CPSC 259: Data Structures and Algorithms for Electrical Engineers

Stack and Queue

(a) Thareja (first edition):9.1-9.6, 9.11, and 9.12
(b) Thareja (second edition): 7.1 – 7.7.2 and 8.1-8.3

Hassan Khosravi

Learning goals

- Differentiate an abstraction from an implementation.
- Determine the time complexities of operations on stacks and queues.
- Manipulate data in stacks and queues (using array and linked list implementation).
- Use stacks and queues to solve real world problems

What is an Abstract Data Type?

- Abstract Data Type (ADT) – a mathematical description of an object and the set of operations on the object.
  - A description of how a data structure works (could be implemented by different actual data structures).
- Example: Dictionary ADT
  - Stores pairs of strings: (word, definition)
  - Operations:
    - Insert(word, definition)
    - Delete(word)
    - Find(word)

Implemented by a data structure

Why so many data structures?

Ideal data structure:

fast, elegant, memory efficient

Trade-offs

- time vs. space
- performance vs. elegance
- generality vs. simplicity
- one operation's performance vs. another's
- serial performance vs. parallel performance

"Dictionary" ADT

- list
- Binary Search Tree
- AVL tree
- Splay tree
- B+ tree
- Red-Black tree
- hash table
- concurrent hash table
- …

Code Implementation

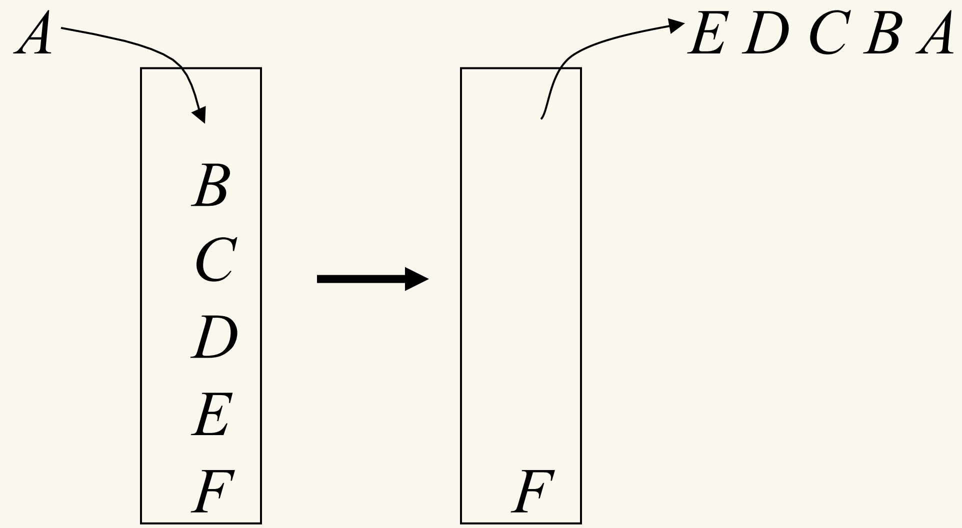
- Theoretically (in programming languages that support OOP)
  - abstract base class describes ADT
  - inherited implementations implement data structures
  - can change data structures transparently (to client code)
- Practice
  - different implementations sometimes suggest different interfaces (generality *vs*. simplicity)
  - performance of a data structure may influence form of client code (time *vs*. space, one operation *vs*. another)

ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - develop a data structure for the ADT
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

Stack ADT

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top/peek
  - is_empty



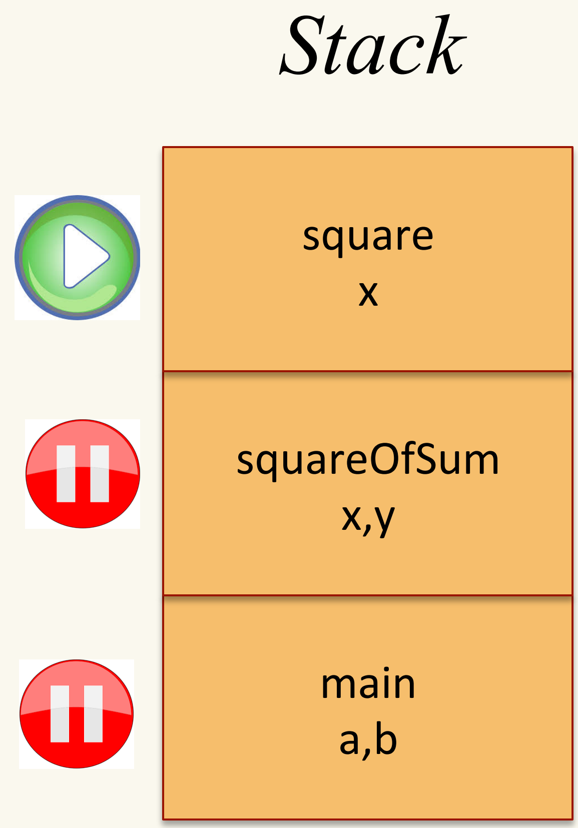
- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

  LIFO: Last In First Out

Demo: http://visualgo.net/list.html

Stacks in Practice (Call Stack)

```
int square (int x){
    return x*x;
}

int squareOfSum(int x, int y){
    return square(x+y);

}

int main() {
    int a = 4;
    int b = 8;
    int total = squareOfSum(a, b);
    cout << total<< endl;
}
```

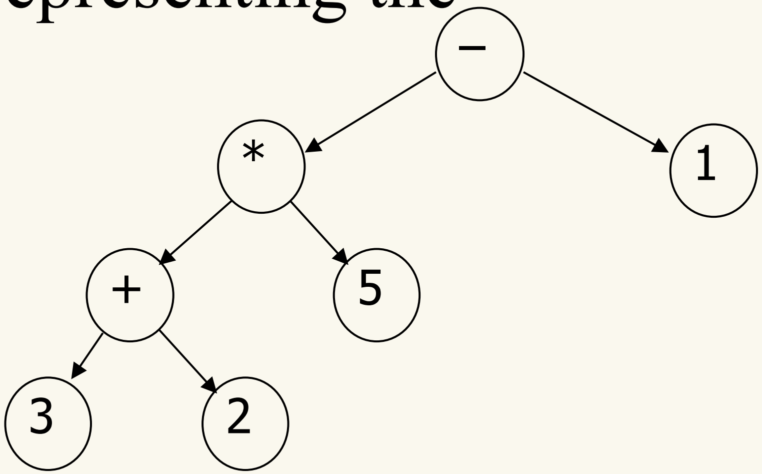


Stacks in Practice (Arithmetic expressions)

- **Application: Binary Expression Trees**

Arithmetic expressions can be represented using binary trees. We will build a binary tree representing the expression:

(3 + 2) * 5 – 1



Now let's print this expression tree using postorder traversal:

3 2 + 5 * 1 -

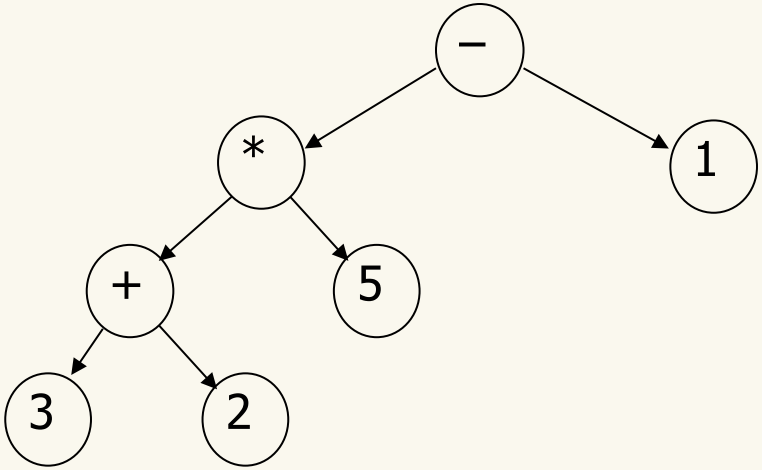
We'll cover this topic later in the course

Stacks in Practice (Arithmetic expressions)

Now let's compute this expression using a Stack

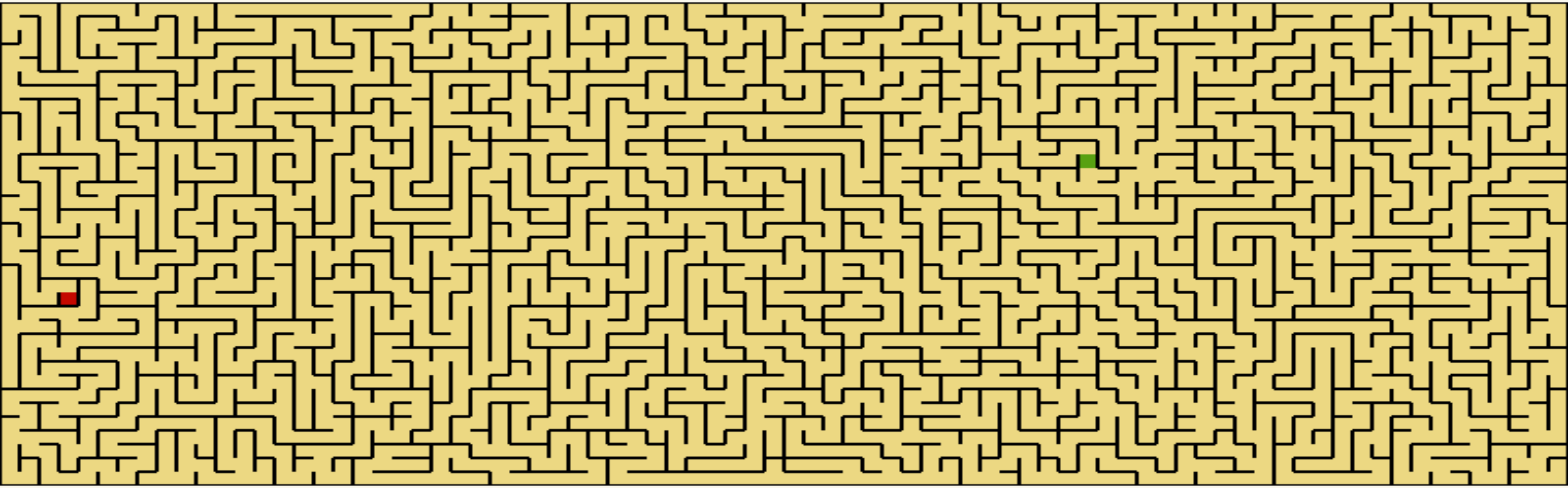
3 2 + 5 * 1 -

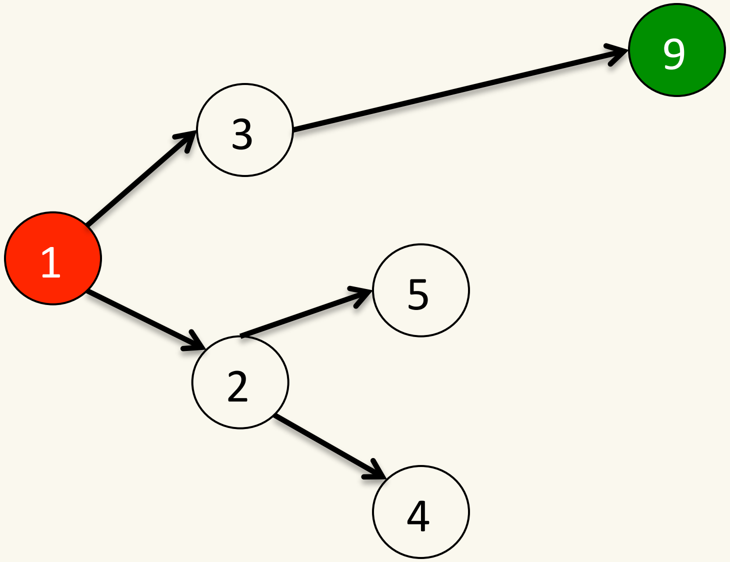
Character scanned	Stack
3	3
2	3, 2
+	5
5	5, 5
*	25
1	25,1
-	24



We'll cover this topic later in the course

Stacks in Practice (Backtracking)





Stack
1
3, 2
3,5,4
3,5
3
9

We'll cover this topic later in the course

Array Representation of Stacks

- In computer's memory stacks can be represented as a linear array.
  - Every stack has a variable TOP associated with it.
    - TOP is used to store the index of the topmost element of the stack. It is this position from where the element will be added or deleted.
  - There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.

Array Representation of Stacks

```
typedef struct
{
    int top;
    int* list;
} Stack;
```

```
#define TRUE 1
#define FALSE 0
```

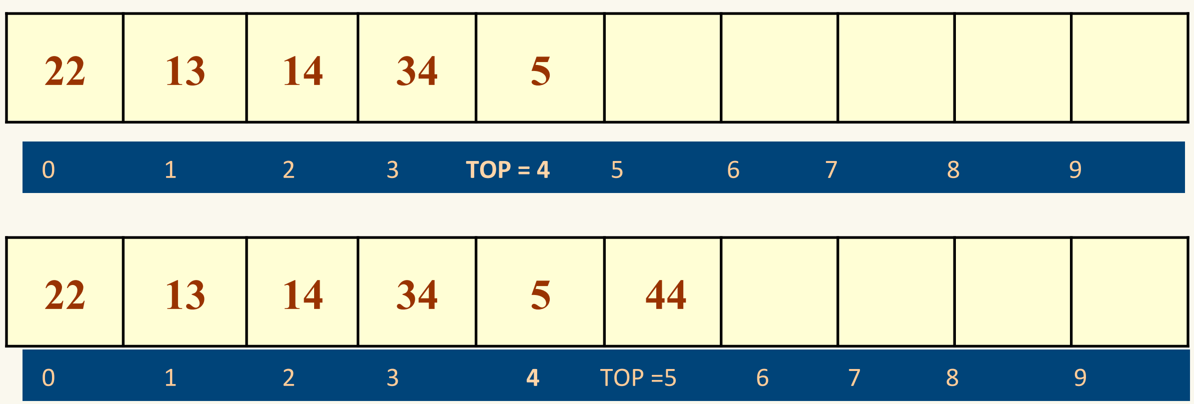
```
void initialize(Stack* stack)
{
   stack->top=-1;
   stack->list = (int*)malloc(sizeof(int)*CAPACITY);
}
```

```
int isEmpty(Stack* stack)
{
    if (stack->top == -1)
        return TRUE;
    else
        return FALSE;
}
```

```
int isFull(Stack* stack)
{
    if (stack->top == MAX-1)
        return TRUE;
    else
        return FALSE;
}
```

Push Operation

- The push operation is used to insert an element into the stack.
  - The new element is added at the topmost position of the stack.
  - However, before inserting the value, we must first check if TOP=MAX-1, because if this is the case then it means the stack is full and no more insertions can further be done.
  - An attempt to insert a value in a stack that is already full causes an overflow error



Push Operation

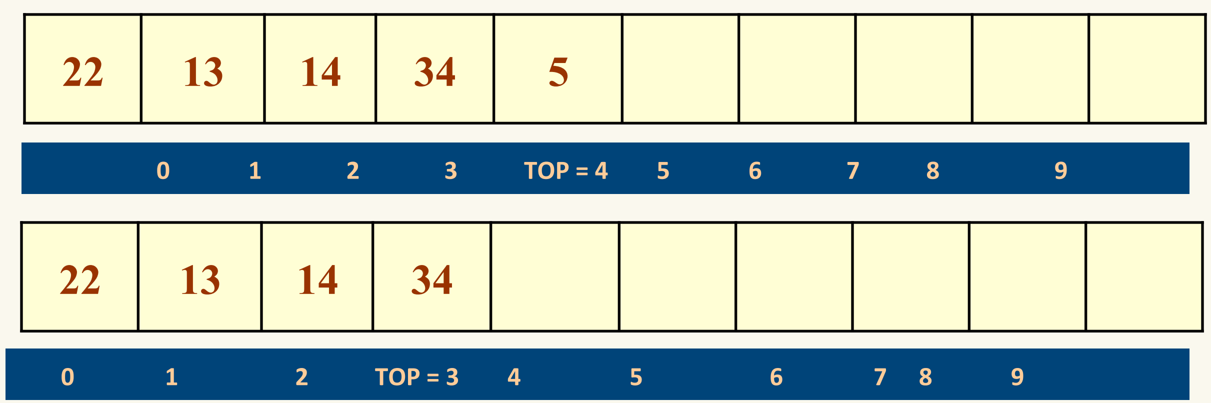
```
int push(Stack* stack, int value)
{
    if (!isFull(stack))
    {
        stack->top++;
        stack->list[stack->top]=value;

        return TRUE;
    }
    else
        return FALSE;
}
```

22	13	14	34	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

22	13	14	34	5	44				
0	1	2	3	**4**	TOP =5	6	7	8	9

Pop Operation

- The pop operation is used to delete the topmost element from the stack.
  - However, before deleting the value, we must first check if TOP=-1, because if this is the case then it means the stack is empty so no more deletions can further be done.
  - An attempt to delete a value from a stack that is already empty causes an underflow error.



Pop Operation

```
int pop(Stack* stack)
{
    if (!isEmpty(stack))
    {
        stack->list[stack->top]=-1;
        stack->top--;
        return TRUE;
    }
    else
        return FALSE;
}
```

22	13	14	34	5					
0	1	2	3	TOP = 4	5	6	7	8	9

22	13	14	34						
0	1	2	TOP = 3	4	5	6	7	8	9

Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
    - However, the peek operation first checks if the stack is empty or contains some elements. If TOP = -1, then an appropriate message is printed else the value is returned

22	13	14	34	5					
0	1	2	3	TOP= 4	5	6	7	8	9

- Here the Peek operation will return 5, as it is the value of the topmost element of the stack.

Peek Operation

```
int peek(Stack* stack)
{
    if (!isEmpty(stack))
        return stack->list[stack->top];
    else
        return FALSE;
}
```

22	13	14	34	5					
0	1	2	3	TOP= 4	5	6	7	8	9

- Here Peek operation will return 5, as it is the value of the topmost element of the stack.

Example Stack with Arrays

push B

pop

push K

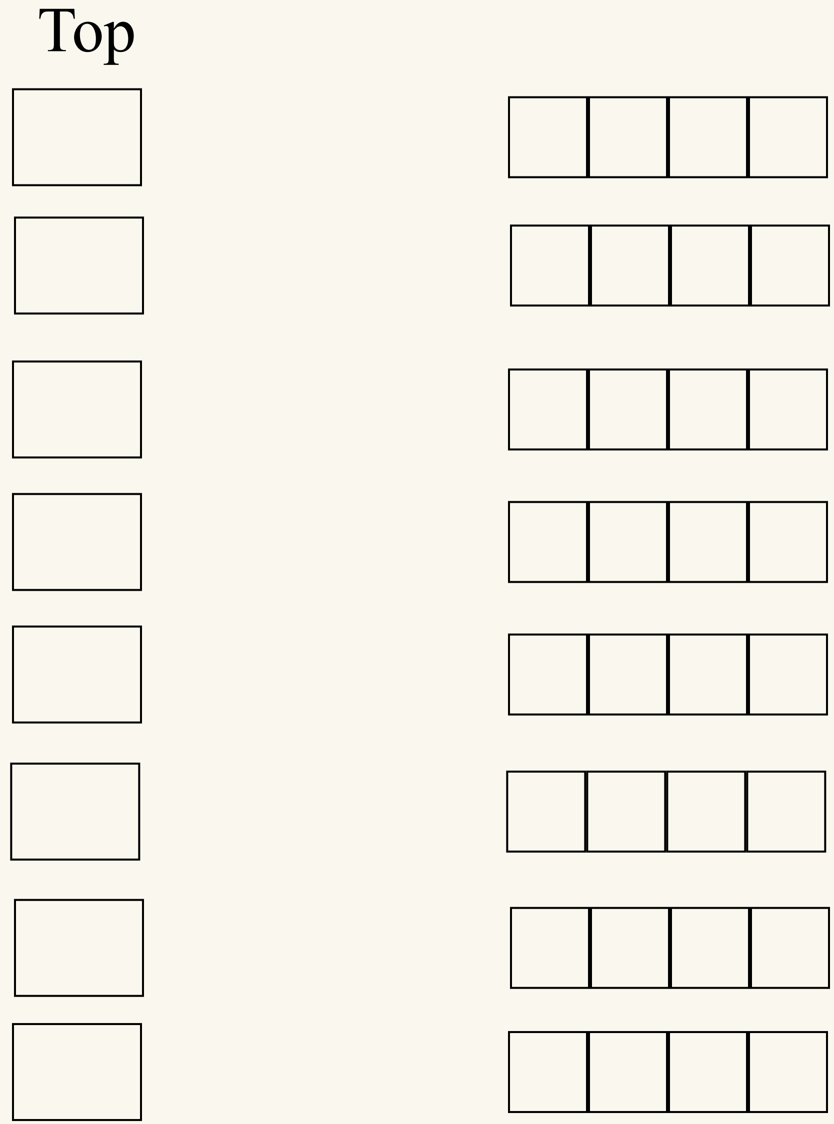
push C

push A

pop

pop

pop



Example Stack with Arrays

	Top				
push B	*0*	*B*			
pop	*-1*				
push K	*0*	*K*			
push C	*1*	*K*	*C*		
push A	*2*	*K*	*C*	*A*	
pop	*1*	*K*	*C*		
pop	*0*	*K*			
pop	*-1*				

CPSC 259 Administrative Notes

- Labs
  - Lab3 – week1 in progress (Oct 13 – Oct 19)
  - Lab3 – Week2 (Oct 26 – Oct 30)
  - No labs (Oct 20 – Oct 23)

- Midterm: On Wednesday (details on course website)
  - Up to and including the Stack and Queue module

- Extra office hour
  - Sean: Tuesday October 20th, 2-4pm, in ICCSX239.

- Exercises/questions on Stack and Queue added to the course website

Stacks

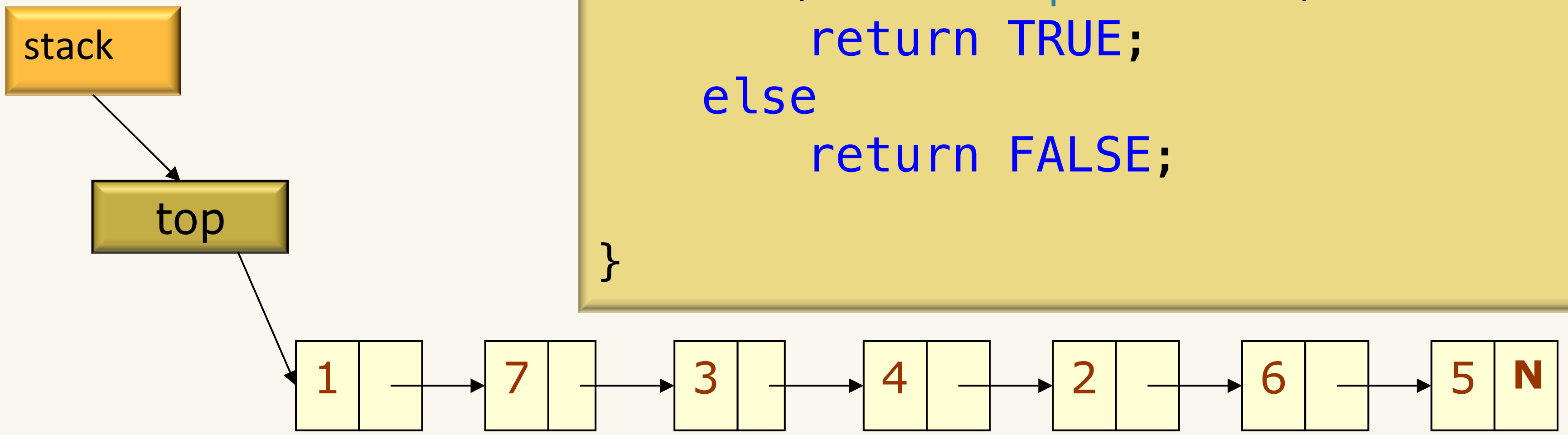
- A stack operates on the LIFO principle: Last In, First Out.
- Some stack operations and their complexities:
  - push(item) ____O(1)____ (add to top)
  - pop() ____O(1)____ (take off top)
  - peek() ____O(1)____ (without removing)
  - isempty() ____O(1)____ (is stack empty?)
  - isfull() ____O(1)____ (is stack full?)

Linked list representation of Stacks

```
typedef struct
{
    struct Node* top;
} Stack_list;
```

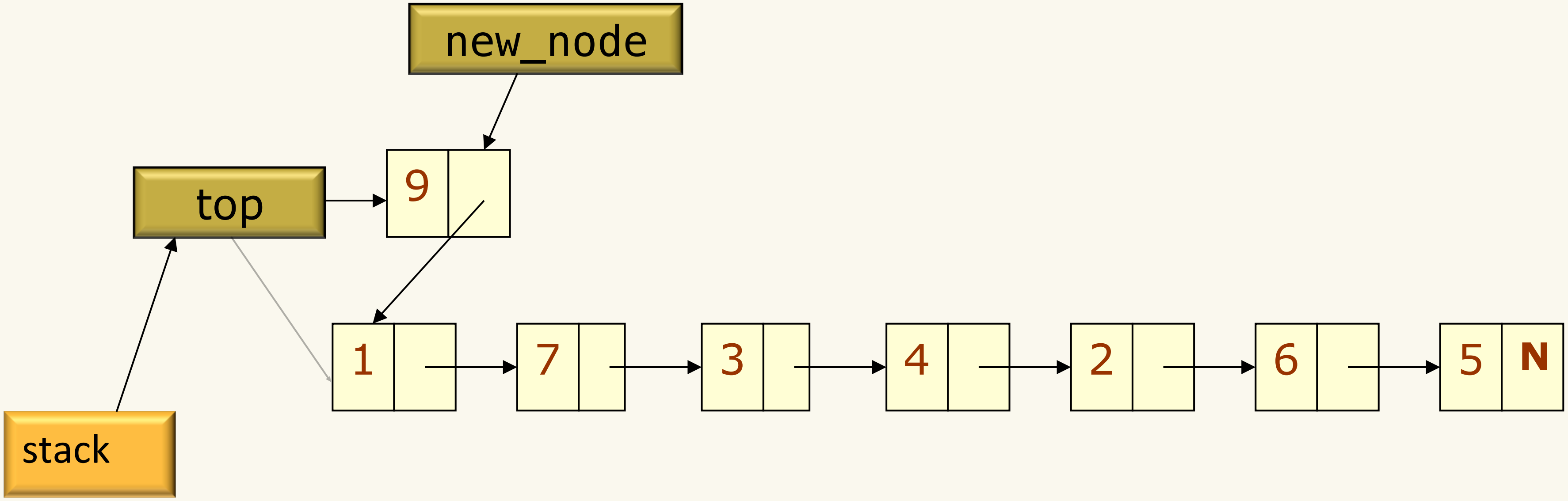
```
struct Node
{
    int data;
    struct Node* next;
};
```

```
int isEmpty_list( Stack_list* stack)
{
    if (stack->top == NULL)
        return TRUE;
    else
        return FALSE;

}
```



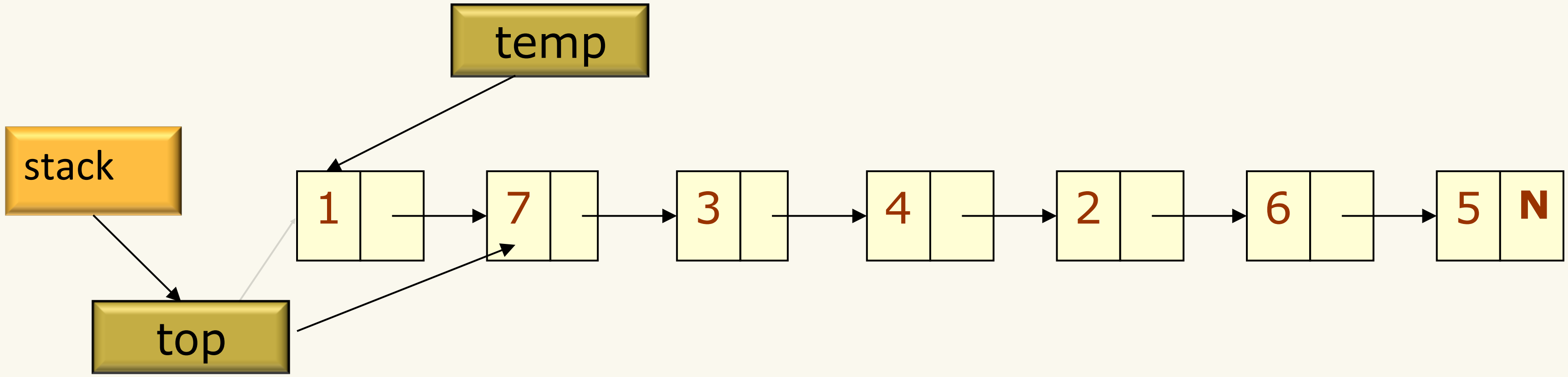
Push Operation on a Linked Stack

```
int push_list( Stack_list* stack, char value)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    if (new_node==NULL)
       return FALSE;

    new_node->data = value;
    new_node->next = stack->top;
    stack->top = new_node;
    return TRUE;
}
```



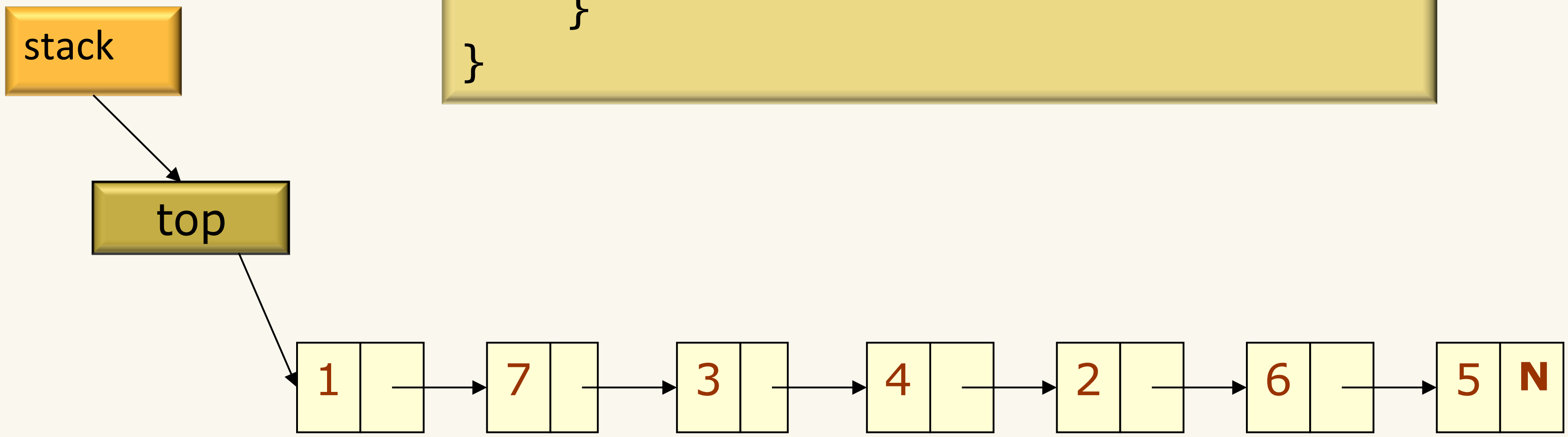
Pop Operation on a Linked Stack

```
int pop_list(Stack_list* stack)
{
    if (!isEmpty_list(stack))
    {
        struct Node* temp = stack->top;
        stack->top = stack->top->next;
        free(temp);
        temp = NULL;
        return TRUE;
    }
    return FALSE;
}
```



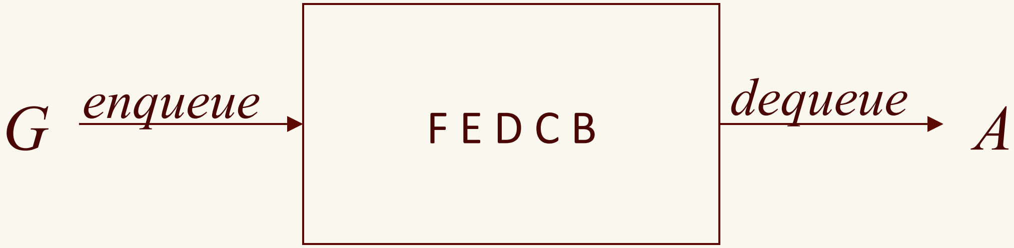
Peek Operation on a Linked Stack

```
int peek_list(Stack_list* stack)
{
    if (!isEmpty_list(stack)) {
        return stack->top->data;
}
    else
    {
        return FALSE;
    }
}
```



Queue ADT

- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty



- Queue property:

  if x is enqueued before y is enqueued,
  then x will be dequeued before y is dequeued.

  FIFO: First In First Out

Applications of the Q

- Hold jobs for a printer
- Store packets on network routers
- Hold memory “freelists”
- Make waitlists fair
- Breadth first search

Abstract Q Example

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

In order, what letters are dequeued?

a. OATE
b. ROTA
c. OTAE
d. None of these, but it **can** be determined from just the ADT.
e. None of these, and it **cannot** be determined from just the ADT.

Abstract Q Example

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

In order, what letters are dequeued?

a. OATE
b. ROTA
c. OTAE
d. None of these, but it **can** be determined from just the ADT.
e. None of these, and it **cannot** be determined from just the ADT.

Array Representation of Queues

- Queues can be easily represented using linear arrays.
- Every queue has front and back variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in figure

12	9	7	18	14	36				

0 1 2 3 4 5 6 7 8 9

front = 0
back = 6

- If we want to add one more value in the list say with value 45, then back would be incremented by 1 and the value would be stored at the position pointed by back.

12	9	7	18	14	36	**45**			

0 1 2 3 4 5 6 7 8 9

front = 0
back= 7

Array Representation of Queues

- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue

	9	7	18	14	36	45			

0 1 2 3 4 5 6 7 8 9

front = 1
back = 7

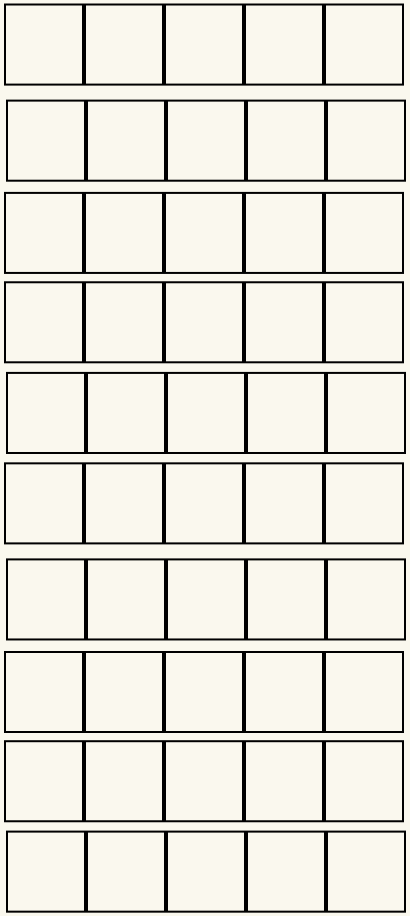
- What is a problem with this implementation?

		7	18	14	36	45	21	99	72

0 1 2 3 4 5 6 7 8 9

Circular Array Q Example 1

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue



Circular Array Q Example 1

enqueue R	R				
enqueue O	R	O			
dequeue	~~R~~	O			
enqueue T	~~R~~	O	T		
enqueue A	~~R~~	O	T	A	
enqueue T	~~R~~	O	T	A	T
dequeue	~~R~~	~~O~~	T	A	T
dequeue	~~R~~	~~O~~	~~T~~	A	T
enqueue E	E	~~O~~	~~T~~	A	T
dequeue	E	~~O~~	~~T~~	~~A~~	T

Circular Array Q Example 2

enqueue R	R				
enqueue O	R	O			
enqueue T	R	O	T		
enqueue A	R	O	T	A	
enqueue T	R	O	T	A	T
enqueue E	E	O	T	A	T

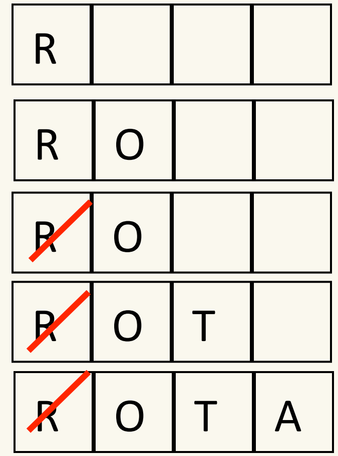
- Before inserting
  - Check is_full()
- Before removing
  - Check is_empty()

Circular Array Q Example 3

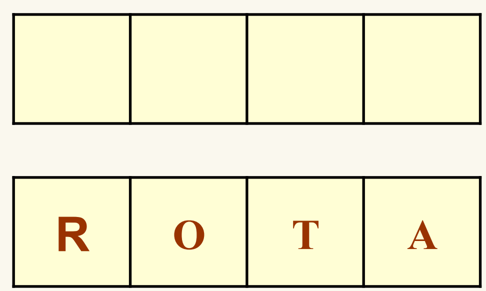
enqueue R				
enqueue O				
dequeue				
enqueue T				
enqueue A				
enqueue T				
dequeue				
dequeue				
enqueue E				
dequeue				

Circular Array Q Example 3

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue



Cannot add the second T
Why?



front = 1
back = 1

front = 1
back = 1

Array Representation of Stacks

```
typedef struct{
  int front;
  int back;
  int* list;
} Queue;
```

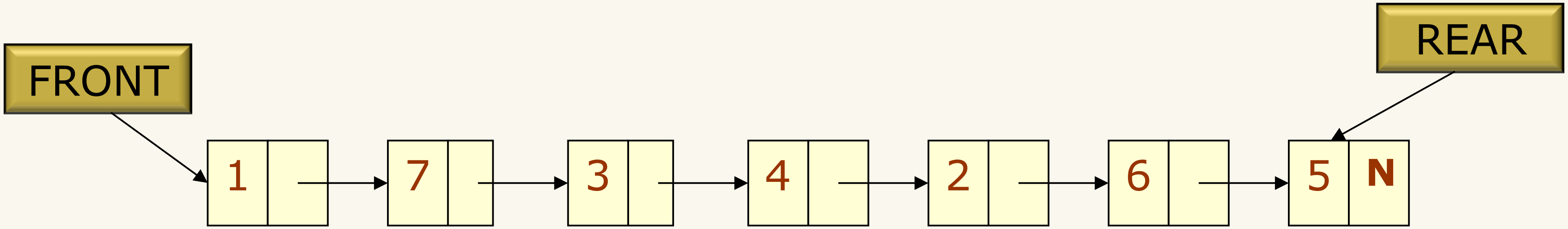
```
void initialize(Queue* queue){
    queue->front=0;
  queue->back=0;
    queue->list = (int*)malloc(sizeof(int)*CAPACITY);
}
```

```
int isEmpty(Queue* queue){
  return(queue->front ==queue->back);
}
```

```
int isFull(Queue* queue){
  return (queue->front == (queue->back + 1) % CAPACITY);
}
```

Linked Representation of Queues

- The START pointer of the linked list is used as FRONT.
- We will also use another pointer called REAR which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If FRONT = REAR = NULL, then it indicates that the queue is empty.



Exercise

- Implement the queue data structure using arrays and linked lists (very similar to the implementation of Stack)

Queues

- Some queues operations and their complexities:
  - push(item) ______O(1)______ (add to top)
  - pop() ______O(1)______ (take off top)
  - peek() ______O(1)______ (without removing)
  - isempty() ______O(1)______ (is queue empty?)
  - isfull() ______O(1)______ (is queue full?)

Popular Interview Question

- Given an expression as a string comprising of opening and closing characters of parentheses - (), curly braces - {} and square brackets - [], check whether symbols are balanced or not.
- You may make use of the following function and a Stack implementation in your code

```
// Function to check whether two characters are opening
// and closing of same type.
int ArePair(char opening,char closing)
{
  if(opening == '(' && closing == ')') return TRUE;
  else if(opening == '{' && closing == '}') return TRUE;
  else if(opening == '[' && closing == ']') return TRUE;
  return FALSE;
}
```

is_balanced

```
int is_balanced(char* exp){
  Stack_list S;
  for(int i =0;i< strlen(exp); i++){
    if(exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
      push_list(&S, exp[i]);
    else if(exp[i] == ')' || exp[i] == '}' || exp[i] == ']'){
      if(isEmpty_list(&S) ||!ArePair(peek_list(&S),exp[i]))
        return FALSE;
      else
        pop_list(&S);
    }
  }
  return isEmpty_list(&S);
}
```

Learning goals revisited

- Differentiate an abstraction from an implementation.
- Determine the time complexities of operations on stacks and queues.
- Manipulate data in stacks and queues (using array and linked list implementation).
- Use stacks and queues to solve real world problems