MODULE 8

STACK

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING

UNIVERSITI TEKNOLOGI MALAYSIA

OBJECTIVES FOR STUDENTS

1. Understand the stack concept and its structure.
2. Understand the operations that can be done on a stack.
3. Understand and know how to implement stack using array and linked list.

KEY CONCEPT

1.0 INTRODUCTION TO STACK

1.1. **Stack definition** –

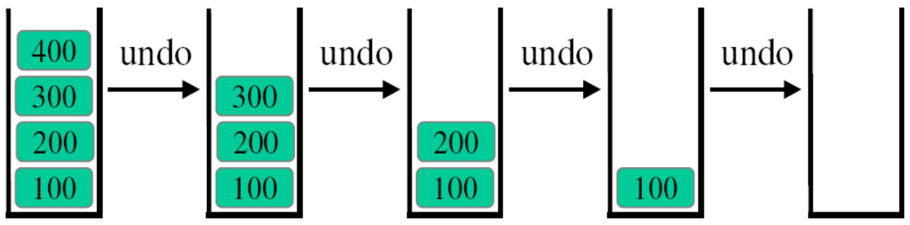
- A linear list whereby all additions and deletions are restricted at one end, called **top.**
- A stack has **LAST IN FIRST OUT** (LIFO) property.
- Items come last will be put at the top and deleted first.
- In Computer Science, stack provides techniques to facilitate the development of computer programs especially in compiler operations.

1.2. **Example of stack** :

- Stack of books or stack of plates.
- Long and narrow driveway. BMW comes in first, Lexus follows, Benz enters in last. When the cars come out, Benz comes out first, then Lexus follows, and BMW comes out last.
- Reverse a string. Example: TOP, Reverse: POT
- Brackets balancing in compiler
- Page-visited history in a Web browser
- Undo operation in many editor tools
- Check nesting structure

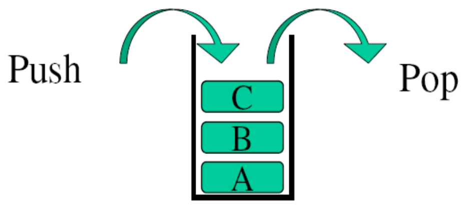
1.3. **More example of stack :**

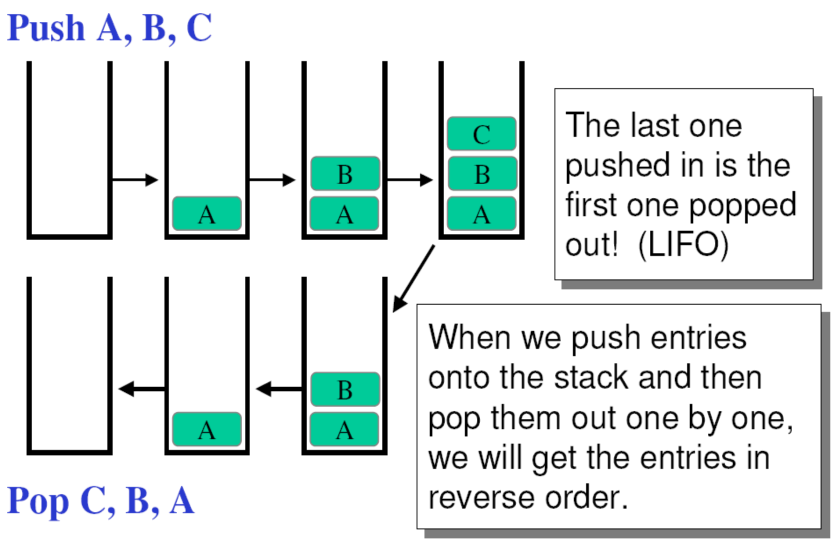
- In an Excel file, input your data in row lets say 100, 200, 300, 400
- If we find something wrong, use undo and return to the previous state



1.4. **LAST IN FIRST OUT (LIFO)**

- Adding an entry on the top is called push
- Deleting an entry from the top is called pop
- A stack is open at one end (the top) only. You can push entry onto the top, or pop the top entry out of the stack.




Push A, B, C
A
B
A
C
B
A
The last one pushed in is the first one popped out! (LIFO)
B
A
A
When we push entries onto the stack and then pop them out one by one, we will get the entries in reverse order.
Pop C, B, A

2.0 STACK IMPLEMENTATION

2.1. Stack Implementation :

- Stack is an abstract data structure
- Items in a stack can be integer, double, string, and also can be any data type, such as Employee, Student etc
- We can implement stack using array or linked list.

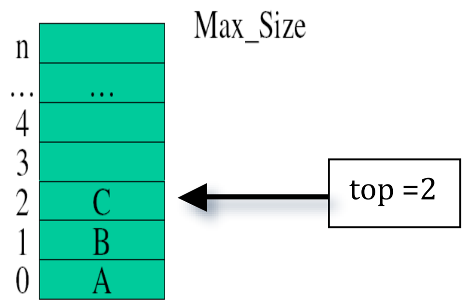
2.2. Implementation of Stack :

- **Array**
  - Size of stack is fixed during declaration
  - Item can be pushed if there is some space available, need **isFull( )** operations.
  - Need a variable called, top to keep track the top of a stack.
  - Stack is empty when the value of top is –1.
- **Linked List**
  - Size of stack implementation linked list is flexible. Item can be pushed and popped dynamically.
  - Need a pointer, called top to point to the top of stack

3.0 ARRAY IMPLEMENTATION OF STACK

3.1 Stack Declaration

- Stack can be visualized as array, BUT the operations can be done at top of the stack only



- We need two data attributes for stack:
  - **Data : array** that store item in the stack. In this example data will store **char** value
  - **top** : as index for top of stack, integer type
- Size of the array that store component of stack is 100. In this case, stack can store up to 100 **char** value.

- Stack class declaration, shown in Program 8.1

```
// Program 8.1
//Stack declaration:
const int size = 100;
class stack
{
  private:  //data declaration
    int top;
    char data[siza];
  public:  //public declaration
  void createStack();
  void push(char);  //insert operation
  void pop();       //delete operation
  char stackTop();  //get top value
  bool isFull();    //check if stack full
  bool isEmpty();   //check if stack empty
};
```

3.2 Stack Implementation

- Thera are 3 things to be considered for stack with array.
- **Stack is Empty** : when top is -1.
- **Push operations -** To insert item into stack, 2 statements must be used
  - **top = top + 1;**
  - **stack[top] = newitem;**
- **pop operations -** To delete item from stack, 2 statements should be used
  - **item = stack[top]; or stackTop();**
  - **top = top – 1;**
- **item = stack[top];** statement is needed if we want to check the value to be popped from the stack

3.3 Stack Operations

- The basic stack operations as declared in Program 8.1 are:
  - **createStack()**
  - **push(item)**
  - **pop()**
  - **isEmpty()**
  - **isFull()**
  - **stackTop()**

3.4 createStack() operation

- Stack will be created by initializing **top** to -1.

- **createStack()** implementation is shown in Program 8.2

- **top** is **–1 -** means that there is no item being pushed into stack yet.

```
// Program 8.2
void stack::createStack()
{
  top=-1;
}
```

3.5 **isFull() operation**

- This operation is needed **only** for implementation of stack using array.
- In an array, size of the array is fixed and to create new item in the array will depend on the space available.
- This operation is needed before any push operation can be implemented on a stack.
- **isFull()** implementation is as in Program 8.3

```
// Program 8.3
bool stack::isFull()
{
  return (top == size-1);
}
```

- In Program 8.3, if the size of the array is 100,
  - **bool isFull()** will return **true**, if **top** is 99 (100 – 1).
  - **bool isFull()** will return **false**, if there is some space available, **top** is less than 100.

3.6 **isEmpty() operation**

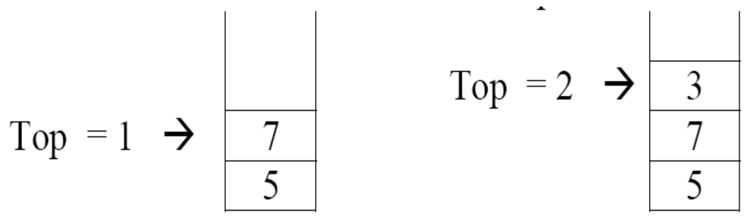
- This operation will check whether the array for stack is empty.
- This operation is needed before pop operation can be done. If the stack is empty, then pop operation cannot be implemented
- **bool isEmpty()** will return **true** if **top** is **–1** and return **false** if **top** is not equal to -1, showing that the stack has element in it.
- **isEmpty()** implementation is shown in Program 8.4

```
// Program 8.4
bool stack::isEmpty()
{
  return (top == -1);
}
```

3.7 **push() operation**

- **push()** operation will **insert an item at the top of stack**. This operation can be done only if there is space available in the array

- Before any item can be inserted into a stack, **isFull()** operation must be called first.
- Insertion operation involve the following steps:
  - **top** will be increased by 1.
    - **top = top + 1;**
  - New item will be inserted at the **top**
    - **data[top] = newItem;**



before **push()**operation

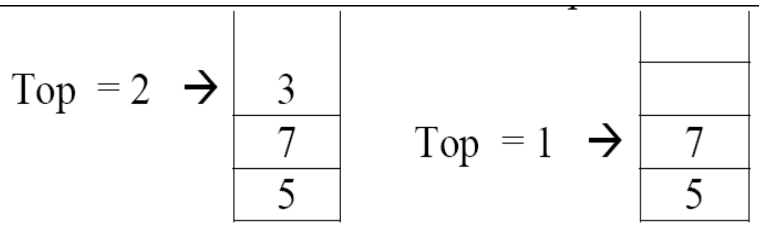
after **push()**item 3 to stack, top increase by 1

- **push()** implementation is as shown in Program 8.5
  - Top will be increased first before item is inserted and it will avoid inserting item at the current top value.

```
// Program 8.5
void stack::push(char newitem)
{
  if(isFull())  //check whether stack is full
    cout << "Sorry, Cannot push item.
           Stack is now full!"<<endl;
  else
  {
    top=top+1;  //Top point to next index
    data[top]=newitem;  //assign new item
  }//end else
}//end push()
```

3.8 pop() operation

- This operation will delete an item at top of stack.
- Function **isEmpty()** will be called first in order to ensure that there is item in a stack to be deleted. If **isEmpty()** return **False**, **pop()** can be implemented on stack
- **pop()** operation will decrease the value of **top** by 1:
  - **top = top - 1;**

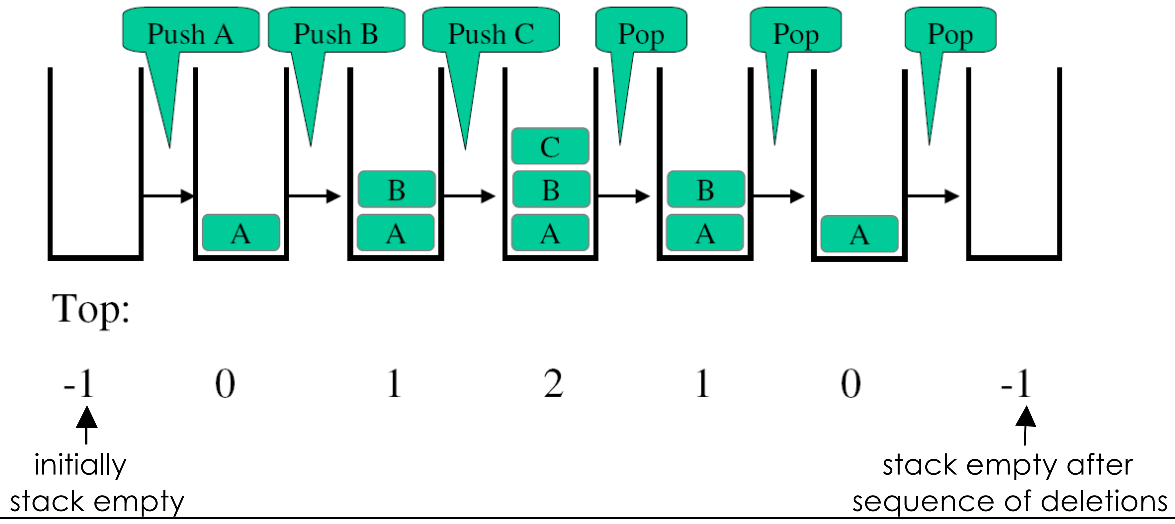


before **pop()** after **pop()**,
item 3 is taken out from stack
and top decrease

- **pop()** implementation is shown in Program 8.6

```
// Program 8.6
void stack::pop()
{
  char item;
  if(isEmpty())
    cout << "Sorry, Cannot pop item.
         Stack is empty!"<<endl;
  else
  { //display value at top to be delete
    cout<<"Popped     value:"<<data[top]<<endl;
    top=top-1;
    //top will hold to new index
  }//end if
}//end pop
```

3.9 **push()**and **pop() operations in stack implementation array**



3.10 stackTop() operation

- **stackTop()** operation is used to get value at the top of the stack
- **stackTop()** doesn't delete item. The function only retrieve item at top of stack
- **stackTop()** implementation is shown in Program 8.7

```
// Program 8.7
char stackTop()
{ //function to get top value of stack
  if(isEmpty())
    cout<<"Sorry, stack is empty!"<<endl;
  else
    return data[top];
}//end stackTop
```

4.0 LINKED LIST IMPLEMENTATION OF STACK

4.1 Introduction

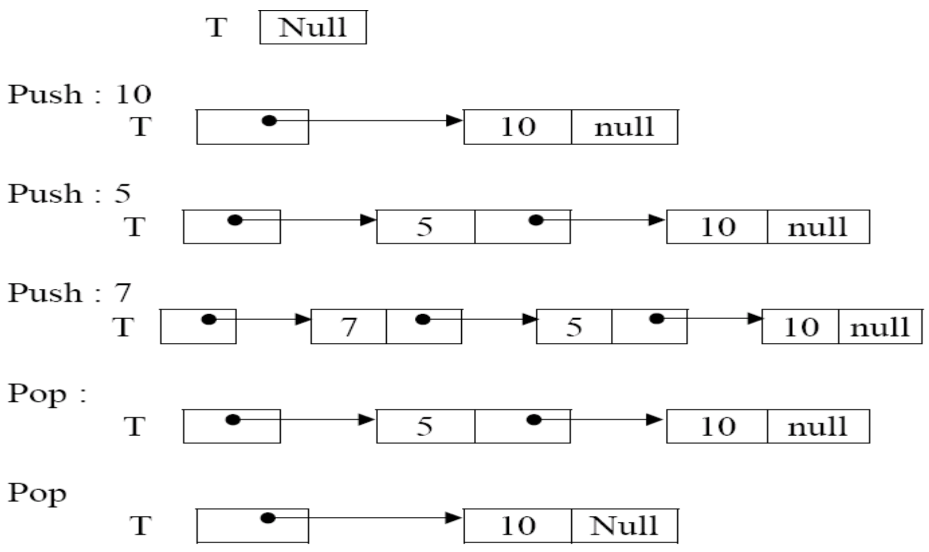
- In stack implemented using linked list, the number of elements in stack is not restricted to certain size.
- It implement dynamic memory creation, whereby memory will be assigned to stack when a new node is pushed into stack, and memory will be released when an element being popped from the stack.
- Stack using linked list implementation can be empty or contains a series of nodes.
- Each node in a stack must contain at least 2 attributes:
    - **data** – to store information in the stack.
    - **pointer next**, which store address of the next node in the stack

4.2 Basic operation of stack

- **createStack()** – to initialize **top**
- **push()** – insert item onto stack
- **pop()** – delete item from stack
- **isEmpty()** – check whether a stack is empty.
- **stackTop()** – get item at **top**
- **isFull()** operation is not needed since elements can be inserted into stack without limitation to the stack size.
- **push()** and **pop**() operations can only be done at the top ~ similar to add

and delete in front of the linked list.

4.3 push() and pop() operations in stack implementation linked list



4.4 Declaration of a linked list implementation of Stack

- Two declarations are needed
  - Declaration of class node
  - Declaration of class stack
  - Class stack has one attribute, which is **top**

- Declaration of stack implementation linked list is as shown in Program 8.8

```
// Program 8.8
class nodeStack
{
  int data;
  nodeStack *next;
};

class stack
{
  private:  //pengisytiharan ahli data
    nodeStack *top;
  public:  //pengisytiharan ahli fungsi
    void createStack(); //set Top to NULL
    void push(int);  //insert item into stack
    void pop();  //delete item from stack
    int stackTop(); //retrieve item at top stack
};
```

4.5 **createStack() operation**

- Creating a stack will initialize **top** to **NULL** – indicating that currently, there is no node in the stack.
- **createStack() i**mplementation is as shown in Program 8.9

```
// Program 8.9
void stack::createStack()
{
  top=NULL;
}
```

4.6 **isEmpty() operation**

- **isEmpty()** will return true if stack is empty, **top** is **NULL**.
- **isEmpty() i**mplementation is as shown in Program 8.10

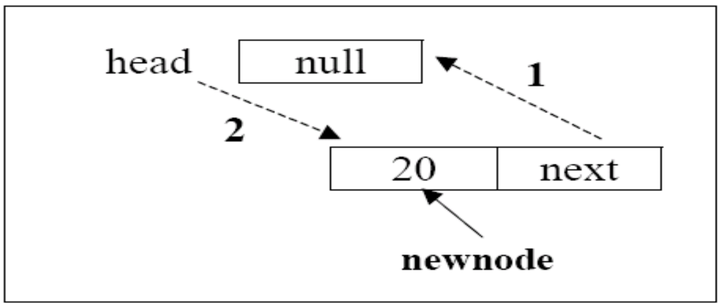
```
// Program 8.10
bool stack::isEmpty()
{
  return (top=NULL);
}
```

4.7 **push() operation**

- There are 2 conditions for inserting element in stack:
  - Insert to empty stack.
  - Insert item to non empty stack : stack with value.

- **Push to empty stack**
  - In this situation the new node being inserted, will become the first item in stack.

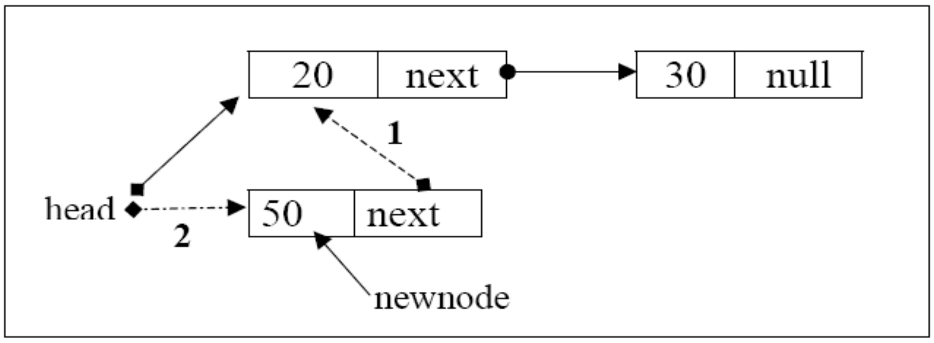
    STEP 1 : newnode-> next = head;
    STEP 2 : head = newnode;



- **Push to non empty stack**
  - This operation is similar to inserting element in front of a linked list. The

next value for the new element will point to the top of stack and head will point to the new element.

STEP 1 : newnode-> next = head;
STEP 2 : head = newnode;

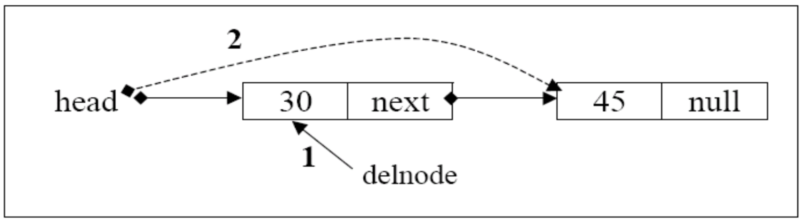


- **push()** implementation is shown in Program 8.11

```
// Program 8.11
void stack::push(int newitem)
{ //create newnode
  nodeStack *newnode;
  newnode = new (nodeStack);
  if(newnode == NULL)
    cout<<"Cannot allocate memory.."<<endl;
  else
  { newnode→data = newitem;
    newnode→next = head;
    head = newnode;
  }// end if
}//end push operation
```

4.8 pop() operation

- Pop operation can only be done to non-empty stack. Before **pop()** operation can be done, **isEempty()** operation must be called in order to check whether the stack is empty or has item in the stack. If **isEmpty()** function return **true**, **pop()** operation cannot be done.
- During **pop()** operation, an external pointer is needed to point to the delete node. In the figure below, **delnode** is the pointer variable to point to the node that is going to be deleted.
- Steps to delete a node in stack implementation linked list
  STEP 1 : delnode = head;
  STEP 2 : head = delnode -> next; or head = head->next;
  STEP 3 : delete(delnode);



- **pop()** implementation is shown in Program 8.12

```
// Program 8.12
void stack::pop()
{ nodeStack *delnode;
  if(isEmpty())
    cout << "Sorry, cannot pop item from stack.
            Stack is still empty!"<<endl;
  else
  { delnode = head;
    Cout << "Item to be popped from stack is:
           "<<stackTop()<<endl;
    head = delnode→next;
    delete(delnode);
  }// end else
}//end pop
```

4.9 **stackTop() operation**

- **stackTop()** is used to retrieve item at the top of stack
- **isEmpty()**function is needed to check whether the stack is empty. If **isEmpty()**return **True**, the stack is empty and **stackTop()** cannot be implemented.
- **stackTop()** implementation is shown in Program 8.13

```
// Program 8.13
int stack::stackTop()
{ if(isEmpty())
    cout<<"Sorry, stack is still empty!"<<endl;
  else
    return head→data;
}//end check item at top
```

5.0 STACK APPLICATION

5.1 Stack Application Examples

- Check whether parentheses are balanced (open and closed parentheses are properly paired)

- Evaluate algebraic expressions.
- Creating simple calculator
- Backtracking (example. Find the way out when lost in a place)

5.2 **Example 1 – To check for balanced parentheses**

•Stack can be used to recognize balanced parentheses in an expression.

•Examples of balanced parentheses.

- **(a+b), (a/b+c), a/((b-c)*d)**
- In the example, open and closed parentheses are properly paired.

•Examples of not balance parentheses.

- **((a+b)*2** and **m*(n+(k/2)))**
- In the example, open and closed parentheses are not properly paired.

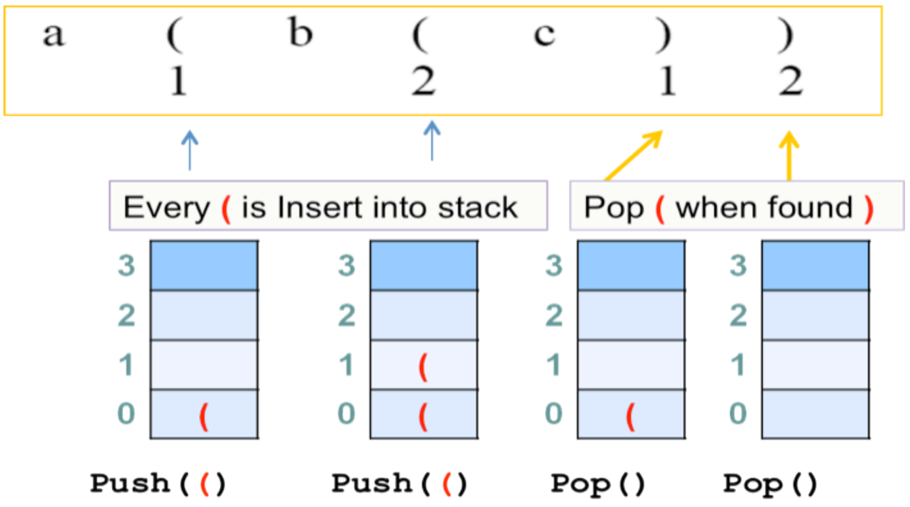
•Check for balanced parentheses algorithm is given in Algorithm 8.1 below

```
// Algorithm 8.1
create(stack);
continue = true;
while(not end of input string) && continue
{  ch = getch();
   if ch = '(' || ch = ')'
   {  if ch = '('
         Push(stack, '(');
      else if isEmpty(stack)
         continue = false;
      else
         Pop(s);
   } // end if
}// end while
if(end of input && isEmpty(stack);
  cout<<"Parentheses are Balanced.."<<endl;
else
  cout<<"Parenthees are Not Balanced.."<<endl;
// algorithm ends
```

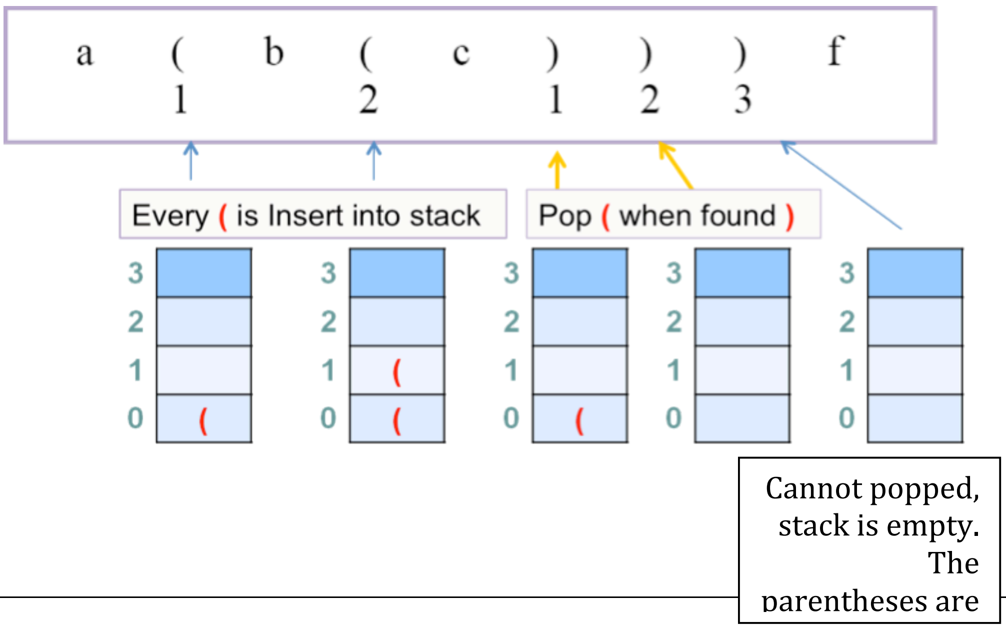
- **Criteria for checking balanced parentheses in an expression**
  - Every '**(**' read from a string will be pushed into stack.
  - The open parentheses '**(**' will be popped from a stack whenever the closed parentheses '**)**' is read from string.
  - An expression have balanced parentheses if **:**
    - Each time a "**)**" is encountered it matches a previously encountered "**(**".
    - When reaching the end of the string, every "**(**" is matched and stack is finally empty.
  - An expression does NOT have balanced parentheses if **:**

- When there is still ‘**)**’ in input string, the stack is already empty.
- When end of string is reached, there is still ‘**(**‘ in stack.

- **Example of checking for balanced parentheses**
  - Expression : **a (b (c))**
  - The expression has balance parentheses since when end of string is found the stack is empty.
  - Example of checking balanced parentheses is as in the figure below:



- **Example of checking for imbalanced parentheses**
  - Expression: **a (b (c))) f**
  - The expression does not have balance parentheses.
  - The third “)” encountered does not has its match since the stack is empty.
  - Example of checking for unbalanced parentheses is in the figure as follows:


a (b (c))) f
1 2 1 2 3
Every (is Insert into stack
Pop (when found)
3 2 1 0
Cannot popped, stack is empty. The parentheses are

5.3 Example 2 – Algebraic expression

- One of the compiler's task is to evaluate algebraic expression.
- Example of assignment statement:

  y = x + z * (w / x + z * (7 + 6))

- The compiler must determine whether the expression is syntactically legal algebraic expression before evaluation can be done on the expression.
- 3 algebraic expressions are :
  - Infix,
  - Prefix
  - Postfix

5.4 Infix Expression

- The algebraic expression commonly used is infix.
- The term infix indicates that every binary operators appears between its operands.
- Example of the syntax:

A	**+**	**B**
operand	operator	operand

- Example of infix expression:

  A + B * C
  A + (B * C)
  (a + b) * c

- To evaluate infix expression, the following rules were applied:
  - Precedence rules.
  - Association rules (associate from *left to right*)
  - Parentheses rules.

5.5 Prefix and postfix expressions

- Alternatives to infix expression
- Prefix : Operator appears before its operand.

+	**A**	**B**
operator	operand	operand

- Prefix Examples:

  *** c d**
  + a * b c

*** + a b c**

- Postfix : Operator appears after its operand.

  A **B** **+**
  operand operand operator

- Postfix Examples:

  b c *
  a b c * +
  a b + c *

- Infix, prefix and postfix examples

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + (b * c)	+ a * b c	a b c * +
(a + b) * c	* + a b c	a b + c *

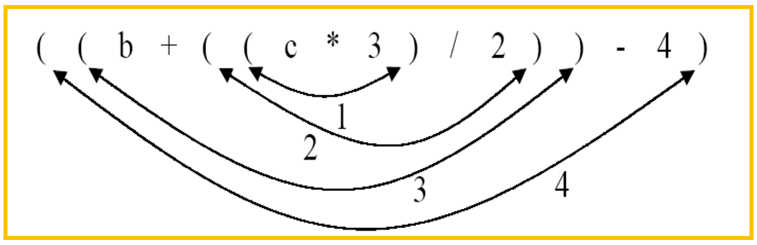
- The advantage of using prefix and postfix is that we don't need to use precedence rules, associative rules and parentheses when evaluating an expression.

5.6 Converting infix to prefix

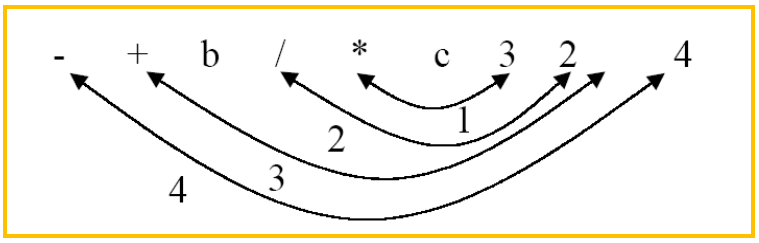
- Steps to convert infix expression to prefix:

  b + c * 3 / 2 - 4

STEP 1 : Determine the precedence. Add parentheses to the infix expression based on the precedence, associative and parentheses rules.



STEP 2: Move forward the operators to closest open parentheses at the left side of them. Then, remove the parentheses.



- More examples

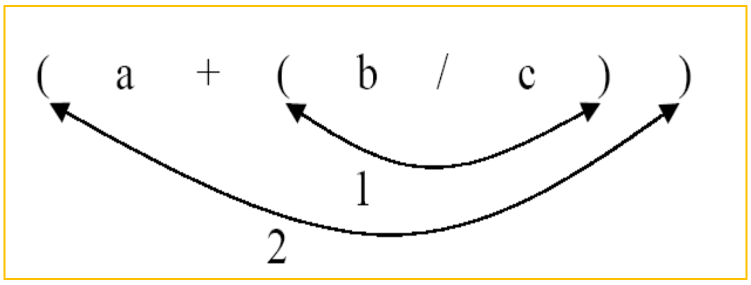
1. a + b = (a + b)
   = + a b
2. a + (b * c) = (a + (b * c))
   = + a * b c

5.7 Converting infix to postfix

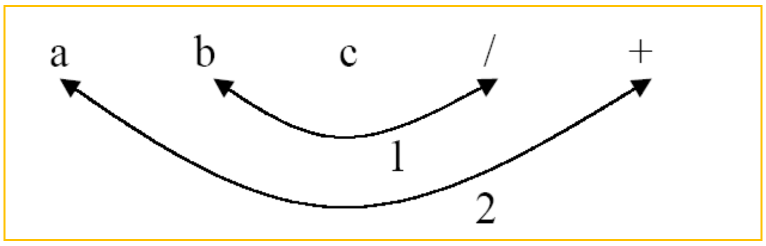
- Steps to convert infix expression to postfix:

  a + b / c

STEP 1: Add parentheses to the postfix expression based on the precedence, associative and parentheses rules.



STEP 2: Move forward the operators to closest close parentheses at the right side of them. Then, remove the parentheses.



- More examples

  1. a + b = (a + b)
     = a b +

  2. a + b * c = (a + (b * c))
     = a b c * +

  3. a + b * (c – d) / (p – r)
     = a + (b * (c – d)) / (p – r)
     = (a + ((b * (c – d)) / (p – r)))
     = a b c d - * p r - / +

5.8 Convert infix to postfix expression using stack

- Stack operations, such as **push()**, **pop()** and **isEmpty()** will be used to solve this problem.
- Algorithm to convert infix to postfix expression is as given in Algorithm 8.2

```
// Algorithm 8.2
create(s);
push(s, '#');
while (not end of infix input)
{       ch = getch  // next input character
        if (ch is an operan)
                add ch to postfix notation;
        if (ch = '(')
                push(s, ch)
        if (ch = ')')
        {       ch = pop(s);
                while (ch != '(')
                {       add ch to postfix notation;
                        ch = pop(s);
                }
     if (ch is an operator)
         {
                while (!isEmpty(s) & (precedence(stacktop()) >=
                    precedence(ch)))
                {       chpop = pop(s);
                    add chpop to postfix notation;
            }
            push(s, ch);
         }
}
while (stacktop() != '#')
{  ch = pop(s);
   add ch to postfix notation;
}
```

Note: Parenthetical expression operators ( ) are in the highest precedence level compared to others.

- Example 1 - Converting infix to postfix expression using stack

A + B * C – D / E

Infix	Stack	Postfix
A + B * C – D / E	#	
+ B * C – D / E	#	
B * C – D / E	# +	
* C – D / E	# +	
C – D / E	# + *	
– D / E	# + *	
D / E	# -	
/ E	# -	
E	# - /	
	# - /	
	#	

- Example 2 - Converting infix to postfix expression using stack

A * B – (C + D) + E

Infix	Stack	Postfix
A * B – (C + D) + E	#	
* B – (C + D) + E	#	A
B – (C + D) + E	# *	A
– (C + D) + E	# *	A B
(C + D) + E	# -	A B *
C + D	# - (	A B *

5.9	) + E		
	+ D) + E	# - (	A B * C
	D) + E	# - (+	A B * C
	) + E	# - (+	A B * C D
	+ E	# -	A B * C D +
	E	# +	A B * C D +
		# +	A B * C D +
		#	A B * C D +

Evaluate postfix expression using stack

- Steps to evaluate postfix expression
  - If the character read from postfix expression is an operand, **push** operand to stack.
  - If the character read from postfix expression is an operator, **pop** the first 2 operand in stack and implement the expression using the following operations:

    pop(opr1) dan pop(opr2)
    result = opr2 operator opr1

  - Push the result of the evaluation to stack.
  - Repeat steps 1 to steps 3 until end of postfix expression
  - Finally, at the end of the operation, only one value left in the stack. The value is the result of postfix evaluation.

- Algorithm to evaluate postfix expression is given in Algorithm 8.3

```
// Algorithm 8.3
create(s);
while(not end of postfix notation)
{  ch = getch();
   if (ch is operand)
     push(ch);
   else  //if ch = operator
   {
     operan1 = pop();
     operan2 = pop();
     result = operan2 ch operan1;
     push(result);
   }  //end else
}
pop(result);
```

- Example 1 - Evaluating postfix expression using stack

2 4 6 + *

Postfix	Ch	Opr	Opn1	Opn2	Result	Stack
2 4 6 + *						
4 6 + *	2					2
6 + *	4					2 4
+ *	6					2 4 6
*	+	+	6	4	10	2 10
	*	*	10	2	20	20

- Example 2 - Evaluating postfix expression using stack

2 7 * 18 - 6 +

Postfix	Ch	Opr	Opn1	Opn2	Result	Stack
2 7 * 18 – 6 +						
7 * 18 – 6 +	2					2
* 18 – 6 +	7					2 7
18 – 6 +	*	*	7	2	14	14
– 6 +	18					14 18
6 +	–	–	18	14	-4	-4
+	6					-4 6
	+	+	6	-4	2	2