Unit III

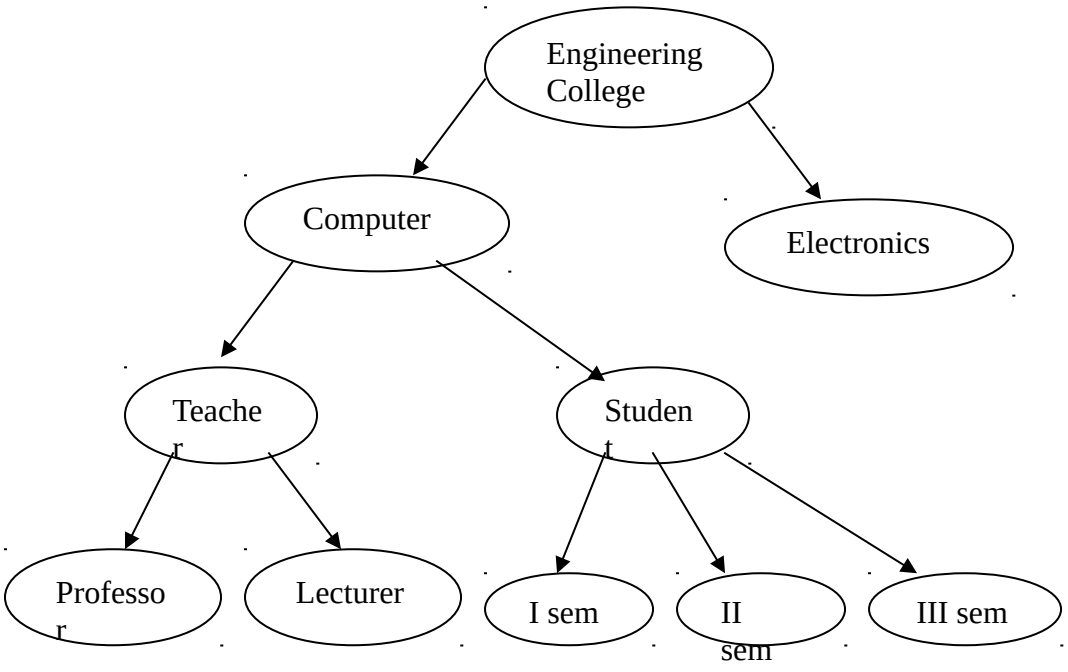
Tree

Nature is man’s best teacher. In every walk of life man has looked and explored the nature, learnt his lessons and then applied the knowledge that nature offered him to solve every-day problems that he faced at work place. It isn’t without reason that there are data structures like trees, binary trees, search trees, AVL trees, forests, etc.

The data structures that we have seen so for (such as linked lists, stack and queues) were linear data structures. As against this, trees are non-linear data structures.

In a linked list each node has a link which points to another node. In a tree structure, however, each node may points to several other nodes (which may then point to several other nodes). Thus tree is a very flexible and powerful data structure that can used for a wide variety of applications.

For example, an engineering college. Such a system has information which can be represented in hierarchical form in the most natural way. A college has n number of departments each for particular branch of engineering. In each department there are students studying in respective semester and there are teachers of various designations such as professor, assistant professors, lecturer. This information system is shown in tree form as below.



A tree structure

Definition Of tree

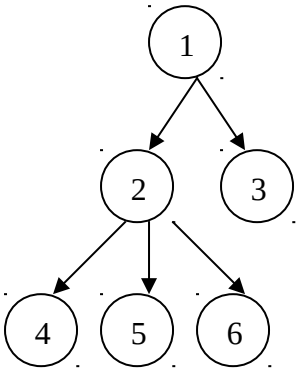
A tree is a finite set of one or more nodes such that

1. There is a specially designated node called root
2. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2, T3,…... Tn

Where T1, T2, T3,….Tn are called the sub trees of the root.

Representation of tree

Basically tree is a set of nodes and every node consists of data. The data of the tree is some information which may be numeric or alphabetic. Each node may or may not have further sub-trees. One can visualize the tree like this



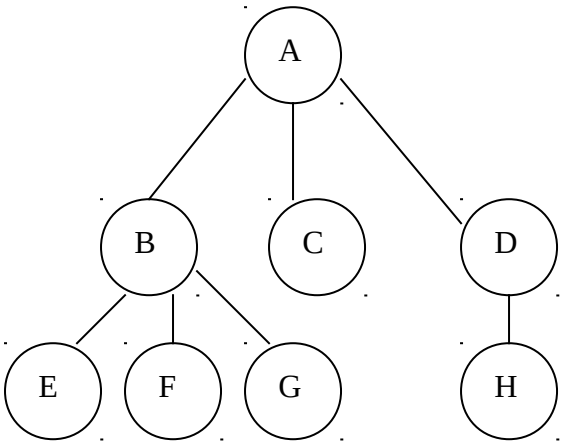
For each node in the tree structure there could be n+ fields

The calculation for n+1 nodes is as below

n – maximum number of branches attached to that node.

1 – data field indicating the information of that node.

Tree terminology



1. **Root**

This is the unique node in the tree to which further sub-trees are attached. In the figure A is a root node.

2. **Degree of the node**

The total number of sub-trees attached to that node is called the degree of the node. For node A the degree is 3, for node E the degree is 0.

3. **Leaves**

These are the terminal nodes of the tree. The nodes with degree 0 always the leaves. Here in our example E,F,C,G,H are the leaf nodes.

4. **Internal nodes**

The nodes other than the root node and the leaves are called the internal nodes. Here B and D are the internal nodes.

5. Parent node

The node which is having further sub-branches is called the parent node of those sub-branches. In the above figure the node B is parent node of E,F,G nodes. And E,F,G are called the children of the parent B.

6. Predecessor

While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. In our example the node E is predecessor of the node B.

7. Successor

The node, which occurs next to some other node. In our example B is a successor of F and G.

8. Level of the tree

The root node is always considered at level zero, then its adjacent children are suppose to be at level 1 and so on. In the above figure A is at level 0, the nodes B,C,D are at level 1, the nodes E,F,G,H, are at level 2.

9. Height of the tree

The maximum level is the height of the tree. Here the height of the tree is 3. The other terminology used for height of the tree is depth of the tree.

10.Forest

A tree may be defined as a forest in which only a single node (root) has no predecessors. Any forest consists of collection of tree.

11.Degree of node

The degree of each node is defined as the total number of sub-branches each node may have. The node A will have the degree 3.

12.Degree of tree

The maximum degree of the node in the tree is called the degree of the tree.

Binary trees

A binary tree consist of a finite set of elements that can be portioned into three distinct sub-sets called the root, the left and the right sub-tree. If there are no elements in the binary tree it is called an empty binary tree.

Types of binary trees

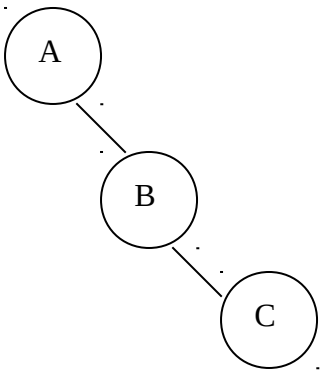
There are three types of binary trees

1. Left Skewed Binary tree:

If the right sub tree is missing in every node of a tree we call it as left skewed binary tree.

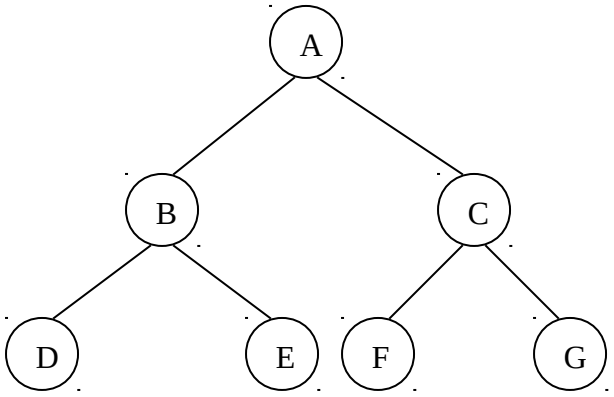
2. Right Skewed Binary Tree:

If the left sub-tree is missing in every node of a tree we call it as right skewed binary tree.



3. Complete Binary Tree

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree of depth d will contain exactly 2^l nodes at each level l, where L is from 0 to d.



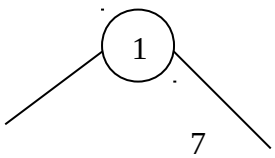
Binary Tree Representation:

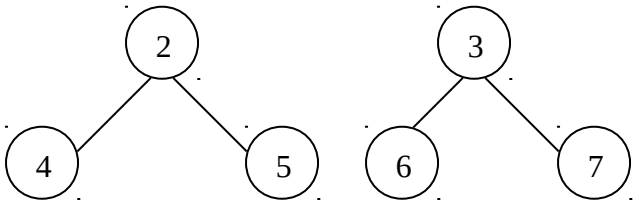
There are two ways of representing the binary tree.

1. Sequential Representation:
2. Linked Representation

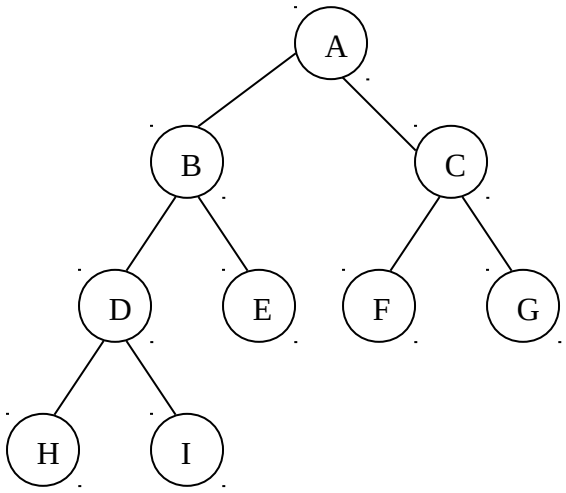
1. Sequential Representation:

Each node is sequentially arranged from top to bottom and from left to right. Let us understand this matter by numbering each node. The numbering will start from root node and then remaining nodes will give ever increasing numbers in level wise direction. The node on the same level will be numbered from left to right. The numbering will be shown in figure.





Now observe this figure carefully. You will get a point that a binary tree of depth n having 2^n-1 number of nodes. In this figure the tree is having the depth 3 and the total number of nodes are 7. thus remember that in a binary tree of depth n there will be maximum 2^n-1 nodes. And so if we know the maximum depth of the tree then we can represent binary tree using arrays data structure.



Root=A=index0

Left child of A i.e.. n=0

B will be at 2*0+1=1st location

Similarly right child of A which will be C

Therefore C will be at 2*0+2=2nd location

I is at 8th location

2n+2=8,2n=6,n=3

that means parent of I is at 3rd location and i.e. D.

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left right children of any particular node is fast because of the random access.

Disadvantage of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have to decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertion and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

Linked representation:

In linked list each node will look like this

Left Child	Data	Right Child

In binary tree each node will have left child, right child and data filed. The left child is nothing but the left link which points to some address of left

sub-tree where as right child is also a right link which points to some address of right sub-tree. And the data filed gives the information about the node.

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree using dynamic memory concept one can create as much memory as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.
2. Insertions and deletions which are the most common operations can be done with out moving the other nodes.

Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

Binary tree traversals:

The traversal of binary tree involves visiting each node in the tree exactly once. In several applications involving binary tree we need to go to each node in the tree systematically. In a linear list, nodes can be visited in a systematic manner from beginning to end. However, such an order is not possible while traversing a tree. Basically there are six ways to traverse a tree. For these traversals we will use some notations as follows :

L means move to left child

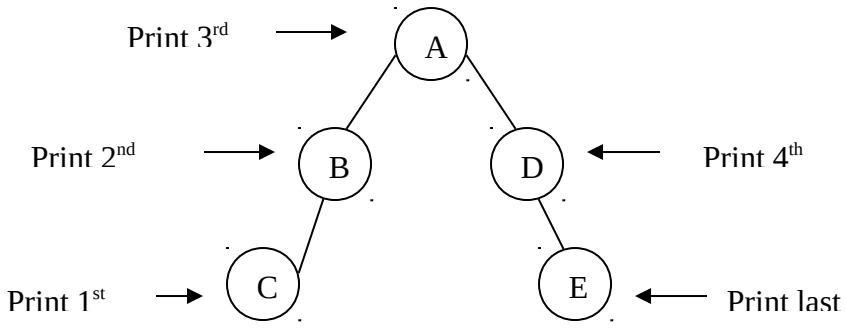
R means move to right child

D means the root/parent node.

Now, with this L,R,D one can have six different combinations of L,R,D nodes.

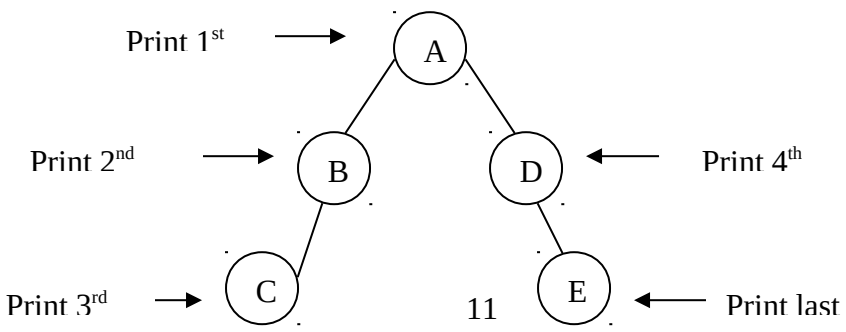
Such as LDR, LRD, DLR, DRL, RLD, RDL. But form computing point of view we will have three different ways of traversing a tree. Those three combinations will be LDR, DLR, LRD. Those are called **in order, preorder, post order**. The methods differs primarily in the order in which they visit the root node the nodes in the left sub-tree and the nodes in the right sub-tree.

1. **In order traversal:**



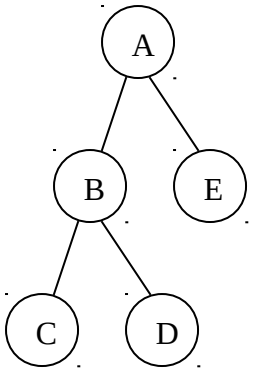
C-B-A-D-E is the in order traversal i.e. first we go towards the leftmost node i.e. C so print the node. Then back to the node B and print B. then root node A then move towards the right sub-tree print D and finally E. thus we are following the tracing sequence of LDR. This type of traversal is called in-order traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

Pre order traversal:



A-B-C-D-E is the preorder traversal of the above figure. We are following DLR path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the leftmost node. Print the data at that node and then move to the right sub-tree. Follow the same DLR principle at each sub-tree and go on printing the data accordingly.

Post order traversal:

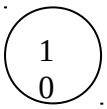


C-D-B-E-A is the post order traversal of the above figure. In the post order traversal we are following the LRD principle i.e. move to the left most node check if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the rightmost node. The key idea here is that at each sub-tree we are following the LRD principle and print the data accordingly.

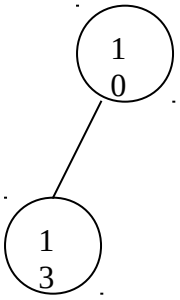
Simple binary tree:

When the data will be entered by the user, the first element will form the root node. For all the next elements user has to enter whether that element has to be inserted as aleft child or a right child of previous node. For the sake of understanding, let us build the simple binary tree for some elements.

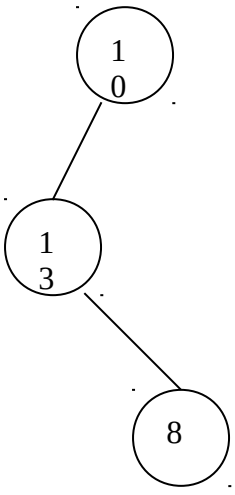
Let us 10, now root will be formed for 10.



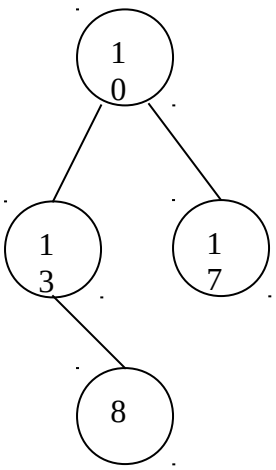
If next element is say 13 then user will be asked for his choice i.e. he will be asked if 13 is attached to the left or right of 10. if user answer is L (that means left) then tree will be



Then if the next element is 8 then again user has to give his choice, whether it is L(left) or R(right). That means first whether user want to attach 8 left or right of 10 will be asked, if user answers L then again “whether user want to attach 8 to left or right of 13 will be asked, if he answers R then, the node 8 will be attached as a right child of 13.



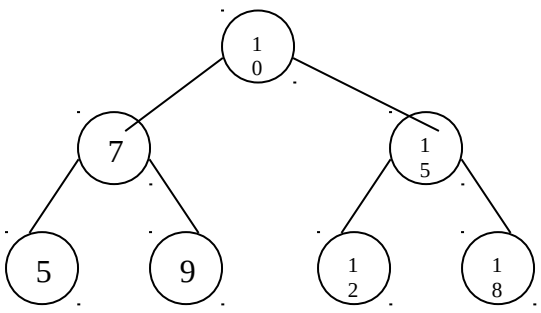
If the next element is 17 and user want to attach it as a right child of 10 then



Thus a simple binary tree will be generated. The principle idea behind this creation is always ask the user where he wants to attach the next node and always start scanning the tree root.

Binary search tree

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left child or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at left sub tree < root node value < right sub tree values.



Construction of binary search tree

Let us take some elements and construct a binary search tree from it.

Step1:

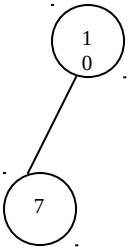
10	7	15	9	5	12	18

Store all the elements in an Array

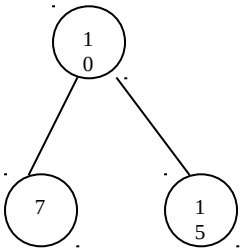
Step 2: Read the first element from the array and form the first node of the tree. Call it as root node.

Root node

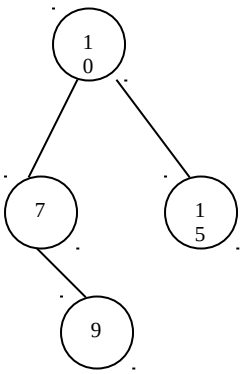
Step 3: Read next element i.e. 7. As 7< 10 attach 7 as left child of 10.



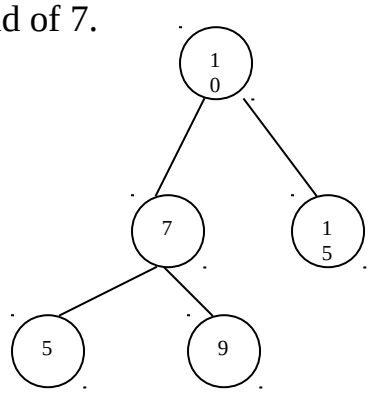
Step 4: Read next element i.e. 15. As 15 > 10 attach 15 as right child of 10



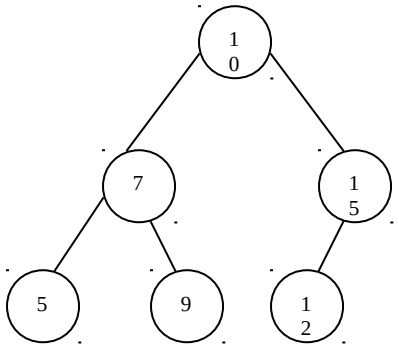
Step 5: Read next element i.e. 9. As 9 < 10 we can attach 9 as left child of 10 but already 7 is attached as left child of 10 so we will again compare 7 and 9. 9 > 7 hence 9 will be attached as right child of 7.



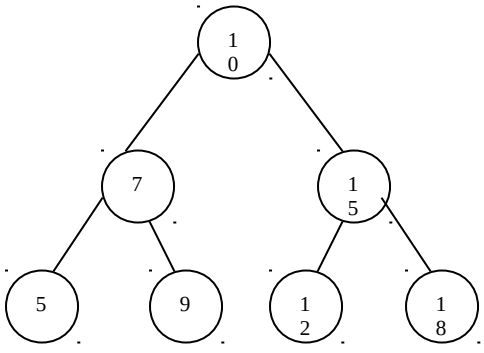
Step 6: Read next element as 5. 5 < 10 and 5 is <7. Hence 5 will be attached as a left child of 7.



Step 7: read the next element i.e. 12. Now 12 will be compared with 10. Since 12 > 10 we can attach 12 as a right child of 10. Then we will compare 12 with 15, then 12 < 15 hence 12 will be attached as left child of 15.



Step 8: Read the next element i.e. 18. As 18 is >10 and 15, we can attach it as right child of 15.

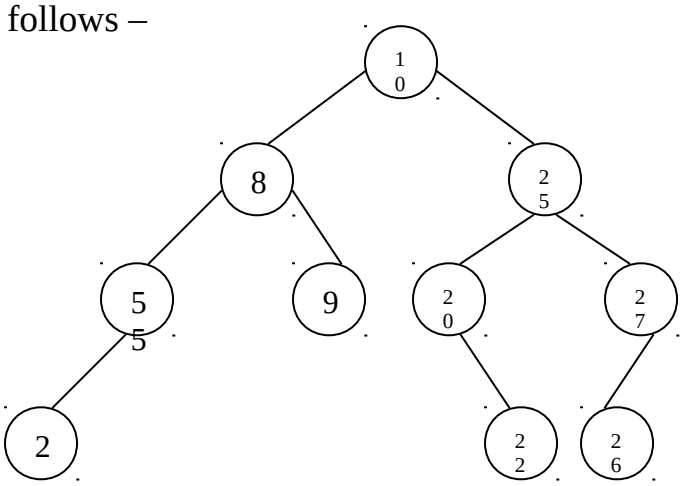


Now there is no further element in the array. Hence we will display the final binary search tree as in step 8.

If you observe the figure carefully, you will find that the left value < parent value < right value is followed throughout the tree.

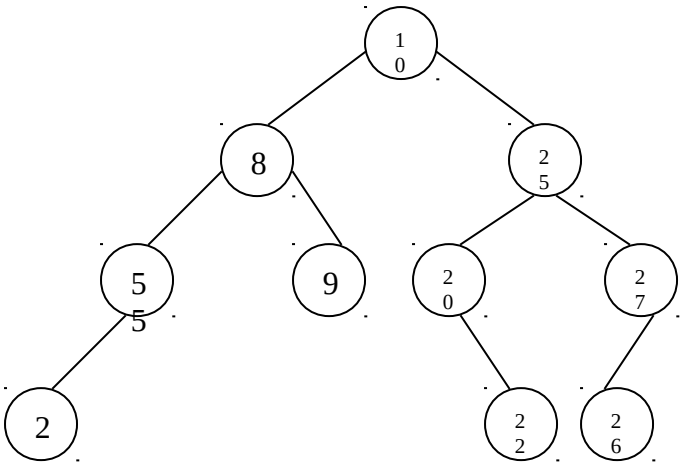
Searching a node from binary search tree.

Let us consider that we have already created a binary search tree as follows –

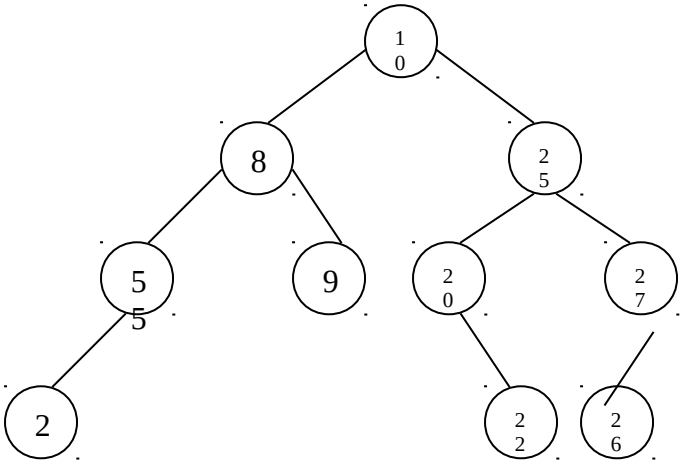


Now we want to search a node 22,from it. Then to search this desired element we will start scanning the tree from root node. Let us call the node element which we want to search as key. I.e.. Key =22. We will proceed in this way.

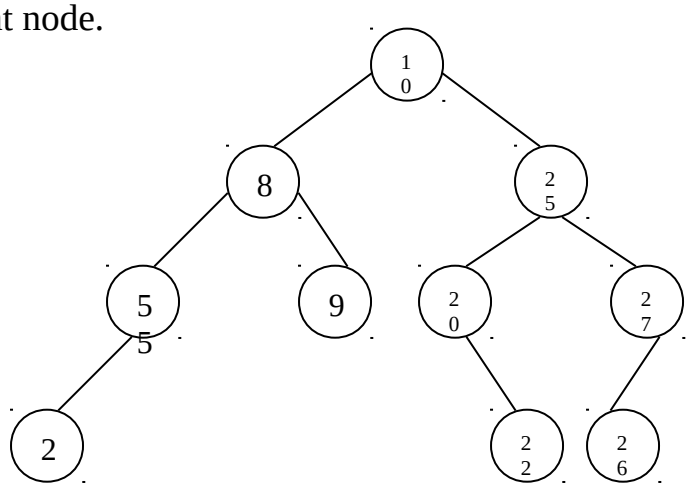
Step 1: Is 22>10. Then move on the right sub branch of tree from current node.



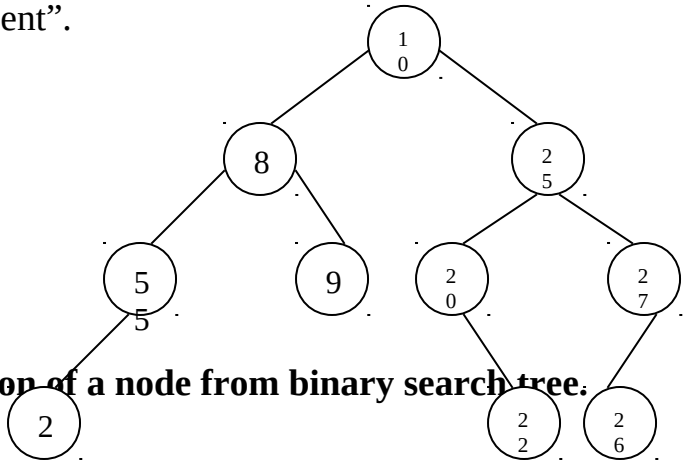
Step2: Is 22<25. Then move the left sub branch of the tree from current node.



Step3: Is 22 > 20 then move on the right sub branch of the tree from the current node.



Step 4: Is 22 == 22 yes!. Match is found. Hence print the message “node 22 is present”.



Deletion of a node from binary search tree.

There are three cases for deletion of any node from binary tree. The case are as follows

Case 1: When a node which we want to delete has no child

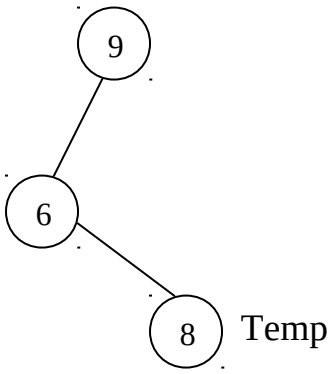
Case 2: When a node which we want to delete has only one child(maybe left or right child)

Case 3: When a node which we want to delete has two children

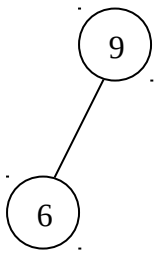
With these cases we can delete any node from binary search tree. Let us discuss each case in detail.

Discussion on case 1:

The given figure shows a binary search tree. we want to delete a node 8 which has no child. This is the simplest deletion. Make the that node as NULL. Temp = NULL

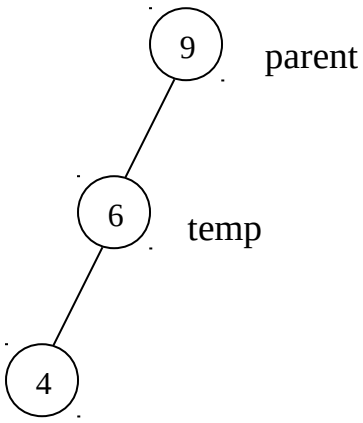


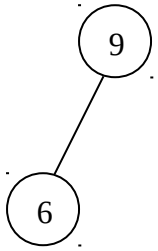
Before deletion



After deletion

Discussion of case 2: when we want to delete such a node which has one child as given in figure.





Before deletion **After deletion**

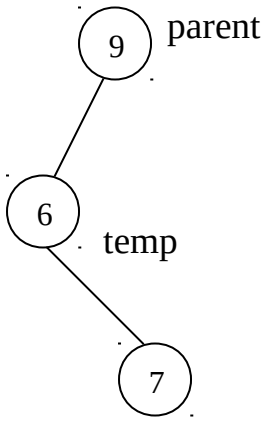
The child of ‘temp’ node (node to be deleted) will be the child of parent node.

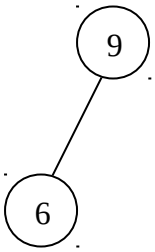
The ‘C’ statements can be

Parent -> left = temp -> left

Free(temp);

Or it could be as follows





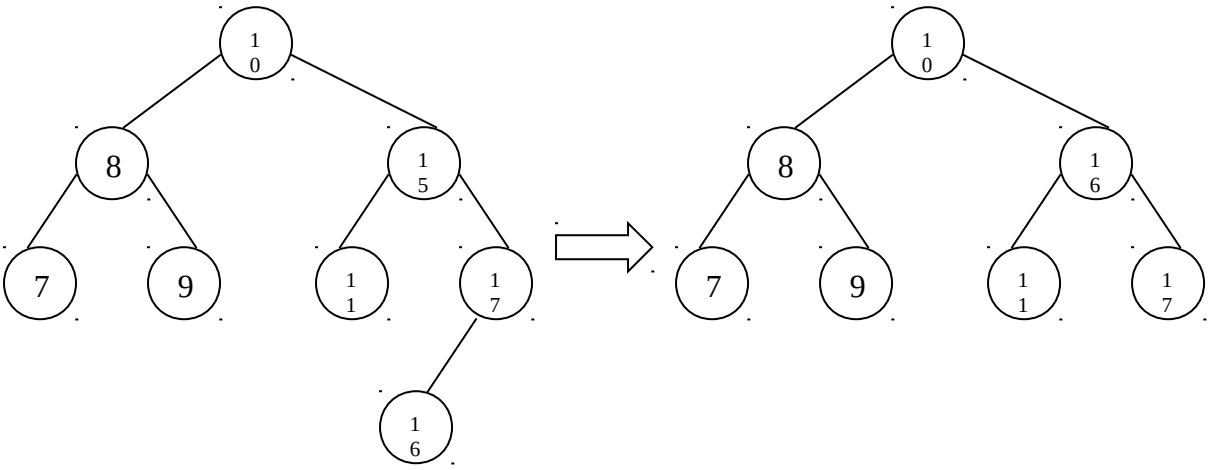
B**efore deletion** **After deletion**

The ‘C’ statements can be

Parent -> left = temp -> right

Free(temp);

Discussion of case 3: when we want to delete such a node which has two children. The figure given below show it.



Before deletion **After deletion**

The simplest technique in this type of deletion is first find the in-order successor of the node which you want to delete. Copy the value of that successor to that place. That means we want to delete the node 15 its in-order successor is 16. Hence copy 16 at 15’s position and then set left pointer of node 17 to NULL.

Applications of binary trees

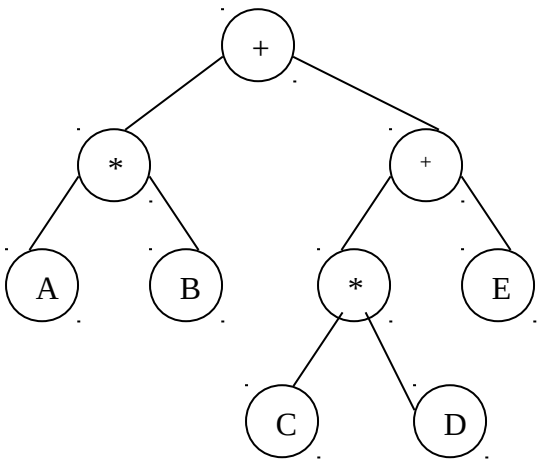
There are many applications of binary tree. For example, whenever we want to take the two-way decisions. Binary tree are the best option. One such application is a binary search tree, which we have already discussed. Binary tree can also be used to represent expressions.

Representing expressions in binary trees:

The arithmetic expressions represented as binary tree are know as expression trees. The root node is operator and the left and right children are operands.

In case of unary operators the left child is absent and the right child is the operand. Since there is no operator available for exponent in C, the operator $ is used to denote exponent. For example the below figure shows a tree for the expression

A * B + C * D + E



When the expression tree is traversed in pre-order then the pre-fix form of the expression is obtained. Similarly, when the expression tree is traversed in post-order then the post-fix form of the expression is obtained. In the same way we get the in-fix form of the expression when the tree is traversed in-order. But the parentheses present in the original arithmetic expression cannot be obtained back from the in-order traversal of the binary tree. This is because the structure of the binary tree itself decides the order of the evaluation of the tree.

Reconstruction of a binary tree:

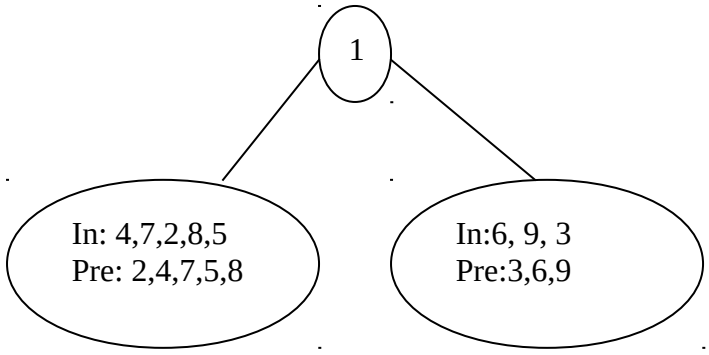
If we know the sequence of nodes obtained through in-order/pre-order/post-order traversal it may not be feasible to reconstruct the binary tree. This is because two different binary tree may yield same sequence of nodes when traversed using post-order traversal. Similarly in-order or pre-order traversal of different binary trees may yield the same sequence of nodes. However,

we can construct a unique binary tree if the result of in-order and pre-order traversal are available. Let us understand this with the help of following set of in-order and pre-order traversal results.

In-order traversal: 4,7,2,8,5,1,6,9,3

Pre-order traversal: 1,2,4,7,5,8,3,6,9

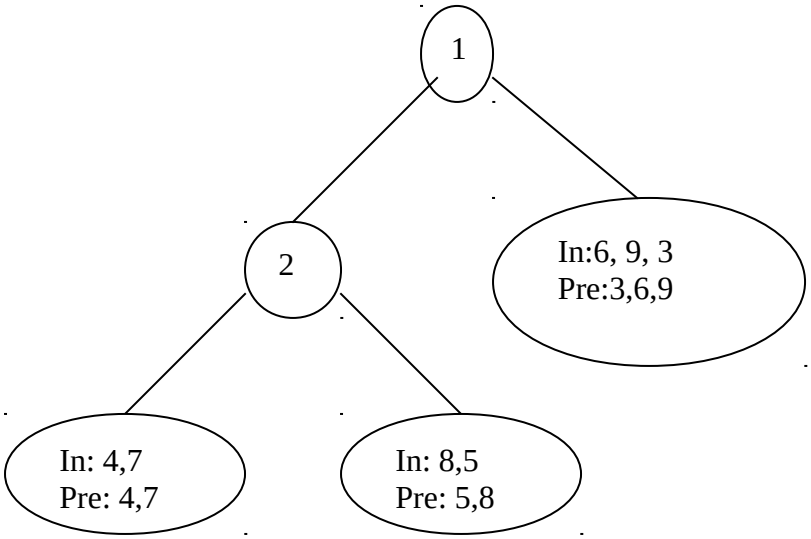
We know that the first value in the pre-order traversal gives us the root of the binary tree. So the node with data 1 becomes the root of the binary tree. In in-order traversal, initially the left sub-tree is traversed then the root node and then the right sub-tree. So the data before 1 in the in-order list (i.e. 4,7,2,8,5) forms the left sub-tree and the data after 1 in the in-order list (i.e. 6,9,3) forms the right sub-tree. In below figure the structure of tree is shown after separating the tree in left and right sub-trees.



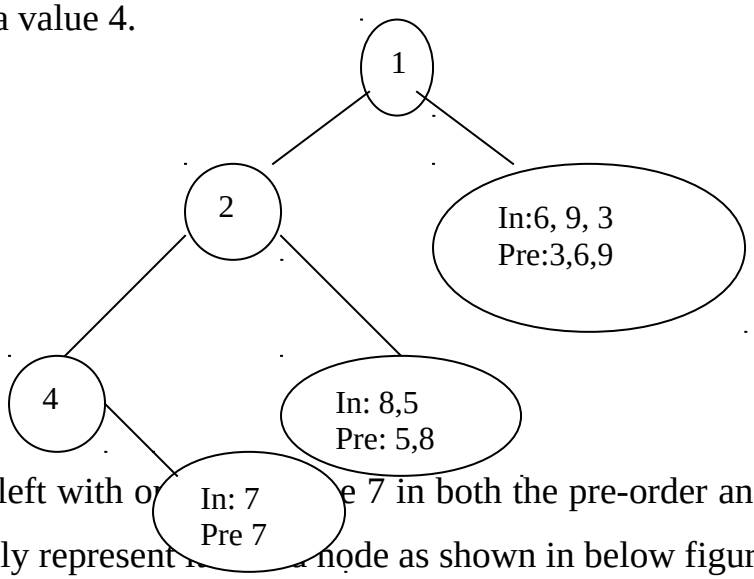
Reconstruction of binary tree.

Now the next data in pre-order list is 2 so the root node of the left-sub-tree is 2. Hence the data before 2 in the in-order list is (i.e. 4,7) forms the left-sub-tree of the node that contains a value 2. The data that comes to the right of 2 in the in-order list (i.e. 8,5) forms the right sub-tree of the node with value 2.

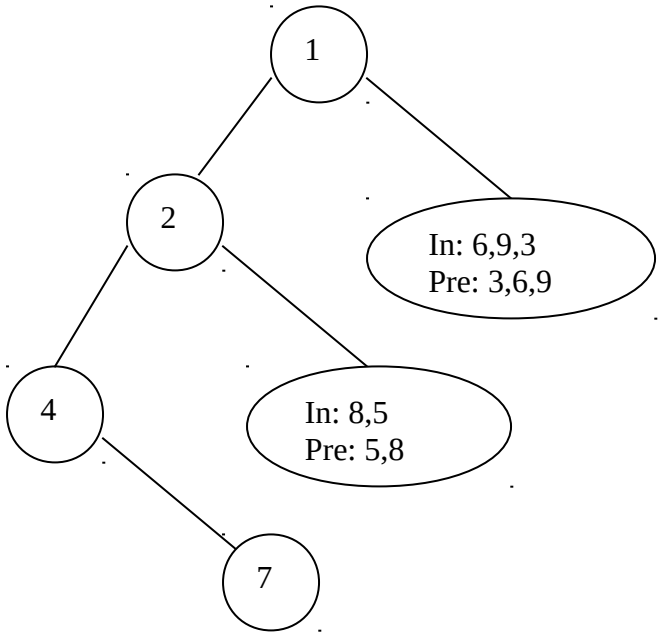
Below figure shows structure of tree after expanding the left and right sub-tree of the node that contains a value 2.



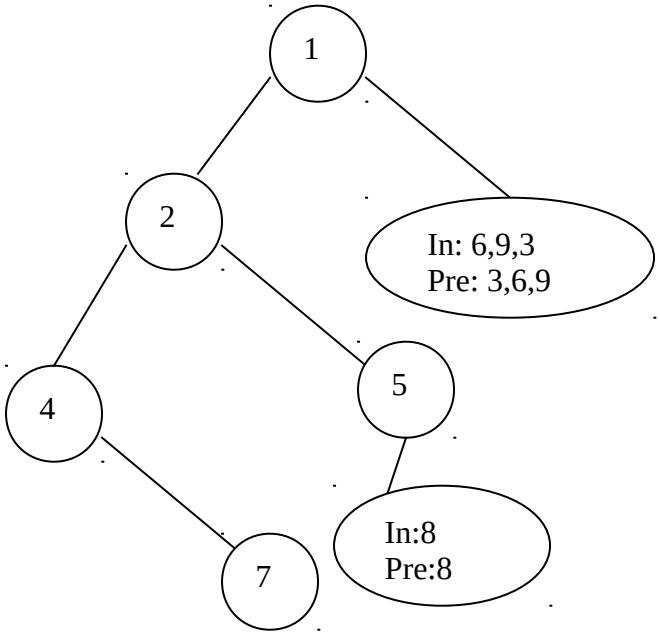
Now the next data in pre-order list is 4, so the root node of the left sub-tree of the node that contains a value 2 is 4. The data before 4 in the in-order list forms the left sub-tree of the node that contains a value 4. But as the there is no data present before 4 in in-order list, the left sub-tree of the node with value 4 is empty. The data that comes to the right of 4 in the in-order list (i.e. 7) forms the right sub-tree of the node that contains a value 4. Below figure shows structure of tree after expanding the left and right sub-tree of the node that contains a value 4.

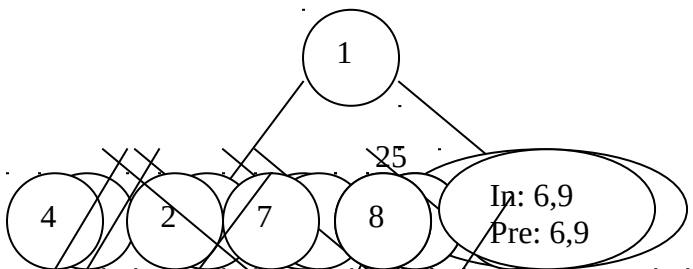


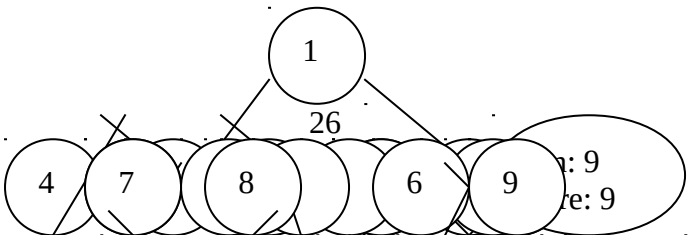
Since we are left with o[illegible]e 7 in both the pre-order and in in-order form we simply represent [illegible] node as shown in below figure



In the same way one by one all the data are picked from the pre-order list and are placed and their respective sub-trees are constructed. As a result, the whole tree is constructed.






1
26
4
7
8
6
9
n: 9
re: 9

AVL Trees:

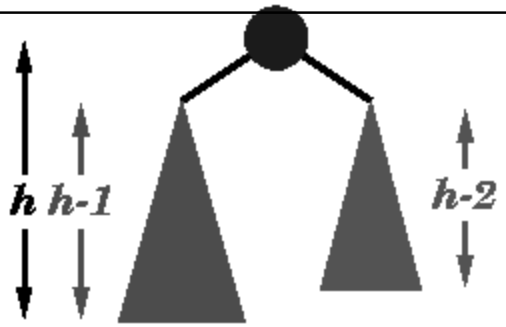
An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an

O(logn) search time. Addition and deletion operations also take ***O(logn)*** time.

Definition of an AVL tree

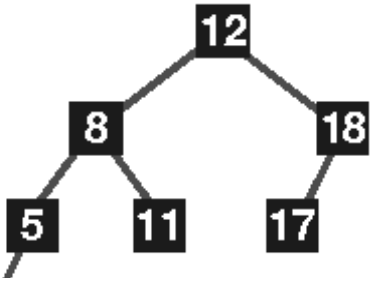
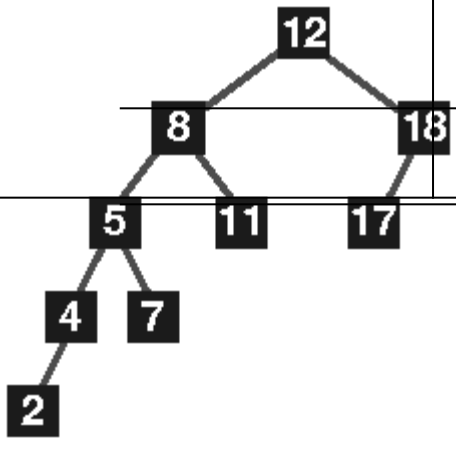
An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

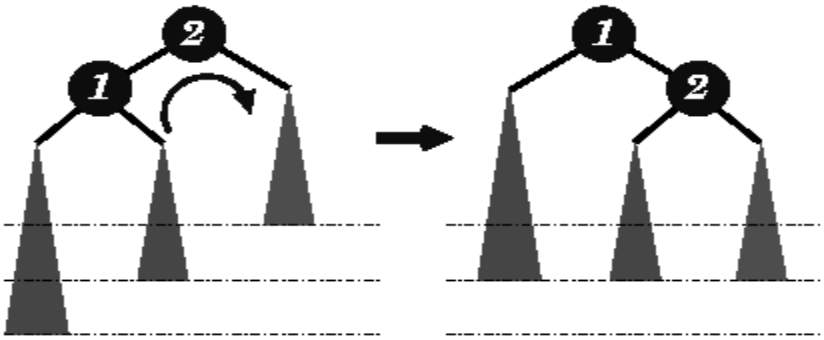
You need to be careful with this definition: it permits some apparently unbalanced trees! For example, here are some trees:

Tree	AVL Tree?
	**Yes** Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree
	**No**

Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.



A new item has been added to the left sub-tree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.

HASHING:

Hashing is an effective way to reduce the number of comparisons. Actually hashing deals with the idea of proving the direct address of the record is likely to store. To understand the idea clearly let take an example

	kev	Record
0		
1		
2	8421002	
3		

395	4618396	
396	4957397	
397		
398	1286399	
996	4618996	
997	4967997	
998	1200992	
999	0001999	

Hashing

Suppose the manufacturing company has an inventory file that consist of less than 1000 parts .

Each part is having unique 7 digit number. The number is called “key” and the particular keyed record consist of that part name. If there are less than 1000 parts then a 1000 element array can be used to store the complete file. Such an array will be indexed from 0 to 999. since the key number is 7 digit it is converted to 3 digits by taking only last 3digits of a key. This is shown in the fig

Observe in fig that the first key 496700 and it is stored at 0^{th} position. The second key is8421002. the last three digits indicate the position 2^{nd} in the array. Let us search the element 4957397.Naturally it will be obtain at

position 397.This method of searching is called hashing. The function that converts the key (7 digit) into array position is called hash function.

Here hash function is

H(key)=key % 1000

Where key % 1000 will be the hash function and the key obtained by hash function is called hash key.

Collision:

From our above discussion, you might be understanding that what I mean by hash function is that function which simply takes some key value, performs some computation on it and gives as the key which is actually the address where the desired record is placed. Thus this function helps us in getting the direct access to the record. Such function is that it should not return the same key address for two different record. The situation in which the hash function returns the same address (same hash keys) for two different records is called the collision. Remember occurrence of collision means design of hash function is poor.

Any way still there are effective ways by which one can resolve the collision which are known as collision handling technique. Let us discuss those.

Collision handling techniques:

1.linear probing: when collision occurs i.e. when two records demand for the same location in the hash table, then the collision can be solved by placing second record linearly down wherever the empty location is found.

For example

Index	data
0	

1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	

Linear probing

In the hash table given in fig the hash function used in number % 10. if the first number which is to be placed is 131 then 131 %10= 1. i.e. remainder is 1 so hash key =1. That means we

Are supposed to place the record at index 1.

Next number is 21 which gives hash key =1 as

21 % 10 =1.But already 131 is placed at index 1.

That means collision is occurred . We will now apply linear probing. In this method , we will search the place for number 21 from location of 131. In this case we can place 21 at index 2 . Then 31 at index 3. Similarly 61 can be stored at 6 because 4 and 5 are stored before 61. Because of this technique , the searching becomes efficient, as we have to search only limited list to obtain the desired number.

Chaining without replacement:

In collision handling method changing is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs we store the second colliding data by linear probing method. The address of this colliding data can be stored with the first colliding element in the chain table, without replacement.

For example consider elements

131, 3, 4, 21, 61, 6, 71, 8, 9

Index	Data	Chain
0	-1	-1
1	-13	2
2	21	5
3	3	-1
4	4	-1
5	61	7
6	6	-1
7	71	-1
8	8	-1
9	9	-1

From the example, you can see that the chain is maintained the number who demands for location 1. First number 131 comes we will place at index 1. Next comes 21 but collision occurs so by linear probing we will place 21 at index 2, and chain is maintained by writing 2 in chain table at index 1 similarly next comes 61 by linear probing we can place 61 at index 5 and chain will be stored by linear probing at empty location but a chain is maintained so that traversing the hash table will be efficient.

The drawback of this method is in finding the next empty location. We are least bothered about the fact when the element which actually belonging to that empty location can not obtain its location. This means logic of hash function gets distributed.

Chaining with replacement:

As previous has a drawback of loosing the meaning of the hash function, to over come this drawback the method known as changing with replacement is introduced. Let us discuss the example to understand the method. Suppose we have to store following elements: 131, 21, 31, 4, 5

Index	Data	Chain
0	-1	-1
1	131	2
2	21	3
3	31	-1
4	4	-1
5	5	-1
6		
7		
8		
9		

Now next elements is 2. As a hash function will indicate hash key as 2 but already at index 2. We have stored element 21. But we also know that 21 is not of that position at which currently it is placed.

Hence we will replace 21 by 2 and accordingly chain table will be updated.

Index	Data	Chain
0	-1	-1
1	131	6
2	2	-1

3	31	-1
4	4	-1
5	5	-1
6	21	3
7	-1	-1
8	-1	-1
9	-1	-1

The value –1 in the hash table and chain table indicate the empty location.

The advantage of this method is that the meaning of hash function is preserved.

But each time some logic is needed to test the element, whether it is at its proper position.