CHAPTER

4

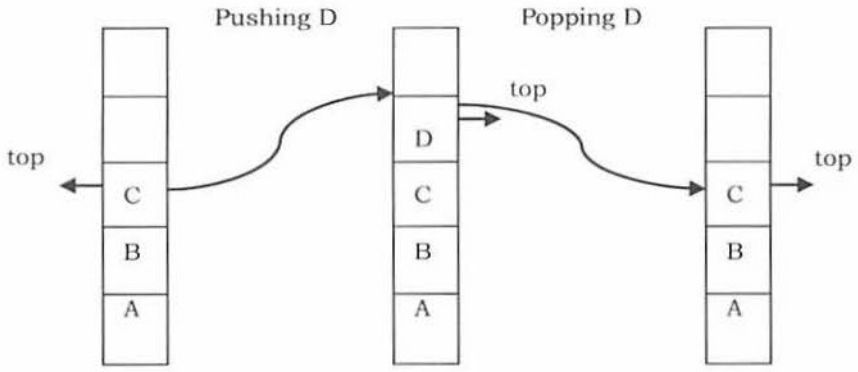
STACKS

4.1 What is a Stack?

A stack is a simple data structure used for storing data (similar to Linked Lists). In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

Definition: A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called *push*, and when an element is removed from the stack, the concept is called *pop*. Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.



4.2 How Stacks are Used

Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task which is more important. The developer puts the long-term project aside and begins work on the new task. The phone rings, and this is the highest priority as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone.

When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

4.3 Stack ADT

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

Main stack operations

- Push (int data): Inserts *data* onto stack.
- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

Exceptions

Attempting the execution of an operation may sometimes cause an error condition, called an exception. Exceptions are said to be "thrown" by an operation that cannot be executed. In the Stack ADT, operations pop and top cannot be performed if the stack is empty. Attempting the execution of pop (top) on an empty stack throws an exception. Trying to push an element in a full stack throws an exception.

4.4 Applications

Following are some of the applications in which stacks play an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to *Problems* section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML

Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
- Component of other data structures (Example: Simulating queues, refer *Queues* chapter)

4.5 Implementation

There are many ways of implementing stack ADT; below are the commonly used methods.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.

The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception*.

```
class Stack(object):
 def __init__(self, limit = 10):
        self.stk = []
        self.limit = limit

 def isEmpty(self):
        return len(self.stk) <= 0

 def push(self, item):
        if len(self.stk) >= self.limit:
                print 'Stack Overflow!'
        else:
                self.stk.append(item)
        print 'Stack after Push',self.stk

 def pop(self):
        if len(self.stk) <= 0:
                print 'Stack Underflow!'
                return 0
        else:
                return self.stk.pop()

 def peek(self):
        if len(self.stk) <= 0:
                print 'Stack Underflow!'
                return 0
        else:
                return self.stk[-1]

 def size(self):
        return len(self.stk)

our_stack = Stack(5)
our_stack.push("1")
our_stack.push("21")
our_stack.push("14")
our_stack.push("31")
our_stack.push("19")
our_stack.push("3")
our_stack.push("99")
our_stack.push("9")
print our_stack.peek()
print our_stack.pop()
print our_stack.peek()
print our_stack.pop()
```

Performance & Limitations

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

Limitations

The maximum size of the stack must first be defined and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Dynamic Array Implementation

First, let's consider how we implemented a simple array based stack. We took one index variable *top* which points to the index of the most recently inserted element in the stack. To insert (or push) an element, we increment *top* index and then place the new element at that index.

Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index. We represent an empty queue with *top* value equal to -1. The issue that still needs to be resolved is what we do when all the slots in the fixed size array stack are occupied?

First try: What if we increment the size of the array by 1 every time the stack is full?

- Push(): increase size of S|| by 1
- Pop(): decrease size of S|| by 1

Problems with this approach?

This way of incrementing the array size is too expensive. Let us see the reason for this. For example, at $n = 1$, to push an element create a new array of size 2 and copy all the old array elements to the new array, and at the end add the new element. At $n = 2$, to push an element create a new array of size 3 and copy all the old array elements to the new array, and at the end add the new element.

Similarly, at $n = n - 1$, if we want to push an element create a new array of size n and copy all the old array elements to the new array and at the end add the new element. After n push operations the total time $T(n)$ (number of copy operations) is proportional to $1 + 2 + \ldots + n \approx O(n^2)$.

Alternative Approach: Repeated Doubling

Let us improve the complexity by using the array *doubling* technique. If the array is full, create a new array of twice the size, and copy the items. With this approach, pushing n items takes time proportional to n (not n^2).

For simplicity, let us assume that initially we started with $n = 1$ and moved up to $n = 32$. That means, we do the doubling at $1, 2, 4, 8, 16$. The other way of analyzing the same approach is: at $n = 1$, if we want to add (push) an element, double the current size of the array and copy all the elements of the old array to the new array.

At $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and at $n = 4$, we do 4 copy operations and so on. By the time we reach $n = 32$, the total number of copy operations is $1 + 2 + 4 + 8 + 16 = 31$ which is approximately equal to $2n$ value (32). If we observe carefully, we are doing the doubling operation *logn* times.

Now, let us generalize the discussion. For n push operations we double the array size *logn* times. That means, we will have *logn* terms in the expression below. The total time $T(n)$ of a series of n push operations is proportional to

$$\begin{aligned} 1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1 \\ &= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right) \\ &= n(2) \approx 2n = O(n) \end{aligned}$$

$T(n)$ is $O(n)$ and the amortized time of a push operation is $O(1)$.

```
class Stack(object):
 def __init__(self, limit = 10):
        self.stk = limit*[None]
        self.limit = limit
 def isEmpty(self):
        return len(self.stk) <= 0
 def push(self, item):
        if len(self.stk) >= self.limit:
            self.resize()
        self.stk.append(item)
        print 'Stack after Push',self.stk
 def pop(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk.pop()
```

```
    def peek(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk[-1]
    def size(self):
        return len(self.stk)
    def resize(self):
        newStk = list(self.stk)
        self.limit = 2*self.limit
        self.stk = newStk

our_stack = Stack(5)
our_stack.push("1")
our_stack.push("21")
our_stack.push("14")
our_stack.push("11")
our_stack.push("31")
our_stack.push("14")
our_stack.push("15")
our_stack.push("19")
our_stack.push("3")
our_stack.push("99")
our_stack.push("9")
print our_stack.peek()
print our_stack.pop()
print our_stack.peek()
print our_stack.pop()
```

Performance

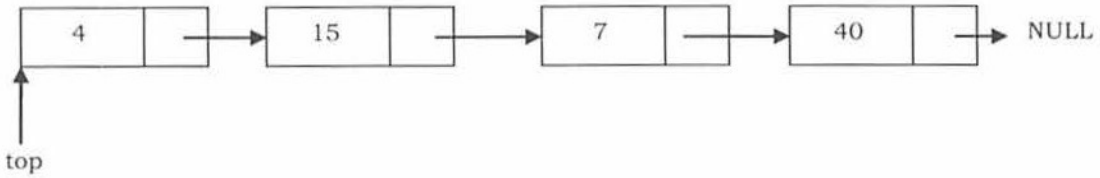
Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1))
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

Note: Too many doublings may cause memory overflow exception.

Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).




```
#Node of a Singly Linked List
class Node:
  #constructor
  def __init__(self):
    self.data = None
    self.next = None
```

```
        #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
      #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
      #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
class Stack(object):
    def __init__(self, data=None):
        self.head = None
        if data:
            for data in data:
                self.push(data)
    def push(self, data):
        temp = Node()
        temp.setData(data)
        temp.setNext(self.head)
        self.head = temp
    def pop(self):
        if self.head is None:
            raise IndexError
        temp = self.head.getData()
        self.head = self.head.getNext()
        return temp
    def peek(self):
        if self.head is None:
            raise IndexError
        return self.head.getData()
our_list = ["first", "second", "third", "fourth"]
our_stack = Stack(our_list)
print our_stack.pop()
print our_stack.pop()
```

Performance

Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	$O(1)$
Time Complexity of Push()	$O(1)$ (Average)
Time Complexity of Pop()	$O(1)$
Time Complexity of Top()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of DeleteStack()	$O(n)$

4.6 Comparison of Implementations

Comparing Incremental Strategy and Doubling Strategy

We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations. We start with an empty stack represented by an array of size 1.

We call *amortized* time of a push operation is the average time taken by a push over the series of operations, that is, $T(n)/n$.

Incremental Strategy

The amortized time (average time per operation) of a push operation is $O(n)$ $[O(n^2)/n]$.

Doubling Strategy

In this method, the amortized time of a push operation is $O(1)$ $[O(n)/n]$.

Note: For analysis, refer to the *Implementation* section.

Comparing Array Implementation and Linked List Implementation

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) – "*amortized*" bound takes time proportional to n.

Linked List Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time $O(1)$.
- Every operation uses extra space and time to deal with references.

4.7 Stacks: Problems & Solutions

Problem-1 Discuss how stacks can be used for checking balancing of symbols.

Solution: Stacks can be used to check whether the given expression has balanced symbols. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [- then it is written to the stack. When a closing delimiter is encountered like), }, or]- the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line. A linear-time $O(n)$ algorithm based on stack can be given as:

Algorithm:

a) Create a stack.
b) while (end of input is not reached) {
    1) If the character read is not a symbol to be balanced, ignore it.
    2) If the character is an opening symbol like (, [, {, push it onto the stack
    3) If it is a closing symbol like),],}, then if the stack is empty report an error. Otherwise pop the stack.
    4) If the symbol popped is not the corresponding opening symbol, report an error.
}
c) At end of input, if the stack is not empty report an error

Examples:

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression has a balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D}}	No	The last closing brace does not correspond with the first opening parenthesis

For tracing the algorithm let us assume that the input is: () (() [()])

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	

(	Push (	((	
)	Pop (Test if (and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop (Test if(and A[i] match? YES	([	
]	Pop [Test if [and A[i] match? YES	(	
)	Pop (Test if(and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

Time Complexity: $O(n)$. Since we are scanning the input only once. Space Complexity: $O(n)$ [for stack].

```
def checkSymbolBalance(input):
    symbolstack = Stack()
    balanced = 0
    for symbols in input:
        if symbols in ["(","{","["]:
            symbolstack.push(symbols)
        else:
            if symbolstack.isEmpty():
                balanced = 0
            else:
                topSymbol = symbolstack.pop()
            if not matches(topSymbol,symbols):
                    balanced = 0
                else:
                    balanced = 1
    return balanced
print checkSymbolBalance("([)]")
'''Output: 0'''
print checkSymbolBalance("{{([][])}()}")
'''Output: 1'''
```

Problem-2 Discuss infix to postfix conversion algorithm using stack.

Solution: Before discussing the algorithm, first let us see the definitions of infix, prefix and postfix expressions.

Infix: An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another Infix string.

A
A+B
(A+B)+ (C-D)

Prefix: A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.

A
+AB
++AB-CD

Postfix: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

A
AB+
AB+CD-+

Prefix and postfix notions are methods of writing mathematical expressions without parenthesis. Time to evaluate a postfix and prefix expression is $O(n)$, where n is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Now, let us focus on the algorithm. In infix expressions, the operator precedence is implicit unless we use parentheses. Therefore, for the infix to postfix conversion algorithm we have to define the operator precedence (or priority) inside the algorithm.

The table shows the precedence and their associativity (order of evaluation) among operators.

Token	Operator	Precedence	Associativity
( ) [ ] → .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right
+ -	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 an 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+

```
class Stack:
    def __init__(self):
        self.items = []

    #method for pushing an item on a stack
    def push(self,item):
        self.items.append(item)

    #method for popping an item from a stack
    def pop(self):
        return self.items.pop()

    #method to check whether the stack is empty or not
    def isEmpty(self):
        return (self.items == [])

    #method to get the top of the stack
    def peek(self):
        return self.items[-1]

    def __str__(self):
        return str(self.items)

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()

            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and
              (prec[opStack.peek()] >= prec[token]):
                postfixList.append(opStack.pop())
            opStack.push(token)
    while not opStack.isEmpty():
```

```
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
```

Problem-3 Discuss postfix evaluation using stacks?

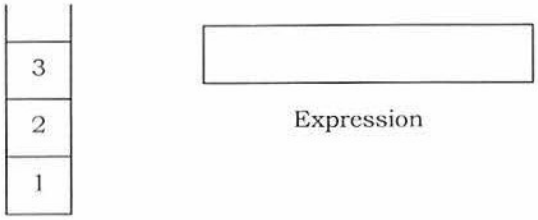
Solution:

Algorithm:

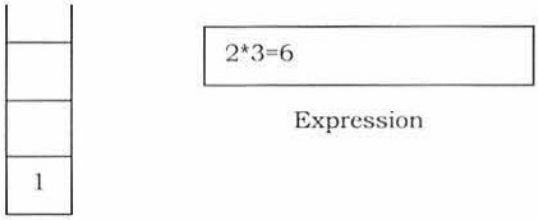
1 Scan the Postfix string from left to right.
2 Initialize an empty stack.
3 Repeat steps 4 and 5 till all the characters are scanned.
4 If the scanned character is an operand, push it onto the stack.
5 If the scanned character is an operator, and if the operator is a unary operator, then pop an element from the stack. If the operator is a 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-mentioned algorithm works using an example. Assume that the postfix string is 123*+5-.

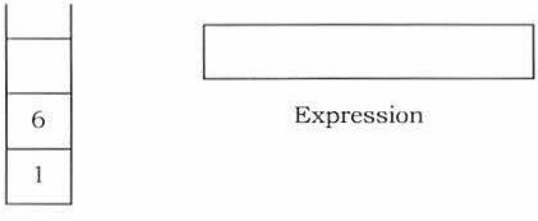
Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



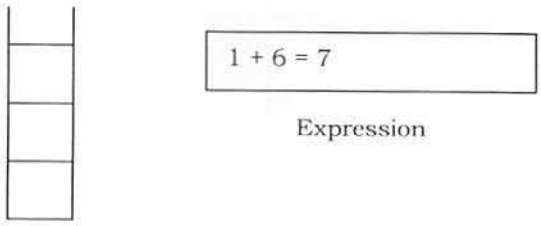
The next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



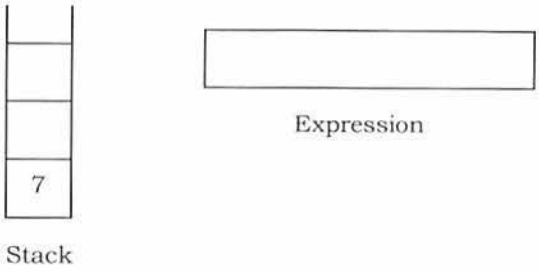
The value of the expression (2*3) that has been evaluated (6) is pushed into the stack.



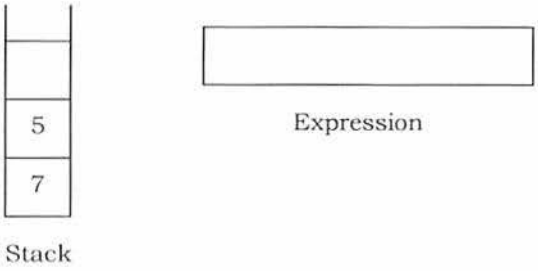
The next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



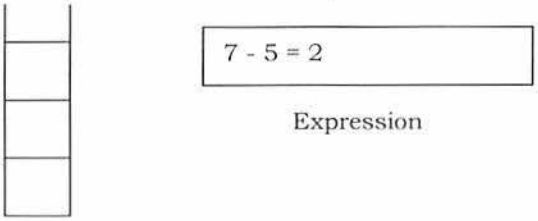
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



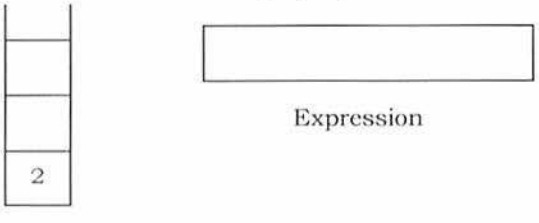
The next character scanned is "5", which is added to the stack.



The 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

```
class Stack:
    def __init__(self):
```

```
        self.items = []
    #method for pushing an item on a stack
    def push(self,item):
        self.items.append(item)
    #method for popping an item from a stack
    def pop(self):
        return self.items.pop()
    #method to check whether the stack is empty or not
    def isEmpty(self):
        return (self.items == [])
    def __str__(self):
        return str(self.items)
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()
    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()
def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
print(postfixEval('1 2 3 * + 5 -'))
```

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 that 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 *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 an example and trace it out. Initially let us assume that we have pushed 2, 6, 4, 1 and 5. Based on the above-mentioned algorithm the *min stack* will look like:

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

```
class SmartStack:
    def __init__(self):
        self.stack= []
        self.min = []

    def stack_push(self,x):
     self.stack.append(x)
     if not self.min or x <= self.stack_min():
                self.min.append(x)
     else:
                self.min.append(self.min[-1])

    def stack_pop(self):
        x = self.stack.pop()
        self.min.pop()
        return x

    def stack_min(self):
        return self.min[-1]
```

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

Solution: **Yes**. The main problem of the 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 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 equal* 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.

```
class SmartStack:
    def __init__(self):
        self.stack= []
        self.min = []
    def stack_push(self,x):
        self.stack.append(x)
        if not self.min or x <= self.stack_min():
            self.min.append(x)
    def stack_pop(self):
        x = self.stack.pop()
        if x == self.stack_min():
            self.min.pop()
        return x
    def stack_min(self):
        return self.min[-1]
```

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 the *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.

Solution: This is one of the simplest algorithms. What we do is, start two indexes, one at the beginning of the string and the other at the end of the string. Each time compare whether the values at both the indexes are the same or not. If the values are not the same then we say that the given string is not a palindrome.

If the values are the 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.

```
def isPalindrome(A):
    i=0
    j = len(A)-1
    while (i < j and A[i] == A[j]):
        i += 1
        j -= 1
    if (i < j ):
        print("Not a Palindrome")
        return 0
    else:
        print("Palindrome")
        return 1
isPalindrome(['m', 'a', 'd','a', 'm'])
```

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

Problem-9 For Problem-8, if the input is in singly linked list then how do we check whether the list elements form a palindrome (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 element's content with top of the stack. If they are the same then pop the stack and go to the next element in the input list.
- If they are not the same then the given string is not a palindrome.
- Continue this process until the stack is empty or the string is not a palindrome.

```
def isPalindrome(str):
    strStack = Stack()
    palindrome = False
    for char in str:
        strStack.push(char)
    for char in str:
        if char == strStack.pop():
            palindrome = True
        else:
            palindrome = False
    return palindrome
print isPalindrome("smadams")
```

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

Problem-11 Given a stack, how to reverse the elements of the stack 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 the stack.

```
class Stack(object):
    def __init__(self,items=[]):
        self.stack = items
    def is_empty(self):
        return not self.stack
    def pop(self):
        return self.stack.pop()
    def push(self,data):
        self.stack.append(data)
    def __repr__(self):
        return "Stack {0}".format(self.stack)
def reverseStack(stack):
    def reverseStackRecursive(stack,newStack=Stack()):
        if not stack.is_empty():
            newStack.push(stack.pop())
            reverseStackRecursive(stack,newStack)
        return newStack
    return reverseStackRecursive(stack)
stk = Stack(range(10))
print stk
print reverseStack(stk)
```

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.

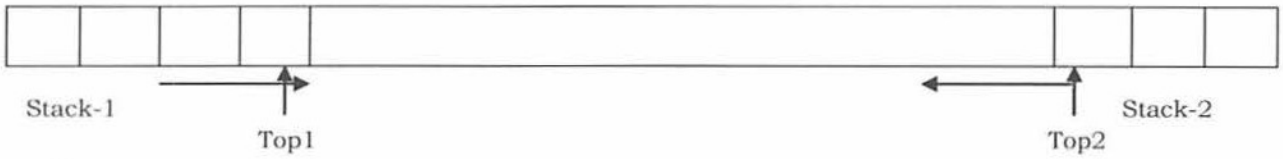
Solution: Refer Queues chapter.

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 *two* stacks using only one array? Our stack routines should not indicate an exception unless every slot in the array is used?

Solution:



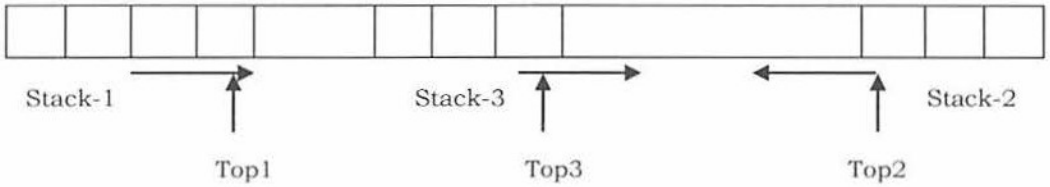
Algorithm:

- Start two indexes one at the left end and the 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 the left index.
- Similarly, if we want to push an element into the second stack then put the element at the right index.
- The first stack grows towards the right, and the second stack grows towards the 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 ways of solving it. Given below is one 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 into 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 of implementing the middle stack?

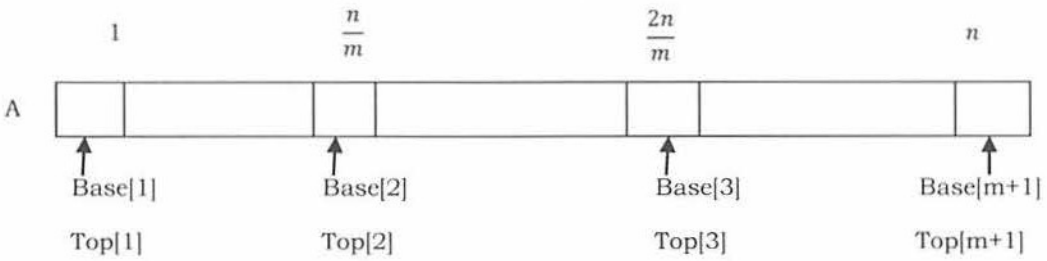
Solution: Yes. When either the left stack (which grows to the right) or the right stack (which grows to the left) bumps into the middle stack, we need to shift the entire middle stack to make room. The same happens if a push on the middle stack causes it to bump into the right stack.

To solve the above-mentioned problem (number of shifts) what we can do is: alternating pushes can 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 we 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 we'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: 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. Similar to the discussion in 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 $1 \leq i \leq m$ 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 $1 \leq i \leq m$.
- 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 the 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 toward the 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 toward the left.
3) If both of them are not possible then we can say that all stacks are full.

```
def push(StackID, 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.

```
def Pop(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 in the order they appear from left to right. Let S indicate a push and X indicate 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 in this chapter, we discussed that for dynamic array implementation of stacks, the 'repeated doubling' approach is used. 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: $30, 40 \ldots$ That means, for a given n value, we are creating the new arrays at: $\frac{n}{10}, \frac{n}{20}, \frac{n}{30}, \frac{n}{40} \ldots$ The total number of copy operations is:

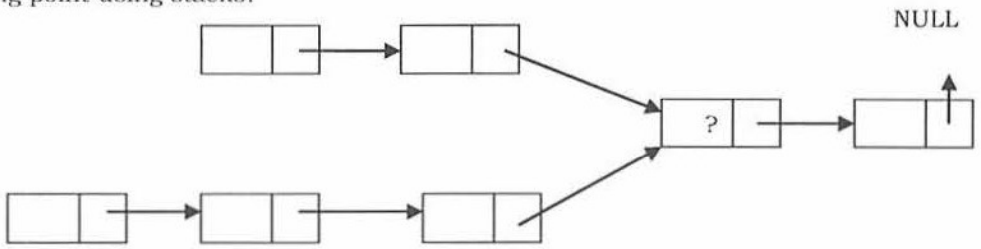
$$= \frac{n}{10} + \frac{n}{20} + \frac{n}{30} + \cdots 1 = \frac{n}{10}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right) = \frac{n}{10} logn \approx O(nlogn)$$

If we are performing n push operations, the cost 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 number 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, the number of push operations (S) should be greater than the number of pop operations (X).

Problem-21 Suppose there are two singly linked lists 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 lists before they intersect are unknown and both lists may have a different number. $List1$ may have n nodes before it reaches the intersection point and $List2$ may have m nodes before it reaches the intersection point where m and n may be $m = n, m < n$ or $m > n$. Can we find the merging point using stacks?

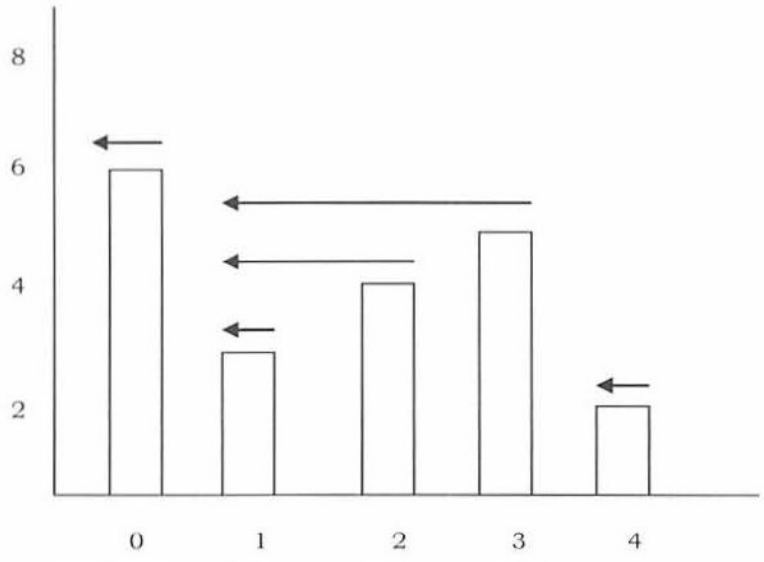


Solution: Yes. For algorithm refer to *Linked Lists* chapter.

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 $A[j] \leq A[i]$?

Another way 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 are used in 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 indicate 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 have a stock price that is less than the current price.

```
class Stack:
    def __init__(self):
        self.items = []

    #method for pushing an item on a stack
    def push(self,item):
        self.items.append(item)

    #method for popping an item from a stack
    def pop(self):
        return self.items.pop()

    #method to check whether the stack is empty or not
    def isEmpty(self):
        return (self.items == [])

    #method to get the top of the stack
    def peek(self):
        return self.items[-1]

    def __str__(self):
        return str(self.items)

def findingSpans(A):
    s = [None]*len(A)
    for i in range(0,len(A)):
        j = 1
        while j <= i and A[i] > A[i-j]:
            j = j + 1
            s[i] = j
        print s

findingSpans(['6', '3', '4', '5', '2'])
```

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

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

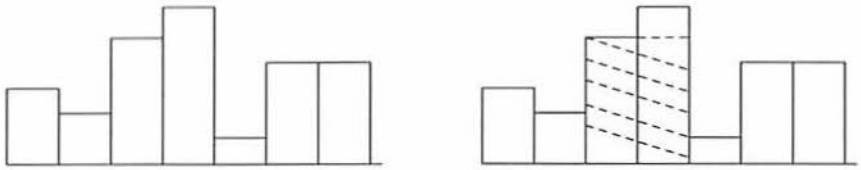
Solution: From the example above, we can see that span $S[i]$ on day i can be easily calculated if we know the closest day preceding i, such that the price is greater on that day than the price on day i. Let us call such a day as P. If such a day exists then the span is now defined as $S[i] = i - P$.

```
class Stack:
    def __init__(self):
        self.items = []

    #method for pushing an item on a stack
    def push(self,item):
        self.items.append(item)

    #method for popping an item from a stack
    def pop(self):
        return self.items.pop()

    #method to check whether the stack is empty or not
    def isEmpty(self):
        return (self.items == [])
```

```
    #method to get the top of the stack
    def peek(self):
        return self.items[-1]
    def __str__(self):
        return str(self.items)
def findingSpans(A):
    D = Stack()
    S = [None]*len(A)
    for i in range (0,len(A)):
            while not D.isEmpty() and A[i] > A[D.peek()]:
                    D.pop()
            if D.isEmpty():
                P = -1
            else:
                    P = D.peek()
            S[i] = i-P
            D.push(i)
    print S
findingSpans(['6', '3', '4', '5', '2'])
```

Time Complexity: Each index of the array is pushed into the stack exactly once 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 the algorithm (trace out an example and see how many times the inner loop becomes successful). 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 have equal widths but may have different heights. For example, the figure on the left shows a 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 rectangle possible. For the given example, the largest rectangle is the shared part.



Solution: A straightforward answer is to go to each bar in the histogram and find the maximum possible area in the 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 sub problems: 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 sub problem 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 sub problem 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 sub problems are finished when 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 sub problems by updating the maximum area with respect to the elements at the top.

```
def largestRectangleArea(self, height):
    stack=[]; i=0; maxArea=0
    while i<len(height):
        if stack==[] or height[i]>height[stack[len(stack)-1]]:
            stack.append(i)
        else:
            curr=stack.pop()
```

```
            width=i if stack==[] else i-stack[len(stack)-1]-1
            maxArea=max(maxArea,width*height[curr])
            i-=1
        i+=1
    while stack!=[]:
        curr=stack.pop()
        width=i if stack==[] else len(height)-stack[len(stack)-1]-1
        maxArea=max(maxArea,width*height[curr])
    return maxArea
```

At the 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].

Problem-26 Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11, 10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

Solution: Refer *Queues* chapter.

Problem-27 Recursively remove all adjacent duplicates: Given a string of characters, recursively remove adjacent duplicate characters from string. The output string should not have any adjacent duplicates.

Input: careermonk	*Input*: mississippi
Output: camonk	*Output*: m

Solution: This solution runs with the concept of in-place stack. When element on stack doesn't match the current character, we add it to stack. When it matches to stack top, we skip characters until the element matches the top of stack and remove the element from stack.

```
def removeAdjacentDuplicates(str):
    stkptr = -1
    i = 0
    size=len(str)
    while i<size:
        if (stkptr == -1 or str[stkptr]!=str[i]):
            stkptr += 1
            str[stkptr]=str[i]
            i += 1
        else:
            while i < size and str[stkptr]==str[i]:
                i += 1
            stkptr -= 1
    stkptr += 1
    str = str[0:stkptr]
    print str
removeAdjacentDuplicates(['6', '2', '4', '1', '2', '1', '2', '2', '1'])
```

Time Complexity: $O(n)$. Space Complexity: O(1) as the stack simulation is done inplace.

Problem-28 Given an array of elements, replace every element with nearest greater element on the right of that element.

Solution: One simple approach would involve scanning the array elements and for each of the elements, scan the remaining elements and find the nearest greater element.

```
def replaceWithNearestGreaterElement(A):
    nextNearestGreater = float("-inf")
    i = j = 0
    for i in range(0,len(A)-1):
            nextNearestGreater = float("-inf")
            for j in range(i+1,len(A)):
                    if A[i] < A[j]:
                            nextNearestGreater = A[j]
```

```
                                    break
                print("For "+ str(A[i]) +", " + str(nextNearestGreater) +" is the nearest greater element")
```

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

Problem-29 For Problem-28, can we improve the complexity?

Solution: The approach is pretty much similar to Problem-22. Create a stack and push the first element. For rest of the elements, mark the current element as *nextNearestGreater*. If stack is not empty, then pop an element from stack and compare it with *nextNearestGreater*. If *nextNearestGreater* is greater than the popped element, then *nextNearestGreater* is the next greater element for the popped element. Keep popping from the stack while the popped element is smaller than *nextNearestGreater*. *nextNearestGreater* becomes the next greater element for all such popped elements. If *nextNearestGreater* is smaller than the popped element, then push the popped element back.

```
def replaceWithNearestGreaterElementWithStack(A):
    i = 0
    S = Stack()
    S.push(A[0])
    for i in range(0,len(A)):
            nextNearestGreater = A[i]
            if not S.isEmpty():
                    element = S.pop()
                    while (element < nextNearestGreater):
                        print(str(element)+"--->"+str(nextNearestGreater)
                        if S.isEmpty():
                            break
                        element = S.pop()
                    if element > nextNearestGreater:
                        S.push(element)
            S.push(nextNearestGreater)
    while (not S.isEmpty()):
            element = S.pop()
            nextNearestGreater = float("-inf")
            print(str(element)+"->"+str(nextNearestGreater)
replaceWithNearestGreaterElementWithStack([6, 12, 4, 1, 2, 111, 2, 2, 10])
```

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

Problem-30 Given a singly linked list L: L_1-> L_2-> L_3...-> L_{n-1}-> L_n, reorder it to: L_1-> L_n-> L_2-> L_{n-1}.....

Solution:

```
def reorderList(self, head):
    if head == None:
        return head
    stack = []
    temp = head
    while temp != None:
        stack.append(temp)
        temp = temp.next
    list = head
    fromHead = head
    fromStack = True
    while (fromStack and list != stack[-1]) or ( not fromStack and list != fromHead):
        if fromStack:
                fromHead = fromHead.next
                list.next = stack.pop()
                fromStack = False
        else:
                list.next = fromHead
                fromStack = True
        list = list.next
    list.next = None
```

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

CHAPTER 5

QUEUES

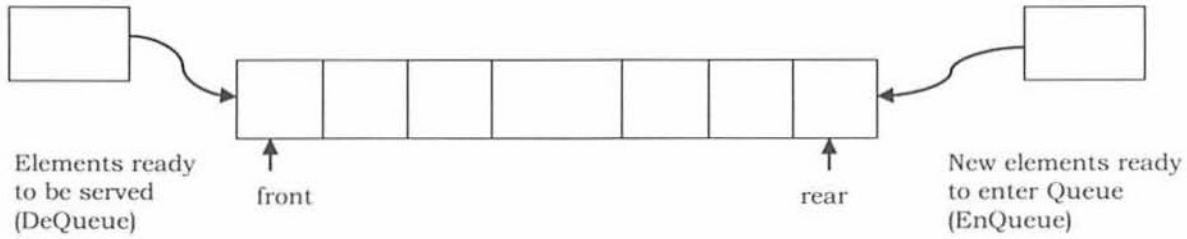
5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which 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 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*.

DeQueueing an empty queue is called *underflow* and *EnQueuing* 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 are Queues Used

The concept of a queue can be explained by observing a line at a reservation counter. When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

As this happens, the next person will come at the head of the line, will exit the queue and will be served. As each person at the head of the line keeps exiting the queue, we move towards the head of the line. Finally we will reach the head of the line and we will exit the queue and be served. This behavior is very useful in 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 the 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 the 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

Similar to other ADTs, executing *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and executing *EnQueue* on a full queue throws a "*Full Queue Exception*".

5.5 Applications

Following are the some of the applications that use queues.

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

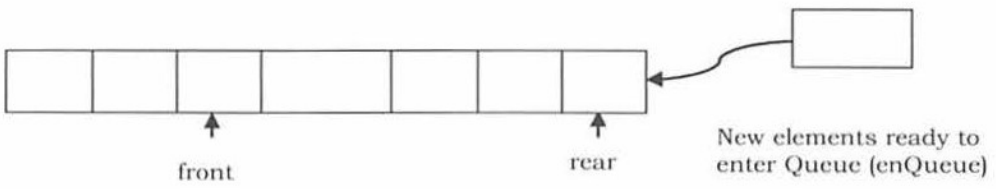
There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

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

Why Circular Arrays?

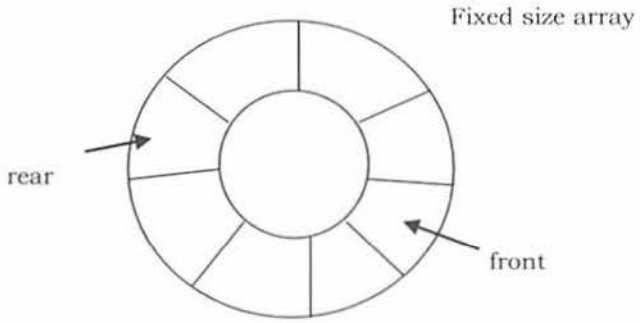
First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes easy to understand.

In the example 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 the last element and the first array elements as 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 similar to stack array implementations. Refer to *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 the 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 an empty queue it will throw *empty queue exception*.

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

```
class Queue(object):
 def __init__(self, limit = 5):
        self.que = []
        self.limit = limit
        self.front = None
        self.rear = None
        self.size = 0
 def isEmpty(self):
        return self.size <= 0
 def enQueue(self, item):
        if self.size >= self.limit:
                print 'Queue Overflow!'
                return
        else:
                self.que.append(item)

        if self.front is None:
                self.front = self.rear = 0
        else:
                self.rear = self.size
        self.size += 1
        print 'Queue after enQueue',self.que
 def deQueue(self):
        if self.size <= 0:
                print 'Queue Underflow!'
                return 0
        else:
                self.que.pop(0)
                self.size -= 1
                if self.size == 0:
                        self.front = self.rear = None
                else:
                        self.rear = self.size-1
                print 'Queue after deQueue',self.que
 def queueRear(self):
        if self.rear is None:
                print "Sorry, the queue is empty!"
```

```
                raise IndexError
            return self.que[self.rear]
    def queueFront(self):
            if self.front is None:
                print "Sorry, the queue is empty!"
                raise IndexError
            return self.que[self.front]
    def size(self):
            return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

Performance and 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 as prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

Dynamic Circular Array Implementation

```
class Queue(object):
 def __init__(self, limit = 5):
            self.que = []
            self.limit = limit
            self.front = None
            self.rear = None
            self.size = 0
 def isEmpty(self):
            return self.size <= 0
 def enQueue(self, item):
            if self.size >= self.limit:
                self.resize()
            self.que.append(item)
            if self.front is None:
                self.front = self.rear = 0
            else:
                self.rear = self.size
            self.size += 1
```

```
            print 'Queue after enQueue',self.que
    def deQueue(self):
            if self.size <= 0:
                    print 'Queue Underflow!'
                    return 0
            else:
                    self.que.pop(0)
                    self.size -= 1
                    if self.size == 0:
                            self.front = self.rear = None
                    else:
                            self.rear = self.size-1
                    print 'Queue after deQueue',self.que
    def queueRear(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.que[self.rear]
    def queueFront(self):
            if self.front is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.que[self.front]
    def size(self):
            return self.size
    def resize(self):
            newQue = list(self.que)
            self.limit = 2*self.limit
            self.que = newQue
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("four")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("five")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("six")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

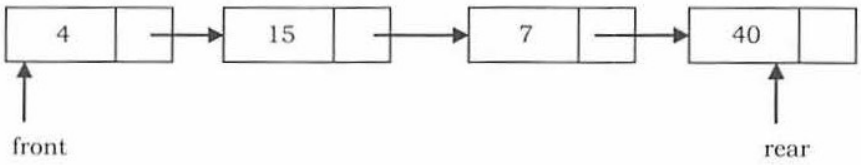
Performance

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

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




```
#Node of a Singly Linked List
class Node:
 #constructor
 def __init__(self, data=None, next=None):
        self.data = data
        self.last = None
        self.next = next
 #method for setting the data field of the node
 def setData(self,data):
        self.data = data
 #method for getting the data field of the node
 def getData(self):
        return self.data
 #method for setting the next field of the node
 def setNext(self,next):
        self.next = next
 #method for getting the next field of the node
 def getNext(self):
        return self.next
 #method for setting the last field of the node
 def setLast(self,last):
        self.last = last
 #method for getting the last field of the node
 def getLast(self):
        return self.last
 #returns true if the node points to another node
 def hasNext(self):
        return self.next != None
class Queue(object):
 def __init__(self, data=None):
        self.front = None
        self.rear = None
        self.size = 0
 def enQueue(self, data):
        self.lastNode = self.front
        self.front = Node(data, self.front)
        if self.lastNode:
                self.lastNode.setLast(self.front)
        if self.rear is None:
                self.rear = self.front
```

```
            self.size += 1
    def queueRear(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.rear.getData()
    def queueFront(self):
            if self.front is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            return self.front.getData()
    def deQueue(self):
            if self.rear is None:
                    print "Sorry, the queue is empty!"
                    raise IndexError
            result = self.rear.getData()
            self.rear = self.rear.last
            self.size -= 1
            return result
    def size(self):
            return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
print "Dequeuing: "+que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```

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 similar to stack implementations and *Stacks* chapter.

5.7 Queues: Problems & Solutions

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:

```
class Stack(object):
    def __init__(self, limit = 10):
            self.stk = []
            self.limit = limit
    def isEmpty(self):
```

```
            return len(self.stk) <= 0
    def push(self, item):
            if len(self.stk) >= self.limit:
                    print 'Stack Overflow!'
            else:
                    self.stk.append(item)
            print 'Stack after Push',self.stk
    def pop(self):
            if len(self.stk) <= 0:
                    print 'Stack Underflow!'
                    return 0
            else:
                    return self.stk.pop()
    def peek(self):
            if len(self.stk) <= 0:
                    print 'Stack Underflow!'
                    return 0
            else:
                    return self.stk[-1]
    def size(self):
            return len(self.stk)
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self, data=None, next=None):
            self.data = data
            self.last = None
            self.next = next
    #method for setting the data field of the node
    def setData(self,data):
            self.data = data
    #method for getting the data field of the node
    def getData(self):
            return self.data
    #method for setting the next field of the node
    def setNext(self,next):
            self.next = next
    #method for getting the next field of the node
    def getNext(self):
            return self.next
    #method for setting the last field of the node
    def setLast(self,last):
            self.last = last
    #method for getting the last field of the node
    def getLast(self):
            return self.last
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
class Queue(object):
    def __init__(self, data=None):
            self.front = None
            self.rear = None
            self.size = 0
    def enQueue(self, data):
            self.lastNode = self.front
            self.front = Node(data, self.front)
            if self.lastNode:
                    self.lastNode.setLast(self.front)
            if self.rear is None:
                    self.rear = self.front
```

```
            self.size += 1
        def queueRear(self):
            if self.rear is None:
                print "Sorry, the queue is empty!"
                raise IndexError
            return self.rear.getData()
        def queueFront(self):
            if self.front is None:
                print "Sorry, the queue is empty!"
                raise IndexError
            return self.front.getData()
        def deQueue(self):
            if self.rear is None:
                print "Sorry, the queue is empty!"
                raise IndexError
            result = self.rear.getData()
            self.rear = self.rear.last
            self.size -= 1
            return result
        def size(self):
            return self.size
        def isEmpty(self):
            return self.size == 0
    que = Queue()
    for i in xrange(5):
      que.enQueue(i)
    # suppose your have a Queue my_queue
    aux_stack = Stack()
    while not que.isEmpty():
      aux_stack.push(que.deQueue())
    while not aux_stack.isEmpty():
      que.enQueue(aux_stack.pop())
    for i in xrange(5):
      print que.deQueue()
```

Time Complexity: $O(n)$.

Problem-2 How can you implement a queue using two stacks?

Solution: The key insight is that a stack reverses order (while a queue doesn't). A sequence of elements pushed on a stack comes back in reversed order when popped. Consequently, two stacks chained together will return elements in the same order, since reversed order reversed again is original order.

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.

EnQueue Algorithm

- Just push on to stack S1

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 a 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.

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). The amortized complexity of pop operation is O(1).

```
class Queue(object):
    def __init__(self):
        self.S1 = []
        self.S2 = []
    def enqueue(self,element):
        self.S1.append(element)
    def dequeue(self):
        if not self.S2:
            while self.S1:
                self.S2.append(self.S1.pop())
        return self.S2.pop()
q = Queue()
for i in xrange(5):
    q.enqueue(i)
for i in xrange(5):
    print q.dequeue()
```

Problem-3 Show how you can 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.

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

Push Operation Algorithm: Insert the element in whichever queue is not empty.

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

Time Complexity: O(1).

Pop Operation Algorithm: Transfer $n-1$ elements to the 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.

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 the other.

```
class Queue(object):
    def __init__(self):
        self.queue=[]
    def isEmpty(self):
        return self.queue==[]
    def enqueue(self,x):
        self.queue.append(x)
    def dequeue(self):
        if self.queue:
            a=self.queue[0]
            self.queue.remove(a)
            return a
        else:
            raise IndexError,'queue is empty'
    def size(self):
        return len(self.queue)
class Stack(object):
    def __init__(self):
        self.Q1=Queue()
```

```
            self.Q2=Queue()
    def isEmpty(self):
            return self.Q1.isEmpty() and self.Q2.isEmpty()
    def push(self,item):
            if self.Q2.isEmpty():
                    self.Q1.enqueue(item)
            else:
                    self.Q2.enqueue(item)
    def pop(self):
            if self.isEmpty():
                    raise IndexError,'stack is empty'
            elif self.Q2.isEmpty():
                    while not self.Q1.isEmpty():
                            cur=self.Q1.dequeue()
                            if self.Q1.isEmpty():
                                    return cur
                            self.Q2.enqueue(cur)
            else:
                    while not self.Q2.isEmpty():
                            cur=self.Q2.dequeue()
                            if self.Q2.isEmpty():
                                    return cur
                            self.Q1.enqueue(cur)
stk = Stack()
for i in xrange(5):
   stk.push(i)
for i in xrange(5):
   print stk.pop()
```

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 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 supports insertion and deletion 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.

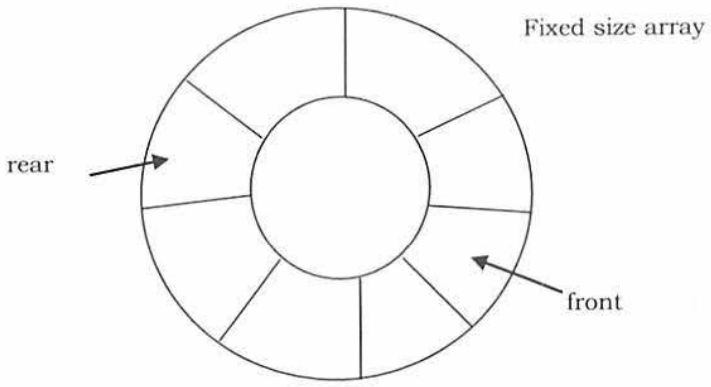
Solution: Assume the elements of queue Q are $a_1, a_2 \ldots a_n$. Dequeuing all elements and pushing them onto the stack will result in a stack with a_n at the top and a_1 at the bottom. This is done in O(n) time as dequeue and each push require constant time per operation. The queue is now empty. By popping all elements and pushing them on 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 figure to get a clear idea of 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:

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

Problem-7 What is the most appropriate data structure to print elements of queue in reverse order?

Solution: Stack.

Problem-8 Implement doubly ended queues. A double-ended queue is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Solution: We will create a new class for the implementation of the abstract data type deque. In removeFront we use the pop method to remove the last element from the list. However, in removeRear, the pop(0) method must remove the first element of the list. Likewise, we need to use the insert method in addRear since the append method assumes the addition of a new element to the end of the list.

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0,item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

Problem-9 Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11,

10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

Solution:

```
import math
def checkStackPairwiseOrder(stk):
    que = Queue()
    pairwiseOrdered = 1
    #Reverse Stack elements
    while not stk.isEmpty():
            que.enQueue(stk.pop())
    while not que.isEmpty():
            stk.push(que.deQueue())
    while not stk.isEmpty():
            n = stk.pop()
            que.enQueue(n)
            if not stk.isEmpty():
                    m = stk.pop()
                    que.enQueue(m)
                    if (abs(n - m) != 1):
                            pairwiseOrdered = 0
                            break
    while not que.isEmpty():
            stk.push(que.deQueue())
    return pairwiseOrdered
stk = Stack()
stk.push(-2)
stk.push(-3)
stk.push(11)
stk.push(10)
stk.push(5)
stk.push(6)
stk.push(20)
stk.push(21)
print checkStackPairwiseOrder(stk)
```

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

Problem-10 Given a queue of integers, rearrange the elements by interleaving the first half of the list with the second half of the list. For example, suppose a queue stores the following sequence of values: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Consider the two halves of this list: first half: [11, 12, 13, 14, 15] second half: [16, 17, 18, 19, 20]. These are combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (11 and 16), then the second values from each half (12 and 17), then the third values from each half (13 and 18), and so on. In each pair, the value from the first half appears before the value from the second half. Thus, after the call, the queue stores the following values: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20].

Solution:

```
def interLeavingQueue(que):
    stk = Stack()
    halfSize = que.size//2
    for i in range(0,halfSize):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
    for i in range(0,halfSize):
        que.enQueue(que.deQueue())
    for i in range(0,halfSize):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
        que.enQueue(que.deQueue())
que = Queue()
```

```
que.enQueue(11)
que.enQueue(12)
que.enQueue(13)
que.enQueue(14)
que.enQueue(15)
que.enQueue(16)
que.enQueue(17)
que.enQueue(18)
que.enQueue(19)
que.enQueue(20)
interLeavingQueue(que)
while not que.isEmpty():
   print que.deQueue()
```

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

Problem-11 Given an integer k and a queue of integers, how do you reverse the order of the first k elements of the queue, leaving the other elements in the same relative order? For example, if k=4 and queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90]; the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

Solution:

```
def reverseQueueFirstKElements(que, k):
    stk = Stack()
    if que == None or k > que.size:
        return
    for i in range(0,k):
        stk.push(que.deQueue())
    while not stk.isEmpty():
        que.enQueue(stk.pop())
    for i in range(0,que.size-k):
        que.enQueue(que.deQueue())
que = Queue()
que.enQueue(11)
que.enQueue(12)
    que.enQueue(13)
que.enQueue(14)
que.enQueue(15)
que.enQueue(16)
que.enQueue(17)
que.enQueue(18)
que.enQueue(19)
que.enQueue(20)
que.enQueue(21)
que.enQueue(22)
reverseQueueFirstKElements(que, 4)
while not que.isEmpty():
   print que.deQueue()
```

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

Problem-12 Implement producer consumer problem with python threads and queues.

Solution:

```
#!/usr/bin/env python
from random import randint
from time import sleep
from Queue import Queue
from myThread import MyThread
def writeQ(queue):
    print 'producing object for Q...',
    queue.put('MONK', 1)
    print "size now", queue.qsize()
```

```
def readQ(queue):
    val = queue.get(1)
    print 'consumed object from Q... size now', queue.qsize()
def producer(queue, loops):
    for i in range(loops):
      writeQ(queue)
      sleep(randint(1, 3))
def consumer(queue, loops):
    for i in range(loops):
      readQ(queue)
      sleep(randint(2, 5))
funcs = [producer, consumer]
nfuncs = range(len(funcs))
nloops = randint(2, 5)
q = Queue(32)
threads = []
for i in nfuncs:
  t = MyThread(funcs[i], (q, nloops),
    funcs[i].__name__)
  threads.append(t)
for i in nfuncs:
  threads[i].start()
for i in nfuncs:
  threads[i].join()
print 'all DONE'
```

As you can see, the producer and consumer do not necessarily alternate in execution. In this solution, we use the Queue. We use random.randint() to make production and consumption somewhat varied.

The writeQ() and readQ() functions each have a specific purpose: to place an object in the queue—we are using the string 'MONK', for example—and to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

The producer() is going to run as a single thread whose sole purpose is to produce an item for the queue, wait for a bit, and then do it again, up to the specified number of times, chosen randomly per script execution. The consumer() will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the producer sleeps is in general shorter than the amount of time the consumer sleeps. This is to discourage the consumer from trying to take items from an empty queue. By giving the producer a shorter time period of waiting, it is more likely that there will already be an object for the consumer to consume by the time their turn rolls around again.

These are just setup lines to set the total number of threads that are to be spawned and executed.

Finally, we have our main() function, which should look quite similar to the main() in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single-threaded program that attempts to do all of the tasks.

We illustrated how a single-threaded process can limit an application's performance. In particular, programs with independent, non-deterministic, and non-causal tasks that execute sequentially can be improved by division into separate tasks executed by individual threads. Not all applications will benefit from multithreading due to overhead and the fact that the Python interpreter is a single-threaded application, but now you are more cognizant of Python's threading capabilities and can use this tool to your advantage when appropriate.

Problem-13 Given a string , write a Python method to check whether it is a palindrome or nor using doubly ended queue.

Solution:

```
class Deque:
  def __init__(self):
    self.items = []
```

```
    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)

def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual

print(palchecker("lsdkjfskf"))
print(palchecker("madam"))
```

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

CHAPTER

6

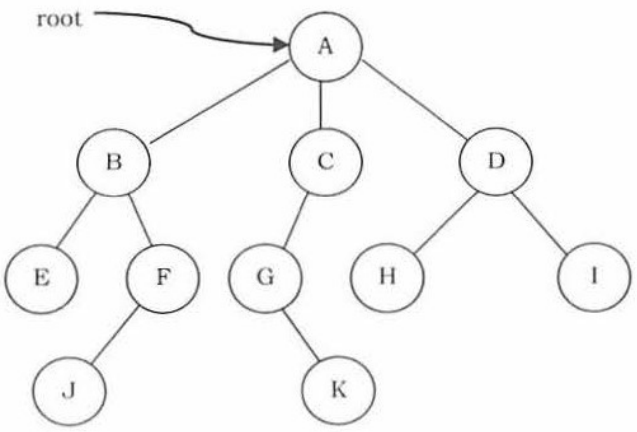
TREES

6.1 What is a Tree?

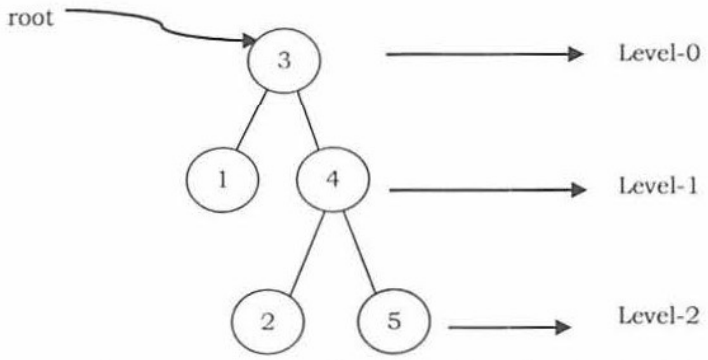
A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Tree is an example of non-linear data structures. A *tree* structure is a way of representing the hierarchical nature of a structure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

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

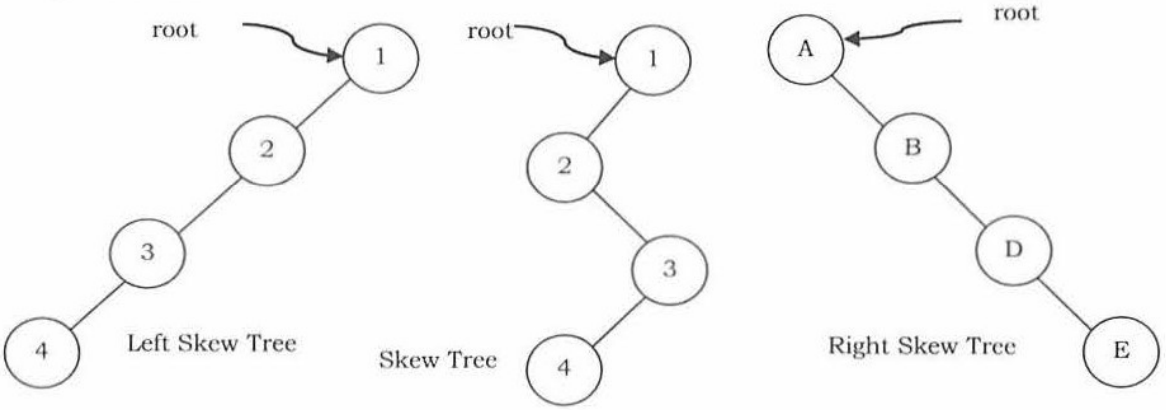
6.2 Glossary



- 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 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 of K.
- The set of all nodes at a given depth is called the *level* of the tree (B, C and D are the same level). The root node is at level zero.



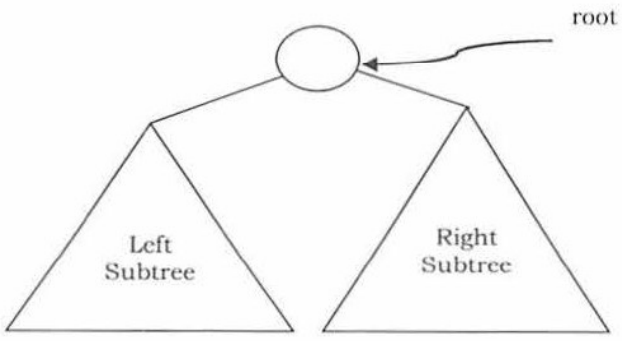
- The *depth* of a node is the length of the path from the root to the node (depth of G is 2, $A - C - G$).
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the 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.
- The size of a node is the number of descendants it has including itself (the 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 *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.



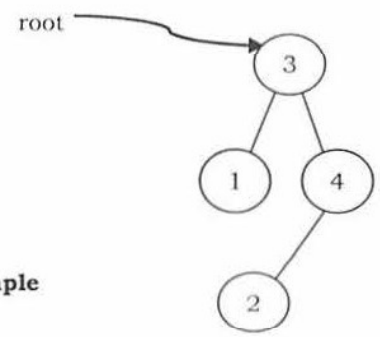
6.3 Binary Trees

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

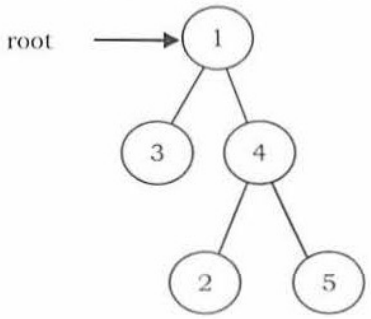


Example

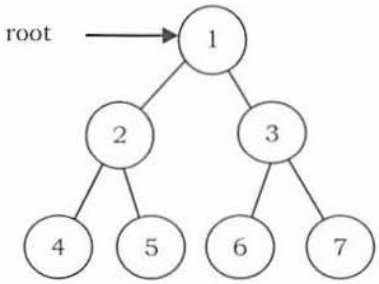


6.4 Types of Binary Trees

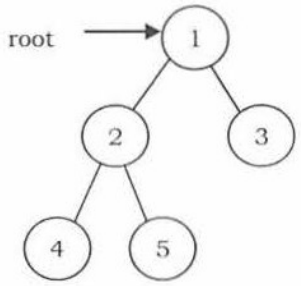
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



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

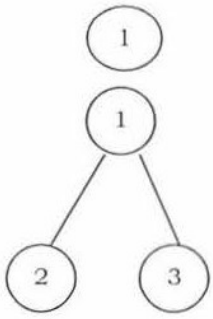


Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes 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.

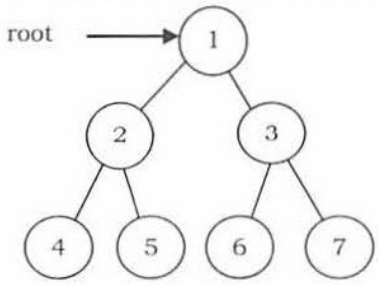


6.5 Properties of Binary Trees

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
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$



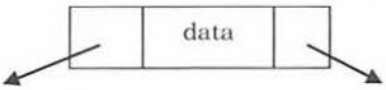
$h = 2$ $2^2 = 4$

From the diagram we can infer the following properties:

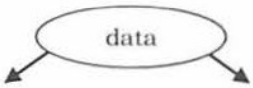
- The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are h levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum). For more information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

Structure of Binary Trees

Now let us define structure of the binary tree. One way to represent a node (which contains data) is to have two links which point to left and right children along with data fields as shown below:

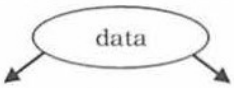


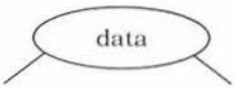
Or




```
'''Binary Tree Class and its methods'''
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data            #root node
        self.left = None            #left child
        self.right = None           #right child
    #set data
    def setData(self, data):
        self.data = data
    #get data
    def getData(self):
        return self.data
    #get left child of a node
    def getLeft(self):
        return self.left
    #get right child of a node
    def getRight(self):
        return self.right
```

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





Operations on Binary Trees

Basic Operations

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

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree

- Finding the level which has maximum sum
- Finding the 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 an important role:

- Expression trees are used in compilers.
- Huffman coding trees that 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 Queue (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 node is processed only once but it may be visited more than once. As we have already seen 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 a particular order. In addition, all nodes are processed in the *traversal by 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 denoted with "*D*"), traversing to the left child node (denoted with "*L*"), and traversing to the right child node (denoted 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

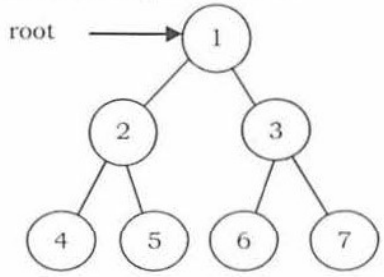
The sequence in which these entities (nodes) are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node (*D*) and if *D* comes in the middle then it does not matter whether *L* is 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 are reduced to 3 and these are:

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

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

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

Let us use the diagram below for the remaining discussion.



PreOrder Traversal

In preorder traversal, each node is processed before (pre) either of its subtrees. 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, 1 is processed first, then the left subtree, and this is followed by the right subtree.

Therefore, processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the 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

```
# Pre-order recursive traversal. The nodes' values are appended to the result list in traversal order
def preorderRecursive(root, result):
    if not root:
        return
    result.append(root.data)
    preorderRecursive(root.left, result)
    preorderRecursive(root.right, result)
```

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

Non-Recursive Preorder Traversal

In the 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 the right subtree. To simulate the same, first we process the current node and before going to the 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.

```
#Pre-order iterative traversal. The nodes' values are appended to the result list in traversal order
def preorder_iterative(root, result):
    if not root:
        return
    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()
        result.append(node.data)
        if node.right: stack.append(node.right)
        if node.left: stack.append(node.left)
```

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

```
# In-order recursive traversal. The nodes' values are appended to the result list in traversal order
def inorderRecursive(root, result):
    if not root:
        return
    inorderRecursive(root.left, result)
```

```
    result.append(root.data)
    inorderRecursive(root.right, result)
```

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

Non-Recursive Inorder Traversal

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

```
# In-order iterative traversal. The nodes' values are appended to the result list in traversal order
def inorderIterative(root, result):
    if not root:
        return
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            1result.append(node.data)
            node = node.right
```

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

```
# Post-order recursive traversal. The nodes' values are appended to the result list in traversal order
def postorderRecursive(root, result):
    if not root:
        return
    postorderRecursive(root.left, result)
    postorderRecursive(root.right, result)
    result.append(root.data)
```

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

Non-Recursive Postorder Traversal

In preorder and inorder traversals, after popping 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 the left subtree we will visit the current node and after processing the right subtree we will visit 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 the left subtree or the right subtree.

We use a *previous* variable to keep track of the earlier traversed node. Let's assume *current* is the current node that is on top of the stack. When *previous* is *current's* parent, we are traversing down the tree. In this case, we try to traverse to *current's* left child if available (i.e., push left child to the stack). If it is not available, we look at *current's* right child. If both left and right child do not exist (ie, *current* is a leaf node), we print *current's* value and pop it off the stack.

If prev is *current's* left child, we are traversing up the tree from the left. We look at *current's* right child. If it is available, then traverse down the right child (i.e., push right child to the stack); otherwise print *current's* value and pop it off the stack. If *previous* is *current's* right child, we are traversing up the tree from the right. In this case, we print *current's* value and pop it off the stack.

```
# Post-order iterative traversal. The nodes' values are appended to the result list in traversal order
def postorderIterative(root, result):
    if not root:
        return
    visited = set()
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            if node.right and not node.right in visited:
                stack.append(node)
                node = node.right
            else:
                visited.add(node)
                result.append(node.data)
                node = None
```

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 the tree are visited in the order: 1 2 3 4 5 6 7

```
import Queue
def levelOrder(root, result):
    if root is None:
        return
    q = Queue.Queue()
    q.put(root )
    node = None
    while not q.empty():
        node = q.get()                    # dequeue FIFO
        result.append(node.getData())
        if node.getLeft()is not None:
            q.put( node.getLeft() )
        if node.getRight() is not None:
            q.put( node.getRight() )
```

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.

Binary Trees: Problems & Solutions

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 the 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.

```
maxData = float("-infinity")
def findMaxRecursive(root): # maxData is the initially the value of root
    global maxData
    if not root:
```

```
        return maxData
    if root.getData() > maxData:
        maxData = root.getData()
    findMaxRecursive(root.getLeft())
    findMaxRecursive(root.getRight())
    return maxData
```

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

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

Solution: Using level order traversal: just observe the element's data while deleting.

```
def findMaxUsingLevelOrder(root):
    if root is None:
        return
    q = Queue()
    q.enQueue( root )
    node = None
    maxElement = 0
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO
        if maxElement < node.getData():
            maxElement = node.getData()
        if node.left is not None:
            q.enQueue( node.left )
        if node.right is not None:
            q.enQueue( node.right )
print maxElement
```

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 data is found in the tree. Recurse down the tree, choose the left or right branch by comparing data with each node's data.

```
def findRecursive(root, data):
    if not root:
        return 0
    if root.getData() == data:
        return 1
    else:
        temp = findRecursive(root.left, data)
        if temp == 1:
            return temp
        else:
            return findRecursive(root.right, data)
```

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.

```
def findUsingLevelOrder(root, data):
    if root is None:
        return -1
    q = Queue()
    q.enQueue( root )
    node = None
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO
        if data == node.getData():
```

```
                return 1
            if node.left is not None:
                q.enQueue( node.left )
            if node.right is not None:
                q.enQueue( node.right )
    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 find the node whose left or right child is NULL.

```
'''Binary Tree Class and its methods'''
class BinaryTree:
    def __init__(self, data):
        self.data = data          #root node
        self.left = None          #left child
        self.right = None         #right child
    #set data
    def setData(self, data):
        self.data = data
    #get data
    def getData(self):
        return self.data
    #get left child of a node
    def getLeft(self):
        return self.left
    #get right child of a node
    def getRight(self):
        return self.right

    def insertLeft(self, newNode):
        if self.left == None:
            self.left = BinaryTree(newNode)
        else:
            temp = BinaryTree(newNode)
            temp.left = self.left
            self.left = temp

    def insertRight(self, newNode):
        if self.right == None:
            self.right = BinaryTree(newNode)
        else:
            temp = BinaryTree(newNode)
            temp.right = self.right
            self.right = temp
# Insert using level order traversal
def insertInBinaryTreeUsingLevelOrder(root, data):
    newNode = BinaryTree(data)
    if root is None:
        root = newNode
        return root

    q = Queue()
    q.enQueue( root )
    node = None
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO

        if data == node.getData():
            return root
        if node.left is not None:
            q.enQueue( node.left )
        else:
```

```
                node.left = newNode
                return root
        if node.right is not None:
                q.enQueue( node.right )
        else:
                node.right = newNode
                return root
```

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.
def findSizeRecursive(root):
    if not root:
            return 0
    return findSizeRecursive(root.left) + findSizeRecursive(root.right) + 1
```

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

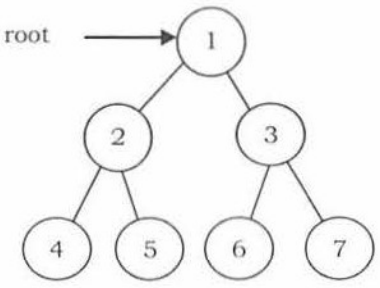
Problem-7 Can we solve Problem-6 without recursion?

Solution: Yes, using level order traversal.

```
def findSizeusingLevelOrder(root):
    if root is None:
            return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()                    # dequeue FIFO
            count += 1
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
    return count
```

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

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



Solution:

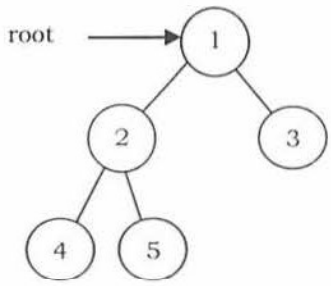
```
def levelOrderTraversalInReverse(root):
    if root is None:
            return 0
    q = Queue()
    s = Stack()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
```

```
            node = q.deQueue()                    # dequeue FIFO
            if node.left is not None:
                    q.enQueue( node.left )

            if node.right is not None:
                    q.enQueue( node.right )
            s.push(node)
    while(not s.isEmpty()):
            print s.pop().getData()
```

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 should we 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.

```
def deleteBinaryTree(root):
    if(root == None) :
        return
    deleteBinaryTree(root.left);
    deleteBinaryTree(root.right);
    del 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).

```
def maxDepth(root):
    if root == None:
       return 0
    return max(maxDepth(root.getLeft()),maxDepth(root.getRight()))+1
```

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

Problem-11 Can we solve 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.

```
def maxDepth(root):
    if root == None:
       return 0
    q = []
    q.append([root, 1])
    temp = 0
    while len(q) != 0:
       node, depth = q.pop()
       depth = max(temp, dep)
       if node.getLeft() != None:
```

```
            q.append([node.getLeft(), depth + 1])
        if node.getRight() != None:
            q.append([node.getRight(), depth + 1])
    return temp
```

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

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

Solution:

```
def deepestNode(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    node = None
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO
        if node.left is not None:
            q.enQueue( node.left )
        if node.right is not None:
            q.enQueue( node.right )
    return node.getData()
```

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 node's 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 nodes whose both left and right children are NULL are called leaf nodes.

```
def numberOfLeavesInBTusingLevelOrder(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO
        if node.left is None and node.right is None:
            count += 1
        else:
            if node.left is not None:
                q.enQueue( node.left )
            if node.right is not None:
                q.enQueue( node.right )
    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.

```
def numberOfFullNodesInBTusingLevelOrder(root):
    if root is None:
        return 0
```

```
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()                          # dequeue FIFO
            if node.left is not None and node.right is not None:
                    count += 1
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
    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 right child (but not both) are called half nodes.

```
def numberOfHalfNodesInBTusingLevelOrder(root):
    if root is None:
            return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()                          # dequeue FIFO
            if (node.left is None and node.right is not None) or (node.left is not None and node.right is None):
                    count += 1
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
    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.
def areStructurullySameTrees(root1, root2):
    if (not root1.left) and not (root1.right) and (not root2.left) and \
            not (root2.right) and root1.data == root2.data:
            return True
    if (root1.data != root2.data) or (root1.left and not root2.left) or \
            (not root1.left and root2.left) or (root1.right and not root2.right) \
            or (not root1.right and root2.right):
            return False
    left = areStructurullySameTrees(root1.left, root2.left) if root1.left and root2.left else True
    right = areStructurullySameTrees(root1.right, root2.right) if root1.right and root2.right else True
    return left and 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 subtrees recursively. Among these two values, we need to send maximum value along with current level (+1).

```
ptr = 0
def diameterOfTree(root):
    global ptr
    if(not root) :
            return 0
    left = diameterOfTree(root.left);
    right = diameterOfTree(root.right);
    if(left + right > ptr):
        ptr = left + right
    return max(left, right)+1

#Alternative Coding
def diameter(root):
    if (root == None):
            return 0
    lHeight = height(root.eft)
    rHeight = height(root.right)
    lDiameter = diameter(root.left)
    rDiameter = diameter(root.right)
    return max(lHeight + rHeight + 1, max(lDiameter, rDiameter))

# The function Compute the "height" of a tree. Height is the number of nodes along
# the longest path from the root node down to the farthest leaf node.
def height(root):
    if (root == None) :
            return 0
```

There is another solution and the complexity is $O(n)$. The main idea of this approach is that the node stores its left child's and right child's maximum diameter if the node's child is the "root", therefore, there is no need to recursively call the height method. The drawback is we need to add two extra variables in the node class.

```
def findMaxLen(root):
    nMaxLen = 0
    if (root == None):
            return 0
    if (root.left == None):
            root.nMaxLeft = 0
    if (root.right == None):
            root.nMaxRight = 0
    if (root.left != None):
            findMaxLen(root.left)
    if (root.right != None):
            findMaxLen(root.right)
    if (root.left != None):
            nTempMaxLen = 0
            nTempMaxLen = max(root.left.nMaxLeft, root.left.nMaxRight)
            root.nMaxLeft = nTempMaxLen + 1
    if (root.right != None):
            nTempMaxLen = 0
            nTempMaxLen = max(root.right.nMaxLeft, root.right.nMaxRight)
            root.nMaxRight = nTempMaxLen + 1
    if (root.nMaxLeft + root.nMaxRight > nMaxLen):
            nMaxLen = root.nMaxLeft + root.nMaxRight
    return nMaxLen
```

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

Problem-19 Give an algorithm for finding the level that has the maximum sum in the binary tree.

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

```
def findLevelwithMaxSum(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    q.enQueue( None )
    node = None
    level = maxLevel= currentSum = maxSum = 0
    while not q.isEmpty():
        node = q.deQueue()                          # dequeue FIFO
        # If the current level is completed then compare sums
        if(node == None):
            if(currentSum> maxSum):
                maxSum = currentSum
                maxLevel = level
            currentSum = 0
            #place the indicator for end of next level at the end of queue
            if not q.isEmpty():
                q.enQueue( None )
                level += 1
        else:
            currentSum += node.getData()
            if node.left is not None:
                q.enQueue( node.left )
            if node.right is not None:
                q.enQueue( node.right )
    return maxLevel
```

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

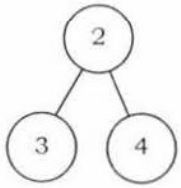
Problem-20 Given a binary tree, print out all its root-to-leaf paths.

Solution: Refer to comments in functions.

```
def pathsAppender(root, path, paths):
    if not root:
        return 0
    path.append(root.data)
    paths.append(path)
    pathsAppender(root.left, path+[root.data], paths)
    pathsAppender(root.right, path+[root.data], paths) # make sure it can be executed!
def pathsFinder(root):
    paths = []
    pathsAppender(root, [], paths)
    print 'paths:', paths
```

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

Problem-21 Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path 1->2->3 which represents the number 123. Find the total sum of all root-to-leaf numbers. For example,

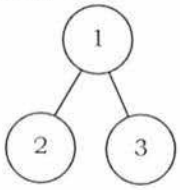


The root-to-leaf path 1->2 represents the number 23.
The root-to-leaf path 1->3 represents the number 24.
Return the sum = 23 + 24 = 47.

Solution:

```
    def sumNumbers(self, root):
        if not root:
                return 0
        current=0
        sum=[0]
        self.calSum(root, current, sum)
        return sum[0]
    def calSum(self, root, current, sum):
        if not root:
                return
        current=current*10+root.data
        if not root.left and not root.right:
                sum[0]+=current
                return
        self.calSum(root.left, current, sum)
        self.calSum(root.right,current, sum)
```

Problem-22 Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree. For example: Given the below binary tree,



Solution:

```
def treeMaximumSumPath(node, is_left=True, Lpath={}, Rpath={}):
    if is_left:
        # left sub-tree
        if not node.left:
            Lpath[node.id] = 0
            return 0
        else:
            Lpath[node.id] = node.data + max(
                treeMaximumSumPath(node.left, True, Lpath, Rpath),
                treeMaximumSumPath(node.left, False, Lpath, Rpath)
            )
            return Lpath[node.id]
    else:
        # right sub-tree
        if not node.right:
            Rpath[node.id] = 0
            return 0
        else:
            Rpath[node.id] = node.data + max(
                treeMaximumSumPath(node.right, True, Lpath, Rpath),
                treeMaximumSumPath(node.right, False, Lpath, Rpath)
            )
            return Rpath[node.id]
def maxsum_path(root):
    Lpath = {}
    Rpath = {}
    treeMaximumSumPath(root, True, Lpath, Rpath)
    treeMaximumSumPath(root, False, Lpath, Rpath)
    print 'Left-path:', Lpath
    print 'Right-path:', Rpath
    path2sum = dict((i, Lpath[i]+Rpath[i]) for i in Lpath.keys())
    i = max(path2sum, key=path2sum.get)
    print 'The path going through node', i, 'with max sum', path2sum[i]
    return path2sum[i]
```

Problem-23 Give an algorithm for checking the existence of path with given sum. That means, given a sum, check whether there exists a path from root to any of the nodes.

Solution: For this problem, the strategy is: subtract the node value from the sum before calling its children recursively, and check to see if the sum is 0 when we run out of tree.

```
def pathFinder(root, val, path, paths):
    if not root:
        return False

    if not root.left and not root.right:
        if root.data == val:
            path.append(root.data)
            paths.append(path)
            return True
        else:
            return False

    left = pathFinder(root.left, val-root.data, path+[root.data], paths)
    right = pathFinder(root.right, val-root.data, path+[root.data], paths) # make sure it can be executed!
    return left or right

def hasPathWithSum(root, val):
    paths = []
    pathFinder(root, val, [], paths)
    print 'sum:', val
    print 'paths:', paths
```

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

Problem-24 Give an algorithm for finding the sum of all elements in binary tree.

Solution: Recursively, call left subtree sum, right subtree sum and add their values to current nodes data.

```
def sumInBinaryTreeRecursive(root):
    if(root == None) :
        return 0
    return root.data+sumInBinaryTreeRecursive(root.left) + sumInBinaryTreeRecursive(root.right)
```

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

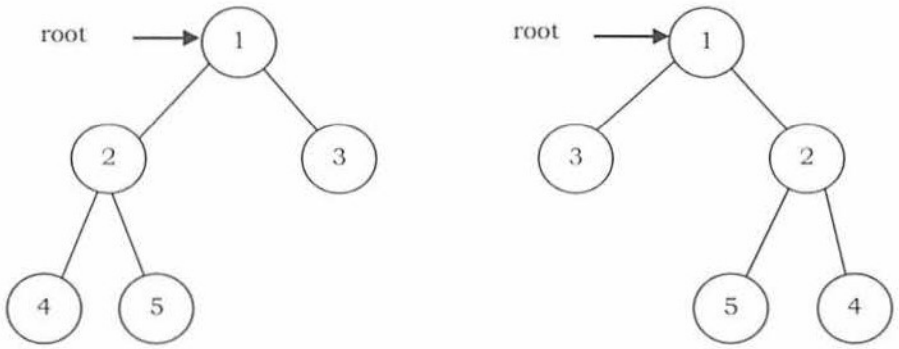
Problem-25 Can we solve Problem-24 without recursion?

Solution: We can use level order traversal with simple change. Every time after deleting an element from queue, add the node's data value to *sum* variable.

```
def sumInBinaryTreeLevelOrder(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    node = None
    sum = 0
    while not q.isEmpty():
        node = q.deQueue()                    # dequeue FIFO
        sum += node.getData()
        if node.left is not None:
            q.enQueue( node.left )
        if node.right is not None:
            q.enQueue( node.right )
    return sum
```

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

Problem-26 Give an algorithm for converting a tree to its mirror. Mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged. The trees below are mirrors to each other.



Solution:

```
def MirrorOfBinaryTree(root):
    if(root != None):
            MirrorOfBinaryTree(root.left)
            MirrorOfBinaryTree(root.right)
            # swap the pointers in this node
            temp  = root.left
            root.left  = root.right
            root.right = temp
    return root
```

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

Problem-27 Given two trees, give an algorithm for checking whether they are mirrors of each other.

Solution:

```
def AreMirrors(root1, root2):
    if(root1 == None and root2 == None):
            return 1
    if(root1 == None or root2 == None):
            return 0
    if(root1.data != root2.data):
            return 0
    else:
            return AreMirrors(root1.left, root2.right) and AreMirrors(root1.right, root2.left)
```

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

Problem-28 Give an algorithm for finding LCA (Least Common Ancestor) of two nodes in a Binary Tree.

Solution:

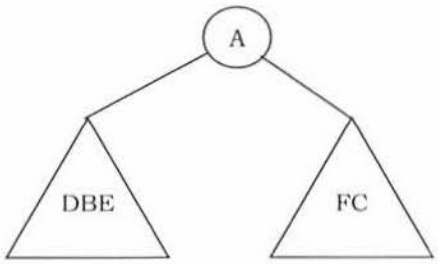
```
def lca(root, alpha, beta):
  if not root:
        return None
  if root.data == alpha or root.data == beta:
        return root
  left = lca(root.left, alpha, beta)
  right = lca(root.right, alpha, beta)
  if left and right:
    # alpha & beta are on both sides
    return root
  else:
    # EITHER alpha/beta is on one side
    # OR alpha/beta is not in L&R subtrees
    return left if left else right
```

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

Problem-29 Give an algorithm for constructing binary tree from given Inorder and Preorder traversals.

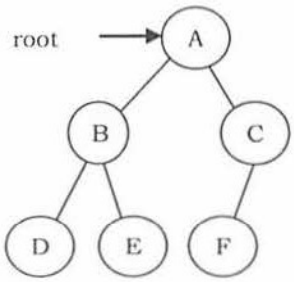
Solution: Let us consider the traversals below:

```
Inorder sequence:  D B E A F C
Preorder sequence: A B D E C F
```



In a Preorder sequence, leftmost element denotes the root of the tree. So we know 'A' is the root for given sequences. By searching 'A' in Inorder sequence we can find out all elements on the left side of 'A', which come under the left subtree, and elements on the right side of 'A', which come under the right subtree. So we get the structure as seen below.

We recursively follow the above steps and get the following tree.



Algorithm: BuildTree()

1 Select an element from *Preorder*. Increment a *Preorder* index variable (*preOrderIndex* in code below) to pick next element in next recursive call.
2 Create a new tree node (*newNode*) from heap with the data as selected element.
3 Find the selected element's index in Inorder. Let the index be *inOrderIndex*.
4 Call BuildBinaryTree for elements before *inOrderIndex* and make the built tree as left subtree of *newNode*.
5 Call BuildBinaryTree for elements after *inOrderIndex* and make the built tree as right subtree of *newNode*.
6 return *newNode*.

```
class TreeNode:
    def __init__(self, data):
        self.val = data
        self.left = None
        self.right = None

class Solution:
    def buildTree(self, preorder, inorder):
        if not inorder:
            return None # inorder is empty
            root = TreeNode(preorder[0])
            rootPos = inorder.index(preorder[0])
            root.left = self.buildTree(preorder[1 : 1 + rootPos], inorder[ : rootPos])
            root.right = self.buildTree(preorder[rootPos + 1 : ], inorder[rootPos + 1 : ])
        return root

# Alternative coding
class Solution2:
    def buildTree(self, preorder, inorder):
        return self.buildTreeRec(preorder, inorder, 0, 0, len(preorder))

    def buildTreeRec(self, preorder, inorder, indPre, indIn, element):
        if element==0:
            return None
        solution = TreeNode(preorder[indPre])
        numElementsLeftSubtree = 0;
        for i in range(indIn, indIn+element):
            if inorder[i] == preorder[indPre]:
```

```
            break
        numElementsLeftSubtree += 1
    solution.left = self.buildTreeRec(preorder, inorder, indPre+1, indIn, numElementsLeftSubtree)
    solution.right = self.buildTreeRec(preorder, inorder, indPre+numElementsLeftSubtree+1, \
        indIn+numElementsLeftSubtree+1, element-1-numElementsLeftSubtree)
    return solution
```

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

Problem-30 If we are given two traversal sequences, can we construct the binary tree uniquely?

Solution: It depends on what traversals are given. If one of the traversal methods is *Inorder* then the tree can be constructed uniquely, otherwise not.

Therefore, the following combinations can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

The following combinations do not uniquely identify a tree.

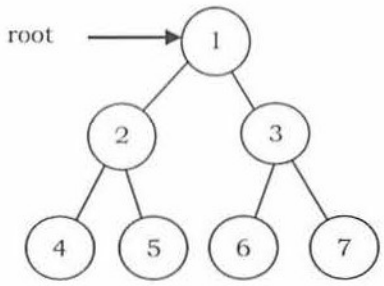
- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

For example, Preorder, Level-order and Postorder traversals are the same for the above trees:

Preorder Traversal = AB Postorder Traversal = BA Level-order Traversal = AB

So, even if three of them (PreOrder, Level-Order and PostOrder) are given, the tree cannot be constructed uniquely.

Problem-31 Give an algorithm for printing all the ancestors of a node in a Binary tree. For the tree below, for 7 the ancestors are 1 3 7.

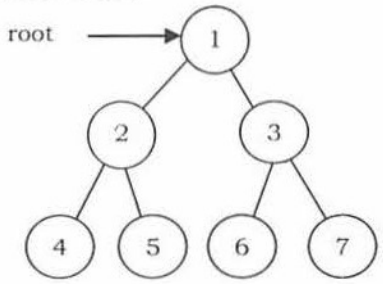


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
def PrintAllAncestors(root, node):
    if(root == NULL):
            return 0
    if(root.left == node or root.right == node or  PrintAllAncestors(root.left, node) or \
                                        PrintAllAncestors(root.right, node)):
            print(root.data)
            return 1
    return 0
```

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

Problem-32 **Zigzag Tree Traversal:** Give an algorithm to traverse a binary tree in Zigzag order. For example, the output for the tree below should be: 1 3 2 4 5 6 7



Solution: This problem can be solved easily using two stacks. Assume the two stacks are: *currentLevel* and *nextLevel*. We would also need a variable to keep track of the current level order (whether it is left to right or right to left).

We pop from *currentLevel* stack and print the node's value. Whenever the current level order is from left to right, push the node's left child, then its right child, to stack *nextLevel*. Since a stack is a Last In First Out (*LIFO*) structure, the next time that nodes are popped off nextLevel, it will be in the reverse order.

On the other hand, when the current level order is from right to left, we would push the node's right child first, then its left child. Finally, don't forget to swap those two stacks at the end of each level (*i.e.*, when *currentLevel* is empty).

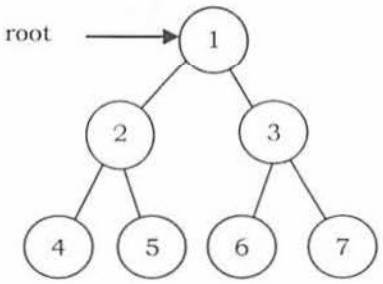
```
def zigzagTraversal(self, root):
    result = []
    currentLevel =[]
    if root != None:
        currentLevel.append(root)
    leftToRight = True
    while len(currentLevel)>0:
        levelresult = []
        nextLevel = []
        while len(currentLevel)>0:
            node = currentLevel.pop()
            levelresult.append(node.val)
            if leftToRight:
                if node.left != None:
                    nextLevel.append(node.left)
                if node.right != None:
                    nextLevel.append(node.right)
            else:
                if node.right != None:
                    nextLevel.append(node.right)
                if node.left != None:
                    nextLevel.append(node.left)
        currentLevel = nextLevel
        result.append(levelresult)
        leftToRight = not leftToRight
    return result
```

Time Complexity: $O(n)$. Space Complexity: Space for two stacks = $O(n) + O(n) = O(n)$.

Problem-33 Give an algorithm for finding the vertical sum of a binary tree. For example,
The tree has 5 vertical lines

Vertical-1: nodes-4 => vertical sum is 4
Vertical-2: nodes-2 => vertical sum is 2
Vertical-3: nodes-1,5,6 => vertical sum is 1 + 5 + 6 = 12
Vertical-4: nodes-3 => vertical sum is 3

Vertical-5: nodes-7 => vertical sum is 7
We need to output: 4 2 12 3 7

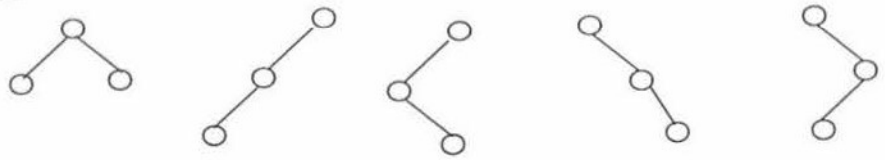


Solution: We can do an inorder traversal and hash the column. We call VerticalSumInBinaryTree(root, 0) which means the root is at column 0. While doing the traversal, hash the column and increase its value by *root → data*.

```
hashTable = {}
def verticalSumInBinaryTree(root, column):
    if not root:
            return
    if not column in hashTable:
            hashTable[column] = 0
    hashTable[column] = hashTable[column] + root.data
    verticalSumInBinaryTree(root.left, column - 1)
    verticalSumInBinaryTree(root.right, column + 1)

verticalSumInBinaryTree(root, 0)
print hashTable
```

Problem-34 How many different binary trees are possible with n nodes?

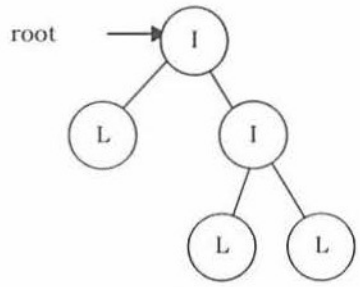
Solution: For example, consider a tree with 3 nodes ($n = 3$). It will have the maximum combination of 5 different (i.e., $2^3 - 3 = 5$) trees.



In general, if there are n nodes, there exist $2^n - n$ different trees.

Problem-35 Given a tree with a special property where leaves are represented with 'L' and internal node with 'I'. Also, assume that each node has either 0 or 2 children. Given preorder traversal of this tree, construct the tree.

Example: Given preorder string => ILILL



Solution: First, we should see how preorder traversal is arranged. Pre-order traversal means first put root node, then pre-order traversal of left subtree and then pre-order traversal of right subtree. In a normal scenario, it's not possible to detect where left subtree ends and right subtree starts using only pre-order traversal. Since every node has either 2 children or no child, we can surely say that if a node exists then its sibling also exists. So every time when we are computing a subtree, we need to compute its sibling subtree as well.

Secondly, whenever we get 'L' in the input string, that is a leaf and we can stop for a particular subtree at that point. After this 'L' node (left child of its parent 'L'), its sibling starts. If 'L' node is right child of its parent, then we need to go up in the hierarchy to find the next subtree to compute.

Keeping the above invariant in mind, we can easily determine when a subtree ends and the next one starts. It means that we can give any start node to our method and it can easily complete the subtree it generates going outside of its nodes. We just need to take care of passing the correct start nodes to different sub-trees.

```
i = 0
def buildTreeFromPreOrder(A):
    global i
    if(A == None or i >= len(A)):                # Boundary Condition
            return None
    newNode = BinaryTree(A[i])
    newNode.data = A[i]
    newNode.left = newNode.right = None
    if(A[i] == "L"):                             # On reaching leaf node, return
            return newNode
    i += 1                                       # Populate left sub tree
    newNode.left = buildTreeFromPreOrder(A)
    i += 1                                       # Populate right sub tree
    newNode.right = buildTreeFromPreOrder(A)
    return newNode
root = buildTreeFromPreOrder(["I","I","L","I","L","L","I","L","L"])
postorderRecursive(root)
```

Time Complexity: $O(n)$.

Problem-36 Given a binary tree with three pointers (left, right and nextSibling), give an algorithm for filling the *nextSibling* pointers assuming they are NULL initially.

Solution: We can use simple queue (similar to the solution of Problem-11). Let us assume that the structure of binary tree is:

```
def fillNextSiblingsWithLevelOrderTraversal(root):
    if root is None:
            return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()                   # dequeue FIFO
            node.nextSibling = q.queueFront()
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
```

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

Problem-37 Is there any other way of solving Problem-36?

Solution: The trick is to re-use the populated *nextSibling* pointers. As mentioned earlier, we just need one more step for it to work. Before we pass the *left* and *right* to the recursion function itself, we connect the right child's *nextSibling* to the current node's nextSibling left child. In order for this to work, the current node *nextSibling* pointer must be populated, which is true in this case.

```
def fillNextSiblings(root):
    if (root == None):
        return
    if root.left:
```

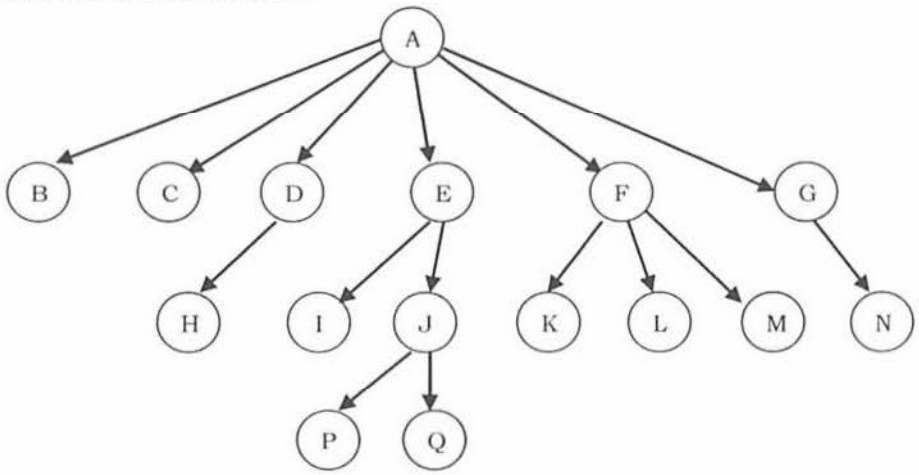
```
        root.left.nextSibling = root.right
    if root.right:
        if root.nextSibling:
            root.right.nextSibling = root.nextSibling.left
        else:
            root.right.nextSibling = None
    fillNextSiblings(root.left)
    fillNextSiblings(root.right)
```

Time Complexity: $O(n)$.

6.7 Generic Trees (N-ary Trees)

In the previous section we discussed binary trees where each node can have a maximum of two children and these are represented easily with two pointers. But suppose if we have a tree with many children at every node and also if we do not know how many children a node can have, how do we represent them?

For example, consider the tree shown below.



How do we represent the tree?

In the above tree, there are nodes with 6 children, with 3 children, with 2 children, with 1 child, and with zero children (leaves). To present this tree we have to consider the worst case (6 children) and allocate that many child pointers for each node. Based on this, the node representation can be given as:

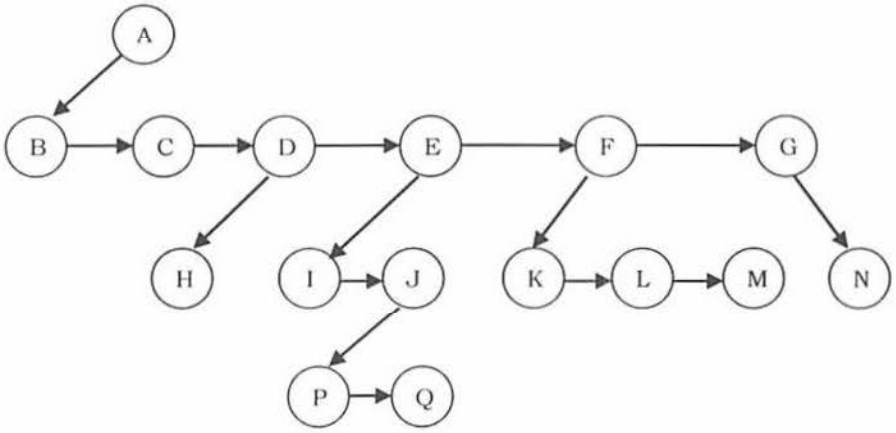
```
#Node of a Generic Tree
class TreeNode:
 #constructor
 def __init__(self, data=None, next=None):
        self.data = data
        self.firstChild = None
        self.secondChild = None
        self.thirdChild = None
        self.fourthChild = None
        self.fifthChild = None
        self.sixthChild = None
```

Since we are not using all the pointers in all the cases, there is a lot of memory wastage. Another problem is that we do not know the number of children for each node in advance. In order to solve this problem we need a representation that minimizes the wastage and also accepts nodes with any number of children.

Representation of Generic Trees

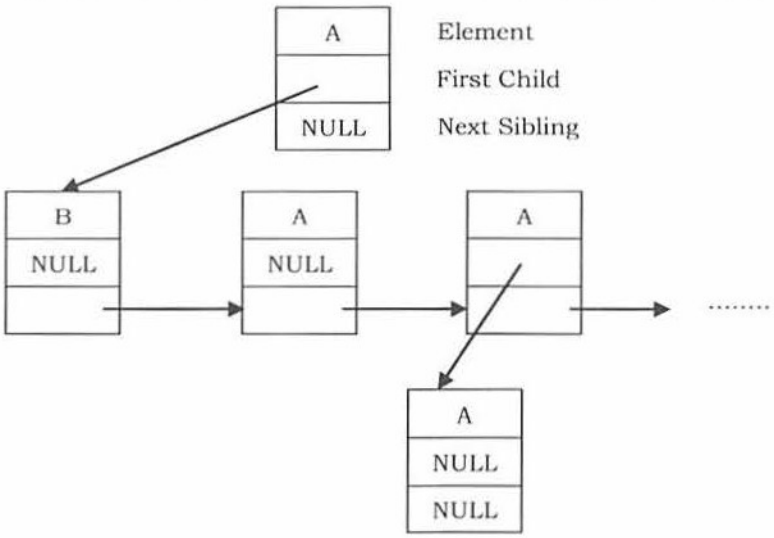
Since our objective is to reach all nodes of the tree, a possible solution to this is as follows:

- At each node link children of same parent (siblings) from left to right.
- Remove the links from parent to all children except the first child.



What these above statements say is if we have a link between children then we do not need extra links from parent to all children. This is because we can traverse all the elements by starting at the first child of the parent. So if we have a link between parent and first child and also links between all children of same parent then it solves our problem.

This representation is sometimes called first child/next sibling representation. First child/next sibling representation of the generic tree is shown above. The actual representation for this tree is:



Based on this discussion, the tree node declaration for general tree can be given as:

```
#Node of a Generic Tree
class TreeNode:
    #constructor
    def __init__(self, data=None, next=None):
        self.data = data
        self.firstChild = None
        self.nextSibling = None
```

Generic Trees: Problems & Solutions

Problem-38 Implement simple generic tree which allows us to add children and also prints the path from root to leaves (nodes without children) for every node.

Solution:

```
import string
class GenericTree:
```

```
    """ Generic n-ary tree node object
        Children are additive; no provision for deleting them.
        The birth order of children is recorded: 0 for the first
        child added, 1 for the second, and so on.
            GenericTree(parent, value=None)   Constructor
            parent          If this is the root node, None, otherwise the parent's GenericTree object.
            childList       List of children, zero or more GenericTree objects.
            value           Value passed to constructor; can be any type.
            birthOrder      If this is the root node, 0, otherwise the index of this child in the parent's .childList
            nChildren()     Returns the number of self's children.
            nthChild(n)     Returns the nth child; raises IndexError if n is not a valid child number.
            fullPath():     Returns path to self as a list of child numbers.
            nodeId():       Returns path to self as a NodeId.
    """
    def __init__ ( self, parent, value=None ):
        self.parent    = parent
        self.value     = value
        self.childList = []
        if parent is None:
            self.birthOrder = 0
        else:
            self.birthOrder = len(parent.childList)
            parent.childList.append ( self )
    def nChildren ( self ):
        return len(self.childList)
    def nthChild ( self, n ):
        return self.childList[n]
    def fullPath ( self ):
        result = []
        parent = self.parent
        kid    = self
        while parent:
            result.insert ( 0, kid.birthOrder )
            parent, kid = parent.parent, parent
        return result
    def nodeId ( self ):
        fullPath = self.fullPath()
        return NodeId ( fullPath )
class NodeId:
    def __init__ ( self, path ):
        self.path = path
    def __str__ ( self ):
        L = map ( str, self.path )
        return string.join ( L, "/" )
    def find ( self, node ):
        return self.__reFind ( node, 0 )
    def __reFind ( self, node, i ):
        if i >= len(self.path):
            return node.value      # We're there!
        else:
            childNo = self.path[i]
        try:
            child = node.nthChild ( childNo )
        except IndexError:
            return None
        return self.__reFind ( child, i+1 )
    def isOnPath ( self, node ):
        if len(nodePath) > len(self.path):
            return 0       # Node is deeper than self.path
        for i in range(len(nodePath)):
```