

Data Structures — Week #3

Stacks

Outline

- Stacks
- Operations on Stacks
- Array Implementation of Stacks
- Linked List Implementation of Stacks
- Stack Applications

Stacks (Yığınlar)

- A *stack* is a list of data with the restriction that *data can be retrieved from or inserted to the “top” of the list.*
- By “*top*” we mean a *pointer pointing to the element that is last added to the list.*
- A stack is a *last-in-first-out (LIFO)* structure.

Operations on Stacks

- Two basic operations related to stacks:
 - *Push* (Put data to the top of the stack)
 - *Pop* (Retrieve data from the top of the stack)

Array Implementation of Stacks

- Stacks can be *implemented by arrays*.
- During the execution, *the stack can*
 - *grow by push operations, or*
 - *shrink by pop operations*
- *within this array.*
- One end of the array is the bottom and insertions and deletions (removals) are made from the other end.
- We also need another field that, at each point, keeps track of the current position of the **top** of the *stack*.

Sample C Implementation

```
#define stackSize ...;
struct dataType {
    ...
}
typedef struct dataType myType;

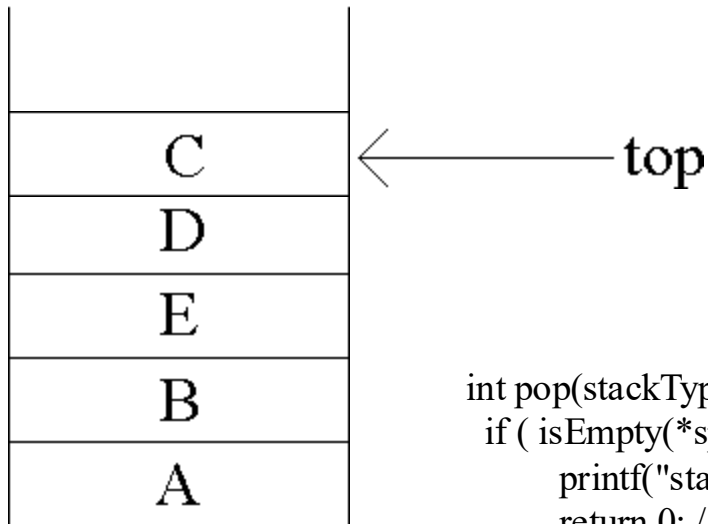
struct stackType {
    int top;
    myType items[stackSize];
}
typedef struct stackType stackType;
stackType stack;
```

} *stack structure*

Sample C Implementation... isEmpty()

```
//Initialize Stack (i.e., set value of top to -1)
stack.top=-1;
int isEmpty(stackType s)
{
    if (s.top == -1)
        return 1; //meaning true
    else return 0; //meaning false
}
```

Pop Operation

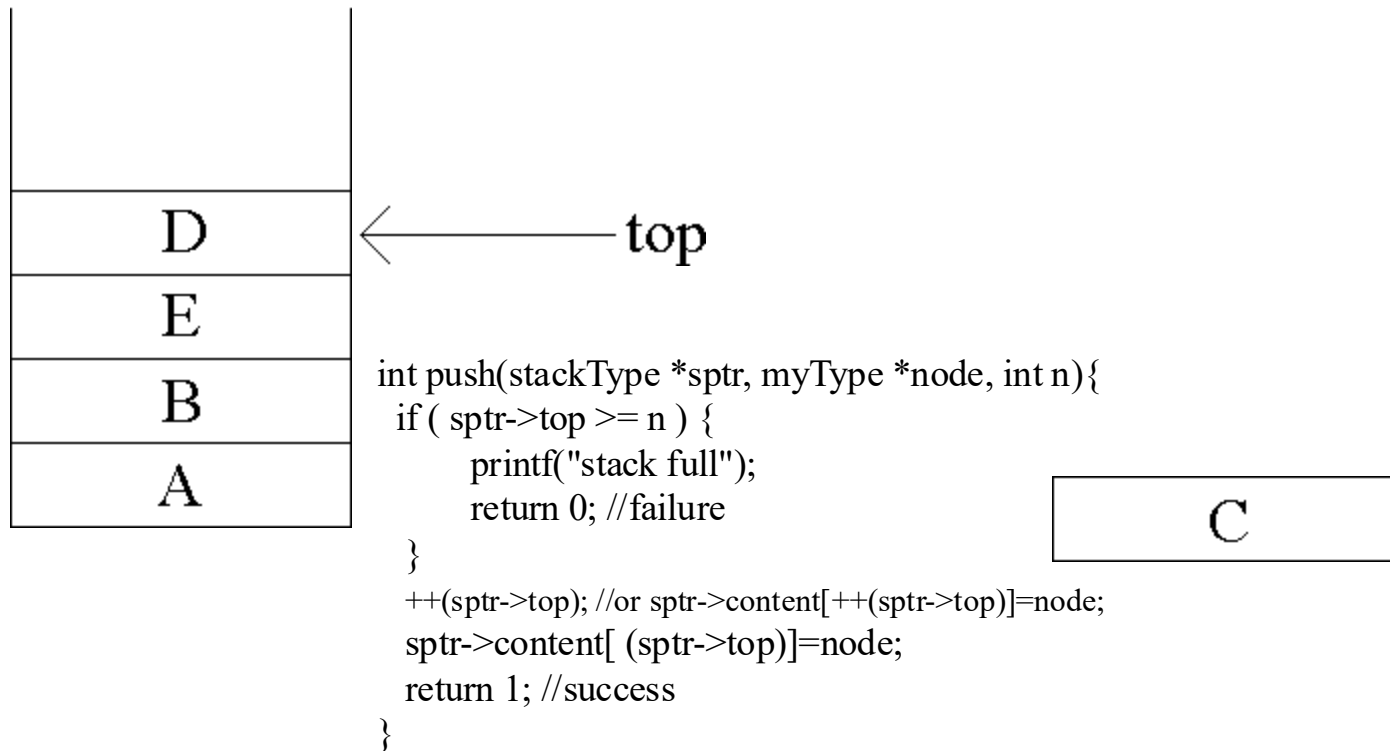


```
int pop(stackType *sptr, myType *node) {  
    if ( isEmpty(*sptr) ) {  
        printf("stack empty");  
        return 0; //failure  
    }  
    *node = sptr->items[sptr->top];  
    sptr->top--; //or *node = sptr->items[sptr->top--];  
    return 1; //success  
}
```

Sample C Implementation... pop()

```
int pop(stackType *sptr, myType *node) {  
    if ( isEmpty(*sptr) ) {  
        printf("stack empty");  
        return 0; //failure  
    }  
    *node = sptr->items[sptr->top--];  
    return 1; //success  
}
```

Push Operation



Sample C Implementation... push()

```
int push(stackType *sptr, myType *node, int n){  
    if ( sptr->top >= n ) {  
        printf("stack full");  
        return 0; //failure  
    }  
    sptr->items[++(sptr->top)]=*node;  
    return 1; //success  
}
```

Linked List Implementation of Stacks

//Declaration of a stack node

```
struct StackNode {  
    int data;  
    struct StackNode *next;  
}  
typedef struct StackNode StackNode;  
typedef StackNode * StackNodePtr;  
...
```

Linked List Implementation of Stacks

```
StackNodePtr NodePtr, top;
```

```
...
```

```
...
```

```
NodePtr = malloc(sizeof(StackNode));
```

```
top = NodePtr;
```

```
NodePtr->data=2;                // or top->data=2
```

```
NodePtr->next=NULL;            // or top->next=NULL;
```

```
Push(&top,&NodePtr); //Nodeptr is an output variable!!!
```

```
...
```

```
Pop(&top);
```

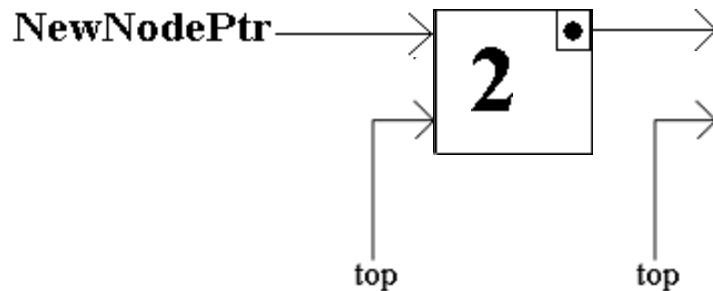
```
...
```

Push and Pop Functions

```
Void Push (StackNodePtr *TopPtr, StackNodePtr *NewNodePtr) {  
    *NewNodePtr = malloc(sizeof(StackNode));  
    // NewNodePtr to pass to invoking function!!!  
    (*NewNodePtr)->data=5;  
    (*NewNodePtr)->next = *TopPtr;  
    *TopPtr = *NewNodePtr;  
}
```

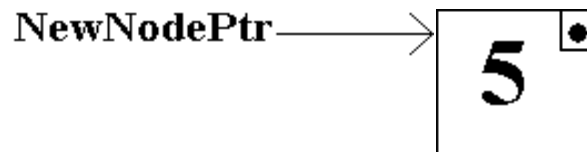
```
Void Pop(StackNodePtr *TopPtr) {  
    StackNodePtr TempPtr;  
    TempPtr= *TopPtr;  
    *TopPtr = *TopPtr->next;  
    free(TempPtr); // or you may return TempPtr!!!  
}
```

Linked List Implementation of Stacks



```
NodePtr = malloc(sizeof(StackNode));  
top = NodePtr;  
NodePtr->data=2; // or top->data=2  
NodePtr->next=NULL; // or top->next=NULL;  
Push(&top,&NodePtr);
```

```
Void Push (StackNodePtr *TopPtr,  
           StackNodePtr *NewNodePtr) {  
    *NewNodePtr = malloc(sizeof(StackNode));  
    (*NewNodePtr)->data=5;  
    (*NewNodePtr)->next = NULL;  
    (*NewNodePtr)->next = *TopPtr;  
    *TopPtr = *NewNodePtr;  
}
```



Stack Applications

- Three uses of stacks
 - Symbol matching in compiler design
 - Return address storage in function invocations
 - Evaluation of arithmetic expressions and cross-conversion into infix, prefix and postfix versions

Symbol Matching in Compiler Design

Algorithm:

1. Create an empty stack.
2. Read tokens until EOF. Ignore all tokens other than symbols.
3. If token is an **opening symbol**,
push it onto the stack.
4. If token is a **closing symbol** *and* stack empty,
report an error.
5. Else
pop the stack.
If symbol popped and opening symbol do not match
report an error
6. If EOF *and* stack not empty,
report an error
7. Else, the symbols are balanced.

Symbol Matching

Example:

```
int pop(Stack *sptr, myType *node) {  
    if ( isEmpty(*sptr) ) {  
        printf("stack empty");  
        return 0; //failure  
    }  
    *node = sptr->items[sptr->top--];  
    return 1; //success  
}
```

(
[
{

Use of Stacks in Function Invocation

- During a function invocation (function call)
 - Each argument value is copied to a local variable called “a dummy variable.” Any possible attempt to change the argument changes the dummy variable, not its counterpart in the caller.
 - Memory space is allocated for local and dummy variables of the called function.
 - Control is transferred to the called. Before this, return address of the caller must also be saved. This is the point where a **system stack** is used.

Use of Stacks in Function Invocation

Returning to the caller, three actions are taken:

1. Return address is retrieved.
2. Data area from the called is cleaned up.
3. Finally, control returns to the caller. Any returned value is also stored in known registers.

A Function Call Example

```
... main(...) {
n11 ...
n12 ...
n13 call f2(...);
r1 ...
n14 ...
}
```

```
... f2(...) {
n24 ...
n25 ...
n26 call f3(...);
r2 ...
n27 ...
}
```

```
... f3(...) {
n37 ...
n38 ...
n39 call f4(...);
r3 ...
}
```

```
... f4(...) {
n41 ...
n42 ...
n43 ...
}
```

Program Counter

n13

Stack Pointer

s0 empty

System Stack

s3

r₃

s2

r₂

s1

r₁

s0

r₀

Infix, Postfix and Prefix Formats of Arithmetic Expressions

The name of the format of arithmetic expression states the location of the operator.

Infix: operator is between the operands (L op R)

Postfix: operator is after the operands (L R op)

Prefix: operator is before the operands (op L R)

Examples to Infix, Postfix and Prefix Formats

Infix	Postfix	Prefix
$A+B$	$AB+$	$+AB$
$A/(B+C)$	$ABC+ /$	$/A+BC$
$A/B+C$	$AB/C+$	$+ /ABC$
$A-B*C+D/(E+F)$	$ABC*-DEF+ /+$	$+ -A*BC/D+EF$
$A*((B+C)/(D-E)+F)-G/(H-I)$	$ABC+DE- /F+*GHI- /-$	$-*A+ /+BC-DEF/G-HI$

Rules to watch during Cross-conversions

Associative Rules

- 1) $+$ and $-$ associate left to right
- 2) $*$ and $/$ associate left to right
- 3) Exponentiation operator ($^$ or $**$) associates from right to left.

Priorities and Precedence Rules

- 1) $+$ and $-$ have the same priority
- 2) $*$ and $/$ have the same priority
- 3) $(*$ and $/)$ precede $(+$ and $-)$

Algorithm for Infix→Postfix Conversion

1. Initialize an operator stack
2. While not EOArithmeticExpression Do
 - i. Get next token
 - ii. case token of
 - a. ‘(’: Push; //assume the lowest precedence for ‘(’
 - b. ‘)’: Pop and place token in the incomplete postfix expression until a left parenthesis is encountered;
If no left parenthesis return with failure
 - c. an operator:
 - a. If empty stack or token has a higher precedence than the top stack element, push token and go to 2.i
 - b. Else pop and place in the incomplete postfix expression and go to c
 - d. an operand: place token in the incomplete postfix expression
3. If EOArithmeticExpression
 - i. Pop and place token in the incomplete postfix expression until stack is empty

Evaluation of Arithmetic Expressions

1. Initialize an operand stack
2. While not EOArithmeticExpression Do
 - i. Get next token;
 - ii. Case token of
 - a. an operand: push;
 - b. an operator:
 - a. if the last token was an operator, return with failure;
 - b. pop twice;
 - c. evaluate expression;
 - d. push result;

Evaluation of Arithmetic Expressions

Example: 9 8 8 6 - / 2*1+- = ?

Token	Stack Content	Operation
9	9	None
8	9 8	None
8	9 8 8	None
6	9 8 8 6	None
-	9 8 2	8-6=2
/	9 4	8/2=4
2	9 4 2	none
*	9 8	4*2=8
1	9 8 1	None
+	9 9	8+1=9
-	0	9-9