CSN 102: DATA STRUCTURES

Stack: Stack fundamentals, implementation of recursion

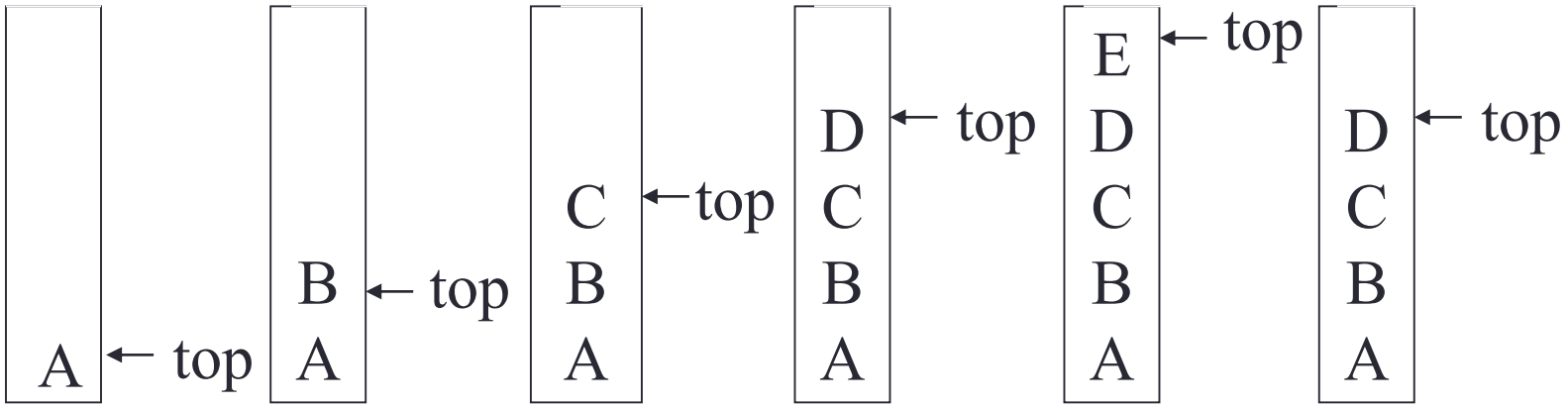
Abstract Data Type (ADT)

- An Abstract Data Type is:
  - Set of values
  - Set of operations which can be uniformly applied to these values
  - Set of Axioms

What is Stack?

- Stores the elements in particular way
- Last In First Out (LIFO)
- push(key): inserts the element key at top of stack
- pop(): deletes the element from top of stack

Last In First Out (LIFO)



Stack Abstract Data Type

- Stack ADT has:
  - Values based on what kind of data stack stores
  - Main operations:
    - new() : creates a new stack
    - push(S, key): inserts element key at top of stack S
    - pop(S): deletes element from top of stack S
    - top(S): returns the top element of stack S without deleting it
  - Supported operations:
    - isEmpty(S): checks whether stack S is empty or not
    - isFull(S): checks whether stack S is full or not
  - Axioms:
    - pop(push(S,v)) = S
    - top(push(S,v)) = v

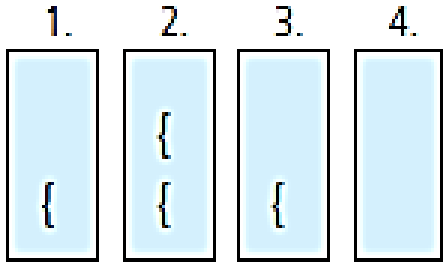
Stack Operations

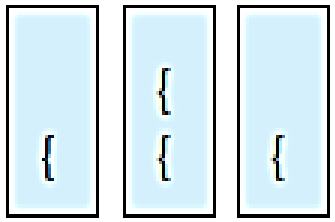
```
push(S, key)
        if (stack is not full)
                increase top by 1
                insert key at top

pop(S)
        if (stack is not empty)
                key = delete element at top
                decrease top by 1
                return (key)
```

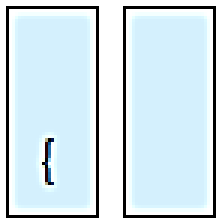
Stack application: Balanced Braces

Traces of algorithm checking balanced braces

Input string	Stack as algorithm executes	

{a{b}c}		1. push {
		2. push {
		3. pop
		4. pop
		Stack empty ⟹ balanced

{a{bc}		1. push {
		2. push {
		3. pop
		Stack not empty ⟹ not balanced

{ab}c}		1. push { 2. pop Stack empty when next "}" encountered ⟹ not balanced

Postfix Expression

- Evaluation of postfix expression is simple and can be performed in single pass
- Operator is after its respective operands
- infix expression postfix expression
- A *op* B A B *op*
- 2+3*4 234*+
- (2+3)*4 23+4*

Infix to postfix conversion: Intuitive

=> If parenthesis exist in infix expression, first performed above conversion inside parenthesis

1. Fully parenthesized expression

   a / b - c + d * e - a * c => ((((a / b) - c) + (d * e)) – (a * c))

2. All operators replace their corresponding right parentheses.

   ((((a / b) - c) + (d * e)) – (a * c))

3. Delete all parentheses.

   ab/c-de*+ac*-

Infix to postfix conversion: Pseudocode

```
while( not end of input ) {
        c = next input character
        if( c is an operand )
                add c to postfix string
        else if ( empty(s) ) {
                push ( s, c )
        }
        else {
                while( !empty(s) && prcd(top(s),c) ) {
                        op = pop(s)
                        add op to the postfix string
        }
                push( s, c ) }
}
while( !empty(s) ) {
        op = pop(s)
        add op to postfix string
}
```

Infix to postfix conversion: Precedence

- prcd(a, b): returns TRUE
  - If precedence of a is greater than b
  - If a and b has same precedence and are left associative
  - Eg. prcd (*, +), prcd (/, *)

- prcd(a, b): returns FALSE
  - If precedence of a is less than b
  - If a and b has same precedence and are right associative
  - Eg. prcd(-, /), prcd (^, ++)

Infix to postfix conversion: Example (1)

Traces of algorithm for converting infix A+B*C/D-E to postfix

Next Symbol	*Postfix String*	*Stack*
A	A	
+	A	+
B	A B	+
*	A B	+ *
C	A B C	+ *
/	A B C *	+ /
D	A B C * D	+ /
-	A B C * D / +	-
E	A B C * D / + E	-
	A B C * D / + E -	

Infix to postfix conversion: Pseudocode

```
while( not end of input ) {
        c = next input character
        if( c is an operand )
                add c to postfix string
        else if ( c is "(" )   {       push (s, c);                    }
         else if ( c is ")" )
        {       while( top(s)!="(") {
                        op = pop(s)
                        add op to the postfix string
                }
        pop(s) }
        else if ( empty(s) ) {
                push ( s, c )
        }
        else {
                while( !empty(s) && prcd(top(s),c) ) {
                        op = pop(s)
                        add op to the postfix string
                }
        push( s, c )
        }
}
while( !empty(s) ) {
        op = pop(s)
        add op to postfix string
}
```

Infix to postfix conversion: Example (2)

Traces of algorithm for converting infix (A+B*(C-D))/E to postfix

Next Symbol	*Postfix String*	*Stack*
(		(
A	A	(
+	A	(+
B	A B	(+
*	A B	(+ *
(	A B	(+ * (
C	A B C	(+ * (
-	A B C	(+ * (-
D	A B C D	(+ * (-
)	A B C D -	(+ *
)	A B C D - * +	
/	A B C D - * +	/
E	A B C D - * + E	/
	A B C D - * + E /	

Exercise

- Convert Below infix expression to postfix expression
- A / B ^ C + D * E – A * C

- (A / B ^ (C + D)) * (E – A * C)

Exercise

- Convert Below infix expression to postfix expression
- A / B ^ C + D * E – A * C
- A B C ^ / D E * + A C * –

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

Postfix Evaluation

- Again an application of stack
- Read postfix expression from left to right
  - If next character is operand, push to stack
  - If next character is operator,
    - pop the top element and store in operand2
    - pop the top element and store in operand1
    - push result of expression operand1 operator operand2 into stack
- Stop when postfix expression is empty
- pop the top element which is results

Postfix evaluation : Example(1)

Infix : (36 – 12) / 4 = 6

postfix: 36 12 – 4 /

Next Symbol	Action	Stack
36	push 36	36
12	push 12	36 12
-	pop; operand2 = 12 pop; operand1 = 36 Result = operand1 - operand2 = 36 – 12 = 24 push 24	24
4	push 4	24 4
/	pop; operand2 = 4 pop; operand1 = 24 Result = operand1 / operand2 = 24 / 4 = 6 push 6	6

Postfix evaluation : Example(2)

Infix : 4 / 2 – 2 + 3 * 3 – 4 * 2 = 1

Postfix evaluation : Example(2)

Infix : 4 / 2 – 2 + 3 * 3 – 4 * 2 = 1

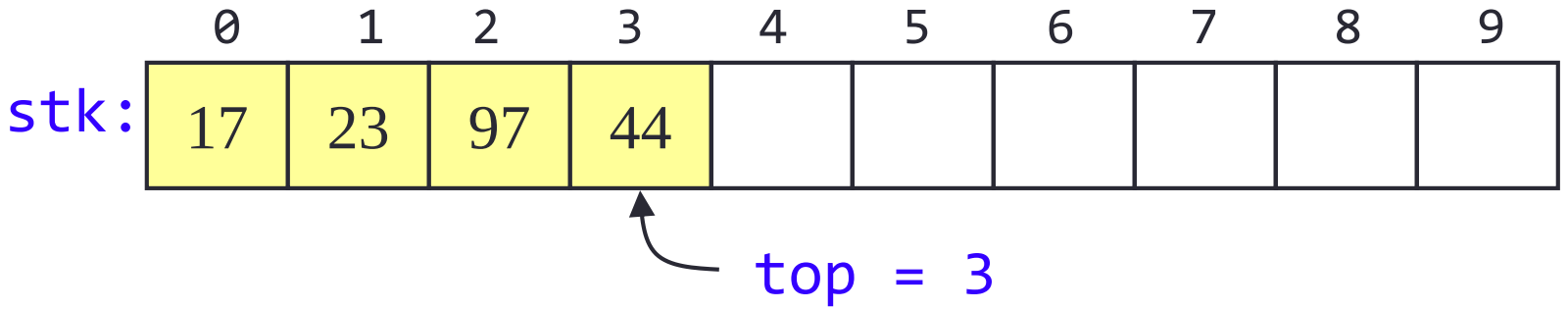
postfix: 4 2 / 2 – 3 3 * + 4 2 * –

Next Symbol	Action	Stack
4	push (4)	4
2	push (2)	4 2
/	op2 = 2; op1 = 4; 4 / 2 = 2; push (2)	2
2	push (2)	2 2
–	op2 = 2; op1 = 2; 2 – 2 = 0; push (0)	0
3	push (3)	0 3
3	push (3)	0 3 3
*	op2 = 3; op1 = 3; 3 * 3 = 9; push (9)	0 9
+	op2 = 9; op1 = 0; 0 + 9 = 9; push (9)	9
4	push (4)	9 4
2	push (2)	9 4 2
*	op2 = 2; op1 = 4; 4 * 2 = 8; push (8)	9 8
–	op2 = 8; op1 = 9; 9 – 8 = s1; push (1)	1

Array Implementation of Stack

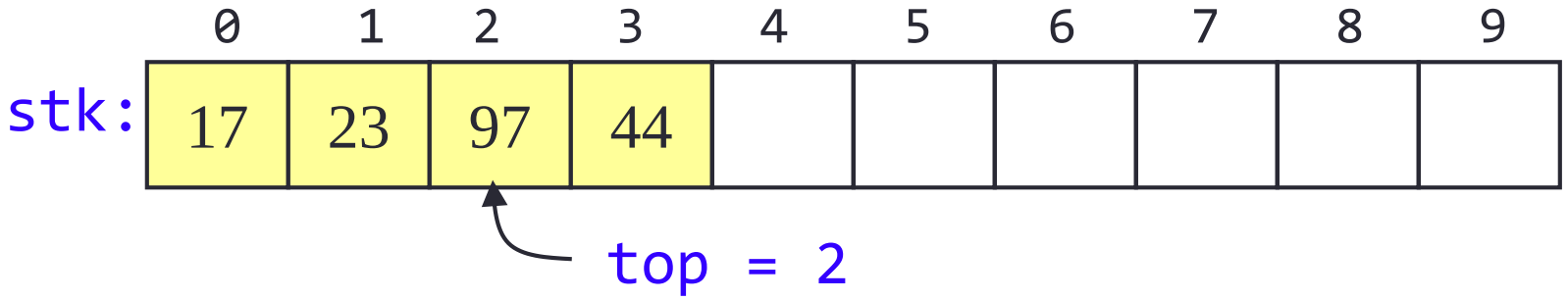
- To implement a stack, items are inserted and removed at the same end (called top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer top which tells the location of element at top of the stack

push and pop Operations



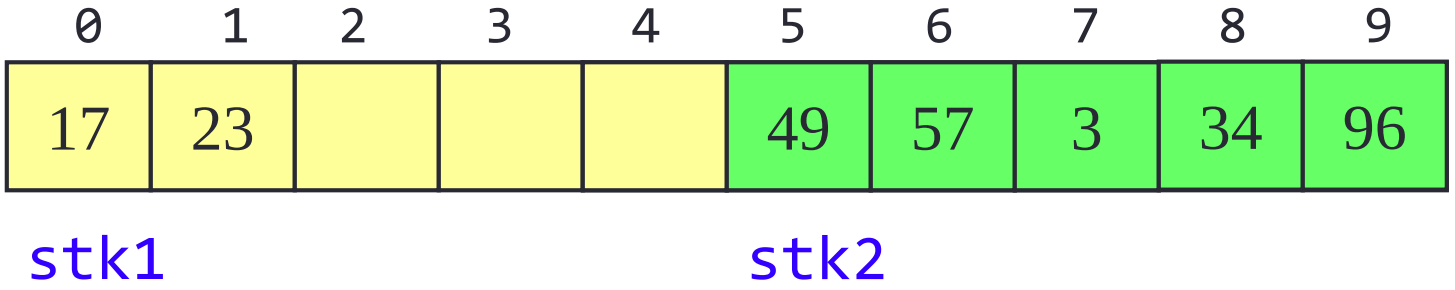
- push elements {17, 23, 97, 44} in stk
- perform a pop operation

push and pop Operations



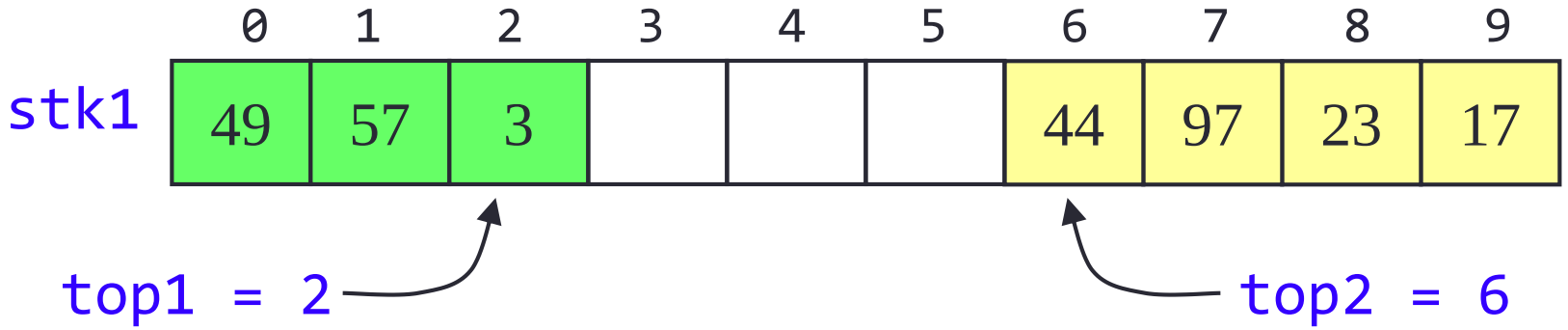
- When we pop an element, do you just leave the “deleted” element sitting in the array?
- The surprising answer is, *“it depends”*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the “deleted” array element to `null`
  - Why? To allow it to be garbage collected!

Implementing Two Stacks



- Initially top1 = -1 and top2 = 10
- Overflow condition for stk1; top1 = 4
- Overflow condition for stk2; top2 = 9
- Even if locations are empty, no more insertion possible in stk2

Implementing Two Stacks



- Initially top1 = -1 and top2 = 10
- Overflow condition for stk1; top1 = top2 – 1
- Overflow condition for stk2; top2 = top1 + 1
- Insertion possible until all the locations are not filled

Prefix Expression

- Operator is before its respective operands
- infix expression prefix expression
- A *op* B *op* A B
- 2+3*4 +2*34
- (2+3)*4 *+234

Infix to prefix conversion: Intuitive

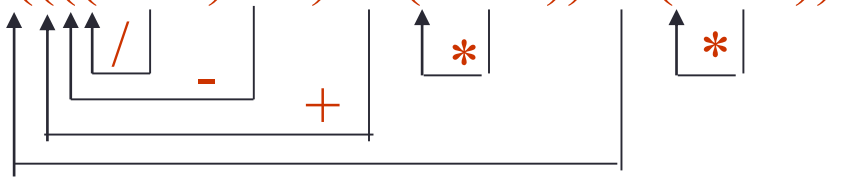
=> If parenthesis exist in infix expression, first performed conversion inside parenthesis

1. Fully parenthesized expression

   a / b - c + d * e - a * c => ((((a / b) - c) + (d * e)) – (a * c))

2. All operators replace their corresponding right parentheses.

   ((((a / b) - c) + (d * e)) – (a * c))



3. Delete all parentheses.

   -+-/abc*de*ac

Implementation of Recursion

```
BinarySearch(a, start, end, key)
        if (start > end)
                return
        middle ← (start + end)/2
        if (a[middle] == key);
        else if (a[middle] > key)
                BinarySearch(a, start, middle-1, key)
        else
                BinarySearch(a, middle+1, end, key)
        print a[middle]
```

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

BS(a,0,9,45)

If (0 > 9)
M=(0+9)/2=4
a[4] > 45

Main()

Program Stack

Output:

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

BS(a,0,9,45)

If (0 > 9)
M=(0+9)/2=4
a[4] > 45

BS(a,0,3,45)

If (0 > 3)
M=(0+3)/2=1
a[1] < 45

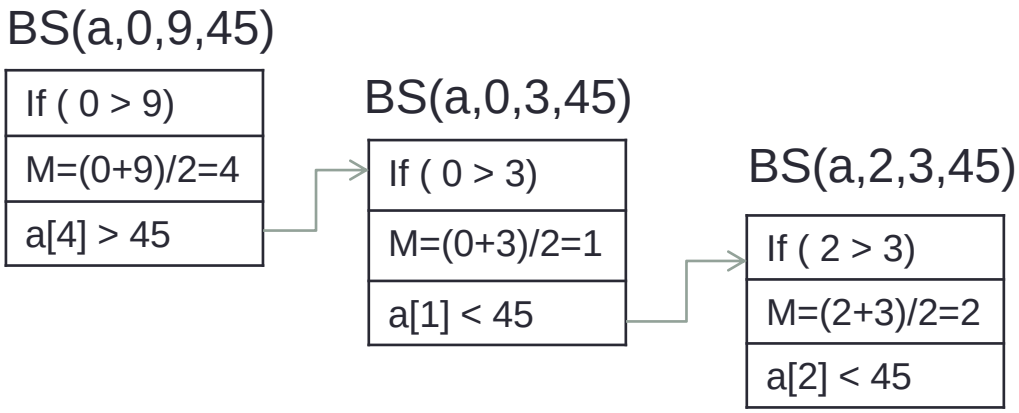
BS(a,0,9,45)
Main()

Program Stack

Output:

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

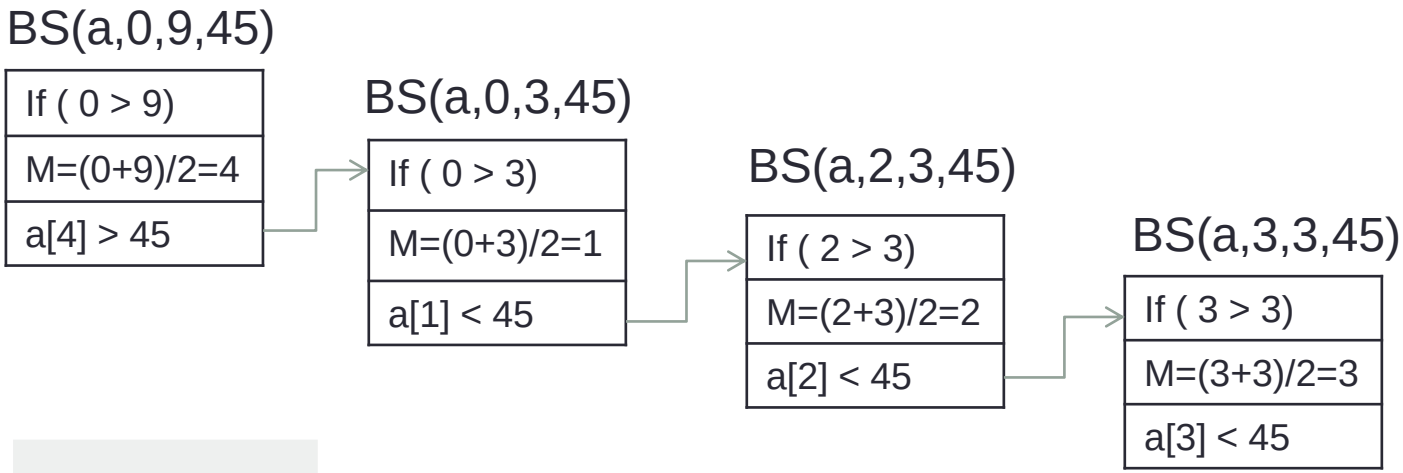


Program Stack

Output:

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

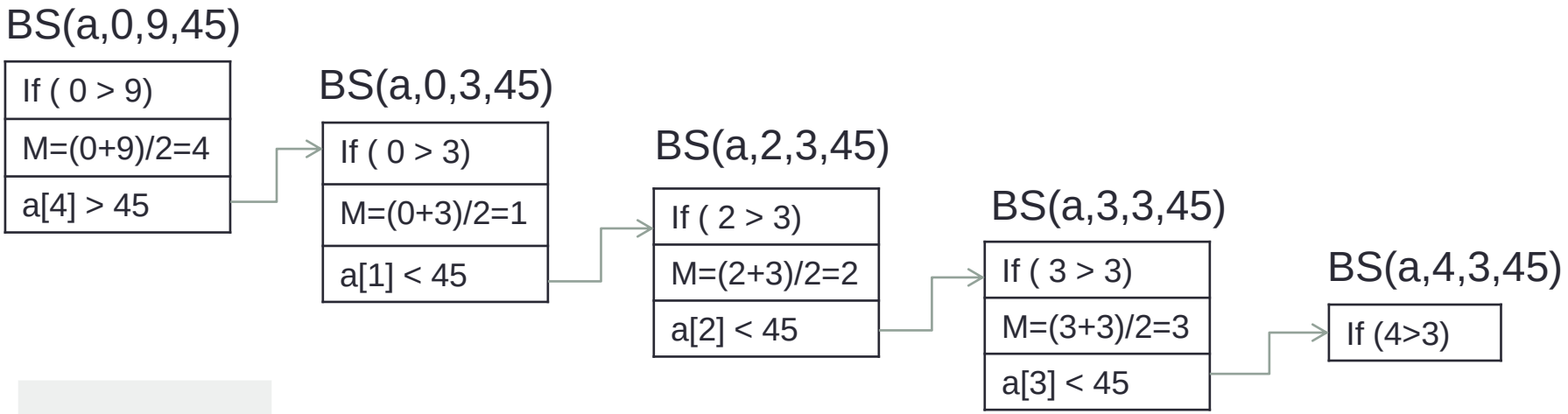


Program Stack

Output:

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

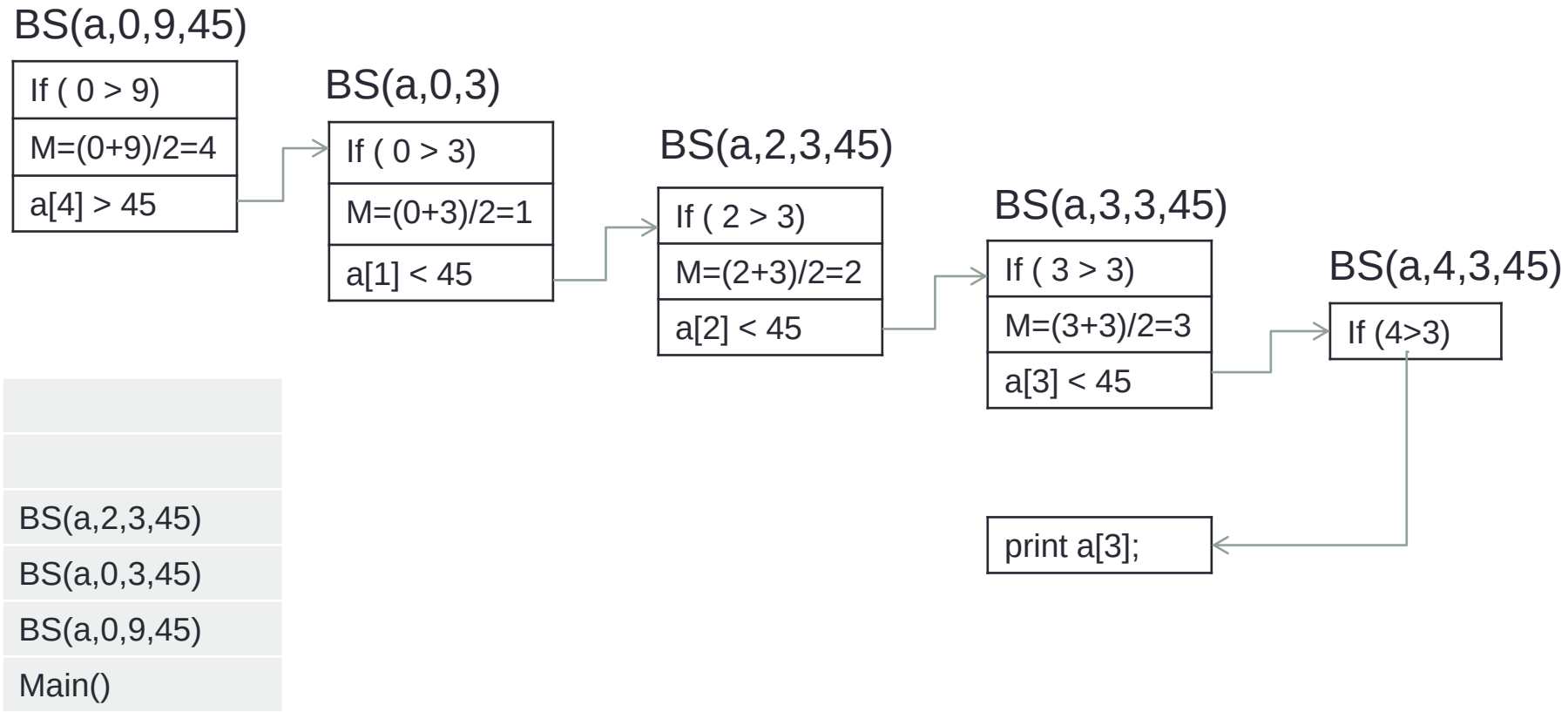


Program Stack

Output:

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

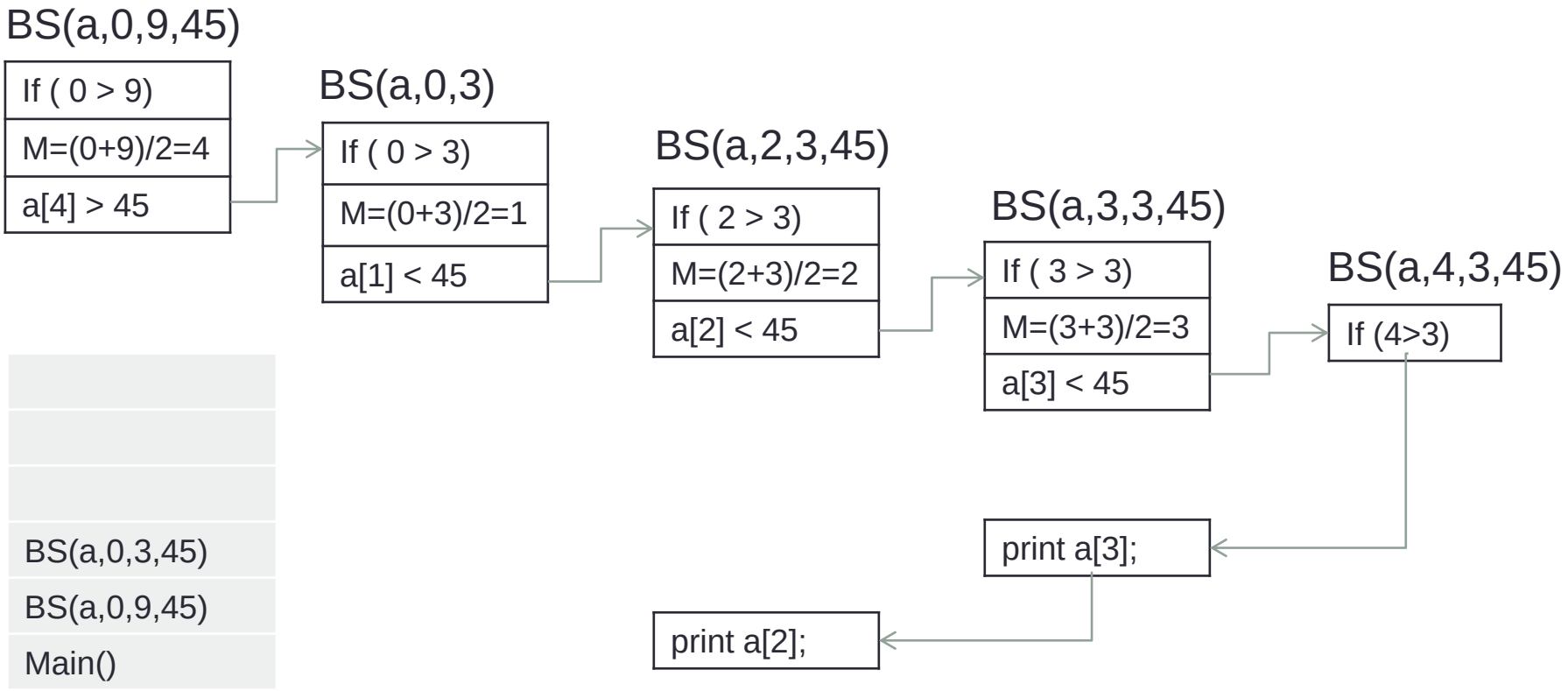


Program Stack

Output: 40

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list

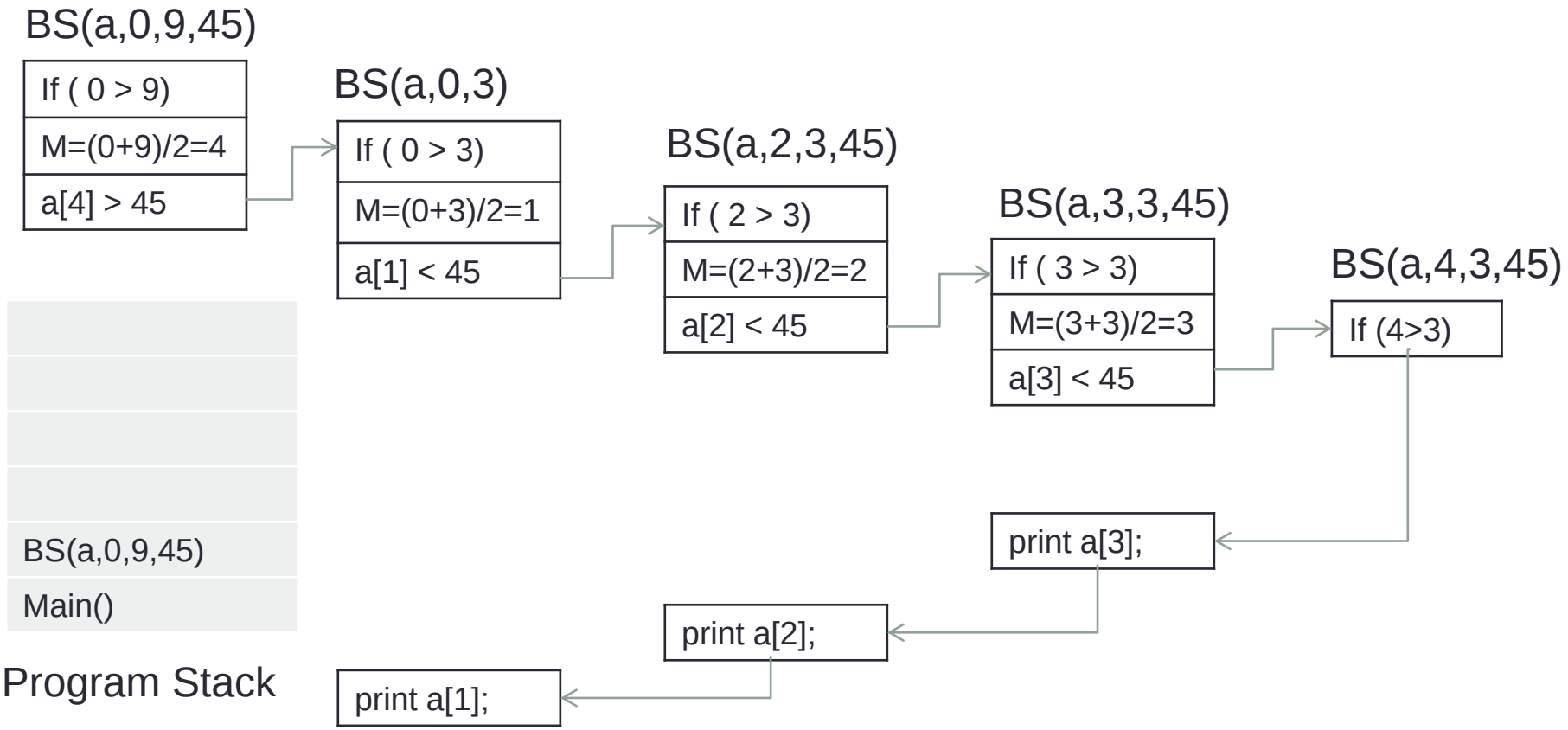


Program Stack

Output: 40 30

a	10	20	30	40	50	60	70	80	90	100

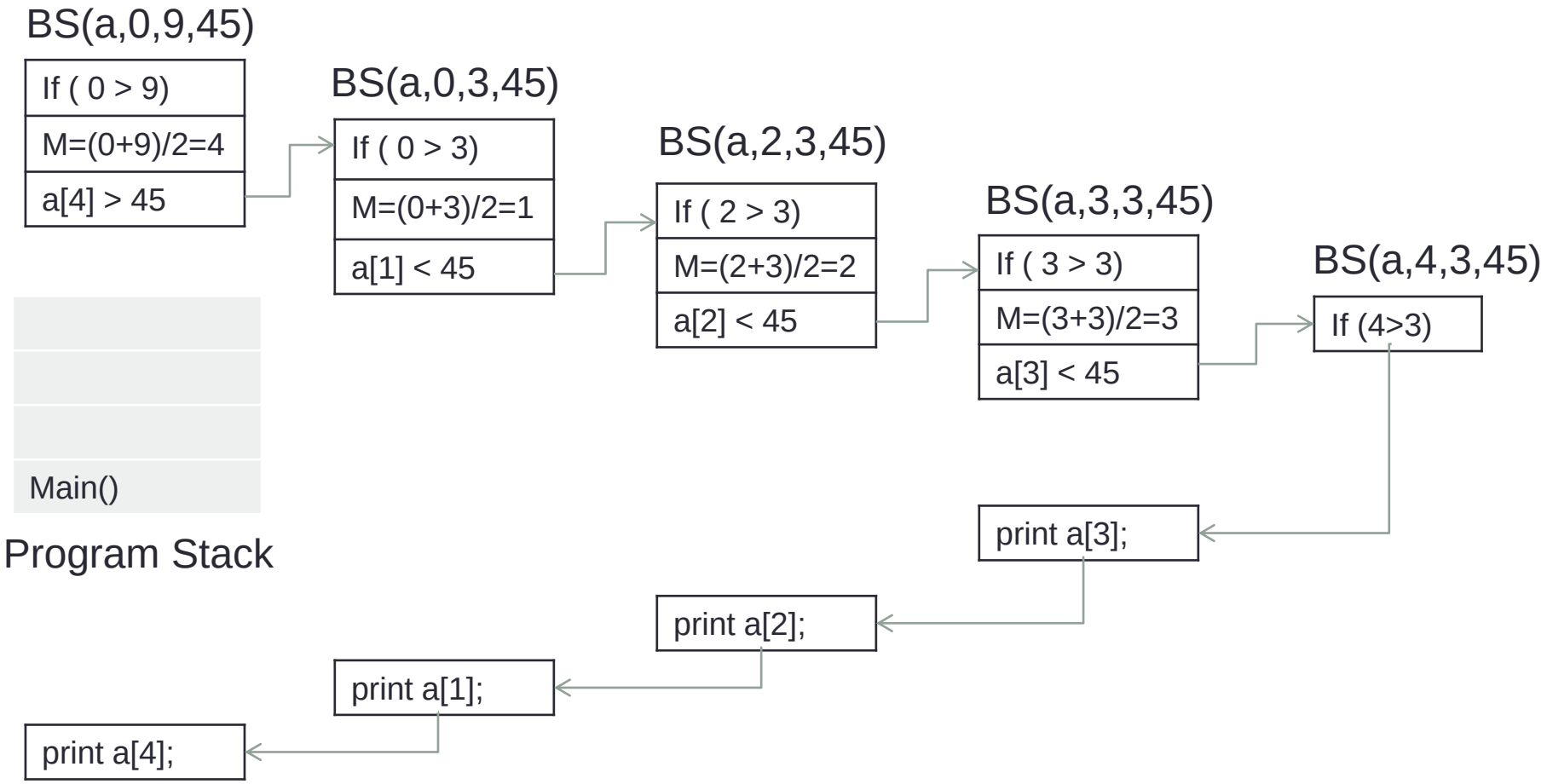
- Search for 45 in above list



Output: 40 30 20

a	10	20	30	40	50	60	70	80	90	100

- Search for 45 in above list



Output: 40 30 20 50

Data Structures (MOOC)

- https://www.coursera.org/learn/data-structures

- Register yourself
- Enrol for the course on above link [Click “audit” instead of paying fee]

- Watch Videos of 1st Week.