Data Structures & Algorithms

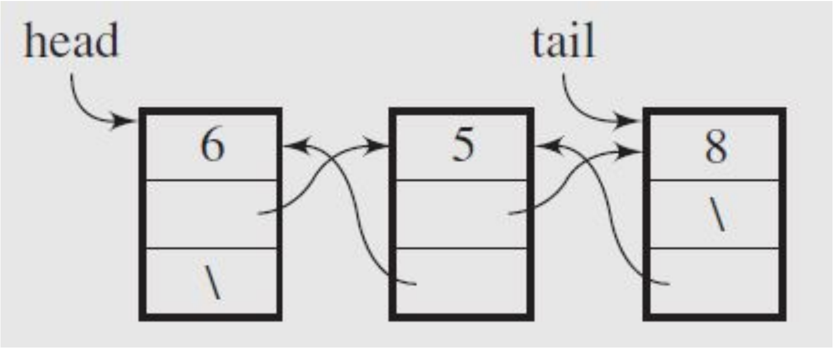
Stack & Its Applications

NUST

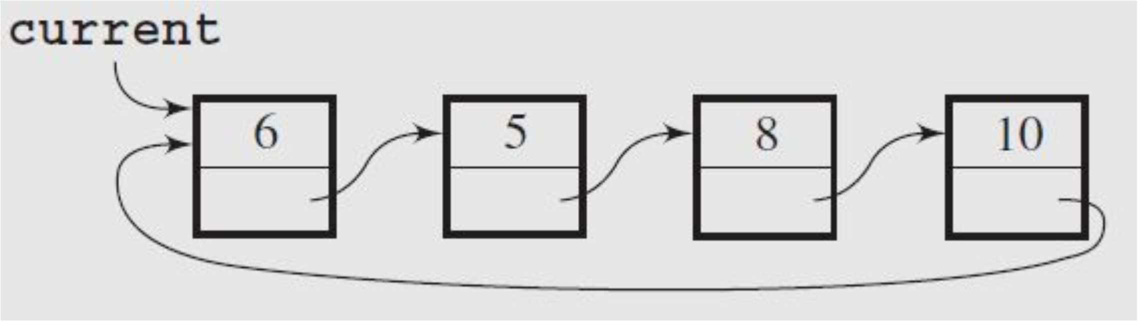
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Recap

- Doubly Linked List



- Circular Linked Lists



Today's Lecture

- Stack
- Practical applications
  - Stacks in validating expressions
  - Infix to Postfix conversion

Stack

Definition:

- Conceptually, a stack is a data structure that allows adding and removing elements in a particular order
  - More specifically, in an order such that items can be inserted or deleted to the collection of already inserted items from one end only, called the **top** of the stack

Stack

- Stack is said to have “**F**irst **I**n, **L**ast **O**ut" (FILO) or “**L**ast **I**n, **F**irst **O**ut” (LIFO) behaviour meaning that the first item added to a stack will be the last item removed from a stack

- Example:
  - Which is the first coin to pick up from the stack of gold coins?

Stack as an ADT

- A **stack** is an ***ordered*** **collection of data items in which** ***access is possible only at one end*** **(called the** ***top*** **of the stack).**
- ***Basic operations:***
- 1. **Construct a stack (usually empty)**
- 2. Check if stack **is empty**
- 3. **Push:** **Add** an element **at the top** of the stack
- 4. **Top:** **Retrieve** the **top element** of the stack
- 5. **Pop:** **Remove** the **top element** of the stack

Related terminology

- Top
  - A pointer that points the top element in the stack.
- Stack Underflow
  - When there is no element in the stack, we cannot pop an element: *stack underflow*
- Stack Overflow
  - When the stack contains equal number of elements as per its capacity and no more elements can be added: *stack overflow*

Basics of stack

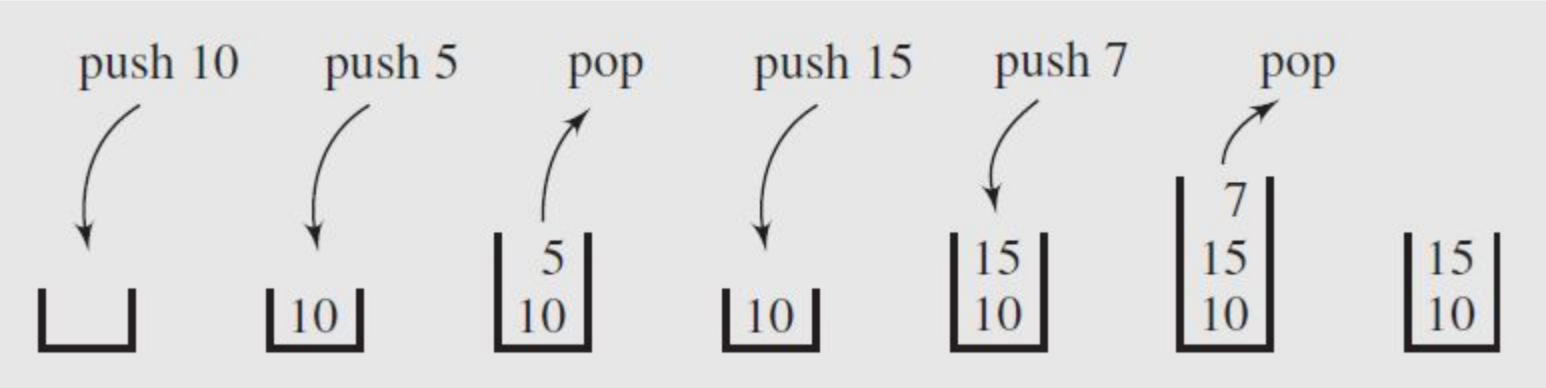
- **Size:** The number of elements on the stack

- **Top:** Points to the top most element on the stack. This refers to NULL if stack is empty or size = 0

Because we think of stacks in terms of the physical analogy,
we usually draw them vertically (so the top is really on top)

Stack Operations

- Primarily two operations performed on one end only
  - **Push:** Add an item on the top of stack
  - **Pop:** Remove an item at the top of stack



Stack Operations

- Push Concerns
  - What if when the stack is full?

- Pop Concerns
  - What if the stack is empty

Solution:

- Before any push, check if the stack is already filled or not
  - If it is filled then generate error message e.g., Stack Overflow
- Before pop, check if stack is not already empty

Stack Operations

push(item)	// Push an item onto the stack
pop( )	// Pop the top item off the stack
isEmpty()	// Return true if stack is empty
isFull()	// Return true if stack is full
top( ) or peek()	// Return value of top item

Applications

- Real life
  - Stack of trays in cafeteria
  - Piles of books in library

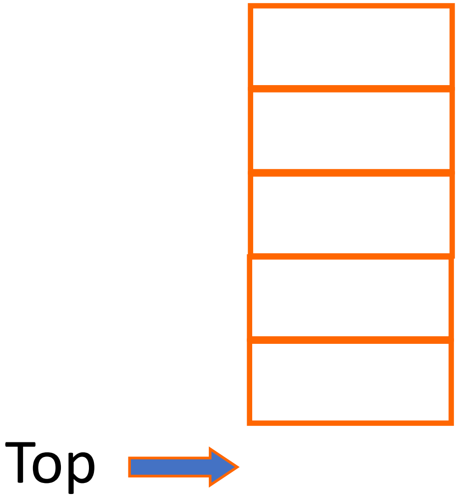
- Computer Science
  - Compilers use stacks to evaluate expressions & syntax parsing
  - Program execution stack
  - Undo operations
  - Expression conversion
  - Infix to post-, pre-fix and vice versa
  - Tower of Hanoi
  - And many more …

Reversing a Sequence

- Decimal to Binary Conversion
- Palindrome Check

Reversing a sequence

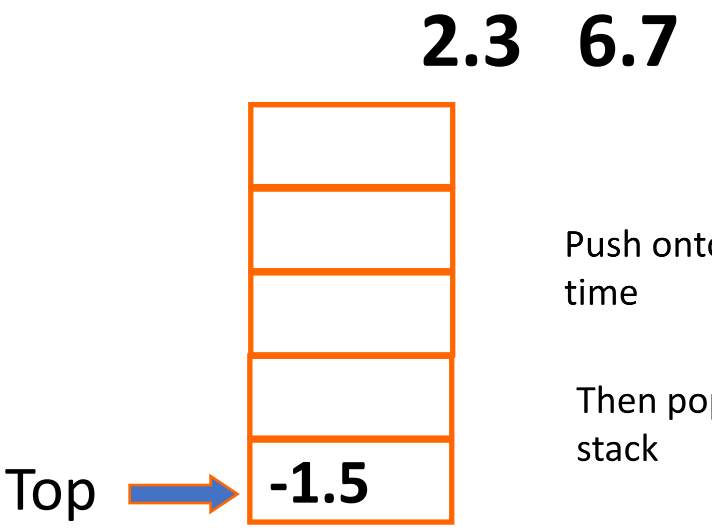
Given a sequence: **-1.5 2.3 6.7**



Push onto a stack one number at a time

Then pop each number off the stack

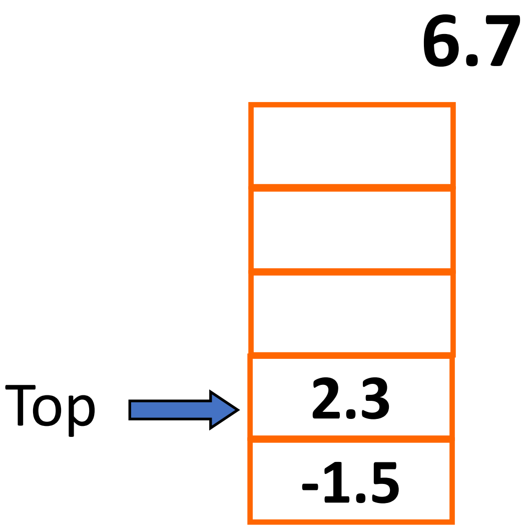
Reversing a Sequence



Push onto a stack one number at a time

Then pop each number off the stack

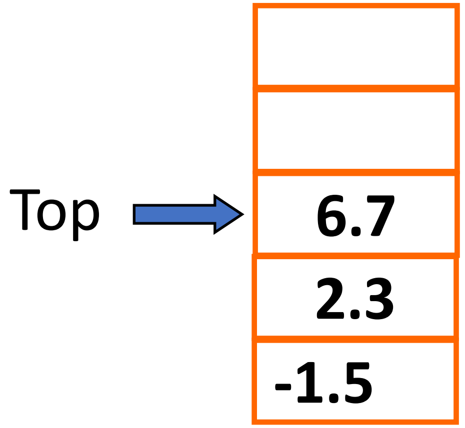
Reversing a Sequence



Push onto a stack one number at a time

Then pop each number off the stack

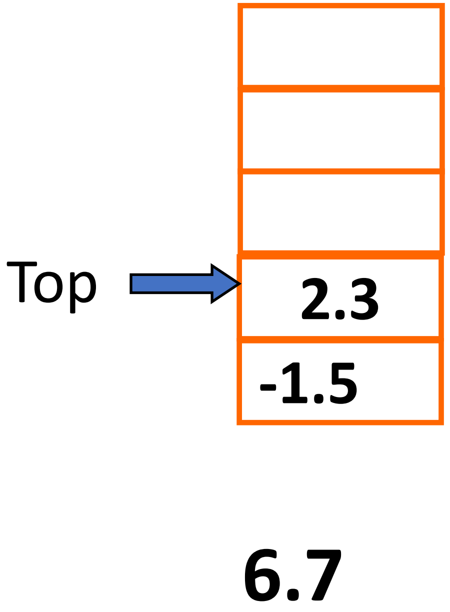
Reversing a Sequence



Push onto a stack one number at a time

Then pop each number off the stack

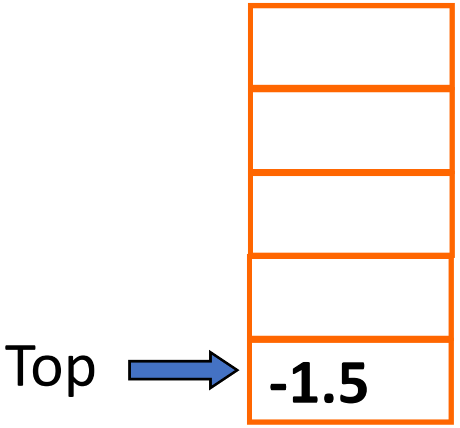
Reversing a Sequence



Push onto a stack one number at a time

Then pop each number off the stack

Reversing a Sequence

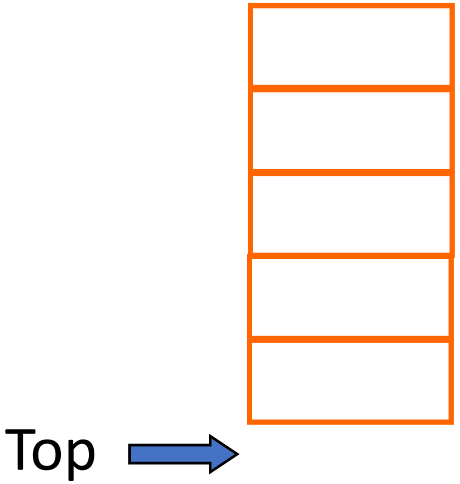


6.7 2.3

Push onto a stack one number at a time

Then pop each number off the stack

Reversing a Sequence



Push onto a stack one element at a time

Then pop each element off the stack

6.7 2.3 -1.5

Example: Conversion from decimal to binary

```
Read (number)
Loop (number > 0)
  digit = number modulo 2
  print (digit)
  number = number / 2
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.) To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Example: Conversion from decimal to binary using Stack

1. Create STACK.
2. Read Num.
3. Repeat while NUM > 0
    - 3.1 SET Digit := Num % 2.
    - 3.2 If STACK is full, then:
        - 3.2.1 Print “Stack Overflow”
        - 3.2.2 Return.
    - 3.3 PUSH(Digit).
    - 3.5 SET Num := Num/2.
4. Repeat while STACK is not empty.
    - 4.1 POP().
    - 4.2 Write Digit.
5. Return.

Choice of implementation

- Array based: Stack Maximum stack size is **known** ahead of time

- Linked List based: Stack Maximum stack size **unknown**

Array based stack

NUST SEECS

- When using an array to implement a stack
  - The array's first element should represent the bottom of the stack
  - The last occupied location in the array represents the stack's top
- This avoids shifting of elements of the array when we push or remove elements from stack

Array based stack

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- Last index in the array is the top
- Increment top when one element is pushed, similarly decrement after each pop

Array based stack operations

NUST SEECS

```
int stack[size];
int top = -1;

void Push(int element) {
if (!isFull()) {
        stack[++top] = element;
        }
}
bool isFull() {
        return top==size-1;
}
```

```
int Pop() {
int element = -1;
If (!isEmpty()) {
        element = stack[top];
        stack[top--] = -1;
}
return element;
}
bool isEmpty() {
        return top<0;
}
```

Linked List based implementation

- LIFO
  - Push = Insert at head
  - Pop = Remove at head
  - IsEmpty = Empty List
  - Clear = Delete List

Linked List based implementation

```
class node
{
public:
   int value;
   node* next;
}; //class node
```

```
class stack
{
   int size;
   node* top;
public:
    stack() {
        size=0;
        top=NULL;
     } //default constructor
     void push(int);
    bool pop();
    bool isEmpty(); };
```

Linked List based implementation

```
void stack::push(int el)
{
  node *temp;
  temp =new node;
  if (top==NULL) {
        temp->next =NULL;
   }
   else {
        temp->next =top;
   }
   temp->value=el;
   top=temp;
   size++; }
```

```
void main ()
{
    ...
    stack();
    stack.push(5);
    ...
}
```

Linked List based implementation

```
int stack::pop()
{    int d;
  if(isEmpty()) {
        cout<<"\nStack is Empty\n";
        return 0;
  }
  else {
    Node *temp = top;
    top=top->next;
    d=temp->value;
    delete temp;
   size--; }
  return d; }
```

```
bool stack::isEmpty()
{
        if(getStackSize()==0)
              return true;

        return false;
}
```

Today's Lecture

- Stack
- Practical applications
  - Stacks in validating expressions
  - Infix to Postfix conversion

Algebraic expression

- An algebraic expression is a legal combination of **operands** and the **operators**
- **Operand** is the quantity (unit of data) on which a mathematical operation is performed
  - E.g., operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- **Operator** is a symbol which signifies a mathematical or logical operation between the operands
  - Example of familiar operators include +,-,*, /, ^
- Considering these definitions of operands and operators, an example of expression may be **x+y*z**

Stacks in problem solving

- Consider a mathematical expression that includes several sets of nested parenthesis, e.g.,

(x + (y – (a +b)))

- We want to ensure that parenthesis are nested correctly and the expression is valid

Validation

- There is an equal number of right and left parentheses
- Every right parenthesis is preceded by a matching left parenthesis

(A+B) * C	Valid
A + B)	Invalid
(A+B]	Invalid

Rules

- Each left parentheses is the opening scope while each right parenthesis is a closing scope

- Parentheses count = 0 at the end means that no scopes have been left open and left and right parentheses exactly match

((A + B)	(A * B)
1 2 2 2 2 1	1 1 1 1 0
Invalid expression	Valid expression

Parsing parenthesis

- We have seen use of stack in validation of expression using only ‘(‘ and ‘)’ scope delimiter

- Let us change the problem slightly
  - Three different kinds of scope delimiters exist

    e.g., **{ x + (y – [a +b]) }**

  - The scope ender must be of same type as scope opener

  - It is necessary to keep track of not only the count of scope but also the types

  - A stack may be used to keep track of the types of scopes encountered

Stacks in validating expression

NUST SEECS

- Whenever a scope opener is encountered
  - it is pushed into the stack
- Whenever a scope ender is encountered,
  - The stack is examined
  - If the stack is empty, the scope ender does not have a matching opener and string is invalid
  - If stack is not empty, we pop an item and check if it corresponds to scope ender
    - If match occurs we continue,
    - otherwise the expression is invalid
- At the end of the string the stack must be empty

NUST SEECS

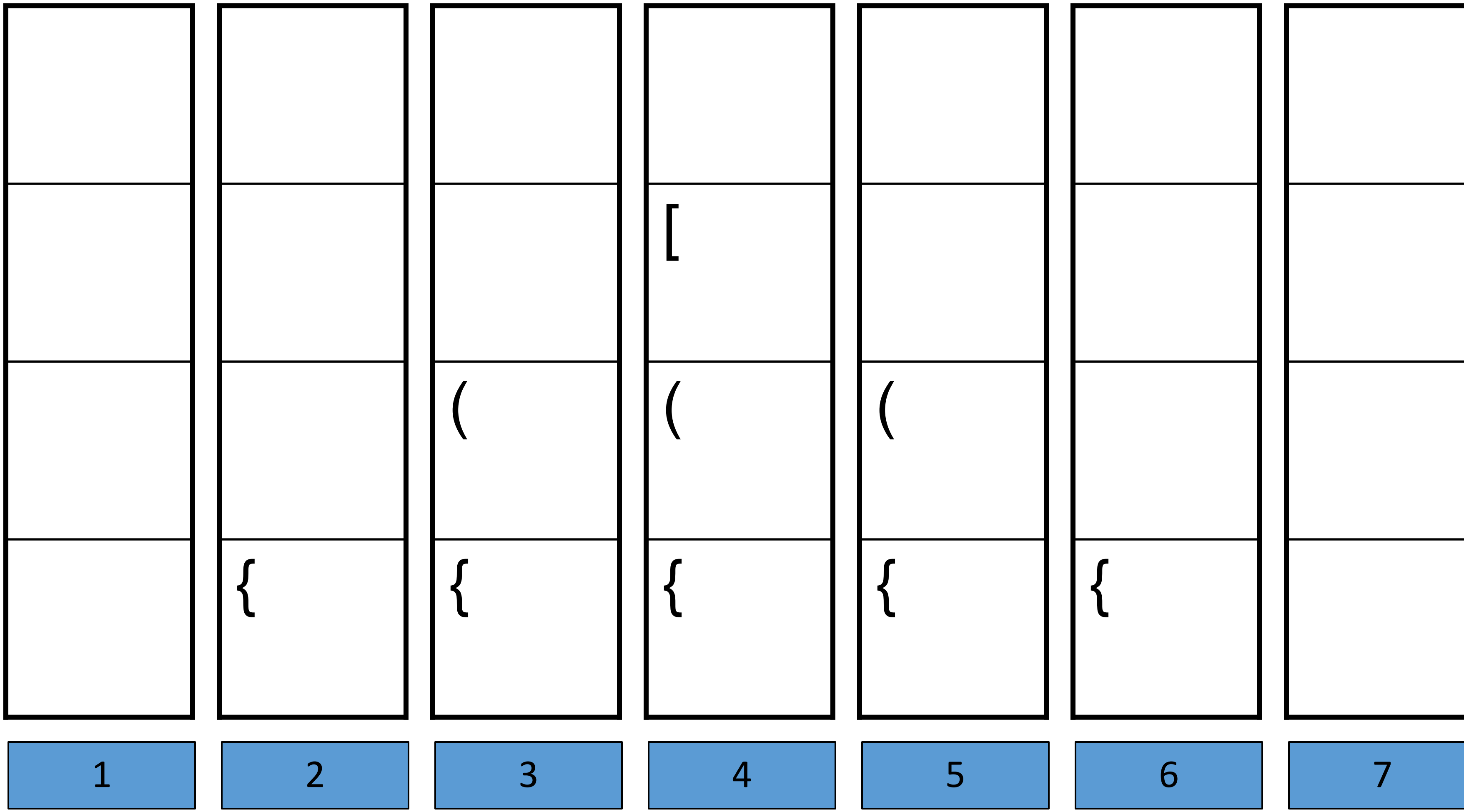
Pseudo code

```
Valid = true
S = the empty stack // array of chars say char s[100]
While (we have not read the entire string) {

        read the next symbol (symb) of the string;
        if (symb == '(' || symb == '[' || symb == '{')
                push (s, symb);
        if (symb == ')' || symb == ']' || symb == '}')

            if (IsEmpty(s))

                valid = false;

            else {

                I = pop (s);
                if ( I is not the matching opener of symb)
                    valid = false;

            }// end of else
} //end while
If (valid)
   cout << "Valid String" << endl;
Else
   cout << "not a valid string"
```

Stacks in validating expression

NUST SEECS

{ x + (y – [a +b]) }



Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d

- This is done by assigning operator priorities
  - priority(*) = priority(/) > priority(+) = priority(-)

- When an operand lies between two operators, the operand is **associated** with the operator that has the higher priority

Tie breaker

- When an operand lies between two operators that have the same priority, the operand is associated with the operator on the **left**
  - a + b - c
  - a * b / c / d

Infix, Prefix & Postfix expressions

- INFIX: Expressions in which **operands** surround the **operator**,
  - e.g., x+y, 6*3 … etc.

- PREFIX: Expressions in which **operator** comes before the **operands**, e.g., +xy, *+xyz etc.
  - Also known as Polish Notation (PN)

- POSTFIX: Expressions in which **operator** comes after the **operands**, e.g., xy+, xyz+* etc.
  - Also known as Reverse Polish Notation (RPN)

Prefix & Postfix notations, why to use?

NUST SEECS

- Why to use these weird looking PREFIX and POSTFIX notations when we have simple INFIX notation?
  - INFIX notations are not as simple as they seem specially while evaluating them
  - To evaluate an infix expression we need to consider **operators' Priority** and **Associative property**

    -E.g., expression 3+5*4 can either be evaluated as 32 i.e., (3+5)*4 or 23 i.e., 3+(5*4)
- To solve this problem **Precedence** or **Priority** of the operators were defined to govern the evaluation order i.e., operator with higher precedence is applied before an operator with lower precedence

Infix expression is hard to parse

NUST SEECS

- Need operator priorities, tie breaker, and delimiters which makes computer evaluation more difficult than is necessary

- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters
  - I.e., need not to consider the Priority and Associative property (order of brackets)
    - E.g. **x/y*z** becomes ***/xyz** in prefix and **xy/z*** in postfix
  - So it is easier to evaluate expressions that are in these forms

Prefix & Postfix notations, why to use?

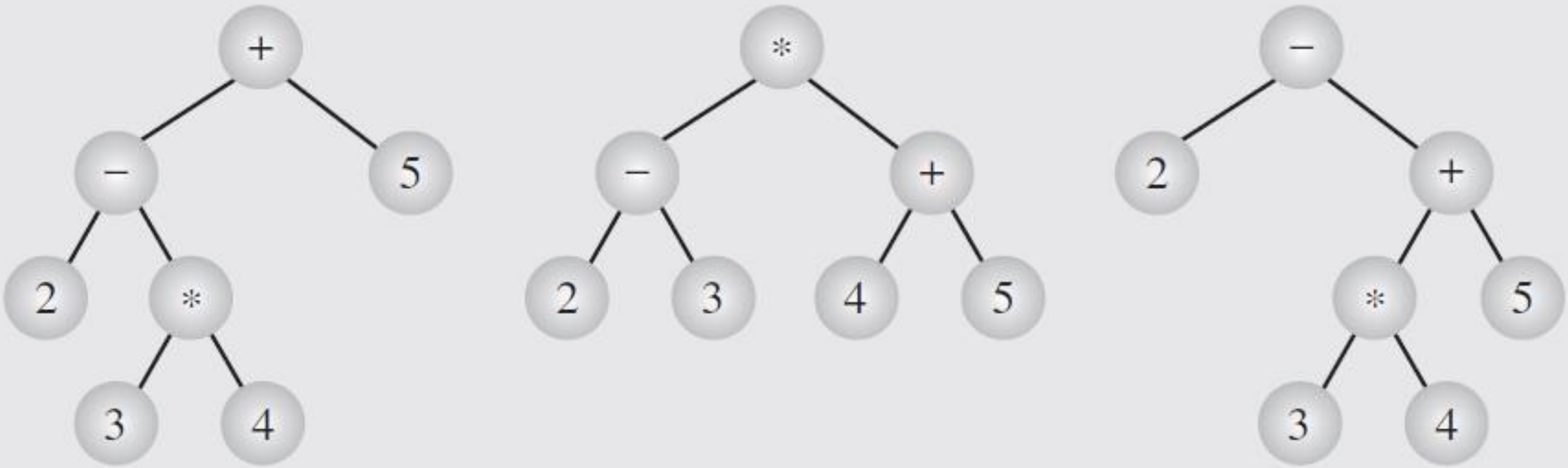
- Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations

- Therefore, all expressions have to be broken down unambiguously into separate operations and put into their proper order

- That is where Polish notation is useful as it allows us to create an expression tree, which imposes an order on the execution of operations

Expression tree

NUST SEECS



$2 - 3 * 4 + 5$	$(2 - 3) * (4 + 5)$	$2 - (3 * 4 + 5)$
$+ - 2 * 3\ 4\ 5$	$* - 2\ 3 + 4\ 5$	$- 2 + * 3\ 4\ 5$
$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$
$2\ 3\ 4 * - 5 +$	$2\ 3 - 4\ 5 + *$	$2\ 3\ 4 * 5 + -$

Prefix Infix Postfix

Infix to Postfix

Example:

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication, Let D=BC*

A+D Convert the addition

AD+

ABC*+ Postfix Form

Examples of Infix to Post- & Prefix

NUST SEECS

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

More examples

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

Infix to Postfix conversion algorithm

- Create an empty stack for keeping operators
- Create an empty list for output
- Scan each element of input expression from left to right
  - If the element is an operand, append it to the end of the output list
  - If the element is a left parenthesis, push it on the stack
  - If the element is a right parenthesis, apply pop operation to the stack until the corresponding left parenthesis is removed & append each operator to the end of the output list
  - If the element is an operator, *, /, +, or -, push it on the stack. However, first remove any operators already on the stack that have higher or equal precedence and append them to the output list
- When the input expression has been completely processed, check the stack. Any operators still on the stack can be removed and appended to the end of the output list

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	**23*21-/53*+**

Evaluating Postfix expression

- Each operator in a postfix string refers to the previous two operands in the string
- Suppose that each time we read an operand we push it into a stack. When we reach an operator, its operands will then 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 back onto the stack so that it will be available for use as an operand of the next operator

Algorithm to evaluate Postfix expression

NUST SEECS

1) Create an empty stack
2) Scan the elements of Postfix expression from left to right
3) If the element is an operand, push the value onto the stack
4) If the element is an operator, *, /, +, or -
   - Pop the stack twice. The first pop is the second operand and the second pop is the first operand
   - Perform the arithmetic operation & push the result back on the stack
4) When the input expression has been completely processed, apply pop operation onto the stack to return the computed value

Algorithm to evaluate Postfix expression

NUST SEECS

```
WHILE more input items exist
{
   If symb is an operand
         then push (stk,symb) // stk is the stack used
   else   //symbol is an operator
   {
         Opnd1=pop(stk);
         Opnd2=pop(stk);
         Value = result of applying symb to opnd1 & opnd2
         Push(stk,value);
   }              //End of else
} // end while
Result = pop (stk);
```

Algorithm to evaluate Postfix expression