointer points to new node

```
TreeNode *create_node(int NewKey)
{
   TreeNode *N = new TreeNode;
   if (N == NULL)
   {
       cout<<"ERROR: Out of memory\n";
       exit(1);
   }
   else
   {
       N->key = NewKey;
       N->left = NULL; N->right = NULL;
       N->parent =NULL;
   }
   return N;
}
```

Create a new node (create_node)

```
TreeNode *create_node(int NewKey)
{
   TreeNode *N = new TreeNode;
   if (N == NULL)
   {
       cout<<"ERROR: Out of memory\n";
       exit(1);
   }
   else
   {
       N->key = NewKey;
       N->left = NULL; N->right = NULL;
       N->parent =NULL;
   }
   return N;
}
```

6.3.3. Operations on BST: Searching

How to search a value on a binary search tree:

(1) Start at the root

(2) Compare the value of the item you are searching for with the value stored at the root:

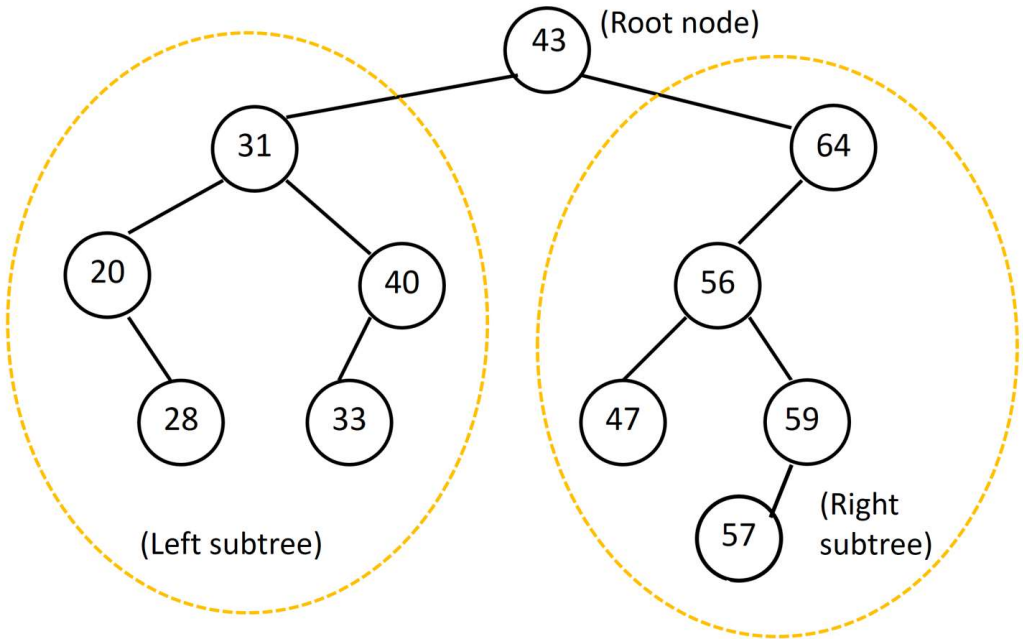
(2.1) If the values are equal, then *item found*;

(2.2) otherwise if it is a leaf node, then *not found*;

(2.3) If it is less than the value stored at the root, then search the left subtree;

(2.4) If it is greater than the value stored at the root, then search the right subtree;

(3) Repeat step 2 for the root of the subtree chosen in the previous step 2.3 or 2.4



6.3.3. Operations on BST: Searching

- The output:
  - If found (it means the target is equal to the key of current node), then return the pointer points to the node with the key is equal to target.
  - Otherwise, return NULL.

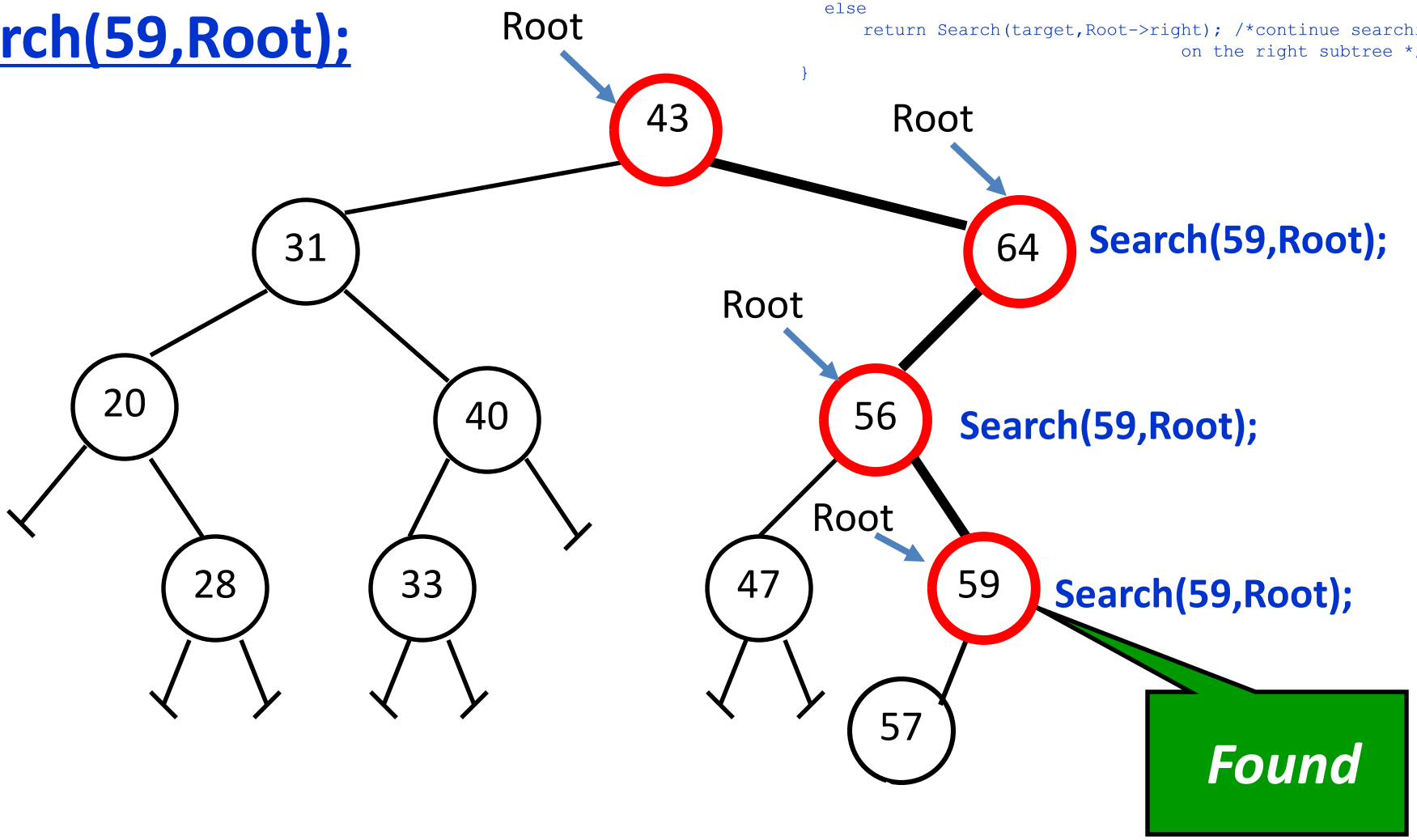
```
TreeNode* Search(int target,TreeNode* Root){
   if (Root == NULL) return NULL; //not found
   else if (target == Root->key) /* target is found */
       return Root;
   else if (target < Root->key)
       return Search(target,Root->left);/*continue searching
                                          on the left subtree*/
   else
       return Search(target,Root->right); /*continue searching
                                          on the right subtree */
}
```

Time Complexity: $O(h)$,
where h is height of the BST

Example 1: search key 59

```
TreeNode* Search(int target,TreeNode* Root){
   if (Root == NULL) return NULL; //not found
   else if (target == Root->key) /* target is found */
       return Root;
   else if (target < Root->key)
       return Search(target,Root->left);/*continue searching
                                          on the left subtree*/
   else
       return Search(target,Root->right); /*continue searching
                                          on the right subtree */
}
```

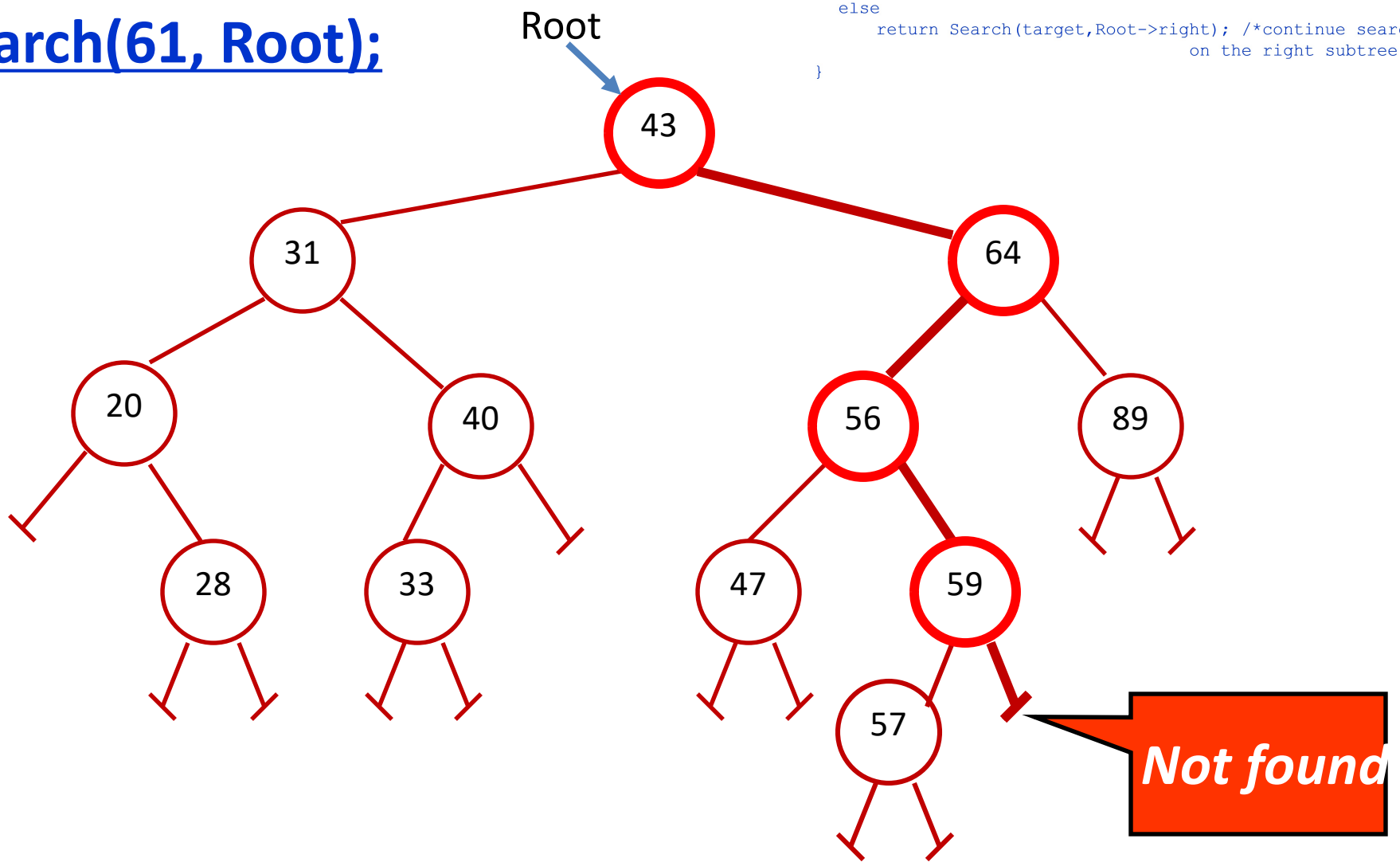
Search(59,Root);



Example 2: find key 61

```
TreeNode* Search(int target,TreeNode* Root){
   if (Root == NULL) return NULL; //not found
   else if (target == Root->key) /* target is found */
       return Root;
   else if (target < Root->key)
       return Search(target,Root->left);/*continue searching
                                          on the left subtree*/
   else
       return Search(target,Root->right); /*continue searching
                                          on the right subtree */
}
```

Search(61, Root);



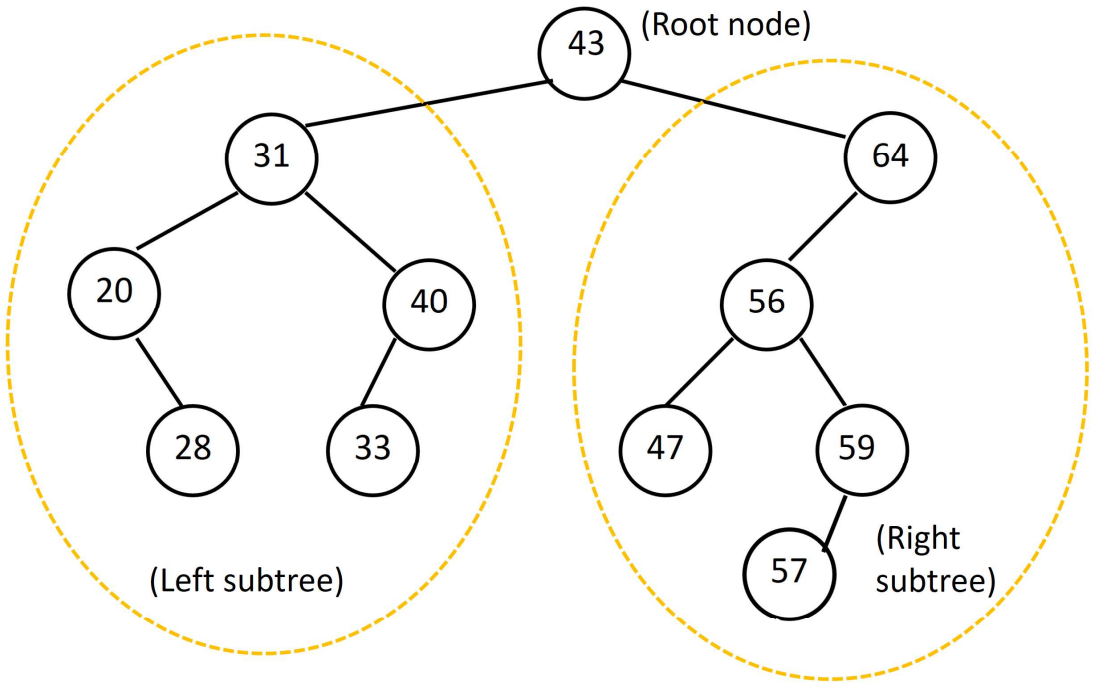
6.3.3. Operations on BST: findMin, findMax

- Where is the smallest element?

Ans: leftmost element

- Where is the largest element?

Ans: rightmost element



All values on the left subtree are **less** than the value in the root node

All values on the right subtree are **greater** than the value in the root node

6.3.3. Operations on BST: findMin, findMax

- To find the smallest key in BST, following left child pointers from the root, until a NULL is encountered

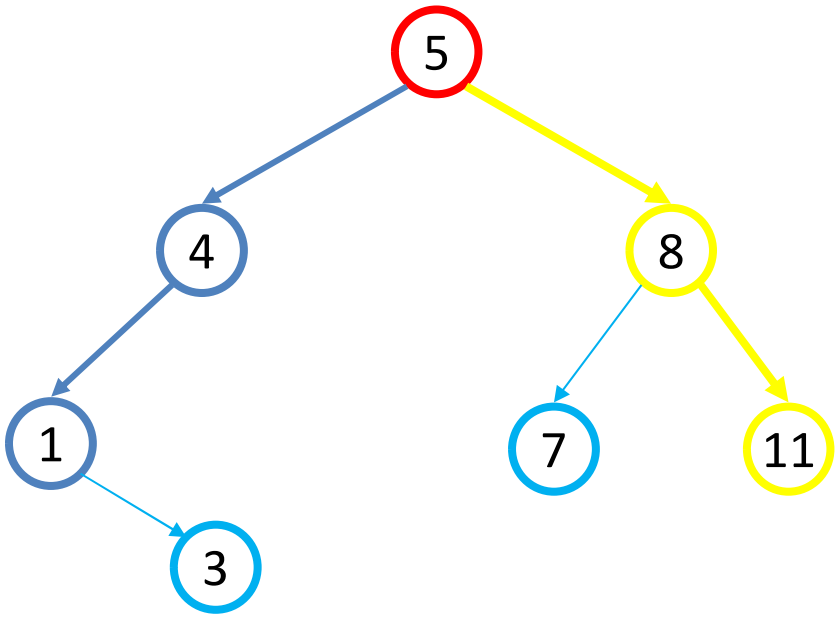
```
TreeNode*   find_min(TreeNode   *root)
```

function returns the node with smallest key on the BST with the root pointed by pointer "`root`"

- To find the largest key in BST, following right child pointers from the root, until a NULL is encountered

```
TreeNode*   find_max(TreeNode   *root)
```

funtion returns the node with largest key on the BST with the root pointed by pointer "`root`"



6.3.3. Operations on BST: findMin, findMax

```
TreeNode*  find_min(TreeNode* root)
{ /* always follow the left */
  if (root == NULL) return NULL;
  else
      if (root->left == NULL) return root;
      else return(find_min(root->left));
}

TreeNode*  find_max(TreeNode* root)
{/* always follow the right */
   if (root != NULL)
       while (root->right !=  NULL) root = root->right;
   return root;
}
```

6.3.3. Operations on BST: traverse

Traverse the BST in depth first search:

- Preorder (Duyệt theo thứ tự trước)

```
void printPreorder(TreeNode *root)
```

- Inorder (Duyệt theo thứ tự giữa)

```
void printInorder(TreeNode *root)
```

- Postorder (Duyệt theo thứ tự sau)

```
void printPostorder(TreeNode *root)
```

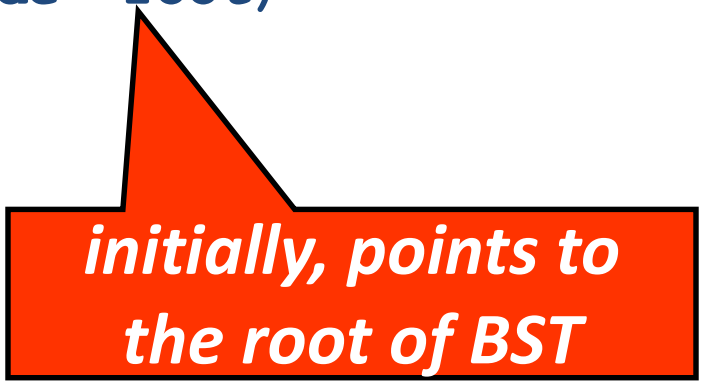
Preorder (Duyệt theo thứ tự trước)

```
void printPreorder(TreeNode *root)
{
      visit the node
      traverse left sub-tree
      traverse right sub-tree
}
```


```
void printPreorder(TreeNode *root)
{
  if (root != NULL) {
      cout<<root->key<<"  ";
      printPreorder(root->left);
      printPreorder(root->right);
  }
}
```

Postorder (Duyệt theo thứ tự sau)

```
void printPostorder(TreeNode *root)
{
     traverse left sub-tree
     traverse right sub-tree
     visit the node
}
```

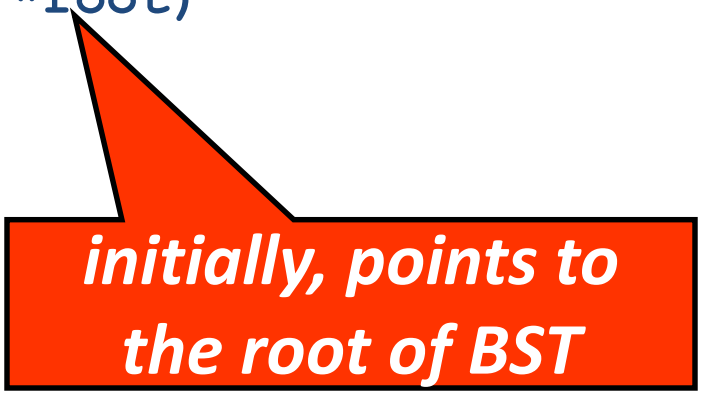



```
void printPostorder(TreeNode *root)
{
  if (root != NULL) {
     printPostorder(root->left);
     printPostorder(root->right);
     cout<<root->key<<"   ";
 }
}
```

Inorder (Duyệt theo thứ tự giữa)

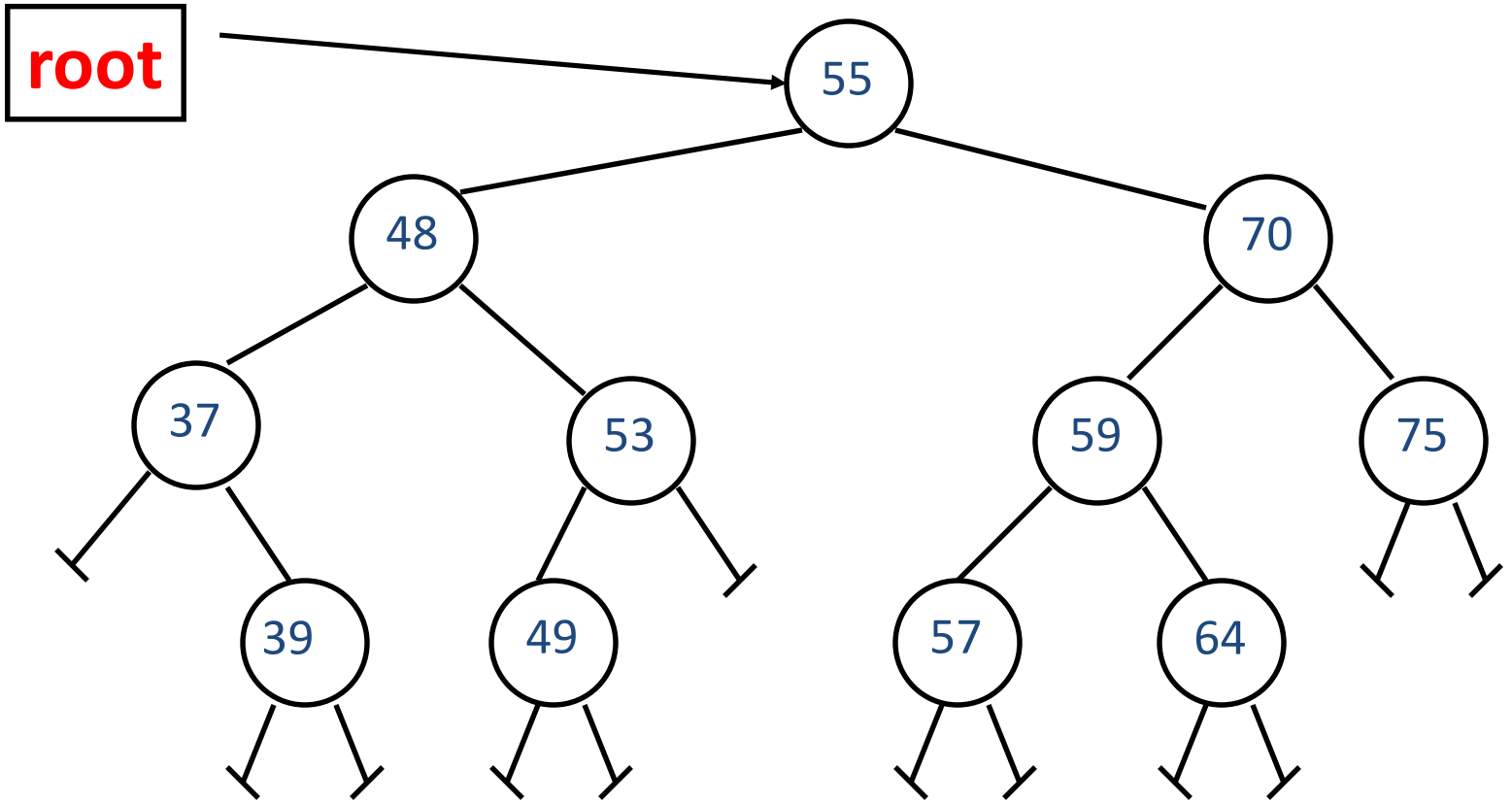
- Inorder on BST always give the sequence of keys is sorted

```
void printInorder(TreeNode *root)
{
      traverse left sub-tree
      visit the node
      traverse right sub-tree
}
```



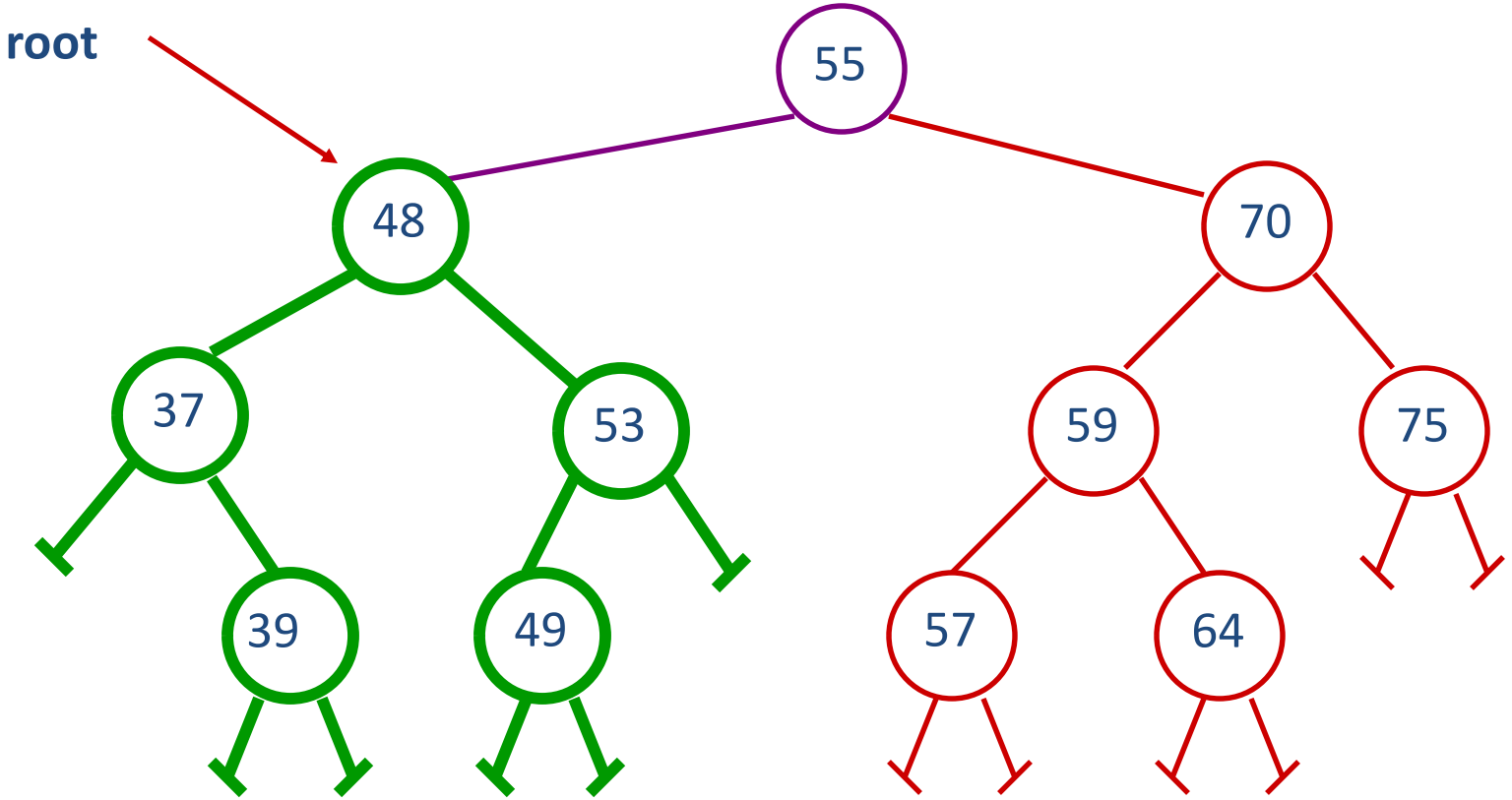

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder



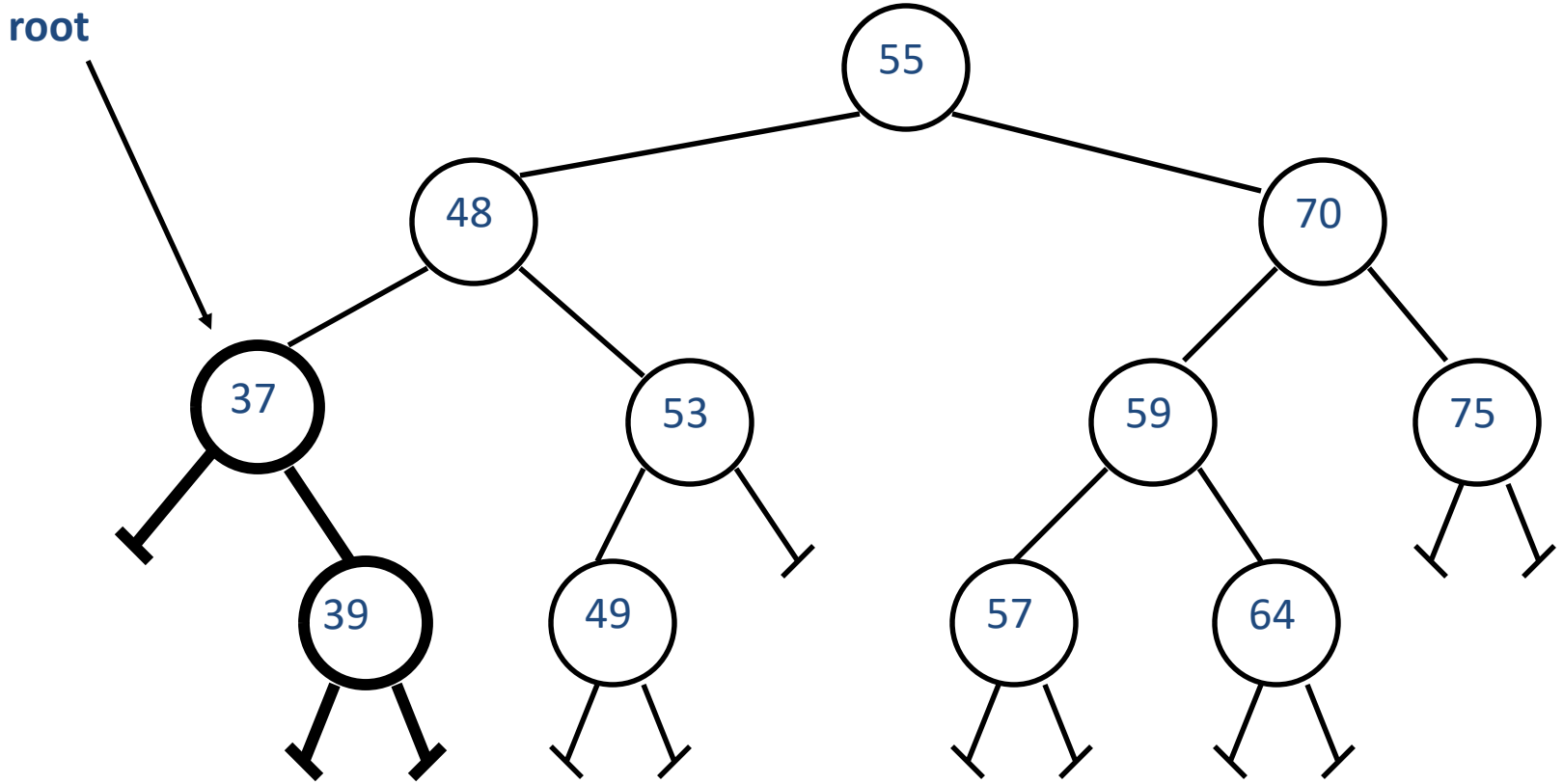

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder



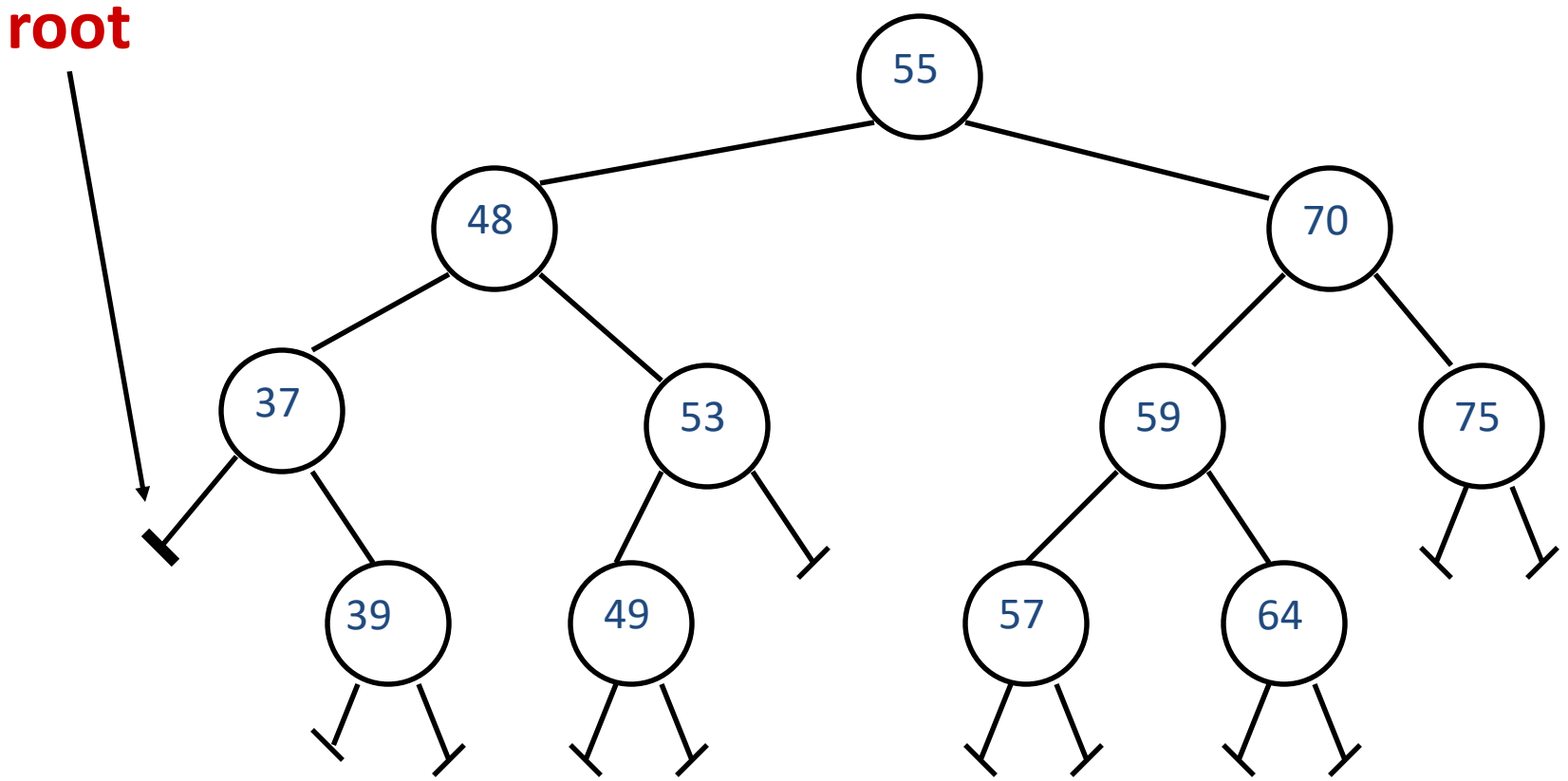

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder



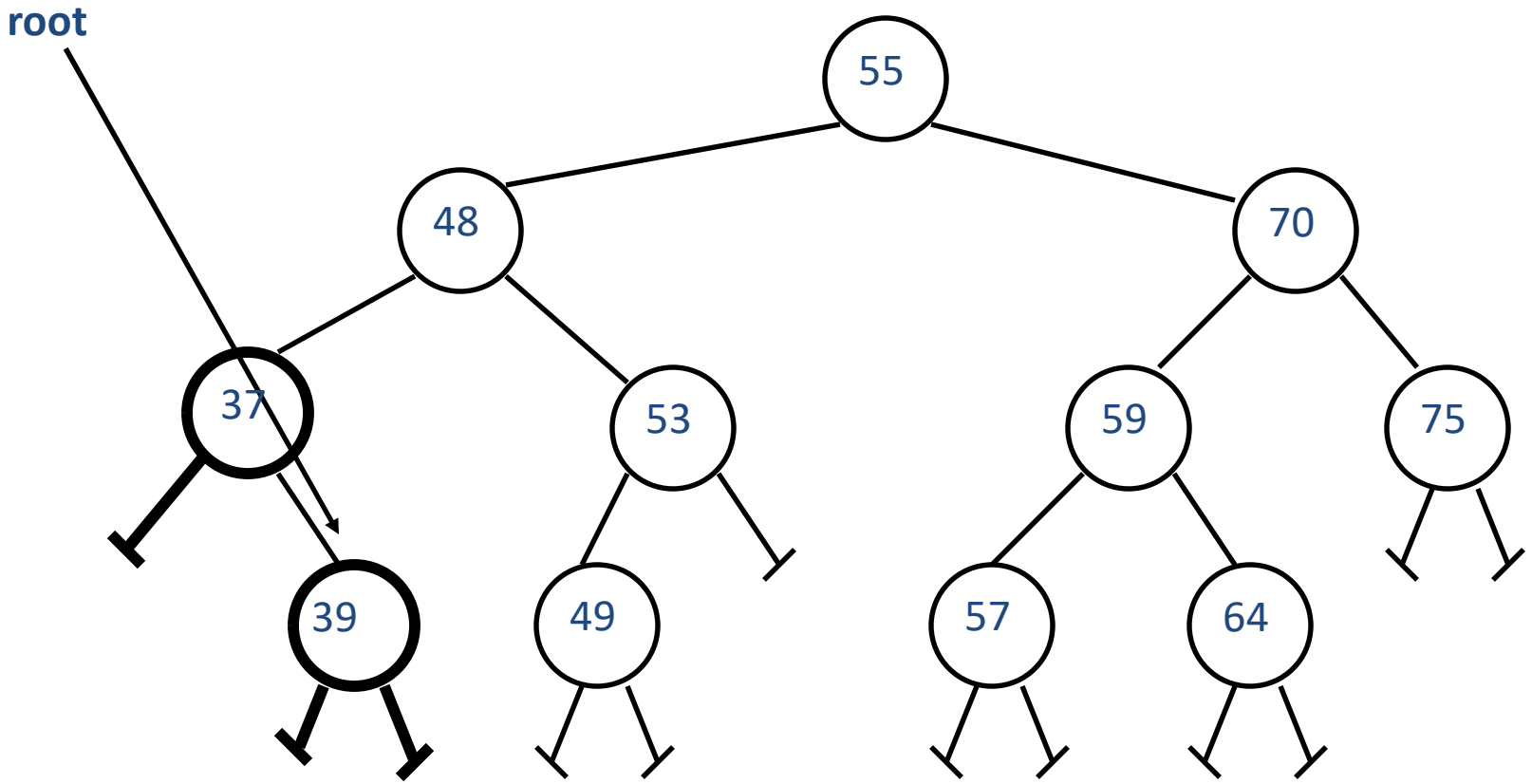

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder



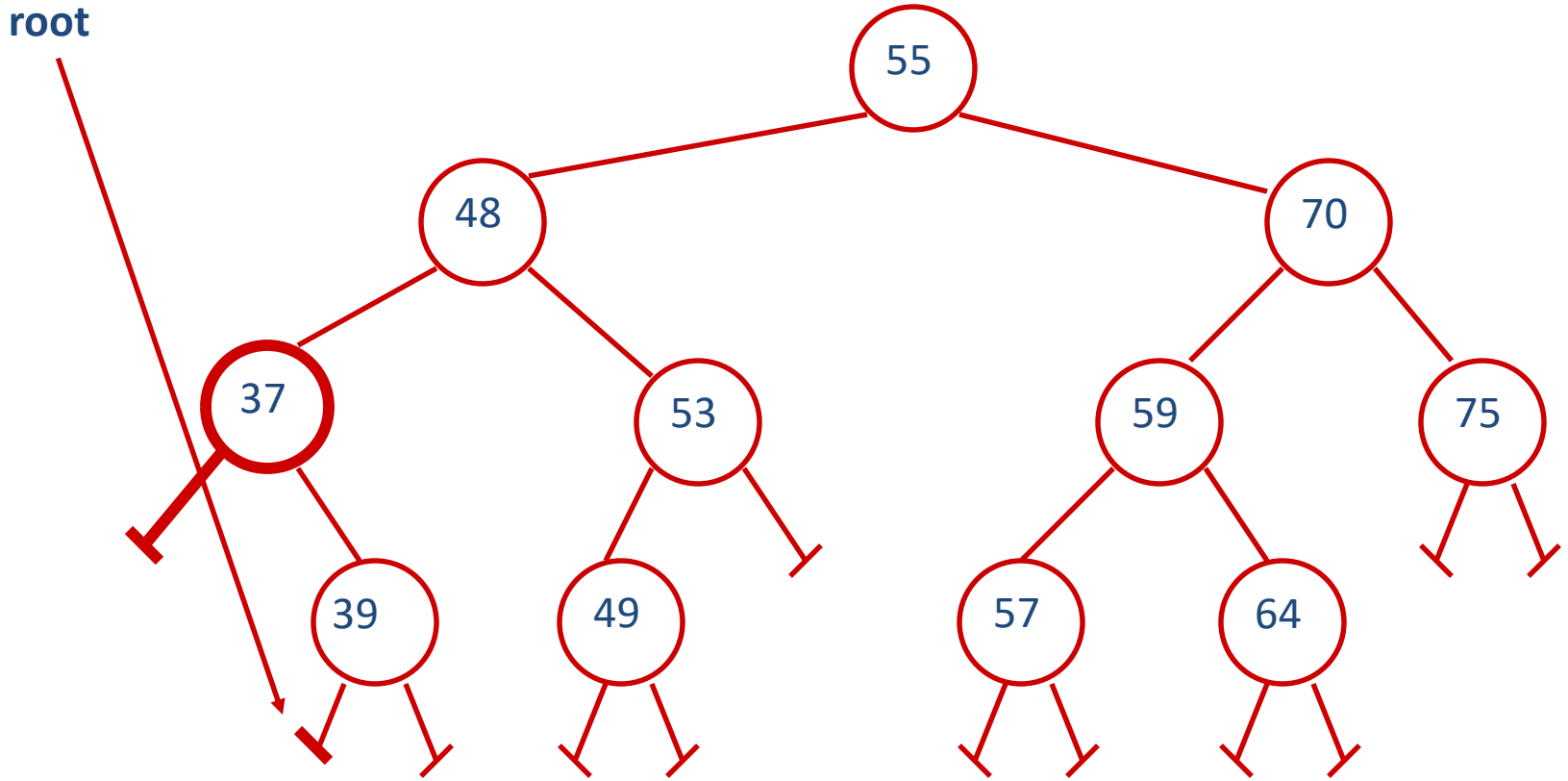

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37,



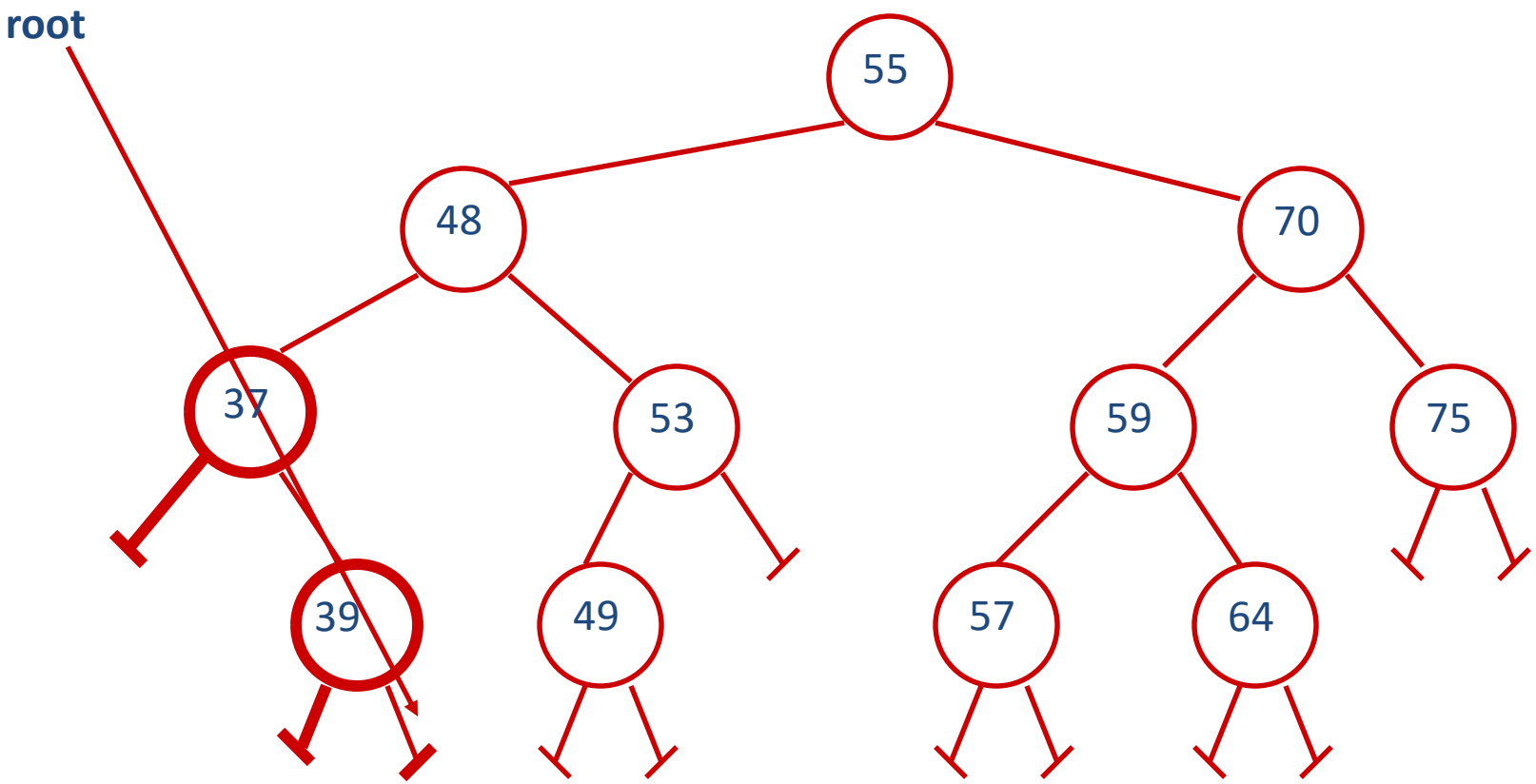

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37,



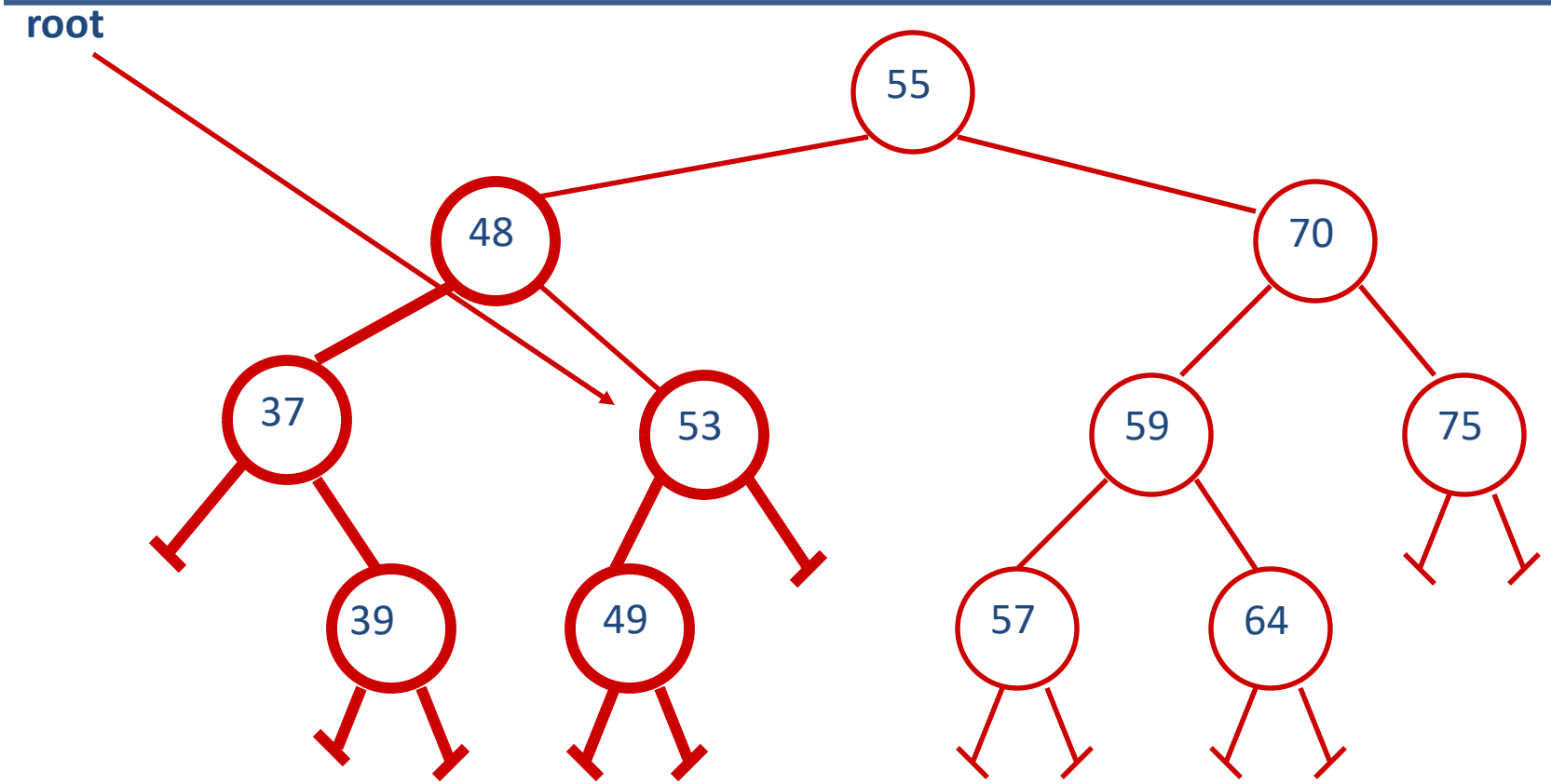

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37,39



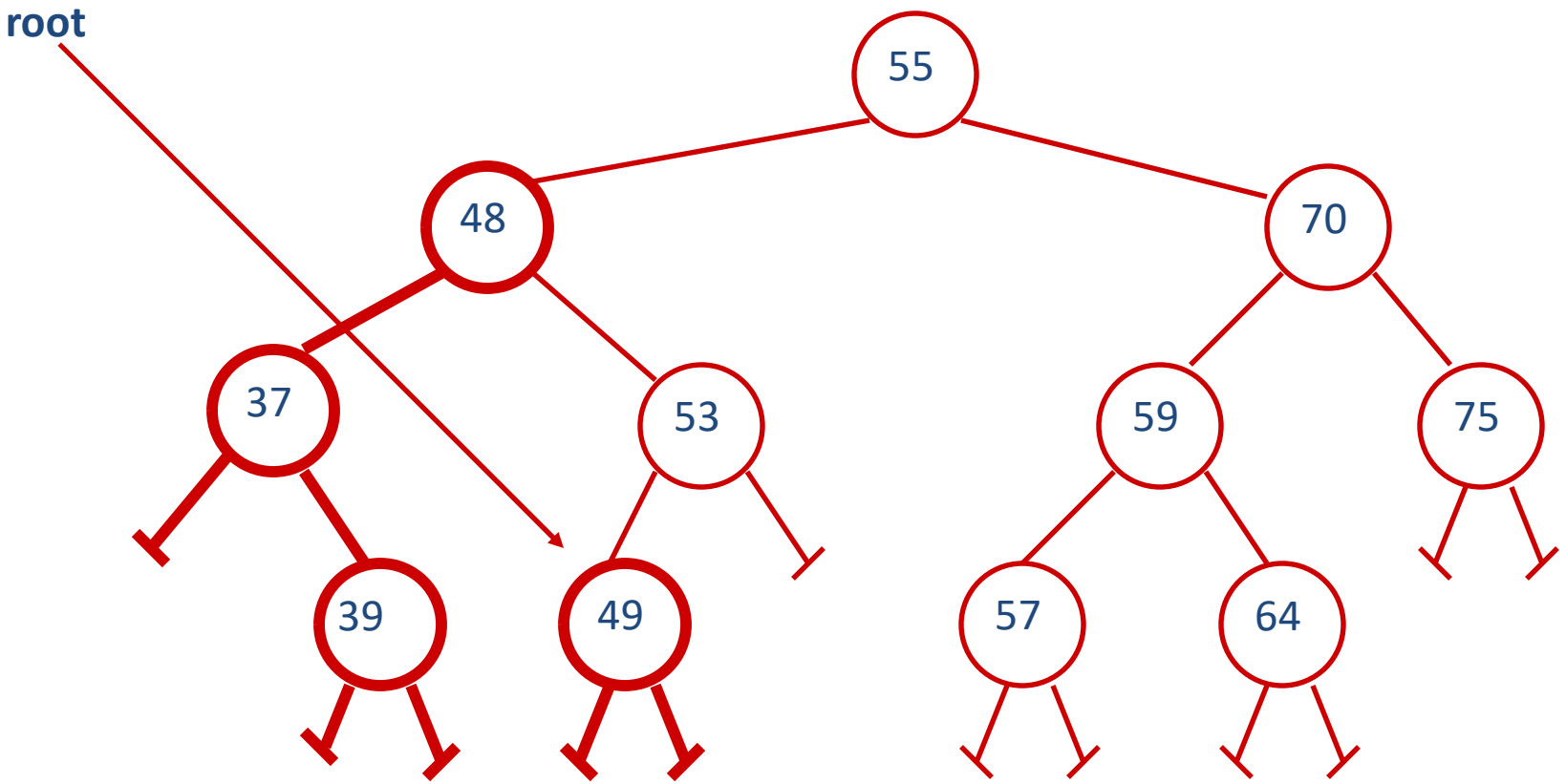

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48



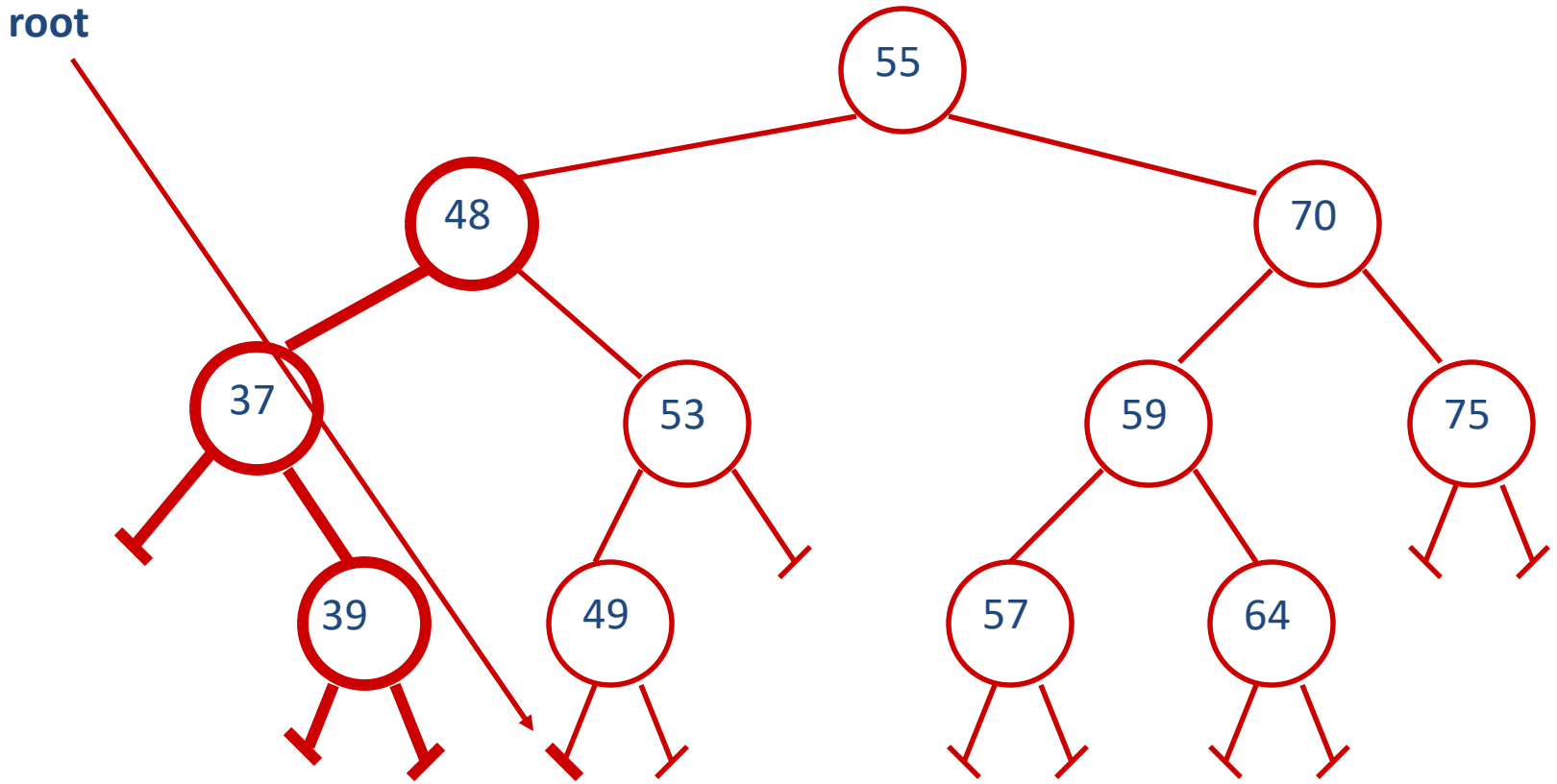

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48



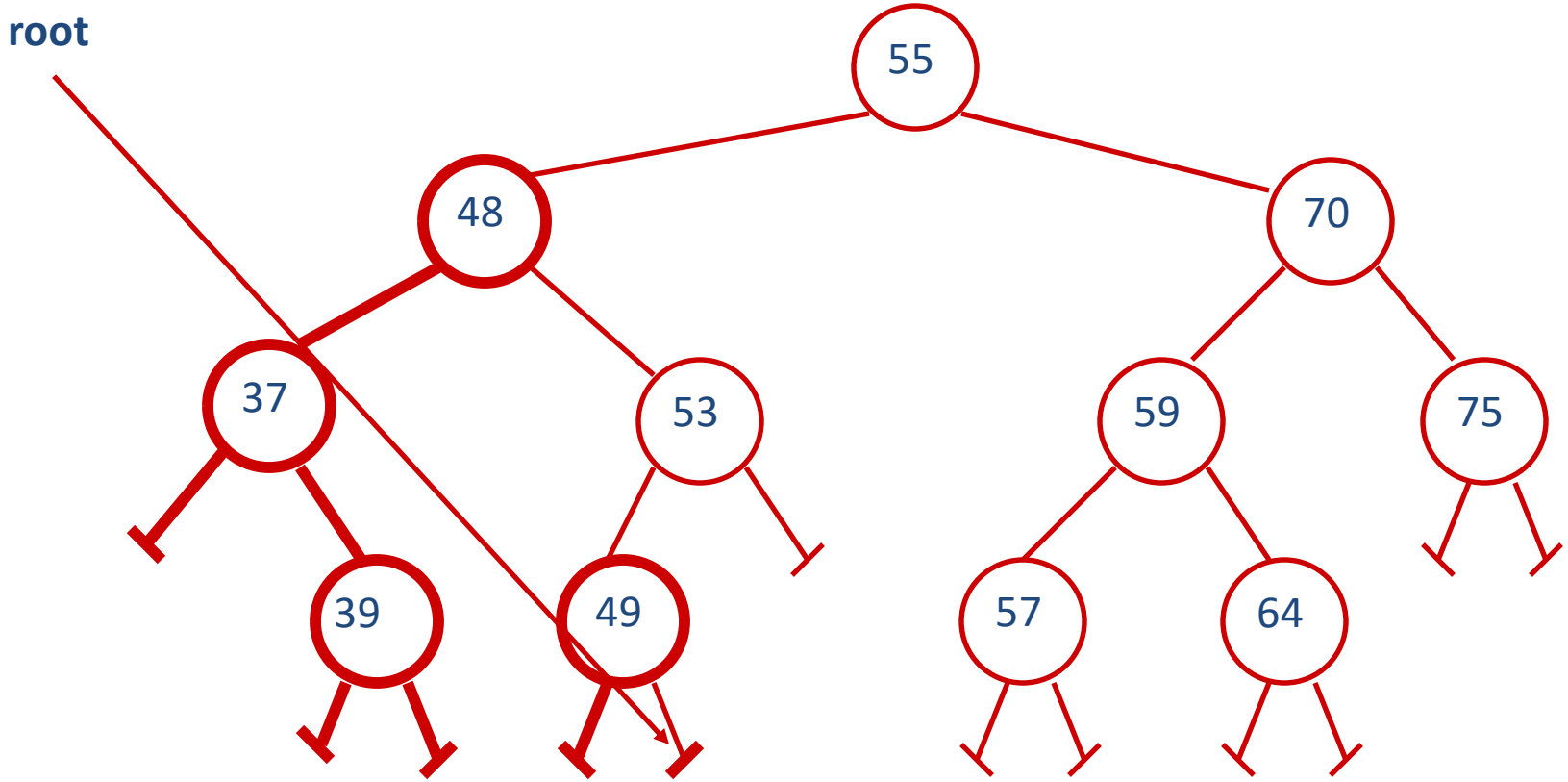

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48



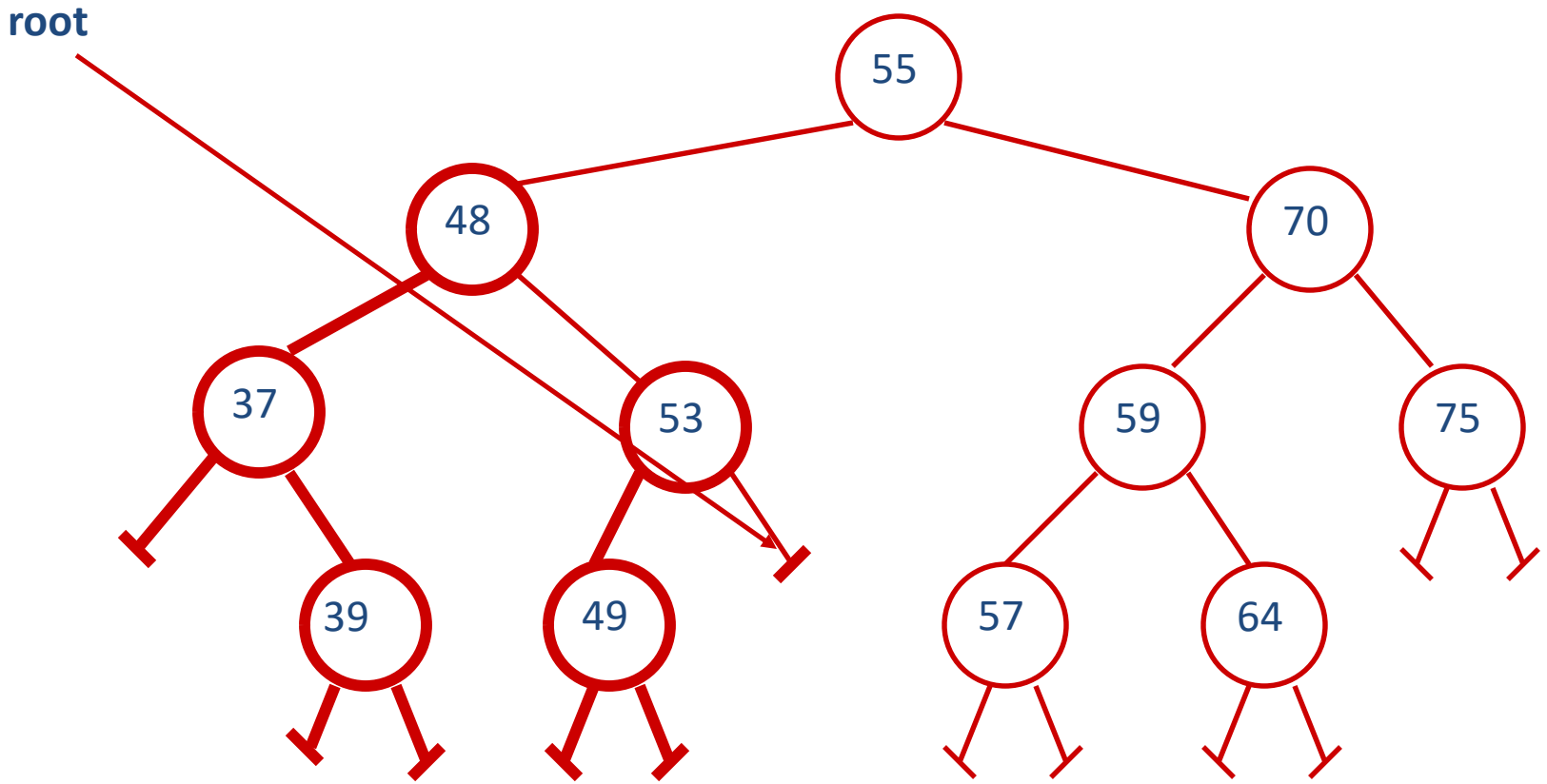

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49



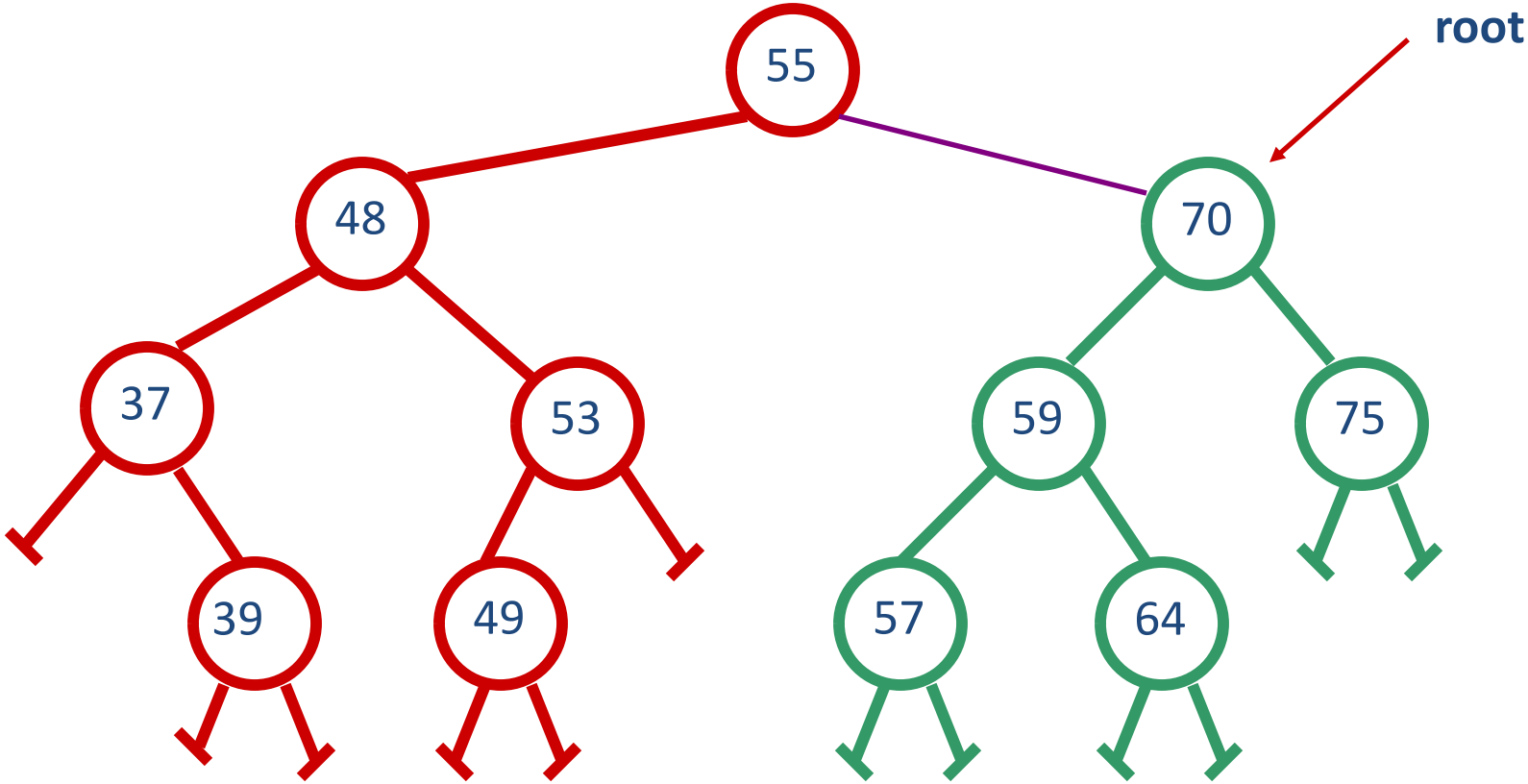

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53



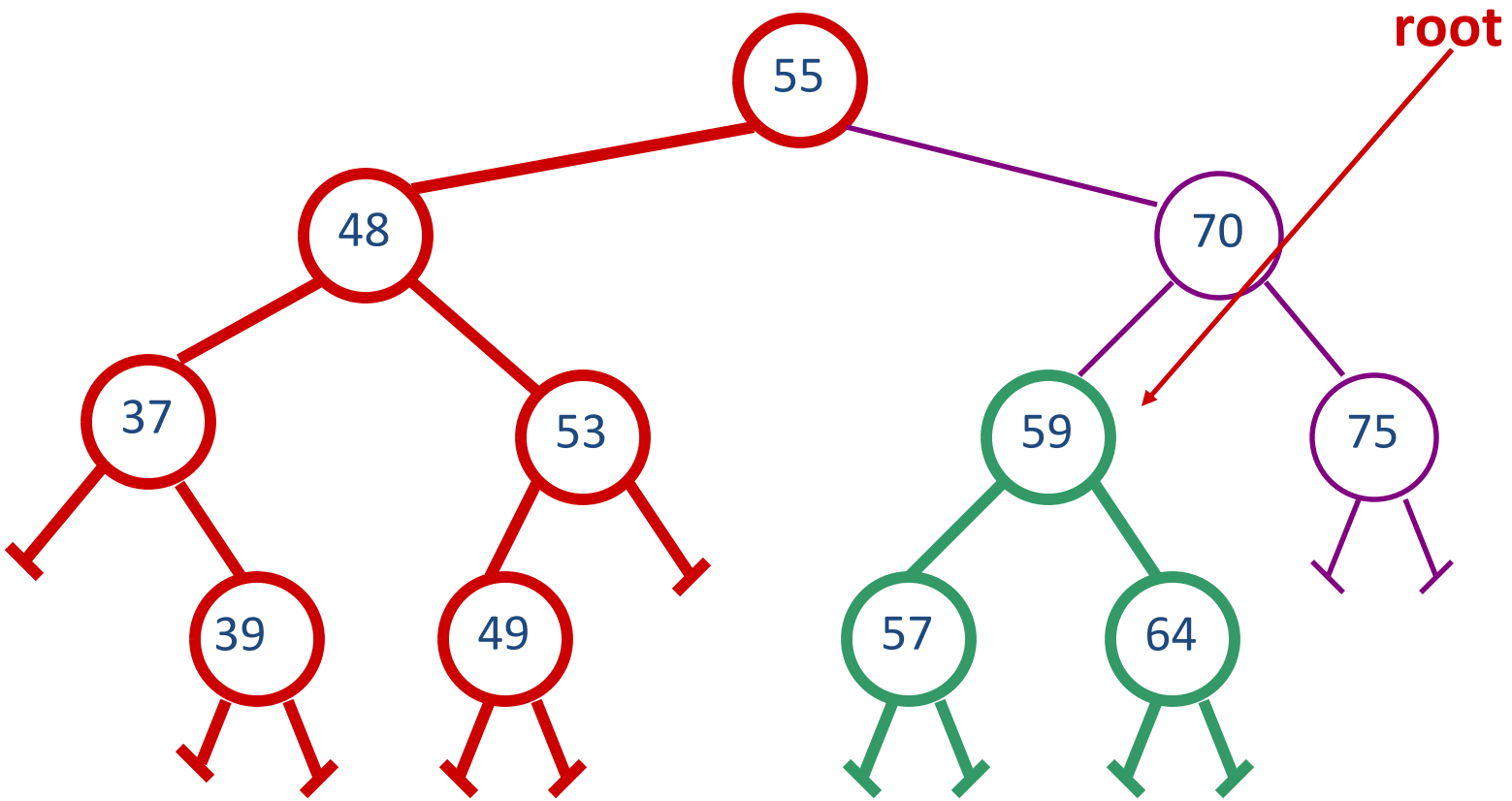

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55



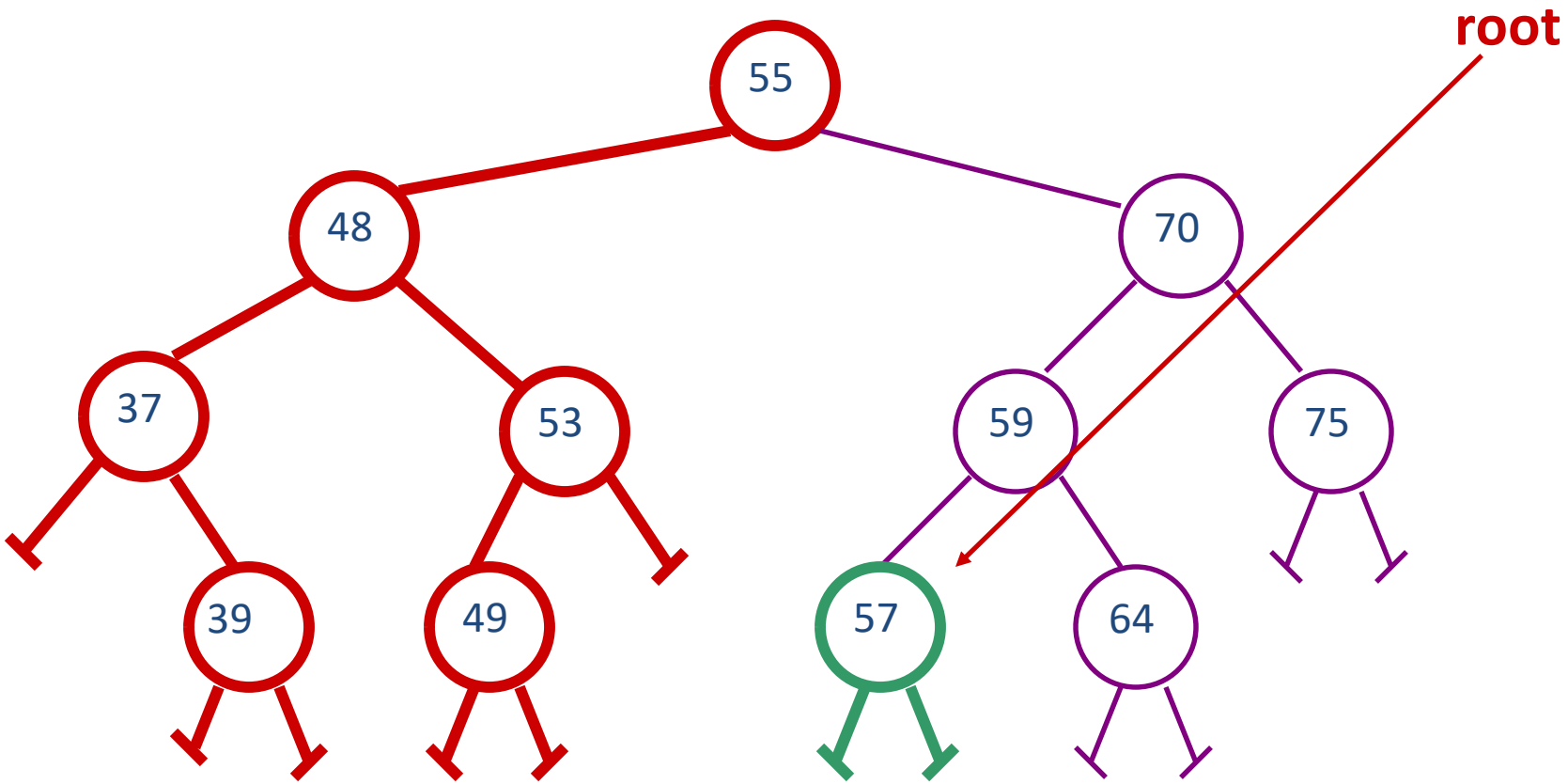

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55



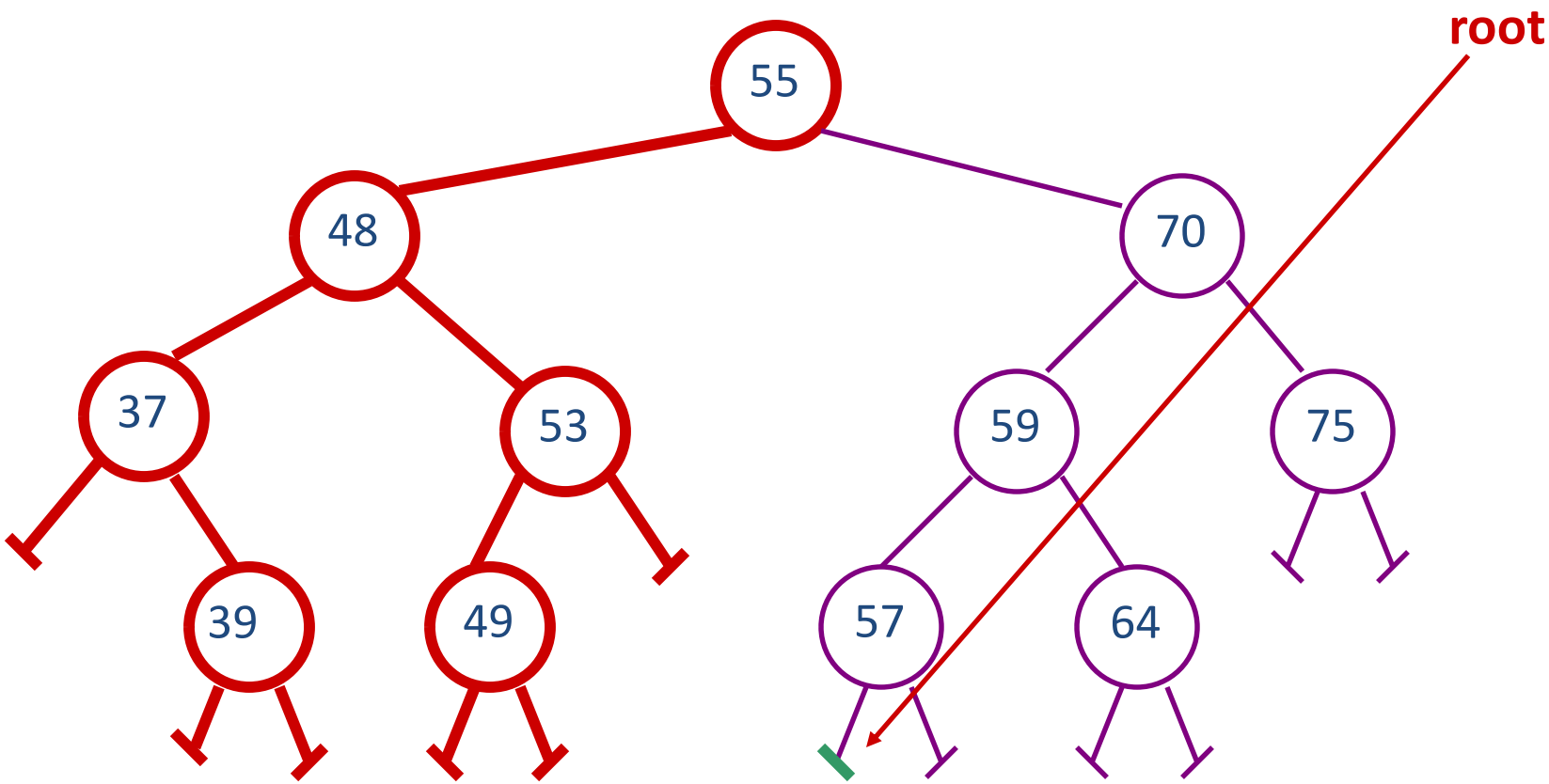

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55



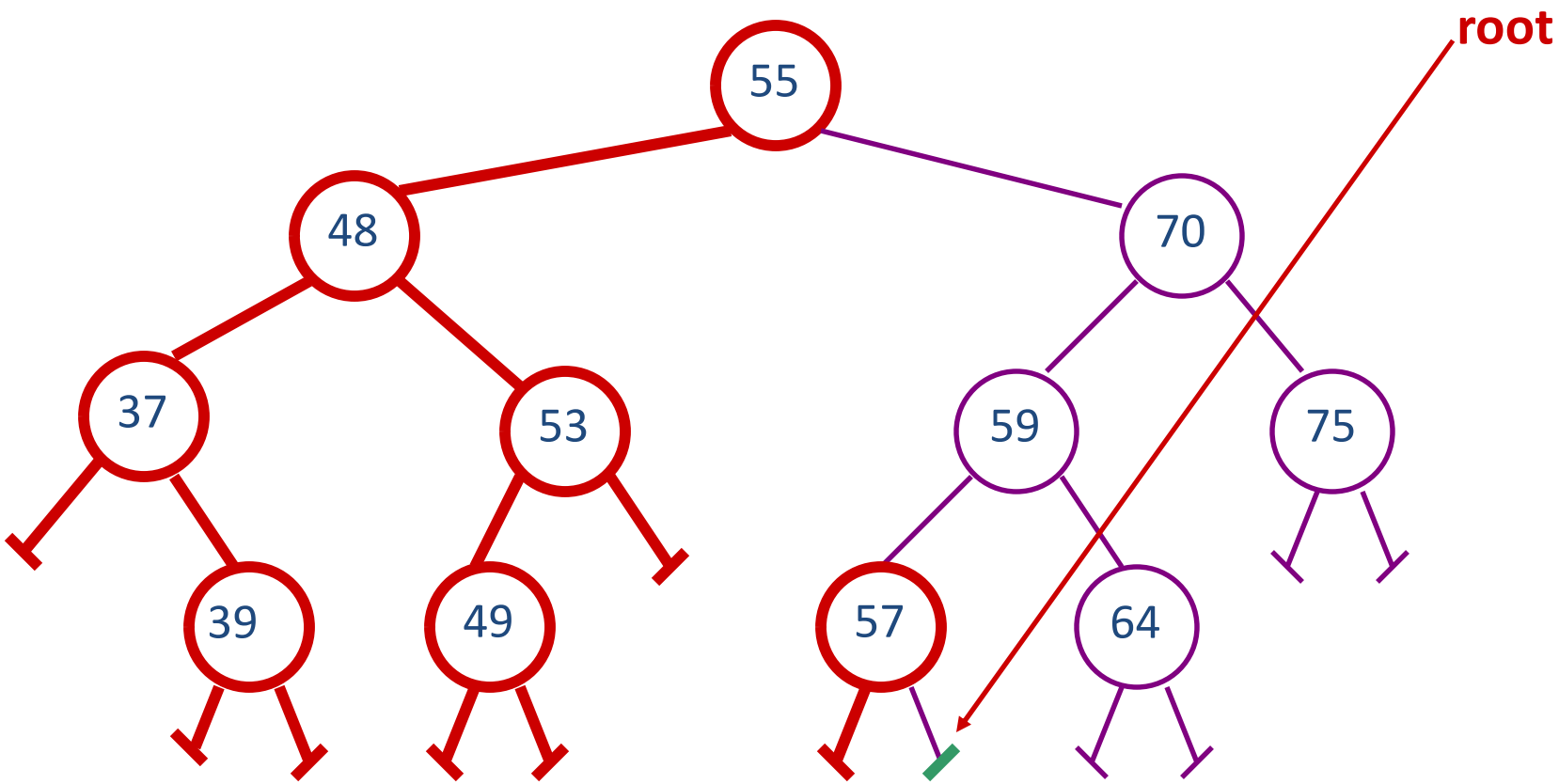

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55



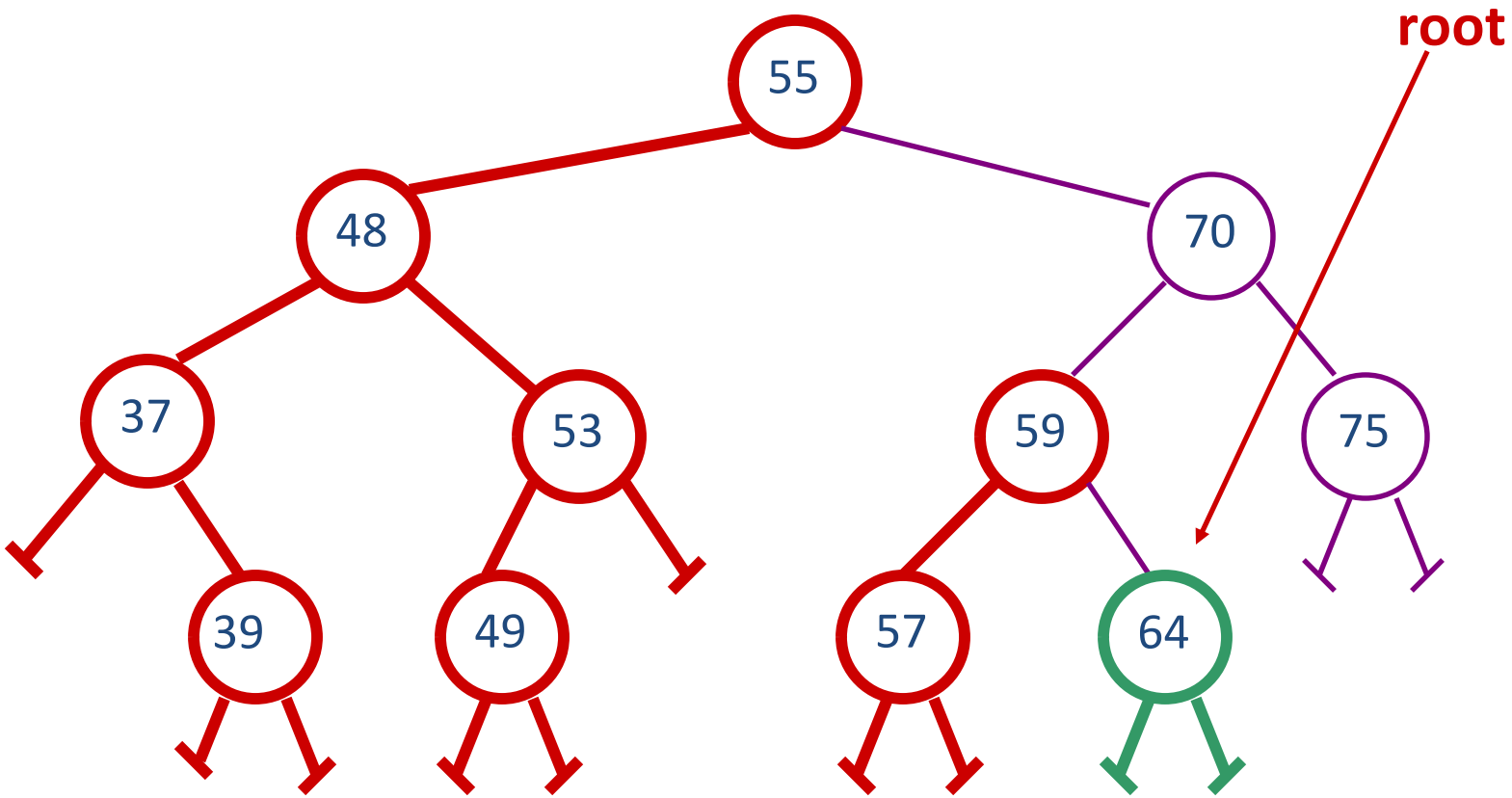

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57



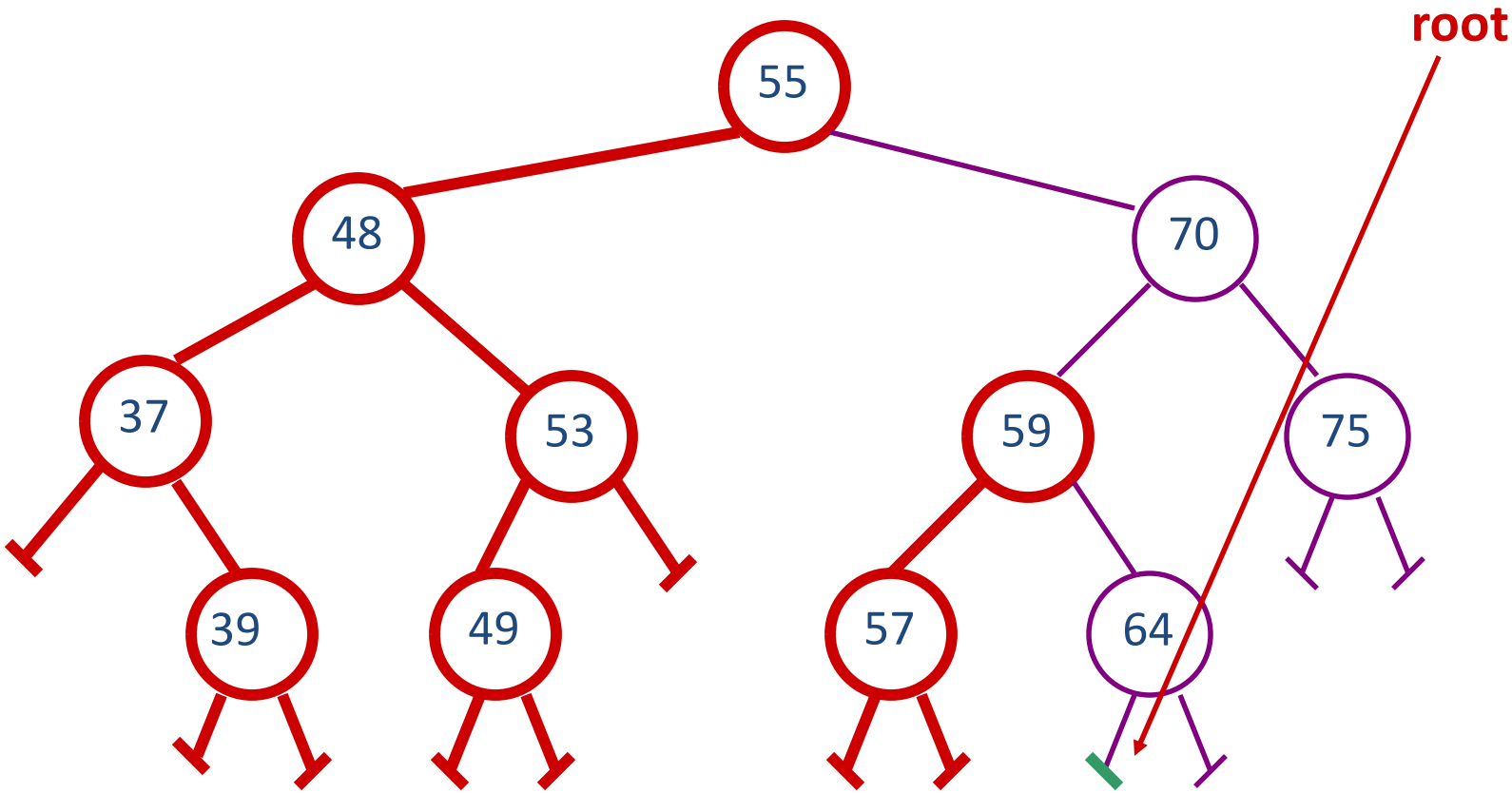

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59



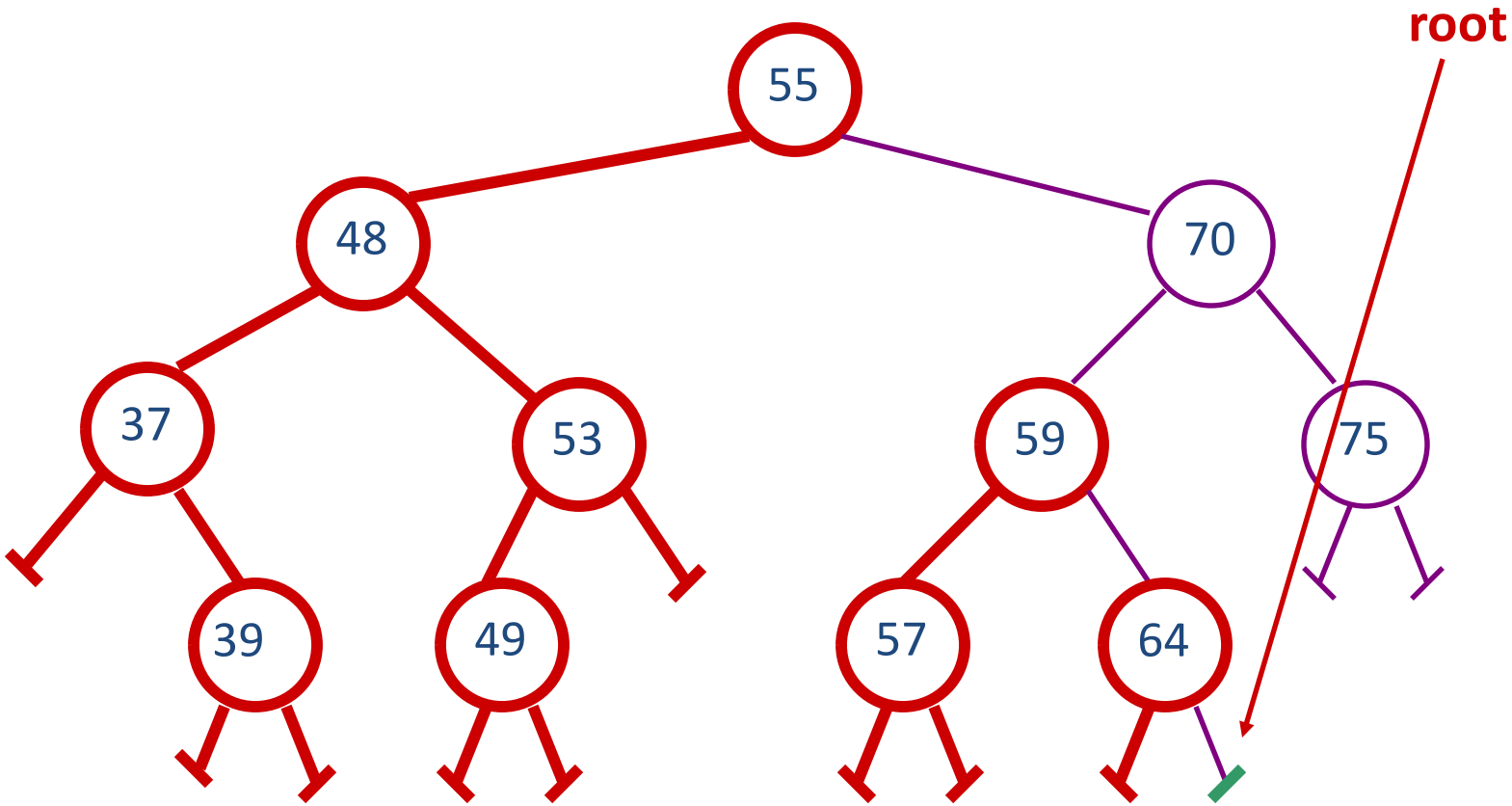

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59



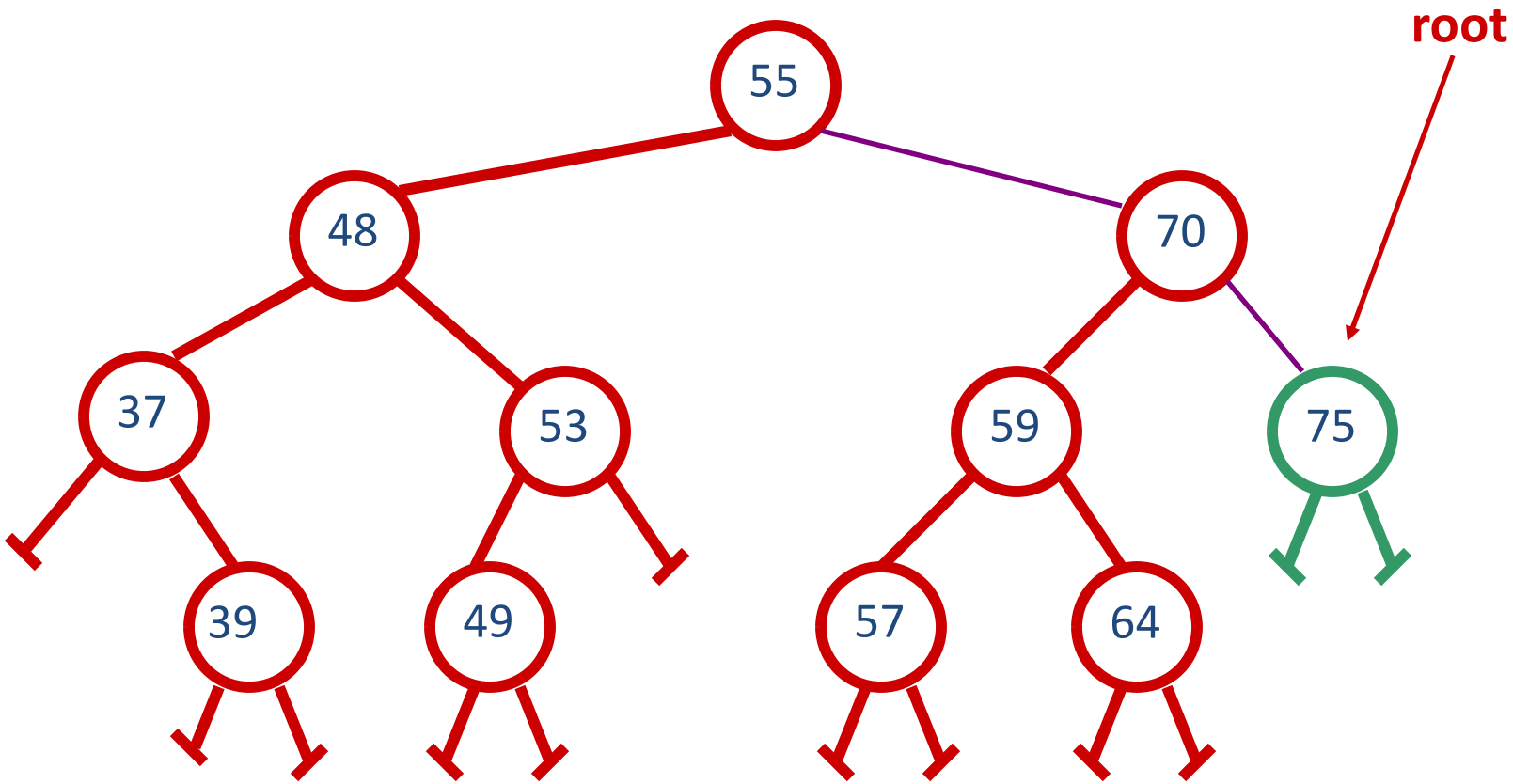

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64



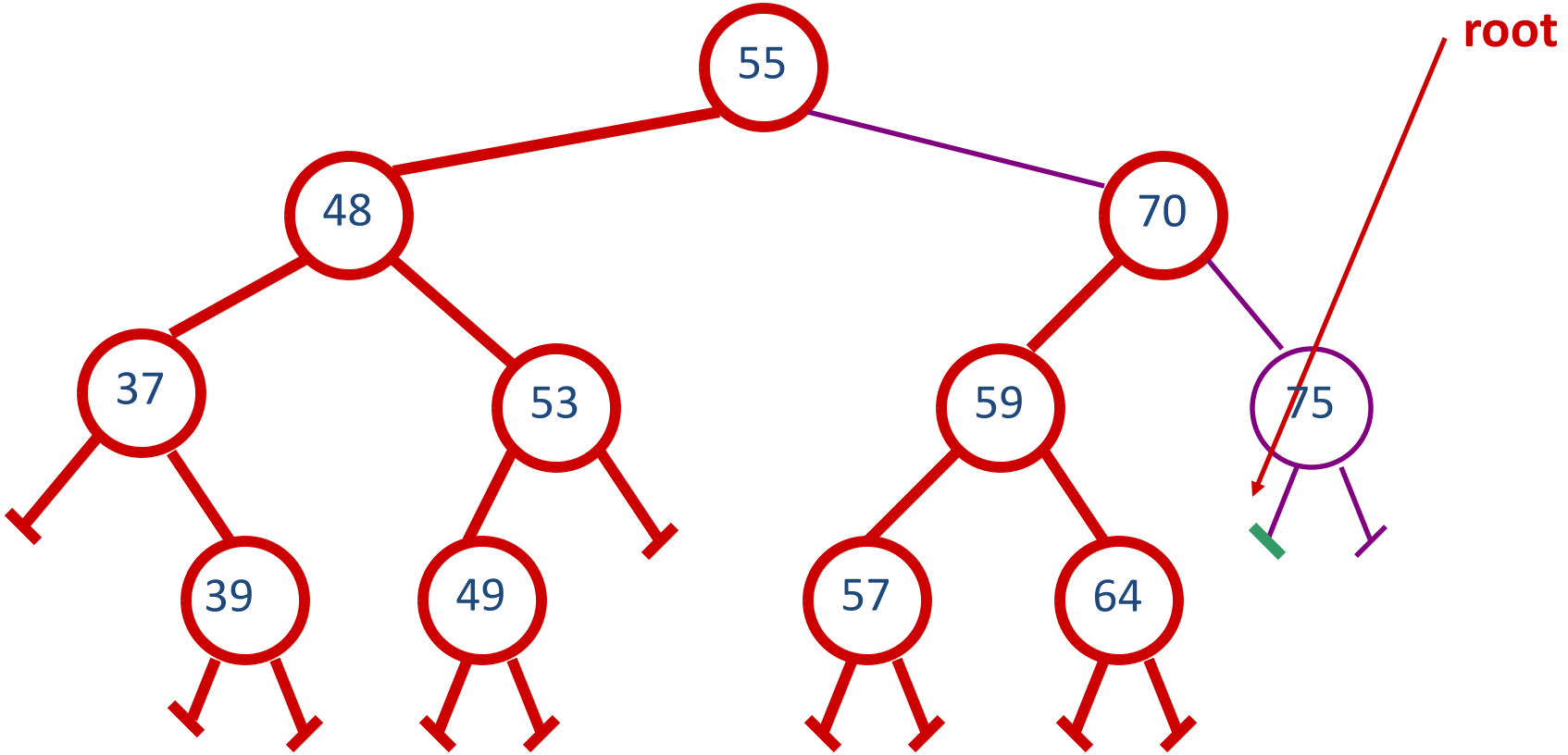

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70



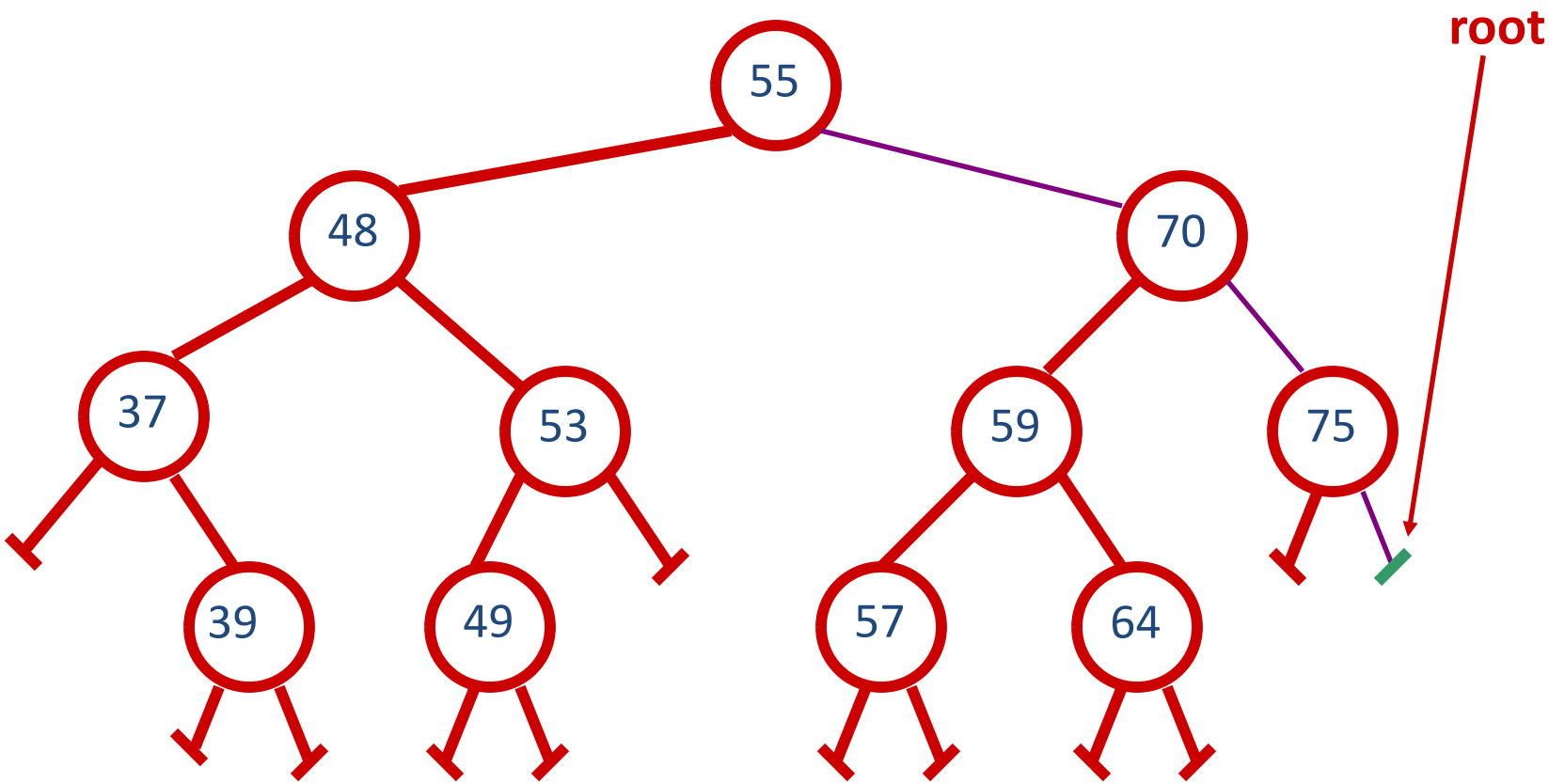

```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70




```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70, 75




```
void printInorder(TreeNode *root)
{
  if (root != NULL) {
     printInorder(root->left);
     cout<<root->key<<"  ";
     printInorder(root->right);
  }
}
```

6.3.3. Operations on BST

1. Search
2. Findmax, Findmin
3. Traversal the BST
4. **Find Predecessor, Successor**
5. Insert
6. Delete

To easy present how above operations work, we use a BST with integer key:

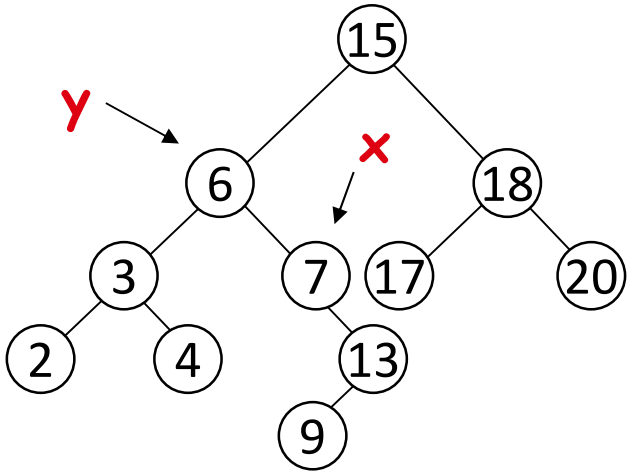
```
struct TreeNode
{
    int  key;
    TreeNode *left, *right, *parent;
};
```

Predecessor

Def: *predecessor* $(x) = y$ such that

key[y] is the biggest key < key [x]

(Predecessor of node x is node y such that *key*[y] is the largest key which is smaller than *key*[x]).



e.g.: *predecessor (7) =* *6*

predecessor (15) = *13*

predecessor (9) = *7*

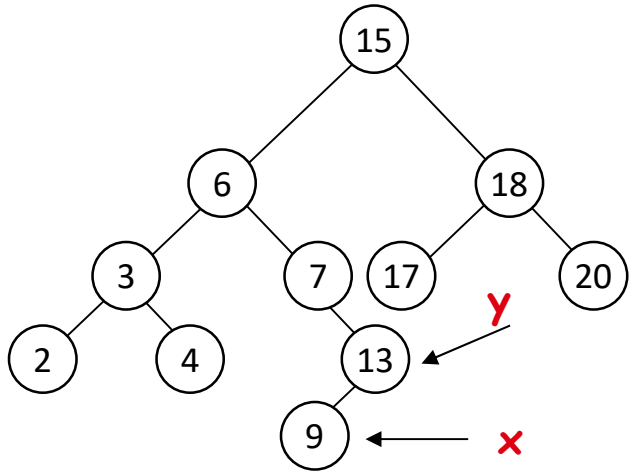
predecessor (2) = ?

Predecessor of the node with smallest key on BST is NULL.

Successor

Def: *successor* (x) = y, such that key[y] is the smallest key > key [x]

(Successor of node x is node y such that $key[y]$ is the smallest key which is greater than $key[x]$).



e.g.: *successor (9) =* 13

successor (15) = 17

successor (13) = 15

successor (20) = ?

Successor of the node with largest key on the BST is NULL.

Predecessor (Kế cận trước) and Successor (Kế cận sau)

- Predecessor of node x is node y such that $key[y]$ is the largest key $< key[x]$.

➔ Predecessor of the node with smallest key is NULL.

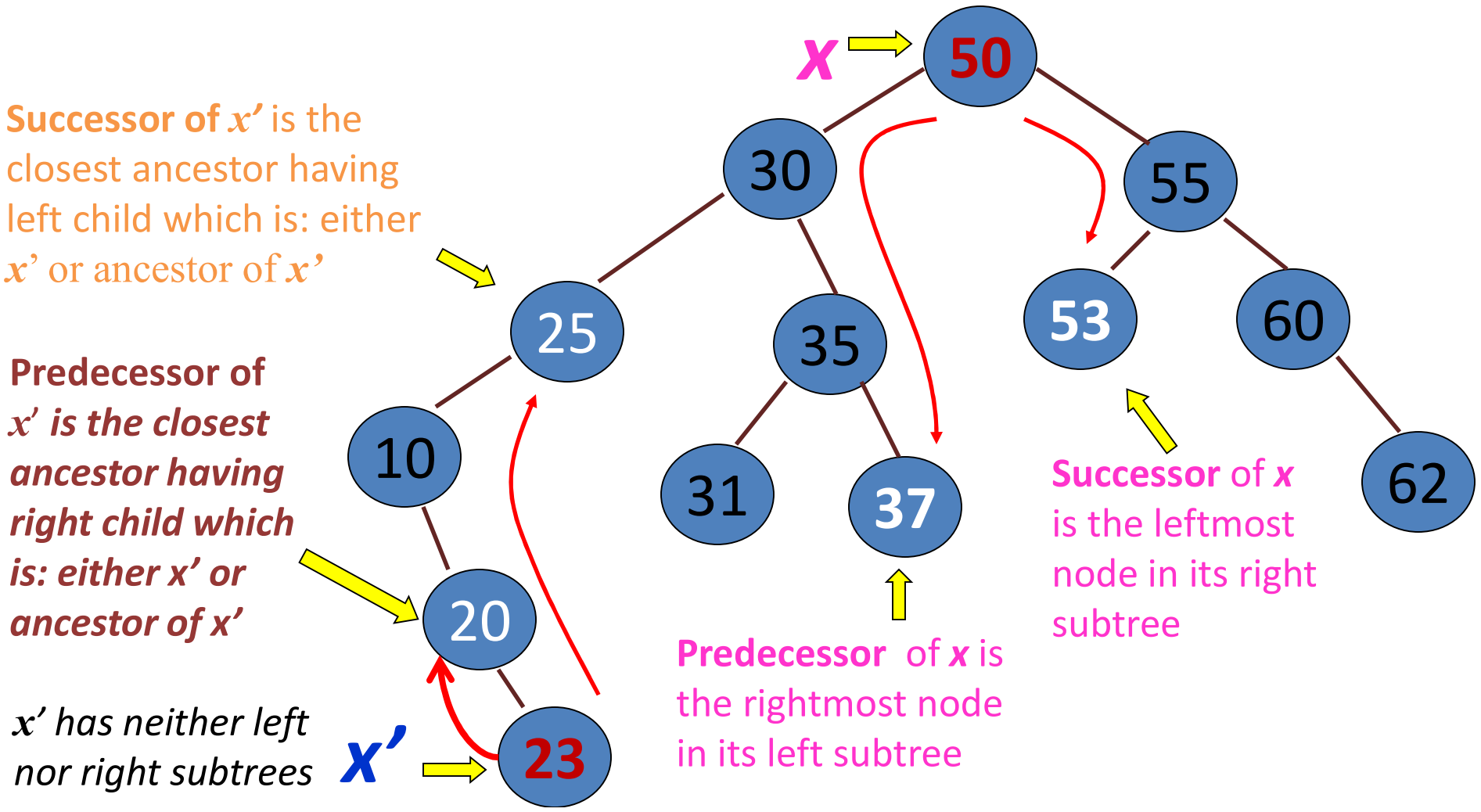
- Successor of node x is node y such that $key[y]$ is the smallest key $> key[x]$.

➔ Successor of the node with largest key is NULL.

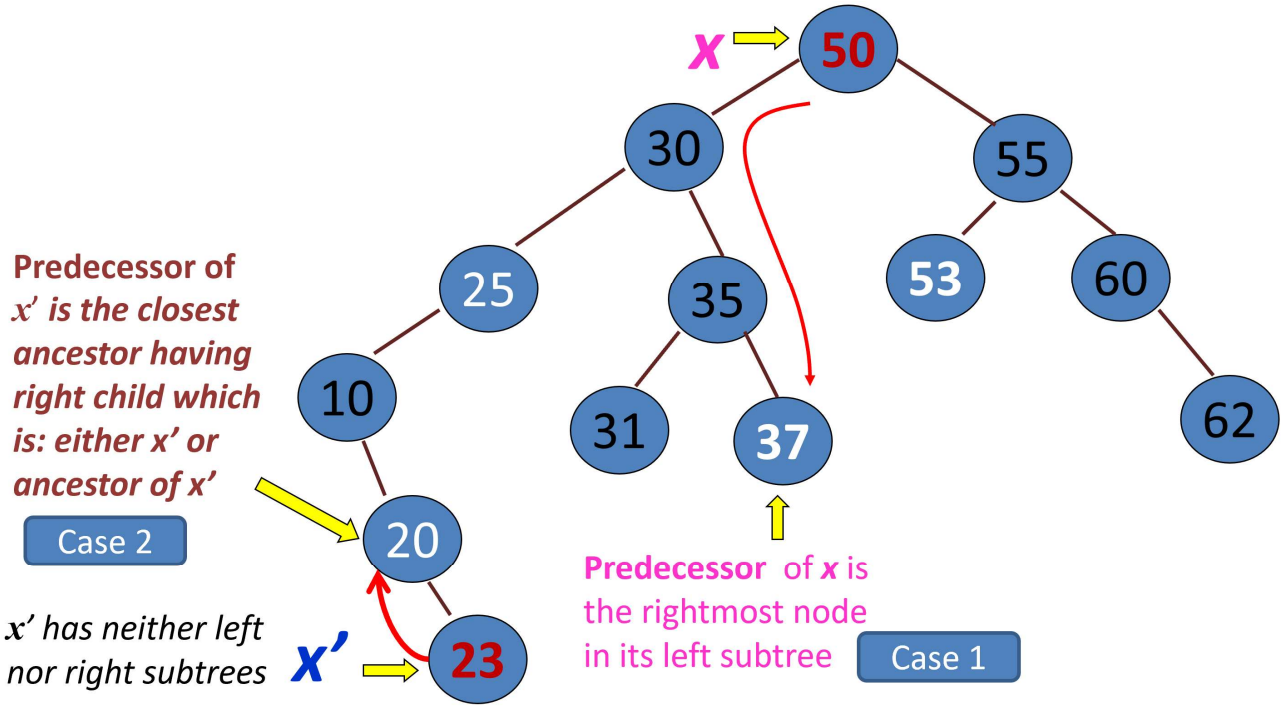
We need to find predecessor/successor of a node on BST without doing comparisons on keys

Predecessor (Kế cận trước) and Successor (Kế cận sau)

10, **20**, **23**, **25**, 30, 31, 35, **37**, **50**, **53**, 55, 60, 62



Predecessor



- Case 1: left (x) is non empty
  - $predecessor\,(x)$ = the maximum in left (x)
- Case 2: left (x) is empty
  - go up the tree until the current node is a right child: $predecessor\,(x)$ is the parent of the current node
  - if you cannot go further (and you reached the root): x is the smallest element

Find predecessor of x

`TreeNode *Predecessor(TreeNode *root, Treenode *x)`: return the node which is the predecessor of x

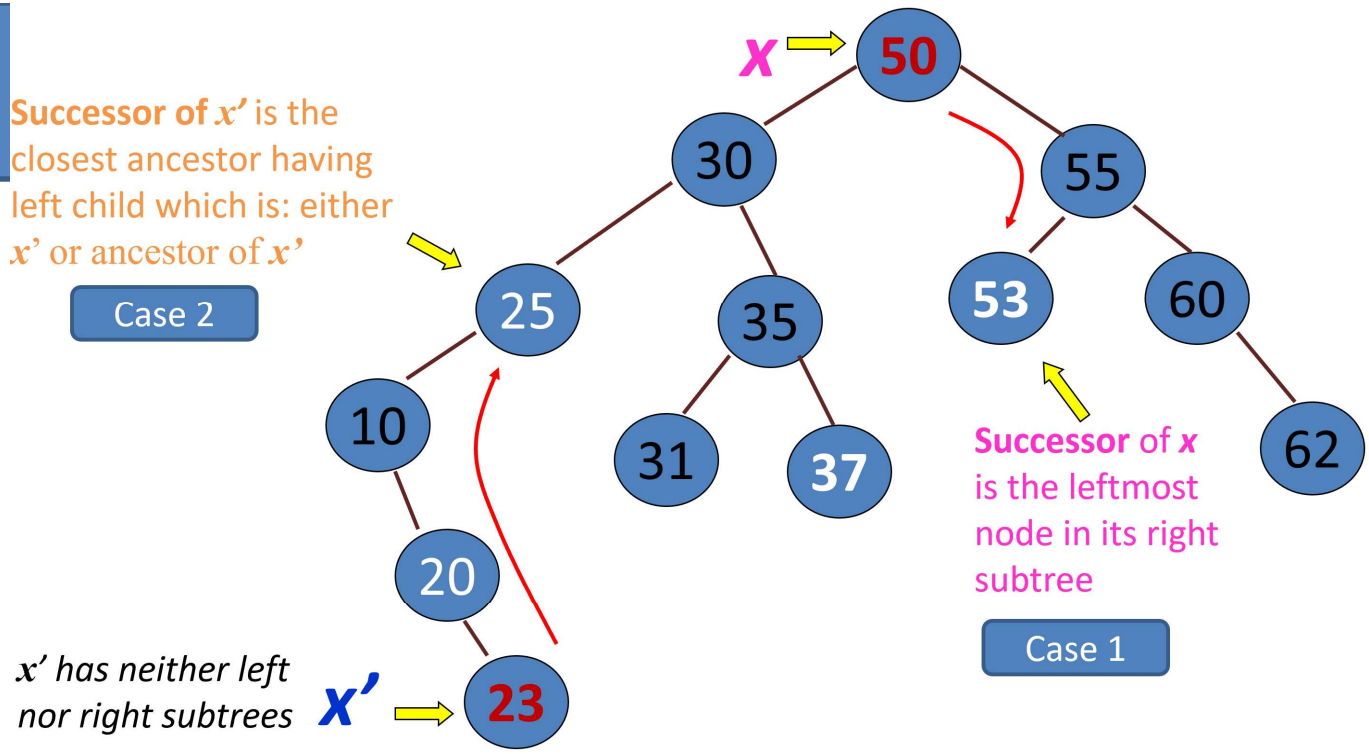
```
{
    if (x ->left != NULL)
        return find_max(x->left);
    TreeNode *p = x->parent;
    while (p!=NULL && x == p->left) {x = p; p = p->parent;}
    return p;
}
```

Running time:

***O* (h)**, h – height of the tree

- Case 1: left (x) is non empty
  - $predecessor\,(x)$ = the maximum in left (x)
- Case 2: left (x) is empty
  - go up the tree until the current node is a right child: $predecessor\,(x)$ is the parent of the current node
  - if you cannot go further (and you reached the root): x is the smallest element

Successor



- Case 1: x->right != NULL

  successor (x) = the minimum in x->right

- Case 2: x->right == NULL
  - go up the tree until the current node is a *left child*: *successor* (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element (no successor!)

Find successor of x

`TreeNode *Successor(TreeNode *root, Treenode *x)`: return the node which is the successor of x

```
{
    if (x ->right != NULL)
        return find_min(x->right);
    TreeNode *p = x->parent;
    while (p!=NULL && x == p->right) {x = p; p = p->parent;}
    return p;
}
```

Running time:

***O* (h)**, h – height of the tree

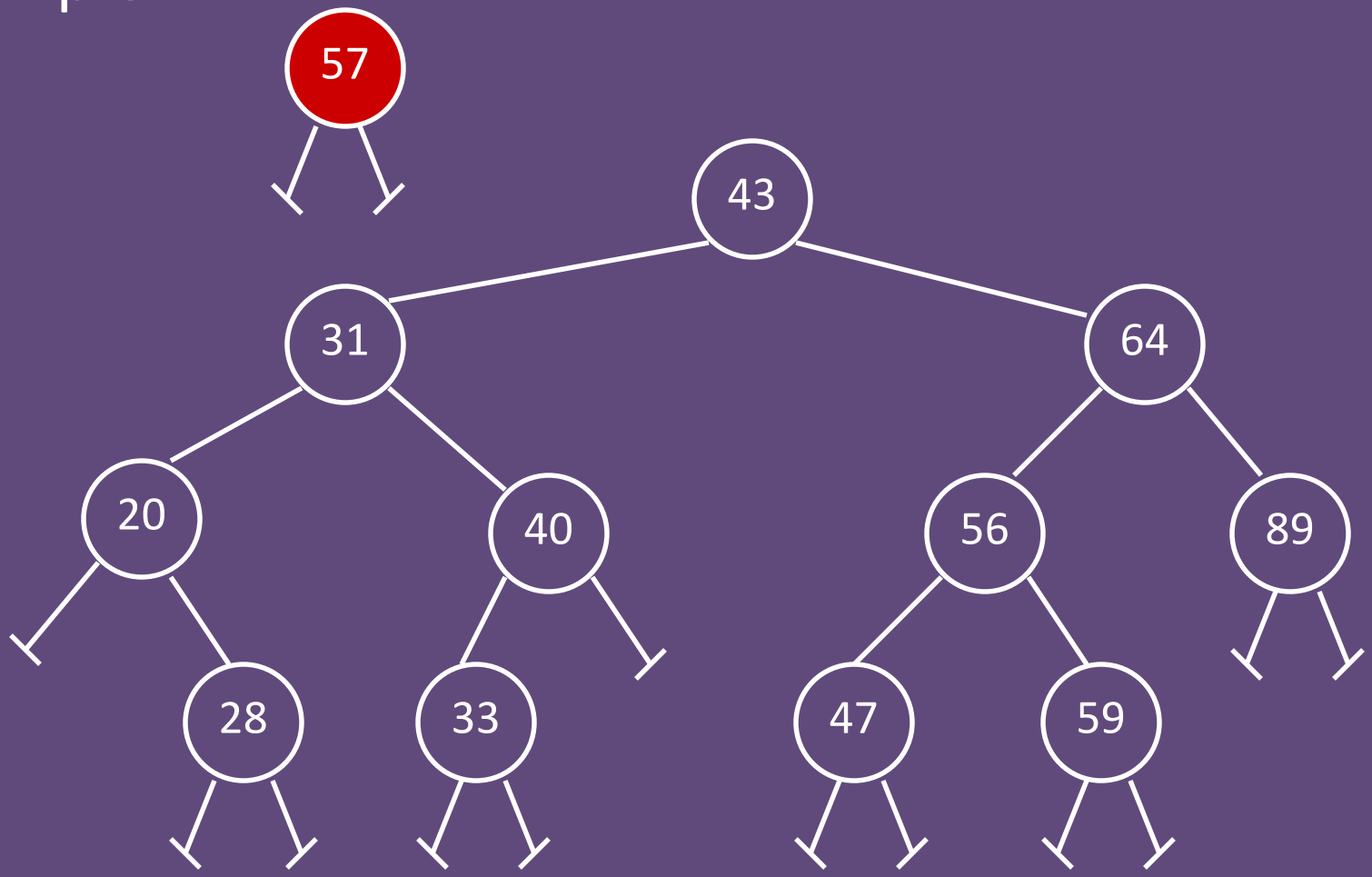
- Case 1: x->right != NULL

  successor (x) = the minimum in x->right

- Case 2: x->right == NULL
  - go up the tree until the current node is a *left child*: *successor* (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element (no successor!)

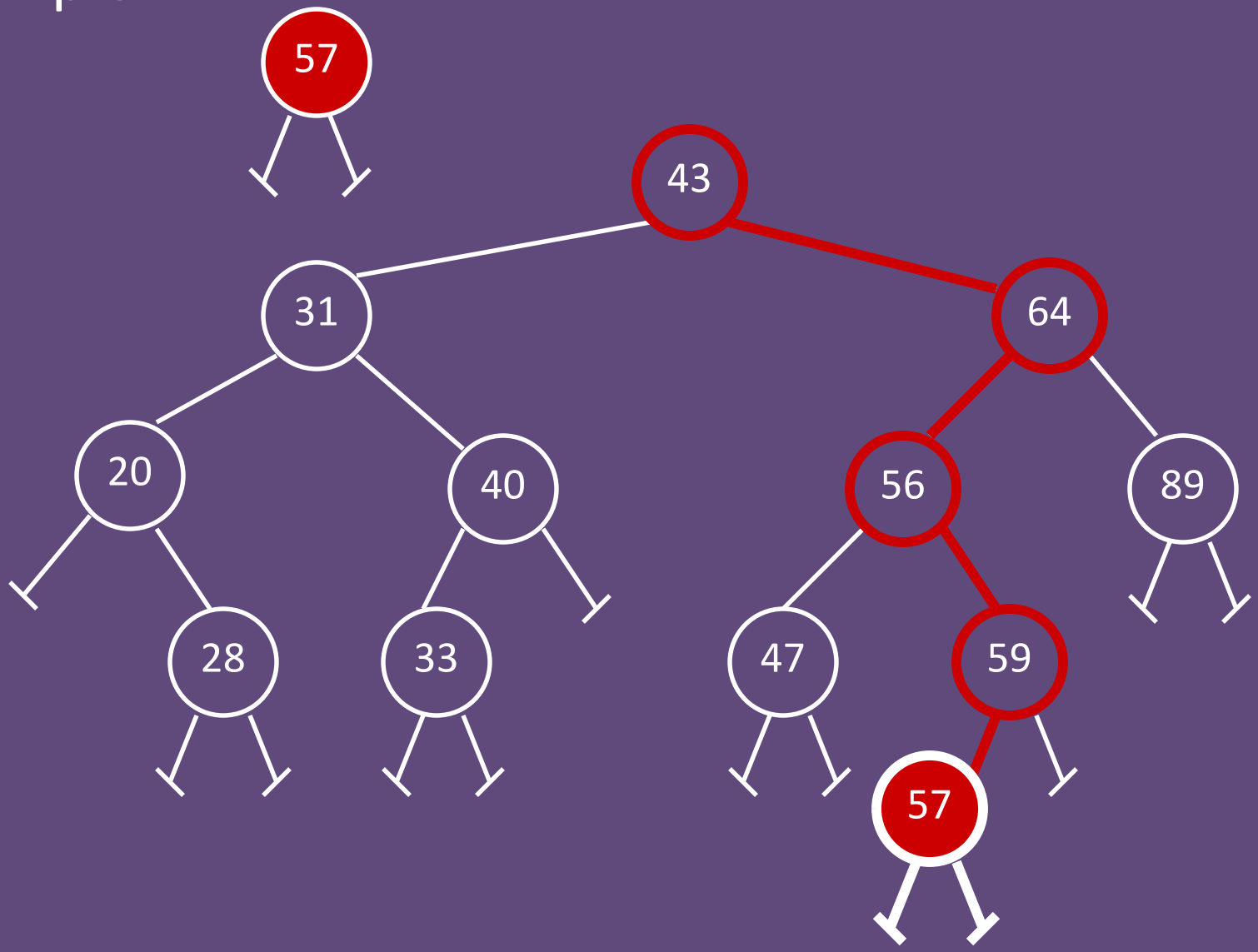
6.3.3. Operations on BST: Insert

Example 1



6.3.3. Operations on BST: Insert

Example 1



6.3.3. Operations on BST: Insert

```
TreeNode* insert(int x, TreeNode* root)
```

- Input parameter:
  - x: Key of the inserted node;
  - root: Pointer points to the current node (initially it is the root of tree).
- Output parameter:
  - Pointer points to the inserted node
- Idea:
  - **If (x < root->key) then** move to the left child of `root`

    else move to the right child of `root`
  - When **root == NULL**, we found the correct position, so create new node and
    - **If (x < y->key) then** insert the new node as y's left child

      else insert it as y's right child

  Beginning at the root, go down the tree and maintain:

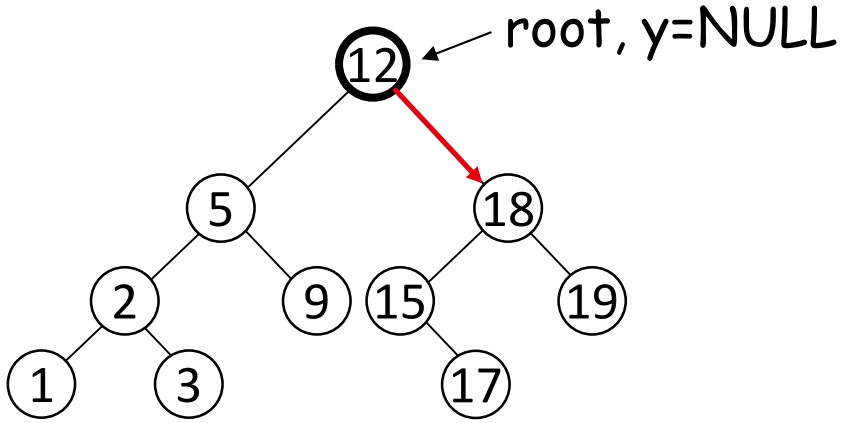
  Pointer root : traces the downward path (current node)

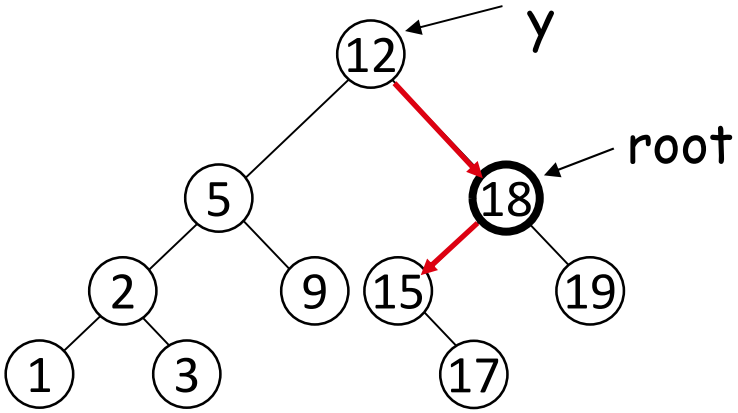
  Pointer y : parent of `root` ("trailing pointer")

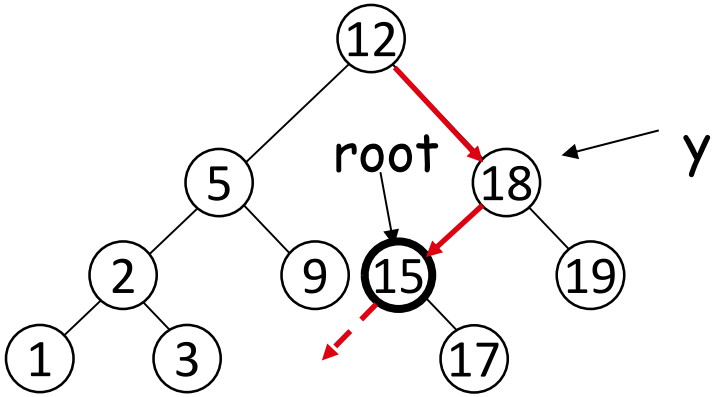
Example:

- **If (x < root->key) then** move to the left child of root
  else move to the right child of root
- When **root == NULL**, we found the correct position, so create new node and
  - **If (x < y->key) then** insert the new node as y's left child
    else insert it as y's right child

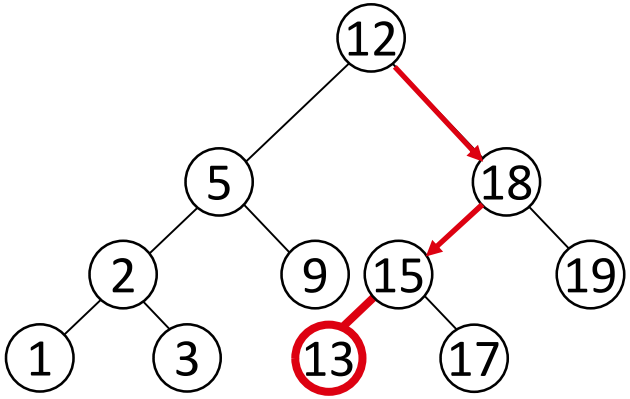
Insert x = 13:







root = NULL
y = 15

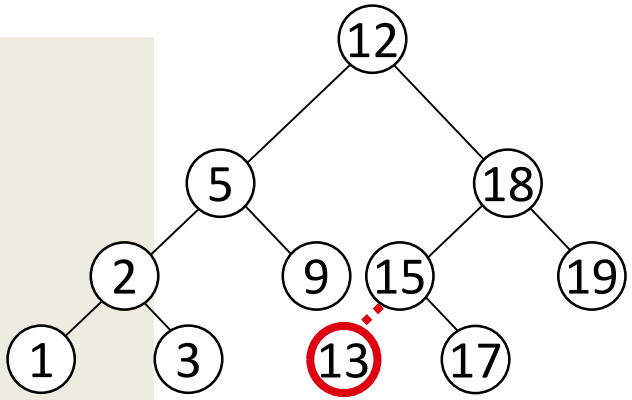


Insertion:

- **If (x < root->key) then** move to the left child of root
  else move to the right child of root
- When **root == NULL**, we found the correct position, so create new node and
  - **If (x < y->key) then** insert the new node as y's left child
    else insert it as y's right child

Alg: **TREE-INSERT(x, root)**

```
1.  y ← NULL
2.  while root ≠ NULL do
3.          y = root
4.          if (x < root->key) then
5.                root = root->left
6.          else root = root->right
7.   newNode = create_node(x)
8.   newNode->parent = y
9.   if (y == NULL) then
10.      root = z   %  Tree BST was empty
11.  else if (x < y->key) then
12.      y->left = newNode
13.  else y->right = newNode
14. return newNode
```



Running time:
$O(h)$

6.3.3. Operations on BST: Insert

```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
      return create_node(x);
    else if (x < root->key)
    {
      TreeNode * newNode = insert(x, root->left);
      root->left = newNode;
      newNode->parent = root; return newNode;
    }
    else if (x > root->key)
    {
      TreeNode * newNode = insert(x, root->right);
      root->right = newNode;
      newNode->parent = root; return newNode;
     }
}
```

Running time:

***O* (h)**, h – height of the tree

6.3.3. Operations on BST: delete a node from BST

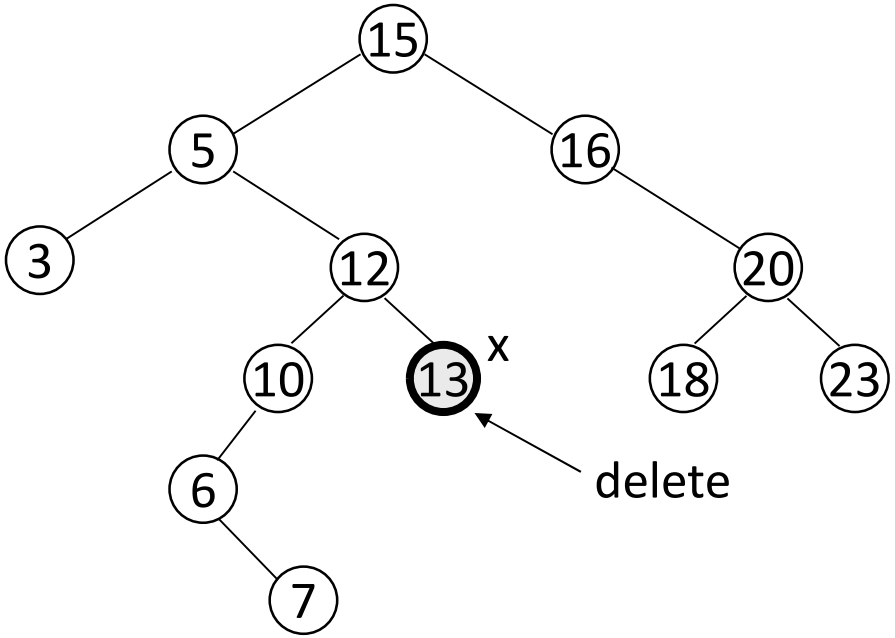
`TreeNode* delete(int x, TreeNode * root)`: delete the node having key = x from the BST with the root pointed by the pointer `root`; function returns the root of the BST after deleting the node

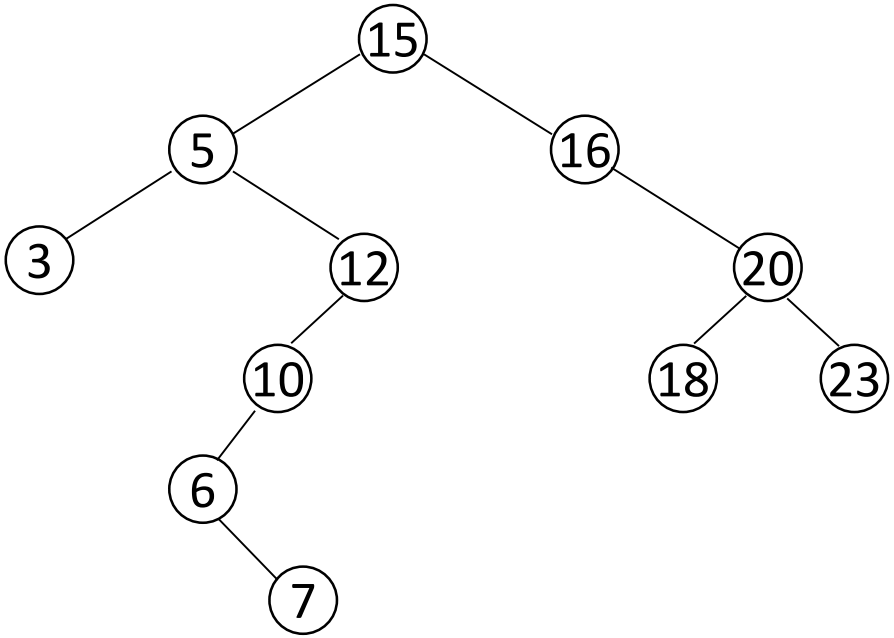
- When deleting a node, the obtained tree must be still BST.
- There are 3 cases need to be considered:
  - Case 1: The deleted node is leaf
  - Case 2: The deleted node only has only one child (left or right)
  - Case 3: The deleted node has both left and right children

6.3.3. Operations on BST: delete a node from BST

Case 1: Deleted node x is leaf (has no children)

Need to do: Delete x by making the parent of x point to NULL.

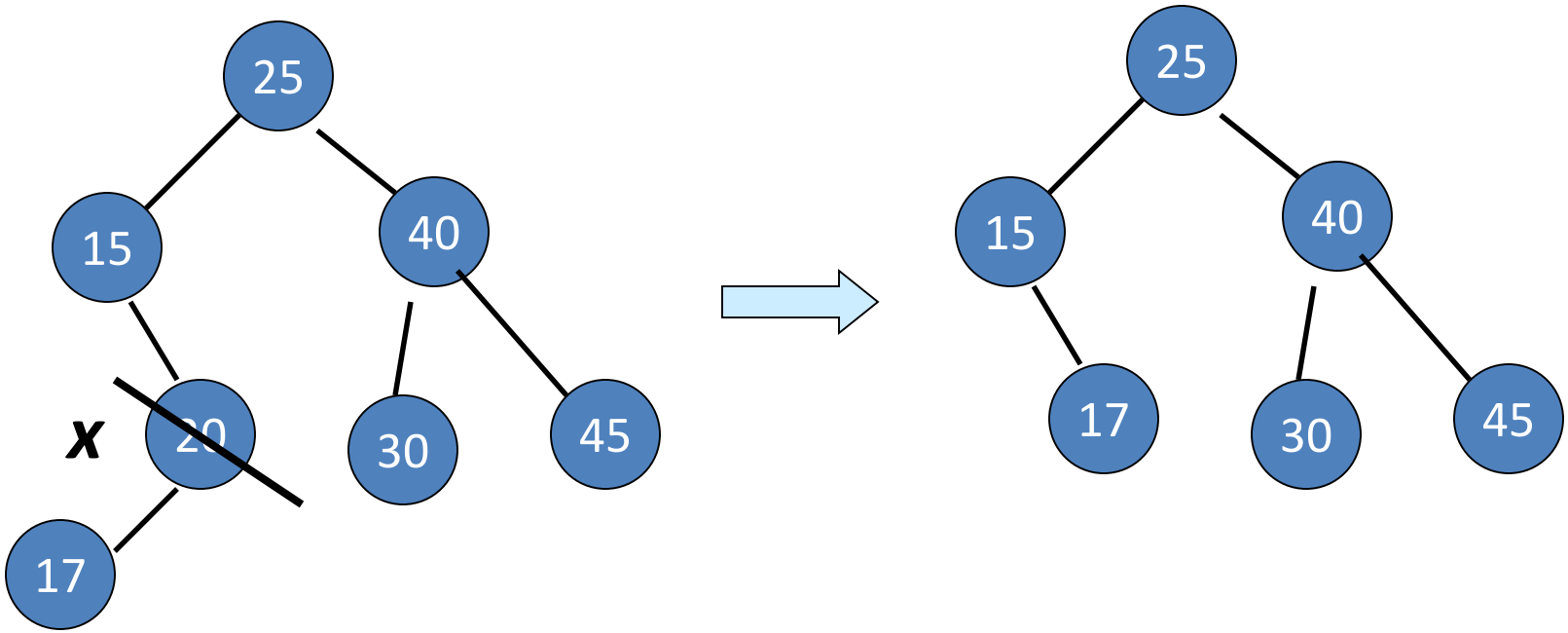




6.3.3. Operations on BST: delete a node from BST

Case 2: deleted node *x* has only one child

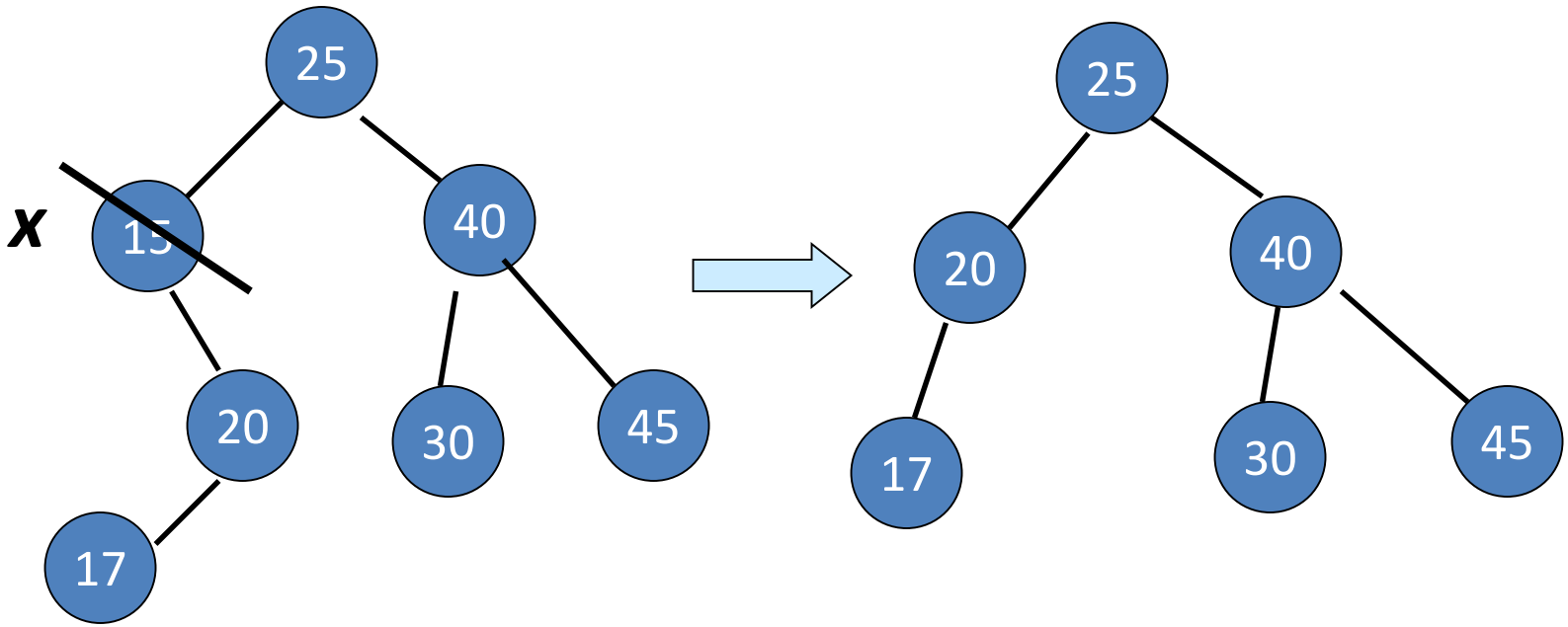
Need to do: Delete *x* by making the parent of *x* point to *x*'s child, instead of to *x* ➔ parent of *x* becomes the parent of *x*'s child.



6.3.3. Operations on BST: delete a node from BST

Case 2: deleted node *x* has only one child

Need to do: Delete *x* by making the parent of *x* point to *x*'s child, instead of to *x* ➔ parent of *x* becomes the parent of *x*'s child.



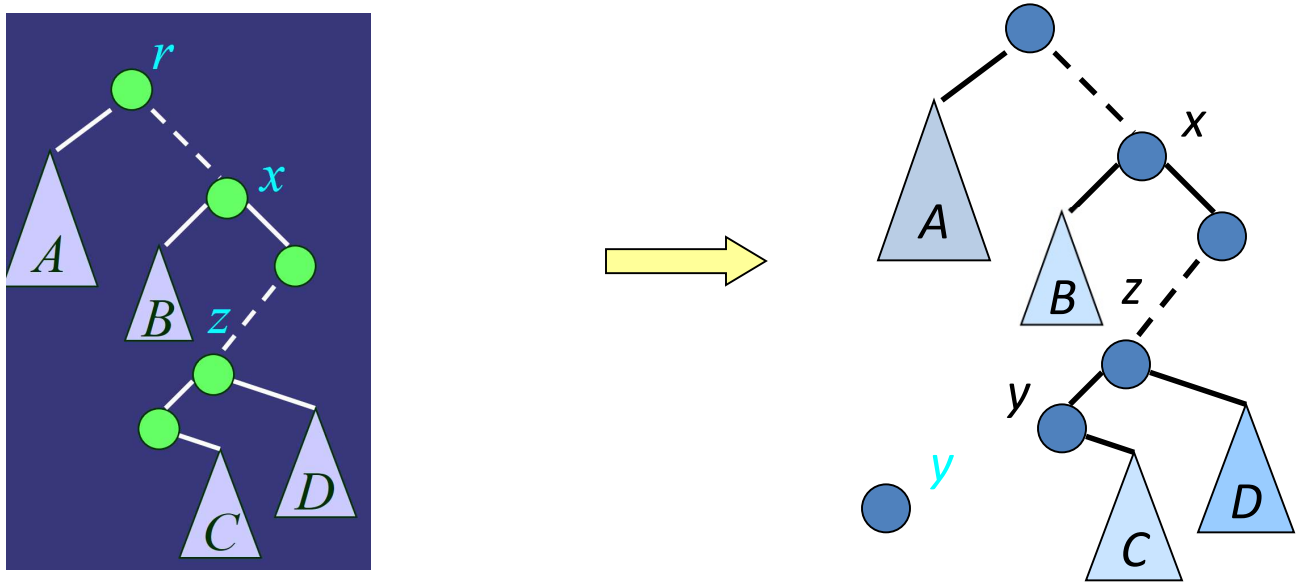
6.3.3. Operations on BST: delete a node from BST

Case 3: deleted node *x* has 2 children

Need to do: 1. Find *x*'s successor (y) to replace *x*, so y is the minimum node in *x*'s right subtree (y has either no children or one right child (but no left child).

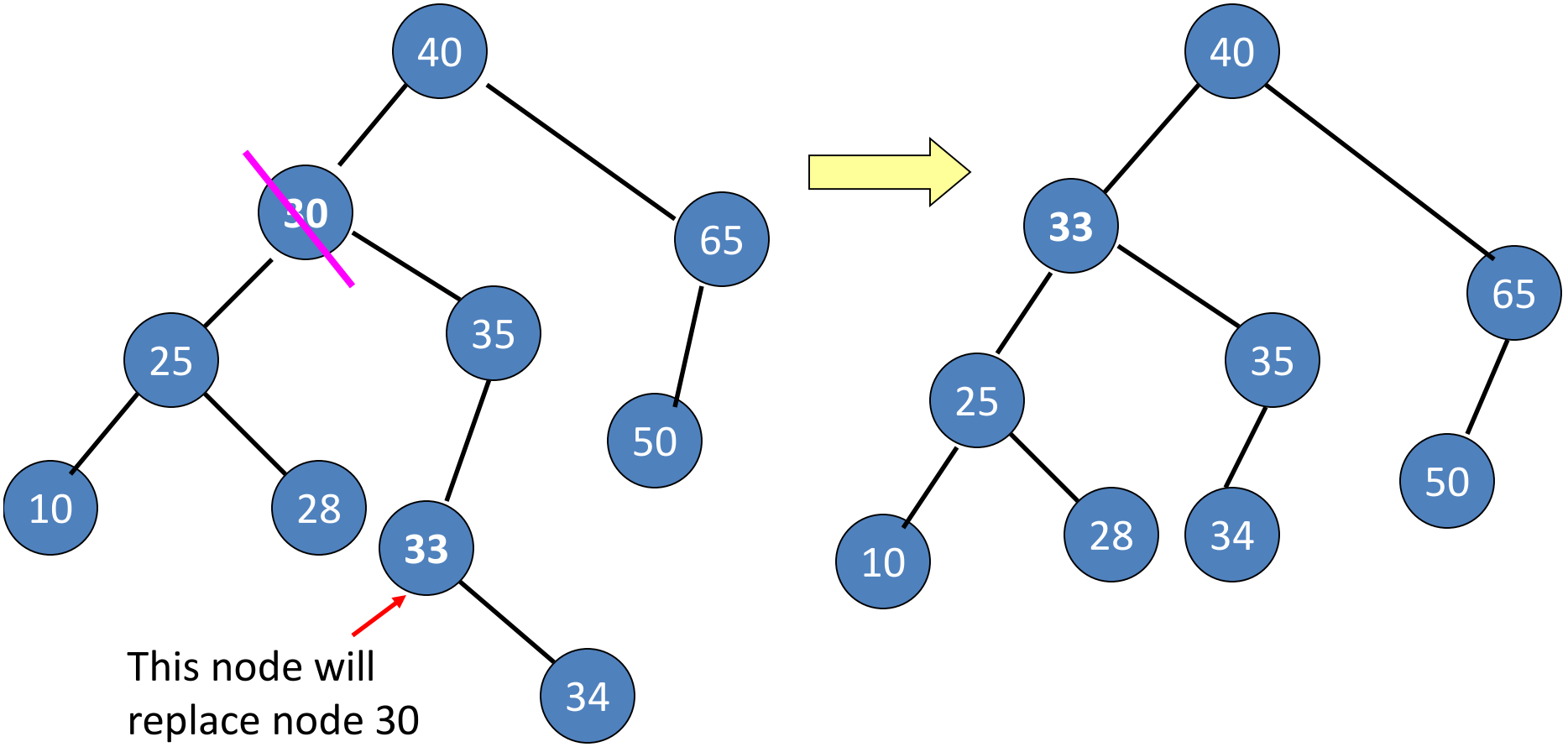
2. Delete *y* from the tree (via Case 1 or 2).

3. Replace *x*'s key and data with *y*'s



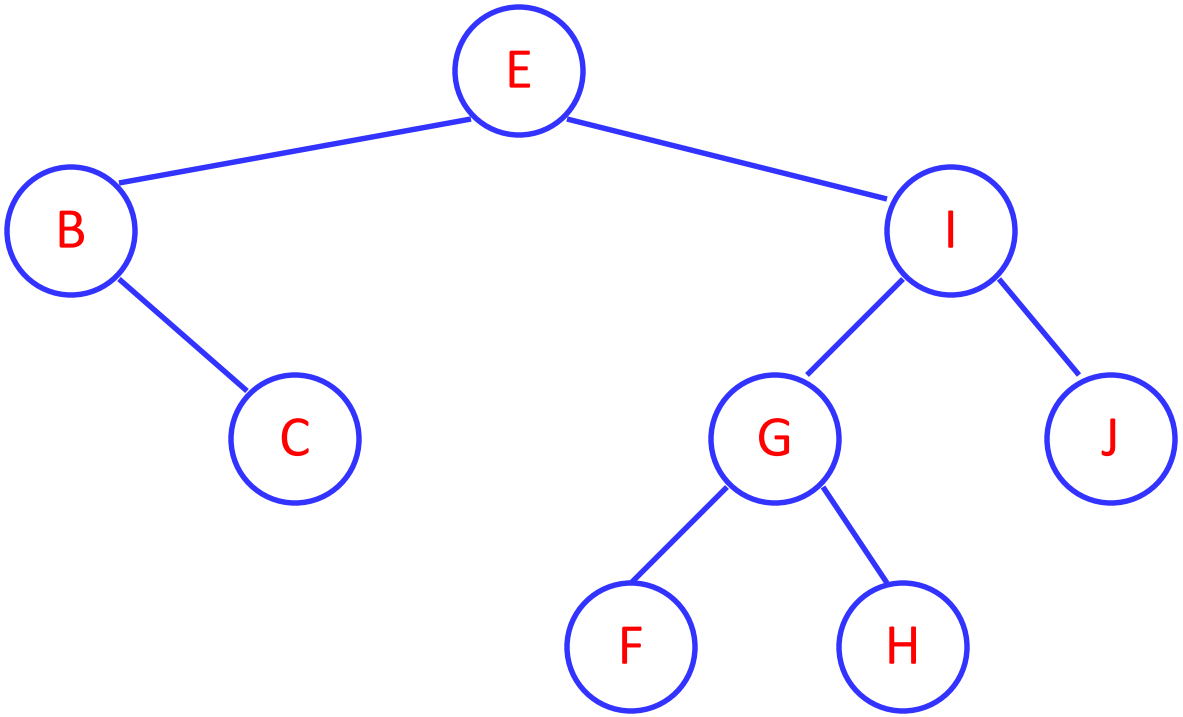
Example 1: Case 4

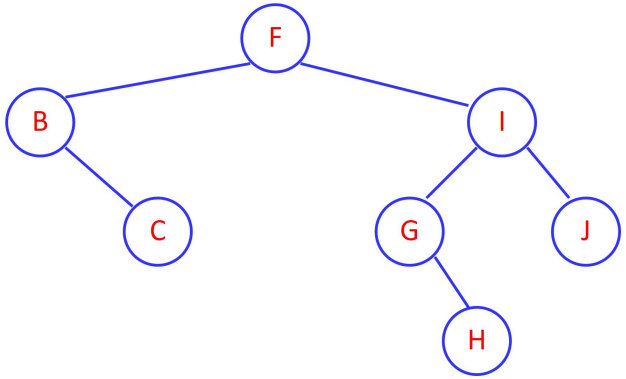
10, 25, 28, **30**, **33**, 34, 35, 40, 50, 65 ⇨ 10, 25, 28, **33**, 34, 35, 40, 50, 65



Example 2: Case 4

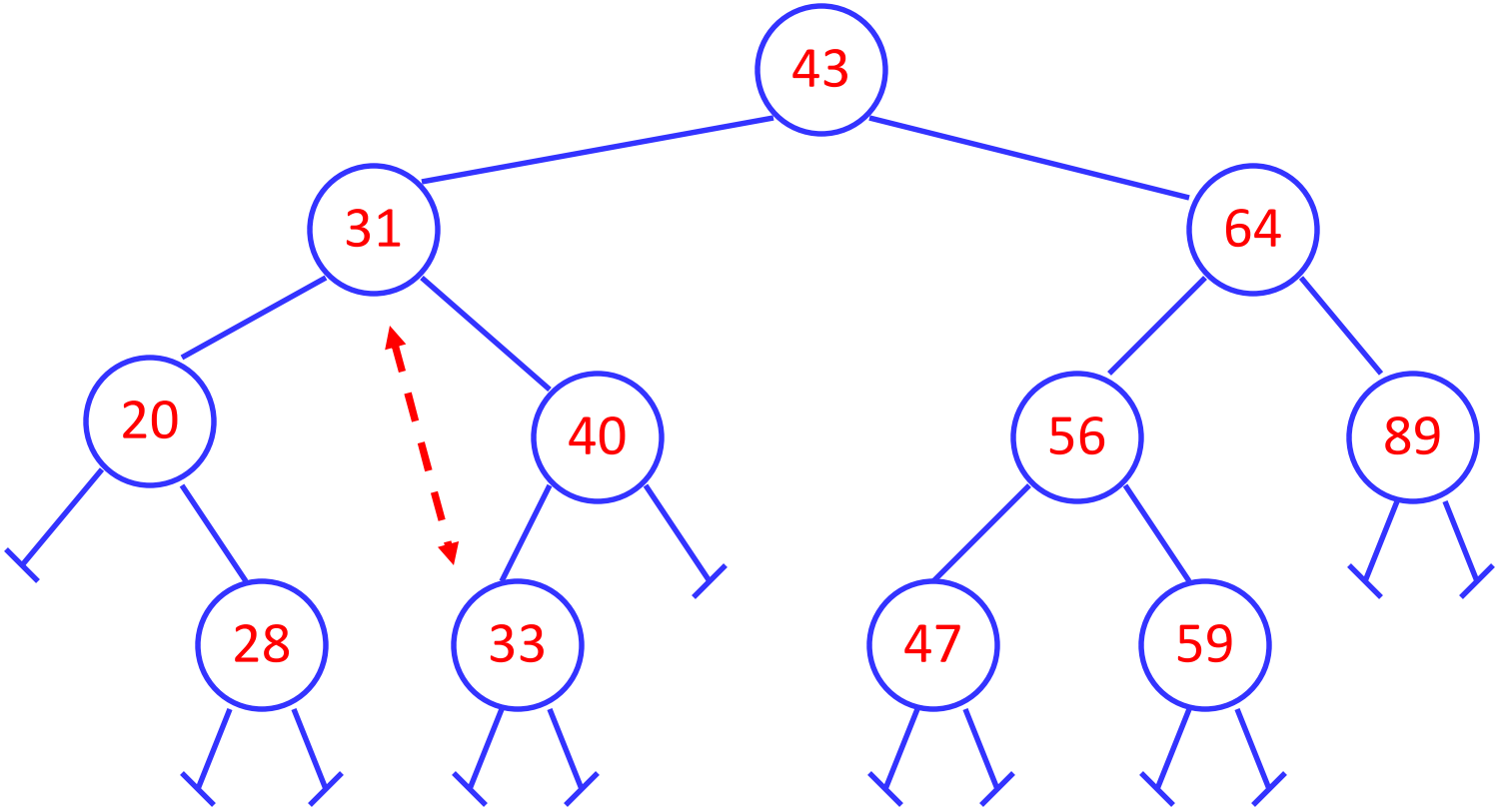
Delete node E





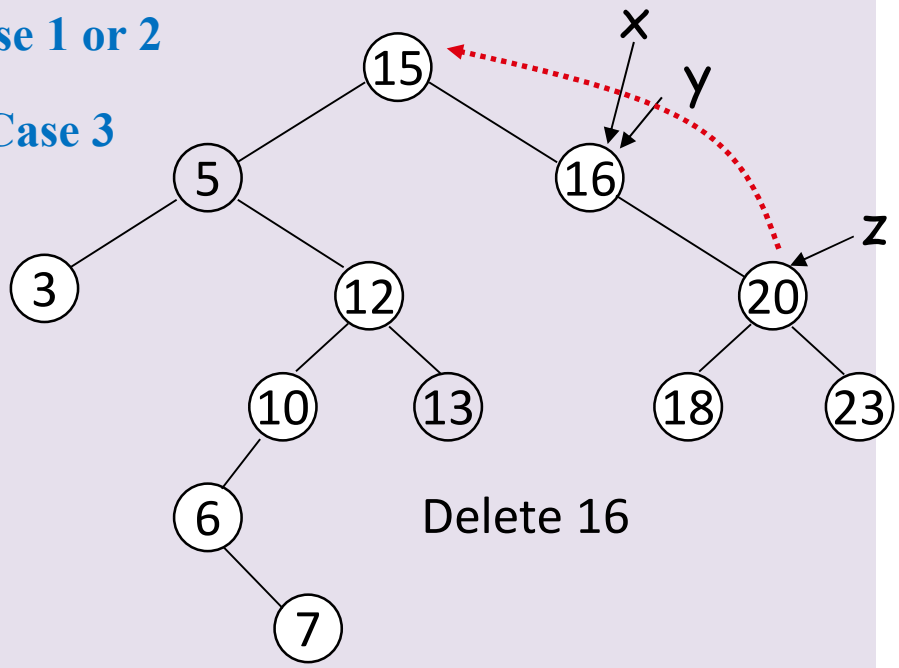
Example 3: Case 4

Delete node 31



TREE-DELETE(root, x): delete node x from the BST

1. **if** (`x->left == NULL or x->right == NULL`) **then**
2. `y = x` //x has at most one child: **Case 1 or 2**
3. **else** y ← Successor(x) //x has 2 children: **Case 3**
   //y will be deleted later
4. **if** (`y->left ≠ NULL`) **then**
5. `z = y->left`
6. **else** `z = y->right`
7. **if** (`z ≠ NULL`) **then**
8. `z->parent = y->parent`



Case 1: Deleted node x is leaf (has no children)

Need to do: Delete x by making the parent of x point to NULL.

Case 2: deleted node *x* has only one child

Need to do: Delete *x* by making the parent of *x* point to *x*'s child, instead of to *x* ➔ parent of *x* becomes the parent of *x*'s child.

Case 3: deleted node *x* has 2 children

Need to do: 1. Find *x*'s successor (y) to replace *x*, so y is the minimum node in *x*'s right subtree (y has either no children or one right child (but no left child).

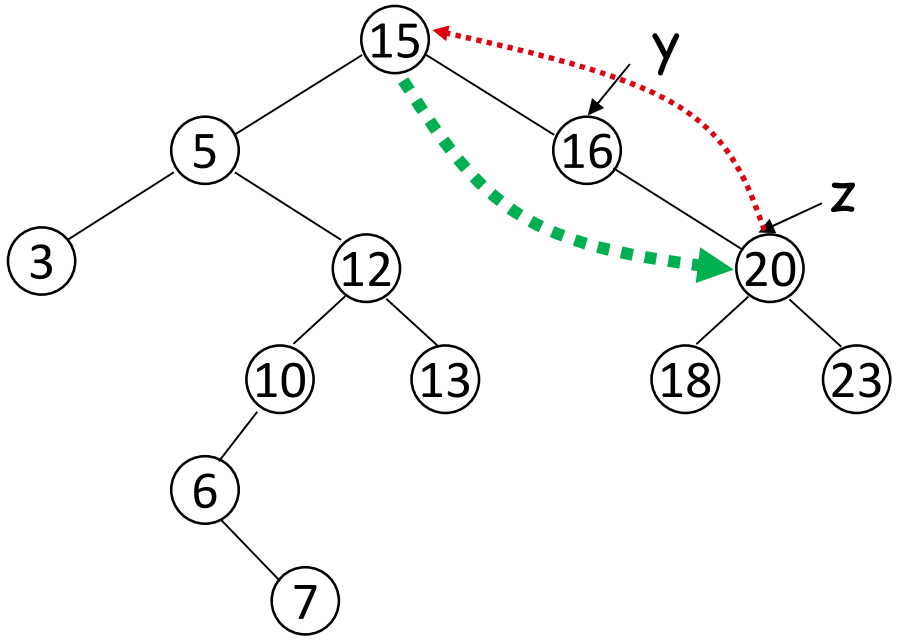
2. Delete *y* from the tree (via Case 1 or 2).

3. Replace *x*'s key and data with *y*'s

TREE-DELETE(root, x): delete node x from the BST

```
9.  if y->parent == NULL
10.    then root = z
11.    else if y == (y->parent)->left
12.            then (y->parent)->left = z
13.            else (y->parent)->right = z
14. if y ≠ x //deal for Case 3:
15.    then x->key = y->key
16. delete y
17. return root
```

Function returns root of the BST after deleting



Running time: $O(h)$ due to TREE-SUCCESSOR operation

Delete a node with *key* = *k*

```
TreeNode* delete(int k, TreeNode *root) {
   TreeNode *y;
   if (root == NULL) cout<<"Not found\n";
   else if (k < root->key)  root->left = delete(k, root->left); /* đi bên trái */
   else if (k > root->key)  root->right = delete(k, root->right); /* đi bên phải */
   else /* tìm được phần tử cần xoá, và trỏ bởi root*/
   if (root->left != NULL && root->right != NULL)
   {
      /* Case 3: phần tử thế chỗ là phần tử min ở cây con phải */
       y = find_min(root->right);
       root->key = y->key;
       root->right = delete(root->key, root->right);
   }
   else /*Case 1,2: có 1 con hoặc không có con */
   {
       y = root;
       if (root->left== NULL) /* chỉ có con phải hoặc không có con */
             root = root->right;
       else if (root->right == NULL) /* chỉ có con trái */
             root = root->left;
       delete y;
    }
    return root;
}
```

Case 1: Deleted node x is leaf (has no children)

Need to do: Delete x by making the parent of x point to NULL.

Case 2: deleted node *x* has only one child

Need to do: Delete *x* by making the parent of *x* point to *x*'s child, instead of to *x* ➔ parent of *x* becomes the parent of *x*'s child.

Case 3: deleted node *x* has 2 children

Need to do: 1. Find *x*'s successor (y) to replace *x*, so y is the minimum node in *x*'s right subtree (y has either no children or one right child (but no left child).

2. Delete *y* from the tree (via Case 1 or 2).

3. Replace *x*'s key and data with *y*'s

Sorting by using BST (Sắp xếp nhờ sử dụng BST)

To sort a list of elements, we can do as follows:

- Insert elements into a BST.
- Traverse BST in preorder to get the elements sorted.

Sorting (Sắp xếp)

Sort this array by using BST

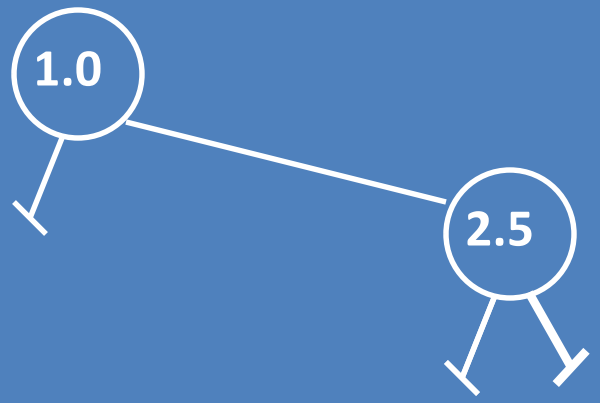
1.0	**2.5**	0.5	0.7	3.6	2.1

Sorting

Sort this array by using BST

1.0	**2.5**	0.5	0.7	3.6	2.1

Sorting



Sort this array by using BST

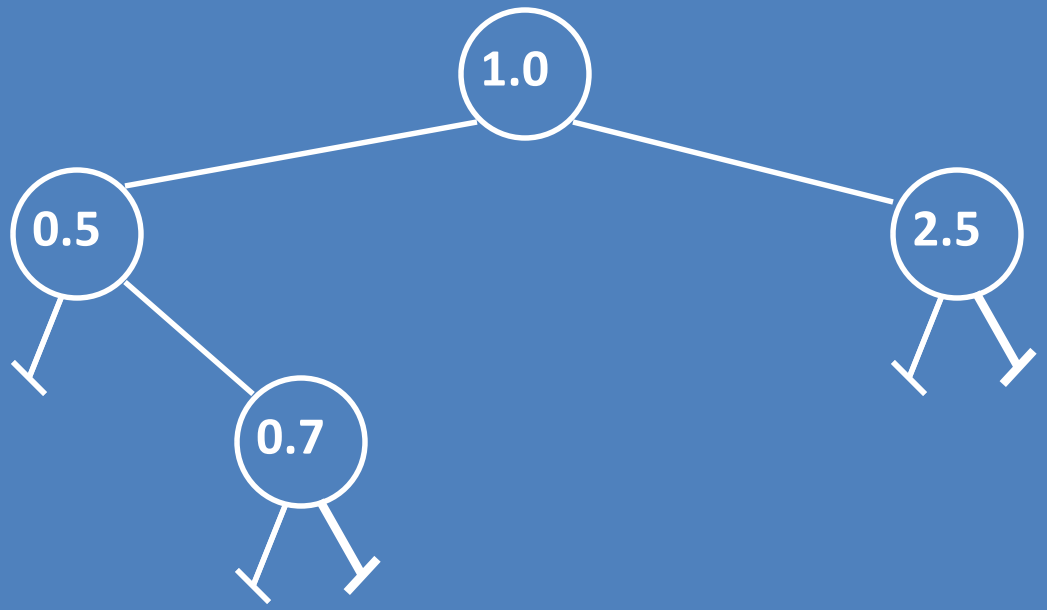
1.0	**2.5**	0.5	0.7	3.6	2.1

Sorting

Sort this array by using BST

1.0	**2.5**	0.5	0.7	3.6	2.1

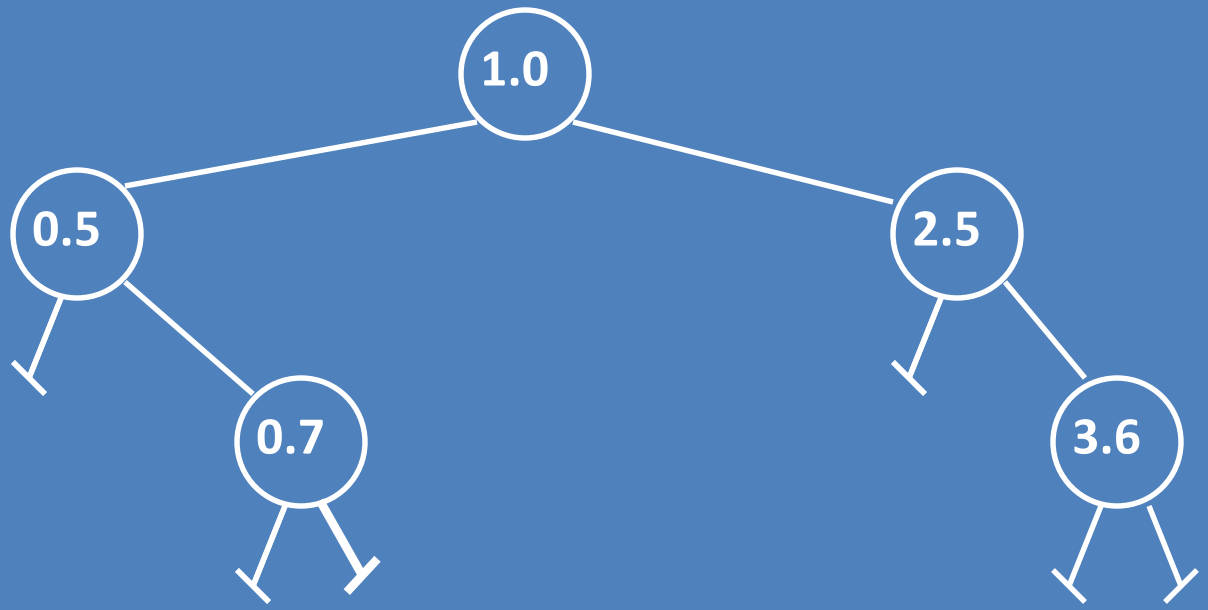
Sorting



Sort this array by using BST

1.0	**2.5**	0.5	0.7	3.6	2.1

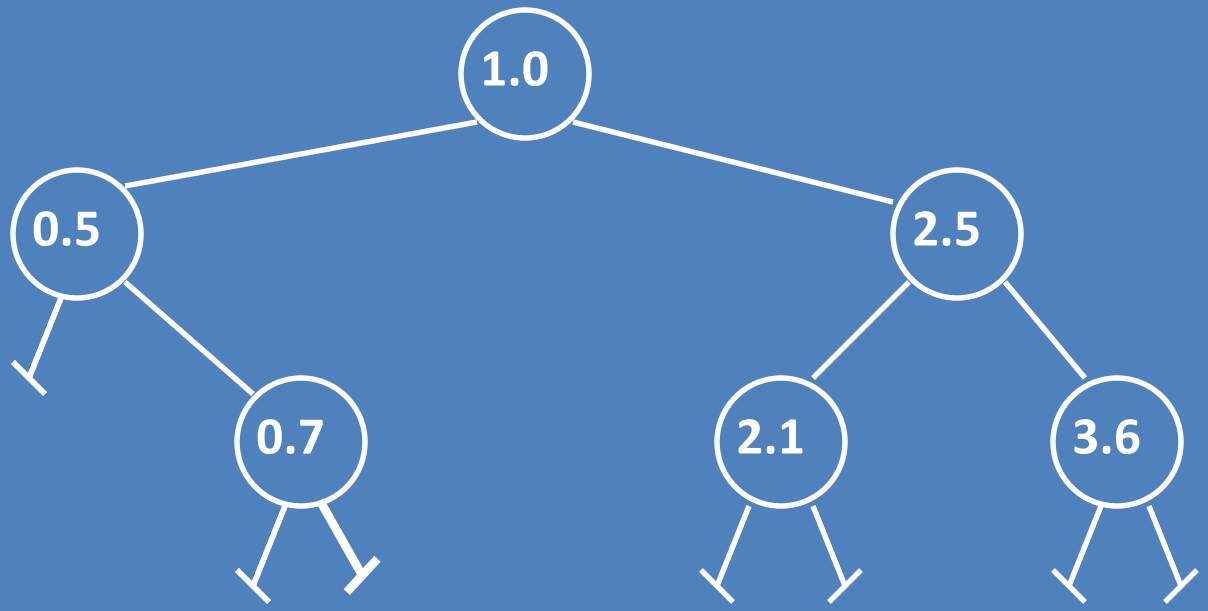
Sorting



Sort this array by using BST

1.0	**2.5**	0.5	0.7	3.6	2.1

Sorting

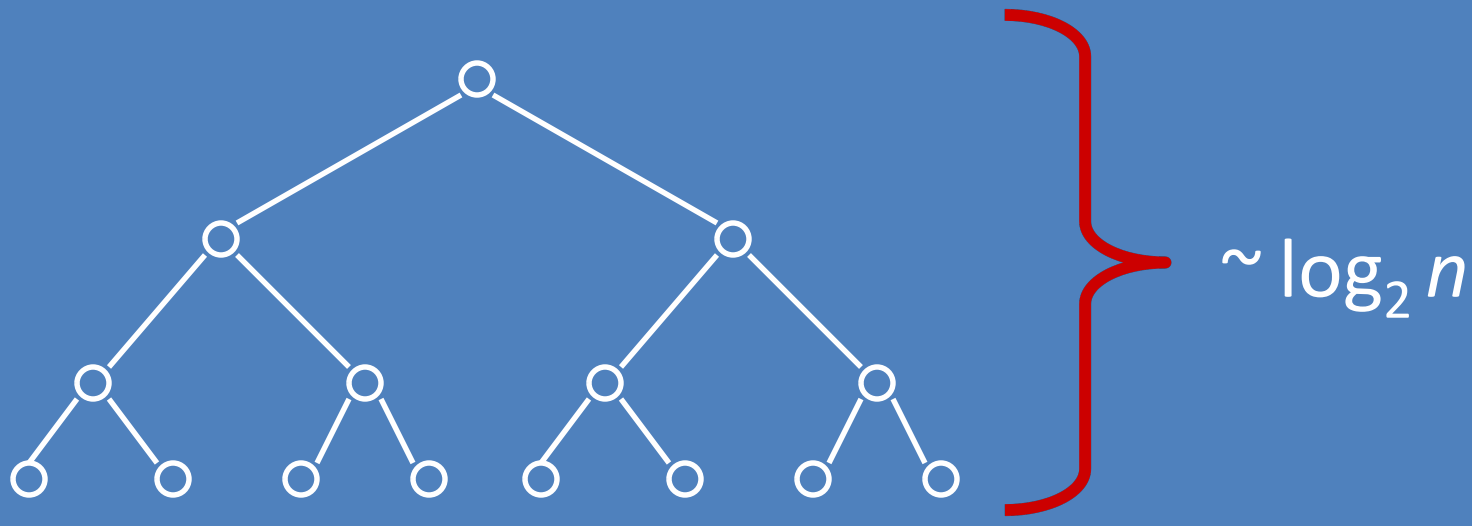


Sort this array by using BST

1.0	**2.5**	0.5	0.7	3.6	2.1

Sorting: Analysis the complexity

- Average case: $O(n \log_2 n)$



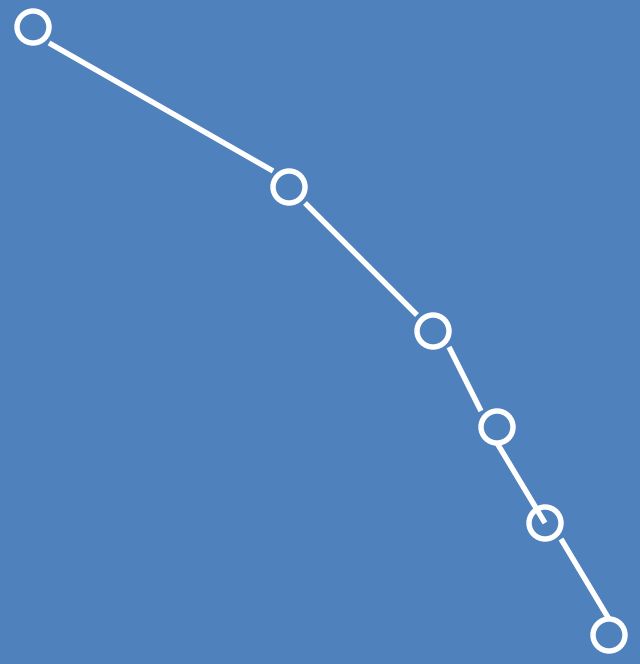
- Inserting $(i+1)$th element takes about $\log_2(i)$ comparisons

Sorting: Analysis the complexity

- Worst case: $O(n^2)$

Insert array: 1, 3, 7, 9, 11, 15

into BST



- Inserting (i+1)th element takes i comparisons

Average computation time for operations on BST (Độ phức tạp trung bình của các thao tác với BST)

- We can show that the average height of BST is

$$h = O(\log_2 n)$$

- Thus, the average complexity for operations on BST are:

Insertion	.....	$O(\log_2 n)$
Deletion	.....	$O(\log_2 n)$
Find Min	.....	$O(\log_2 n)$
Find Max	.....	$O(\log_2 n)$
BST Sort	.....	$O(n\log_2 n)$

Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESSOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$

- These operations are fast if the height of the tree is small

Theorem

The **expected height** of a randomly built **binary search tree** on **n** distinct keys is **O(*lg*n)**