University of Zurich[UZH]

Department of Informatics
Database Technology Group
Prof. Dr. M. Böhlen

Informatics II
Exercise 7

April 01, 2019

Abstract Data Types (ADT)

Task 1. Create a new datatype `stack` of positive integers that is able to store an arbitrary number of elements. You should implement stack using dynamic array, which initial size is `INITIAL_STACK_SIZE`, a constant equal to 5. When stack becomes full, increase the size of stack array for `INITIAL_STACK_SIZE` elements. You don't have to take care of stack size when removing elements from stack. A stack is of the following type:

```
typedef struct stackADT {
    int *elements;
    int size;
    int count;
} stack;
```

Along with the above datatype create the following functions:

- *void initialize(stack *s)* which initializes stack `s`.
- *int isEmpty(stack *s)* which checks if stack `s` is empty or not.
- *int push(stack *s, int value)* which inserts element `value` into `s` and returns the position where it was added in the `elements` array.
- *int pop(stack *s)* which removes an element from `s` and returns its value. In case of an error, -1 should be returned.
- *int compareStack(stack *sA, stack *sB)* which returns 1 if `sA` and `sB` are equal and 0 in any other case.
- *int printStack(stack *s)* which prints the values of the stack starting from the top element of `s`.

Test your implementation by performing the following operations:

1. Create and initialize stacks sA, sB.
2. Insert 1, 2, 3, 4, 5, 6, 7, 8 into sA, remove two elements and print the values of the removed elements.

3. Insert 10, 11, 12, 13 into sB
4. Print sB
5. Remove one element of sB and print the value of the removed element.
6. Insert 8 and 9 into sB.
7. Compare sA with sA and print the result.
8. Compare sA with sB and print the result.
9. Remove 4 elements from sA and print the values of the removed elements.
10. Delete the stacks.

Task 2. Consider having a stack implementation from the Task 1. What does the function *void func(stack *s, int n, int curr)* do? What is the output of the following program?

```
Algo: FUNC(s, n, curr)
-----------------------------------------
if isEmpty(s) or (curr == n) then
  return ;
value = pop(s);
func(s, n, curr+1);
if curr != n/2 then
  push(s, val);
```

```
Algo: MAIN()
-----------------------------------------
initialize(s);
push(s, 1);
push(s, 2);
push(s, 3);
push(s, 4);
push(s, 5);
push(s, 6);
push(s, 7);
push(s, 8);
push(s, 9);
curr = 0;
n = 9; //Stack size
func(s, 9, curr);
printStack(s);
```

Task 3. Consider a datatype `queue` corresponding to a queue of positive integers able to store `QUEUE_SIZE` elements, where `QUEUE_SIZE` is defined as a constant. A queue is of the following structure:

```
typedef struct queueADT {
    int elements[QUEUE_SIZE];
    int head;
```

```
    int count;
} queue;
```

Considering the `queue` datatype write the pseudocode for the following functions:

- *void initialize(queue *q)* which initializes the `head` and the `count` of `q`.
- *int enqueue(queue *q, int value)* which inserts the element `value` in `q` and returns the position it was added in the `elements` array. In case of an error, -1 should be returned.
- *int dequeue(queue *q)* which removes an element from `q` and returns its value. In case of an error, -1 should be returned.
- *int compareQueue(queue *qA, queue *qB)* that returns 1 if `qA` and `qB` are equal and 0 in any other case.

Task 4. The LIFO behaviour of a stack can be simulated using two FIFO queues. The elements of the stack are moved from one queue to another to properly perform the *push* and *pop* operations. After the completion of a *push* and *pop* operation, all elements are stored in only one of the two queues. Assume that a stack is implemented using two queues of identical size. Initially the queues, and thus the stack, are empty. Make a table to illustrate the sequence of queue operations needed to simulate the following sequence of stack operations: `push(4); push(5); push(6); pop(); pop(); push(7);` Draw the state of the queues after each stack operation.

Queue A	**Queue B**	**Stack Op**	**Queue Op**