Data Structure and Algorithms

Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

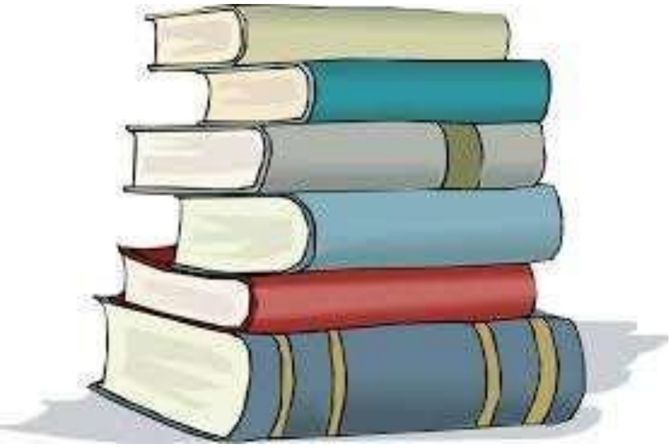
Stack

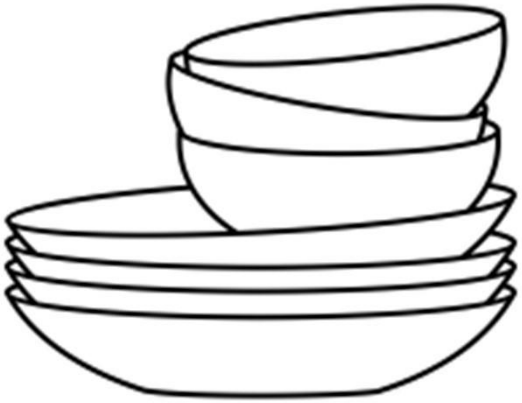
- A stack is a special kind of list
  - Insertion and deletions takes place at one end called top

- Other names
  - Push down list
  - Last In First Out (LIFO)

Stack Examples

- Books on floor
- Dishes on a shelf



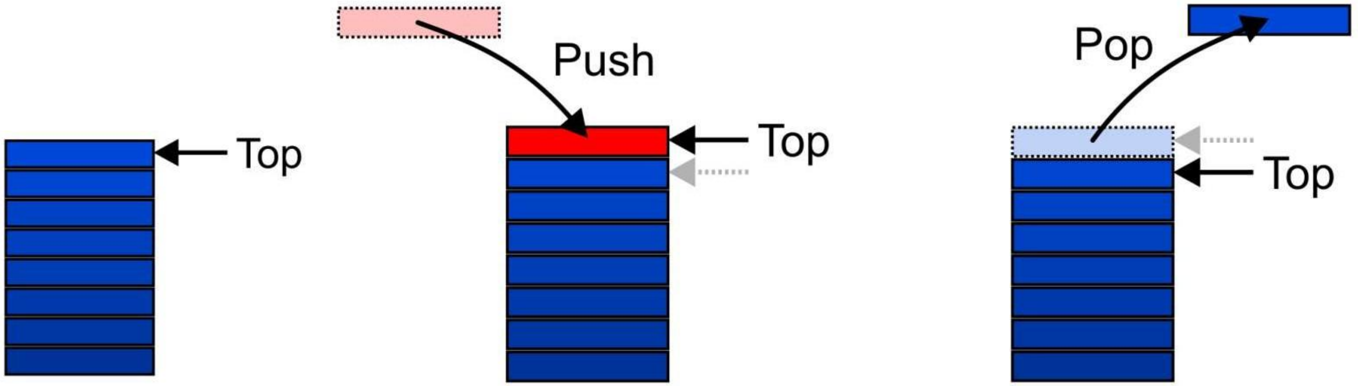


Stack ADT

- Stack ADT emphasizes specific operations
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually
  - Inserted objects are pushed onto the stack
  - Top of the stack is the most recently object pushed onto the stack
  - When an object is popped from the stack, the current top is erased

Stack ADT – Operations

- Graphically, the stack operations are viewed as follows:



Stack ADT – Operations

- MAKENULL(S)
  - Make Stack S be an empty stack

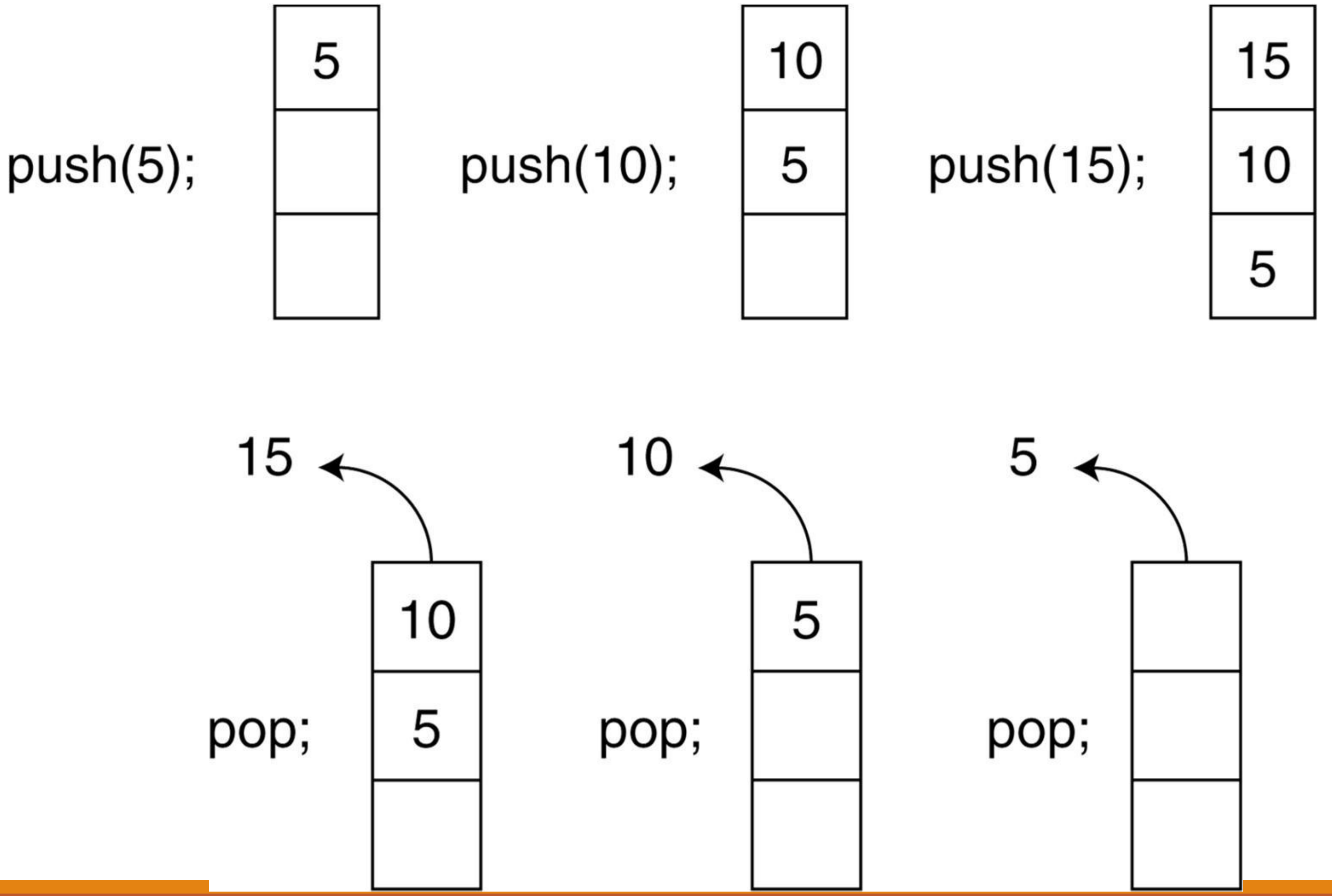
- TOP(S)
  - Return the element at the top of stack S

- POP(S)
  - Remove the top element of the stack

- PUSH(S,x)
  - Insert the element x at the top of the stack

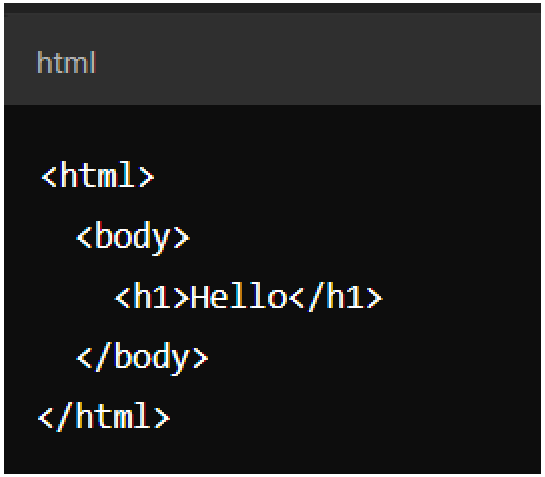
- EMPTY(S)
  - Return true if S is an empty stack and return false otherwise

Push and Pop Operations of Stack



Applications

- Many applications
  - Parsing code
    - ➢ Matching parenthesis
    - ➢ XML (e.g., XHTML)
  - Tracking function calls
  - Dealing with undo/redo operations



```html
<html>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

- The stack is a very simple data structure
  - Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Applications

- Problem solving
  - Solving one problem may lead to subsequent problems
  - These problems may result in further problems
  - As problems are solved, focus shifts back to the problem which lead to the solved problem

- Notice that function calls behave similarly
  - A function is a collection of code which solves a problem

Use of Stack in Function Calls

- When a function begins execution an activation record is created to store the current execution environment for that function

- Activation record contains all the necessary information about a function call, including
  - Parameters passed by the caller function
  - Local variables
  - Content of the registers
  - (Callee) Function’s return value(s)
  - Return address of the caller function
    - ➢ Address of instruction following the function call

Use of Stack in Function Calls

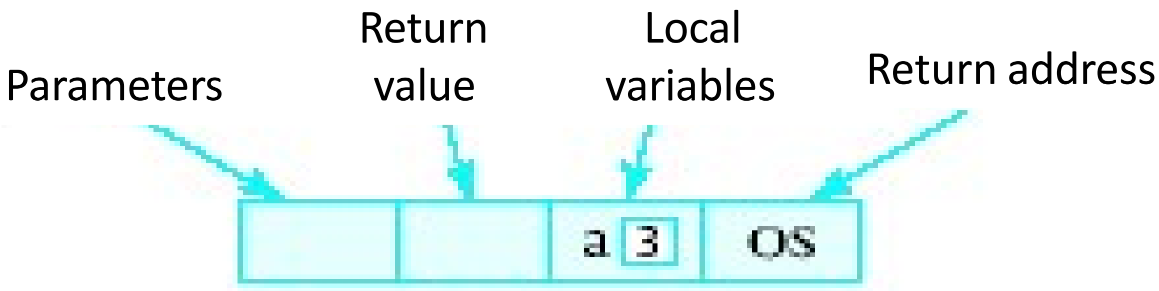
- Each invocation of a function has its own activation record
- Recursive/Multiple calls to the functions require several activation records to exist simultaneously
- A function returns only after all functions it calls have returned Last In First Out (LIFO) behavior
- A program/OS keeps track of all the functions that have been called using run-time stack

Runtime Stack Example

```
void main(){
int a=3;
    f1(a); // statement A
    cout << endl; }

void f1(int x){
    cout << f2(x+1); // statement B
}

int f2(int p){
    int q=f3(p/2); // statement C
    return 2*q;
}

int f3(int n){
    return n*n+1; }
```

Runtime Stack

- When a function is called …
  - Copy of activation record pushed onto run-time stack
  - Arguments copied into parameter spaces
  - Control transferred to starting address of body of function



Parameters

Return value

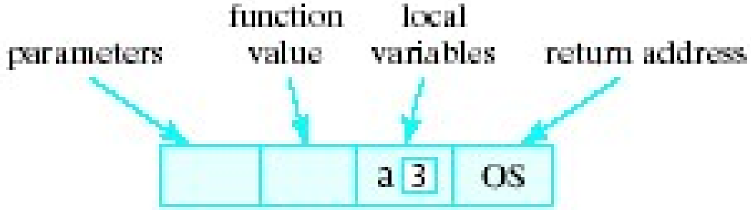
Local variables

Return address

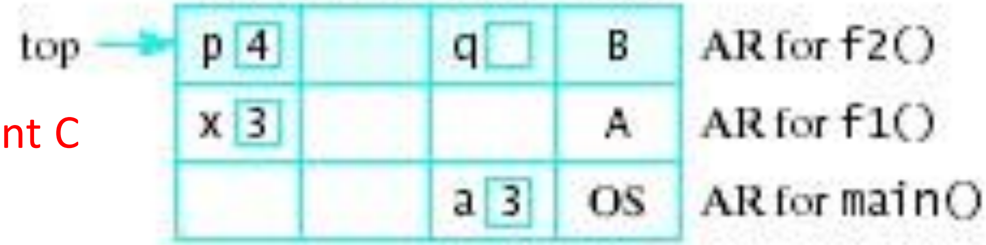
OS denotes that when execution of main() is completed, it returns to the operating system

Runtime Stack Example

```
void main(){
int a=3;
    f1(a); // statement A
    cout << endl; }

void f1(int x){
    cout << f2(x+1); // statement B
}

int f2(int p){
    int q=f3(p/2); // statement C
    return 2*q;
}

int f3(int n){
    return n*n+1; }
```



Function call f2(x + 1)



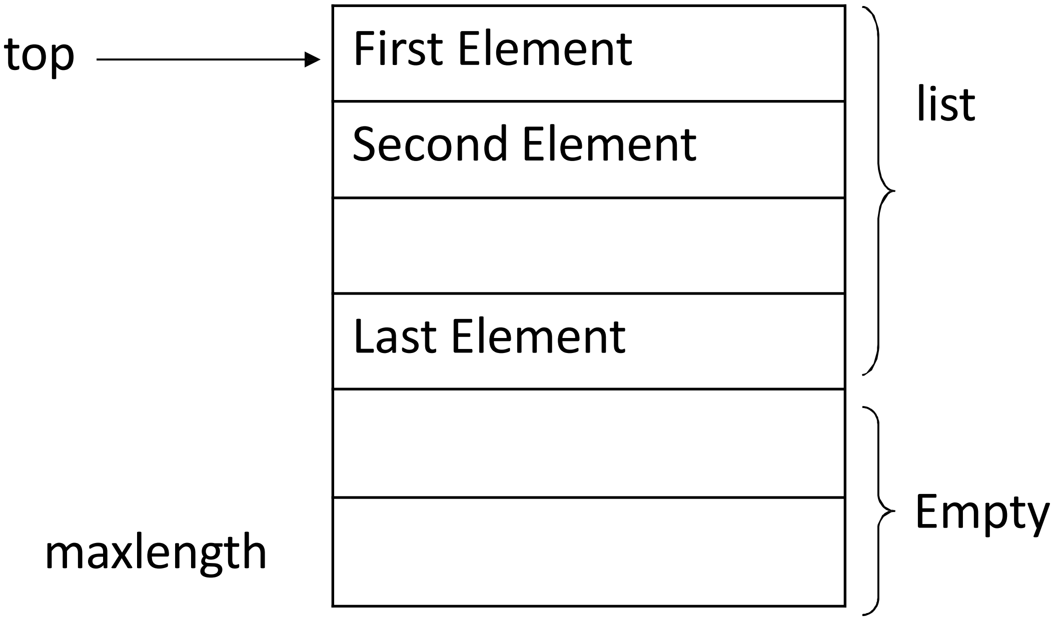
p 4		q □	B	AR for f2()
x 3			A	AR for f1()
		a 3	OS	AR for main()

Static and Dynamic Stacks

- Two possible implementations of stack data structure
  - Static, i.e., fixed size implementation using arrays
  - Dynamic implementation using linked lists

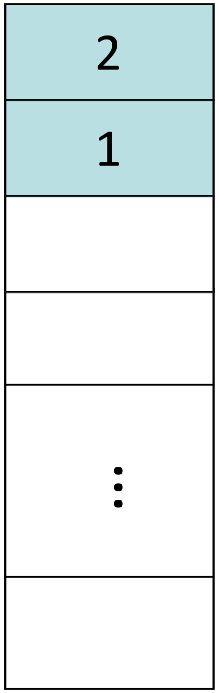
Array Implementation – First Solution

- Elements are stored in contiguous cells of an array
- New elements can be inserted to the top of the list



Array Implementation – First Solution

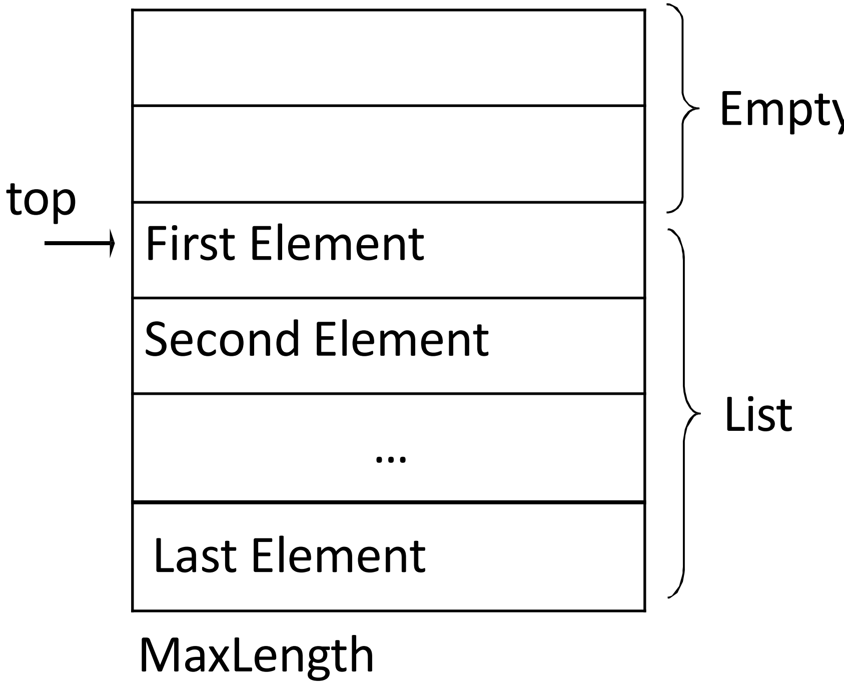
- Problem
  - Every PUSH and POP requires moving the entire array up and down



Array Implementation – Better Solution

Idea

- Anchor the top of the stack at the bottom of the array
- Let the stack grow towards the top of the array
- Top indicates the current position of the first stack element



top

First Element

Second Element

…

Last Element

MaxLength

Empty

List

Array Implementation – Code

```
class IntStack {
    private:
        int  *stackArray;
        int  stackSize;
        int  top;
    public:
        IntStack(int);
        ~IntStack( );
        void  push(int);
        void  pop(int  &);
        bool  isFull(void);
        bool  isEmpty(void); };
```

Array Implementation – Code

- Constructor

```
IntStack::IntStack(int size) {
        stackArray = new int[size];
        stackSize = size;
        top = -1;
}
```

- Destructor

```
IntStack::~IntStack(void) {
        delete  [] stackArray;
}
```

Array Implementation – Code

- isFull function

```
bool IntStack::isFull(void)
{
    bool status;
    if (top == stackSize - 1)
        status = true;
    else
        status = false;
    return status; // return (top == stackSize-1);
}
```

- isEmptyfunction

```
bool IntStack::isEmpty(void)
{
    return (top == -1);
}
```

Array Implementation – Code

- push function inserts the argument numonto the stack

```
void  IntStack::push(int  num)
{
    if  (isFull())
    {
        cout << "The stack is full.\n";
    }
    else
    {
        top++;
        stackArray[top] = num;
    }}
```

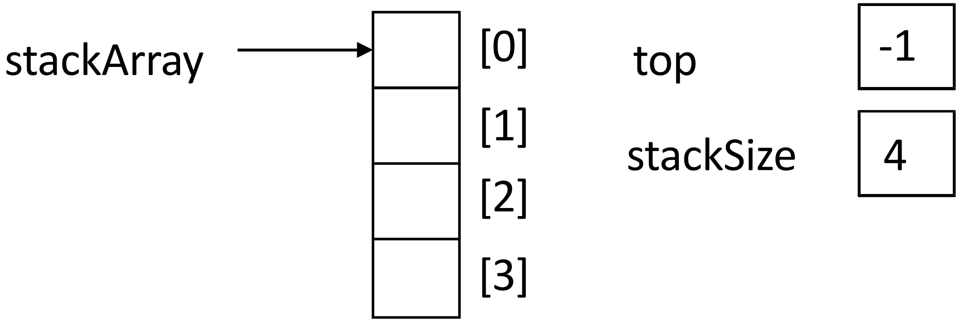
Array Implementation – Code

- Pop function removes the value from top of the stack and returns it as a reference

```
void  IntStack::pop(int  &num)
{
    if  (isEmpty())
    {
        cout << "The stack is empty.\n";
    }
    else
    {
        num = stackArray[top];
        top--;
    }
}
```

Using Stack

```
void main(void)
{
   IntStack stack(4); }
```



stackArray

[0]

[1]

[2]

[3]

top

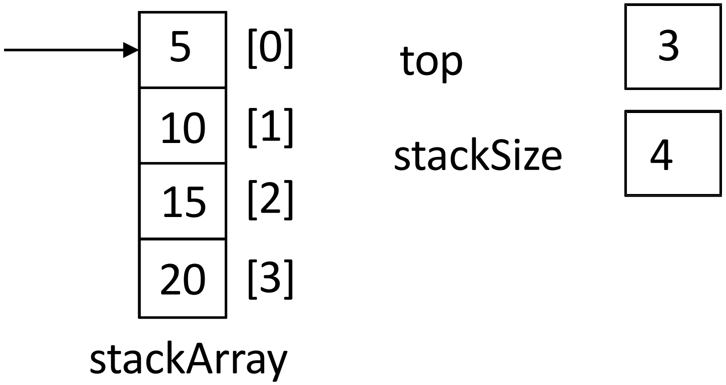
-1

stackSize

4

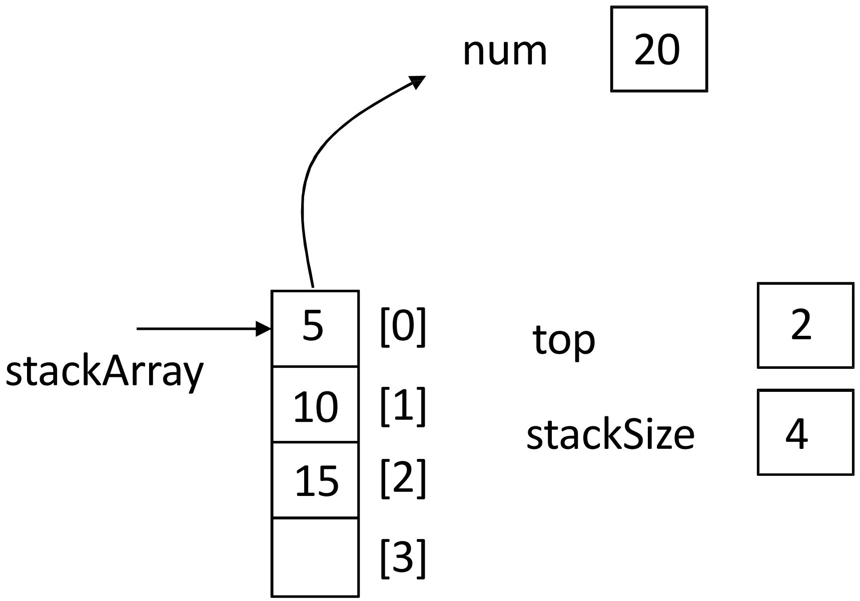
Using Stack

```
void main(void)
{
   IntStack stack(4);
   int catchVar;
   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20); }
```



Using Stack

```
void main(void)
{
   IntStack stack(4);
   int catchVar;
   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20);
   cout << "Popping...\n";
   stack.pop(catchVar);
   cout << catchVar << endl;
   }
```



Using Stack

```
void main(void)
{
   IntStack stack(4); int catchVar;

   cout << "Pushing Integers\n";
   stack.push(5);
   stack.push(10);
   stack.push(15);
   stack.push(20);

   cout << "Popping...\n";
   stack.pop(catchVar);
   cout << catchVar << endl;
   stack.pop(catchVar);
   cout << catchVar << endl;
    stack.pop(catchVar);
   cout << catchVar << endl;
   stack.pop(catchVar);  cout << catchVar << endl;

}
```

Output: Pushing Integers Popping…

20

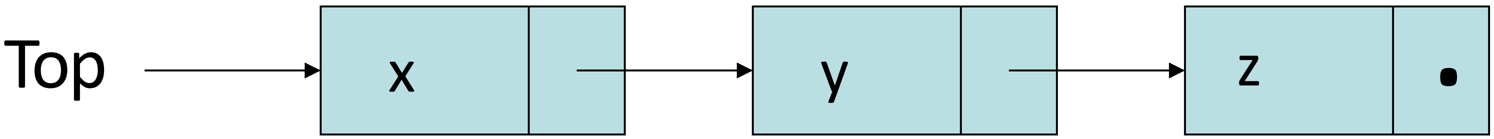
15

10

5

Pointer-based Implementation of Stacks

- Stack can expand or shrink with each push or pop operation
- Push and pop operate only on the header cell, i.e., the first cell of the list



Pointer Implementation – Code

```
class Node {
public:
    int data;
    Node* next; };
class Stack {
    Node* top;
public:
   Stack() : top(nullptr) {}
   void Push(int newElement);
   void Pop(int& removedElement);
   bool IsEmpty(); };
```

Pointer Implementation – Code

- `IsEmpty` function returns true if the stack is empty

```
bool Stack::IsEmpty()
{
    if (top==NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Pointer Implementation – Code

- `Push` function inserts a node at the top/head of the stack

```
void Stack::Push(int newelement) {


node *newptr;

newptr=new node;

    newptr->data=newelement;

    newptr->next=top;

    top=newptr;

}
```

Pointer Implementation – Code

- `Pop` function deletes the node from the top of the stack and returns its data by reference

```
void Stack:Pop(int& returnvalue) {
   if (IsEmpty()) {
      cout<<“underflow error”;
      return;
   }

   tempptr = top;
   returnvalue = top->data;
   top = top->next;

   delete tempptr;
}
```

Algebraic Expressions

- An algebraic expression is combination of operands and operators

- Operand is the object of mathematical operation
  - Quantity that is operated on

- Operator is a symbol that signifies a mathematical or logical operation

Infix, Postfix and Prefix Expressions

- Infix
  - Expressions in which operands surround the operators
  - Example: A+B-C

- Postfix or Reverse Polish Notation (RPN)
  - Operators comes after the operands
  - Example: AB+C-

- Prefix or Polish Notation
  - Operator comes before the operands
  - Example: -+ABC

Example: Conversion From Infix to Postfix

- Infix: A+B*C

- Conversion: Applying the rules of precedence

A+(B*C)	Parentheses for emphasis
A+(BC*)	Convert the multiplication
ABC*+	Postfix Form

Example: Conversion From Infix to Postfix

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

- Conversion: Applying the rules of precedence

  ((AB+)*C-(DE-)) $ (FG+)

  ((AB+C*)-(DE-)) $ (FG+)

  (AB+C*DE--) $ (FG+)

  AB+C*DE- -FG+$

- Exercise: Convert the following to Postfix
  - (A + B) * (C – D)
  - A / B * C – D + E / F / (G + H)

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

Infix, Postfix and Prefix Expressions – Examples

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

Why Do We Need Prefix and Postfix?

- Normally, algebraic expressions are written using Infix notation
  - For example: (3 + 4) × 5 – 6

- Appearance may be misleading, Infix notations are not as simple as they seem
  - Operator precedence
  - Associativity property

- Operators have precedence: Parentheses are often required
  - (3 + 4) × 5 – 6 = 29
  - 3 + 4 × 5 – 6 = 17
  - 3 + 4 × (5 – 6) = –1
  - (3 + 4) × (5 – 6) = –7

Why Do We Need Prefix and Postfix?

- Infix Expression is Hard To Parse and difficult to evaluate

- Postfix and prefix   do not rely on operator priority and are easier to parse
  - No ambiguity and no brackets are required

- Many compilers first translate algebraic expressions into some form of postfix notation
  - Afterwards translate this postfix expression into machine code

```
MOVE.L #$2A, D1          ;  Load    42 into Register D1
MOVE.L #$100, D2         ;  Load 256 into Register D2
ADD D2, D1               ;  Add D2 into D1
```

Conversion of Infix Expression to Postfix

- Precedence function
  - prcd(op1, op2)
  - op1and op2are characters representing operators

- Precedence function returns TRUE
  - If op1has precedence over op2
  - Otherwise function returns FALSE

- Examples
  - prcd(‘*’,’+’) returns TRUE
  - prcd(‘+’,’+’) returns TRUE
  - prcd(‘+’,’*’) returns FALSE

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
  symb = next input character;
  if   (symb is an operand)
              add  symb to the postfix string
  else  {
             while  (!empty(opstk) &&
             prcd(stacktop(opstk),symb) ) {
             topsymb = pop(opstk);
             add  topsymb to the postfix string; }
  push(opstk, symb); } }

while  (!empty(opstk) ) {
  topsymb = pop(opstk);
  add  topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
  symb = next input character;
  if  (symb is an operand)
              add  symb to the postfix string
  else  {
             while  (!empty(opstk) &&
             prcd(stacktop(opstk),symb) ) {
             topsymb = pop(opstk);
             add  topsymb to the postfix string; }
  push(opstk, symb); } }

while  (!empty(opstk) ) {
  topsymb = pop(opstk);
  add  topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
  symb = next input character;
  if  (symb is an operand)
              add  symb to the postfix string
  else  {
             while  (!empty(opstk) &&
             prcd(stacktop(opstk),symb) ) {
             topsymb = pop(opstk);
             add  topsymb to the postfix string; }
  push(opstk, symb); } }

while  (!empty(opstk) ) {
  topsymb = pop(opstk);
  add  topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
          add symb to the postfix string
  else {
          while (!empty(opstk) &&
          prcd(stacktop(opstk),symb) ) {
          topsymb = pop(opstk);
          add topsymb to the postfix string; }
  push(opstk, symb); } }

while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+
B	AB	+

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
             add symb to the postfix string
  else {
            while (!empty(opstk) &&
            prcd(stacktop(opstk),symb) ) {
            topsymb = pop(opstk);
            add topsymb to the postfix string; }
  push(opstk, symb); } }

while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+
B	AB	+
*	AB	+ *

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
            add symb to the postfix string
  else {
            while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
            topsymb = pop(opstk);
            add topsymb to the postfix string; }
  push(opstk, symb); } }

while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
  symb = next input character;
  if   (symb is an operand)
                add  symb to the postfix string
  else  {
              while  (!empty(opstk) &&
              prcd(stacktop(opstk),symb) ) {
              topsymb = pop(opstk);
              add  topsymb to the postfix string; }
  push(opstk, symb); } }

while  (!empty(opstk) ) {
  topsymb = pop(opstk);
  add  topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC*	+

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
  symb = next input character;
  if   (symb is an operand)
              add  symb to the postfix string
  else  {
             while  (!empty(opstk) &&
             prcd(stacktop(opstk),symb) ) {
             topsymb = pop(opstk);
             add  topsymb to the postfix string; }
  push(opstk, symb); } }

while  (!empty(opstk) ) {
  topsymb = pop(opstk);
  add  topsymb to the postfix string;

}
```

EXAMPLE: A+B*C

symb	Postfix string	opstk
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC*	+
	ABC*+	

What If Expression Contains Parenthesis?

- Precedence function prcd(op1, op2) has to be modified

– prcd(‘(‘ , op) = FALSE	For any operator op
– prcd(op, ‘(‘) = FALSE	For any operator opother than ‘)’
– prcd(op, ‘)‘) = TRUE	For any operator opother than ‘(‘
– prcd(‘)‘ ,op) = undef	For any operator op(an error)

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
            add symb to the postfix string
  else {
           while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
             topsymb = pop(opstk);
             add topsymb to the postfix string; }
         if ( empty(opstk)|| symb != ‘)’ )
           push(opstk, symb);
        else
           topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
   symb = next input character;
   if   (symb is an operand)
                 add  symb to the postfix string
   else  {
                while   (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
                   topsymb = pop(opstk);
                   add  topsymb to the postfix string; }
            if   ( empty(opstk)|| symb != ‘)’ )
               push(opstk, symb);
            else
               topsymb = pop(opstk); }}
  while   (!empty(opstk) ) {
   topsymb = pop(opstk);
   add  topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
            add symb to the postfix string
  else {
            while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
              topsymb = pop(opstk);
              add topsymb to the postfix string; }
          if ( empty(opstk)|| symb != ‘)’ )
            push(opstk, symb);
          else
            topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
              add symb to the postfix string
  else {
             while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
               topsymb = pop(opstk);
               add topsymb to the postfix string; }
         if ( empty(opstk)|| symb != ‘)’ )
             push(opstk, symb);
         else
             topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while   (not end of input) {
   symb = next input character;
   if    (symb is an operand)
                    add   symb to the postfix string
   else  {
                 while   (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
                     topsymb = pop(opstk);
                     add   topsymb to the postfix string; }
             if    ( empty(opstk)|| symb != ')' )
                 push(opstk, symb);
            else
                 topsymb = pop(opstk); }}
  while   (!empty(opstk) ) {
   topsymb = pop(opstk);
   add   topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
              add symb to the postfix string
  else {
             while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
               topsymb = pop(opstk);
               add topsymb to the postfix string; }
          if ( empty(opstk)|| symb != ')' )
             push(opstk, symb);
          else
             topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
            add symb to the postfix string
  else {
           while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
              topsymb = pop(opstk);
              add topsymb to the postfix string; }
        if ( empty(opstk)|| symb != ‘)’ )
           push(opstk, symb);
        else
           topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	
*	AB+	*

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while (not end of input) {
  symb = next input character;
  if (symb is an operand)
            add symb to the postfix string
  else {
            while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
              topsymb = pop(opstk);
              add topsymb to the postfix string; }
        if ( empty(opstk)|| symb != ‘)’ )
            push(opstk, symb);
        else
            topsymb = pop(opstk); }}
 while (!empty(opstk) ) {
  topsymb = pop(opstk);
  add topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	
*	AB+	*
C	AB+C	*

Convert Infix to Postfix

ALGORITHM

```
opstk = the empty stack;
while  (not end of input) {
   symb = next input character;
   if   (symb is an operand)
              add  symb to the postfix string
   else  {
             while  (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
                topsymb = pop(opstk);
                add  topsymb to the postfix string; }
          if   ( empty(opstk)|| symb != ‘)’ )
             push(opstk, symb);
          else
             topsymb = pop(opstk); }}
  while  (!empty(opstk) ) {
      topsymb = pop(opstk);
      add  topsymb to the postfix string; }
```

EXAMPLE: (A+B)*C

symb	Postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB+	
*	AB+	*
C	AB+C	*
	AB+C*	

Conversion of Infix Expression to Postfix – Rules

- Token is an operand
  - Append it to the end of postfix string
- Token is a left parenthesis
  - Push it on the opstk
- Token is a right parenthesis
  - Pop the opstk until the corresponding left parenthesis is removed
  - Append each operator to the end of the postfix string
- Token is an operator, *, /, +, or –
  - Push it on the opstk
  - First remove any operators already on the opstk that have higher or equal precedence and append them to the postfix string
- Input expression has been completely processed
  - Any operators still on the opstk can be removed and appended to the end of the postfix string

Any Question So Far?