Data Structure

Contents

N.B:

The Algorithm and steps noted here are for better understanding. Only practice the application of these algorithms for exam.

STRING PROCESSING

Initialization:

Method 1:

Character st1*10, st2*14

St1 = "The end"

St2 = "To be or not to be"

Then st1 and st2 will appear in memory as follows:

St1:

T	h	e		e	n	d			

St2:

T	o		b	e		o	r		n	o	t		t

Method 2:

Character st1, st2

St1 = "The end"

St2 = "To be or not to be";

Then st1 and st2 will appear in memory as follows:

St1:

T	h	e		e	n	d

St2:

T	o		b	e		o	r		n	o	t		t	o		b	e

SUBSTRING

Substring (string, initial, length)

SUBSTRING ('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'

SUBSTRING ('THE END', 4, 4) = ' END'

INDEXING

Index (text, pattern)

T = 'HIS FATHER IS THE PROFESSOR'

Index (T, 'THE) = 7

Index (T, 'THEN') = 0

Index (T, ' THE ') = 14

LENGTH

LENGTH (text)

T = 'HIS FATHER IS THE PROFESSOR'

Length (T) = 27

Concatenation

S1 = "MARK", S2 = "TWAIN"

The concatenation of S1 and S2 is denoted by S1//S2.

S1//S2 = "MARKTWAIN'

S1//' ▭ '//S2 = "MARK TWAIN'

Insertion

INSERT (text, position, string)

INSERT ('ABCDEFG', 3, 'XYZ') = 'ABXYZCDEFG'

INSERT (T, K, S) = SUBSTRING (T, 1, K – 1) //S// SUBSTRING (T, K, LENGTH(T) – K + 1)

DELETE

DELETE (text, position, length)

DELETE ('ABCDEFG', 4, 2) = 'ABCFG'

DELETE (T, K, L) = SUBSTRING (T, 1, K – 1) // SUBSTRING (T, K + L, LENGTH(T) – (K + L – 1))

DELETE (When text and pattern are given)

DELETE (T, INDEX (T, P), LENGTH (P))

T = 'ABCDEFG', P = 'CD'

INDEX (T, P) = 3 and LENGTH (P) = 2

So, DELETE (T, 3, 2) = 'ABEFG'

Algorithm:

A text T and pattern P are in memory. This algorithm deletes every occurrence of P in T.

1) Find index of P, Set K = INDEX (T, P)
2) Repeat while K ≠ 0:
   a) Delete P from T, Set T = DELETE (T, INDEX (T, P), LENGTH (P))
   b) Update index, Set K = INDEX (T, P)

   End of loop.
3) Write T
4) Exit

REPLACEMENT

REPLACE (text, pattern1, pattern2)

REPLACE ('XABYABZ', 'AB', 'C') = 'XCYABZ'

Algorithm:

A text T and patterns P and Q are in memory. This algorithm replaces every occurrence of P in T by Q.

1) Find index of P, Set K = INDEX (T, P)
2) Repeat while K ≠ 0:
   a) Replace P by Q: Set T = REPLACE (T, P, Q)
   b) Update index, Set K = INDEX (T, P)

   End of loop

3) Write T
4) Exit

Arrays, Records and Pointers

Bubble Sort

Algorithm:

(Bubble Sort) BUBBLE (DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N – 1.

2. Set PTR:= 1. [Initializes pass pointer PTR.]

3. Repeat while PTR ≤ N – K: [Executes pass.]

    (a) If DATA[PTR] < DATA [PTR + 1], then:

    Interchange DATA[PTR] and DATA [PTR + 1].

    [End of If structure.]

    (b) Set PTR:= PTR + 1.

    [End of inner loop.]

[End of Step 1 outer loop.]

4. Exit.

Process:

Suppose the list of numbers A [l], A [2], …, A[N] is in memory. The bubble sort algorithm works as follows:

Step 1. Compare A [1] and A [2] and arrange them in the desired order, so that A[l] < A [2]. Then compare A [2] and A [3] and arrange them so that A [2] < A [3]. Then compare A [3] and A [4] and arrange them so that A [3] < A [4]. Continue until we compare A [N – 1] with A[N] and arrange them so that A [N – 1] < A[N].

Observe that Step 1 involves n – 1 comparison. (During Step 1, the largest element is "Bubbled up" to the nth position or "sinks" to the nth position.) When Step 1 is completed, A[N] will contain the largest element.

Step 2. Repeat Step 1 with one less comparison; that is, now we stop after we compare and possibly rearrange A [N – 2] and A[N – 1]. (Step 2 involves N – 2 comparisons and, when Step 2 is completed, the second largest element will occupy A [N – 1].)

Step 3. Repeat Step 1 with two fewer comparisons; that is, we stop after we compare and possibly rearrange A [N – 3] and A [N – 2]. Step N – 1. Compare A[l] with A [2] and arrange them so that A [l] < A [2]. After n – 1 step, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass," so each of the above steps is called a pass. Accordingly, the bubble sort algorithm requires n – 1 pass, where n is the number of input items.

Time Complexity: O (n^2).

Example:

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57
Apply bubble sort to the array.

Solution:

Pass 1:

Comparison 1: 32 < 51, so no change.

Comparison 2: 51 > 27, so 32, **27, 51**, 85, 66, 23, 13, 57.

Comparison 3: 51 < 85, so no change.

Comparison 4: 85 > 66, so 32, 27, 51, **66, 85**, 23, 13, 57.

Comparison 5: 85 > 23, so 32, 27, 51, 66, **23, 85**, 13, 57.

Comparison 6: 85 > 13, so 32, 27, 51, 66, 23, **13, 85,** 57.

Comparison 7: 85 > 57, so 32, 27, 51, 66, 23, 13, **57, 85**.

Pass 2:

Comparison 1: 32 > 27, so **27, 32**, 51, 66, 23, 85, 13, 57.

Comparison 2: 32 < 51, so no change.

Comparison 3: 51 < 66, so no change.

Comparison 4: 66 > 23, so 27, 32, 51, **23, 66**, 85, 13, 57.

Comparison 5: 66 < 85, so no change.

Comparison 6: 85 > 13, so 27, 32, 51, 23, 66, **13, 85**, 57.

Comparison 7: 85 > 57, so 27, 32, 51, 23, 66, 13, **57, 85**.

Pass 3:

Comparison 1: 27 < 32, so no change.

Comparison 2: 32 < 51, so no change.

Comparison 3: 51 > 23, so 27, 32, **23, 51,** 66, 13, 57, 85.

Comparison 4: 51 < 66, so no change.

Comparison 5: 66 > 13, so 27, 32, 23, 51, **13, 66**, 57, 85.

Comparison 6: 66 > 57, so 27, 32, 23, 51, 13, **57, 66**, 85.

Comparison 7: 66 < 85, so no change.

Pass 4:

Comparison 1: 27 < 32, so no change.

Comparison 2: 32 > 23, so 27, **23, 32**, 51, 13, 57, 66, 85.

Comparison 3: 32 < 51, so no change.

Comparison 4: 51 > 13, so 27, 23, 32, **13, 51**, 57, 66, 85.

Comparison 5: 51 < 57, so no change.

Comparison 6: 57 < 66, so no change.

Comparison 7: 66 < 85, so no change.

Pass 5:

Comparison 1: 27 > 23, so **23, 27**, 32, 13, 51, 57, 66, 85.

Comparison 2: 27 < 32, so no change.

Comparison 3: 32 > 13, so 23, 27, **13, 32**, 51, 57, 66, 85.

Comparison 4: 32 < 51, so no change.

Comparison 5: 51 < 57, so no change.

Comparison 6: 57 < 66, so no change.

Comparison 7: 66 < 85, so no change.

Pass 6:

Comparison 1: 23 < 27, so no change.

Comparison 2: 27 > 13, so 23, **13, 27**, 32, 51, 57, 66, 85.

Pass 7:

Comparison 1: 23 > 13, so **13, 23,** 27, 32, 51, 57, 66, 85.

Since the list has 8 elements; it is sorted after the seventh pass.

Linear Search

Algorithm:

(Linear Search) LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA [N + 1] := ITEM.

2. [Initialize counter.] Set LOC := 1.

3. [Search for ITEM.]

Repeat while DATA[LOC] ≠ ITEM: Set LOC := LOC + 1.

[End of loop.]

4. [Successful?] If LOC = N + 1, then: Set LOC := 0.

5. Exit.

Time Complexity: O (n).

Binary Search

Algorithm:

(Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]

Set BEG := LB, END := UB and MID = INT((BEG + END)/2).

2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.

3. If ITEM < DATA[MID], then: Set END := MID – 1.

Else: Set BEG := MID + 1.

[End of If structure.]

4. Set MID := INT((BEG + END)/2).

[End of Step 2 loop.]

5. If DATA[MID] = ITEM, then: Set LOC := MID.

Else: Set LOC := NULL.

[End of If structure.]

6. Exit.

Time Complexity: O (log n).

Example: Let DATA be the following sorted 13-element array:

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99.

Find ITEM = 85 by following binary search.

Solution:

Step 1: Beg = 1, End = 13, Mid = (1 + 13) / 2 = 7, Data [Mid] = 55

Step 2: As 85 > 55, Beg = Mid + 1 = 7+1 = 8, End = 13, Mid = (8+13)/2 = 10, Data [Mid] = 77.

Step 3: As 85 > 77, Beg = 10 + 1 = 11, End = 13, Mid = (11+13)/2 = 12, Data [Mid] = 88.

Step 4: As 85 < 88, Beg = 11, End = Mid – 1 = 12 – 1 = 11, Mid = (11+11)/2 = 11, Data [Mid] = 80.

Step 5: As 85 > 80, Beg = Mid+1 = 11 + 1 = 12, End = 11.

Here, Beg > End. So, 85 is not found in the array.

Limitations of the Binary Search Algorithm:

Since the binary search algorithm is very efficient (e.g., it requires only about 20 comparisons with an initial list of 1 000 000 elements), why would one want to use any other search algorithm?

ANSWER: Observe that the algorithm requires two conditions: (1) the list must be sorted and (2) one must have direct access to the middle element in any sub list. This means that one must essentially use a sorted array to hold the data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. Accordingly, in such situations, one may use a different data structure, such as a linked list or a binary search tree, to store the data.

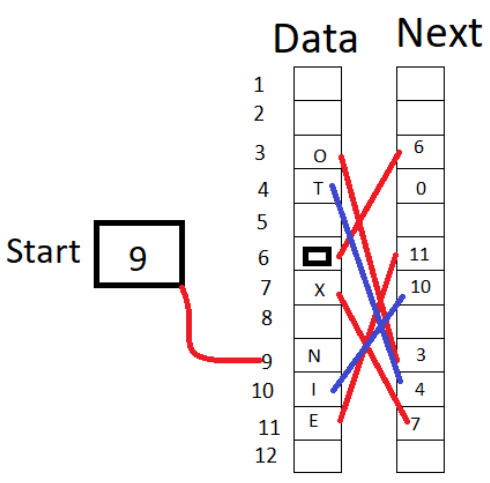
Linked List

Differences between array and Linked list:

Linear Array	Linked List
• Deletion and Insertions are difficult.	• Deletion and Insertions can be done easily.
• For insertion and deletion, it needs movements	• For insertion and deletion, it does not require movement of nodes
• In it space is wasted	• In it space is not wasted
• It is expensive	• It is not expensive
• It cannot be reduced or extended according to requirements	• It can be reduced or extended according to requirements
• To avail each element same amount of time is required.	• To avail each element different amount of time is required.
• In consecutive memory locations elements are stored.	• Elements may or may not be stored in consecutive memory locations
• We can reach there directly if we have to go to a particular element	• To reach a particular node, you need to go through all those nodes that come before that node.

- **Why Linked List is Better?**
  - ➔ It is relatively expensive to insert and delete elements in an array. Since an array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. On the other hand, in a linked list in memory each element contains a link or pointer, which contains the address of the next element. Thus, successive elements in the list need not occupy adjacent space in memory. This will make it easier to insert or delete elements in the list.

- **Representation of linked list in memory:**



Here, Start = 9: Data [9] = N is the 1st character.

Next [9] = 3: Data [3] = O is the 2nd character.

Next [3] = 6: Data [6] = □ is the 3rd character.

Next [6] = 11: Data [11] = E is the 4th

Next [11] = 7: Data [7] = X is the 5th

Next [7] = 10: Data [10] = I is the 6th

Next [10] = 4: Data [4] = T is the 7th

Next [4] = 0, the null value, so the list has ended.

• Traversing a Linked List:

Time Complexity: O (n)

Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1) Set PTR = START
2) While PTR ≠ NULL
    a) Apply PROCESS to INFO [PTR]
    b) Set PTR = LINK [PTR] (PTR now points to the next node)

    End of loop
3) Exit.

• Print

PRINT (INFO, LINK, START)

This procedure prints the information at each node of the list

1) Set PTR = START
2) While PTR ≠ NULL
    a) Write INFO [PTR]
    b) Set PTR = LINK [PTR] (Updates pointer)

    End of loop
3) Exit

• Count:

Time Complexity: O (n)

COUNT (INFO, LINK, START, NUM)

1) Set NUM = 0 (Initializes counter)
2) Set PTR = START (Initializes pointer)
3) While PTR ≠ NULL
    c) Set NUM = NUM + 1
    d) Set PTR = LINK [PTR] (Updates pointer)

    End of loop
4) Return NUM

Searching

Time Complexity: O (n)

Unsorted List:

SEARCH (INFO, LINK, START, ITEM, LOC)

LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1) Set PTR = START
2) While PTR ≠ NULL
    a) If ITEM = INFO [PTR], then:
        Set LOC = PTR, and exit.

b) Else, Set PTR = LINK [PTR] (Updates pointer)

End of loop

3) (Search is unsuccessful) Set LOC = NULL
4) Exit

Sorted List:

SEARCH (INFO, LINK, START, ITEM, LOC)

LIST is a sorted list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1) Set PTR = START
2) While PTR ≠ NULL
   If ITEM = INFO [PTR], then:
   Set LOC = PTR, and exit.
   Else If ITEM < INFO [PTR],
   Set PTR = LINK [PTR] (Updates pointer)
   Else: Set LOC = NULL and exit.

   End of loop
3) Set LOC = NULL
4) Exit

Overflow and Underflow:

Overflow will occur when AVAIL = NULL and there is an insertion.

Underflow will occur when START = NULL and there is a deletion.

Insertion

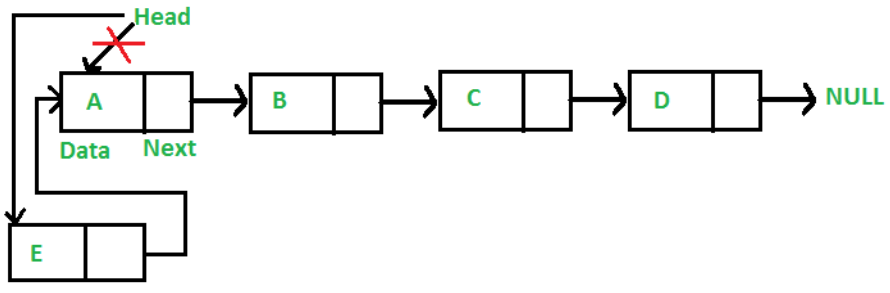
- **Inserting at the beginning of the list:**

Time Complexity: O (1)

INSFIRST (INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts ITEM as the first node in the list.

1) If AVAIL = NULL, Write: OVERFLOW and Exit.
2) (Remove first node from AVAIL list)
   Set NEW = AVAIL and AVAIL = LINK [AVAIL]
3) Set INFO [NEW] = ITEM (copies new data into new node)
4) Set LINK [NEW] = START (new node now points to original first node)
5) Set START = NEW (changes START so it points to the new node)
6) Exit



- **Inserting after a given node:**

Time Complexity: O (n)

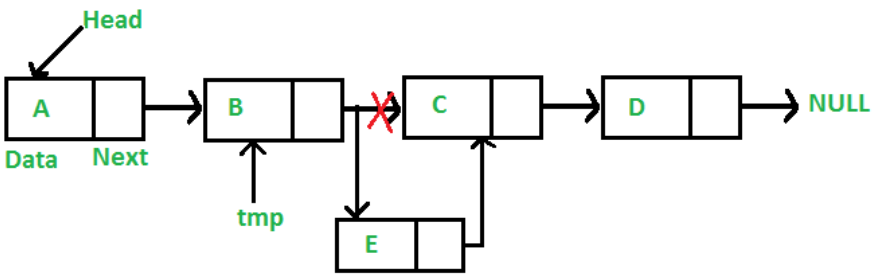
INSFIRST (INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

1) If AVAIL = NULL, Write: OVERFLOW and Exit.
2) (Remove first node from AVAIL list)
   Set NEW = AVAIL and AVAIL = LINK [AVAIL]
3) Set INFO [NEW] = ITEM (copies new data into new node)
4) If LOC = NULL
   a) Set LINK [NEW] = START (new node now points to original first node)
   b) Set START = NEW (changes START so it points to the new node)

   Else:

   a) Set LINK [NEW] = LINK [LOC]
   b) LINK [LOC] = NEW
5) Exit



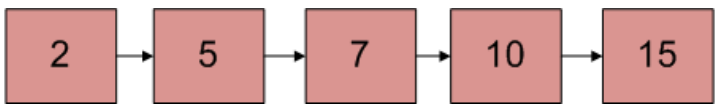
- **Inserting into a sorted linked list:**

Time Complexity: O (n)

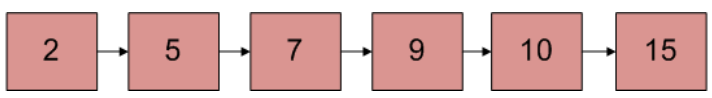
Space Complexity: O (1)

- **Given a sorted linked list and a value to insert, write a function to insert the value in a sorted way.**

Initial Linked List:



Linked List after insertion of 9:



Algorithm:

1) If Linked list is empty then make the node as head and return it.

2) If the value of the node to be inserted is smaller than the value of the head node, then insert the node at the start and make it head.

3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted. To find the appropriate node start from the head, keep moving until you reach a node GN (10 in the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).

4) Insert the node (9) after the appropriate node (7) found in step 3.

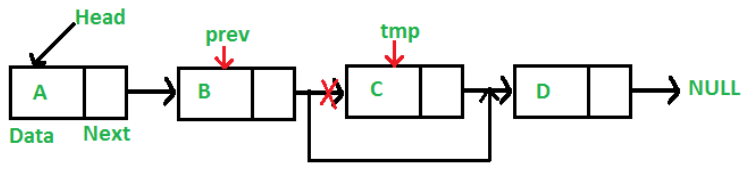
Deleting

To delete a node from the linked list, we need to do the following steps.

1) Find the previous node of the node to be deleted.

2) Change the next of the previous node and set prev_node->next = node_to_delete->next.

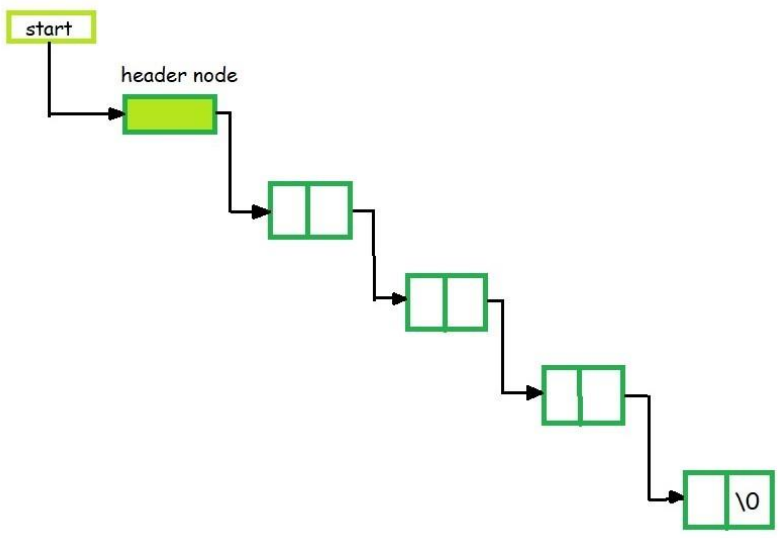
3) Free memory for the node to be deleted.



Header Linked List

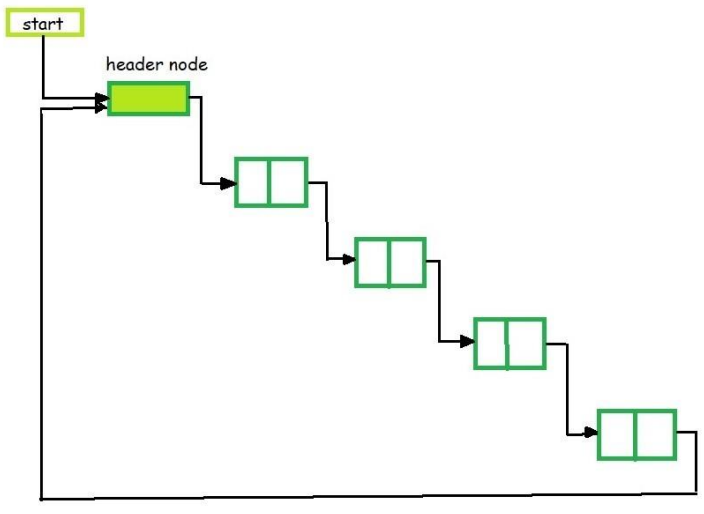
1) Grounded Header Linked List

It is a list whose last node contains the NULL pointer. In the header linked list the start pointer always points to the header node. start -> next = NULL indicates that the grounded header linked list is empty.



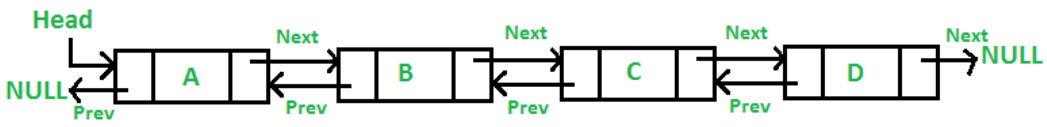
2) Circular Header Linked List

A list in which last node points back to the header node is called circular linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does not contain the NULL pointer.



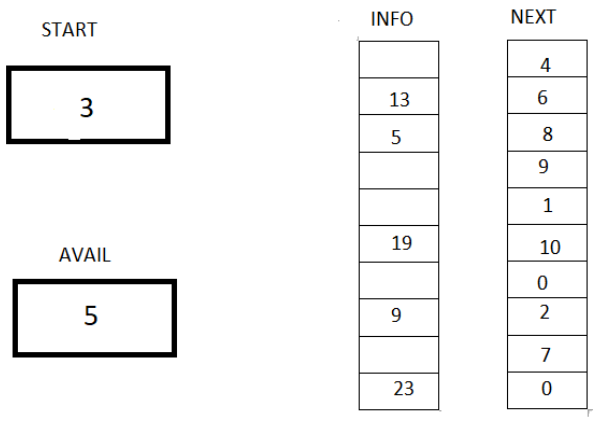
Two Way Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

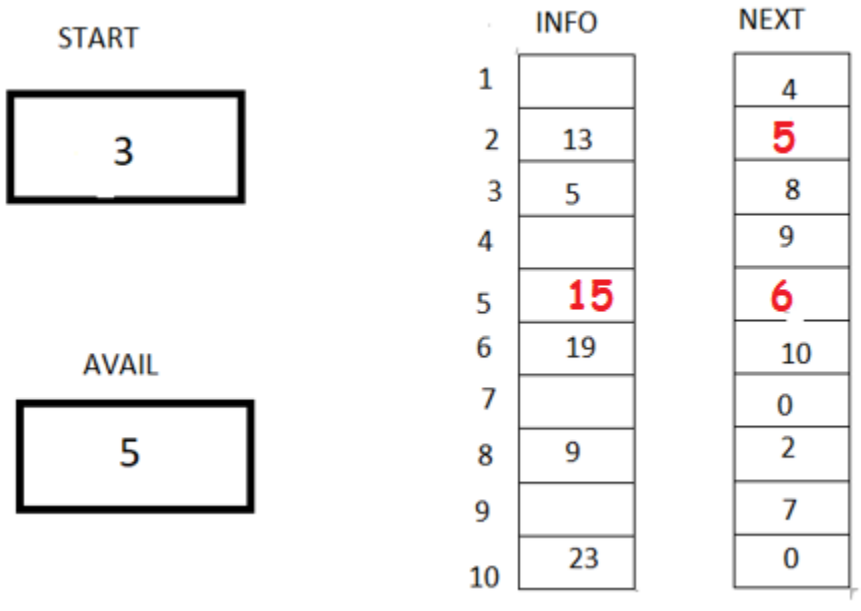


Practice:

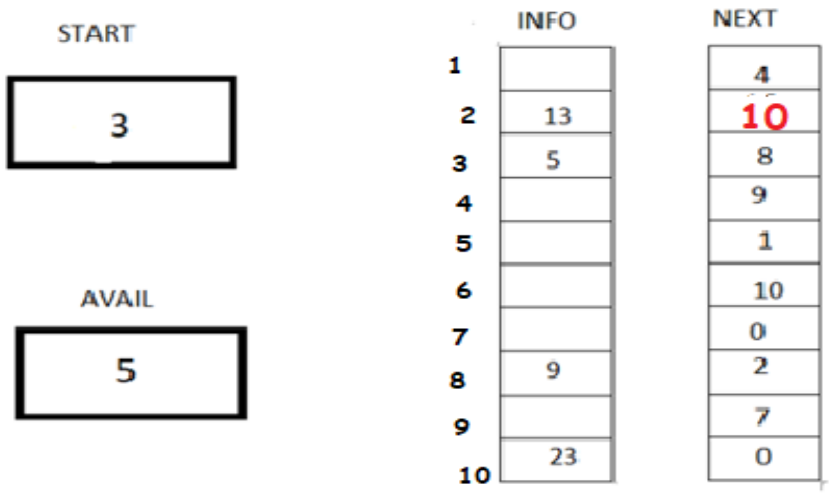
Consider the following sorted linked list represented using two linear arrays INFO and NEXT.



1) Redraw the given figure so that it represents the sorted linked list after inserting a node containing the value 15 in the INFO field.



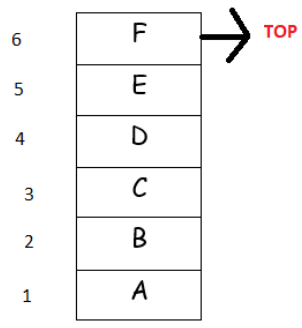
2) Redraw the given figure so that it represents the sorted linked list after deleting a node containing the value 19 in the INFO field.



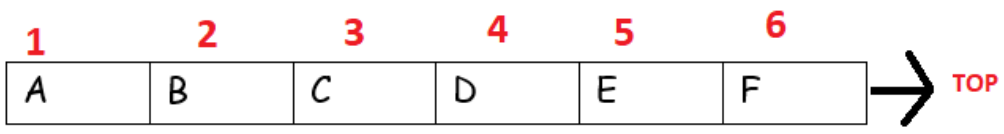
Stacks, Queues and Recursion

- **Stacks follow LIFO (last in first out) method.**
- Push – insert an element into a stack.
- Pop – delete an element from a stack.

Stack: A, B, C, D, E, F



In array:



Here, top = 6 and size = 6

Push:

PUSH (STACK, TOP, SIZE, ITEM)

This procedure pushes an ITEM into a stack.

1) If TOP = SIZE, print: overflow and return
2) Set TOP = TOP + 1
3) Set STACK [TOP] = ITEM (inserts ITEM in new TOP position)
4) Return

Pop:

POP (STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the carriable ITEM.

1) If TOP = 0, print: underflow and return.
2) Set ITEM = STACK [TOP] (assigns TOP element to ITEM)
3) Set TOP = TOP – 1
4) Return

Practice:

- Consider the following stack of city names (stack is allocated n = 6 memory cells)

Stack: London, Berlin, Rome, Paris, _____, _____

Describe the stack as the following operations take place:

1) PUSH (Athens)
2) POP ()
3) POP ()
4) PUSH (Madrid)
5) PUSH (Moscow)
6) POP ()
7) PUSH (Dhaka)
8) PUSH (Kabul)
9) PUSH (Thimpu)
10) PUSH (Beijing)

Solution:

1) London, Berlin, Rome, Paris, Athens, _____
2) London, Berlin, Rome, Paris, _____, _____
3) London, Berlin, Rome, _____, _____, _____
4) London, Berlin, Rome, Madrid, _____, _____
5) London, Berlin, Rome, Madrid, Moscow, _____
6) London, Berlin, Rome, Madrid, _____, _____
7) London, Berlin, Rome, Madrid, Dhaka, _____
8) London, Berlin, Rome, Madrid, Dhaka, Kabul
9) Overflow
10) Overflow

Evaluation of Postfix Expression:

Steps:

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.

a) If the element is a number, push it into the stack

b) If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack.

3) When the expression is ended, the number in the stack is the final answer.

Example:

Let the given expression be "2 3 1 * + 9 -". We scan all elements one by one.

1) Scan '2', it's a number, so push it to stack. Stack contains '2',

2) Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)

3) Scan '1', again a number, push it to stack, stack now contains '2 3 1'

4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.

5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.

6) Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.

7) Scan '-', it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.

8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Practice:

Given: 5, 6, 2, +, *, 12, 4, /, -

Solution:

Postfix	*Stack*	*Comment*
5	*5*	*Push 5*
6	*5, 6*	*Push 6*
2	*5, 6, 2*	*Push 2*
+	*5, 8*	*Pop 2 and 6* *2 + 6 = 8 and push*
***	*40*	*Pop 8 and 5* *8 * 5 = 40 and push*
12	*40, 12*	*Push 12*
4	*40, 12, 4*	*Push 4*
/	*40, 3*	*Pop 4 and 12* *12 / 4 = 3 and push*
-	*37*	*Pop 3 and 40* *40 – 3 = 37 and push*

Answer is 37.

Infix to Postfix Conversion

Step 0. Tokenize the infix expression. i.e Store each element i.e (operator / operand / parentheses) of an infix expression into a list / queue.

Step 1. Push "(" onto a stack and append ")" to the tokenized infix expression list / queue.

Step 2. For each element (operator / operand / parentheses) of the tokenized infix expression stored in the list/queue repeat steps 3 up to 6.

Step 3. If the token equals "(", push it onto the top of the stack.

Step 4. If the token equals ")", pop out all the operators from the stack and append them to the postfix expression till an opening bracket i.e "(" is found.

Step 5. If the token equals "*" or "/" or "+" or "-" or "^", pop out operators with higher precedence at the top of the stack and append them to the postfix expression. Push current token onto the stack.

Step 6. If the token is an operand, append it to the postfix expression. (Positions of the operands do not change in the postfix expression so append an operand as it is.)

Practice:

1) A + (B * C – (D / E ^ F) * G)* H

INFIX	STACK	POSTFIX
A		A
+	+	A
(	+ (	A
B	+ (	AB
*	+ (*	AB
C	+ (*	ABC
-	+ (-	ABC*
(	+ (- (	ABC*
D	+ (- (	ABC*D
/	+ (- (/	ABC*D
E	+ (- (/	ABC*DE
^	+ (- (/ ^	ABC*DE
F	+ (- (/ ^	ABC*DEF
)	+ (-	ABC*DEF^/
*	+ (- *	ABC*DEF^/

G	+ (- *	ABC*DEF^/G
)	+	ABC*DEF^/G*-
*	+ *	ABC*DEF^/G*-
H	+ *	ABC*DEF^/G*-H
		ABC*DEF^/G*-H*+

2) (A – 2 * (B + C) ^ 3 / D * E) + F ^ G

INFIX	STACK	POSTFIX
(	(	
A	(	A
-	(-	A
2	(-	A 2
*	(- *	A 2
(	(- * (	A 2
B	(- * (	A 2 B
+	(- * (+	A 2 B
C	(- * (+	A 2 B C
)	(- *	A 2 B C +
^	(- * ^	A 2 B C +
3	(- * ^	A 2 B C + 3
/	(- /	A 2 B C + 3 ^ *
D	(- /	A 2 B C + 3 ^ * D
*	(- *	A 2 B C + 3 ^ * D /
E	(- *	A 2 B C + 3 ^ * D / E
)		A 2 B C + 3 ^ * D / E * -
+	+	A 2 B C + 3 ^ * D / E * -
F	+	A 2 B C + 3 ^ * D / E * - F
^	+ ^	A 2 B C + 3 ^ * D / E * - F
G	+ ^	A 2 B C + 3 ^ * D / E * - F G
		A 2 B C + 3 ^ * D / E * - F G ^ +

Recursion

Fibonacci

FIBONACCI (FIB, N)

This procedure calculates FN and returns the value in the first parameter FIB.

1. If N = 0 or N = 1, then: Set FIB := N, and Return.

2. Call FIBONACCI (FIBA, N – 2).

3. Call FIBONACCI (FIBB, N – 1).

4. Set FIB := FIBA + FIBB.

5. Return

Factorial

FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the

variable FACT.1. If N = 0, then: Set FACT := 1, and Return.

2. Set FACT := 1. [Initializes FACT for loop.]

3. Repeat for K = 1 to N.

Set FACT := K*FACT.

[End of loop.]

4. Return.

Tower of Hanoi

N = Total number of discs.

```
TOH (n, source, medium, destination)
{
        If (n == 0) return.
        TOH (n - 1, source, destination, medium).
        TOH (n - 1, destination, medium, source).
}
```

The solution to the Towers of Hanoi problem for $n = 3$ appears in Fig. 6.15. Observe that it consists of the following seven moves:

$n = 3$: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.

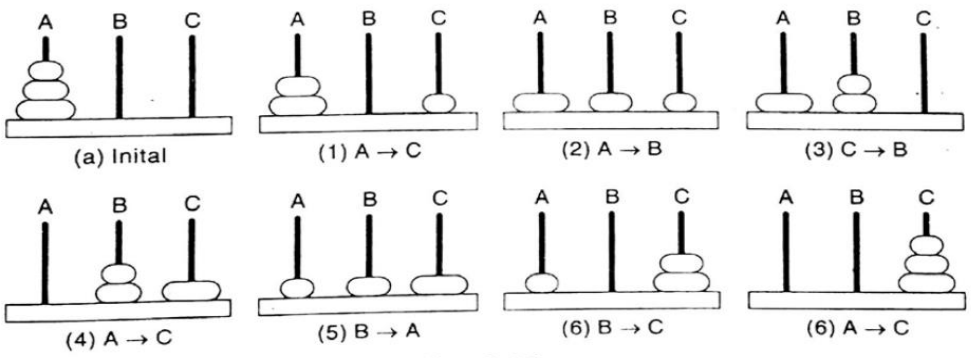


Fig. 6.15

For n = 4:

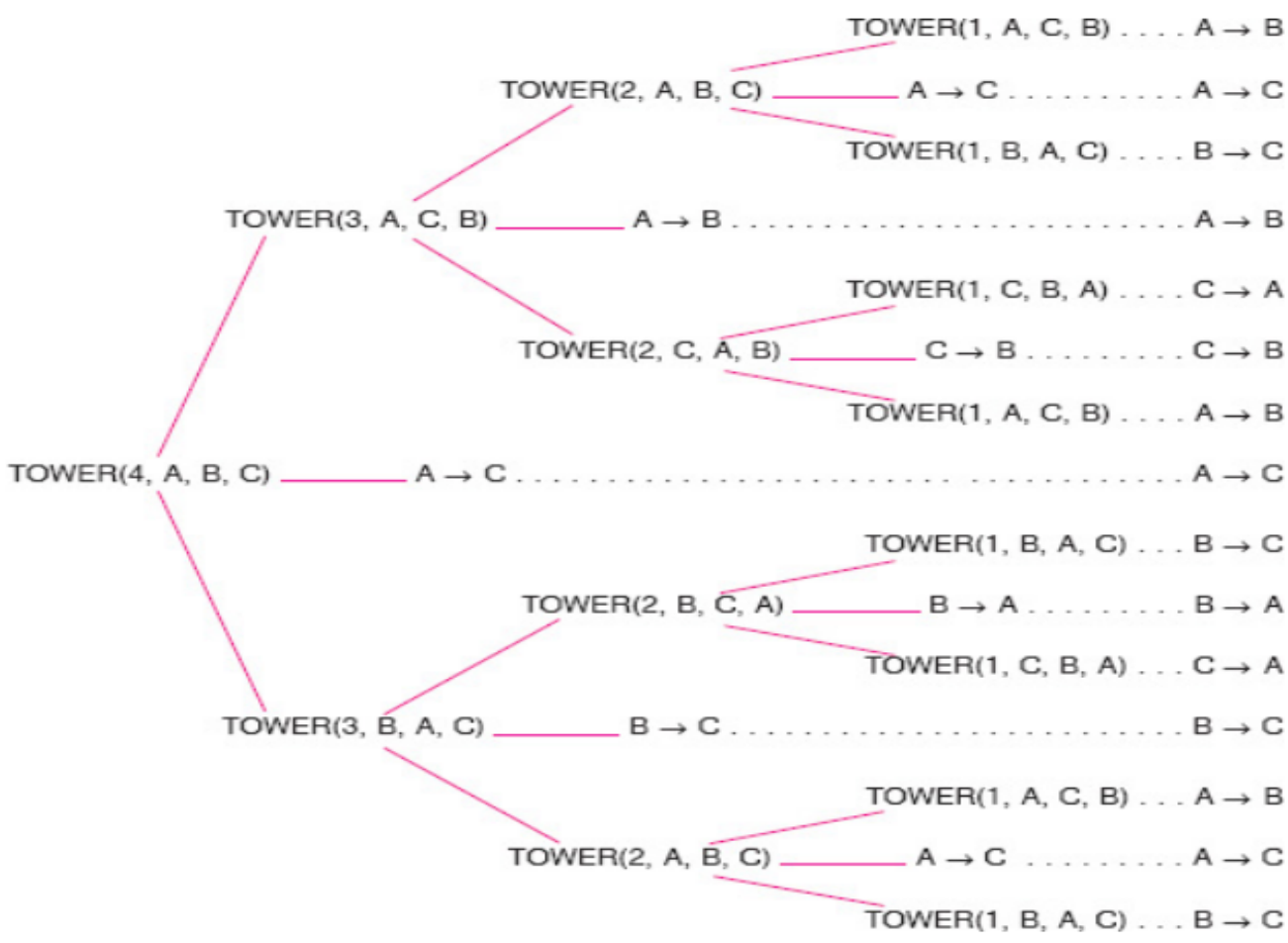


Fig. 6.17 ***Recursive Solution to Towers of Hanoi Problem for n = 4***

Observe that the recursive solution for n = 4 disks consists of the following 15 moves:

$A \to B \quad A \to C \quad B \to C \quad A \to B \quad C \to A \quad C \to B \quad A \to B \quad A \to C$
$B \to C \quad B \to A \quad C \to A \quad B \to C \quad A \to B \quad A \to C \quad B \to C$

In general, this recursive solution requires $f(n) = 2^n - 1$ moves for n disks.

Ackermann Function

(a) If m = 0, then A (m, n) = n + 1.

(b) If m ≠ 0 but n = 0, then A (m, n) = A (m – 1, 1).

(c) If m ≠ 0 and n ≠ 0, then A (m, n) = A (m – 1, A (m, n – 1))

Practice for RECURSION:

- Let J and K be integers and suppose Q (J, K) is recursively defined by

$$Q(J, K) = \begin{cases} 5 & \text{if } J < K \\ Q(J - K, K + 2) + J & \text{if } J \geq K \end{cases}$$

Find Q (2, 7), Q (5, 3) and Q (15, 2).

Solution:

Q (2, 7):

J = 2, K = 7, so J < K. So, Q (2, 7) = 5

Q (5, 3):

J = 5, K = 3, so J >= K

Q (5, 3) = Q (5-3, 3+2) + 5 = Q (2, 5) + 5 = 5 + 5 (J < K) = 10

Q (15, 2):

J = 15, K = 2, so J >= K

Q (15, 2) = Q (15-2, 2+2) + 15 = Q (13, 4) + 15 = Q (13-4, 4+2) + 13 + 15 = Q (9, 6) + 28 = Q (9-6, 6+2) + 9 + 28 = Q (3, 8) + 37 = 5 + 37 = 42

Queues

Follows **FIFO** (First in First Out) methods.

Algorithm:

QINSERT (QUEUE, N, FRONT, REAR, ITEM)

This procedure inserts an element ITEM into a queue.

1. [Queue already filled?]

If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then: Write: OVERFLOW, and Return.

2. [Find new value of REAR.]

If FRONT := NULL, then: [Queue initially empty.] Set FRONT := 1 and REAR := 1.

Else if REAR = N, then: Set REAR := 1.

Else: Set REAR := REAR + 1.

[End of If structure.]

3. Set QUEUE[REAR] := ITEM. [This inserts new element.]

4. Return.

QDELETE (QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue and assigns it to the variable ITEM.

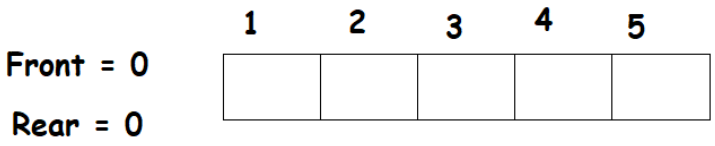
1. [Queue already empty?]

If FRONT := NULL, then: Write: UNDERFLOW, and Return.

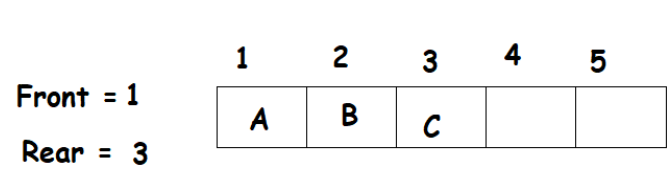
2. Set ITEM := QUEUE[FRONT].

Example:

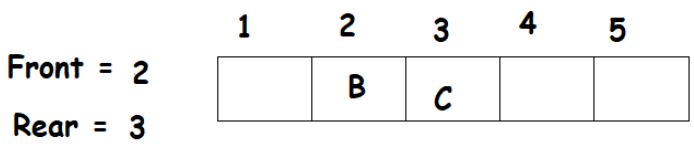
1) Initially empty:



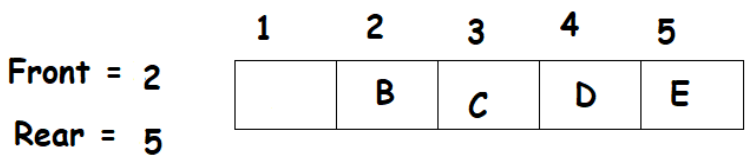
2) A, B, C inserted:



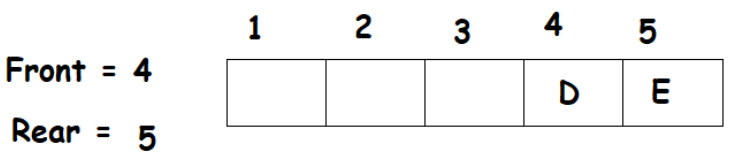
3) A deleted:



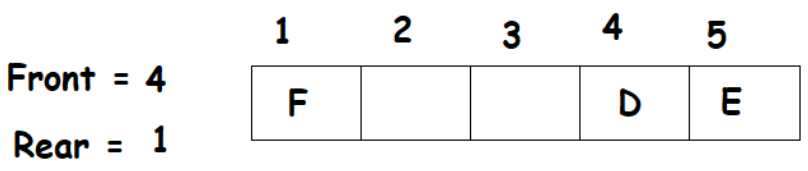
4) D, E inserted:



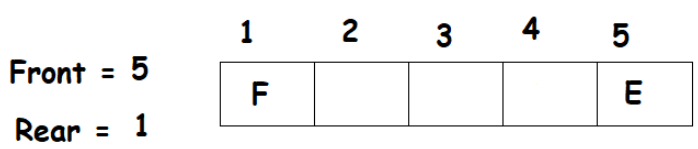
5) B and C deleted:



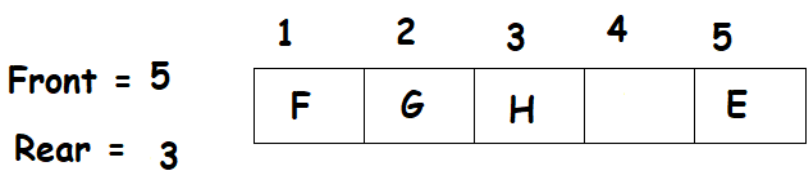
6) F inserted:



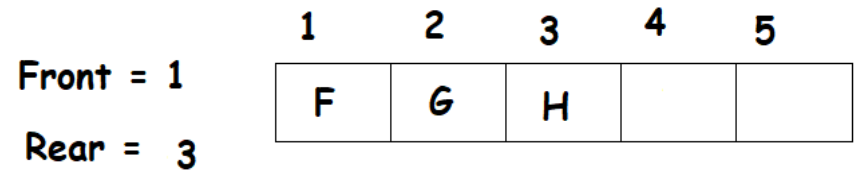
7) D deleted:



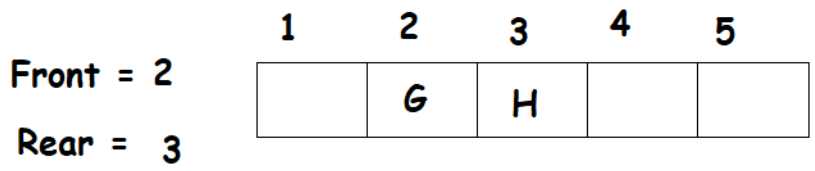
8) G, H inserted:



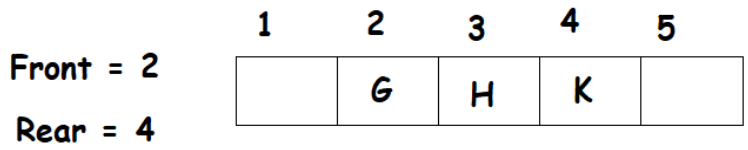
9) E deleted:



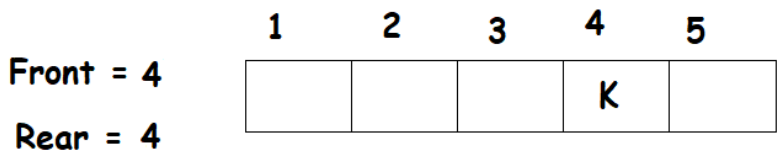
10) F deleted:



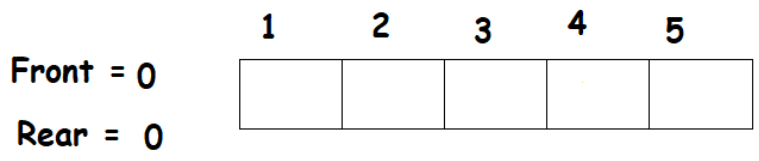
11) K inserted:



12) G and H deleted:



13) K deleted:



Practice: P – 281 (Colorful Book)

For more exercise: P – 307

Trees

Tree Traversal

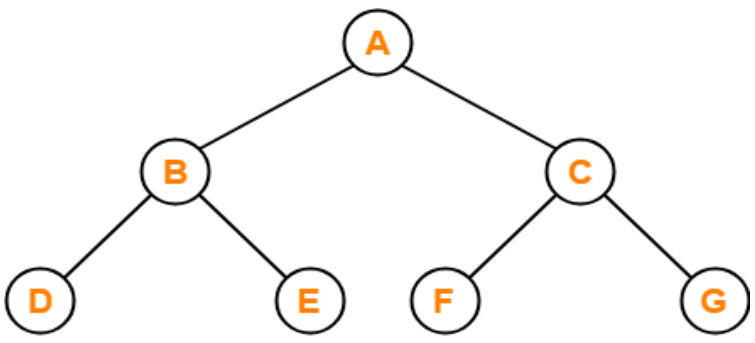
1. Preorder Traversal-

Algorithm-

1. Visit the root.
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

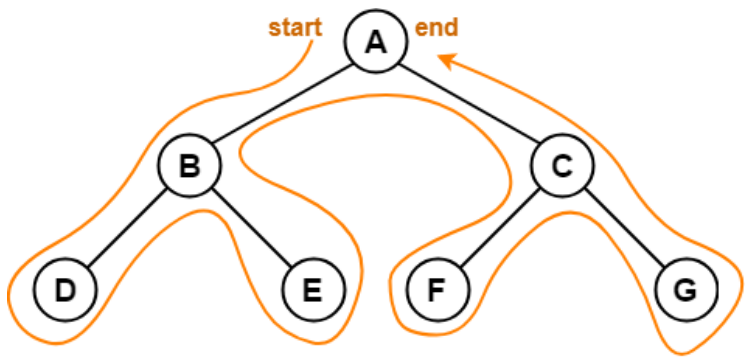
Example-



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut:

Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Applications-

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

2. Inorder Traversal-

Algorithm-

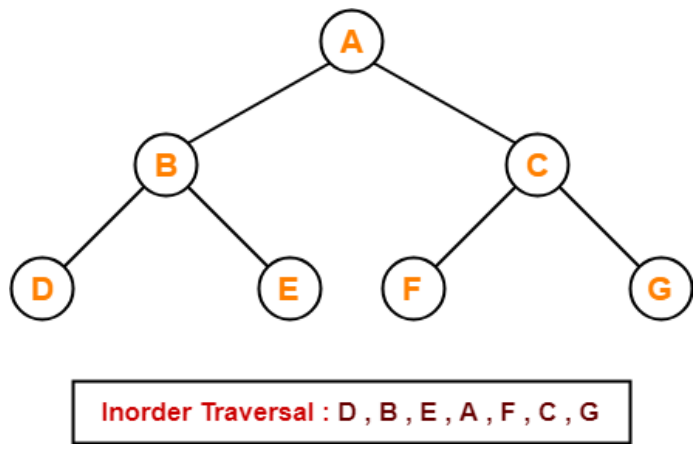
1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root.

3. Traverse the right sub tree i.e. call In order (right sub tree)

Left → Root → Right

Example-

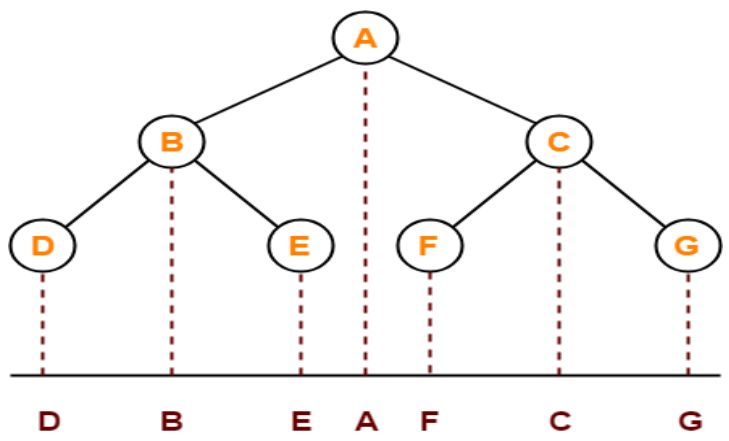
Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut:

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

3. Postorder Traversal-

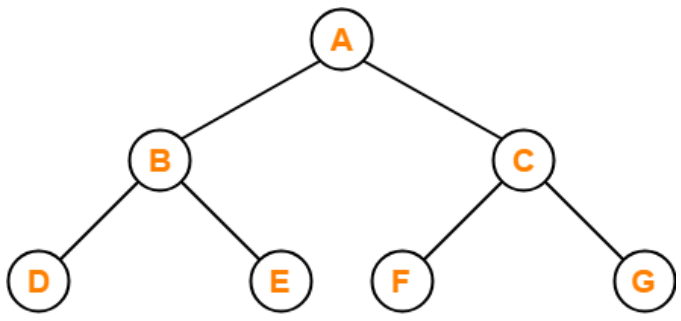
Algorithm-

1. Traverse the left sub tree i.e., call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example-

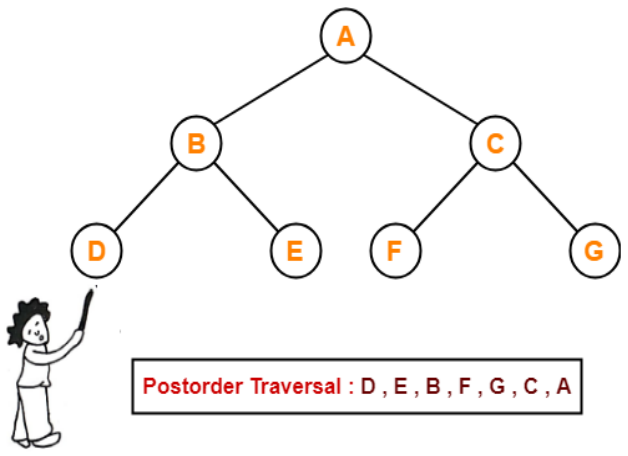
Consider the following example-



Postorder Traversal : D , E , B , F , G , C , A

Postorder Traversal Shortcut:

Pluck all the leftmost leaf nodes one by one.



Postorder Traversal : D , E , B , F , G , C , A

Traversal Using Stacks

- **Pre-order Traversal**

Steps:

1) Push the root into stack.
2) While stack is not empty
   a) pop the top element and process it.
   b) Push its (the popped element) right and then left child.

Example:

```
        1
      /   \
     2     3
   /  \    /\
  4    5  6  7
```

Step:

Node of Tree	Stack	Output (Popped elements)
Push 1	1	
Pop 1, push 3 and 2	3, 2	1
Pop 2, push 5 and 4	3, 5, 4	1, 2

Pop 4	3, 5	1, 2, 4
Pop 5	3	1, 2, 4, 5
Pop 3, push 7 and 6	7, 6	1, 2, 4, 5, 3
Pop all elements		1, 2, 4, 5, 3, 6, 7

- **In-order Traversal**

Steps:

1) Set current node = root and push root

2) Continue pushing the left child of current node until there is no left child. Then step 3.

3) If stack is not empty then

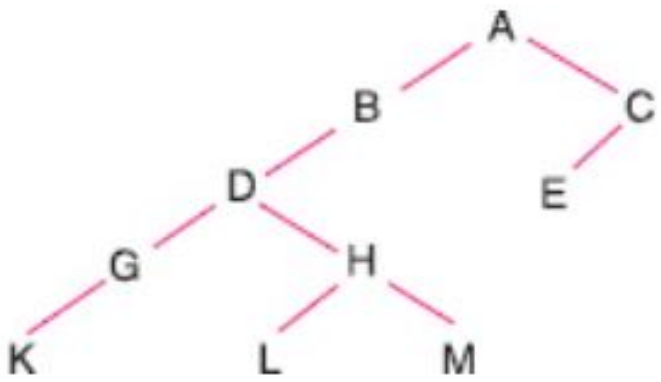
   a) Pop the top item from stack.

   b) Print the popped item, set current node = popped_item->right

   c) Go to step 2.

4) If current is NULL and stack is empty then we are done.

Example:



Step	Stack	Print
Push A	A	
Push B	A, B	
Push D	A, B, D	
Push G	A, B, D, G	
Push K	A, B, D, G, K	
Pop till there is node with right child	A, B	K, G, D
Push H	A, B, H	K, G, D
Push L	A, B, H, L	K, G, D
Pop till there is node with right child	A, B	K, G, D, L, H
Push M	A, B, M	K, G, D, L, H
Pop till there is node with right child		K, G, D, L, H, M, B, A
Push C	C	K, G, D, L, H, M, B, A
Push E	C, E	K, G, D, L, H, M, B, A
Pop till there is node with right child		K, G, D, L, H, M, B, A, E, C

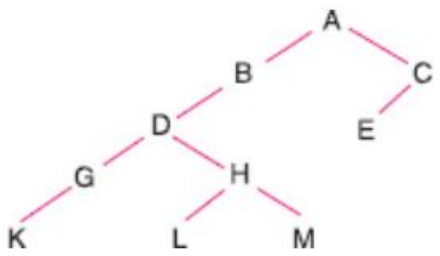
Traversal is done now as stack S is empty and current is NULL.

• Post-order Traversal

Steps:

1. Push root to first stack.
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack.
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack.

Example:



Step	1st Stack	2nd Stack
A	A	
Pop A, Push B and C	B, C	A
Pop C, Push E	B, E	A, C
Pop E	B	A, C, E
Pop B, Push D	D	A, C, E, B
Pop D, Push G and H	G, H	A, C, E, B, D
Pop H, Push L and M	G, L, M	A, C, E, B, D, H
Pop M	G, L	A, C, E, B, D, H, M
Pop L	G	A, C, E, B, D, H, M, L
Pop G, Push K	K	A, C, E, B, D, H, M, L, G
Pop K		A, C, E, B, D, H, M, L, G, K

Pop and print the elements from the 2nd stack: K, G, L, M, H, D, B, E, C, A

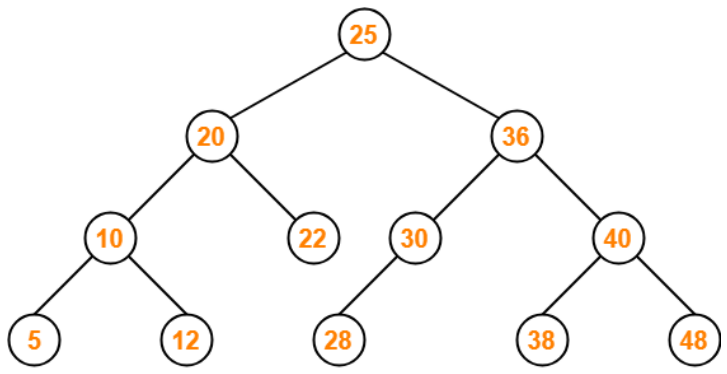
Binary Search Tree

Binary Search Tree is a special kind of binary tree in which nodes are arranged in a specific order.

In a binary search tree (BST), each node contains-

- Only smaller values in its left sub tree
- Only larger values in its right sub tree

Example-



Binary Search Tree

Binary Search Tree Construction-

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(T: binary search tree, x: item)
v := root of T
{a vertex not present in T has the value *null*}
while $v \neq null$ and $label(v) \neq x$
  if $x < label(v)$ **then**
    if left child of $v \neq null$ **then** v := left child of v
    else add *new vertex* as a left child of v and set $v := null$
  else
    if right child of $v \neq null$ **then** v := right child of v
    else add *new vertex* as a right child of v and set $v := null$
if root of $T = null$ **then** add a vertex v to the tree and label it with x
else if v is null or $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {v = location of x}

Let us understand the construction of a binary search tree using the following example-

Example:

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

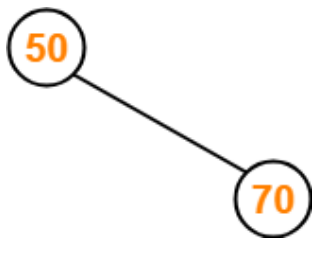
- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

The binary search tree will be constructed as explained below-

Insert 50-

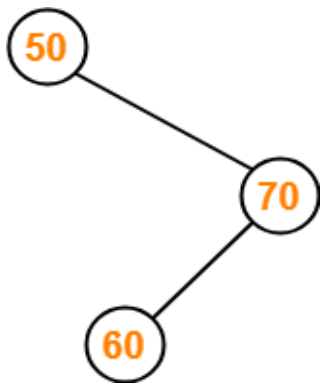
Insert 70-

- As 70 > 50, so insert 70 to the right of 50.



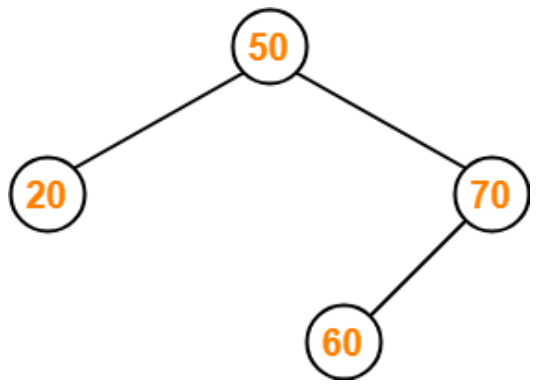
Insert 60-

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



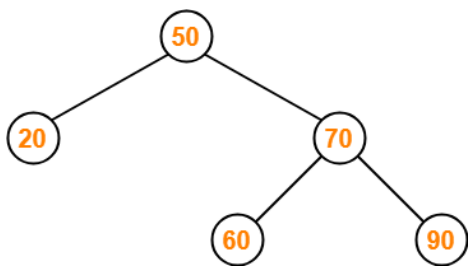
Insert 20-

- As 20 < 50, so insert 20 to the left of 50.



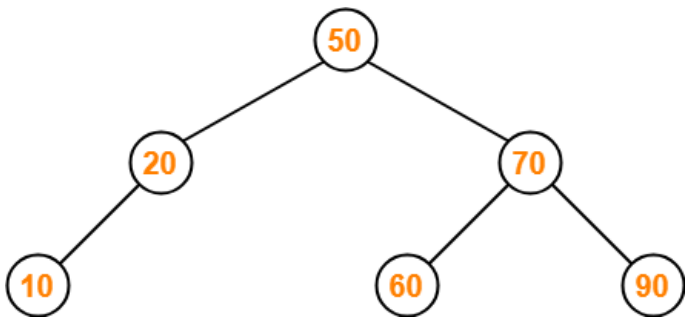
Insert 90-

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



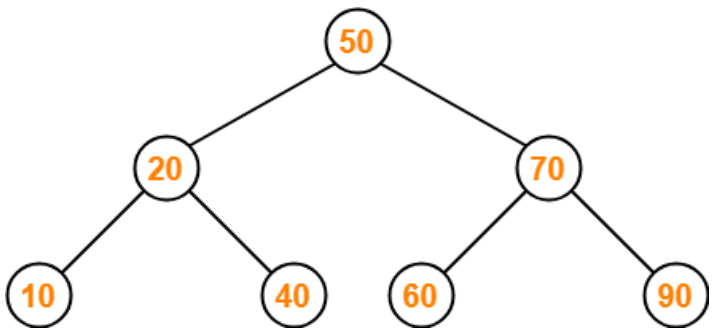
Insert 10-

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



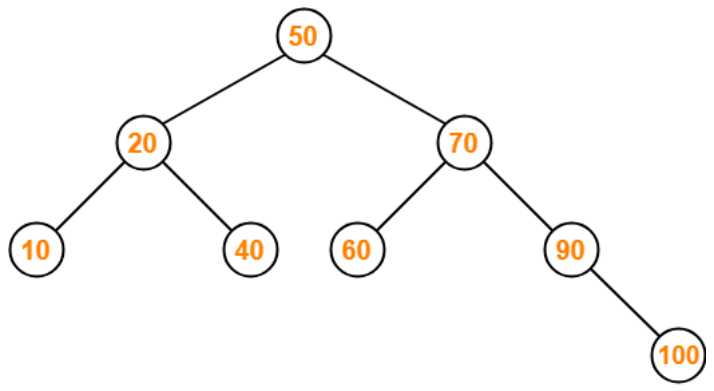
Insert 40-

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Insert 100-

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



Binary Search Tree

PRACTICE PROBLEMS BASED ON BINARY SEARCH TREES-

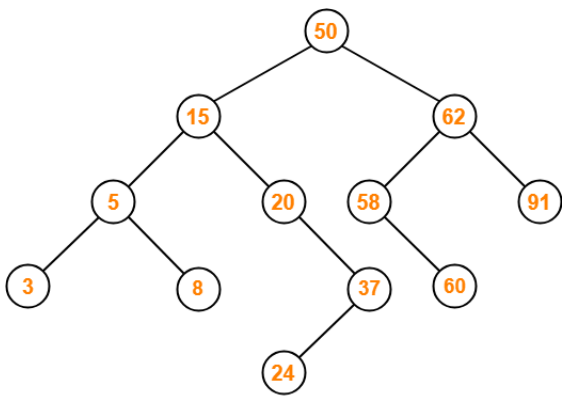
Problem-01: A binary search tree is generated by inserting in order of the following integers-

50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

The number of nodes in the left subtree and right subtree of the root respectively is _____.

a) (4, 7)
b) (7, 4)
c) (8, 3)
d) (3, 8)

Solution- Using the above discussed steps, we will construct the binary search tree. The resultant binary search tree will be-



Binary Search Tree

Clearly,

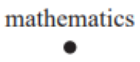
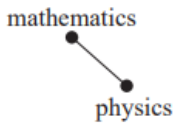
- Number of nodes in the left subtree of the root = 7

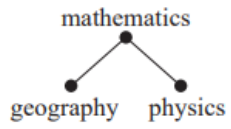
- Number of nodes in the right subtree of the root = 4

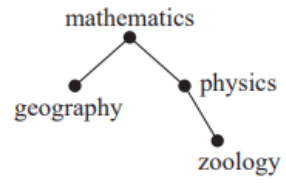
Thus, Option (B) is correct.

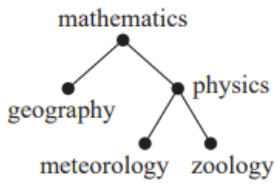
Problem-02: Form a binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).

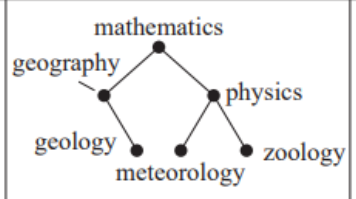
Solution:

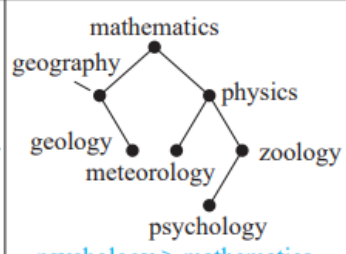
<table>
<tr>
<td> 
</td>
<td> 

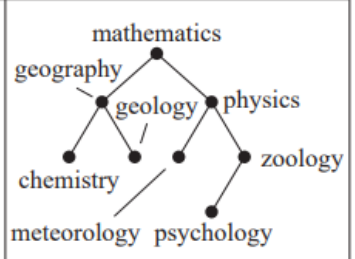
physics > mathematics</td>
<td> 

geography < mathematics</td>
<td> 

zoology > mathematics
zoology > physics</td>
</tr>
<tr>
<td> 

meteorology > mathematics
meteorology < physics</td>
<td> 

geology < mathematics
geology > geography</td>
<td> 

psychology > mathematics
psychology > physics
psychology < zoology</td>
<td> 

chemistry < mathematics
chemistry < geography</td>
</tr>
</table>

Binary search tree (BST) is a special kind of binary tree where each node contains-

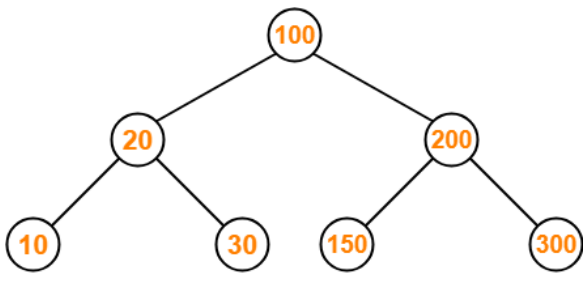
- **Only larger values in its right subtree.**
- **Only smaller values in its left subtree.**

Binary Search Tree Traversal-

- A binary search tree is traversed in exactly the same way a binary tree is traversed.
- In other words, BST traversal is same as binary tree traversal.

Example-

Consider the following binary search tree-



Binary Search Tree

Now, let us write the traversal sequences for this binary search tree-

Preorder Traversal- 100, 20, 10, 30, 200, 150, 300

Inorder Traversal- 10, 20, 30, 100, 150, 200, 300

Postorder Traversal- 10, 30, 20, 150, 300, 200, 100

Important Notes-

Note-01:

- Inorder traversal of a binary search tree always yields all the nodes in increasing order.

Note-02:

Unlike Binary Trees,

- A binary search tree can be constructed using only preorder or only postorder traversal result.
- This is because in order traversal can be obtained by sorting the given result in increasing order.

PRACTICE PROBLEMS BASED ON BINARY SEARCH TREE (BST) Traversal-

Problem-01:

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers.

What is the inorder traversal sequence of the resultant tree?

a) 7, 5, 1, 0, 3, 2, 4, 6, 8, 9
b) 0, 2, 4, 3, 1, 6, 5, 9, 8, 7
c) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
d) 9, 8, 6, 4, 2, 3, 0, 1, 5, 7

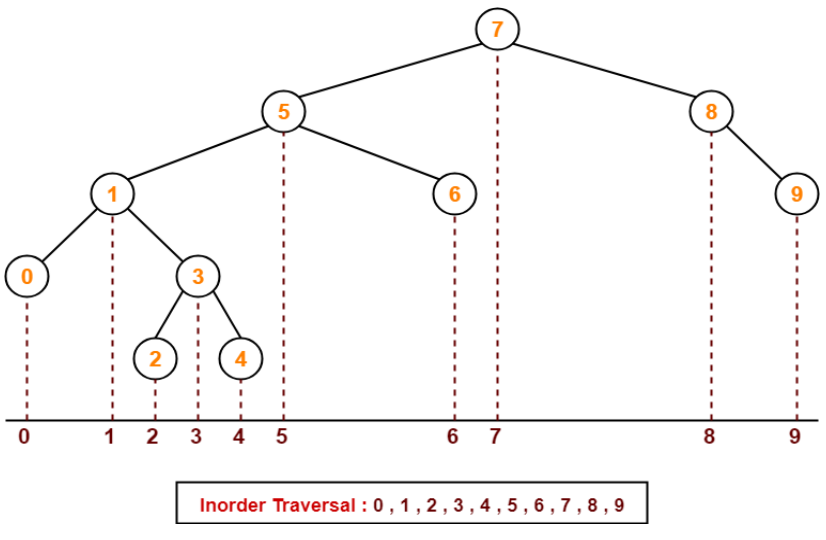
Solution-

This given problem may be solved in the following two ways-

Method-01:

- We construct a binary search tree for the given elements.
- We write the inorder traversal sequence from the binary search tree so obtained.

Following these steps, we have-



Thus, Option (C) is correct.

Method-02:

We know, in order traversal of a binary search tree always yields all the nodes in increasing order.

Using this result,

- We arrange all the given elements in increasing order.
- Then, we report the sequence so obtained as inorder traversal sequence.

Inorder Traversal:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Thus, Option (C) is correct.

Problem-02:

The preorder traversal sequence of a binary search tree is-

30, 20, 10, 15, 25, 23, 39, 35, 42

What one of the following is the postorder traversal sequence of the same tree?

a) 10, 20, 15, 23, 25, 35, 42, 39, 30
b) 15, 10, 25, 23, 20, 42, 35, 39, 30
c) 15, 20, 10, 23, 25, 42, 35, 39, 30
d) 15, 10, 23, 25, 20, 35, 42, 39, 30

Solution-

In this question,

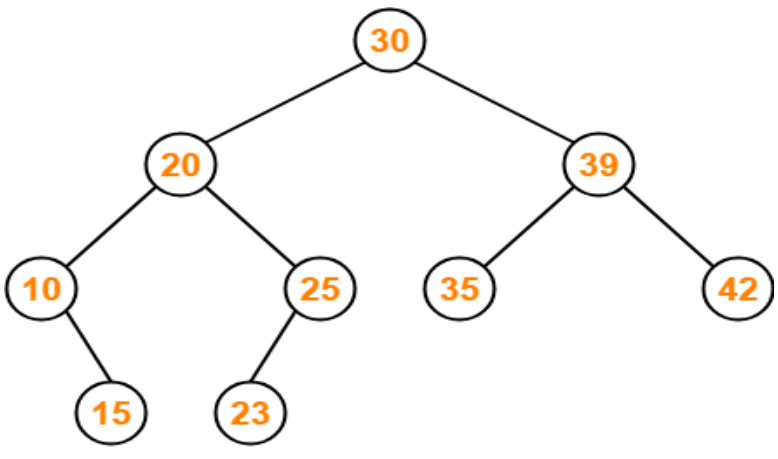
- We are provided with the preorder traversal sequence.
- We write the inorder traversal sequence by arranging all the numbers in ascending order.

Then-

- Preorder Traversal: 30, 20, 10, 15, 25, 23, 39, 35, 42
- Inorder Traversal: 10, 15, 20, 23, 25, 30, 35, 39, 42

Now,

- We draw a binary search tree using these traversal results.
- The binary search tree so obtained is as shown-



Binary Search Tree

Now, we write the postorder traversal sequence-

Postorder Traversal:

15, 10, 23, 25, 20, 35, 42, 39, 30

Thus, Option (D) is correct

Binary Search Tree Operations-

Commonly performed operations on binary search tree are-

1. **Search Operation**
2. **Insertion Operation**
3. **Deletion Operation**

1. Search Operation-

Search Operation is performed to search a particular element in the Binary Search Tree.

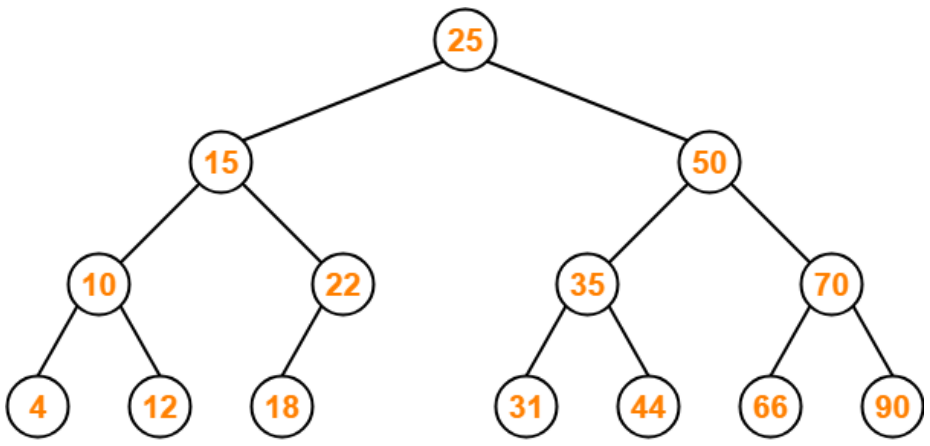
Rules-

For searching a given key in the BST,

- Compare the key with the value of root node.
- If the key is present at the root node, then return the root node.
- If the key is greater than the root node value, then recur for the root node's right subtree.
- If the key is smaller than the root node value, then recur for the root node's left subtree.

Example-

Consider key = 45 has to be searched in the given BST-



Binary Search Tree

- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

2. Insertion Operation-

Insertion Operation is performed to insert an element in the Binary Search Tree.

Rules-

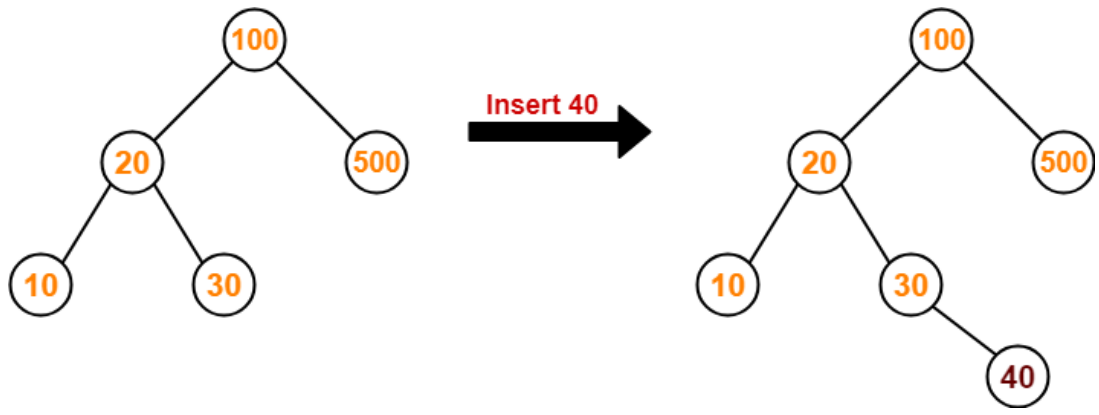
The insertion of a new key always takes place as the child of some leaf node.

For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.
- Once a leaf node is reached, insert the key as child of that leaf node.

Example-

Consider the following example where key = 40 is inserted in the given BST-



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

3. Deletion Operation-

Deletion Operation is performed to delete a particular element from the Binary Search Tree.

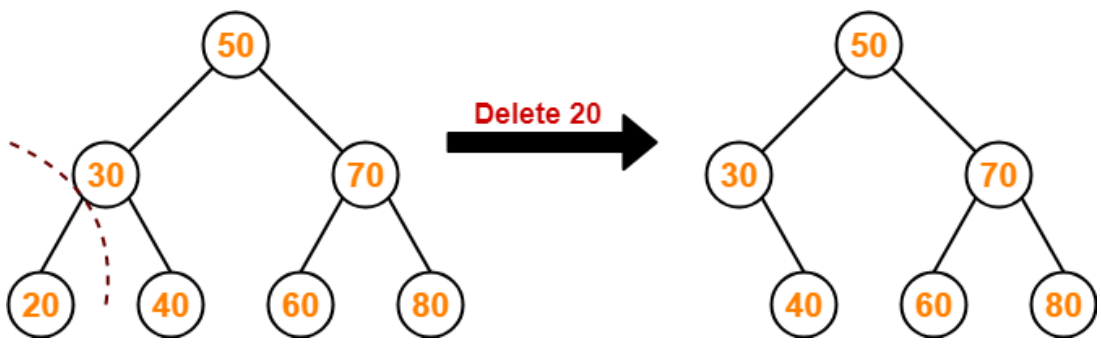
When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to delete from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

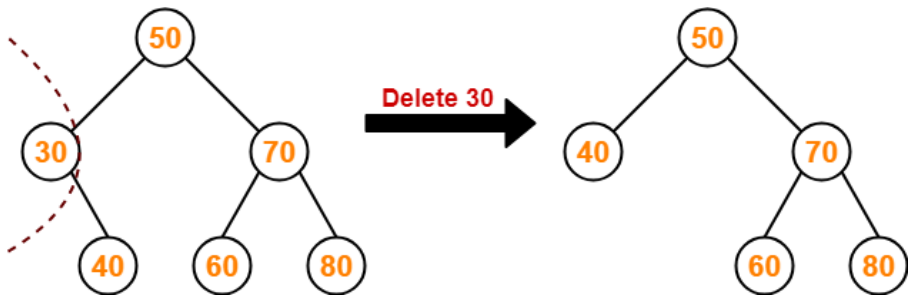


Case-02: Deletion Of A Node Having Only One Child-

Just make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-



Case-03: Deletion Of A Node Having Two Children-

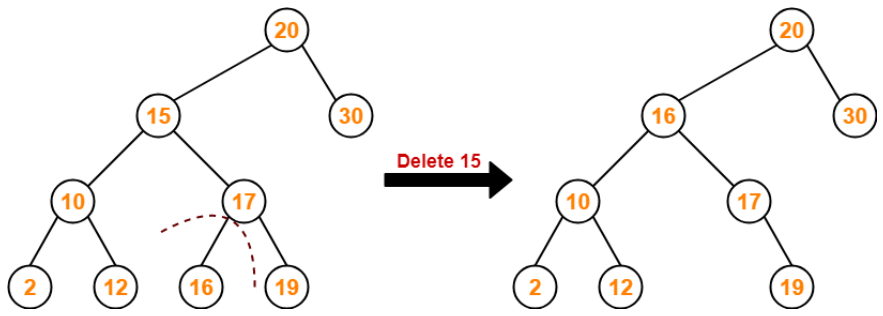
A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the **right subtree** of the deleting node.
- Pluck the **least value element** called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-

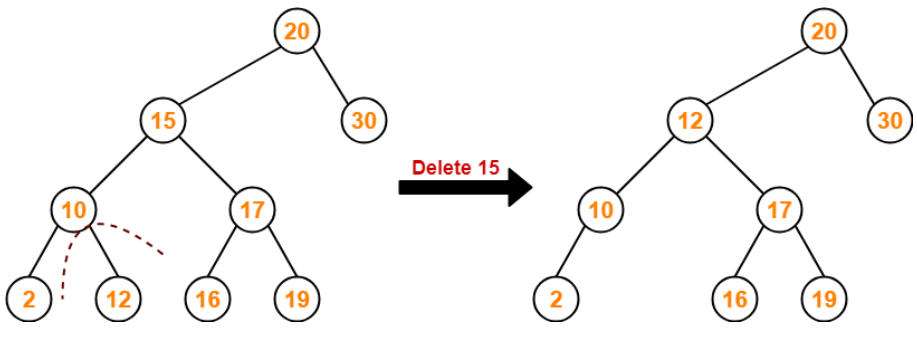


Method-02:

- Visit to the **left subtree** of the deleting node.
- Pluck the **greatest value element** called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-



Time Complexity-

- Time complexity of all BST Operations = O(h).

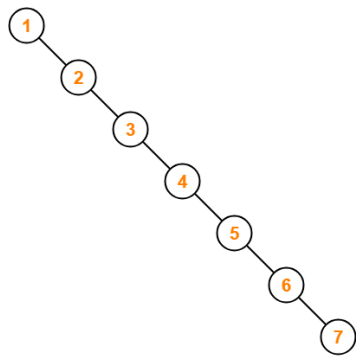
- Here, h = Height of binary search tree

Now, let us discuss the worst case and best case.

Worst Case-

- The binary search tree is a skewed binary search tree.
- Height of the binary search tree becomes n.
- So, time complexity of BST Operations = O(n).

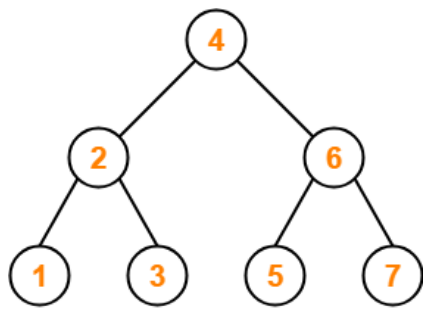
In this case, binary search tree is as good as unordered list with no benefits.



Skewed Binary Search Tree

Best Case-

- The binary search tree is a balanced binary search tree.
- Height of the binary search tree becomes log(n).
- So, time complexity of BST Operations = O(logn).



Balanced Binary Search Tree

Heapsort

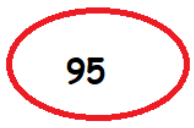
Given elements are: 95, 85, 70, 55, 33, 30, 65, 15, 20, 15, 22.

- Construct the min-heap.
- Heap sort.

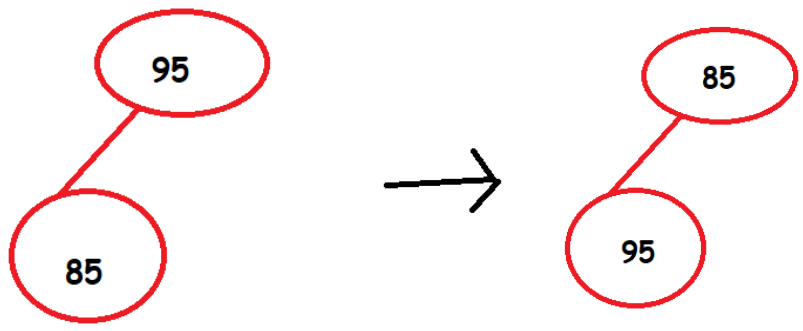
Solution:

1) **Min heap:**

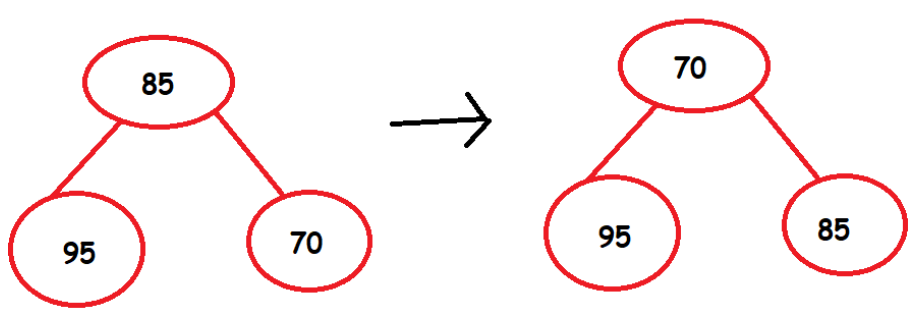
Step 1: Inserting 95:



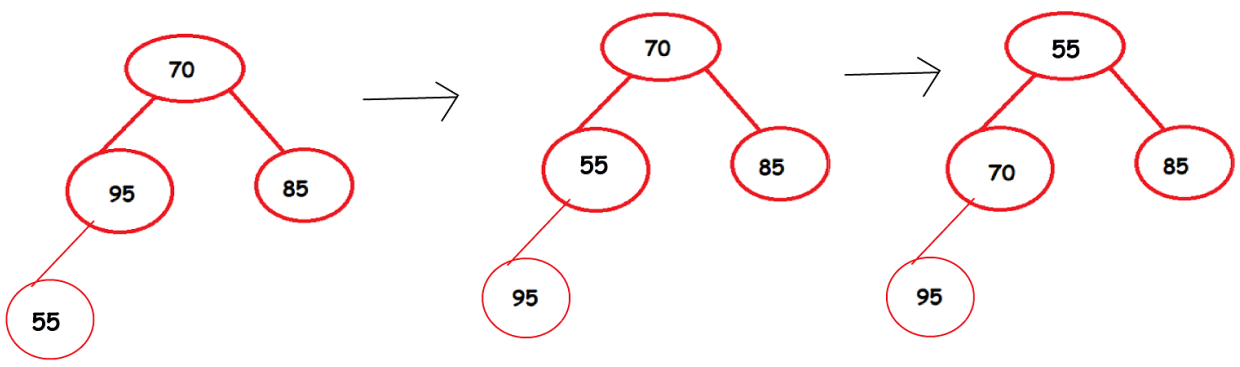
Step 2: Inserting 85:



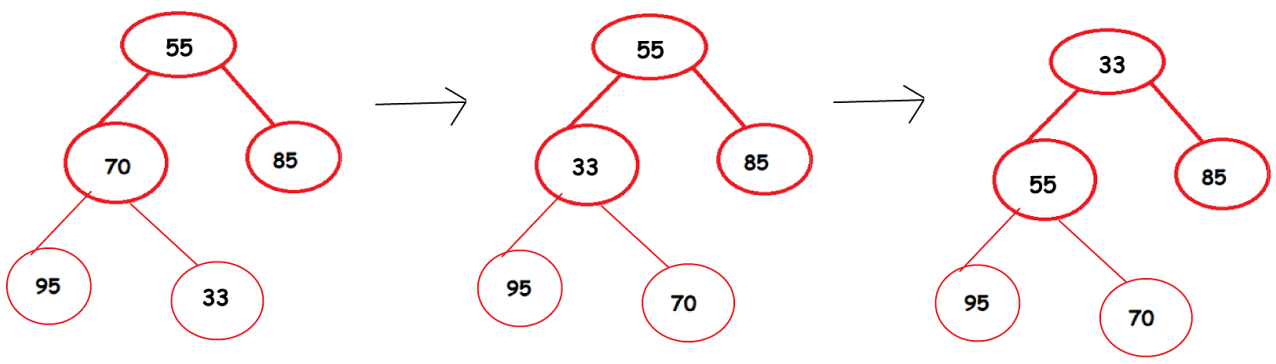
Step 3: Inserting 70



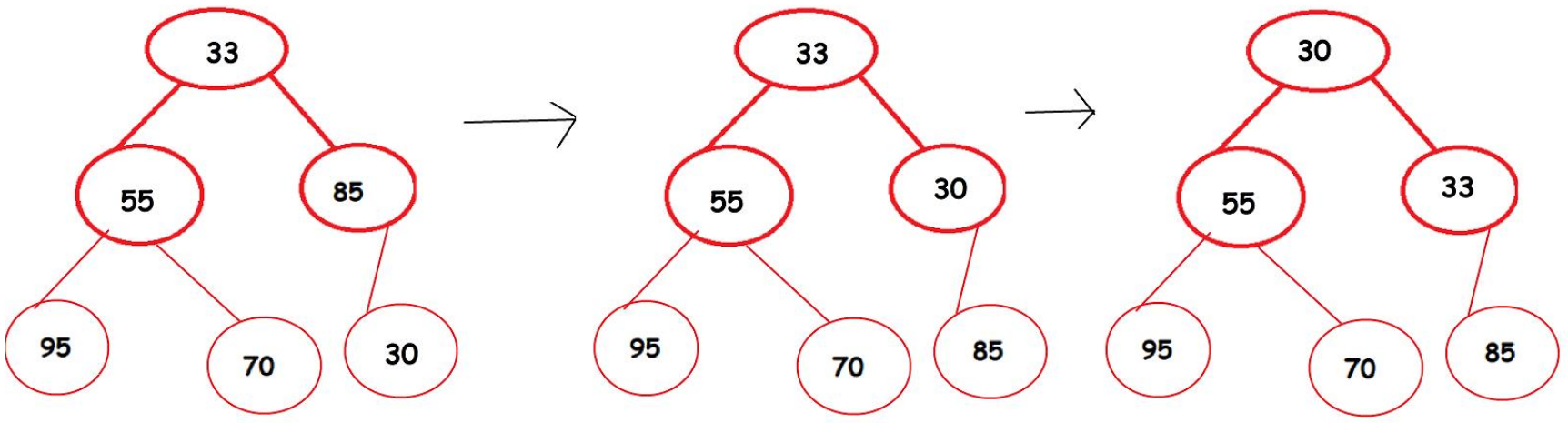
Step 4: Inserting 55



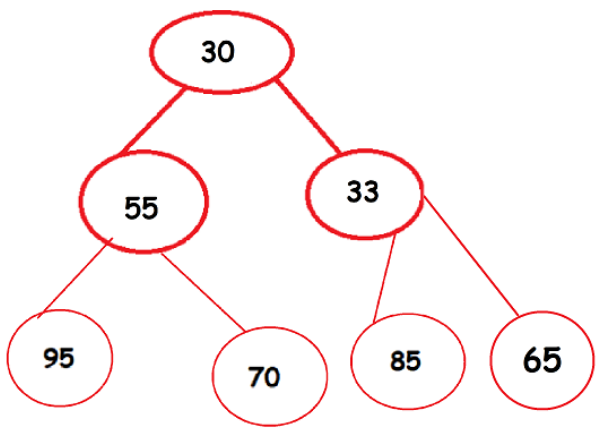
Step 5: Inserting 33



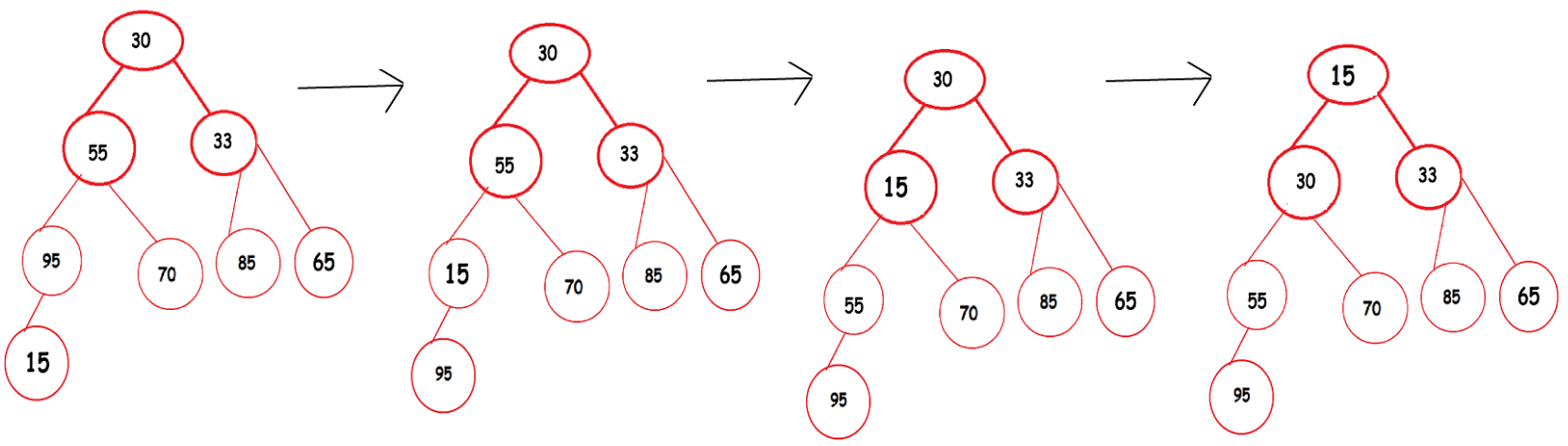
Step 6: Inserting 30



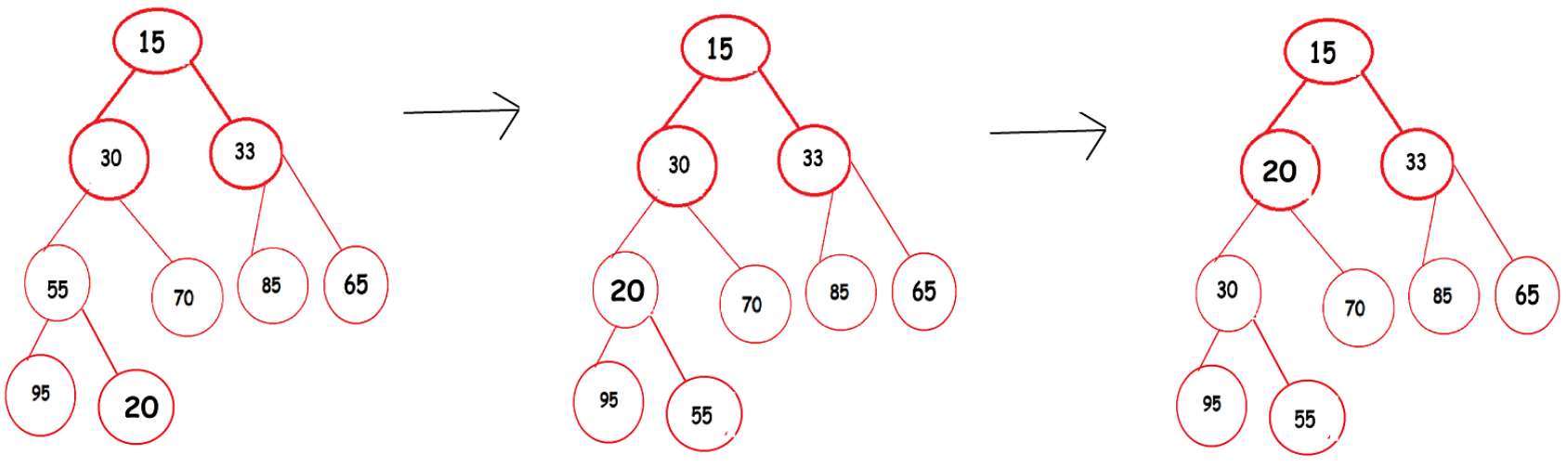
Step 7: Inserting 65



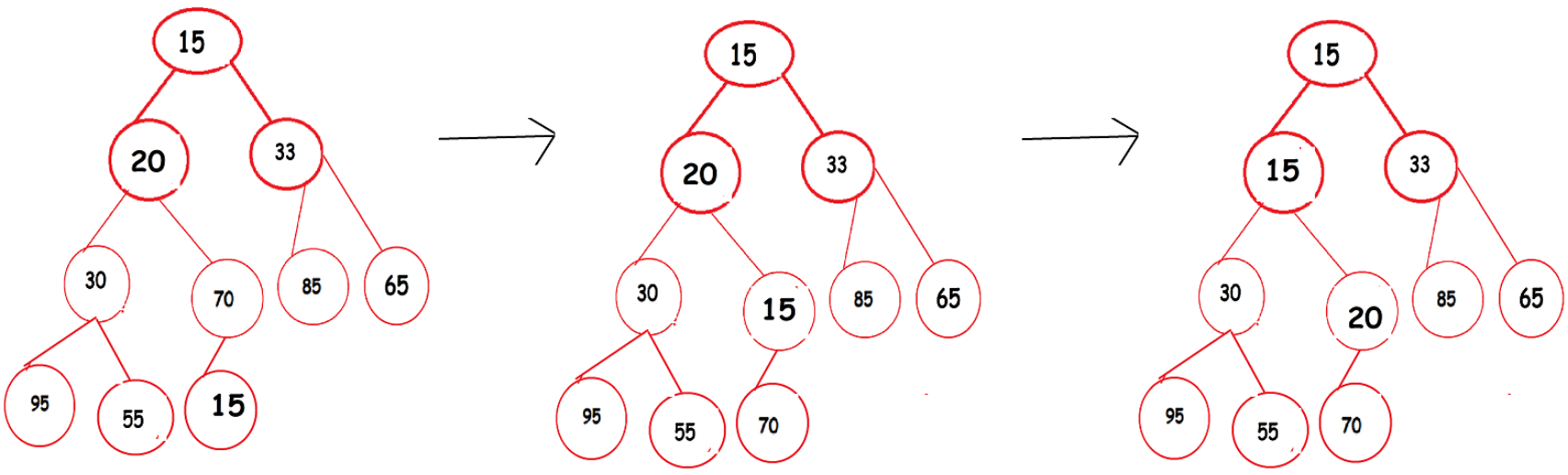
Step 8: Inserting 15



Step 9: Inserting 20

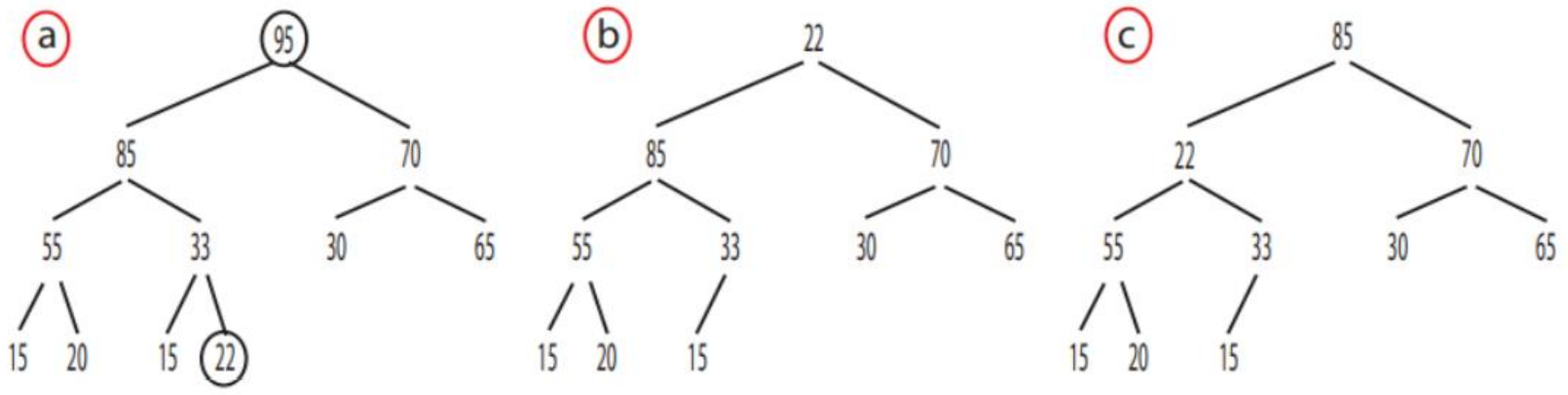


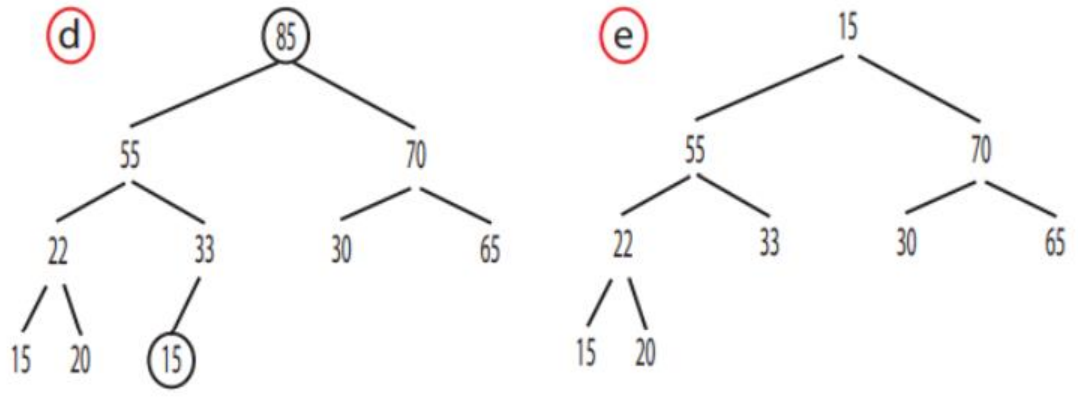
Step 10: Inserting 15

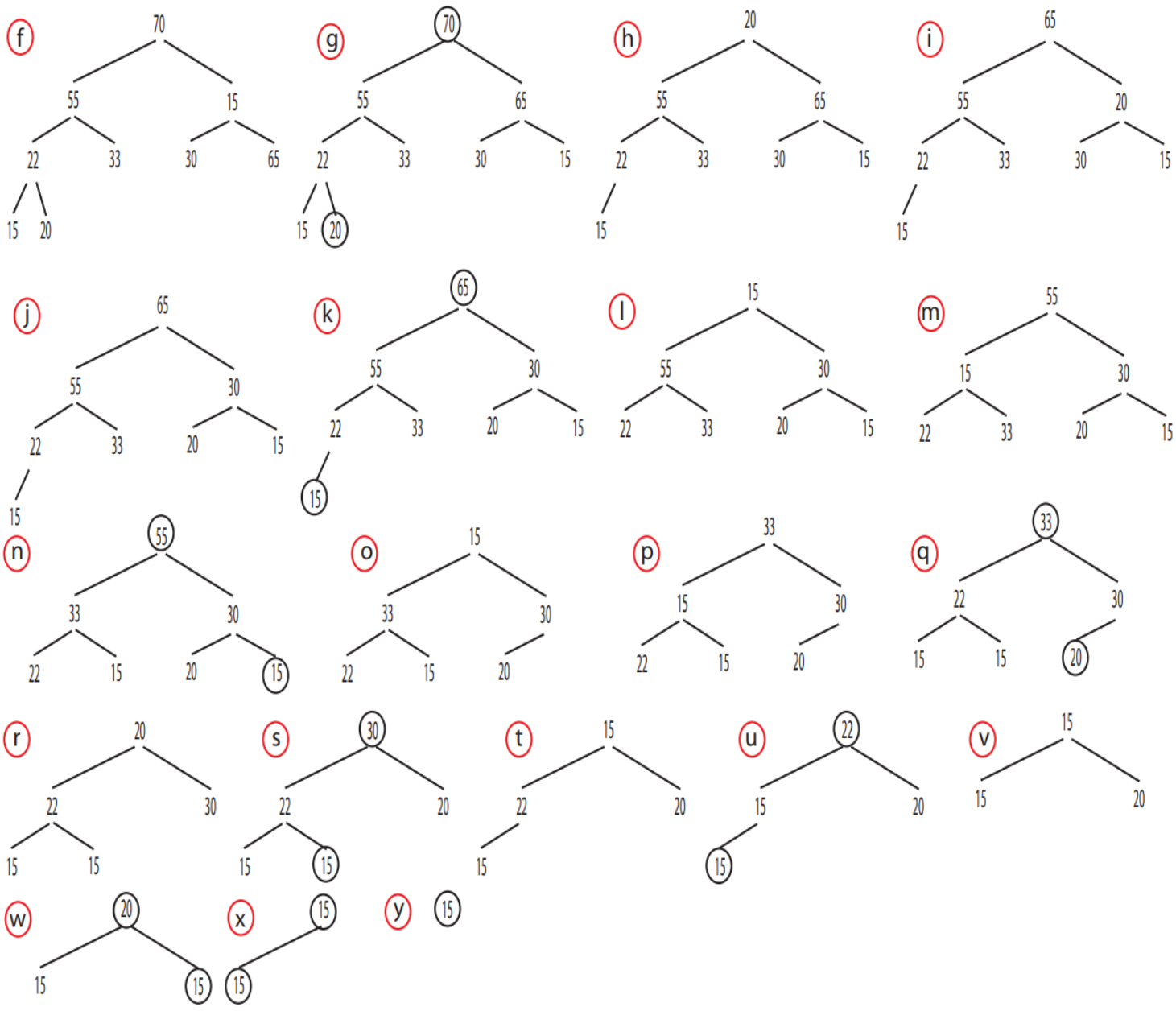


Step 11: Inserting 22

2) Heap sort (Max heap):







Step No	Removed Element
a	95
d	85
g	70
k	65
n	55
q	33
s	30
u	22
w	20
x	15
y	15

Sorted list: 15, 15, 20, 22, 30, 33, 55, 65, 70, 85, 95.

Huffman Coding (Variable Length Coding)

A B B C D B C C D A A B B E E E B E A B -> 20 Characters

A = 65 = 01000001 -> 8 bits

Total bits required = 20 x 8 = 160 (for fixed length coding)

 # With n bits, we can use 2^n characters.

Example: with 2 bits, four combinations (00,01,10,11) are possible.

 # For the above example, 3 bits are required as there are 5 ($5 > 2^2 = 4$ and $5 < 2^2 = 8$) different characters.

So, total bits required for this message = 20 x 3 = 60

Encryption Table

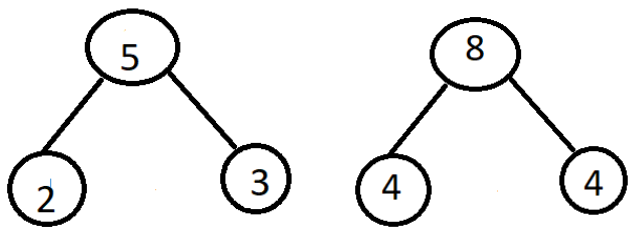
A – 000 B – 001 C – 010 D – 011 E – 100	For these alphabets, 5 x 8 = 40 bits For the codes, 5 x 3 = 15 bits

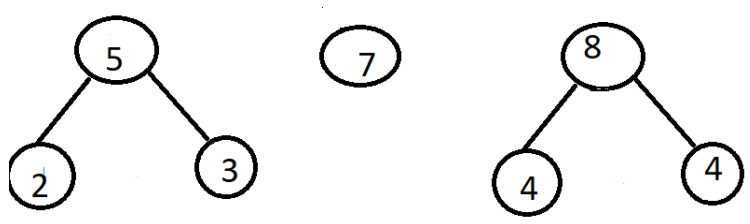
Total = 60 + 40 + 15 = 115 bits

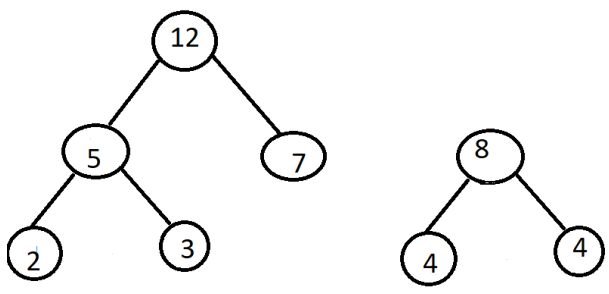
Drawbacks of fixed-length codes

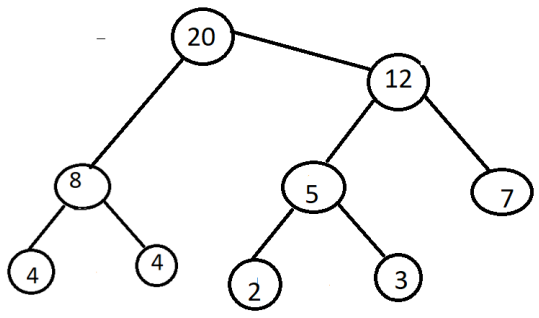
- Wasted space:
  - Unicode uses twice as much space as ASCII.
    - inefficient for plain-text messages containing only ASCII characters.
- Same number of bits used to represent all characters.
  - 'a' and 'e' occur more frequently than 'q' and 'z'.
- **Potential solution**: use variable-length codes
  - variable number of bits to represent characters when frequency of occurrence is known.
  - short codes for characters that occur frequently.

Characters	Counts
A	4
B	7
C	3
D	2
E	4

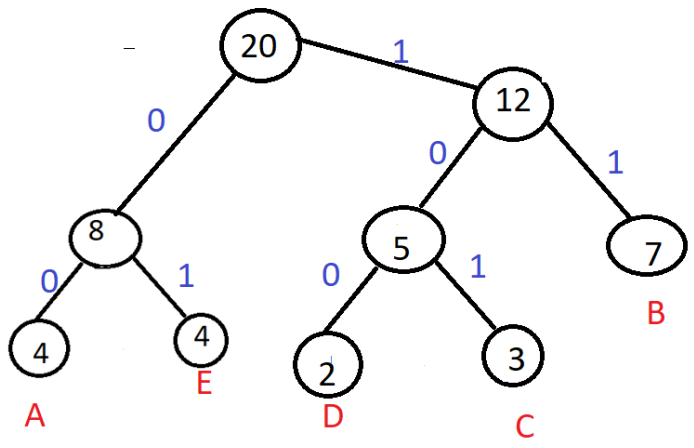








For left subtree assign 0, and for right subtree assign 1.



Characters	Counts	Code
A	4	00
B	7	11
C	3	101
D	2	100
E	4	01

Total = 4x2 + 7x2 + 3x3 + 2x3 + 4x2 = 47

Again, 5x8 + 12 = 52

So, 47+52 = 97 bits.

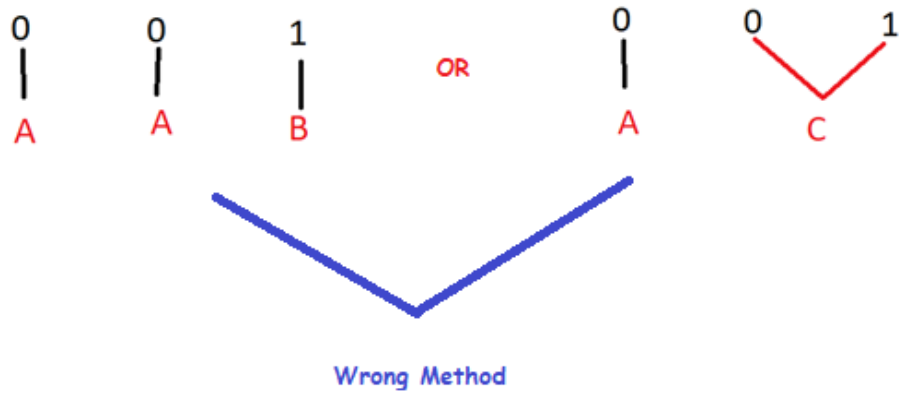
<u>Prefix Code:</u> No code should be prefix code of another code.

Example: A-0, B-1, C-01

Prefix of C

A – 0, B – 1, C – 0 1

- Suppose the message is: 001.
- We can encrypt in in two ways:



Huffman coding always follows the prefix rules.

Time complexity: O (n log n)

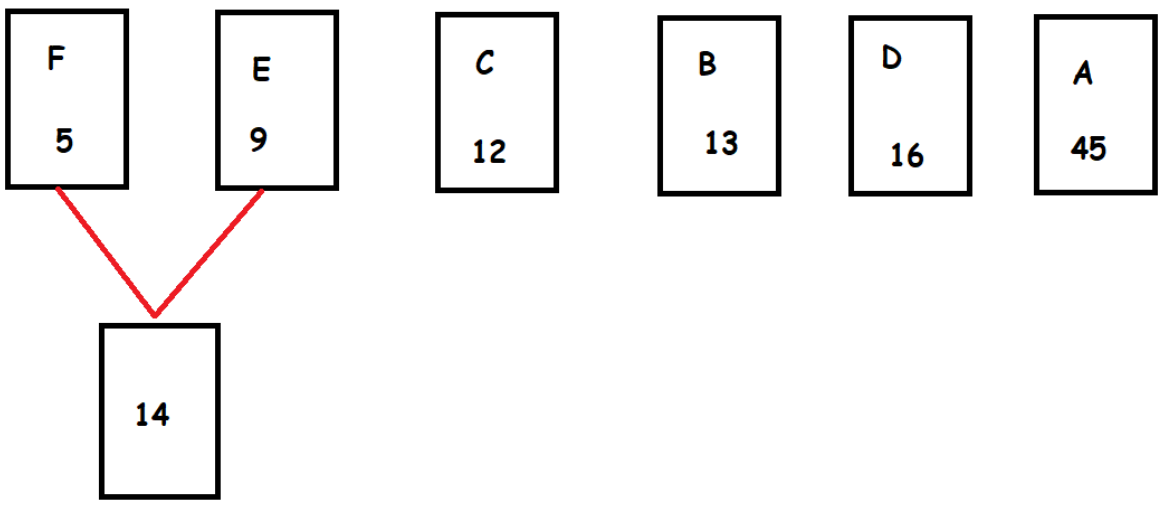
The character with large frequency produces small code and the character with small frequency produces large code.

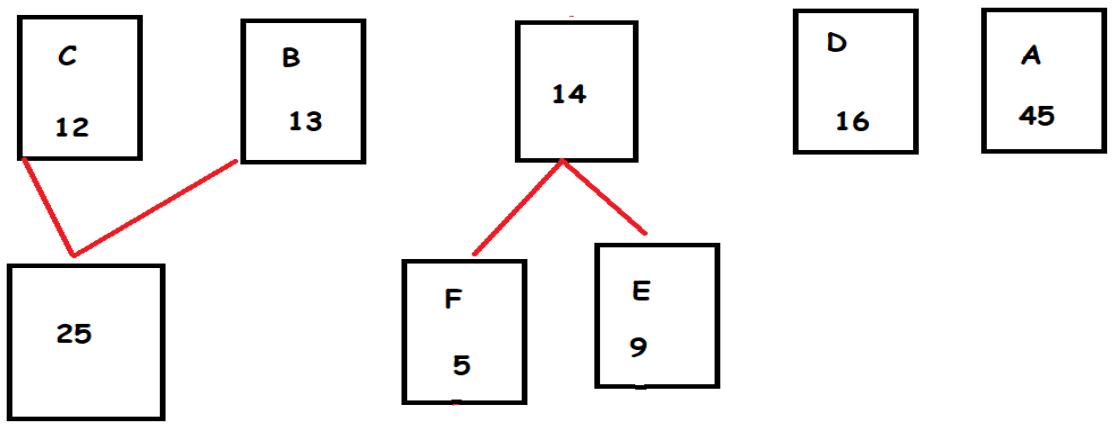
Practice:

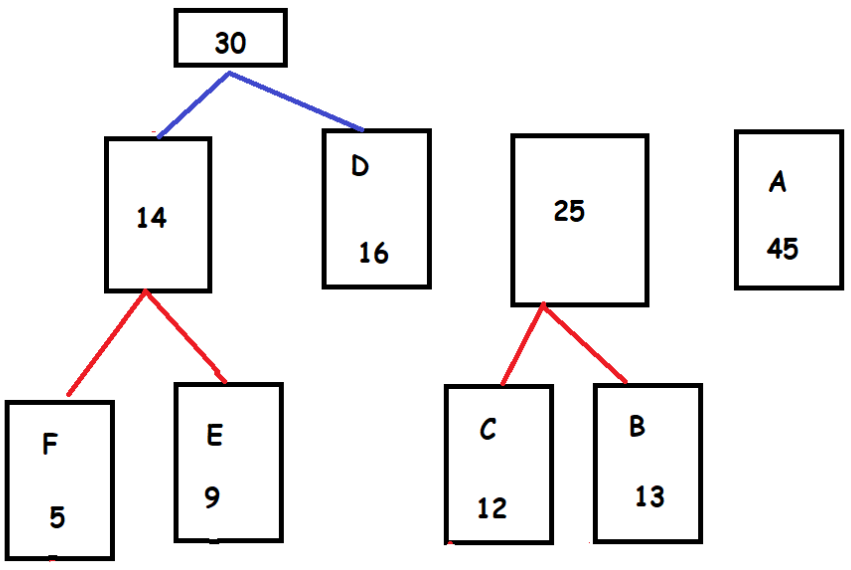
- Suppose that the following characters are given with their corresponding frequency: E-9, F-5, D-16, A-45, C-12, B-13. Using Huffman Coding Algorithm, find the code for each character. Compare the result of variable length code word with fixed length code word.

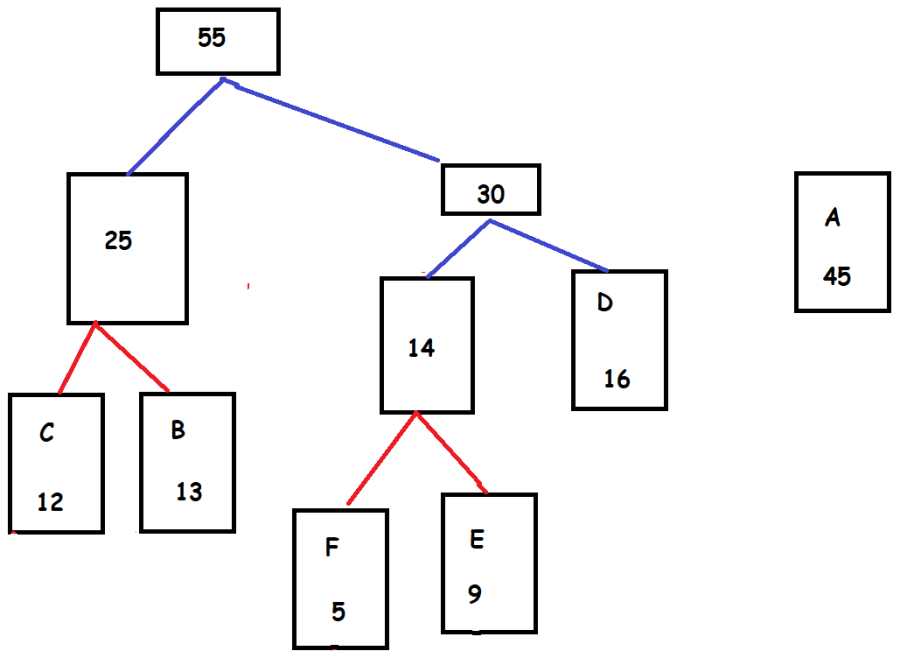
Solution:

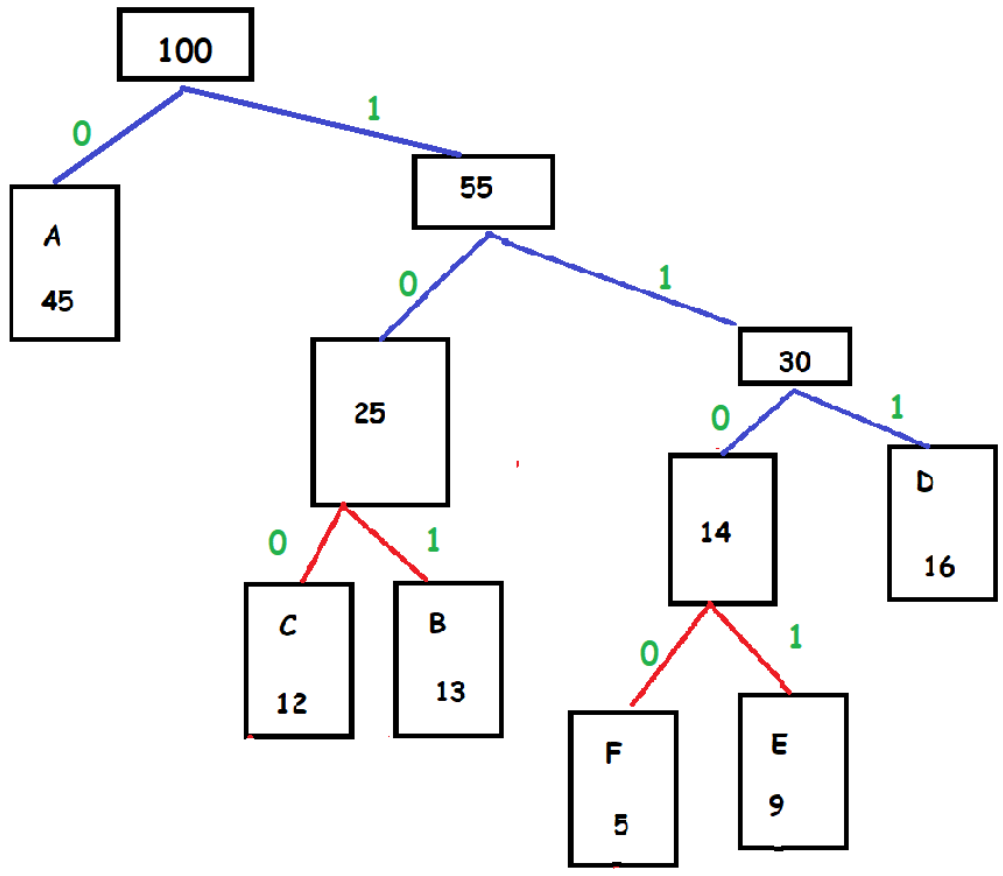
Variable length coding:











A – 0, B – 101, C – 100, D – 111, E – 1101, F – 1100

Total Bits = 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224

For encryption table = 6 * 8 + 18 = 66

Total = 224 + 66 = 290

Fixed length coding:

With 3 bits, 2^3 = 8 combinations are possible. So, A – 000, B-001, C-010, D-011, E-100, F-101

Total Bits = 100 * 3 = 300

For encryption table = 6*8 + 6*3 = 48 + 18 = 66

Total = 300 + 66 = 366

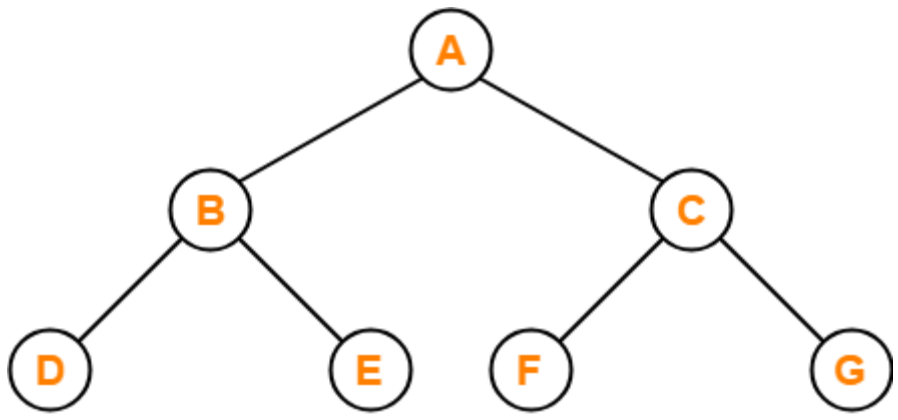
So, variable length code is better.

Graph

Breadth First Traversal

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.

Example-



Level Order Traversal : A , B , C , D , E , F , G

Applications-

- Level order traversal is used to print the data in the same order as stored in the array representation of complete binary tree.

Algorithm (Using Queue):

This algorithm executes a breadth-first search on a

graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).

2. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).

3. Repeat Steps 4 and 5 until QUEUE is empty:

4. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).

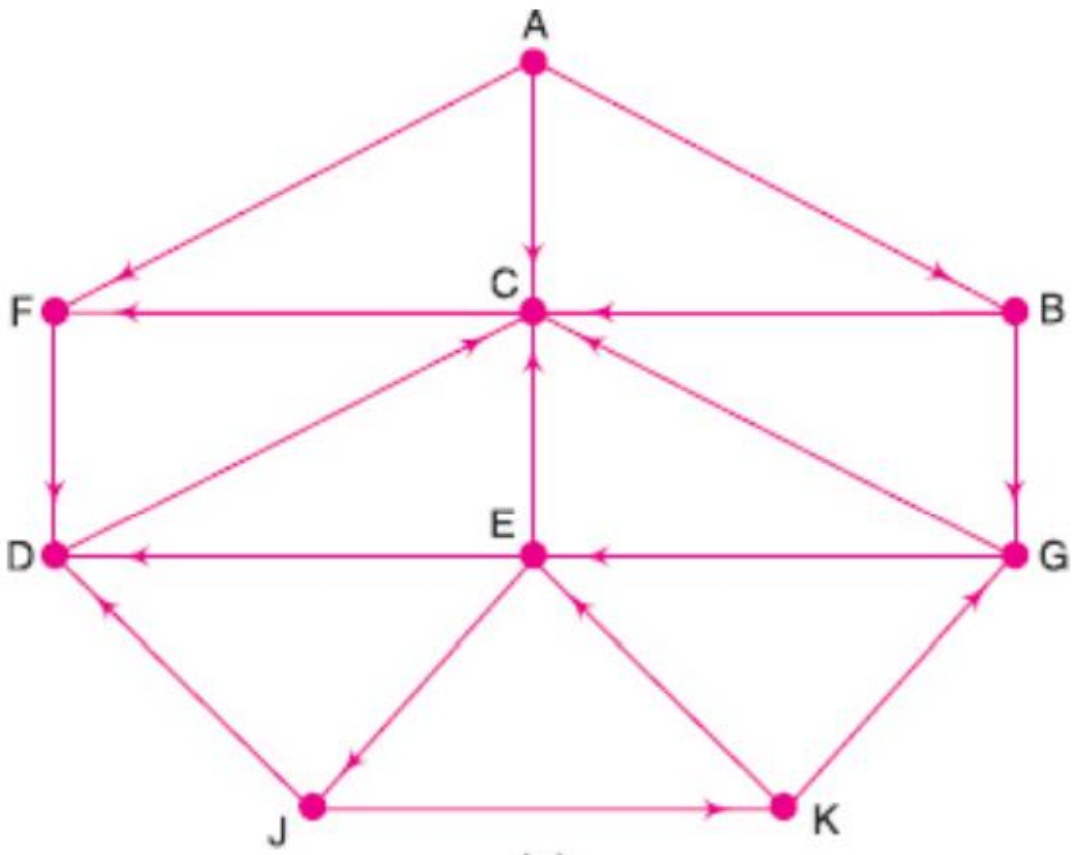
5. Add to the rear of QUEUE all the neighbors of N that are in the steady state (STATUS = 1), andchange their status to the waiting state (STATUS = 2).

[End of Step 3 loop.]

6. Exit.

Example:

Consider the graph G in Fig.(a). (The adjacency lists of the nodes appear in Fig.(b).) Suppose G represents the daily flights between cities of some airline, and suppose we want to fly from city A to city K with the minimum number of stops. In other words, we want the minimum path P from A to J (Where each edge has length 1).



(a)

Adjacency lists	
A:	F, C, B
B:	G, C
C:	F
D:	C
E:	D, C, J
F:	D
G:	C, E
J:	D, K
K:	E, G

(b)

Solution:

1) Insert the origin A to the queue.

	1	2	3	4	5	6	7	8	9	10
Queue:	A									
Origin:	ϕ									

Front = 1
Rear = 1

2) Remove A and insert its' neighboring nodes (Direction should be A to them).

	1	2	3	4	5	6	7	8	9	10
Queue:		F	C	B						
Origin:	ϕ	A	A	A						

Front = 2
Rear = 4

3) Remove F and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:			C	B	D					
Origin:	ϕ	A	A	A	F					

Front = 3
Rear = 5

4) Remove C and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:				B	D					
Origin:	ϕ	A	A	A	F					

Front = 4
Rear = 5

5) Remove B and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:					D	G				
Origin:	ϕ	A	A	A	F	B				

Front = 5
Rear = 6

6) Remove D and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:						G				
Origin:	ϕ	A	A	A	F	B				

Front = 6
Rear = 6

7) Remove G and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:							E			
Origin:	ϕ	A	A	A	F	B	G			

Front = 7
Rear = 7

8) Remove E and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:								J		
Origin:	ϕ	A	A	A	F	B	G	E		

Front = 8
Rear = 8

9) Remove J and insert its' neighboring nodes (if available)

	1	2	3	4	5	6	7	8	9	10
Queue:									K	
Origin:	ϕ	A	A	A	F	B	G	E	J	

Front = 9
Rear = 9

We stop as soon as K is added to QUEUE, since K is our destination. We now backtrack from K, using the array ORIGIN to find the path P. Thus **K ← J ← E ← G ← B ← A** is the required path P.

Depth First Traversal

Algorithm (Using Stack):

This algorithm executes a depth-first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).

2. Push the starting node A onto STACK and change its status to the waiting state (STATUS = 2).

3. Repeat Steps 4 and 5 until STACK is empty.

4. Pop the top node N of STACK. Process N and change its status to the processed state (STATUS = 3).

5. Push onto STACK all the neighbors of N that are still in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

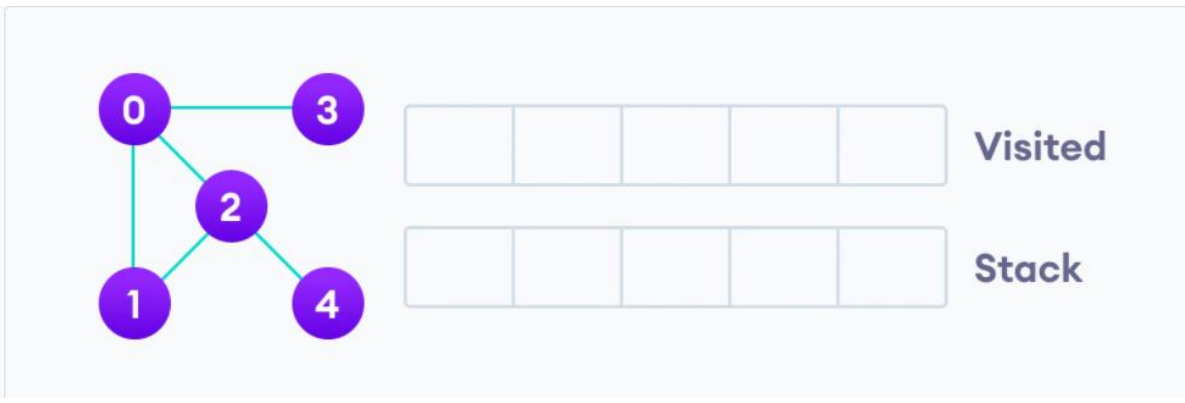
[End of Step 3 loop.]

6. Exit.

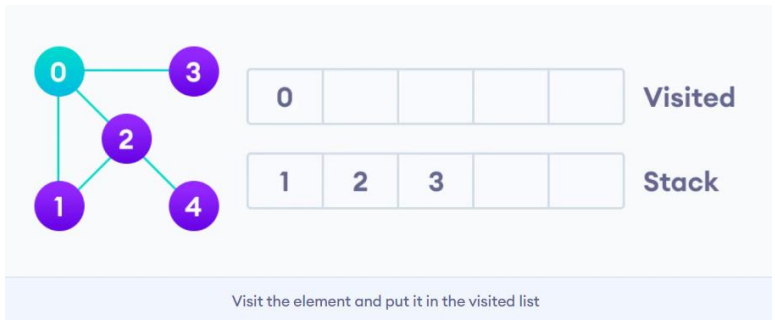
Exercise:

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

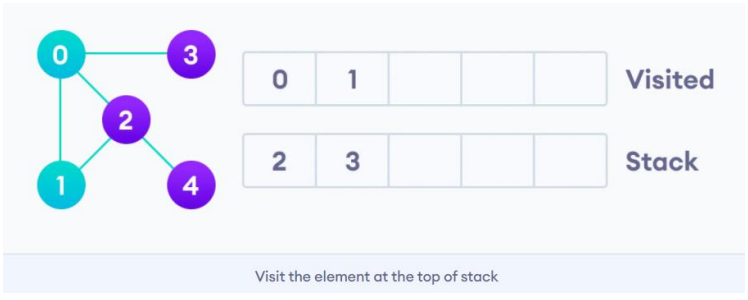
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack. Start by putting it in the Visited list and putting all its adjacent vertices in the stack.

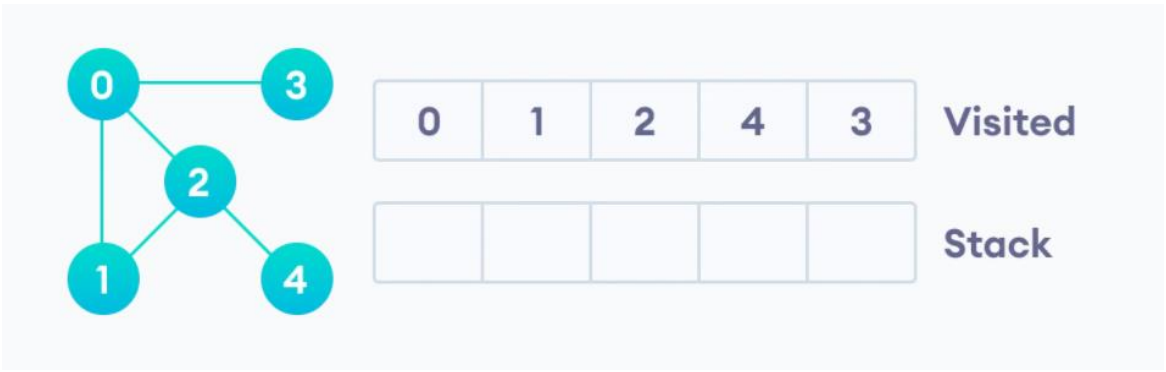


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

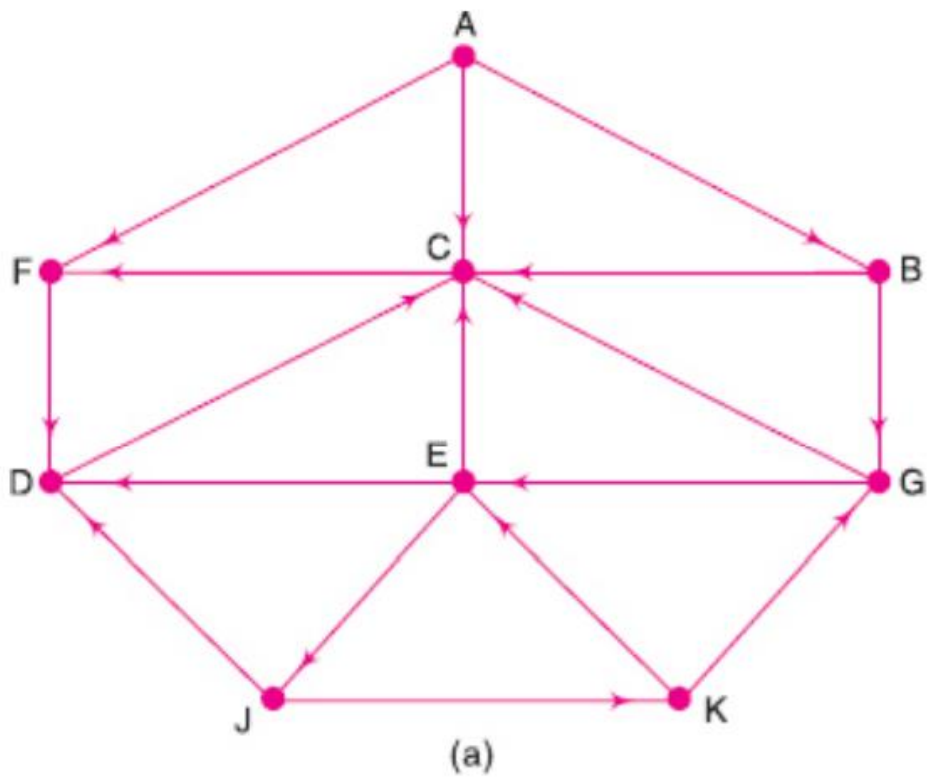


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



Practice: Consider the graph G in Fig. (a). Suppose we want to find and print all the nodes reachable from the node J (including J itself). One way to do this is to use a depth-first search of G starting at the node J.



(a)

Adjacency lists	
A:	F, C, B
B:	G, C
C:	F
D:	C
E:	D, C, J
F:	D
G:	C, E
J:	D, K
K:	E, G

(b)

Solution:

(a) Initially, push J onto the stack as follows:

STACK: J

(b) Pop and print the top element J, and then push the neighbors of J:

Print J STACK: D, K

(c) Pop and print the top element K, and then push the neighbors of K:

Print K STACK: D, E, G

(d) Pop and print the top element G, and then push the neighbors of G:

Print G STACK: D, E, C

(e) Pop and print the top element C, and then push the neighbors of C:

Print C STACK: D, E, F

(f) Pop and print the top element F, and then push the neighbors of F:

Print F STACK: D, E

(g) Pop and print the top element E, and push the neighbors of E:

Print E STACK: D

(h) Pop and print the top element D, and push the neighbors of D:

Print D STACK: _____

The stack is now empty, so the depth-first search of G starting at J is now complete. Accordingly, the nodes which were printed, J, K, G, C, F, E, D are precisely the nodes which are reachable from J.

Sorting and Searching

Worst and Average Case of Sorting Algorithm:

Sorting Algorithm	Worst Case	Average Case	
Bubble Sort	$\frac{n(n-1)}{2}$ = O(n²)	$\frac{n(n-1)}{2}$ = O(n²)	
Quick Sort	$\frac{n(n+3)}{2}$ = O(n²)	1.4 n (log n) = O (log n)	
Heap Sort	3n (log n) = O (log n)	3n (log n) = O (log n)	
Insertion Sort	$\frac{n(n-1)}{2}$ = O(n²)	$\frac{n(n-1)}{2}$ = O(n²)	
Selection Sort	$\frac{n(n-1)}{2}$ = O(n²)	$\frac{n(n-1)}{2}$ = O(n²)	
Merge Sort	O (n log n)	O (n log n)	Extra Memory: O(n)

- **There is no algorithm which can sort n items in time of order less than O (n log n).**

Quicksort Algorithm

- Always pick first element or last element or a random element or median as pivot.

- The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

- **Algorithm for partition:**

```
partition(int a[], int lb, int ub)
{
pivot = a[lb];
start = lb, end = ub;
        while (start < end)
        {
                While (pivot >= a[start]) start++;
                While (pivot < a[end]) end--;
                if (start < end) Swap(a [start], a [end]
        }
        Swap (a[lb], a[end])
return end.
}
```

Partition 1	Pivot	Partition 2

Example:

Suppose S is the following list of 14 alphabetic characters:

(D) A T A S T R U C T U R E (S)

Suppose the characters in S are to be sorted alphabetically. Use the quicksort algorithm to find the final position of the first character D. Beginning with the last character S, scan the list from right to left until finding a character which precedes D alphabetically. It is C. Interchange D and C to obtain the list:

(C) A T A S T R U (D) T U R E S

Beginning with this C, scan the list toward D, i.e., from left to right, until finding a character which succeeds D alphabetically. It is T.

Interchange D and T to obtain the list:

C A (D) A S (T) R U T T U R E S

Beginning with this T, scan the list toward D until finding a character which precedes D. It is A. Interchange D and A to obtain the list:

C A (A) (D) S T R U T T U R E S

Beginning with this A, scan the list toward D until finding a character which succeeds D. There is no such letter. This means D is in its final position. Furthermore, the letters before D form a sub list consisting of all letters preceding D alphabetically. and the letters after D form a sub list consisting of all the letters succeeding D alphabetically, as follows:

C A A (D) S T R U T T U R E S

Sublist Sublist

Sorting S is now reduced to sorting each sub list.

Example: Use quick sort algorithm to find the final position of 10.

10	**15**	**1**	**2**	**9**	**16**	**11**

Insertion Sort Algorithm: (for index 0)

```
insertion_sort(A, n)
{
        for (int i = 1; i < n; i++)
        {
                index = i;
                value = a[i];
                while (index > 0 && a [index - 1] > value)
                {
                        a[index] = a[index - 1];
                        index--;
```

```
        }
        a[index] = value;
    }
}
```

- Apply insertion sort and show the step for 77, 33, 44, 11, 88, 22, 66, 55.

Pass	A [0]	A [1]	A [2]	A [3]	A [4]	A [5]	A [6]	A [7]	A [8]
K = 1	-∞	77	33	44	11	88	22	66	55
K = 2	-∞	77	33	44	11	88	22	66	55
K = 3	-∞	33	77	44	11	88	22	66	55
K = 4	-∞	33	44	77	11	88	22	66	55
K = 5	-∞	11	33	44	77	88	22	66	55
K = 6	-∞	11	33	44	77	88	22	66	55
K = 7	-∞	11	22	33	44	77	88	66	55
K = 8	-∞	11	22	33	44	66	77	88	55
Sorted:	-∞	11	22	33	44	55	66	77	88

Selection Sort Algorithm

```
selection_sort(int a[], int n)
{
    for(i = 0; i < n; i++)
    {
        index_min = i;
        for(j = i + 1; j < n; j++)
        {
            if(a[j] < a[index_min]) index_min = j;
        }
        int temp = a[i];
        a[i] = a[index_min];
        a[index_min] = temp;
    }
}
```

- Apply selection sort and show the step for 77, 33, 44, 11, 88, 22, 66, 55.

Pass	A [1]	A [2]	A [3]	A [4]	A [5]	A [6]	A [7]	A [8]
K = 1, Loc = 4	77	33	44	11	88	22	66	55
K = 1, Loc = 4	11	33	44	77	88	22	66	55
K = 1, Loc = 4	11	22	44	77	88	33	66	55
K = 1, Loc = 4	11	22	33	77	88	44	66	55
K = 1, Loc = 4	11	22	33	44	88	77	66	55
K = 1, Loc = 4	11	22	33	44	55	77	66	88
K = 1, Loc = 4	11	22	33	44	55	66	77	88

Algorithm:

```
Merge (int a[], int l, int mid, int h)
{
        int i, j, k;
        int L [mid - l + 1], R [h - mid];
        for (i = 0; i < mid - l + 1; i++) L[i] = a [l + i];
        for (j = 0; j < h - mid; j++) R[j] = a [mid + 1 + j];
        i = 0, j = 0, k = l;
        while (i < mid - l + 1 && j < h - mid)
        {
                if(L[i] <= R[j]) a[k++] = L[i++];
                else a[k++] = R[j++];
        }
        for (; i < mid - l + 1; i++) a[k++] = L[i];
        for (; j < h - mid; j++) a[k++] = R[j];
}
```

Complexity of Merging: O (n)

Merge Sort Algorithm:

```
merge_sort(int a[], int l, int h)
{
        If (l < h)
        {
                int mid = (l + h) / 2;
                merge_sort(a, l, mid);
```

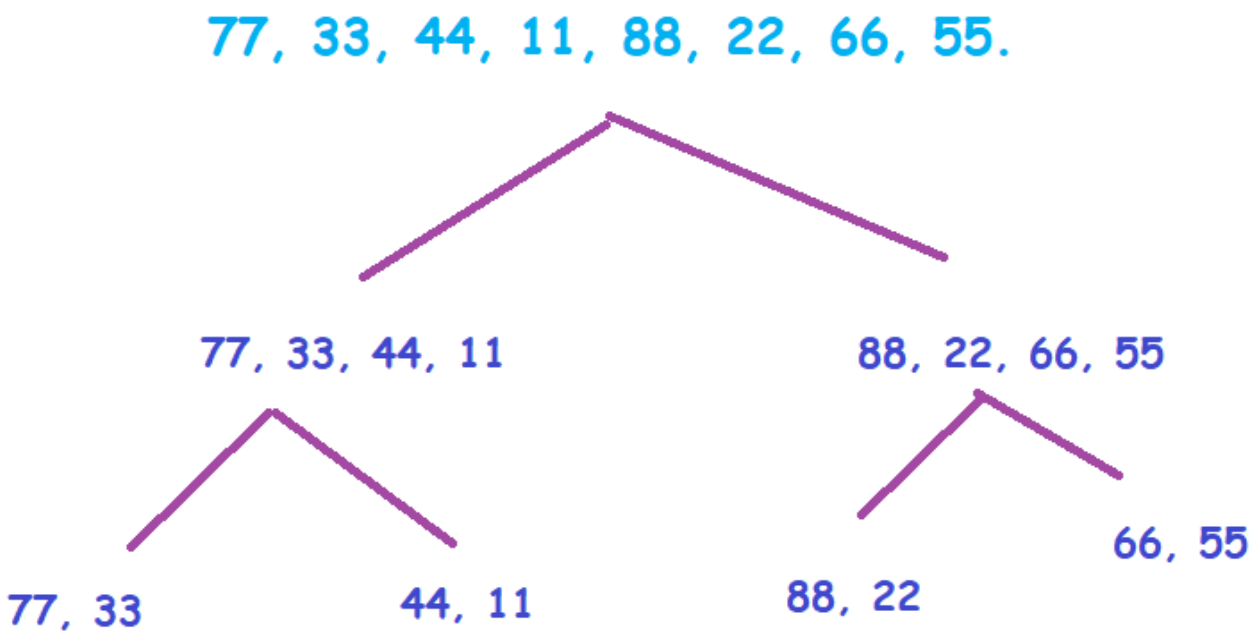
```
            merge_sort(a, mid + 1, h);
            merge (a, l, mid, h);
        }
}
```

Example:

Apply merge sort and show the step for 77, 33, 44, 11, 88, 22, 66, 55.

Steps:

Splitting:



Sorting and Merging: