CS 571 Homework 1

Total Score : 55

Instructions

This homework consists of 4 theoretical questions and 1 programming question

Please provide your answers to all questions in a PDF format.

- Upload the PDF containing your answers to Gradescope.
- Submit the programming source tarball on Brightspace under the appropriate assignment.

1 The XOR Linked List — Total Score: 10

This section concerns the implementation of linked list data structures in C. For your reference, we have provided descriptions and implementations of linked lists in the supplementary Section A at the end of this document.

Part A

There is a way to provide the benefits of a *doubly-linked list* using the same space as a *singly-linked list* that is sometimes used on very space-constrained systems. The technique takes advantage of the properties of the bitwise XOR function. Specifically, it uses the property that (a^b)^b is equivalent to a, where a and b are numeric variables and ^ is the bitwise XOR operator in C. An example definition of a node for this type of linked list is given below:

```c
struct xdll_node {
  int elem;
  struct xdll_node * nxp;
};
```

Please provide implementations for the *iter* and *rotate* (definitions for these functions on a standard doubly-linked list are in Section A) functions for the XOR *doubly-linked list*.

Hint: The following is a function which swaps two variables in-place with the XOR instruction:

```c
void xor_swap(int * a, int * b)
{
  *a = *a ^ *b;
  *b = *b ^ *a;
```

```
    *a = *a ^ *b;
}
```

Part B

What is the disadvantage of the XOR doubly-linked list over the traditional doubly-linked list? Explain.

2 Higher-order Pointers **Total Score: 10**

In the following program, identify all instances of a) garbage, and b) references to dangling pointers.

Instructions:

- State clearly which problems occur, referring to line number(s) in the code where relevant.
- Assume we have unlimited memory, and under all circumstances, memory will be allocated successfully.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  
4  int main() {
5      int **ptr1 = (int **)malloc(sizeof(int *));
6      if (ptr1 != NULL) {
7          *ptr1 = (int *)malloc(sizeof(int));
8          if (*ptr1 != NULL) {
9              **ptr1 = 42;
10             printf("Value: %d\n", **ptr1);
11             int *ptr2 = (int *)malloc(sizeof(int));
12             free(*ptr1);
13             printf("ptr 2 Value: %d\n", *ptr2);
14             printf("ptr 1 Value: %d\n", **ptr1);
15             *ptr1 = ptr2;
16             **ptr1 = 99;
17         }
18     }
19     return 0;
20 }
```

3 Stack Size Estimation **Total Score: 10**

One of the advantages of C is that the low-level control it provides can enable you to precisely determine the performance properties of programs. Consider the following C program:

```c
int f(int x) {
    if (x == 1) {
        return 0;
    } else if (x % 2 == 0) {
        return f(x/2) + 1;
    } else {
        return f((3 * x) + 1);
    }
}
int g() {
    int v1 = f(4);
    int v2 = f(5);
    return v1 + v2;
}
```

For the purpose of this question, you may assume that: 1) integers are 4 bytes, 2) all pointers are 8 bytes, and 3) the stack contains no extraneous information such as stack canaries or padding. How many bytes of stack space does executing a call to g() require? Justify your answer.

4 Smart Pointers — Total Score: 10

Consider the linked list and doubly-linked list presented in the supplementary Section A at the end of this document. Imagine you were to replace all pointer definitions with smart pointer definitions in the structs and the *insert front*, *iter*, and *rotate* functions, and consider the consequences in terms of memory safety and garbage.

1. Is it possible to implement a (singly-) linked list with unique pointers?
2. Is it possible to implement a (singly-) linked list with shared pointers?
3. Is it possible to implement a doubly-linked list with unique pointers?
4. Is it possible to implement a doubly-linked list with shared pointers?

Explain your reasoning for all answers.

Programming Assignment: Abstract Syntax Tree Manipulation

Introduction

In this programming assignment, you will work on manipulating an Abstract Syntax Tree (AST) data structure for a simple calculator language. The assignment consists of three parts, each focusing on different aspects of AST construction, memory management, and expression evaluation.

All necessary code resources to get started will be uploaded to bright space under : content/assignment1

Part 1: AST Construction **Score: 5**

In `expr.c`, fill in the functions `mk_expr1`, `mk_expr2`, and `mk_expr3` to construct the listed expression ASTs. Utilize the provided helper constructors in `expr.c` appropriately. The expected output from running `make; ./expr` should match the documentation in the comments in `main.c`, excluding the parts dependent on `eval`.

```
// Example:
// make; ./expr
// Output: ...
```

Part 2: Memory Management **Score: 5**

Complete the `free_expr` function in `expr.c`. Ensure that this function frees all memory associated with the input expression AST, without causing any double-free or dangling-pointer reference errors. Running `make; valgrind --tool=memcheck ./expr` should show no memory leaks.

```
// Example:
// make; valgrind --tool=memcheck ./expr
// Output: ...
```

Part 3: Expression Evaluation **Score: 5**

Complete the `eval` function in `expr.c`. This function should evaluate an input expression AST. After completing this part, the output from running `make; ./expr` should align with the comments in `main.c`.

```
// Example:
// make; ./expr
// Output: ...
```

Testing

Test your code on various cases beyond the provided examples. The program must compile with the `-Wall -Wextra -Werror` options enabled.

Instructions for Remote Server Validation

This README provides instructions for testing and validating your code on the remote server (`remote.cs.binghamton.edu`). Ensure your code works correctly on the remote machine before submission.

Accessing the Remote Server

Connect to the remote server using SSH. Replace `your_username` with your actual username and `remote.cs.binghamton.edu` with the server address.

```
ssh your_username@remote.cs.binghamton.edu
```

Enter your password when prompted.

Navigating to Your Assignment Directory

Navigate to the directory where you have stored your assignment files.

```
cd path/to/your/assignment
```

Compiling Your Code

Compile your code using the provided `Makefile`. Ensure that you have included the necessary compiler options (`-Wall -Wextra -Werror`) for good programming practices.

```
make
```

Testing the Program

Run the executable on the remote machine to ensure it works as expected. Replace `./expr` with the name of your compiled executable.

```
./expr
```

Compare the output with the expected results mentioned in the comments in `main.c`.

Memory Leak Check

Use `valgrind` to check for memory leaks.

```
valgrind --tool=memcheck ./expr
```

Ensure there are no memory leaks reported.

Submitting Your Solution

- Ensure your code meets the requirements and adheres to good programming practices.
- Test thoroughly on the remote machine before submission.
- **Note:** No re-grading is encouraged if the solution is correct but does not work on the remote server.
- Compress all the programming assignment files into a tarball.
- Submit the tarball on Brightspace under `assignments/assignment1`.
- Submit the PDF with all the solutions for non-programming parts to Gradescope.

A Additional Resources

Linked Lists

A *linked list* is a data structure type that is rather important for C programming. The basic structure of a linked list is a *node*, which contains an element and a pointer to the next node in the list. The first node is often referred to as the *head*. The following is an example of a node definition and some interface functions for a linked list:

```c
typedef struct ll_node {
    int elem;
    struct ll_node * next;
};

/* Insert an element at the front of the list
   Returns a new head pointer
*/
struct ll_node * insert_front(struct ll_node * head, int elem)
{
    // Returns a pointer to a new node
    struct ll_node * new_head = malloc(sizeof(struct ll_node));
    new_head->elem = elem;
    new_head->next = head;
    return new_head;
}

/* Iterate through the list, and call the
   passed-in function on each element.
*/
void iter(struct ll_node * list,
          void (*iter_fn)(struct ll_node *elem)) {
    struct ll_node * cur = list;
    while(cur != NULL) {
        iter_fn(cur);
        cur = cur->next;
    }
}
```

Similarly, there is a structure called a *doubly-linked list*, which maintains both a *next* pointer and a pointer to the previous element. An example definition of a *doubly-linked list* node is given below:

```c
struct dll_node {
    int elem;
    struct dll_node * next;
    struct dll_node * prev;
};

```

```
/* Insert an element at the front of the list
   Returns a new head pointer
*/
struct dll_node * insert_front(struct dll_node * head, int elem)
{
    // Returns a pointer to a new node
    struct dll_node new_head = malloc(sizeof(struct dll_node));
    new_head->elem = elem;
    new_head->next = head;
    new_head->prev = NULL;
    return new_head;
}

/* Iterate through the list, and call the
   passed-in function on each element.
*/
void iter(struct dll_node * list,
          void (*iter_fn)(struct dll_node *elem)) {
    struct dll_node * cur = list;
    while(cur != NULL) {
        iter_fn(cur);
        cur = cur->next;
    }
}

/* Rotate the back of the list to the head
   Returns the new head
*/
struct dll_node * dll_rotate(struct dll_node * head)
{
    struct dll_node * temp = head;
    struct dll_node * tail;
    if(temp->next == NULL)
        return head;
    while(temp->next != NULL)
        temp = temp->next;
    temp->next = head->next;
    head->prev = temp->prev;
    head->next = NULL;
    temp->prev = NULL;
    return temp;
}
```

Copying Files to the Remote Server

If you need to transfer your files from your local machine to the remote server (`remote.cs.binghamton.edu`), you can use the `scp` command. Follow the steps below:

Copy a Single File

To copy a single file, use the following command. Replace `local_file` with the path to your local file and `remote_directory` with the destination directory on the remote server.

```
scp local_file your_username@remote.cs.binghamton.edu:remote_directory/
```

Enter your password when prompted.

Copy a Directory

To copy an entire directory, use the `-r` option. Replace `local_directory` with the path to your local directory and `remote_directory` with the destination directory on the remote server.

```
scp -r local_directory your_username@remote.cs.binghamton.edu:remote_directory/
```

Enter your password when prompted.

Example

Here's an example for copying a file and a directory:

```
# Copy a single file
scp myfile.c your_username@remote.cs.binghamton.edu:~/assignment/

# Copy a directory
scp -r myproject/ your_username@remote.cs.binghamton.edu:~/assignment/
```

Adjust the paths and filenames based on your actual file structure.