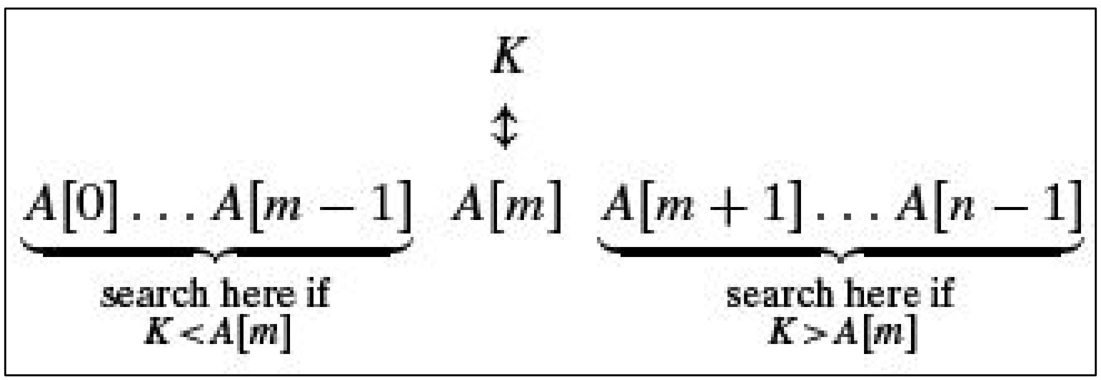
Module 7:
Binary Search

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Binary Search

- Binary search is a Θ(log n), highly efficient search algorithm, in a sorted array.
- It works by comparing a search key K with the array's middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$.
- The number of comparisons to search for a key in an array of size n is $C(n) = C(n/2) + 1$, for $n > 1$. $C(n) = 1$ for $n = 1$.



Binary Search

Example

Search Key
K = 70

l	r	m
l=0	r=12	m=6
l=7	r=12	m=9
l=7	r=8	m=7

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	91	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

ALGORITHM *BinarySearch*$(A[0..n-1], K)$
//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
// a search key K
//Output: An index of the array's element that is equal to K
// or -1 if there is no such element
$l \leftarrow 0$; $r \leftarrow n-1$
while $l \le r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **return** m
  else if $K < A[m]$ $r \leftarrow m-1$
  else $l \leftarrow m+1$
return -1

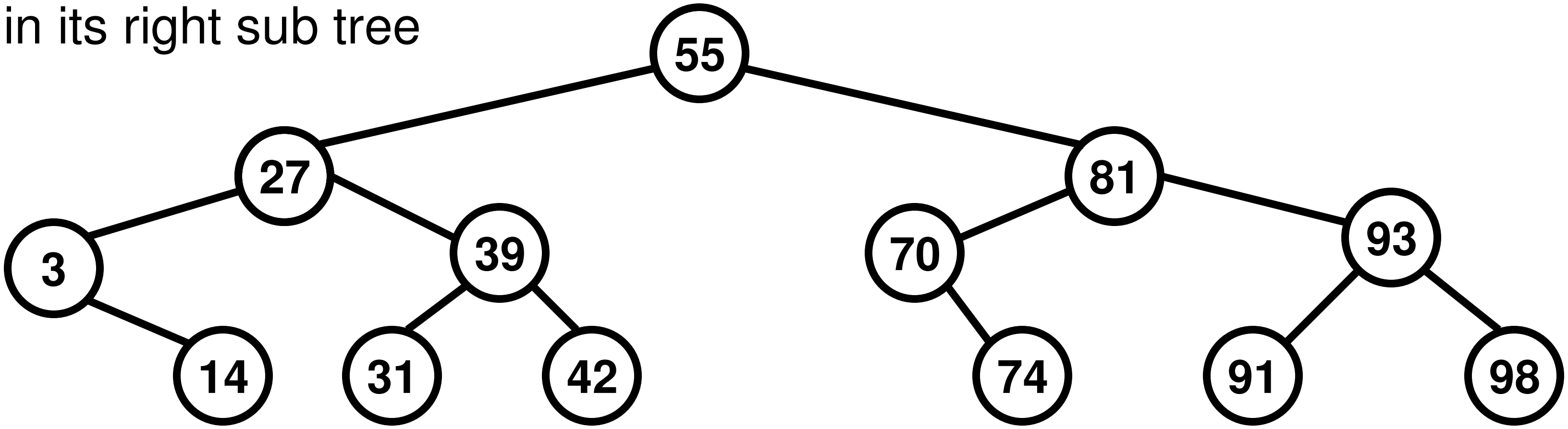
Note that the "search space" reduces by half in each iteration.
Hence, the # iterations is proportional to log(n), where 'n' is the # elements

The algorithm is run until the left index is less than or equal to the right index The search key should be found by then.

The moment the left index becomes greater than the right index, we stop and declare the search key is not there.

Binary Search Tree (BST)

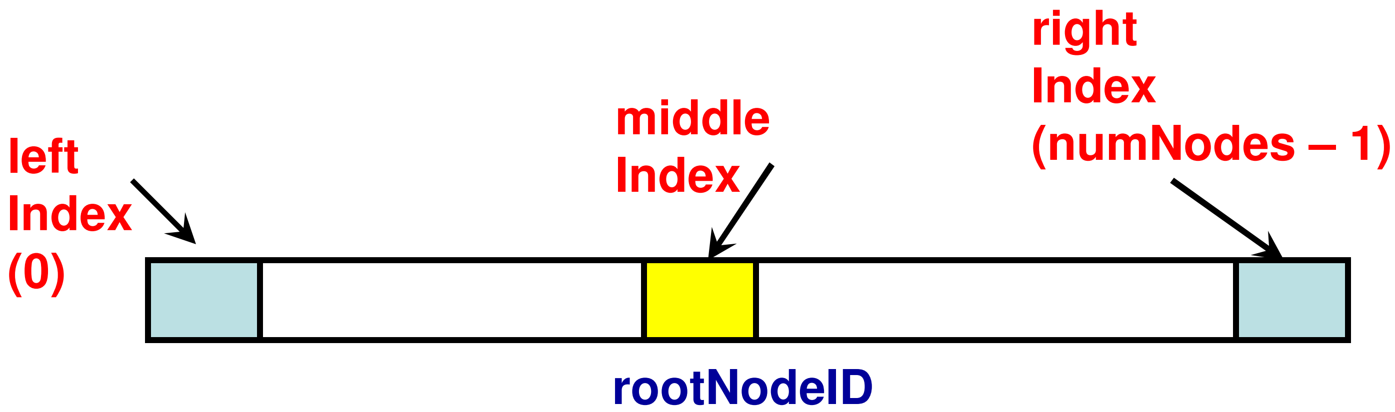
- A binary search tree is a binary tree in which the value for an internal node is greater than or equal to the values of the nodes in its left sub tree and is lower than or equal to the values of the nodes in its right sub tree



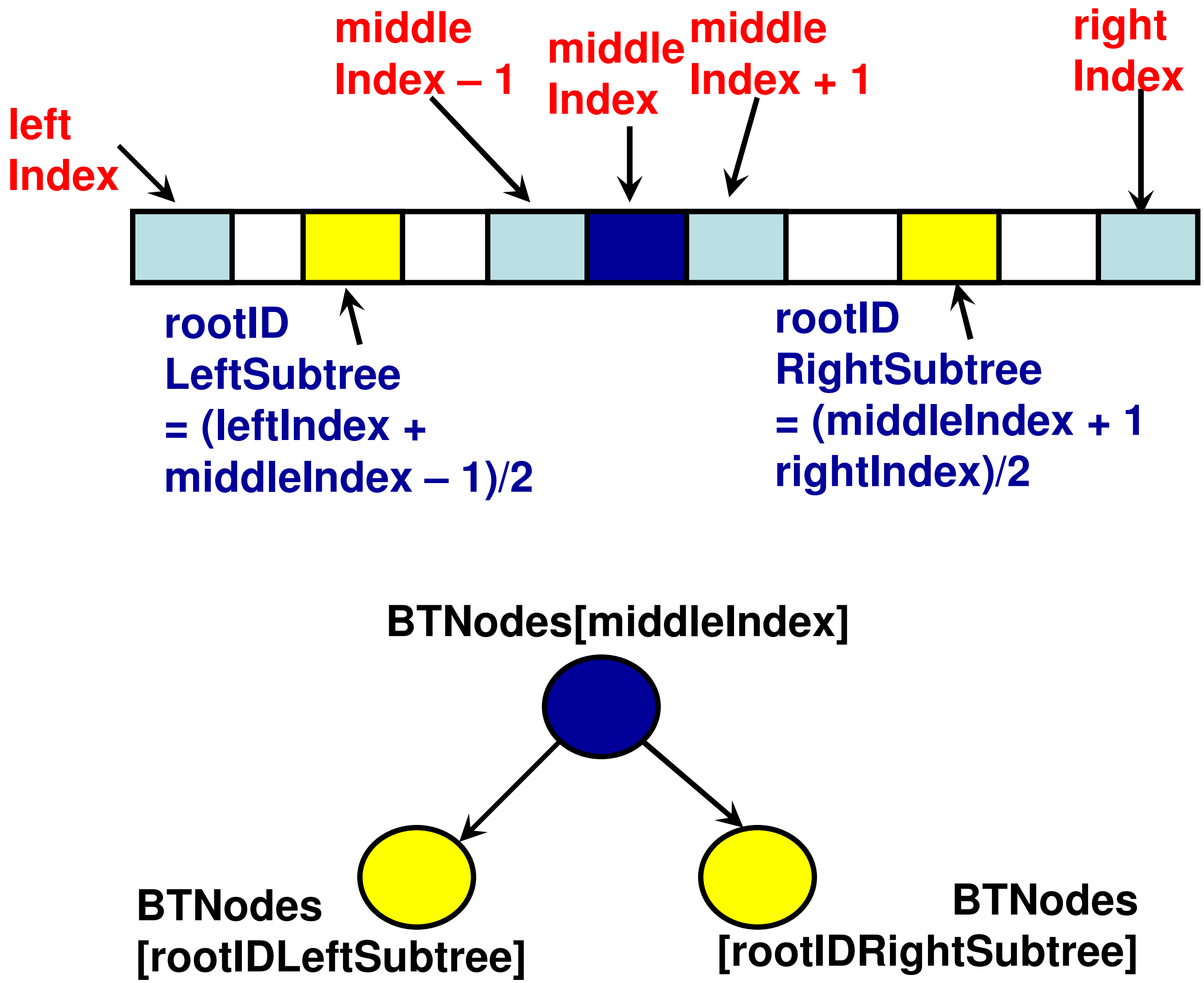
- Both hash tables and BSTs are data structures to implement a Dictionary ADT
- A hash table is an unordered collection of data items as a hash table could be constructed for any arbitrary array and the search could be conducted on a specific linked list to which the search element indexes (hash index) into.
- A BST is an ordered collection of data items (satisfying the property mentioned above). The number of comparisons it takes for a successful search or an unsuccessful search is bounded by the height of the binary search tree, which is proportional to log(# nodes).

Algorithm to Construct a BST

```
Begin BST Construction(Array A, numNodes)
    int leftIndex = 0
    int rightIndex = numNodes – 1
    int middleIndex = (leftIndex  + rightIndex) / 2

    rootNodeID = middleIndex
    BSTree[middleIndex].setData(A[middleIndex])

    ChainNodes(A, middleIndex, leftIndex, rightIndex)
End BST Construction
```



Logic behind the ChainNodes Function



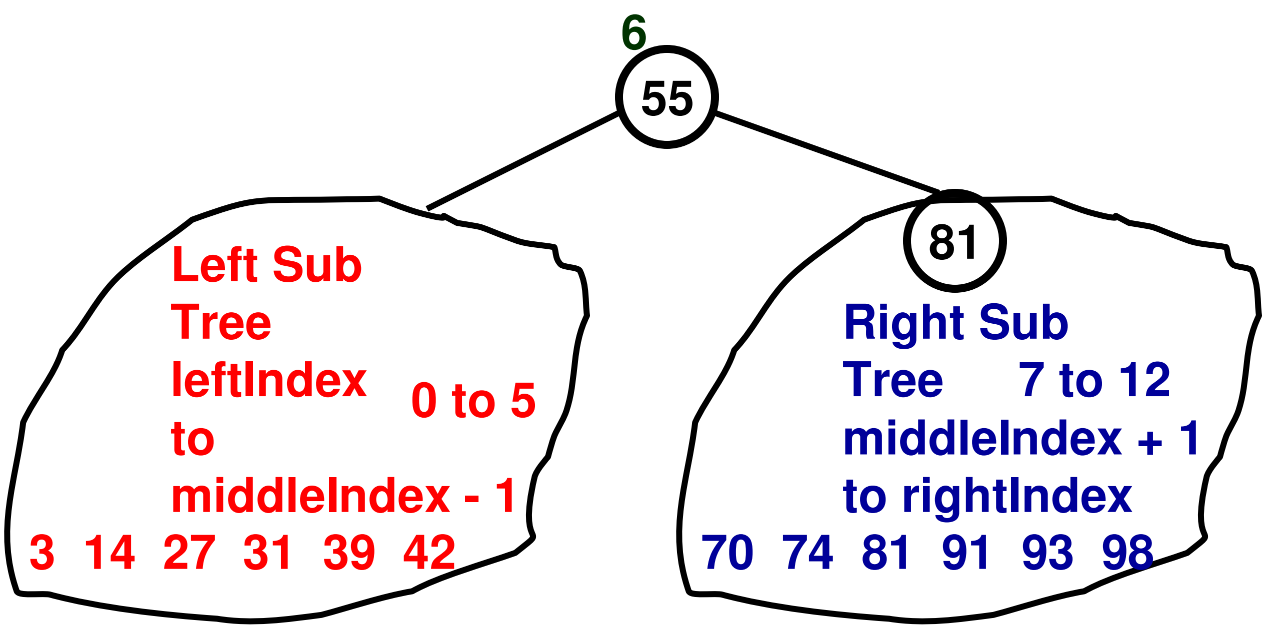
Pseudo Code: ChainNodes Function

```
ChainNodes(A, middleIndex, leftIndex, rightIndex)
    if (leftIndex < middleIndex) then  // a left sub tree exists for the node
                                       // at middleIndex
        rootIDLeftSubtree = (leftIndex + middleIndex – 1) / 2
        BTNodes[rootIDLeftSubtree].setData(A[rootIDLeftSubtree])
        setLeftLink(middleIndex, rootIDLeftSubtree)
        ChainNodes(A, rootIDLeftSubtree, leftIndex, middleIndex – 1)
    end if

    if (rightIndex > middleIndex) then // a right sub tree exists for the node
                                       // at middleIndex
        rootIDRightSubtree = (middleIndex + 1 + rightIndex) / 2
        BTNodes[rootIDRightSubtree].setData(A[rootIDRightSubtree])
        setRightLink(middleIndex, rootIDRightSubtree)
        ChainNodes(A, rootIDRightSubtree, middleIndex + 1, rightIndex)
    end if
```

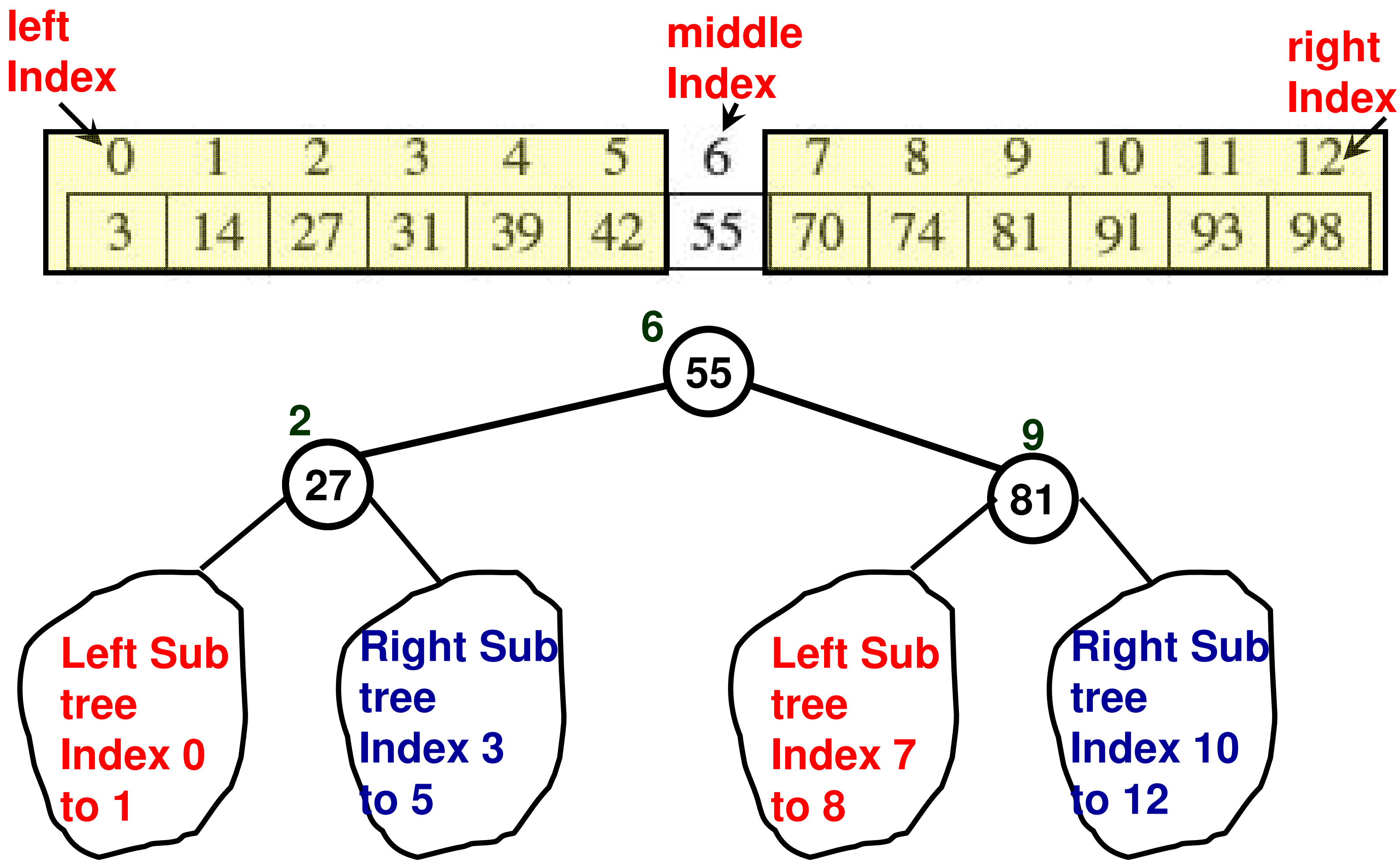
Example 1: Construction of BST

left Index → 0, middle Index → 6, right Index → 12

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98

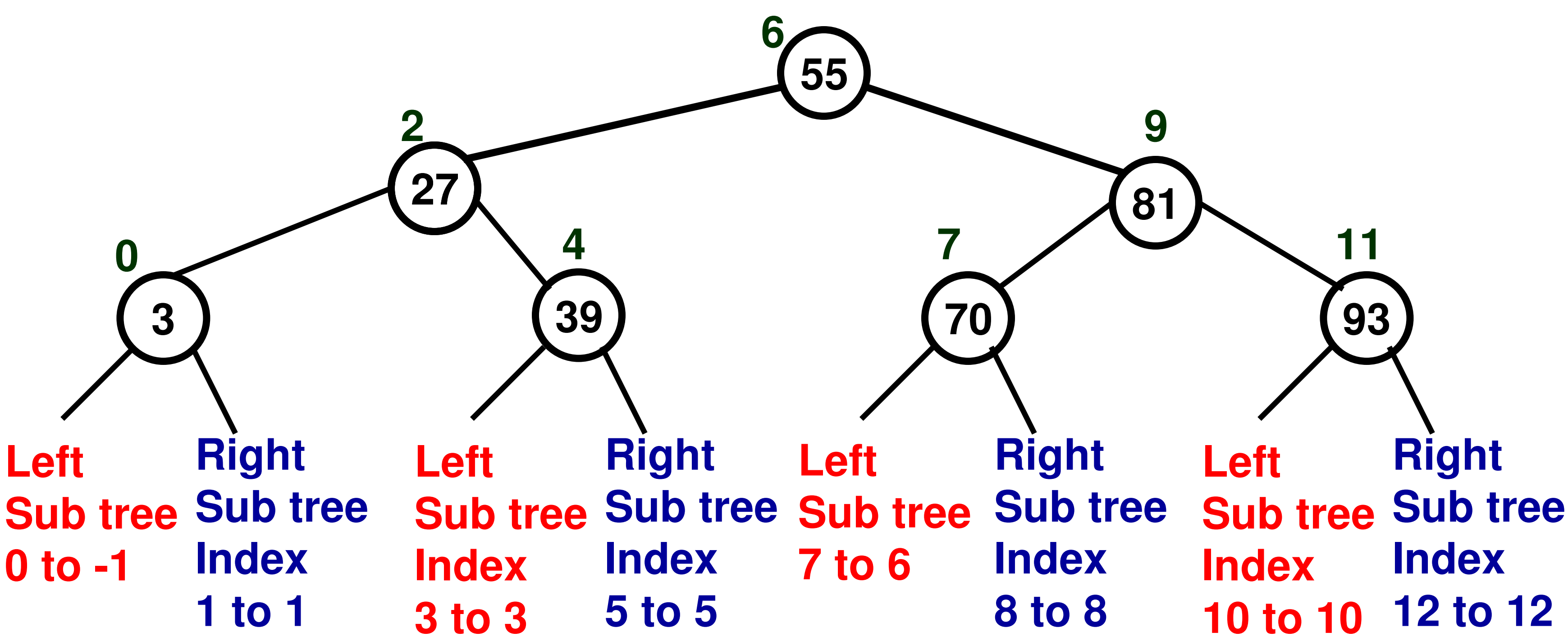


Example 1: Construction of BST



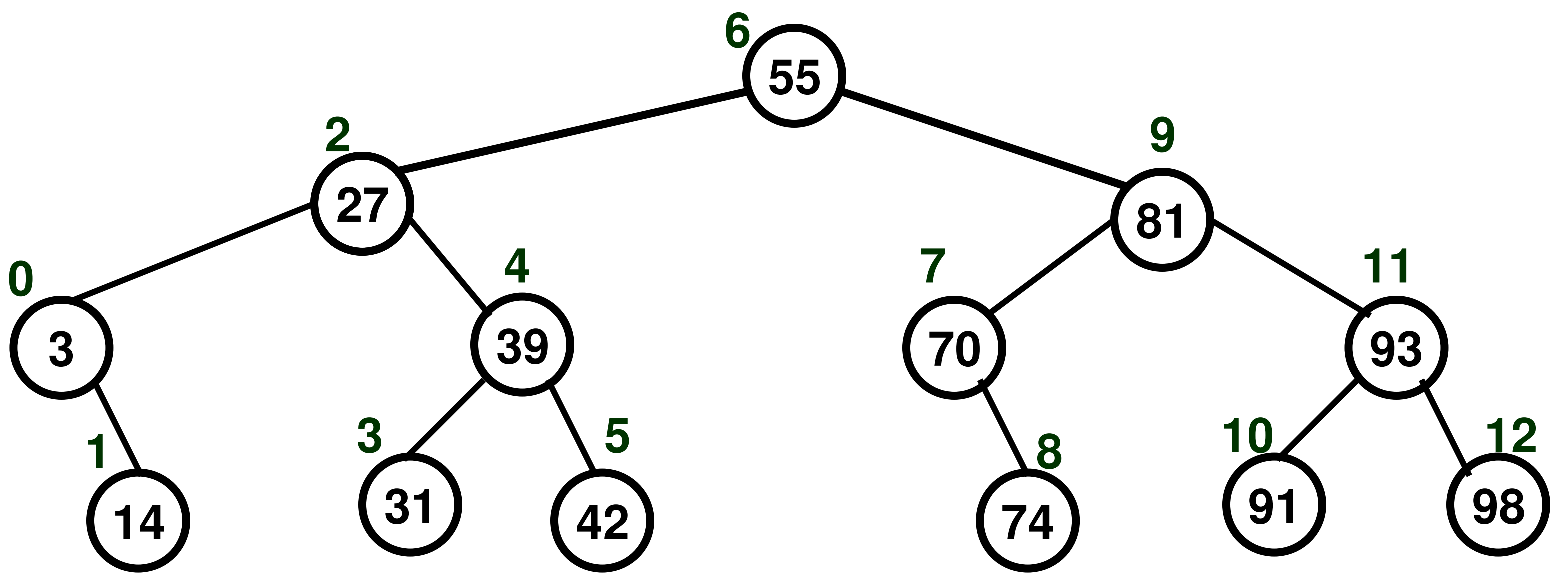
Example 1: Construction of BST

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98



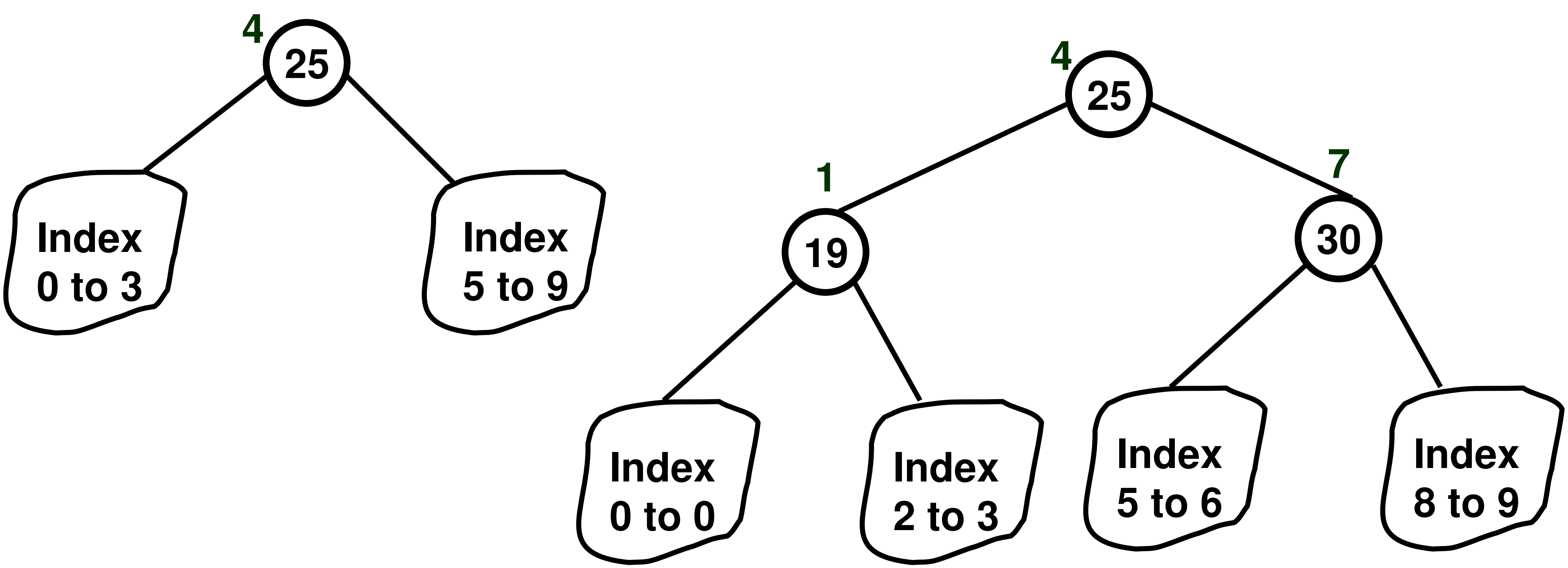
Example 1: Construction of BST

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98



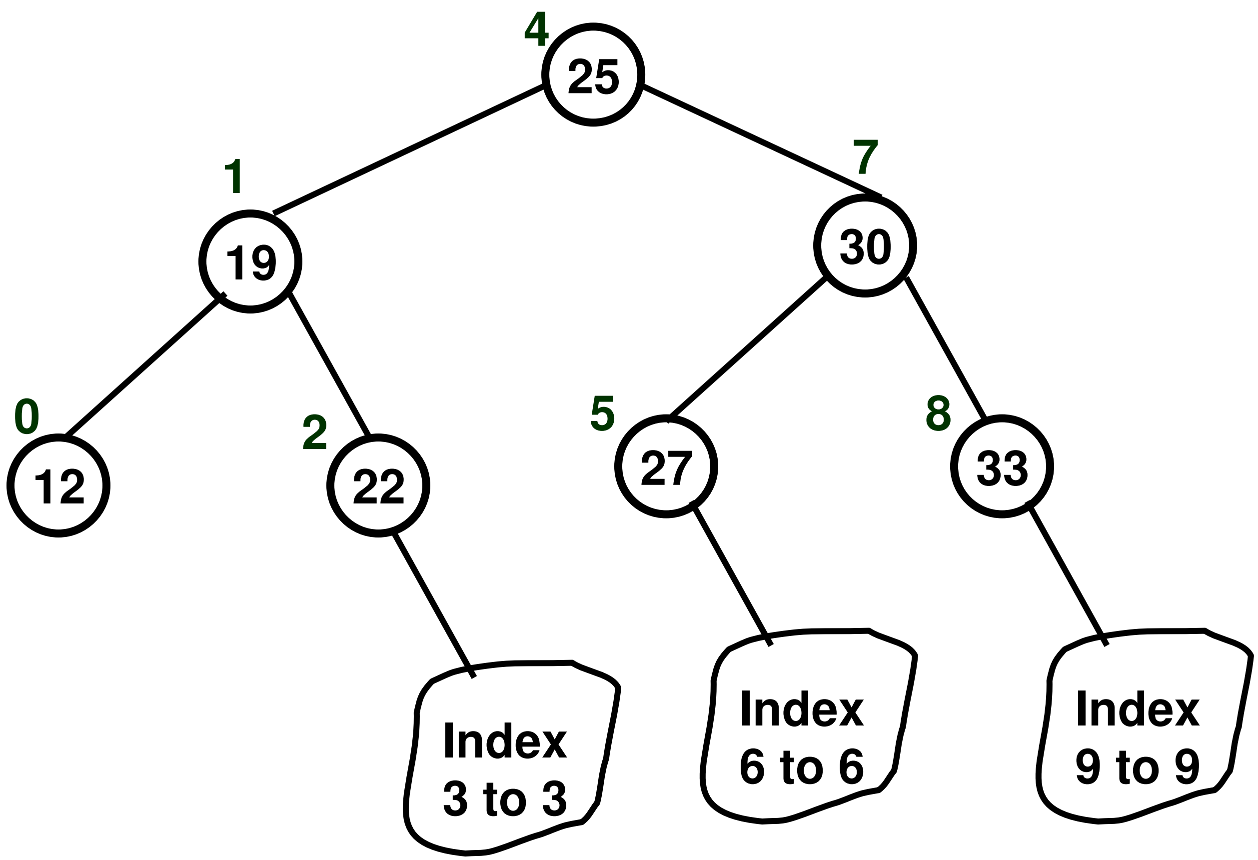
Example 2: Construction of BST

0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37



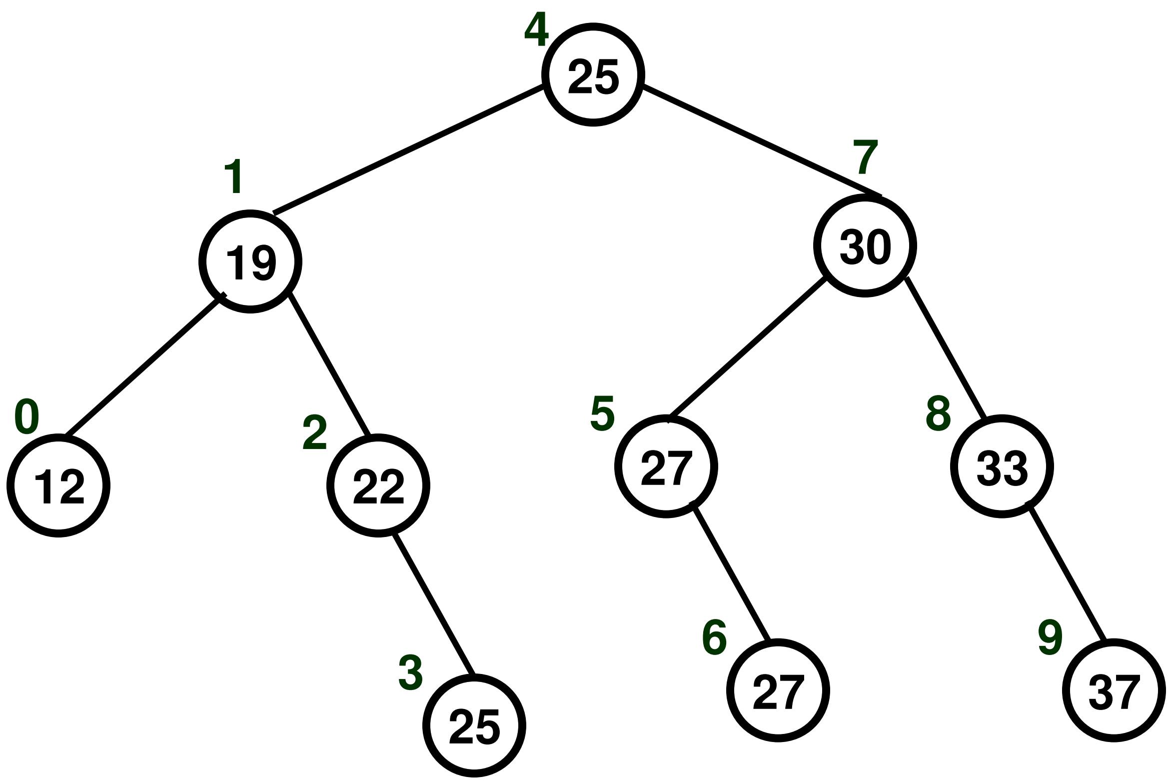
Example 2: Construction of BST

0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37



Example 2: Construction of BST

0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37



Binary Search Tree (BST) Construction

- We will create a class called BinarySearchTree that will be similar to the BinaryTree class created in the other module as much as possible.
- Differences
  - There will be a member variable called root node id (the root node id of a BST need not be 0)
  - We will add two member functions called constructBSTree( ) that will get the input array of sorted integers from the user, determines the root node and calls the ChainNodes(…) function, which is implemented in a recursive fashion.
    - The ChainNodes(…) function will link a node to its left child node and right child node, if any exists, and will call itself to do the same on its left sub tree and right sub tree.

BST Implementation (C++: Code 7.1)

BTNode
int nodeid
int data
int levelNum
BTNode* leftChildPtr
BTNode* rightChildPtr

BinarySearchTree
int numNodes
BTNode* arrayOfBTNodes
int rootNodeID

```
BinarySearchTree(int n){
        numNodes = n;
        arrayOfBTNodes = new BTNode[numNodes];

        for (int index = 0; index < numNodes; index++){

                arrayOfBTNodes[index].setNodeId(index);
                arrayOfBTNodes[index].setLeftChildPtr(0);
                arrayOfBTNodes[index].setRightChildPtr(0);
                arrayOfBTNodes[index].setLevelNum(-1);

        }
}
```

```
void setLeftLink(int upstreamNodeID, int downstreamNodeID){
    arrayOfBTNodes[upstreamNodeID].setLeftChildPtr(&arrayOfBTNodes[downstreamNodeID]);
}
```

```
void setRightLink(int upstreamNodeID, int downstreamNodeID){
    arrayOfBTNodes[upstreamNodeID].setRightChildPtr(&arrayOfBTNodes[downstreamNodeID]);
}
```

constructBSTree Function (Code 7.1)

C++

```
void constructBSTree(int* array){
        int leftIndex = 0;
        int rightIndex = numNodes-1;
        int middleIndex = (leftIndex + rightIndex)/2;

        rootNodeID = middleIndex;
        arrayOfBTNodes[middleIndex].setData(array[middleIndex]);

        ChainNodes(array, middleIndex, leftIndex, rightIndex);
}
```

Assumes the array is already sorted

ChainNodes Function (C++ Code 7.1)

```
void ChainNodes(int* array, int middleIndex, int leftIndex, int rightIndex){

        if (leftIndex < middleIndex){
                int rootIDLeftSubtree = (leftIndex + middleIndex-1)/2;
                setLeftLink(middleIndex, rootIDLeftSubtree);

                arrayOfBTNodes[rootIDLeftSubtree].setData(array[rootIDLeftSubtree]);
                ChainNodes(array, rootIDLeftSubtree, leftIndex, middleIndex-1);
        }

        if (rightIndex > middleIndex){
                int rootIDRightSubtree = (rightIndex + middleIndex + 1)/2;
                setRightLink(middleIndex, rootIDRightSubtree);

                arrayOfBTNodes[rootIDRightSubtree].setData(array[rootIDRightSubtree]);
                ChainNodes(array, rootIDRightSubtree, middleIndex+1, rightIndex);
        }
}
```

constructBSTree Function (called without the data array)

```
void constructBSTree(){

        int leftIndex = 0;
        int rightIndex = numNodes-1;
        int middleIndex = (leftIndex + rightIndex)/2;

        rootNodeID = middleIndex;

        ChainNodes(middleIndex, leftIndex, rightIndex);

}
```

ChainNodes Function (called without the data array)

```
void ChainNodes(int middleIndex, int leftIndex, int rightIndex){

        if (leftIndex < middleIndex){
                int rootIDLeftSubtree = (leftIndex + middleIndex-1)/2;
                setLeftLink(middleIndex, rootIDLeftSubtree);
                ChainNodes(rootIDLeftSubtree, leftIndex, middleIndex-1);
        }

        if (rightIndex > middleIndex){
                int rootIDRightSubtree = (rightIndex + middleIndex + 1)/2;
                setRightLink(middleIndex, rootIDRightSubtree);
                ChainNodes(rootIDRightSubtree, middleIndex+1, rightIndex);
        }

}
```

Selection Sort: Example

Given Array: 12 5 1 4 18 9 7 15

	Index: 0	1	2	3	4	5	6	7
	Data: 12	5	1	4	18	9	7	15
Iteration 0	1	5	12	4	18	9	7	15
Iteration 1	1	4	12	5	18	9	7	15
Iteration 2	1	4	5	12	18	9	7	15
Iteration 3	1	4	5	7	18	9	12	15
Iteration 4	1	4	5	7	9	18	12	15
Iteration 5	1	4	5	7	9	12	18	15
Iteration 6	1	4	5	7	9	12	15	18

Code 7.2 Selection Sort (C++)

```
void selectionSort(int *array, int arraySize){

    for (int iterationNum = 0; iterationNum < arraySize-1; iterationNum++){

        int minIndex = iterationNum;

        for (int j = iterationNum+1; j < arraySize; j++){

            if (array[j] < array[minIndex])
                minIndex = j;

        }

        // swap array[minIndex] with array[iterationNum]
        int temp = array[minIndex];
        array[minIndex] = array[iterationNum];
        array[iterationNum] = temp;
    }
}
```

```cpp
int numElements;
cout << "Enter the number of elements: ";
cin >> numElements;

int *array = new int[numNodes];

int maxValue;
cout << "Enter the maximum value for an element: ";
cin >> maxValue;

srand(time(NULL));

cout << "array generated: ";

for (int index = 0; index < numNodes; index++){
        array[index] = rand() % maxValue;
        cout << array[index] << " ";
}

cout << endl;

selectionSort(array, numNodes);

BinarySearchTree bsTree(numElements);
bsTree.constructBSTree(array);
```

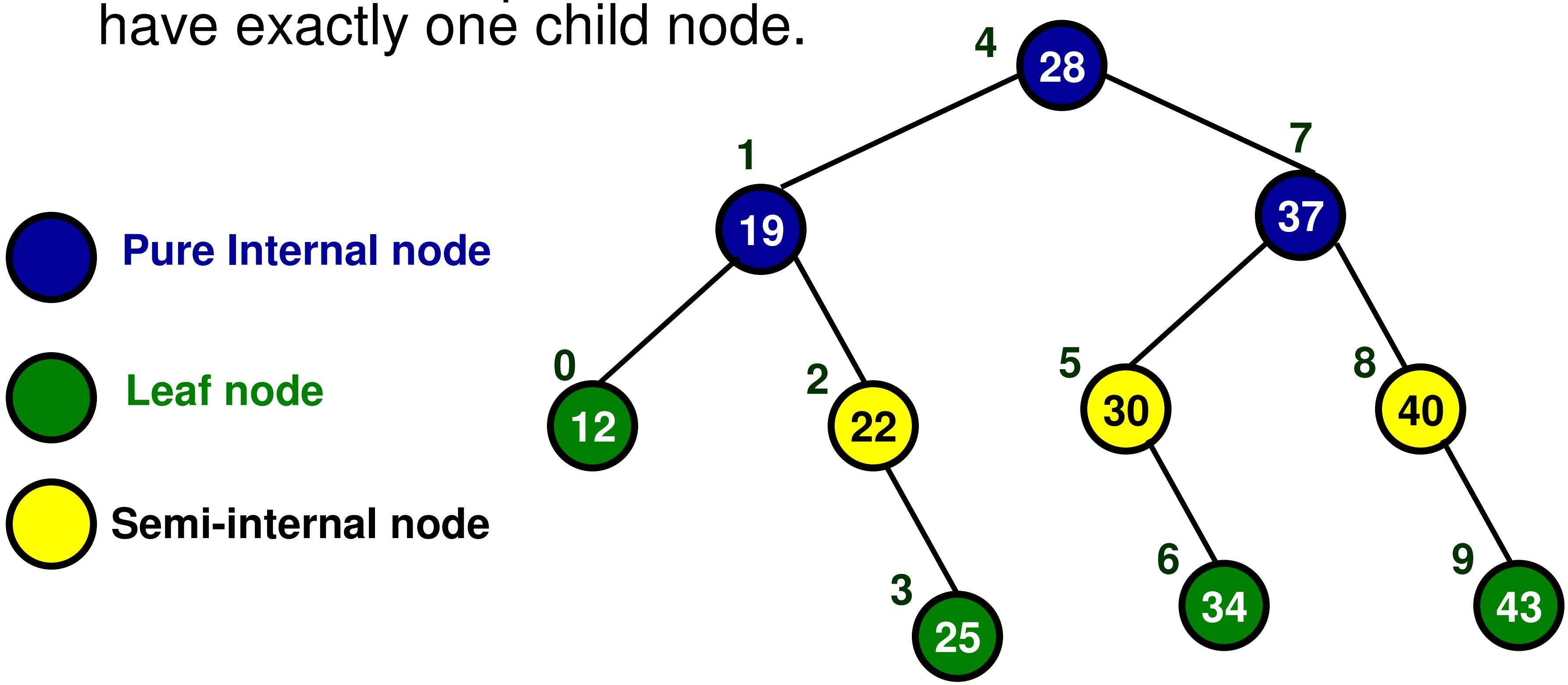
Main Function for BST Implementation based on a Randomly Generated and Sorted Array (Code 7.3: C++)

getIndex(int searchKey) Method C++ Code: 7.4

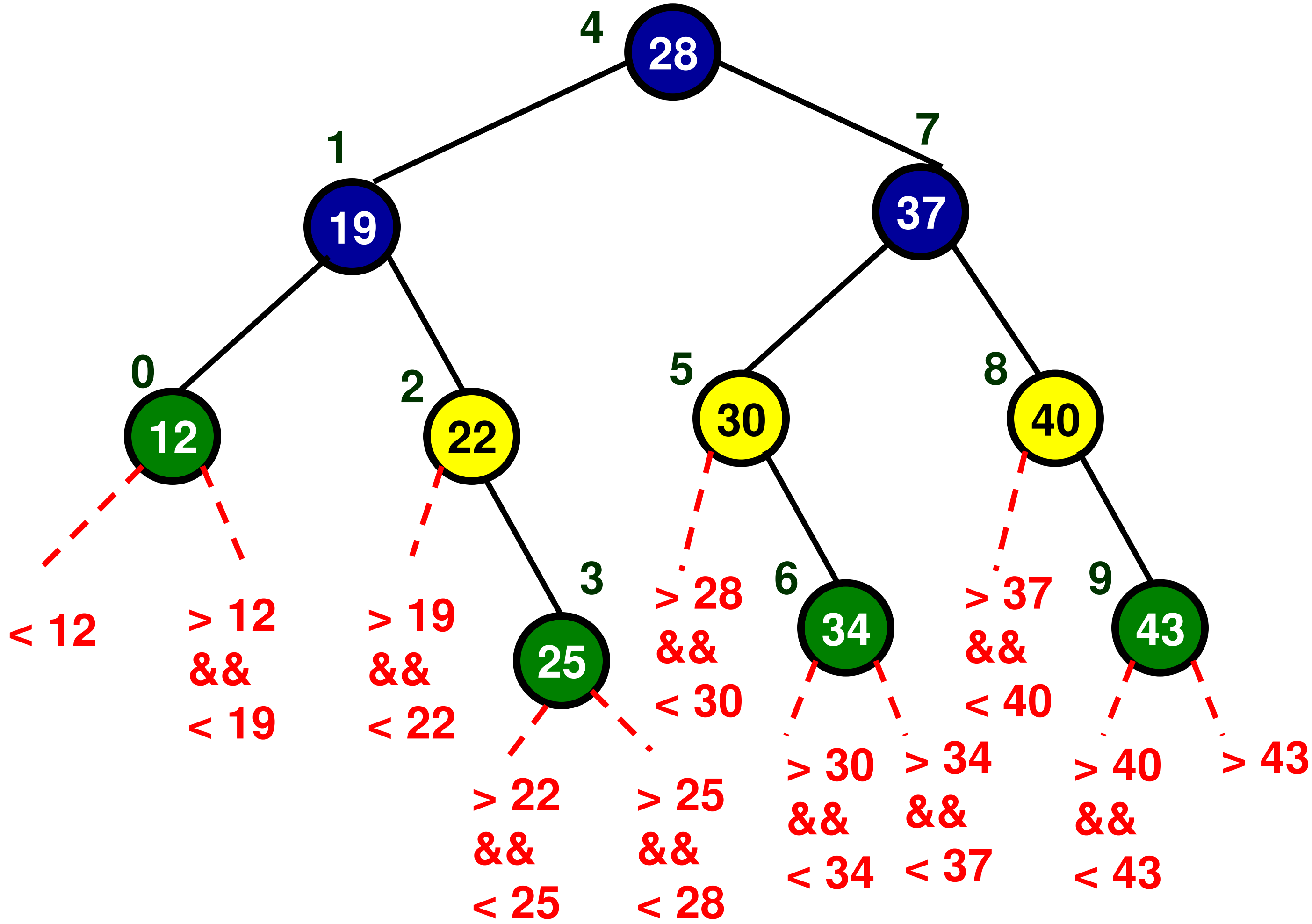
```
int getKeyIndex(int searchKey){

    int searchNodeID = rootNodeID;

    while (searchNodeID != -1){

        if (searchKey == arrayOfBTNodes[searchNodeID].getData())
            return searchNodeID;
        else if (searchKey < arrayOfBTNodes[searchNodeID].getData())
            searchNodeID = arrayOfBTNodes[searchNodeID].getLeftChildID();
        else
            searchNodeID = arrayOfBTNodes[searchNodeID].getRightChildID();

    }

    return -1;
}
```

Avg. # Comparisons: Successful Search and Unsuccessful Search

- A leaf node is a node with no child nodes
- Let us refer to a node as a "pure internal node" if it has both a left child as well as a right child.
- A "semi-internal node" is a node that is not a leaf node as well as not a pure internal node and is considered to have exactly one child node.



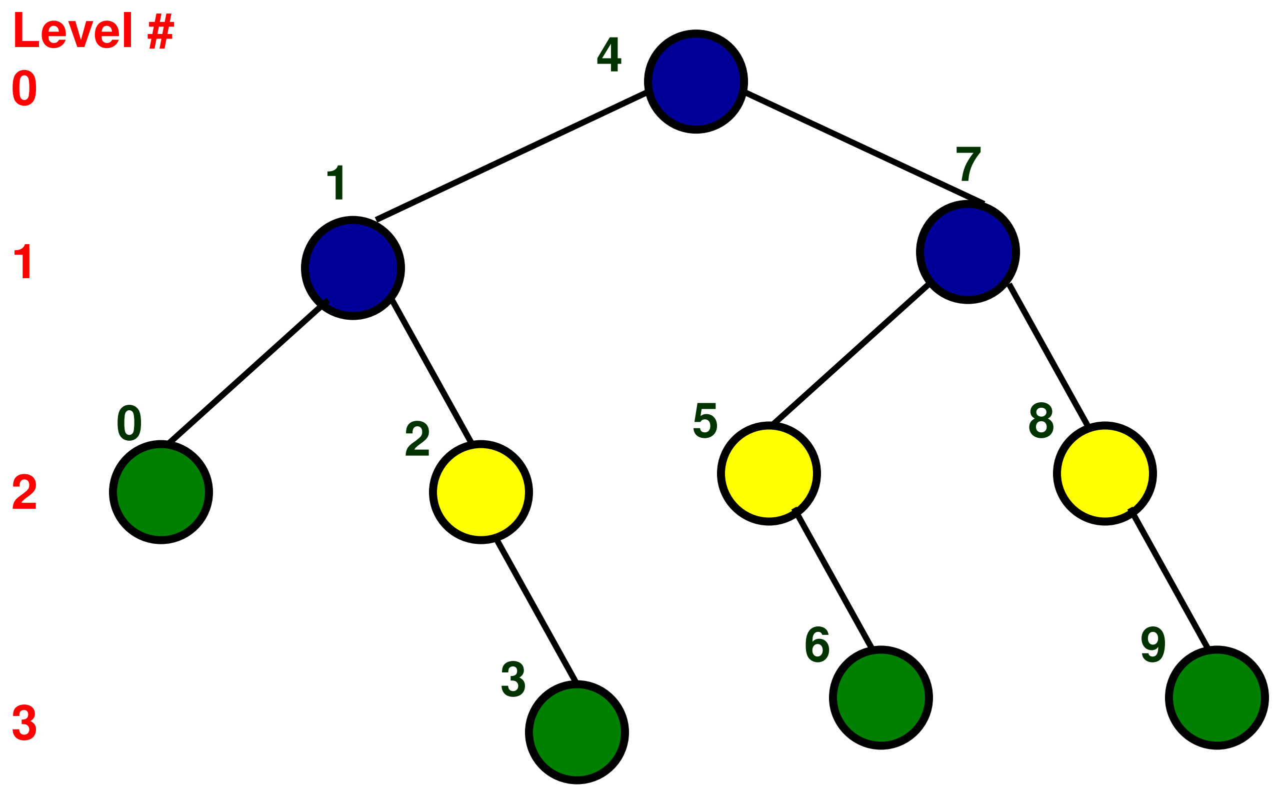
- A **successful search** is a search for a data that corresponds to one of the nodes in the BST.
- An **unsuccessful search** is a search for a missing data that if at all present could be in either the left or right sub tree of a leaf node or in the missing sub tree of a semi-internal node.
  - Each such missing sub trees would constitute a range in which the data for an unsuccessful search could be located.



Avg. # Comparisons based only on the Structure of the BST

<u>Successful Search</u>
 # Comparisons for a node is 1 + the level number for the node

# Comp	# Nodes
1	1
2	2
3	4
4	3



Average # Comparisons for a successful search

Level 0 Level 1 Level 2 Level 3

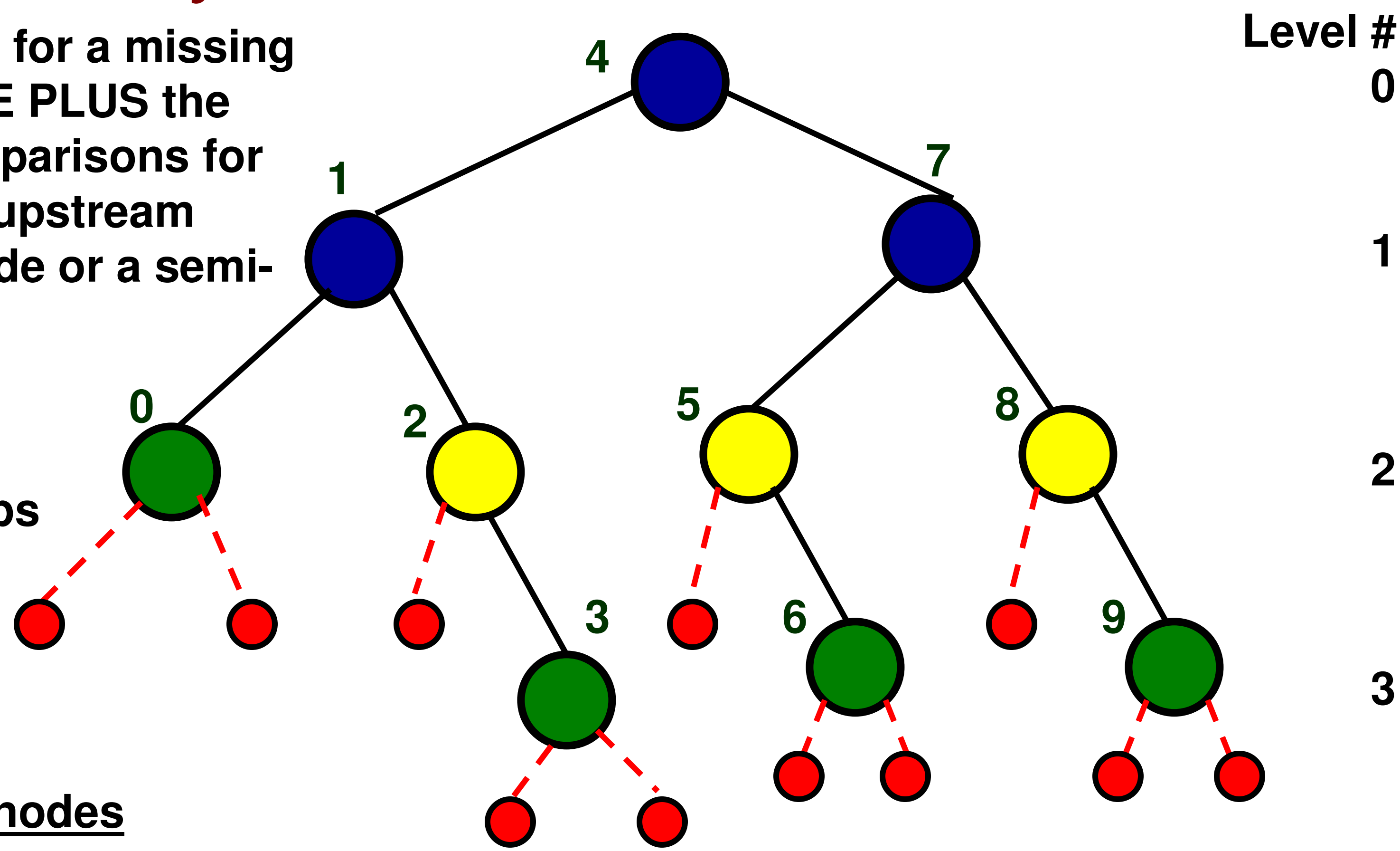
(1 comp * 1 node) + (2 comp * 2 nodes) * (3 comp * 4 nodes) + (4 comp * 3 nodes)

Total number of nodes (10)

= 2.90

Avg # Comparisons: Unsuccessful Search based only on the Structure of the BST

 # Comparisons for a missing Sub tree is ONE PLUS the number of comparisons for the immediate upstream node (a leaf node or a semi-internal node).



Leaf nodes

Node	# Comps
0	3
3	4
6	4
9	4

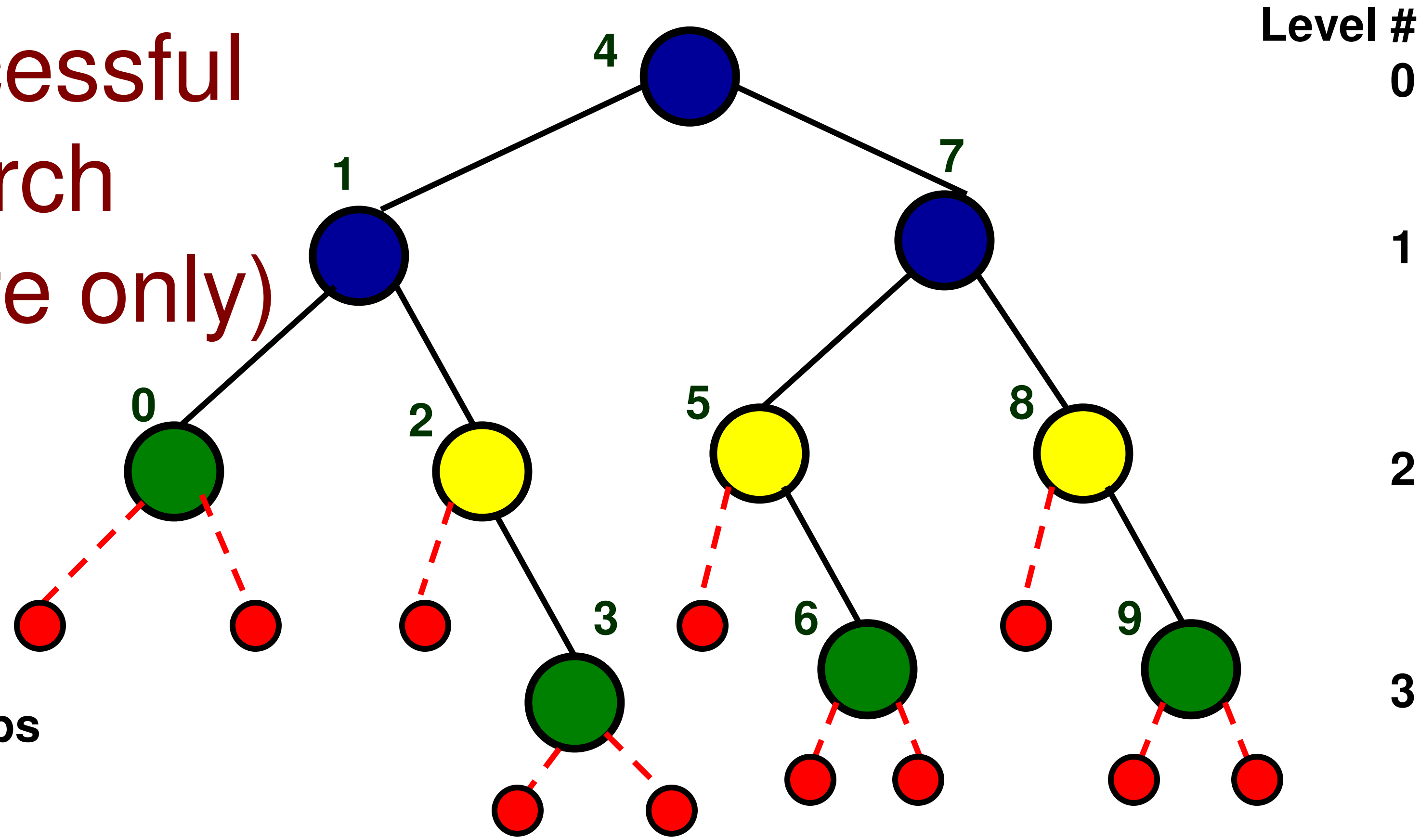
Semi-Internal nodes

Node	# Comps
2	3
5	3
8	3

Avg # comparisons for an unsuccessful search

= (2* Sum of #comps for each leaf node) +
(1* Sum of # comps for each semi-internal node)
----
2 * Number of leaf nodes + 1 * Number of semi-internal nodes

Unsuccessful Search (structure only)



Leaf nodes

Node	# Comps
0	3
3	4
6	4
9	4

Semi-Internal nodes

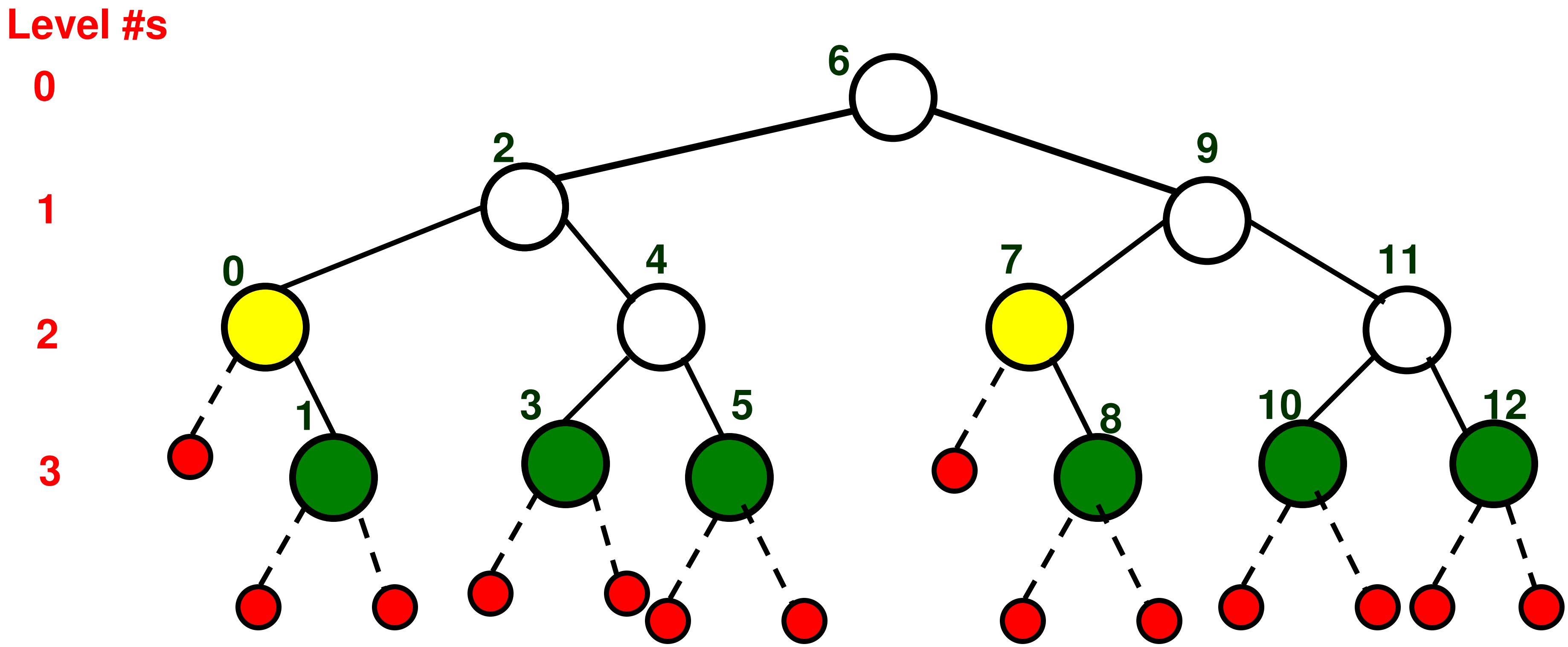
Node	# Comps
2	3
5	3
8	3

Avg # comparisons for an unsuccessful search

$$= \frac{(2^* \text{ Sum of \#comps for each leaf node}) + (1^* \text{ Sum of \# comps for each semi-internal node})}{2 * \text{\# leaf nodes} + 1 * \text{\# semi-internal nodes}}$$

$$= \frac{2 * (3 + 4 + 4 + 4) + 1 * (3 + 3 + 3)}{(2*4 + 1*3)} = 3.55$$

Avg # Comparisons (structure only): Ex. 2



L-0 L-1 L-2 L-3

Avg # Comparisons for Successful Search

$$= \frac{(1*1) + (2*2) + (3*4) + (4*6)}{13} = 3.15$$

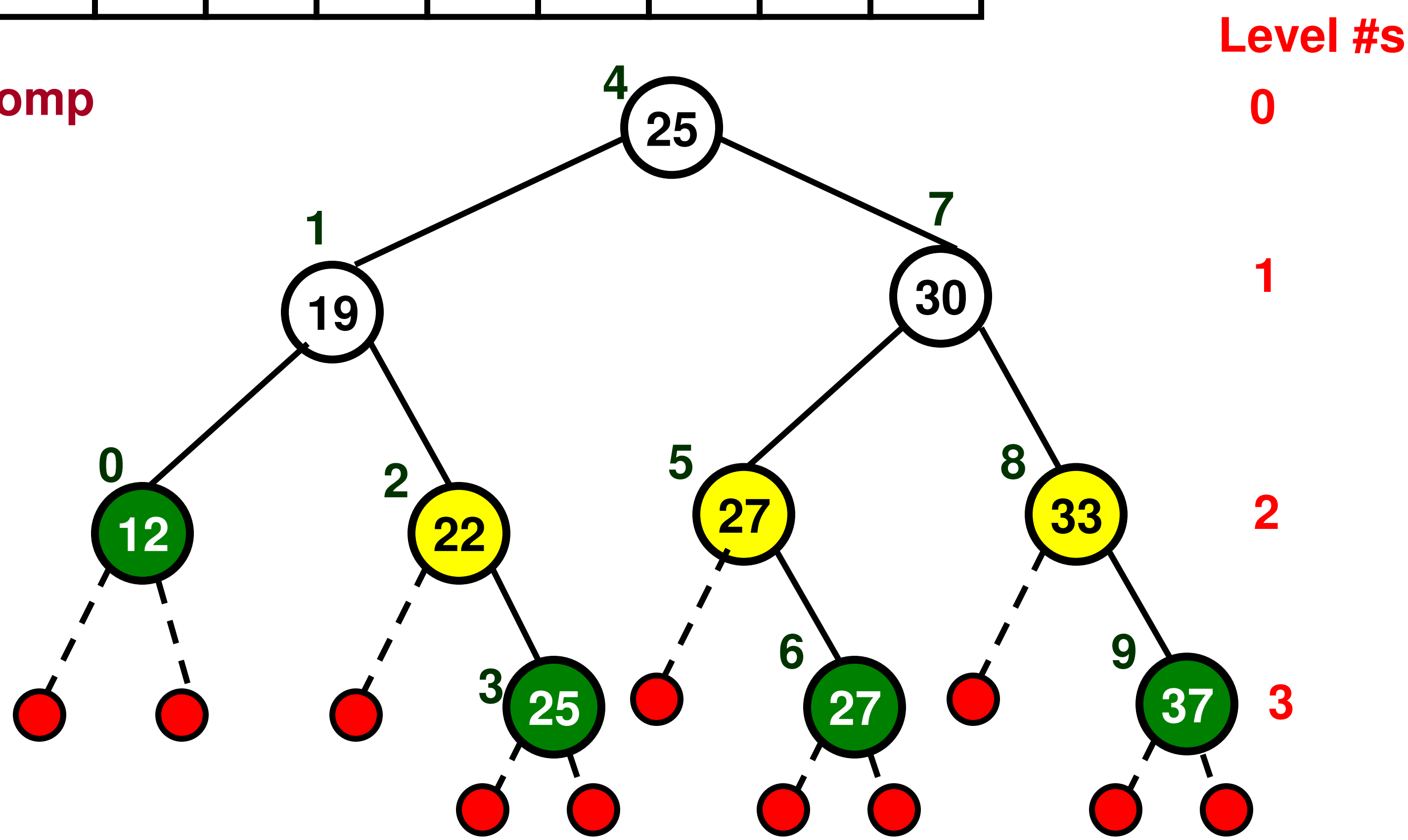
Avg # Comparisons for Unsuccessful Search

$$= \frac{2 * (4 + 4 + 4 + 4 + 4 + 4) + 1 * (3 + 3)}{2 * 6 + 1 * 2} = 3.86$$

Considering both structure and data: Ex. 1

0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37

Index	Data	# Comp
0	12	3
1	19	2
2	22	3
3	25	1
4	25	1
5	27	3
6	27	3
7	30	2
8	33	3
9	37	4



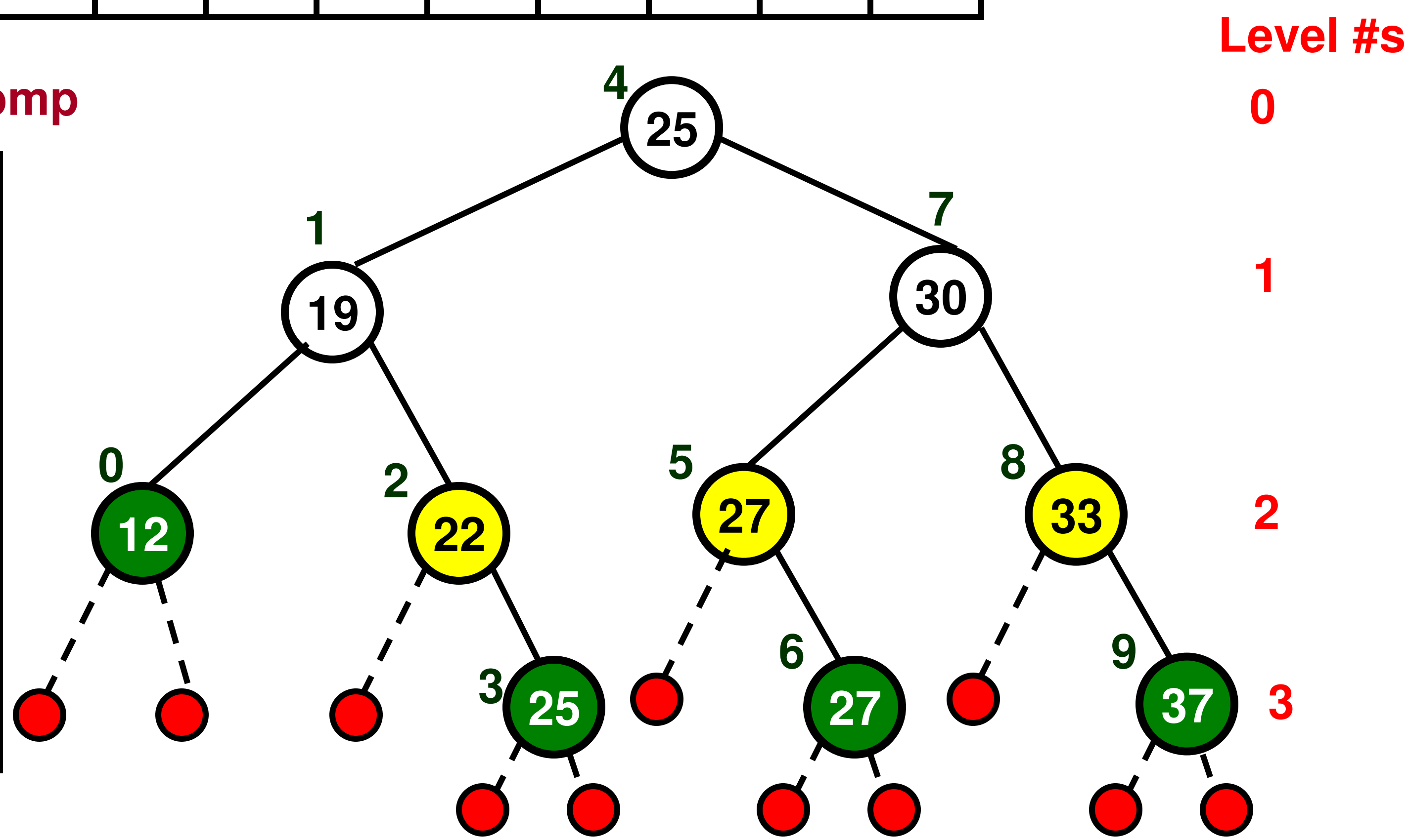
Avg. # Comparisons for a Successful Search

$$= \frac{(3 + 2 + 3 + 1 + 1 + 3 + 3 + 2 + 3 + 4)}{10} = 2.50$$

Considering both structure and data: Ex. 1

0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37

Range	# Comp
< 12	3
> 12 && < 19	3
> 19 && < 22	3
> 22 && < 25	4
> 25 && < 25	0
> 25 && < 27	3
> 27 && < 27	0
> 27 && < 30	4
> 30 && < 33	3
> 33 && < 37	4
> 37	4



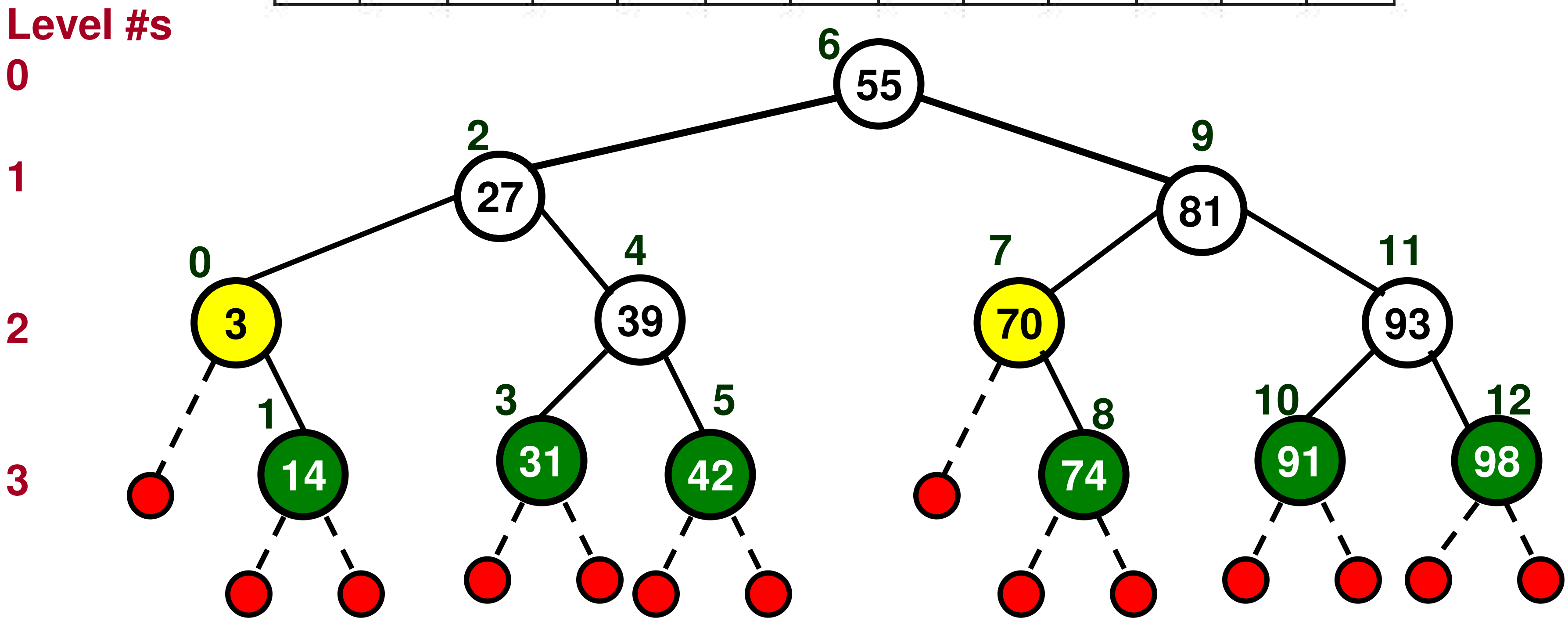
Avg. # Comparisons for an Unsuccessful Search

$$= \frac{(3 + 3 + 3 + 4 + 3 + 4 + 3 + 4 + 4)}{9} = 3.44$$

(Considering only the ranges for which the # comparisons is > 0)

Considering both structure and data: Ex. 2

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98



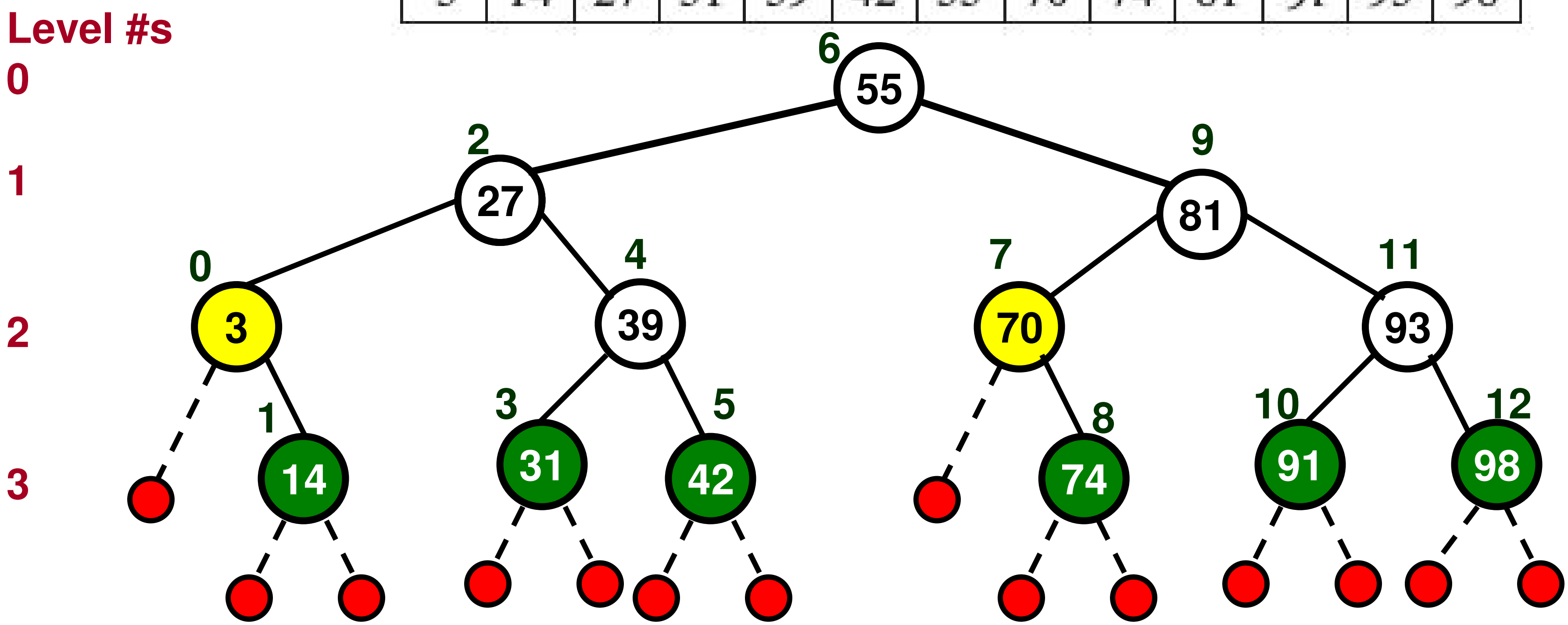
Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Data	3	14	27	31	39	42	55	70	74	81	91	93	98
# Comps	3	4	2	4	3	4	1	3	4	2	4	3	4

Avg. # Comparisons for a Successful Search

$$= \frac{(3+4+2+4+3+4+1+3+4+2+4+3+4)}{13} = 3.15$$

Ex. 2 (contd..)

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98



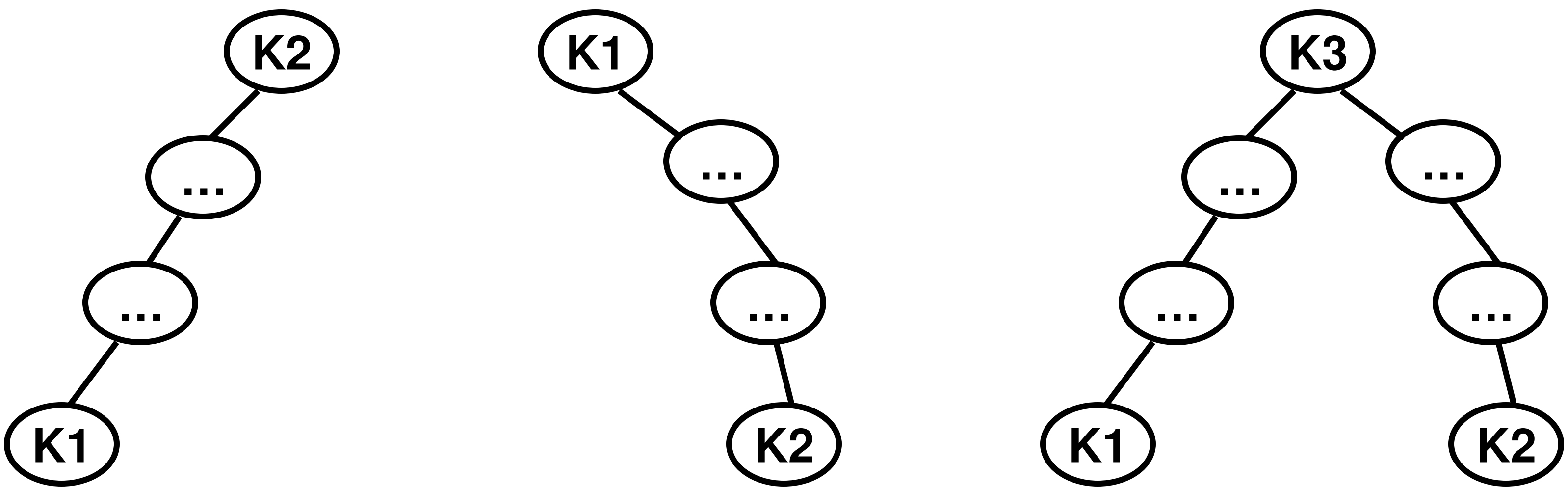
Range	<3	>3 && <14	>14 && <27	>27 && <31	>31 && <39	>39 && <42	>42 && <55	>55 && <70	>70 && <74	>74 && <81	>81 && <91	>91 && <93	>93 && <98	>98
# Comps	3	4	4	4	4	4	4	3	4	4	4	4	4	4

Avg. # Comparisons for An Unsuccessful Search

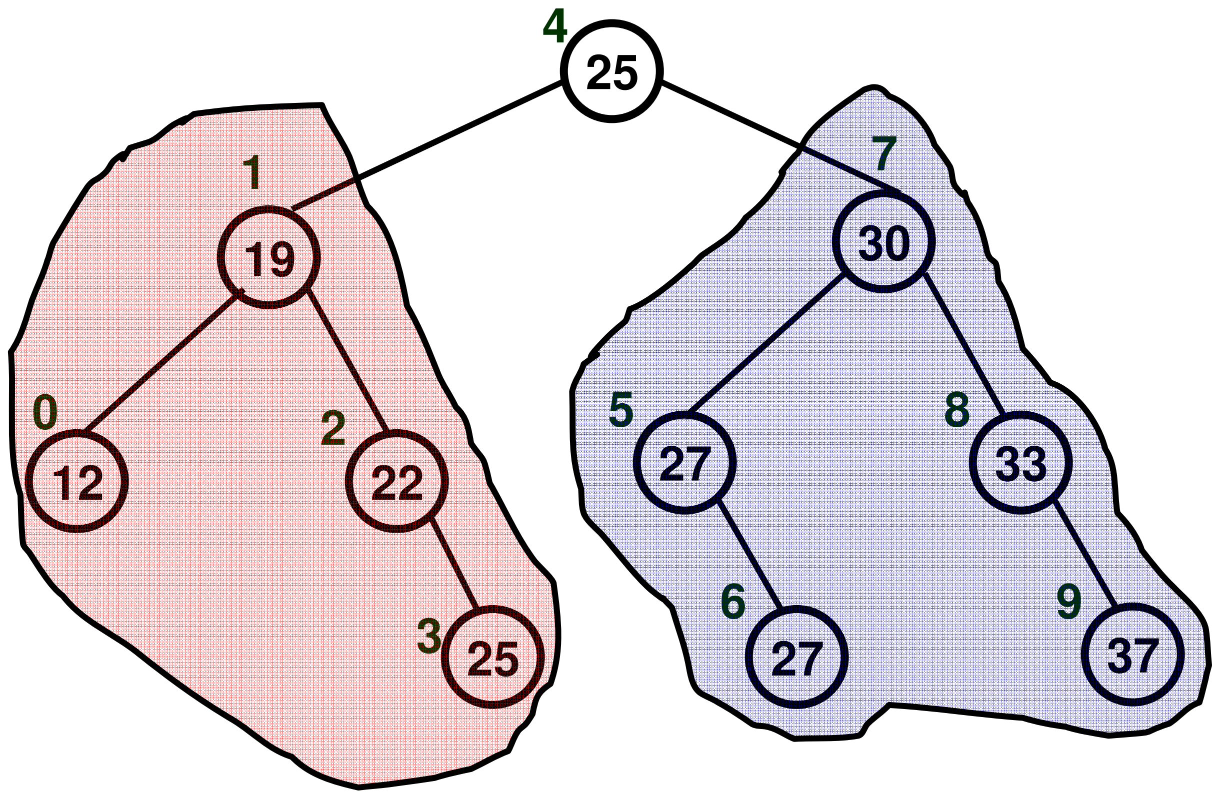
$$= \frac{(3+4+4+4+4+4+4+3+4+4+4+4+4+4)}{14} = 3.86$$

inorder Traversal of a BST (see Code 7.3)

- inorder traversal of a BST will list the keys of the BST in a sorted order.
- Proof: Let K1 < K2 be the two keys in a BST. We want to prove that K1 will appear before K2 in an inorder traversal of the BST.
- There are three scenarios:
  - K2 is in the right sub tree of K1
  - K1 is in the left sub tree of K2
  - K1 and K2 have a common ancestor (say K3) such that K1 < K3 < K2.
- For each of the three scenarios, if we were to do an inorder traversal, K1 will appear before K2.



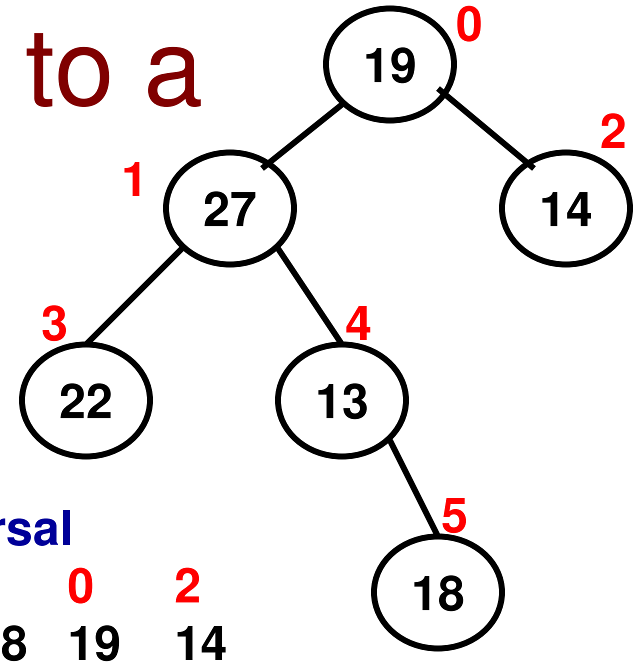
inorder Traversal of a BST



{Left sub tree} {root} {Right sub tree}

Left sub tree				Root	Right sub tree				
0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37

Converting a Binary Tree to a Binary Search Tree (preserving the structure)



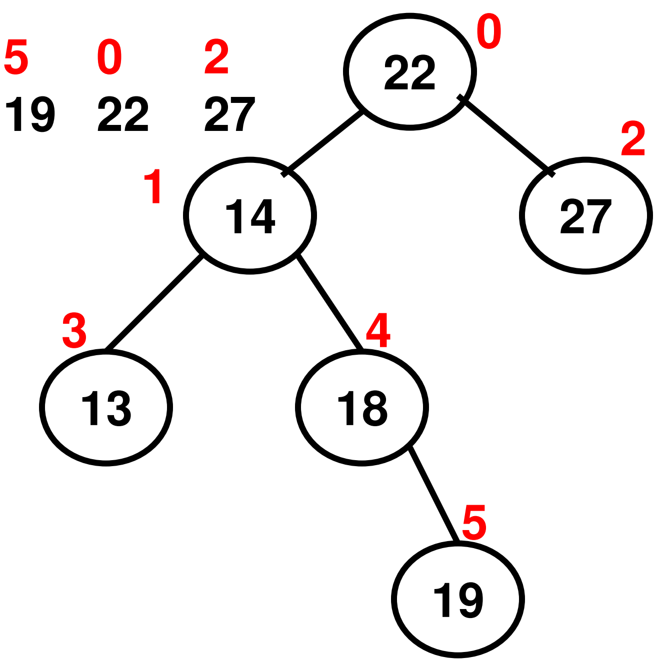
- Do an inorder traversal of the given binary tree and get an array of data corresponding to the nodes of the tree in the order they are visited (i.e., the index entries of the nodes)
- Sort the data using a sorting algorithm
- Do an inorder traversal of the binary tree again. For each node that is about to be listed (as per the index entries), replace their data with the data in the sorted array.

inorder Traversal

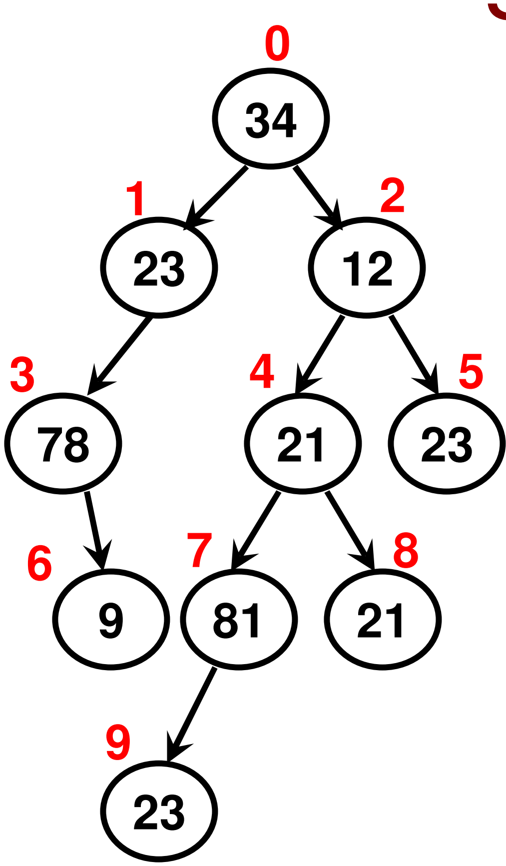
3	1	4	5	0	2
22	27	13	18	19	14

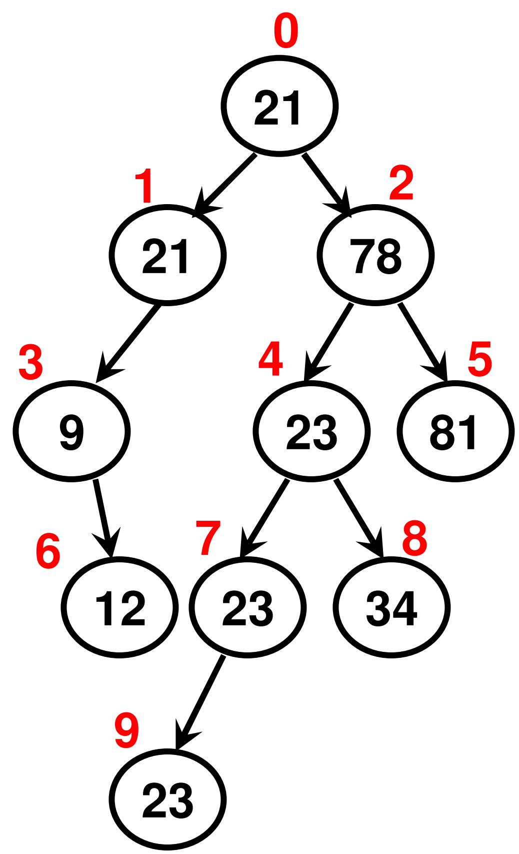
Sorted Order

3	1	4	5	0	2
13	14	18	19	22	27



Converting a Binary Tree to a BST: Example 2





inorder Traversal

3	6	1	0	9	7	4	8	2	5
78	9	23	34	23	81	21	21	12	23

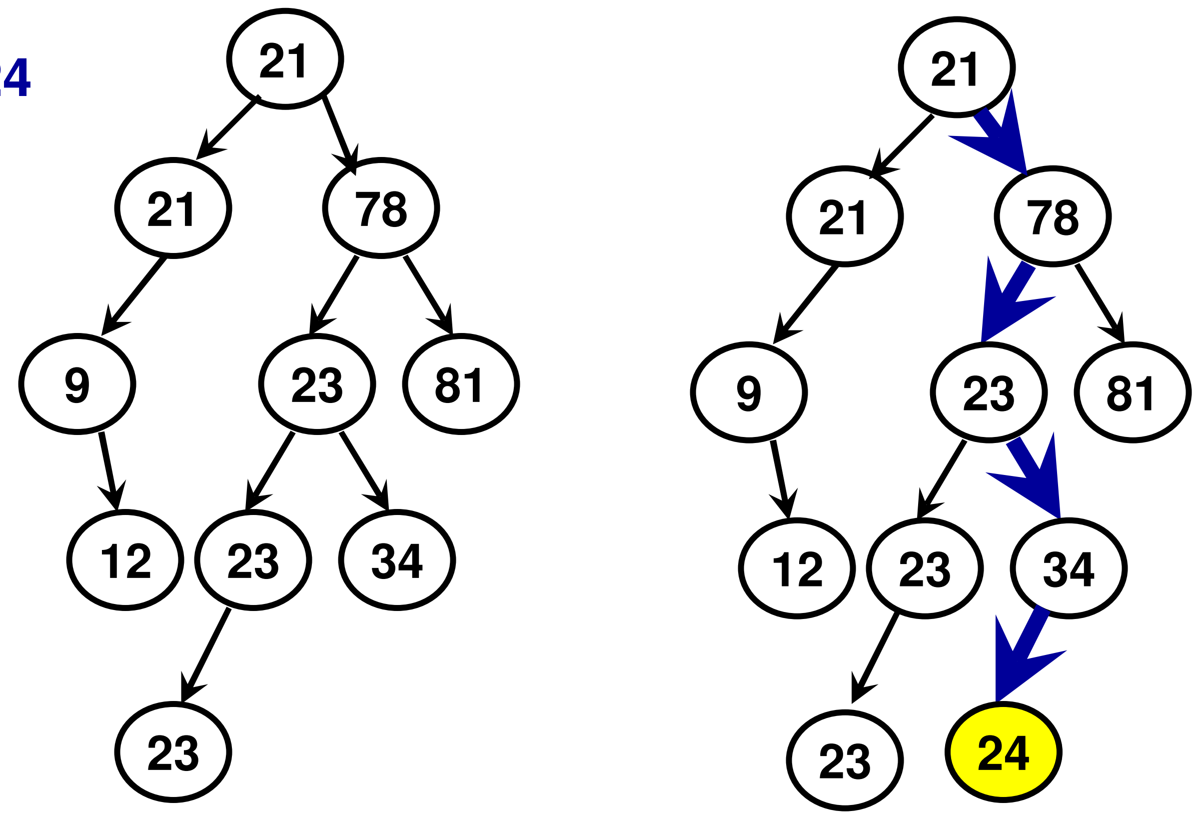
Sorted Order of the inorder Traversed Data

3	6	1	0	9	7	4	8	2	5
9	12	21	21	23	23	23	34	78	81

Inserting an Element in a BST (1)

- Let K be the data to be inserted. Traverse the BST as if we are searching for the data element K. When we come to a leaf node or a semi-internal node, we insert to its left or right depending on the case. If there is a tie, we insert a node as the left child.

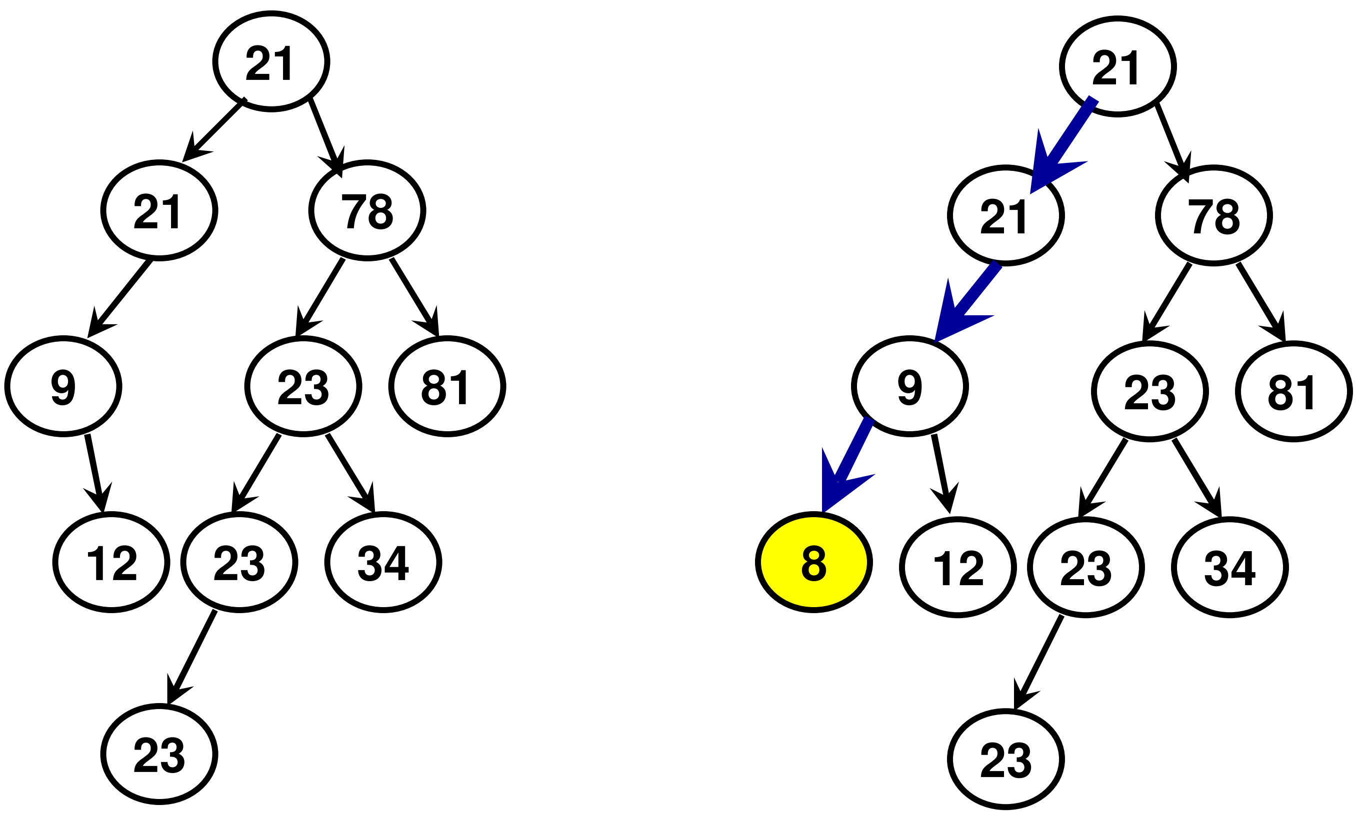
Let K = 24



Inserting an Element in a BST (2)

- Let K be the data to be inserted. Traverse the BST as if we are searching for the data element K. When we come to a leaf node or a semi-internal node, we insert to its left or right depending on the case. If there is a tie, we insert a node as the left child.

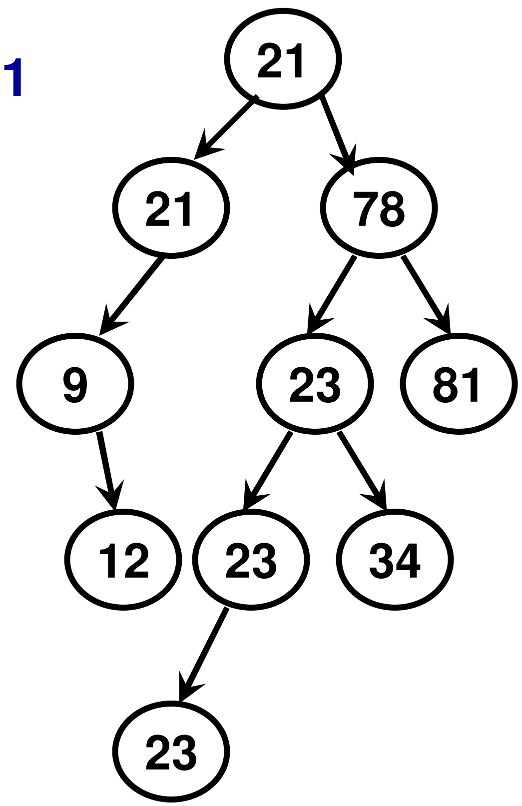
Let K = 8

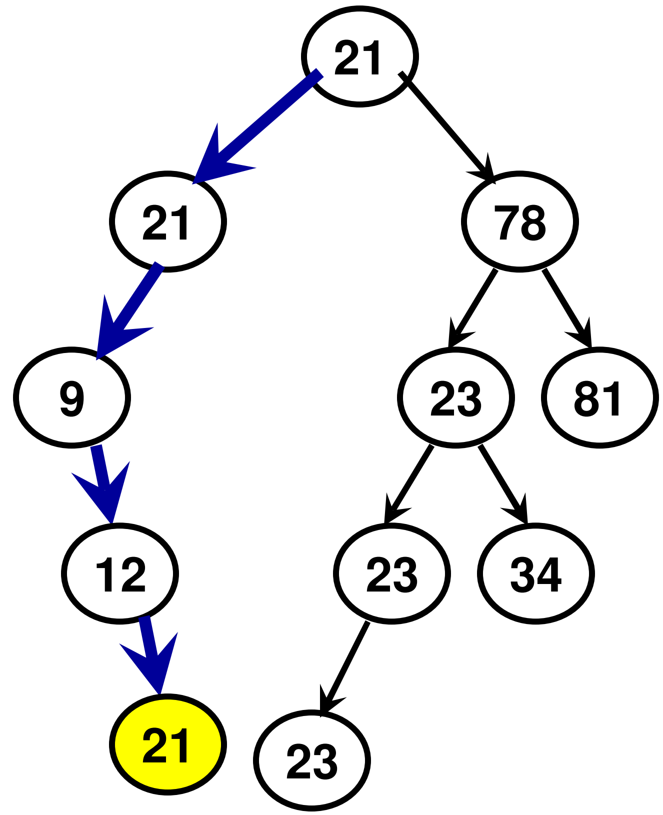


Inserting an Element in a BST (3)

- If the data to insert is already there, then proceed to its left sub tree. Traverse the left sub tree as if you are searching for the data in the sub tree. Follow the rules for insertion as mentioned before.

Let K = 21

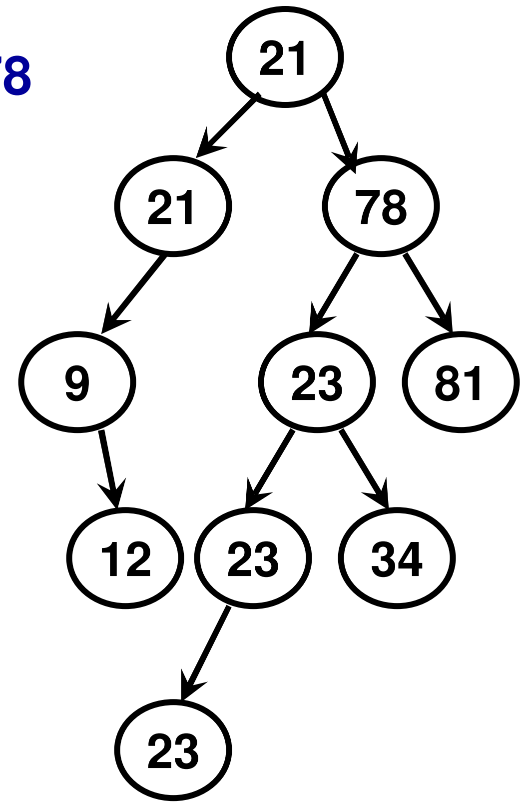


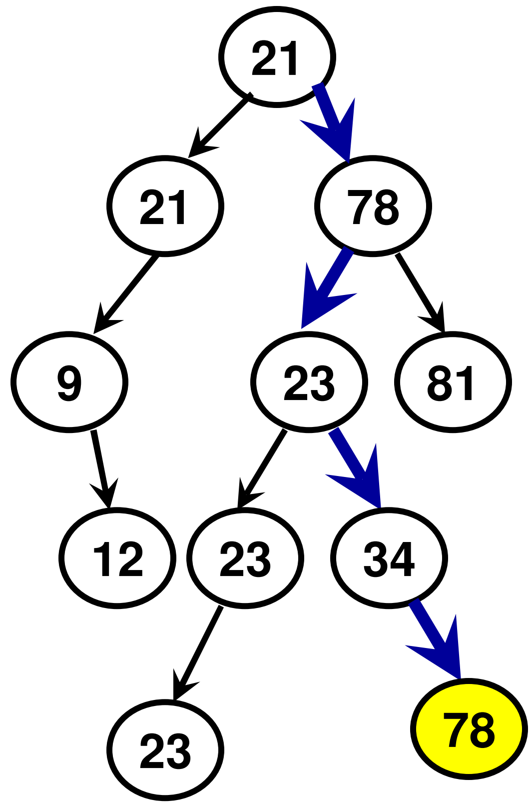


Inserting an Element in a BST (4)

- If the data to insert is already there, then proceed to its left sub tree. Traverse the left sub tree as if you are searching for the data in the sub tree. Follow the rules for insertion as mentioned before.

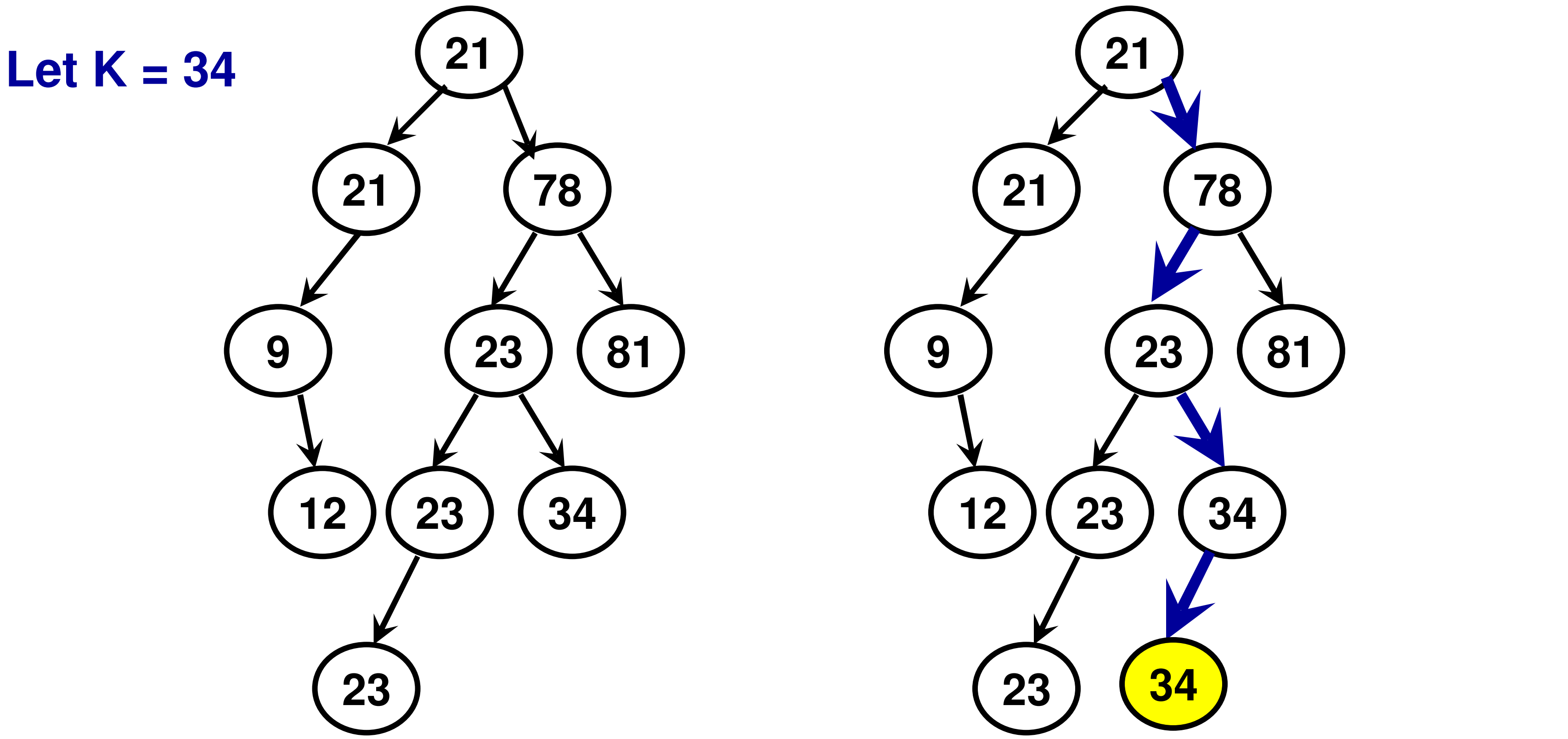
Let K = 78





Inserting an Element in a BST (5)

- If the data to insert is already there, then proceed to its left sub tree. Traverse the left sub tree as if you are searching for the data in the sub tree. Follow the rules for insertion as mentioned before.



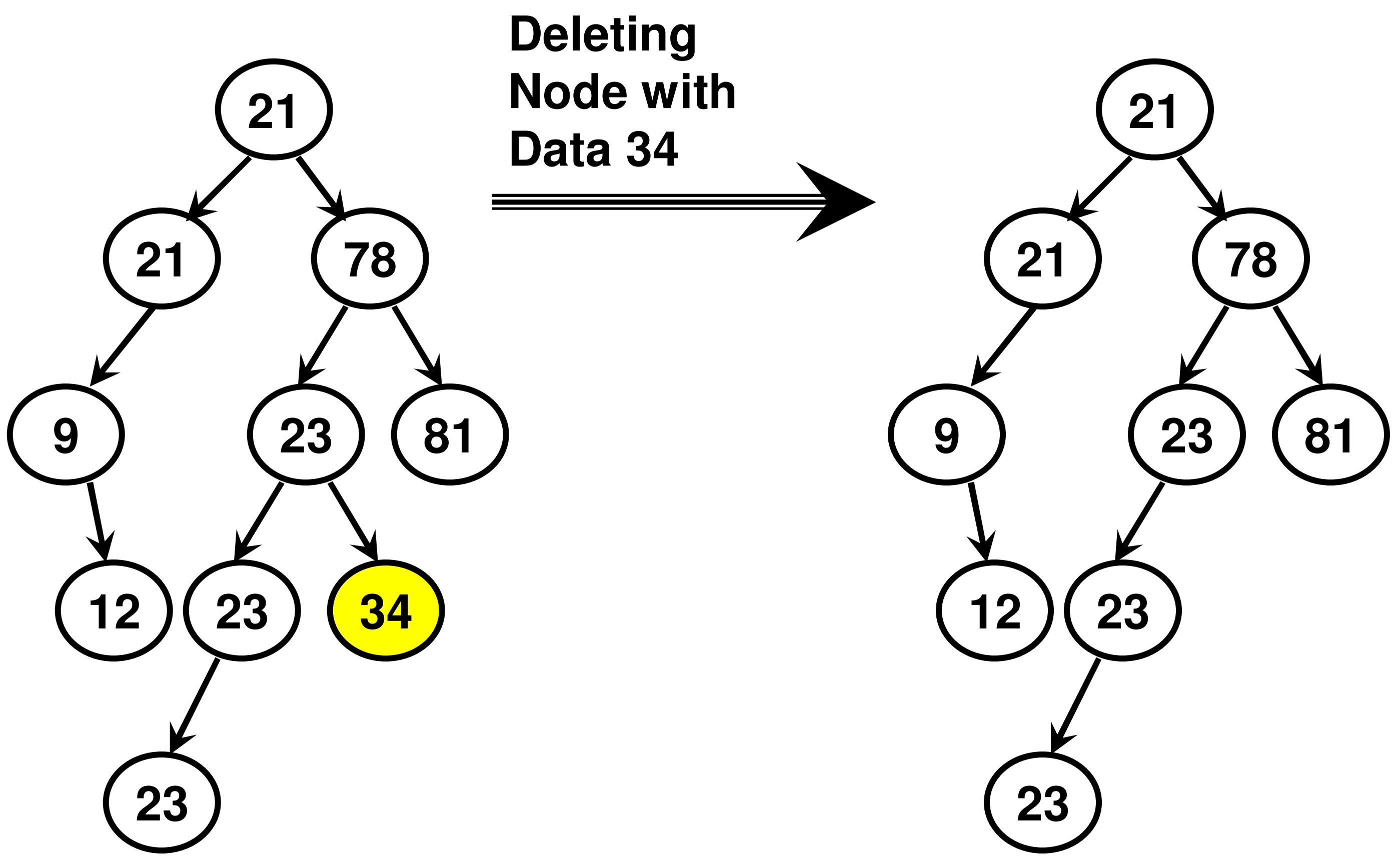
Deleting a Node from a BST

- Three scenarios arise
  - <u>Scenario 1:</u> The node to be deleted is a leaf node:
    - Just delete the node from the BST
  - <u>Scenario 2:</u> The node to be deleted has only one child node (i.e., is a semi-internal node)
    - Replace the node to be deleted with the child node and its sub tree, if any exists
  - <u>Scenario 3:</u> The node to be deleted has two child nodes: Find the inorder successor of the node to be deleted
    - <u>**Scenario 3.1:**</u> If the inorder successor is a leaf node, simply copy its value to the node to be deleted and delete the inorder successor.
    - <u>**Scenario 3.2:**</u> If the inorder successor is an internal node (other than the root), then copy its value to the node to be deleted and link the sub tree rooted at the inorder successor to be the left sub tree of the parent node of the inorder successor.

Deleting a Node from a BST: Ex-1

(Scenario 1: Deleting a leaf node)

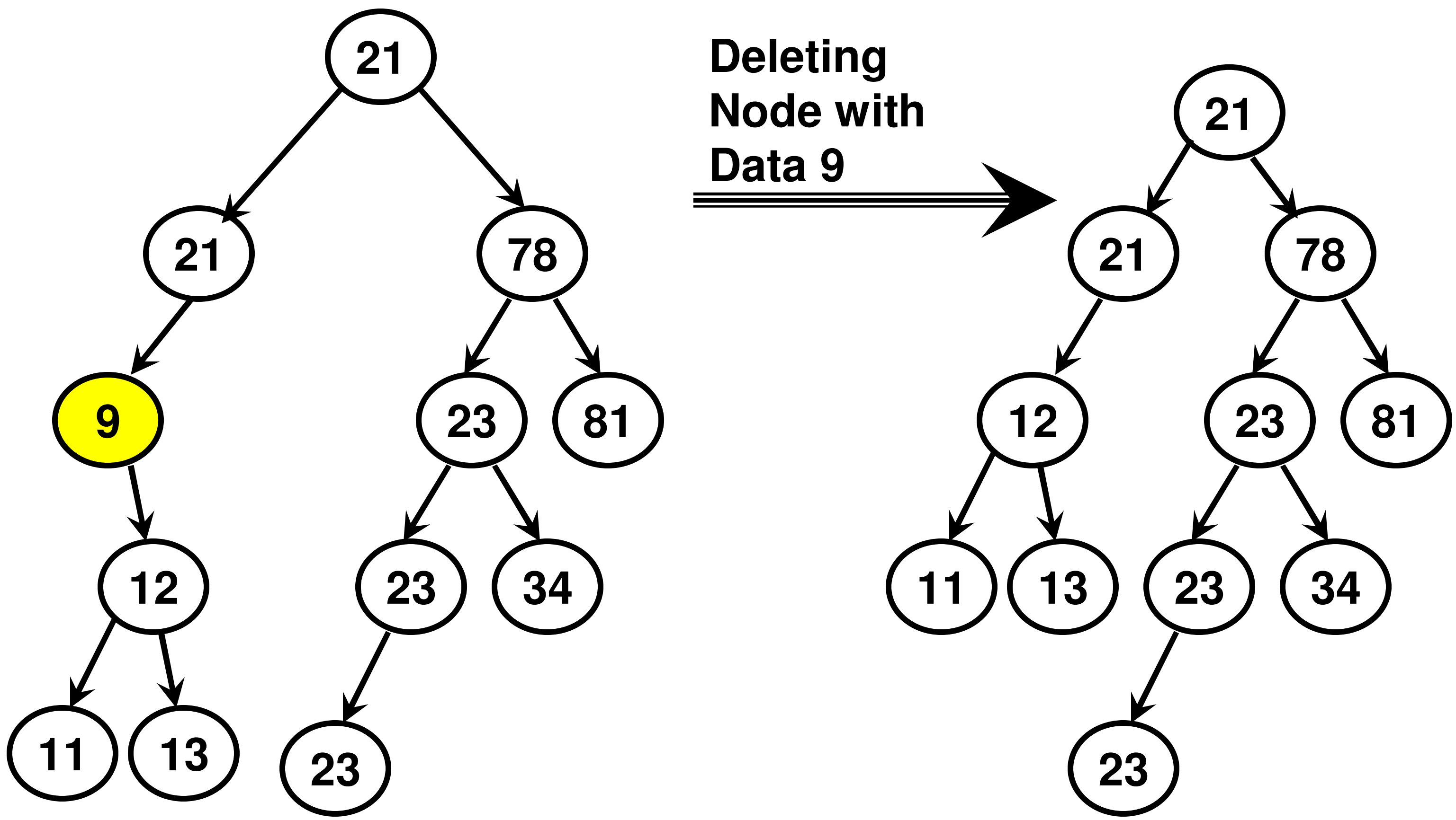
Rule: Just delete the node from the BST



Deleting a Node from a BST: Ex-2

(Scenario 2: Deleting a semi-internal node: an internal node with one child node)

Rule: Replace the node to be deleted with the child node and its sub tree, if any exists

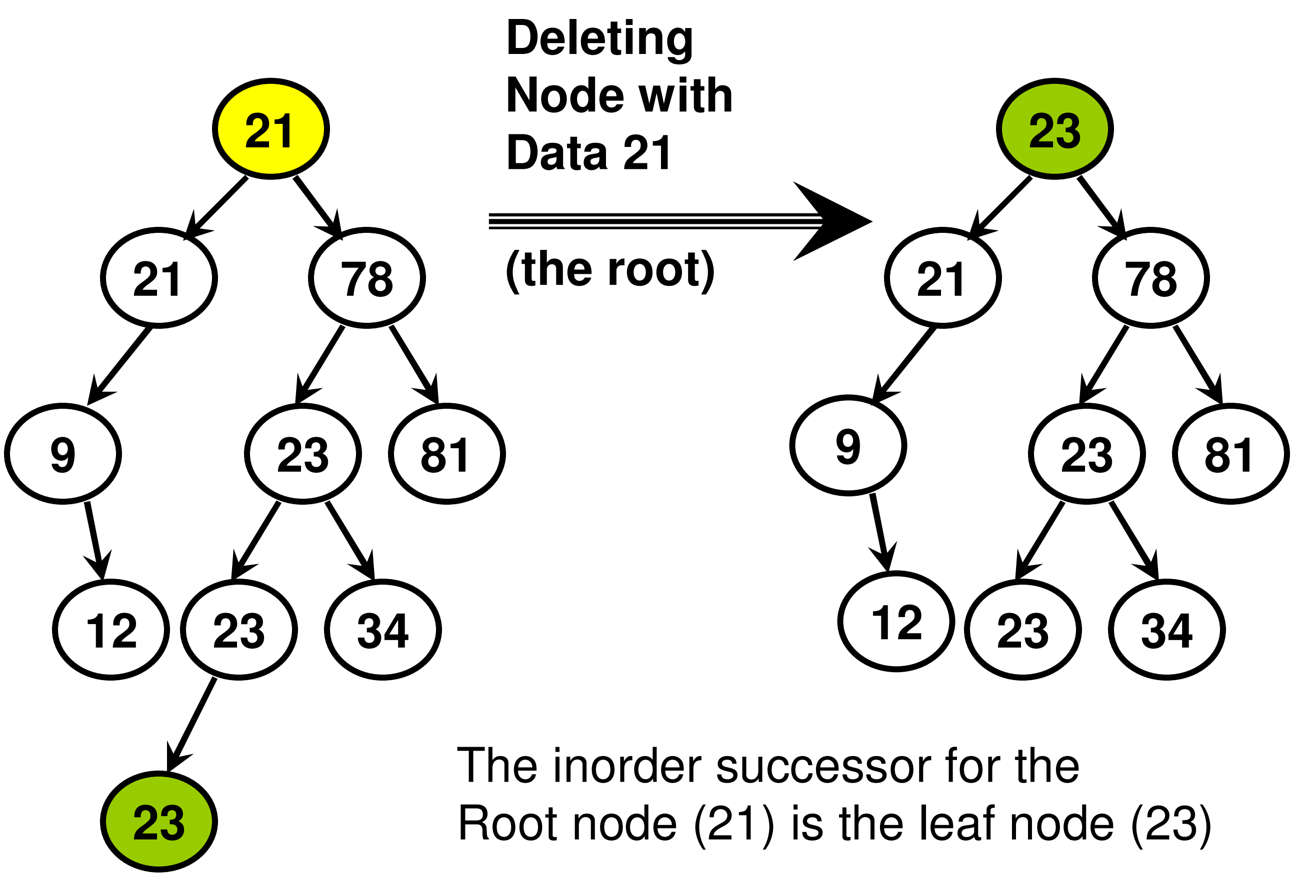


Deleting a Node from a BST: Ex-3

(Scenario 3: Deleting a "pure" internal node with two child nodes)

Scenario 3.1: The inorder successor is a leaf node

Rule: Simply copy the value of the inorder successor to the node to be deleted and delete the inorder successor.



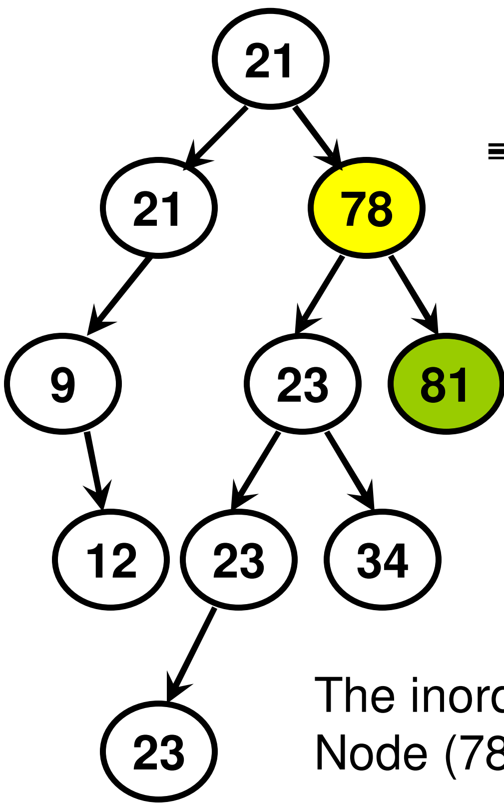
The inorder successor for the Root node (21) is the leaf node (23)

Deleting a Node from a BST: Ex-4

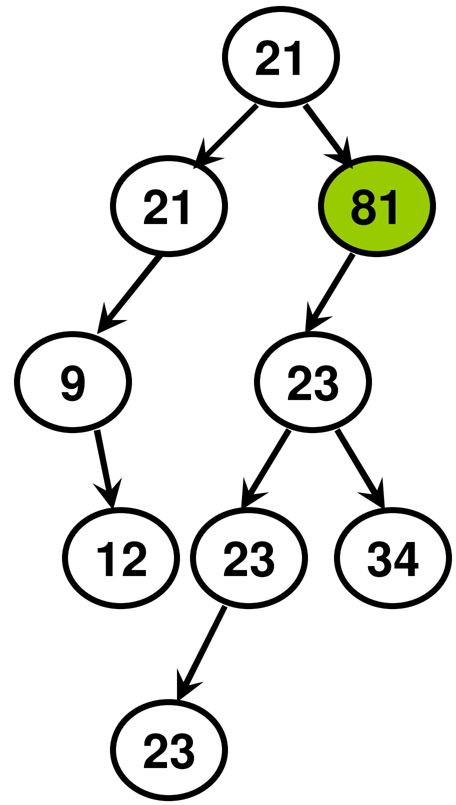
(Scenario 3: Deleting a “pure” internal node with two child nodes)

Scenario 3.1: The inorder successor is a leaf node

Rule: Simply copy the value of the inorder successor to the node to be deleted and delete the inorder successor.



Deleting Node with Data 78



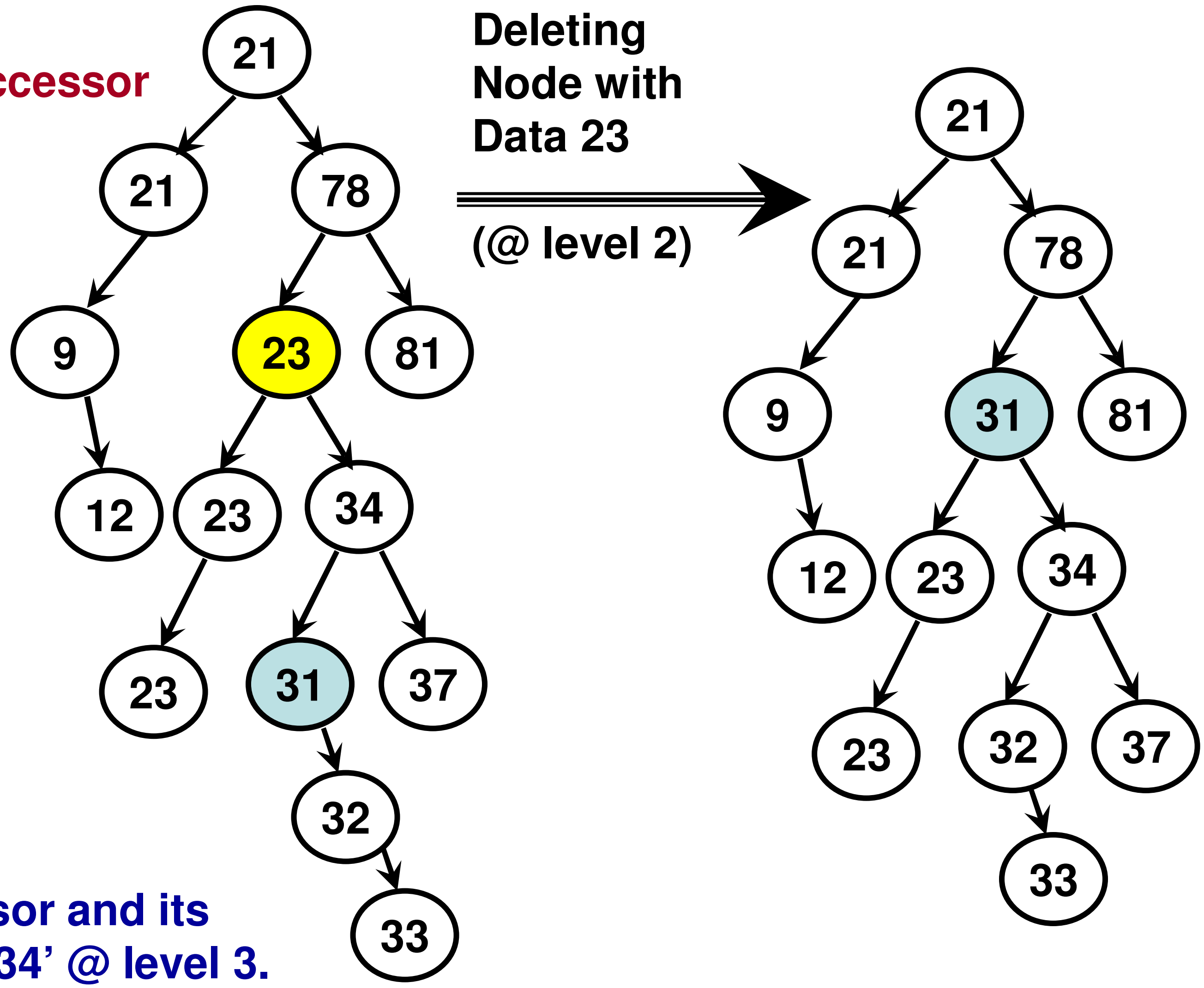
The inorder successor for the Node (78) is the leaf node (81)

Deleting a Node from a BST: Ex-5

(Scenario 3: Deleting a "pure" internal node with two child nodes)

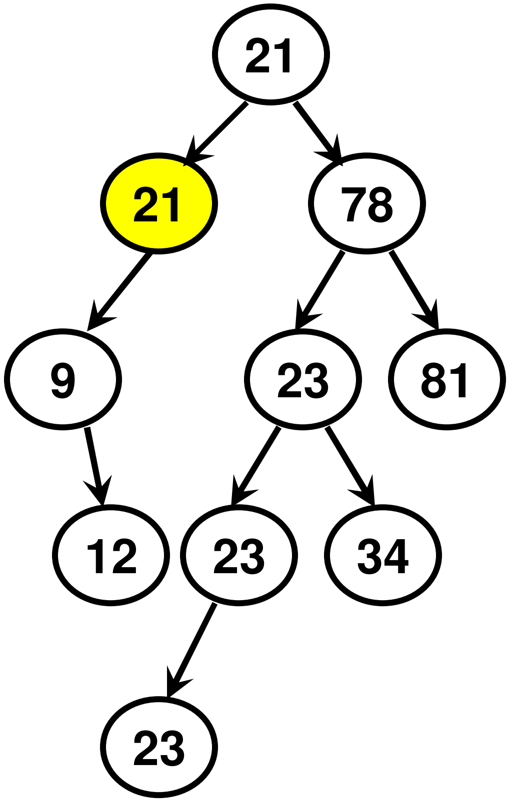
Scenario 3.2: The inorder successor is not a leaf node

Rule: If the inorder successor is an internal node (other than the root), then copy its value to the node to be deleted and link the sub tree rooted at the inorder successor to be the left sub tree of the parent node of the inorder successor.



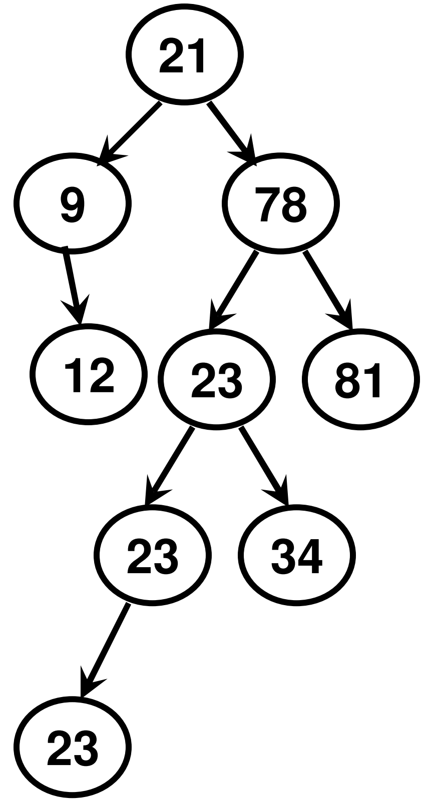
Node '31'@ level 4 is the Inorder successor and its parent node is Node '34' @ level 3.

Deleting a Node from a BST
Ex. 6

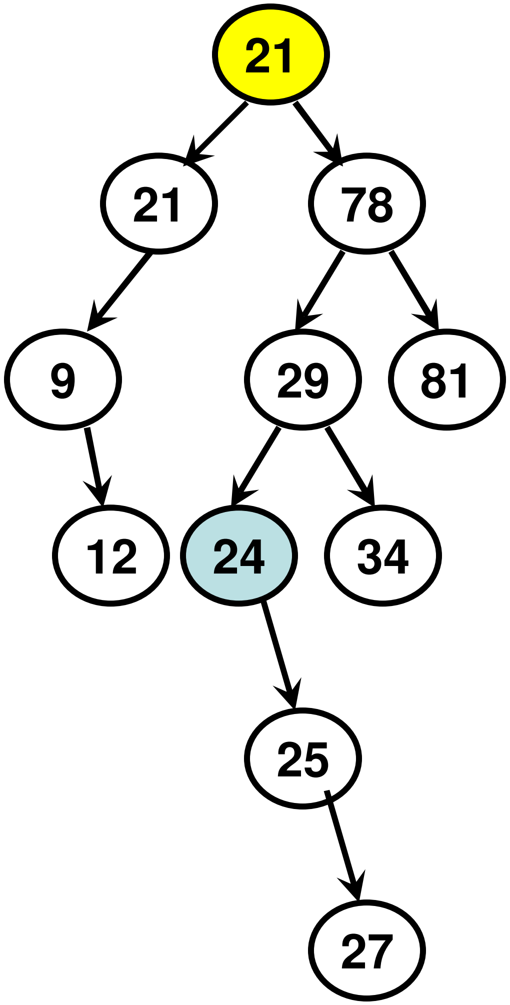


Deleting Node 21 @ Level 1 (a semi-internal node with one child node)

Replace the node to be deleted with the child node and its sub tree, if any exists



Deleting a Node from a BST
Ex. 7



Deleting Node 21 (the root) (an internal node with two child nodes)

If the inorder successor is an internal node, then copy its value to the node to be deleted and link the sub tree rooted at the inorder successor to be the left sub tree of the parent node of the inorder successor.