Lecture No.6

Data Structures & Algorithms

STACK

Stacks

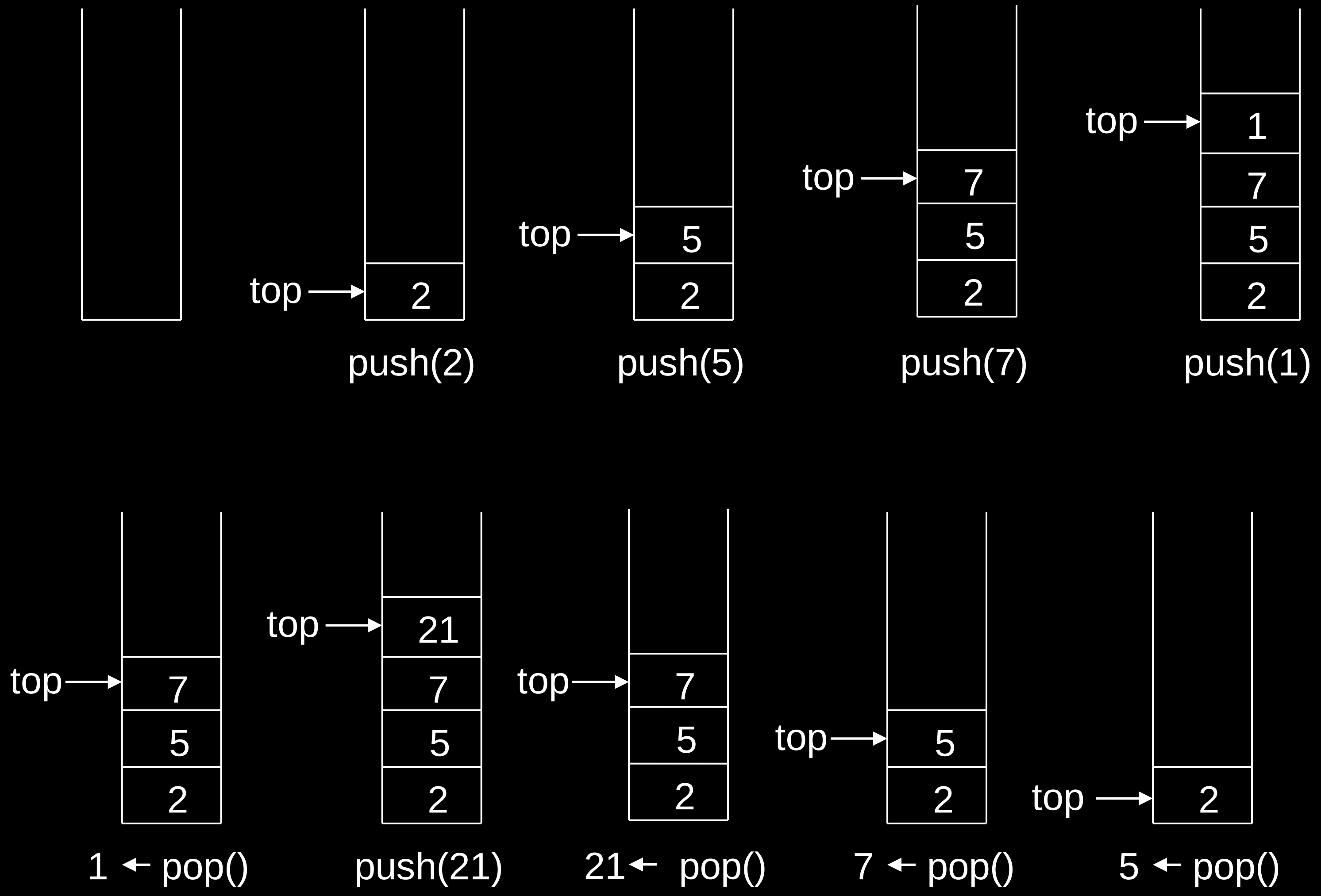
The fundamental operations involved in a stack are “push” and “pop”.

- *Push(x)*: insert X as the top element on the stack
- *Pop()*: removes an element from top of the stack and returns it

Other operations include

- *Top():* Returns top element from top without removing it from the stack
- *Getsize()*: tracks the number of elements in the stack
- *isEmpty()*: Checks if the Stack is empty, must call before *pop()*
- *IsFull()*: checks if the stack is full, must call before *push(x)*

Stack Operations



top → 2
push(2)

top → 5, 2
push(5)

top → 7, 5, 2
push(7)

top → 1, 7, 5, 2
push(1)

top → 7, 5, 2
1 ← pop()

top → 21, 7, 5, 2
push(21)

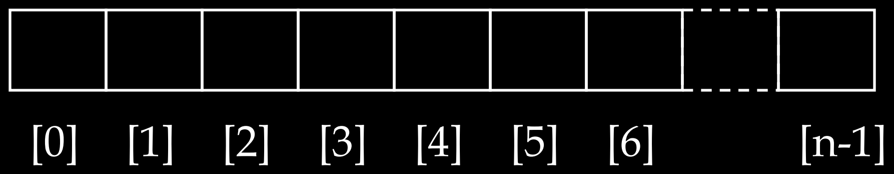
top → 7, 5, 2
21 ← pop()

top → 5, 2
7 ← pop()

top → 2
5 ← pop()

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.



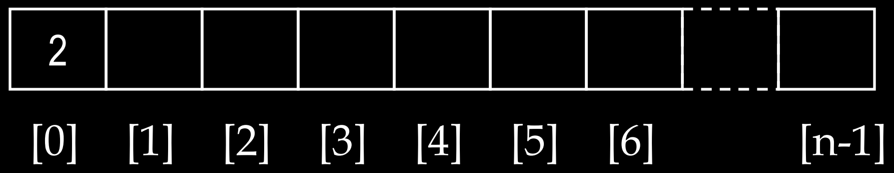
Top = 0

Size = 0

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

Push(2);



2							----	
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

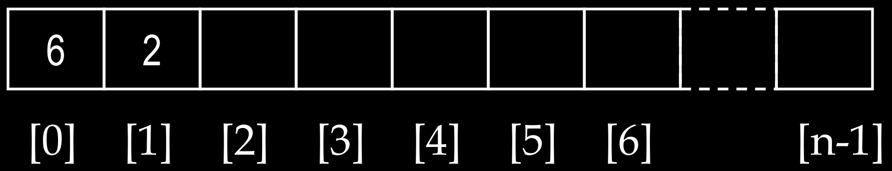
Top = 0

Size = 1

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

Push(6);



6	2							
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

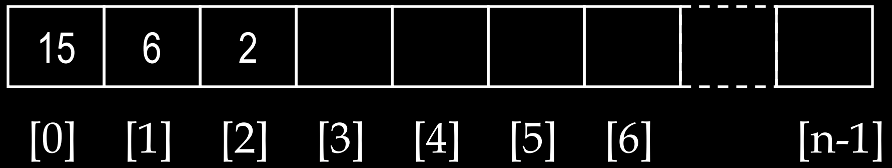
Top = 0

Size = 2

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

Push(15);

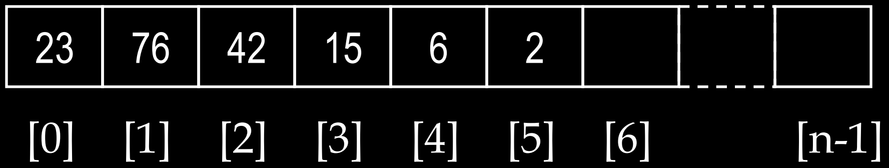


Top = 0

Size = 3

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.



23	76	42	15	6	2			
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

Top = 0

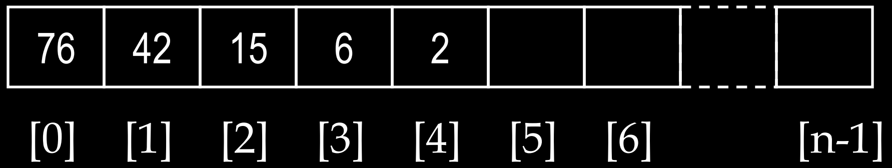
Size = 6

a =

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

a = pop();



76	42	15	6	2				
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

Top = 0

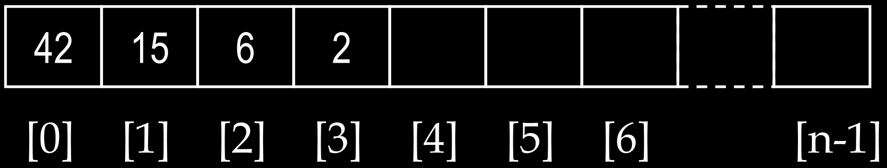
Size = 5

a = 23

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

a = pop();



42	15	6	2					
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

Top = 0

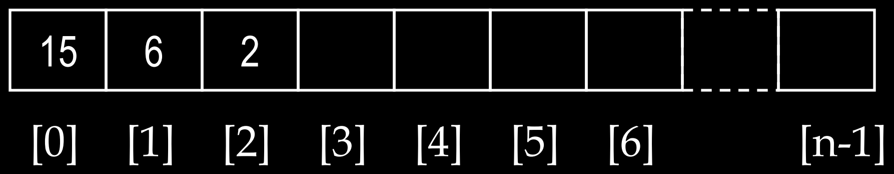
Size = 4

a = 76

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the right.

a = pop();



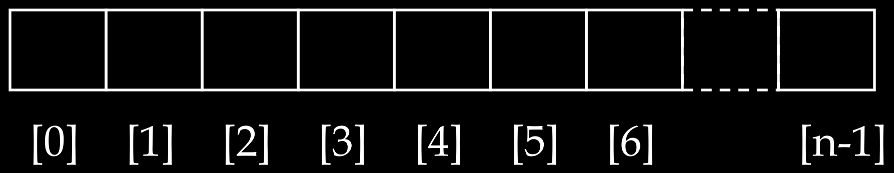
Top = 0

Size = 3

a = 42

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.



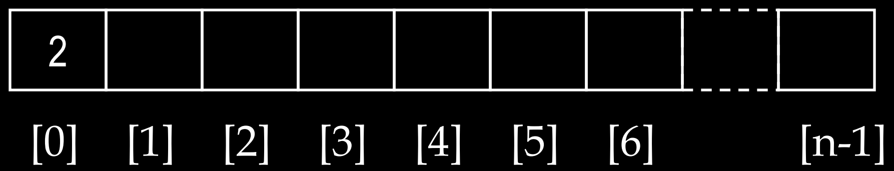
current = -1 // current top

Size = 0

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

Push(2);



2							…	
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

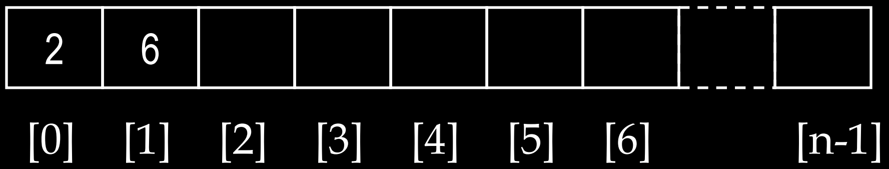
current = 0 // current top

Size = 1

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

Push(6);



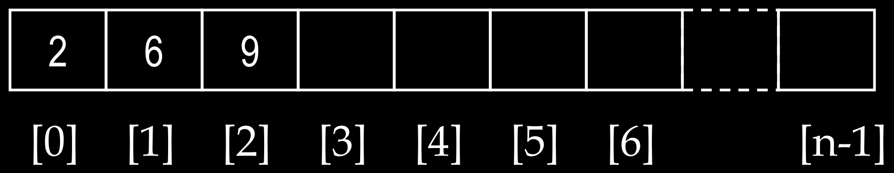
current = 1 // current top

Size = 1

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

Push(9);



2	6	9					- - - -	
[0]	[1]	[2]	[3]	[4]	[5]	[6]		[n-1]

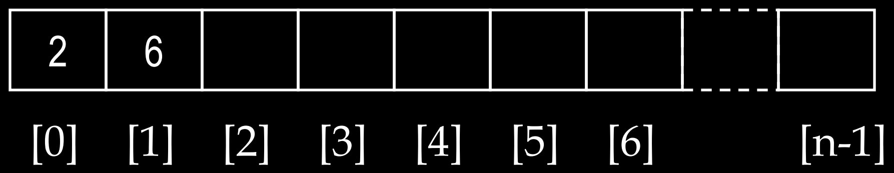
current = 2 // current top

Size = 1

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

a = pop();



current = 1 // current top

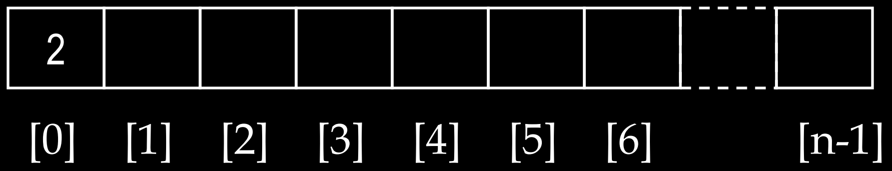
Size = 1

a = 9

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

a = pop();



current = 0 // current top

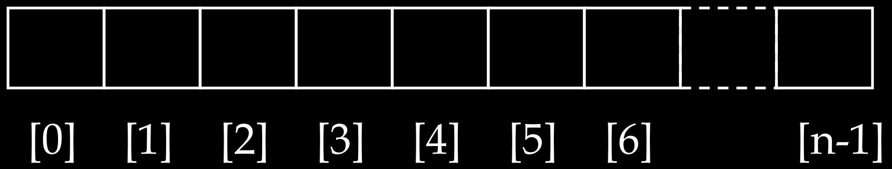
Size = 1

a = 6

Stack Implementation: Array

- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

a = pop();



current = -1 // current top

Size = 1

a = 2

Stack Implementation: Array

```
class stack
{
private:
   int current, size, maxsize;
   int *A;

public:
stack(int x)
{
   maxsize=x;
   A=new int[maxsize];
   current=-1;
   size=0;
}
```

Stack Implementation: Array

```
int pop()
{
    --size;
    return A[current--];
}

void push(int x)
{
    A[++current] = x;
    ++size;
}

int top()
{
    return A[current];
}

int IsEmpty()
{
    return ( current == -1 );
}
```

Stack Implementation: Array

```
int IsFull()
{
    return ( current == maxsize-1);
}

void print()
{
    for (int i=current; i>=0; i--);
     cout << (this->A[i]) << " “;
    cout << endl;
}

};
```

Array Stack Class Implementation

```
#include<iostream.h>


void main()
{
stack S(5);
int a=0;

while (S.IsFull()==0)
      { cout << "Enter value to push, 999 to terminate: ";
   cin >> a; if (a==999) break;
   S.push(a);
      }
if (S.IsEmpty()==0)
     { cout << "Stack is: ";
       S.print();
      }
```

Array Stack Class Implementation

```
int choice=1;
while (!((choice>2)||(choice<1)))
 {
   cout << "1. push     2. pop    3. Exit   :Enter your choice  ";
   cin  >> choice;
   if (choice==1)
   {if (S.IsFull()==0)
       { cout << "\nEnter No. to push: "; cin >> a; S.push(a);
         cout << "\nStack is: "; S.print();}
    else
         cout << "\nStack is full";

   }
   if (choice==2)
   {if (S.IsEmpty()==0)
       { a=S.pop();cout << "\n" << a << " has been poped from stack";
         cout << "\nStack is: "; S.print();}
    else
         cout << "\nStack is empty";
   }
  }
}
```

Stack Using Linked List

- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.

- As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

Stack Using Linked List

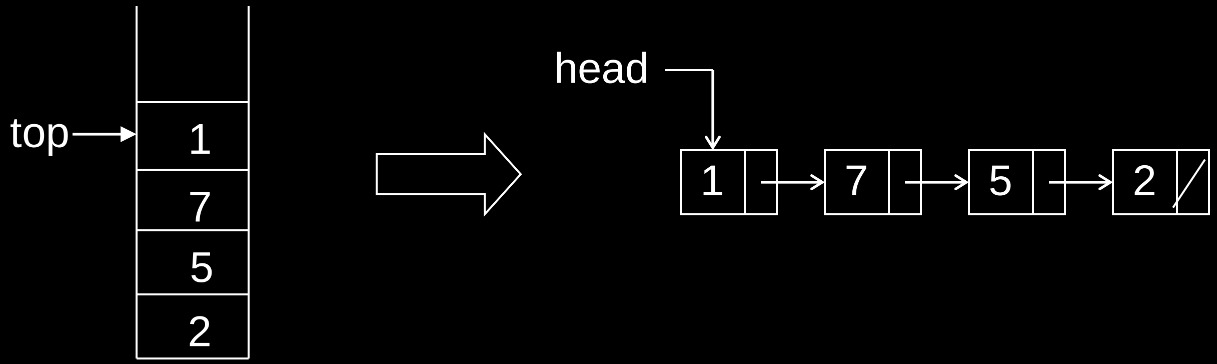
- For a singly-linked list, insert at start or end takes constant time using the head and current pointers respectively.

- Removing an element at the start is constant time but removal at the end required traversing the list to the node one before the last.

- Make sense to place stack elements at the start of the list because insert and removal are constant time.

Stack Using Linked List

- No need for the current pointer; head is enough.

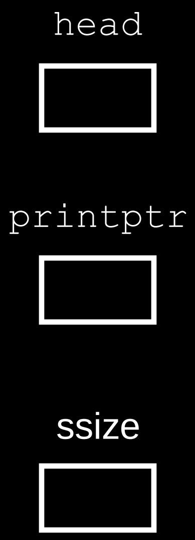


Stack Class using Linked List

```
class Node {
private:
   int data;
   Node *nextNode;

public:
   int get() { return data; }
   void set(int data) { this->data = data; }

   Node *getNext() { return nextNode; }
   void setNext(Node *nextNode)
                    { this->nextNode = nextNode; }
};
```

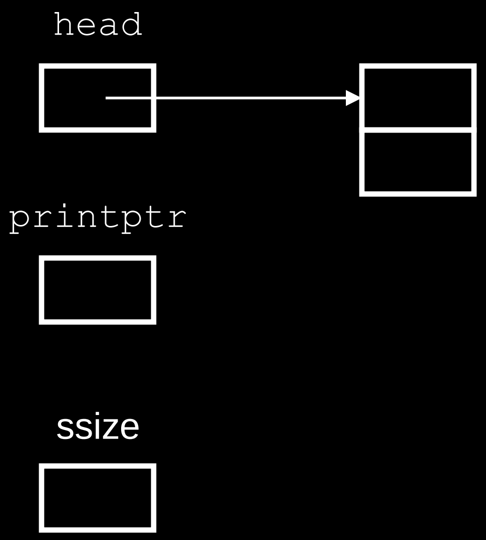
Stack Class using Linked List

```
class stack {

private:
   int ssize;
   Node *head;
   Node *printptr;

public:
   // Constructor
   stack() {
       head = new Node();
       head->setNext(NULL);
       printptr = head;
       ssize = 0;
   };
```



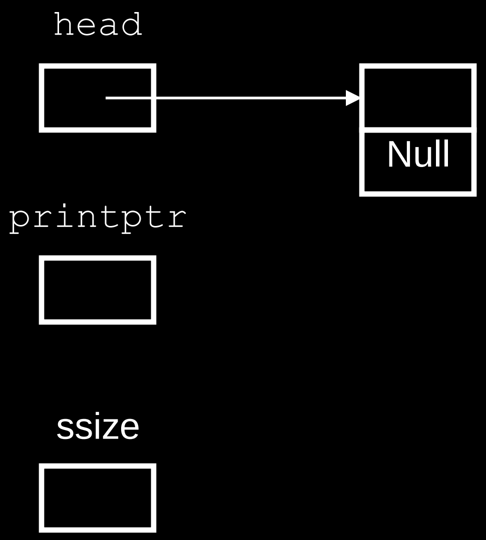
Stack Class using Linked List

```
class stack {

private:
   int ssize;
   Node *head;
   Node *printptr;

public:
   // Constructor
   stack() {
       head = new Node();
       head->setNext(NULL);
       printptr = head;
       ssize = 0;
   };
```



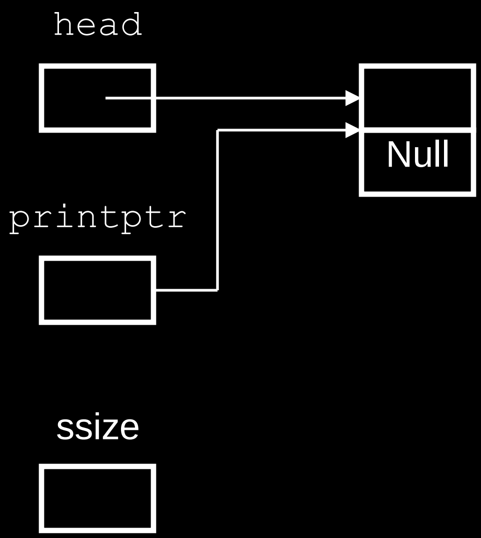
Stack Class using Linked List

```
class stack {

private:
   int ssize;
   Node *head;
   Node *printptr;

public:
   // Constructor
   stack() {
        head = new Node();
        head->setNext(NULL);
        printptr = head;
        ssize = 0;
   };
```



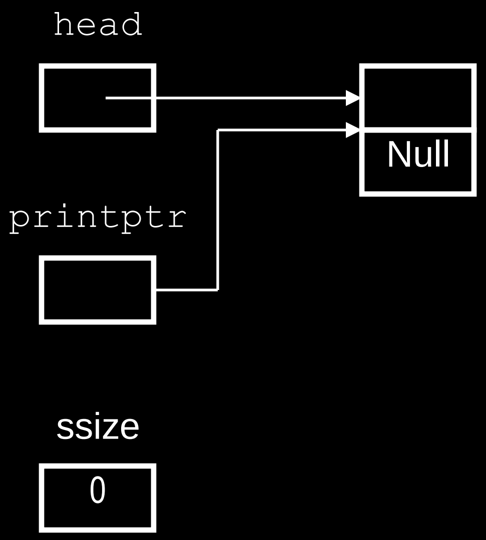
Stack Class using Linked List

```
class stack {

private:
   int ssize;
   Node *head;
   Node *printptr;

public:
   // Constructor
   stack() {
        head = new Node();
        head->setNext(NULL);
        printptr = head;
        ssize = 0;
   };
```



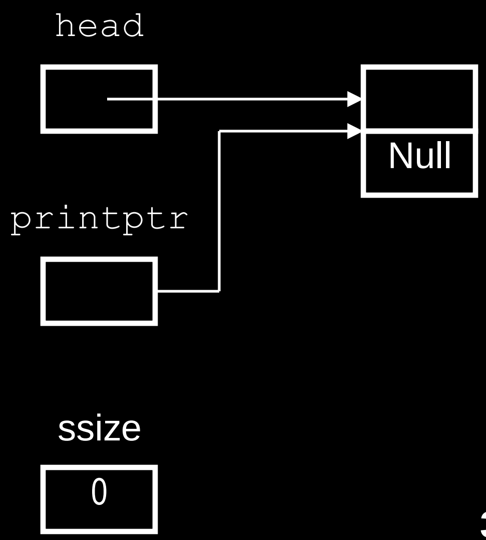
Stack Class using Linked List

```
class stack {

private:
   int ssize;
   Node *head;
   Node *printptr;

public:
   // Constructor
   stack() {
       head = new Node();
       head->setNext(NULL);
       printptr = head;
       ssize = 0;
   };
```



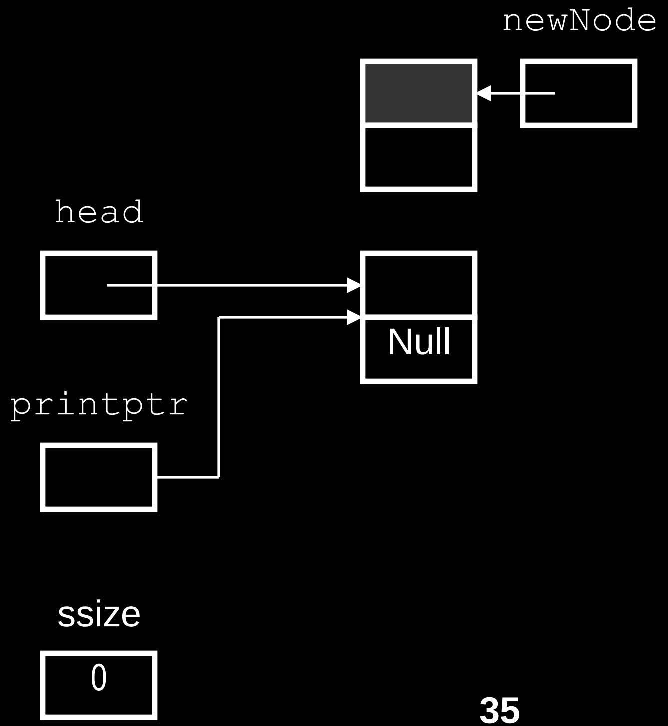
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



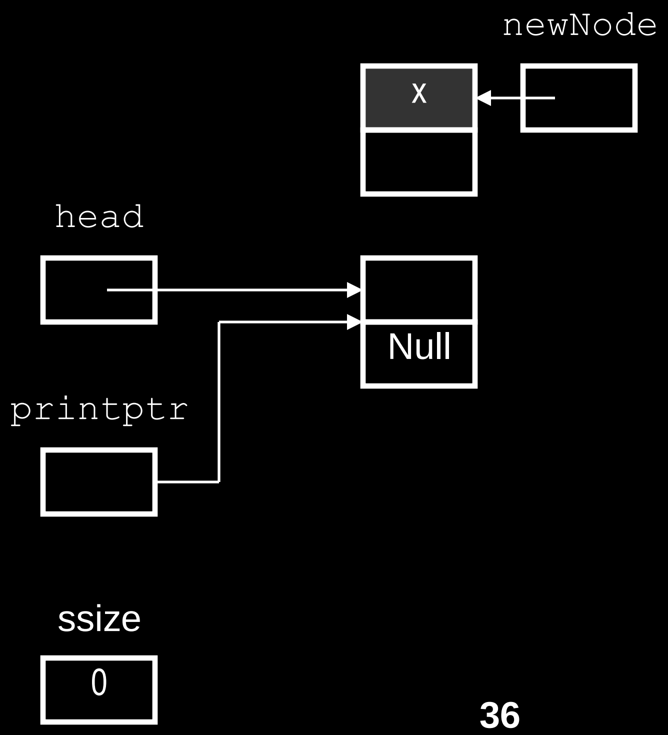
Stack Class using Linked List

```
void push(int x)
{
     Node* newNode = new Node();
     newNode->set(x);
     newNode->setNext(head);
     head = newNode;
     ssize++;
}
```



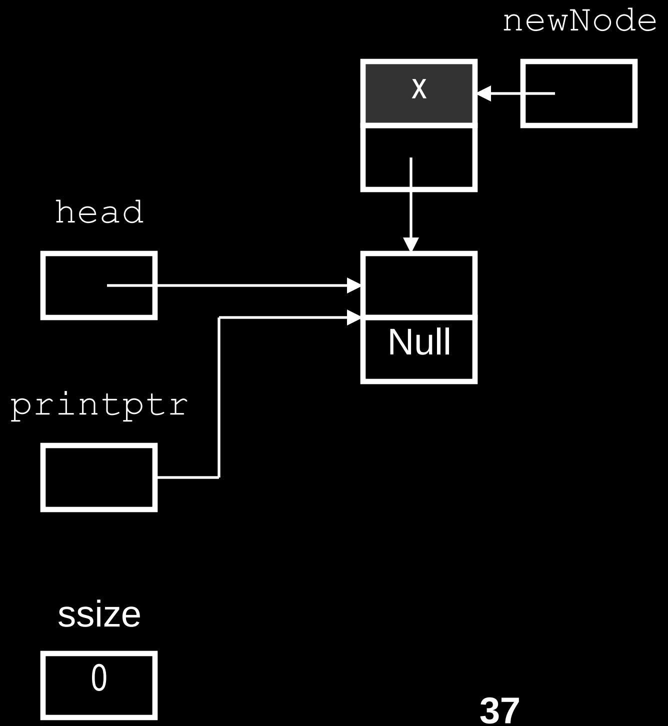
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



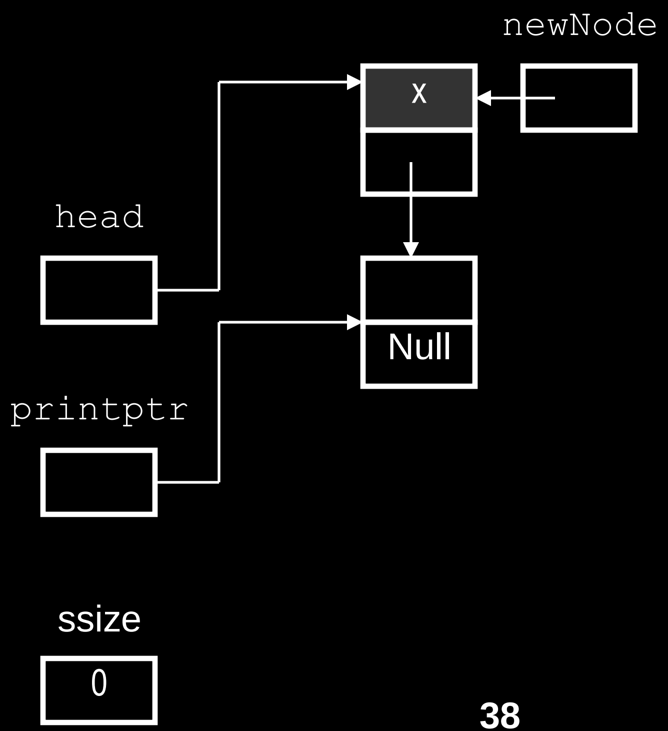
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



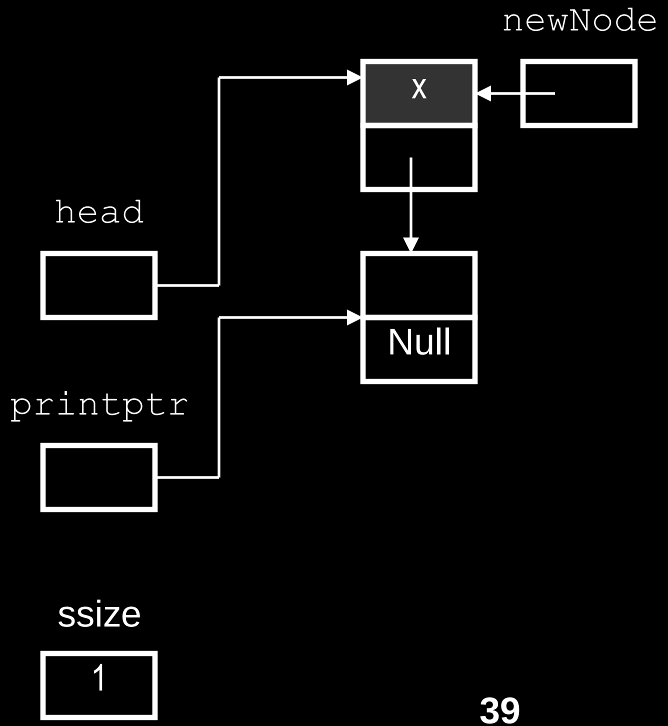
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



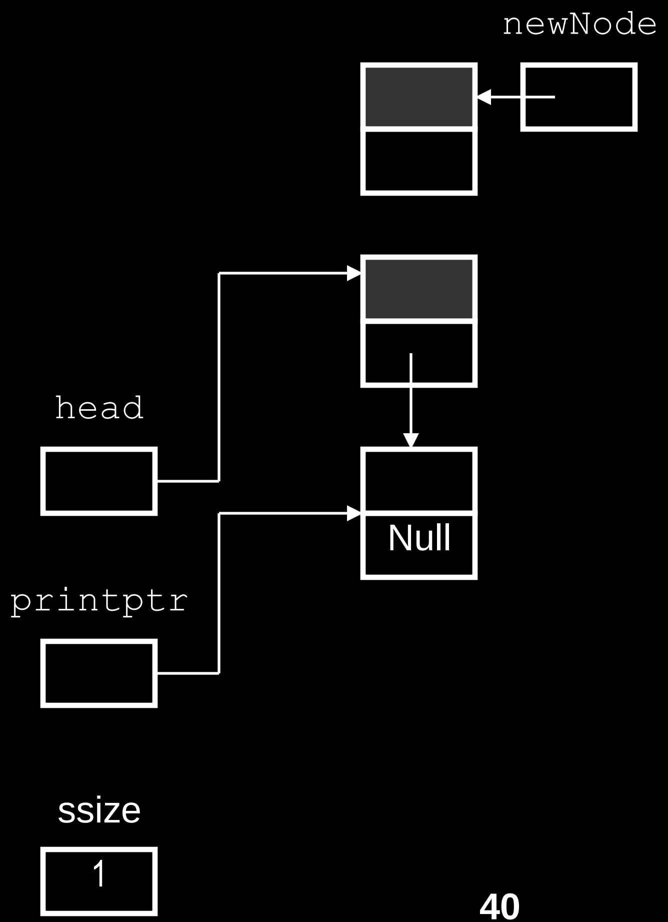
Stack Class using Linked List

```
void push(int x)
{
     Node* newNode = new Node();
     newNode->set(x);
     newNode->setNext(head);
     head = newNode;
     ssize++;
}
```



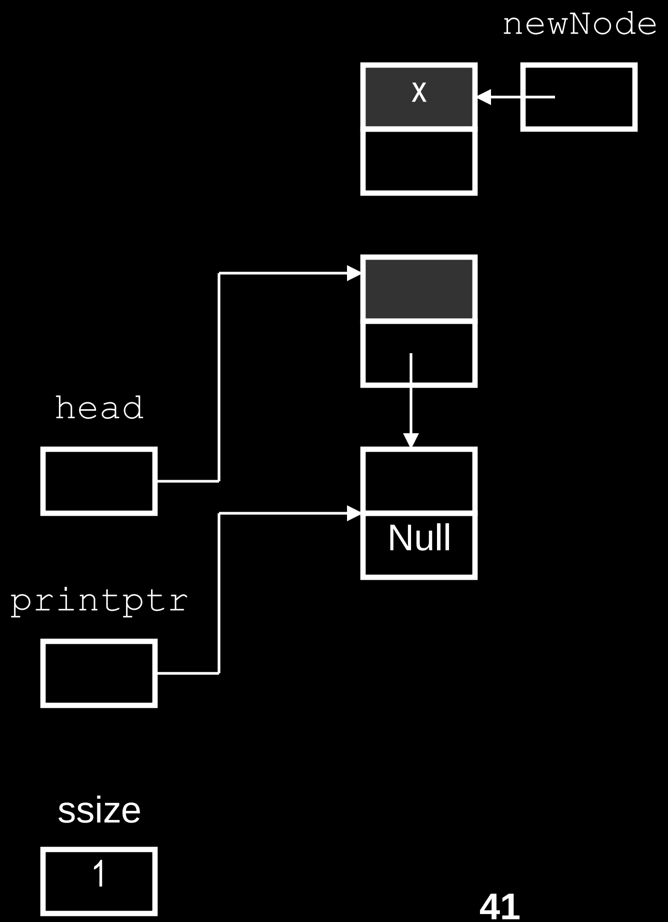
Stack Class using Linked List

```
void push(int x)
{
     Node* newNode = new Node();
     newNode->set(x);
     newNode->setNext(head);
     head = newNode;
     ssize++;
}
```



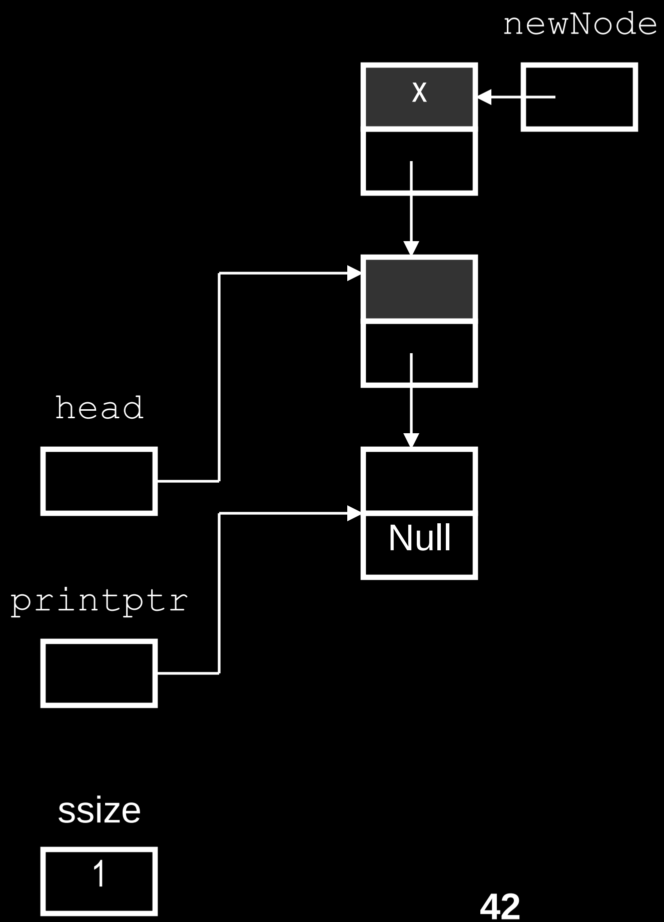
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



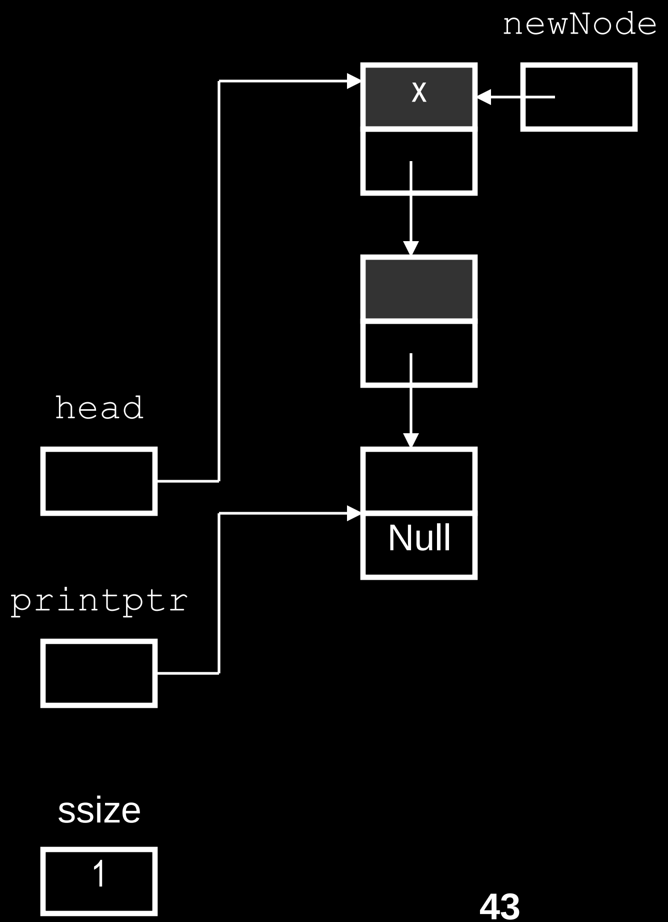
Stack Class using Linked List

```
void push(int x)
{
     Node* newNode = new Node();
     newNode->set(x);
     newNode->setNext(head);
     head = newNode;
     ssize++;
}
```



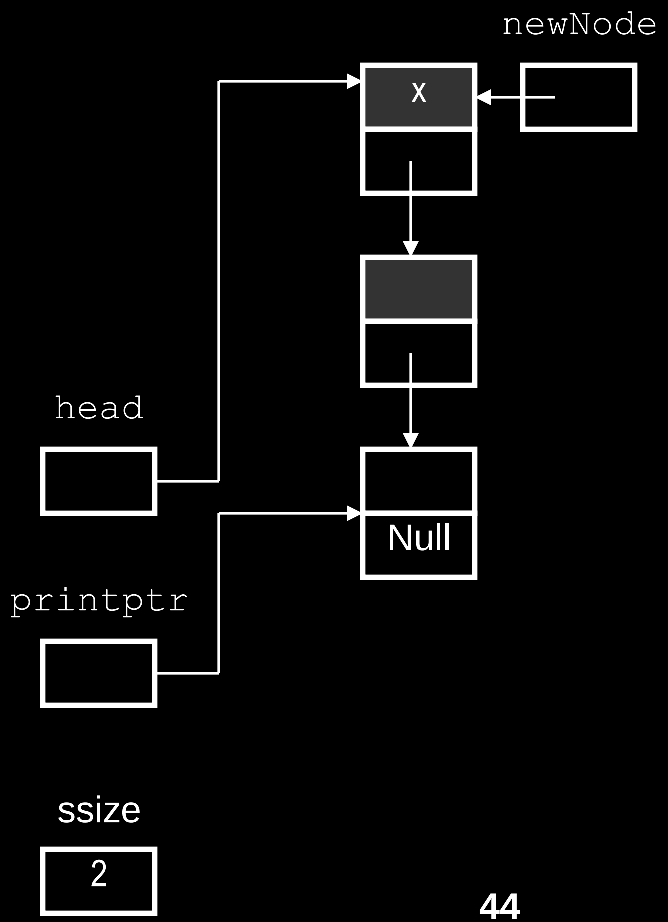
Stack Class using Linked List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
    ssize++;
}
```



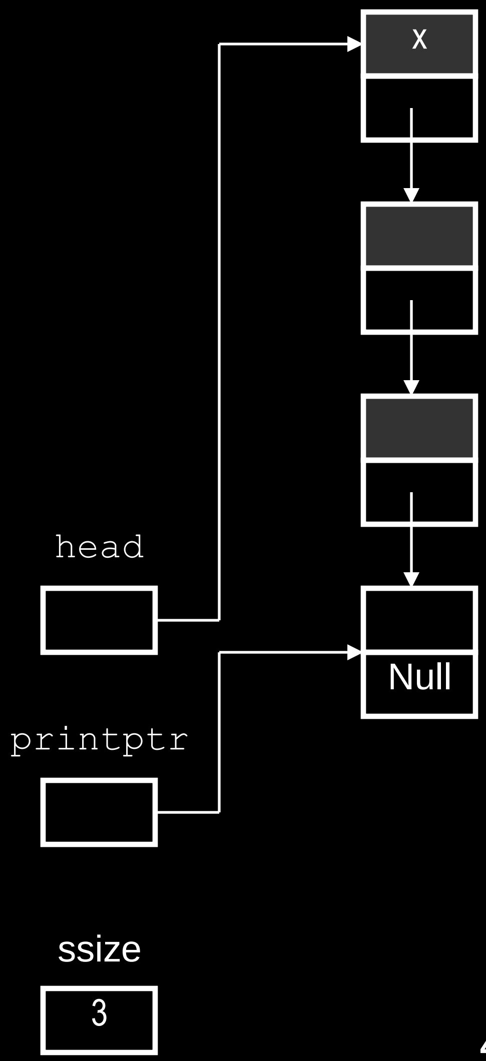
Stack Class using Linked List

```
void push(int x)
{
     Node* newNode = new Node();
     newNode->set(x);
     newNode->setNext(head);
     head = newNode;
     ssize++;
}
```



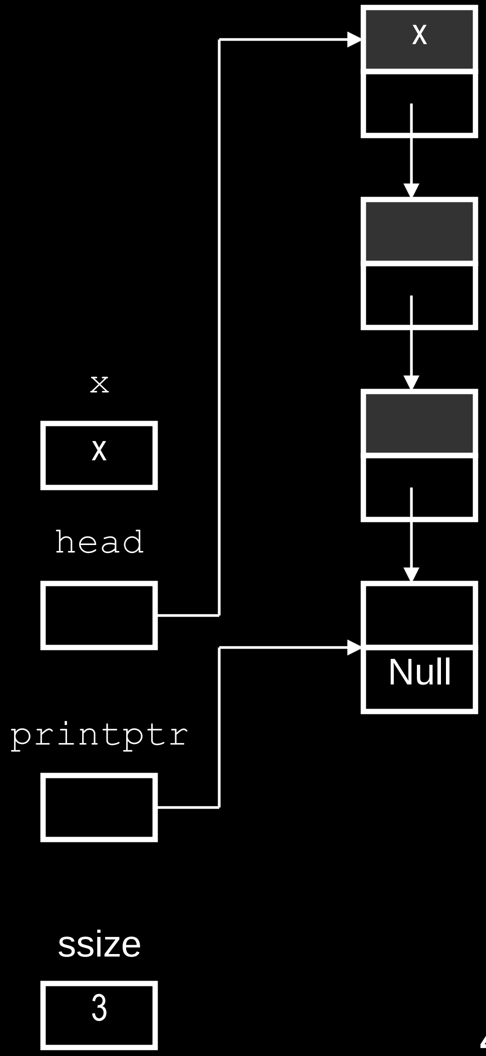
Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```



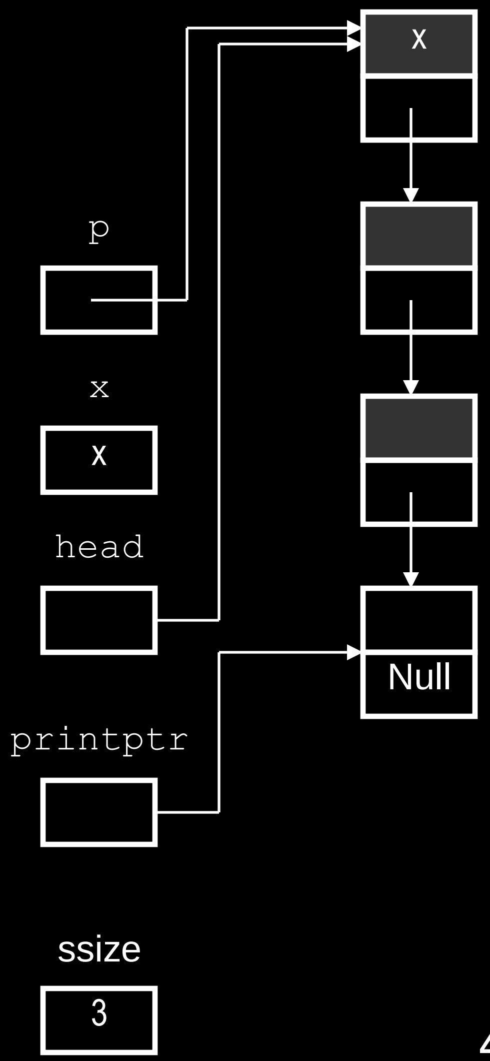
Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```



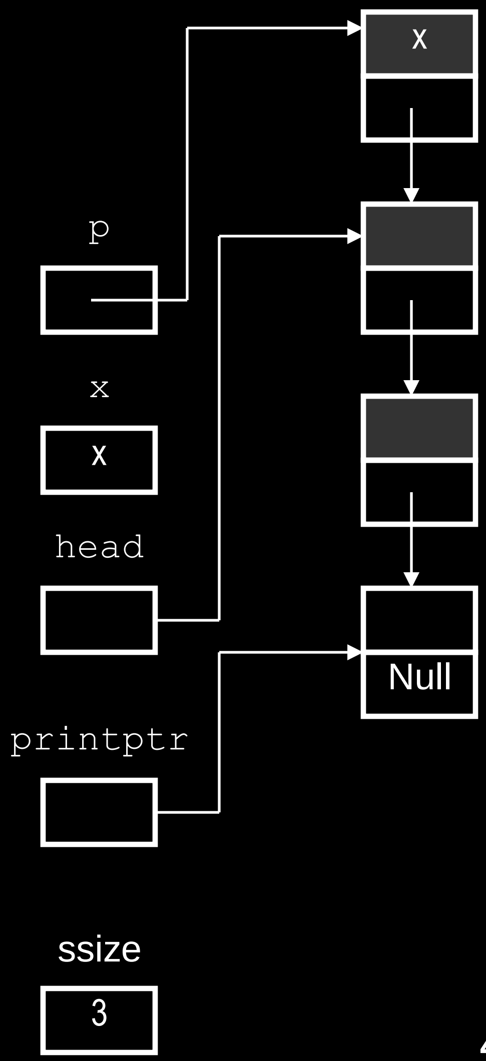
Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```



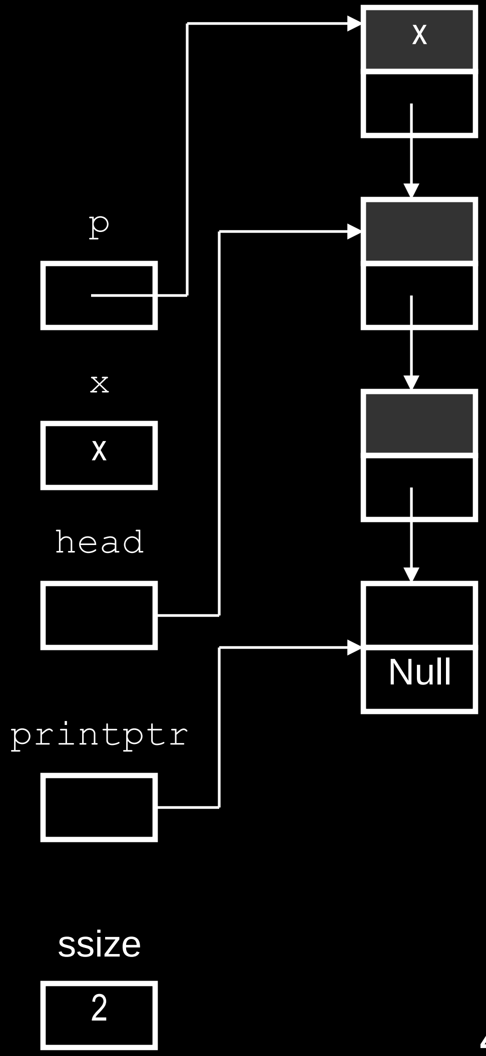
Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```



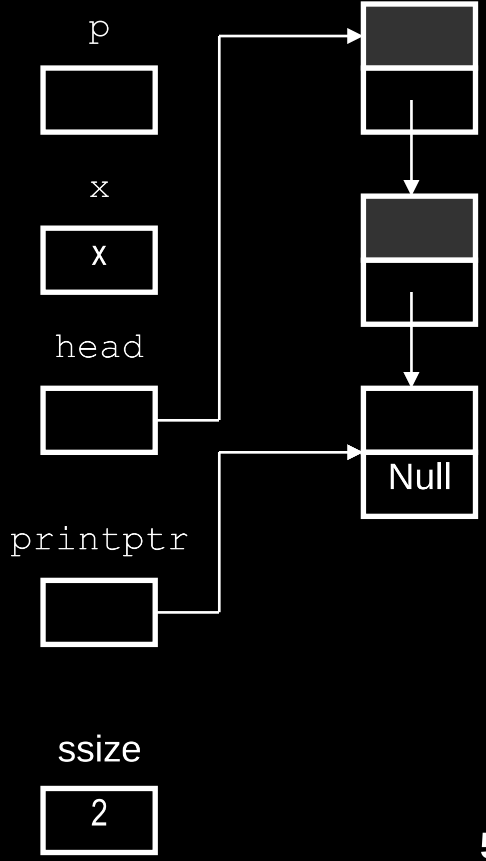
Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```



Stack Class using Linked List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    ssize--;
    delete p;
    return x;
}
```

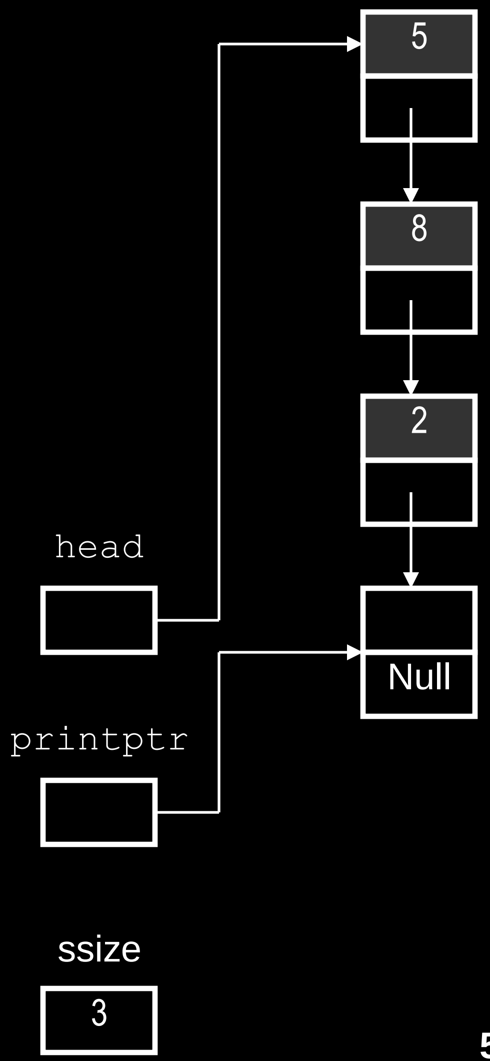


Stack Class using Linked List

```
int top()
{
    return head->get();
}

int IsEmpty()
{
    return ( head->getNext() == NULL );
}

int size()
{
    return ssize;
}
```

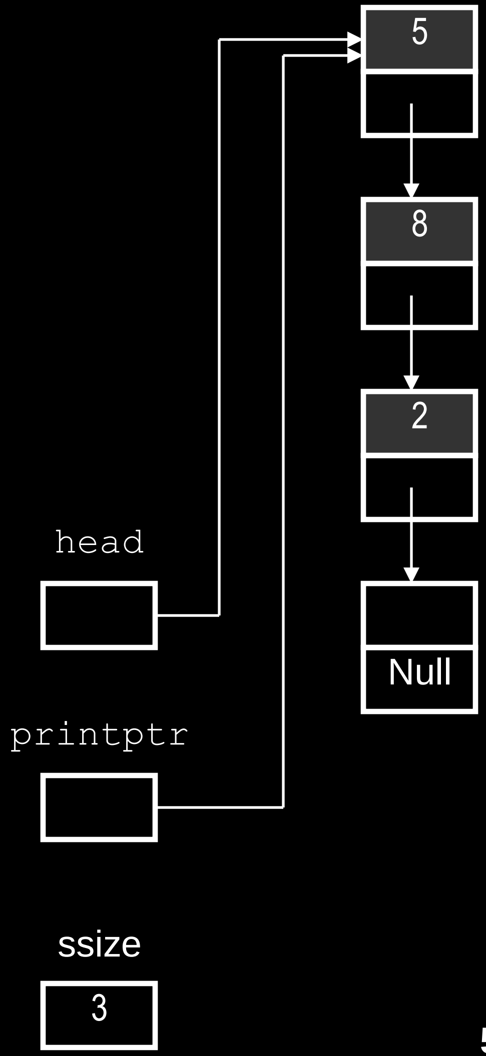
Stack Class using Linked List

```
void print()
{
   printptr=head;
   while ((printptr->getNext())!=NULL)
      {
      cout << printptr->get() << " ";
      printptr=printptr->getNext();
      }
}
};
```



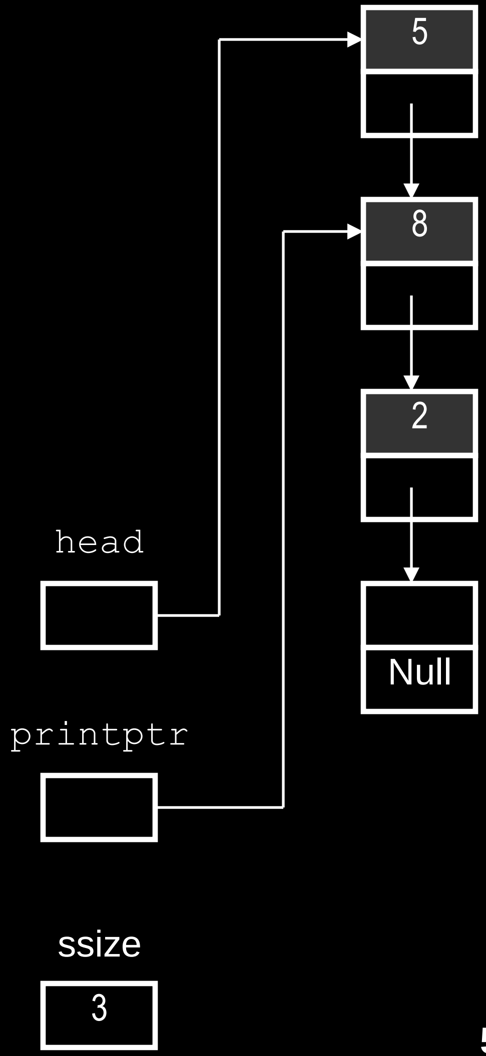
Stack Class using Linked List

```
void print()
{
   printptr=head;
   while ((printptr->getNext())!=NULL)
      {
      cout << printptr->get() << " ";
      printptr=printptr->getNext();
      }
}
};
```



Stack Class using Linked List

```
void print()
{
  printptr=head;
  while ((printptr->getNext())!=NULL)
    {
    cout << printptr->get() << " ";
    printptr=printptr->getNext();
    }
}
};
```



Stack Class using Linked List: Implementation

```
#include<iostream.h>

void main()
{
 stack S;
 int a = 0;
while (a!=999)
 {
 cout << "Enter value to push, 999 to terminate: ";
 cin >> a;
 if (a!=999) S.push(a);
 }
if (S.IsEmpty()==0)
     { cout << "Stack is: ";
       S.print();
      }
```

Stack Class using Linked List: Implementation

```
int choice=1;
while (!((choice>2)||(choice<1)))
 {
   cout << "\n1. push    2. pop   3. Exit  :Enter your choice  ";
   cin  >> choice;
   if (choice==1)
   {     cout << "\nEnter No. to push: "; cin >> a; S.push(a);
         cout << "\nStack is: "; S.print();
   }

   if (choice==2)
   {if (S.IsEmpty()==0)
     { a =S.pop();cout << "\n" << a << " has been poped from stack";
         cout << "\nStack is: "; S.print();}
    else
         cout << "\nStack is empty";
   }
  }
}
```

Stack: Array or List

- Since both implementations support stack operations in constant time, any reason to choose one over the other?
- Allocating and deallocating memory for list nodes does take more time than preallocated array.
- List uses only as much memory as required by the nodes; array requires allocation ahead of time.
- List pointers (head, next) require extra memory.
- Array has an upper limit; List is limited by dynamic memory allocation.

Use of Stack

- Example of use: prefix, infix, postfix expressions.

- Consider the expression A+B: we think of applying the *operator* “+” to the *operands* A and B.

- “+” is termed a *binary operator*: it takes two operands.

- Writing the sum as A+B is called the *infix* form of the expression.

Prefix, Infix, Postfix

- Two other ways of writing the expression are

  + A B *prefix*
  A B + *postfix*

- The prefixes “pre” and “post” refer to the position of the operator with respect to the two operands.

Prefix, Infix, Postfix

- Consider the infix expression
  A + B * C

- We “know” that multiplication is done before addition.

- The expression is interpreted as
  A + (B * C)

- Multiplication has *precedence* over addition.

- Conversion to postfix
  A + (B * C)   infix form

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)　　　infix form

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition
A B C * +	postfix form

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C infix form

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication
A B + C *	postfix form

Precedence of Operations

- The four binary operators are: addition, subtraction, multiplication and division.

- parenthesis can also present in infix form

- The order of precedence is (highest to lowest)

• Parenthesis	(,)
• Multiplication/division	* , /
• Addition/subtraction	+ , -

Precedence of Operators

- For operators of same precedence, the left-to-right rule applies:

  A+B+C means (A+B)+C.

Infix to Postfix

For operators of same precedence, the left-to-right rule applies:

A+B+C means (A+B)+C.

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *

- Note that the postfix form an expression does not require parenthesis.

Infix to Postfix – Developing Algorithm - i

Let we have Two Arrays – `infixString[]` and `postfixString[]`

`infixString[]` contains the infix expression (string) to be converted to postfix

`postfixString[]` will hold the postfix notation

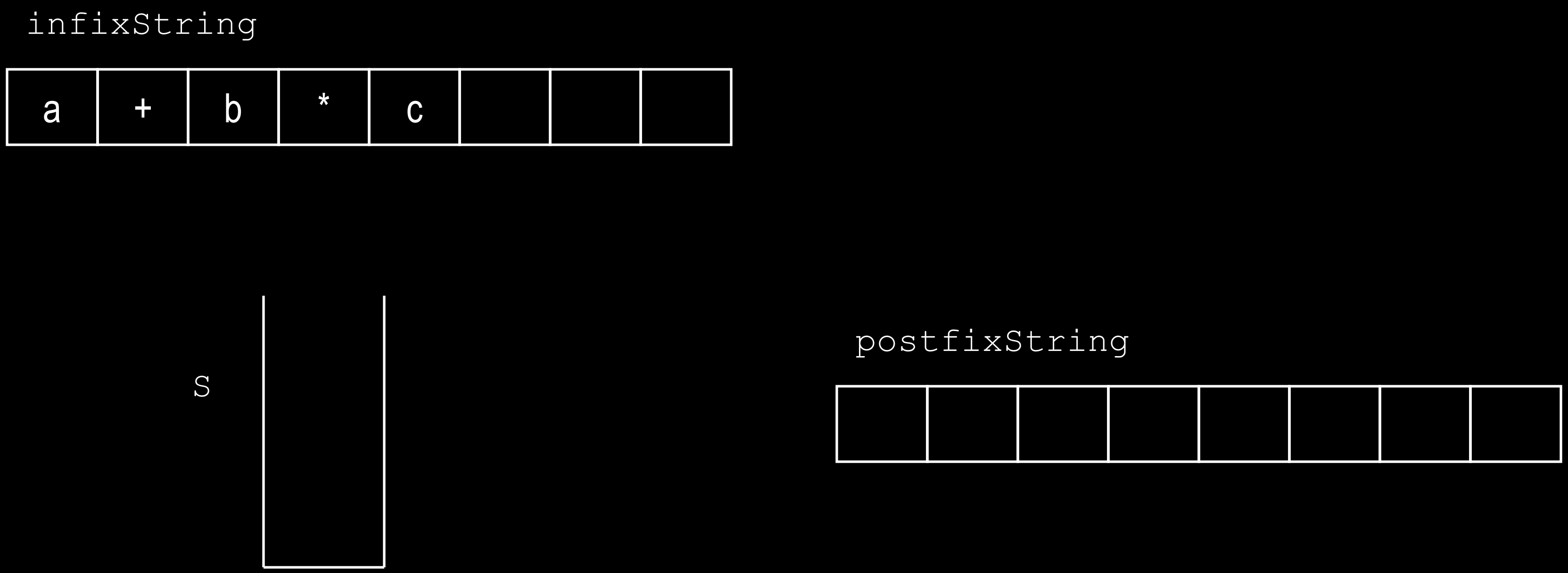
Let we have One Stack `S` which will be used for holding operators during conversion

A. Scan infixString from left to right and hold each character scanned in a char variable `c`

1. If `c` is an operand then pass it to postfixString
2. If `c` is an operator then we compare it with operator on top of stack
   - i. if operator on top of stack `S` has higher precedence than `c` , pop all the operators from stack `S` which have higher or equal precedence than `c` and pass them to `postfixString`
   - ii. push `c` on the stack `S`

B. Pop all remaining operators from `S` and pass them to `postfixString`

Infix to Postfix – Developing Algorithm – example 1



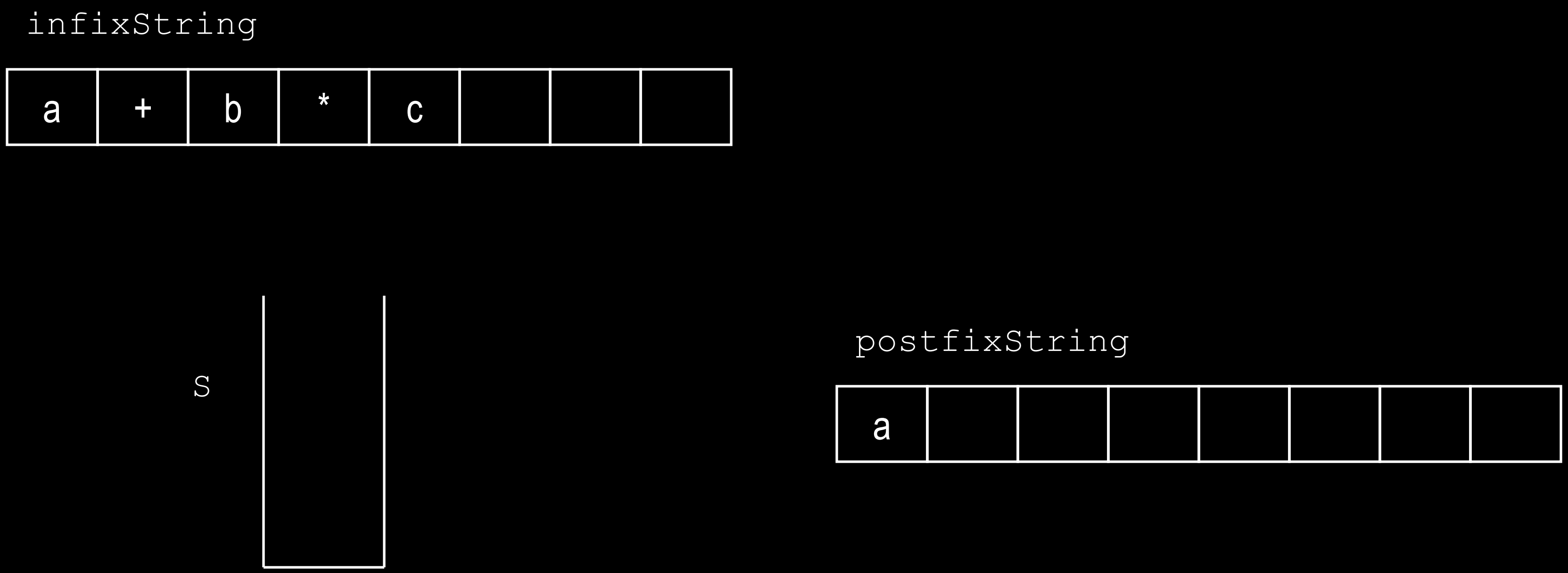
infixString

a	+	b	*	c			

S

postfixString

Infix to Postfix – Developing Algorithm – example 1



infixString

a	+	b	*	c			

S

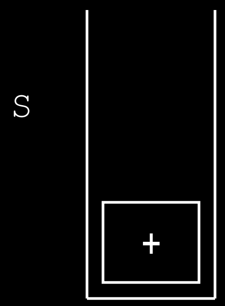
postfixString

a							

Infix to Postfix – Developing Algorithm – example 1

infixString

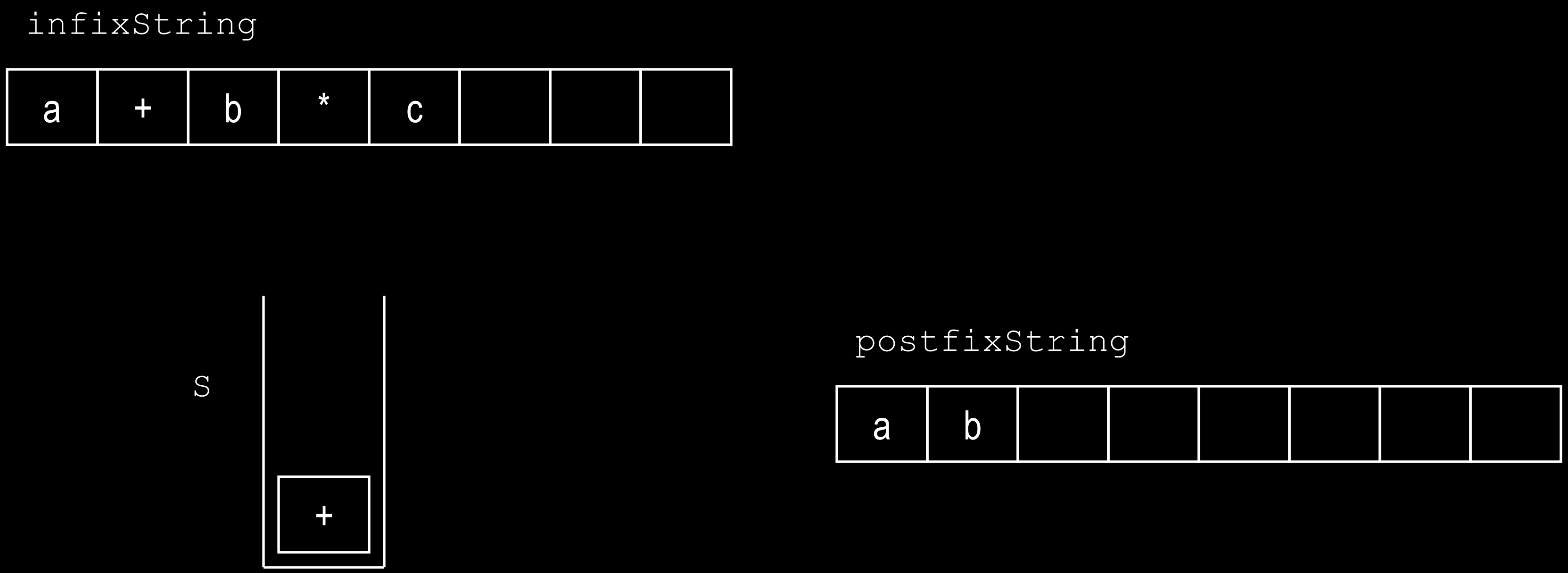
a	+	b	*	c			



postfixString

a							

Infix to Postfix – Developing Algorithm – example 1



infixString

a	+	b	*	c			

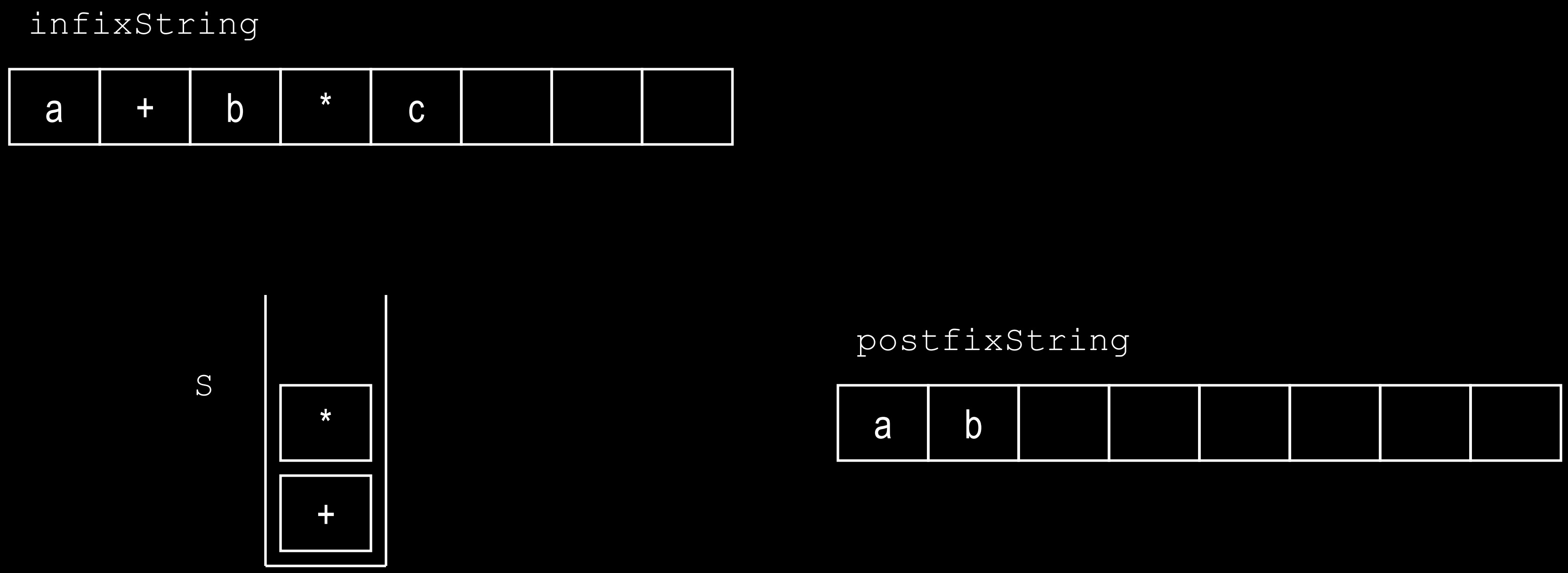
S

+

postfixString

a	b						

Infix to Postfix – Developing Algorithm – example 1



infixString

a	+	b	*	c			

S

*
+

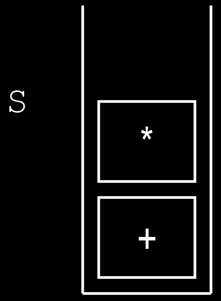
postfixString

a	b						

Infix to Postfix – Developing Algorithm – example 1

infixString

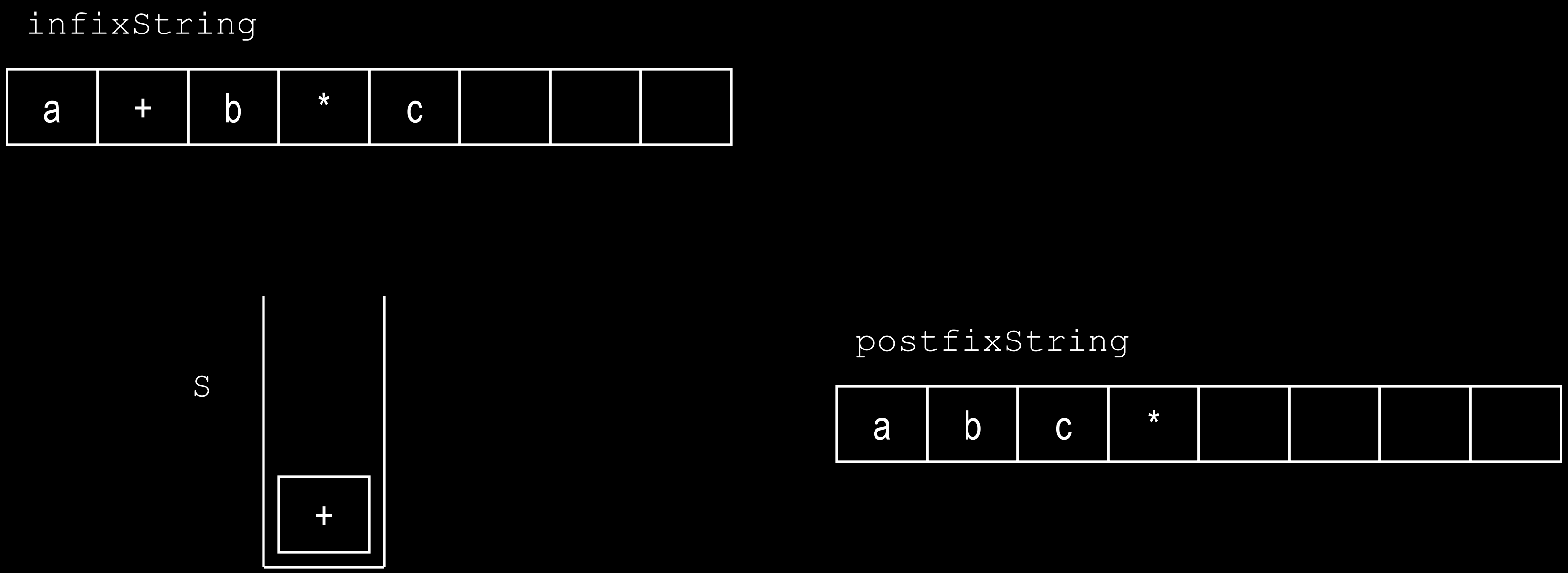
a	+	b	*	c			



postfixString

a	b	c					

Infix to Postfix – Developing Algorithm – example 1



infixString

a	+	b	*	c			

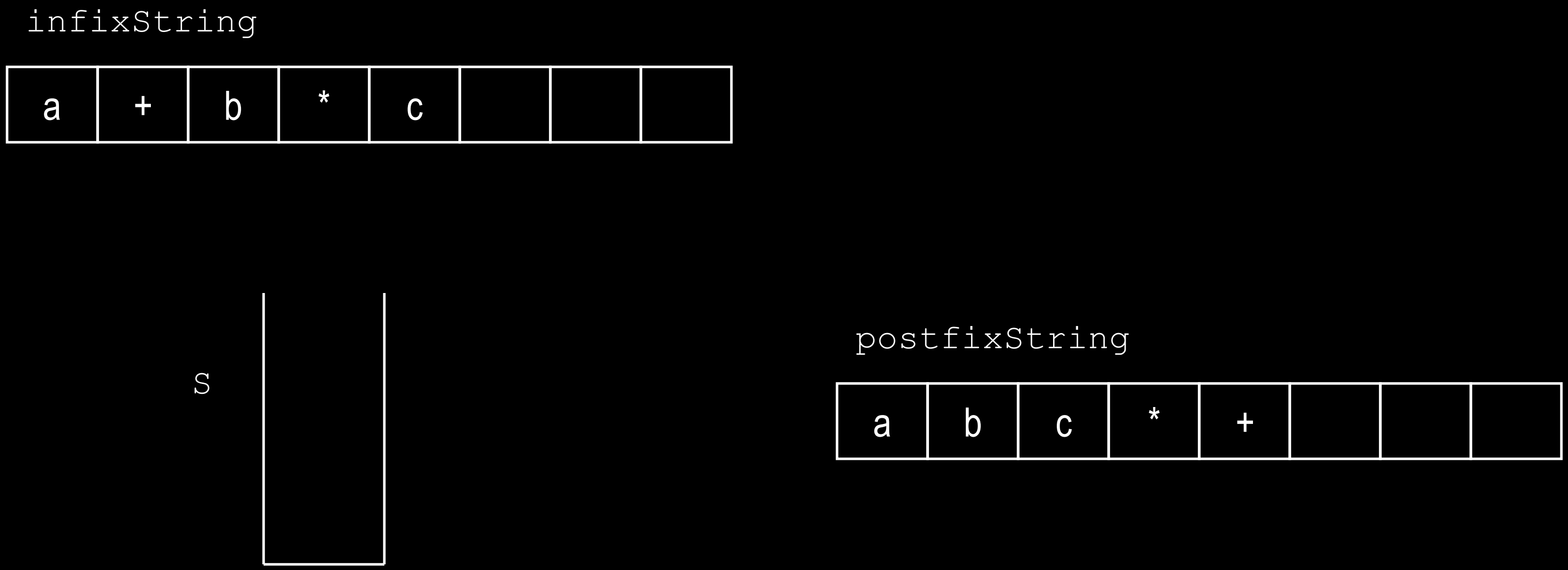
S

+

postfixString

a	b	c	*				

Infix to Postfix – Developing Algorithm – example 1



infixString

a	+	b	*	c			

S

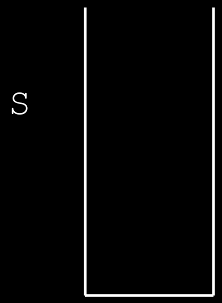
postfixString

a	b	c	*	+			

Infix to Postfix – Developing Algorithm – example 2

infixString

a	*	b	+	c			



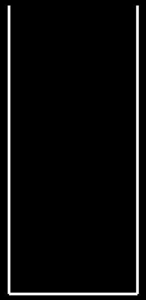
postfixString

Infix to Postfix – Developing Algorithm – example 2

infixString

a	*	b	+	c			

S



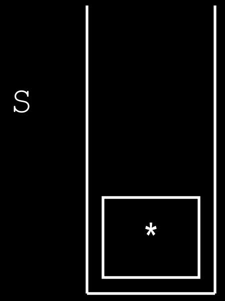
postfixString

a							

Infix to Postfix – Developing Algorithm – example 2

infixString

a	*	b	+	c			



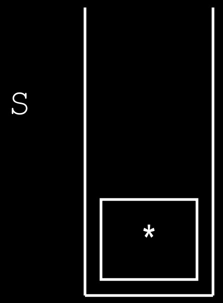
postfixString

a							

Infix to Postfix – Developing Algorithm – example 2

infixString

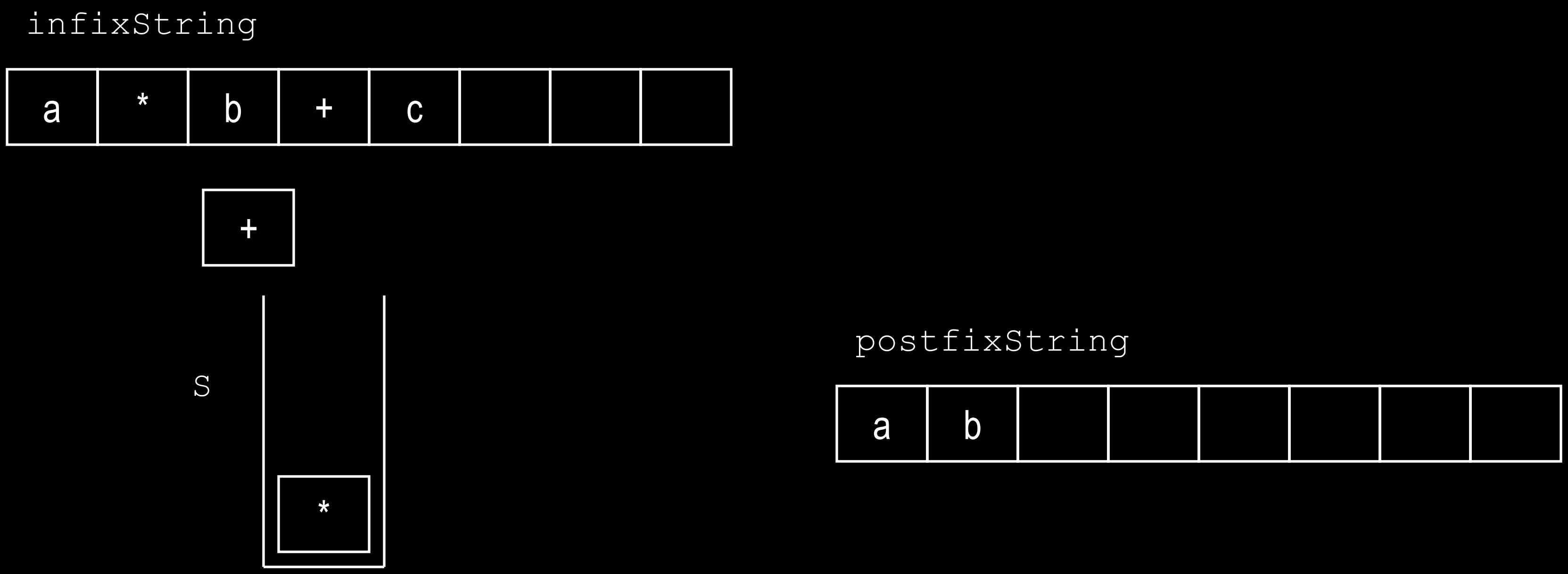
a	*	b	+	c			



postfixString

a	b						

Infix to Postfix – Developing Algorithm – example 2



infixString

a	*	b	+	c			

+

S

*

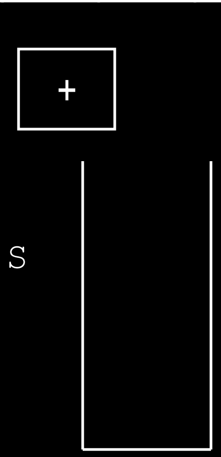
postfixString

a	b						

Infix to Postfix – Developing Algorithm – example 2

infixString

a	*	b	+	c			



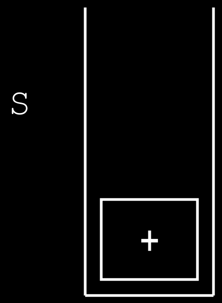
postfixString

a	b	*					

Infix to Postfix – Developing Algorithm – example 2

infixString

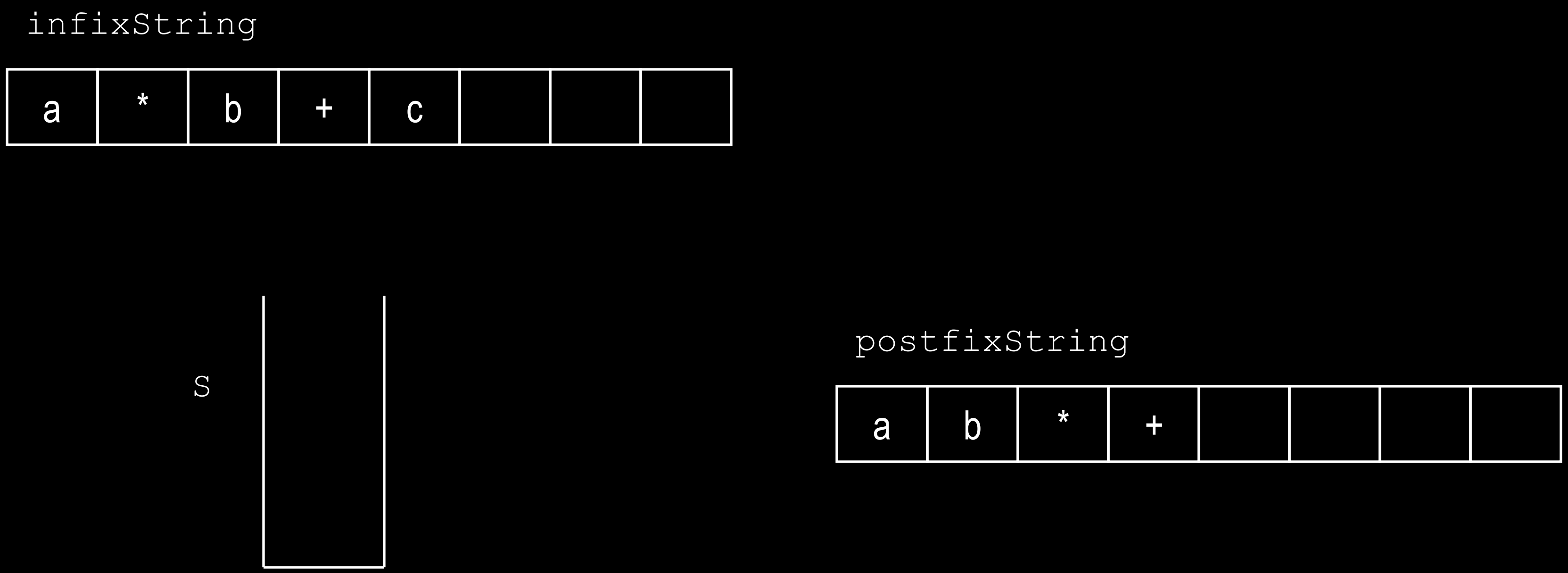
a	*	b	+	c			



postfixString

a	b	*					

Infix to Postfix – Developing Algorithm – example 2



infixString

a	*	b	+	c			

S

postfixString

a	b	*	+				

Evaluating Postfix

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we push it on a stack.
- When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

Evaluating Postfix

```
Stack s;
while( not end of input ) {
    e = get next element of input
    if( e is an operand )
        s.push( e );
    else {
        op2 = s.pop();
        op1 = s.pop();
        value = result of applying operator ‘e’ to op1 and op2;
        s.push( value );
    }
}
finalresult = s.pop();
```

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Infix to Postfix – Developing Algorithm - i

How to calculate the precedence of operators

- Assume the existence of a function `prcd(op1,op2)` where op1 and op2 are two operators.

  `Prcd(tp,rd)` returns 1 if `tp` has precedence over `rd`,

  returns 0 otherwise.

`prcd(‘*’,’+’)`	is	1
`prcd(‘+’,’+’)`	is	1
`prcd(‘+’,’*’)`	is	0

Converting Infix to Postfix – operator precedendence

prcd(‘+’, ‘+’) == 1
prcd(‘+’, ‘-’) == 1
prcd(‘+’, ‘*’) == 0
prcd(‘+’, ‘/’) == 0

prcd(‘*’, ‘+’) == 1
prcd(‘*’, ‘-’) == 1
prcd(‘*’, ‘*’) == 1
prcd(‘*’, ‘/’) == 1

prcd(‘(’, ‘+’) == 0
prcd(‘(’, ‘-’) == 0
prcd(‘(’, ‘*’) == 0
prcd(‘(’, ‘/’) == 0

prcd(‘-’, ‘+’) == 1
prcd(‘-’, ‘-’) == 1
prcd(‘-’, ‘*’) == 0
prcd(‘-’, ‘/’) == 0

prcd(‘/’, ‘+’) == 1
prcd(‘/’, ‘-’) == 1
prcd(‘/’, ‘*’) == 1
prcd(‘/’, ‘/’) == 1

prcd(‘)’, ‘+’) == 1
prcd(‘)’, ‘-’) == 1
prcd(‘)’, ‘*’) == 1
prcd(‘)’, ‘/’) == 1

Converting Infix to Postfix – Algorithm 1 (without parenthesis)

```
Stack S;
While( not end of infixString ) {
    c = next input character;
    if( c is an operand )
        add c to postfixString;
    else {
        while( S.IsEmpty == 0) && ( prcd( S.top(), c) ==1 )
          {
            opfromstack = S.pop();
            add opfromstack to the postfixString;
          }
         S.push( c );
    }
    while( S.IsEmty == 0 ) {
        opfromstack = S.pop( );
        add opfromstack to postfixString;
    }
```

Infix to Postfix – Developing Algorithm – ii
Handling Parenthesis

A. Scan infixString from left to right and hold each character scanned in a char variable c

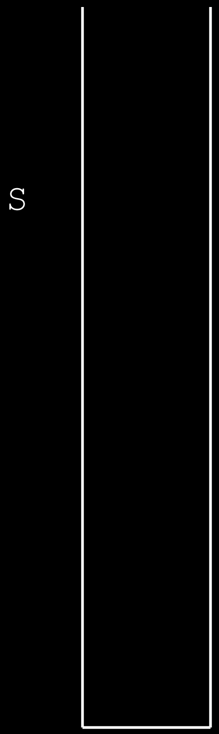
1. If c is an operand then pass it to postfixString
2. If c is an operator then we compare it with operator on top of stack
    - i. if operator on top of stack S has higher precedence than c , pop all the operators from stack S which have higher or equal precedence than c and pass them to postfixString
    - ii. push c on the stack S
3. If c is '(' push it on the stack
4. If c is ')' pop all the operators from the stack and pass to postfixstring until '(' is on the top of stack, discard '('

B. Pop all remaining operators from S and pass them to postfixString

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g

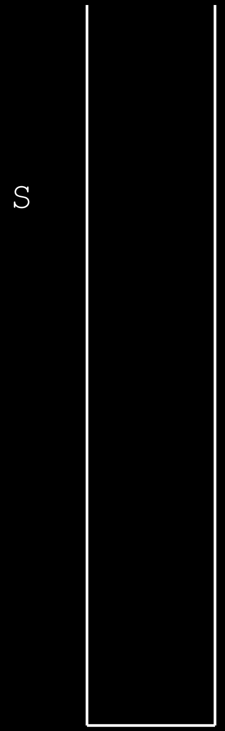


postfixString

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



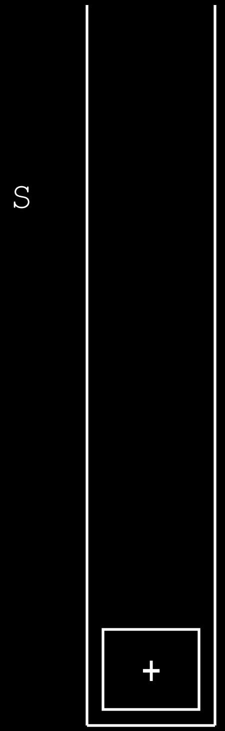
postfixString

a														

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



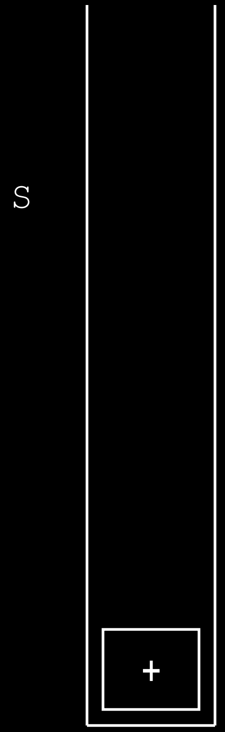
postfixString

a														

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



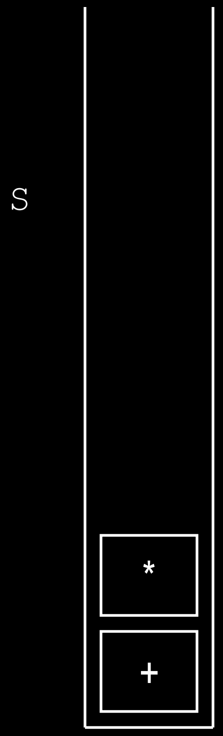
postfixString

a	b													

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



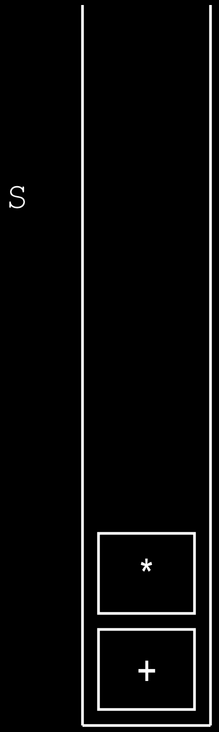
postfixString

a	b													

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



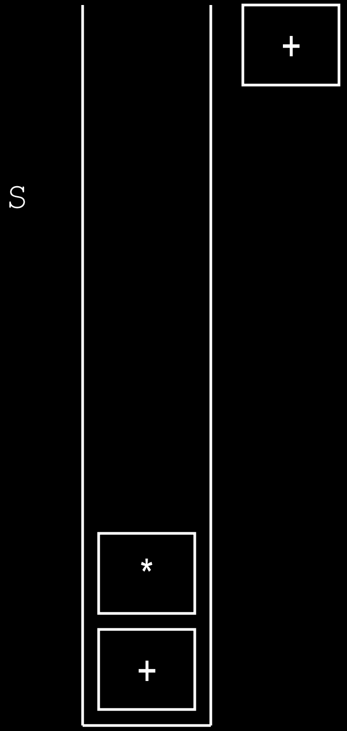
postfixString

a	b	c												

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



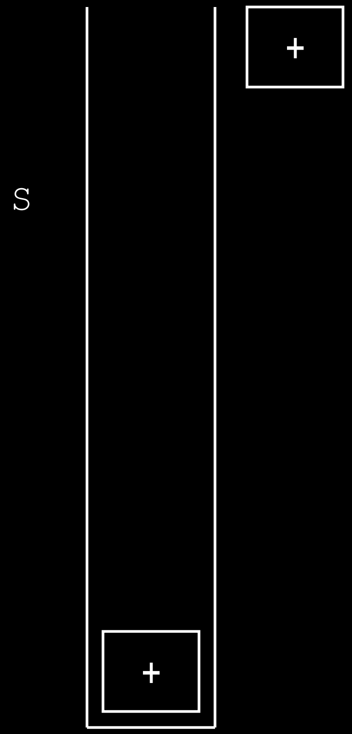
postfixString

a	b	c												

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



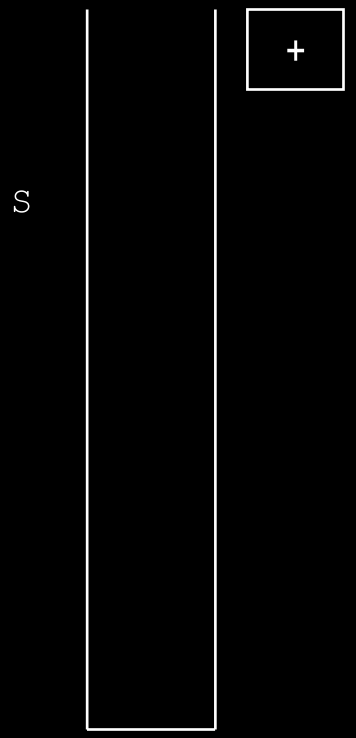
postfixString

a	b	c	*											

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



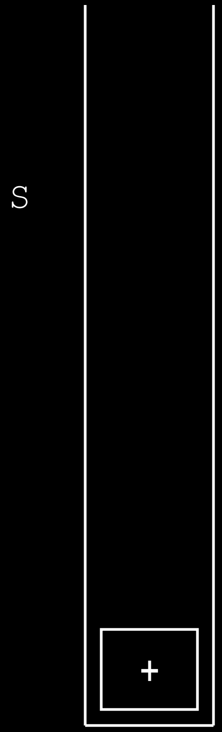
postfixString

a	b	c	*	+										

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



S

+

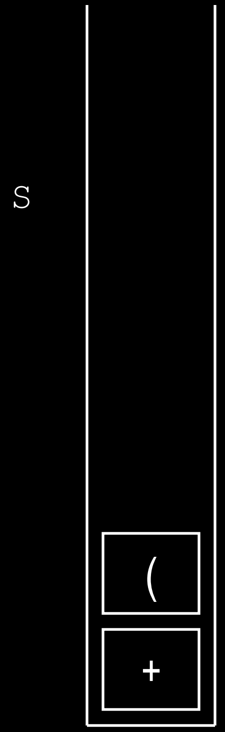
postfixString

a	b	c	*	+										

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



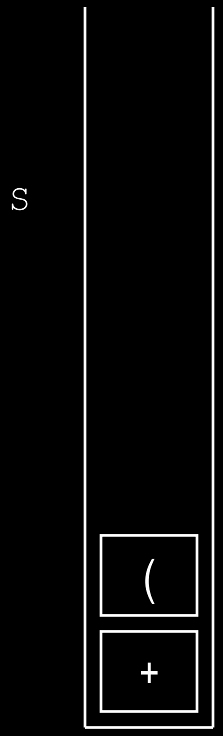
postfixString

a	b	c	*	+										

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



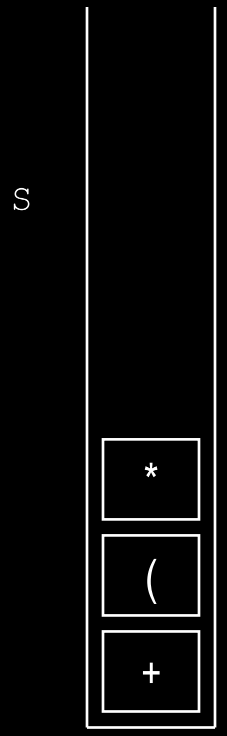
postfixString

a	b	c	*	+	d									

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



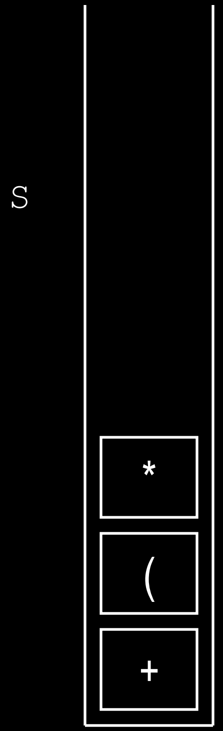
postfixString

a	b	c	*	+	d									

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



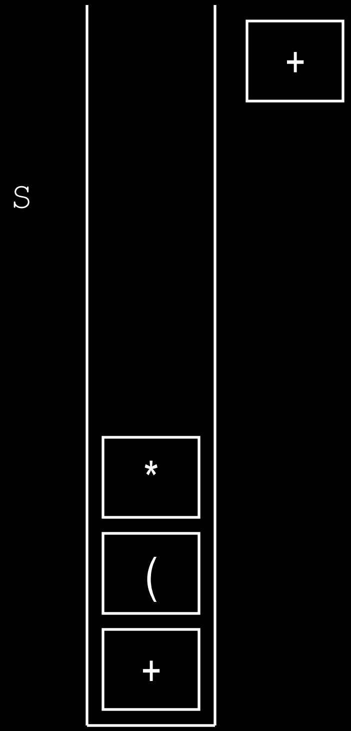
postfixString

a	b	c	*	+	d	e								

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



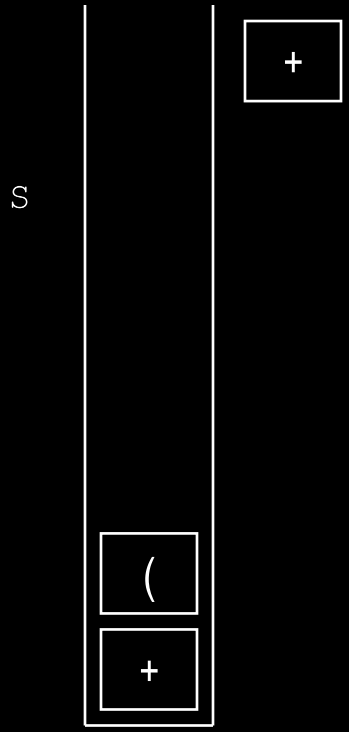
postfixString

a	b	c	*	+	d	e								

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



postfixString

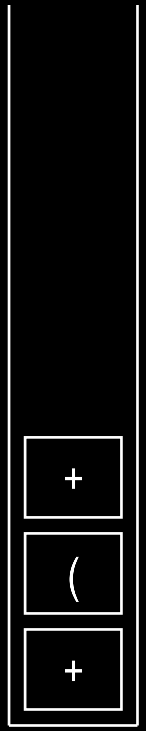
a	b	c	*	+	d	e	*							

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g

S



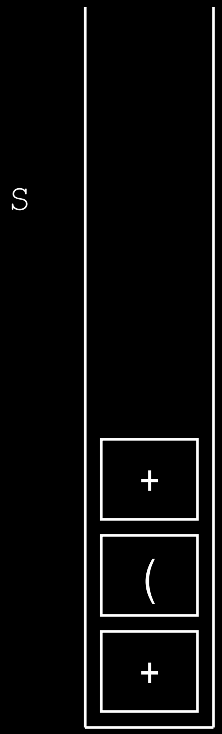
postfixString

a	b	c	*	+	d	e	*							

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



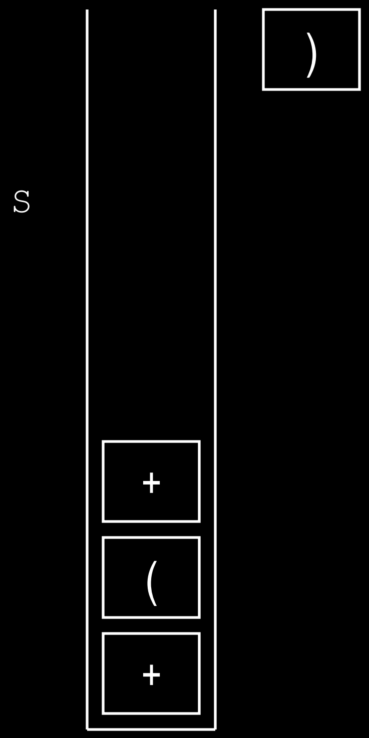
postfixString

a	b	c	*	+	d	e	*	f						

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



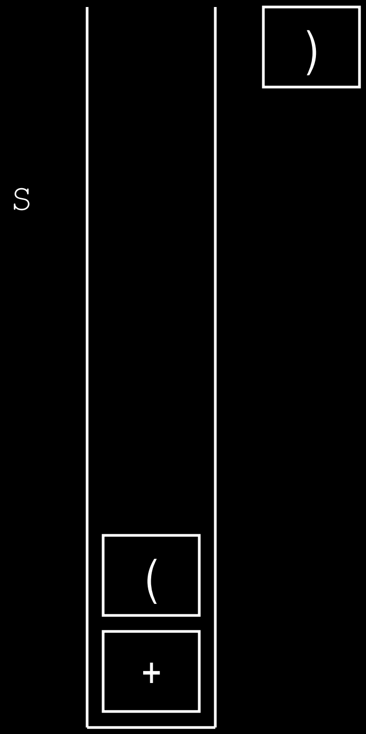
postfixString

a	b	c	*	+	d	e	*	f						

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



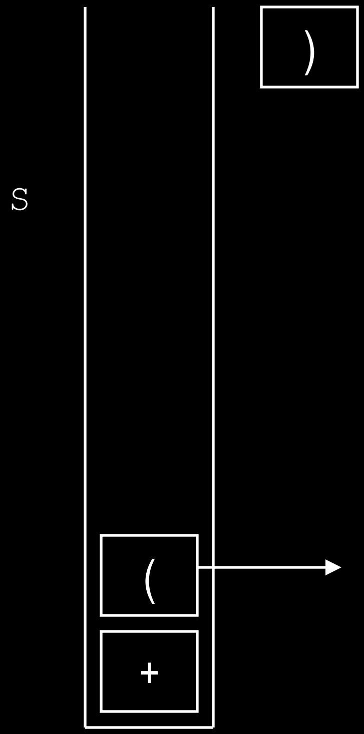
postfixString

a	b	c	*	+	d	e	*	f	+					

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



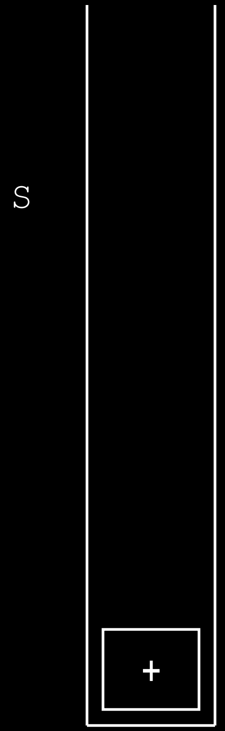
postfixString

a	b	c	*	+	d	e	*	f	+					

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



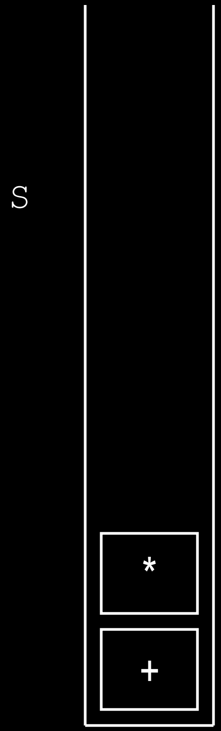
postfixString

a	b	c	*	+	d	e	*	f	+					

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



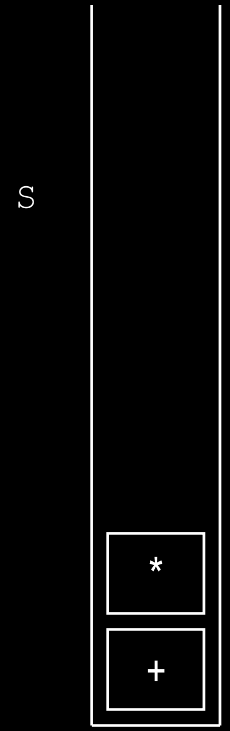
postfixString

a	b	c	*	+	d	e	*	f	+					

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



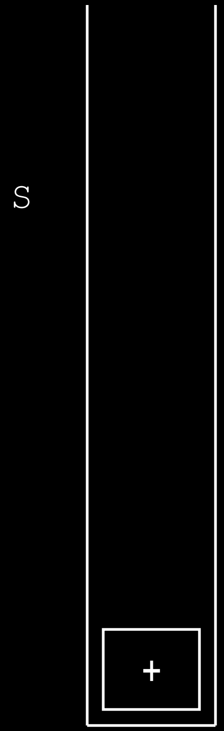
postfixString

a	b	c	*	+	d	e	*	f	+	g				

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



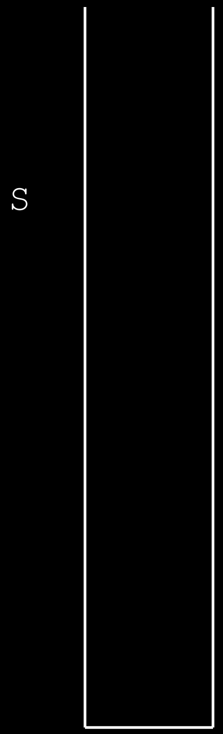
postfixString

a	b	c	*	+	d	e	*	f	+	g	*			

Infix to Postfix – Developing Algorithm ii – example 1

infixString

a	+	b	*	c	+	(	d	*	e	+	f	)	*	g



postfixString

a	b	c	*	+	d	e	*	f	+	g	*	+		

Converting Infix to Postfix - ii

Handling parenthesis

- When an open parenthesis ‘(‘ is read, it must be pushed on the stack

  This can be done by setting **prcd(op,’(‘) == 0.**
- Also, **prcd(‘(‘,op)== 0** which ensures that an operator after ‘(‘ is pushed on the stack.
- When a ‘)’ is read, all operators up to the first ‘(‘ must be popped and placed in the postfix string.

  To do this, **`prcd( op,’)’)== 1`**.
- Both the ‘(‘ and the ‘)’ must be discarded: **`prcd(‘(‘,’)’)==0`**.

prcd(‘(‘, op) = 0   for any operator

prcd(op, ‘(’) = 0   for any operator other than ‘)’

prcd(‘)’, ‘(‘) = 1

prcd(op, ‘)’) = 1   for any operator other than ‘(‘

prcd(op, ‘(‘) = 0

We have to change the algorithm also

Converting Infix to Postfix – ii
operator precedendence with parenthesis

```
prcd( ‘+’, ‘+’ ) == 1
prcd( ‘+’, ‘-’ )  == 1
prcd( ‘+’, ‘*’ )  == 0
prcd( ‘+’, ‘/’ )  == 0
prcd( ‘+’, ‘(’ )  == 0
prcd( ‘+’, ‘)’ )  == 1
```

```
prcd( ‘-’, ‘+’ ) == 1
prcd( ‘-’, ‘-’ )  == 1
prcd( ‘-’, ‘*’ )  == 0
prcd( ‘-’, ‘/’ )  == 0
prcd( ‘-’, ‘(’ )  == 0
prcd( ‘-’, ‘)’ )  == 1
```

```
prcd( ‘*’, ‘+’ ) == 1
prcd( ‘*’, ‘-’ )  == 1
prcd( ‘*’, ‘*’ )  == 1
prcd( ‘*’, ‘/’ )  == 1
prcd( ‘*’, ‘(’ )  == 0
prcd( ‘*’, ‘)’ )  == 1
```

```
prcd( ‘/’, ‘+’ ) == 1
prcd( ‘/’, ‘-’ )  == 1
prcd( ‘/’, ‘*’ )  == 1
prcd( ‘/’, ‘/’ )  == 1
prcd( ‘/’, ‘(’ )  == 0
prcd( ‘/’, ‘)’ )  == 1
```

```
prcd( ‘(’, ‘+’ ) == 0
prcd( ‘(’, ‘-’ )  == 0
prcd( ‘(’, ‘*’ )  == 0
prcd( ‘(’, ‘/’ )  == 0
prcd( ‘(’, ‘(’ )  == 0
prcd( ‘(’, ‘)’ )  == 0
```

```
prcd( ‘)’, ‘+’ ) == 1
prcd( ‘)’, ‘-’ )  == 1
prcd( ‘)’, ‘*’ )  == 1
prcd( ‘)’, ‘/’ )  == 1
prcd( ‘)’, ‘(’ )  == 0
prcd( ‘)’, ‘)’ )  == 1
```

Converting Infix to Postfix

```
Stack S;
While( not end of infixString ) {
    c = next input character;
    if( c is an operand )
        add c to postfixString;
    else {
        while ( S.IsEmpty()==0) && ( prcd( S.top(), c) ==1 )
           {
            opfromstack = S.pop();
            if (opfromstack!= ‘ (‘ )  then add opfromstack  to the postfixString;
          }
         if ( c != ‘)‘ )      then S.push( c );
    }
    while( S.IsEmpty() == 0 ) {
        opfromstack = S.pop();
        if (opfromstack != ‘(‘ )    then add opfromstack to postfixString;
    }
```

Converting Infix to Postfix

- Example: (A + B) * C

symb	postfix	stack
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB +	
*	AB +	*
C	AB + C	*
	AB + C *	

Converting Infix to Postfix

```
#include<iostream.h>
#iclude<ctype.h>
int prcd(char,char);
Using namespace std;
int main()
{
 stack S;
 int i=0, p=0;
 char c,opfromstack;
 char infixString[80];
 char postfixString[80];
 cout << "Enter Infix expression:\n";
 cin >> infixString;
```

Converting Infix to Postfix

```
while (infixString[i]!='\0')
 {
 c=infixString[i];

 if (isalpha(c)||isdigit(c))
     { postfixString[p++]=c;}

 else
     { while ((S.IsEmpty()==0)&&(prcd(S.top(),c)==1))
           { opfromstack=S.pop();
             if (opfromstack!='(')
                 postfixString[p++]=opfromstack;}
       if (c!=‘)’) S.push(c);
     }
     i++;
 }

 while (S.IsEmpty()==0)
       {opfromstack=S.pop();
       if (opfromstack!='(') postfixString[p++]=opfromstack;}

 postfixString[p]='\0';
 cout << postfixString << endl;

}
```

Converting Infix to Postfix

```
int prcd(char tp, char rd)
{
 switch (rd)
   {
    case '+':
    case '-':
              if (tp=='(') return 0;
              else return 1;
    case '*':
    case '/':
              if ( (tp=='+‘)||(tp=='-‘)||(tp=='(‘) ) return 0;
              else return 1;

    case '(':
             return 0;
    case ')':
             if (tp=='(') return 0;
              else return 1;
   }
}
```

Function Call Stack

- Stacks play a key role in implementation of function calls in programming languages.
- In C++, for example, the “call stack” is used to pass function arguments and receive return values.
- The call stack is also used for “local variables”