Unit 4

Stacks

	Syllabus	Guidelines	Suggested number of lectures
1	Array and linked representation of stacks, comparison of the operations on stacks in the two representations, implementing multiple stacks in an array *VScode	Chapter 5, Section 5.1.1-5.1.5, **Ref 2**	8
2	Applications of stacks: prefix, infix and postfix expressions, utility and conversion of these expressions from one to another	Chapter 2, Section 2.3 (upto page no. 106) **Additional resource 4**	
3	Applications of stacks to recursion: developing recursive solutions to simple problems, advantages and limitations of recursion	Chapter 5, Section 5.1 - 5.7 (snowflake example in section 5.5 not to be discussed), 6.4.2(non-recursive Depth first traversal) **ref 1**	

References

1. Ref 1: . Drozdek, A., (2012), *Data Structures and algorithm in C++*. 3rd edition. Cengage Learning. Note: 4th edition is available. Ebook is 4th edition

2. Ref 2.: Goodrich, M., Tamassia, R., & Mount, D., (2011). *Data Structures and Algorithms Analysis in C+ +*. 2nd edition. Wiley.

3. Additional Resource 3: Sahni, S. (2011). Data Structures, Algorithms and applications in C++. 2ndEdition, Universities Press

4. Additional Resource 4: Tenenbaum, A. M., Augenstein, M. J., & Langsam Y., (2009), *Data Structures Using C and C++*. 2nd edition. PHI.

Note: Ref1, Additional resource etc. as per the LOCF syllabus for the paper.

ch-4. stacks and Queues (A. Drozdek).

4.1 stacks

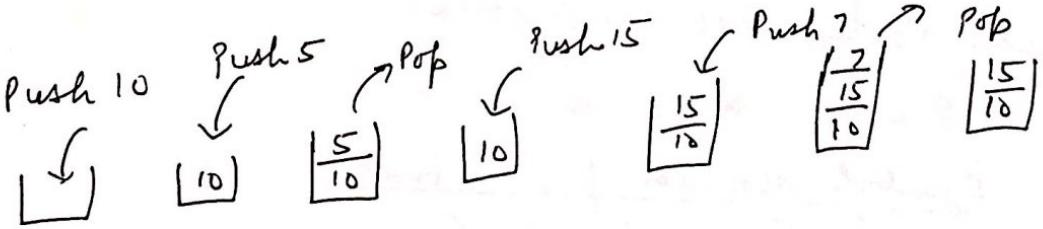
A stack is a linear data structure that can be accessed only at one of its ends for storing and retreiving data. eg:- stack of trays.

stack is a LIFO structure - Last In / first Out.

operations to be performed :-

1) clear() - clear the stack.
2) isEmpty() - check to see if stack is empty.
3) push(el) - Put the element el on top of the stack.
4) pop() - Take the topmost element from the stack.
5) topEl() - Return the topmost element from the stack without removing it.

eg:-



Applications of stacks.

1) matching delimiters :- ( ) , [ ], { }, /* */.
2) Adding very large nos.
3) Infix to Postfix notation

Stacks are the simplest of all data structures, yet they are also among the most important, since they are used in a host of different applications that include many more sophisticated data structures. Formally, a stack is an abstract data type (ADT) that supports the following operations:

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Example 5.3: *The following table shows a series of stack operations and their effects on an initially empty stack of integers.*

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

stack operations using arrays

```
#include <iostream.h>
#include <conio.h>
#include <process.h>
const int size = 30;
class stack
{
    int s[size];
    int top;
    public:
        stack () { top = -1; }
        void push (int);
        int pop ();
        void display ();
};
void stack :: push (int item)
{ if (top == size - 1)
    { cout << " stack full ";
      exit (1);
    }
  else
      s[++top] = item;
}
int stack :: pop()
{
    int item;
    if (top == -1)
      { cout << " stack empty ";
        exit (1);
      }
    else
      item = s[top--];
    return item;
}
void stack :: display ()
{ if (top == -1)
    cout << "\n nothing can be
            displayed ";
  else
  { cout << "\n elements are:\n";
    for (int i = top ; i >= 0; i--)
        cout << s[i];
  }
}
void main ()
{
}
```

eg. Add 592 + 3784 using stacks. → sum = rem + s1.top() + s2.top();

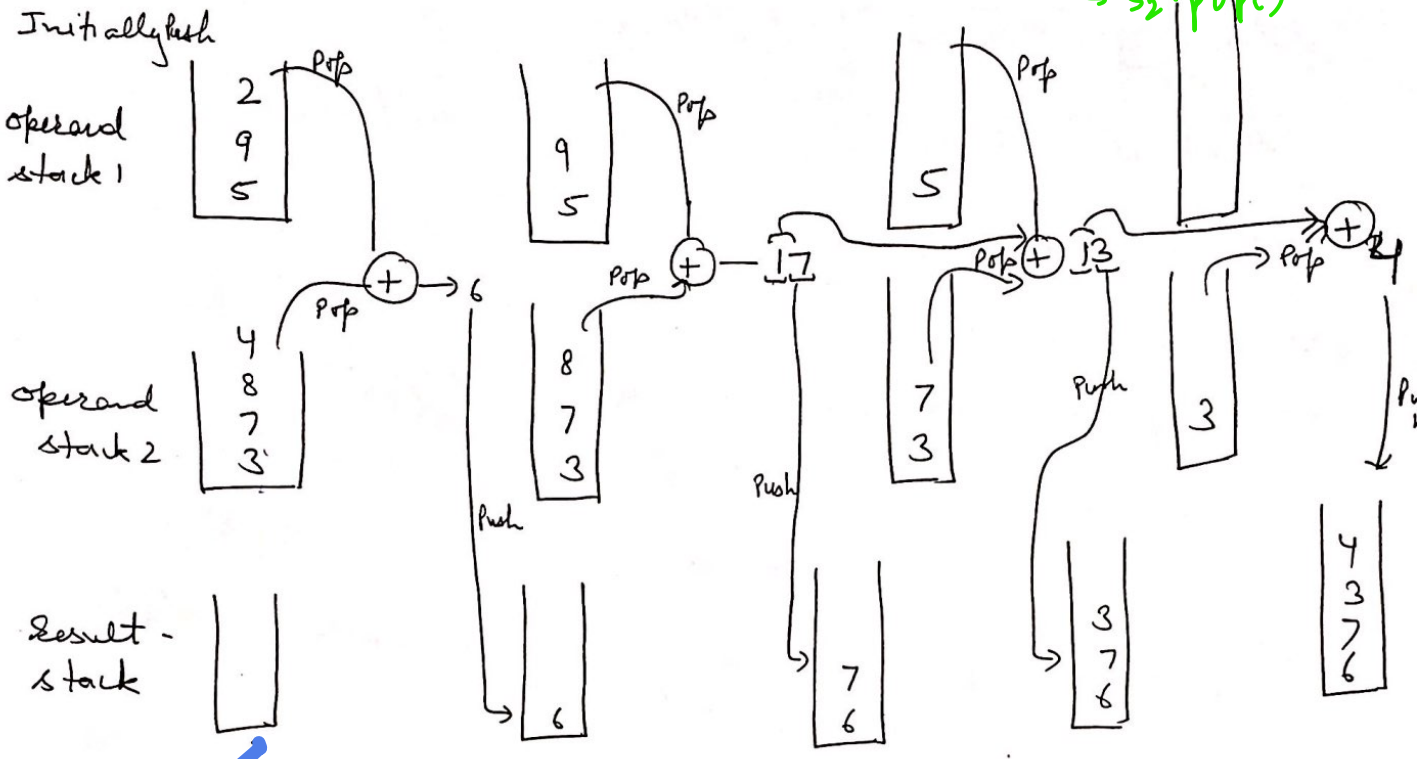
```
  592
+3784
-----
 4376
```

```
  2
+ 4
---
  6
```

```
  9
+ 8
----
 17
```

```
  1
+ 5
+ 7
----
 13
```

→ rem = sum / 10;
→ s1.pop()
→ s2.pop()



eg. Evaluate Postfix exp. using stacks.

5 6 2 + * 12 4 / -

when operand comes, pop last two numbers, perform operation and push back to the stack.

Symbol scanned	stack.
5	5
6	5 6
2	5 6 2 (Pop then +, then Push result)
+	5 8
*	40
12	40 12
4	40 12 4
/	40 3
-	37

Stacks - Infix to Postfix Notation

1. To evaluate Postfix expression using stack.
2. To add two large integers using stack.

erators +, −, *, and /. The fifth, exponentiation, is represented by the operator $. The value of the expression $A \$ B$ is A raised to the B power, so that $3 \$ 2$ is 9. For these binary operators the following is the order of precedence (highest to lowest):

Exponentiation
Multiplication/division
Addition/subtraction

When unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left. Thus $A + B + C$ means $(A + B) + C$, whereas $A \$ B \$ C$ means $A \$ (B \$ C)$. By using parentheses we can override the default precedence.

We give the following additional examples of converting from infix to postfix. Be sure that you understand each of these examples (and can do them on your own) before proceeding to the remainder of this section.

Infix	Postfix
A + B	AB +
A + B − C	AB + C −
(A + B) * (C − D)	AB + CD − *
A $ B * C − D + E / F / (G + H)	AB $ C * D − EF / GH + / +
((A + B) * C − (D − E)) $ (F + G)	AB + C * DE − − FG + $
A − B/(C * D $ E)	ABCDE $ * / −

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that the operator is placed before the operands rather than after them. We present the prefix forms of the foregoing expressions. Again, you should attempt to make the transformations on your own.

Infix	Prefix
A + B	+ AB
A + B − C	− + ABC
(A + B) * (C − D)	* + AB − CD
A $ B * C − D + E / F / (G + H)	+ − * $ ABCD / / EF + GH
((A + B) * C − (D − E)) $ (F + G)	$ − * + ABC − DE + FG
A − B/(C * D $ E)	− A / B * C $ DE

Note that the prefix form of a complex expression is not the mirror image of the postfix form, as can be seen from the second of the foregoing examples, $A + B - C$. We will henceforth consider only postfix transformations and leave to the reader as exercises most of the work involving prefix.

One point immediately obvious about the postfix form of an expression is that it requires no parentheses. Consider the two expressions $A + (B * C)$ and $(A + B) * C$. Although the parentheses in one of the two expressions is superfluous [by convention $A + B * C = A + (B * C)$], the parentheses in the second expression are necessary to avoid confusion with the first. The postfix forms of these expressions are

Evaluate the Postfix expression:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

symb	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
−	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

symb	stack
6	6
2	6, 2
3	6, 2, 3
+	6, 5
-	1
3	1, 3
8	1, 3, 8
2	1, 3, 8, 2
/	1, 3, 4
+	1, 7
*	7
2	7, 2
$	49
3	49, 3
+	52

Convert infix to Postfix

((A-(B+C))*D)$(E+F)

EXAMPLE 3

symb	*postfix string*	*opstk*
(		(
(		((
A	*A*	((
−	*A*	((−
(	*A*	((− (
B	*AB*	((− (
+	*AB*	((− (+
C	*ABC*	((− (+
)	*ABC* +	((−
)	*ABC* + −	(
*	*ABC* + −	(*
D	*ABC* + −*D*	(*
)	*ABC* + −*D* *	
$	*ABC* + −*D* *	$
(	*ABC* + −*D* *	$ (
E	*ABC* + −*D* **E*	$ (
+	*ABC* + −*D* **E*	$ (+
F	*ABC* + −*D* **EF*	$ (+
)	*ABC* + −*D* **EF* +	$
	ABC + −*D* **EF* + $	

symb | st | opstack

(| | (

(| | ((

A | A | ((

− | A | ((−

(| A | ((− (

B | AB | ((− (

+ | AB | ((− (+

C | ABC | ((− (+

) | ABC+ | ((−

) | ABC+− | (

* | ABC+− | (*

D | ABC+D | (*

) | ABC+−D* |

$ | ABC+−D* | $

(| ABC+−D* | $ (

E | ABC+−D*E | $ (

+ | | $ (+

F | ABC+−D*EF | $ (+

) | ABC+−D*EF+ | $

ABC+−D*EF+$

UNIT-4 stacks (5.1 - 5.7). [leave snowflakes in 5.5] (47)
A. Drozdek.

Ch-5 Recursion

A. Drozdek.
(5.1 - 5.4)
(w.e.f → Jan '12.)

Recursion :- A f^n calling itself.

eg:-
```
int factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Activation Record :- A Record that maintains state of each function (including main). ie contents of automatic variables, values of f^n parameters, return address indicating where to restart in the calling function. & return value for a function not declared as void.

This activation Record or stack frame is allocated on Run-time stack.

eg:- To compute x^n ie $x^n = \begin{cases} 1 & \text{if } n=0 \\ x.x^{n-1} & \text{if } n>0 \end{cases}$

```
int power(int x, int n)
{
    if (n==0)
        return 1;
    else
        return x * power(x, n-1);
}
```

Types of Recursion → Types of Direct Recursion

(1) **Tail Recursion** :- A Recursive f^n that is having only one recursive call at the very end of the function.

The recursive call is not only the last statement but there ~~is~~ ~~no~~ are no earlier recursive calls, direct or indirect.

eg:-

```
void tail (int i)
{
    if (i > 0)
    {
        cout << i;
        tail (i-1);
    }
}
```

i = 4

4, 3, 2, 1

```
void iter_tail (int i)
{
    for ( ; i>0 ; i--)
        cout << i;
}
```

i = 4

4, 3, 2, 1

5 9 10

(2) Non-tail Recursion :- Another stat. after Recursive Call

→ Type of Direct Recursion

```
Void nonTail(int i)
{
    if (i>0)
    {
        nonTail(i-1);
        cout<<i;
        nonTail(i-1);
    }
}
```

i=4

```
f1()
{
    f1();
    // another stat. after the call
    f1();
}
```

1, 2, 1, 3, 2, 1, 4, 3, 2, 1

eg:- Printing an input line in Reverse order. (Recursive Implementation)

```
Void Reverse()
{
    char ch;
    cin.get(ch);
    if (ch!='\n')
    {
        reverse();
        cout.put(ch);
    }
}
```

eg:- Printing an I/P line in Reverse order (Iterative Implementation)

```
void non_Rec_Reverse()
{
    char ch;
    cin.get(ch);
    while (ch!='\n')
    {
        s.push(ch);
        cin.get(ch);
    }
    while (!s.empty())
        cout.put(s.Pop());
}
```

Application - Von-Koch snowflake.

3) Indirect Recursion :- If a function call itself indirectly via a chain of other calls. f() not calls f()

eg:- f() call g() & g() calls f().

The chain of intermediate calls can be of any length, as

$$f() \to f1() \to f2() \to f3() \to \dots \to fn() \to f().$$

4) Nested Recursion :- A function is not only defined in terms of itself, but also is used as one of the parameters.

eg:-

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(2n)) & \text{if } n \le 4 \end{cases}$$

eg:- Ackermann f^n

$$A(n, m) = \begin{cases} m + 1 & \text{if } n = 0 \\ A(n-1, 1) & \text{if } n > 0,\ m = 0 \\ A(n-1, A(n, m-1)) & \text{otherwise} \end{cases}$$

Eqs. of Recursion

(1) To calculate factors of no. (using Tail Recursion)

Recursive # initialize i=1

```
void factor(int num, int i)
{ If (num < i)
        return;
  else
    { If (num % i == 0)
        cout << i;
    }
  factor(num, ++i);
}
```

Iterative

```
void factor(int num)
{
  for(int i=1; i<=num; i++)
   { If (num % i == 0)
        cout << i;
   }
}
```

(2) To calculate factorial of a no.

Recursive

```
int fact(int num)
{ if (num == 1)
        return 1;
  else
      return (num * fact(num-1));
}
```

Iterative

```
void fact(int num)
{ int ~~fact~~ prod = 1;
  while(num != 0)
  { Prod = Prod * num;
    -- num;
  }
  cout << Prod;
}
```

(3) To generate fibonacci series

iterative

```
void fib()
{
    a=0; b=1;
    cout<<a<<b;
    for (i=2; i<n; i++)
        { c = a+b;
          cout<<c;
          a = b;
          b = c;
        }
}
```

Recursive

```
void fibonacci()
{
    for (i=1; i<=n ; i++)
      { fib = series (i);
        cout << fib;
      }
}
int series (int n)
{
    if (n == 1)
          return 0;
    else if (n == 2)
          return 1;
    else
        return (series(n-1)+
                series (n-2));
```

O/P :- If n = 5 then fib. series is :-

0 1 1 2 3.

Q. To find maximum of a Array using Recursion.

(4) To find GCD of two nos.

Recursive.

```
void main()
{
    cout << "Enter nos:";
    cin >> a >> b;
    if (a > b)
            g = gcd(a,b);
    else
            g = gcd(b,a);
    cout << "GCD is:" << g;
}

int gcd(int a,int b)
{
    int c,g;
    c = a % b;
    if (c == 0)
            return b;
    else
      {
          a = b;
          b = c;
          g = gcd(a,b);
      }
    return g;
}
```

Backtracking

Problem is to find a valid path that leads to success. After trying one path unsuccessfully, we return and try to find another path that leads to solution. This technique is called backtracking. (eg - Sudoku)

Application:- Eight Queen's Problem.

The problem attempts to place 8 Queens on a 8 × 8 chessboard in such a way that no Queen is attacking any other. (vertically, horizontally or diagonally).

Soln.