**Knapsack Problem and Memory functions**

**Knapsack Problem**

Recall that the knapsack problem is an optimization problem. The goal is to fill a knapsack with capacity *W* with the maximum value from a list of items each with weight and value. The discrete knapsack includes the restriction that items cannot be spit, meaning the entire item or none of the item can be selected, the weights, values and capacities have integer values.

To design a dynamic programming algorithm we need to find a recursive relation from the smaller sub problems to larger problem.

Specify the weights and values in arrays *w*[1*...n*] and *v*[1*...n*]. This implies that we have listed the items in a sequence. The sub problems *i j* is then be the knapsack problem for items 1, ... ,*i* with values *v*[1 ...  *i*], weights *w*[1 ... *i*] and the capacity of the knapsack is *j*. The value of the optimal solution is *V*[*i*, *j*]. The feasible solutions can be divided into two sets, those composed of only the *i*-1 proceeding items and those using the *i*-th item.

1. The optimal solution the for the sub problem that does not include the *i*-th item and capacity *j* is *V*[*i*-1, *j*].

2. The optimal solution for the sub set that include the *i*-th item can be written

*V*[*i*-1, *j*-*w*[*i*]] + *v*[i]

where *j*-*w*[*i*] should not be negative and assures that the sub-problem knapsack has remaining capacity to include the *i*-th item.

Then the recurrence relation is

*V*[*i*, *j*] = max{ *V*[*i*-1, *j*], *V*[*i*-1, *j*-*w*[*i*]] + *v*[i] } if *j*-*w*[*i*] ≥ 0

= *V*[*i*-1, *j*] if *j*-*w*[*i*] < 0

We need initial conditions (IC) and they are

*V*[0,*j*] = 0 for *j* ≥ 0 and *V*[*i*, 0] = 0 for *i* ≥ 0

which means that an empty knapsack has no value.

Then we can fill out *V* row wise

illustrate the algorithm

illustrate which previous cells are being used.

Cost is Θ(*nW*)

How would you construct the list of selected items?

Work backward and check the value above, meaning *V*[*i*-1, *j*].

If  *V*[*i*-1, *j*] == *V*[*i*, *j*] then the *i*-th item was not chosen, so move on to *V*[*i*-1, *j*].

If *V*[*i*-1, *j*] ≠ *V*[*i*, *j*] then the *i*-th item was chosen, so move on to *V*[*i*-1, *j*-*w*[*i*]] + *v*[i].

Repeat the check.

What is the cost? Θ(*n+W*)

**Memory Functions**

Dynamic programming solves problems that have a recurrence relation.  Using the recurrences directly in a recursive algorithm is a **top-down technique**. It has the disadvantage that it solves common sub problem multiple times. This leads to poor efficiency, exponential. The dynamic programming technique is bottom-up, and solving all the sub-problems only once. This has the disadvantage that some of the sub-problems may not have been necessary to solve. Illustrate in the table. We would like to have the best of both worlds, i.e. all the necessary sub-problem solved only once. This is possible using **memory** **functions**.

The technique uses a top-down approach, recursive algorithm, with table of sub-problem solution. Before determining the solution recursively the algorithm checks if the sub problem has already been solved by checking the table. If the table has a valid value then the algorithm uses the table value else it proceeds with the recursive solution.

Memory function algorithm for the knapsack problem initializes *V*[*i*, *j*] to -1 except row 0 and column 0, which is initialize to 0.

Algorithm *MFKnapsack*(*i*, *j*) // *i*, *j* represent the sub problem

**if** *V*[*i*, *j*] < 0 // meaning not already calculated

**if** *j* < *Weights*[i] **then**

*value* ← *MFKnapsack*(*i*-1, *j*)

**else**

*value* ← max(*MFKnapsack*(*i*-1, *j*), *Values*[*i*] + *MFKnapsack*(*i*-1, *j*-*Weights*[*i*])

*V*[*i*, *j*] ← *value* // put valid value in the table for both cases

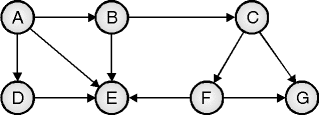
**return** *V*[*i*, *j*]

**Topological Sorting**

Topological sort is applicable on directed acyclic graphs only. So let us first understand what we mean by the directed acyclic graph.

**Directed Acyclic Graph**

A graph is called a Directed Acyclic Graph (DAG) if the graph is a directed graph and it has no directed cycles. DAG is used to find common sub expressions in optimizing compilers. The graph shown below is an example of a directed acyclic graph.

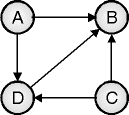


**Working Mechanism**

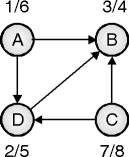
* Topological sorting of directed graph is linear ordering of its vertices. In this sorting, for every directed edge from u to v, u comes before v in ordering. This is used to simulate tasks and their ordering. In graph, each node may represent one task and every edge may represent constraint that one task should be complete before another.
* Topological sorting is possible only if graph is DAG. Every DAG has at least one topological sorting sequence. If graph is not DAG, topological sorting is not possible.
* Topological sorting is different than DFS. In topological sorting, vertex is print before its adjacent vertices.

In simplest way, to find topological ordering of graph, we should perform following steps :

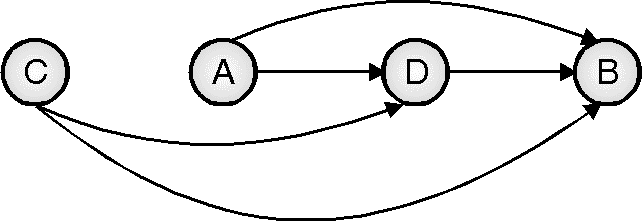
* Visit graph in DFS order and label the vertices with their discovery and finish time.
* Linearly arrange vertices in decreasing order of their finish time from left to right
* Join the vertices using forward arcs in linear ordering, corresponding to graph

****

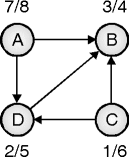
* There may exist more than one DFS forest of graph, so multiple topological ordering for same graph is possible. Arc can only go in forward direction. For above graph, two different DFS traversal and their linear ordering is shownbelow :

****

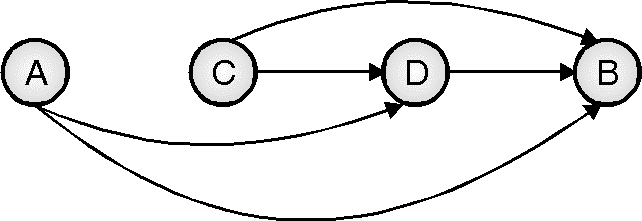
**DFS traversal sequence 1**

****

**Topological sequence**

****

**DFS traversal sequence 2**

****

**Topological sequence**

Algorithm for Topological Sort

The algorithm for topological sorting is shown below:

Algorithm

TOPOLOGICAL\_SORT

// Input to algorithm is graph G = <V, E>

Call DFS

* Display vertices in decreasing order of their finish timeComplexity Analysis
* This algorithm scans edges and nodes once to order them. So running time is linear in a number of edges and vertices, i.e. O(|V| + |E|).
* Applications of Topological Sort
* Project management.
* Event management.
* Job scheduling.
* Instruction scheduling.
* Order of compilation tasks.
* Data serialization.
* Ordering of cell evaluation in formulas in spread sheet.

**Transform and conquer: Presorting**

**Balanced search trees**

Balanced Binary trees are computationally efficient to perform operations on.

A balanced binary tree will follow the following conditions:

* The absolute difference of heights of left and right subtrees at any node is less than 1.
* For each node, its left subtree is a balanced binary tree.
* For each node, its right subtree is a balanced binary tree.

**Height-balanced Binary Trees**

Balanced binary trees are also known as height-balanced binary trees. Height balanced binary trees can be denoted by HB(k), where k is the difference between heights of left and right subtrees. ‘k’ is known as the balance factor.

If for a tree, the balance factor (k) is equal to zero, then that tree is known as a fully balanced binary tree. It can be denoted as HB(0).

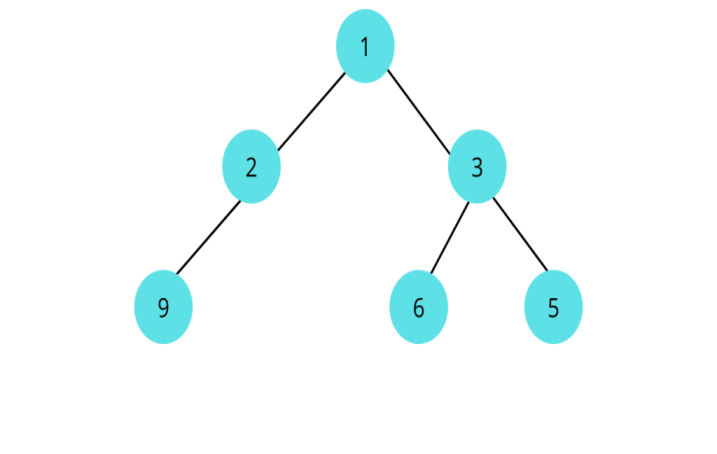
## 

**Self Balancing Binary Search Tree**

If a binary search tree has a balance factor of one then it is an AVL ( Adelso-Velskii and Landis) tree. This means that in an AVL tree the difference between left subtree and right subtree height is at most one.

AVL tree is a self-balancing binary search tree. In an AVL tree if the difference between left and right subtrees is greater than 1 then it performs one of the following 4 rotations to rebalance itself:

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation



**How to Check if a Binary Tree is balanced?**

To check if a Binary tree is balanced we need to check three conditions :

The absolute difference between heights of left and right subtrees at any node should be less than 1.

For each node, its left subtree should be a balanced binary tree.

For each node, its right subtree should be a balanced binary tree.

We will need a function that can calculate the height of the tree. One way to do this is to write a separate function for calculating the height and call it every time height is needed. This is going to be computationally inefficient.

The better way to implement this will be returning height in the same function.

For each node, we will return -1 if it is not balanced and the height of that node/subtree if it is balanced.

**The algorithm is as follows :**

If node == null -> return 0

Check left subtree. If not balanced -> return -1

Check right subtree. If not balanced -> return -1

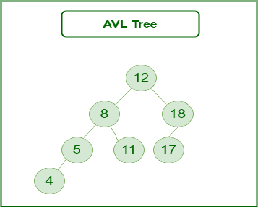
The absolute between heights of left and right subtrees. If it is greater than 1 -> return -1.

If the tree is balanced -> return height

**AVL trees -2-3 Trees**

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

### Example of AVL Trees:

****

Operations on an AVL Tree:

* Insertion
* Deletion
* Searching

**Insertion in AVL Tree:**

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.   
Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

* Left Rotation
* Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

y x

/ \ Right Rotation / \

x T3 - - - - - - - > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

# Deletion in an AVL Tree

**Steps to follow for deletion**.   
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation
2. Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)

or x (on right side)

y x

/ \ Right Rotation / \

x T3 – - – - – - – > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from [insertion](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/)here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   1. y is left child of z and x is left child of y (Left Left Case)
   2. y is left child of z and x is right child of y (Left Right Case)
   3. y is right child of z and x is right child of y (Right Right Case)
   4. y is right child of z and x is left child of y (Right Left Case)

**Insert Operation in an AVL Tree**

The insertion procedure is similar to that of a normal AVL tree without a parent pointer, but in this case, the parent pointers need to be updated with every insertion and rotation accordingly. Follow the steps below to perform insert operation:

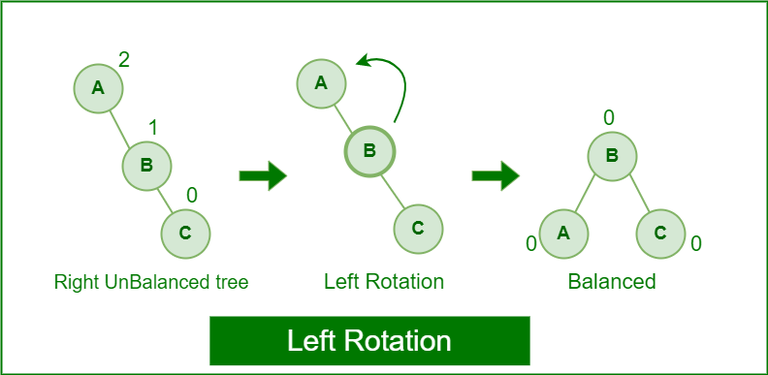
* [Perform standard BST insert for the node](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) to be placed at its correct position.
* Increase the height of each node encountered by 1 while finding the correct position for the node to be inserted.
* Update the parent and child pointers of the inserted node and its parent respectively.
* Starting from the inserted node till the root node check if the AVL condition is satisfied for each node on this path.
* If **w** is the node where the AVL condition is not satisfied then we have 4 cases:
  + **Left Left Case:** (If the left subtree of the left child of **w** has the inserted node)
  + **Left Right Case:** (If the right subtree of the left child of **w** has the inserted node)
  + **Right Left Case:** (If the left subtree of the right child of **w** has the inserted node)
  + **Right Right Case:** (If the right subtree of the right child of **w** has the inserted node)

**Rotating the subtrees in an AVL Tree:**

An AVL tree may rotate in one of the following four ways to keep itself balanced:

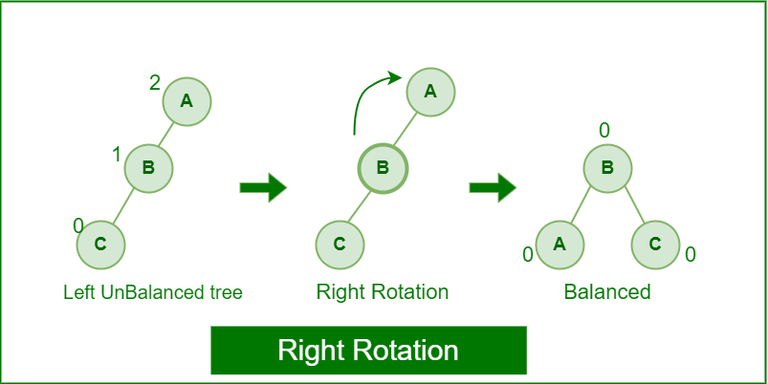
**Left Rotation:**

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



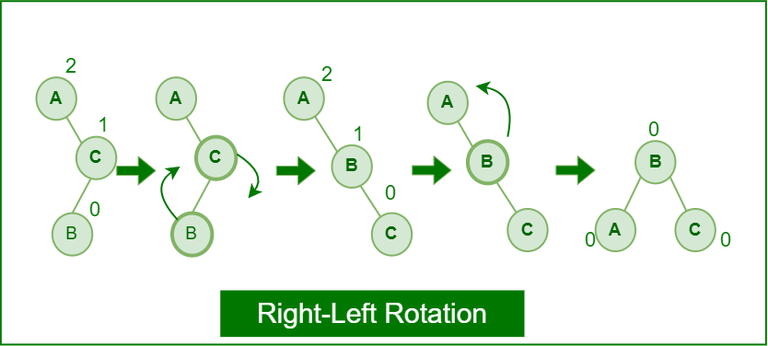
**Right Rotation:**

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



**Right-Left Rotation:**

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



### Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary
4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

### Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

### Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.