+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^=	assignment	2	right-to-left
,	Comma	1	left-to-right

Important Properties

- Let us consider the infix expression 2 + 3 * 4 and its postfix equivalent 2 3 4 * +. Notice that between infix and postfix the order of the numbers (or operands) is unchanged. It is 2 3 4 in both cases. But the order of the operators * and + is affected in the two expressions.
- Only one stack is enough to convert an infix expression to postfix expression. The stack that we use in the algorithm will be used to change the order of operators from infix to postfix. The stack we use will only contain operators and the open parentheses symbol '('. Postfix expressions do not contain parentheses. We shall not output the parentheses in the postfix output.

Algorithm

```
a) Create a stack
b) for each character t in the input stream{
       if(t is an operand)
             output t
       else if(t is a right parenthesis){
             Pop and output tokens until a left parenthesis is popped (but not output)
       }
       else // t is an operator or left parenthesis{
             pop and output tokens until one of lower priority than t is encountered or a left parenthesis is
             encountered or the stack is empty
             Push t
       }
   }
c) pop and output tokens until the stack is empty
```

For better understanding let us trace out some example: A * B- (C + D) + E

Input Character	Operation on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C

D			AB*CD
)	Pop and append to postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End of input	Pop till empty		AB*CD+-E+

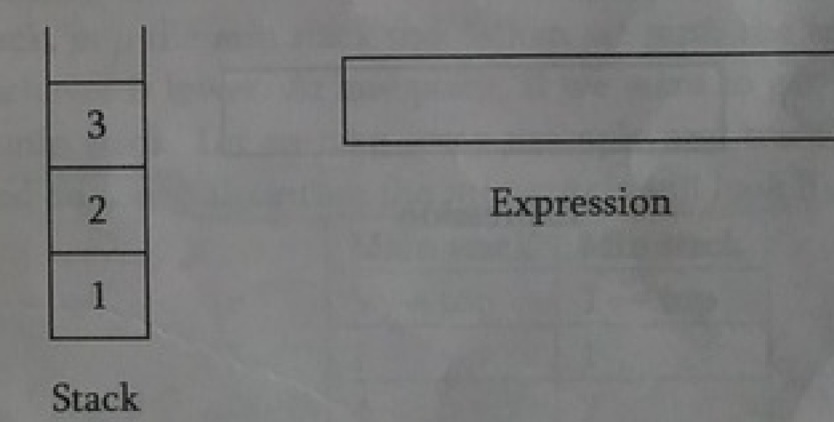
Problem-3 Discuss postfix evaluation using stacks?

Solution:

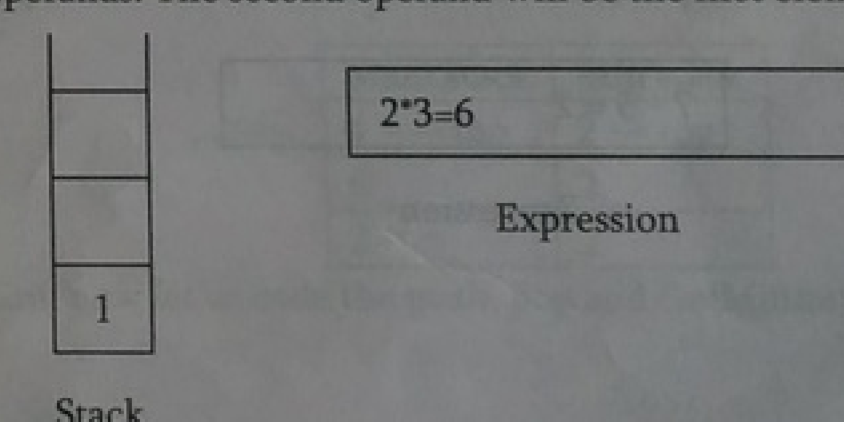
Algorithm

1. Scan the Postfix string from left to right.
2. Initialize an empty stack.
3. Repeat the below steps 4 and 5 till all the characters are scanned.
4. If the scanned character is an operand, push it onto the stack.
5. If the scanned character is an operator, and if the operator is unary operator then pop an element from the stack. If the operator is binary operator then pop two elements from the stack. After popping the elements, apply the operator to those popped elements. Let the result of this operation be retVal onto the stack.
6. After all characters are scanned, we will have only one element in the stack.
7. Return top of the stack as result.

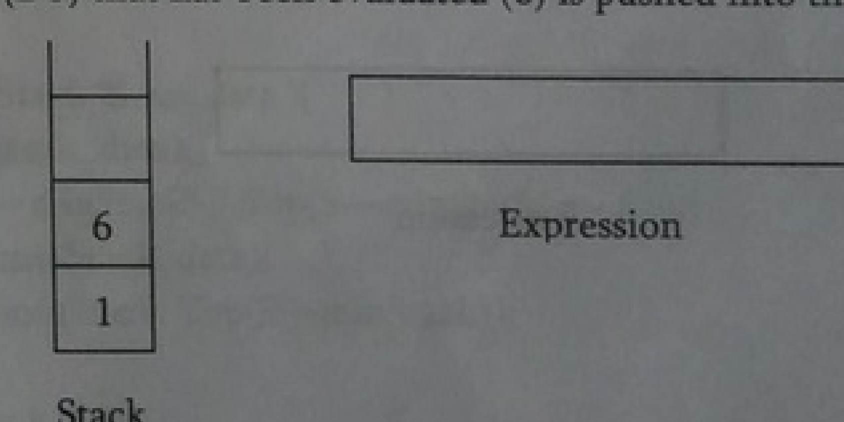
Example: Let us see how the above algorithm works using an example. Assume that the postfix string is 123*+5-. Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



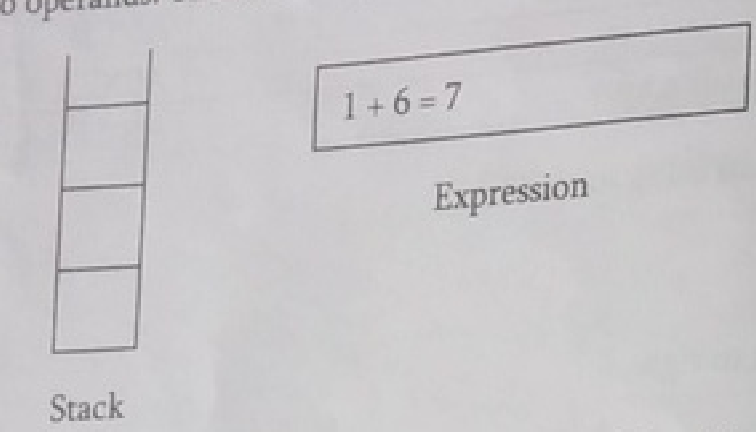
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



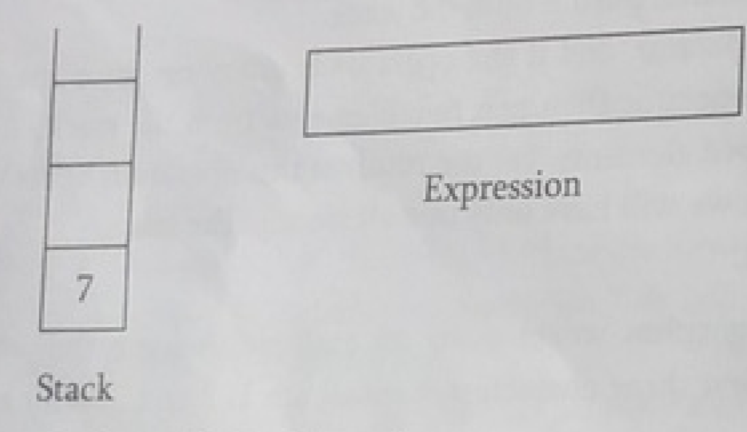
The value of the expression (2*3) that has been evaluated (6) is pushed into the stack.



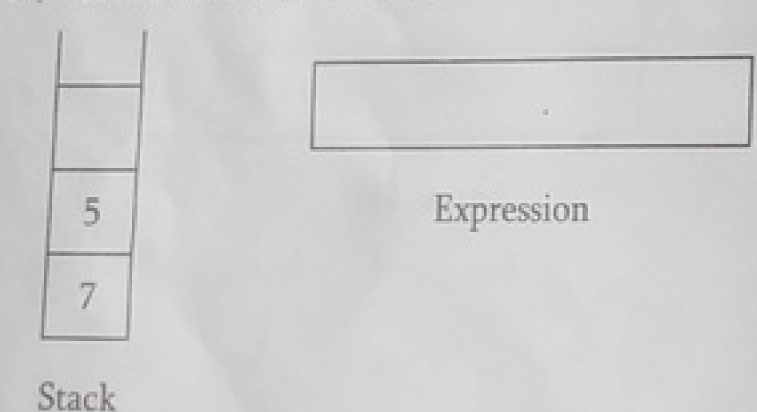
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



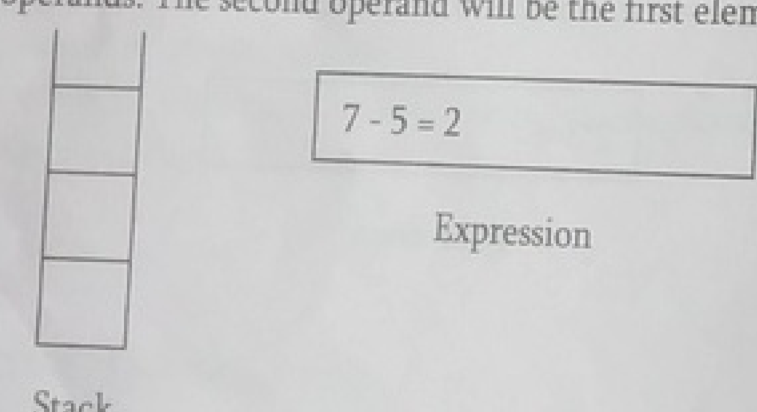
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



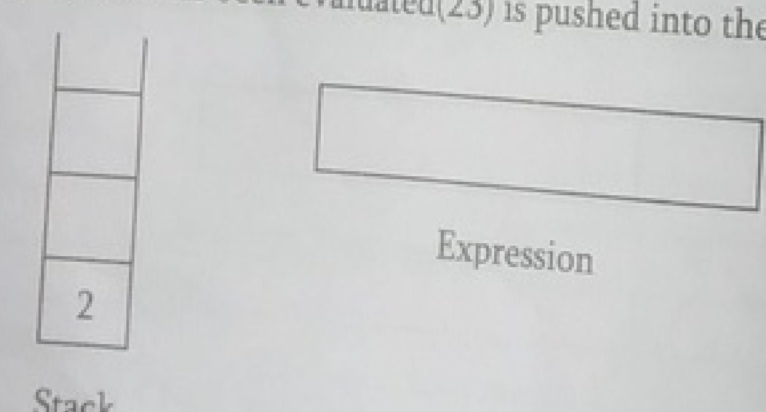
Next character scanned is "5", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-5) that has been evaluated(23) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned. End result:

- Postfix String : 123*+5-
- Result : 2

Problem-4 Can we evaluate the infix expression with stacks in one pass?

Solution: Using 2 stacks we can evaluate an infix expression in 1 pass without converting to postfix.

Algorithm

1) Create an empty operator stack
2) Create an empty operand stack
3) For each token in the input string
   a. Get the next token in the infix string
   b. If next token is an operand, place it on the operand stack
   c. If next token is an operator
      i. Evaluate the operator (next op)
4) While operator stack is not empty, pop operator and operands (left and right), evaluate left operator right and push result onto operand stack
5) Pop result from operator stack

Problem-5 How to design a stack such that GetMinimum( ) should be O(1)?

Solution: Take an auxiliary stack which maintains the minimum of all values in the stack. Also, assume that, each element of the stack is less than its below elements. For simplicity let us call the auxiliary stack as *min stack*.

When we *pop* the main stack, *pop* the min stack too. When we push the main stack, push either the new element or the current minimum, whichever is lower. At any point, if we want to get the minimum then we just need to return the top element from the min stack. Let us take some example and trace out. Initially let us assume that we have pushed 2, 6, 4, 1 and 5. Based on above algorithm the *min stack* will look like:

TOP

Main stack	Min stack
5 → top	1 → top
1	1
4	2
6	2
2	2

After popping twice we get:

Main stack	Min stack
4 → top	2 → top
6	2
2	2

Based on the above discussion, now let us code the push, pop and GetMinimum() operations.

```
struct AdvancedStack{
        struct Stack elementStack;
        struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data ){
        Push (S→elementStack, data);
        if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
                Push (S→minStack, data);
        else    Push (S→minStack, Top(S→minStack));
}
int Pop(struct AdvancedStack *S ){
```

```
    int temp;
    if(IsEmptyStack(S→elementStack))
        return -1;
    temp = Pop (S→elementStack);
    Pop (S→minStack);
    return temp;
}
int GetMinimum(struct AdvancedStack *S){
    return Top(S→minStack);
}
struct AdvancedStack *CreateAdvancedStack(){
    struct AdvancedStack *S = (struct AdvancedStack *)malloc(sizeof(struct AdvancedStack));
    if(!S)
        return NULL;
    S→elementStack = CreateStack();
    S→minStack = CreateStack();
    return S;
}
```

Time complexity: O(1). Space complexity: O(n) [for Min stack]. This algorithm has much better space usage if we rarely get a "new minimum or equal".

Problem-6 For the Problem-5 is it possible to improve the space complexity?

Solution: Yes. The main problem of previous approach is, for each push operation we are pushing the element on to *min stack* also (either the new element or existing minimum element). That means, we are pushing the duplicate minimum elements on to the stack.

Now, let us change the above algorithm to improve the space complexity. We still have the min stack, but we only *pop* from it when the value we pop from the main stack is equal to the one on the min stack. We only *push* to the min stack when the value being pushed onto the main stack is less than *or equa* to the current min value. In this modified algorithm also, if we want to get the minimum then we just need to return the top element from the min stack. For example, taking the original version and pushing 1 again, we'd get:

Main stack	Min stack
1 → top	
5	
1	
4	1 → top
6	1
2	2

Popping from the above pops from both stacks because 1 == 1, leaving:

Main stack	Min stack
5 → top	
1	
4	
6	1 → top
2	2

Popping again *only* pops from the main stack, because 5 > 1:

Main stack	Min stack
1 → top	
4	
6	1 → top

2	2

Popping again pops both stacks because 1 == 1:

Main stack	Min stack
4 → top	
6	
2	2 → top

Note: The difference is only in push & pop operations.

```
struct AdvancedStack {
    struct Stack elementStack;
    struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data){
    Push (S→elementStack, data);
    if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
        Push (S→minStack, data);
}
int Pop(struct AdvancedStack *S ){
    int temp;
    if(IsEmptyStack(S→elementStack)) return -1;
    temp = Top (S→elementStack);
    if(Top(S→ minStack) == Pop(S→elementStack))
        Pop (S→ minStack);
    return temp;
}
int GetMinimum(struct AdvancedStack *S){
    return Top(S→minStack);
}
Struct AdvancedStack * AdvancedStack(){
    struct AdvancedStack *S = (struct AdvancedStack) malloc (sizeof (struct AdvancedStack));
    if(!S) return NULL;
    S→elementStack = CreateStack();
    S→minStack = CreateStack();
    return S;
}
```

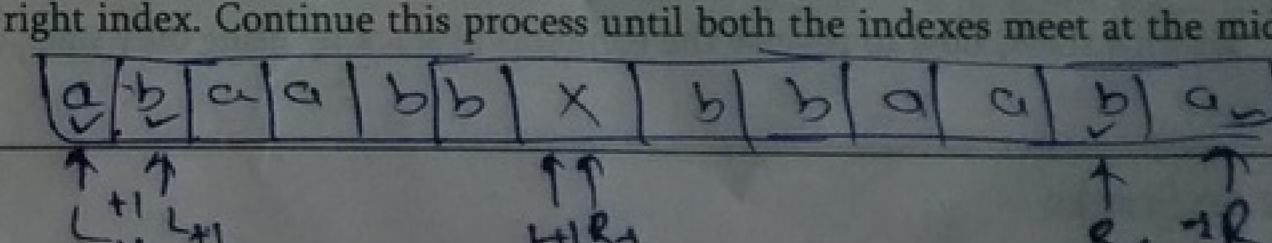
Time complexity: O(1). Space complexity: O(n) [for Min stack]. But this algorithm has much better space usage if we rarely get a "new minimum or equal".

Problem-7 For a given array with n symbols how many stack permutations are possible?

Solution: The number of stack permutations with n symbols is represented by *Catalan number* and we will discuss this in *Dynamic Programming* chapter.

Problem-8 Given an array of characters formed with a's and b's. The string is marked with special character X which represents the middle of the list (for example: ababa…ababXbabab…..baaa). Check whether the string is palindrome or not?

Solution: This is one of the simplest algorithms. What we do is, start two indexes one at the beginning of the string and other at the ending of the string. Each time compare whether the values at both the indexes are same or not. If the values are not same then we say that the given string is not a palindrome. If the values are same then increment the left index and decrement the right index. Continue this process until both the indexes meet at the middle (at X) or if the string is not palindrome.



```
int IsPalindrome(char *A){
    int i=0, j = strlen(A)-1;
    while(i < j && A[i] == A[j]) {
        i++;
        j--;
    }
    if(i < j ) {
        printf("Not a Palindrome");
        return 0;
    }
    else {  printf("Palindrome");
        return 1;
    }
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problem-9 For the Problem-8, if the input is in singly linked list then how do we check whether the list elements form a palindrome or not? (That means, moving backward is not possible).

Solution: Refer *Linked Lists* chapter.

Problem-10 Can we solve Problem-8 using stacks?

Solution: Yes.

Algorithm

- Traverse the list till we encounter X as input element.
- During the traversal push all the elements (until X) on to the stack.
- For the second half of the list, compare each elements content with top of the stack. If they are same then pop the stack and go to the next element in the input list.
- If they are not same then the given string is not a palindrome.
- Continue this process until the stack is empty or the string is not a palindrome.

```
int IsPalindrome(char *A){
    int i=0;
    struct Stack S= CreateStack();
    while(A[i] != 'X') {
        Push(S, A[i]);
        i++;
    }
    i++;
    while(A[i]) {
        if(IsEmptyStack(S) ||A[i] != Pop(S)) {
            printf("Not a Palindrome");
            return 0;
        }
        i++;
    }
    return IsEmptyStack(S);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n/2) \approx O(n)$.

Problem-11 Given a stack, how to reverse the elements of stack by using only stack operations (push & pop)?

Solution: Algorithm

- First pop all the elements of the stack till it becomes empty.
- For each upward step in recursion, insert the element at the bottom of stack.

```
void ReverseStack(struct Stack *S){
    int data;
    if(IsEmptyStack(S)) return;
    data = Pop(S);
    ReverseStack(S);
    InsertAtBottom(S, data);
}
void InsertAtBottom(struct Stack *S, int data){
    int temp;
    if(IsEmptyStack(S)) {
        Push(S, data);
        return;
    }
    temp = Pop(S);
    InsertAtBottom(S, data);
    Push(S, temp);
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(n)$, for recursive stack.

Problem-12 Show how to implement one queue efficiently using two stacks. Analyze the running time of the queue operations.

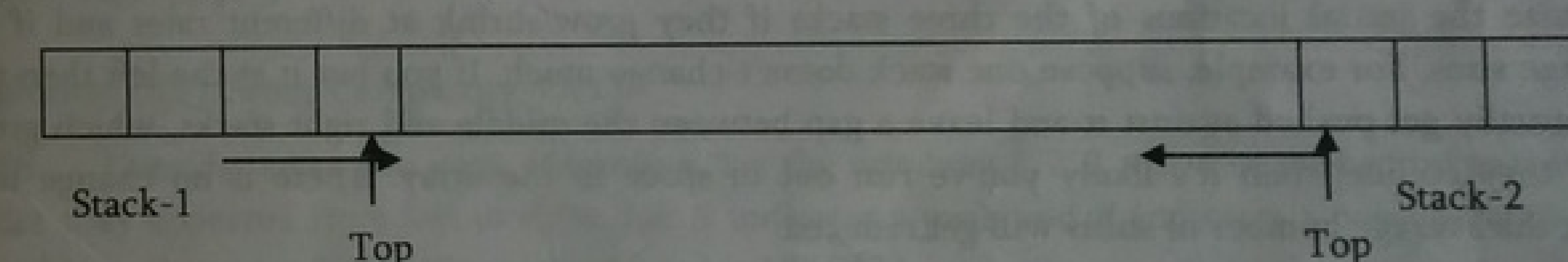
Solution: Refer *Queues* chapter.

Problem-13 Show how to implement one stack efficiently using two queues. Analyze the running time of the stack operations.

Solution: Refer *Queues* chapter.

Problem-14 How do we implement 2 stacks using only one array? Our stack routines should not indicate an exception unless every slot in the array is used?

Solution:



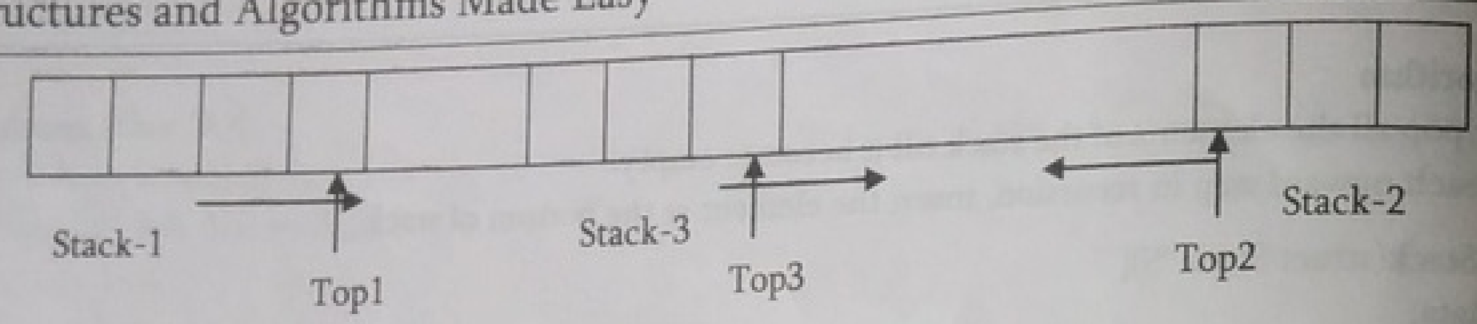
Algorithm:

- Start two indexes one at the left end and other at the right end.
- The left index simulates the first stack and the right index simulates the second stack.
- If we want to push an element into the first stack then put the element at left index.
- Similarly, if we want to push an element into the second stack then put the element at right index.
- First stack gets grows towards right, second stack grows towards left.

Time Complexity of push and pop for both stacks is $O(1)$. Space Complexity is $O(1)$.

Problem-15 3 stacks in one array: How to implement 3 stacks in one array?

Solution: For this problem, there could be other way of solving it. Below is one such possibility and it works as long as there is an empty space in the array.



To implement 3 stacks we keep the following information.

- The index of the first stack (Top1): this indicates the size of the first stack.
- The index of the second stack (Top2): this indicates the size of the second stack.
- Starting index of the third stack (base address of third stack).
- Top index of the third stack.

Now, let us define the push and pop operations for this implementation.

Pushing:

- For pushing on to the first stack, we need to see if adding a new element causes it to bump into the third stack. If so, try to shift the third stack upwards. Insert the new element at (start1 + Top1).
- For pushing to the second stack, we need to see if adding a new element causes it to bump into the third stack. If so, try to shift the third stack downward. Insert the new element at (start2 – Top2).
- When pushing to the third stack, see if it bumps the second stack. If so, try to shift the third stack downward and try pushing again. Insert the new element at (start3 + Top3).

Time Complexity: $O(n)$. Since, we may need to adjust the third stack. Space Complexity: $O(1)$.

Popping: For popping, we don't need to shift, just decrement the size of the appropriate stack.

Time Complexity: $O(1)$. Space Complexity: $O(1)$.

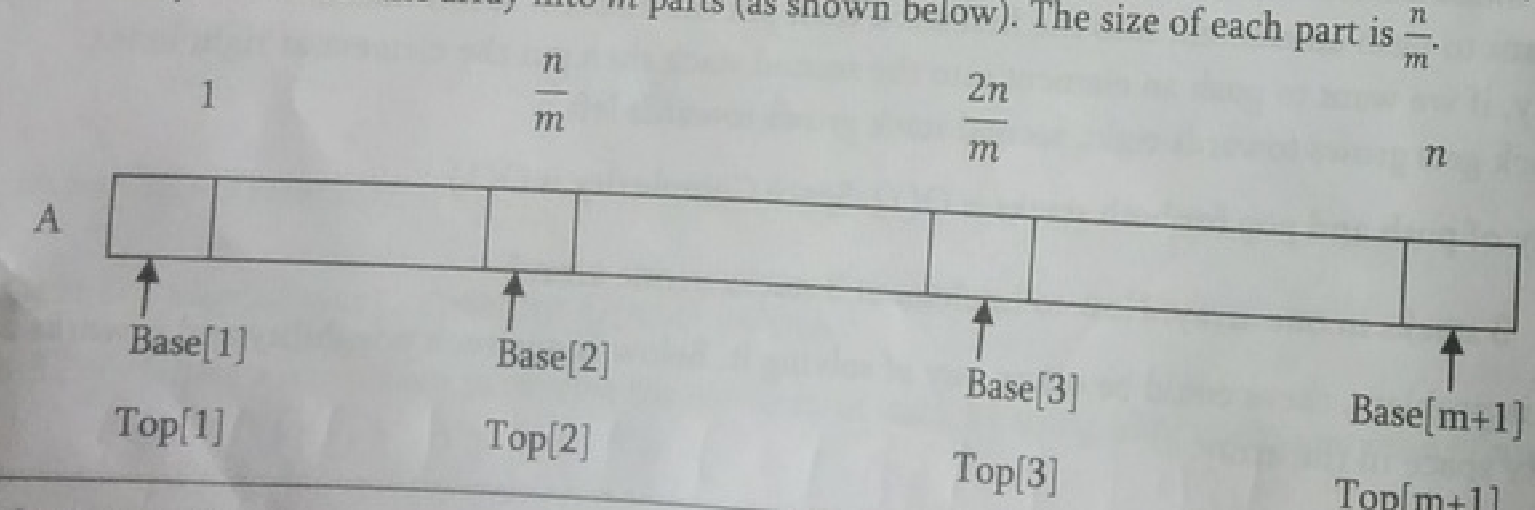
Problem-16 For Problem-15, is there any other way implementing middle stack?

Solution: Yes. When either the left stack (which grows to the right) or the right stack (which grows to the left) bumps into the middle stack, we need to shift the entire middle stack to make room. The same thing happens if a push on the middle stack causes it to bump into the right stack. To solve the above problem (number of shifts) what we can do is, alternating pushes could be added at alternating sides of the middle list (For example, even elements are pushed to the left, odd elements are pushed to the right).This would keep the middle stack balanced in the center of the array but it would still need to be shifted when it bumps into the left or right stack, whether by growing on its own or by the growth of a neighboring stack.

We can optimize the initial locations of the three stacks if they grow/shrink at different rates and if they have different average sizes. For example, suppose one stack doesn't change much. If you put it at the left then the middle stack will eventually get pushed against it and leave a gap between the middle and right stacks, which grow toward each other. If they collide, then it's likely you've run out of space in the array. There is no change in the time complexity but the average number of shifts will get reduced.

Problem-17 Multiple (m) stacks in one array: As similar to Problem-15, what if we want to implement m stacks in one array?

Solution: Let us assume that array indexes are from 1 to n. As similar to the discussion of Problem-15, to implement m stacks in one array, we divide the array into m parts (as shown below). The size of each part is $\frac{n}{m}$.



From the above representation we can see that, first stack is starting at index 1 (starting index is stored in Base[1]), second stack is starting at index $\frac{n}{m}$ (starting index is stored in Base[2]), third stack is starting at index $\frac{2n}{m}$ (starting index is stored in Base[3]) and so on. Similar to *Base* array, let us assume that *Top* array stores the top indexes for each of the stack. Consider the following terminology for the discussion.

- Top[i], for $1 \le i \le m$ will point to the topmost element of the stack i.
- If Base[i] == Top[i], then we can say the stack i is empty.
- If Top[i] == Base[i+1], then we can say the stack i is full.
  Initially Base[i] = Top[i] = $\frac{n}{m}(i-1)$, for $1 \le i \le m$.
- The i^{th}stack grows from Base[i]+1 to Base[i+1].

Pushing on to i^{th} stack:

1) For pushing on to the i^{th} stack, we check whether top of i^{th} stack is pointing to Base[i+1] (this case defines that i^{th} stack is full). That means, we need to see if adding a new element causes it to bump into the $i+1^{th}$ stack. If so, try to shift the stacks from $i+1^{th}$ stack to m^{th} stack towards right. Insert the new element at (Base[i] + Top[i]).
2) If right shifting is not possible then try shifting the stacks from 1 to $i-1^{th}$ stack towards left.
3) If both of them are not possible then we can say that all stacks are full.

```
void Push(int StackID, int data) {
    if(Top[i] == Base[i+1])
        Print i^th Stack is full and does the necessary action (shifting);
    Top[i] = Top[i]+1;
    A[Top[i]] = data;
}
```

Time Complexity: $O(n)$. Since, we may need to adjust the stacks. Space Complexity: $O(1)$.

Popping from i^{th} stack: For popping, we don't need to shift, just decrement the size of the appropriate stack. The only case to check is stack empty case.

```
int Pop(int StackID) {
    if(Top[i] == Base[i])
        Print i^th Stack is empty;
    return A[Top[i]--];
}
```

Time Complexity: $O(1)$. Space Complexity: $O(1)$.

Problem-18 Consider an empty stack of integers. Let the numbers $1, 2, 3, 4, 5, 6$ be pushed on to this stack only in the order they appeared from left to right. Let S indicates a push and X indicates a pop operation. Can they be permuted in to the order 325641(output) and order 154623?

Solution: SSSXXSSXSXXX outputs 325641. 154623 cannot be output as 2 is pushed much before 3 so can appear only after 3 is output.

Problem-19 Earlier of this chapter, we have seen that, for dynamic array implementation of stack, we have used repeated doubling approach. For the same problem what is the complexity if we create a new array whose size is $n + K$ instead of doubling?

Solution: Let us assume that the initial stack size is 0. For simplicity let us assume that $K = 10$. For inserting the element we create a new array whose size is $0 + 10 = 10$. Similarly, after 10 elements we again create a new array whose size is $10 + 10 = 20$ and this process continues at values: $30, 40$ ... That means, for a given n value, we are creating the new arrays at: $\frac{n}{10}, \frac{n}{20}, \frac{n}{30}, \frac{n}{40}$ ... The total number of copy operations are:

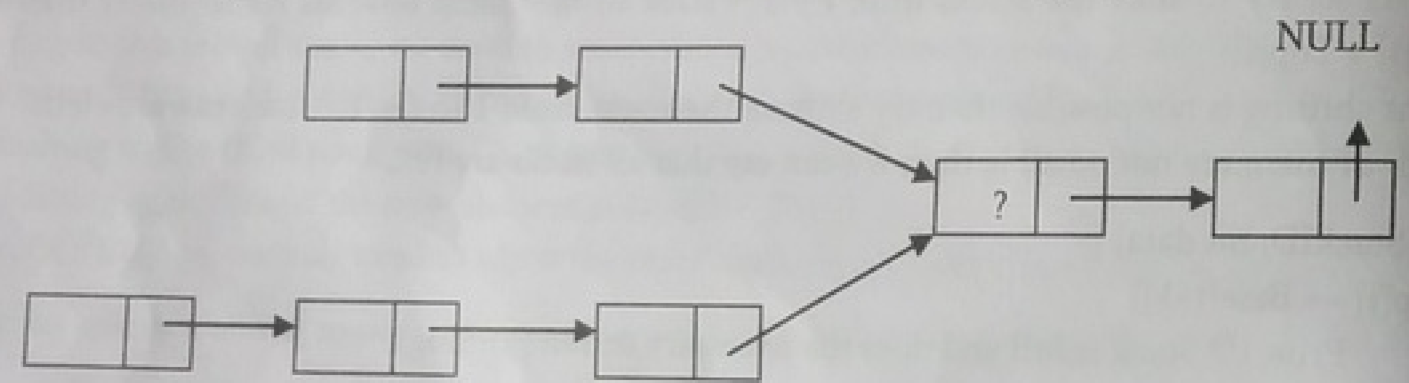
$$= \frac{n}{10} + \frac{n}{20} + \frac{n}{30} + \cdots 1 = \frac{n}{10}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right) = \frac{n}{10} logn \approx O(nlogn)$$

If we are performing n push operations, the cost of per operation is $O(logn)$.

Problem-20 Given a string containing n S's and n X's where S indicates a push operation and X indicates a pop operation, and with the stack initially empty, Formulate a rule to check whether a given string S of operations is admissible or not?

Solution: Given a string of length $2n$, we wish to check whether the given string of operations is permissible or not with respect to its functioning on a stack. The only restricted operation is pop whose prior requirement is that the stack should not be empty. So while traversing the string from left to right, prior to any pop the stack shouldn't be empty which means the no of S's is always greater than or equal to that of X's. Hence the condition is at any stage of processing of the string, number of push operations (S) should be greater than number of pop operations (X).

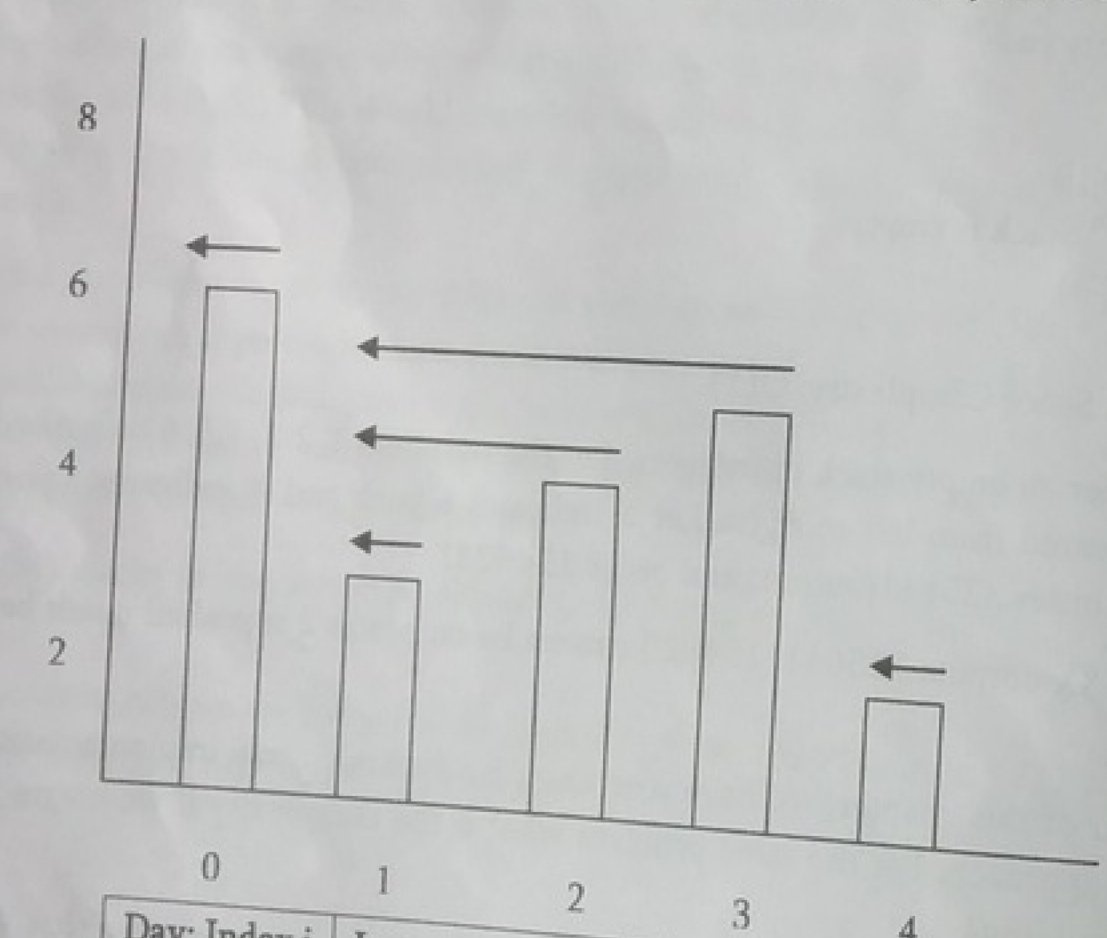
Problem-21 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the list before they intersect are unknown and both list may have it different. $List1$ may have n nodes before it reaches intersection point and $List2$ might have m nodes before it reaches intersection point where m and n may be $m = n, m < n$ or $m > n$. Can we find the merging point using stacks?



Solution: Yes. For algorithm refer *Linked Lists* chapter.

Problem-22 **Finding Spans:** Given an array A, the span $S[i]$ of $A[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and such that $A[j] \le A[i]$?
Otherway of asking: Given an array A of integers, find the maximum of $j - i$ subjected to the constraint of $A[i] < A[j]$.

Solution:



Day: Index i	Input Array A[i]	S[i]: Span of A[i]
0	6	1
1	3	1
2	4	2
3	5	3
4	2	1

This is a very common problem in stock markets to find the peaks. Spans have applications to financial analysis (E.g., stock at 52-week high). The span of a stock price on a certain day, i, is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on i. As an example, let us consider the table and the corresponding spans diagram. In the figure the arrows indicates the length of the spans. Now, let us concentrate on the algorithm for finding the spans. One simple way is, each day, check how many contiguous days are with less stock price than current price.

```
Algorithm: FindingSpans(int A[], int n) {
        //Input: array A of n integers, Output: array S of spans of A
        int i, j, S[n]; //new array of n integers;
        for (i = 0; i < n; i++) {                          //Executes n times
            j = 1;                                          n
            while j <= i && A[i] > A[i-j]                   1 + 2 + ...+ (n – 1)
                    j = j + 1;                              1 + 2 + ...+ (n – 1)
            S[i] = j;                                       n
        }
        return S;                                           1
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

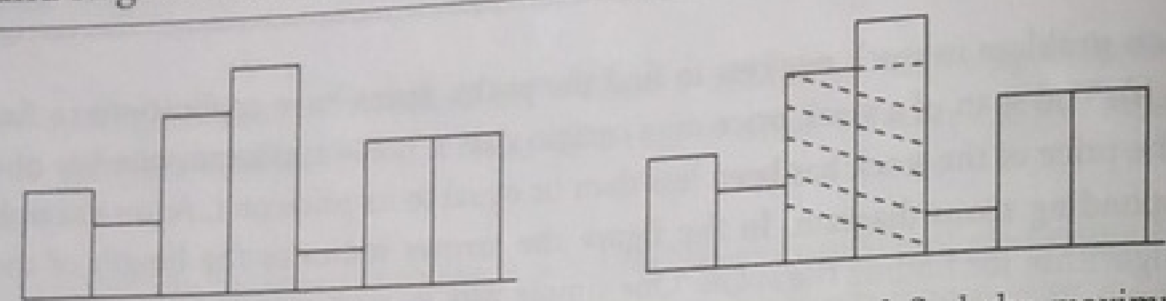
Problem-23 Can we improve the complexity of Problem-22?

Solution: From the above example, we can see that span $S[i]$ on day i can be easily calculated if we know the closest day preceding i, such that the price is greater than on that day than the price on day i. Let us call such a day as P. If such a day exists then the span is now defined as $S[i] = i - P$.

```
Algorithm: FindingSpans(int A[], int n) {
        struct stack *D = CreateStack();
        int P;
        for (int i = 0 i< n; i++) {
            while (!IsEmptyStack(D)) {
                if(A[i] > A[Top(D)])
                    Pop(D);
            }
            if(IsEmptyStack(D))
                P = -1;
            else  P = Top(D);
            S[i] = i-P;
            Push(D, i);
        }
        return S;
}
```

Time Complexity: Each index of the array is pushed into the stack exactly one and also popped from the stack at most once. The statements in the while loop are executed at most n times. Even though the algorithm has nested loops, the complexity is $O(n)$ as the inner loop is executing only n times during the course of algorithm (trace out an example and see how many times the inner loop is becoming success). Space Complexity: $O(n)$ [for stack].

Problem-24 **Largest rectangle under histogram:** A histogram is a polygon composed of a sequence of rectangles aligned at a common base line. For simplicity, assume that the rectangles are having equal widths but may have different heights. For example, the figure on the left shows the histogram that consists of rectangles with the heights 3, 2, 5, 6, 1, 4, 4, measured in units where 1 is the width of the rectangles. Here our problem is: given an array with heights of rectangles (assuming width is 1), we need to find the largest rectabgle possible. For the given example the largest rectangle is the shared part.



Solution: A straightforward answer is to go for each bar in the histogram and find the maximum possible area in histogram for it. Finally, find the maximum of these values. This will require $O(n^2)$.

Problem-25 For Problem-24, can we improve the time complexity?

Solution: Linear search using a stack of incomplete subproblems: There are many ways of solving this problem. *Judge* has given a nice algorithm for this problem which is based on stack. Process the elements in left-to-right order and maintain a stack of information about started but yet unfinished sub histograms.

If the stack is empty, open a new subproblem by pushing the element onto the stack. Otherwise compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element. If the new one is less, we finish the topmost subproblem by updating the maximum area with respect to the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element. This way, all subproblems are finished until the stack becomes empty, or its top element is less than or equal to the new element, leading to the actions described above. If all elements have been processed, and the stack is not yet empty, we finish the remaining subproblems by updating the maximum area with respect to to the elements at the top.

```
struct StackItem {
        int height;
        int index;
};
int MaxRectangleArea(int A[], int n) {
        int i, maxArea=-1, top = -1, left, currentArea;
        struct StackItem *S = (struct StackItem *) malloc(sizeof(struct StackItem) * n);
        for(i=0; i<=n; i++) {
                while(top >= 0 && (i==n || S[top]→height > A[i])) {
                        if(top > 0)
                                left = S[top-1]→index;
                        else    left = -1;
                        currentArea = (i - left-1) * S[top]→height;
                        --top;
                        if(currentArea > maxArea)
                                maxArea = currentArea;
                }
                if(i<n) { ++top;
                        S[top]→height = A[i];
                        S[top]→index = i;
                }
        }
        return maxArea;
}
```

In first impression, this solution seems to be having $O(n^2)$ complexity. But if we look carefully, every element is pushed and popped at most once and in every step of the function at least one element is pushed or popped. Since the amount of work for the decisions and the update is constant, the complexity of the algorithm is $O(n)$ by amortized analysis. Space Complexity: $O(n)$ [for stack].

QUEUES

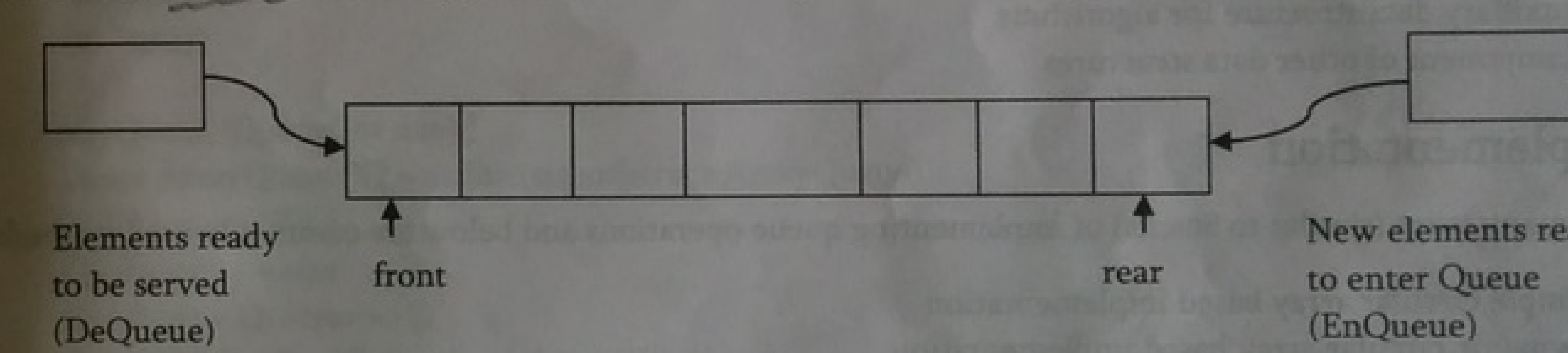
Chapter-5

5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which the data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition: A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called as First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*. Trying to *DeQueue* an empty queue is called *underflow* and trying to *EnQueue* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.



5.2 How is Queues Used?

Line at a reservation counter explains the concept of a queue. When we enter the line we put ourselves at the end of the line and the person who is at the front of the line is the next who will be served. The person will exit the queue and will be served.

In the meanwhile the queue is served and next person at head of the line will exit the queue and will be served. While the queue is served, we move towards the head of the line since each person that is served will be removed from the head of the queue. Finally we will reach head of the line and we will exit the queue and be served. This behavior is very useful in any cases where there is a need to maintain the order of arrival.

5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in queue must follow the FIFO scheme. For simplicity we assume the elements are integers.

Main Queue Operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations

- int Front(): Returns the element at front without removing it

- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueue(): Indicates whether no elements are stored in the queue or not

5.4 Exceptions

As similar to other ADTs, attempting execution of *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and attempting execution of *EnQueue* on an full queue throws an "*Full Queue Exception*".

5.5 Applications

Following are the some of the applications in which queues are being used.

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

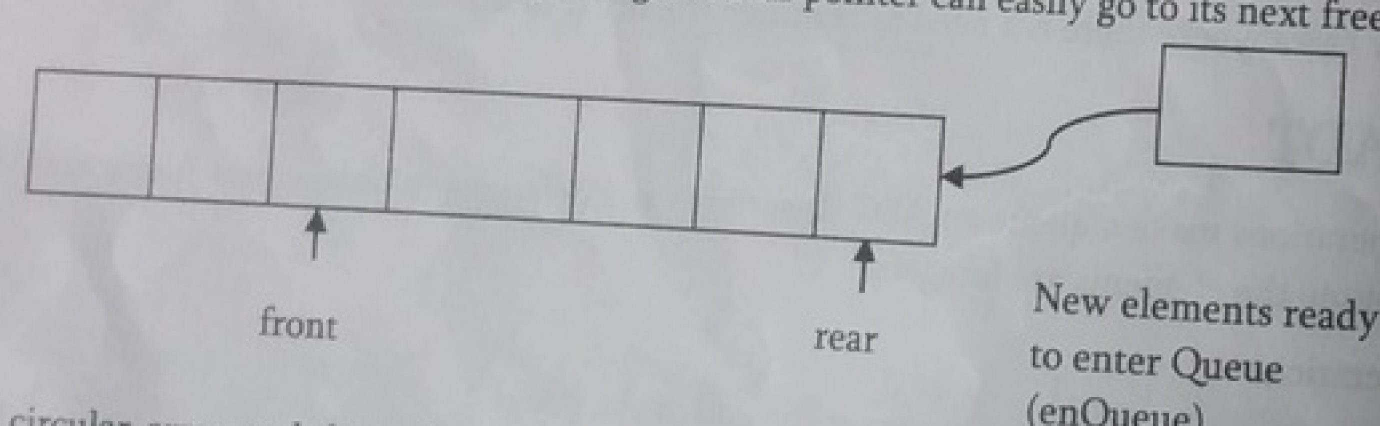
5.6 Implementation

There are many ways (similar to Stacks) of implementing queue operations and below are commonly used methods.

- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation

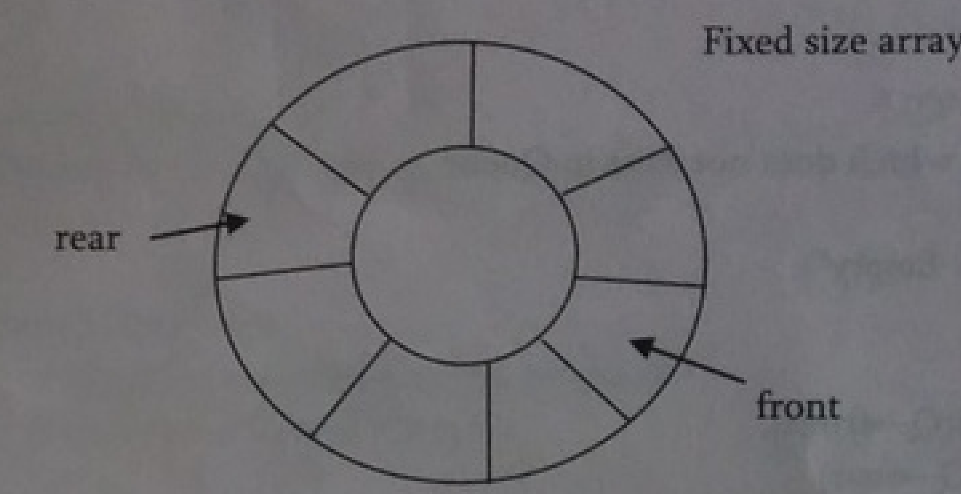
Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues which we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at other end. After some insertions and deletions it is easy to get the situation as shown below. It can be seen clearly that, the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat last element and first array elements are contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.



Note: The simple circular array and dynamic circular array implementations are very much similar to stack array implementations. Refer *Stacks* chapter for analysis of these implementations.

Simple Circular Array Implementation



This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue. The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from empty queue then it will throw *empty queue exception*.

Note: Initially, both front and rear points to -1 which indicates that the queue is empty.

```
struct ArrayQueue {
        int front, rear;
        int capacity;
        int *array;
};
struct ArrayQueue *Queue(int size) {
        struct ArrayQueue *Q = malloc(sizeof(struct ArrayQueue));
        if(!Q) return NULL;
        Q→capacity = size;
        Q→front = Q→rear = -1;
        Q→array= malloc(Q→capacity * sizeof(int));
        if(!Q→array)
            return NULL;
        return Q;
}
int IsEmptyQueue(struct ArrayQueue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == -1);
}
int IsFullQueue(struct ArrayQueue *Q) {
        //if the condition is true then 1 is returned else 0 is returned
        return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
        return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct ArrayQueue *Q, int data) {
        if(IsFullQueue(Q))
                printf("Queue Overflow");
        else {  Q→rear = (Q→rear+1) % Q→capacity;
                Q→ array[Q→rear]= data;
                if(Q→front == -1)
```

```
		Q→front = Q→rear;
	}
}
int DeQueue(struct ArrayQueue *Q) {
	int data = 0;//or element which does not exist in Queue
	if(IsEmptyQueue(Q)) {
		printf("Queue is Empty");
		return 0;
	}
	else {	data = Q→array[Q→front];
		if(Q→front == Q→rear)
			Q→front = Q→rear = -1;
		else	Q→front = (Q→front+1) % Q→capacity;
	}
	return data;
}
void DeleteQueue(struct ArrayQueue *Q) {
	if(Q) {
		if(Q→array)
			free(Q→array);
		free(Q);
	}
}
```

Performance & Limitations

Performance: Let n be the number of elements in the queue:

Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	$O(1)$
Time Complexity of DeQueue()	$O(1)$
Time Complexity of IsEmptyQueue()	$O(1)$
Time Complexity of IsFullQueue()	$O(1)$
Time Complexity of QueueSize()	$O(1)$
Time Complexity of DeleteQueue()	$O(1)$

Limitations: The maximum size of the queue must be defined a prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

Dynamic Circular Array Implementation

```
struct DynArrayQueue {
	int front, rear;
	int capacity;
	int *array;
};
struct DynArrayQueue *CreateDynQueue() {
	struct DynArrayQueue *Q = malloc(sizeof(struct DynArrayQueue));
	if(!Q) return NULL;
	Q→capacity = 1;
	Q→front = Q→rear = -1;
	Q→array= malloc(Q→capacity * sizeof(int));
	if(!Q→array)
```



```
		return NULL;
	return Q;
}
int IsEmptyQueue(struct DynArrayQueue *Q) {
	// if the condition is true then 1 is returned else 0 is returned
	return (Q→front == -1);
}
int IsFullQueue(struct DynArrayQueue *Q) {
	//if the condition is true then 1 is returned else 0 is returned
	return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
	return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct DynArrayQueue *Q, int data) {
	if(IsFullQueue(Q))
		ResizeQueue(Q);
	Q→rear = (Q→rear+1)% Q→capacity;
	Q→ array[Q→rear]= data;
	if(Q→front == -1)
		Q→front = Q→rear;
}
void ResizeQueue(struct DynArrayQueue *Q) {
	int size = Q→capacity;
	Q→capacity = Q→capacity*2;
	Q→array = realloc (Q→array, Q→capacity);
	if(!Q→array) {
		printf("Memory Error");
		return;
	}
	if(Q→front > Q→rear ) {
		for(int i=0; i < Q→front; i++) {
			Q→array[i+size] =Q→array[i];
		}
		Q→rear = Q→rear + size;
	}
}
int DeQueue(struct DynArrayQueue *Q) {
	int data = 0;//or element which does not exist in Queue
	if(IsEmptyQueue(Q)) {
		printf("Queue is Empty");
		return 0;
	}
	else {	data = Q→array[Q→front];
		if(Q→front== Q→rear)
			Q→front= Q→rear = -1;
		else
			Q→front = (Q→front+1) % Q→capacity;
	}
	return data;
```

```
}
void DeleteQueue(struct DynArrayQueue *Q) {
        if(Q) {
                if(Q→array)
                        free(Q→array);
                free(Q→array);
        }
}
```

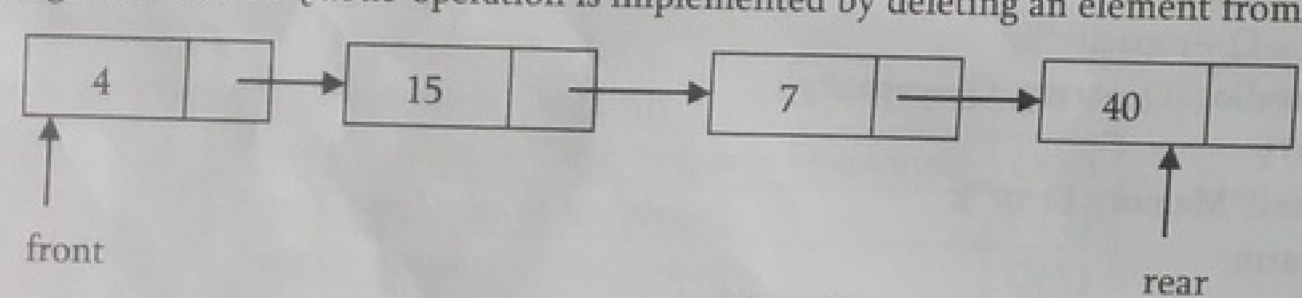
Performance

Let n be the number of elements in the queue.

Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Linked List Implementation

The other way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting element at the ending of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.




```
struct ListNode {
   int data;
   struct ListNode *next;
};
struct Queue {
   struct ListNode *front;
   struct ListNode *rear;
};
struct Queue *CreateQueue() {
        struct Queue *Q;
        struct ListNode *temp;
        Q = malloc(sizeof(struct Queue));
        if(!Q) return NULL;
        temp = malloc(sizeof(struct ListNode));
        Q→front = Q→rear = NULL;
        return Q;
}
int IsEmptyQueue(struct Queue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == NULL);
}
void EnQueue(struct Queue *Q, int data) {
        struct ListNode *newNode;
        newNode = malloc(sizeof(struct ListNode));
```



```
        if(!newNode)
                return NULL;
        newNode→data = data;
        newNode→next = NULL;
        Q→rear→next = newNode;
        Q→rear = newNode;
        if(Q→front == NULL)
                Q→front = Q→rear;
}
int DeQueue(struct Queue *Q) {
        int data = 0;    //or element which does not exist in Queue
        struct ListNode *temp;
        if(IsEmptyQueue(Q)) {
                printf("Queue is empty");
                return 0;
        }
        else {  temp = Q→front;
                data = Q→front→data;
                Q→front== Q→front→next;
                free(temp);
        }
        return data;
}
void DeleteQueue(struct Queue *Q) {
        struct ListNode *temp;
        while(Q) {
                temp = Q;
                Q = Q→next;
                free(temp);
        }
        free(Q);
}
```

Performance

Let n be the number of elements in the queue, then

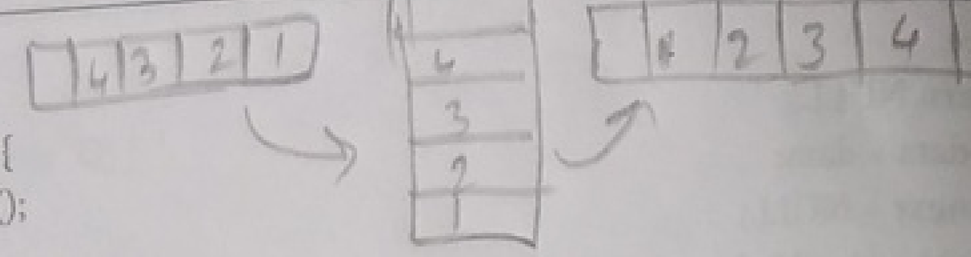
Space Complexity (for n EnQueue operations)	$O(n)$
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

Comparison of Implementations

Note: Comparison is very much similar to stack implementations and *Stacks* chapter.

5.7 Problems on Queues

Problem-1 Give an algorithm for reversing a queue Q. To access the queue, we are only allowed to use the methods of queue ADT.



Solution:

```
void ReverseQueue(struct Queue *Q) {
        struct Stack *S = CreateStack();
        while (!IsEmptyQueue(Q))
                Push(S, DeQueue(Q))
        while (!IsEmptyStack(S))
                EnQueue(Q, Pop(S));
}
```

Time Complexity: $O(n)$.

Problem-2 How to implement a queue using two stacks?

Solution: Let S1 and S2 be the two stacks to be used in the implementation of queue. All we have to do is to define the EnQueue and DeQueue operations for the queue.

```
struct Queue {
        struct Stack *S1; // for EnQueue
        struct Stack *S2; // for DeQueue
}
```

EnQueue Algorithm

- Just push on to stack S1

```
void EnQueue(struct Queue *Q, int data) {
        Push(Q→S1, data);
}
```

Time Complexity: $O(1)$.

DeQueue Algorithm

- If stack S2 is not empty then pop from S2 and return that element.
- If stack is empty, then transfer all elements from S1 to S2 and pop the top element from S2 and return that popped element [we can optimize the code little by transferring only $n-1$ elements from S1 to S2 and pop the n^{th} element from S1 and return that popped element].
- If stack S1 is also empty then throw error.

```
int DeQueue(struct Queue *Q) {
        if(!IsEmptyStack(Q→S2))
                return Pop(Q→S2);
        else {  while(!IsEmptyStack(Q→S1))
                        Push(Q→S2, Pop(Q→S1));
                return Pop(Q→S2);
        }
}
```

Time Complexity: From the algorithm, if the stack S2 is not empty then the complexity is $O(1)$. If the stack S2 is empty then, we need to transfer the elements from S1 to S2. But if we carefully observe, the number of transferred elements and the number of popped elements from S2 are equal. Due to this the average complexity of pop operation in this case is $O(1)$. Amortized complexity of pop operation is $O(1)$.

Problem-3 Show how to efficiently implement one stack using two queues. Analyze the running time of the stack operations.

Solution: Let Q1 and Q2 be the two queues to be used in the implementation of stack. All we have to do is to define the push and pop operations for the stack.

```
struct Stack {
        struct Queue *Q1;
```



```
        struct Queue *Q2;
}
```

In below algorithms, we make sure that one queue is empty always.

Push Operation Algorithm: Whichever is queue is not empty, insert the element in it.

- Check whether queue Q1 is empty or not. If Q1 is empty then Enqueue the element into Q2.
- Otherwise EnQueue the element into Q1.

```
Push(struct Stack *S, int data) {
        if(IsEmptyQueue(S→Q1))
                EnQueue(S→Q2, data);
        else    EnQueue(S→Q1, data);
}
```

Time Complexity: $O(1)$.

Pop Operation Algorithm: Transfer $n-1$ elements to other queue and delete last from queue for performing pop operation.

- If queue Q1 is not empty then transfer $n-1$ elements from Q1 to Q2 and then, DeQueue the last element of Q1 and return it.
- If queue Q2 is not empty then transfer $n-1$ elements from Q2 to Q1 and then, DeQueue the last element of Q2 and return it.

```
int Pop(struct Stack *S) {
        int i, size;
        if(IsEmptyQueue(S→Q2)) {
                size = size(S→Q1);
                i = 0;
                while(i < size-1) {
                        EnQueue(S→Q2, DeQueue(S→Q1));
                        i++;
                }
                return DeQueue(S→Q1);
        }
        else {  size =  size(S→Q2);
                while(i < size-1) {
                        EnQueue(S→Q1, DeQueue(S→Q2));
                        i++;
                }
                return DeQueue(S→Q2);
        }
}
```

Time Complexity: Running time of pop operation is $O(n)$ as each time pop is called, we are transferring all the elements from one queue to other.

Problem-4 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5

1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output**: An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]. **Requirement**: Find a good optimal way to get B[i]

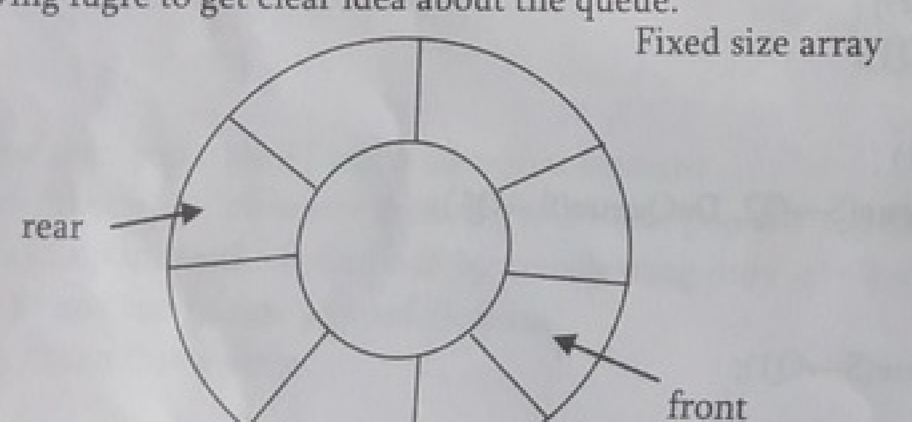
Solution: This problem can be solved with doubly ended queue (which support insertion and deletions at both ends). Refer *Priority Queues* chapter for algorithms.

Problem-5 Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue and push and pop operations for the stack, outline an efficient O(n) algorithm to accomplish the above task, using only a constant amount of additional storage.

Solution: Assume the elements of queue Q are $a_1, a_2 \dots a_n$. Dequeuing all elements and pushing them onto the stack will result in a stack with an at the top and a_1 at the bottom. This is done in O(n) time as dequeue and push each require constant time per operation. The queue is now empty. By popping all elements and pushing them on the the queue we will get a_1 at the top of the stack. This is done again in O(n) time. As in big-oh arithmetic we can ignore constant factors, the process is carried out in O(n) time. The amount of additional storage needed here has to be big enough to temporarily hold one item.

Problem-6 A queue is set up in a circular array A[0..n - 1] with front and rear defined as usual. Assume that $n-1$ locations in the array are available for storing the elements (with the other element being used to detect full/empty condition). Give a formula for the number of elements in the queue in terms of *rear*, *front*, and *n*.

Solution: Consider the following fugre to get clear idea about the queue.



- Rear of the queue is somewhere clockwise from the front
- To enqueue an element, we move rear one position clockwise and write the element in that position
- To dequeue, we simply move front one position clockwise
- Queue migrates in a clockwise direction as we enqueue and dequeue
- Emptiness and fullness to be checked carefully.
- Analyze the possible situations (make some drawings to see where *front* and *rear* are when the queue is empty, and partially and totally filled). We will get this:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if rear == front} \\ rear - front + n & \text{otherwise} \end{cases}$$

Chapter-6

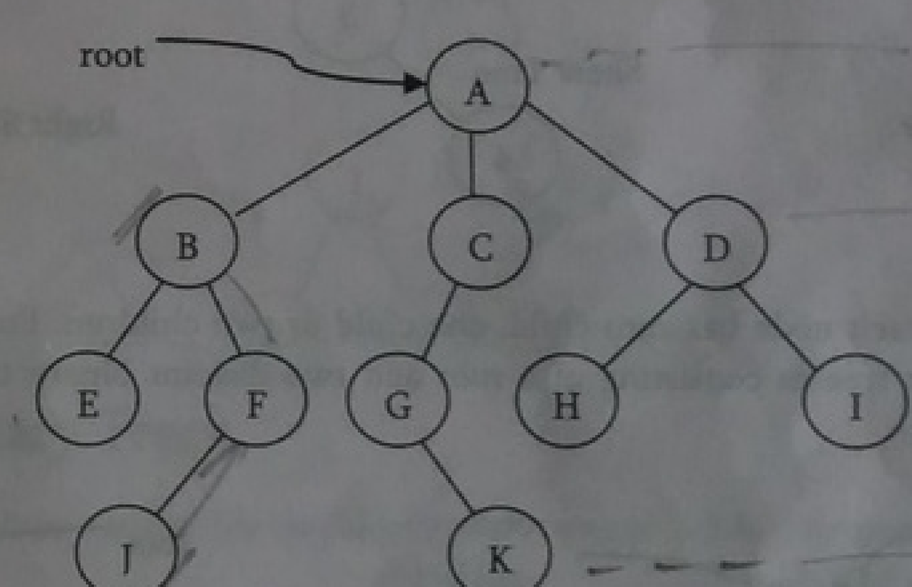
TREES

6.1 What is a Tree?

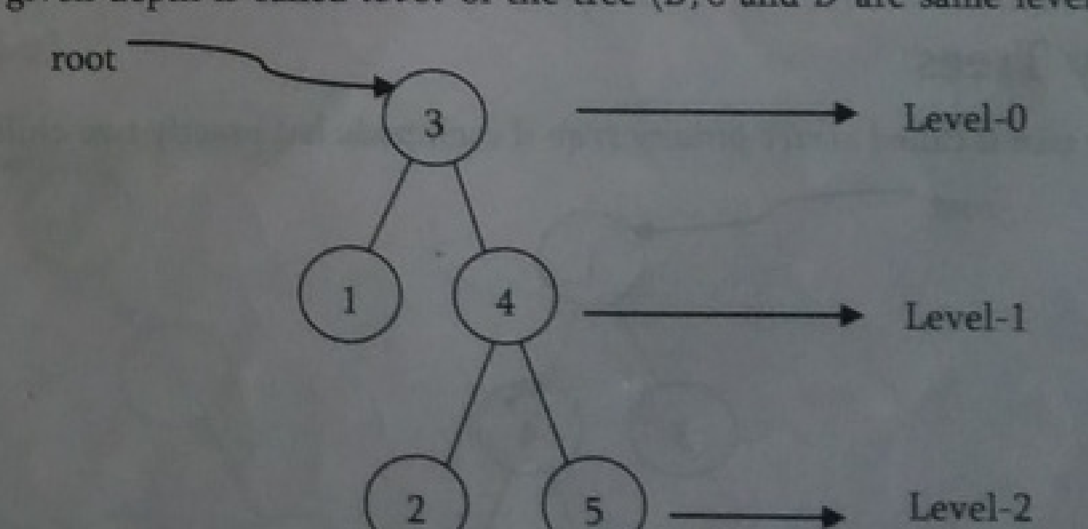
A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Tree is an example of non-linear data structures. A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.

In trees ADT (Abstract Data Type), order of the elements is not important. If we need ordering information linear data structures like linked lists, stacks, queues, etc. can be used.

6.2 Glossary

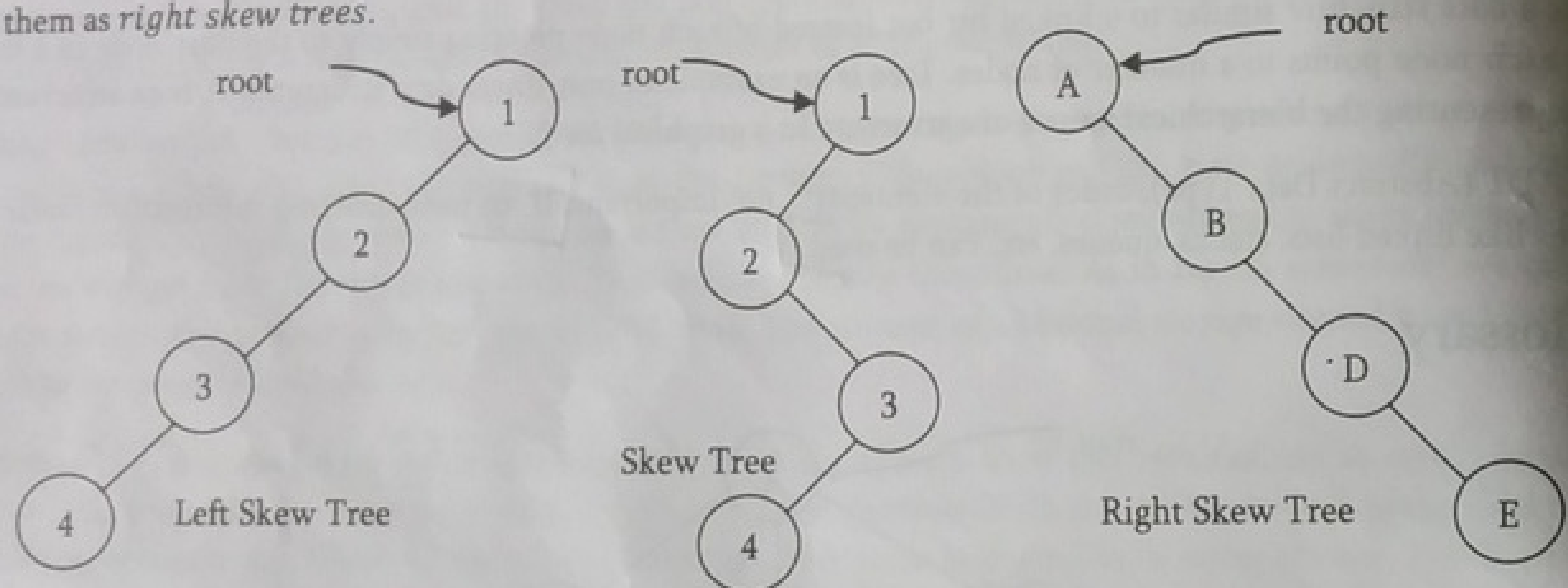


- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (node *A* in the above example).
- An *edge* refers to the link from parent to child (all links in the figure).
- A node with no children is called *leaf* node (E, J, K, H and I).
- Children of same parent are called *siblings* (B, C, D are siblings of A and E, F are the siblings of B).
- A node p is an *ancestor* of a node q if there exists a path from *root* to q and p appears on the path. The node q is called a *descendant* of p. For example, A, C and G are the ancestors for K.
- Set of all nodes at a given depth is called *level* of the tree (B, C and D are same level). The root node is at level zero.



- The *depth* of a node is the length of the path from the root to the node (depth of G is 2, $A-C-G$).

- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, height of B is 2 ($B - F - J$).
- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree. For a given tree depth and height returns the same value. But for individual nodes we may get different results.
- Size of a node is the number of descendants it has including itself (size of the subtree C is 3).
- If every node in a tree has only one child (except leaf nodes) then we call such trees as *skew trees*. If every node has only left child then we call them as *left skew trees*. Similarly, if every node has only right child then we call them as *right skew trees*.

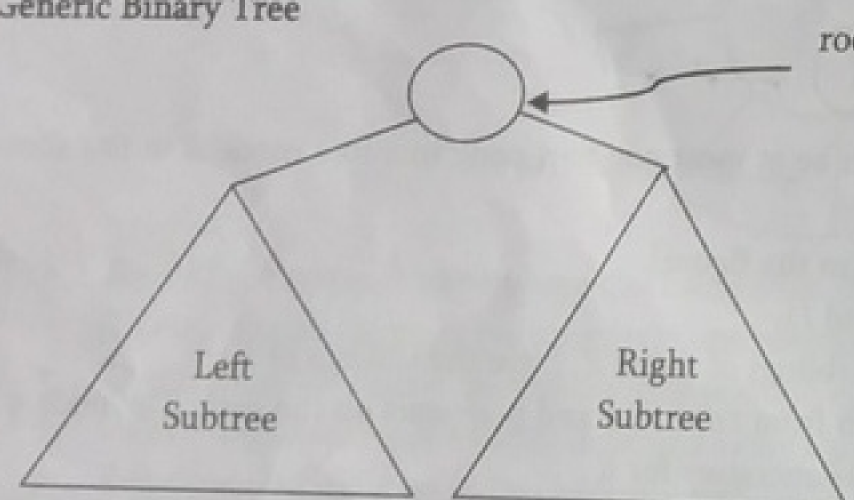


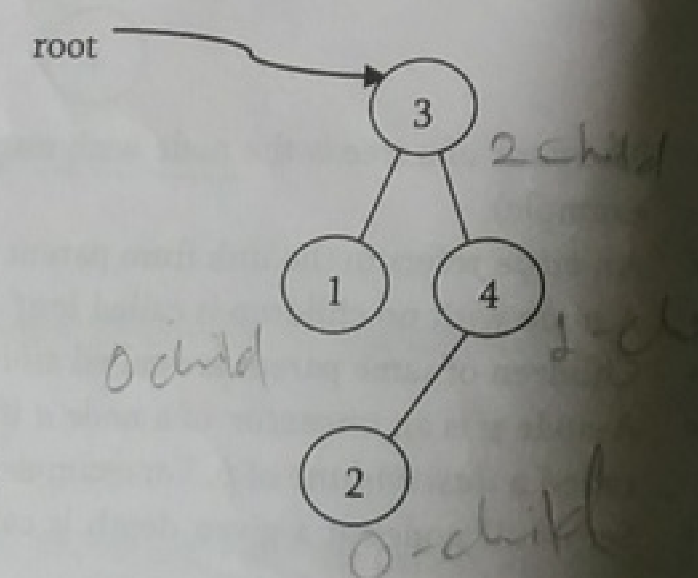
Left Skew Tree Skew Tree Right Skew Tree

6.3 Binary Trees

A tree is called *binary tree* if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Generic Binary Tree

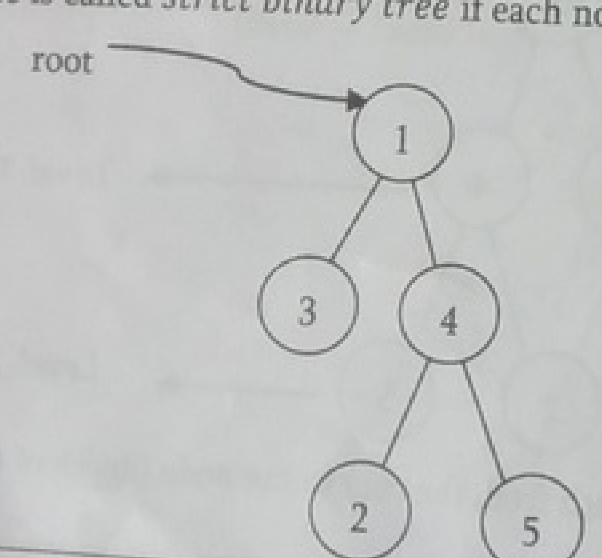




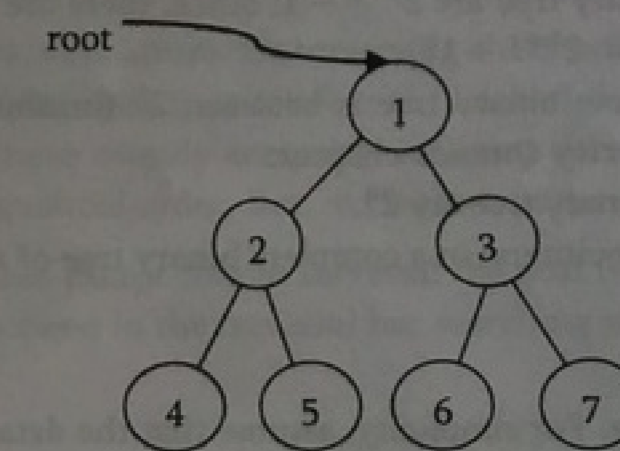
Example

6.4 Types of Binary Trees

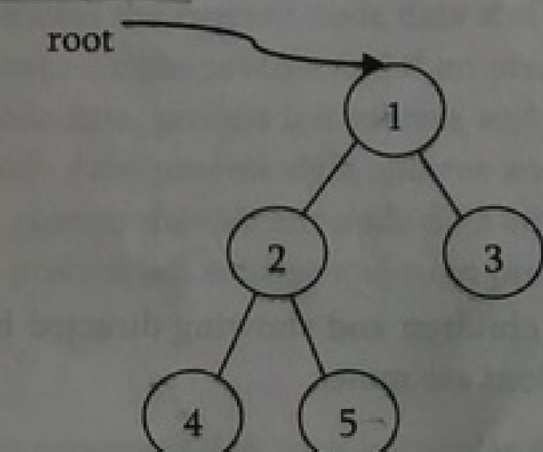
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at same level.

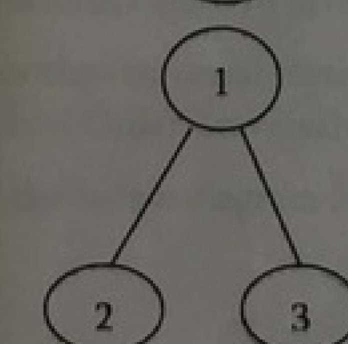
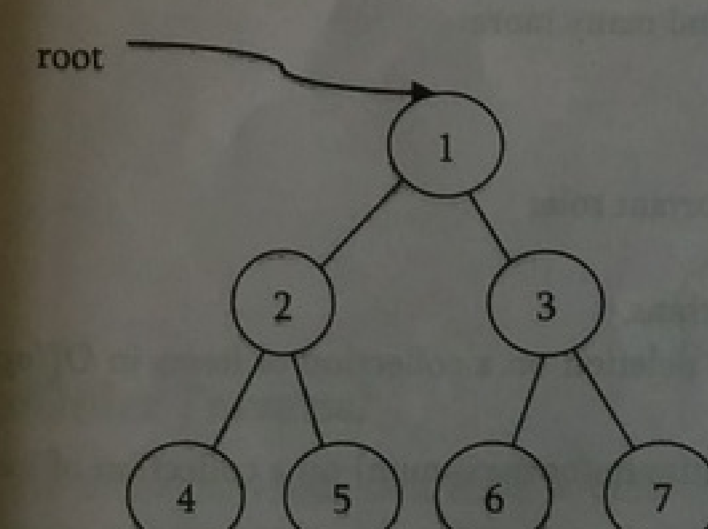


Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at root (let us say the root node has 1) then we get a complete sequence from 1 to number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called complete binary tree if all leaf nodes are at height h or $h - 1$ and also without any missing number in the sequence.



6.5 Properties of Binary Trees

For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero.

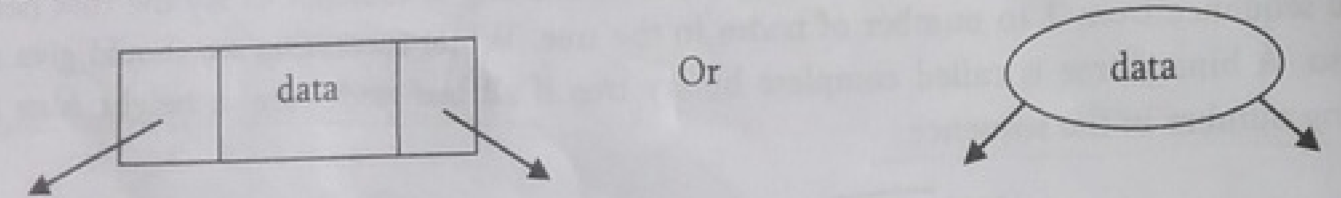
	Height	Number of nodes at level h
	$h = 0$	$2^0 = 1$
	$h = 1$	$2^1 = 2$
	$h = 2$	$2^2 = 4$

From the diagram we can infer the following properties:

- The number of nodes n in a full binary tree are $2^{h+1}-1$. Since, there are h levels we need to add all nodes at each level $[2^0+2^1+2^2+\cdots+2^h = 2^{h+1}-1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1}-1$ (maximum). For more information on this, refer *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree are 2^h.
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes are $n+1$.

Structure of Binary Trees

Now let us define structure of the binary tree. For simplicity, assume that the data of the nodes are integers. One way to represent a node (which contains the data) is to have two links which points to left and right children along with data fields as shown below:




```
struct BinaryTreeNode {
  int data;
  struct BinaryTreeNode *left;
  struct BinaryTreeNode *right;
};
```

Note: In trees, the default flow is from parent to children and showing directed branches is not compulsory. For our discussion, we assume both the below representations are same.

Operations on Binary Trees

Basic Operations

- Inserting an element in to a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding size of the tree
- Finding height of the tree
- Finding the level which has maximum sum
- Finding least common ancestor (LCA) for a given pair of nodes and many more.

Applications of Binary Trees

Following are the some of the applications where *binary trees* play important role:

- Expression trees are used in compilers.
- Huffman coding trees which are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in $O(logn)$ (average).
- Priority Queues (PQ), which supports search and deletion of minimum(or maximum) on a collection of items in logarithmic time (in worst case).

6.6 Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them and that forms the subject of this section. The process of visiting all nodes of a tree is called *tree traversal*. Each of the nodes is processed only once but they may be visited more than once. As we have already seen that in linear data structures (like linked lists, stacks, queues, etc...), the elements are visited in sequential order. But, in tree structures there are many different ways.

Tree traversal is like searching the tree except that in traversal the goal is to move through the tree in some particular order. In addition, all nodes are processed in the traversal but searching stops when the required node is found.

Traversal Possibilities

Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denotes with "D"), traversing to the left child node (denotes with "L"), and traversing to the right child node (denotes with "R"). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

1. *LDR*: Process left subtree, process the current node data and then process right subtree
2. *LRD*: Process left subtree, process right subtree and then process the current node data
3. *DLR*: Process the current node data, process left subtree and then process right subtree
4. *DRL*: Process the current node data, process right subtree and then process left subtree
5. *RDL*: Process right subtree, process the current node data and then process left subtree
6. *RLD*: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals

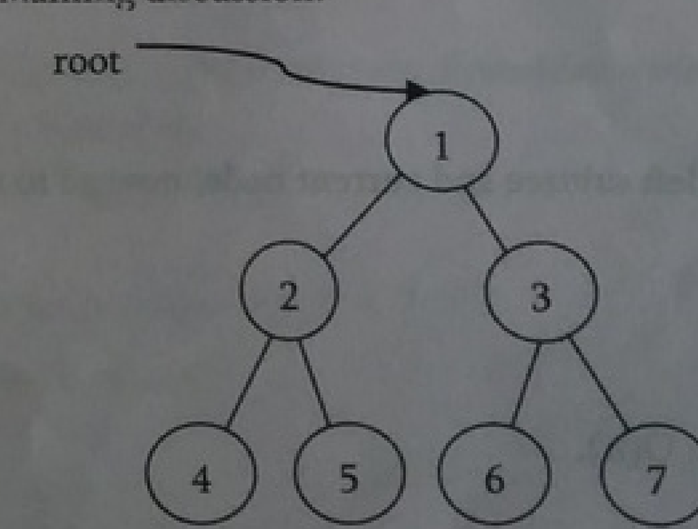
The sequence in which these entities processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L on left side of D or R is on left side of D. Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities were reduced to 3 and they are:

- Preorder (*DLR*) Traversal
- Inorder (*LDR*) Traversal
- Postorder (*LRD*) Traversal

There is another traversal method which does not depend on above orders and it is:

- Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Let us use the below diagram for remaining discussion.



PreOrder Traversal

In pre-order traversal, each node is processed before (pre) either of its sub-trees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree. In the example above, the 1 is processed first, then the left sub-tree followed by the right subtree. Therefore, processing must return to the right sub-tree after finishing the processing of the left subtree. To move to right subtree after processing left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in the reverse order.

Preorder traversal is defined as follows:

- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

```
void PreOrder(struct BinaryTreeNode *root){
	if(root) {
		printf("%d",root→data);
		PreOrder(root→left);
		PreOrder (root→right);
	}
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Non-Recursive Preorder Traversal

In recursive version a stack is required as we need to remember the current node so that after completing the left subtree we can go to right subtree. To simulate the same, first we process the current node and before going to left subtree, we store the current node on stack. After completing the left subtree processing, *pop* the element and go to its right subtree. Continue this process until stack is nonempty.

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
	struct Stack *S = CreateStack();
	while(1) {
		while(root) {
			//Process current node
			printf("%d",root→data);
			Push(S,root);
			//If left subtree exists, add to stack
			root = root→left;
		}
		if(IsEmptyStack(S))
			break;
		root = Pop(S);
		//Indicates completion of left subtree and current node, now go to right subtree
		root = root→right;
	}
	DeleteStack(S);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

InOrder Traversal

In Inorder traversal the root is visited between the subtrees. Inorder traversal is defined as follows:

- Traverse the left subtree in Inorder.
- Visit the root.

- Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

```
void InOrder(struct BinaryTreeNode *root){
	if(root) {
		InOrder(root→left);
		printf("%d",root→data);
		InOrder(root→right);
	}
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Non-Recursive Inorder Traversal

Non-recursive version of Inorder traversal is very much similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which indicates after completion of left subtree processing).

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
	struct Stack *S = CreateStack();
	while(1) {
		while(root) {
			Push(S,root);
			//Got left subtree and keep on adding to stack
			root = root→left;
		}
		if(IsEmptyStack(S))
			break;
		root = Pop(S);
		printf("%d", root→data);		//After popping, process the current node
		//Indicates completion of left subtree and current node, now go to right subtree
		root = root→right;
	}
	DeleteStack(S);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

PostOrder Traversal

In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:

- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

The nodes of tree would be visited in the order: 4 5 2 6 7 3 1

```
void PostOrder(struct BinaryTreeNode *root){
	if(root) {
		PostOrder(root→left);
		PostOrder(root→right);
		printf("%d",root→data);
	}
}
```

Time Complexity: O(n). Space Complexity: O(n).

Non-Recursive Postorder Traversal

In preorder and inorder traversals, after poping the stack element we do not need to visit the same vertex again. But in postorder traversal, each node is visited twice. That means, after processing left subtree we will be visiting the current node and also after processing the right subtree we will be visiting the same current node. But we should be processing the node during the second visit. Here the problem is how to differentiate whether we are returning from left subtree or right subtree?

Trick for this problem is: after popping an element from stack, check whether that element and right of top of the stack are same or not. If they are same then we are done with processing of left subtree and right subtree. In this case we just need to pop the stack one more time and print its data.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while (1) {
        if (root) {
            Push(S,root);
            root=root→left;
        }
        else {  if(IsEmptyStack(S)) {
                    printf("Stack is Empty");
                    return;
                }
                else    if(Top(S)→right == NULL) {
                            root = Pop(S);
                            printf("%d",root→data);
                            if(root == Top(S)→right) {
                                printf("%d",Top(S)→data);
                                Pop(S);
                            }
                        }
                if(!IsEmptyStack(S))
                    root=Top(S)→right;
                else    root=NULL;
        }
    }
    DeleteStack(S);
}
```

Time Complexity: O(n). Space Complexity: O(n).

Level Order Traversal

Level order traversal is defined as follows:

- Visit the root.
- While traversing level l, keep all the elements at level $l+1$ in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

The nodes of tree would be visited in the order: 1 2 3 4 5 6 7

```
void LevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
```



```
    struct Queue *Q = CreateQueue();
    if(!root)
        return;
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //Process current node
        printf("%d", temp→data);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

Time Complexity: O(n). Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Problems on Binary Trees

Problem-1 Give an algorithm for finding maximum element in binary tree.

Solution: One simple way of solving this problem is: find the maximum element in left subtree, find maximum element in right sub tree, compare them with root data and select the one which is giving the maximum value. This approach can be easily implemented with recursion.

```
int FindMax(struct BinaryTreeNode *root) {
    int root_val, left, right, max = INT_MIN;
    if(root !=NULL) {
        root_val = root→data;
        left = FindMax(root→left);
        right = FindMax(root→right);
        // Find the largest of the three values.
        if(left > right)
            max = left;
        else    max = right;
        if(root_val > max)
            max = root_val;
    }
    return max;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-2 Give an algorithm for finding maximum element in binary tree without recursion.

Solution: Using level order traversal: just observe the elements data while deleting.

```
int FindMaxUsingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int max = INT_MIN;
    struct Queue *Q = CreateQueue();
    EnQueue(Q,root);
```

```
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        // largest of the three values
        if(max < temp→data)
            max = temp→data;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return max;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-3 Give an algorithm for searching an element in binary tree.

Solution: Given a binary tree, return true if a node with the data is found in the tree. Recurse down the tree, choose the left or right branch by comparing the data with each nodes data.

```
int FindInBinaryTreeUsingRecursion(struct BinaryTreeNode *root, int data) {
    int temp;
    // Base case == empty tree, in that case, the data is not found so return false
    if(root == NULL)
        return 0;
    else {  //see if found here
        if(data == root→data)
            return 1;
        else {  // otherwise recur down the correct subtree
            temp = FindInBinaryTreeUsingRecursion (root→left, data)
            if(temp != 0)
                return temp;
            else    return(FindInBinaryTreeUsingRecursion(root→right, data));
        }
    }
    return 0;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-4 Give an algorithm for searching an element in binary tree without recursion.

Solution: We can use level order traversal for solving this problem. The only change required in level order traversal is, instead of printing the data we just need to check whether the root data is equal to the element we want to search.

```
int SearchUsingLevelOrder(struct BinaryTreeNode *root, int data){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root)
        return -1;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
```



```
        //see if found here
        if(data == root→data)
            return 1;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return 0;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-5 Give an algorithm for inserting an element into binary tree.

Solution: Since the given tree is a binary tree, we can insert the element wherever we want. To insert an element, we can use the level order traversal and insert the element wherever we found the node whose left or right child is NULL.

```
void InsertInBinaryTree(struct BinaryTreeNode *root, int data){
    struct Queue *Q;
    struct BinaryTreeNode *temp;
    struct BinaryTreeNode *newNode;
    newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
    newNode→left = newNode→right = NULL;
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    if(!root) {
        root = newNode;
        return;
    }
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(temp→left)
            EnQueue(Q, temp→left);
        else {  temp→left=newNode;
            DeleteQueue(Q);
            return;
        }
        if(temp→right)
            EnQueue(Q, temp→right);
        else {  temp→right=newNode;
            DeleteQueue(Q);
            return;
        }
    }
    DeleteQueue(Q);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-6 Give an algorithm for finding the size of binary tree.

Solution: Calculate the size of left and right subtrees recursively, add 1 (current node) and return to its parent.

```
// Compute the number of nodes in a tree.
int SizeOfBinaryTree(struct BinaryTreeNode *root) {
    if(root==NULL)
        return 0;
    else    return(SizeOfBinaryTree(root→left) + 1 + SizeOfBinaryTree(root→right));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

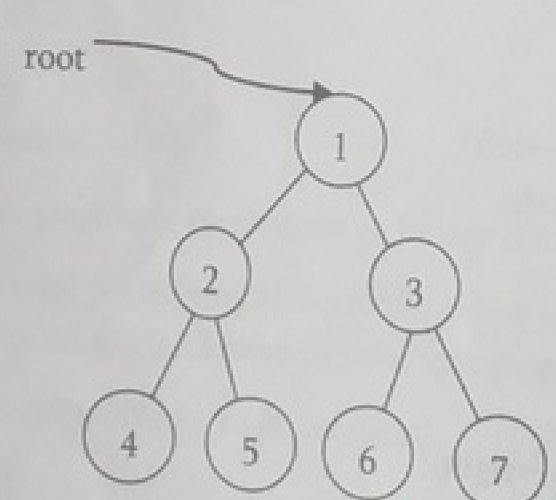
Problem-7 Can we solve the Problem-6 without recursion?

Solution: Yes, using level order traversal.

```
int SizeofBTUsingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        count++;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-8 Give an algorithm for printing the level order data in ~~reserve~~ reverse order. For example, the output for the below tree should be: 4 5 6 7 2 3 1



Solution:

```
void LevelOrderTraversalInReverse(struct BinaryTreeNode *root){
    struct Queue *Q;
    struct Stack *s = CreateStack();
    struct BinaryTreeNode *temp;
    if(!root) return;
    Q = CreateQueue();
    EnQueue(Q, root);
```



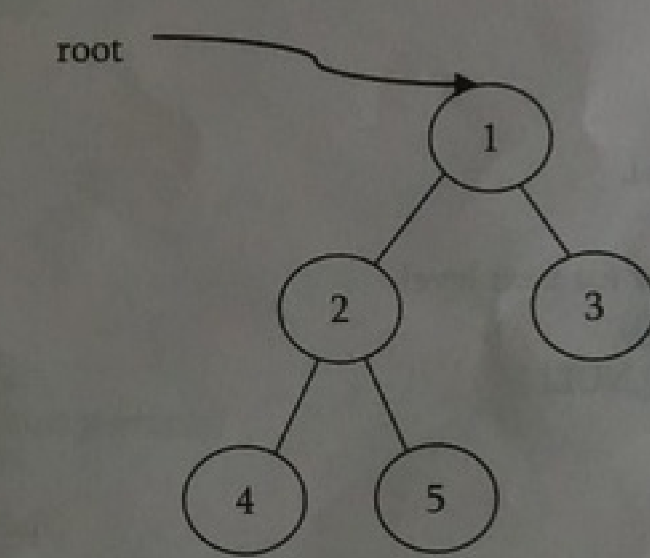
```
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(temp→right)
            EnQueue(Q, temp→right);
        if(temp→left)
            EnQueue (Q, temp→left);
        Push(s, temp);
    }
    while(!IsEmptyStack(s))
        printf("%d",Pop(s)→data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-9 Give an algorithm for deleting the tree.

Solution: To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use Inorder, Preorder, Postorder or Level order Traversal?

Before deleting the parent node we should delete its children nodes first. We can use postorder traversal as it does the work without storing anything. We can delete tree with other traversals also with extra space complexity. For the following tree nodes are deleted in order – 4, 5, 2, 3, 1.




```
void DeleteBinaryTree(struct BinaryTreeNode *root){
    if(root == NULL)
        return;
    /* first delete both subtrees */
    DeleteBinaryTree(root→left);
    DeleteBinaryTree(root→right);
    //Delete current node only after deleting subtrees
    free(root);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-10 Give an algorithm for finding the height (or depth) of the binary tree.

Solution: Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. This is similar to *PreOrder* tree traversal (and *DFS* of Graph algorithms).

```
int HeightOfBinaryTree(struct BinaryTreeNode *root){
    int leftheight, rightheight;
    if(root == NULL)
        return 0;
    else {    /* compute the depth of each subtree */
```

```
        leftheight = HeightOfBinaryTree(root→left);
        rightheight = HeightOfBinaryTree(root→right);
        if(leftheight > rightheight)
                return(leftheight + 1);
        else    return(rightheight + 1);
    }
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-11 Can we solve the Problem-10 without recursion?

Solution: Yes. Using level order traversal. This is similar to *BFS* of Graph algorithms. End of level is identified with NULL.

```
int FindHeightofBinaryTree(struct BinaryTreeNode *root){
    int level=1;
    struct Queue *Q;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    // End of first level
    EnQueue(Q,NULL);
    while(!IsEmptyQueue(Q)) {
        root=DeQueue(Q);
        // Completion of current level.
        if(root==NULL) {
            //Put another marker for next level.
            if(!IsEmptyQueue(Q))
                EnQueue(Q,NULL);
            level++;
        }
        else {  if(root→left)
                    EnQueue(Q, root→left);
                if(root→right)
                    EnQueue(Q, root→right);
        }
    }
    return level;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-12 Give an algorithm for finding the deepest node of the binary tree.

Solution:

```
struct BinaryTreeNode *DeepestNodeinBinaryTree(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root) return NULL;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(temp→left)
```



```
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
    return temp;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-13 Give an algorithm for deleting an element (assuming data is given) from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Starting at root, find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest nodes data with node to be deleted.
- Then delete the deepest node.

Problem-14 Give an algorithm for finding the number of leaves in the binary tree without using recursion.

Solution: The set of all nodes whose both left and either right are NULL are called leaf nodes.

```
int NumberOfLeavesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(!temp→left && !temp→right)
            count++;
        else {  if(temp→left)
                    EnQueue(Q, temp→left);
                if(temp→right)
                    EnQueue(Q, temp→right);
        }
    }
    DeleteQueue(Q);
    return count;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-15 Give an algorithm for finding the number of full nodes in the binary tree without using recursion.

Solution: The set of all nodes with both left and right children are called full nodes.

```
int NumberOfFullNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
```

```
        temp = DeQueue(Q);
        if(temp→left && temp→right)
                count++;
        if(temp→left)
                EnQueue (Q, temp→left);
        if(temp→right)
                EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-16 Give an algorithm for finding the number of half nodes (nodes with only one child) in the binary tree without using recursion.

Solution: The set of all nodes with either left or either right child (but not both) are called half nodes.

```
int NumberOfHalfNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root)
        return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //we can use this condition also instead of two temp→left ^ temp→right
        if(!temp→left && temp→right || temp→left && !temp→right)
                count++;
        if(temp→left)
                EnQueue (Q, temp→left);
        if(temp→right)
                EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-17 Given two binary trees, return true if they are structurally identical.

Solution:

Algorithm:

- If both trees are NULL then return true.
- If both trees are not NULL, then compare data and recursively check left and right subtree structures.

```
//Return true if they are structurally identical.
int AreStructurullySameTrees(struct BinaryTreeNode *root1, struct BinaryTreeNode *root2) {
    // both empty→1
    if(root1==NULL && root2==NULL)
        return 1;
    if(root1==NULL || root2==NULL)
```



```
        return 0;
    // both non-empty→compare them
    return(root1→data == root2→data && AreStructurullySameTrees(root1→left, root2→left) &&
            AreStructurullySameTrees(root1→right, root2→right));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for recursive stack.

Problem-18 Give an algorithm for finding the diameter of the binary tree. The diameter of a tree (sometimes called the *width*) is the number of nodes on the longest path between two leaves in the tree.

Solution: To find the diameter of a tree, first calculate the diameter of left subtree and right sub trees recursively. Among these two values, we need to send maximum along with current level (+1).

```
int DiameterOfTree(struct BinaryTreeNode *root, int *ptr){
    int left, right;
    if(!root)
        return 0;
    left = DiameterOfTree(root→left, ptr);
    right = DiameterOfTree(root→right, ptr);
    if(left + right > *ptr)
        *ptr = left + right;
    return Max(left, right)+1;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-19 Give an algorithm for finding the level which is having maximum sum in the binary tree.

Solution: The logic is very much similar to finding number of levels. The only change is, we need to keep track of sums as well.

```
int FindLevelwithMaxSum(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int level=0, maxLevel=0;
    struct Queue *Q;
    int currentSum = 0, maxSum = 0;
    if(!root) return 0;
    Q=CreateQueue();
    EnQueue(Q,root);
    EnQueue(Q,NULL);                    //End of first level.
    while(!IsEmptyQueue(Q)) {
        temp =DeQueue(Q);
        // If the current level is completed then compare sums
        if(temp == NULL) {
            if(currentSum> maxSum) {
                maxSum = currentSum;
                maxLevel = level;
            }
            currentSum = 0;
            //place the indicator for end of next level at the end of queue
            if(!IsEmptyQueue(Q))
                EnQueue(Q,NULL);
            level++;
        }
```