SEARCHING

BINARY SEARCH TREE

Binary search tree is a binary tree (where each node has at most two children) in which a new node is added at the left of the tree if it is smaller and to the right if it is bigger in value. Example:

Create a binary tree (binary search tree) from the following values:

23, 10, 12, 5, 4, 91, 18, 2, 28 First value = 23

Let us make it the root

Second value = 10 which is less than 23 Therefore, it will be positioned to the left of the root (23). Third value = 12. This is less than 23 and hence will be moved to its left. But 12 is greater than 10, and will be moved to its right. Now next value is 5; 5 is less than 23, that is, root → value. Therefore, we move to left, that is, ptr = ptr → left Now ptr → value = 10. But 5 is less than ptr → value. Therefore, we create a new node on the left of 10.

Next value is 4 moving in the same way as we started

Now we encounter 91 which is

> root → value

Therefore, we create a new node in the right

Now next is 18

Now next is 2

and finally we have 28.

Figure below shows the complete binary search tree.

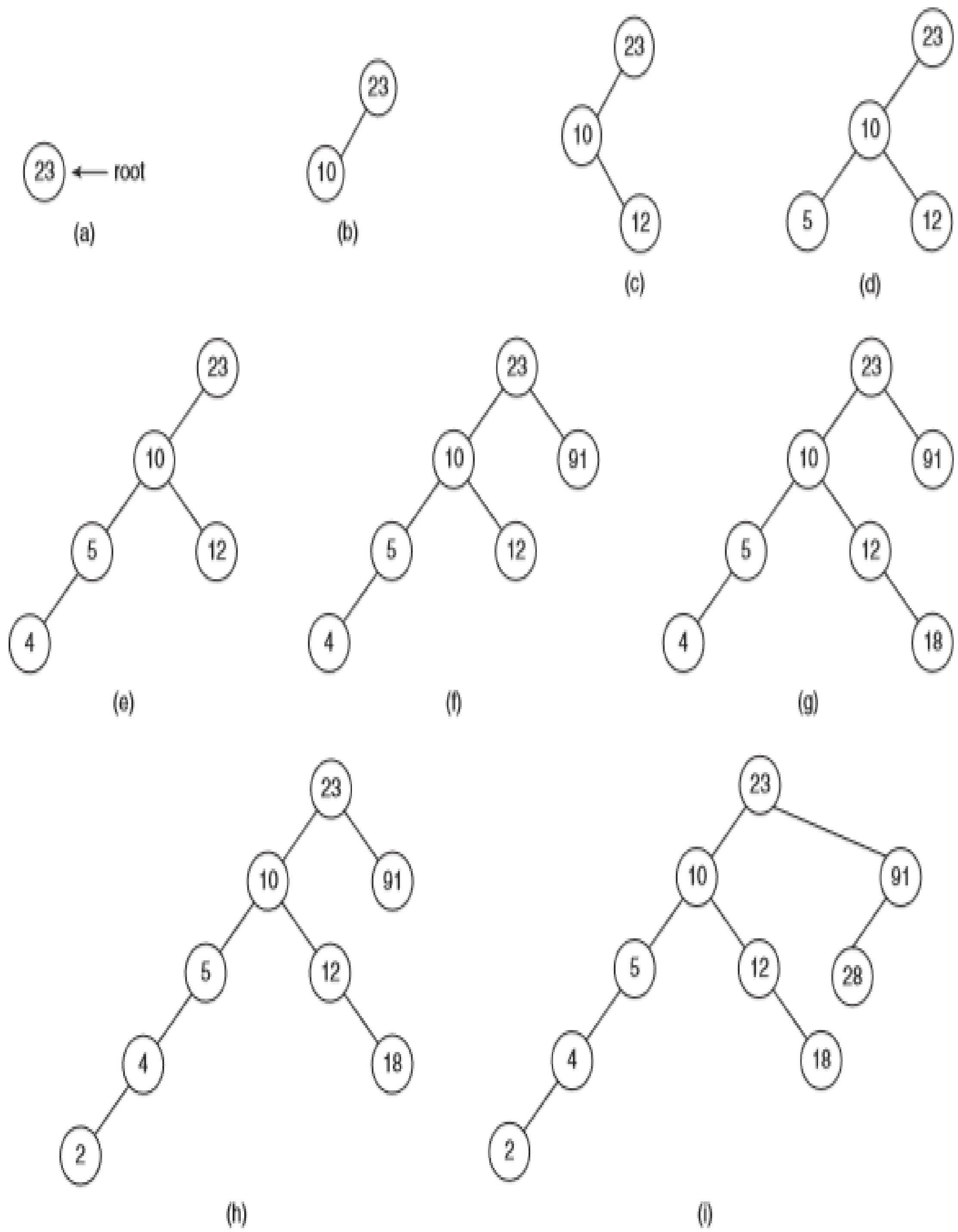


Figure Show Creation of a binary search tree

Algorithm: Creation of binary search tree

Input: Binary search tree and the item which needs to be inserted
Output: The tree with an additional element.

```
Add node (x,root)

{
ptr=root;
if (ptr = = NULL)
    {
     Create a new node;
     node→value = x;
     node→left = node→right = NULL;
}
else
 {
 If (node→value < x)
    {
    Ptr = node→left
    if (ptr = NULL) create
    new node and new node→value = x
    else add node (x, ptr)
    }
else
    {
    ptr = node→right;
    if (ptr = NULL) create new node & new node→value = x
    else
    add node (x, ptr);
    }
}
}  end of algorithm.
```

An Illustration for algorithm

Algorithm BinarySearch(A, first, last , key)

- Input: A , an sorted array , **a[0]≤a[1]≤a[2]≤. . . ≤ a[finalIndex]**

and n>= 1, the number of elements. The range of indexes is 0,....n-1, and key is the searched value.

Output: the index of searched value if found in the Array or return -1 if not found.

```
If   first > last   return -1
else
 {
     mid← (first+last)/2
      If (key  equal  A[mid] )   return mid
     Else   if ( key  < A[mid]  ) return  BinarySearch(A, first, mid-1 , key)
     Else   return  BinarySearch(A, mid+1, last , key)
 }
```

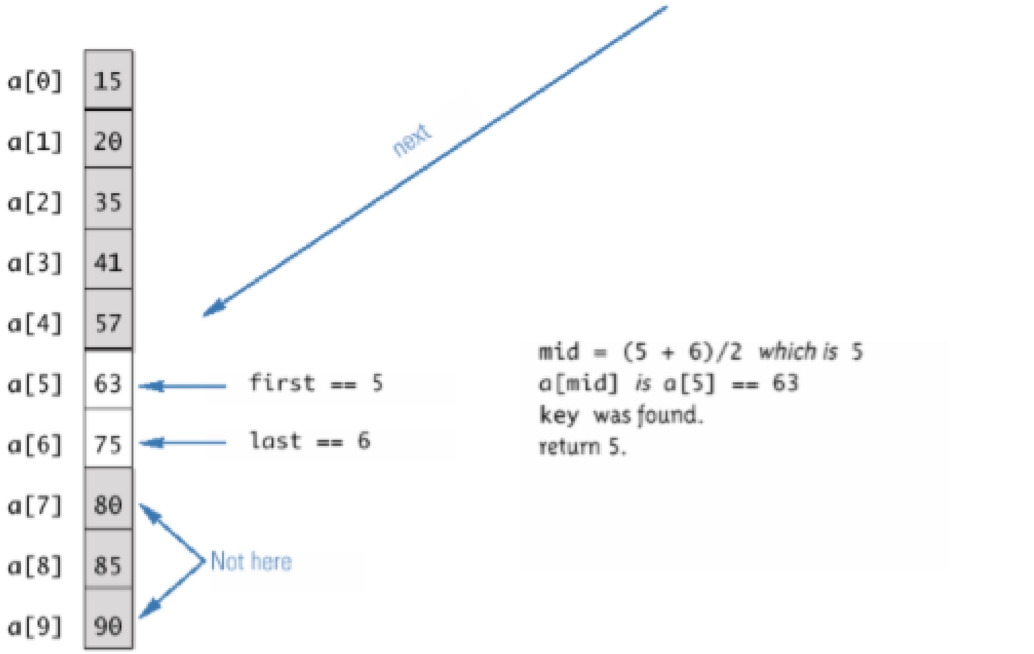
Efficiency of Binary Search

- The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order .

  Around half the array is eliminated from consideration right at the start . Then a quarter of the array, then an eighth of the array, and so forth.

- The binary search algorithm has a worst-case running time that is logarithmic: O(log n).

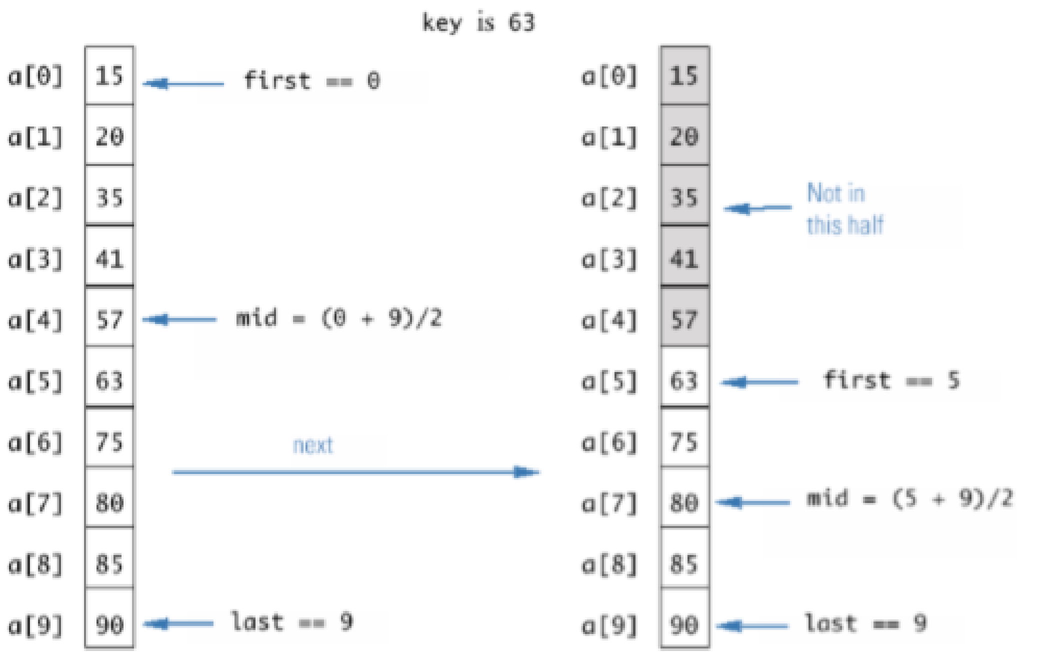
Display 11.7 **Execution of the Method search** (continued)



Example: use the binary search algorithm to search number 63 in the following list :

[15, 20 ,35,41,57,63,75,80,85,90] :

Display 11.7 **Execution of the Method search**



The sequential search algorithm on an array is :

–Sequentially scan the array, comparing each array item with the searched value .

–If a match is found; return the index of the matched element; otherwise return –1 .

Note: sequential (linear) search can be applied to both sorted and unsorted arrays.

Algorithm SequentialSearch(A,n,x) (unsorted list)

Input: A , an array , and n>= 1, the number of elements. The range of indexes is 0,....n-1.and x is the searched value.

Output: the index of searched value if found in the Array or return -1 if not found.

```
For  i ← 0 to n
        if A[i] equals x
                return i
      return -1;
```

Analysis of SequentialSearch algorithm

Best case O(1) and Worst case O(n)

Binary Search

- Binary search uses a recursive method to search an array to find a specified value.
- The array must be a sorted array: **a[0]≤a[1]≤a[2]≤. . . ≤ a[finalIndex]**
- If the value is found, its index is returned
- If the value is not found, -1 is returned

Algorithm SequentialSearch(A,n,x) (sorted list)

Input: A , an sorted array , and n>= 1, the number of elements. The range of indexes is 0,....n-1.and x is the searched value.

Output: the index of searched value if found in the Array or return -1 if not found.

```
For  i ← 0 to n
  {
        if A[i] equals x
                    return i
        if  A[i] > x   return -1
  }
      return -1;
```

Note: Each execution of the recursive method reduces the search space by about a half.

An algorithm to solve this task looks at the middle of the array or array segment first

- If the value looked for is smaller than the value in the middle of the array
  - Then the second half of the array or array segment can be ignored
  - This strategy is then applied to the first half of the array or array segment
- If the value looked for is larger than the value in the middle of the array or array segment
  - Then the first half of the array or array segment can be ignored
  - This strategy is then applied to the second half of the array or array segment
- If the value looked for is at the middle of the array or array segment, then it has been found
- If the entire array (or array segment) has been searched in this way without finding the value, then it is not in the array