Data Structure & Algorithms

Lec 03: Stacks

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. Motivation & Background
2. Stacks Definition, Properties and Applications
3. Stack as an ADT
4. Stack Array based Implementation
5. Self Assessment

Learning Outcome

By the end of this class, you should be able to

- Explain the stack operations and its applications
- Implement the stack as ADT
- Use stack to solve problems that require stack insertion and deletion behaviors

Motivation and Background

A stack is a linear data structure similar to arrays and linked list. The stack defines one restriction over elements insertion and deletion. The elements are added and removed from the stack at one end.

The last element inserted is the first element to be removed from the stack.

The way the stack handle elements insertion and deletion is called Last In First Out (LIFO) or First In Last Out (FILO).

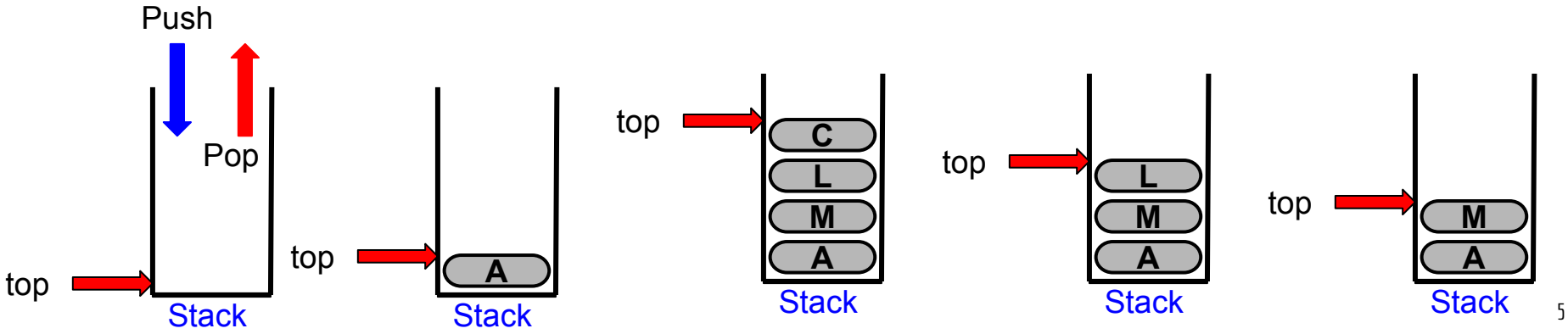
The insertion of an element into the stack called a push operation and the deletion of an element from the stack called a pop operation.

How does the Stack work?

We need a special pointer | index called top. We add and remove elements to/from the top of the stack.

Trying to add an element to a full stack result in overflow condition

Trying to remove an element from an empty stack result in underflow condition



Stack Applications

Many real-life applications require stack behaviors (LIFO) structures.

1. Execution or call Stack.
2. Balancing of symbols.
3. Undo operations in IDEs, Text editors, etcs
4. Retrieving social network history (e.g facebook post, tweets, etc)
5. Evaluation of Polish Notation (e.g. + 3 x 5 2)
6. Matching Tags markup languages (e.g HTML, XML)
7. Graph traversing algorithms (e.g. Depth First Search)

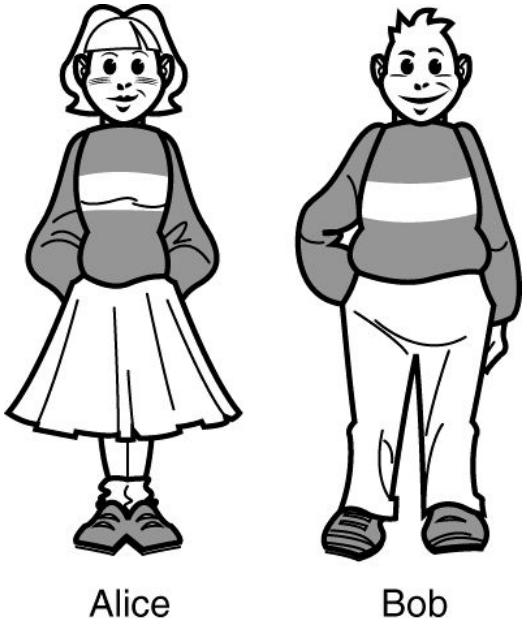
Stack Applications: Guess What Happened?

```
Exception in thread "main" java.lang.StackOverflowError
    at java.lang.String.length(Unknown Source)
    at java.lang.AbstractStringBuilder.append(Unknown Source)
    at java.lang.StringBuilder.append(Unknown Source)
    at java.lang.StringBuilder.<init>(Unknown Source)
    at com.examscam.model.Tester.toString(Tester.java:13)
    at java.lang.String.valueOf(Unknown Source)
    at java.lang.StringBuilder.append(Unknown Source)
```

Execution or call Stack Exception.

Stack as ADT

Alice and **Bob** are software developers working in data science startup. They come up with an algorithm to collect and analyze events (e.g., facebook posts, tweets, stock price, news feed, etc.) to extract insights. They called it Algorithm X. They figure out that they need a linear data structure that provides Last In First Out insertion and deletion behaviors. For efficiency, they decided to use C/C++ to implement their algorithm. For productivity, Bob will implement a stack data structure as an ADT and Alice will focus on implementing Algorithm X that will use Bob's Stack ADT.



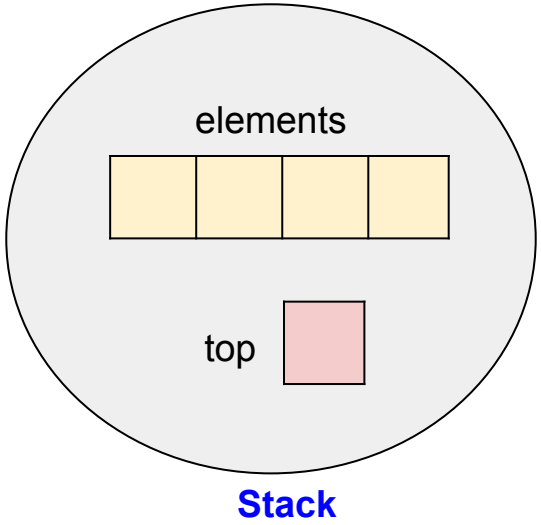
Stack as ADT

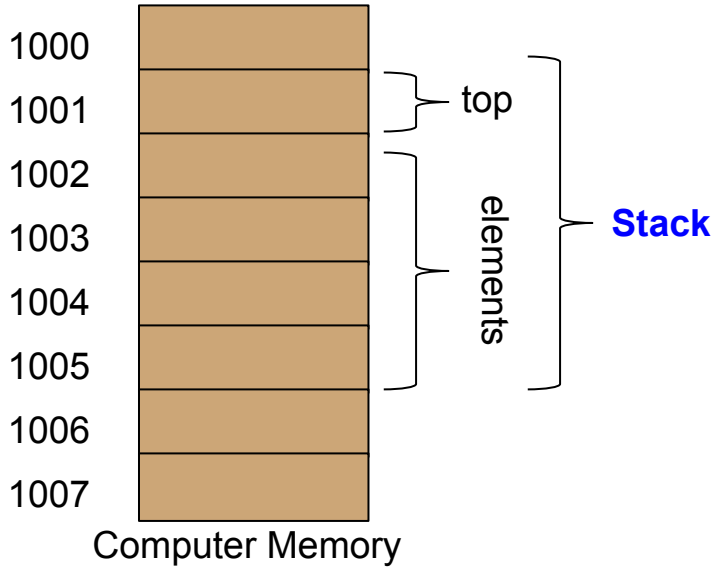
Bob and Alice decided that they need the following operations in their Stack ADT:

1. Push Element (Event) into the stack
2. Pop Element (Event) from the stack
3. Initialize the stack
4. Check if the stack is full
5. Check if the stack is empty
6. Measure the size of the stack
7. Processing stack elements
8. Delete all elements in the stack
9. Check the top of the stack

Stack ADT: Array based Implementation

Bob decided to implement the stack ADT using an array. In this case elements will be added from left to right and a variable "top" is used to keep track of the stack top.

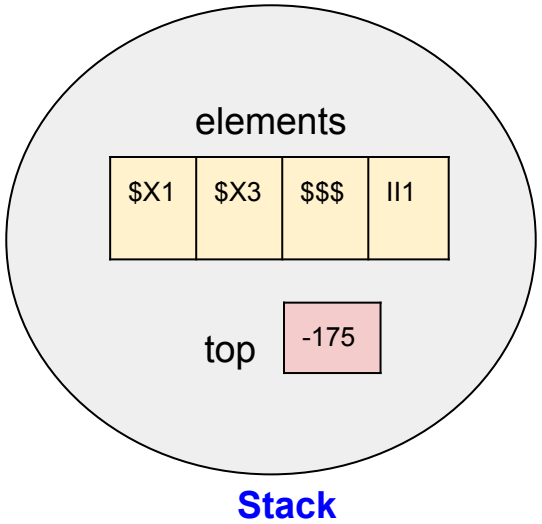




Stack ADT: Create Stack

```
 7 int main() {
 8
 9     Stack mystack;
10
11     return 0;
12 }
```

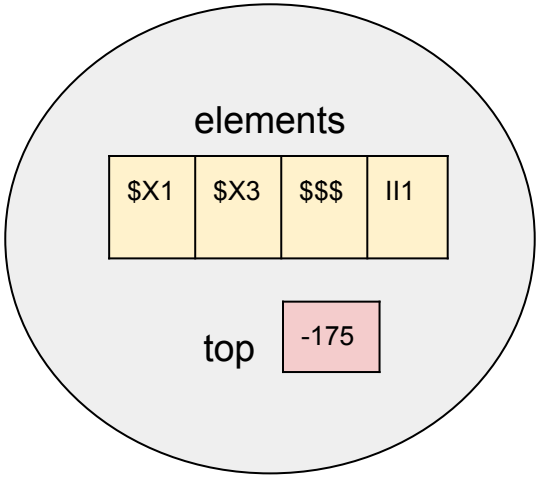
Alice Code "Algorithm X"



Stack ADT: Create Stack

```
3 typedef struct stack{
4
5     int top;
6
7     StackElement elements[MaxStack];
8
9 }Stack;
```

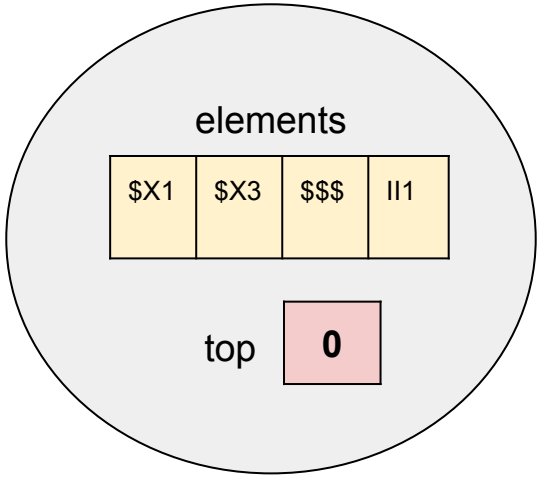
Bob's Code "Stack ADT"



Stack ADT: Stack Initialization

```
19 int main() {
20
21     Stack mystack;
22
23     InitiateStack(myStack);
24
25
26     return 0;
27 }
```

Alice Code "Algorithm X"



Stack

What is wrong??

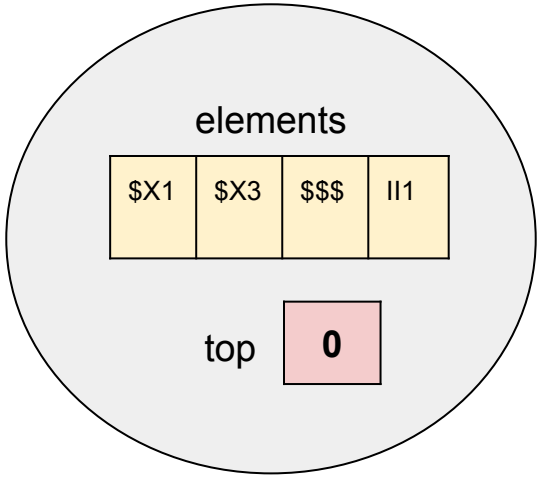
```
13 void InitiateStack(Stack stack){
14
15     stack.top = 0;
16
17 }
```

Bob's Code "Stack ADT"

Stack ADT: Stack Initialization

```
20 int main() {
21
22     Stack mystack;
23
24     InitiateStack(&myStack);
25
26
27     return 0;
28 }
```

Alice Code "Algorithm X"



Stack

```
13 void InitiateStack(Stack *ptrStack){
14
15     ptrStack->top = 0;
16
17 }
```

Bob's Code "Stack ADT"

Stack ADT: Push Element

```
34      StackElement e;
35
36      /*Alice set data for StackElment e */
37
38      Push(e, &myStack);
39
```

Alice Code "Algorithm X"

What is wrong??

```
20 void Push(StackElement e, Stack *ptrStack){
21
22      ptrStack->elements[ptrStack->top] = e;
23
24      ptrStack->top++;
25 }
```

Bob's Code "Stack ADT"

Stack ADT: Is Stack Full

```
48      if(!IsFullStack(&myStack)){
49
50          Push(e, &myStack);
51
52      }
```

Alice Code "Algorithm X"

What else could we do to improve the code?

```
20 int IsFullStack(Stack *ptrStack){
21
22      if(ptrStack->top >= MAXSIZE){
23          return 1;
24      }
25      else{
26          return 0;
27      }
28 }
```

Bob's Code "Stack ADT"

Stack ADT: Pop an Element

```
61      StackElement e;
62
63      Pop(&e, &myStack);
```

Alice Code "Algorithm X"

What is wrong??

```
37 void Pop(StackElement *e, Stack *ptrStack){
38
39      *e = ptrStack->elements[ptrStack->top-1];
40
41      ptrStack->top--;
42 }
```

Bob's Code "Stack ADT"

Stack ADT: Is Stack Empty

```
73      StackElement e;
74
75      if(!IsEmptyStack(&myStack)){
76
77          Pop(&e, &myStack);
78      }
79
80      return 0;
81  }
```

Alice Code "Algorithm X"

What else could we do to improve the code?

```
45  int IsEmptyStack(Stack *myStack){
46
47      if(myStack->top>0){
48          return 0;
49      }
50      else{
51          return 1;
52      }
53  }
```

Bob's Code "Stack ADT"

Stack ADT: Delete | Empty Stack

```
88      DeleteStack(&myStack);
89
90      return 0;
```

Alice Code "Algorithm X"

Why not calling initiate stack?

```
21 void DeleteStack(Stack *ptrStack){
22
23     ptrStack->top = 0;
24
25 }
```

Bob's Code "Stack ADT"

Stack ADT: Stack Size

```
95      int size;
96
97      size = StackSize(&myStack);
```

Alice Code "Algorithm X"

```
63 int StackSize(Stack *myStack){
64
65      return mystack->top;
66 }
```

Bob's Code "Stack ADT"

Stack ADT: Stack Top

```
107     StackElement e;
108
109     StackTop(&e, &myStack);
110
111     /*
112      Alice process e in her code without
113      removing it from the stack
114     */
```

Alice Code "Algorithm X"

```
52 void StackTop(StackElement *e, Stack *ptrStack){
53
54     *e = ptrStack->elements[ptrStack->top-1];
55
56 }
```

Bob's Code "Stack ADT"

Stack ADT: Traverse Elements

Bob wants to provide Alice with a feature that allows her to traverse the elements in the stack without removing them. This will allow Alice to process the elements when needed, for example, to display the elements, store them into a file or database, etc.

Bob does not know what Alice exactly going to do and wants to provide access flexibility to Alice.

What should Bob do?

Stack ADT: Traverse Elements

What should Bob do?

He should implement the traverse function in a way that enable processing each element in the stack with a user-defined function (e.g. with a function defined by Alice)

```
13 void Foo1(int x) {
14
15     cout<<x<<endl;
16 }
17
18 void Foo2(void (*pf) (int), int arr[], int size){
19
20     for (int i=0; i<size; i++){
21         (*pf)(arr[i]);
22     }
23 }
24
25 int main() {
26
27     int arr[5]= {4,2,8,9,1};
28
29     Foo2(&Foo1,arr,5);
30
31     return 0;
32 }
```

Stack ADT: Traverse Elements

```
70 void PrintStack(StackElement e){
71
72     /*
73         print the stack element data
74         on the screen
75     */
76 }
```

Alice Code "Algorithm X"

```
79 void TraverseStack(Stack *ptrStack, void (*ptrFun) (StackElement)){
80
81     for(int i=ptrStack->top; i>0; i--){
82
83         (*ptrFun)(ptrStack->elements[i-1]);
84     }
85 }
```

Bob's Code "Stack ADT"

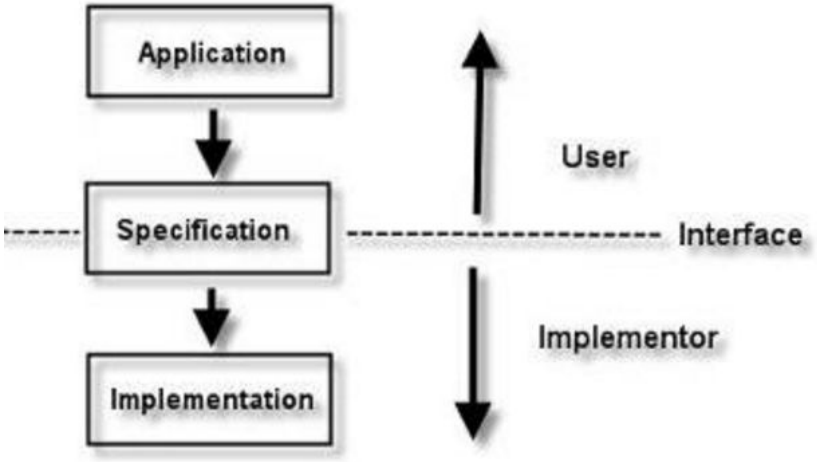
Run-Time Complexity

The run time complexity of the stack operations with array-based implementation are:

Operation	Run-Time Complexity
Push an Element	O(1)
Pop an Element	O(1)
Stack Size	O(1)
Is Empty Stack	O(1)
Is Full StacK	O(1)
Delete | Empty Stack	O(1)

Abstract Data Types

- Abstract data Type is a mathematical and logical model for data types. The data type is defined by its behaviors (operations) from the user point of view.
- The implementation details are hidden from the users.
- Bob implements a Stack ADT for Alice. Alice is the user of Bob's Stack and Bob is the Stack ADT implementor. Bob is responsible for providing the user of



Abstract Data Type

Abstract Data Types

- Bob provides Alice with his Stack ADT specifications that inform Alice about the available interfaces and how she can use them.
- Alice does not care how Bob implements every method or operation.
- Bob can easily change the implementation of his Stack without requiring Alice to change her implementation.
- ADT emphasize on applying encapsulation and information hiding.

```c
#define MAXSTACK 100

typedef  struct StackElement{

    int data;

}StackElement;


typedef struct stack{
    int top;

    StackElement elements[MAXSTACK];

}Stack;

void InitiateStack(Stack *);

void Push(StackElement, Stack *);

int  IsFullStack(Stack *);

void Pop(StackElement *, Stack *);

int  IsEmptyStack(Stack *);

int  StackSize(Stack *);

void DeleteStack(Stack *);

void StackTop(StackElement *, Stack*);

void TraverseStack(Stack *, void (*)(StackElement));
```

stack.h

Pre and Post Conditions

While Alice does not need to know the implementation details of the Stack operations, she needs to know the pre-conditions and post-conditions of the Stack operations. Bob must provide the pre and post conditions of the stack operations as part of the specification.

For example, the pre conditions of the Push operation are stack must be initialized and not full. The post conditions are the element added to the top of the stack and the stack top increased by one.

Self-Assessment

- In your opinion what are the limitations and advantages of using an array-based approach to implement a Stack ADT?
- Using a stack data structure design an algorithm to check if a given equation or expression has balanced symbols or not.

Expression	Is Valid
(A+B) * C-1)^2+4	No
Hello World!	Yes
(A^B) * ((C+D)*(E-A)	No