Midterm Exam Info

- Time: Oct. 23, during class time, 45mintues
- Closed book, written exam
- Coverage: Weeks 0-5
- Types of questions: Multiple choices, True/false, short answer, open-ended questions.
  - More details on practice and sample questions
- How to prepare
  - Lecture content, Assignments, Labs
  - Practice questions coming one week before the midterm
  - Oct. 21 lecture, review and sample questions

Midterm Exam Info

- Makeup midterm
  - Oct. 28 (Mon.) class time
  - Requests must be sent to the course email **before** the midterm
  - No other makeup exam offered. If missed, the weight of midterm will be shifted to final. (Note: your final will have a 60% of the total grade)

Plans for Week of Oct. 14 - 28

- Week 6 (Oct. 14-18)
  - No class on Oct. 14 (holiday)
  - No labs and office hours on Oct 14&15
  - Classes, labs, and office hours are as usual on Oct. 16-18.
- Week 7 (Oct. 21-25)
  - Review lecture on Oct. 21
  - Midterm on Oct. 23
  - Labs and office hours are as usual on Oct. 21-23
  - No classes, labs, and office hours on Oct. 24&25

Plans for Week of Oct. 14 - 28

- Week 8 (Oct. 28 - Nov. 1)
  - Oct. 28 (Mon.) makeup midterm
  - Course lecture on Oct. 28 will be uploaded as videos.
  - No labs and office hours on Oct. 28&29
  - Classes, labs, and office hours are as usual on Oct. 30 – Nov. 1.

Information will be posted as CourseLink Announcement later as well.

Resources on CourseLink

- Garbage Collection

Stacks

Notes from Charlie Obimbo and revised by Yan Yan.

Contents

1. Definition of Stacks
2. Stacks Applications
3. Implementation

Learning Objectives

1. Define the data structure of the Stack ADT
2. Implement the operations of the Stack ADT.
3. Describe the advantages and disadvantages of linked list implementation and array implementation of the stack.

Stacks & Queues

- **Stacks & Queues**
  - – Stacks: Last in First Out

- Queues: First in First Out

Stack: Definition

- **Stack**: A stack is an ADT in which items are only inserted on or removed from the top of a stack.
- **Push** operation inserts an item on the top of the stack.
- **Pop** operation removes and returns the item at the top of the stack.
- A *linear* data structure, in which elements are accessed using the LIFO (Last in First Out) Order.

Stack: Some Applications

- Page-visited history in a Web browser
- Undo and Redo mechanism in a text editor
- Backtracking, i.e., to check parenthesis matching in an expression.
- Saving local variables when one function calls another, and this one calls another, and so on.
- String Reversal

Queue: Definition

- **Queue**: A queue is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.
- **Enqueue** operation inserts an item at the end of the queue.
- **Dequeue** operation removes and returns the item at the front of the queue.
- FIFO (First in First Out) ADT.

Queue: Some Applications

- Waiting lines
- Access to shared resources (e.g., printer)
- Maintaining the playlist in media players.

Stacks: Operations

- Mainly the following basic operations are performed in the stack:
  - **Push (stack, X)**
  - **Pop (stack)**
  - **Peek (stack)**
  - **isEmpty (stack)**

Push

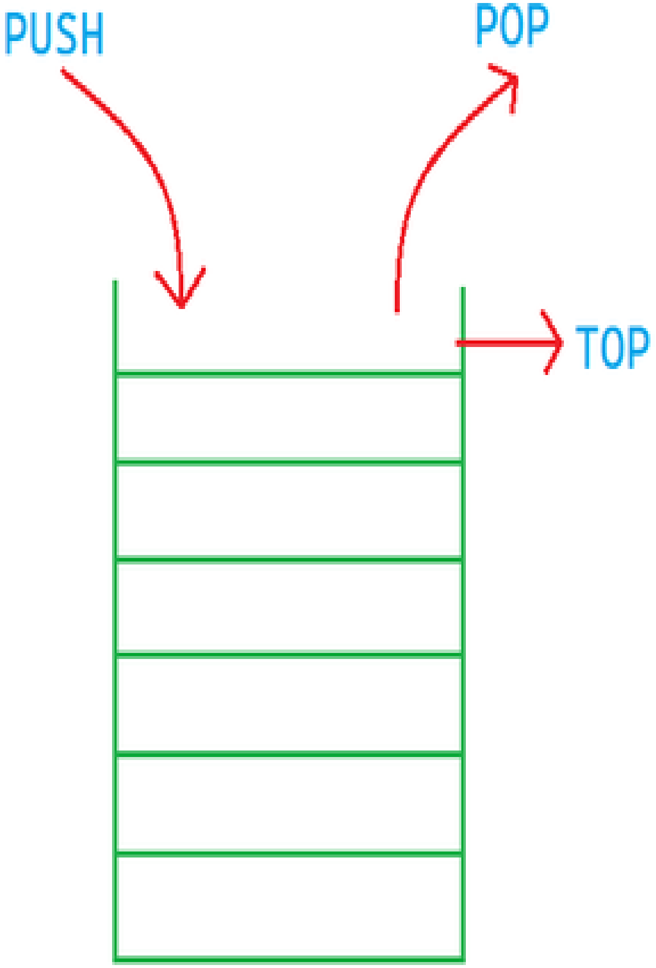
- Push: Adds an item to the stack. If the stack is full, then it is said to be an overflow condition.

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
   increment top
   stack[top] assign value
end else
end procedure
```

Stack

Insertion & Deletion
happens on the same end.



Pop

- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **underflow** condition.

Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
   store value of stack[top]
   decrement top
   return value
end else
end procedure
```

isEmpty

- **isEmpty**: Returns true if the stack is empty, else false.

Algorithm for isEmpty:

```
begin
if top < 1
   return true
 else
   return false
end procedure
```

Peek

- **Peek** or **Top**: Returns the top element of the stack.

Algorithm for peek:

```
begin
   if top == null // (or if isEmpty)
            return “stack is empty”
   else
  return stack[top]
end procedure
```

Time Complexities

- What are the time complexities of the operations on the stack?

- push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Stacks: Exercise - 1

Given numStack: 5, 9, 1 (top is 5)

- What is the stack after a pop operation Pop(numStack) ?
- Following the previous operation, what is the stack after a push operation Push(numStack, 8) ?

What operation determines if the stack contains no items?

Stacks: Exercise - 1 Solution

Given numStack: 5, 9, 1 (top is 5)

- What is the stack after a pop operation Pop(numStack) ?   9,1
- Following the previous operation, what is the stack after a push operation Push(numStack, 8) ? 8, 9,1

What operation determines if the stack contains no items? IsEmpty

Stacks: Exercise - 2

- What are the output of the following operations?

```
route = new Stack
Push(route, Tokyo)
Push(route, Osaka)
Push(route, Nara)
print Pop(route)
print Pop(route)
```

- What is the route stack after the above operations?

Stacks: Exercise - 3

- What are the output of the following operations?

```
route = new Stack
Push(route, Tokyo)
Push(route, Osaka)
Push(route, Nara)
print Peek(route)
print Pop(route)
```

- What is the route stack after the above operations?

Stacks

- **Implementation:**
  - Using array
  - Using linked list

```
// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};
//______________________________________________________________________________
// function creates a stack of given capacity. It initializes size to 0
struct Stack* createStack(unsigned capacity){
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
//______________________________________________________________________________
// Stack is full when top is equal to the last index
int isFull(struct Stack* stack){
    return stack->top == stack->capacity - 1;
}
```

```
// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack){
    return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item){
    if (isFull(stack))
        printf("Overflow\n");
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack){
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}
```

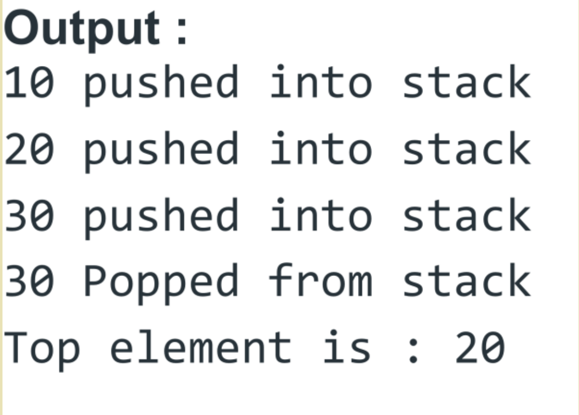
```
// Function to return the top from stack without removing it
int peek(struct Stack* stack){
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main() {
    struct Stack* stack = createStack(100);

   push(stack, 10);
    push(stack, 20);
    push(stack, 30);

  printf("%d popped from stack\n", pop(stack));
  printf(”Top element is: %d\n", peek(stack));

   return 0;
}
```



Output :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is : 20
```

Complete code see https://github.com/Yathish27/Data-Structure-using-C/blob/master/stack_g.c

Pros & Cons of Array Implementation

- **Pros**: Easy to implement. No additional pointer for each element is needed (save memory).

- **Cons**: It is not dynamic. It doesn’t grow and shrink depending on needs at runtime.

Stack Linked List Implementation

- **Head** of the list is the top of the stack.
- Push/Pop: update the top of the stack. Similar to adding/deleting the first node in a linked list.

Knowledge Prep: pointer to the pointer

- **Pointer (*p)** stores the information of an address (can be null)
- Pointer to the pointer (**p) stores the address of that pointer (*p)
- Use ** when you want to preserve (or retain change in) the memory-allocation or assignment even outside of a function call

Knowledge Prep: pointer to the pointer -- Example

```
#include <stdio.h>

int  main () {
      int c = 1;
      int d = 2;
      int e = 3;
      int * a = &c;
      int * b = &d;
      int * f = &e;
      int ** pp = &a; // pointer to pointer ‘a’

      a = b; // a stores d’ address
      cant_change(a, f);
      change(pp, f);
      return 0;
}
```

```
void cant_change(int * x, int * z){
      x = z;
      printf("\n ----> value of 'a' is: %x inside function, same as 'f', BUT NOT the same outside of this function \n", x);
}
void change(int ** x, int * z){
      *x = z;
      printf("\n ----> value of 'a' is: %x inside function, same as ‘f’,and the same outside of this function\n", *x);
}
```

Script adapted from https://stackoverflow.com/questions/5580761/why-use-double-indirection-or-why-use-pointers-to-pointer

```
// C program for linked list implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct StackNode {
    int data;
    struct StackNode* next;
};

// create a new node in the stack (implemented by linked list)
struct StackNode* newNode(int data) {
    struct StackNode* stackNode =
      (struct StackNode*)
      malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}
```

```
int isEmpty(struct StackNode* top) {
    return !top;
}
void push(struct StackNode** top, int data) {
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *top;
    *top = stackNode;
    printf("%d pushed to stack\n", data);
}
int pop(struct StackNode** top) {
    if (isEmpty(*top))
        return INT_MIN;
    struct StackNode* temp = *top;
    *top = (*top)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}
```

```
int peek(struct StackNode* top) {
    if (isEmpty(top))
        return INT_MIN;
    return top->data;
}

int main() {
    struct StackNode* top = NULL;

    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
     printf("%d popped from stack\n", pop(&top));

    printf("Top element is %d\n", peek(top));

   return 0;
}
```

Pros & Cons of LL Implementation

- **Pros**: The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- **Cons**: Requires extra memory due to involvement of pointers.

Exercise - 1

- An empty stack is indicated by a list head pointer value of ______.

A. newNode

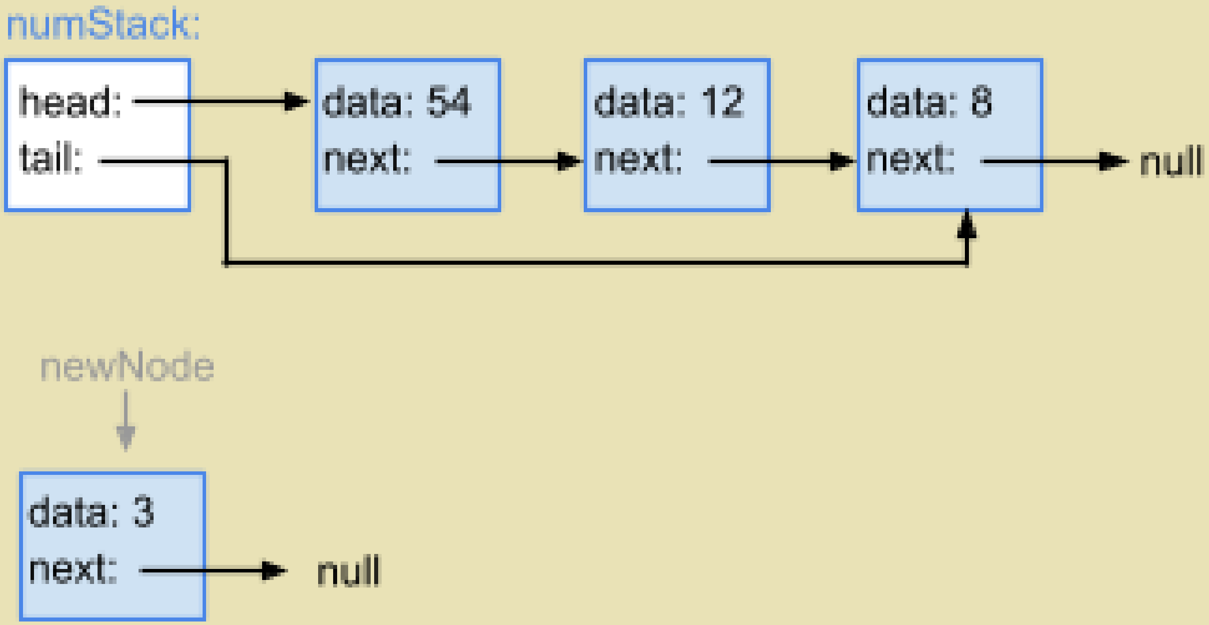
[✓ B. null]

C. Unknown

Exercise is from the textbook 5.2 Stacks using linked lists

Exercise - 2

- For StackPush(numStack, 3), newNode's next pointer is pointed to? **54**



Exercise is from the textbook 5.2 Stacks using linked lists

References and Useful Resources

- Zybook Data Structures 5) Stacks and Queues
- Signed and unsigned in Type Define
  - https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)IntegerTypes.html#:~:text=Unsigned%20variables%2C%20which%20can%20be,the%20whim%20of%20the%20compiler.
  - https://ece.uwaterloo.ca/~dwharder/icsrts/C/07/
- Array implementation of Stacks
  - https://www.geeksforgeeks.org/array-implementation-of-stack-in-c/

That's about this lecture!