Data Structure & Algorithms

Lec 04: Stacks Linked List-based implementation

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. Stack Linked-Lists based Implementation
2. Stack Application:Parsing Arithmetic Expressions
3. 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

Stacks Array-Based Implementation

The array based implementation of a stack has the following limitations:

1. Preallocation of memory
2. Fixed capacity during runtime

The array based implementation has the following advantages:

1. All access mechanism (operations) has O(1) except traverse
2. Implementation is straightforward
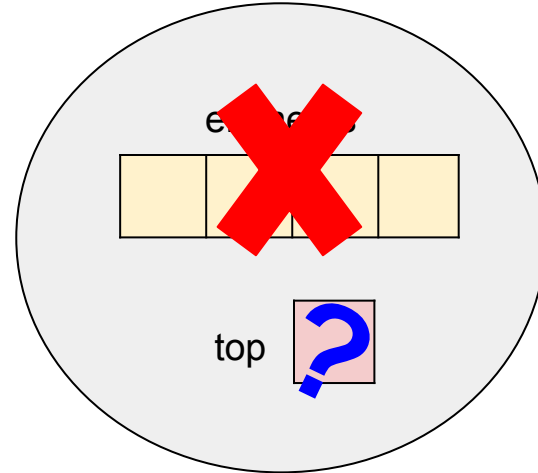
Stack Linked List-Based Implementation

Using a linked-list structure to implement a stack will avoid the need to preallocate memory in advance and provide a dynamic size to the stack where the stack could grow or shrink during runtime based on the application requirements.

What could be the major limitation of using linked lists to implement the stack?
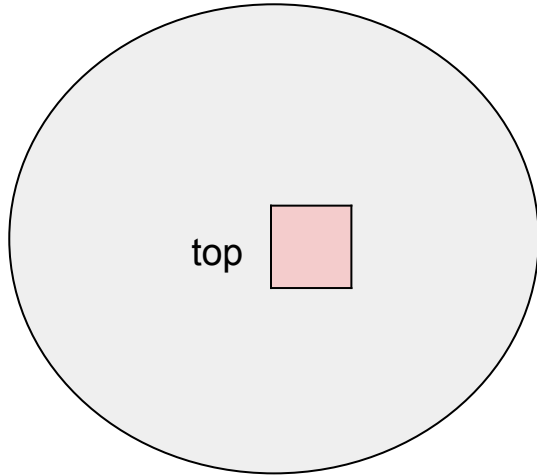
Stack Implementation

elements
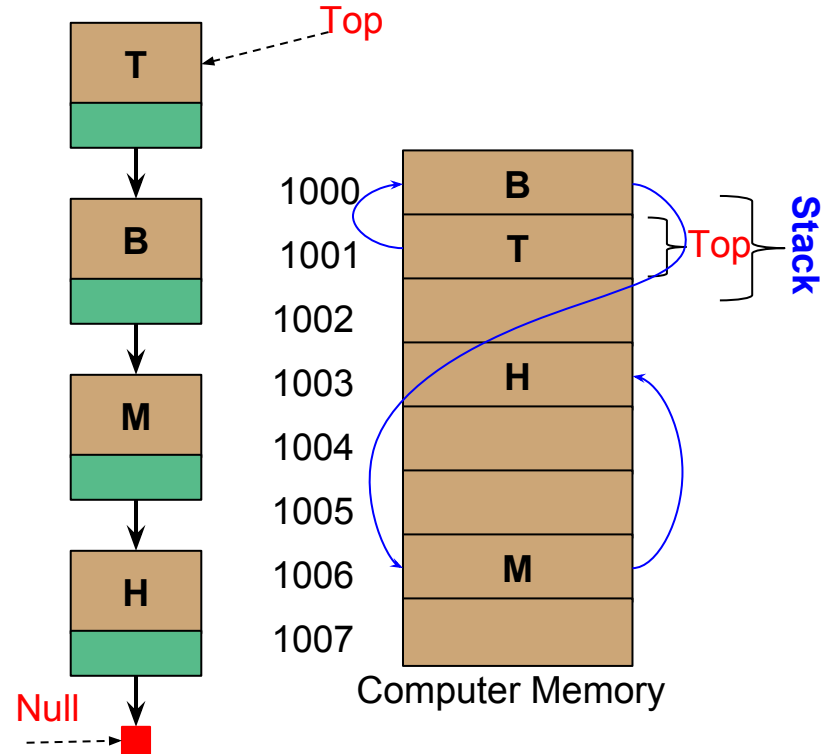
top
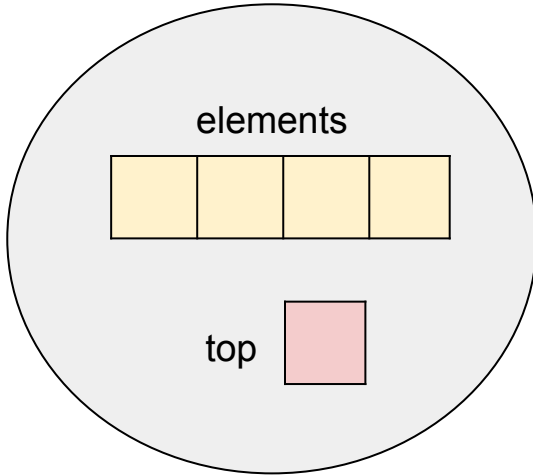
Stack Array based Implementation

elements

top

Stack Linked Lists based Implementation

Stack Linked Lists Implementation

The stack in this case is defined by a pointer top that points to the top (head) node in the linked List.

Stack Linked Lists based Implementation

Abstract Data Types

- Bob provided Alice with his Stack ADT specifications that inform Alice about the available interfaces and how she can use them.
- Bob will change his Stack ADT from array-based implementation to linked lists based Implementation
- Alice does not need to change her code at all. Because Bob will maintain his Stack ADT interfaces (specification)

```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));
```

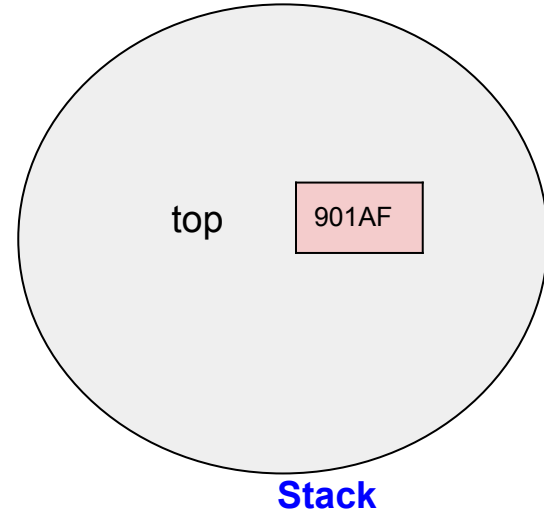**stack.h**

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
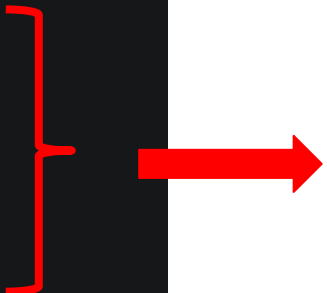
Stack ADT: Create Stack

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

Alice Code "Algorithm X"

top   901AF

Stack

Stack ADT Specification: Stack.h

```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));
```

```c
#define MAXSTACK 100

typedef  struct StackElement{

    int data;

}StackElement;


typedef struct stack{
    int top;

    StackElement elements[MAXSTACK];

}Stack;
```

Stack ADT: Create Stack

```c
2  typedef struct stack{
3
4      StackNode *top;
5
6  }Stack;
```

Bob's Code "Stack ADT"

```c
2  typedef struct stack{
3
4      StackElement *top;
5
6  }Stack;
```

Bob's Code "Stack ADT"

Which one is correct? And why?

New Stack ADT Specification: Stack.h

```c
typedef struct stackelement{

    int value;

}StackElement;

typedef struct stacknode{

    StackElement element;

    struct stacknode *next;

}StackNode;


typedef struct stack{

    StackNode *top;

}Stack;
```

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

top NULL

What is wrong??
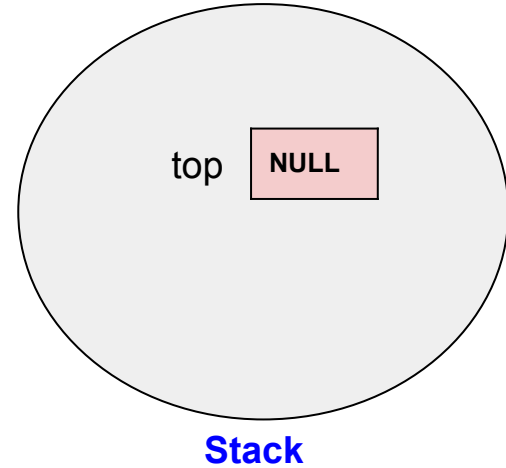
```
2  void InitiateStack(Stack *ptrStack){
3
4      ptrStack->top = nullptr;
5  }
```

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 are the pre & post conditions?

```
 2
 3   void Push(StackElement e, Stack *ptrStack){
 4
 5       StackNode * node = (StackNode*)malloc(sizeof(StackNode));
 6
 7       node->element =e;
 8
 9       node->next = ptrStack->top;
10
11       ptrStack->top = node;
12   }
```

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"

Why Is full stack return always ZERO ? How could we fix it?

```
2 int IsFullStack(Stack *ptrStack){
3
4     return 0;
5 }
6
```

Bob's Code "Stack ADT"

Stack ADT: Pop an Element

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

Alice Code "Algorithm X"

What is happening?

```
2  void Pop(StackElement *e, Stack *ptrStack){
3
4      *e = ptrStack->top->element;
5
6      StackNode * temp = ptrStack->top;
7
8      ptrStack->top = ptrStack->top->next;
9
10      free(temp);
11  }
```

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"

```
13 int IsEmptyStack(Stack *ptrStack){
14
15     return ptrStack->top == nullptr;
16 }
```

Bob's Code "Stack ADT"

Stack ADT: Delete | Empty Stack

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

Alice Code "Algorithm X"

What is the run-time complexity of deleting the stack?

```
4  void DeleteStack(Stack *ptrStack){
5
6      StackNode *crr = ptrStack->top;
7
8      while(crr!= nullptr){
9
10         crr = crr->next;
11
12         free(ptrStack->top);
13
14         ptrStack->top = crr;
15     }
16 }
```

Bob's Code "Stack ADT"

Stack ADT: Stack Size

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

Alice Code "Algorithm X"

What is the run-time complexity of this function?

```
3  int StackSize(Stack *ptrStack){
4
5      StackNode *crr = ptrStack->top;
6      int size =0;
7
8      while(crr!= nullptr){
9          crr = crr->next;
10         size++;
11     }
12
13     return size;
14 }
```

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"

```
2
3  void StackTop(StackElement *e, Stack *ptrStack) {
4
5      *e = ptrStack->top->element;
6  }
```

Bob's Code "Stack ADT"

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"

```
2 void TraverseStack(Stack *ptrStack, void (*ptrFun)(StackElement)){
3
4     StackNode *crr = ptrStack->top;
5
6     while(crr!= nullptr){
7
8         (*ptrFun)(crr->element);
9
10        crr = crr->next;
11    }
12 }
```

Bob's Code "Stack ADT"

Run-Time Complexity

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

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

Improving the Runtime of Stack Size

The runtime complexity of stack size is O(n).

How could we improve the runtime?

```cpp
int StackSize(Stack *ptrStack){

    StackNode *crr = ptrStack->top;
    int size =0;

    while(crr!= nullptr){
        crr = crr->next;
        size++;
    }

    return size;
}
```

Improving the Runtime of Stack Size

```
1
2  int StackSize(Stack *ptrStack){
3
4      return ptrStack->size;
5  }
```

```
12  typedef struct stack{
13
14      StackNode *top;
15
16      int size;
17
18  }Stack;
```

What else we need to change to support this new design?

Stacks Application: Parsing Arithmetic Expressions

- **Infix Notation:** arithmetic expressions are written with an operator (+, -, *, / , ^) placed between two operands.
- **Prefix Notation:** the operator written before the two operands.
- **Postfix Notation:** the operator written after the two operad

Infix Notation: A+B

Prefix Notation: +AB

Postfix Notation: AB+

Converting from Infix to Postfix

Infix	Postfix
A+B-C	AB+C-
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	?
A*B+C*D	?
(A+B)*(C-D)	?
((A+B)*C-D)	?

Converting from Infix to Postfix

Infix	Postfix
A+B-C	AB+C-
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C-D)	AB+CD-*
((A+B)*C-D)	AB+C*D-

How Humans Evaluate Infix

1. We read (scan) the expression from left to right.
2. When you have read enough to evaluate two operands and an operator, you calculate and substitute the answer for these two operands and operator
3. You continue this process until the end of expression.

3+4*5

Read	Parse and Evaluate
3	3
+	3+
4	3+4
*	3 + 4 *
5	3 + 4 * 5
	3 + 20
	23

Using Stack to Convert Infix to Postfix

A*B-C+E/D

A	*****
AB	*****
AB*	**-**
AB*C	**-**
AB*C-	**+**
AB*C-E	**+**
AB*C-E	**+/**
AB*C-ED/	**+**
AB*C-ED/+	

A + B * C

A	**+**
AB	**+**
AB	**+***
ABC	**+***
ABC*	**+**
ABC*+	

Using Stack to Evaluate Postfix

AB*C-ED/+

2 3 * 4 - 9 3 / +

2	
2 3	*
6	
6 4	-
2	
2 9	
2 9 3	/
2 3	
2 3	+
5	

ABC*+

2 3 4 * +

2	
2 3	
2 3 4	
2 3 4	*
2 12	
2 12	+
14	

Self Assessment

Give an example when using array-based implementation is more appropriate than using linked-list implementation.

Convert the following expression from infix to postfix

$$A+B*(C-D)/(E+F))$$

Write an algorithm to convert from infix to postfix using a stack

What is the advantage of Postfix notation (reverse Polish Notation)?