Indian Institute of Information Technology Allahabad

Data Structures and Algorithms

Stacks

Dr. Shiv Ram Dubey
Associate Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

Stacks

- Abstract Data Types (ADTs)
- Stacks
- Application to the analysis of a time series
- Growable stacks

Abstract Data Types (ADTs)

- ***ADT*** is mathematically specified entity that defined a set of *its instances*, with:
  - a specific *interface* – a collection of signatures of operations that can be invoked on an instance.
  - a set of *axioms* (*preconditions* and *postconditions*) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but now how)

Abstract Data Types (ADTs)

- Why do we need to talk about ADTs in a DS course?
  - They serve as *specification of requirements* for the building blocks of solutions to algorithmic problems
  - Provides a language to talk on a higher level of abstraction
  - ADTs encapsulate *data structures* and algorithms that *implement* them
  - Separate the issues of *correctness* and *efficiency*

Example – Dynamic Sets

- We will deal with ADTs, instances of which are sets of some type of elements.
  - Operations are provided that change the set.
- We call such class of ADTs *dynamic sets*

Dynamic Sets

- An example dynamic set ADT
  - Methods:
    - **New()**:*ADT*
    - **Insert(S**:*ADT*, **v**:*element***)**:*ADT*
    - **Delete(S**:*ADT*, **v**:*element***)**:*ADT*
    - **IsIn(S**:*ADT*, **v**:*element***)**:*boolean*
- **Insert** and **Delete** – *manipulation* operations
- **IsIn** – *Access* method

Dynamic Sets

- Axioms that define the methods:
  - **IsIn(New(), v)** = false
  - **IsIn(Insert(S, v), v)** = true
  - **IsIn(Insert(S, u), v)** = **IsIn(S, v)**, *if $u \neq v$*
  - **IsIn(Delete(S, v), v)** = false
  - **IsIn(Delete(S, u), v)** = **IsIn(S, v)**, *if $u \neq v$*

Abstract Data Types (ADTs)

- There are lots of formalized and standard ADTs.

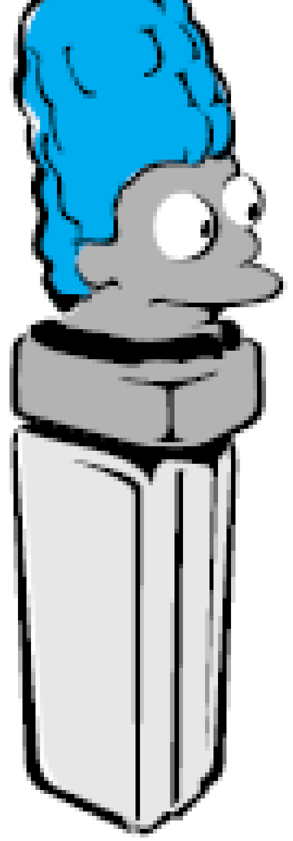
- In this course we are going to learn a lot of different standard ADTs.
  - ***stacks***,
  - queues,
  - trees
  - …

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “**pushing**” onto the stack. “**Popping**” off the stack is synonymous with removing an item.

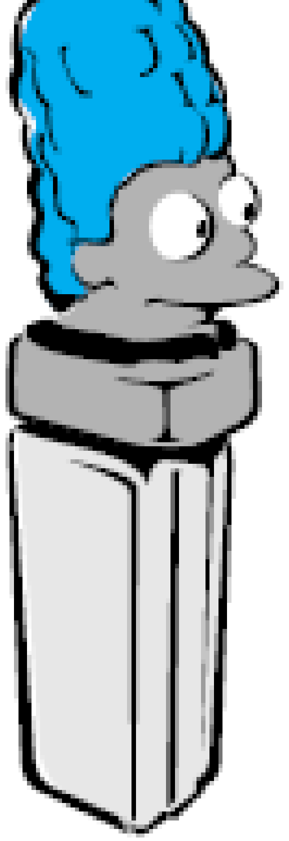
Stacks

- A PEZ ® dispenser is an analogy:

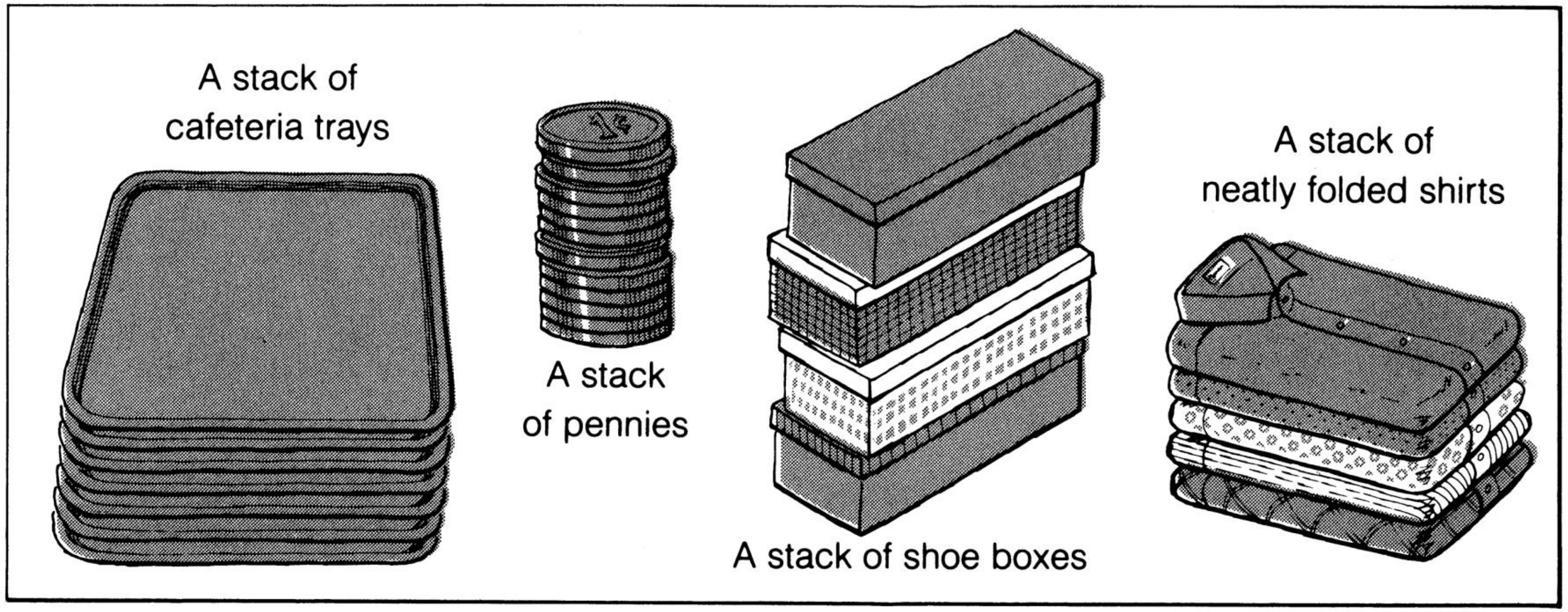


Stacks

- A PEZ ® dispenser is an analogy:



- Other examples:



Stacks

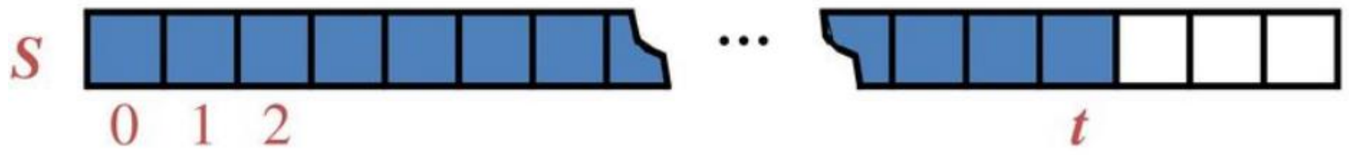
- A stack is an ADT that supports four main methods:
  - **new()**:*ADT* – Creates a new stack
  - **push(S**:*ADT*, **o**:*element***)**:*ADT* – Inserts object o onto top of stack S
  - **pop(S**:*ADT***)**:*ADT* – Removes the top object of stack S; if the stack is empty an error occurs, so take care.
  - **top(S**:*ADT***)**:*element* – Returns the top object of the stack, without removing it; if the stack is empty an error occurs, so take care.

Stacks

- The following support methods should also be defined:
  - **size(S**:*ADT***)**:*integer* - Returns the number of objects in stack S
  - **isEmpty(S**:*ADT***)**:*boolean* - Return a boolean indicating if stack S is empty.
- Axioms
  - **pop(push(S, v)) = S**
  - **top(push(S, v)) = v**

Array-Based Stack

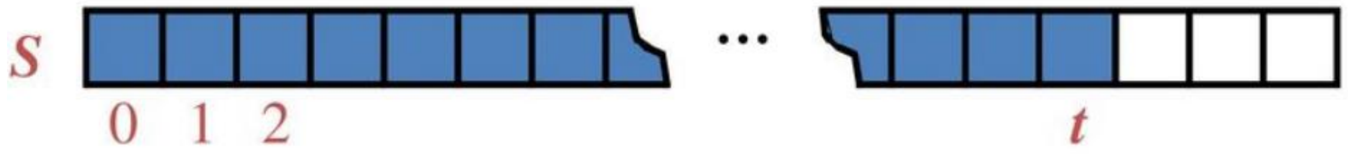
- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array S and an integer variable *t*, the index of the top element in array S.



- Array indices start at 0, so we initialize *t* to -1.

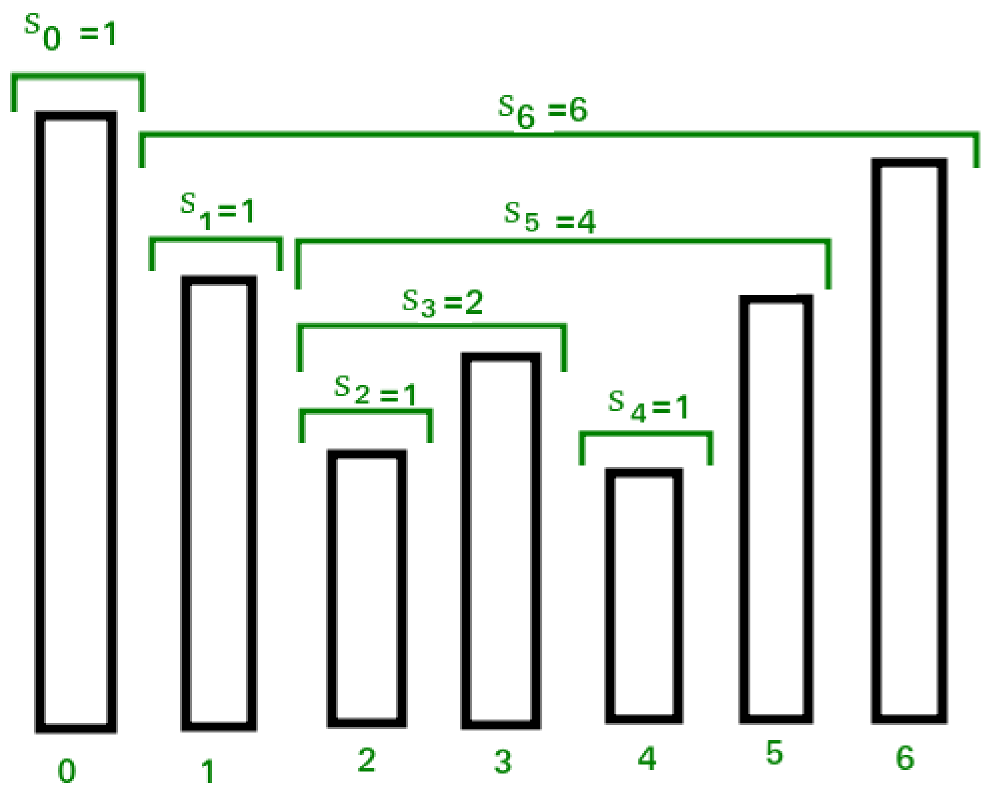
Array-Based Stack

- The array implementation is simple and efficient (methods performed in O(1)).
- There is an upper bound, *N*, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.
- Stack Empty case is required to be dealt.
- Stack Full is particular to this implementation.



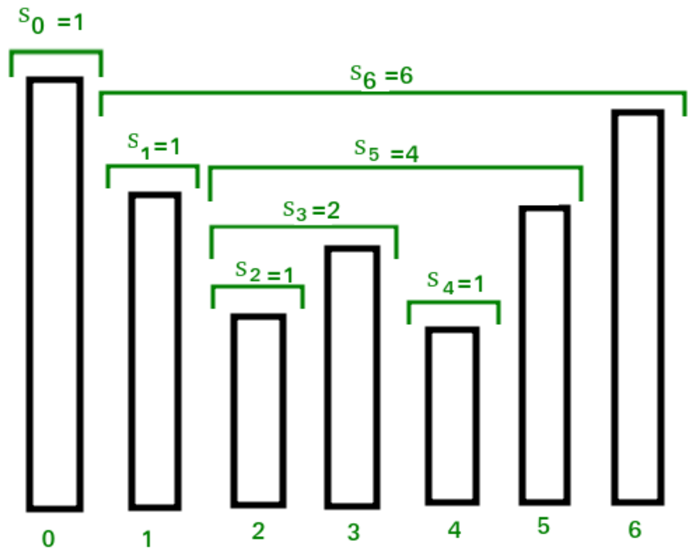
Application: Time Series

- The **span s_i** of a stock's price on a certain day i is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day i.



An Inefficient Algorithm

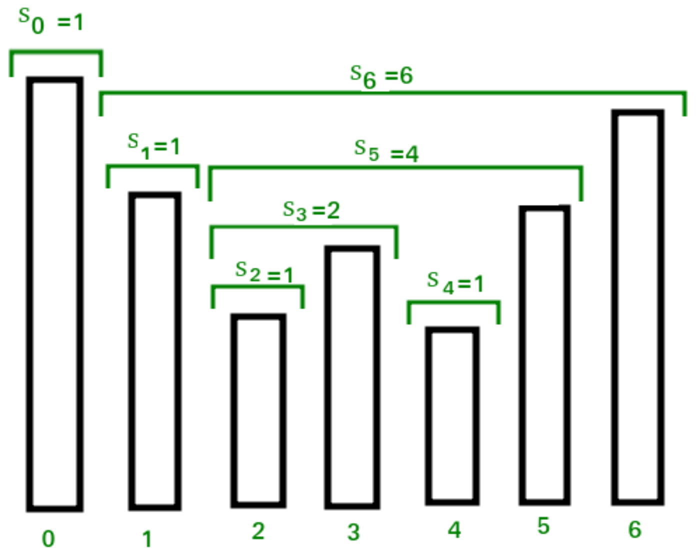
```
Algorithm computeSpans1(P):
for i = 0 to n-1 do
        k = 0; done = false
        repeat
                if P[i-k] <= P[i] then k = k+1
                                else done = true
        Until (k == i) or done
        S[i] = k
return S
```



Input P -> Stock price in an array
Output S -> Span in an array

An Inefficient Algorithm

```
Algorithm computeSpans1(P):
for i = 0 to n-1 do
        k = 0; done = false
        repeat
                if P[i-k] <= P[i] then k = k+1
                                    else done = true
        Until (k == i) or done
        S[i] = k
return S
```



Input P -> Stock price in an array
Output S -> Span in an array

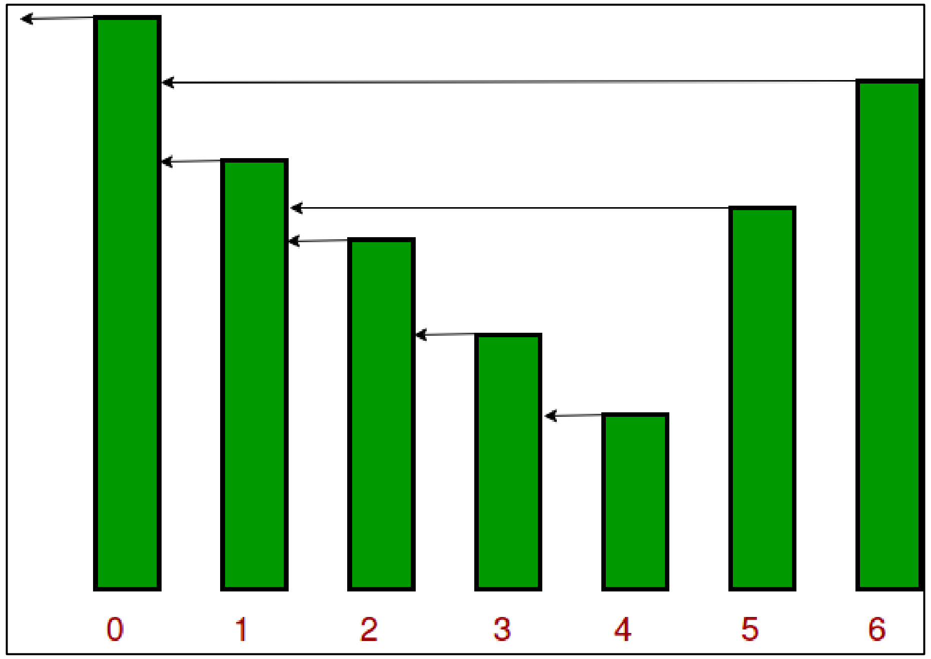
The running time of this algorithm is $O(n^2)$. Why?

A Stack Can Help

- S_i can be easily computed if we know the closest day preceding i, on which the price is greater than the price on day i.

- If such a day exists, let’s call it h(i), otherwise, we conventionally define h(i) = -1.

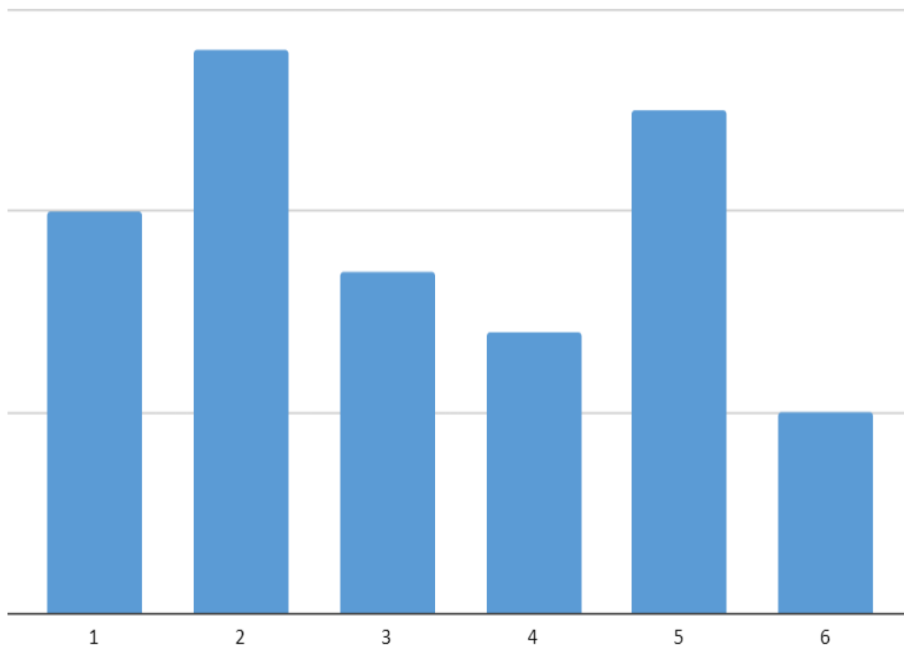
- In the fugure h(3)=2, h(5)=1 and h(6)=0.
- The span is now computed as $s_i = i - h(i)$



What to do with the Stack?

What are possible values of h(7)?

Can it be 1 or 3 or 4?

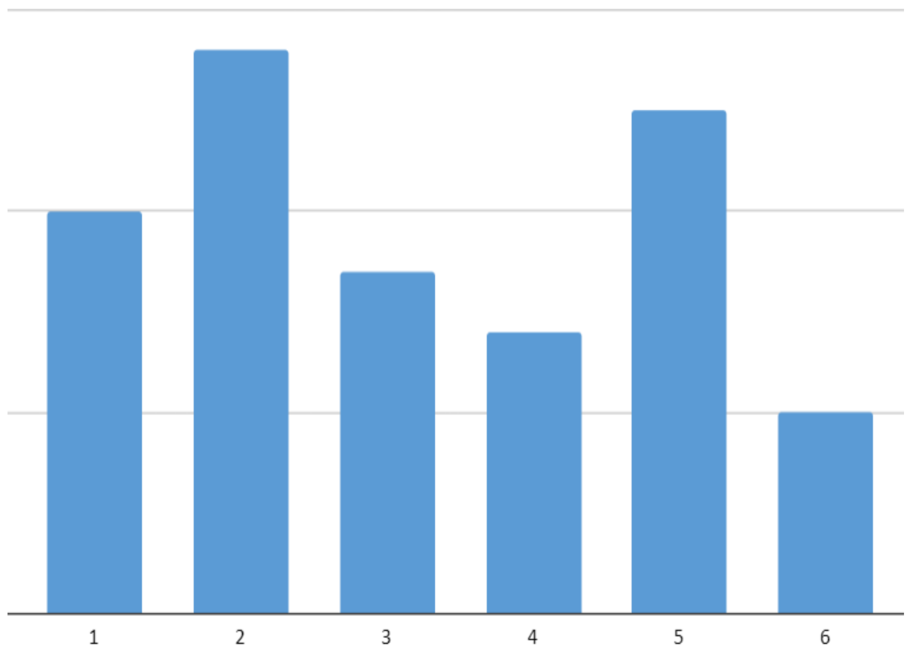


What to do with the Stack?

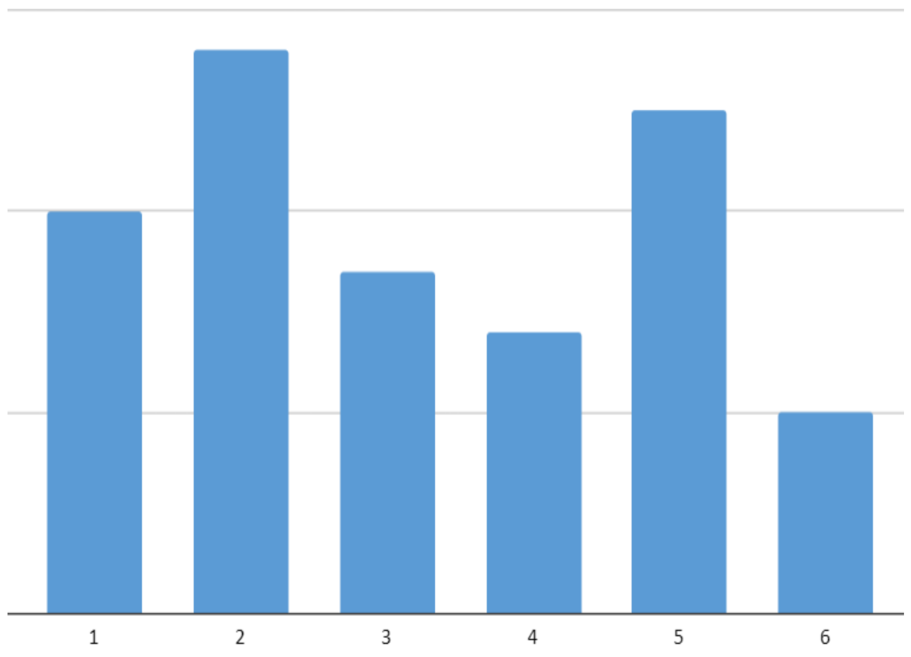
What are possible values of h(7)?

Can it be 1 or 3 or 4?

No, h(7) can only be 2 or 5 or 6



What to do with the Stack?



What are possible values of h(7)?

Can it be 1 or 3 or 4?

No, h(7) can only be 2 or 5 or 6

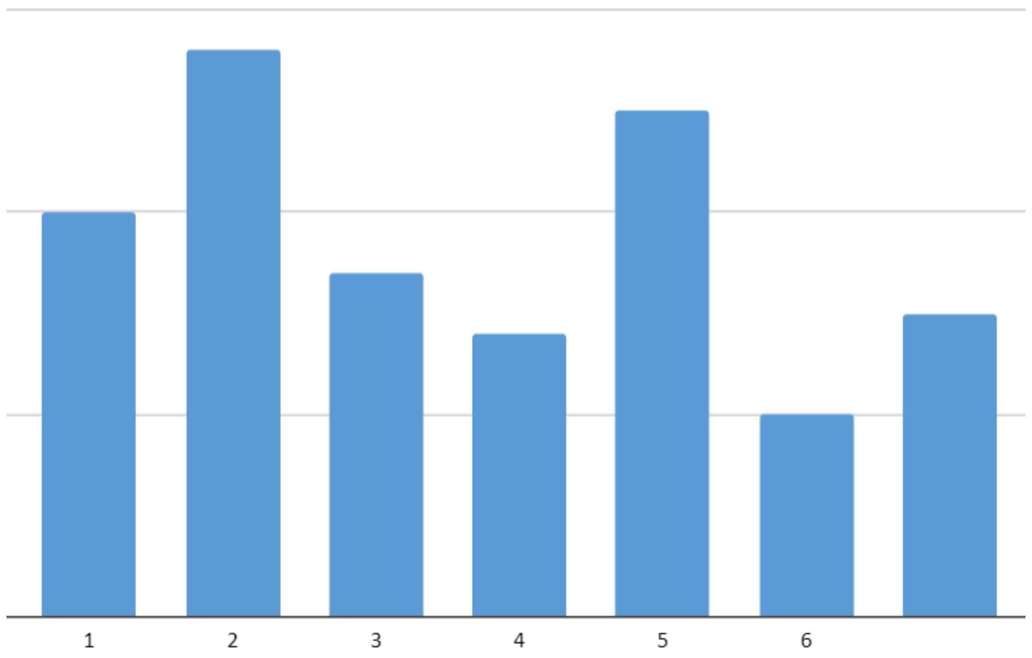
- We store indices 2,5,6 in the stack
- To determine h(7) we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.

What to do with the Stack?

What are possible values of h(7)?

Can it be 1 or 3 or 4?

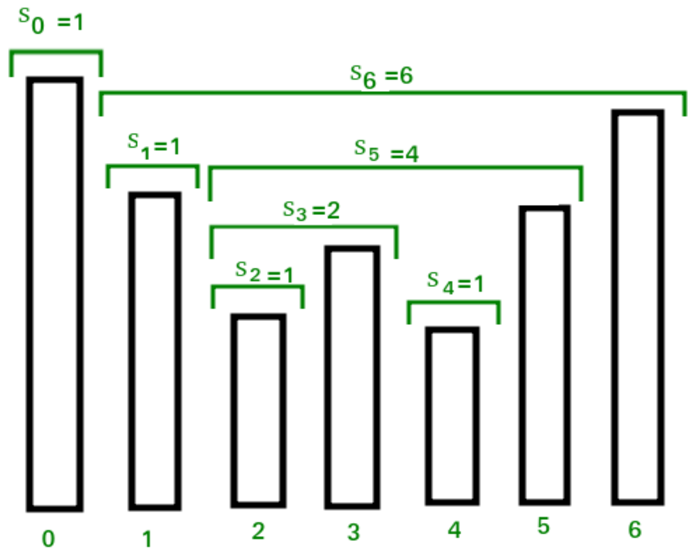
No, h(7) can only be 2 or 5 or 6



- We store indices 2,5,6 in the stack
- To determine h(7) we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.
- The first price larger than the price on day 7 gives h(7)
- The stack should be updated to reflect the price of day 7
- It should now contains 2,5,7

An Efficient Algorithm

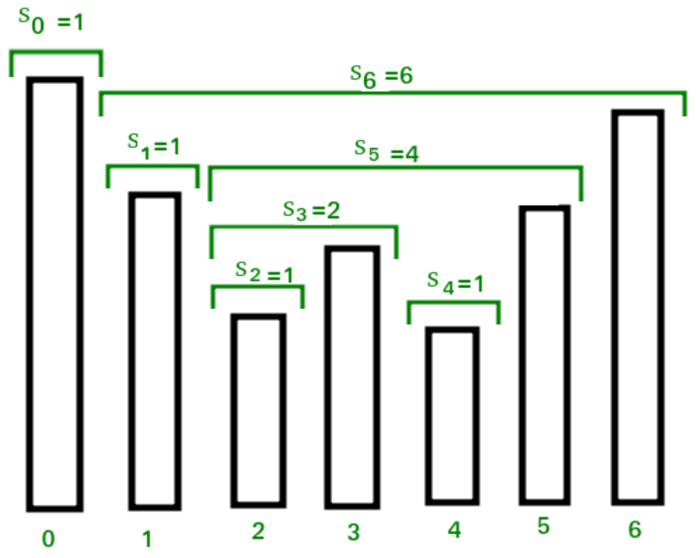
```
Algorithm computeSpans2(P):
for i = 0 to n-1 do
        k = 0; done = false
        while not (D.isEmpty() or done) do
                if P[i] >= P[D.top()] then D.pop()
                                else done = true
        if D.isEmpty() then h=-1
                        else h=D.top()
        S[i] = i-h
        D.push(i)
return S
```



Let D be an empty stack

An Efficient Algorithm

```
Algorithm computeSpans2(P):
for i = 0 to n-1 do
        k = 0; done = false
        while not (D.isEmpty() or done) do
                if P[i] >= P[D.top()] then D.pop()
                                        else done = true
        if D.isEmpty() then h=-1
                        else h=D.top()
        S[i] = i-h
        D.push(i)
return S
```



Let D be an empty stack

The running time of this algorithm is O(n). Why?

A Growable Array-Based Stack

There are two strategy for growable stack:

- **Tight Strategy :** Add a constant amount to the old stack (N+c)
- **Growth Strategy :** Double the size of old stack (2N)

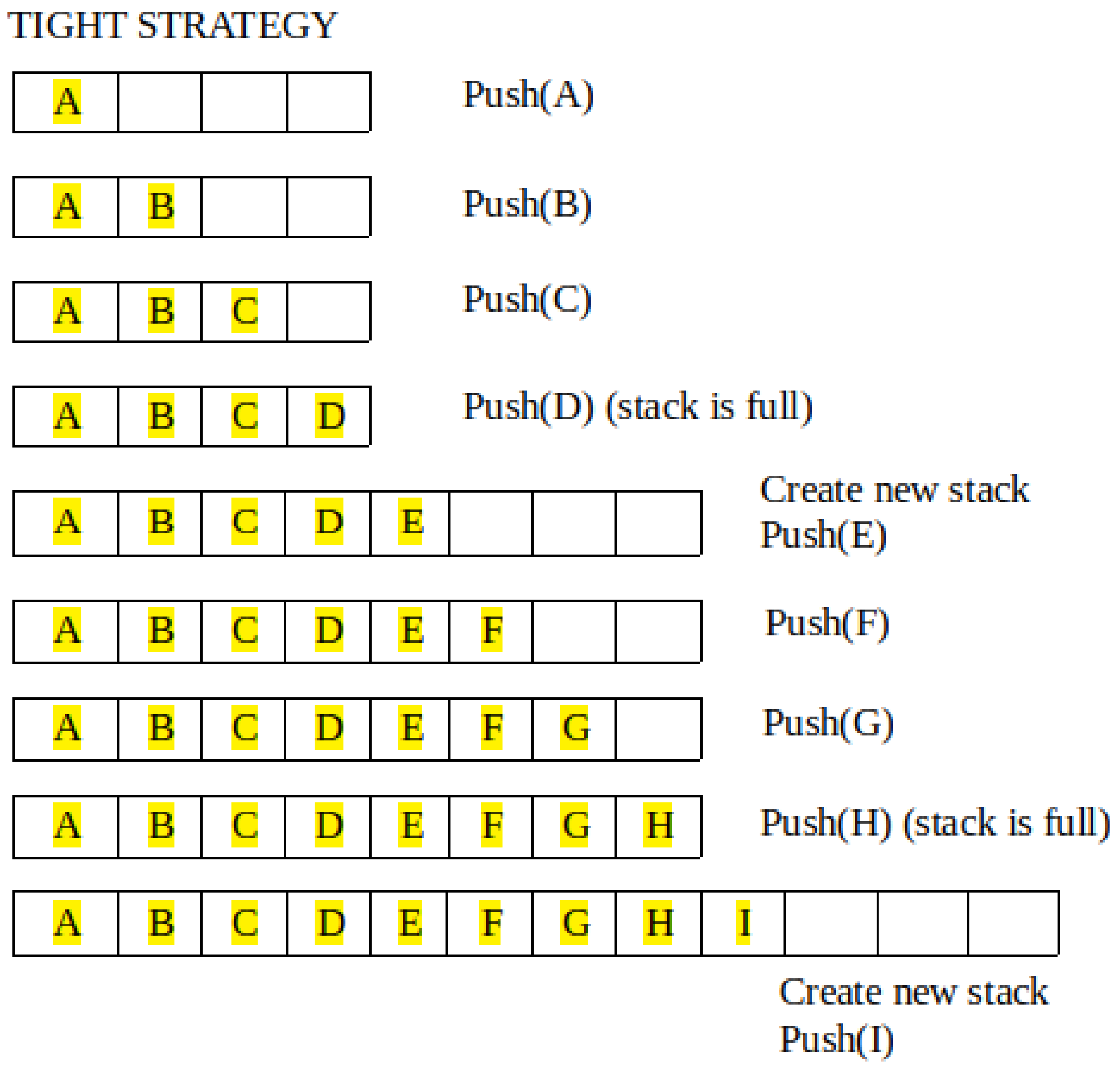


Figure from geeksforgeeks

A Growable Array-Based Stack

We can replace the array S with a larger one and continue processing push operations.

```
Algorithm push(o):
if size()==N then A = new array if length f(N)
for i = 0 to N-1 do
    A[i] = S[i]
S=A; t=t+1
S[t]=o
```

Tight vs. Growth Strategies: Comparison

To compare the two strategies, we use the following cost model:

- **A Regular Push Operation:** Adds one element at top of stack. It costs one unit.
- **A Special Push Operation:** Create a new stack (using array) of size greater than old stack (according to one of the strategy above, f(N)) and copy all N elements from old stack and then push the new element to the new stack. It costs f(N)+N+1 units

Tight Strategy (c=4)

Start with an array of size 0. Cost of special push is 2N+5.

a			
a	b		
a	b	c	
a	b	c	d

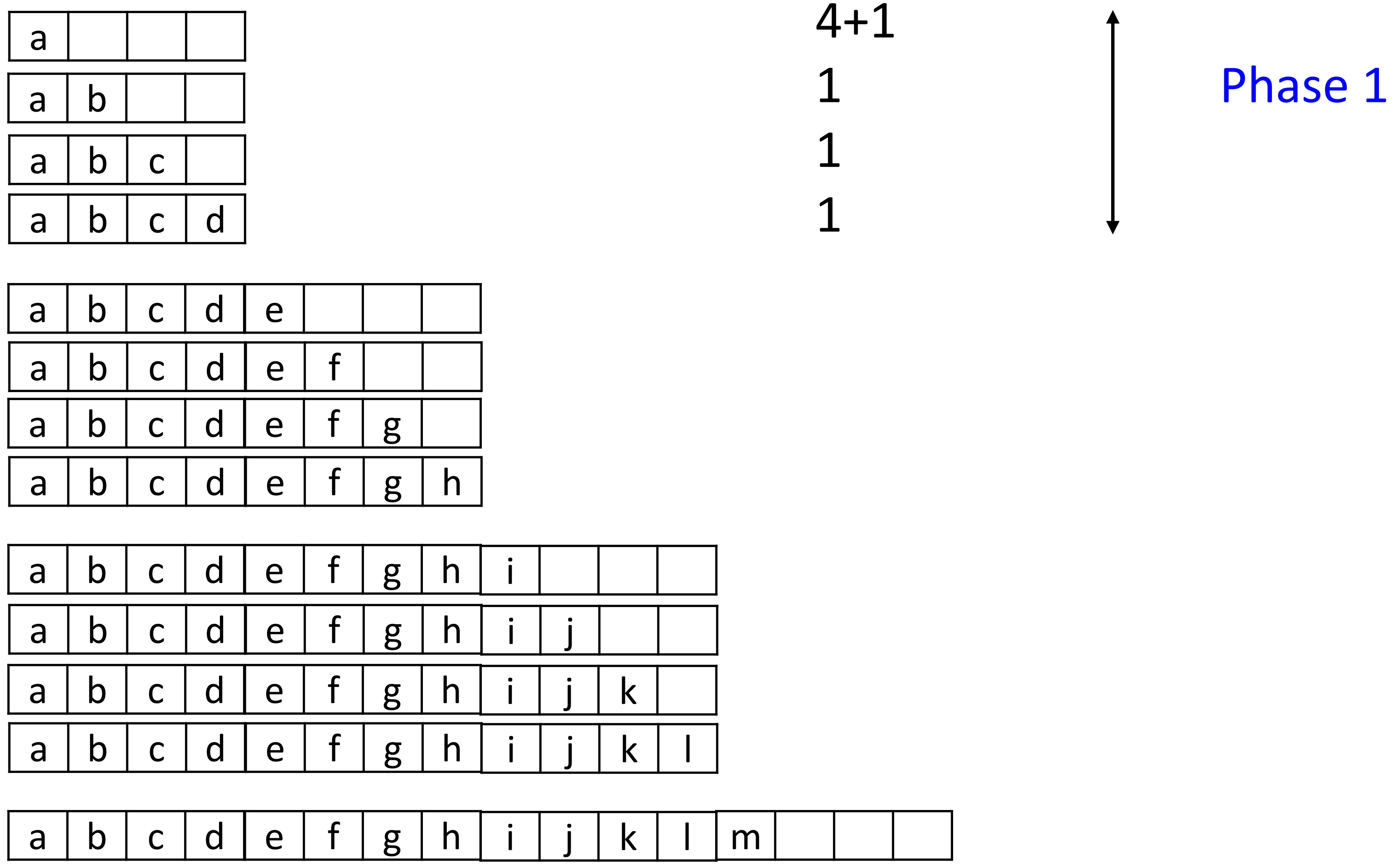
a	b	c	d	e			
a	b	c	d	e	f		
a	b	c	d	e	f	g	
a	b	c	d	e	f	g	h

a	b	c	d	e	f	g	h	i			
a	b	c	d	e	f	g	h	i	j		
a	b	c	d	e	f	g	h	i	j	k	
a	b	c	d	e	f	g	h	i	j	k	l

a	b	c	d	e	f	g	h	i	j	k	l	m			

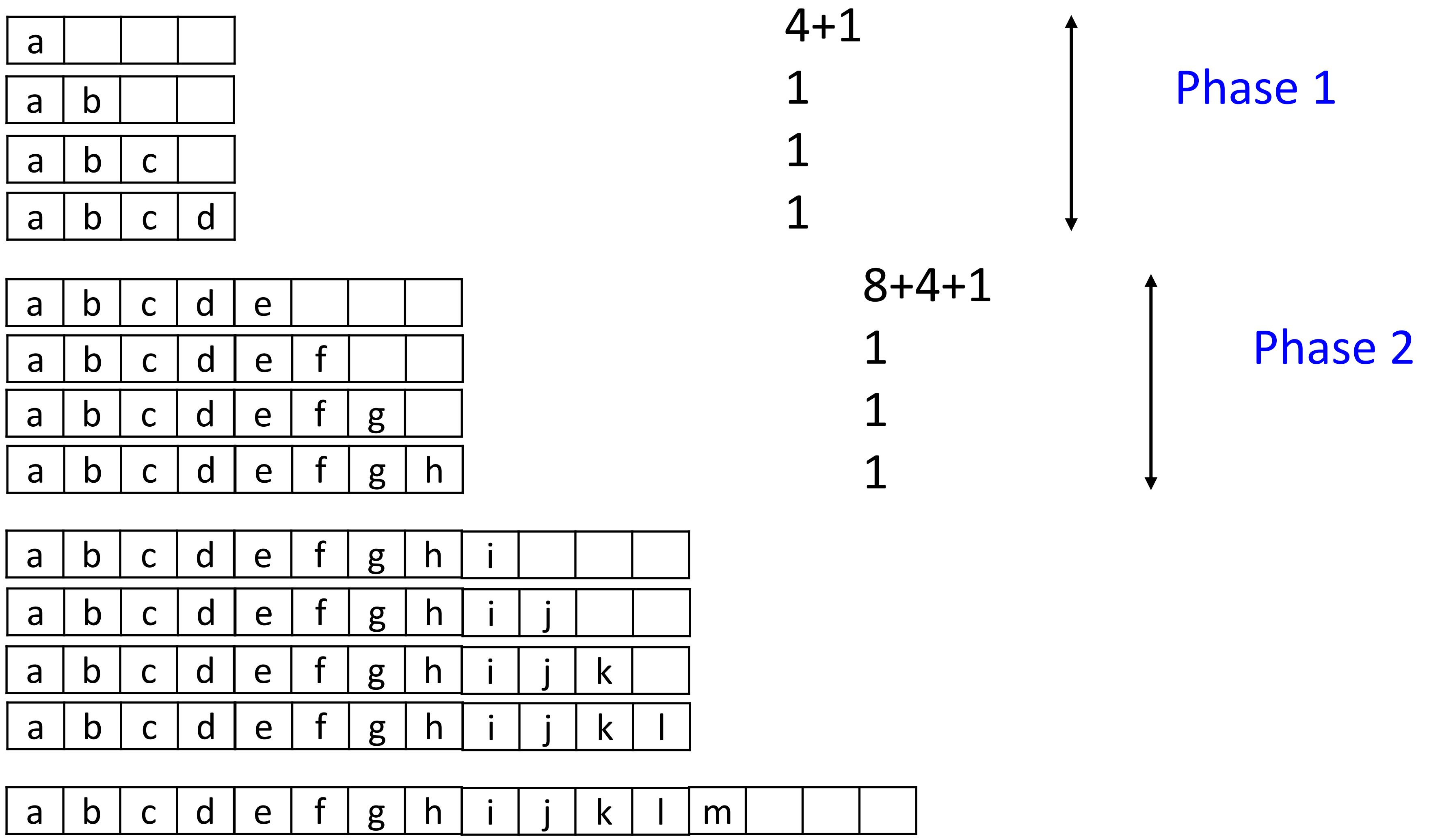
Tight Strategy (c=4)

Start with an array of size 0. Cost of special push is 2N+5.



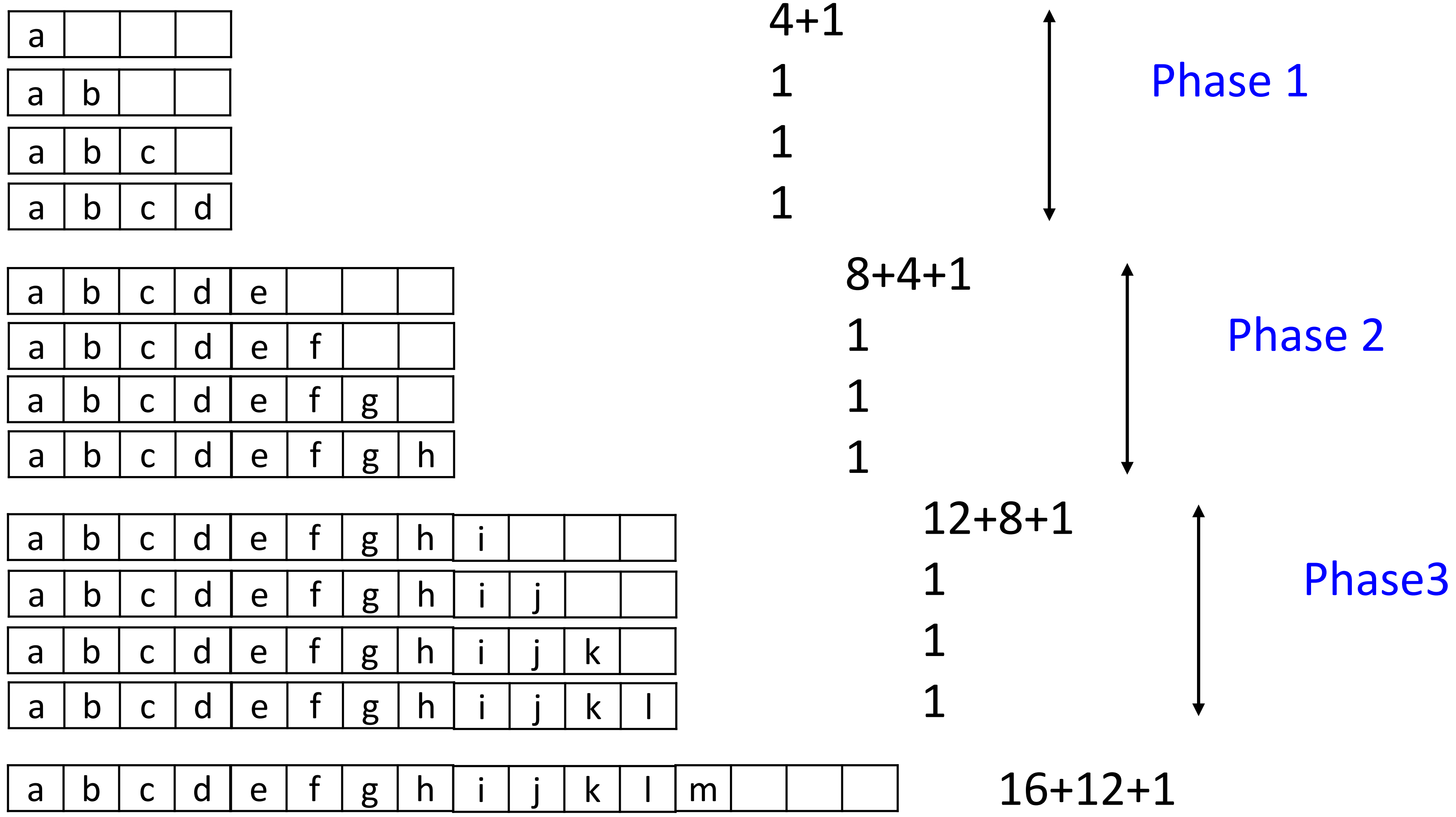
Tight Strategy (c=4)

Start with an array of size 0. Cost of special push is 2N+5.



Tight Strategy (c=4)

Start with an array of size 0. Cost of special push is 2N+5.



Performance of the Tight Strategy

- In phase i the array has size c×i

Performance of the Tight Strategy

- In phase i the array has size $c \times i$
- Total cost of phase i is:
  - $c \times i$ is the cost of creating the array
  - $c \times (i-1)$ is the cost of copying elements into new array
  - c is the cost of c pushes
- Hence, the cost of phase i is 2ci.

Performance of the Tight Strategy

- In phase i the array has size c×i
- Total cost of phase i is:
  - c×i is the cost of creating the array
  - c×(i-1) is the cost of copying elements into new array
  - c is the cost of c pushes
- Hence, the cost of phase i is 2ci.

- In each phase we do c pushes. Hence for n pushes, we need n/c phases.

Performance of the Tight Strategy

- In phase i the array has size c×i
- Total cost of phase i is:
  - c×i is the cost of creating the array
  - c×(i-1) is the cost of copying elements into new array
  - c is the cost of c pushes
- Hence, the cost of phase i is 2ci.

- In each phase we do c pushes. Hence for n pushes, we need n/c phases.
- Total cost of these n/c phases is:

  $= 2c(1+2+3+....+n/c) = O(n^2/c)$

Growth Strategy

Start with an array of size 0. Cost of special push is 3N+1.

$\boxed{a}$ 1+0+1 $\updownarrow$ Phase 1

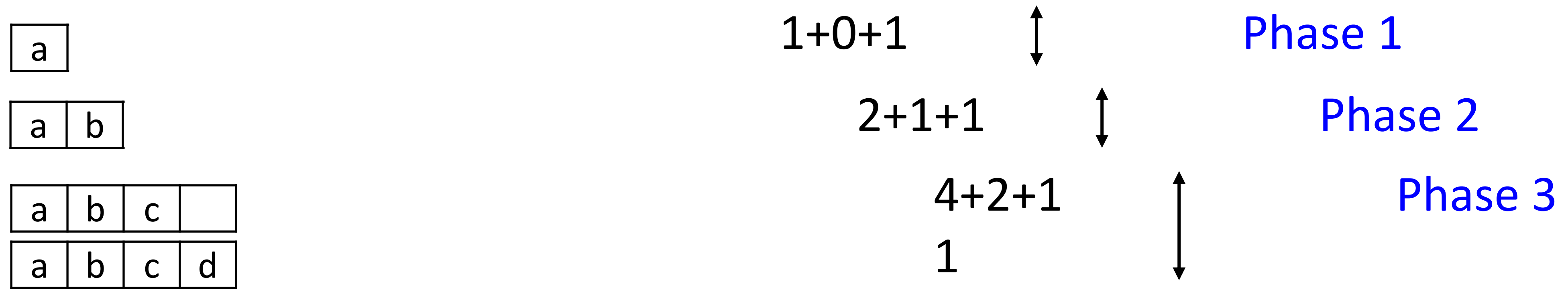
Growth Strategy

Start with an array of size 0. Cost of special push is 3N+1.

$\boxed{a}$	1+0+1	$\updownarrow$	Phase 1
$\boxed{a}\boxed{b}$	2+1+1	$\updownarrow$	Phase 2

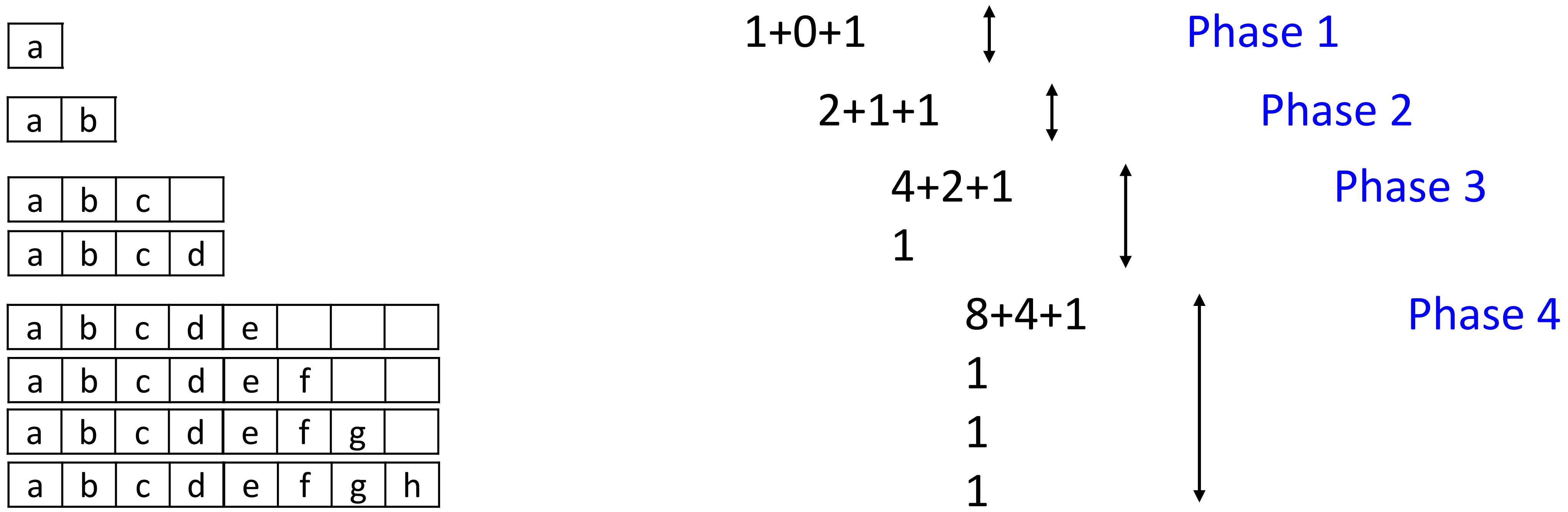
Growth Strategy

Start with an array of size 0. Cost of special push is 3N+1.



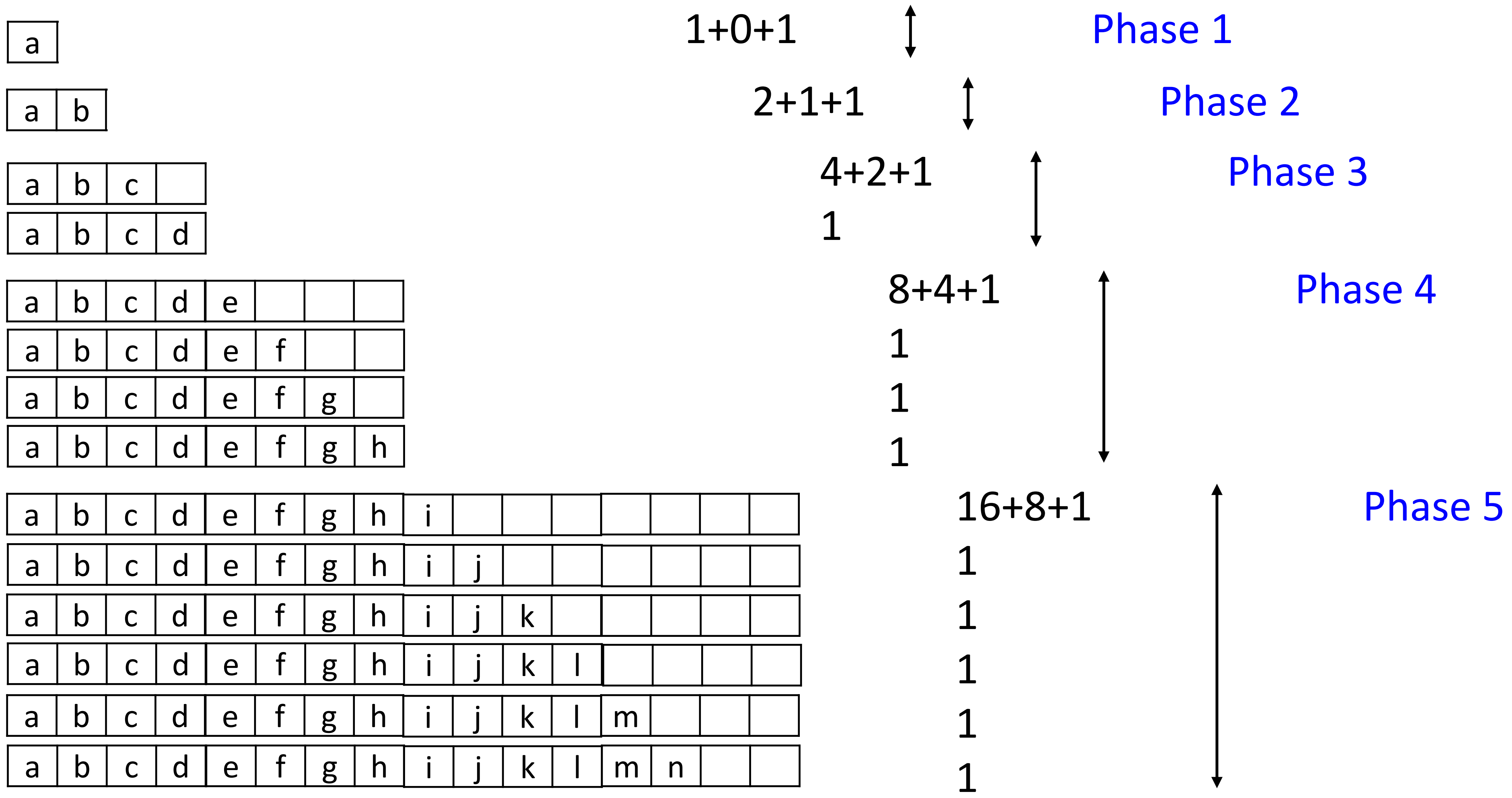
Growth Strategy

Start with an array of size 0. Cost of special push is 3N+1.



Growth Strategy

Start with an array of size 0. Cost of special push is 3N+1.



Performance of the Growth Strategy

- In phase i the array has size 2^i

Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is:
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of 2^{i-1} pushes done in this phase
- Hence, the cost of phase i is 2^{i+1}.

Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is:
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of 2^{i-1} pushes done in this phase
- Hence, the cost of phase i is 2^{i+1}.

- If we do n pushes we will have log n phases.
- Total cost of n pushes is:

  $= 2+4+8+\dots+2^{\log n+1} = 4n-1$

Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is:
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of 2^{i-1} pushes done in this phase
- Hence, the cost of phase i is 2^{i+1}.

- If we do n pushes we will have log n phases.
- Total cost of n pushes is:

  $= 2+4+8+\dots+2^{\log n+1} = 4n-1$
- **The growth strategy wins!**

Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML

- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

Infix and Postfix Notations

- Infix: operators placed between operands:

  A+B*C

- Postfix: operands appear before their operators:-

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(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 → (A + (B * C)) → (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 +

Infix to Postfix Conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

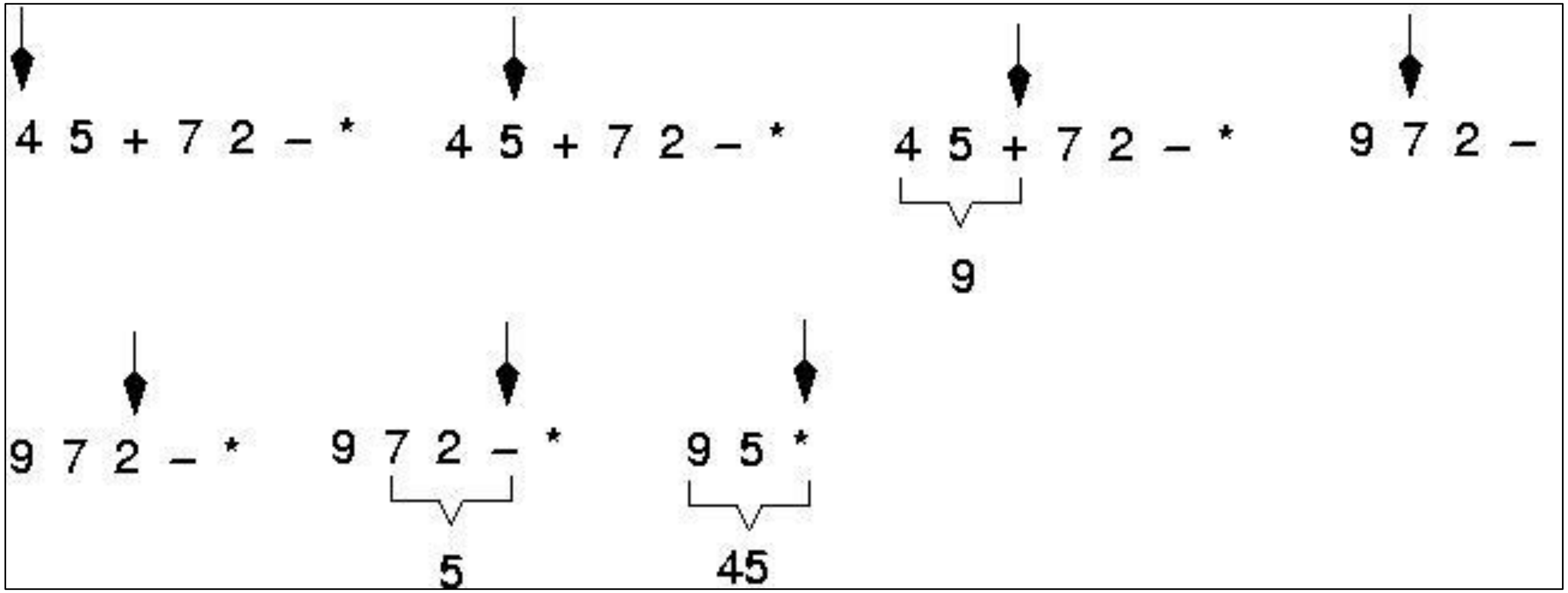
becomes

A B C D * + * E +

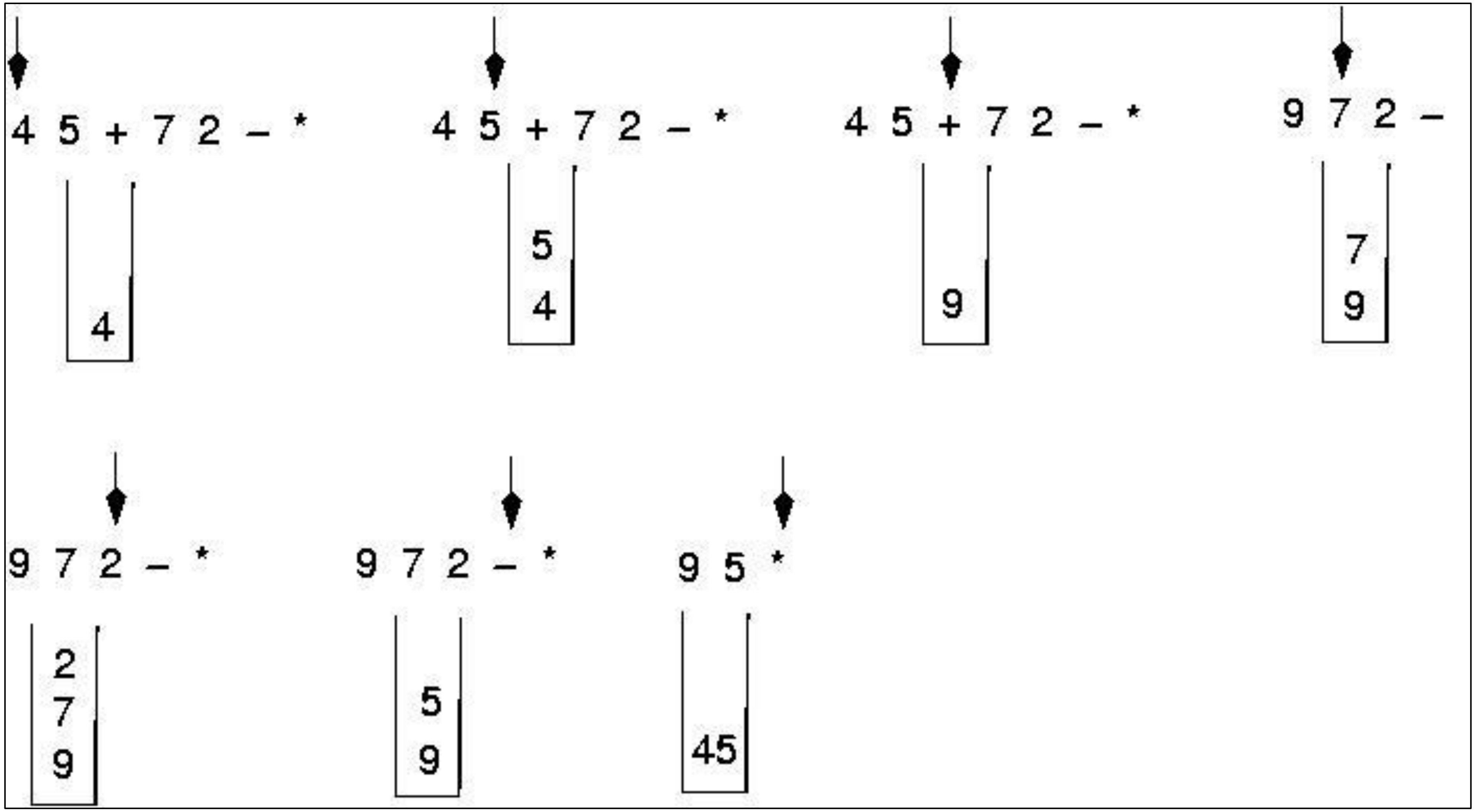
Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Postfix Expression Evaluation



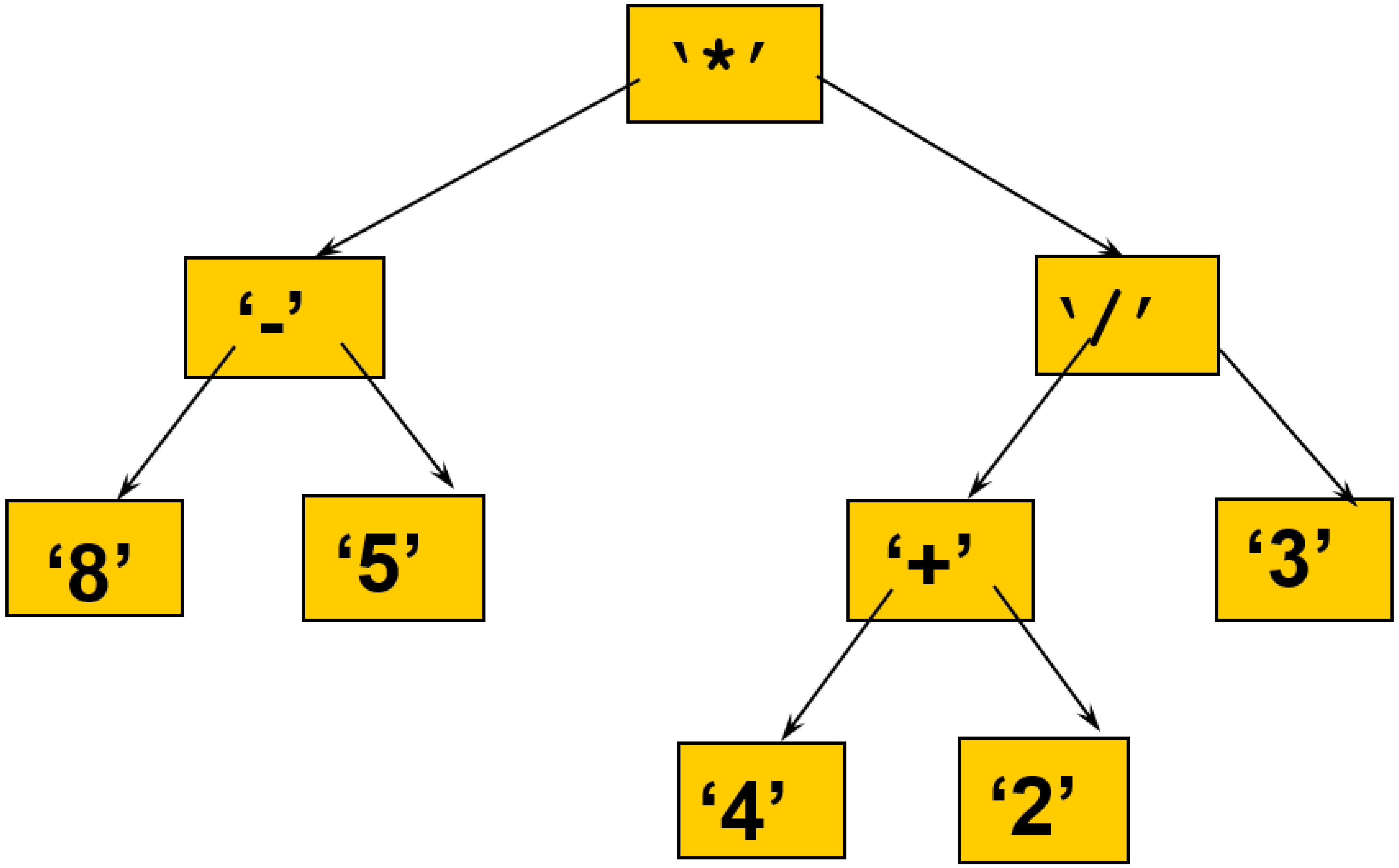
Postfix Expression Evaluation using Stack



Binary Expression Tree

1. Each **leaf node** contains a single operand
2. Each **nonleaf node** contains a single binary operator
3. The left and right subtrees of an operator node represent **subexpressions** that must be evaluated **before** applying the operator at the root of the subtree.

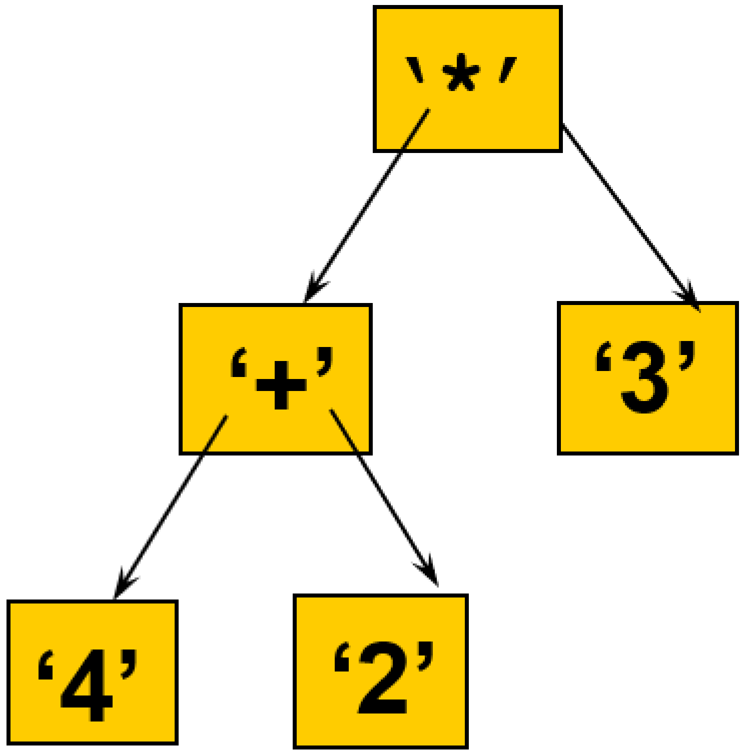
A Four-Level Binary Expression



Levels Indicate Precedence

- **The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).**
- **Operations at higher levels of the tree are evaluated later than those below them.**
- **The operation at the root is always the last operation performed.**

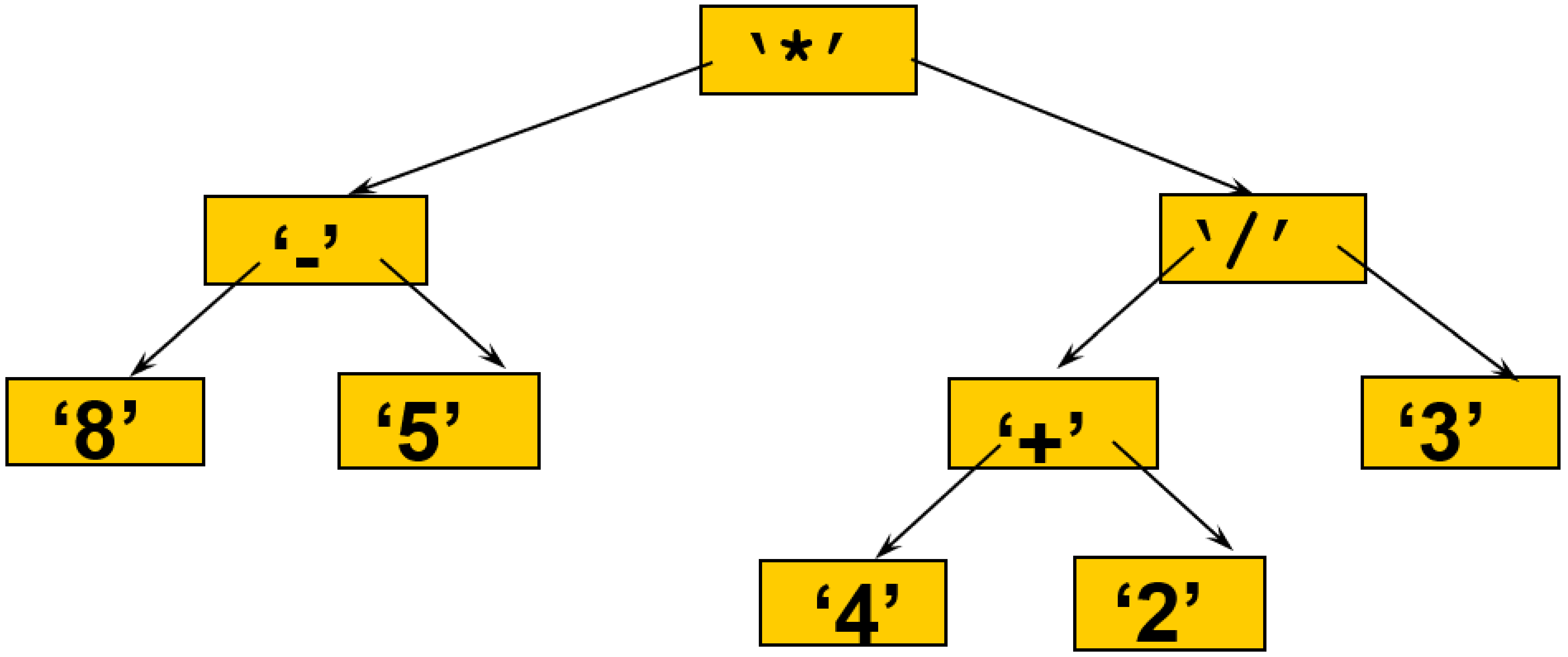
A Binary Expression Tree



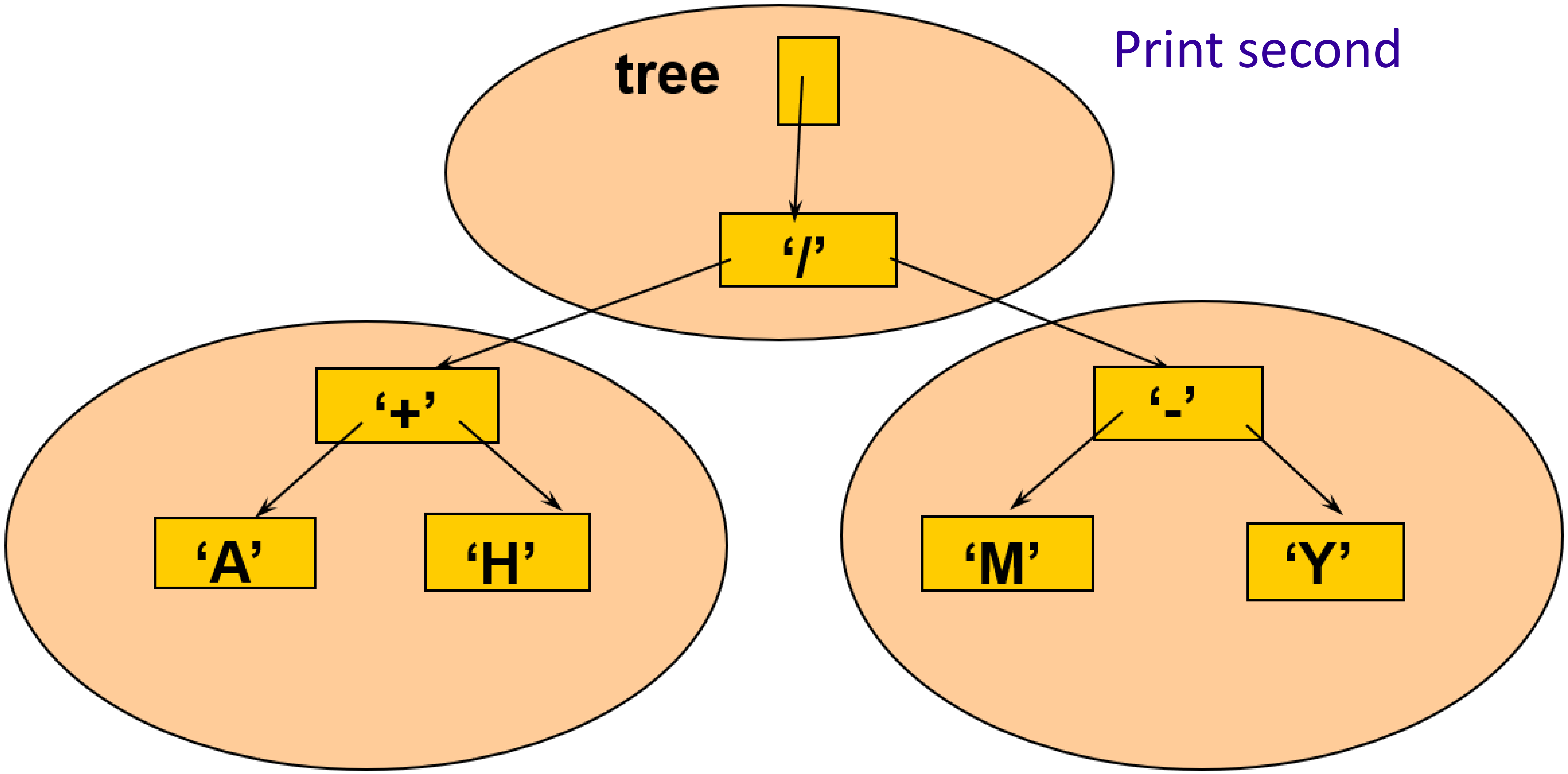
What value does it have?

(4 + 2) * 3 = 18

Easy to generate the infix, prefix, postfix expressions (how?)



Inorder Traversal: (A + H) / (M - Y)

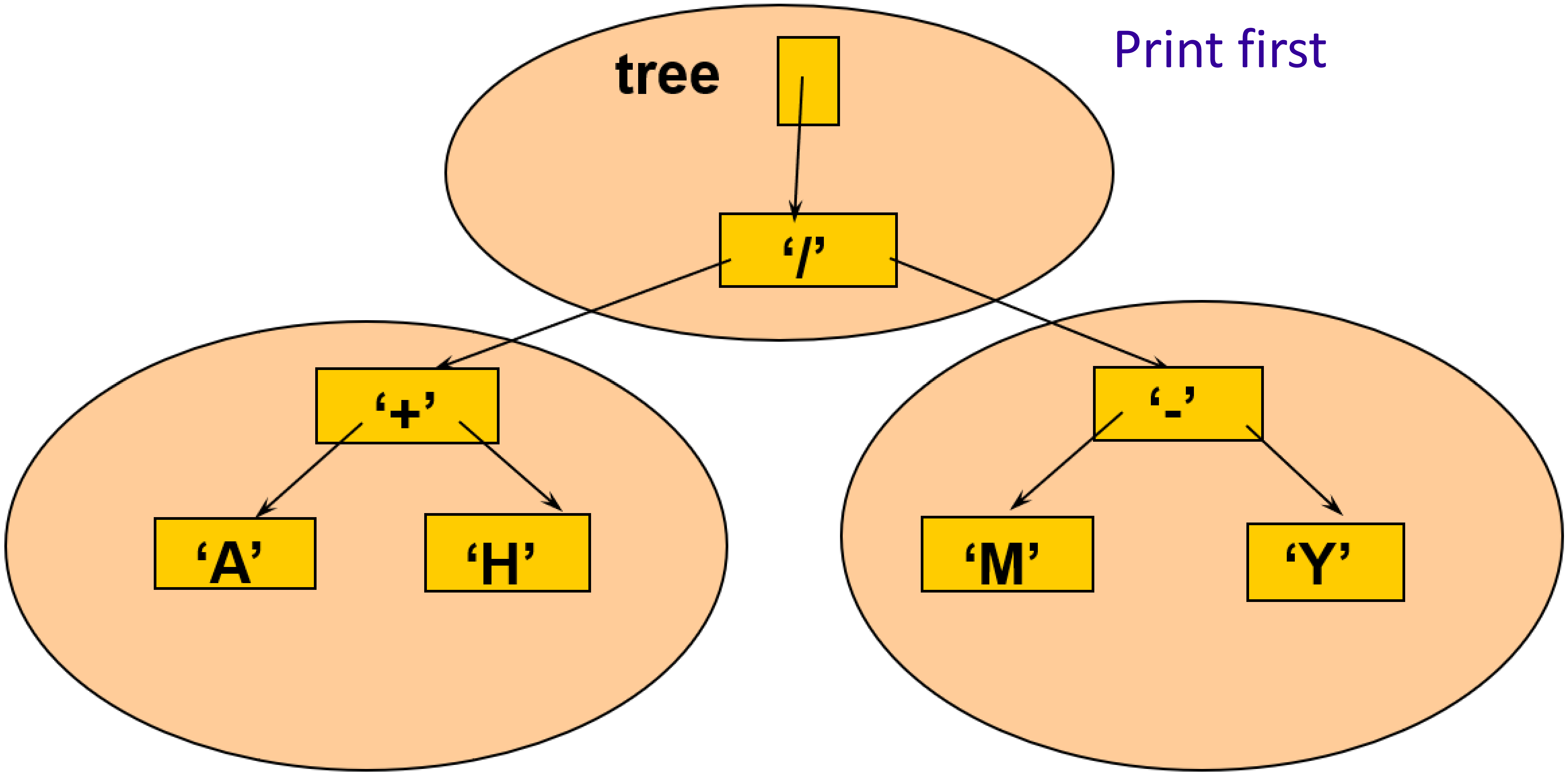


Print second

Print left subtree first

Print right subtree last

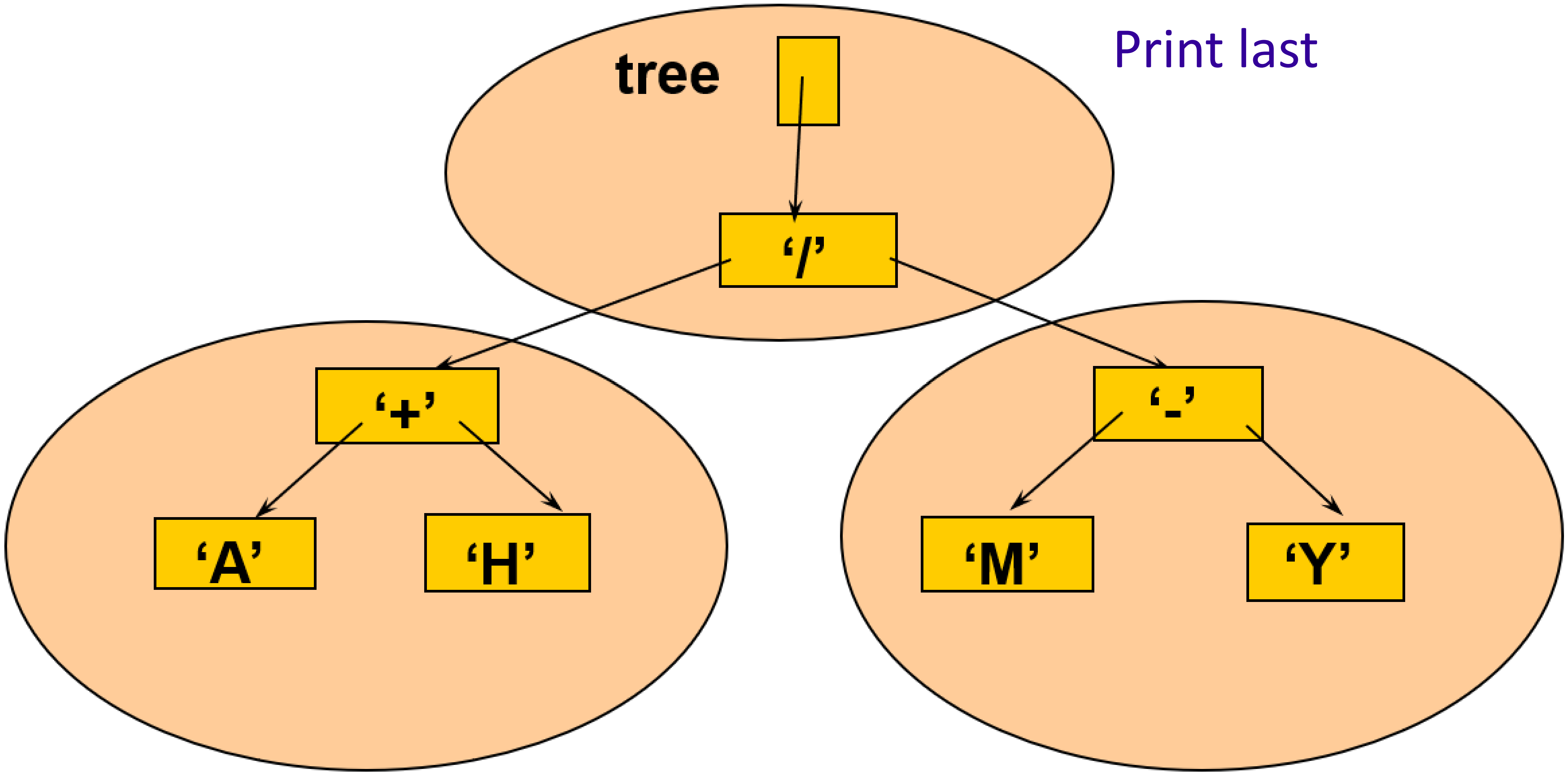
Preorder Traversal: / + A H - M Y



Print left subtree second

Print right subtree last

Postorder Traversal: A H + M Y - /

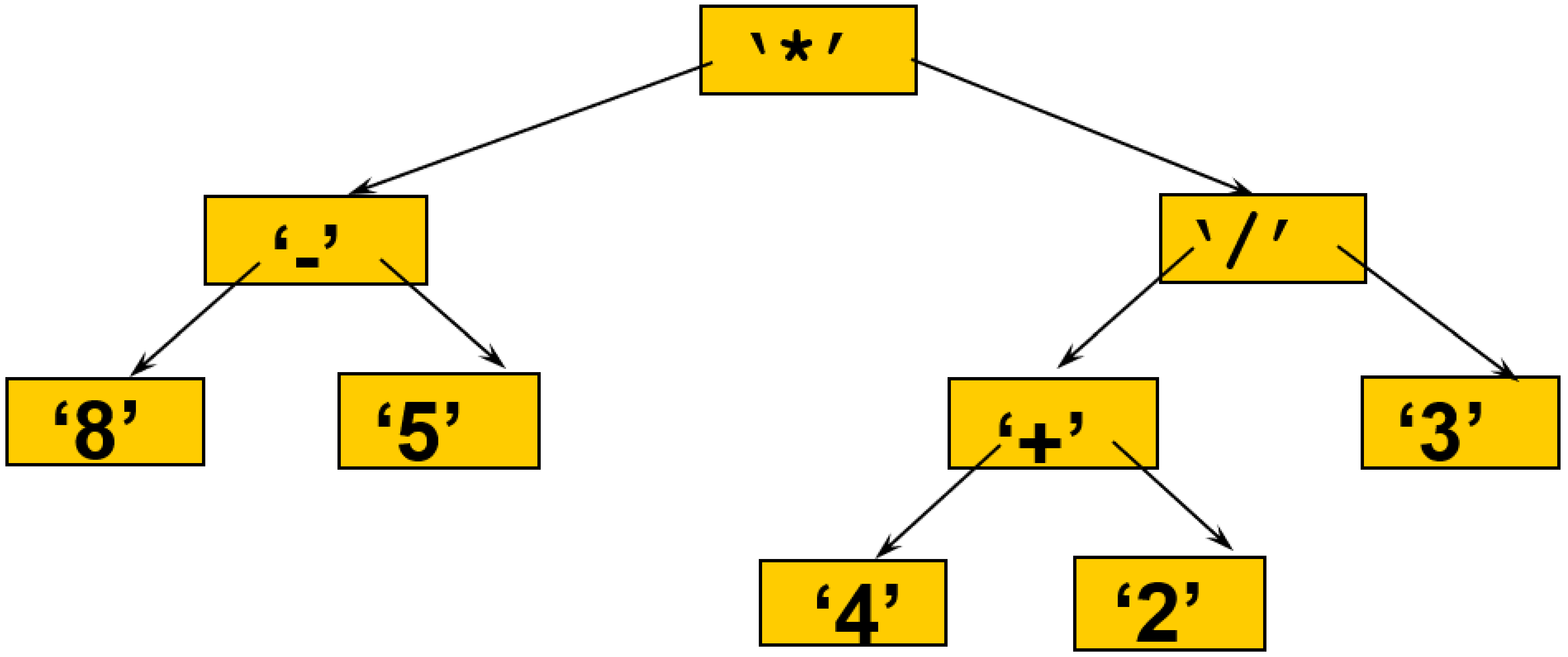


Print last

Print left subtree first

Print right subtree second

The infix, prefix, postfix expressions



Infix: ((8 - 5) * ((4 + 2) / 3))

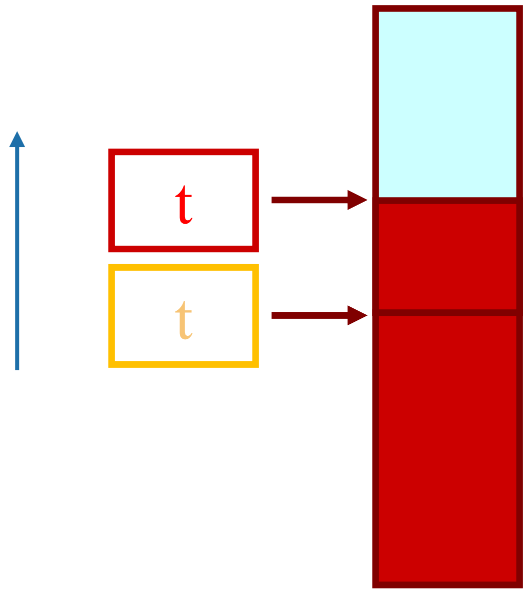
Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

Building a Binary Expression Tree from an expression in prefix notation

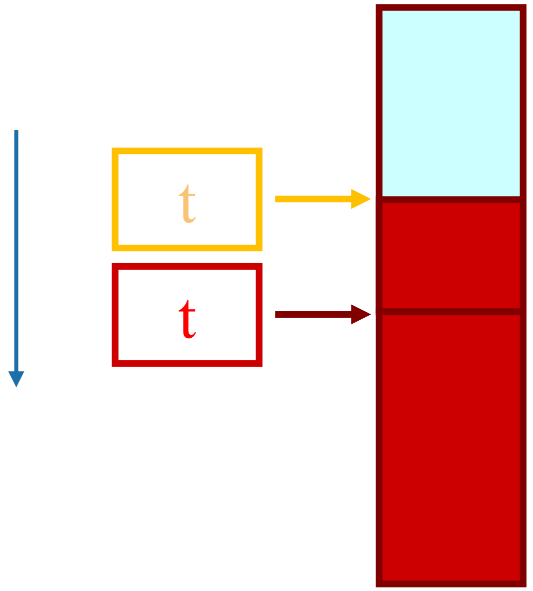
- Insert new nodes, each time moving to the left until an operand has been inserted.
- Backtrack to the last operator, and put the next node to its right.
- Continue in the same pattern.

Push using Stack



PUSH

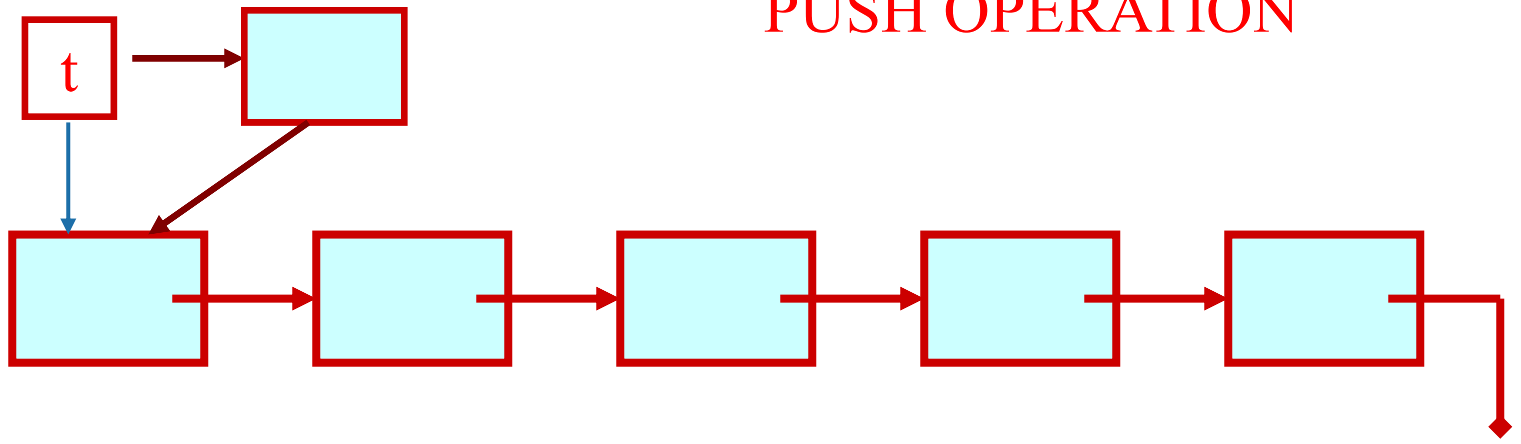
Pop using Stack



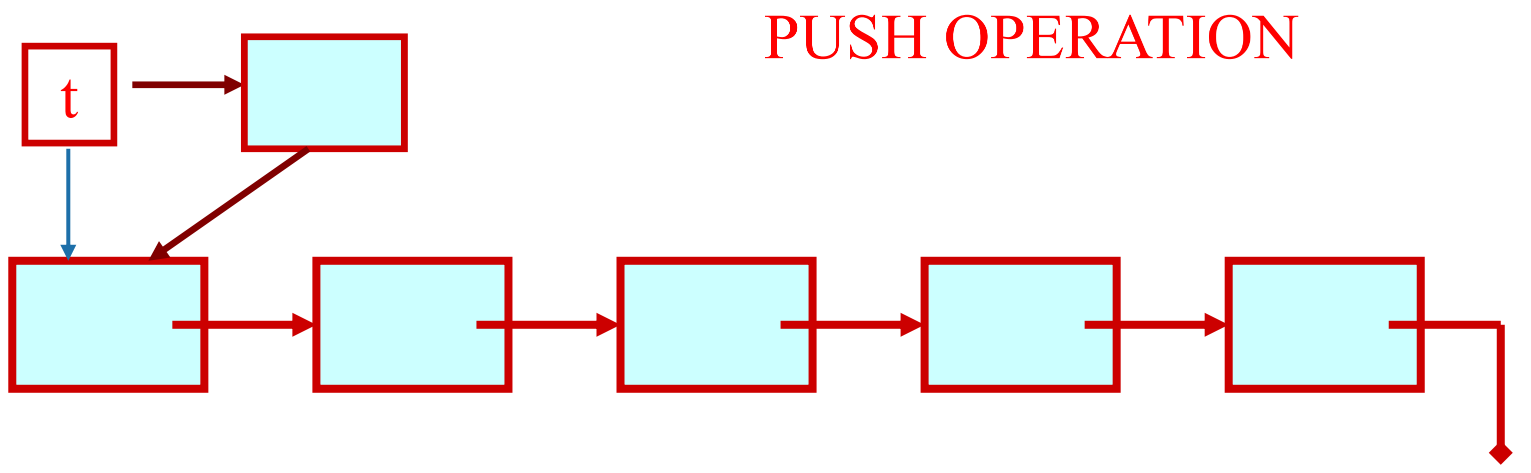
POP

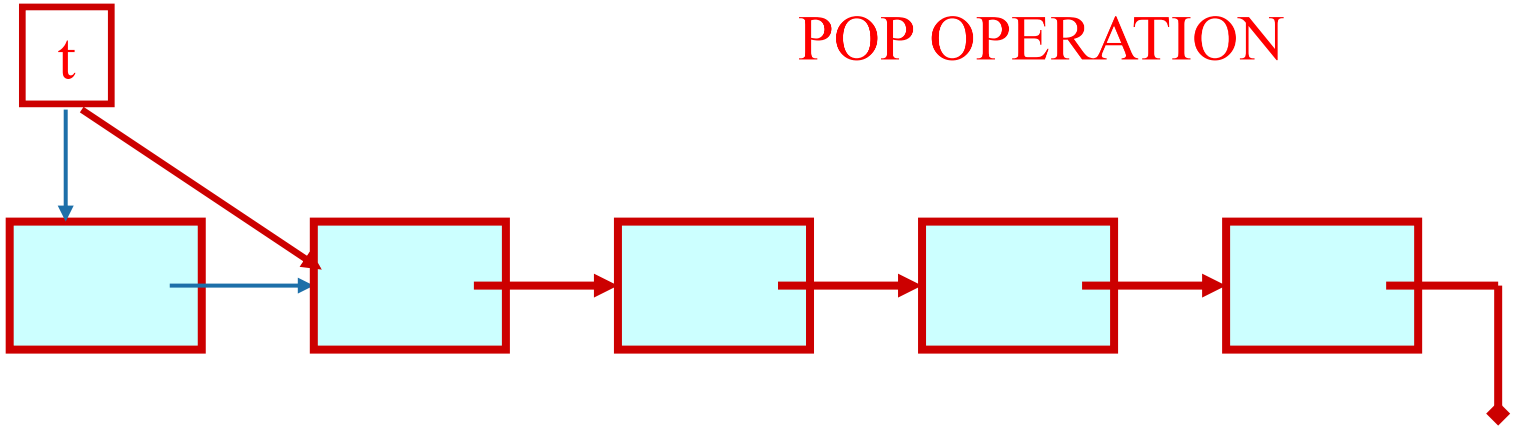
Stack using Linked List

PUSH OPERATION



Stack using Linked List





Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the "top" of the stack.
    - Contains the array index of the "top" element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Stack: Declaration

```
#define MAXSIZE 100

struct lifo
{
    int  st[MAXSIZE];
    int  t;
};
typedef struct lifo
                 stack;
stack s;
```

ARRAY

```
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo
                 stack;

stack *t;
```

LINKED LIST

Stack: Creation

```
void new(stack *s)
{
    s->t = -1;

    /* s->t points to
       last element
       pushed in;
       initially -1 */
}
```

ARRAY

```
void new(stack **t)
{
    *t = NULL;

    /* top points to NULL,
       indicating empty
       stack            */
}
```

LINKED LIST

Stack: Pushing an element into stack

```
void push (stack *s, int element)
{
  if (s->t == (MAXSIZE-1))
  {
    printf ("\n Stack overflow");
    exit(-1);
  }
  else
  {
    s->t++;
    s->st[s->t] = element;
  }
}
```

ARRAY

```
void push (stack **t, int element)
{
  stack *new;
  new = (stack *)malloc
                  (sizeof(stack));
  if (new == NULL)
  {
    printf ("\n Stack is full");
    exit(-1);
  }

  new->value = element;
  new->next = *t;
  *t = new;
}
```

LINKED LIST

Stack: Popping an element from stack

```
int pop (stack *s)
{
   if (s->t == -1)
   {
      printf("\n Stack
                     underflow");
      exit(-1);
   }
   else
   {
      return (s->st[s->t--]);
   }
}
```

ARRAY

```
int pop (stack **t)
{
 int tv; stack *p;

 if (*t == NULL)
 {
    printf("\n Stack is empty");
    exit(-1);
 }
 else
 {
    tv = (*t)->value;
    p = *t;
    *t = (*t)->next;
    free (p);
    return tv;
 }
}
```

LINKED LIST

Stack: Return top element from stack

```
int top (stack *s)
{
  if (s->t == -1)
  {
    printf("\n Stack
                  underflow");
    exit(-1);
  }
  else
  {
    return (s->st[s->t]);
  }
}
```

ARRAY

```
int top (stack **t)
{

 if (*t == NULL)
 {
   printf("\n Stack is
                    empty");
   exit(-1);
 }
 else
 {
   return (*t)->value;
 }
}
```

LINKED LIST

Stack: Checking for stack empty

```
int isEmpty (stack *s)
{
   if (s->t == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isEmpty (stack *t)
{
   if (t == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Stack using Array: Example

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int t;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A); create(&B);
  push(&A,10); push(&A,20); push(&A,30);
  push(&B,100); push(&B,5);

  printf (“%d %d”, pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf (“\n B is empty”);
  return;
}
```

Stack using Linked List: Example

```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;

main() {
  stack *A, *B;
  create(&A); create(&B);
  push(&A,10); push(&A,20); push(&A,30);
  push(&B,100); push(&B,5);

  printf (“%d %d”, pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf (“\n B is empty”);
  return;
}
```

Acknowledgement

- IIT Delhi
- IIT Kharagpur

Thank You