DATA STRUCTURES

HASHING

Hashing is basically a searching technique with the least amount of complexity. For example, Linear Search and Binary Search have worst case time complexity of $O(n)$ and $O(\log n)$ respectively. However, for hashing the complexity is $\Theta(1)$ for **each and every case.** The concept of Big O and Big Θ will be dealt in Algorithms. In short, the complexity is **independent of the size of the array.**

Direct Address Table (DAT)

The DAT enables the hashing process to search with the complexity of $\Theta(1)$. The DAT is basically an array –

DATA										
INDEX	0	1	2	3	4	5	6	7	8	9

Now, let us assume we have data files as shown below –

KEY	**FILE_NAME**
0	ABC
2	DEF
4	GHI
7	JKL
9	MNO

Then, the data will be stored in the DAT with the key and the index being the same as shown below –

DATA	ABC		DEF		GHI			JKL		MNO
INDEX	0	1	2	3	4	5	6	7	8	9

Now, if we need to find the file with key 0, we can simply access DAT[0] and get the file. Hence, the complexity for the search will be $\Theta(1)$. While we explore this, we can see a major **drawback** with DAT. Suppose we now have another file with key 1024. In this case, we will need to have a DAT of 1024 size so as to accommodate the file. Hence, we will end up with a DAT of size 1024 to store 6 files.

Therefore, **DAT works best if the keys are continuous or are comparable**. If there are cases where the keys have a large variance, then the DAT is not a feasible solution.

Hash Function

This is the solution to fix the drawback of the DAT space problem. The hash function basically performs a MOD division of the file key with the length of the DAT. In short, **Hash function is a function to map between file keys and DAT index.** This way, we scale down the file key to the range of the DAT key. For example, let us assume the files as follows –

KEY	**FILE_NAME**
100	ABC

13252	DEF
567332244	GHI
77	JKL
110229	MNO

In this case, the length of DAT is 10. So, we take the keys and MOD it by 10 and use the result as the key storage in DAT as shown below –

DATA	ABC		DEF		GHI			JKL		MNO
INDEX	0	1	2	3	4	5	6	7	8	9

So now, even if the file keys are large and have a large variance, we have compressed them to a value between 0 – 9. The hash function we applied now is called **Division Modulo Function.**

However, there is a major drawback with the Division Modulo Hash function. Suppose we have two keys – 125 and 625. In this case, the hash function will return index 5 for both the keys. So both the keys will try to be stored in the given index. This is called **collision.**

Types of Hash Function

We have already seen that the Division modulo hash function is not the true solution here. We can use the following suggestions to improve the Division Modulo method –

- Don't pick the length of the DAT to be a power of 2. Suppose the length of the DAT is 2^k, then the modulo will be the last k – bits from the LSB side. Thus, the rest of the bits are not considered hence causing more collisions.
- Pick the length of DAT to be a prime number that is not close to a power of 2.

Additionally, we have various other hash functions as well –

- Digit Extraction Method
- Mid square method
- Folding method

Digit Extraction Method

This is also called the **Truncation method** as well. In this method, we extract digits from the key instead of performing modulo. This ensures that we don't ignore the MSB bits and also prevents collisions. For example, let the length of the DAT be **1000.** Hence, the indices range from **0 – 999.** Now, if we extract 1 digits, we get 10 unique addresses. Similarly, if we extract 2 digits, we get 90 unique addresses. Finally, if we extract 3 digits, we can get 900 unique addresses. Hence, **taking 3 digits will ensure minimum collision.** Suppose, we have –

$$Key = 725648891$$

Here, let us extract the 3rd, 6th and 9th digits to get the index (just an example). Then, we get the key to be stored in index **581.** Since this method doesn't involve any calculations, it is rigid and is therefore worse of the lot. It is also worse than Division Modulo function.

Mean Square Method

Here is the step-by-step process for this function –

- Take the key and square it
- Then take the middle digit of the squared number. If the number of digits in the square is even, then there will be 2 middle digits.
- Then, take the $k - 1$ digits from the middle to get the index (given the number of addresses in the DAT is k digits).

For example, let us have a 1000 ($k = 4$) length DAT where we need to store 8312 key. Squaring the key, we get 69089344. Here, we have 2 middle digits – 8 and 9. Now, we select the 3-digit index from the middle to get indices = 089 and 893.

Since we are squaring and then selecting the digits, it is difficult to get data with the same index. Hence, this has lesser collisions.

Fold Boundary Method

Here is the step-by-step process for this function –

- Take the first and last $k - 1$ digits of the key. This is given the length of DAT is of k digits.
- Add the 2 $k - 1$ digit numbers.
- If the result is of $k - 1$ digits, then that is the index.
- Else if the result is more than $k - 1$ digits, then repeat the process for the sum.

For example, let the size of DAT be 1000 and the key be 998123767. Then we select the first and last 3 digit groups – 998 and 767. By adding them, we get the sum as 1765. Since the sum is more than 3 digits, then we repeat the process to get the 3 digits groups – 176 and 5. Adding them, we get $\mathbf{181}$ which is the index.

Fold Shifting Method

This is the same as the Flod Boundary method but in this case, the groups are made across the key and not just at the front and back. So in our previous example, there will be 3 groups – 998, 123 and 767. Adding them gives us 1888. Repeating the process, we get 2 groups = 188 and 8. Hence, we get the final index as **196.**

Overall, we get the following statement –

"*Mid-square and Fold shifting methods are the best as they involve all the digits. The next comes Division Modulo as it involves the value of the entire key. Next comes Fold Boundary method as it involves only the first and last groups of digits. The worst of the lot is the Digit extraction method which involves minimum number of digits.*"

COLLISION RESOLUTION TECHNIQUES

No matter which hash function we use, we can never achieve 0% collision. There will always be some chance of collision occurring. Hence, we need to use a collision resolution technique to resolve any collision that might occur.

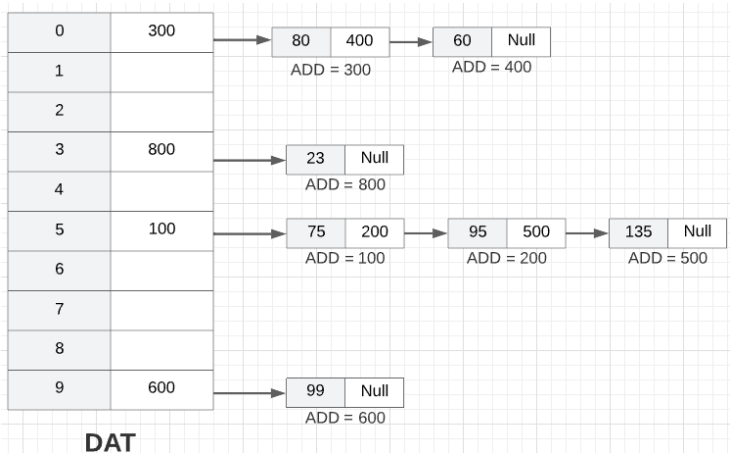
CHAINING TECHNIQUE

Let us assume there is a DAT with size 10 and the following keys are need to be stored –

$$Keys = 75, 80, 95, 99, 23, 135, 60$$

Assume we are using the Division Modulo hash function. Here, we can see that there is a clear lack of space and there will be collisions. So, to resolve these collisions, we shall resort to the **Chaining technique.**

Basically, we are no longer storing the data of the keys in the DAT. Instead, we shall use an **external linked-list** structure to store this data. In short, the keys will be stored as shown below –



$$Average\ chain\ length = \frac{2+0+0+1+0+3+0+0+0+1}{10} = \mathbf{0.7}$$

As we can see, no element is stored in the DAT. This is all **external storage.** We only store the addresses in the DAT.

Maximum chain length (worst case) possible for n keys and m slots is $\boldsymbol{n}$**.** This will be the case when the hash function will assign all the keys to the same index slot. The **Minimum chain length** (best case) possible will be $\mathbf{1}$**.** Thus, we need to use a hash function that is closest to the best case scenario.

In the above example, let us assume we need to find key number 135. To do so, we first apply the hash function which will let us know that the value is stored in Index 5. In Index 5, we need to perform a linear search on the linked lists till we can find the value. Hence, for the best case scenario the element will be present as the first element in the index. Hence, the time complexity will be $\boldsymbol{O(1)}$**.** On the other hand, for worst case scenario we need to parse through the index via a Linear Search to get the element. Hence, the time complexity becomes $\boldsymbol{O(n)}$**.**

Now, let us look at insertion time complexity. Suppose we need to insert another key in the DAT. In this case, we will simply insert the element at the beginning of the linked lists. We don't care for the order which means that regardless if it is the best or worst case, we insert the element in the beginning

itself. Hence, the time complexity will be $\boldsymbol{O(1)}$**.** However, if we need to insert the element **only if the element is not present in the index,** then we need to do a linear search through the linked lists to see if the element to be inserted is already present or not. In the worst case for this scenario, we get the complexity as $\boldsymbol{O(n)}$**.**

Finally, let us look into Deletion time complexity. Deletion can only occur after we search the linked lists using linear search and then delete the element. So, in worst case scenario, we get the complexity as $\boldsymbol{O(n)}$**.** In the best-case scenario, the element will be the first in the linked list array. So, the best-case scenario will be $\boldsymbol{O(1)}$**.**

NOTE – As we saw in the Chaining technique, we need to allocate memory to store the keys. There are multiple ways to allocate memory in C –

COMMAND	EXPLANATION
`int a`	The command will allocate 2B of memory for int in the **Stack area.**
`malloc(2)`	The command will dynamically allocate 2B of memory in the **Heap area.**
`static int a`	The command will allocate 2B of memory for int in the **Static area.**

Drawbacks and Advantages

For chaining, we can see that the complexity even for hashing has been increased to $\boldsymbol{O(n)}$ for worst case scenarios. At the same time, we need additional storage for linked lists. However, the biggest advantage of chaining is that despite the small size of the DAT, it can **accommodate infinite collisions** as it stores the keys in external linked lists.

OPEN ADDRESS TECHNIQUES

In these techniques, the data is stored in the DAT itself and hence, the maximum number of data keys stored is equal to the size of DAT. Open address techniques is actually an umbrella term and includes a few techniques under it.

Linear Probing

In this case, the mapping function is given as follows –

$$LP(key, i) = [HF(key) + i] \; mod \; M \qquad \forall \; i \in [0, 1, 2 \ldots M-1]$$

Where $HF(key)$ is the hash function applied for that key. For simplicity, we shall assume a Division Modulo HF. Also, M stands for the size of DAT. Let us take an example –

$$M = 10$$

$$Keys = [55, 79, 62, 85, 96, 90, 45, 57, 78, 69]$$

Like mentioned before, we are using Division modulo HF. Let us take the first element i.e. **55**

$$HF(55) = 55 \; mod \; 10 = 5$$

$$LP(55, 0) = [5 + 0] \; mod \; 10 = 5$$

Since the slot at index 5 is free, there is no collision and 55 gets stored at index 5. Similarly, elements 79 and 62 also get stored in indices 9 and 2 respectively without collisions.

Now, we encounter the element **85.** We know that $HF(85) = 5$. So, we get –

$$LP(85, 0) = [5 + 0]\ mod\ 10 = 5$$

Here, we face a collision. Thus, we increment the value of i and try again –

$$LP(85, 1) = [5 + 1]\ mod\ 10 = 6$$

We can see that index 6 is not filled yet so we can store 85 in index 6. Hence, element 85 is stored at index 6 **after 1 collision.**

Now that we have a hang of this, we can write –

ELEMENT/KEY	FINAL INDEX	NO OF COLLISIONS
55	5	0
79	9	0
62	2	0
85	6	1
96	7	1
90	0	0
45	8	3
57	1	4
78	3	5
69	4	5

The DAT was mapped completely but with **19 collisions** in total. Usually, in GATE exam they ask a NAT based on this. The linear probing is a **slower mapping** technique but at the same time will ensure all elements will **get mapped.**

In Linear Probing, the time required for searching and storing **is the same** as the process is the same. In best case, we can find an empty slot (for storing) or the required element (for searching) in the first check. Hence, the best case time complexity is $\boldsymbol{O(1)}$**.** On the other hand, for worst case scenario we would have to traverse the entire DAT to either get an empty slot (for storing) or the required element (for searching). Hence, for worst case scenario we have time complexity of $\boldsymbol{O(m)}$**.**

<u>Quadratic Probing</u>

In this type of probing, we simply replace i with i^2. So, the formula becomes –

$$QP(key, i) = [HF(key) + i^2]\ mod\ M \qquad \forall\ i \in [0, 1, 2 \dots M - 1]$$

Now if we solve the above problem, we will get –

ELEMENT/KEY	FINAL INDEX	NO OF COLLISIONS
55	5	0
79	9	0
62	2	0
85	6	1
96	7	1
90	0	0

45	4	3
57	8	1
78	3	5
69	NA	10

NOTE – As we can see, for the same problem the Linear Probing was able to map all the values but Quadratic Probing wasn't able to map all the keys. Thus, we can conclude that Linear Probing will always map all the values if there is an empty slot but Quadratic Probing may not map all the elements. That is why Linear Probing is also called **Closed Hashing.** At the same time, Quadratic Probing is **faster than** Linear Probing.

Double Hashing

As the name suggests, this collision resolution technique requires 2 Hash functions. For the above-mentioned example, let us define the 2 hash functions as follows –

$$HF_1(key) = key \ mod \ M$$

$$HF_2(key) = 1 + [key \ mod \ (M-2)]$$

Now, we can define the double hashing function as follows –

$$DH(key, i) = \left[HF_1(key) + \left(i * HF_2(key)\right)\right] mod \ M \qquad \forall \ i \in [0, 1, 2 \dots M-1]$$

Let us try to map the first element i.e. 55 as follows –

$$DH(55{,}0) = [(55 \ \% \ 10) + (0 * \{1 + 55 \ \% \ 8\})] \ \% \ 10 = \mathbf{5}$$

Since the DAT is empty to begin with, there are no collisions and we can map 55 to index 5. Similarly, we can map 79 and 62 to indices 9 and 2 respectively without any collisions. Now, let us try mapping key 85 –

$$DH(85{,}0) = [(85 \ \% \ 10) + (0 * \{1 + 85 \ \% \ 8\})] \ \% \ 10 = \mathbf{5} - \boldsymbol{Collision}$$

$$DH(85{,}1) = [(85 \ \% \ 10) + (1 * \{1 + 85 \ \% \ 8\})] \ \% \ 10 = \mathbf{1}$$

Thus, element 85 gets mapped to index 1 after 1 collision. We can write the same for the rest of the elements as well –

ELEMENT/KEY	FINAL INDEX	NO OF COLLISIONS
55	5	0
79	9	0
62	2	0
85	1	1
96	6	0
90	0	0
45	7	2
57	3	3
78	8	0
69	NA	10

PRIMARY CLUSTERING PROBLEM (LINEAR PROBING)

To understand this, let us take a problem as follows –

- $M = 10$
- Keys = 54, 63, 90, 81, 22, 80, 50
- Hash function = Division Modulo
- Collision Resolution technique = Linear Probing

In this case, we will have no collision till key 22. At that time, the DAT will look like this –

INDEX	KEY
0	90
1	81
2	22
3	63
4	54
5	
6	
7	
8	
9	

Now, we take the case of key **80 –**

$$LP(80,0) = 0 - Collision$$

$$LP(80,1) = 1 - Collision$$

$$LP(80,2) = 2 - Collision$$

$$LP(80,3) = 3 - Collision$$

$$LP(80,4) = 4 - Collision$$

$$LP(80,5) = 5 - No\ Collision$$

Hence, the DAT now becomes –

INDEX	KEY
0	90
1	81
2	22
3	63
4	54
5	80
6	
7	
8	
9	

Now, for key **50** we have –

$$LP(50,0) = 0 - Collision$$

$$LP(50,1) = 1 - Collision$$

$$LP(50,2) = 2 - Collision$$

$$LP(50,3) = 3 - Collision$$

$$LP(50,4) = 4 - Collision$$

$$LP(50,5) = 5 - Collision$$

$$LP(50,6) = 6 - No\ collision$$

Finally, the DAT looks like this –

INDEX	KEY
0	90
1	81
2	22
3	63
4	54
5	80
6	50
7	
8	
9	

The problem that arises here is that for both 80 and 50, the LP technique had to perform redundant checks from index 0 to 5/6. This was because the elements are all **clustered** in a sequence together and this causes unnecessary checks and collisions. This is called the **Primary Clustering problem** and is a drawback of the Linear Probing technique since LP checks the slots one by one in a sequence (linearly).

In short, "*If two keys contain the same starting hash address, then they will follow the same path and have the same collisions in a linear manner. This increases the average search time and is known as* ***Primary Clustering.***"

As we have seen in Linear Probing, we saw that the average searching/storing time complexity is $\boldsymbol{O(1)}$**.** However, we haven't accommodated the primary clustering problem. The time taken in the primary clustering problem can be given as –

$$Time\ taken = 1(for\ 1^{st}\ element) + 2\big(for\ 2^{nd}\ element\big) + \cdots + M = \frac{M(M-1)}{2}$$

Average time taken will be –

$$T_{avg} = \frac{M(M-1)}{2M} \approx \frac{M}{2}$$

Hence, now the time complexity has been changed to $\boldsymbol{O(m)}$**.**

DELETION IN OPEN ADDRESSING

Let us take the case of DAT as follows –

INDEX	KEY
0	90
1	81

2	22
3	63
4	54
5	80
6	50

Now, let us assume that element **22** was deleted. In that case, the DAT becomes –

INDEX	KEY
0	90
1	81
2	
3	63
4	54
5	80
6	50

Now, we are trying to search for element **80.** Using Division modulo, we first reach index 0. The value is 90, so we go for the next slot. The value is 81 and we go for the next slot. Here, we see that there is an empty slot. In Open addressing, if we encounter an empty slot, we can assume that the element was not mapped. Hence, **even though 80 is present in the DAT, we assume that the element is not mapped as we encounter an empty slot.**

Hence, deletion is cumbersome in this scenario. This was not an issue in Chaining as we used to store the keys externally. To overcome this scenario, we replace the deleted element with **$** symbol. That indicates that initially an element was present but has been subsequently deleted. Once we delete enough values, we will have more **$** symbols compared to actual value. In that case, we can just do a re-hashing to clean up the table.

SECONDARY CLUSTERING PROBLEM (QUADRATIC PROBING)

This is faced in quadratic probing. If two keys start from the same hash address in a quadratic manner, then they will follow the same path (redundant). This causes the searching time to increase, but not as much as Linear Probing. Hence, this is termed as **Secondary Clustering.**

Just like Linear Probing, the time complexity for searching, insertion and deletion in best and worst case scenarios are $\boldsymbol{O(1)}$ and $\boldsymbol{O(M)}$ respectively.

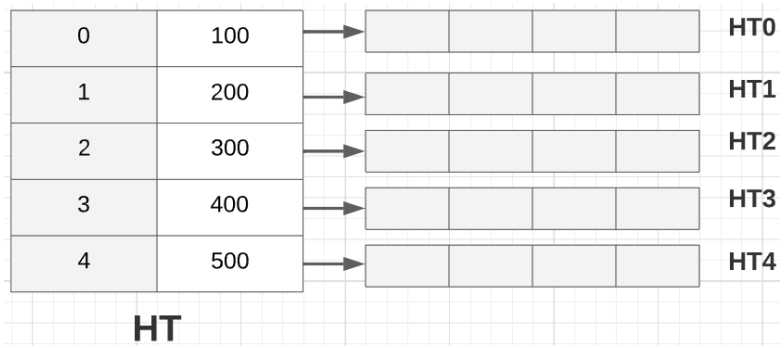
CLUSTERING IN DOUBLE HASHING

In Double hashing, we use two different hash functions instead of one. So, even if two keys start at the same hash address, then the paths they take need not be the same. Hence, there is **no redundancy and no clustering problem.**

Hence, the average time complexity is the **same as the best case** complexity which is $\boldsymbol{O(1)}$**.**

PERFECT HASHING (Not asked usually)

This is basically a nested hash table i.e. the hash table stores other hash tables. Each of the tables have their own hash functions which are the **best hash function** aka which cause minimum collision. In perfect hashing, there are minimum collisions and hence the time complexity is $\boldsymbol{O(1)}$ for all cases.



UNIVERSAL HASHING

This is a technique where we randomly select a hash function from a given bank of hash functions independent of the keys. This ensure very low number of collisions and is preferred.

LOAD FACTOR (AVERAGE)

Let us take a hashing situation where we have M slots in the DAT and we need to store N keys. Then, we get –

$$Load\ factor\ (\alpha) = Number\ of\ keys\ per\ slot = \frac{N}{M}$$

In open addressing, we can have a minimum of 0 keys and a maximum of 1 key per slot. On the other hand, for chaining we can keep infinite number of keys in 1 slot. Hence, we get –

$$0 \leq \alpha \leq 1 \qquad for\ Open\ Addressing$$

$$0 \leq \alpha \leq \infty \qquad for\ Chaining$$

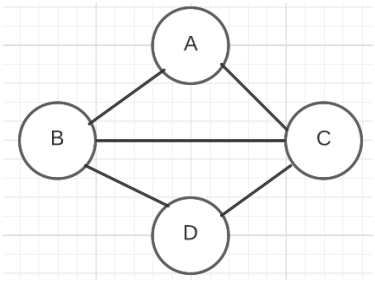
The expected number of probes (attempts) of an open addressing technique for –

$$Unsuccesful\ search = \frac{1}{1-\alpha}$$

$$Successful\ search = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

GRAPH BASICS

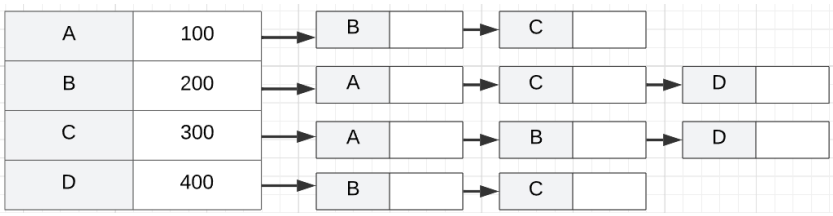
A graph is a combination of vertices (nodes) and edges. For example, let us take the graph below –



For this graph, we can write the **Adjacency Matrix** which is a **square matrix** that represents how many nodes are adjacent to each other –

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

In a computer system, the graph can either be stored as an Adjacency matrix or as an **Adjacency List** as shown below –



For a graph of V vertices and E edges, we have –

$$Space\ for\ Adjacency\ Matrix = V^2$$

$$Space\ for\ Adjacency\ List = V + E$$

The number of nodes adjacent to the given node is called **Degree.** So for the given graph, we have –

$$Deg(A) = Deg(D) = 2$$

$$Deg(B) = Deg(C) = 3$$

We also have –

$$Number\ of\ edges, E = \frac{Sum\ of\ degrees}{2} = \frac{2+2+3+3}{2} = \mathbf{5}$$

The sum of degrees is ALWAYS EVEN. There can be graphs with vertices but NO EDGES. Such graphs are called **Null graphs** and the degree of each vertex is **zero.** On the other hand, a graph in which each vertex is adjacent to each other vertex (adjacency matrix has all values as 1) is called a $\boldsymbol{NV}$ **complete graph** where N is the number of vertices. For a N-vertex complete graph, we have –

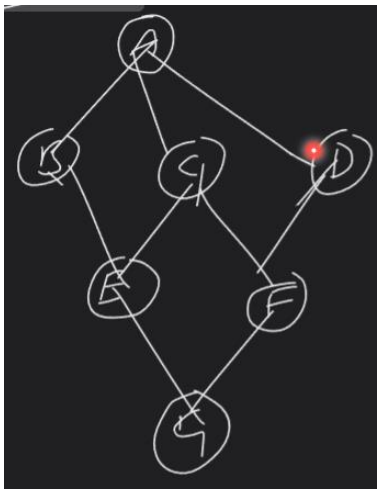
$$Number\ of\ Edges, E = \frac{N(N-1)}{2}$$

GRAPH TRAVERSAL

Graph traversal is the way to cover all the vertices. There are two ways to do so –

- Breadth First Traversal (BFT)
- Depth First Traversal (DFT)

Basically, traversal is the process of searching for ALL NODES of a graph. Let us take the example of the graph below –



In this case,

- If we traverse all nodes at a level and then move down, then it is called **BFT.** So the BFT traversal will be $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$
- On the other hand, if we go straight down the levels, then it is called **DFT.** So, the DFT traversal will be $A \rightarrow B \rightarrow E \rightarrow G$.

There are multiple ways to traverse under both BFT and GFT and there is no single right answer.

BFT ALGORITHM

To implement the BFT algorithm, we need two data structures –

- A visited array to note if a node has been visited before or not. This is to prevent any duplication.
- A node queue to print out the nodes as they are visited.

Both visited array and node queue are to be of size V.

```
#Initialization
visited = [0, 0, 0, 0, 0, 0, 0]
nodeQueue = []

#Now, we push the source node to the nodeQueue and mark that node as visited
#in the visited array.
nodeQueue.append('A')
visited = [1, 0, 0, 0, 0, 0, 0]

#While we still have nodes to be traversed.
while (nodeQueue):
```

```
    #Next, we check the row of the Adjacency Matrix to find the nodes that
    #are adjacent to the source node A. In our case, the adjacent nodes are
    #B, C, and D. Out of these, we consider only those nodes which are NOT
    #VISITED before. Once we have these, we can perform the following ops -
        #1. Dequeue and print top of queue.
        #2. Push the adjacent not visited nodes to the queue.
        #3. Change the visited info in the array
    print(nodeQueue[0])
    nodeQueue.dequeue()
    nodeQueue.append('B', 'C', 'D')
    visited = [1, 1, 1, 1, 0, 0, 0]

#This process is repeated till the queue is empty.
```

Here is how the algorithm will play out for the given graph –

nodeQueue	Visited	Adjacent nodes	PRINTED
A	1,0,0,0,0,0,0	B C D	-
B C D	1,1,1,1,0,0,0	A E	A
C D E	1,1,1,1,1,0,0	A E F	A B
D E F	1,1,1,1,1,1,0	A F	A B C
E F	1,1,1,1,1,1,0	B C G	A B C D
F G	1,1,1,1,1,1,1	C D G	A B C D E
G	1,1,1,1,1,1,1	E F	A B C D E F
Null	1,1,1,1,1,1,1	Null	A B C D E F G

Finally, we got the BFT as $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$ which is what was expected 😊.

Another thing to note is the method to get the adjacent nodes. We have two choices in this case –

- In an adjacency matrix, we will have to parse through the entire row to get the adjacent nodes. Hence, the complexity to get adjacent nodes becomes $O(V)$. Since we need to find the adjacent nodes for all the V nodes, **the BFT time complexity becomes $\boldsymbol{O(V^2)}$**
- In an adjacency list, we already know that the list of nodes for a particular index node are the adjacent nodes. So, there is no need to parse any list and we can get the adjacent nodes directly. Hence, the complexity to get the adjacent nodes becomes $O(1)$ and thus the **BFT complexity becomes $\boldsymbol{O(V+E)}$** as the list also takes care of edges.
- In a tree traversal, we have the same complexity as BFT. However, since there are no cycles in the tree which means that $E = V - 1$. Therefore, $V + E = 2V - 1$. Thus, for tree traversal, the complexity becomes $\boldsymbol{O(V)}$**.**

Additionally, we need to have an extra space for Queue and Visited array. Since they are the same size as V, the space complexity will be $\boldsymbol{O(V)}$**.** As we are traversing the graph level by level, BFT is also called **Level Order Traversal.**

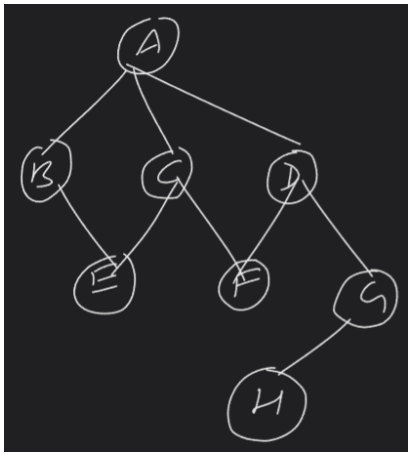
GRAPH vs TREE

Tree is a special type of graph which has the following major features –

- A tree will NEVER have cycling or loops.

- A tree is always a **connected graph** which refers to graphs in which one can find a path between any two nodes.
- To traverse a tree, we must always start from the **root node.**

Question



For the graph displayed above, find 4 unique BHT sequences that begin from the node H.

Answer

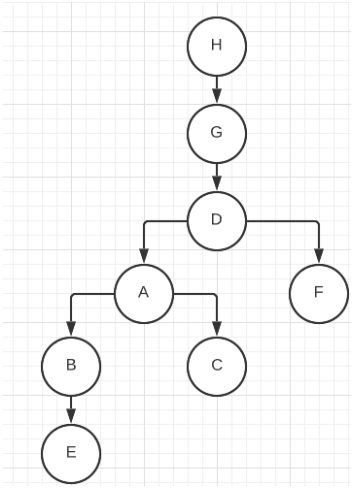
Using the algorithm mentioned above, we can get 4 BHTs as follows –

- HGDAFBCE
- HGDAFCBE
- HGDFACBE

In this case, there are only 3 unique sequences possible. Given the 1st BHT sequence, we can write –

- H is the source node.
- G is adjacent to H.
- D is adjacent to G.
- A and F are adjacent to D.
- B and C are adjacent to A.
- E is adjacent to B.

Using this info, we can draw a **BFT Tree** as shown below –



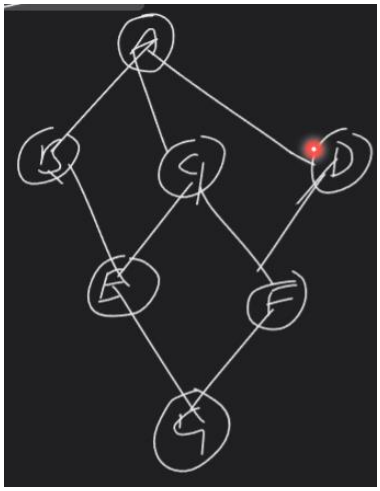
This is a sub-graph and is also called the **Spanning Tree.** The properties of the spanning tree are –

- It's a tree (DUH). So it has a root node, no cycles and is connected.
- It is a sub – graph of the given graph.
- It has $V - 1$ edges.

APPLICATIONS OF BFT

- BFT can be used to create the Spanning Tree with the same time complexity as BFT.
- BFT can be used to look for disjoint graphs. Disjoint graphs are the ones which have unreachable nodes. After BFT, if a node is unreachable then the visited value for that node will NOT be 1. Hence, with the same time complexity as BFT, we can figure out if the graph is disjoint or NOT connected. The Maximal sub – graph that is connected of a disjoint graph is called a **Connected component.**
- Similar to the second point, BFT can also be used to find the reachability of nodes in an Automate or Linked lists as these both are basically directed graphs.
- Let us say that we start the BFT and one of the adjacent nodes already has been visited by someone else. Then we can safely conclude that the graph is **cyclic.**
- When we create a BFT tree, we can find the shortest path between 2 nodes. This is called **Single Source Shortest Path.** This works for both directed and undirected graphs but not for weighted graphs. A general rule of thumb is as follows –
    - For unweighted graph, find SSSP using BFT.
    - For weighted graph with positive weights, find SSSP using Dijkstra's algorithm.
    - For weighted graph with negative weights as well, find SSSP using Bellman Fort's algorithm.
    - If we have weighted graph but all the weights are positive and **equal,** then we can still use BFT as the graph is effectively unweighted.
- Using BFT, we can verify if a graph is **bipartite** or not. A Bipartite graph is one where the nodes are divided into two disjoint sets and the edges are formed to connect nodes between the two sets and no edges exist within 1 set.

DFT ALGORITHM



Here, unlike BFT, we would be using a **Stack** instead of a queue to print the elements. This is because we don't print all the nodes of a level.

```
#As mentioned, we have 2 DS here - a visited array and a nodeStack stack.
visited = [0,0,0,0,0,0,0]
nodeStack = []

#Now, we define the DFT function. Let us assume we are starting with A.
    #1. We first set the visited value to 1.
    #2. We print out A
    #3. Then, we push the adjacent nodes on top of the stack.
def DFT(A):
    visited['A'] = 1
    print('A')

    adj = Adjacent(A) #Function to get adjacent nodes of Node A.
    for node in adj:
        nodeStack.push(node)

#Now, we take the nodes one by one from the stack. If a node has
#not been visited previously, we call the DFT function again for that node.
    for w in nodeStack:
        if (visited[w] == 0):
            DFT(w) #Recurssion
        else:
            nodeStack.pop()

#The program ends when the stack is empty.
```

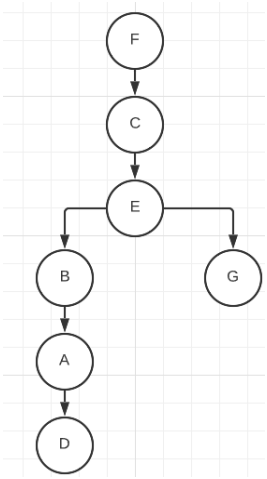
Let us follow this algorithm for the given graph –

Current node	Visited	nodeStack	PRINTED
A	1,0,0,0,0,0,0	B C D	A
B	1,1,0,0,0,0,0	A E C D	A B
E	1,1,0,0,1,0,0	B G C D	A B E
G	1,1,0,0,1,0,1	E F C D	A B E G
F	1,1,0,0,1,1,1	G C D	A B E G F
C	1,1,1,0,1,1,1	A E F D	A B E G F C
D	1,1,1,1,1,1,1	A F	A B E G F C D

Now, since all the nodes in the stack have been visited already, we are no longer calling DFT function and rather we would be popping those elements off. Now, we got one DFT sequence. We can get other such sequences as well –

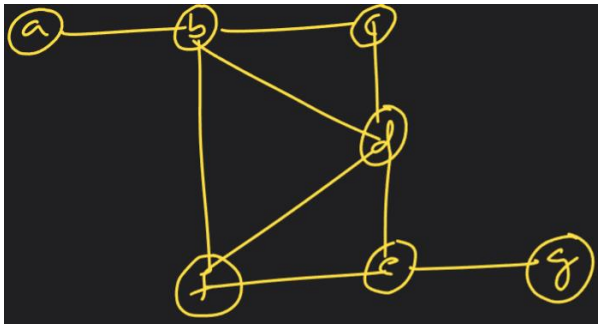
- A C F D G E B
- A B E C F G D
- B A D F C E G
- F C E G B A D

For the last sequence, let us make the DFT tree –



In this sequence, when we reached the node G, all its neighbors had been visited previously. Hence, we had to break out of the recursion and back track to node E. This is represented as a fork in the graph at node E. In this tree, we have **6 levels** and therefore, we need the **stack space of 6 elements.**

Question



For the above graph, 5 DFT sequences are given. Find the sequence which is **INCORRECT.**

1. A B C D E F G
2. E D F B C A G
3. C D F B A E G
4. E D C F B A G
5. G E D A B C F

Also, find the Maximum and Minimum stack size required for this graph.

Answer

In Option 4, once we reach node C, we have an option to travel to node B since it has not been visited previously. However, the sequence back-tracks to D and then goes on to F. Hence, Option 4 is **INCORRECT.**

The maximum stack size will occur for DFTs without back-tracking and minimum stack size will occur for DFTs with maximum back tracking. In our case, using trial and error (and some common sense), we can get –

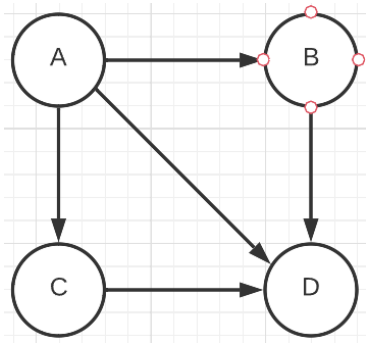
1. A B C D F E G – No back tracking. Maximum stack size 7.
2. D F E G B A C – Several back tracking. Minimum stack size 4.

APPLICATIONS OF DFT

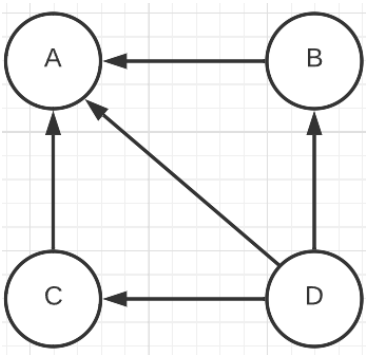
These are similar to the applications of BFT –

- To check if the graph is cyclic or not.
- To check if the graph is connected or not. We can also find the number of connected components.
- To find the spanning aka DFT tree.
- In BFT, we take all the adjacent nodes at a time which enables us to find the SSSP in an unweighted graph. However, DFT takes one adjacent node at a time and hence, will **NOT** give the SSSP for an unweighted graph.
- We can use DFT to get the **articulation points (cut vertices).** Articulation points are nodes that when deleted, will make the graph a disconnected graph. The check for articulation points can only be done by DFT and **not by BFT.** FYI, in a tree, every node except for the leaf nodes is an articulation point.
- A **directed** graph is said to be **strongly connected** if there is a path from any vertex to any other vertex. Given a directed graph, first apply DFT and then reverse the edge directions and apply DFT again. If in both cases, we are able to cover all nodes, then the directed graph is said to be strongly connected.

Let us assume the directed graph as shown below –



Applying DFT starting from Node A gives us - $ABDC$. Notice, this is a directed graph and hence there is no direct path from D to C. Instead, we need to back track to B and then to A and then get to C. Reversing the directions, we get –

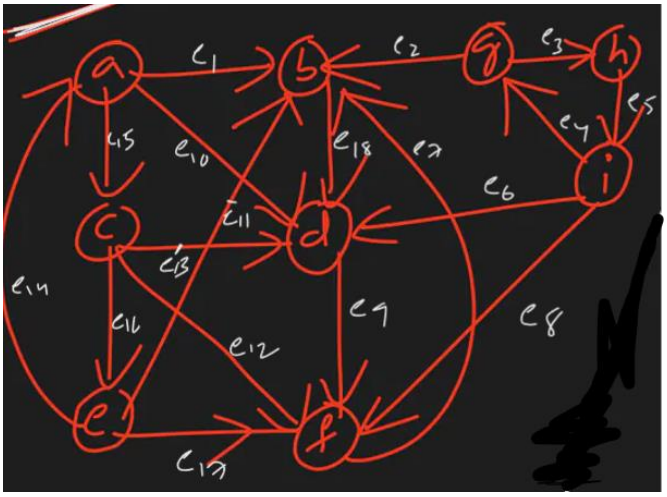


Let us again apply DFT to node A. Here, we can see that there are no adjacent nodes to A and there are no paths originating from A to any other node. Hence, the DFT sequence stops at A. Since we are unable to cover all the nodes, this graph is said to be **NOT Strongly Connected.**

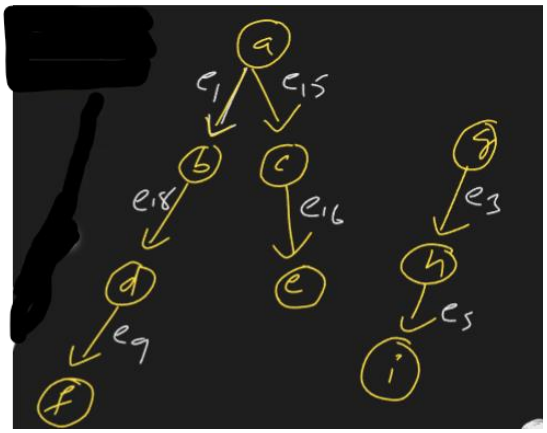
In case we replace the directed paths with undirected paths to make this an undirected graph, then the DFT will cover all nodes. This makes the graph **weakly connected.**

EDGE CLASSIFICATION (DFT + Directed Graph)

Let us take the direct graph as shown below –



Now for this graph, let us try performing DFT –



- We can see that not all edges are present in the DFT tree. In fact, out of 18 original edges, 11 edges are not present in the DFT tree. The edges present in the DFT tree are called **Tree edges.**
- We can see that Node A is the ancestor of Node E from the DFT tree. The edge e_{14} in the original graph connects the child to its ancestor which is **NOT present in the DFT**. These edges are called **Back edges.** Presence of Back edges indicates the presence of **cycles.**
- Similarly, an edge like e_{10} which goes from Node A to its descendent Node D and is **NOT a part of the DFT** is called a **Forward Edge.**
- Let us now look at edge e_{13}. It is connecting Node C to Node D and these nodes have no relation between them. These edges are called **Cross edges.**

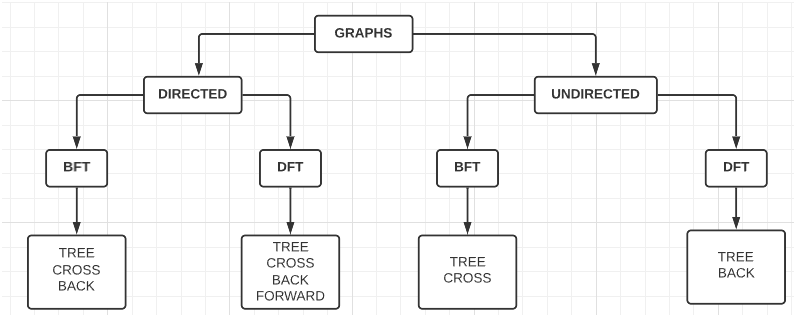
In short, we have –

TYPES OF EDGES	EDGE NUMBER
Tree	1, 3, 5, 9, 15, 16, 18
Backward	4, 7, 14
Forward	10
Cross	2, 6, 8, 11, 12, 13, 17

The above analysis was done for a Directed Graph using DFT. We can also perform the same edge classification for –

- Directed Graph using BFT
- Undirected Graph using DFT
- Undirected Graph using BFT

Since undirected graphs don't have any direction, both back and forward edges are the same. So, there is no need to separately calculate both. Also, in Directed graph, we used to check for Back edges to check for cycling. However, in an undirected graph, the **Cross edges indicate cycling.**



1-D ARRAYS

Array is a continuous memory storage and is a collection of **similar datatypes.** In an array, the elements have indices and hence, we can **randomly access** the elements. Since the array has continuous allocation of memory, we can calculate the address of an element N as follows –

$$Address\ of\ element\ N = Base\ Address + [(N - 1^{st}\ index) * element\ size]$$

So, we can get the address of each element in the array without using a loop making the time complexity $O(1)$. However, this formula can only be used for continuous data structures like arrays.

For example, let us take an array with the following details –

- Starting Index = 10
- Index of the element whose address is to be found = 100
- Size of elements = 2B (like int)
- Base address = 1000

Then, we can use the formula as follows =

$$Address\ of\ indedx\ 100 = 1000 + [(100 - 10) * 2] = \mathbf{1180}$$

NOTE – Suppose we start the indexing from 0, then we have –

$$Offset = N - 1^{st}\ index = N - 0 = N$$

Thus, indexing from 0 saves us a subtraction operation. This is the reason why languages start their indexing from 0.

2-D ARRAYS

These are basically matrices. Suppose we have a matrix as follows –

$$\begin{matrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{matrix}$$

Then, we can represent the matrix in either row major or column major order –

ADDRESS	1000	1002	1004	1006	1008	1010	1012	1014	1016
ROW MAJ	$a11$	$a12$	$a13$	$a21$	$a22$	$a23$	$a31$	$a32$	$a33$
COL MAJ	$a11$	$a21$	$a31$	$a12$	$a22$	$a32$	$a13$	$a23$	$a33$

We can use a formula to find the address of a specific element. Let us assume that the element is present in row i and column j. Take the base address to be BA and the size of each element to be S. Finally, the total order of the array is $m \times n$ and the starting indices for rows and columns are r and c respectively. In that case, we have –

For Row Major Addressing

$$Address\ of\ A_{ij} = BA + \{\{[(i - r) * n] + (j - c)\} * S\}$$

For Column Major Addressing

$$Address\ of\ A_{ij} = BA + \{\{[(j - c) * m] + (i - r)\} * S\}$$

In the above example, address of element a_{32} would be –

For Row Major Addressing

$$Address\ of\ A_{ij} = 1000 + \{\{[(3 - 1) * 3] + (2 - 1)\} * 2\} = 1014$$

For Column Major Addressing

$$Address\ of\ A_{ij} = 1000 + \{\{[(2 - 1) * 3] + (3 - 1)\} * S\} = 1010$$

This is the same values as seen from the table above. In both row major and column major addressing, the number of additions, multiplications and subtractions are the same and hence they give the same performance.

The C language is programmed by default to use the Row Major ordering. Hence, while declaring a 2-D array, even if we miss the row values, the program will not throw an error. But if we miss the column values, then it will throw an error as it won't know how many elements are in the rows.

```
int a[2][3] = {1, 2, 3, 4, 5, 6}        VALID
int a[][3] = {1, 2, 3, 4, 5, 6}         VALID
```

```
int a[2][] = {1, 2, 3, 4, 5, 6}
```
INVALID

In the second case, we don't need the row number as we are using Row Major addressing. Hence, the system will assign the 3 elements for the 3 columns and then move on to the next row.

3 – D ARRAYS

1-D arrays are a collection of elements. 2-D arrays are a collection of 1-D arrays. Similarly, 3-D arrays are a collection of 2-D arrays. For 3 – D array, we can extend the previous expression to get the address of an element in 2-D formula.

LOWER TRIANGULAR MATRIX

The lower part (including the principle diagonal) is non – zero but the upper part is all zeroes. For a $n \times n$ square matrix, the number of non – zero elements are $\frac{n(n+1)}{2}$ which is the sum of n natural numbers. This is because 1st row will have 1 non – zero element, the 2nd row will have 2 non – zero elements and so on.

Another way to represent the lower triangular matrix is –

$$A[i][j] = 0 \quad if\ i < j$$

This way, we know which elements will be 0 and which elements will not be 0. So, when storing the lower triangular matrix, the system stores only the non-zero elements to save space. Hence, let us take the case of a matrix as follows –

$$\begin{matrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{matrix}$$

Since we will be storing only the non – zero elements, the matrix will be stored in the memory as follows –

ADDRESS	1000	1002	1004	1006	1008	1010
ROW MAJ	$a11$	$a21$	$a22$	$a31$	$a32$	$a33$
COL MAJ	$a11$	$a21$	$a31$	$a22$	$a32$	$a33$

Even though this saves space, one must be careful while retrieving the address for the element as each row has unequal number of elements.

Question

Let us assume an array as shown below –

$$A[75 \ldots 180\,;75 \ldots 180]$$

The base address is 1000 and the size of each element is 10B. The array represents a lower triangular matrix and is stored using Row major ordering. Find the address of element at indices $A[170][150]$.

Answer

First, we find the number of non – zero elements that complete row till row 170 –

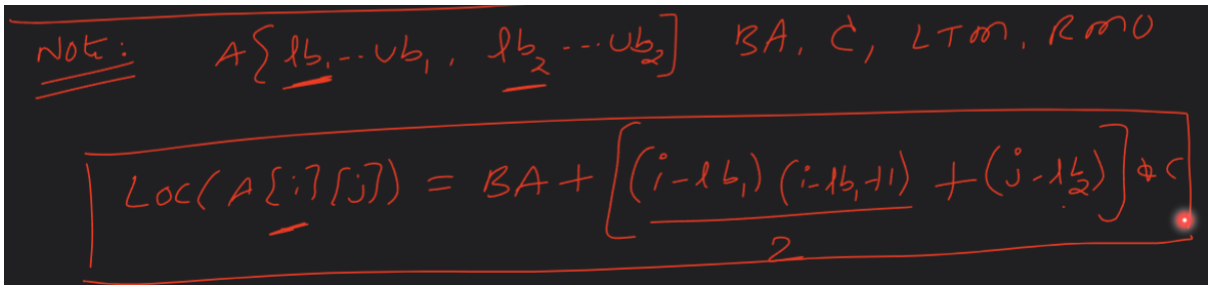
$$No\ of\ elements = \frac{(170-75)(170-75+1)}{2} = 4560$$

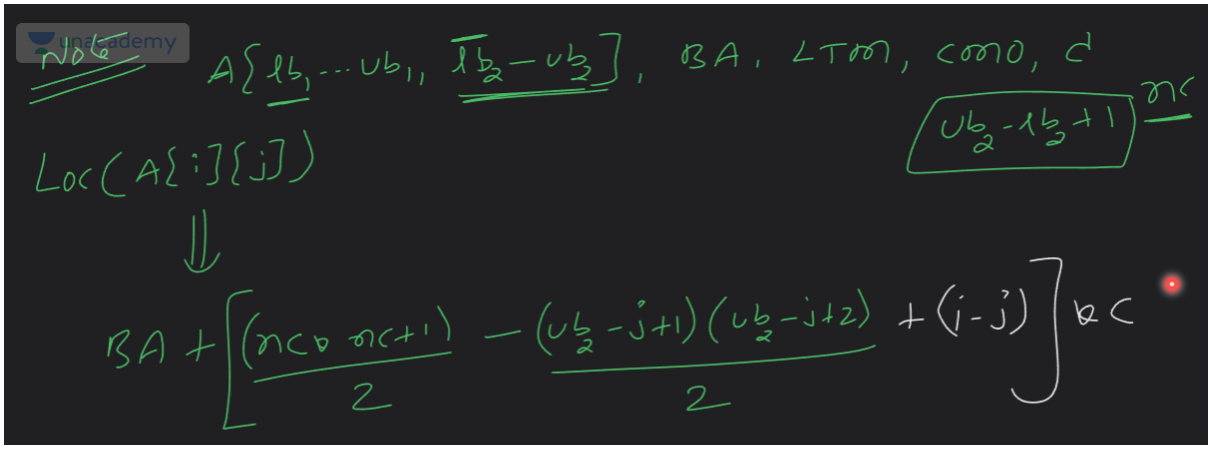
The addressing for the first 4560 elements in the array will be –

$$Address = (4559 * 10) + 1000 = 46590$$

Now, in the 170th row, we need to navigate till column 150. Hence, the address becomes –

$$Loc(A[170][150]) = 46590 + [(150-75+1) * 10] = \mathbf{47350}$$





The above formulae can be done for an **Upper triangular matrix** but we just need to replace the Row Major and Column Major approaches.

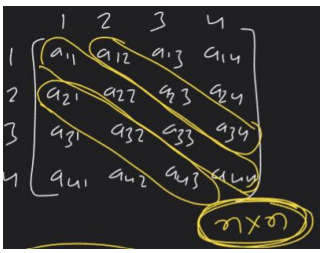
TRI – DIAGONAL MATRIX

The non – zero elements in the matrix are present in the Principle diagonal and the diagonals above and below it. The rest of the elements are zero. For a square matrix of order $n \times n$, we have –

$$Number\ of\ elements\ in\ Principle\ Diagonal = n$$

$$Number\ of\ elements\ in\ the\ diagonal\ below\ the\ Principle\ diagonal = n-1$$

Thus, we get the total number of non – zero elements to be $3n-2$.



One interesting thing to note is that apart from the first and the last row, every row has 3 elements. The first and the last rows have 2 elements. In the same way, the first and last columns have 2 elements and each of the other columns have 3 elements. For this case, we can write –

$$Loc(A[i][j]) = BA + \{[3(i - lb_1) - 1 + j - (i - 1)] * C\}$$

For column major accessing, we can write –

$$Loc(A[i][j]) = BA + \{[3(j - i) - 1 + i - (j - 1)] * C\}$$

But this is a cumbersome process. Instead, we can store the 3 diagonal elements directly instead of storing row-wise or column-wise as shown below –

- n elements of Principle diagonal can be stored. For these elements, $i = j$
- $n - 1$ elements of the lower diagonal can be stored. For these elements, $i - j = 1$
- $n - 1$ elements of the upper diagonal can be stored. For these elements, $j - i = 1$

Usually, we prefer to store in the following order in the memory –

1. Principle diagonal elements
2. Lower diagonal elements
3. Upper diagonal elements

PROGRAMMING

We will be focusing on Programming in C. To understand the concepts ahead, we need to understand the operator precedence –

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

SCOPE OF A VARIABLE

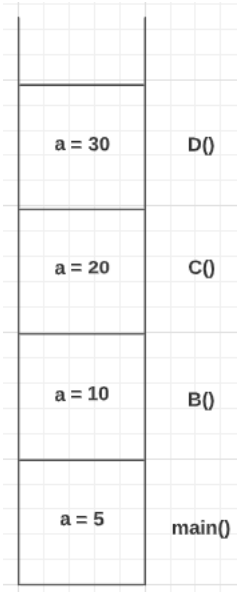
To understand scope of variables, let us assume a program as follows –

```c
#include <stdio.h>
int a = 10;

int main()
{
    int a = 5;
    B();
}

int B()
{
    int a = 10;
    C();
}
```

```c
int C()
{
    int a = 20;
    D()
}

int D()
{
    int a = 30;
    printf("%d", a);
}
```

In this case, we have a global declaration for variable a and 4 local declarations for variable a as well. When declaring a global variable, 2B of memory (coz int) will be allocated in the **static area** during **compilation by compiler.**

Now, we call the main() function. A note to remember is that a function call gets pushed in the **stack memory.** So, we get the following stack information –



So, now when we print the value of a, then the value at the top of the stack will be printed aka **30.**

NOTE – If we declare a variable, then we are basically allocating a memory location for that variable. However, if we don't initialize any value to this variable, the variable will store the previous program value at that memory location, aka **garbage value.**

SCOPE PROBLEM

In the above program, let us assume that the D() function is shown as below –

```
int D()
{
    printf("%d", a);
}
```

Basically, we are no longer initializing the value of a but we are still printing a. Since this variable was not declared in the scope, we call this a **Scope Problem** and a is also called a **Free variable.** There are two ways to solve the scope problem –

1. The Scope problem can be solved by the **compiler** itself. In this case, the compiler simply pulls the value of a which was declared in the global scope. Hence, if the scope problem is solved by compiler, the value of a printed will be **10.** This is called **Static Scoping.**
2. On the other hand, the scope problem can be solved by **processor** as well. In this case, the processor goes to the calling of D() and enters function C(). Basically, it backtracks. In C(), we can see that a has been declared as 20. So, in this case, the value of a printed will be **20.** If C() also didn't have any declaration, it would have back tracked further to B() and so on till it gets a value of a. This is called **Dynamic Scoping.**

Dynamic Scoping has more overhead and is slower when compared to Static scoping. Hence, most of the languages use Static scoping and hence **assume Static scoping until and unless mentioned otherwise.**

Question

Find the output of the following program –

```
int a;
int main()
{
    int i = 10;
    f();
}

int f()
{
    int i = 20;
    g();
}

int g()
{
    printf("%d", i);
}
```

Answer

We can see that there is a scoping problem and that i is a free variable. In this case, if we use Static scoping then we will be receiving an error stating that variable i has not been declared as there is no global scope declaration for i. On the other hand, if we use Dynamic scoping, then we will receive output as **20** after back tracking to function $f()$.

Suppose, we add `int i;` in the global scope and ran the program again, then we will get garbage value for variable i. By default, the garbage value of i is **0** as static area is a small memory space and we can't afford to have large values. So, we will actually be printing **0.**

Question

Find the output of the following program –

```
int i = 10;
int j = 20;

int main()
{
    int i = 5;
    printf("P1 - %d %d\n", i, j);
    A();
    printf("P2 - %d %d\n", i, j);
    B();
}
```

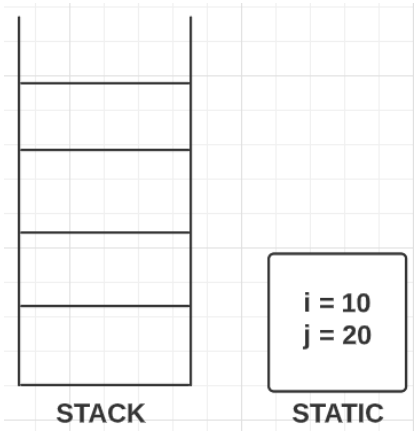
```
int A()
{
    printf("P3 - %d %d\n", i, j);
    i = 7;
    j = 8;
    B(i);
    i = 77;
    j = 88;
}

int B(int i)
{
    printf("P4 - %d %d\n", i, j);
    i = 3;
    j = 4;
    D(i);
    printf("P5 - %d %d\n", i, j);
}
```

```
int D(int j)
{
    printf("P6 - %d %d\n", i, j);
    i = 30;
    j = 90;
}
```

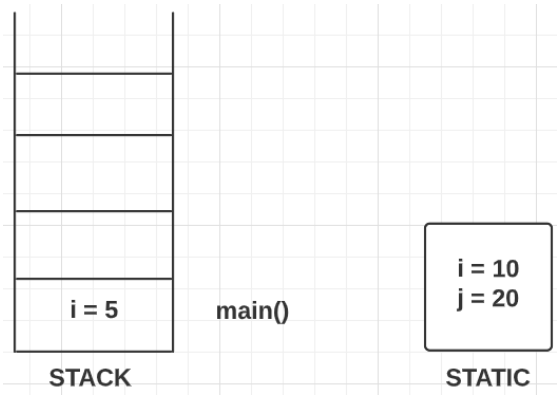
Answer

This is quite an interesting question. So, let us deal with this in **Static Scoping method.**

First we define the two address spaces – Static (for global variables) and Stack (for functions and local vars).



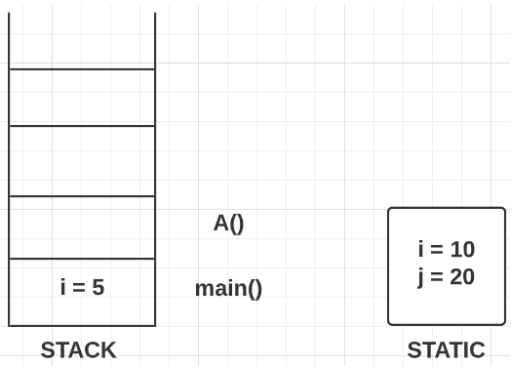
Now, we enter the **main()** function and then we initialize $i = 5$ in the local scope –



Next line is the print statement –

$$P1 - 5\ 20$$

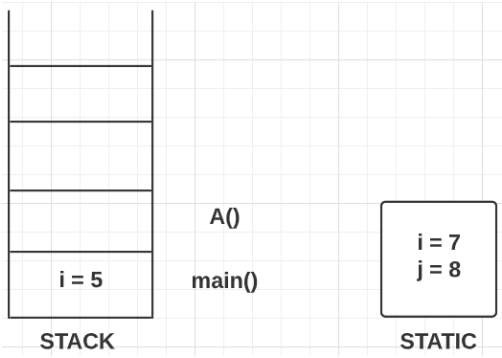
Now, we are calling function **A()** and that gets pushed onto the stack.



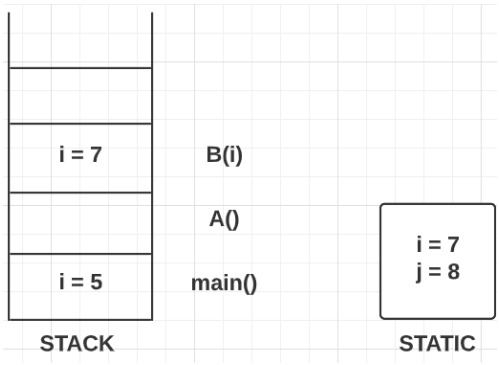
Since A has no variable declaration, we will be printing the global variables (coz Static scoping) –

$$P3 - 10\ 20$$

Now, we are updating $i = 7$ and $j = 8$. Since there are no local vars declared under A(), these values are updated globally.



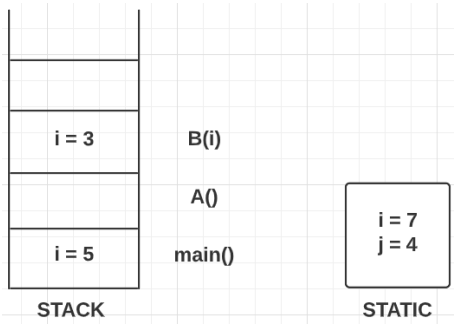
Now, we are calling function **B(i).** One thing to note here is that we are passing the value of i to the function. This is equivalent to declaring and initializing a variable in the local scope itself.



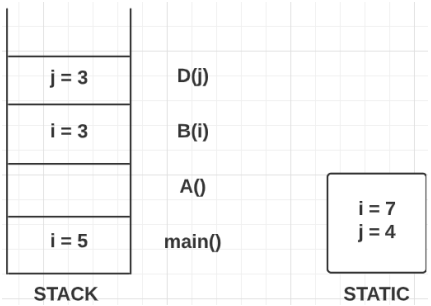
So, if we print now, we will get –

$$P4 - 7\ 8$$

Next, we are updating $i = 3$ and $j = 4$. The value of j will be globally updated in the static space but as far as i is concerned, the value will be updated locally.



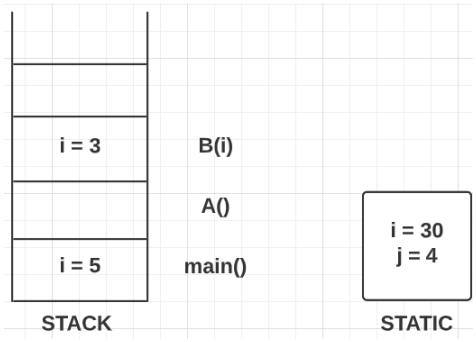
Next, we are calling function $\boldsymbol{D(j)}$**.** This function is interesting as it is taking the value of i as the argument but as per the function definition, the function is storing this value in the local scope as j. Hence, we are effectively declaring and initializing $j = 3$



Now, we are printing –

$$P6 - 7\ 3$$

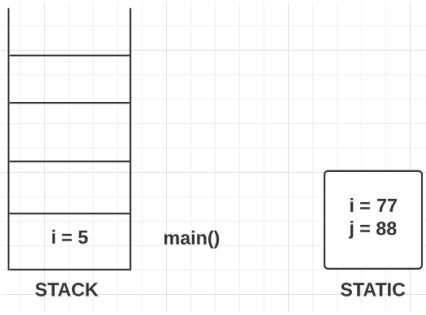
Post the printing, we are updating the value of $i = 30$ globally and value of $j = 90$ locally. However, we break out of D function and hence, that memory is deleted and made available.



Now, we are back to function B and if we print now, the local value of i and the global value of j will be printed –

$$P5 - 3\ 4$$

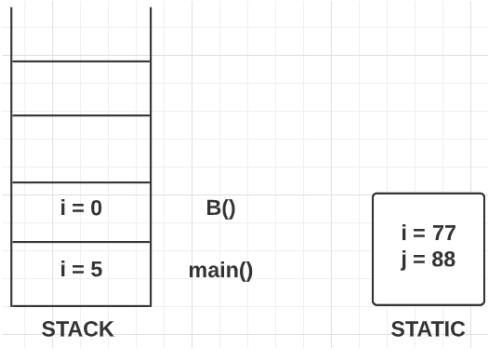
After this, we are finishing B and breaking out back to A. Thus, the memory allocated to B is now made available. At the same time, the values of i and j are updated globally as A has no local var declaration. However, right after that we break out of A as well and go back to $main()$.



Now printing, we get the local value of i and the global value of j –

$$P2 - 5\ 88$$

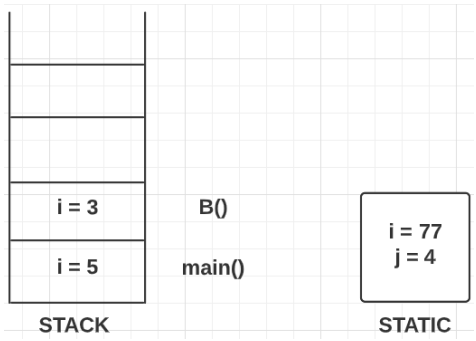
We are almost done here, but we still have a one line of code left. That is a function call for B again! Interestingly, this time there is no parameter provided while calling B. So now, stack memory will be allocated to B but the value of i in the function will be a **garbage value** which we can assume to be anything. Let us assume it to be **0** for the time being.



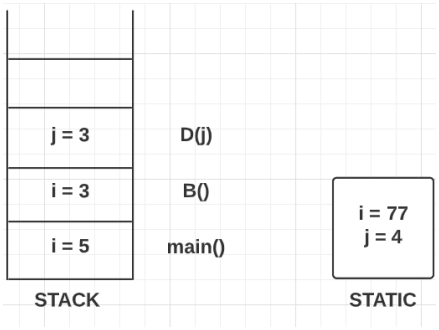
Now, printing we get –

$$P4 - 0\ 88$$

After this, we are updating the values of $i = 3$ and $j = 4$ locally and globally respectively.



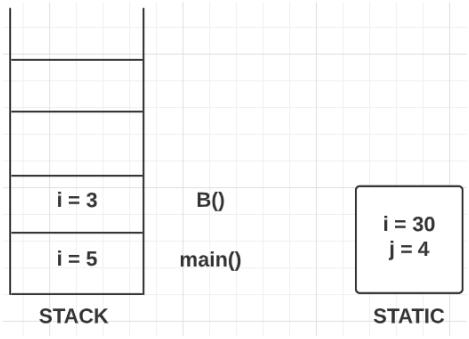
Next, we will be calling the function D by passing the parameter value of **3** which will be stored locally as $j = 3$.



Printing, we get –

$$P6 - 77\ 3$$

Finally, we set the value of $i = 30$ globally and set the value of $j = 90$ locally. However, we are breaking out of D so all this memory will be made available.



Printing, we get –

$$P5 - 3\ 4$$

Finally, we are done and we break out back into A and then back into $main()$ and finally out. So, the final output is as follows –

$$P1 - 5\ 20$$

$$P3 - 10\ 20$$

$$P4 - 7\ 8$$

$$P6 - 7\ 3$$

$$P5 - 3\ 4$$

$$P2 - 5\ 88$$

$$P6 - 77\ 3$$

$$P5 - 3\ 4$$

If we are using Dynamic scoping, then we can write the output as –

$$P1 - 5\ 20$$

$$P3 - 5\ 20$$

$$P4 - 7\ 8$$

$$P6 - 3\ 3$$

$$P5 - 30\ 4$$

$$P2 - 77\ 88$$

$$P4 - 77\ 88$$

$$P6 - 3\ 3$$

$$P5 - 30\ 4$$

INCREMENT

There are 2 types of increment operators in C –

- Pre-increment
- Post-increment

In **Pre-increment,** the expression is written as –

$$y = ++x$$

This expression states that first we increment x and then assign it to y. Thus, we can expand the pre-incrementation as follows –

$$x = x + 1$$

$$y = x$$

In **Post-increment,** the expression is written as –

$$y = x++$$

This expression states that first we assign the value of x to y and then increment the value of x. Thus, we can expand the pre-incrementation as follows –

$$y = x$$

$$x = x + 1$$

STATIC VARIABLE

A local variable can be declared with the **static** keyword such that it is stored in the **Static** area in the memory. That means, the value will be stored in the same memory vicinity as the global variables. Let us take an example to understand this –

```
int main()
{
    int n = 5;
    printf("P1 - %d\n", f(n));
    printf("P2 - %d\n", f(n));
}
```

```
int f(int n)
{
    int m = 20;
    m = m + n;
    printf("P3 - %d\n", m);
    return(m);
}
```

In this case, we have a regular program. Here, the variable of interest if m. Since m is a local variable, it is stored in the **stack** area and hence, whenever the control breaks out of $f(n)$, the value of m will be erased and it will be re-assigned when the function gets called again. Hence, the output in this case will be –

$$P3 - 25$$

$$P1 - 25$$

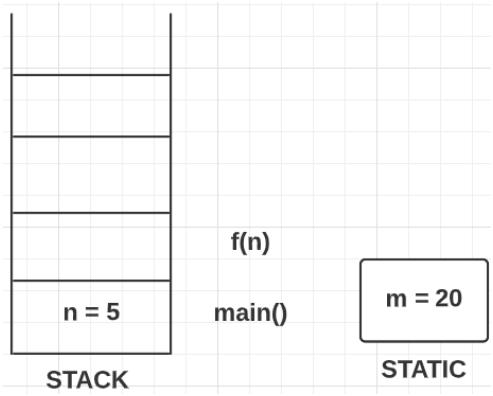
$$P3 - 25$$

$$P2 - 25$$

Now, we change the function $f(n)$ as follows –

```
int f(int n)
{
    static int m = 20;
    m = m + n;
    printf("P3 - %d\n", m);
    return(m);
}
```

In this case, the local variable m is stored in the **static** area (at compile time by compiler) and is no longer stored in the stack area (at run time by processor). Now, the first time we call $f(n)$, the stack and static area look like this –

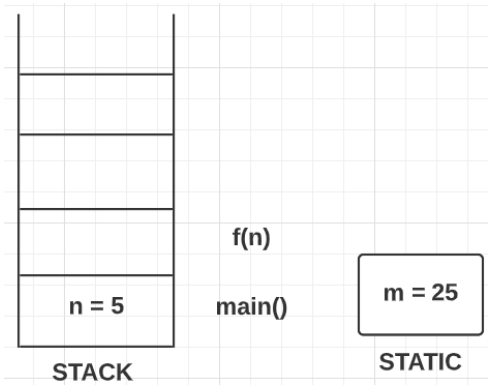


In this case, the first iteration will print the following –

$$P3 - 25$$

$$P1 - 25$$

Now, for the second iteration (second time $f(n)$ is called in $main()$), the memory allocation looks something like this –

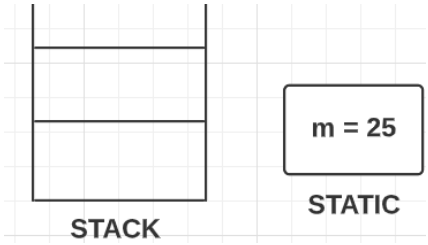


After the first iteration, the value of m was updated to 25 and since it is declared with the **static** keyword, the value is updated in the **static memory space** similar to global variables update process. Now, the following lines will be printed –

$$P3 - 30$$

$$P2 - 30$$

At the end of the execution, we will have –



In general, we have the following points –

- For a static variable, the memory will be allocated during **compile time.** Since a program is compiled only once, the memory allocation will also occur **once.**

- Since static memory space has small size, the garbage value stored is usually **0.** Hence, a static variable declared without initialization will be stored with **0** value.

Question

Find the output of the following program.

```
int main()
{
    static int n = 5;
    if (n) {
        printf("%d", n--);
        main();
    }
}
```

Answer

As we are using **static** keyword, the value of n will be stored in the static space. In 1st iteration, the if condition passes and therefore, we are printing 5. However, we also have a post-decrement operator. So the value of n will be set to 4. After that, the $main()$ function is called again aka **recursion.** IN this case however, the value of n will remain 4 and will not be re-assigned as it is static. Hence, the value printed will be 4. Similarly, this will go on till $n = 0$. Hence, the output will be –

$$Output = 5\ 4\ 3\ 2\ 1$$

Question

In the previous program, remove the **STATIC** keyword. What is the output now?

Answer

In this case, everything remains the same expect for the fact that the value of n will be stored now in the stack space. Hence, with every recursion, the value of n will be re-initialized to 5. Hence, it will produce an **infinite recursion** and will continuously print 5.

Question

Find the output of the following program.

```
int main()
{
    static int n = 5;
    if (n) {
        main();
        printf("%d", n--);
    }
}
```

Answer

In this program, there is one major bug. The $main()$ recursion is occurring before the decrement operator. Hence, whether we use **static** or not, there will be no change in the value of n and hence, the program will keep on running.

As we keep calling $main()$ function, the slot will be pushed to the top of the stack over and over infinitely. At some point, irrespective of static is used or not, the program will display **Stack Overflow** error. The Stack overflow error is a **run – time/logical error.**

Question

Find the output of the following program.

```
int main()
{
    static int n = 5;
    if (n--) {
        main();
        printf("%d", n);
    }
}
```

Answer

For the above program, we can go iteration by iteration as shown below –

```
n = 5
if (5) {
    n = 4 #Because of post decrement
    #main called again
    if (4) {
        n = 3
        if (3) {
            n = 2
            if (2) {
                n = 1
                if (1) {
                    n = 0
                    if (0) {
                        #Here the function fails. But remember, the decrement operator will still work.
                    }
                    n = -1
                    print(-1)
                }
                print(-1)
            }
            print(-1)
        }
        print(-1)
    }
    print(-1)
}
```

Hence, the program will print $-\mathbf{1}$ a total of 5 times.

Question

Find the output of the following program.

```c
int main()
{
    static int n = 5;
    if (--n) {
        main();
        printf("%d", n);
    }
}
```

Answer

Again, we can go iteration by iteration and we get the output.

```
n = 5
#Iteration1
n = 4
if (4) {
    #Iteration2
    n = 3
    if (3) {
        #Iteration3
        n = 2
        if (2) {
            #Iteration4
            n = 1
            if (1) {
                #Iteration5
                n = 0
                if (0) {
                    #FAILS
                }
                print(0)
            }
            print(0)
        }
        print(0)
    }
    print(0)
}
```

Hence, the output is 0000.

Question

Find the output of the following program.

```c
int f(int n) {
    static int r = 10;
    if (n <= 0) {
        return(7);
    }
    if (n > 2) {
        return(f(n-1)+r);
    }
    else {
        r = 17;
        return(n+f(n-1));
    }
}

int main() {
    int n = 7;
    printf("%d", f(n));
}
```

Answer

Follow the same procedure we have been following up until now. Here is a list of the intermediate outputs –

$$f(0) = 7 \quad and \quad r = 17$$

$$f(1) = 1 + f(0) = 8$$

$$f(2) = 2 + f(1) = 10$$

$$f(3) = f(2) + r = 27$$

$$f(4) = f(3) + r = 44$$

$$f(5) = f(4) + r = 61$$

$$f(6) = f(5) + r = 78$$

$$f(7) = f(6) + r = \mathbf{95} = \boldsymbol{Output}$$

Question

Find the output of the given code snippet.

```
void main ()
{
    int x = 5;
    int y = 10;
    int i;
    for (i=1 ; i<=2 ; i++) {
        y += f(x) + g(x);
        printf("%d\n", y);
    }
}
```

```
int f(int x) {
    int y;
    y = g(x);
    return(x + y);
}

int g(int x) {
    static int y = 20;
    y += 3;
    return (x + y);
}
```

Answer

Till now we have done so many problems that I didn't bother writing this one down. It is fairly simple. I got it on my first try and you will get it too future Bow.

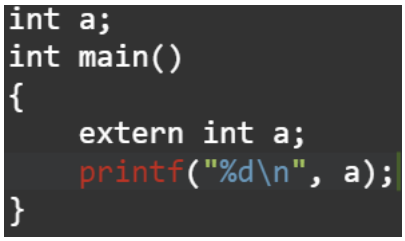
$$Output = 74\ 150$$

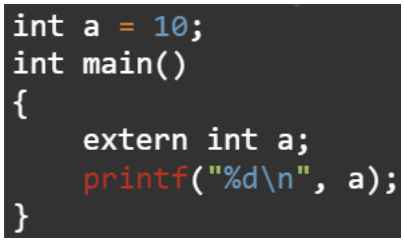
EXTERN VARIABLE

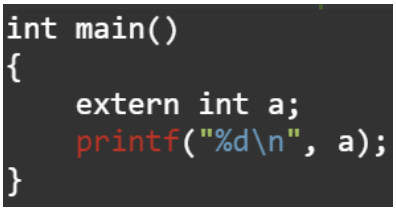
Before we proceed, let us list out the various key words –

DECLARATION	MEMORY LOCATION
int a	Stack for Local. Static for Global
auto int a	Stack. Same as Local declaration
register int a	Stack. Same as Local declaration. Stores in register to make the program faster.
static int a	Static

If a local variable is declared with **extern** keyword, it is saying that there is no need to create any memory and instead it would like to use the memory present in the global scope. Let us take the code snippets to get better clarify –





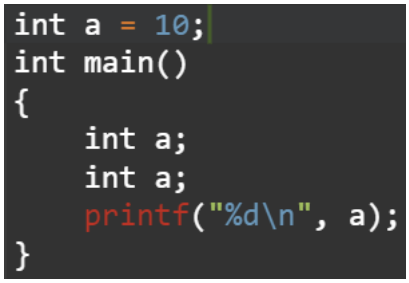


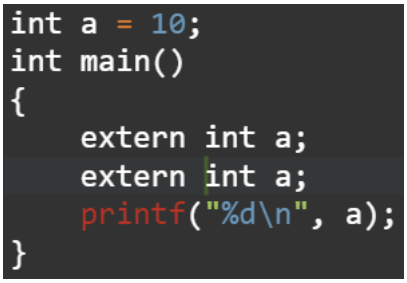
The outputs of the following codes will be as follows –

- 0. This is because the local variable will use the memory allocated globally and global variable a is stored with garbage value 0.
- Similar to the first one, this code will output $a = 10$
- Since there are no global a variable, the extern line will return an error stating a is not initialized.

NOTE – If we declare a global variable using extern and initialize with some value, then it will allocate memory for the variable and work as expected. However, if we declare a local variable using extern and initialize with some value, then the code will throw error.

Let us take the case of 2 code snippets below –





In the first code snippet, there are 2 local variables with the same name. This means that there are 2 different address locations with the same variable. So, if we try to execute the print statement, the program doesn't know which address to pull the value from and hence, we **face an error.**

On the other hand, in the second code snippet we get 2 local variables that are declared using extern. In this case, no memory is allocated for the variables as they want to use the space provided for a on a global space. Hence, when printed there is no address ambiguity and hence, **we don't face an error** and the value printed is **10.**

This is the biggest advantage of **extern** where it enables the code to declare the same named variables without throwing any errors.

Question

Find the output of the following program.

```c
int main()
{
    int i = -1;
    int j = -1;
    int k = -1;
    int l = 2;
    int m;
    m = ((i++ && j++ && k++) || (l++));
    printf("%d%d%d%d%d", i, j, k, l, m);
}
```

Answer

This is a fairly straight forward program, but there is a small catch down the line. Since $i = j = k = -1$, then we have –

$$i++ \;\&\&\; j++ \;\&\&\; k++ = 1$$

And then the value of i, j and k will be updated to 0. Now here is the interesting part. Since the first bracket returns 1, the value of the OR condition is already 1 and hence, **there is no need to check the condition $l++$.** This means that $l++$ doesn't execute (as it is not needed) and hence the **value of l remains 2.** Hence, the final output will be –

$$Output = 00021$$

Question

Take the program in the question above but this time i is initialized to 0 instead of -1. Find the output now.

Answer

Now, we know that $i = 0$. That means we already know that the AND statements will return false and there is no need to check for the value of $j++$ or $k++$. Therefore, we directly go to $l++$ to see if it will change the OR condition to true or not. Hence, the final output will be –

$$0 - 1 - 131$$

Question

Find the output of the following program.

```c
int main()
{
    int i;
    for (i=1 ; i<=30 ; i++) {
        switch (i)
        {
            case 1: i += 5;
            case 2: i += 3;
            case 3: i += 7;
            default:
                i += 2;
                break;
        }

        printf("%d\n", i);
    }
}
```

Answer

This is also a simple program, but one thing to note is that there are **no break** statements for each case. That means, the program will run for all the cases (including the default). Let's see the operation –

First Iteration

The value of $i = 1$. Hence, it satisfied for Case 1. However, since there are no break statements the rest of the case statements from that point onwards will be executed. So,

$$i = 1 + 5 + 3 + 7 + 2 = \mathbf{18}$$

That will be printed by the first iteration. After the switch case, the value of i will be increments as we are still inside for loop.

Second Iteration

The value of $i = 19$. So, none of the case statements are satisfied. Hence, we directly execute the default case –

$$i = 19 + 2 = \mathbf{21}$$

Then, i gets incremented by the for loop to 22.

The process keeps on executing till the for loop fails and the final output becomes –

$$Output = 18\ 21\ 24\ 27\ 30$$

NOTE – Here are a few things about the switch – case statements –

- After every case, break statements are required to prevent the switch case from executing every other case.
- The default statement can be added at any point in the switch case and need not be added in the end. However, it will fire only if all the other cases have failed.
- Since default is always the last to fire in the cases, there is no need for break statement after default.

Question

Take the same program as above, but here assume that there are break statements added after all switch cases. Find the output.

Answer

In this case, the output will be as follows –

$$Output = 6\ 9\ 12\ 15\ 18\ 21\ 24\ 27\ 30$$

Question

Find the output of the following program.

```
int main()
{
    int n = 13;
    int c = 0;
    while (n > 0)
    {
        c += n & 1;
        n >>= 1;
    }
    printf("%d\n", c);
}
```

Answer

In this program, the following operations are occurring in the loop –

- n is BITWISE ANDed with 1. This is same as extracting the LSB. The value can be either 0 or 1.
- The value of the BITWISE AND is added to c.
- Then we right shift n by 1.

Here is the flow of operations –

n	$n \,\&\, 1$	c	$n \gg= 1$
1101	0001	0001	0110
0110	0000	0001	0011
0011	0001	0010	0001
0001	0001	0011	0000

So the output is $\mathbf{3}$. One interesting thing to note is that we can write the program flow as follows –

- Get the LSB bit of n
- If n is 1, increment counter c. Else, don't change the counter.
- Right shift n so that the 2nd bit from right becomes the LSB.

In short, this example was a **program to find the number of 1's in a binary number.**

IMPLICIT TYPECASTING

We know that by default C allocated 2B for int data and 1B for character data. However, let us assume that there is a code as follows –

```
int a = 'c';
```

In this case, we are storing a character in an integer variable. This is not an error and in fact, C compiler will store character c in 2B even though it is a char to fit the int requirements. This is called **implicit typecasting** as it is done by the compiler implicitly. If we print a, then the ASCII value will be printed which is 99.

EXPLICIT TYPECASTING

Suppose we have the following code snippet –

```
float n = 10.5;
int a = n;
```

In this case, we are trying to fit a float value into an int variable. Since there is truncation involved, this can't be done by the compiler as it will result in loss of data. So, the compiler will throw an error. However, the user has the liberty to store this as an int variable as shown below –

```
int a = (int)n;
```

In this case, the user is changing the float variable to int variable and this is called **Explicit Typecasting.** In the case of Explicit Typecasting, there is a loss of data.

POINTERS (V V IMP)

Pointers are variables that store the address information instead of storing the value. It is also called **pointer variable or address variable.** Since the pointers store the address of data, then they should also have enough space to store the address information. Hence, even though an int is stored in 2B of memory, the address may need more space to be stored. In the question, the address size will be mentioned. For the notes ahead, we will be assuming **8B** address size until and unless mentioned otherwise.

Now, let us assume an integer variable a declared as follows –

```
int a = 10;
```

Suppose, we need to create a pointer variable b to store the address of a, then we can write the command as –

```
int *b = &a;
```

The above statement says that a **pointer variable b** is being created which is initialized with the **address of variable a.** Suppose, we declared the variable like this instead –

```
int c = &a;
```

In this case, the compiler will throw an error. This is because without the * declaration, c is a regular integer and has been allocated 2B of memory. However, $\&a$ returns the address of a which will be of 8B size. Hence, the compiler will throw an error for the same.

NOTE – When we write $\&a$, we are extracting the address of variable a. When we write $*b$, then we are trying to get the data at the address whose value is present in the variable b. To understand this, let us have a code snippet and output as shown below –

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int *b = &a;
    printf("%d\n", a); //Immediate Addressing
    printf("%d\n", b);
    printf("%d\n", *b); //Direct Addressing
}
```

```
10
2060022476
10
```

So as we can see, Line 6 and Line 8 both print **10** as the output. Line 6 is immediate addressing as it directly pulls the value of a. Line 8 is Direct Addressing as it goes to b, gets the address and since * is present, it gets the value at that address hence getting 10.

Let us assume that the there is a variable b which is a pointer. After that, we declare another pointer variable c to point to the address of b. This is called a **double pointer** and can be declared as shown below –

```
int **c = &b
```

```c
int main()
{
    int a = 10;
    int *b = &a;
    int **c = &b;
    printf("%d\n", a); //Immediate Addressing
    printf("%d\n", b);
    printf("%d\n", *b); //Direct Addressing
    printf("%d\n", c);
    printf("%d\n", *c);
    printf("%d\n", **c); //Indirect Addressing
}
```

```
10
-1312644428
10
-1312644424
-1312644428
10
```

NOTE – In the above code snippets, we can see that addresses are coming in as negative. That is not possible. The reason we are getting negative because we are printing the address as a regular integer **(%d).** Instead, we usually print the address either as **Unsigned Int (%u)** or as a **Pointer (%p).**

```c
int main()
{
    int a = 10;
    int *b = &a;
    int **c = &b;
    printf("%d\n", a); //Immediate Addressing
    printf("%p\n", b);
    printf("%d\n", *b); //Direct Addressing
    printf("%p\n", c);
    printf("%p\n", *c);
    printf("%d\n", **c); //Indirect Addressing
}
```

```
10
0x7fff3064e3a4
10
0x7fff3064e3a8
0x7fff3064e3a4
10
```

Question

Find the output of the following code snippet.

```
int a = 10;
int *b = &a;
int **c = &b;
int S1 = a**b;
int S2 = a***c;
int S3 = a*b;
int S4 = a**c;
```

Answer

This program can be solved easily by knowing that **pointers have more precedence than arithmetic ops.** So, if a pointer variable has * next to it, we first take it as a pointer and then we consider arithmetic ops. So, the above code snippet becomes –

```
S1 = 10 * 10 = 100
S2 = 10 * 10 = 100
S3 = 1010 //Error
S4 = 1010 //Error
```

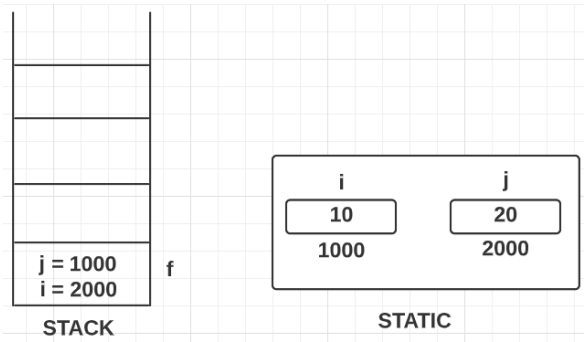
Question

Find the output of the following program

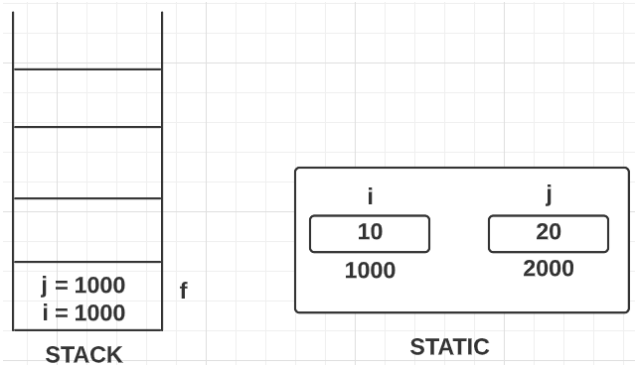
```
int i = 10;
int j = 20;
int main()
{
    f(&i, &j);
    printf("%d %d\n", i, j);
}

int f(int *j, int *i)
{
    i = j;
    *i = *j;
    *j = *i + 3;
}
```

Answer

Let us assume vars i and j are stored in locations 1000 and 2000 respectively in the static space. Now when function f is called, the values are stored in the stack –



Now we perform the operations under function f. First, we change the value of i in the stack to 1000. Then, we assign the value at address j (1000) to address i (1000). By the end of these two operations, the global variables don't change.



Now, we perform the final operation –

$$*j = *i + 3$$

$$*1000 = *1000 + 3$$

$$10 = 10 + 3 = \mathbf{13}$$

Therefore, at address 1000 the value being stored will now be **13.** Hence, the final output becomes –

$$Output = 13\ 20$$

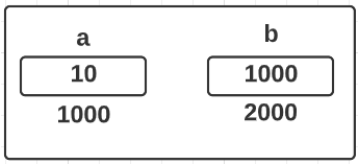
Question

Find the output of the following program –

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int *b = &a;
    printf("Enter the number = ");
    scanf("%d", b);
    printf("%d\n", a + 75);
}
```

Answer

This question may seem like an easy bag…but it is actually quite an interesting and maybe confusing one. So, let's take this slow. First, let's define the static memory for a and b as shown below –



We know that a is in integer **data var** and b is an integer **pointer var.** Now, let us move on to the next part of the program which is the scanf line. Usually, the scanf line (for int) has the following syntax –

```
scanf("%d", <address of int var>);
```

In the code, we can see that in the place of address, we are passing b which is 1000. In short, we will be passing the **scanf value to address 1000 which is nothing but a.** Effectively, the scanf can be written as –

```
Scanf("%d", &a);
```

This was the tricky part here. Hence, we can conclude that whatever the user inputs, that value will be stored in a and then it will be used to print the final result.

```
Enter the number = 4
79
```

Question

```
int main()
{
    int a, *b;
    float c, *d;
    char e, *f;
    printf("Vars - %d %d %d\n", sizeof(a), sizeof(c), sizeof(e));
    printf("Pointers - %d %d %d\n", sizeof(b), sizeof(d), sizeof(f));
    printf("Pointers values - %d %d %d\n", sizeof(*b), sizeof(*d), sizeof(*f));
}
```

Answer

This is a simple program. Like we already assumed, the data types and sizes are as follows –

- Char – 1B
- Int – 2B
- Float – 4B
- Address – 8B

So, the output will be as follows –

$$Vars - 2\ 4\ 1$$

$$Pointers - 8\ 8\ 8$$

$$Pouinter\ values - 2\ 4\ 1$$

The one thing to note here is that **whether we are storing an integer pointer, char pointer or a float pointer var, the addressing in the system doesn't change and hence all of them have the same size.**

Question

Find the output of the following code snippet –

```
int main()
{
    int a = 10, *b = &a;
    char c = 'A',  *d = &c;
    float e = 10.5, *f = &e;

    printf("%d %c %f\n", a, c, e);
    printf("%p %p %p\n", b, d, f);
    printf("%d %c %f\n", a+1, c+1, e+1);
    printf("%p %p %p\n", b+1, d+1, f+1);
}
```

Answer

In the above case, let us assume that a, c and d are stored in locations 1000, 2000 and 3000 respectively. Thus, the first three lines of the output are fairly straightforward –

```
10 'A' 10.5
1000 2000 3000
11 'B' 11.5
```

For the final line, things get interesting. If you notice, we are basically incrementing the pointer variable values. So in short, we are incrementing the address aka we are moving to the next address. Hence, we will have –

- For char variable, the address will move by 1 as the size of char is 1B
- For int variable, the address will move by 2 as the size of int is 2B
- For float variable, the address will move by 4 as the size of float is 4B

In short, the final line of the output will be –

```
1002 2001 3004
```

Question

Find the output of the following program –

```
#include <stdio.h>
int main()
{
    char a[] = "gate2011";
    printf("%s", a + a[3] - a[1]);
}
```

Answer

This is a really interesting and tricky one as it not only involves the concept of array variable being a pointer, but also the concepts of how strings and characters are dealt with in the C language. As per the first line, we are creating a **character array** a where each element is a character from the string **gate2011.**

ADDRESS	1000	1001	1002	1003	1004	1005	1006	1007	1008
VALUE	g	a	t	e	2	0	1	1	\0
INDEX	0	1	2	3	4	5	6	7	8

One major thing to note is that every string also has a **Null character** at the end of the string. This means that the string has a total of **9 characters.** Now, suppose we have the following code line –

```
printf("%s", a);
```

In this case, $a = 1000$. So, the code line will start at address 1000 and start printing in the order of addresses till it hits a Null character. Hence, the above code line will print **gate2011.** Similarly, if we have the line below –

```
printf("%s", a + 5);
```

In this case, the code line will print the values from address 1005 and go on till it reaches a Null character. So, it will print **011** in this case. Now, we can tackle the problem at hand. To do so, first we need to calculate the value to be printed –

$$a + a[3] - a[1] = 1000 + e - a = 1004$$

We can calculate $e - a = 4$ using the ASCII values. Hence, the statement effectively becomes –

```
printf("%s", 1004);
```

Hence, the output in this case will be **2001.**

NOTE – Pertaining to the above problem, we have a few things to also look out for. Suppose, we create a character array and don't provide a Null character at the end. Then, the program will print till the end of the array and then **continue to print garbage values till it encounters a Null character.** On the other hand, if we try to print a character as a string, it will throw us an error (on most compilers).

```
printf("%c", a[3]);          ERROR
```

Question

Find the output of the following program.

```c
#include <stdio.h>
int main()
{
    char a[] = "GATECSIT2017";
    printf("%d\n", strlen(a + a[2] - a[6] - 1));
}
```

Answer

In this case, we first write the character array as follows –

ADDRESS	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012
VALUE	G	A	T	E	C	S	I	T	2	0	1	7	\0
INDEX	0	1	2	3	4	5	6	7	8	9	10	11	12

Now, we can evaluate the expression as follows –

$$a + a[2] - a[6] - 1 = 1000 + 'T' - 'I' - 1 = 1010$$

Hence, the code effectively becomes –

```
printf("%d", strlen(1010));
```

Now, one important thing to note is here is that for the string from 1010 onwards, the **strlen is 2 but the sizeof will return 3.** This is because sizeof needs to calculate the memory allocated and hence includes the null character as well. However, the strlen needs to find just the length and doesn't include the null character.

Therefore, the output of this code snippet will be **2.**

Question

Find the output of the following program –

```
int main()
{
    char a[] = "abcdef";
    char b[] = "abcdef";
    if (a == b)
    {
        printf("HI");
    }
    else
    {
        printf("BYE");
    }
}
```

Answer

In this case, the same string is being broken into a character array and stored in two separate arrays a and b. However, we must remember that when we access a and b, it gives us the address and not the value. So, in this case $\boldsymbol{a}$ **is NOT EQUAL to** $\boldsymbol{b}$ **as these two arrays are stored at different memory locations**. Hence, the output is **BYE.**

We can replace the if condition with the following statements and the respective outputs are listed below –

IF Statement	OUPTUT
`if (*a == *b)`	HI. This is because the first elements of the two arrays are the same.
`if (*a = *b)`	HI. This is because we are simply assigning the value of b[0] to a[0] and that returns 1.
`if (a = b)`	Error. This is because now we are trying to change the address instead of the value which is not possible and it throws and error.

NOTE

Let us take the condition as follows –

`If (a = b)`

The following statement first assigns the value of b to a. Then, we are checking condition $if\ (a)$. So, if we have $a = 0$ and $b = 10$, then the statement will return **True.** However, if we have $a = 10$ and $b = 0$, then a will be assigned to 0 and then when checked, it will return **False.**

NOTE – Let us assume there is an array A with size 9B and base address 1000. Then, we have –

- `Sizeof(A) = 9`
- `Print(A) = 1000`
- `Print(&A) = 1000`
- `Print(A+1) = 1001` **`Skips to next element`**
- `Print(&A+1) = 1009` **`Skips full array`**

NOTE – No matter what happens, the **base address of an array CAN'T be changed!**

Question

```c
#include <stdio.h>
int main()
{
    int a[] = {10};
    int *b = a;
    printf("%d\n%d\n%d\n", a, b, *b);
}
```

Answer

In the above case, a will have the **base address of array a.** That is what will be printed first. However, when we initialize the pointer variable b, we are storing the base address of a into it. Hence, when we print b, it will again print the **base address of array a.** Finally, when we do $*\ b$, then the program will print the value at address b. Since address b is pointing to the base address of a, we get **10** as the output. Hence, the final output will be – $1000\ 1000\ 10$

Question (V V IMP)

Find the output of the following code snippet –

```
1  int main()
2  {
3      float c[] = {70, 20, 10, 50, 60, 5};
4      float *a[] = {c+2, c+3, c+4, c, c+5, c+1};
5      float **b = a;
6      printf("%d %d %d\n", b-a, *b-c, *(c+2));
7      ++*++*b+1;
8      printf("%d %d %d\n", b-a, *b-c, *c+2);
9      c + 1;
10     printf("%d %d %d\n", b-a, *b-c, **b+2);
11 }
```

Answer

In the above program, we have –

- c is a **float data array.**
- a is a **float pointer array.**
- b is a **float pointer variable.**

Let us assume that these variables are stored in the memory as shown below –

ADDRESS	1000	1004	1008	1012	1016	1020
VALUE	70	20	10	50	60	5

ARRAY C

ADDRESS	2000	2008	2016	2024	2032	2040
VALUE	1008	1012	1016	1000	1020	1004

ARRAY A

Here are some very important things to note –

- Since Array A is a float data array, each element occupies **4B.**
- Since Array C is a float pointer array, each element is an address and hence occupied **8B.**
- The case of b is very interesting. Since it is a pointer variable, we can write –

$$\boldsymbol{b = a = 2000}$$

Now that we have this information, we are effectively done till **Line 5** in the program. Let us now move forward.

Line 6

We are printing the values. The three print statements are quite similar with slight changes, so understanding 1 of them should be a good start. As we know from previous discussion, $b = 2000$. At the same time, we have $a = base\ address\ of\ A = 2000$. Hence, we get –

$$b - a = 2000 - 2000 = \mathbf{0}$$

Also, we have $* b = Value\ at\ address\ 2000 = 1008$ and $c = base\ address\ of\ C = 1000$. Hence, we get –

$$* b - c = 1008 - 1000 = 8$$

One thing to note is that each element occupies 4B in array C. So the value printed will be 8/4 = **2**

Now, we have –

$$* (c + 2) = * (1000 + 8) = \mathbf{10}$$

Hence, the final output at Line 6 will be - **0 2 10.**

Line 7

There are multiple operators involved here. Two points to note are –

- Unary operators have a higher priority as compared to binary operator.
- The unary operators have a Right – to – Left associativity.

So, Line 7 can be written as –

$$++ * ++ * b + 1 = \Big(++\big(*\big(++(*b)\big)\big)\Big) + 1 = \Big(++\big(*(++1008)\big)\Big) + 1$$
$$= \big(++(*1012)\big) + 1 = (++50) + 1 = 51 + 1 = 52$$

We can see that the value at address 2000 (to which b points to) has been incremented and updates from 1008 to 1012. At the same time, in array C, the value 50 is incremented to 51. However, **adding 1 using a binary + is not the same as increment. Hence, the value in array C is 51 and NOT 52 as binary + doesn't update address locations.**

ADDRESS	1000	1004	1008	1012	1016	1020
VALUE	70	20	10	**51**	60	5

ARRAY C

ADDRESS	2000	2008	2016	2024	2032	2040
VALUE	**1012**	1012	1016	1000	1020	1004

ARRAY A

Line 8

This is again just a print statement.

$$b - a = 2000 - 2000 = \mathbf{0}$$

$$* b - c = 1012 - 1000 = 12 = \mathbf{3}\ (divided\ by\ 4)$$

$$* c + 2 = 70 + 2 = \mathbf{72}$$

Hence, the final output becomes –

0 3 72

Line 9

Line 9 is a simple add 1 operation. Since it is a binary operator, it doesn't directly change the address location. Also, this line is not returning anything. Hence, we can successfully conclude that this **line does ABSOLUTELY FUCKING NOTHING.**

Line 10

This is again a print statement. Since there haven't been any changes from Line 8 to Line 10, we can write –

$$b - a = 0$$

$$*b - c = 3$$

Now, we have the final expression –

$$**b + 2 = *1012 + 2 = 51 + 2 = \mathbf{53}$$

Hence, the full and final output becomes –

$$0\ 2\ 10$$

$$0\ 3\ 72$$

$$0\ 3\ 53$$

NOTE

Any two pointers can be subtracted to get **the number of elements** between the two pointers as follows –

$$No\ of\ elements = \frac{P_2 - P_1}{each\ element\ size}$$

This can only work if P_1 and P_2 point to elements of the same array and all elements are of same type. However, **we can't add, multiply or divide two pointers.**

Question

Find the output of the following code snippet.

```
2  int main()
3  {
4      char *a[] = {"ABCDEF", "DEFGHI", "GHIJKL", "JKLMNO", "MNOPQR", "PQRSTU", "STUVWX"};
5      char **b[] = {a, a+2, a+1, a+3, a+4, a+5, a+6};
6      char ***c = b;
7      printf("%s\n", *(*c+2)+3);
8      printf("%c\n", *(*(*c+2)+2));
9      printf("%s\n", **(c+2)+3);
10     printf("%c\n", *(*c[1]+2)+1);
11 }
```

Answer

From the variable declaration and initialization, we can see that –

- a is a **char pointer array** and it stores the addresses of the strings.
- b is a **char double – pointer array** and it stores the addresses of elements of a.
- c is a **char triple – pointer variable** and it stores the first address of b

ADDRESS	10	20	30	40	50	60	70
VALUE	ABCDEF	DEFGHI	GHIJKL	JKLMNO	MNOPQR	PQRSTU	STUVWX

ADDRESSES OF STRINGS

ADDRESS	1000	1008	1016	1024	1032	1040	1048
VALUE	10	20	30	40	50	60	70

ARRAY A

ADDRESS	2000	2008	2016	2024	2032	2040	2048
VALUE	1000	1016	1008	1024	1032	1040	1048

ARRAY B

Also, we have –

$$c = b = 2000$$

Line 7

```
*(*c+2)+3 = *(1000+2)+3 = *1016+3 = 30+3 = 33
```

The data present at address 30 is string **GHIJKL.** Since, it is a string, it will be stored as follows –

ADDRESS	30	31	32	33	34	35	36
VALUE	G	H	I	J	K	L	\0

Hence, Line 7 will print **the string from address 33 till the null character.** And therefore, the output for Line 7 will be ***GHI***

Line 8

```
*(*(*c+2)+2) = *(*(1000+2)+2) = *(*1016+2) = *(30+2) = *32
```

As seen from Line 7, the data at address 32 is ***I*.** Also, since we print a character instead of a string, the output will be ***I*.**

Line 9

```
**(c+2)+3 = **(2016)+3 = *(1008)+3 = 20+3 = 23
```

The data present at address 20 is string **DEFGHI.** Since, it is a string, it will be stored as follows –

ADDRESS	20	21	22	23	24	25	26
VALUE	D	E	F	G	H	I	\0

The line prints the string from 23 till it encounters a null character. Hence, the output will be ***GHI.***

Line 10

```
*(*c[1]+2)+1 = *(*(*(c+1))+2)+1 = *(*(*2008))+2+1 = *(*1016)+3 =
                                  *30+3 = 33
```

Since we are printing the character, the output will be ***J.***

Hence, the final output will be as follows –

$$GHI$$

$$I$$

$$GHI$$

$$J$$

NOTE

Let us take a C code line as follows –

```
char *b = "string";
```

In the above line, *b is actually called a **string constant.** So a string constant is a char pointer var which points to the base address of a string. One thing to note is that we can't change the contents of a string constant (hence the name). So, if we run the following command –

```
b[1] = 'w';
```

The output will be undefined. There will be no error, but the output can't be predicted and changes from compiler to compiler. If we wish to change the string values, then we define the string as follows –

```
char b[] = "string";
```

Question

Find the output of the following program.

```
int main()
{
    char a[10];
    char *b = "string";
    int len = strlen(b);
    int i;
    for (i=0 ; i<len ; i++) {
        a[i] = b[len-i-1];
    }
    printf("%s", a);
}
```

Answer

ADDRESS	1000	1001	1002	1003	1004	1005	1006
VALUE	s	t	r	i	n	g	\0

We can notice from the initialization that b is a **string constant.** Hence, we have $b = 1000$. Luckily, we are not changing the value of the string at b, so the output will be defined. Also, it is helpful to remember that –

$$b[i] = *(b + i)$$

As we have discussed before, $strlen(b)$ will first go to address 1000. Then, it will go on and count the elements till it encounters a \0 (not including \0). Hence,

$$strlen(b) = 6$$

Now, we can iterate as follows –

$\boldsymbol{i}$	$\boldsymbol{b[len - i - 1]}$	$\boldsymbol{a[i]}$
0	g	g
1	n	n
2	i	i
3	r	r
4	t	t
5	s	s

Hence, the output will be $\boldsymbol{gnirts}$.

NOTE

In the above question, let us replace the line under for loop as follows –

```
a[i] = b[len-i]
```

In that case, array a will become $\{\backslash 0,\ g,\ n,\ i,\ r,\ t,\ s\}$. Now, if we try printing, then it will encounter the null character at the beginning itself and will not print. So, **no output will be printed.**

STRUCTURES AND UNIONS

These are also called **User defined Datatype** aka, these are custom datatypes that can be created by the user itself. For example, let us take the code snippet below –

```
struct node
{
    int a;
    float b;
    char c;
};

struct node d;
```

The **struct node definition** is the part where the user creates a data type. Since the user is defining a data type, **there is NO MEMORY ALLOCATED SO FAR.** The memory is allocated in the next line where we declare variable d of the **node datatype.**

$$Size\ of\ d = 2(int) + 4(float) + 1(char) = \mathbf{7B}$$

The best way to understand this concept is to imagine humans as a user defined Datatype. Every human is comprised of 1 heart, 2 lungs and 1 brain. So the organs are pre-defined datatypes and using them, we created a user defined datatype called human.

We can initialize and print the elements of the structure using **membership/select (dot) operator** as shown below –

```
d.a = 10;
d.b = 11.12;
d.c = 'S';
printf("%d\n%0.2f\n%c\n", d.a, d.b, d.c);
printf("%d", d);
```

In this case, Line 4 will print –

10

11.12

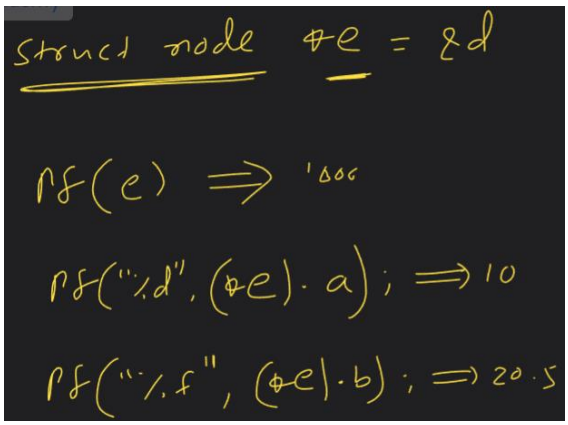
S

However, Line 5 is an erroneous line since it is not specifying which element to print. So, in this case the program will print the first element which is **10.** Now, suppose we have d stored as follows –

ADDRESS	1000	1002	1006
VALUE	10	11.12	'S'
VARIABLE	a	b	c

In this case, if we execute `printf(&d)`, then the code will print **1000.** One important thing to note is that d is a **variable** of type struct node. This means that is **NOT AN ARRAY** and hence when we print d, we are not printing the address but the element. But, d has three members inside it, which is why we can't simply provide print(d) but rather we need to use the membership operator to get the actual value from the struct node.

We can also use pointers in conjunction with structures –



NOTE

If we check the operator precedence table, we can see that membership operator (dot) has a higher priority when compared to * and hence, if we take the expression below –

```
*e.a = *(e.a) = *10 = undefined
```

Therefore, to access the element a inside d, we need to write it as `(*e).a`

The addition of brackets around e may cause confusion in larger programs and will reduce readability. Hence, we can replace –

```
(*e).a      =>    e->a
```

Hence, we prefer using the -> operator instead of * and dot.

```
struct node *e = &d;
printf("%d\n", (*e).a);
printf("%d\n", e->a);
```

In both the print statements, the output will be **10.**

ARRAY OF STRUCTURES

Suppose we have the following structure –

```
struct node
{
    int a;
    float b;
    char c;
};
```

Now, suppose we create a struct node variable as follows –

```
struct node d[3];
```

Then, this is called **array of structures.** This means that d is a array of 3 elements where each element is of struct node data type. The data will be stored as follows –

ADDRESS	1000	1002	1006	1007	1009	1013	1014	1016	1020
VALUE	10	20.5	'A'	11	30.5	'B'	12	40.5	'C'
VARIABLE	a	b	c	a	b	c	a	b	c
INDEX	0			1			2		

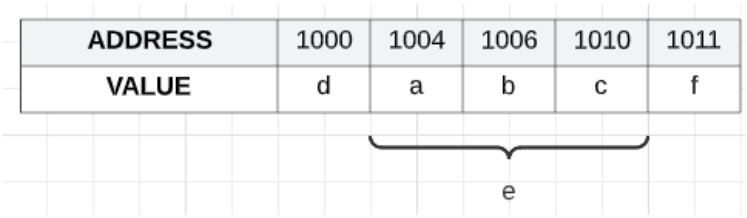
So, if we need to access element **30.5,** then we can access them using **`d[1].b`**

NESTED STRUCTURES

There can be a case where we declare a structure and use that struct data type in the declaration of another structure. It is shown below –

```c
int main()
{
    struct s1
    {
        int a;
        float b;
        char c;
    };

    struct s2
    {
        float d;
        struct s1 e;
        float f;
    }

    struct s2 g;
}
```

Here, the struct s2 variable *g* will be stored as shown below –



So, we can see that the variable *g* which is of datatype struct s2 has a variable *e* which of datatype struct s1 nested in it. Now, we can proceed with initialization and printing –

```c
//Initialization
g.d = 20.5;
g.e.a = 10;
g.e.b = 10.5;
g.e.c = 'B';
g.f = 30.5;
```

```c
//Printing
printf("Output\n");
printf("%f\n", g.d);
printf("%d\n", g.e.a);
printf("%f\n", g.e.b);
printf("%c\n", g.e.c);
printf("%f\n", g.f);
```

```
Output
20.500000
10
10.500000
B
30.500000
```

We can now try out the pointer for struct datatype. One important things to note is that **since the pointer variable stores the address, it will be of size 8B irrespective of the size of the struct variable.**

```c
struct s2 *h;
h = &g;
printf("Output\n");
printf("%f %f\n", (*h).d, h->d);
printf("%d %d\n", (*h).e.a, (h->e).a);
printf("%f %f\n", (*h).e.b, (h->e).b);
printf("%c %c\n", (*h).e.c, (h->e).c);
printf("%f %f\n", (*h).f, h->f);
```

```
Output
20.500000 20.500000
10 10
10.500000 10.500000
B B
30.500000 30.500000
```

In the above code snippet, we can see how the **->** operator works. One thing to note is that –

```
h->e->a = *((*h).e).a
```

The above line makes no fucking sense. So, we don't write h->e->a in C.

NOTE

In the above example, the program works well when we declare S1 before S2. Now, if we reverse the order, we will get **compiler error** because S2 will not know how much space to be reserved in the memory for *e* as the S1 datatype has not been defined yet.

However, suppose we replace e with $*e$, then the cod works fine. This is because $*e$ becomes a pointer and it always occupies 8B irrespective of the datatype of the variable the pointer is pointing to. So, we can declare S1 after S2 only if S2 has the following variable declaration in its definition –

```
struct s1 *e
```

Also, when we use struct, we are **creating a datatype and NOT CREATING a VARIABLE.** This means that no memory is assigned when defining the struct datatype. So, we **can't perform initialization inside structure definition.**

UNIONS

Unions are **EXACTLY THE SAME AS STRUCTURES.** The only difference between the Structures and Unions is in the way they allocate memory. For example, let us get the following structure and union definitions –

```
struct s
{
    int a;
    float b;
    char c;
}
```

```
union u
{
    int d;
    float e;
    char f;
}
```

The definition is the same for both of them. In the case of **structures, all members are allocated their own space.** So,

$$Strructure\ size = 2 + 4 + 1 = \mathbf{7B}$$

On the other hand, **in the case of Unions, all members will share the same space.** This means that the union allocated enough space to accommodate the largest element and the other members will also share the space. So, if we need to store the members, we need to **overwrite the space.**

$$Size\ of\ Union = Size\ of\ Largest\ element = \mathbf{4B}$$

This difference in memory allocation is the only difference between structures and unions.

Question

Find the output of the following program.

```c
#include <stdio.h>
int main()
{
    int a[5][3] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150};
    printf("%u %u %d\n", a, *a, **a);
    printf("%u %u %d\n", a+1, *a+1, **a+1);
    printf("%u %u %d\n", a+2, *(a+2), **(a+2));
    printf("%u %u %d\n", a+2, *(a+2), *(*(a+2)+2));
}
```

Answer

In the above question, we are given a 2D array. Of course, in the system it will be stored in a 1D format, but let us try to visualize it better. Also, as it is not explicitly mentioned, we assume the system stores this information in a Row Major Format.

1000 **10**	1002 **20**	1004 **30**
1006 **40**	1008 **50**	1010 **60**
1012 **70**	1014 **80**	1016 **90**
1018 **100**	1020 **110**	1022 **120**
1024 **130**	1026 **140**	1028 **150**

Now, we know that $a = 1000$. But what about $a + 1$? This is where things get interesting!! For a 1D array, we know that $a + 1$ would be 1002 as this is an integer array. Basically, we have –

$$*(\boldsymbol{a + 1}) = \boldsymbol{a[1]}$$

In a 1D array, this means that we are going to $a[1]$ which is effectively skipping by 1 element. But, what does $a[1]$ mean in the case of a 2D array? In the case of a 2D array,

$$*(\boldsymbol{a + 1}) = \boldsymbol{a[1]} \rightarrow \boldsymbol{Effecitvely\ skipping\ 1\ row}$$

So, we will be skipping 1 row. Hence, $a + 1 = 1006$ and not 1002. With this concept in mind, we can make the following observations –

EXPRESSION	GENERAL NOTATION	ADDRESS
a	a[0][0]	1000
*a+1	a[0][1]	1002
*a+2	a[0][2]	1004
a+1	a[1][0]	1006
*(a+1)+1	a[1][1]	1008
*(a+1)+2	a[1][2]	1010
a+2	a[2][0]	1012
*(a+2)+1	a[2][1]	1014
*(a+2)+2	a[2][2]	1016
a+3	a[3][0]	1018
*(a+3)+1	a[3][1]	1020
*(a+3)+2	a[3][2]	1022
a+4	a[4][0]	1024
*(a+4)+1	a[4][1]	1026
*(a+4)+2	a[4][2]	1028

Also, we have –

```
*a = *(a+0) = a[0][0] = a
```

Now, with this incredible information, the output of the program becomes really clear –

```
1000 1000 10
1006 1002 11
1012 1012 70
1012 1012 90
```

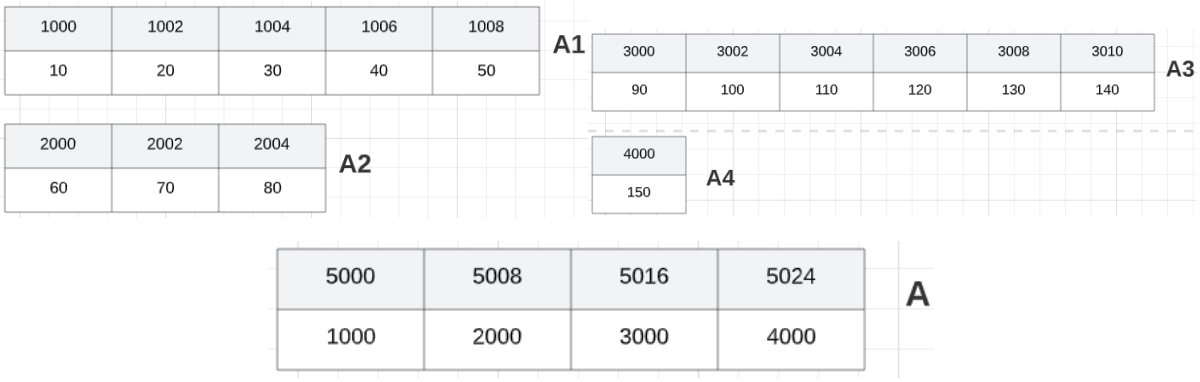
Question

Find the output of the following code snippet –

```c
#include <stdio.h>
int a1[] = {10, 20, 30, 40, 50};
int a2[] = {60, 70, 80};
int a3[] = {90, 100, 110, 120, 130, 140};
int a4[] = {150};
int *a[] = {a1, a2, a3, a4};

int main()
{
    printf("%d\n", a[2][3]);
    printf("%d\n", *a+3);
    printf("%d\n", *(*(a+2)+3));
}
```

Answer

The arrays will be stored in the memory as shown below –



Now, we can write –

```
a[2][3] = *(*(a+2)+3) = *(3000+3) = *3006 = 120
```

```
*a+3 = *(a+0)+3 = 1000+3 = 1006
```

```
*(*(a+2)+3) = *(3000+3) = *3006 = 120
```

NOTE

In the above question, we are basically using pointers to **implementing a 2D array with every row having unequal number of columns.**

ADDRESS TYPECASTING

Let us take the example below –

```
int main()
{
    int a[] = {300, 301, 302, 303, 304};
    int *b = a;
    char *c = a;
}
```

In this case, we know that a is the base address of array a. So, the second line is correct. But what about the third line? We are basically declaring a **character pointer** which is of size 8B and we are storing the base address of a which is also 8B.

Since, we are storing an address in a pointer, **there will be no syntax error message.** However, it will result in a **semantic warning** because the array a is an integer array and we are tyring to store its address in a character pointer. To resolve this semantic warning, we can perform **address typecasting** as shown below –

```
char *c = (char *)a;
```

In the above line, we have transformed the **integer address a** to a **character address** using the typecasting. This will resolve the warning. One very important thing to remember is that `(char *)` is used to convert integer address to character address. So the size will still be 8B. On the other hand, if we use `(char)` then it means we are changing the integer address to **plain character value.** In short, it will change the 8B address to 1B character and hence throw an error.

Suppose we declare a line as follows –

```
void *e = a
```

In this case, we don't need typecasting as the **void pointer accepts address of all datatypes.** Now, if we print a, b, c, d and e, we will get **1000** in all cases as they all point to the base address of a. Let us assume that a is stored in the memory in the **Little-Endian** format. This means that out of the 2B, the Lower byte is stored first and then the Upper byte. So for example, we have –

$$(300)_{10} = (0000000100101100)_2$$

So, the lower byte (00101100) will be stored in the first address and then the upper byte (00000001) will be stored in the next address. Using this, we can now imagine how a will be stored in the memory –

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
00101100	00000001	00101101	00000001	00101110	00000001	00101111	00000001	00110000	00000001

A

Now, here is the interesting part. Let us run the following command –

```
printf("%d %d %d %d %d\n", *a, *b, *c, *d, *e);
```

In this case –

- When we print *a, we will get the element at $a[0]$ which is **300**

- When we print *b, the pointer will take 2B from 1000 address as it is an integer pointer. Hence, it will also print **300.**
- When we print *c, the pointer will take 1B from 1000 address as it is a char pointer. Hence, it will print 00101100 which is nothing but **44.**
- When we print *d, the pointer will take 4B from 1000 address as it is a float pointer. It will return the 32bit or 4B value.
- When we print *e, the pointer is not aware of the datatype and hence doesn't know how many bytes to read. So, it **throws an error.**

The above logic can be used in increment as well –

$$a = b = c = d = e = 1000$$

$$b + 1 \rightarrow 1002$$

$$c + 1 \rightarrow 1001$$

$$d + 1 \rightarrow 1004$$

$$e + 1 \rightarrow \boldsymbol{ERROR}$$

$e + 1$ will give error as it is not aware of the datatype and hence doesn't know how many bytes to skip.

Question

Find the output of the following program –

```
int main()
{
    int a = 300;
    char *b = (char *) &a;
    b++;
    *b = 2;
    printf("%d", a);
}
```

Answer

Variable a is an integer which means it will be stored in 2B. Assuming Little – Endian method is used, we get –

1000	1001
00101100	00000001

Now, a char pointer b is declared and it points to address 1000. Since typecasting is used, there shouldn't be any warnings. Next, b is incremented. Since, b is a char pointer, the increment will add 1B and the new value of $b = 1001$.

Next, we are changing the value at address $b = 1001$ to 2 which is 00000010. Hence, the memory location looks like this –

1000	1001
00101100	00000010

Therefore, we have –

$$a = (0000001000101100)_2 = (556)_{10}$$

NOTE

We can see that the void pointer is useful as it can store address of any datatype. However, every time we use $*ptr$, we get an error as the pointer is not sure which datatype to use. In this case, we can perform typecasting while printing itself. For example, let us assume the following code –

```
int a = 10;
void *ptr = &a;
printf("%d", *ptr); //ERROR
printf("%d", *((int *) ptr)); //Prints 10
```

Hence, we can use void pointers as long as we can perform type casting while printing.

SELF – REFERRENTIAL DATA STRUCTURE

Suppose we use structures to create a datatype as shown below –

```
struct node {
    int a;
    struct node *b;
}
```

In this case, we have created a datatype called **struct node** which has 2 elements – Integer (2B) and a Pointer (8B). So, the total size of the struct node is **10B.** However, if we notice then we can see that struct node refers pointer ptr points to struct node itself. These datatypes are called **self – referential data structures.** One of the most commonly used self – referential data structure is a **Linked List.**

Now, let us create a node for the data type –

```
struct node a1 = {10, Null};
struct node a2 = {20, Null};
```

As we can see, we have declared the pointer to struct node as Null. This means that b will be pointing to the address of ASCII value of null character which is **Zero.** Hence, if we do $*b$, then we will get an **error as Address 0 is a special address which is reserved and not free for user to change.**

LINKED LISTS

Let us create a datatype using structures as shown below –

```
struct node {
    int data;
    struct node *next;
}
```

Like seen previously, this is a self – referential data structure. Now, let us initialize 6 nodes from a to f for this datatype –

```
struct node a = {10, NULL};
struct node b = {20, NULL};
struct node c = {30, NULL};
struct node d = {40, NULL};
struct node e = {50, NULL};
struct node f = {60, NULL};
```

Now we have 6 nodes with NULL addresses pointers and integer data. Now, we write the following code lines –

```
a.next = &b;
b.next = &c;
c.next = &d;
d.next = &e;
e.next = &f;
```

This means that a stores the address of node b, node b stores the address of node c and so on. Hence, we can represent the nodes as follows –

This is called a **Linked List.** Since the association is in a single direction only, it is called a **Singly Linked List.**

Question

Write a program to declare, traverse and print the values of singly linked list.

Answer

First, let us define the struct datatype and the different nodes in the linked list.

```
struct node {
    int data;
    struct node *next;
}

struct node a = {10, NULL};
struct node b = {20, NULL};
struct node c = {30, NULL};
struct node d = {40, NULL};
struct node e = {50, NULL};
struct node f = {60, NULL};
```

```
a.next = &b;
b.next = &c;
c.next = &d;
d.next = &e;
e.next = &f;
```

Now that we have the nodes and linked list ready, we can now declare a pointer that will be used to traverse the linked list.

```
struct node *s = &a;
int i;
for(i=0 ; i<6 ; i++)
{
    printf("%d\n", s->data);
    s = s->next;
}
```

As we can see, the pointer $* s$ is declared to point at a node to begin with. Then, will each iteration, it is printing the data of the node and then moving on to the next node when s is updated to the next address. The output will print 10 to 60 in separate lines.

Question

Write the program to find the address of the last node in a singly linked list.

Answer

Let us use the same struct node, nodes and s pointer as in the previous question. For the last node, the $next$ address value will be NULL. Hence, we can write the program as follows –

```
if(s == NULL) //To check if LL is empty
{
    printf("Linked List is empty");
}
else
{
    while (s->next != NULL)
    {
        s = s->next;
    }
    printf("%p", s);
}
```

As we can see in the above program, we are in an infinite loop which will run when the LL is not empty and when the next address of the current node is NULL. One thing to note is that if we try to print the data at s for an empty linked list, then the compiler will return **segmentation error.**

The program to check the last node address will have a time complexity of $O(n)$ since it depends on the number of nodes in the linked list.

Question

WAP to print the address of the second last node.

Answer

We can have 3 conditions to check here –

- First, we need to return if the LL is empty.
- Next, we need to return if the LL has just one element since then we don't have second last element.
- Once the above two conditions fail, then we know the LL has at least 2 nodes. So then we can go ahead and check for second last node.

```
if(s == NULL) //To check if LL is empty
{
    printf("Linked List is empty");
}
else if (s->next == NULL) //To check if LL has only 1 element
{
    printf("LL has 1 element");
}
else {
    while (s->next->next != NULL)
    {
        s = s->next;
    }
    printf("%p", s);
}
```

Just like the previous program, this program also has time complexity of $O(n)$.

Question

What does the following function do?

```
int f(struct node *p)
{
    return ((p==NULL) || (p->next==Null) || ((p->data<=p->next->data) && f(p->next)));
}
```

Answer

The function operates as follows –

- If LL is Null **or** the LL has a single element, then the function returns 1.
- Otherwise, we are checking if the current element is lesser than or equal to next element and the recursively calling the function.

Hence, this function returns 1 if –

- **LL is empty**

- **LL has 1 element**
- **LL elements are in ascending order.**

Question

WAP to add a node to the end of the Linked List with data x.

Answer

Here is a step – by – step on how to approach this problem –

1. Create a function to add a node which takes s pointer and value x as inputs
2. Create a new node.
    a. First dynamically allocate the required memory.
    b. Then set the value to x and next to NULL.
3. Check if LL is null. If that is the case, then simply point s to the new node.
4. If LL is not empty, traverse the LL till the last node.
5. At the last node, point to new node.
6. Return the LL aka first address of LL.

```
struct node *AddNodeAtEnd(struct node *s, int x) //Struct Node function
{
    struct node *p; //Pointer pointing to the new node
    p = (struct node *) malloc(sizeof(struct node)); //Memory allocation
    p->data = x; //Setting data as x
    p->next = NULL; //Since this is the new last node, setting Next to NULL
    if (s == NULL) //LL is initially empty
    {
        s = p; //P is the only node
    }
    else
    {
        //Since we need to return beginning of LL, store s in a temp var s1
        struct node *s1 = s;
        //Traverse to last node
        while(s1->next != NULL)
            s1 = s1->next;
        s1->next = p; //Set last node as P
    }

    return(s);
}
```

The program is pretty self – explanatory. However, the interesting part of the code is the dynamic memory allocation –

```
p = (struct node *) malloc(sizeof(struct node));
```

The above line reads as follows – "*Dynamically allocate memory (malloc) of the size of the struct node datatype which is **10B.** Since we know that malloc returns a **void pointer,** there is a need for typecasting to make it a struct node pointer. Store this pointer in variable **p**"*

NOTE

When we use *malloc,* the memory will be allocated in the **heap area.** Let us compare this to other memory location. If a local variable is created, then it is stored in the **stack area.** This memory will be

allocated till the function is executing. As soon as the control of the program goes out of the scope of the local variable, the memory will be de-allocated.

Next, we can have memory in the **static area.** This memory will be allocated and remain there till the entire program finishes execution.

The **heap area** is a mixture of the two. The memory allocated here will remain till the program finishes execution. However, if the user wants, they can de-allocate the memory in heap area **midway through the program execution** using the $free$ command. One more reason to use the heap area is that we need to preserve the changes done in the function to be present outside the function as well. So, we **can't use stack area.**

Question

WAP to add a node with data x before the node with data y.

Answer

```
struct node *AddNodeBeforeAnotherNode(struct node *s, int x, int y)
{
    if (s == NULL) { //If the LL is empty, then we return 0
        return 0;
    }

    struct node *p;
    p = (struct node *) malloc(sizeof(struct node));
    p->data = x;
    struct node *s1 = s;
    struct node *s2 = s;
    while(s1->data != y && s1->next != NULL) {
        s2 = s1;
        s1 = s1->next;
    }

    if (s1->data == y) {
        p->next = s1;
        if (s2 == s) {
            return(p);
        }
        else {
            s2->next = p;
        }
    }
    else {
        printf("Element %d not present in the LL", y);
    }
    return(s);
}
```

Question

WAP to delete last node of the given SLL.

Answer

```
struct node *DeleteLastNode(struct node *s)
{
    if (s == NULL) {
        printf("LL is empty");
    }
    else {
        struct node *p = s;
        struct node *q = s;
        while(p->next != NULL) {
            q = p;
            p = p->next;
        }

        if (q == s) { //LL has only 1 element
            s = NULL;
        }
        else {
            q->next = NULL;
        }
    }

    return(s);
}
```

Since we are deleting the last node which is pointed by p, we can go ahead and free up the memory of the last node using the following code –

```
free(p);
```

So, the location pointed by p is being de-allocated by using the $free$ command and that memory location is **now available.** Hence, pointer p **is pointing to an available memory location** and it is not pointing to any data. Such a pointer is called a **dangling pointer.** Let us take the following code snippet –

```
printf("%p %d %p\n", p, p->data, p->next);
free(p);
printf("%p %d %p\n", p, p->data, p->next);
p = null;
printf("%p %d %p\n", p, p->data, p->next);
```

In the above code snippet, let us assume p is pointing to address 7000. Then, the first print statement will print –

```
7000 g NULL
```

Next, we are freeing up p. This means that p is pointing to 7000, but the memory has been de-allocated aka address 7000 is empty. Hence, the next print statement will print –

```
7000 <garbage> <garbage>
```

Finally, we are setting p to be NULL. This means that we can no longer access the data and next value in that address and hence, the third print statement will give **error.**

NOTE

Suppose we have the following code snippet –

```
s1 = malloc();
s1 = malloc();
```

First, let us assume that dynamic memory location 1000 is allocated and $s1$ point to 1000 memory location. Now, in the next line we are again dynamically allocating memory (say 2000) to $s1$. So now, $s1$ is pointing to address 2000.

However, we have not freed the memory address 1000. In this case, we have no pointer pointing to 1000 memory address and so as a user we can't access that address. However, since that address has not been freed, the OS also can't allocate that memory to someone else. In short, the memory address 1000 is now useless. This is called **memory leak.** To avoid this, we need to use $free(s1)$ to free that memory address.

Question

WAP to delete a node with data x in a SLL

Answer

```
struct node *DeleteXNode(struct node *s, int x)
{
    if (s == NULL) {
        printf("LL is empty\n");
        return (s);
    }
    else if (s->next == NULL && s->data == x) {
        s = NULL;
    }
    else {
        struct node *s1 = s;
        struct node *s2 = s;
        while (s1->data != x && s1->next != NULL) {
            s2 = s1;
            s1 = s1->next;
        }
        if (s1->data == x) {
            if (s1 == s) {
                s = s1->next;
            }
            else {
                s2->next = s1->next;
                s1->next = NULL;
            }
        }
        else {
            printf("Element not found\n");
        }
    }
    return(s);
}
```

Question

WAP to move the last node to the front of the SLL.

Answer

```
struct node *PushLastToFirst(struct node *s)
{
    if (s == NULL || s->next == NULL) {
        return(s);
    }
    struct node *s1 = s;
    struct node *s2 = s;
    while (s1->next != NULL) {
        s2 = s1;
        s1 = s1->next;
    }

    s2->next = NULL;
    s1->next = s;
    s = s1;
    return(s);
}
```

Question

WAP to reverse a SLL.

Answer

```
struct node *ReverseList(struct node *s) {
    if (s == NULL || s->next == NULL) {
        return (s);
    }

    struct node *s1 = NULL;
    struct node *s2 = NULL;
    while (s->next != NULL) {
        s2 = s1;
        s1 = s;
        s = s->next;
        s1->next = s2;
    }
    s->next = s1;
    return(s);
}
```

Question

WAP to find the middle element of a SLL.

Answer

The conventional way to solve this problem would be to create a temporary pointer and counter. Then, we parse the SLL and keep incrementing the counter. This would give us the length of the SLL and dividing that number by 2 will give us the number of elements to traverse to reach the middle element. Then, we traverse that many elements and find the middle element.

However, there is a better way. It is the **hare and the tortoise** method. Basically, we have one pointer that is incremented by 1 (tortoise) and one pointer which is incremented by 2 (hare). This means that tortoise will always be half the amount as the hare. So, if the hare has reached the end of the SLL, then the tortoise points to the middle element 😊

```
struct node *FindMiddleElement(struct node *s) {
    if (s == NULL || s->next == NULL) {
        return (s);
    }

    struct node *turtle = s;
    struct node *hare = s;
    while (hare->next != NULL) {
        turtle = turtle->next;
        hare = hare->next->next;
    }
    return(turtle);
}
```

Question

WAP to perform Binary Search in a Linked List.

Answer

```
struct node *GetMidElement(struct node *low, struct node *high) {
    struct node *mid = low;
    struct node *temp = low->next;
    while (temp != high) {
        temp = temp->next;
        if (temp != high) {
            mid = mid->next;
            temp = temp->next;
        }
    }

    return(mid);
}
```

```
int BinarySearch(struct node *s, struct node *low, struct node *high, int x) {
    if (s == NULL) {
        return(0);
    }
    else if (s->next == NULL) {
        if (s->data == x) {
            return(1);
        }
        else {
            return(0);
        }
    }
    else {
        while (low != high) {
            struct node *mid = GetMidElement(low, high);
            if (mid->data == x) {
                return(1);
            }
            if (x < mid->data) {
                high = mid;
            }
            else {
                low = mid->next;
            }
        }
        return(0);
    }
```

For a normal array, the time complexity of Binary search in Worst case or Average case is $O(\log n)$. However, in best case the element will be present as the mid element itself. Since, mid calculation for an array is a simple arithmetic operation, the best case time complexity will be $O(1)$.

However, in the linked list the calculation of middle element itself has a complexity of $O(n)$ since we are using a loop to find the mid element. Hence, every case time complexity for binary search in linked list is $O(n)$. Hence, **binary search in linked list is possible but is not efficient.**

NOTE

Merge sorting has 2 parts –

- First part is to divide the array into two parts.
- Second part is to merge the arrays based on ascending ordering between the two arrays.

We can also apply merge sort in a Linked List. The time complexity for Linked List and array is **same** and is equal to $\boldsymbol{O(n \log n)}$**.** However, if we have space constraints, then it is better to use Linked Lists as there is no need for continuous memory allocation and we can simply use linking.

Question

WAP to detect a cycle in the given SLL.

Answer

There can be two solutions to this problem. The first solution is to have a separate array where we store the nodes that have been visited. So, we need to traverse the linked list $O(n)$ and for every node, we need to check in the array if it has been visited before or not $O(n)$. If the node has been visited before, then there is a cycle. This is not a good method as time complexity is $\boldsymbol{O(n^2)}$ and since we need an additional array, the space complexity is $\boldsymbol{O(n)}$**.**

The second method is much better as we will be using the **hare – tortoise pointers** again. The hare pointer increments by 2 and the tortoise pointer increments by 1. If these two pointers meet again, a cycle is present. Here, the time complexity and space complexity will be $\boldsymbol{O(n)}$ and $\boldsymbol{O(1)}$ respectively.

```
int CyclePresent(struct node *s) {
    struct node *hare = s;
    struct node *tortoise = s;
    while (hare->next != NULL) {
        hare = hare->next->next;
        tortoise = tortoise->next;
        if (hare == tortoise) {
            return(1);
        }
    }
    return(0);
}
```

DRAWBACKS OF SLL

- We can only traverse in one direction. Which means we can't traverse back to the previous node. This is because each node contains only 1 pointer that will point to the next node in the SLL.
- In the SLL, the last node always points to NULL. Which means that the last node is not being utilized properly.

CIRCULAR SINGLE LINKED LIST

This is introduced to eliminate the above two drawbacks. In this, the last node of the SLL is pointing to the first node, thus forming a loop. Due to this, we can also traverse to the previous node...but that will take n time since we need to go forward till we reach the previous node.

Question

WAP to add a node at the end of the CLL.

Answer

```
struct node *AddNodeAtEnd(struct node *s, int x) {
    struct node *p = (struct node *) malloc(sizeof(struct node));
    p->data = x;
    if (s == NULL) {
        s = p;
        s->next = s;
    }
    else {
        struct node *s1 = s;
        while (s1->next != s) {
            s1 = s1->next;
        }
        s1->next = p;
        p->next = s;
    }

    return (s);
}
```

Question

WAP to delete the first node of the C-SLL.

Answer

```
struct node *DeleteFirstNode(struct node *s) {
    if (s == NULL) {
        return(s);
    }

    struct node *s1 = s;
    while (s1->next != s) {
        s1 = s1->next;
    }
    s1->next = s->next;
    free(s);
    s = s->next;
    return (s);
}
```

DOUBLE LINKED LIST

It is the type of linked list that has 2 pointers – **next** to point to the next node in the LL and **pre** to point to the previous node in LL. It is easier to navigate through the DLL, but it occupies more space.

```
struct DLL {
    int data;
    struct node *pre;
    struct node *next;
};
```

```
struct DLL a = {10, NULL, NULL};
struct DLL b = {20, NULL, NULL};
struct DLL c = {30, NULL, NULL};
struct DLL d = {40, NULL, NULL};
struct DLL e = {50, NULL, NULL};
```

```
a.next = &b;
b.pre = &a;
b.next = &c;
c.pre = &b;
c.next = &d;
d.pre = &c;
d.next = &e;
e.pre = &d;
```

Question

WAP to insert a node with data x before a node with data y.

Answer

```
struct DLL *InsertNode(struct DLL *s, int x, int y) {
    if (s == NULL || (s->next == NULL && s->data != y)) {
        return(s);
    }

    struct DLL *s1 = s;
    while (s1->data != y && s1->next != NULL) {
        s1 = s1->next;
    }
    if (s1->next == NULL && s1->data != y) {
        printf("Element %d not present in DLL\n", y);
    }
    else {
        struct DLL *p = (struct DLL *) malloc(sizeof(struct DLL));
        p->data = x;
        p->next = s1;
        s1 = s1->pre;
        s1->next = p;
        p->pre = s1;
    }

    return (s);
}
```

Question

WAP to delete a node with data x from a DLL.

Answer

```
struct DLL *DeleteNode(struct DLL *s, int x) {
    if (s == NULL || (s->next == NULL && s->data != x)) {
        return(s);
    }
```

```
    struct DLL *s1 = s;
    while (s1->data != x && s1->next != NULL) {
        s1 = s1->next;
    }
    if (s1->next == NULL && s1->data != x) {
        printf("Element %d not present in DLL\n", x);
    }
    else if (s1->pre == NULL && s1->data == x) {
        s = s1->next;
        s->pre = NULL;
    }
    else {
        s1->pre->next = s1->next;
        s1->next->pre = s1->pre;
    }

    return (s);
}
```

CIRCULAR DLL

This is a DLL where the last node next is the first node and first node pre is the last node. Everything else is the same as the DLL.

Question

WAP to add a node with data x to the end of a C-DLL

Answer

```
struct DLL *InsertNodeAtEnd(struct DLL *s, int x) {
    struct DLL *p = (struct DLL *) malloc(sizeof(struct DLL));
    p->data = x;
    if (s == NULL) {
        s = p;
        s->next = s;
        s->pre = s;
    }
    else {
        struct DLL *s1 = s->pre;
        s1->next = p;
        p->pre = s1;
        p->next = s;
        s->pre = p;
    }

    return(s);
}
```

As we can see in the above case, there are **No Loops** involved. That means, this program has a constant time complexity of $\boldsymbol{O(1)}$**.** This is better than a regular DLL which has the time complexity of $\boldsymbol{O(n)}$**.**

Question

WAP to delete the first node of C-DLL.

Answer

```
struct DLL *DeleteFirstNode(struct DLL *s) {
    if (s == NULL) {
        printf("C-DLL is empty.\n");
    }
    else {
        struct DLL *s1 = s;
        s = s1->next;
        s->pre = s1->pre;
        s1->pre->next = s;
        free(s1);
        s1 = NULL;
    }

    return(s);
}
```

BINARY TREE

Binary tree is defined like a DLL with the following components –

- Integer data for the given node.
- RC pointer to point to the right child node.
- LC pointer to point to the left child node.

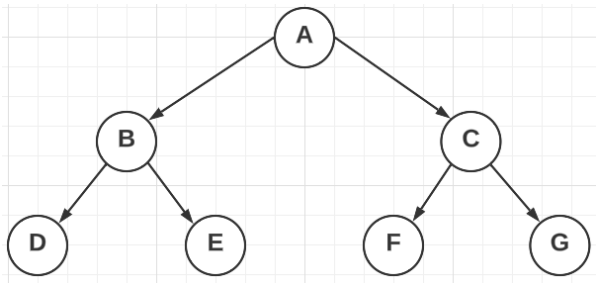
```
struct BTnode {
    int data; //Data in the tree node
    struct BTnode *rc; //Points to the right child node
    struct BTnode *lc; //Points to the left child node
};
```

In DLL, the two pointers point to the next node and to the previous node. However, in the BT the two pointers point to new nodes. If a node in BT has both LC and RC as null, then it is called a **leaf node.**

```
struct BTnode a = {10, NULL, NULL};
struct BTnode b = {20, NULL, NULL};
struct BTnode c = {30, NULL, NULL};
struct BTnode d = {40, NULL, NULL};
struct BTnode e = {50, NULL, NULL};
struct BTnode f = {60, NULL, NULL};
struct BTnode g = {70, NULL, NULL};
```

```
a->lc = &b;
a->rc = &c;
b->lc = &d;
b->rc = &e;
c->lc = &f;
c->rc = &g;
```

The above declarations result in a graph as shown below –



Question

WAP to find the number of leaf nodes in the BT.

Answer

To get the number of leaf nodes, we need to find the nodes which have both LC and RC as NULL. If the node has both LC and RC as NOT null, then we recursively apply the same function to its children.

```
10  int LeafNodes(struct BTnode *s) {
11      if (s == NULL)
12          return 0;
13      if (s->rc == NULL && s->lc == NULL) {
14          return(1);
15      }
16      else {
17          int leftNum = LeafNodes(s->lc);
18          int rightNum = LeafNodes(s->rc);
19          int total = leftNum + rightNum + 1;
20          return(total);
21      }
22  }
```

If we have a balanced BT (where in each node has 2 children – one left and one right), then the above program will achieve the **best case scenario.** This is also the **average case scenario** as that is how most BTs will be. On the other hand, the **worst-case scenario** will be when the BT is skewed to either left or right. In **every case, the time complexity is $O(n)$.**

Also, as this is a recursive program, we will need extra space in stack as well since the function is being called multiple times. The **space complexity** depends on the number of levels of the BT. In **best case,** the number of levels in $\log n$ and hence space complexity will be $O(\log n)$**.** In the **worst-case scenario**, each level has only 1 node. So, the number of levels is n and hence the space complexity becomes $O(n)$**.**

There can be several variations of the above program –

To find the number of Non – leaf nodes

First off, we need to change line 14 to `return 0` instead of 1 since Line 14 is only executed for leaf nodes and we don't need them anymore. The rest of the program remains the same except for Line 19 which becomes –

```
int total = leftNum + rightNum + 1
```

The additional 1 is added otherwise the total will always be 0.

To find the total number of Nodes

Line 14 and Line 19 will be changed as follows –

```
return 1;
```

```
int total = leftNum + rightNum + 1
```

To find the total number of nodes in the Right – most path

Line 14 and Line 19 will be changed as follows –

```
return 1;
```

```
int total = rightNum + 1
```

To find the total number of nodes in the Left – most path

Line 14 and Line 19 will be changed as follows –

```
return 1;
```

```
int total = leftNum + 1
```

Question (V V IMP)

WAP to find the height of the BT.

Answer

Height of a BT is the longest route between the root node and a leaf node. To get this, we can simply keep recursively going from node to node till we hit a lead node. Then, we can get the bigger value among the left and right values.

```
int FindHeight(struct BTnode *s) {
    if (s == NULL)
        return 0;
    else {
        int leftNum = FindHeight(s->lc);
        int rightNum = FindHeight(s->rc);
        if (leftNum < rightNum) {
            return rightNum + 1;
        }
        else {
            return leftNum + 1;
        }
    }
}
```

A few things to note about the height of a binary tree –

- Height of a leaf node is the same as the height of an NULL BT which is **zero.**
- Height of the BT is **one less than the number of levels.**

Question

WAP to find whether the given BT is a strict BT or not.

Answer

A strict BT is a BT where every node either has 0 or 2 children.

```
int StrictBT(struct BTnode *s) {
    if (s == NULL)
        return 0;
    else {
        if (s->lc == NULL && s->rc == NULL) {
            return 1;
        }
        if ((s->lc != NULL && s->rc == NULL) || (s->lc == NULL && s->rc != NULL)) {
            return 0;
        }
        else {
            int leftNum = StrictBT(s->lc);
            int rightNum = StrictBT(s->rc);
        }
    }
}
```

In the above program, for the worst case scenario we will have to parse till the leaf nodes making the time complexity to be $\boldsymbol{O(n)}$**.** This is also the average case. However, in the best case scenario, the root node itself may have only 1 child. In that case, we don't even need to enter the recursion. Hence, the best case time complexity is $\boldsymbol{O(1)}$**.**

Question

WAP to check if two BT are equal or not.

Answer

There are three scenarios to look into for this program –

- If the two BTs are null, then they are equal.
- If one of the BTs is null and the other is not, then they are not equal.
- If both the BTs are not null, then we need to recursively check all the nodes and edges.

```
int BTEquality(struct BTnode *s1, struct BTnode *s2) {
    if (s1 == NULL && s2 == NULL) {
        return 1;
    }
    if ((s1 == NULL && s2 != NULL) || (s1 != NULL && s2 == NULL)) {
        return 0;
    }

    return (s1->data == s2->data
        && BTEquality(s1->lc, s2->lc)
        && BTEquality(s1->rc, s2->rc));
}
```

On average and even in the worst case, the time complexity will be $\boldsymbol{O(n)}$**.** On the other hand, the best case scenario can be achieved in the following cases –

- Both the trees are NULL.
- One of the tree is null but the other is not.
- Both trees are not null but the root nodes are different.

For all these cases, we need a single if condition to determine the result. Hence, the time complexity becomes $\boldsymbol{O(1)}$**.**

Question

WAP to print the mirror image of a BT.

Answer

```
struct BTnode *MirrorImage(struct BTnode *s) {
    if (s == NULL) {
        return(s);
    }
    struct BTnode *s1 = s;
    struct BTnode *temp = s1->lc;
    s1->lc = s1->rc;
    s1->rc = temp;
    MirrorImage(s1->lc);
    MirrorImage(s1->rc);

    return(s);
}
```

BINARY SEARCH TREE

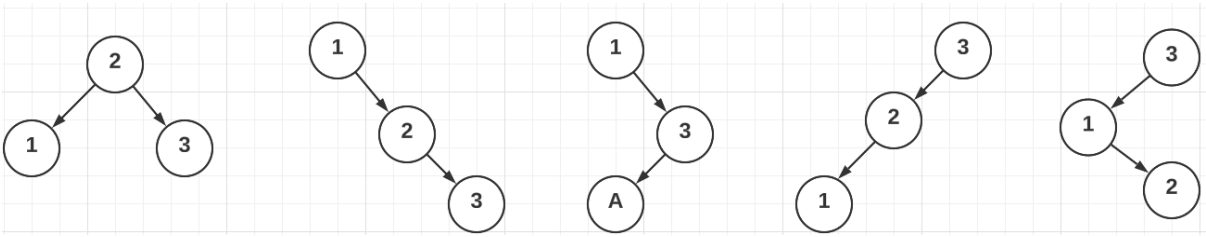
A BST is the same as the binary tree, but for any node c, the following condition is satisfied –

$$c \rightarrow left \rightarrow data > c \rightarrow data > c \rightarrow right \rightarrow data$$

Also, each node **is unique** and data is not repeated in any of the nodes. For a set of n nodes, there are at least n unique BSTs. However, there is no way of knowing the max number of BSTs. For example, let us take –

$$nodes = \{1, 2, 3\}$$

The unique BSTs that can be formed are –



Hence, we can have 5 BSTs. Similarly, for 4 nodes and 5 nodes, the possible BSTs are 14 and 42 respectively. In general, we can write –

$$Number\ of\ BSTs\ \ T(n) = \sum_{i=0}^{n-1} T(i) * T(n-i-1)$$

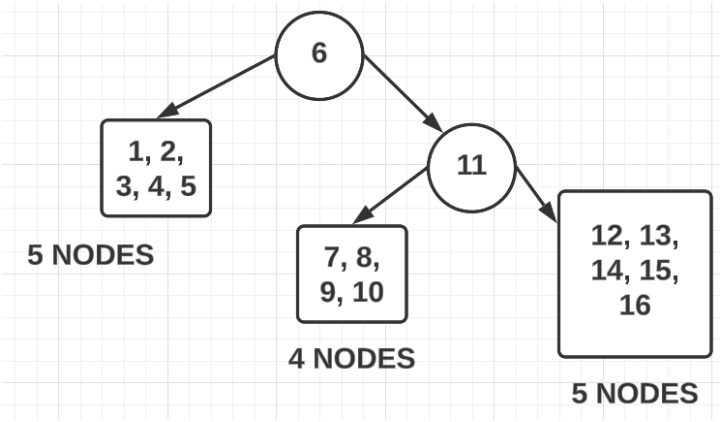
Also, $T(0) = T(1) = 1$

Question

How many BSTs are possible with 16 nodes such that the root node is 6 and the right sub – tree root node is 11.

Answer

The BST will be as follows –



So, total number of possibilities –

$$N = 5 - nodes * 4 - nodes * 5 - nodes = 42 * 14 * 42 = \mathbf{24696}$$

Question

How many BSTs are possible with 7 nodes in such a way that the height of the BSTs is 6.

Answer

This is a tricky situation. Since we want the height to be 6, the number of levels will be $6 + 1 = \mathbf{7}$**.** Hence, we need to have just one node per level. That means that every level will have either the smallest remaining or the largest remaining node. Hence, **all levels can have 2 possibilities except for the last node which will have the remaining element.**

$$Number\ of\ BST = 2 * 2 * 2 * 2 * 2 * 2 * 1 = \mathbf{64}$$

NOTE

$$No\ of\ unlabelled\ BT = No\ of\ BST = \frac{C_n^{2n}}{n+1}$$

Question

Suppose we have an unlabelled binary tree with n nodes and we have a set of values $S = \{1, 2, 3 \dots n\}$. Then, how many ways are there to insert the values from set S into the unlabelled binary tree so that it becomes a BST?

Answer

In the above question, it is stated that we have an unlabelled binary tree to fill. We also know from the discussion above this question that the number of unlabelled binary trees and the number of BSTs for n nodes are the same. Hence, it is safe to say that **each unlabelled binary tree corresponds to a BST.** Hence, there is only **one way to fill an unlabelled binary tree to make it a BST.**

NOTE

In a BST, the left – most node and the right – most node will be the **smallest** and the **largest elements** in the BST. In best case scenario, the first node itself is the smallest element and it has no nodes at the left of it. Hence, the time complexity becomes $\boldsymbol{O(1)}$**.** For the worst case, we might need to traverse all the nodes to get the left – most node. Hence, the time complexity becomes $\boldsymbol{O(n)}$**.** Finally, in an average case, the BST contains only $\boldsymbol{\log n}$ levels itself. So, the time complexity becomes $\boldsymbol{O(\log n)}$**.**

Question

Find the time complexity to find the minimum element.

Answer

In general, the time complexity to find the minimum element is given as follows –

	BEST CASE	WORST CASE	AVERAGE CASE
BINARY TREE	n	n	n
BST	1	n	$\log n$
AVL TREE	$\log n$	$\log n$	$\log n$
Min Heap	1	1	1
Max Heap	n	n	n

AVL tree is a balanced BST. This means that we need to find the left – most node, but since the tree is balances, we can't get a NULL left node in the beginning. Hence, the $\log n$ complexity. As for min heap, the root node is the minimum element. Hence, the time complexity will always be 1. As for max heap, the root node is the maximum element. So, we don't know where the minimum element will be in the max heap.

Question

Find the time complexity to find the element with data x.

Answer

	BEST CASE	WORST CASE	AVERAGE CASE
BINARY TREE	1	n	n
BST	1	n	$\log n$
AVL TREE	1	$\log n$	$\log n$
Min Heap	1	n	n
Max Heap	1	n	n

In all the cases, the best case will be that root node itself has data x. This would mean that the best case will have time complexity of $\boldsymbol{O(1)}$. In the worst – case scenario, we would have to traverse all the elements to get the element with x data. Hence, the time complexity becomes $\boldsymbol{O(n)}$. The exception to this is the AVL tree since the AVL tree is a balanced BST. This means that in the worst case scenario, it has a max of $\log n$ levels. As for average case, the same logic follows. However, in BST also the average number of levels are usually $\log n$.

Question

Find the time complexity to know that element x is **NOT** present.

Answer

	BEST CASE	WORST CASE	AVERAGE CASE
BINARY TREE	n	n	n
BST	1	n	$\log n$
AVL TREE	$\log n$	$\log n$	$\log n$
Min Heap	1	n	n
Max Heap	1	n	n

For BST, the best case scenario will be that x is greater than the root node value but the root node has no right child. That means, the root node is the max element in the BST and hence the value x is not present in the BST. Hence, the best case scenario time complexity becomes $\boldsymbol{O(1)}$. On the other hand, the worst case will be that all the nodes are on the right and we would have to traverse to the end of the tree. Hence, the time complexity becomes $\boldsymbol{O(n)}$. The average will remain $\log n$ as we have an average of that many levels.

Binary tree on the other hand is like an unsorted array. Which means, we need to parse through the whole tree to make sure that the element is not present. Hence, the value of $\boldsymbol{O(n)}$ will be the time complexity for every case.

AVL tree is a balanced BST. Which means, that the root will have 2 children. Hence, the best case, worst case and the average case becomes $\boldsymbol{O(\log n)}$.

In min heap, the root element is the minimum value. So if x is lesser than the root element, we know that element x will not be present. Similarly for max heap, if x is greater than the root element, we know that element x will not be present. Hence, the best case scenario will be $\boldsymbol{O(1)}$. The other scenarios will be $\boldsymbol{O(n)}$.

Question

In BST, what is the time taken to find **any element** that is neither minimum nor maximum?

Answer

This may seem like an abstract question, but the logic is very simple. We already know that a BST is a balanced tree which means the max and the min elements are on the right-most and the left-most positions of the tree. Since we need to find **any** elements that is not these two, we can simply take the root node, the root right sub-tree node and the root left sub-tree node –

- If the root node has both left and right children, then it is neither max nor min and we have found the solution.
- If the root node has no right child, then it means that the root node is the max element and the node to the left will be the solution
- Similarly, if the root node has no left child, then it means that the root node is the min element and the node to the right will be the solution.

One very important thing to node is that this works only in the case of a BST with **3 or more nodes** since a BST with 2 nodes will have one element as the max and one as the min (DUH!). Since, we are not parsing the tree in any of the cases and we know which nodes to access, the time complexity for every case in this question will be $\boldsymbol{O(1)}$**.**

BST INSERTION

The insertion of a new element in BST is tricky since there is an order to be maintained. So, here is the brief way of approaching this problem –

- First create the new node using malloc.
- Then, from node start traversing as if you are searching for the new node. If the value is greater, then move to the right. Else, move the left.
- Stop where we encounter the next address as Null.
- Insert the new node there.

The time complexity is as follows –

	BEST CASE	WORST CASE	AVERAGE CASE
To insert 1 node	1	n	$\log n$
To insert n nodes	$n \log n$	n^2	$n \log n$

NOTE

In – order traversal of a tree is when we first print the left child, then the root node and then the right child. Since, the BST has the smaller element to the left and the larger element to the right, the in – order traversal of BST will result in **an ascending order sequence.** To find the predecessor of a node in the in-order traversal, we need to find the **maximum value in the left sub-tree**. Similarly, to find the successor of a node, we need to find the **minimum value in the right sub-tree.**

DELETION IN BST

Deletion in BST can be ranked in 3 different levels of difficulty –

- Deleting a leaf node without any children
- Deleting a node with 1 child
- Deleting a node with 2 children

For a leaf node, we simply first find the node in the BST. Then, we change the address in the parent node to NULL and then free up the leaf node. Simple, straight to the point.

For a node with one child, we use pointer p to point to the node to be deleted and the pointer q to point to the parent node. Let us assume that p is the right child of q. Also, let us assume that the node p has a right child. Then, we can delete p node as follows –

```
q->rc = p->rc;
free(p);
p = NULL
```

So, now the parent node q still has 1 right child but that is not p, but the child of p. So, we have successfully deleted a node with one child.

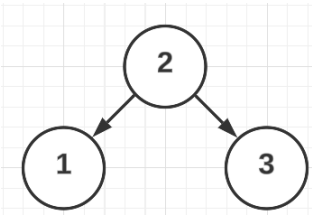
Now, let us look at the condition where we need to delete a node with 2 children. This is a tough one as we don't know where to point since we need to maintain the BST order. To do so, we simply need to delete the node and **replace it with either the predecessor or the successor as per the in-order traversal.**

AVL TREE

It is a balanced BST tree. That means, for a group of n nodes, the number of levels will always be $\log n$. At a node x, the **balancing factor** can be given as –

$$BF = Height\ of\ Left\ subtree - Height\ of\ Right\ subtree$$

In an AVL tree, the BF at a node can be either -1, 0 or 1. Let us take examples to understand this concept.



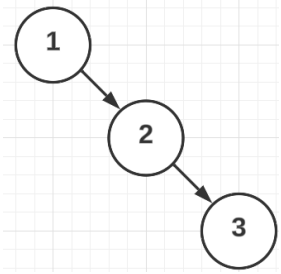
In the above BST, we calculate the balancing factor for the nodes –

$$BF_1 = Height(Left) - Height(Right) = 0 - 0 = \mathbf{0}$$

$$BF_2 = Height(Left) - Height(Right) = 1 - 1 = \mathbf{0}$$

$$BF_3 = Height(Left) - Height(Right) = 0 - 0 = \mathbf{0}$$

Since the balancing factor at each node is 0, then the above graph is said to be a AVL graph. Now, let us take another graph with the same three nodes –



In the above BST, we calculate the balancing factor for the nodes –

$$BF_1 = Height(Left) - Height(Right) = 0 - 2 = \mathbf{-2}$$

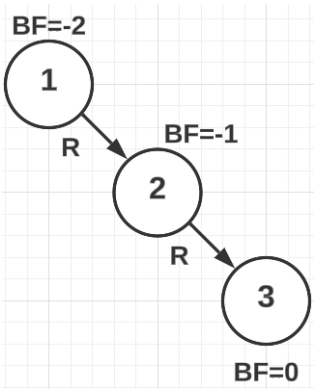
$$BF_2 = Height(Left) - Height(Right) = 0 - 1 = \mathbf{-1}$$

$$BF_3 = Height(Left) - Height(Right) = 0 - 0 = \mathbf{0}$$

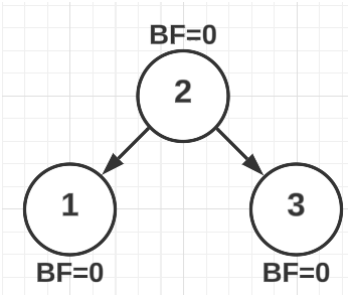
As we can see, the balancing factor for Node 1 is **-2** and is not in the set $\{-1, 0, 1\}$, hence the above BST is **NOT** an AVL tree.

RR – PROBLEM

Suppose we take the graph as shown below –



In the above graph, we can see that the node 1 has a BF of -2 and hence the BST is not an AVL. This is because the node 1 has 2 Right (RR) edges. Hence, this is called the **RR – problem.** To solve this problem, we need to perform **Left Rotation** on the problem node which is **Node 1.** Performing Left rotation will result in the following graph –

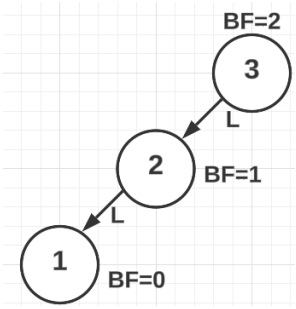


Now the BST is an AVL. This can be written in code as follows –

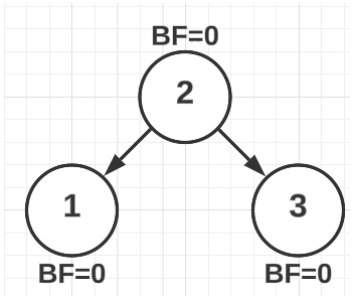
```
2->lc = 1;
1->rc = NULL;
```

LL – PROBLEM

Similar to the above problem, in this case there are 2 Left edges thus causing a BF of +2 at the root node.



In this case, we perform **Right rotation** on **Node 3** and create a BST as shown below –

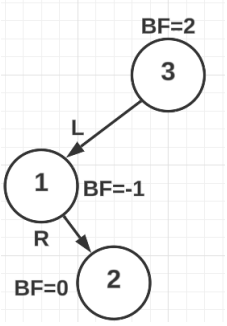


This is an AVL. The code can be written as –

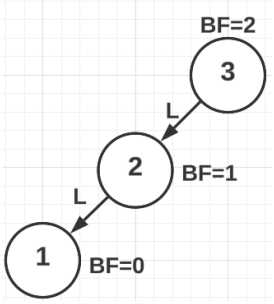
```
2->rc = 3;
3->lc = NULL;
```

LR PROBLEM

This is where the nodes have a left and right edge as shown below –

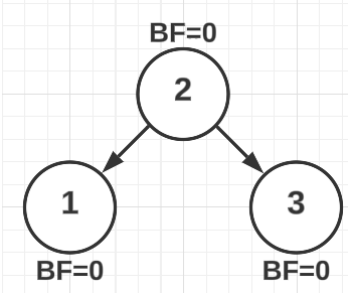


In this case, we have a left and a right edge, but at the same time we have a BST that is not an AVL. Since we have both left and right, we need **2 rotations** to resolve this. First, we make a left rotation between Nodes 1 and 2 –



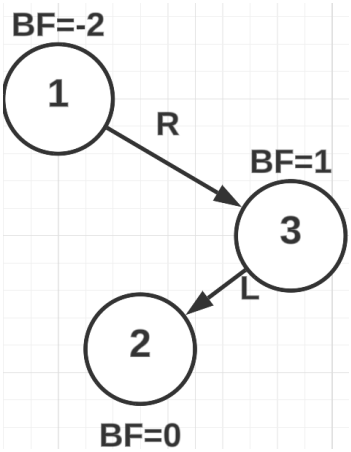
```
2->lc = 1;
1->rc = NULL;
```

After this, we can solve the LL problem by performing a Right rotation as discussed previously –

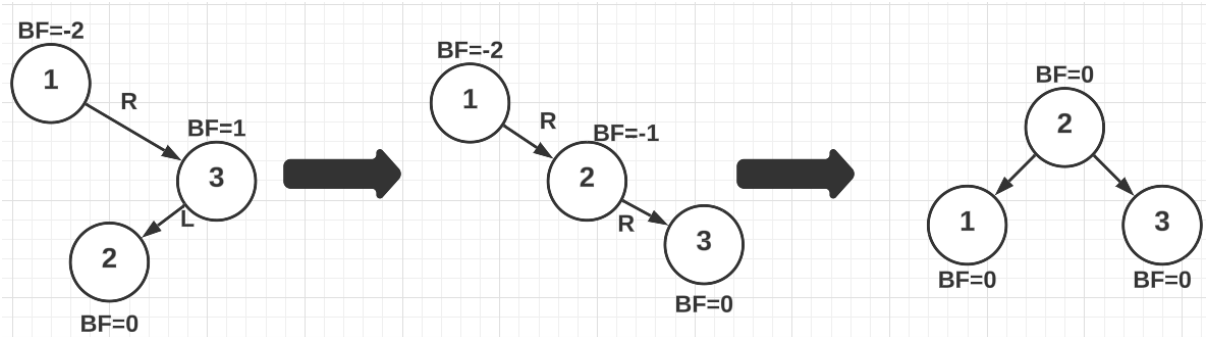



```
2->rc = 3;
3->lc = NULL;
```

RL PROBLEM



Just like the previous problem, we will have 2 rotations – first to the right and then to the left.



NOTE

When inserting in an AVL tree, we need to follow the given procedures –

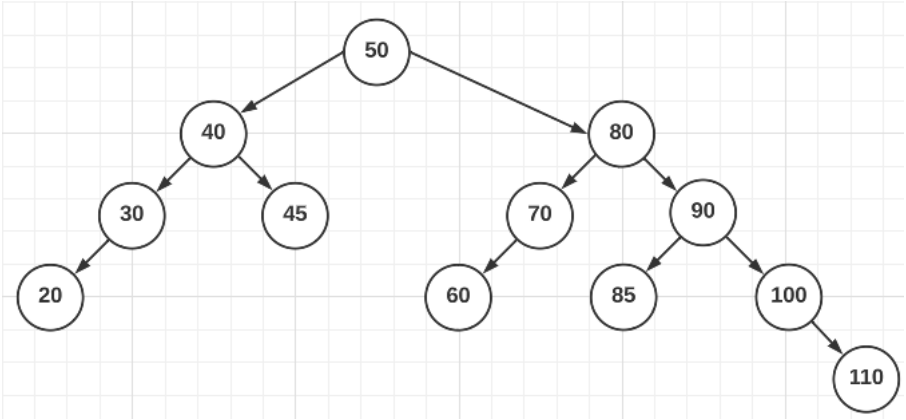
- First, traverse the tree to get to the node where insertion needs to be made $(O(\log n))$

- Next, insert the node. $\left(O(1)\right)$
- Next, we traverse back to the top while calculating the BF of all nodes post the insertion $\left(O(\log n)\right)$
- If there is either LL, RR, LR or RL problem, perform rotation to resolve the problem.

As seen above, the insertion of 1 element in a n node AVL tree has the time complexity of $\boldsymbol{O(\log n)}$**.** Since inserting 1 node takes $\log n$ time, then construction of a n node AVL tree will take $\boldsymbol{O(n \log n)}$ time complexity.

Question

In the AVL tree shown below, element with value 150 is inserted. After insertion, how would the AVL tree look?

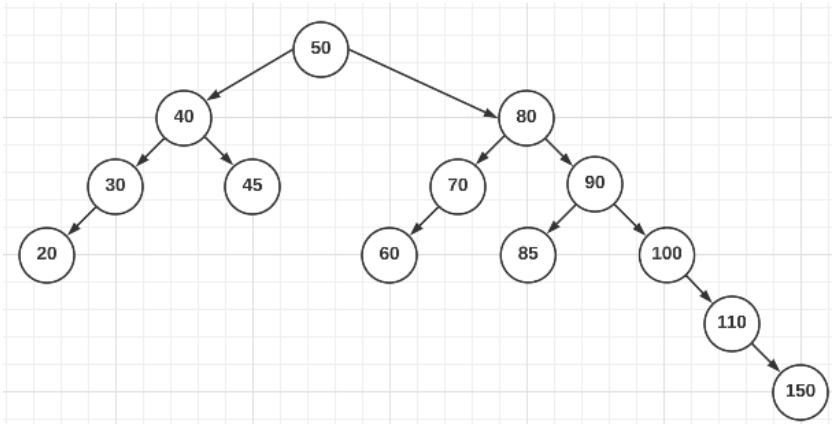


Answer

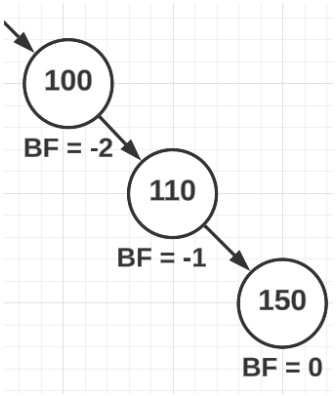
As we can see, in the above tree the BF of the nodes are either 0, -1 or 1. Hence, it is an AVL tree. Now, to insert 150, we start from the root node and go all the way till we get the right position for it –

- 150 > 50. Move right.
- 150 > 80. Move right.
- 150 > 90. Move right.
- 150 > 100. Move right.
- 150 > 110. Move right.

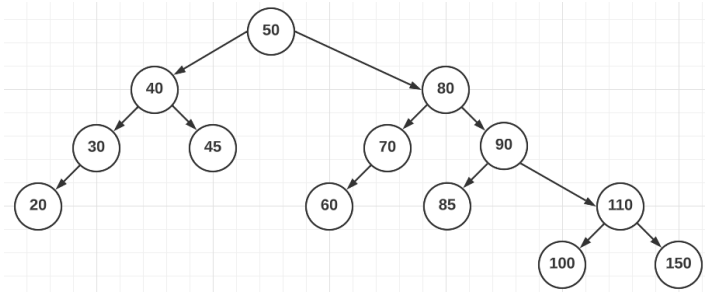
Since we have reached a leaf node, the node with 150 will be a right child of the node with value 110.



In the next step, we now need to retrace our steps and find the BFs of the nodes now to check if there are any discrepancies. Once we do that, we can see one problem –



Since we encountered a problem, we will fix this first before moving ahead. As we can see, this is a **RR problem.** So, it can be resolved by performing a **left rotation.** Hence, the tree now becomes –



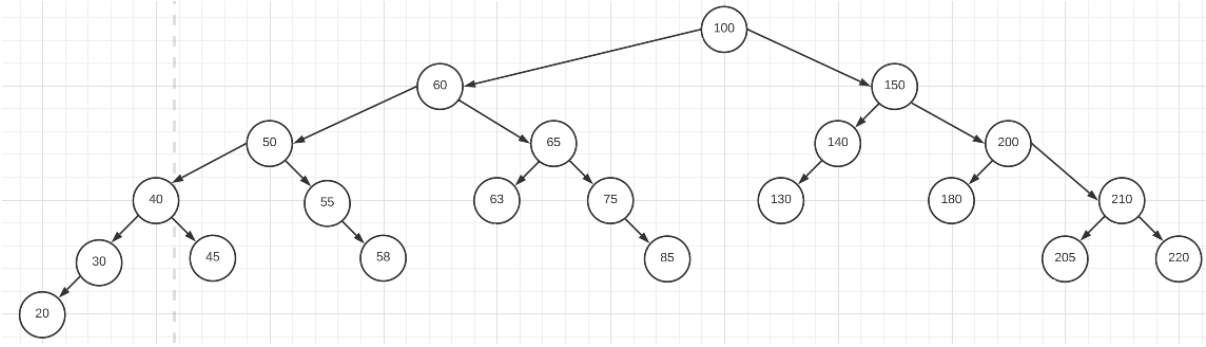
Now, if we check, we can see that there are no more problems and all the BFs are either 0, -1 or 1. Hence, this is the **final AVL tree.**

NOTE

During insertion or deletion, a maximum of $\log n$ problems can occur and a maximum of $2 \log n$ rotations can be done.

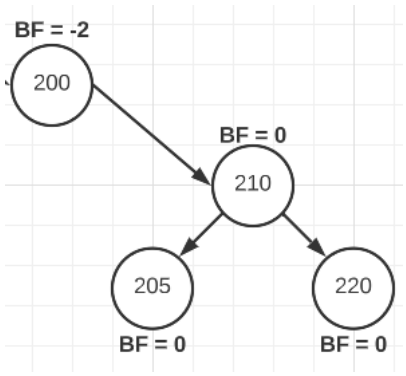
Question

In the below AVL tree, let us assume that the node 180 is deleted. Find the new AVL.

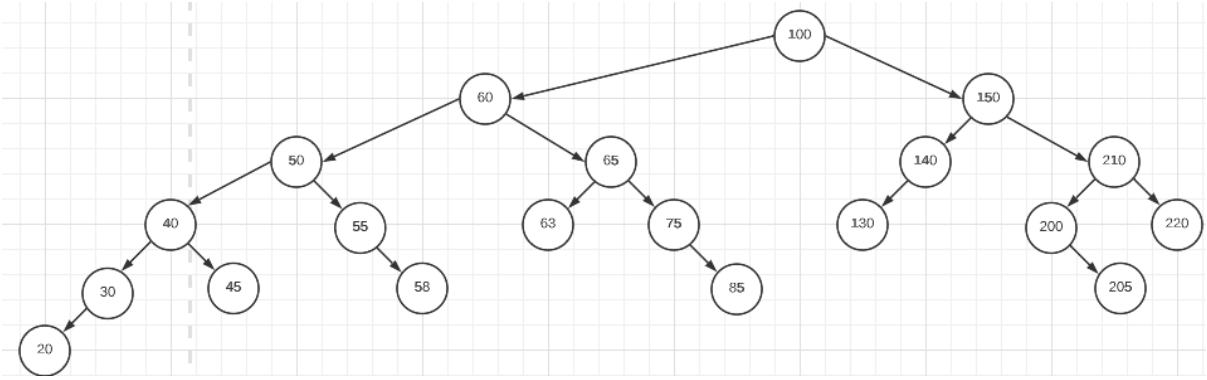


Answer

Let us delete node 180. Since it is a leaf node, there should be no issues. However, we can see that there will be an issue with the BF –



In this case, we have both RR and RL problems here. However, we usually solve the RR problems since we would need just one rotation. Then, we have the following tree –



NOTE

To get the minimum number of nodes to achieve an AVL tree of height –

$$MNN(H) = \begin{Bmatrix} 1 & if\ H = 0 \\ 2 & if\ H = 1 \\ MNN(H-1) + MNN(H-2) + 1 & if\ H > 1 \end{Bmatrix}$$

To get the maximum number of nodes to achieve an AVL tree of height –

$$MXNN(H) = 2^H - 1$$

NODES	MAX HEIGHT
0	0
1	0
2	1
4	2
7	3
12	4
20	5
33	6
54	7

TREE TRAVERSAL

There are three ways to perform tree traversal –

- Pre-order
  - First Root
  - Second Left sub-tree
  - Third Right sub-tree
- Post-order
  - First Left sub-tree
  - Second Right sub-tree
  - Third Root
- In-order
  - First Left sub-tree
  - Second Root
  - Third Right sub-tree

Since this is traversal, we are **visiting all nodes exactly once** and hence, the time complexity for all cases becomes $\boldsymbol{O(n)}$**.** The codes for the traversal will be given as follows –

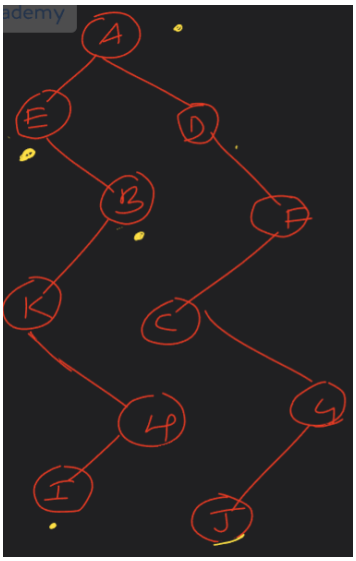
```c
// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}
```

```c
// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

```c
// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
```

Question

Find the pre-order, post-order and in-order traversal of the following tree.



Answer

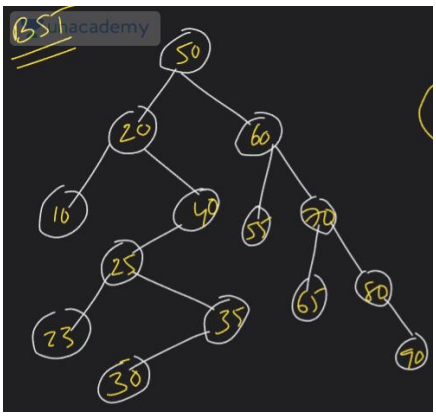
$$Pre - order = A \rightarrow E \rightarrow B \rightarrow K \rightarrow H \rightarrow I \rightarrow D \rightarrow F \rightarrow C \rightarrow G \rightarrow J$$

$$Post - order = I \rightarrow H \rightarrow K \rightarrow B \rightarrow E \rightarrow J \rightarrow G \rightarrow C \rightarrow F \rightarrow D \rightarrow A$$

$$In - order = E \rightarrow K \rightarrow I \rightarrow H \rightarrow B \rightarrow A \rightarrow D \rightarrow C \rightarrow J \rightarrow G \rightarrow F$$

Question

Write the in – order traversal of the BST tree –



Answer

$$In - order\ traversal = 10 \rightarrow 20 \rightarrow 23 \rightarrow 25 \rightarrow 30 \rightarrow 35 \rightarrow 40 \rightarrow 50 \rightarrow 55 \rightarrow 60 \rightarrow 65 \rightarrow 70 \rightarrow 80 \rightarrow 90$$

As we can see, the in – order traversal of a BST tree will result in an ascending order.

NOTE

The first element and the last element of pre-order and post-order traversal respectively give the **root node** of the tree. If we take the root node and find its location in the in-order traversal, then we can find the left and right sub tree –

$$In - order = < left\ sub - tree > < Root\ node > < Right\ sub - tree >$$

Question

Consider the following Binary Tree data –

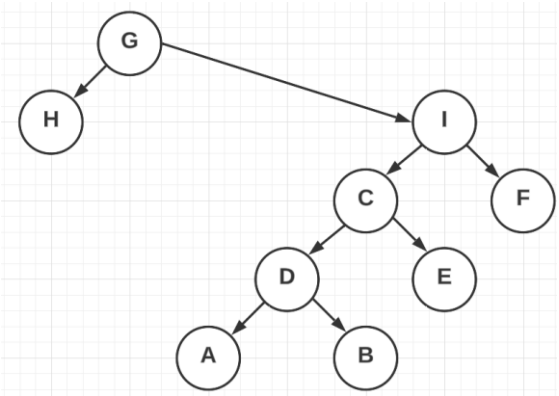
$$In - order = H \rightarrow G \rightarrow A \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow I \rightarrow F$$

$$Pre - order = G \rightarrow H \rightarrow I \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow E \rightarrow F$$

Find, the post – order traversal order.

Answer

From the pre-order sequence, we know that node **G** is the root node. Now, using that info in the in-order sequence, we realize that the left sub-tree consists of only node **H** and the right sub-tree has the rest of the nodes. We will use that information to now build a tree as follows –



Using the tree above, we can write the post-order traversal sequence –

$$H \rightarrow A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow I \rightarrow G$$

Question

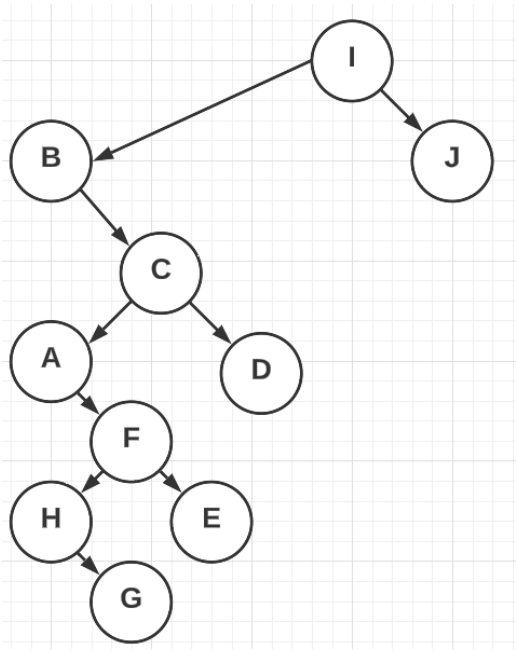
Find the Binary tree given the Post and in-order sequences –

$$Post = GHEFADCBJI$$

$$In = BAHGFECDIJ$$

Answer

We can see that Node I is the root node and node G is the lowest leaf node. Then, going backwards from I in the post – order sequence, we take the nodes and then compare their position in the in – order sequence and can draw a tree.



NOTE

To create a Binary Tree from either the Pre-order, post-order or the in-order expressions needs a time complexity of $O(n^2)$ as we are traversing the list of n nodes to find the position of the given node. And this process is then repeated for each of the n nodes.

Question

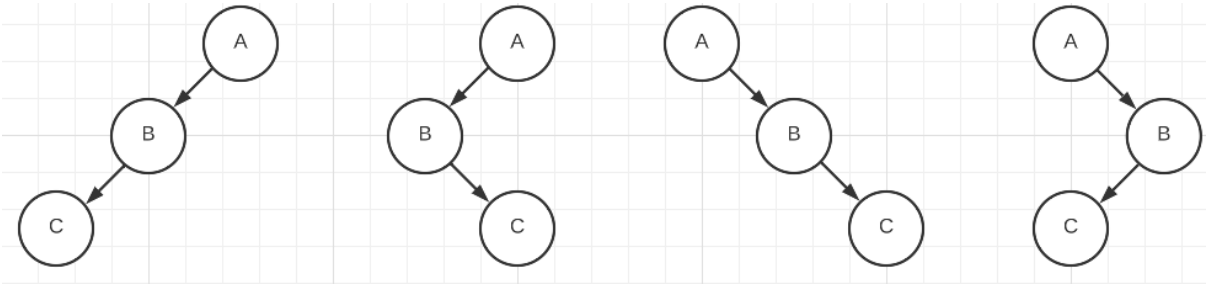
Draw the tree for the following sequences –

$$Pre - order = ABC$$

$$Post - order = CBA$$

Answer

In this case, we have the root node as A. However, we can see that there are multiple trees for which these two sequences can be correct –



From this, we can conclude that **to get a unique binary tree, In-order sequence is needed.** With just pre-order and post-order sequences, we can't get an **unique binary tree.**

Question

Given a BST with the following pre-order traversal sequence –

$$50, 30, 20, 10, 28, 25, 29, 35, 38, 45, 48, 80, 55, 85$$

Find the post – order traversal sequence.

Answer

The clue in the above question is that this is a sequence for a BST. This means, we have the in – order sequence which is just the nodes in ascending order –

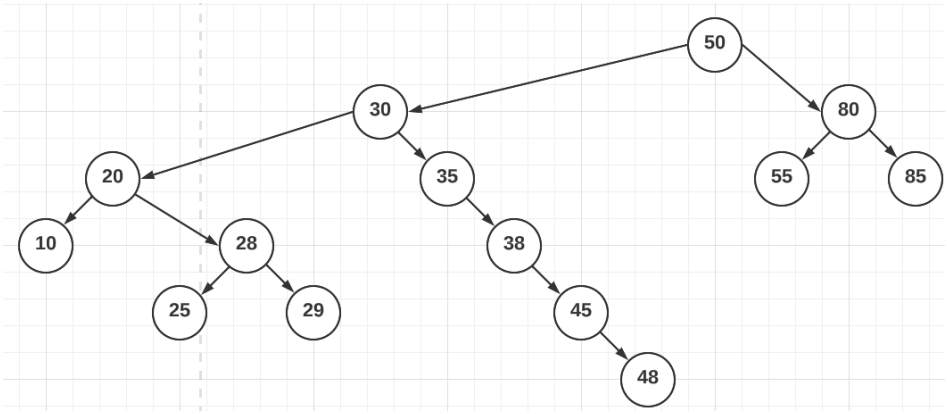
$$10, 20, 25, 28, 29, 30, 35, 38, 45, 48, 50, 55, 80, 85$$

Since we have the in – order sequence as well, this means we have a **unique solution.** From the pre-order sequence, we know that **Node 50** is the root node. Hence, from the in-order sequence, we have –

$$Left\ sub - tree = 10, 20, 25, 28, 29, 30, 35, 38, 45, 48$$

$$Right\ sub - tree = 55, 80, 85$$

Hence, the tree will be as follows –



Hence, we get –

$$Post - order = 10, 25, 29, 28, 20, 48, 45, 38, 35, 30, 55, 85, 80, 50$$

In this case, we have the time complexity of sorting $O(n \log n)$ to get the in-order sequence. Then, we need to search for the elements and that will take another $O(n \log n)$ which is because the tree is sorted now and we can apply binary search. Hence, the total time complexity will be $\boldsymbol{O(n \log n)}$**.**

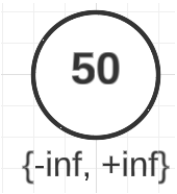
NOTE

As we saw, the time complexity comes up to $O(n \log n)$. However, we can do the same problem in the time complexity of $O(n)$ as this is a BST tree and we can perform comparisons to sort the nodes. To do so, we go as per the pre-order sequence and store two values in each node –

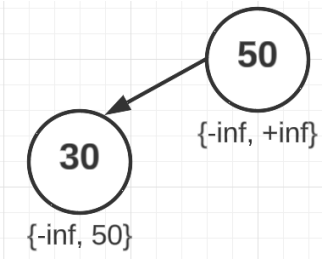
$$Node = \{max, min\}$$

We start with $min = -\infty$ and $max = \infty$ at the root node. With every left step, the max value becomes the value of the node previous to it. With every right step, the min value becomes the value of the node previous to it. Let us take the above example and understand.

We start with the root node as follows –



The next node is 30. Since this is a BST, 30 will be in the Left side of 50. Hence, there is a step to the left. So, the value of max changes to the value of the node before it.



At node 30, we can have the next value anywhere between $-\infty$ to 50. Like this, we keep building the tree with the nodes and we will end up with the BST. Since, there is a single comparison at each node, the complexity becomes $\boldsymbol{O(n)}$.

Question

Consider the following in-order sequence of min heap. Find the post – order sequence.

$$In - order = 100, 40, 110, 30, 130, 50, 120, 10, 90, 70, 80, 20, 60$$

Answer

For a min heap, the minimum element becomes the root node. So initially, the root node is 10. As per the in – order sequence, we get –

$$Left\ sub - tree = 100, 40, 110, 30, 130, 50, 120$$

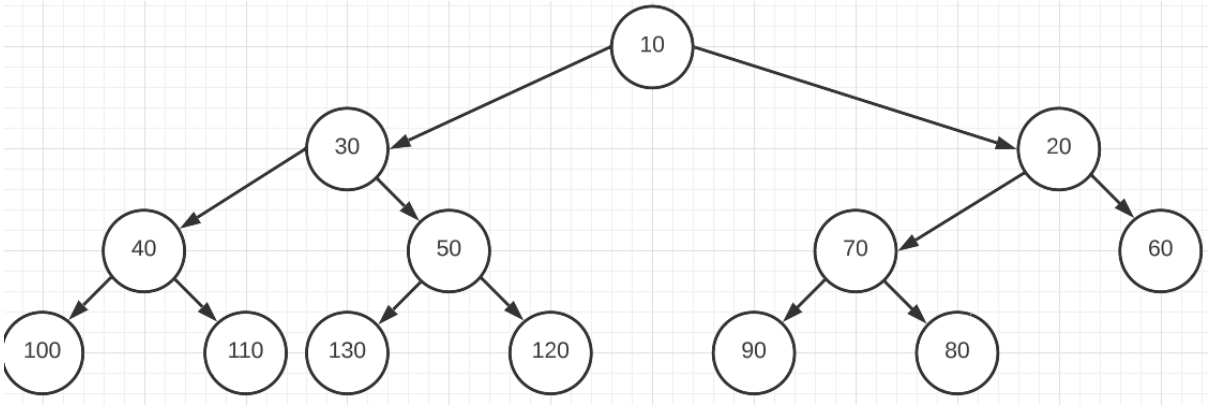
$$Right\ sub - tree = 90, 70, 80, 20, 60$$

Next, let us take the right sub-tree. In that, the root node will be 20. Thus, we have –

$$Left\ sub - tree = 90, 70, 80$$

$$Right\ sub - tree = 60$$

Like this, we continue and can form the entire tree –



NOTE

If we have a graph with n children, then the left most node is the left sub-tree and the rest of the $n - 1$ children can be considered as the right sub-tree. If a parent has a single child, then the child is considered as a left sub-tree.

STACK & QUEUE

A stack is a data structure where the insertion and deletion happen from the same side (top). In short, it is LIFO.

A queue is a data structure where the insertion happens on one side (rear) and deletion happens on the other end (front). In short, it is a FIFO.

Abstract Data type – It is the list of operations that can be performed in the data structure. For example, ADT of Queue is Enqueue() and Dequeue() while the ADT of Stack is Push() and Pop().

IMPLEMENTING QUEUE USING ARRAY

First, we set the initial values –

```
int front = -1;
int rear = -1;
int n = 10;
int q[n];
```

Now, we can write the Enqueue and the Dequeue functions –

```
void EnQueue(int d) { //Add element d at the rear of the queue
    if (rear+1 == n) { //Queue is full
        printf("Queue overflow\n");
    }
    else {
        if (rear == -1) { //First element insertion.
            front++;
            rear++;
        }
        else { //Regular insertion
            rear++;
        }

        q[rear] = d;
    }
}
```

```
int Dequeue() { //Remove element from the front
    if (front == -1) { //Queue is empty
        printf("Queue Underflow\n");
    }
    else {
        int y = q[front];
        if (front == rear) { //Last element deletion
            front = -1;
            rear = -1;
        }
        else { //Regular element deletion
            front++;
        }

        return(y);
    }
}
```

Since there are no loops in the Enqueue or the Dequeue functions, the time complexity for both the functions is $\boldsymbol{O(1)}$**.**

The above implementation has a problem. Suppose, we first use Enqueue to fill up the queue and then start deleting the elements. At this point, the front will keep incrementing with each deletion, the rear will remain constant at $n-1$. Hence, even though there are multiple deletions, the queue can no longer use Enqueue as rear value will return Queue Overflow. This is the biggest **drawback of Linear Queue.**

CIRCULAR QUEUE

To improve the queue efficiency and overcome this drawback, we can use the **Circular Queue.** For example, let us assume a queue of size 5 with each element full. At this point,

$$front = 0 \quad ; \quad rear = 4$$

Now, let us assume we perform 2 deletions. Then, it makes –

$$front = 2 \quad ; \quad rear = 4$$

As we know, there are 2 empty slots present in the queue. Hence, to access those slots, we use the following formula to update rear instead of simple incrementation –

$$rear = (rear + 1)\ \%\ n$$

For example, let us assume a queue as follows –

VALUE	10	20	30	40	50
INDEX	0	1	2	3	4

The queue is full and we have $front = 0$ and $rear = n - 1 = 5 - 1 = 4$. Now, let us perform 2 dequeues. The queue now looks like –

VALUE			30	40	50
INDEX	0	1	2	3	4

Now, we have 2 empty slots and $front = 2$. If this was a linear queue, then the queue would still show overflow message as it doesn't allow the rear to come back. However, for a circular queue –

$$rear = (rear + 1)\ \%\ n = 5\%5 = \mathbf{0}$$

Hence, the $rear$ now point to index 0. So, we can perform another enqueue –

VALUE	60		30	40	50
INDEX	0	1	2	3	4

At the same time, we need to update the condition to check if the circular queue is full. For this, there are 2 conditions –

- If $rear = n - 1$ and at the same time $front = 0$, then the queue is full.
- If the rear goes back to the beginning and then becomes equal to front, then the deleted element slots are also filled now. Hence, the queue is full

```
void Enqueue(int d) {
    if ((rear + 1 == front) || (front == 0 && rear + 1 == n)) {
        printf("Queue Overflow\n");
    }
    else {
        if (rear == -1) {
            front++;
            rear++;
        }
        else {
            rear = (rear + 1) % n;
        }

        q[rear] = d;
    }
}
```

This was the enqueue function. Similarly, we need to update the condition below for the dequeue as well –

$$front = (front + 1)\ \%\ n$$

IMPLEMENTING QUEUE WITH LINKED LIST

In the case of linked list, the enqueue operation has the following three operations –

```
p = malloc;

rear->next = p;

rear = p;
```

In short, we are creating a memory location, setting it as the next in list and then updating the $rear$ location. If the queue is full, then there is no more memory to allocate which results in p being null.

As for dequeue, we again has three conditions –

- If $front$ is NULL, then the queue is empty.
- Else, if $front = rear$, then there is only 1 element in the queue. Hence, we $free(front)$ and then set both $front$ and $rear$ to NULL.
- Else, free up the $front$ and go to the next node.

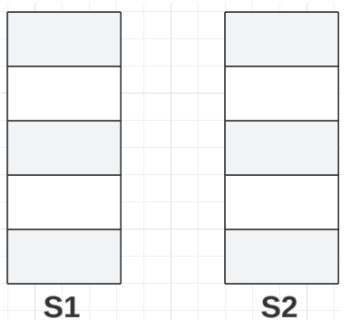
```
void Enqueue(int d) {
    struct node *p = (struct node *) malloc(sizeof(struct node));
    if (p == NULL) {
        printf("Queue Overflow\n");
    }
    else {
        p->data = d;
        p->next = NULL;
        if (rear == NULL) {
            front = p;
            rear = p;
        }
        else {
            rear->next = p;
        }
    }
}
```

```c
int Dequeue() {
    if (front == NULL) {
        printf("Queue underflow\n");
    }
    else {
        int d = front->data;
        if (front == rear) {
            free(front);
            front = NULL;
            rear = NULL;
        }
        else {
            struct node *p = front;
            front = front->next;
            free(p);
            p = NULL;
        }

        return(d);
    }
}
```

In the above case, the enqueue and the dequeue functions don't have any loops thus making the time complexity to be $\boldsymbol{O(1)}$.

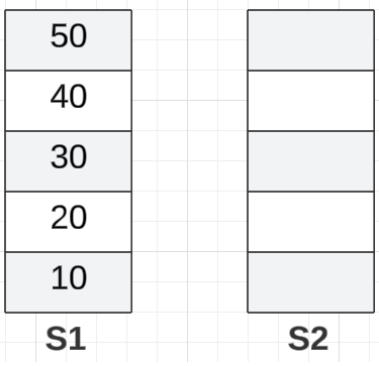
IMPLEMENTING QUEUE USING STACK

To implement queue, we need **two stacks** – mainly for dequeue operation. Let us assume two stacks as shown below –

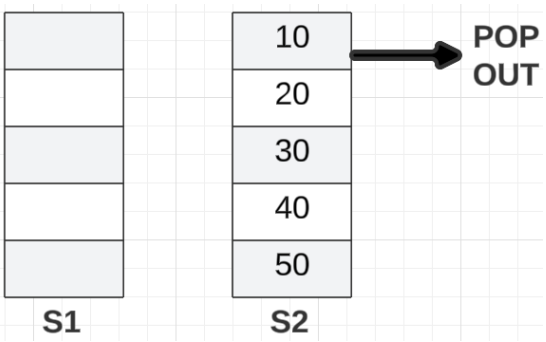


Let us now perform Enqueue. It is simple the same as pushing the element into Stack S1. Hence, we can enqueue 5 elements as follows –

$$Enqueue(x) = push(x, S1)$$



Now, if we try to Dequeue, then the problem occurs. We would expect element 10 to be deleted upon Dequeue, but pop(s1) will return 50. This is where the stack S2 comes into play. First, we pop all the elements from S1 into S2. Then, we pop from S2 to get the dequeue element.



Hence, the code can be given as follows –

```
void Enqueue(int d) {
    push(d, S1);
}

int Dequeue() {
    if (S2 != NULL) { //There are elements in S2
        return(S2.pop());
    }
    else { //S2 is empty. So we need to pop S1 and push S2
        while (S1 != NULL) {
            int x = S1.pop();
            push(x, S2);
        }
        return(S2.pop());
    }
}
```

In the case of Enqueue, we have a constant time complexity $\boldsymbol{O(1)}$**.** On the other hand, for Dequeue, we have two cases –

- First, if S2 is not null, then the dequeue operation is a simple pop. Hence, time complexity becomes $\boldsymbol{O(1)}$ for the best and average cases.
- Else, if S2 is null, then we need to pop from S1 and push into S2 and then pop from S2. This is the worst – case scenario with the time complexity of $\boldsymbol{O(n)}$**.**

NOTE

The reason we need 2 stacks in this case is because we first need to reverse the order of the elements and then pop the element out during a dequeue. Since a standard stack can't perform reverse, we need another stack and hence the time complexity becomes $\boldsymbol{O(n)}$**.** However, in one of the GATE exams, the question stated that the ADT of the stack was as follows –

- Push
- Pop
- Reverse

Hence, in this case, we can simply reverse the stack and we don't need another stack. Therefore, in this case the time complexity becomes $\boldsymbol{O(1)}$**.** So, please read the question carefully.

PRIORITY QUEUE

In these queues, the Enqueue operation remains as is, but the dequeue process happens in some priority or order –

- **Ascending –** The dequeue happens such that the result is a list of values in ascending order.
- **Descending –** The dequeue results in a list of values in the descending order.

Implementing Ascending Priority Queue using Unsorted Array

As seen above, the Enqueue is simply appending the values like a regular queue. Now for, a dequeue, we perform the following operations –

- Scan the array and find the index of the minimum element (k).
- Store index of last element (j)
- Print the element at position k
- $q[k] = q[j]$
- $j = j - 1$

This will result in an ascending order being returned. Since for a dequeue we need to scan the array, the time complexity becomes $\boldsymbol{O(n)}$ for every case.

Implementing Ascending Priority Queue using Sorted Array

In this case, whenever a new element is added using enqueue, the array needs to be sorted. So the queue will be stored as a sorted array. At the end, if we dequeue, it will always result in an ascending order.

For enqueue, the worst case scenario will be $\boldsymbol{O(n)}$ as we are sorting. However, the best case scenario will be when there is no sorting needed and the elements are enqueueing in the correct order itself. This makes the time complexity as $\boldsymbol{O(1)}$**.**

Implementing Ascending Priority Queue using Min Heap

If we use min heap, the sorting can be done with time complexity $\boldsymbol{O(\log n)}$ in the worst and average case scenario. However, in the best case there might be no need for sorting, hence making the time complexity as $\boldsymbol{O(1)}$**.**

The interesting part here is that when we dequeue, we are effectively deleting a node from the min heap. Thus, there is a need to re-sort the min heap again. Hence, both enqueue and dequeue would need to have $\boldsymbol{O(\log n)}$ complexity. Since $\log n$ is better than n, this is a better choice amongst the others.

DOUBLE ENDED QUEUE (DEQueue)

This kind of queue allows for front and rear to be decremented instead of just incrementing. That means, we can now insert using front and delete using rear. FIFO is smashed right out the bloody window. No need to worry too much as this is rarely asked in GATE.

STACK

As discussed previously, stack is LIFO and has an ADT of push() and pop(). Let us discuss its implementation.

Implementing Stack using Array

It is the same as queue implementation using array but instead of front and rear, we have just top.

```
void push(int d) {
    if (top + 1 == n) {
        printf("Stack Overflow\n");
    }
    else {
        top++;
        q[top] = d;
    }
}
```

```
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
    }
    else {
        int y = q[top];
        top--;
    }
    return(y);
}
```

In both Push() and Pop(), there are no loops and hence the time complexity becomes $\boldsymbol{O(1)}$ for every case.

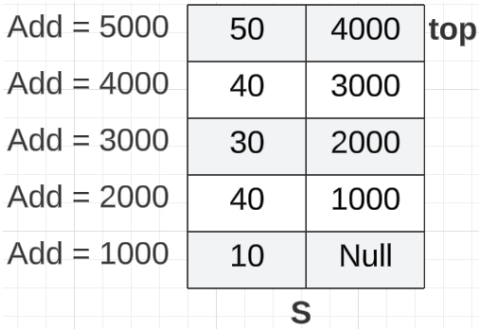
Implementing Stack using SLL

In this case, we will use malloc and struct node.

```
void push(int d) {
    struct node *p = (struct node *) malloc(sizeof(struct node));
    if (p == NULL) {
        printf("Stack Overflow\n");
    }
    else {
        p->data = d;
        p->next = top;
        top = p;
    }
}
```

```
int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
    }
    else {
        int y = top->data;
        struct node *p = top;
        top = top->next;
        free(p);
        p = NULL;
        return(y);
    }
}
```

In this case, the linking happens from top to bottom rather than bottom to top as shown below –



The reason for this is that during pop, we can use top itself. If we were linking it the other way, then we would have to traverse from Add 1000 till we find the last element. So, with this, we are restricting both the push() and pop() time complexity to $\boldsymbol{O(1)}$.

NOTE

In the push() function, we are using malloc and if there is no more memory to assign, we are printing stack overflow. However, we have discussed before that malloc assigns memory in the heap area. So technically, this is a heap overflow, but since we are implementing a Stack, we print Stack overflow.

Implementing Stack using Queue

To implement a stack, we need 2 queues to reverse the order the elements. So, let us assume there are 2 queues - q_1 and q_2. In that case –

Push operation

- Insert element in queue q_2.
- Insert the elements from q_1 into q_2
- Now, insert the elements from q_2 back into q_1. This will reverse the elements

Pop operation

- Simple dequeue operation

In this case, since we are traversing the queue with every Push or Pop, then the time complexity here becomes $\boldsymbol{O(n)}$.

Average Lifetime of Elements in Stack

To understand this concept, we need to define a few terms –

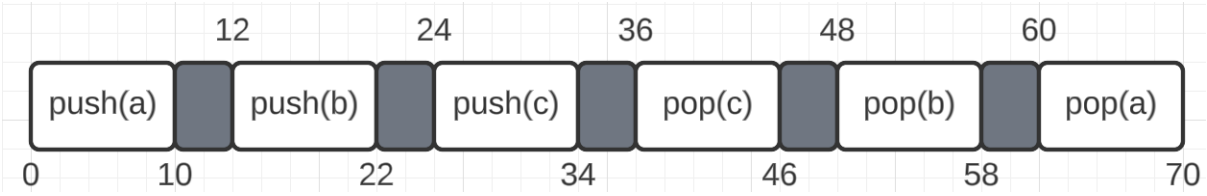
- **Operation Time –** It is the amount of time required to perform either a Push or a Pop operation. It is denoted as x
- **Elapsed Time –** It is the amount of time elapsed between any 2 stack operations. It is denoted as y.

Let us take an example where $x = 10$ and $y = 2$. The operation being performed is –

```
n = 3(a, b, c)
```

The above line is a representation of the following command – "*We are first pushing a, b and then c in sequence and then popping the three elements".*

Given this information, we can form a timeline as follows –



Now, we can define the **lifetime of an element** as follows –

$$Lifetime(a) = (time\ when\ pop(a)\ just\ began) - (time\ when\ push(a)\ was\ completed)$$

Using the above timeline, we can define –

$$Lifetime(a) = 60 - 10 = 50$$

$$Lifetime(b) = 48 - 22 = 26$$

$$Lifetime(c) = 36 - 34 = 2$$

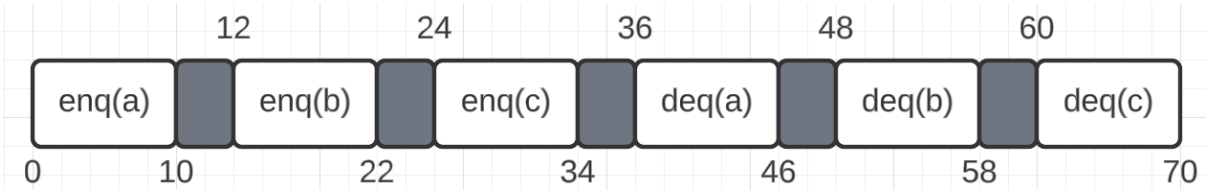
Hence, we have –

$$\boldsymbol{Avg.\ Lifetime} = \frac{Sum\ of\ lifetimes}{No\ of\ elements} = \frac{50 + 26 + 2}{3} = \mathbf{26}$$

In general, we can write a formula for the average lifetime as follows –

$$Avg.\ Lifetime = n(x + y) - x$$

Now, suppose the above example is the same except this time we are talking about a queue rather than a stack –



In this case, we have –

$$Lifetime(a) = 36 - 10 = 26$$

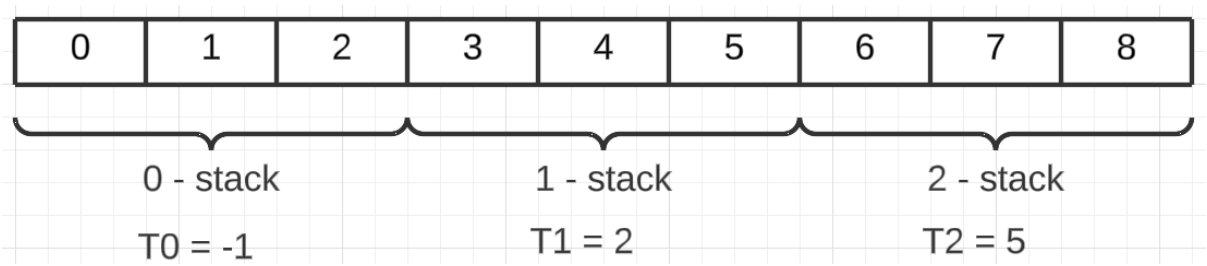
$$Lifetime(b) = 48 - 22 = 26$$

$$Lifetime(c) = 60 - 34 = 26$$

In this case as well, the **average lifetime** is still the same as 26. However, **for a queue, the lifetimes of the elements remain the same.**

Implementing multiple stacks in a single array

Basically, we are tyring to split an array into multiple continuous stacks. For example, let us assume an array of length 9. We can use this array to have 3 stacks with equal size as follows –



As we can see, for each stack we have a separate top pointer. So now, in 1 array itself we are able to implement multiple stacks.

Suppose we have an array of size n and each stack to be implemented is of size m, then –

$$No\ of\ stacks = \frac{n}{m}$$

$$Top\ element\ of\ x^{th}\ stack = \left(x * \frac{n}{m}\right) - 1$$

For the above example, $n = 9$ and $m = 3$. Using the above formula, we can see –

$$T_0 = \left(0 * \frac{9}{3}\right) - 1 = 0$$

$$T_1 = \left(1 * \frac{9}{3}\right) - 1 = 2$$

$$T_2 = \left(2 * \frac{9}{3}\right) - 1 = 5$$

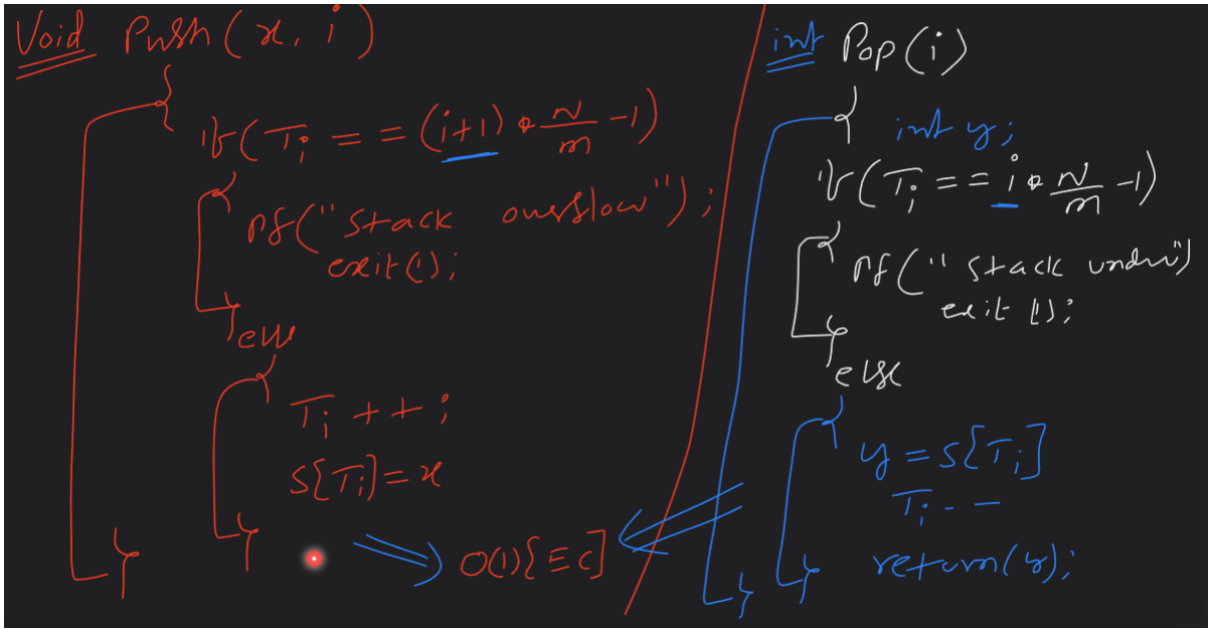
In short,

$$if\ \left(T_x == \left[\left(x * \frac{n}{m}\right) - 1\right]\right) \qquad Underflow$$

$$if\ \left(T_x == \left[\left((x+1) * \frac{n}{m}\right) - 1\right]\right) \qquad Overflow$$

In the above example, if $T_2 == 5$, then the 2nd stack is empty. On the other hand, if $T_2 == 8$, then the 2nd stack is full.

NOTE – When implementing multiple stacks in an array, the push() and pop() operations need an additional argument to mention the stack number being addressed.



The advantage of this scenario is that we can have multiple users use stacks for different applications at the same time. However, since we are dividing memory, there is a space restriction and a user can't use any other user's memory even though it might be free.

Implementing multiple stacks in a single array EFFECTIVELY

In this case, we have an array in which one stack starts at the beginning and then grows to the right while the other stack starts at the end and grows to the left. So eventually, the two stacks meet up with each other which is called the **critical area.** One of the stacks will set a **lock** variable to let the other stack know that they are going to use the critical area.

In the end, the array full condition can be checked using the following condition –

$$if\ (T_R == T_L + 1)$$

If $T_L == -1$, then the left stack is empty. Also, if $T_R == n - 1$, then the right stack is empty.

APPLICATIONS OF STACKS

- Recursion
  - Tail Recursion
  - Non – tail recursion
  - Indirect recursion
  - Nested recursion
- Conversions
  - Prefix to Postfix
  - Infix to Postfix
  - Postfix evaluation
- Towers of Hanai
- Fibonacci series

RECURSION

It is the program where a function calls itself. As mentioned above, it can be of various types –

- **Tailed Recursion –** In this type of recursion program, the function is called at the last line of the program. There are no more code lines after the function call. It has a drawback as it uses a lot of stack space due to repeated function calls. At the same time, it is quite easy to covert the recursive program to a non – recursive program (using loop) that doesn't use a lot of stack space.
- **Non-tail Recursion –** In this type of recursion program, there are some code lines executed after the recursive function call.
- **Indirect Recursion –** In this type of recursion, a function calls another function which in turn calls the first function again.
- **Nested Recursion –** When the function is passed as a parameter in the recursion.

For a given recursive program, if we are able to write an equivalent non – recursive program and are also able to save up on stack space, then it is a **Tail Recursive program.** On the other hand, for a non – tail recursive program, stack space is required when converted to a non – recursive program.

INFIX, POSTFIX and PREFIX

In Infix, the operator is **in-between** the operands. In prefix, the operator is **pre-operand.** Finally, in postfix the operator is **post** the operands.

$$Infix: a + b$$

$$Prefix: +ab$$

$$Postfix: ab +$$

No matter which notation, the order of the operands are the same i.e. first comes a and then comes b. Humans prefer the infix notation but the system prefers the postfix notation. To covert between notations, one must make groups based on priority and then perform the conversion for each group.

Question

Convert the following infix expressions to postfix and prefix –

$$a + b * c$$

$$a + b - c$$

Answer

In the first expression, we have –

$$a + b * c = a + (b * c)$$

Infix to Postfix

$$a + (b * c) = a + (bc *) = \boldsymbol{abc} * +$$

Infix to Prefix

$$a + (b * c) = a + (* bc) = +a * bc$$

In the second expression, we have –

$$a + b - c = (a + b) - c$$

Infix to Postfix

$$(a + b) - c = (ab +) - c = \boldsymbol{ab} + \boldsymbol{c} -$$

Infix to Prefix

$$(a + b) - c = (+ab) - c = - + \boldsymbol{abc}$$

Infix to Postfix Algorithm

As we can see above, the manual method can work for simple expressions. However, when the expressions become more and more complex, there is a need for algorithm –

1. Start with an empty stack
2. If **operand** comes, then directly print it
3. Else if **operand** comes, push it to the stack
    a. If the incoming operator has a **higher** priority compared to the top of the stack, then push the incoming operator on to the stack.
    b. If the incoming operator has a **lower or equal** priority compared to the top of the stack, then pop the top of the stack and push the incoming operator onto the stack.

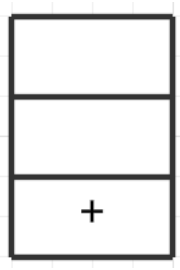
Question

Convert the infix to postfix expression –

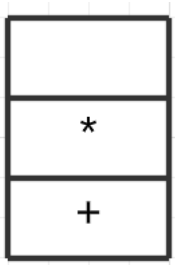
```
a + b * c - d / e
```

Answer

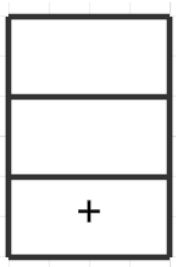
First, we have the operand a, so we print a first. Next, we have the $+$ operator. Since the stack is empty, we push the operator to the stack.



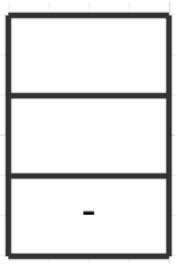
Next, we print the operand b. After that, we encounter $*$ operator. Since the incoming operator has a higher priority as the top of the stack, the incoming operator gets pushed onto the stack.



Next, we print operand c. After that, we get operator $-$. Since the incoming operator has a lower priority as compared to the top of the stack, the top of the stack gets popped.



Again, the incoming operator has equal priority to the top of the stack. So the top of the stack gets popped and finally the incoming operator gets pushed onto the stack.



Next, we print the operand d. After that, we push / operator onto the stack and print operand e. Finally, we pop out the operator stack. The final postfix expression is –

abc*+de/-

As we can see, this conversion took **2 stack units.**

Question

Infix to postfix –

```
a = b = c = d = e
```

Answer

Here, we have only one operator which is the assignment operator. Since they are on the same priority level, we need to check the associativity. For the assignment operator, the associativity goes from right to left. Hence, the rightmost operator has the highest priority and the leftmost operator has the lowest priority. In short, the operators will keep getting pushed onto the stack when we parse the expression from left to right. Thus, the final postfix expression becomes –

abcd====

The above conversion takes **4 stack units.**

NOTE

The maximum (worst – case) number of stack units used in this algorithm will be equal to the number of operators in the expression. Additionally, we are parsing through the expression once during this conversion. Hence, this algorithm has a time complexity of $\boldsymbol{O(n)}$**.**

Question

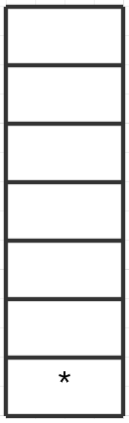
Infix to postfix –

```
a * (b – c) / (d + e * (g = h = i)) + j
```

Answer

In this case, we shall encounter parenthesis operator in the operator stack. Actually, the parenthesis are not printed in the final postfix expression and hence, the parenthesis act as a boundary in the stack. Let us take this question.

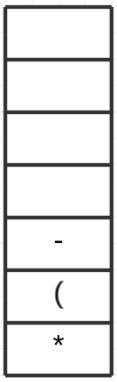
First, we print operand a and add $*$ to the operator stack.



Now, we are pushing the parenthesis to the stack. When that happens, the open parenthesis acts as a partition. Anything before that is no longer considered and when a closed parenthesis is encountered, then every operator between the two parenthesis is popped out.

Now, we perform the following steps –

- Print operand b
- Push operator –
- Print operand c



Now, we encounter a closed parenthesis. This means that all the operators between the two parenthesis will be popped out.

We continue with the same process for the rest of the expression and end up with the following postfix expression –

abc-*deghi==*+/j+

Prefix to Postfix

This is a simple process –

- First parse the prefix expression.
- When an operator comes, check its immediate neighbours for the operands.
- If the immediate 2 neighbours are operands, then shift the operator to the post of the operands.
- Else, keep looking forward.

For example,

$$* ab => ab *$$

Question

```
*-cd/ef
```

Answer

In this expression, first we encounter *. The immediate 2 elements are – and c. Since we have now encountered another operator —, we need to look again for the 2 operands. We get –

$$-cd => cd-$$

Next, we get –

$$/ef => ef/$$

Finally, we get the postfix expression as –

cd-ef/*

Question

```
*-+*-/-+a+b-cdefgh*ijkl
```

Answer

abcd-++e_f/g-h*ij*+k-l*

Question

```
*a/b-c**+-ghi/de+fh
```

Answer

abcgh-i+de/*fh+*-/*

Postfix Evaluation Algorithm

1. If an operand is encountered, push them in the stack
2. When an operator is encountered (let's say op), then do the following –
    a. `Opd2 = stack.pop()`
    b. `Opd1 = stack.pop()`
    c. `Result = Opd1 op Opd2`
    d. `stack.push(Result)`
3. Finally, the top of the stack is the result.

For example, take the postfix expression below –

```
234*+51/2*-30+
```

Initially, we push the operands 2, 3 and 4 into the stack –

Next, we encounter operator $op = *$. So, we follow the algorithm –

```
Opd2 = 4;
Opd1 = 3;
Result = 3*4 = 12;
Stack.push(Result);
```

So now, the stack looks as follows –

We continue this down the expression and the final result will be **34.**

NOTE

Since we are parsing through the expression just once, the time complexity will be $\boldsymbol{O(n)}$**.**

Prefix/Postfix to Infix

In infix expression, the operator comes in between the operands. For example, if we have the prefix expression as –

```
*-ab+cd
```

We can write the infix expression as follows –

```
a-b * c+d
```

However, in the infix expression the * operation will happen first as per priority but as per the prefix expression, we need the – and + to happen before the *. So, when converting to infix, make sure of putting the parenthesis –

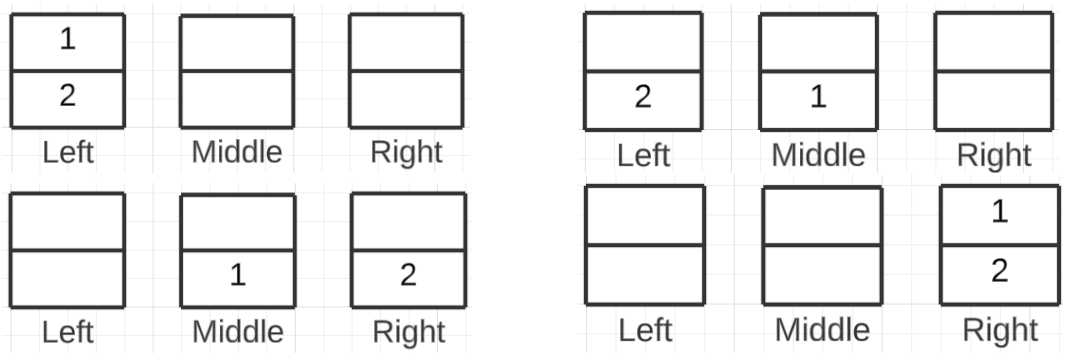
```
(a-b) * (c+d)
```

TOWERS OF HANAI

This is a classic problem that is solved using stacks. Imagine that there are three towers – Left, Middle and Right. Now, we have a bunch of discs that are all stacked up in Left Tower. The goal is to move the discs from the Left tower to the Right Tower **one disc at a time** and **result should be in the same order**.

For example, let us assume there are two discs in Left tower – Disc 1 and Disc 2. To move the discs from Left tower to the Right tower, we perform 3 moves –

- Move Disc 1 from Left to Middle
- Move Disc 2 from Left to Right
- Move Disc 1 from Middle to Right



As we can see, we would need **3 moves** to move 2 nodes. Now, let us assume there are 3 discs – Disc 1, Disc 2 and Disc 3. In this case, the process will be as follows –

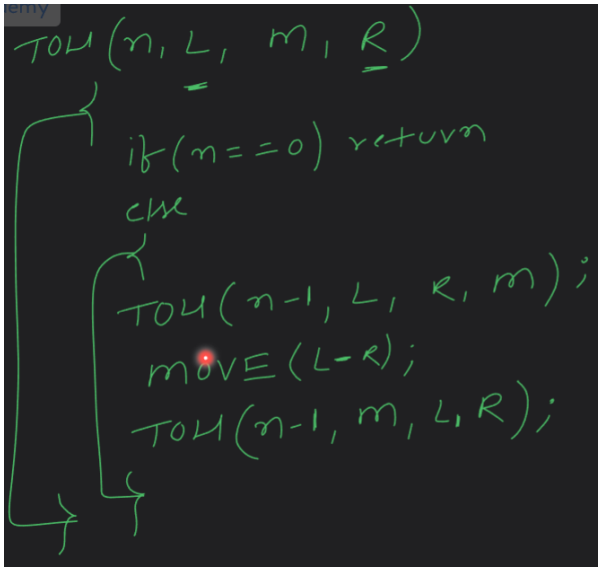
- Disc 1 and Disc 2 are sent from Left to Middle. As seen above, moving 2 nodes from one tower to another takes **3 moves.**
- Disc 3 is moved from Left to Right. This takes **1 move.**
- Disc 1 and 2 are moved from Middle to Right. This again takes **3 moves.**

Hence, for moving 3 nodes we require **7 moves.** Now, we can define a function as follows –

$$TOH(n, src, inter, dest)$$

Where,

- n is the number of discs
- src is the source tower
- $inter$ is the intermediate tower
- $dest$ is the destination tower



Hence, using recursion (aka stack) makes the towers of hanai problem a much easier solution. The above solution has the time complexity as $\boldsymbol{O(2^n)}$ and the space complexity of $\boldsymbol{O(n)}$**.**

NOTE

In every programming language, the function calling occurs in a **pre order fashion** but the function execution occurs in the **post order fashion.** Also, in our execution of the TOH function, we first call the function $n + 1$ times before making a move. Additionally, this is a **tail recursion** as after the second function call, there are no other statements.

We also have the following observation when it comes to number of moves –

n	No of moves	No of func calls
0	$0 = 2^0 - 1$	$1 = 2^1 - 1$
1	$1 = 2^1 - 1$	$3 = 2^2 - 1$
2	$3 = 2^2 - 1$	$7 = 2^3 - 1$
3	$7 = 2^3 - 1$	$15 = 2^4 - 1$
4	$15 = 2^4 - 1$	$31 = 2^5 - 1$
5	$31 = 2^5 - 1$	$63 = 2^6 - 1$

In short, we have –

$$No\ of\ moves\ for\ n\ nodes = \mathbf{2^n - 1}$$

$$No\ of\ fucntion\ calls\ for\ n\ nodes = \mathbf{2^{n+1} - 1}$$

Suppose that a hash table of m slots contains a single element with key k and the rest of the slots are empty. Suppose further we search r times in the table for various other keys not equal to k. assuming simple uniform hashing, what is the probability that one of the r searches probes the slot containing the single element stored in the table?

Select your answer

- ✗ r/m
- B $(1-1/m)^r$
- C $(1/m)^r$
- D $1- (1-1/m)^r$

Which of the following is not a factor which affects the efficiency of lookup operations in a hash table?

Select your answer

- A Number of elements stored in the hash table.
- B Size of elements stored in the hash table.
- C Number of buckets present in the hash table.
- ✗ Quality of hash function.

Question 15

Which of the following is the best choice of hash table size for storing 200 elements in a hash table?

Select your answer

- ✗ 200
- B 228
- C 263
- D 251

Question 21

Consider a hash table with m slots that uses chaining for collision resolution. The table is initially empty. What is the probability that after 10 keys are inserted that at least a chain of 9 is created?

Select your answer

A $\frac{1}{m^8}$

✗ $\frac{m-1}{m^9}$

C $\frac{10m-9}{m^9}$

D $\frac{3}{m}$

Question 22

Suppose that the following keys are inserted in some order into an initially empty linear-probing hash table.

Key	Hash
B	1
E	5
M	6
N	0
O	1
T	6
Y	4

Following is the resultant hash table:

0	1	2	3	4	5	6
T	N	O	B	Y	E	M

If the initial size of the hash table was 7, which of the following keys is definitely not the last key inserted? (Assume that the hash table did not grow or shrink)

Select your answer

A B

B Y

C M

D E

Which type of object lifetimes defines the scope of a static variable during the run-time?

Select your answer

✓ Static Lifetime

B Dynamic Lifetime

C Automatic Lifetime

D None of the Above

Question

Find the output of the following program

```c
#include <stdio.h>
#define x 4+2
int main()
{
    int i;
    i = x*x*x;
    printf("%d", i);
}
```

Answer

In this case, x will literally be substituted by $4 + 2$. Hence, we have –

$$i = 4 + 2 * 4 + 2 * 4 + 2$$

Using operator precedence, we get $\boldsymbol{i = 22.}$

Question

Find the output of the following program

```c
#include <stdio.h>
#define x 2
int main()
{
    const x = 10;
    printf("%d", x);
}
```

Answer

In this case, the output will be a **Compilation Error.**

Question

```c
#include <stdio.h>
int main()
{
    printf("%f\n", 4.89);
    printf("%0.0f\n", 4.89);
    printf("%0.1f\n", 4.89);
    printf("%2.0f\n", 4.89);
    printf("%0.2f\n", 4.89);
}
```

Answer

```
4.890000
5
4.9
 5
4.89
```

Question

```c
#include <stdio.h>
void india();
int main()
{
    void (*fun)();
    fun = india;
    (*fun)();
    fun();
    return 0;
}

void india()
{
    printf("Dhoni");
}
```

Answer

$fun()$ is assigned to call $india()$. So two lines will execute function $india()$ –

- `(*fun)()`
- `fun()`

Hence, $india()$ executes **twice** and the final output will be **DhoniDhoni.**