```
struct BinarySearchTreeNode *kthSmallestInBST(struct BinarySearchTreeNode *root, int k, int *count){
        if(!root) return NULL;
        struct BinarySearchTreeNode *left = kthSmallestInBST(root→left, k, count);
        if( left ) return left;
        if(++count == k)
                return root;
        return kthSmallestInBST(root→right, k, count);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problem-60 **Floor and ceiling:** If a given key is less than the key at the root of a BST then floor of key (the largest key in the BST less than or equal to key) must be in the left subtree. If key is greater than the key at the root then floor of key could be in the right subtree, but only if there is a key smaller than or equal to key in the right subtree; if not (or if key is equal to the key at the root) then the key at the root is the floor of key. Finding the ceiling is similar with interchanging right and left.For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then

For $x = 0$: floor doesn't exist in array, ceil = 1, For $x = 1$: floor = 1, ceil = 1

For $x = 5$: floor = 2, ceil = 8, For $x = 20$: floor = 19, ceil doesn't exist in array

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the values being visited. If the roots data is greater than the given value then return the previous value which we have maintained during traversal. If the roots data is equal to the given data then return root data.

```
struct BinaryTreeNode *FloorInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return FloorInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *FloorInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
          if(!root)
                  return NULL;
          if(!FloorInBSTUtil(root→left, prev, data))
                 return 0;
          if(root→data == data)
                  return root;
          if(root→data > data)
                  return prev;
          prev = root;
          return FloorInBSTUtil(root→right, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

For ceiling, we just need to call the right subtree first and then followed by left subtree.

```
struct BinaryTreeNode *CeilingInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return CeilingInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *CeilingInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
          if(!root)
                  return NULL;
          if(!CeilingInBSTUtil(root→right, prev, data))
                   return 0;
          if(root→data == data) return root;
```

```
        if(root→data < data)  return prev;
        prev = root;
        return CeilingInBSTUtil(root→left, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-61 Give an algorithm for finding the union and intersection of BSTs. Assume parent pointers are available (say threaded binary trees). Also, assume the lengths of two BSTs are m and n respectively.

Solution: If parent pointers are available then the problem is same as merging of two sorted lists. This is because if we call inorder successor each time we get the next highest element. It's just a matter of which InorderSuccessor to call.

Time Complexity: $O(m + n)$. Space complexity: $O(1)$.

Problem-62 For Problem-61, what if parent pointers are not available?

Solution: If parent pointers are not available then, one possibility is converting the BSTs to linked lists and then merging.

1 Convert both the BSTs into sorted doubly linked lists in $O(n + m)$ time. This produces 2 sorted lists.
2 Merge the two double linked lists into one and also maintain the count of total elements in $O(n + m)$ time.
3 Convert the sorted doubly linked list into height balanced tree in $O(n + m)$ time.

Problem-63 For Problem-61, is there any alternative way of solving the problem?

Solution: Yes, using inorder traversal.

- Perform inorder traversal on one of the BST.
- While performing the traversal store them in table (hash table).
- After completion of the traversal of first *BST*, start traversal of the second *BST* and compare them with hash table contents.

Time Complexity: $O(m + n)$. Space Complexity: $O(Max(m, n))$.

Problem-64 Given a *BST* and two numbers $K1$ and $K2$, give an algorithm for printing all the elements of *BST* in the range $K1$ and $K2$.

Solution:

```
void RangePrinter(struct BinarySearchTreeNode *root, int  K1, int K2) {
        if(root == NULL)
                return;
        if(root→data >= K1)
                RangePrinter(root→left, K1, K2);
         if(root→data >= K1 && root→data <= K2)
                        printf("%d", root→data);
        if(root→data <= K2)
                RangePrinter(root→right, K1, K2);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-65 For Problem-64, is there any alternative way of solving the problem?

Solution: We can use level order traversal: while adding the elements to queue check for the range.

```
void  RangeSeachLevelOrder(struct BinarySearchTreeNode *root, int  K1, int K2){
        struct BinarySearchTreeNode *temp;
        struct Queue *Q = CreateQueue();
        if(!root)
                return NULL;
        Q = EnQueue(Q, root);
```

```
        while(!IsEmptyQueue(Q)) {
                temp=DeQueue(Q);
                if(temp→data >= K1 && temp→data <= K2)
                        printf("%d",temp→data);
                if(temp→left && temp→data >= K1)
                        EnQueue(Q, temp→left);
                if(temp→right && temp→data <= K2)
                        EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
        return NULL;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for queue.

Problem-66 For Problem-64, can we still think of alternative way for solving the problem?

Solution: First locate $K1$ with normal binary search and after that use InOrder successor until we encounter $K2$. For algorithm, refer problems section of threaded binary trees.

Problem-67 Given root of a Binary Search tree, trim the tree, so that all elements in the new tree returned are between the inputs A and B.

Solution: It's just another way of asking the Problem-64.

Problem-68 Given two BSTs, check whether the elements of them are same or not. For example: two BSTs with data 10 5 20 15 30 and 10 20 15 30 5 should return true and the dataset with 10 5 20 15 30 and 10 15 30 20 5 should return false. **Note:** BSTs data can be in any order.

Solution: One simple way is performing a traversal on first tree and storing its data in hash table. As a second step perform traversal on second tree and check whether that data is already there in hash table or not. During the traversal of second tree if we find any mismatch return false.

Time Complexity: $O(max(m,n))$, where m and n are the number of elements in first and second BST. Space Complexity: $O(max(m,n))$. This depends on the size of the first tree.

Problem-69 For Problem-68, can we reduce the time complexity?

Solution: Instead of performing the traversals one after the other, we can perform $in-order$ traversal of both the trees in parallel. Since the $in-order$ traversal gives the sorted list, we can check whether both the trees are generating the same sequence or not.

Time Complexity: $O(max(m,n))$. Space Complexity: $O(1)$. This depends on the size of the first tree.

Problem-70 For the key values $1 \ldots n$, how many structurally unique BSTs are possible that store those keys.

Solution: Strategy: consider that each value could be the root. Recursively find the size of the left and right subtrees.

```
int CountTrees(int n) {
        if (n <= 1)  return 1;
        else {  // there will be one value at the root, with whatever remains on the left and right
                // each forming their own subtrees.Iterate through all the values that could be the root...
                int sum = 0;
                int left, right, root;
                for (root=1; root<=n; root++) {
                        left = CountTrees(root - 1);
                        right = CountTrees(numKeys - root);
                        // number of possible trees with this root == left*right
                        sum += left*right;
```

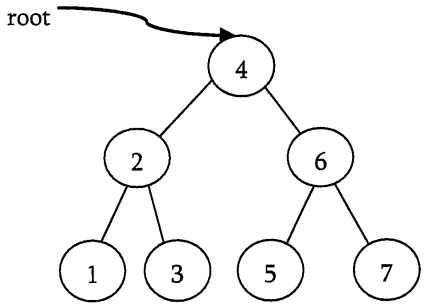
```
            }
            return(sum);
        }
}
```

6.12 Balanced Binary Search Trees

In earlier sections we have seen different trees whose worst case complexity is $O(n)$, where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to $O(logn)$ by imposing restrictions on the heights. In general, the height balanced trees are represented with $HB(k)$, where k is the difference between left subtree height and right subtree height. Sometimes k is called balance factor.

Complete Balanced Binary Search Trees

In $HB(k)$, if $k = 0$ (if balance factor is zero), then we call such binary search trees as $full$ balanced binary search trees. That means, in $HB(0)$ binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Note: For constructing $HB(0)$ tree refer problems section.

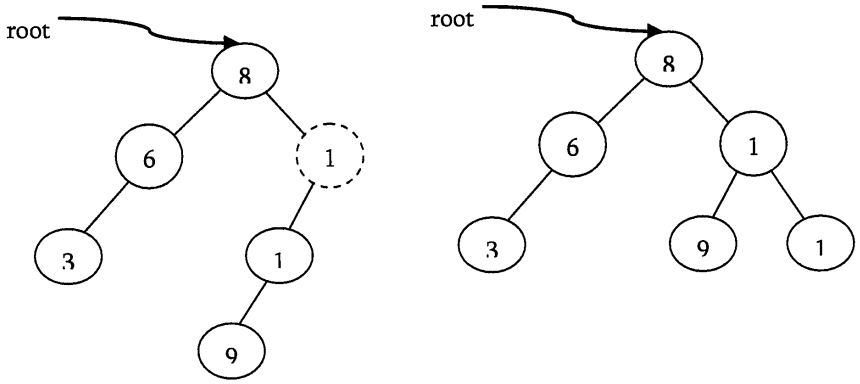
6.13 AVL (Adelson-Velskii and Landis) Trees

In $HB(k)$, if $k = 1$ (if balance factor is one), such binary search tree is called an AVL tree. That means an AVL tree is a binary search tree with a *balance* condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.



As an example among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get minimum number of nodes with height h, we should fill the tree with as minimum nodes as possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

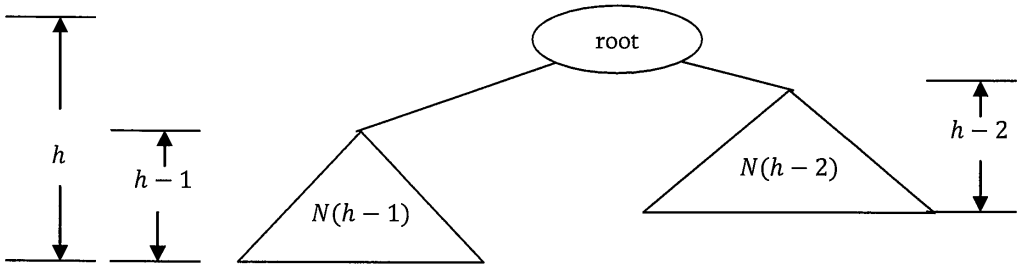
$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44 logn \approx O(logn)$$



Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(logn)$. Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get

$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is $O(logn)$.

AVL Tree Declaration

Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we include the height also as part of declaration.

```
struct AVLTreeNode{
        struct AVLTreeNode *left;
        int data;
        struct AVLTreeNode *right;
        int height;
};
```

Finding Height of an AVL tree

```
int Height(struct AVLTreeNode *root ){
        if( !root)  return -1;
        else return root→height;
}
```

Time Complexity: O(1).

Rotations

When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves

adding/deleting a single node, this can only increase/decrease the height of some subtree by 1. So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. That means, we need to apply the rotations for the node X.

Observation: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to root of the tree. While moving to root, we need to consider the first node whichever is not satisfying the AVL property. From that node onwards every node on the path to root will have the issue. Also, if we fix the issue for that first node, then all other nodes on the path to root will automatically satisfy the AVL tree property. That means we always need to care for the first node whichever is not satisfying the AVL property on the path from insertion point to root and fix it.

Types of Violations

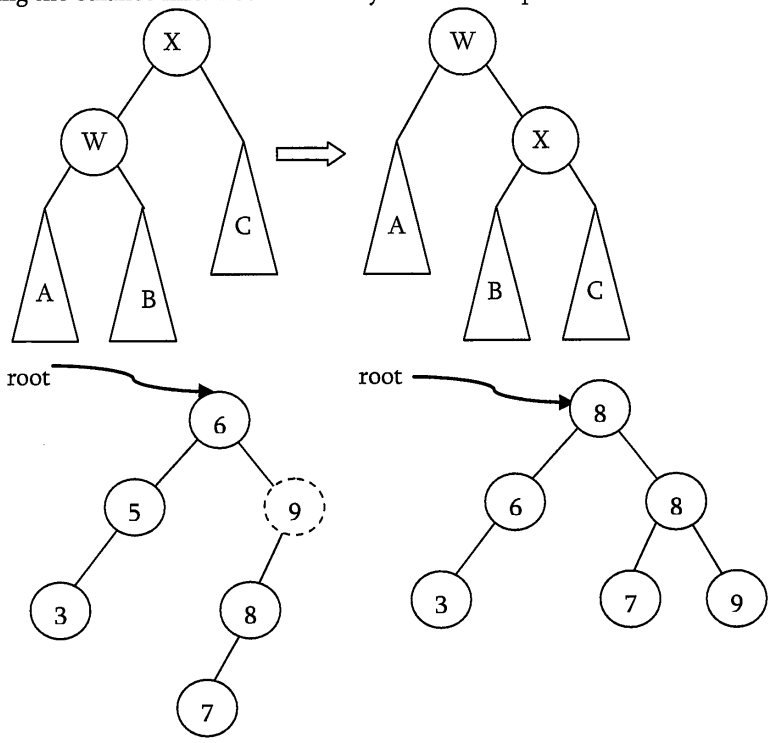
Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that X's two subtrees' heights differ by two. We can easily observe that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of X.
2. An insertion into the right subtree of the left child of X.
3. An insertion into the left subtree of the right child of X.
4. An insertion into the right subtree of the right child of X.

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (needs two single rotations).

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the below case, at node X, the AVL tree property is not satisfying. As discussed earlier, rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

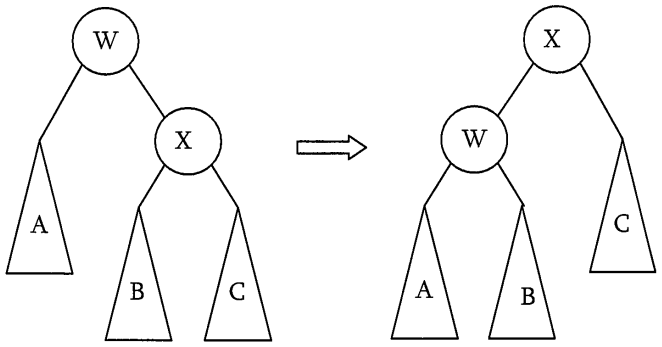


For example, in above figure, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

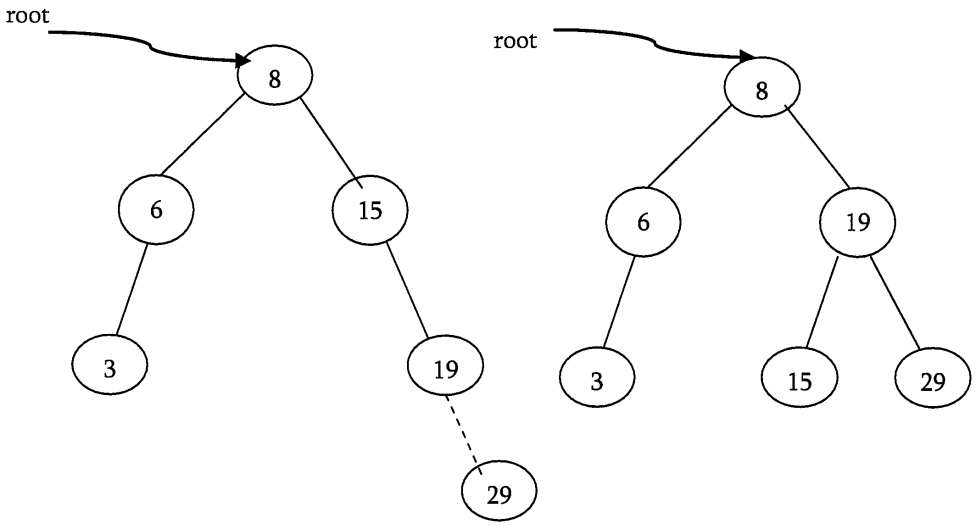
```
struct AVLTreeNode *SingleRotateLeft(struct AVLTreeNode *X ){
        struct AVLTreeNode *W = X→left;
        X→left = W→right;
        W→right = X;
        X→height = max( Height(X→left), Height(X→right) ) + 1;
        W→height = max( Height(W→left), X→height ) + 1;
        return W; /* New root */
}
```

Time Complexity: O(1). Space Complexity: O(1).

Right Right Rotation (RR Rotation) [Case-4]: In this case, the node *X* is not satisfying the AVL tree property.



For example, in above figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

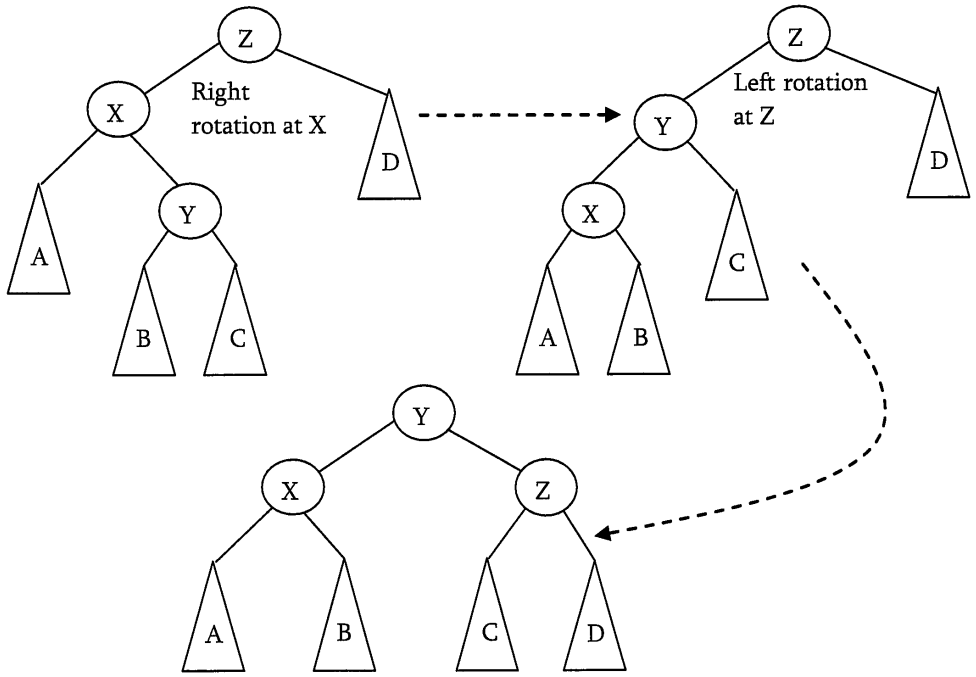



```
struct AVLTreeNode *SingleRotateRight(struct AVLTreeNode *W ) {
        struct AVLTreeNode *X  = W→right;
        W→right = X→left;
        X→left = W;
        W→height = max( Height(W→right), Height(W→left) ) + 1;
        X→height = max( Height(X→right), W→height) + 1;
        return X;
```

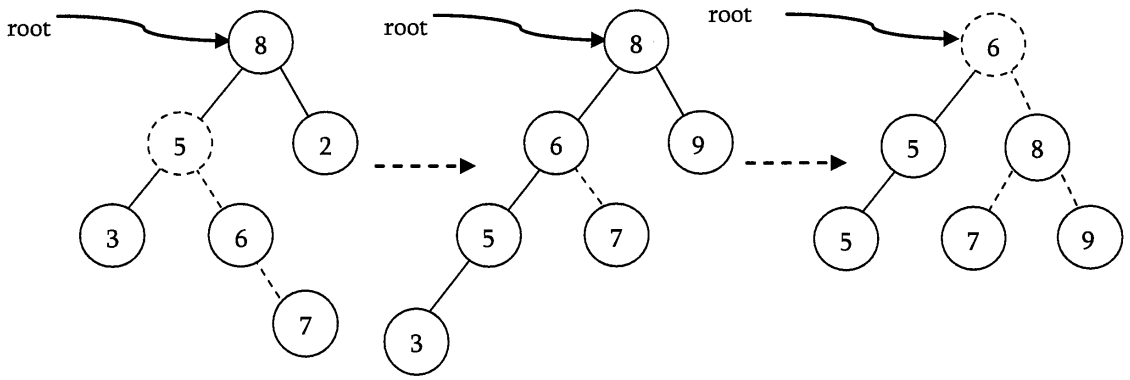
}
Time Complexity: O(1). Space Complexity: O(1).

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



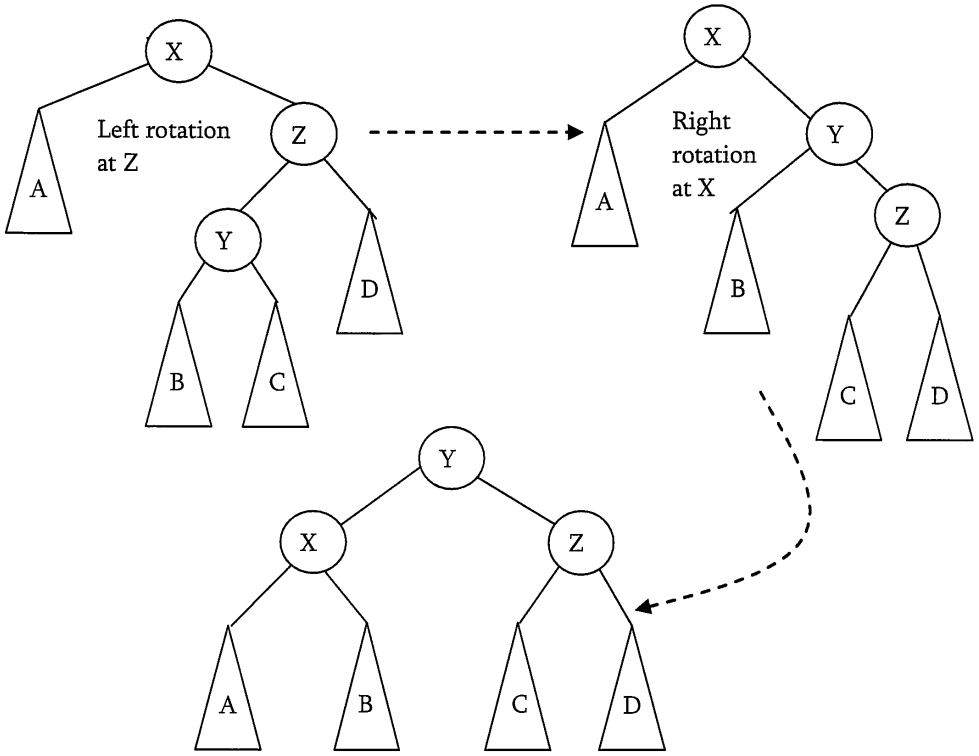
As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right side tree is the one after double rotation.



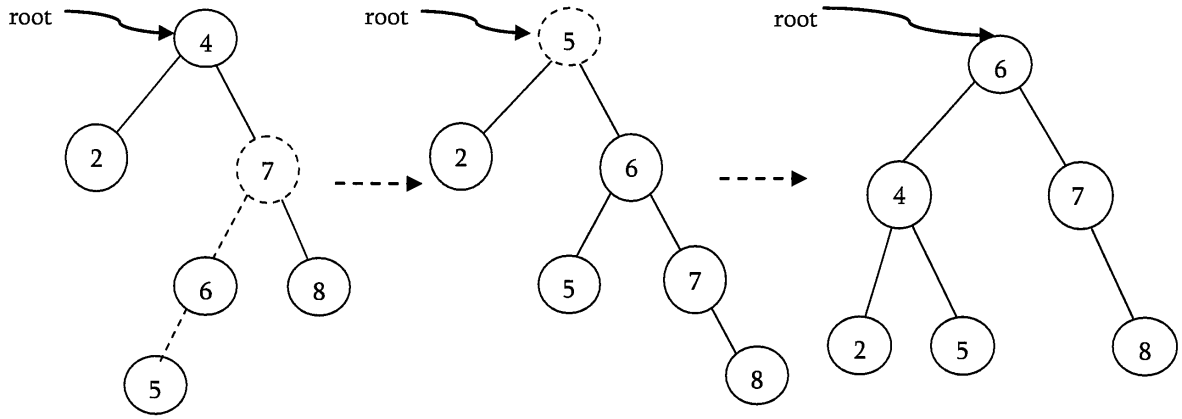
Code for left-right double rotation can be given as:

```
struct AVLTreeNode *DoubleRotatewithLeft( struct AVLTreeNode *Z ){
        Z→left = SingleRotateRight( Z→left );
        return SingleRotateLeft(Z);
}
```

Right Left Rotation (RL Rotation) [Case-3]: As similar to case-2, we need to perform two rotations for fixing this scenario.



As an example, let us consider the following tree: Insertion of 6 is creating the case-3 scenario and right side tree is the one after double rotation.



Insertion into an AVL tree

Insertion in AVL tree is very much similar to BST insertion. After inserting the element, we just need to check whether there is any height imbalance. If there is any imbalance, call the appropriate rotation functions.

```
struct AVLTreeNode *Insert( struct AVLTreeNode *root, struct AVLTreeNode *parent, int data){
        if( !root) {
                root = (struct AVLTreeNode*) malloc(sizeof (struct AVLTreeNode*) );
                if(!root) { printf("Memory Error");
                        return;
                }
                else {  root→data = data;
                        root→height = 0;
                        root→left = root→right = NULL;
```

```
            }
        }
        else if( data < root→data ) {
            root→left = Insert( root→left, root, data );
            if( ( Height( root→left ) - Height( root→right ) ) == 2 ) {
                if( data < root→left→data )
                    root = SingleRotateLeft( root );
                else    root = DoubleRotateLeft( root );
            }
        }
        else if( data > root→data ) {
            root→right = Insert( root→right, root, data );
            if( ( Height( root→right ) - Height( root→left ) ) == 2 ) {
                if( data < root→right→data )
                    root = SingleRotateRight( root );
                else    root = DoubleRotateRight( root );
            }
        }
        /* Else data is in the tree already. We'll do nothing */
        root→height = max( Height(root→left), Height(root→right) ) + 1;
        return root;
}
```

Time Complexity: O(n). Space Complexity: O($logn$).

Problems on AVL Trees

Problem-71 Given a height h, give an algorithm for generating the $HB(0)$.

Solution: As we have discussed, $HB(0)$ is nothing but generating full binary tree. In full binary tree the number of nodes with height h are: $2^{h+1} - 1$ (let us assume that the height of a tree with one node is 0). As a result the nodes can be numbered as: 1 to $2^{h+1} - 1$.

```
struct BinarySearchTreeNode *BuildHB0(int h){
        struct BinarySearchTreeNode *temp;
        if(h == 0)
            return NULL;
        temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
        temp→left = BuildHB0 (h-1);
        temp→data = count++;                    //assume count is a global variable
        temp→right = BuildHB0 (h-1);
        return temp;
}
```

Time Complexity: O(n). Space Complexity: O($logn$), where $logn$ indicates the maximum stack size which is equal to height of tree.

Problem-72 Is there any alternative way of solving Problem-71?

Solution: **Yes**, we can solve following Mergesort logic. That means, instead of working with height, we can take the range. With this approach we do not need any global counter to be maintained.

```
Struct BinarySearchTreeNode *BuildHB0(int l, int r){
        struct BinarySearchTreeNode *temp;
        int mid = l + (r-l)/2;
```

```
    if( l > r)
        return NULL;
    temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
    temp→data = mid;
    temp→left = BuildHB0(l, mid-1);
    temp→right = BuildHB0(mid+1, r);
    return temp;
}
```

The initial call to $BuildHB0$ function could be: $BuildHB0(1, 1 \ll h)$. $1 \ll h$ does the shift operation for calculating the $2^{h+1} - 1$.

Time Complexity: $O(n)$. Space Complexity: $O(logn)$. Where $logn$ indicates maximum stack size which is equal to height of the tree.

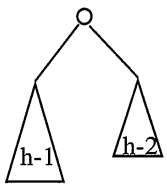
Problem-73 Construct minimal AVL trees of height 0, 1, 2, 3, 4, and 5. What is the number of nodes in a minimal AVL tree of height 6?

Solution Let $N(h)$ be the number of nodes in a minimal AVL tree with height h.

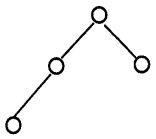
$N(0) = 1$

$N(1) = 2$

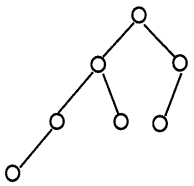
$N(h) = 1 + N(h-1) + N(h-2)$



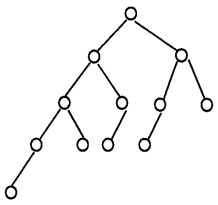
$N(2) = 1 + N(1) + N(0)$
$= 1 + 2 + 1 = 4$



$N(3) = 1 + N(2) + N(1)$
$= 1 + 4 + 2 = 7$



$N(4) = 1 + N(3) + N(2)$
$= 1 + 7 + 4 = 12$



$N(5) = 1 + N(4) + N(3)$
$= 1 + 12 + 7 = 20$

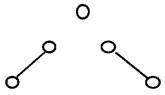
...

Problem-74 For the Problem-71 how many different shapes of a minimal AVL tree of height h can have?

Solution: Let $NS(h)$ be the number of different shapes of a minimal AVL tree of height h.

$NS(0) = 1$

$NS(1) = 2$

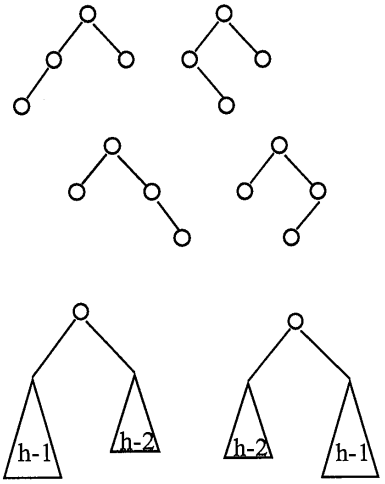


$$NS(2) = 2 * NS(1) * NS(0)$$
$$= 2 * 2 * 1 = 4$$

$$NS(3) = 2 * NS(2) * NS(1)$$
$$= 2 * 4 * 1 = 8$$

...

$$NS(h) = 2 * NS(h-1) * NS(h-2)$$



Problem-75 Given a binary search tree check whether the tree is an AVL tree or not?

Solution: Let us assume that *IsAVL* is the function which checks whether the given binary search tree is an AVL tree or not. *IsAVL* returns -1 if the tree is not an AVL tree. During the checks each node sends height of it to their parent.

```
int IsAVL(struct BinarySearchTreeNode *root){
        int left, right;
        if(!root)
            return 0;
        left = IsAVL(root→left);
        if(left == -1)
                return left;
        right = IsAVL(root→right);
        if(right == -1)
                return right;
        if(abs(left-right)>1)
                return -1;
        return Max(left, right)+1;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

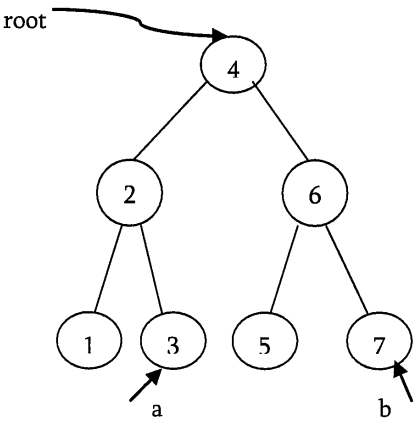
Problem-76 Given a height h, give an algorithm to generate an AVL tree with min number of nodes.

Solution: To get minimum number of nodes, fill one level with $h-1$ and other with $h-2$.

```
struct AVLTreeNode *GenerateAVLTree(int h){
        struct AVLTreeNode *temp;
        if(h == 0)
                return NULL;
        temp = (struct AVLTreeNode *)malloc (sizeof(struct AVLTreeNode));
        temp→left = GenerateAVLTree(h-1);
        temp→data = count++; //assume count is a global variable
        temp→right = GenerateAVLTree(h-2);
        temp→height = temp→left→height+1; // or temp→height = h;
        return temp;
}
```

Problem-77 Given an AVL tree with n integer items and two integers a and b, where a and b can be any integers with $a <= b$. Implement an algorithm to count the number of nodes in the range $[a, b]$.

Solution:



The idea is to make use of the recursive property of binary search trees. There are three cases to consider, whether the current node is in the range $[a, b]$, on the left side of the range $[a, b]$ or on the right side of the range $[a, b]$. Only subtrees that possibly contain the nodes will be processed under each of the three cases.

```
int RangeCount(struct AVLNode *root, int a, int b) {
        if(root == NULL)
                return 0;
        else if(root→data > b)
                return RangeCount(curr→left, a, b);
        else if(root→data < a)
                return RangeCount(root→right, a, b);
        else if(root→data >= a && root→data <= b)
                return RangeCount(root→left, a, b) + RangeCount(root→right, a, b) + 1;
}
```

The complexity is similar to $in-order$ traversal of the tree but skipping left or right sub-trees when they do not contain any answers. So in the worst case, if the range covers all the nodes in the tree, we need to traverse all the n nodes to get the answer. The worst time complexity is therefore $O(n)$.

If the range is small, which only covers few elements in a small subtree at the bottom of the tree, the time complexity will be $O(h) = O(logn)$, where h is the height of the tree. This is because only a single path is traversed to reach the small subtree at the bottom and many higher level subtrees have been pruned along the way.

Note: Refer similar problem in BST.

6.14 Other Variations in Trees

In this section, let us enumerate the other possible representations of trees. In the earlier sections, we have seen AVL trees which is a binary search tree (BST) with balancing property. Now, let us see few more balanced binary search trees: Red-Black Trees and Splay Trees.

Red-Black Trees

In red-black trees each node is associated with extra attribute: the color, which is either red or black. To get logarithmic complexity we impose the following restrictions.

Definition: A red-black tree is a binary search tree that satisfies the following properties:

- Root Property: the root is black
- External Property: every leaf is black
- Internal Property: the children of a red node are black
- Depth Property: all the leaves have the same black

As similar to AVL trees, if the Red-black tree becomes imbalanced then we perform rotations to reinforce the balancing property. With Red-black trees, we can perform the following operations in O($logn$) in worst case, where n is the number of nodes in the trees.

- Insertion, Deletion
- Finding predecessor, successor
- Finding minimum, maximum

Splay Trees

Splay-trees are BSTs with self-adjusting property. Another interesting property of splay-trees is: starting with empty tree, any sequence of K operations with maximum of n nodes takes O($Klogn$) time complexity in worst case.

Splay trees are easier to program and also ensures faster access to recently accessed items. As similar to AVL and Red-Black trees, at any point if the splay tree becomes imbalanced then we perform rotations to reinforce the balancing property.

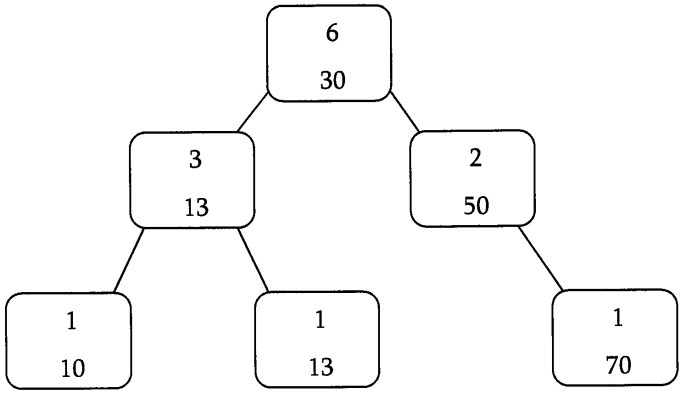
Splay-trees cannot guarantee the O($logn$) complexity in worst case. But it gives amortized O($logn$) complexity. Even though individual operations can be expensive, any sequence of operations gets the complexity of logarithmic behavior. One operation may take more time (a single operation may take O(n) time) but the subsequent operations may not take worst case complexity and on the average *per operation* complexity is O($logn$).

Augmented Trees

In earlier sections, we have seen the problems like finding K^{th} –smallest element in the tree and many other similar problems. For all those problems the worst complexity is O(n), where n is the number of nodes in the tree. To perform such operations in O($logn$) augmented trees are useful. In these trees, extra information is added to each node and that extra data depends on the problem we are trying to solve. For example, to find K^{th} –smallest in binary search tree, let us see how augmented trees solves the problem. Let us assume that we are using Red-Black trees as balanced BST (or any balanced BST) and augment the size information in the nodes data. For a given node X in Red-Black tree with a field $size(X)$ equal to the number of nodes in the subtree and can be calculated as:

$$size(X) = size(X \rightarrow left) + size(X \rightarrow right)) + 1$$

Example: With the extra size information, the augmented tree will look like:



Kth-smallest operation can be defined as:

```
struct BinarySearcTreeNode *KthSmallest (struct BinarySearcTreeNode *X, int K) {
        int r = size(X→left) + 1;
        if(K == r)
                return X;
        if(K < r)
                return KthSmallest (X→left, K);
        if(K > r)
```

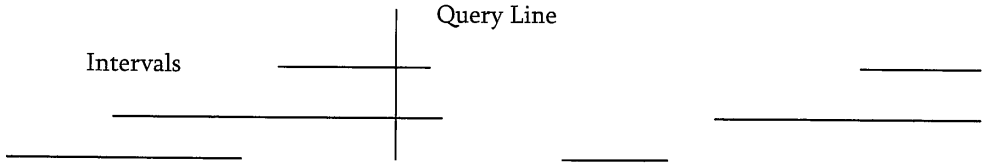
```
        return KthSmallest (X→right, K-r);
}
```

Time Complexity: O($logn$). Space Complexity: O($logn$).

Interval Trees

Interval trees are also binary search trees and stores interval information in the node structure. That means, we maintain a set of n intervals $[i_1, i_2]$ such that one of the intervals containing a query point Q (if any) can be found efficiently. Interval trees are used for performing range queries efficiently.

Example: Given a set of intervals: S = {[2-5], [6-7], [6-10], [8-9], [12-15], [15-23], [25-30]}. A query with $Q = 9$ returns $[6, 10]$ or $[8, 9]$ (assume these are the intervals which contains 9 among all the intervals). A query with $Q =$ 23 returns [15, 23].

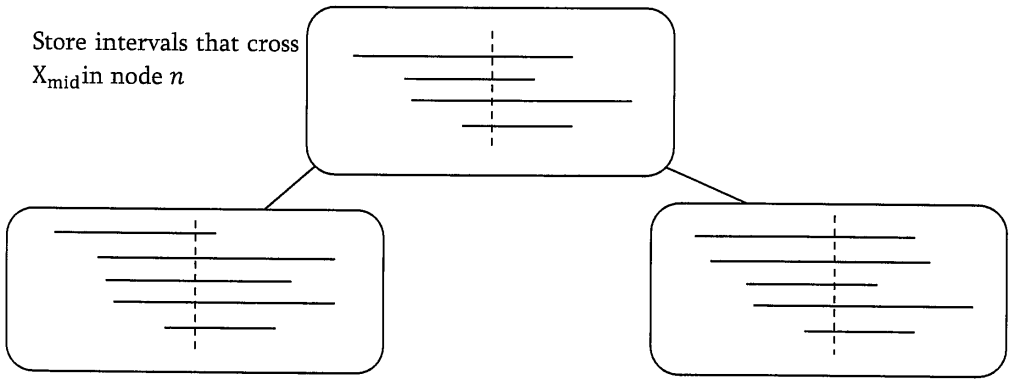


Construction of Interval Trees: Let us assume that we are given a set S of n intervals (also called segments). These n intervals will have $2n$ endpoints. Now, let us see how to construct the interval tree.

Algorithm:

Recursively build tree on interval set S as follows:

- Sort the $2n$ endpoints
- Let X_{mid} be the median point



Time Complexity for building interval trees: O($nlogn$). Since we are choosing the median, Interval Trees will be approximately balanced. This ensures that, we split the set of end points up in half each time. The depth of the tree is O($logn$). To simplify the search process, generally X_{mid} is stored with each node.

Chapter 12 SEARCHING

What is Searching?

In computer science, searching is the process of finding an item with specified properties among a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs or may be elements of other search space.

Why Searching?

Searching is one of core computer science algorithms. We know that today's computers store lot of information. To retrieve this information efficiently we need very efficient searching algorithms.

There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in some proper order then it is easy to search the required element. Sorting is one of the techniques for making the elements ordered.

In this chapter we will see different searching algorithms.

Types of Searching

The following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

Unordered Linear Search

Let us assume that given an array whose elements order is not known. That means the elements of the array are not sorted. In this case if we want to search for an element then we have to scan the complete array and see if the element is there in the given list or not.

int UnsorteddLinearSearch (int A[], int n, int data)

```
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array.
Space complexity: $O(1)$.

Sorted/Ordered Linear Search

If the elements of the array are already sorted then in many cases we don't have to scan the complete array to see if the element is there in the given array or not. In the below algorithm, it can be seen that, at any point if the value at $A[i]$ is greater than the $data$ to be searched then we just return -1 without searching the remaining array.

```
int SortedLinearSearch(int A[], int n, int data)
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
                else if(A[i] > data)
                        return -1;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is same.
Space complexity: $O(1)$.

Note: For the above algorithm we can make further improvement by incrementing the index at faster rate (say, 2). This will reduce the number of comparisons for searching in the sorted list.

Binary Search

If we consider searching of a word in a dictionary, in general we directly go some approximate page [generally middle page] start searching from that point. If the *name* that we are searching is same then we are done with the search. If the page is before the selected pages then apply the same process for the first half otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.

```
//Iterative Binary Search Algorithm
int BinarySearchIterative[int A[], int n, int data)
{
        int low =  0;
        int high = n-1;
        while (low <= high)
        {
                mid  = low + (high-low)/2; //To avoid overflow

                if (A[mid] == data)
                        return mid;
                else if (A[mid] < data)
                        low =  mid + 1;
                else
                        high =  mid - 1;
        }
        return -1;
}

//Recursive Binary Search Algorithm
int BinarySearchRecursive[int A[], int low, int high, int data)
{
        int mid  = low + (high-low)/2; //To avoid overflow

        if (A[mid] == data)
                return mid;
        else if (A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else
                return BinarySearchRecursive (A, low, mid - 1 , data);

        return -1;
}
```

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using *Divide and Conquer* master theorem, we get, $T(n) = O(logn)$.

Time Complexity: $O(logn)$.
Space Complexity: $O(1)$ [for iterative algorithm].

Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Avg. Case
Unordered Array	n	$\frac{n}{2}$
Ordered Array	$logn$	$logn$
Unordered List	n	$\frac{n}{2}$
Ordered List	n	$\frac{n}{2}$
Binary Search (arrays)	$logn$	$logn$
Binary Search Trees (for skew trees)	n	$logn$

Note: For discussion on binary search trees refer *Trees* chapter.

Symbol Tables and Hashing

Refer *Symbol Tables* and *Hashing* chapters.

String Searching Algorithms

Refer *String Algorithms* chapter.

Problems on Searching

Problem-1 Given an array of n numbers. Give an algorithm for checking whether there are any duplicated elements in the array or not?

Solution: This is one of the simplest problems. One obvious answer to this is, exhaustively searching for duplicated in the array. That means, for each input element check whether there is any element with same value. This we can solve just by using two simple *for* loops. The code for this solution can be given as:

```
void CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
```

```
        for(i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        if(A[i] == A[j])
                        {
                                printf("Duplicates exist: %d", A[i]);
                                return;
                        }
                }

        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-2 Can we improve the complexity of Problem-1's solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see if there are elements with same value and adjacent.

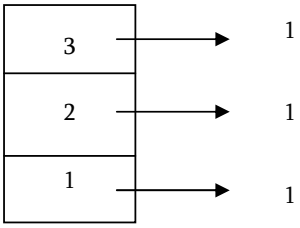
```
void CheckDuplicatesBruteForce(int A[], int n)
{
        //sort the array
        Sort(A, n);

        for(int i = 0; i < n-1; i++)
        {
                if(A[i] == A[i+1])
                {
                        printf("Duplicates exist: %d", A[i]);
                        return;
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(nlogn)$. This is because of sorting.
Space Complexity: $O(1)$.

Problem-3 Is there any other way of solving the Problem-1?

Solution: Yes, using hash table. Hash tables are a simple and effective method to implement dictionaries. $Average$ time to search for an element is $O(1)$, while worst-case time is $O(n)$. Refer $Hashing$ chapter for full details on hashing algorithms.

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. Scan the input array and insert the elements into the hash. For inserted element, keep the $counter$ as 1. This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting first three elements 3, 2 and 1):



If we try inserting 2, since the counter value of 2 is already 1, then we can say the element is appearing twice.

Time Complexity: $O(n)$.
Space Complexity: $O(n)$.

Problem-4 Can we further improve the complexity of Problem-1's solution?

Solution: Let us assume that the array elements are positive numbers and also all the elements are in the range 0 to $n - 1$. For each element $A[i]$, we go to the array element whose index is $A[i]$. That means we select $A[A[i]]$ and mark - $A[A[i]]$ (that means we negate the value at $A[A[i]]$). We continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

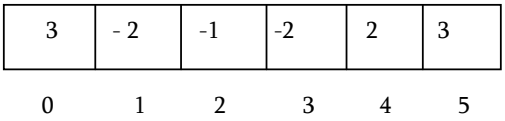
At step-1, negate A[abs(A[0])],

3	2	1	-2	2	3
0	1	2	3	4	5

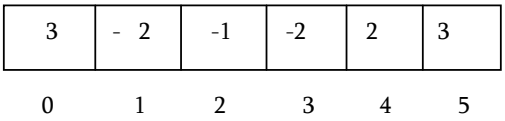
At step-2, negate A[abs(A[1])],

3	2	-1	-2	2	3
0	1	2	3	4	5

At step-3, negate A[abs(A[2])],



At step-4, negate A[abs(A[3])],



At step-4, we can observe that $A[abs(A[3])]$ is already negative. That means we have encountered the same value twice.

The code for this algorithm can be given as:

```
void CheckDuplicates(int A[], int n)
{
        int i =  0;
        for(i = 0; i < n; i++)
        {
                if(A[abs(A[i])] < 0)
                {
                        printf("Duplicates exist:%d", A[i]);
                        return;
                }
                else
                {
                        A[A[i]] = - A[A[i]];
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n)$. Since, only one scan is required.
Space Complexity: $O(1)$.

Note:

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

Problem-5 Given an array of n numbers. Give an algorithm for finding the first element in the array which is repeated?

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. In this array the first repeated number is 3 (not 2). That means, we need to return the first element among the repeated elements.

Solution: We can use the brute force solution of Problem-1. Because it for each element it checks whether there is a duplicate for that element or not. So, whichever element duplicates first then that element is returned.

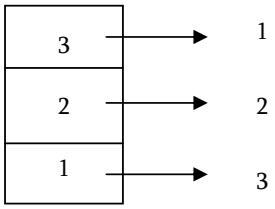
Problem-6 For Problem-5, can we use sorting technique?

Solution: No. For proving the failed case, let us consider the following array. For example, $A = \{3, 2, 1, 2, 2, 3\}$. Then after sorting we get $A = \{1, 2, 2, 2, 3, 3\}$. In this sorted array the first repeated element is 2 but the actual answer is 3.

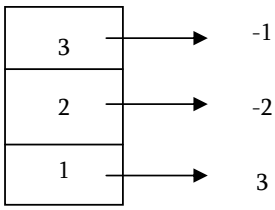
Problem-7 For Problem-5, can we use hashing technique?

Solution: Yes. But the simple technique which we used for Problem-3 will not work. For example, if we consider the input array as $A = \{3, 2, 1, 2, 3\}$, in this case the first repeated element is 3 but using our simple hashing technique we the answer as 2. This is because of the fact that 2 is coming twice before 3. Now let us change the hashing table behavior so that we get the first repeated element.

Let us say, instead of storing 1 value, initially we store the position of the element in the array. As a result the hash table will look like (after inserting 3, 2 and 1):



Now, if we see 2 again, we just negate the current value of 2 in the hash table. That means, we make its counter value as -2. The negative value in the hash table indicates that we have seen the same element two times. Similarly, for 3 (next element in input) also, we negate the current value of hash table and finally the hash table will look like:



After scanning the complete array, we scan the hash table and return the highest negative indexed value from it (i.e., -1 in our case). The highest negative value indicates that we have seen that element first (among repeated elements) and also repeating.

What if the element is repeated more than two times?

In this case, what we can do is, just skip the element if the corresponding value i already negative.

Problem-8 For Problem-5, can we use Problem-3's technique (negation technique)?

Solution: No. As a contradiction example, for the array $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated element is 3. But with negation technique the result is 2.

Problem-9 Given an array of n elements. Find two elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach
One simple solution to this is, for each input element check whether there is any element whose sum is K. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int K)
{
        int i =  0, j = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i; j < n; j++)
                {
                        if(A[i]+A[j] == K)
                        {
                                printf("Items Found:%d %d", i, j);
                                return;
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-10 Does the solution of Problem-9 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we get the pair of numbers if they exist.

Problem-11 For the Problem-9, can we improve the time complexity?

Solution: Yes. Let us assume that we have sorted the given array. This operation takes $O(n\ logn)$. On the sorted array, maintain indices $loIndex = 0$ and hiIndex = $n - 1$ and compute $A[loIndex] + A[hiIndex]$. If the sum equals K, then we are done with the solution. If the sum is less than K, decrement $hiIndex$, if the sum is greater than K, increment $loIndex$.

```
void Search[int A[], int n, int K)
{
        int i, j, temp;
        Sort(A, n);
        for(i = 0, j = n-1; i < j; )
        {
                temp = A[i] + A[j];
                if (temp == K)
                {
                        printf("Elements Found: %d %d", i, j);
                        return;
                }
                else if (temp < K)
                        i = i + 1;
                else
                        j = j - 1;
        }
        return;
}
```

Time Complexity: $O(nlogn)$. If the given array is already sorted then the complexity is $O(n)$.
Space Complexity: $O(1)$.

Problem-12 Is there any other way of solving the Problem-9?

Solution: Yes, using hash table.

Since our objective is to find two indexes of the array whose sum is K. Let us say those indexes are X and Y. That means, $A[X] + A[Y] = K$.

What we need is, for each element of the input array $A[X]$, check whether $K - A[X]$ also exists in input array. Now, let us simplify that searching with hash table.

Algorithm

- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Before proceeding to the next element we check whether $K - A[X]$ also exists in hash table or not.