Data Structures using [R1UC308B]

Module-VI: Stack

Dr. Subhash Chandra Gupta

GALGOTIAS UNIVERSITY
School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

July 31, 2025

Contents

Introduction to Stack

- A Stack is a linear data structure that follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.
- A variable name **Top** is used to keep the position of the topmost element in the stack.
- we can access the elements only on the top of the stack

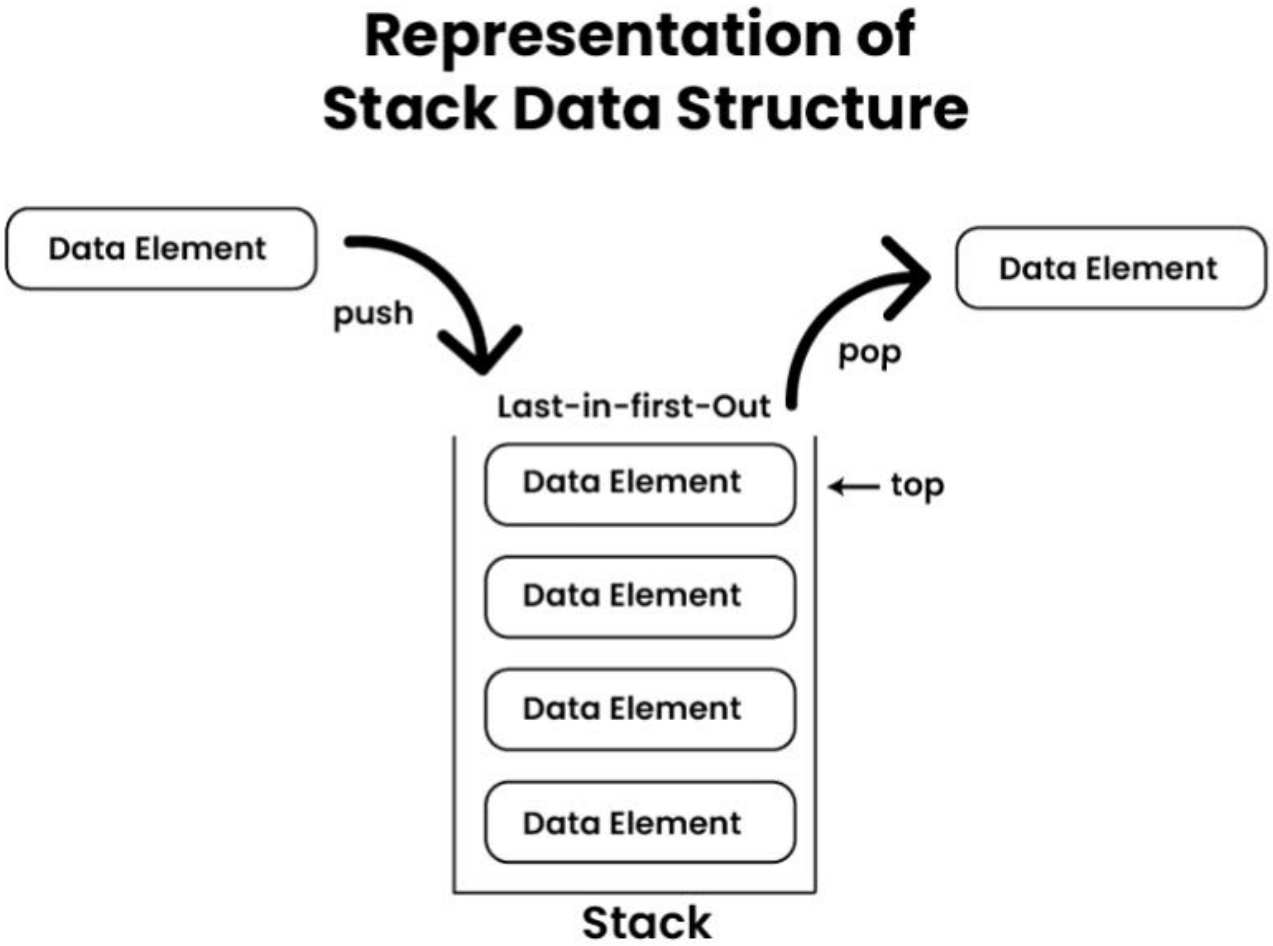

Figure: Stack Structure

Types of Stack based on Size:

- **Fixed Size Stack** : As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- **Dynamic Size Stack** : A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Abstract Data Type

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

- In Stack ADT implementation instead of data being stored in each node, the pointer/reference to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT.
- The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to top.

```java
public class StackADT {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
        while(!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

Figure: Stack ADT

Primitive Stack operations

- Push: Adds an element to the top of the stack
- Pop: Removes the top element from the stack
- Peek: Returns the top element without removing it
- IsEmpty: Checks if the stack is empty
- IsFull: Checks if the stack is full (in case of fixed-size arrays)

Stack Implementation

- There are two types of Stack based on Size: Fixed Size Stack and Dynamic Size Stack
- Arrays are used to implement the fixed size Stack.
- Size of the array declared at the time of declaration of the array in Java is the size of the Stack.
- For example int Stack[20], here maximum stack size is limited upto 20. and the valid range of the Top is from 0 to 19.
- Top having value -1 is the condition for empty stack and Top having value 20 is the condition for full stack.
- When stack is empty, nothing can be popped.
- When stack is full, nothing can be pushed.
- Dynamic size stack can be implemented using the link link.

Array Implementation of Stack

- In push, we check the overflow condition. If Top=size-1 then stack is full and insertion of item is not possible, the stack is in overflow state.
- In pop, we check the underflow condition. If Top=-1 then stack is empty and deletion is not possible, the stack is in underflow state.
- Below is the complete program of stack implementation using array.


```
class Stack {
    static final int MAX = 2;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
     if(top<0) return true; else return false;
        //return (top < 0);
    }
    Stack()
    {
        top = -1;
    }

    boolean push(int x)
    {
```


```
    if (top >= (MAX - 1)) {
        System.out.println("Stack Overflow");
        return false;
    }
    else {
     top=top+1;
        a[top] = x;
        System.out.println(x + " pushed into stack");
        return true;
    }
}

int pop()
{
    if (top < 0) {
        System.out.println("Stack Underflow");
        return 0;
```

```
        }
        else {
            int x = a[top];
            top=top-1;
            return x;
        }
    }

    int peek()
    {
        if (top < 0) {
            System.out.println("Stack Underflow");
            return 0;
        }
        else {
            int x = a[top];
            return x;
```

```
        }
    }
    boolean isFull() {
     if(top==MAX-1) return true; else false;
    }

}

class StackArray {
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.pop();//Stack underflow
        s.push(10);
        s.push(20);
        s.push(30); //Stack overflow
        System.out.println(s.pop() + " Popped from stack");
```


```
            System.out.println("Top element is :" + s.peek());
        }
    }
```

Linked List Implementation of Stack

- To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.
- In singly linked list if we perform only addition and deletion at front i.e. at Head and if we rename Head as Top then singly linked list will be behave just like a Stack.
- In the stack Implementation, a stack contains a top pointer which is the "head" of the stack where pushing and popping items happens at the head of the list.

- The first node(right most/bottom most) has a null in the link field and second node-link has the first node address in the link field and so on and the last node(left most/top most) address is in the "top" pointer.

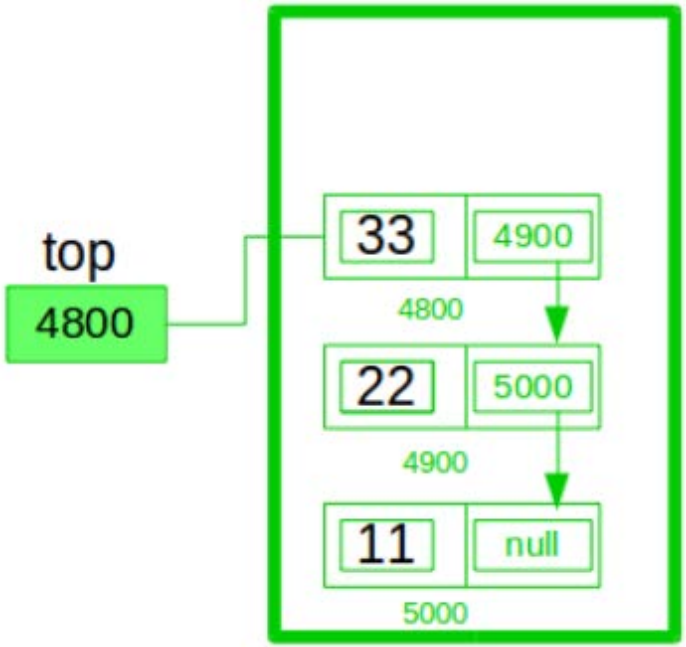

Figure: Stack using SLL

- The main advantage of using a linked list over arrays is that it is possible to implement a stack that can shrink or grow as much as needed.
- Using an array will put a restriction on the maximum capacity of the array which can lead to stack overflow.
- Here each new node will be dynamically allocated, so overflow is not possible.

Stack Operations using SLL

- push(): Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.

```
void push(int new_data) {
    Node new_node = new Node(new_data);
    if (new_node == null) {
        System.out.println("\nStack Overflow");
        return;
    }
    new_node.next = top;
    top = new_node;
}
```

Figure: push()

- pop(): Return the top element of the Stack i.e simply delete the first element from the linked list.

```
void pop() {
    if (top==null) {
        System.out.println("\nStack Underflow");
        return;
    }
    else {
        Node temp = top;
        top = top.next;
        System.out.println("Popped element is " + temp.data);
        temp = null;
    }
}
```

Figure: pop()

- peek(): Return the top element.

```
int peek() {
    if (top!=null)
        return top.data;
    else {
        System.out.println("\nStack is empty");
        return Integer.MIN_VALUE;
    }
}
```

Figure: peek()

- isFull(): if newly created node is null then Stack is full.

```
boolean isFull() {
    Node new_node = new Node();
    if(new_node==null) return true;
            else return false;
}
```

Figure: isFull()

- isEmpty(): if top is null then Stack is empty.

```
boolean isEmpty() {
    if(top==null) return true;
            else return false;
}
```

Figure: isEmpty()

- Below is the complete program of stack implementation using singly linked list.


```
class Node {
    int data;
    Node next;
    Node(){
       data = 0;
       next = null;
    }
    Node(int d) {
       data = d;
       next = null;
    }

class StackL {
Node top;
StackL() { //constructor
top = null;
}
```


```
boolean isEmpty() {
if(top==null) return true;
else return false;
}

boolean isFull() {
Node new_node = new Node();
if(new_node==null) return true;
else return false;
}

void push(int new_data) {
Node new_node = new Node(new_data);
if (new_node == null) {
System.out.println("\nStack Overflow");
return;
```


```
}
new_node.next = top;
top = new_node;
}

void pop() {
if (top==null) {
System.out.println("\nStack Underflow");
return;
}
else {
Node temp = top;
top = top.next;
System.out.println("Popped element is " + temp.data);
temp = null;
}
}
```


```
int peek() {
if (top!=null)
return top.data;
else {
System.out.println("\nStack is empty");
return Integer.MIN_VALUE;
}
}
}

public class StackLink {
public static void main(String[] args)
{
// Creating a stack
StackL st = new StackL();
st.pop();
```


```
st.push(11);
st.push(22);
st.push(33);
st.push(44);
System.out.println("Top element is " + st.peek());
System.out.println("Removing two elements...");
st.pop();
st.pop();
System.out.println("Top element is " + st.peek());
}
}
```

Application of Stack

- **Expression evaluation and conversion**: Stacks can evaluate expressions with operators and operands, and convert expressions from one form to another
- **Backtracking**: Stacks can check for matching parentheses in an expression
- **Function call management**: Stacks manage function calls in real-time applications
- **Memory allocation**: Stacks are used for systematic memory management
- **Task scheduling**: Stacks are used for task scheduling in real-time applications
- **Program flow management**: Stacks are crucial for managing program flow

- **History maintenance**: Stacks are used to maintain history
- **Undo functionality**: Stacks are used to implement undo functionality
- **Web browsing history**: Stacks help manage web browsing history in browsers
- **System memory architecture**: Stacks are used in system memory architecture
- Postfix Expressions
- Prefix Expressions
- Evaluation of postfix expression

Postfix Expressions

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
   - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a '(', push it to the stack.
5. If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-5 until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Prefix Expressions

- Step 1: Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.
- Step 2: Convert the reversed infix expression to "nearly" postfix expression. While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
- Step 3: Reverse the postfix expression.

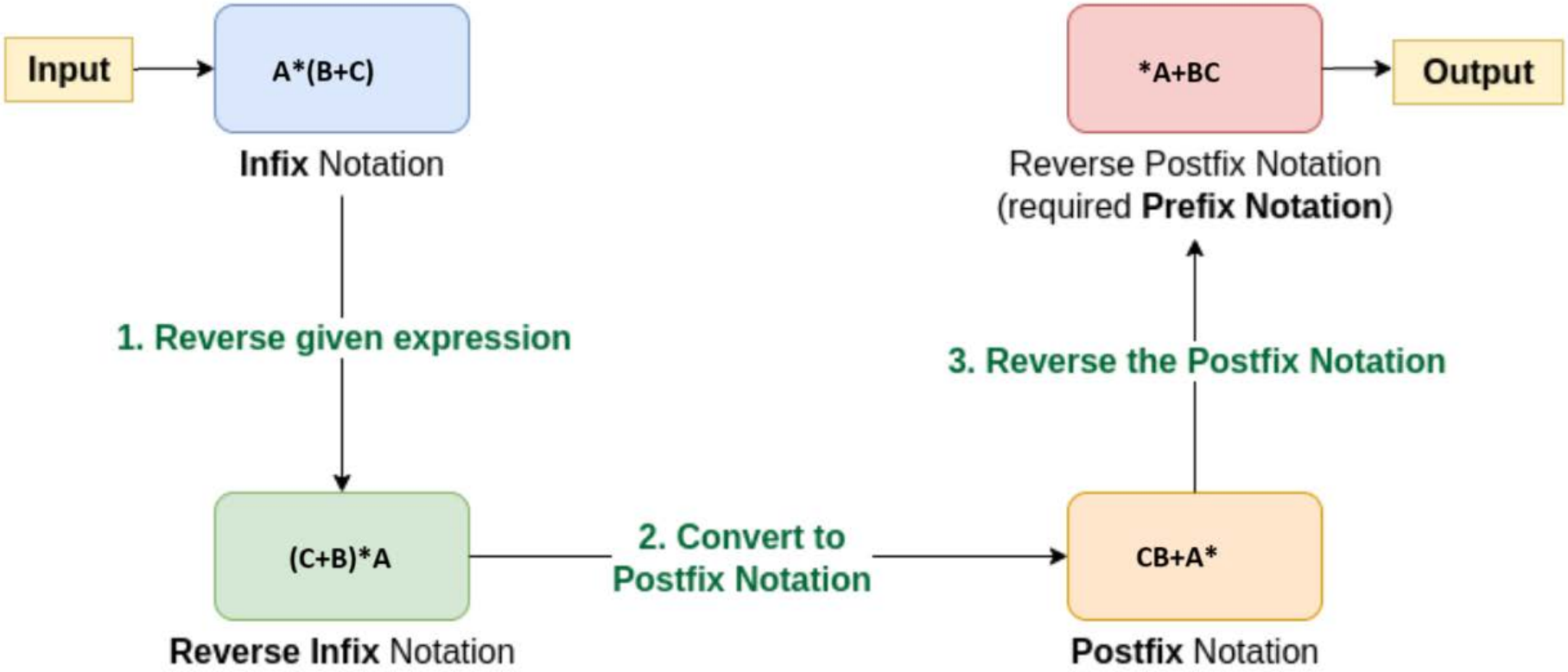

Figure: Convert infix expression to prefix expression

Evaluation of postfix expression

- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
  - If the element is a number, push it into the stack.
  - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- When the expression is ended, the number in the stack is the final answer.

Thank you

Please send your feedback or any queries to
subhash.chandra@galgotiasuniversity.edu.in