Stack

Overview

- **Basics**
- **Operations**
- **Implementation using array**

Stack - Basics

- Stack of plates
  - Stacked one on top of the other
  - Topmost plate
  - New plate placed on top, top most plate taken out (normally)

Function Calls / Returns

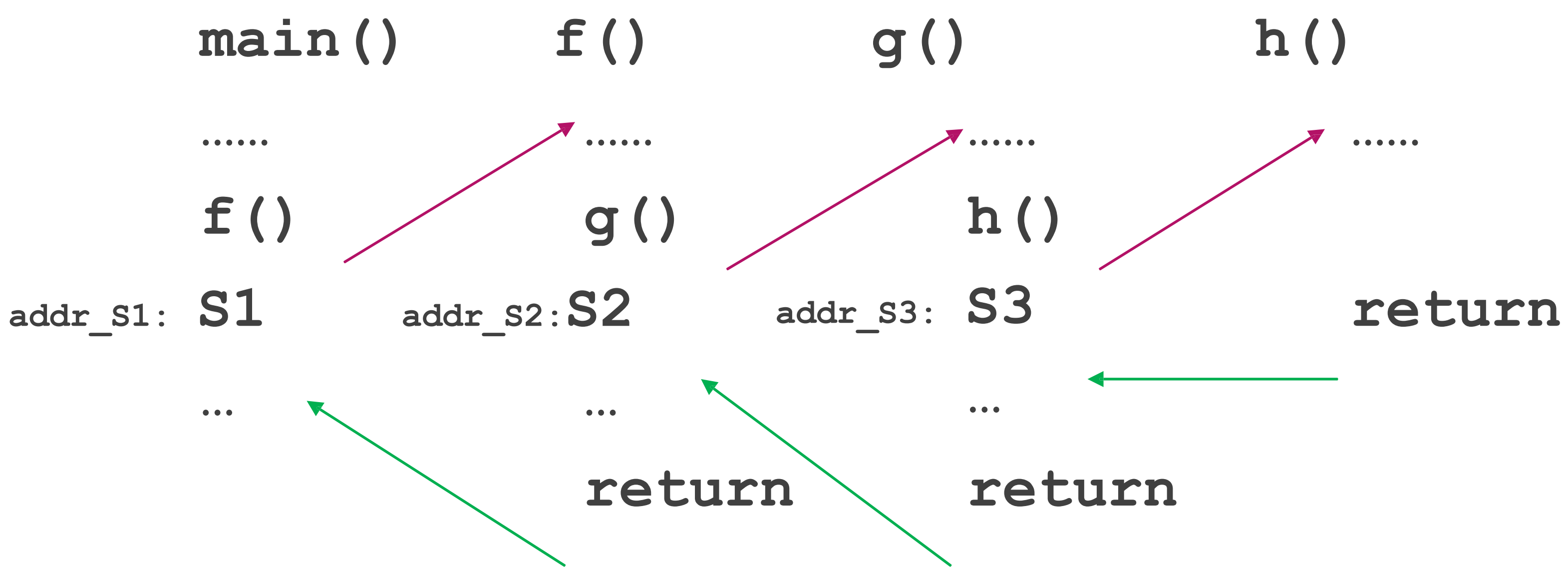
```
Call Sequence :
        main()→  f()→   g()→ h()

Return Sequence:
        main()←  f()←   g()← h()
```

Function Calls / Returns

main()	f()	g()	h()
……	……	……	……
f()	g()	h()	
addr_S1: S1	addr_S2: S2	addr_S3: S3	return
…	…	…	
	return	return	

Return Address : Address of the instruction in the caller function to which control should return

Function Calls / Returns



Upon each call, store Return Address

Upon return, retrieve the last stored address

Return Address

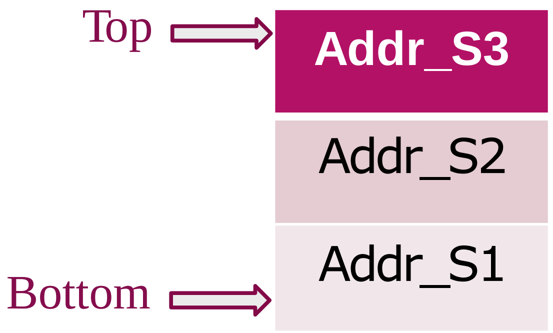
```
          main()        f()           g()           h()
          ......        ......        ......        ......
          f()           g()           h()
addr_S1:  S1    addr_S2:S2    addr_S3: S3           return
          ...           ...           ...
                        return        return
```

Store return addresses: ***addr_S1, addr_S2, addr_S3***

first to be retrieved **:** ***addr_S3***

Stack – Store return addresses

Stack the return addresses

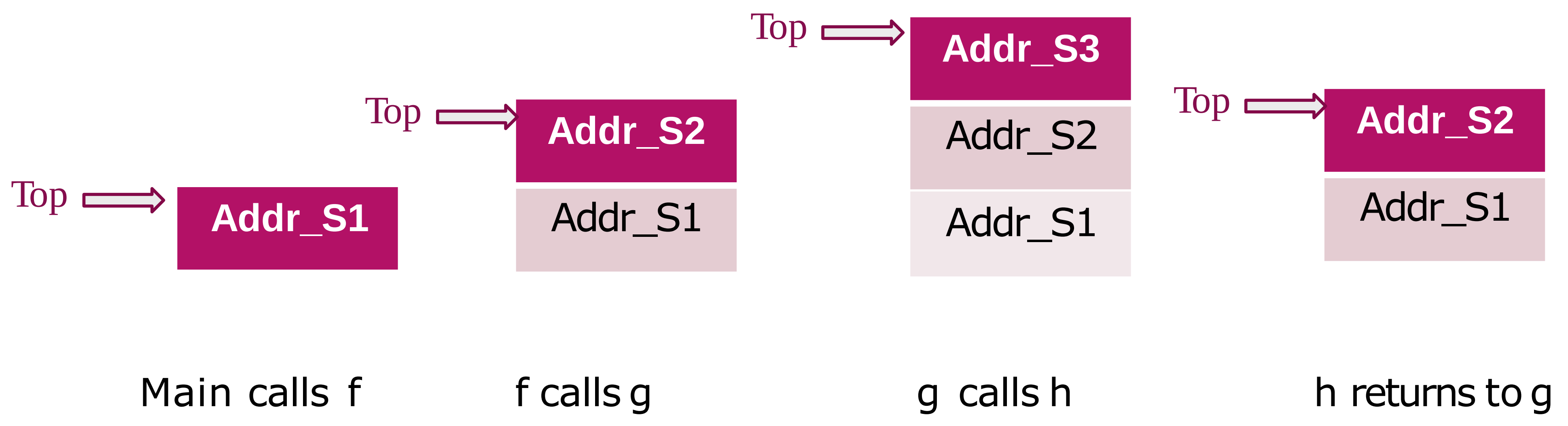


Stack of Activation Records, return address as a field in the AR

Stack - Store return addresses

- Functions calls
  - Return address to be stored
    - In a new call the return address of caller added to the top
  - Returning in the reverse sequence of calls
    - The topmost return address removed first

Stack - Store return addresses



Stack - Basics

- List of elements, INSERT / DELETE only at one end of the list
  - Access restriction
- Last-in, First-out (LIFO)
  - The last inserted element is the first one to be removed

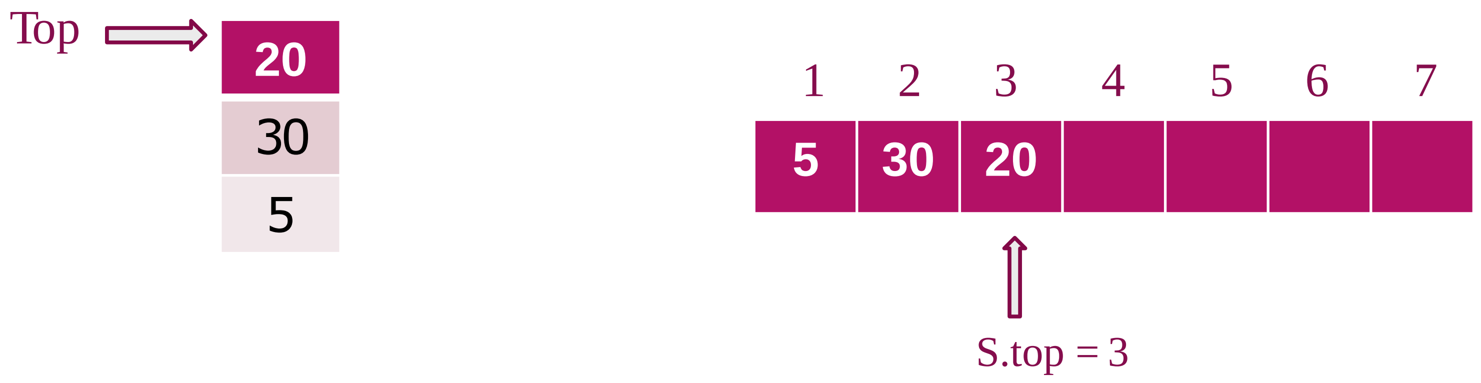
Stack - Operations

- **PUSH() – Insert a new element at the top**
- **POP() – Pop out the top most element (deleted)**

Stack - Implementations

- Array Based
  - S [1..n]
  - Top - an array index
- Pointer Based
  - As a linked list
  - Top - a pointer to the topmost node

Stack – Implementation using Array



Stack – Implementation using Array

1	2	3	4	5	6	7
5	30	20				

PUSH(S, 15)

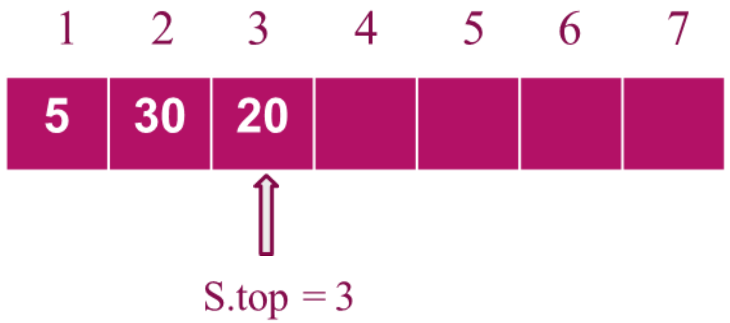
1	2	3	4	5	6	7
5	30	20	15			

POP(S)

1	2	3	4	5	6	7
5	30	20				

Stack - Array Based Implementation

- Array Based
  - Array ***S [1..n]*** : an array of at most n elements
  - An attribute ***S.top*** : index of the top element
  - Elements from ***S[1.. S.top]***
  - ***S[1]*** : element at the bottom
  - ***S[S.top]*** : element at the top

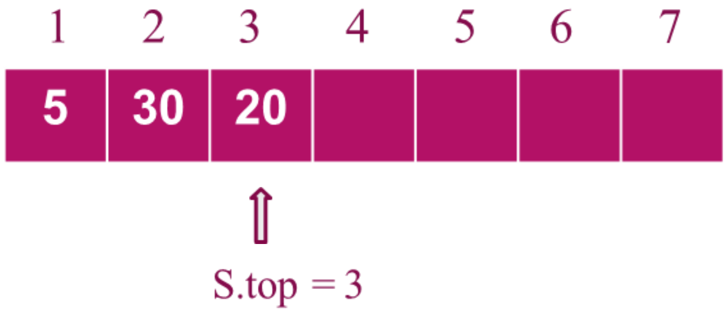


Stack - Array Based Implementation

PUSH (S, x)

S. top = S. top + 1

S [S. top] = x



Stack - Array Based Implementation

POP (S)

- Pops out the topmost element
- If no elements (stack is empty)?
  - POP() has to do an initial check to ensure that Stack has at least one element

Stack - Array Based Implementation

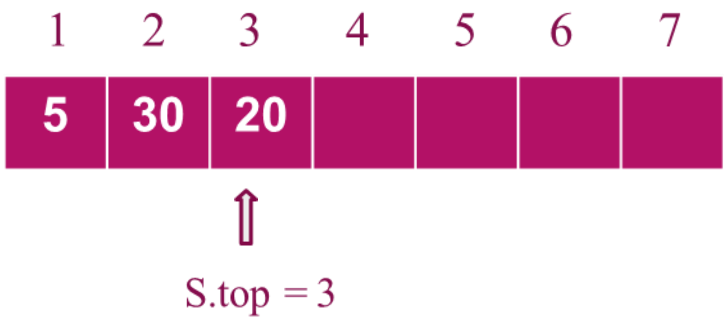
POP (S)

If (STACK-EMPTY (S))

error “underflow”

else S. top = S. top - 1

return S[S. top+1]



Stack - Array Based Implementation

```
STACK-EMPTY(S)
    If S. top == 0
        return TRUE
    else
      return FALSE
```

Stack - Array Based Implementation

- **Time Complexity of operations**
  - **PUSH ?**
  - **POP ?**
  - **STACKEMPTY() ?**

Stack - Array Based Implementation

- **Time Complexity of operations**
  - **PUSH - O(1)**
  - **POP - O(1)**
  - **STACKEMPTY() – O(1)**

Stack - Array Based Implementation

- **Can the stack be full?**
  - **STACKFULL() required?**

Implementation Details

- **C implementation – as a struct with attributes top and array**

```
struct stack {
        int top;
        int   elems [100]
        }
```

Implementation Details

- **C implementation – as a struct with attributes top and array**

```
struct Stack {
    int top;
    ElemType  elems [100]
    };
```

Implementation Details

- **C implementation – as a struct with attributes top and array**

```
struct Stack {
    int top;
    ElemType  *elems;
    };
```

Implementation Details

```
struct Stack {
    int top;
    ElemType  *elems;
    };

    PUSH(struct Stack * s, ElemType x){
        ….
    }
```

Stack - Application

- Reverse a string
  - Push each character
  - Pop out (in the reverse order)
- Check if a string is palindrome
- Expression evaluation
- Tree / Graph traversals
- Compilers

Queue

Overview

- **Basics**
- **Operations**
- **Implementation using array**

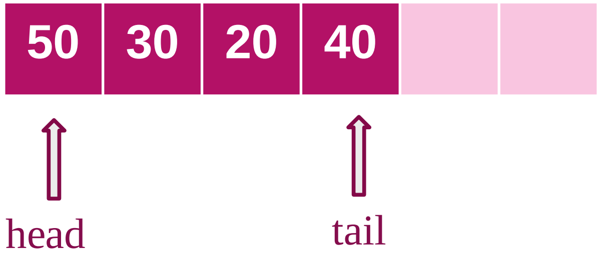
Queue -Basics

- **Queue - List with access restrictions**
  - **FIFO – First-In First-Out**
  - **Double ended- Head and Tail (front and rear)**
  - **Insertion always to the tail**
  - **Deletion always from the head**

Queue -Applications

- Simulating real life scenarios
- Scheduling jobs – print queue
- Tree/ Graph traversal

Queue –Basics



Queue -Operations

- **ENQUEUE(Q, x)**
  - **Add element x to the tail of Queue Q**

- **DEQUEUE(Q)**
  - **Delete the element at the head of Q**

Queue -Implementations

- **Array Based**
  - **Q [1..n]**
  - **Head, Tail - array indices**

- **Pointer Based**
  - **As a linked list**
  - **Head, Tail - pointers to the nodes at front and rear respectively**

Queue - Array Based Implementation #1

- Array Based
  - Array ***Q [1..n]*** - an array of at most n elements
  - An attribute ***Q.head*** – index of the head element
  - An attribute ***Q.tail*** – index of the tail element
  - Elements from ***Q[Q.head..Q.tail]***

Queue –Implementation using Array #1

1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40								

⇧ Q.head = 1

⇧ Q.tail = 4

Queue – Implementation using Array #1

1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40								

⇧ Q.head = 1 ⇧ Q.tail = 4

Enqueue(Q, 10)

1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40	10							

⇧ Q.head = 1 ⇧ Q.tail = 5

Enqueue(Q, 15)

1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40	10	15						

Q.head = 1

Q.tail = 6

Dequeue(Q)

1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40	10	15						

Q.head = 2

Q.tail = 6

Dequeue(Q)

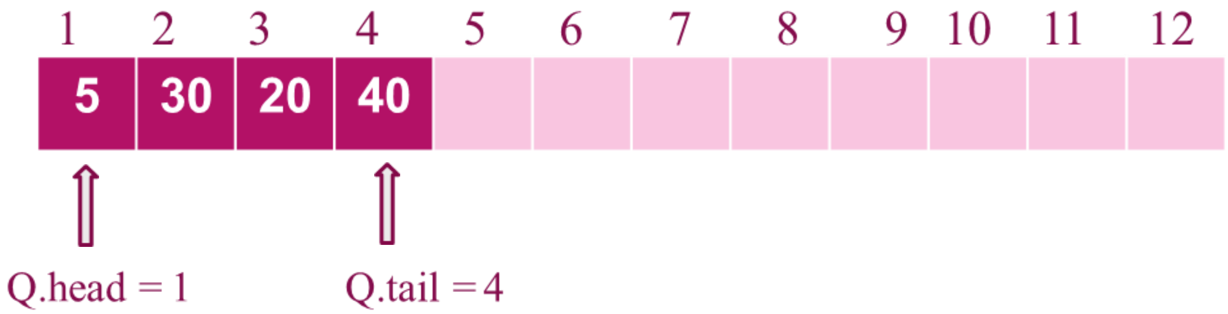
1	2	3	4	5	6	7	8	9	10	11	12
5	30	20	40	10	15						

Q.head = 3

Q.tail = 6

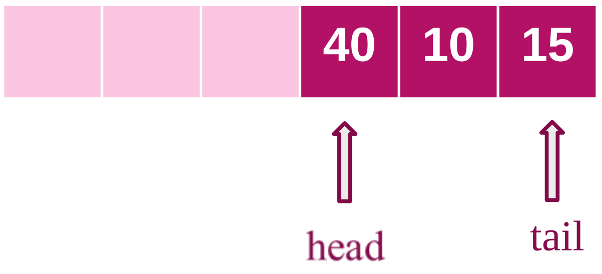
Queue -Operations

```
ENQUEUE (Q, x)  // check the correctness
    if (QUEUE-FULL(Q))
        error "overflow"
    else
        Q. tail= Q. tail + 1
        Q [Q. tail] = x
```



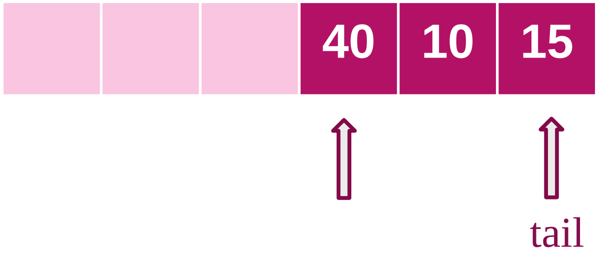
Queue -Operations

QUEUE-FULL(Q) ?



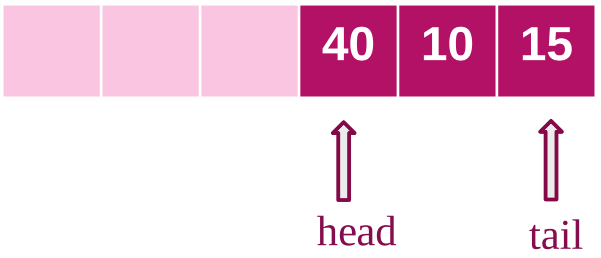
- **No further EnQueue possible even though the queue is not full**

Queue -Operations

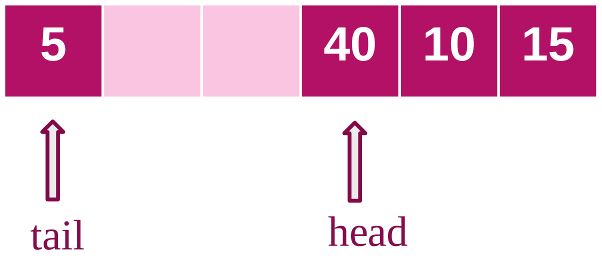


- No further EnQueue possible even though the queue is not full
  - Left Shift the element
    - Takes linear time

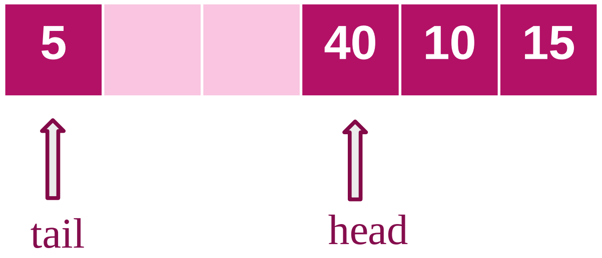
Queue -Operations



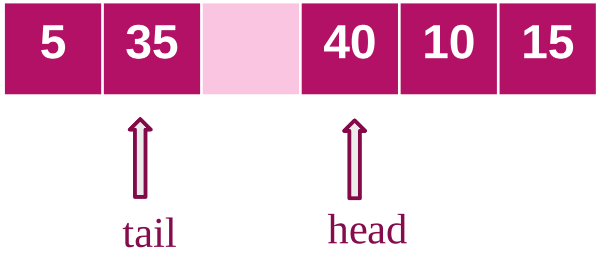
Start adding from left



Queue -Operations

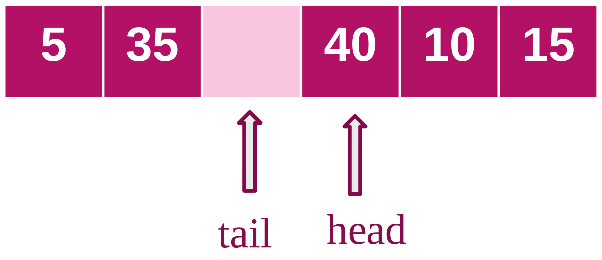


EnQueue(Q, 35)



One more EnQueue ?

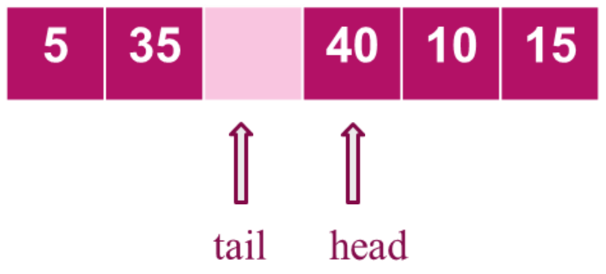
Queue -Operations



- No more EnQueue()
- One slot left vacant (maximum *n-1* elements can be queued in an *n* element array)
- Q.tail points to the next location where a new element can be added
- Makes checking QueueFull() / QueueEmpty() easier

Queue - Array Based Implementation

- Array Q[1..n]
- Attributes - Q. head, Qtail
- Q.head points to actual head
- Q.tail points to the next location for insertion
- Initially Q.head =Q.tail =1
- Elements from Q.head, Q.head+1,...Q.tail -1



Reference

T H Cormen, C E Leiserson, R L Rivest, C Stein *Introduction to Algorithms,* 3rd ed., PHI, 2010