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, <u>in a sorted array</u>.
- 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.

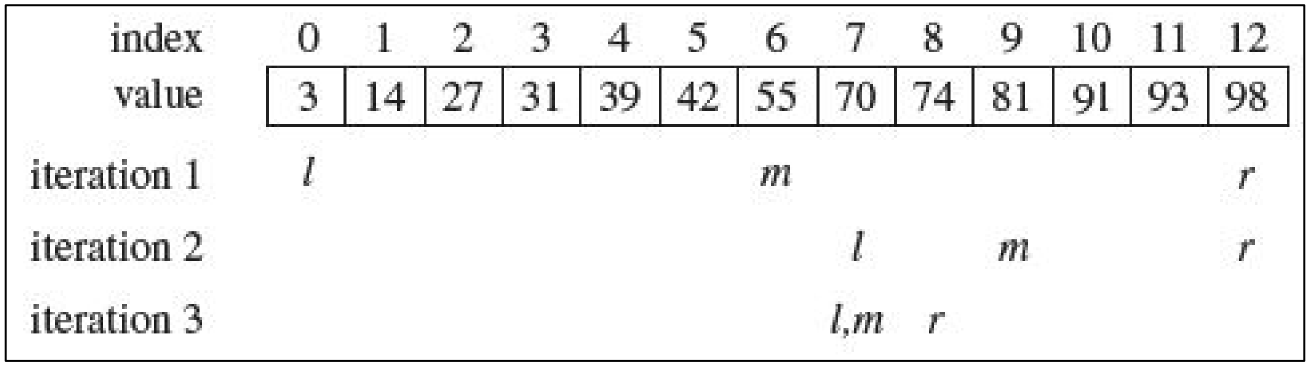
$$\boxed{\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\quad A[m]\quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}} \end{array}}$$

Binary Search

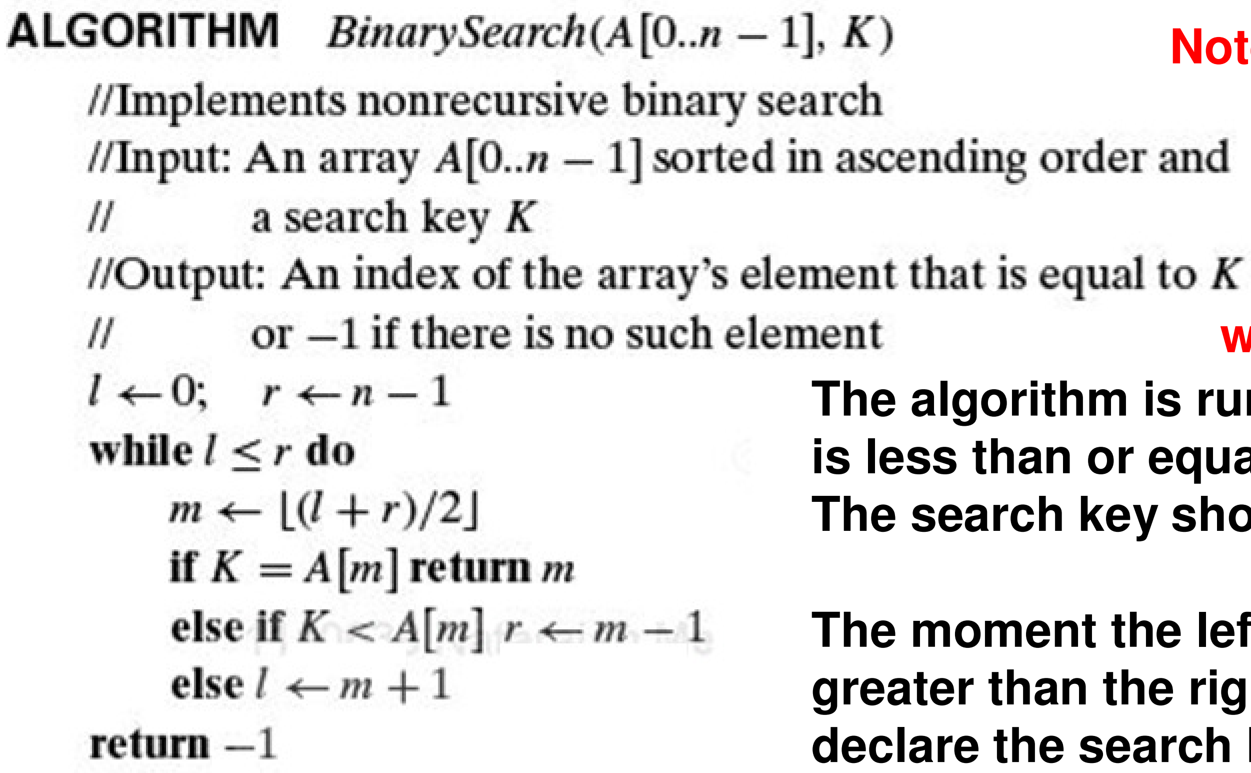
Example

Search Key
K = 70

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 $l \leq r$ 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 $-1$
```

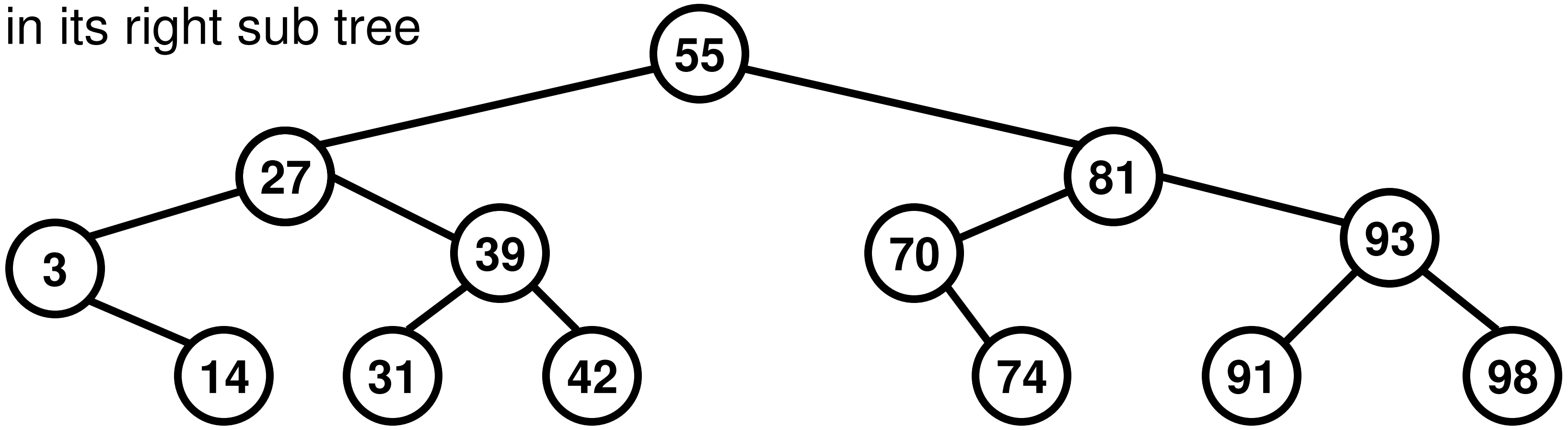
Note that the "search space" reduces by half in each iteration.
Hence, the # iterations is proportional to log(n), where 'n' is the # elements

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)

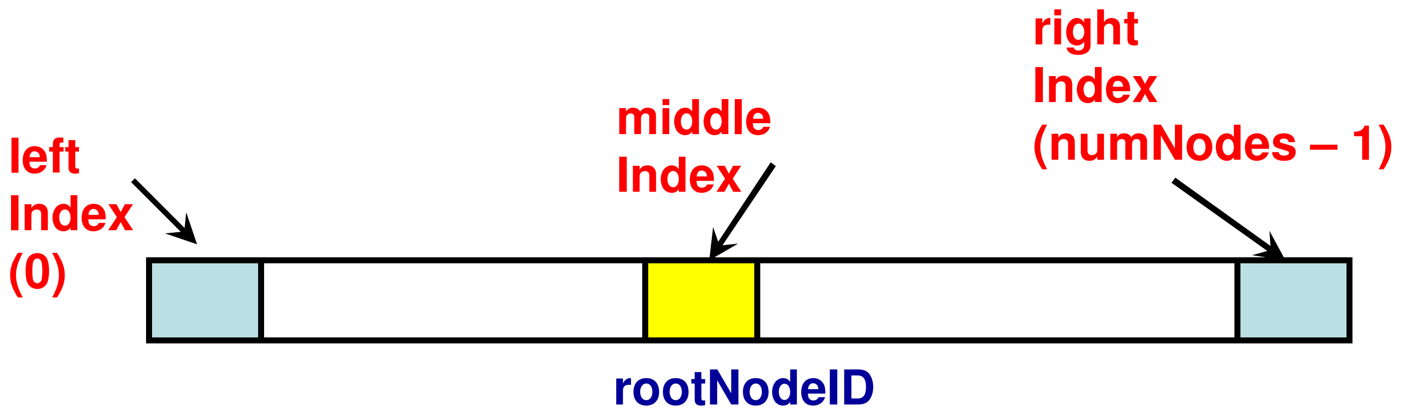
- A binary search tree is a binary tree in which the value for an internal node is greater than or equal to the values of the nodes in its left sub tree and is lower than or equal to the values of the nodes in its right sub tree



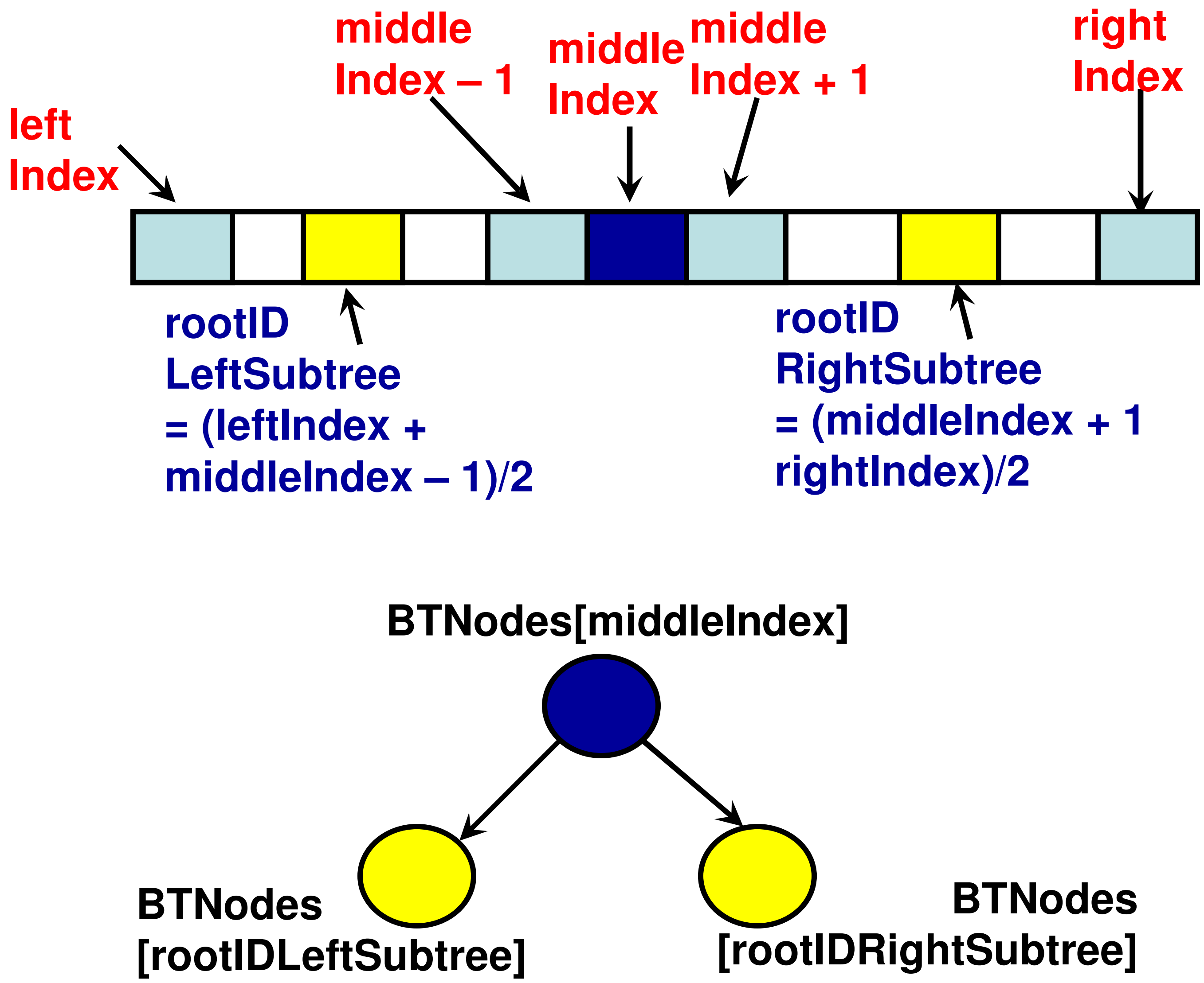
- Both hash tables and BSTs are data structures to implement a Dictionary ADT
- A hash table is an unordered collection of data items as a hash table could be constructed for any arbitrary array and the search could be conducted on a specific linked list to which the search element indexes (hash index) 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
```



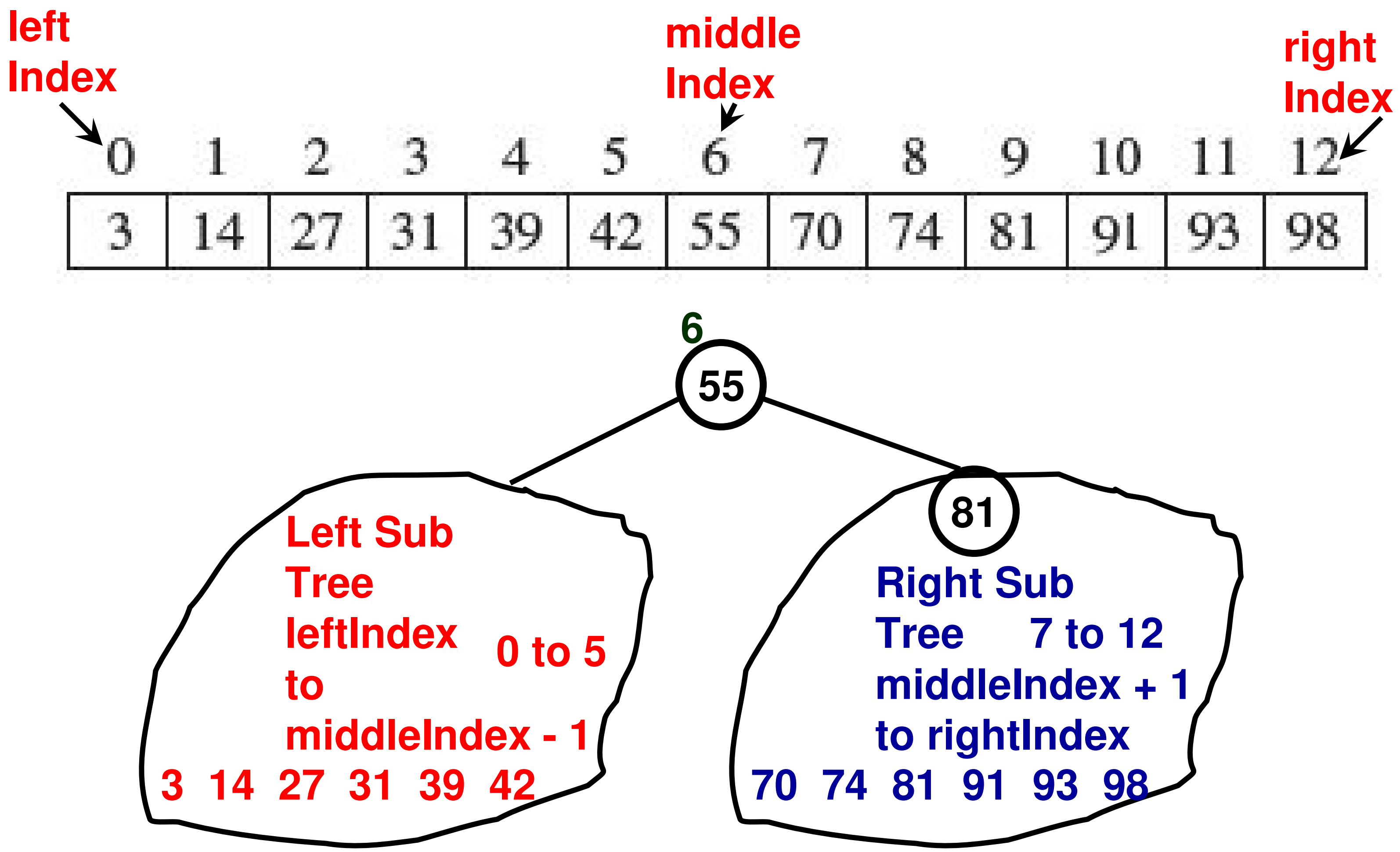
Logic behind the ChainNodes Function


middle Index – 1
middle Index
middle Index + 1
right Index
left Index
rootID LeftSubtree = (leftIndex + middleIndex – 1)/2
rootID RightSubtree = (middleIndex + 1 rightIndex)/2
BTNodes[middleIndex]
BTNodes [rootIDLeftSubtree]
BTNodes [rootIDRightSubtree]

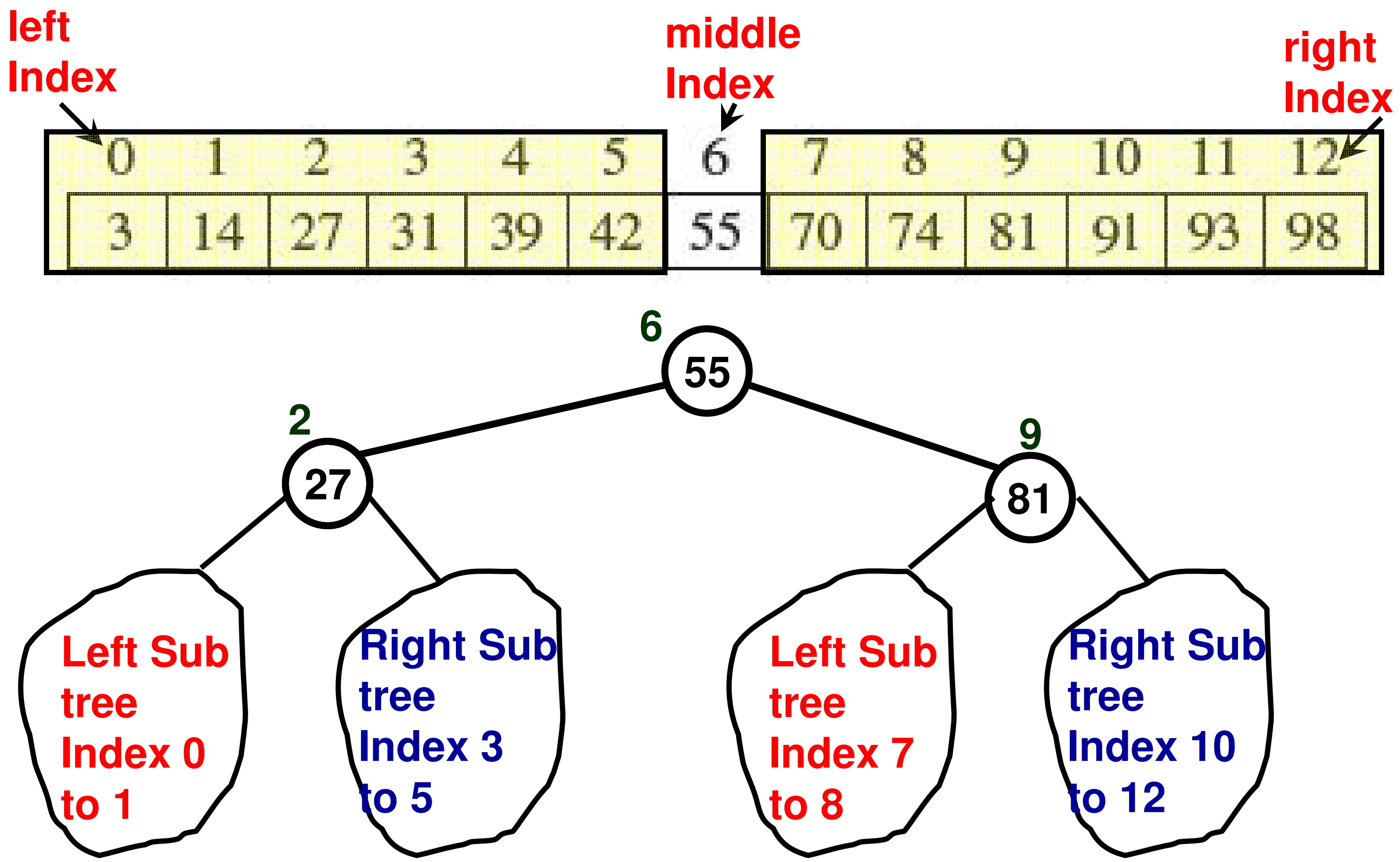
Pseudo Code: ChainNodes Function

```
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

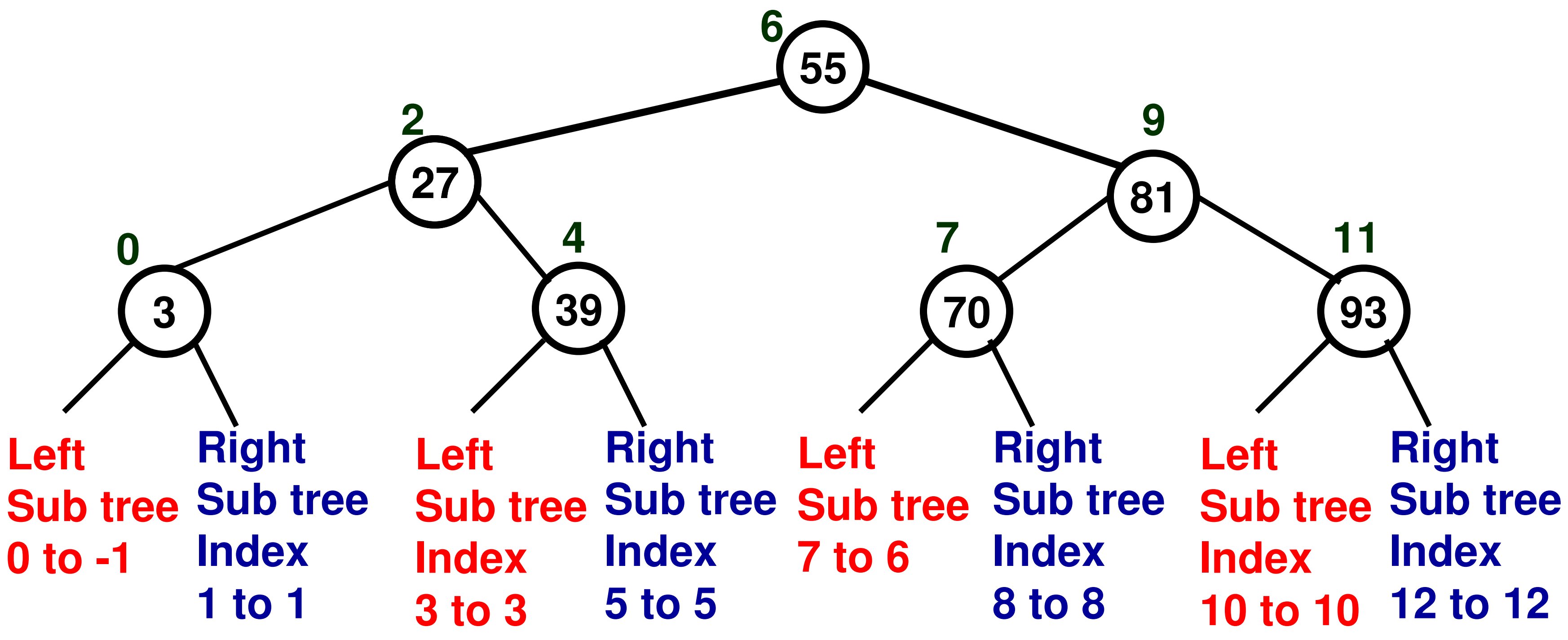


Example 1: Construction of BST

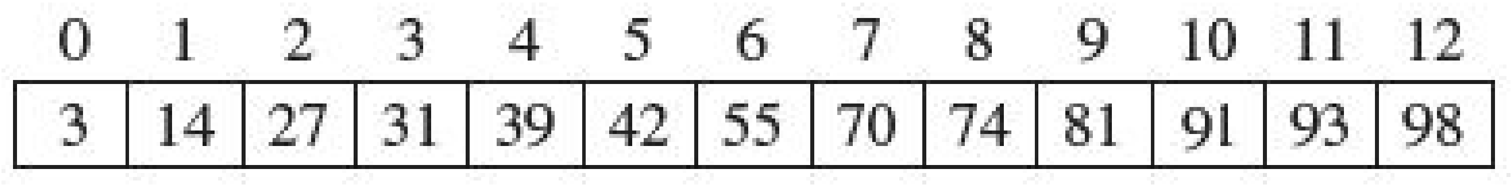

left Index
middle Index
right Index
0 1 2 3 4 5 6 7 8 9 10 11 12
3 14 27 31 39 42 55 70 74 81 91 93 98
6
55
2
27
9
81
Left Sub tree Index 0 to 1
Right Sub tree Index 3 to 5
Left Sub tree Index 7 to 8
Right Sub tree Index 10 to 12

Example 1: Construction of BST

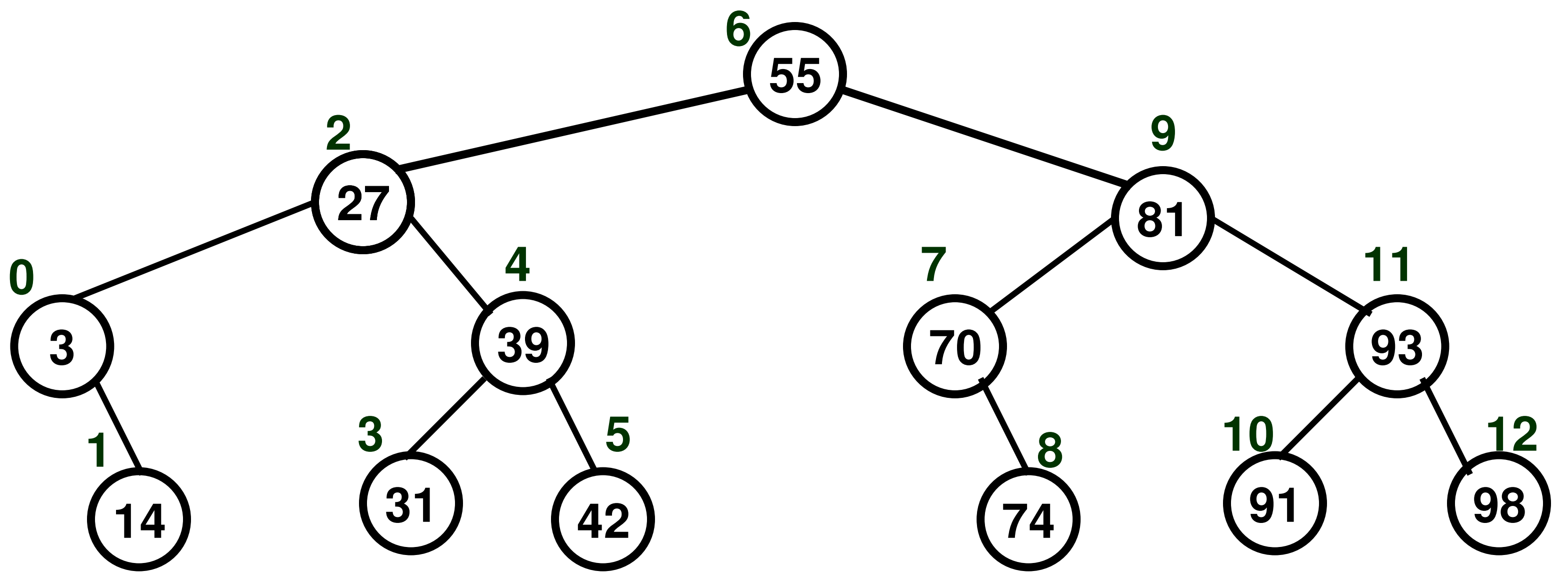
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98


6
55
2
27
9
81
0
3
4
39
7
70
11
93
Left Sub tree 0 to -1
Right Sub tree Index 1 to 1
Left Sub tree Index 3 to 3
Right Sub tree Index 5 to 5
Left Sub tree 7 to 6
Right Sub tree Index 8 to 8
Left Sub tree Index 10 to 10
Right Sub tree Index 12 to 12

Example 1: Construction of BST

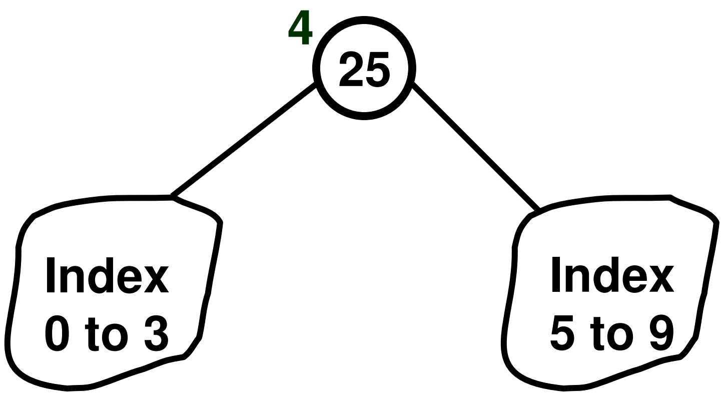


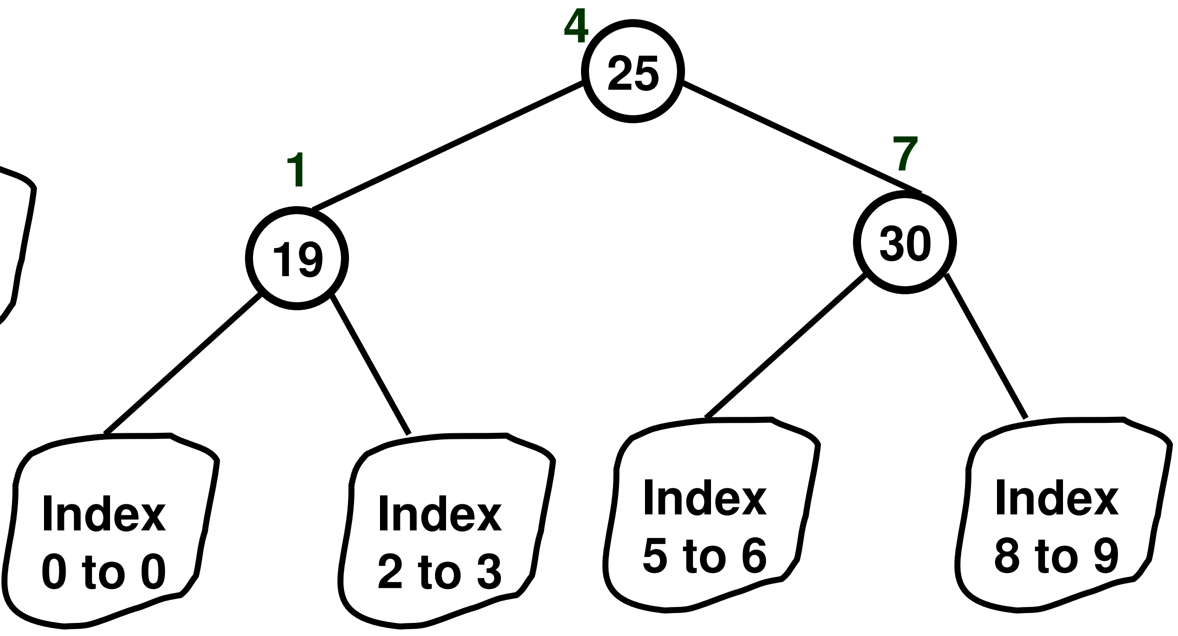
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	91	93	98



Example 2: Construction of BST

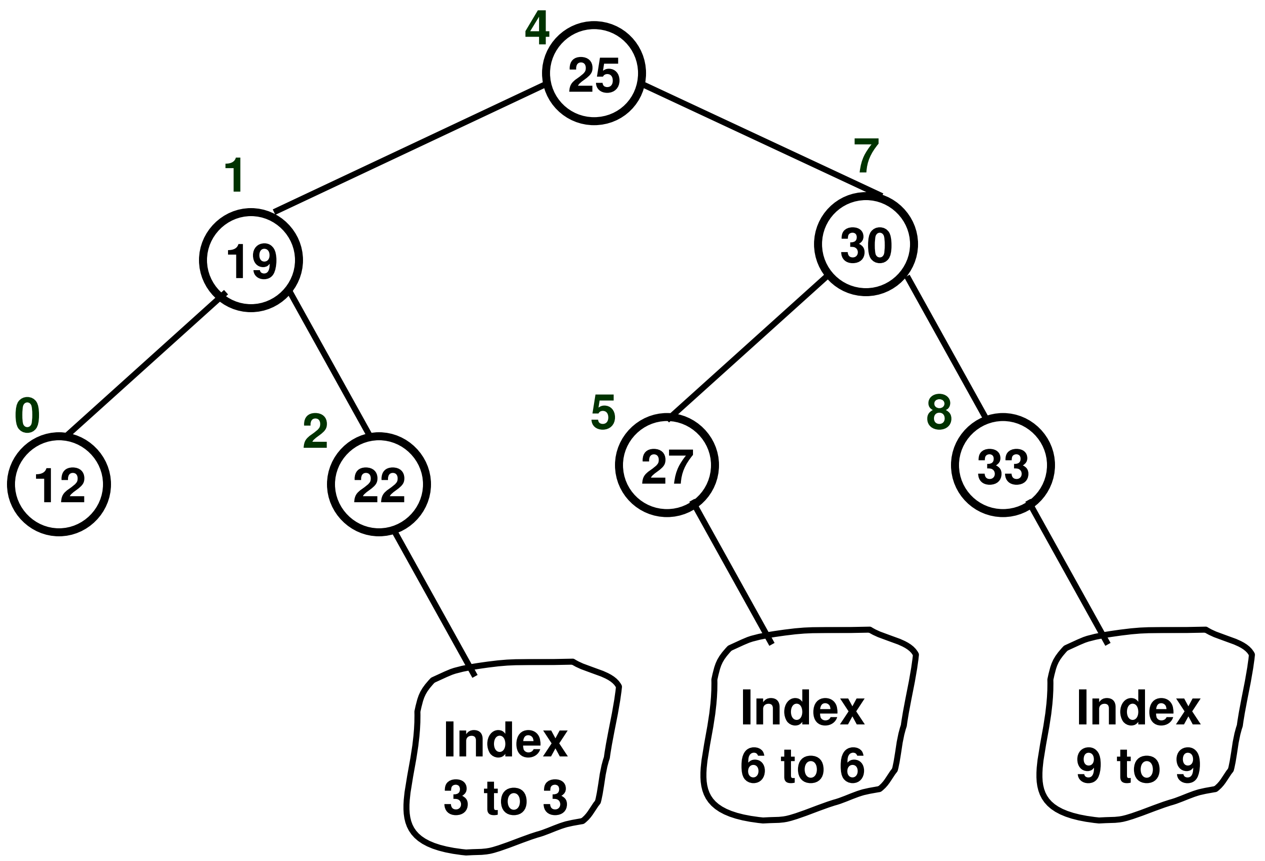
0	1	2	3	4	5	6	7	8	9
12	19	22	25	25	27	27	30	33	37





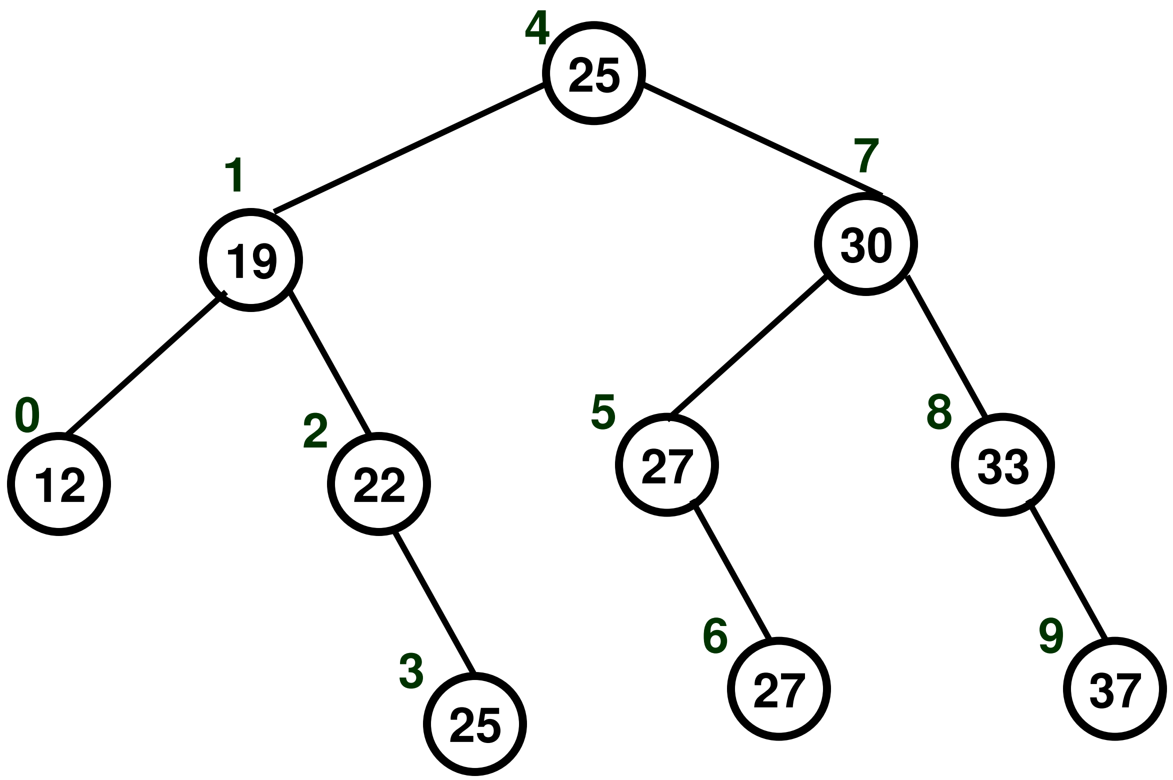
Example 2: Construction of BST

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



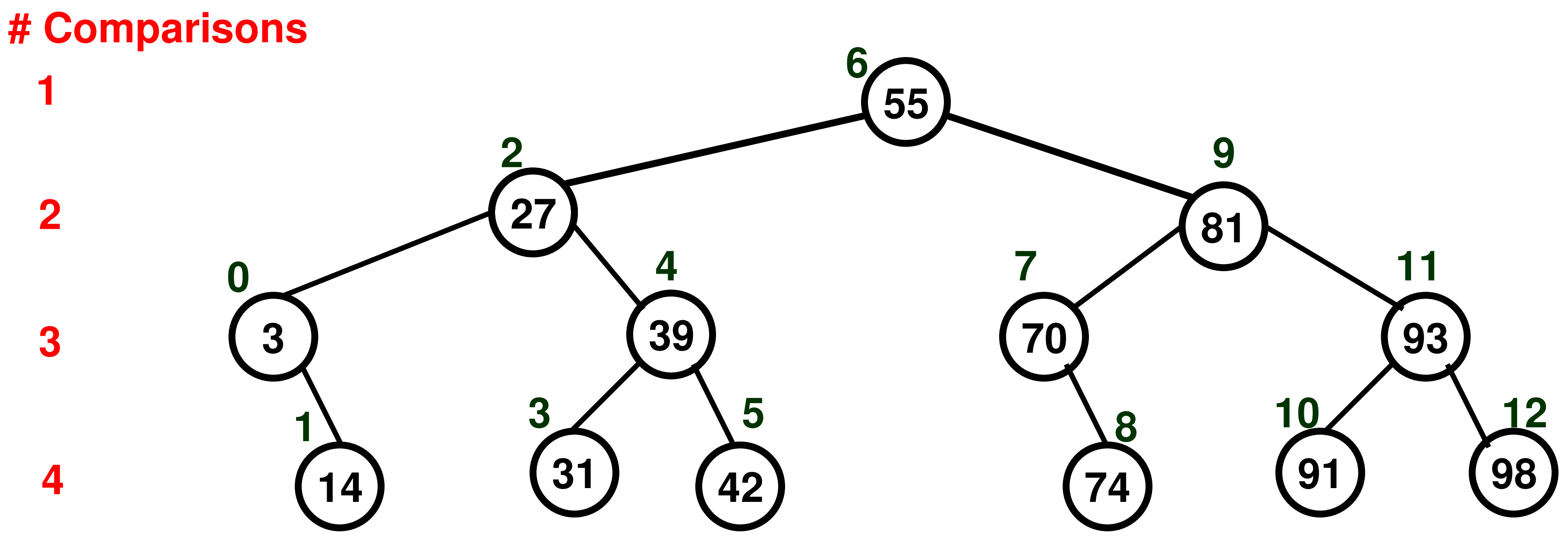
Example 2: Construction of BST

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



# Comparisons for Successful Search Example 1

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

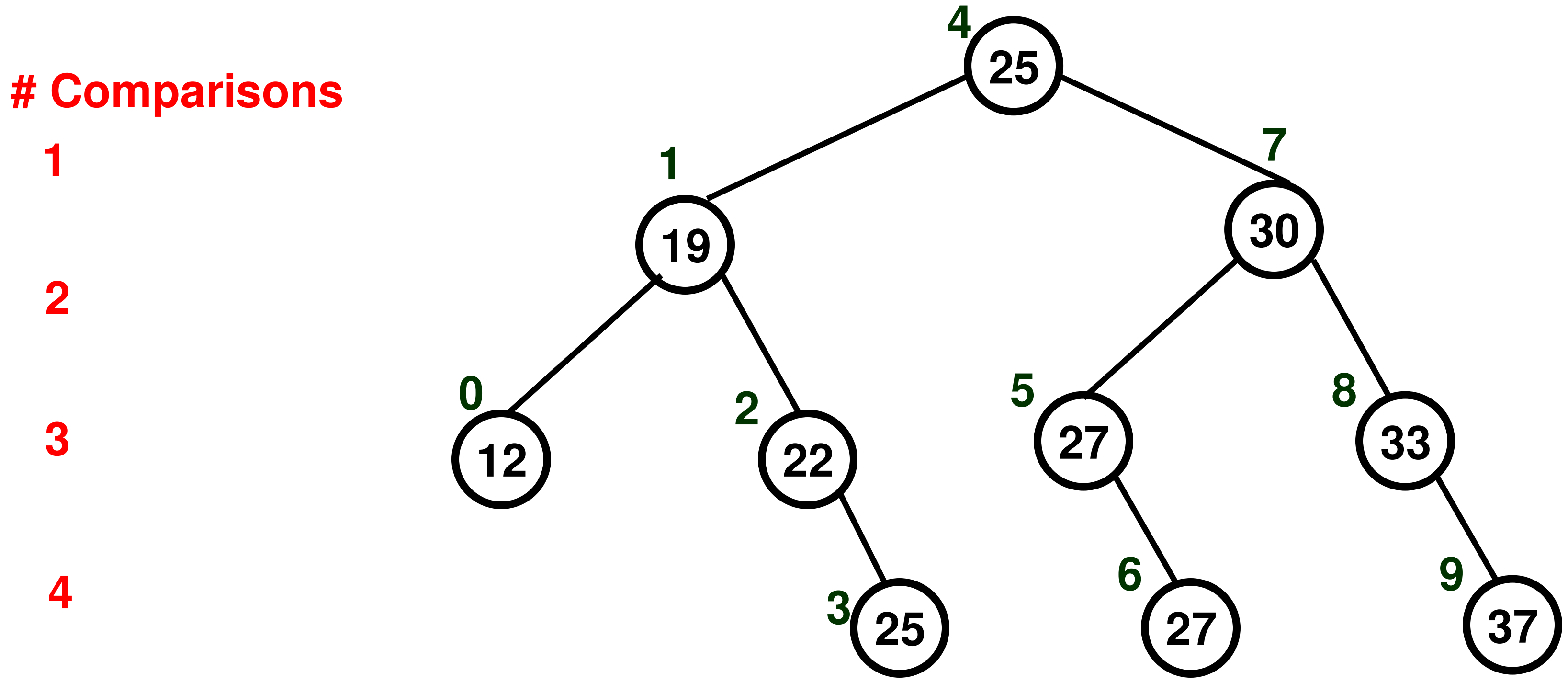


$$\frac{(1\text{ key})(1\text{ comp}) + (2\text{ keys})(2\text{ comps}) + (4\text{ keys})(3\text{ comps}) + (6\text{ keys})(4\text{ comps})}{13\text{ keys}} = 3.15$$

Note that in case of a successful search, the number of comparisons for a key is one more than the level number of the node representing the key in the BST

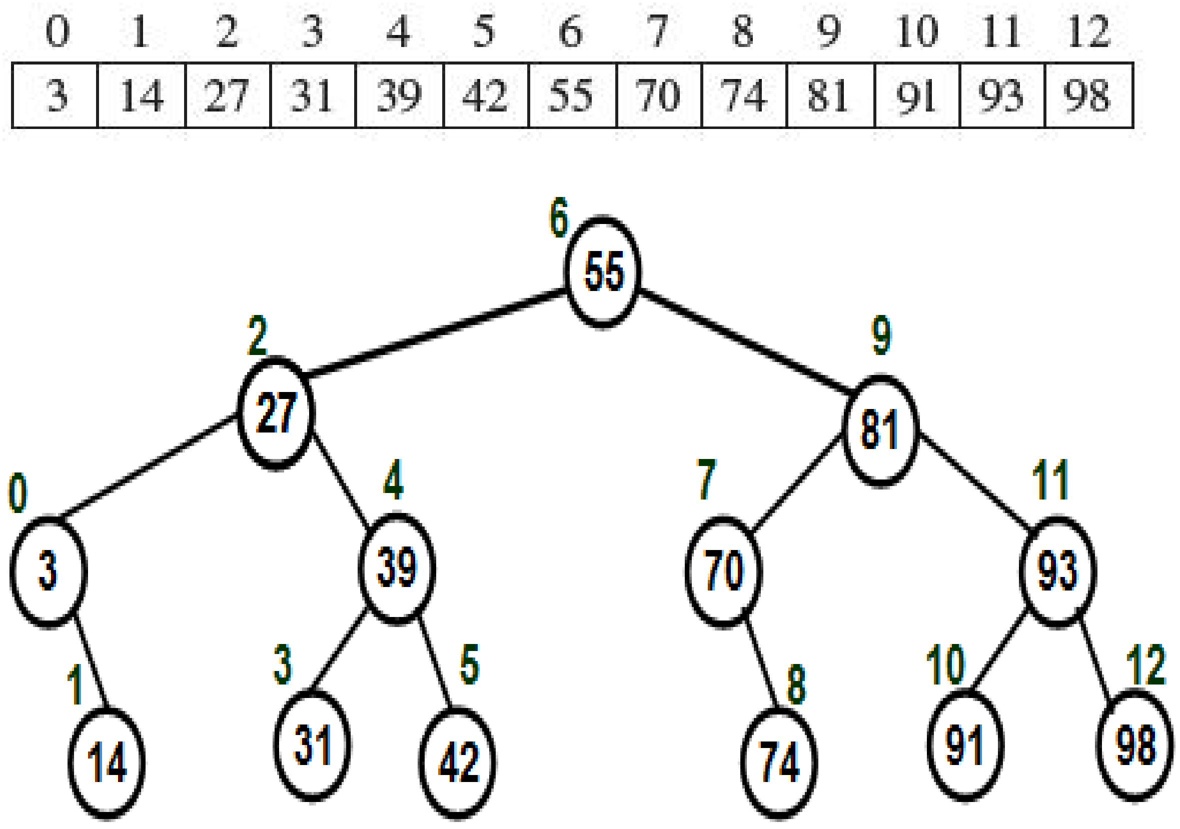
# Comparisons for Successful Search Example 2

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



$$\frac{(1\ \text{key})(1\ \text{comp}) + (2\ \text{keys})(2\ \text{comps}) + (4\ \text{keys})(3\ \text{comps}) + (3\ \text{keys})(4\ \text{comps})}{10\ \text{keys}} = 2.90$$

# Comparisons for Unsuccessful Search: Example 1



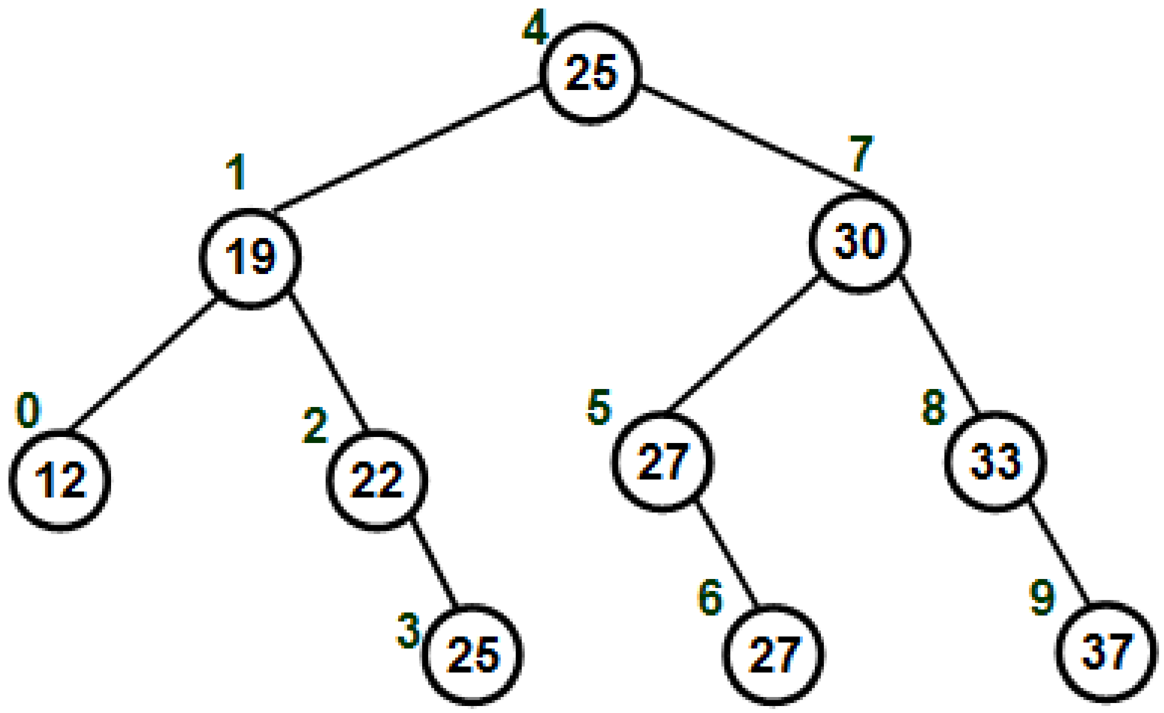
Range	# Comps
< 3	3
> 3 && < 14	4
> 14 && < 27	4
> 27 && < 31	4
> 31 && < 39	4
> 39 && < 42	4
> 42 && < 55	4
> 55 && < 70	3
> 70 && < 74	4
> 74 && < 81	4
> 81 && < 91	4
> 91 && < 93	4
> 93 && < 98	4
> 98	4

$$\text{Avg \# Comparisons for an unsuccessful search} = \frac{\text{Sum of all Comps above}}{\text{\# Ranges}}$$

= { (4*12) + (3*2) } / 14 = 3.86

# Comparisons for Unsuccessful Search Example 2

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



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

Avg # Comparisons for an unsuccessful search = $\dfrac{\text{Sum of all Comps above}}{\text{\# Ranges}}$

= { (4*4) + (3*5 } / 9 = 3.44

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

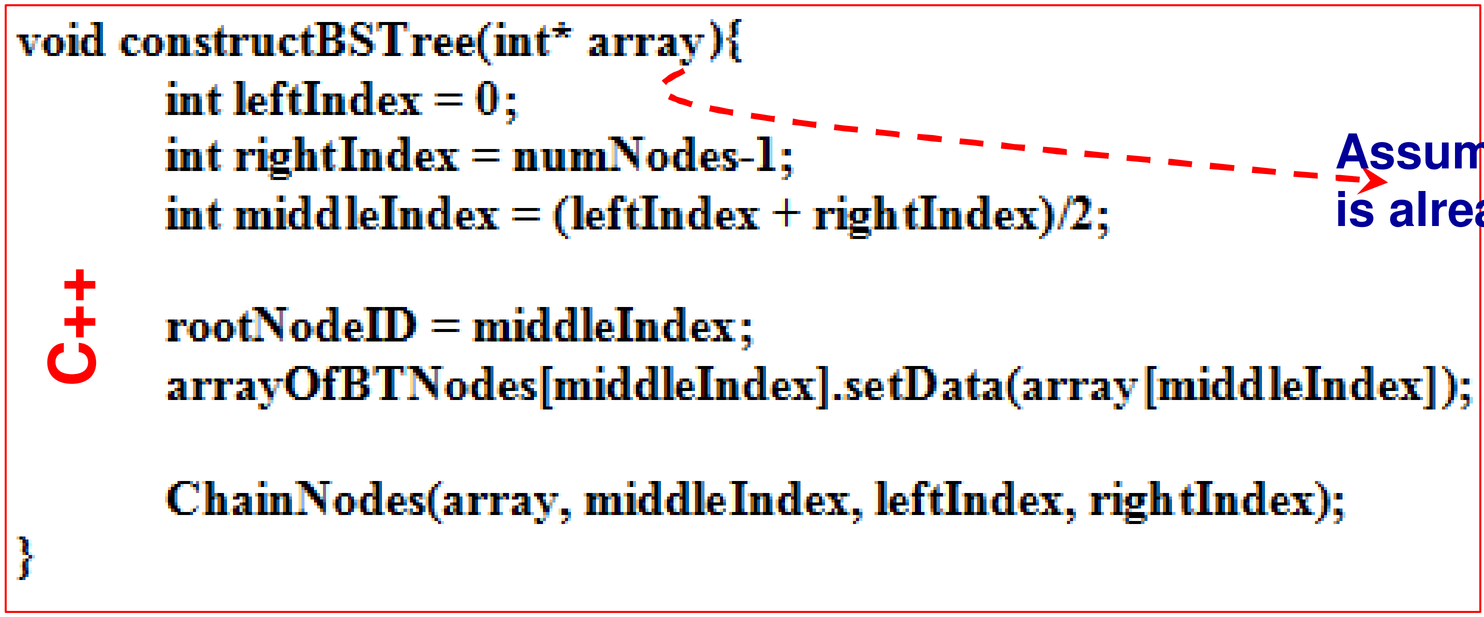
BinarySearchTree
int numNodes
BTNode* arrayOfBTNodes
int rootNodeID

```
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)



```
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);
}
```

C++

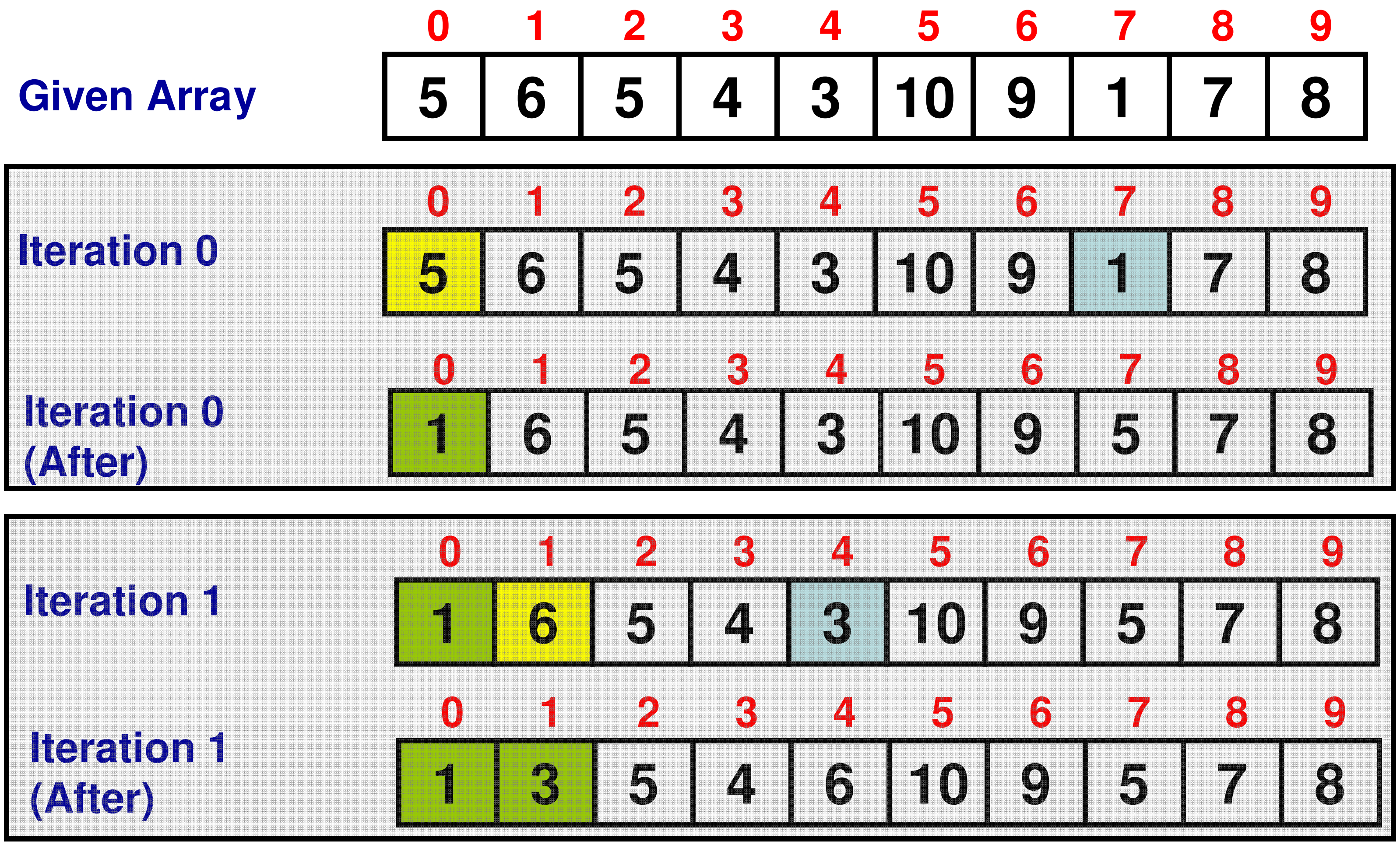
Assumes the array is already sorted

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);
        }
}
```

Sorting Algorithm: Selection Sort

- Given an array A[0…n-1], we proceed for a total of n-1 iterations)
- In iteration i (0 ≤ i < n-1), we assume A[i] is the minimum element and seek to find whether there exists an element at index i+1…n-1 so that we can swap that element with A[i], if such an element exists.



	0	1	2	3	4	5	6	7	8	9
Given Array	5	6	5	4	3	10	9	1	7	8
Iteration 0	5	6	5	4	3	10	9	1	7	8
Iteration 0 (After)	1	6	5	4	3	10	9	5	7	8
Iteration 1	1	6	5	4	3	10	9	5	7	8
Iteration 1 (After)	1	3	5	4	6	10	9	5	7	8

Iteration 2

0	1	2	3	4	5	6	7	8	9
1	3	5	4	6	10	9	5	7	8

Iteration 2 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 3

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 3 (After)

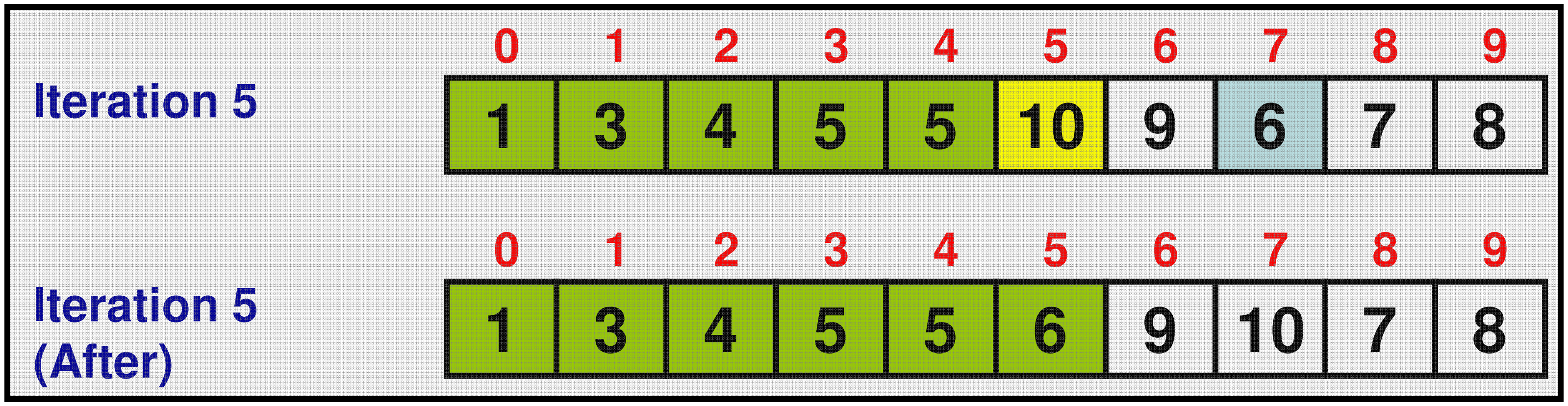
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

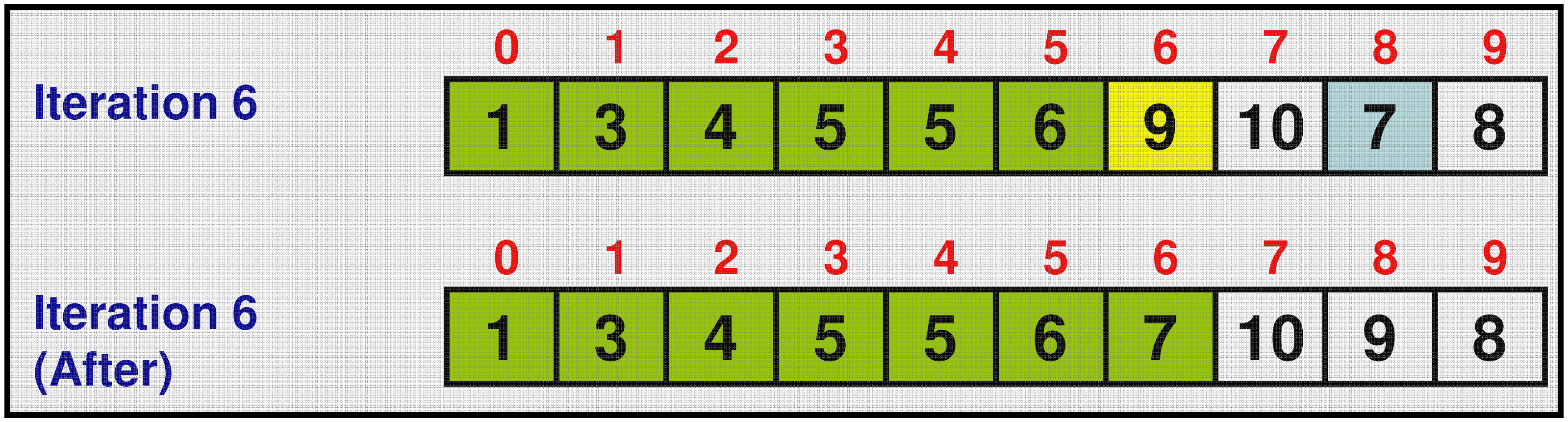
Iteration 4

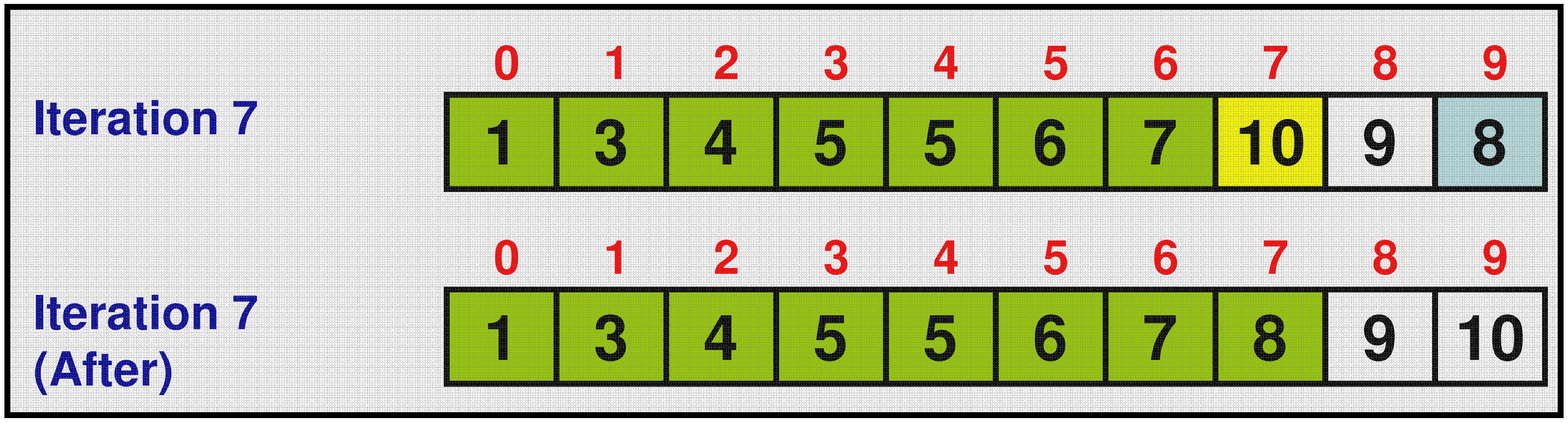
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 4 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	10	9	6	7	8


Iteration 5
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 10 9 6 7 8
Iteration 5 (After)
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 6 9 10 7 8


Iteration 6
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 6 9 10 7 8
Iteration 6 (After)
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 6 7 10 9 8


Iteration 7
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 6 7 10 9 8
Iteration 7 (After)
0 1 2 3 4 5 6 7 8 9
1 3 4 5 5 6 7 8 9 10

Iteration 8

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Iteration 8 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Final Sorted Array

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Selection Sort

ALGORITHM

```
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ to $n-2$ do
    $min \leftarrow i$
    for $j \leftarrow i+1$ to $n-1$ do
        if $A[j] < A[min]$   $min \leftarrow j$
    swap $A[i]$ and $A[min]$
```

# Comparisons

$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = \Theta(n^2)$

There is no best or worst case. In the ith Iteration, we have to find if there exists any element that is less than the element at index i.

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;
    }
}
```

Selection Sort: Example 2

Given Array

0	1	2	3	4	5
8	9	1	3	2	7

It # 0

0	1	2	3	4	5
8	9	1	3	2	7

It # 0 (After)

0	1	2	3	4	5
1	9	8	3	2	7

It # 1

0	1	2	3	4	5
1	9	8	3	2	7

It # 1 (After)

0	1	2	3	4	5
1	2	8	3	9	7

It # 2

0	1	2	3	4	5
1	2	8	3	9	7

It # 2 (After)

0	1	2	3	4	5
1	2	3	8	9	7

It # 3

0	1	2	3	4	5
1	2	3	8	9	7

It # 3 (After)

0	1	2	3	4	5
1	2	3	7	9	8

It # 4

0	1	2	3	4	5
1	2	3	7	9	8

It # 4 (After)

0	1	2	3	4	5
1	2	3	7	8	9

Final Sorted Array

0	1	2	3	4	5
1	2	3	7	8	9

Selection Sort: Example 3 (How to show the work in an exam)

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 1		**1**	**5**	**12**	**4**	**18**	**9**	**7**	**15**
Iteration 2		**1**	**4**	**12**	**5**	**18**	**9**	**7**	**15**
Iteration 3		**1**	**4**	**5**	**12**	**18**	**9**	**7**	**15**
Iteration 4		**1**	**4**	**5**	**7**	**18**	**9**	**12**	**15**
Iteration 5		**1**	**4**	**5**	**7**	**9**	**18**	**12**	**15**
Iteration 6		**1**	**4**	**5**	**7**	**9**	**12**	**18**	**15**
Iteration 7		**1**	**4**	**5**	**7**	**9**	**12**	**15**	**18**

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

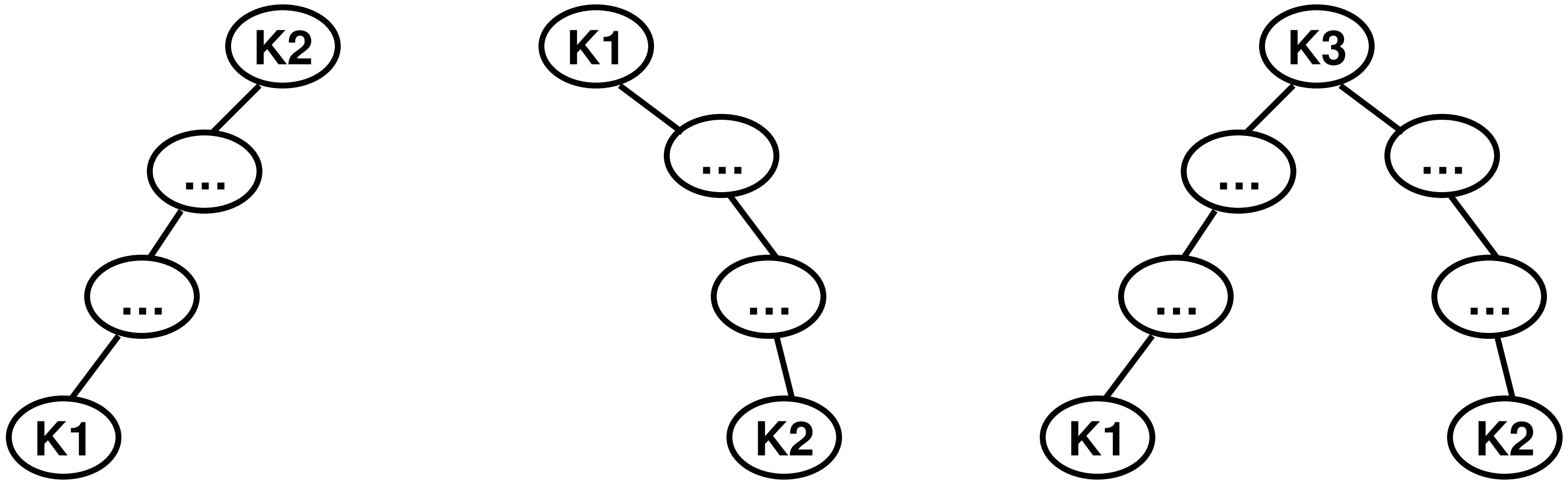
```
int numElements;
cout << "Enter the number of elements: ";
cin >> numElements;

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);
```

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

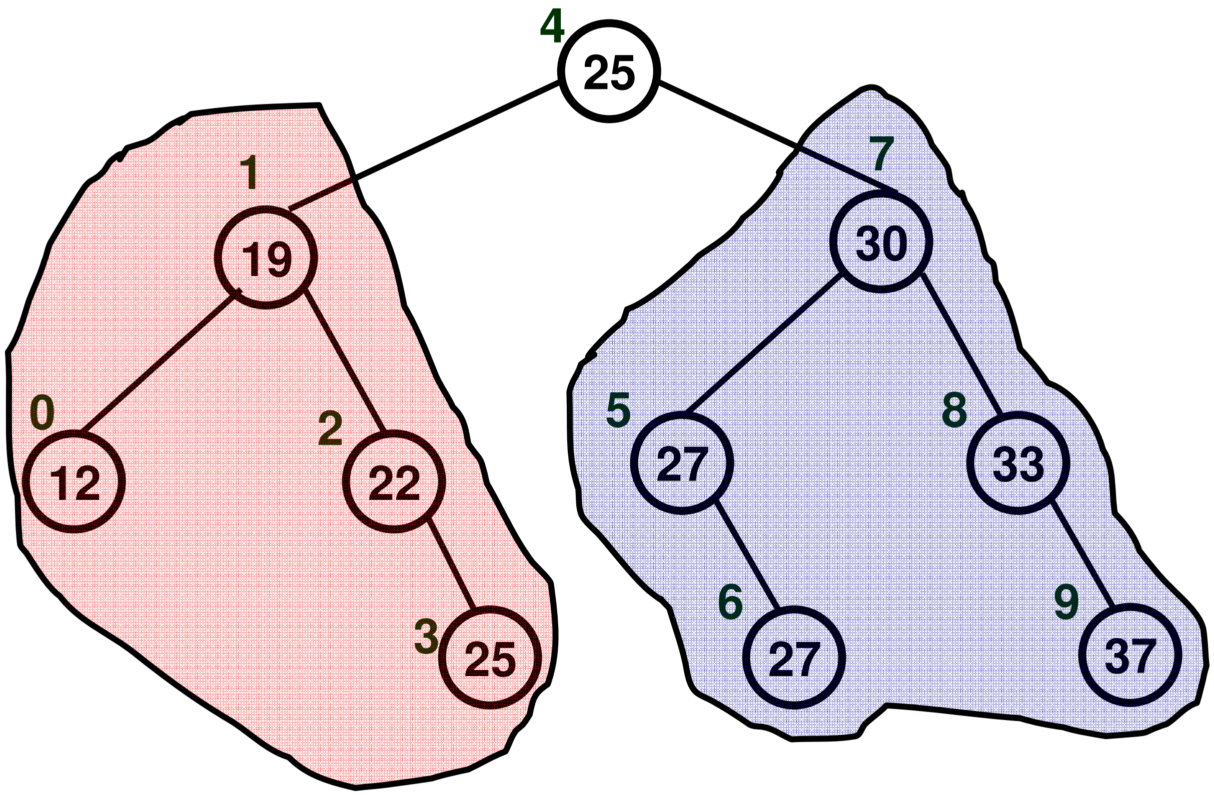
```
int getKeyIndex(int searchKey){

        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;
}
```

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.



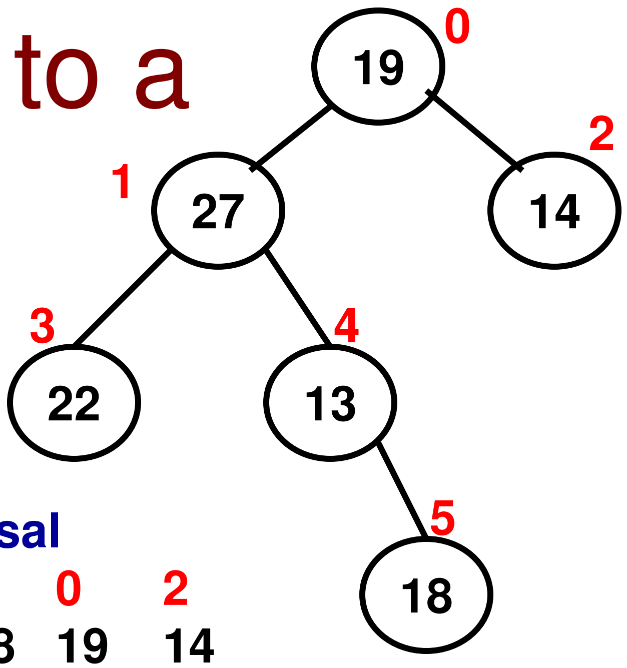
inorder Traversal of a BST



{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)



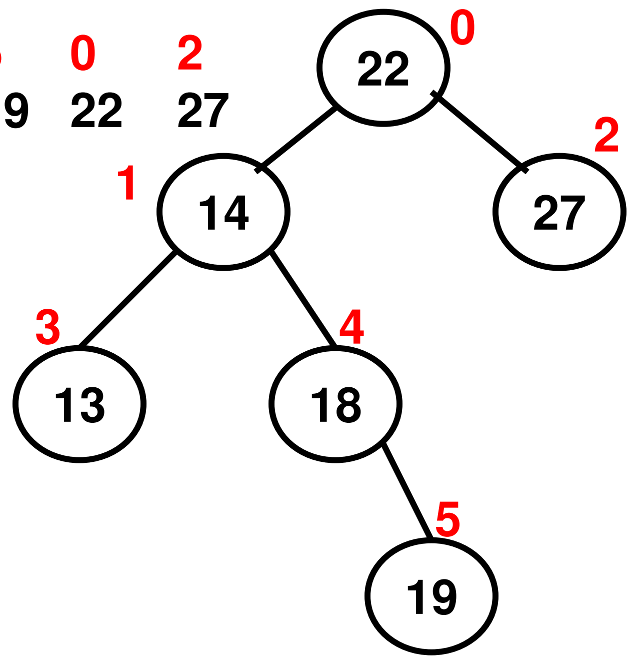
- Do an inorder traversal of the given binary tree and get an array of data corresponding to the nodes of the tree in the order they are visited (i.e., the index entries of the nodes)
- Sort the data using a sorting algorithm
- Do an inorder traversal of the binary tree again. For each node that is about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

inorder Traversal

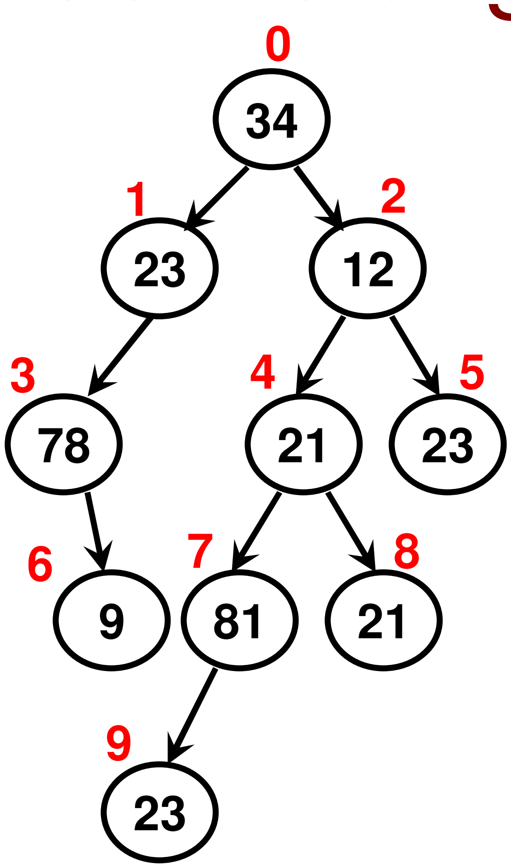
3	1	4	5	0	2
22	27	13	18	19	14

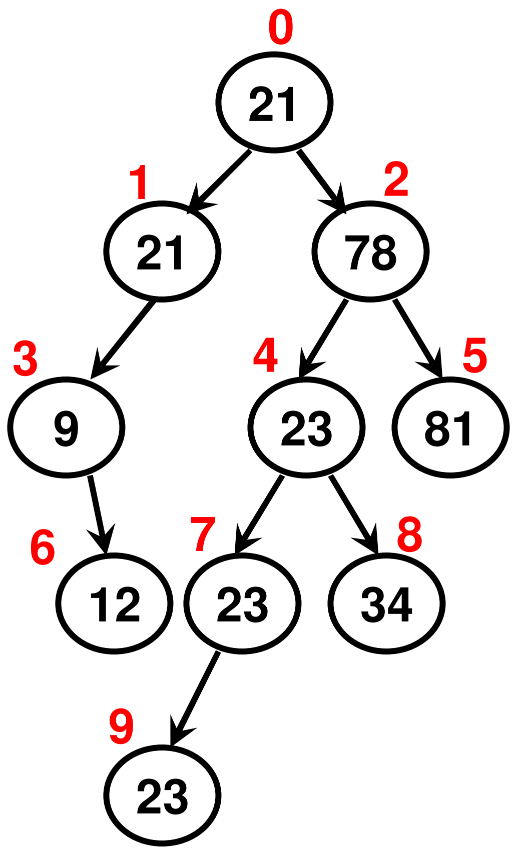
Sorted Order

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