Unit-IV
Search Trees

Contents

- Symbol Table-Representation of Symbol Tables- Static tree table and Dynamic tree table
- Weight balanced tree - Optimal Binary Search Tree (OBST), OBST as an example of Dynamic Programming,
- Height Balanced Tree- AVL tree
- Red-Black Tree, AA tree, K-dimensional tree, Splay Tree

Search Tree

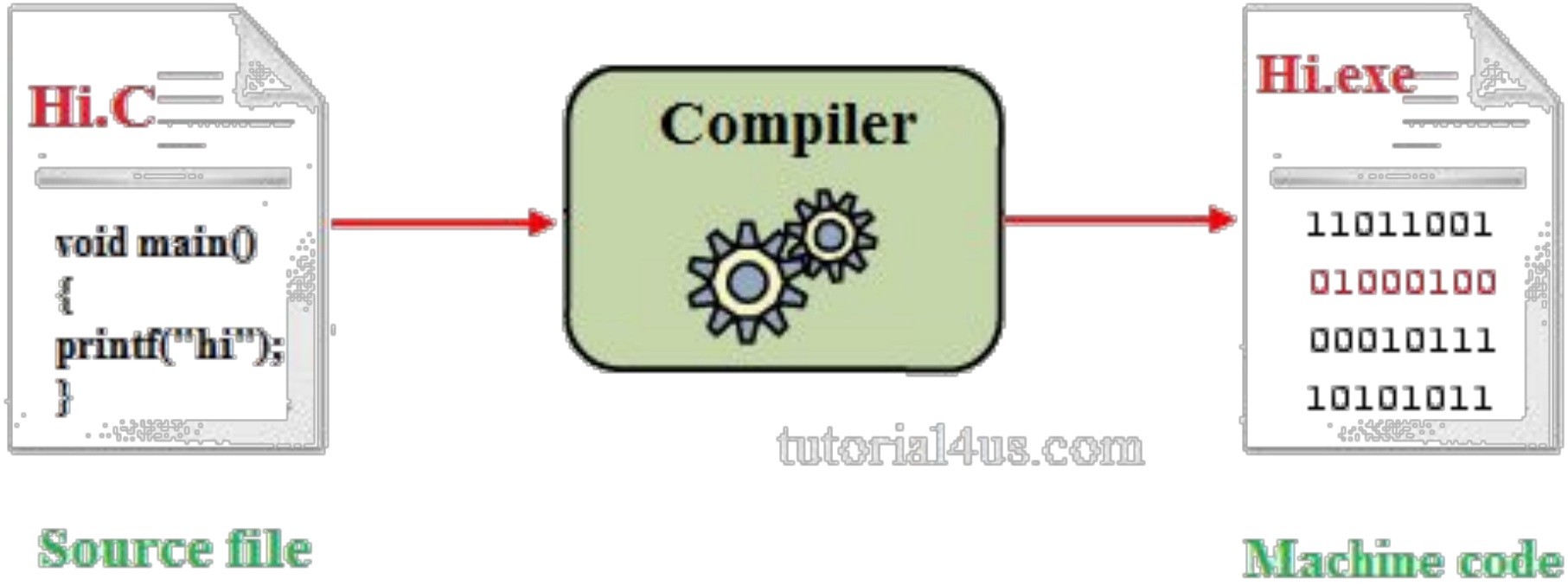
- A tree data structure used for locating specific keys from within a set.
- In order for a tree to function as a search tree, the key for each node must be greater than any keys in subtrees on the left, and less than any keys in subtrees on the right.

Symbol Table

- The symbol table is defined as the set of Name and Value pairs.
- Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the **analysis** and the **synthesis** parts of a compiler.

What is a compiler?

- The simplest definition of a compiler is a program that translates code written in a high-level programming language (like JavaScript or Java) into low-level code (like Assembly) directly executable by the computer or another program such as a virtual machine.



Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by the compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, a pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations on Symbol Table :

Following operations can be performed on symbol table-

1. Insertion of an item in the symbol table.
2. Deletion of any item from the symbol table.
3. Searching of desired item from symbol table.

Representation of Symbol Tables-

Data structure for symbol table

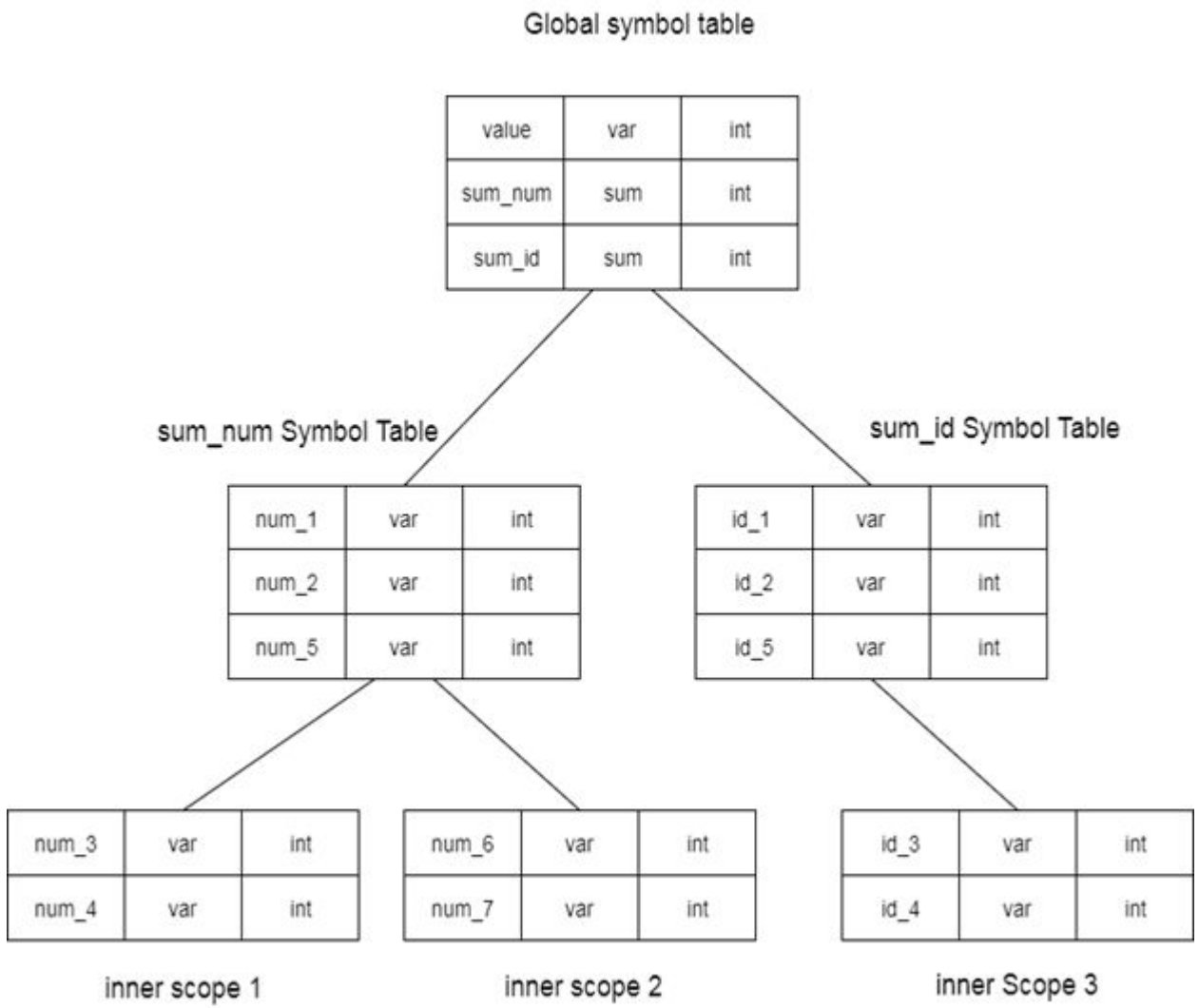
A compiler contains two type of symbol table

- global symbol table
- scope symbol table.
- Global symbol table can be accessed by all the procedures and scope symbol table.

Example

```
int value=10;

void sum_num()
    {
      int num_1;
      int num_2;

          {
           int num_3;
           int num_4;
          }

      int num_5;

          {
           int_num 6;
           int_num 7;
          }
```

```
    }

Void sum_id
    {
      int id_1;
      int id_2;

          {
           int id_3;
           int id_4;
          }

      int num_5;
    }
```



Symbol Table-

Representation of Symbol Tables

Consider the following program written in C:

```c
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double  sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1;  i <= count;  i++)
        sum += bar((double) i);
    return sum;
}
```

A C compiler that parses this code will contain at least the following symbol table entries:

Symbol name	Type	Scope
`bar`	function, double	extern
`x`	double	function parameter
`foo`	function, double	global
`count`	int	function parameter
`sum`	double	block local
`i`	int	for-loop statement

Static Tree Table

- Symbols are known in advance
- No insertion and deletions are allowed
- Ex. Huffman tree and OBST

Dynamic Tree Table

- Symbols are inserted as and when the come
- Ex. AVL tree

Introduction to Dynamic Programming

- Who invented dynamic programming?
- Richard Ernest Bellman
- Richard Ernest Bellman (August 26, 1920 – March 19, 1984) was an American applied mathematician, who introduced dynamic programming in 1953, and important contributions in other fields of mathematics
- What is dynamic programming?
- Dynamic Programming refers to a very large class of algorithms. The idea is to break a large problem down (if possible) into incremental steps so that, at any given stage, optimal solutions are known to sub-problems.

Difference between Divide & conquer and Dynamic programming

Divide & Conquer

1. The divide-and-conquer paradigm involves three steps at each level of the recursion:
• **Divide** the problem into a number of sub problems.
• **Conquer** the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
• **Combine** the solutions to the sub problems into the solution for the original problem.

2. They call themselves recursively one or more times to deal with closely related sub problems.

3. D&C does more work on the sub-problems and hence has more time consumption.

4. In D&C the sub problems are independent of each other.

5. Example: Merge Sort, Binary Search

Difference between Divide & conquer and Dynamic programming

- **Dynamic Programming**

1. The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

- a. Characterize the structure of an optimal solution.
- b. Recursively define the value of an optimal solution.
- c. Compute the value of an optimal solution in a bottom-up fashion.
- d. Construct an optimal solution from computed information

2. Dynamic Programming is not recursive.

3. DP solves the sub problems only once and then stores it in the table.

4. In DP the sub-problems are not independent.

5. Example : Matrix chain multiplication

Dynamic Programming vs. Recursion and Divide & Conquer

Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Dynamic Programming

- Principal: Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- Following are the **two main properties** of a problem that suggest that the given problem can be solved using Dynamic programming.
    1) Overlapping Subproblems
    2) Optimal Substructure

Dynamic Programming

- **1) Overlapping Subproblems:** Dynamic Programming is mainly used when **solutions of same subproblems are needed again and again**. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed.

- **2) Optimal Substructure:** A given problems has **Optimal Substructure** Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

The principle of optimality

- Dynamic programming is a technique for finding an *optimal* solution

- The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems

Differences between Greedy, D&C and Dynamic

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Divide and conquer Vs Dynamic Programming

- Divide-and-Conquer: a top-down approach.
- Many smaller instances are computed more than once.

- Dynamic programming: a bottom-up approach.
- Solutions for smaller instances are stored in a table for later use.

Weight balanced tree - Optimal Binary Search Tree (OBST)

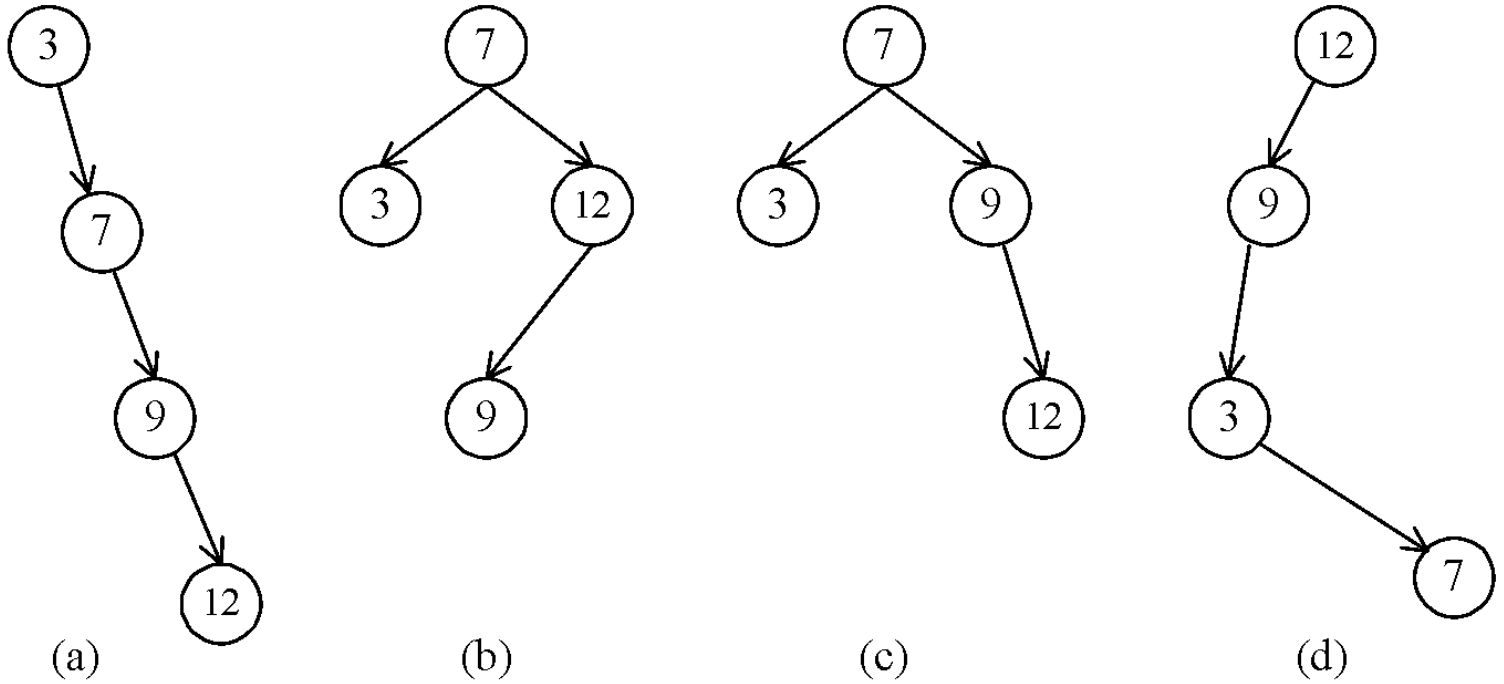
- A binary tree is weight balanced if the weight of the right and left subtree in each node differ by at most one.
- balanced based on the weight on the edges of the tree.
- the absolute difference between the weight of the left subtree and the right subtree should be minimum.

Optimal Binary Search Tree (OBST)

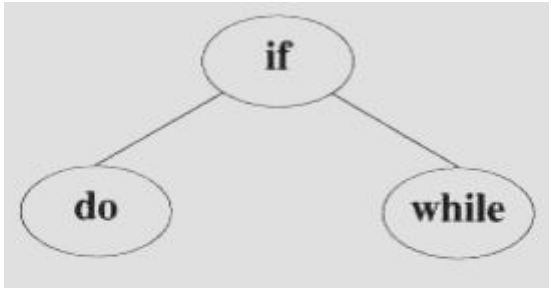
- An optimal binary search tree (OBST) is a binary search tree (BST) for which the nodes are arranged on levels such that the tree **cost is minimum**

Optimal binary search trees

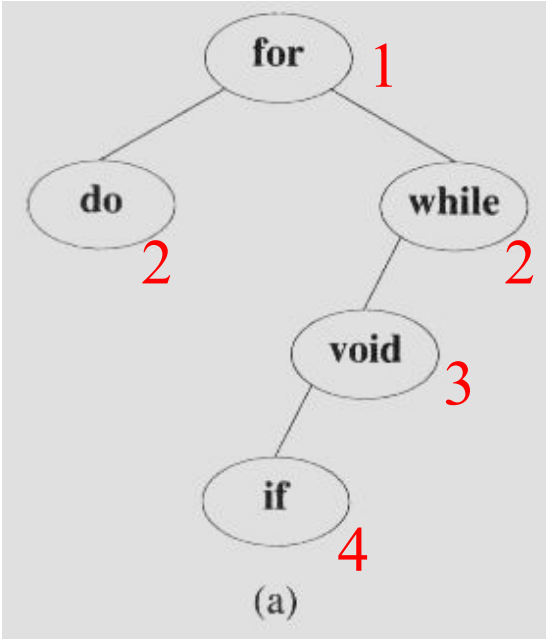
- **Example: binary search trees for 3, 7, 9, 12;**



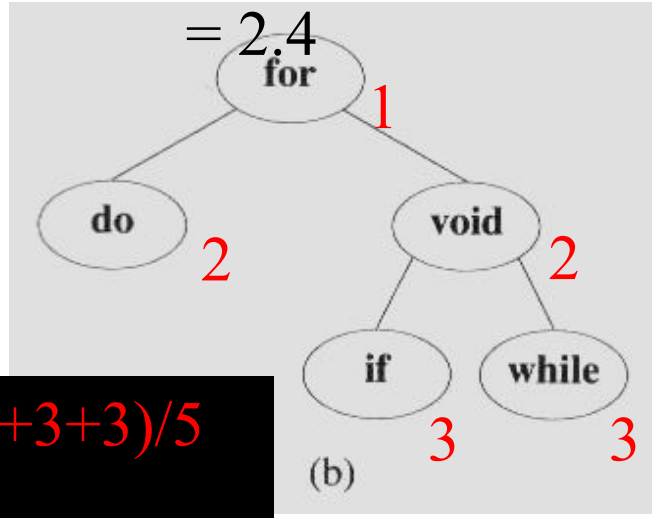
- A full binary tree may not be an optimal binary search tree if the identifiers are searched for with different frequency



- Consider these two search trees, If we search for each identifier with equal probability
  - In first tree, the average number of comparisons for successful search is 2.4.
  - Comparisons for second tree is 2.2.
- The second tree has
  - a better worst case search time than the first tree.
  - a better average behavior.



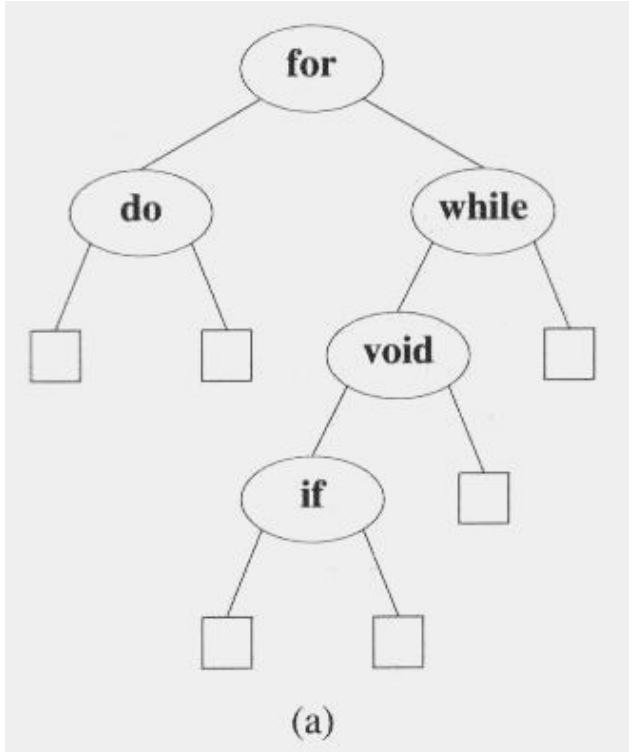
(1+2+2+3+4)/5
= 2.4



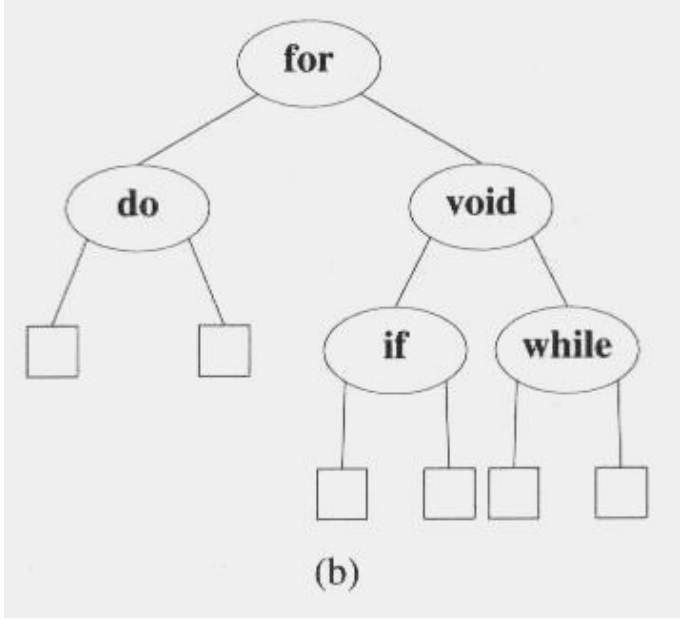
(1+2+2+3+3)/5
= 2.2

Optimal binary search trees

- In evaluating binary search trees, it is useful to add a special square node at every place there is a null links.
  - We call these nodes *external nodes*.
  - We also refer to the external nodes as *failure* nodes.
  - The remaining nodes are *internal nodes*.
  - A binary tree with external nodes added is an *extended binary tree*



(a)



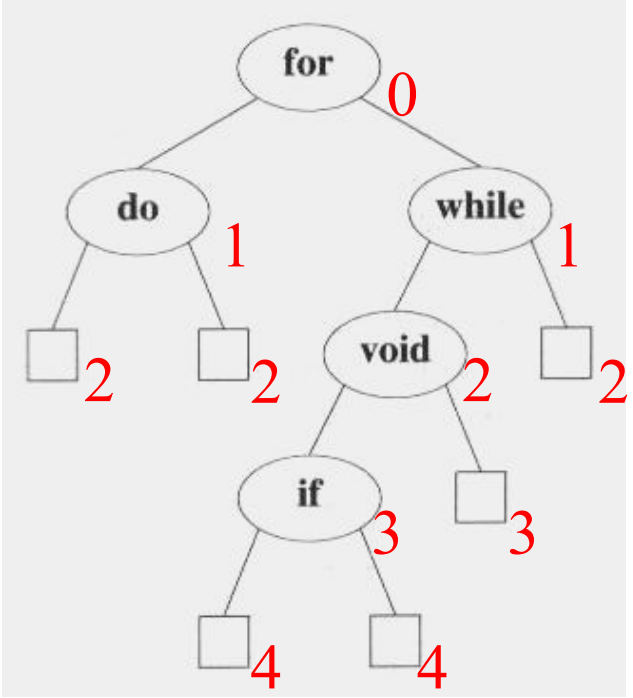
(b)

Optimal binary search trees

- *External / internal path length*
  - The sum of all external / internal nodes' levels.
- For example
  - Internal path length, *I*, is:

    $I = 0 + 1 + 1 + 2 + 3 = 7$
  - External path length, *E*, is :

    $E = 2 + 2 + 4 + 4 + 3 + 2 = 17$
- A binary tree with *n* internal nodes are related by the formula $E = I + 2n$



Optimal binary search trees

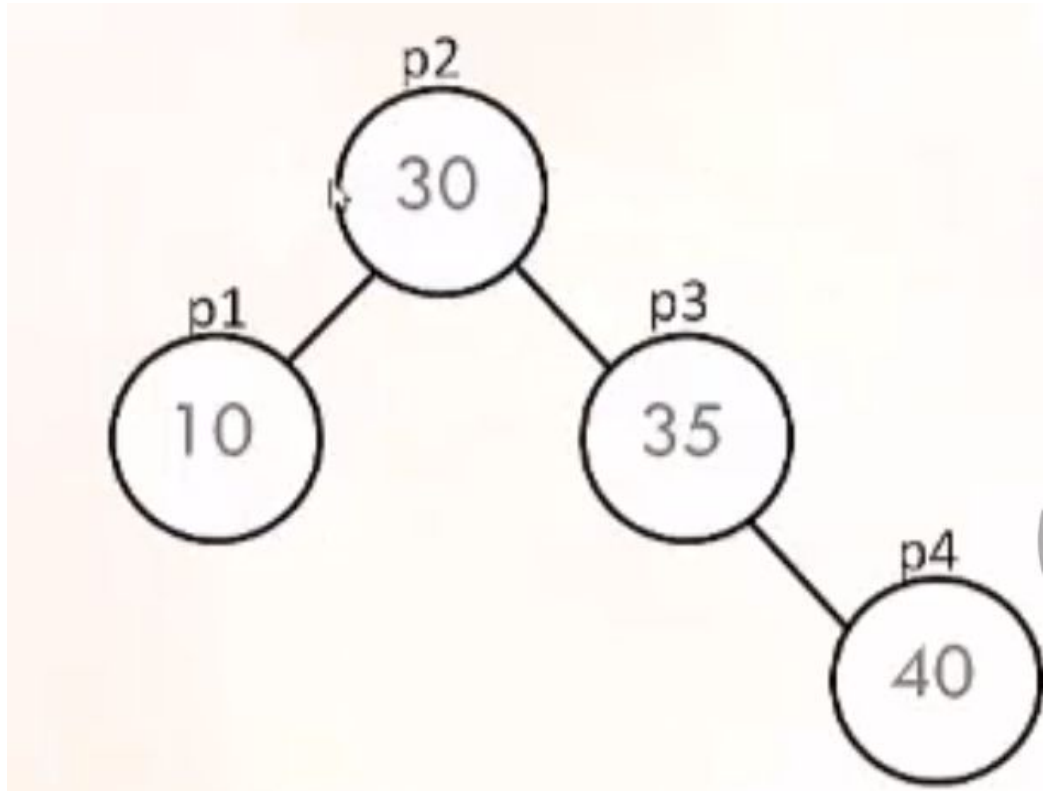
- n identifiers are given.

P_i, $1 \le i \le n$: Successful Probability

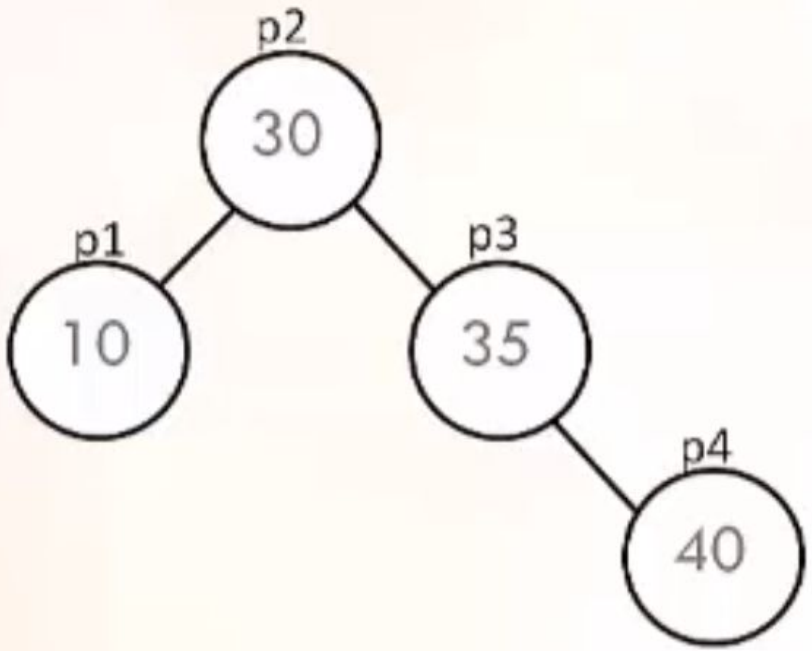
Q_i, $0 \le i \le n$: Unsuccessful Probability

Where,

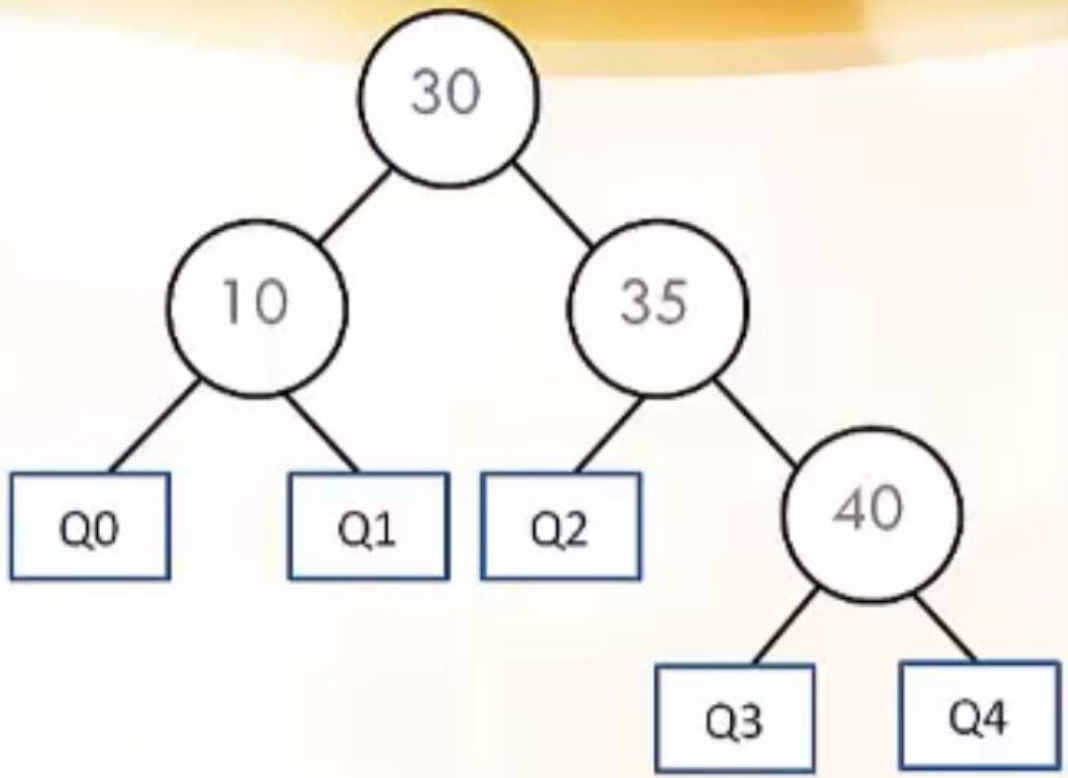
$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1$$



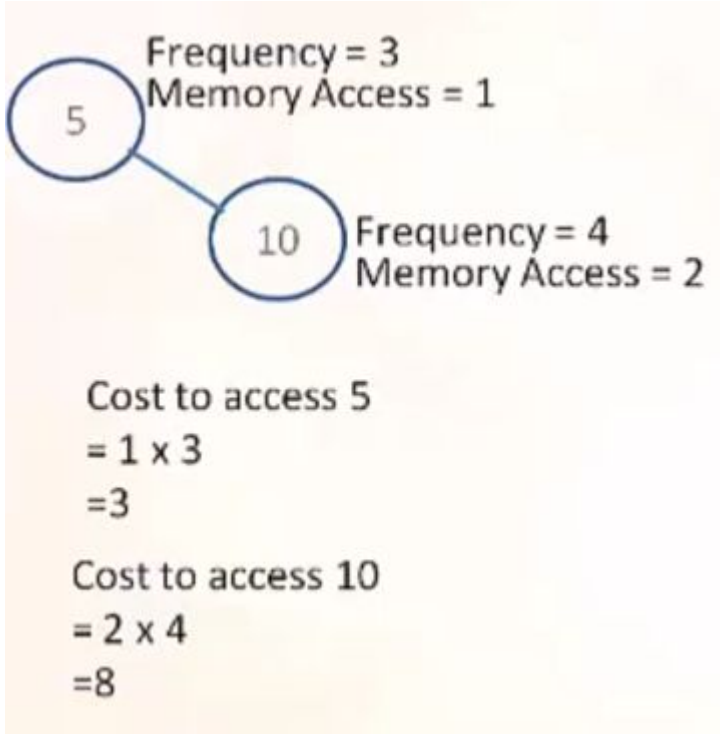
Probability of successful search



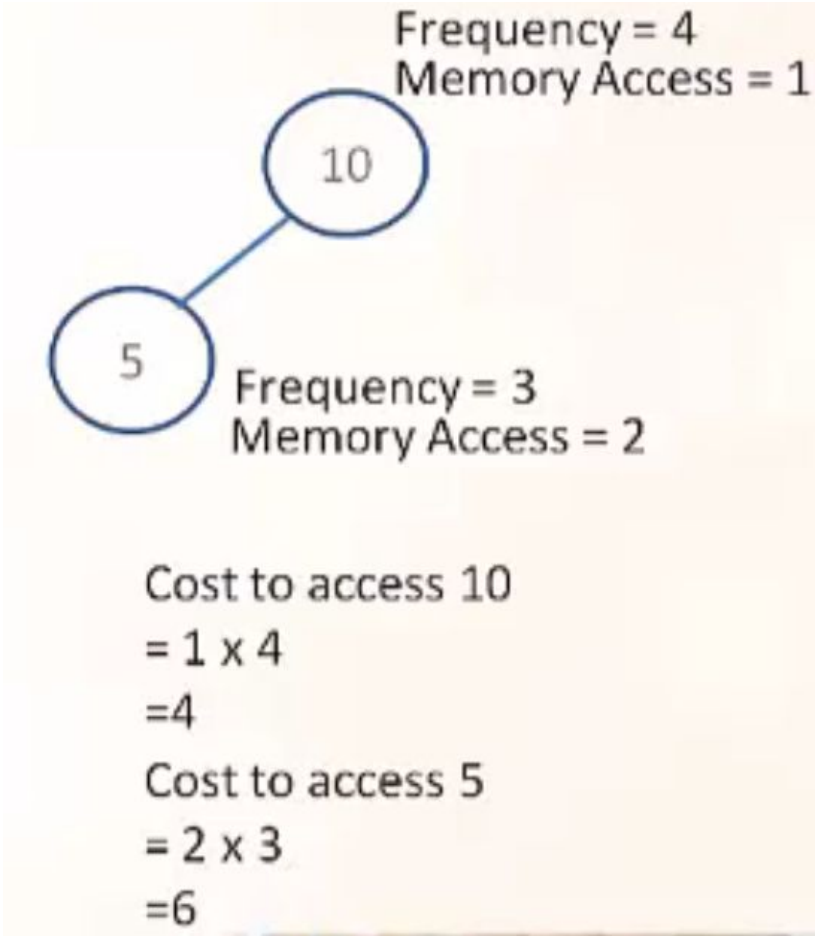
Probability of successful search



Probability of unsuccessful search



Total Cost = 3+8=11



Total Cost = 4+6=10

The dynamic programming approach for Optimal binary search trees

- To solve OBST , requires to find answer for Weight (w),Cost (c) and Root (r) by:
- W(i,j) =p(j) +q(j) +w(i,j-1)
- C(i,j)= min {c(i,k-1)+c(k,j)}+w(i,j) ….. i<k<=j
- r = k (for which value of c(i,j) is small

- C(i,i) = 0
- r (i,i) = 0
- w (i,i) = q(i)

Example

OBST Example

- Let $n = 4$,
- $(a_1, a_2, a_3, a_4) =$ (**do**, **if**, **int**, **while**)
- Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$
- $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$.

$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
$w_{01}=$ $c_{01}=$ $r_{01}=$	$w_{12}=$ $c_{12}=$ $r_{12}=$	$w_{23}=$ $c_{23}=$ $r_{23}=$	$w_{34}=1$ $c_{34}=0$ $r_{34}=0$	
$W_{02}=$ $c_{02}=$ $r_{02}=$	$W_{13}=$ $c_{13}=$ $r_{13}=$	$w_{24}=$ $c_{24}=$ $r_{24}=$		
$W_{03}=$ $c_{03}=$ $r_{03}=$	$W_{14}=$ $c_{14}=$ $r_{14}=$			
$W_{04}=$ $c_{04}=$ $r_{04}=$				

- C(i,i) = 0
- r (i,i) = 0
- w (i,i) = q(i)

$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$.

$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
$w_{01}=$ $c_{01}=$ $r_{01}=$	$w_{12}=$ $c_{12}=$ $r_{12}=$	$w_{23}=$ $c_{23}=$ $r_{23}=$	$w_{34}=1$ $c_{34}=0$ $r_{34}=0$	
$W_{02}=$ $c_{02}=$ $r_{02}=$	$W_{13}=$ $c_{13}=$ $r_{13}=$	$w_{24}=$ $c_{24}=$ $r_{24}=$		
$W_{03}=$ $c_{03}=$ $r_{03}=$	$W_{14}=$ $c_{14}=$ $r_{14}=$			
$W_{04}=$ $c_{04}=$ $r_{04}=$				

$$W(i,j) = p(j) + q(j) + w(i,j-1)$$

$$C(i,j) = \min \{c(i,k-1)+c(k,j)\}+w(i,j) \quad \ldots\ldots \quad i<k<=j$$

OBST Example

$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

W(i,j) =p(j) +q(j) +w(i,j-1)

$$w_{01} = p_1 + q_1 + w_{00} = 3 + 3 + 2 = 8$$

C(i,j)= min {c(i,k-1)+c (k,j)}+w(i,j) i<k<=j

$$c_{01} = \min_{0<k<=1} \{c_{00} + c_{11}\} + w_{01}$$

$$= \min \{0 , 0) + 8 = 8$$

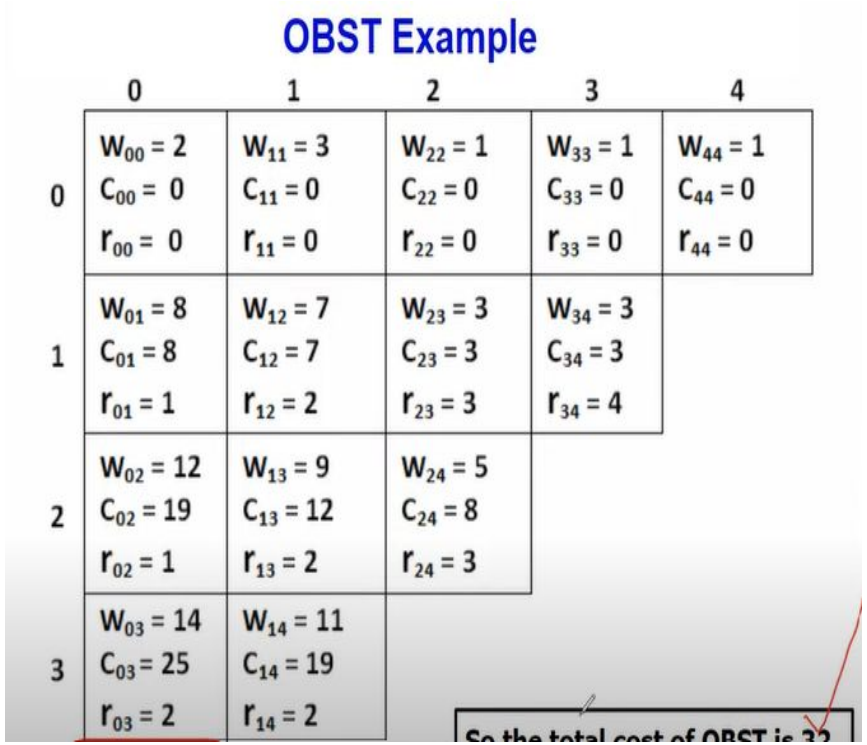
r = k (for which value of c(i,j) is small

$$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8$$
$$c_{01} = w_{01} + \min\{c_{00} + c_{11}\} = 8, r_{01} = 1$$

$$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7$$
$$c_{12} = w_{12} + \min\{c_{11} + c_{22}\} = 7, r_{12} = 2$$

$$w_{23} = p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3$$
$$c_{23} = w_{23} + \min\{c_{22} + c_{33}\} = 3, r_{23} = 3$$

$$w_{34} = p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3$$
$$c_{34} = w_{34} + \min\{c_{33} + c_{44}\} = 3, r_{34} = 4$$



W04=16
C04=32
R04=2

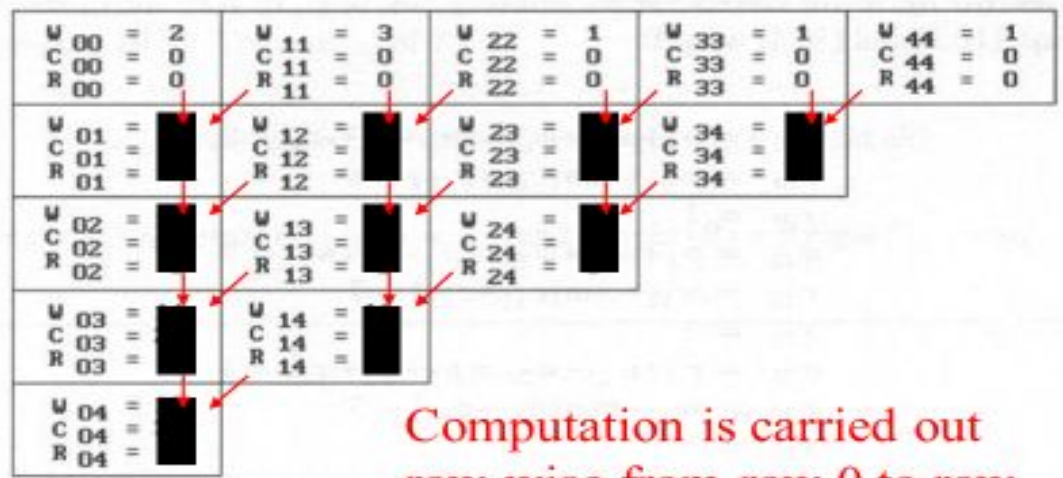
Optimal binary search trees

- $w_{ii} = q_i$
- $w_{ij} = p_k + w_{i,k-1} + w_{kj}$
- $c_{ij} = w_{ij} +$ [redacted]
- $c_{ii} = 0$
- $r_{ii} = 0$
- $r_{ij} = l$

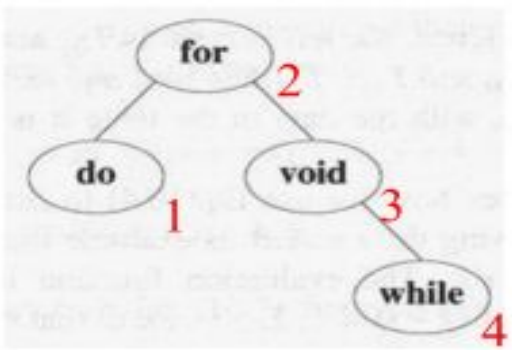
(a1, a2, a3, a4) = (do,for,void,while)

(p1, p2, p3, p4) = (3, 3, 1, 1)

(q0, q1, q2, q3, q4) = (2, 3, 1, 1, 1)



Computation is carried out row-wise from row 0 to row 4



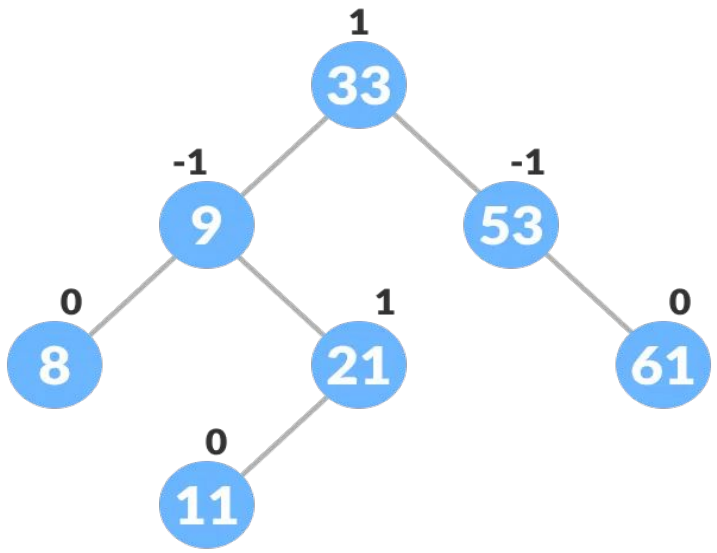
The optimal search tree as the result

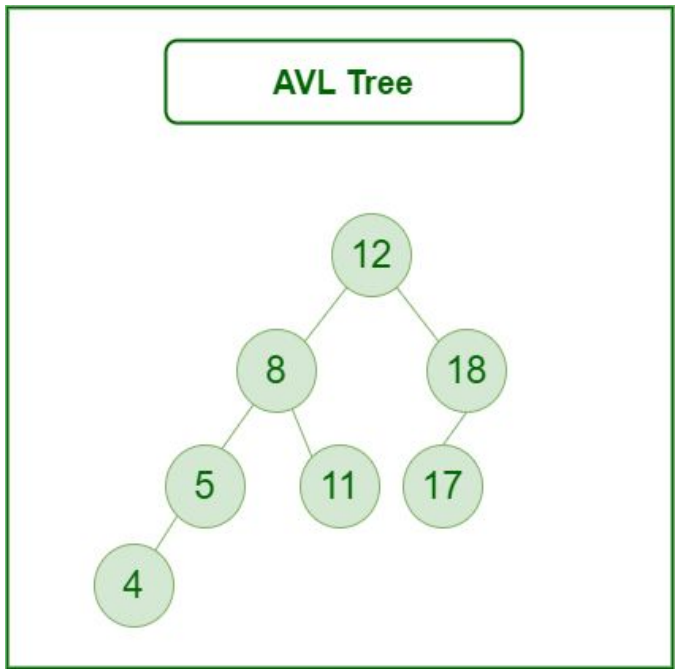
AVL

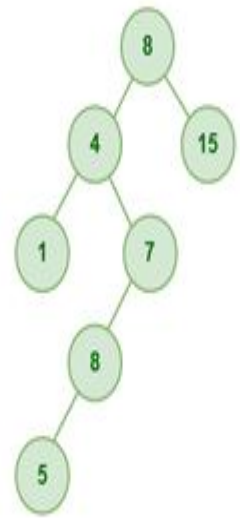
- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.
- Every AVL tree is also a Binary Search Tree but every BST is not AVL Tree
- time complexities(search, insert and delete, max, min, floor and ceiling) O(Log n)

Balance Factor

- Balance Factor = (Height of Left Subtree - Height of Right Subtree)
- The value of balance factor should always be **-1, 0 or +1.**







Not an AVL Tree

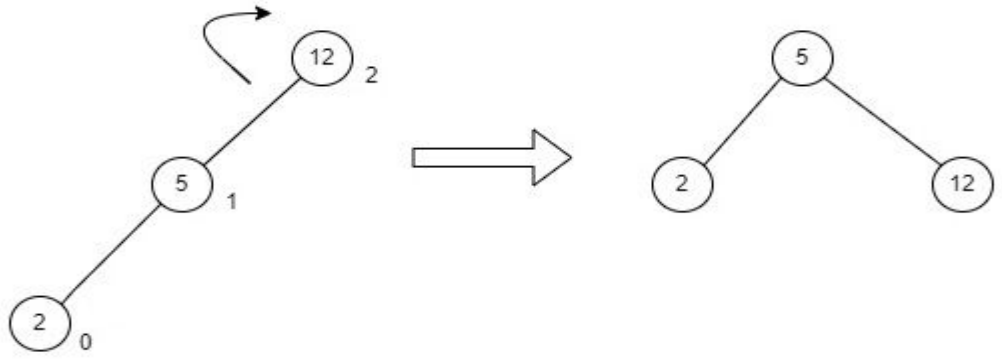
AVL Rotations

- Four cases of rotation in the balancing algorithm of AVL trees:

1. LL ->Left-Left
2. RR ->Right-Right
3. LR->Left-Right
4. RL->Right-Left

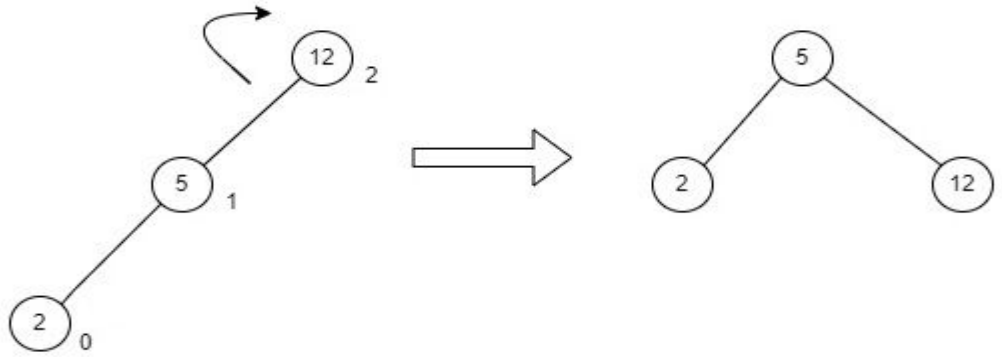
LL Rotation

- performed when the node is inserted into the right subtree leading to an unbalanced tree.
- This is a single left rotation to make the tree balanced again



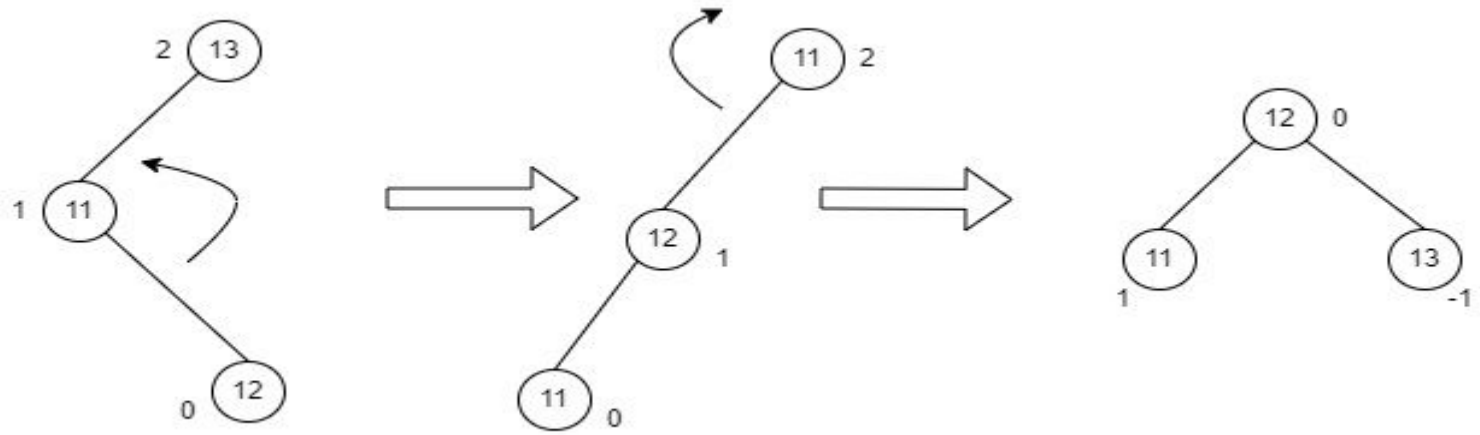
RR Rotation

- performed when the node is inserted into the left subtree leading to an unbalanced tree.
- This is a single right rotation to make the tree balanced again –



LR Rotation

- Double Rotation.
- It is performed when a node is inserted into the right subtree of the left subtree.
- The LR rotation is a combination of the **left rotation followed by the right rotation.**



RL Rotation

- Double Rotation.
- it is performed if a node is inserted into the left subtree of the right subtree.
- The RL rotation is a combination of the **right rotation followed by the left rotation**