Worksheet 4

Overview

The w orkshop for Week 5 w ill start w ith a tutorial on Stacks and Queues.

Tutorial Questions

Question 4.1 A *stack* is an abstract data structure that behaves much like a stack of books on a desk: The new est additions go on top, and the new est additions are also the first to come off the stack.

Show :

- How you could implement a stack using an array as the underlying data structure. Think about how you w ould deal w ith the situation w here the stack is full.
- How you w ould implement the stack using a linked list as the underlying data structure.

Question 4.2 A *queue* is an abstract data structure that behaves much like a queue in a grocery shop: The new est additions to the queue go at the end (tail, and items are taken out of the queue and processed from the front (head).

Show :

- How you could implement a queue using an array as the underlying data structure. Think about how you w ould deal w ith the situation w here the queue is full. Discuss w hether a *circular array* can be used.
- How you w ould implement a queue using a linked list as the underlying data structure.

Discuss the pros and cons of the linked list and array implementations.

Programming exercises

Programming 4.1 In the tutorial section, w e discussed *stacks* and *queues* implemented using arrays or linked lists as the underlying data structure. In this exercise, you are asked to implement a stack based on a linked list. This exercise should give you practice in using *malloc* to make nodes, filling up the fields w ithin nodes, and abstracting implementation details into functions.

You should have at least the operations (functions) *makeStack*, *push*, *pop*. Having an operation to check w hether the stack is *empty* is also a good idea. Separate each stack operation into a different function! The functions should take a pointer to the stack (list) as an argument, so that multiple queues can operate w ithin a program using the same functions, and possibly arguments, such as the item to be pushed.

For an extra challenge, consider how you w ould free all the memory used by the stack once your program is finished w ith it.

```
/* stack.h */
struct stack;

/* Returns a pointer to an empty stack */
struct stack *makeStack();

/* Adds an item to the top of the stack. */
void push(struct stack *stack, int item);

/* Returns the top item from the stack. */
int pop(struct stack *stack);

/* Checks whether the stack is empty. */
int empty(struct stack *stack);
```

```c
/* Frees the stack and sets the pointer at the location
    provided to NULL. */
void freeStack(struct stack **stack);

/* stack.c */
#include <stdlib.h>
#include <assert.h>

struct stack {
    /* FILL IN: Define stack struct here. */
};

struct stack *makeStack(){
    /* FILL IN */
    return NULL;
}

void push(struct stack *stack, int item){
    /* FILL IN */
}

int pop(struct stack *stack){
    /* FILL IN */
    return 0;
}

int empty(struct stack *stack){
    /* FILL IN */
    return 1;
}

void freeStack(struct stack **stack){
    /* FILL IN */
}

/* main.c */
/* #include "stack.h" */
#include <stdio.h>
#include <stdlib.h>

#define SIZE 7

int main(int argc, char **argv){
    int i;
    int current;
    struct stack *stack1 = makeStack();
    struct stack *stack2;

    if(empty(stack1)){
        printf("Stack 1 created and empty.\n");
    } else {
        printf("ERROR: Stack 1 not empty when created!\n");
    }

    printf("Adding 1 to stack 1.\n");
    push(stack1, 1);

    printf("Taking top item from stack 1.\n");
    printf("Top item on stack 1 was: %d.\n",pop(stack1));

    printf("Adding numbers %d to %d to stack 1.\n",1,SIZE);
    for(i = 0; i < SIZE; i++){
        push(stack1, i + 1);
    }

    if(empty(stack1)){
        printf("ERROR: Stack 1 empty.\n");
    } else {
        printf("Stack 1 not empty after items added.\n");
    }

    stack2 = makeStack();
    if(empty(stack2)){
        printf("Stack 2 created and empty.\n");
    } else {
        printf("ERROR: Stack 2 not empty when created!\n");
    }

    printf("Taking items from stack 1 and adding them to stack 2.\n");
    printf("Stack 1: ");
    for(i = 0; i < SIZE; i++){
        current = pop(stack1);
        printf("%d",current);
        /* Intercalate commas */
        if(i != (SIZE - 1)){
            printf(", ");
        }

        push(stack2, current);
    }
    printf("\n");

    if(empty(stack1)){
        printf("Stack 1 emptied.\n");
    } else {
        printf("ERROR: Stack 1 not empty when all items removed!\n");
    }

    if(empty(stack2)){
        printf("ERROR: Stack 2 erroneously empty after items being added.\n");
    }

    printf("Taking items from stack 2 and printing them.\n");
    printf("Stack 2: ");
    for(i = 0; i < SIZE; i++){
        current = pop(stack2);
        printf("%d",current);
        /* Intercalate commas */
        if(i != (SIZE - 1)){
            printf(", ");
        }
    }
    printf("\n");

    if(empty(stack2)){
        printf("Stack 2 emptied.\n");
    } else {
        printf("ERROR: Stack 2 not empty when all items removed!\n");
    }

    freeStack(&stack1);
    freeStack(&stack2);

    return 0;
}
```

Programming Challenge 4.1 In this challenge, you will write part of the LZ77 compression algorithm, used in a number of leading compression algorithms today. The process will be to add characters to a stack one by one, when you meet a different character, or the end of the string, you should pop all items from the stack, counting how many items you added and then printing out the character followed by the number of times it repeats. The output of your program could be used as a preliminary step to build an LZ77 implementation which could quickly consider more input at the same time.

So for example "Testting" should be turned into "T1e1s1t2i1n1g1".

```
/* Add your stack implementation from Programming 4.1
    and change the data type from integer to character */

/* Compresses all duplicate characters into character and
    count pairs in given input and prints the results. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void compress(char *input);

void compress(char *input){
    /* FILL IN HERE */
}

int main(int argc, char **argv){
    char *input = "Aaaaaa, screamed the amateur magician as the aardvark ate the apple he'd prepared...";

    compress(input);

    return 0;
}
```

Programming Challenge 4.2 Another common problem in the real w orld is bracket matching. For this problem, you'll need to w rite a function w hich reads through a string of text and prints out w hether the brackets are balanced or not.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PROBLEMCOUNT 9
/*
    Returns 1 if the null-terminated string given has
    matched brackets and 0 otherwise. Only parentheses
    "()" are considered brackets for the purposes of
    this problem (though others are, in principle,
    relatively trivial extensions of logic you've
    applied before, so if you'd like to give it a
    go, feel free).
*/
int bracketCheck(char *input);

int bracketCheck(char *input){
    /* FILL IN HERE */
}

int main(int argc, char **argv){
    int i;
    char *problems[PROBLEMCOUNT];
    problems[0] = "No brackets.";
    problems[1] = "(One level of brackets)";
    problems[2] = "(Missing end bracket";
    problems[3] = "((Two levels of bracket))";
    problems[4] = "((((Four levels of bracket))))";
    problems[5] = "((Set)(Of)(Brackets)(at)(same)(level))";
    problems[6] = "(((Set)(Of))(Brackets)(at(different)((levels))))";
    problems[7] = "(((Set)(Of))(Mismatched(Brackets)(at(different)((levels))))";
    problems[8] = "Missing start bracket)";

    for(i = 0; i < PROBLEMCOUNT; i++){
        if(bracketCheck(problems[i])){
            printf("Matched:   %s\n",problems[i]);
        } else {
            printf("Unmatched: %s\n",problems[i]);
        }
    }

    return 0;
}
```