DSA(Lecture#3)

Prepared By Bal Krishna Subedi

The stack

a. concept and definition
   - primitive operations
   - Stack as an ADT
   - Implementing PUSH and POP operation
   - Testing for overflow and underflow conditions

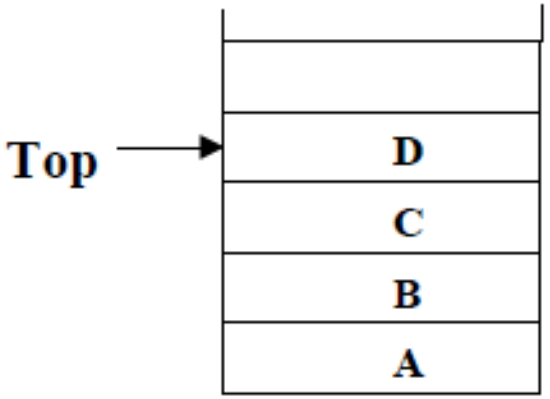
b. The infix, postfix and prefix
   - Concept and definition
   - Evaluating the postfix operation
   - Converting from infix to postfix

c. Recursion
   - Concept and definition
   - Implementation of:
     - ✓ Multiplication of natural numbers
     - ✓ Factorial
     - ✓ Fibonacci sequences
     - ✓ The tower of Hanoi

Introduction to Stack

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the* top *of the stack. The deletion and insertion in a stack is done from top of the stack.

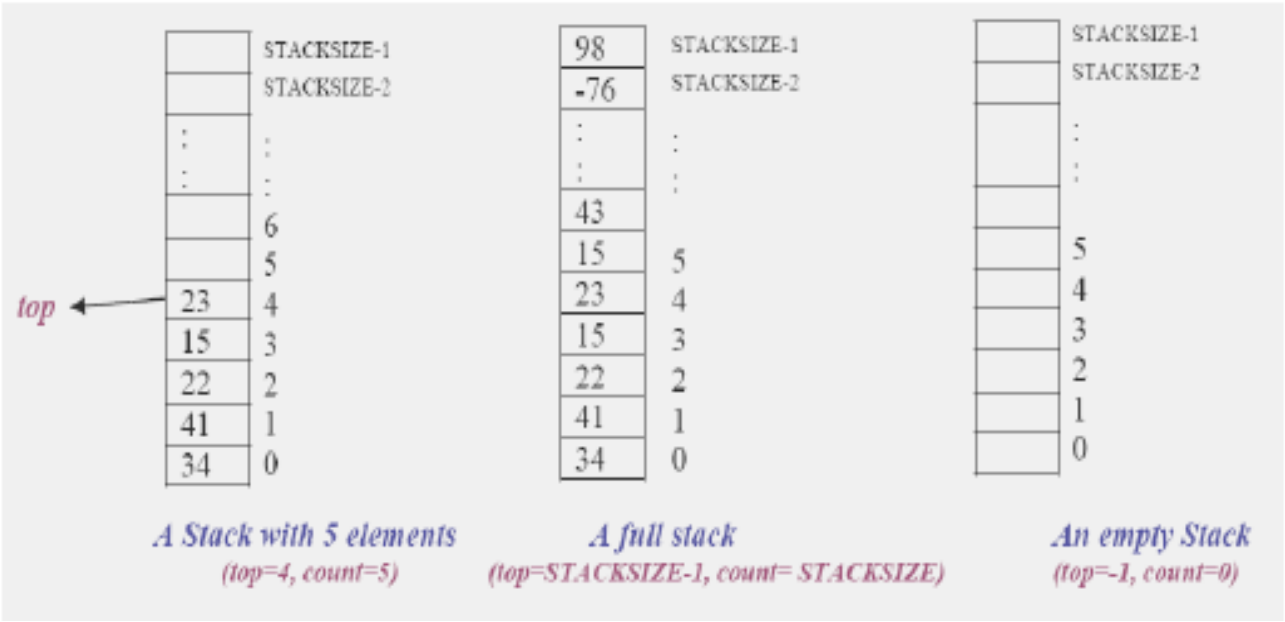
The following fig shows the stack containing items:



(Fig: A stack containing elements or items)

Intuitively, a stack is like a pile of plates where we can only (conveniently) remove a plate from the top and can only add a new plate on the top.

In computer science we commonly place numbers on a stack, or perhaps place records on the stack



Applications of Stack:

Stack is used directly and indirectly in the following fields:

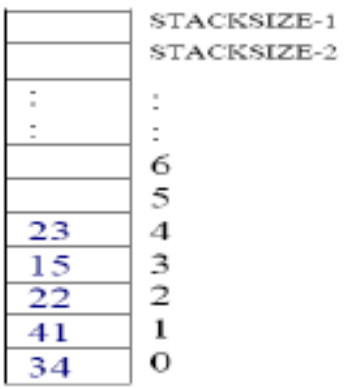
- To *evaluate the expressions (postfix, prefix)*
- To keep the p*age-visited history in a Web browser*
- To perform the u*ndo sequence in a text editor*
- Used in *recursion*
- To pass the parameters between the f*unctions in a C program*
- Can be used as an a*uxiliary data structure for implementing algorithms*
- Can be used as a c*omponent of other data structures*

Stack Operations:

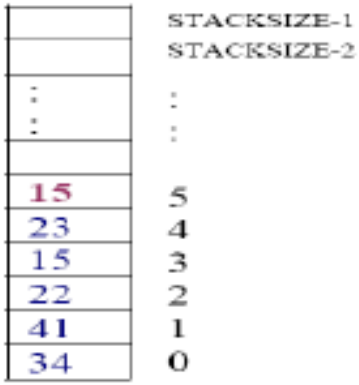
The following operations can be performed on a stack:

❖ ***PUSH*** operation: The push operation is used to add (or push or insert) elements in a stack

- When we add an item to a stack, we say that we ***push*** it onto the stack
- The last item put into the stack is at the top



Before PUSH
(top=4, count=5)

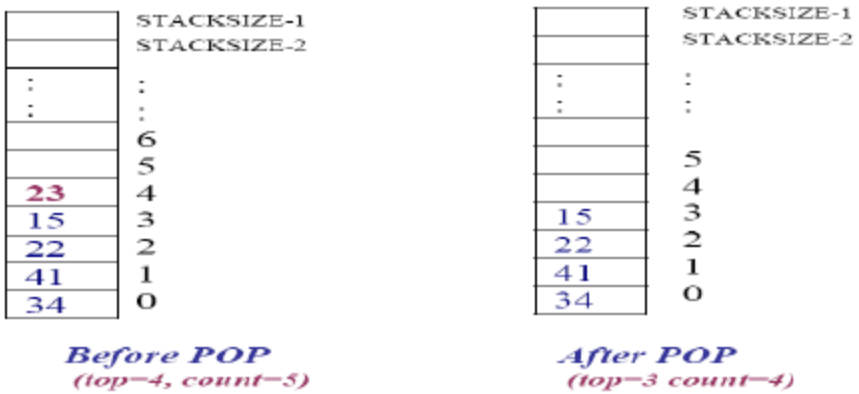


After PUSH
(top=5, count= 6)

❖ ***POP*** operation: The pop operation is used to remove or delete the top element from the stack.

- When we remove an item, we say that we ***pop*** it from the stack

- When an item popped, it is always the top item which is removed



The **PUSH** and the **POP** operations are the ***basic or primitive*** operations on a stack. Some others operations are:

- ***CreateEmptyStack*** operation: This operation is used to create an empty stack.
- ***IsFull*** operation: The isfull operation is used to check whether the stack is full or not (i.e. stack overflow)
- ***IsEmpty*** operation: The isempty operation is used to check whether the stack is empty or not. (i. e. stack underflow)
- ***Top operations:*** This operation returns the current item at the top of the stack, it doesn't remove it

The Stack ADT:

A **stack** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations

- **CreateEmptyStack(S):** Create or make stack ***S*** be an empty stack
- **Push(S, x):** Insert ***x*** at one end of the stack, called its **top**
- **Top(S):** If stack ***S*** is not empty; then retrieve the element at its **top**
- **Pop(S):** If stack ***S*** is not empty; then delete the element at its **top**
- **IsFull(S):** Determine if ***S*** is full or not. Return **true** if ***S*** is full stack; return **false** otherwise
- **IsEmpty(S):** Determine if ***S*** is empty or not. Return **true** if ***S*** is an empty stack; return **false** otherwise.

Implementation of Stack:

Stack can be implemented in two ways:

1. Array Implementation of stack (or static implementation)
2. Linked list implementation of stack (or dynamic)

Array (static) implementation of a stack:

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1(top=-1).

```
#define MAX 10
sruct stack
{
        int items[MAX];  //Declaring an array to store items
        int top;        //Top of a stack
};
typedef struct stack st;
```

Creating Empty stack :

The value of top=-1 indicates the empty stack in C implementation.

```
/*Function to create an empty stack*/
    void create_empty_stack(st *s)
    {
        s->top=-1;
    }
```

Stack Empty or Underflow:

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int isempty(st *s)
{
    if(s->top==-1)
        return 1;
    else
        return 0;
}
```

Stack Full or Overflow:

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAXSIZE-1) of the stack. The following function returns true (1) if stack is full false (0) otherwise.

```
int isfull(st *s)
{
      if(s->top==MAX-1)
            return 1;
      else
            return 0;
}
```

Algorithm for PUSH and POP operations on Stack

Let Stack[MAXSIZE] be an array to implement the stack. The variable top denotes the top of the stack.

i) Algorithm for PUSH (inserting an item into the stack) operation:

This algorithm adds or inserts an item at the top of the stack

1. *[Check for stack overflow?]*
   if top=MAXSIZE-1 then
   print "Stack Overflow" and Exit
   else
   Set top=top+1 [Increase top by 1]

Set Stack[top]:= item [Inserts item in new top position]

2. *Exit*

ii) Algorithm for POP (removing an item from the stack) operation

This algorithm deletes the top element of the stack and assign it to a variable *item*

```
1.[Check for the stack Underflow]
  If top<0 then
        Print "Stack Underflow" and Exit
  else
        [Remove the top element]
        Set item=Stack [top]
        [Decrement top by 1]
        Set top=top-1
        Return the deleted item from the stack
2. Exit
```

The PUSH and POP functions

The C function for **push** operation

```
void push(st *s, int element)
{
      if(isfull(s))      /* Checking Overflow condition */
            printf("\n \n The stack is overflow: Stack Full!!\n");
      else
s->items[++(s->top)]=element;  /* First increase top by 1 and store element at top position*/
}
```

OR

Alternatively we can define the push function as give below:

```
void push()
{
     int item;
     if(top == MAXSIZE - 1)    //Checking stack overflow
          printf("\n The Stack Is Full");
     else
     {
          printf("Enter the element to be inserted");
          scanf("%d",&item); //reading an item
          top= top+1;              //increase top by 1
          stack[top] = item;     //storing the item at the top of
                                        the stack
     }
}
```

The C function for POP operation

```
void pop(stack *s)
{
    if(isempty(s))
        printf("\n\nstack Underflow: Empty Stack!!!");
    else
       printf("\nthe deleted item is %d:\t",s->items[s->top--]);/*deletes top element and
       decrease top by 1 */
}
```

OR

Alternatively we can define the push function as give below:

```
void pop()
{
        int item;
        if(top <01)   //Checking Stack Underflow
                printf("The stack is Empty");
        else
        {
                item = stack[top];  //Storing top element to item variable
                top = top-1;       //Decrease top by 1
                printf("The popped item is=%d",item); //Displaying the deleted item
        }
}
```

Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix
- Prefix
- Postfix

- **Infix expression**: It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: A+B. Here '+' is an ***operator*** and ***A*** and ***B*** are called ***operands***
- **Prefix notation:** In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. Example: +AB
- **Postfix notation**: In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands. Example: AB+

Examples:

A + B (Infix)
+ AB (Prefix)
AB + (Postfix)

- Both prefix and postfix are parenthesis free expressions. For example

(A + B) * C Infix form
* + A B C Prefix form
A B + C * Postfix form

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

Converting an Infix Expression to Postfix

First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.

- Remember, to convert an infix expression to its postfix equivalent, we first convert the innermost parenthesis to postfix, resulting as a new operand
- In this fashion parenthesis can be successively eliminated until the entire expression is converted
- The last pair of parenthesis to be opened within a group of parenthesis encloses the first expression within the group to be transformed
- This last in, first-out behavior suggests the use of a stack

Precedence rule:

While converting infix to postfix you have to consider the **precedence rule**, and the precedence rules are as follows

1. Exponentiation (the expression A$B is A raised to the B power, so that 3$2=9)
2. Multiplication/Division
3. Addition/Subtraction

When un-parenthesized 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.

- A+B+C means (A+B)+C
- ABC means A$(B$C)

By using parenthesis we can override the default precedence.

Consider an example that illustrate the converting of infix to postfix expression, A + (B* C).

Use the following **rule** to convert it in postfix:

1. Parenthesis for emphasis
2. Convert the multiplication
3. Convert the addition
4. Post-fix form

Illustration:

A + (B * C). Infix form
A + (B * C) Parenthesis for emphasis
A + (BC*) Convert the multiplication
A (BC*) + Convert the addition
ABC*+ Post-fix form

Consider an example:

(A + B) * ((C - D) + E) / F Infix form
(AB+) * ((C – D) + E) / F
(AB+) * ((CD-) + E) / F
(AB+) * (CD-E+) / F
(AB+CD-E+*) / F
AB+CD-E+*F/ Postfix form

Examples

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 $ * / -

Algorithm to convert infix to postfix notation

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack top and poststack top respectively.

1. Scan one character at a time of an infix expression from left to right
2. opstack=the empty stack
3. Repeat till there is data in infix expression
   - 3.1 if scanned character is '(' then push it to opstack
   - 3.2 if scanned character is operand then push it to poststack
   - 3.3 if scanned character is operator then
     - if(opstack!=-1)
     - while(precedence (opstack[otos])>precedence(scan character)) then
       - pop and push it into poststack
     - otherwise
       - push into opstack
   - 3.4 if scanned character is ')' then
     - pop and push into poststack until '(' is not found and ignore both symbols
4. pop and push into poststack until opstack is not empty.
5. return

Trace of Conversion Algorithm

The following tracing of the algorithm illustrates the algorithm. Consider an infix expression

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

Scan character	Poststack	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	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
……	ABC+-D*EF+$ (postfix)	……

Converting an Infix expression to Prefix expression

The precedence rule for converting from an expression from infix to prefix are identical.

Only changes from postfix conversion is that the operator is placed before the operands rather than after them. The prefix of

A+B-C is -+ABC.

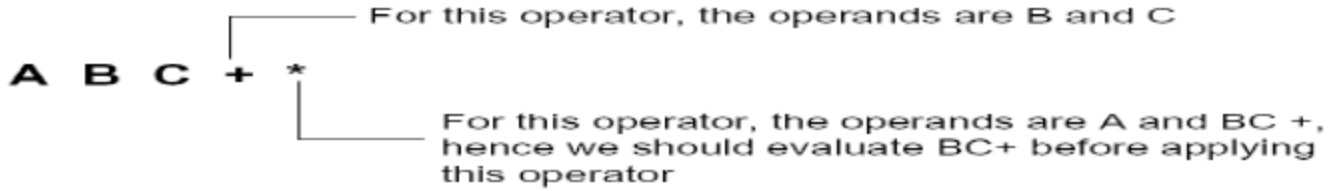
A+B-C (infix)
=(+AB)-C
=-+ABC (prefix)

Example Consider an example:

A $ B * C – D + E / F / (G + H) infix form
= A $ B * C – D + E / F /(+GH)
=$AB* C – D + E / F /(+GH)
=*$ABC-D+E/F/(+GH)
=*$ABC-D+(/EF)/(+GH)
=*$ABC-D+//EF+GH
= (-*$ABCD) + (//EF+GH)
=+-*$ABCD//EF+GH which is in prefix form.

Evaluating the Postfix expression

Each operator in a postfix expression refers to the previous two operands in the expression.



To evaluate the postfix expression we use the following procedure:

Each time we read an operand we push it onto a stack. When we reach an operator, its operand s will be the top two elements on the stack. We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

Consider an example

3 4 5 * +
=3 20 +
=23 (answer)

Evaluating the given postfix expression:

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

=6 5 - 3 8 2 / + * 2 $ 3 +

=1 3 8 2 / + * 2 $ 3 +

=1 3 4 + * 2 $ 3 +

=1 7 * 2 $ 3 +

=7 2 $ 3 +

=49 3 +

= 52

Algorithm to evaluate the postfix expression

Here we use only one stack called vstack(value stack).

1. Scan one character at a time from left to right of given postfix expression
   1.1 if scanned symbol is operand then
   read its corresponding value and push it into vstack
   1.2 if scanned symbol is operator then
   – pop and place into op2
   – op and place into op1
   – compute result according to given operator and push result into vstack
2. pop and display which is required value of the given postfix expression
3. return

Trace of Evaluation:

Consider an example to evaluate the postfix expression tracing the algorithm

ABC+*CBA-+*

123+*321-+*

Scanned character	value	Op2	Op1	Result	vstack
A	1	……	…….	…….	1
B	2	……	……	…..	1 2
C	3	……	……	…..	1 2 3
+	…….	3	2	5	1 5
*	……	5	1	5	5
C	3	…….	……	……	5 3
B	2	…	…….		5 3 2
A	1	…….	…….	……	5 3 2 1
-	…….	1	2	1	5 3 1
+	…….	1	3	4	5 4
*	…….	4	5	20	20

Its final value is 20.

Evaluating the Prefix Expression

To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms. One stack is used to store operators and another is used to store the operands. Consider an example for this

+ 5 *3 2 prefix expression

= +5 6
=11

Illustration: Evaluate the given prefix expression
/ + 5 3 – 4 2 prefix equivalent to (5+3)/(4-2) infix notation
= / 8 – 4 2
= / 8 2
= 4

Recursion:

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

Example:

```
/*calculation of the factorial of an integer number using recursive function*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int n;
        long int facto;
        long int factorial(int n);
        printf("Enter value of n:");
        scanf("%d",&n);
        facto=factorial(n);
        printf("%d! = %ld",n,facto);
        getch();
}
long int factorial(int n)
{
        if(n == 0)
                return 1;
        else
                return n * factorial(n-1);
}
```

Let's trace the evaluation of factorial(5):

Factorial(5)=
5*Factorial(4)=
5*(4*Factorial(3))=
5*(4*(3*Factorial(2)))=
5*(4*(3*(2*Factorial(1))))=
5*(4*(3*(2*(1*Factorial(0)))))=
5*(4*(3*(2*(1*1))))=
5*(4*(3*(2*1)))=
5*(4*(3*2))=
5*(4*6)=
5*24=
120

Example:

```
/*calculation of the factorial of an integer number without using recursive function*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int n;
        long int facto;
        long int factorial(int n);
        printf("Enter value of n:");
        scanf("%d",&n);
        facto=factorial(n);
        printf("%d! = %ld",n,facto);
        getch();
}
```

```
long int factorial(int n)
{
        long int facto=1;
        int i;
        if(n==0)
                return 1;
        else {
                for(i=1;i<=n;i++)
                        facto=facto*i;
                return facto;
        }
}
```

```
/* Program to find sum of first n natural numbers using recursion*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int n;
        int sum_natural(int );
        printf("n = ");
        scanf("%d",&n);
        printf("Sum of first %d natural numbers = %d",n,sum_natural(n));
        getch();
}
int sum_natural(int n)
{
        if(n == 1)
                return 1;
        else
                return n + sum_natural(n-1);
}
```

Tower of Hanoi problem:

Initial state:

- There are three poles named as origin, intermediate and destination.
- **n** number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4, ……,n.

Objective:

- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions:

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

Algorithm: - To move a tower of *n* disks from *source* to *dest* (where *n* is positive integer):

1. If *n* ===1:
    1.1. Move a single disk from *source* to *dest*.
2. If n > 1:
    2.1. Let *temp* be the remaining pole other than *source* and *dest*.
    2.2. Move a tower of (*n* – 1) disks form *source* to *temp*.
    2.3. Move a single disk from *source* to *dest*.
    2.4. Move a tower of (*n* – 1) disks form *temp* to *dest*.
3. Terminate.

Example: Recursive solution of tower of Hanoi:

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); //Function prototype
void main()
{
        int n;
        printf("Enter number of disks");
        scanf("%d",&n);
        TOH(n,'O','D','I');
        getch();
}

void TOH(int n, char A, char B, char C)
{
        if(n>0)
        {
                TOH(n-1, A, C, B);
                Printf("Move disk %d from %c to%c\n", n, A, B);
                TOH(n-1, C, B, A);
        }
}
```

Advantages of Recursion:

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as tower of Hanoi.

Disadvantages of Recursion:

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

Thanks You