Roleen Ferrer 1731261

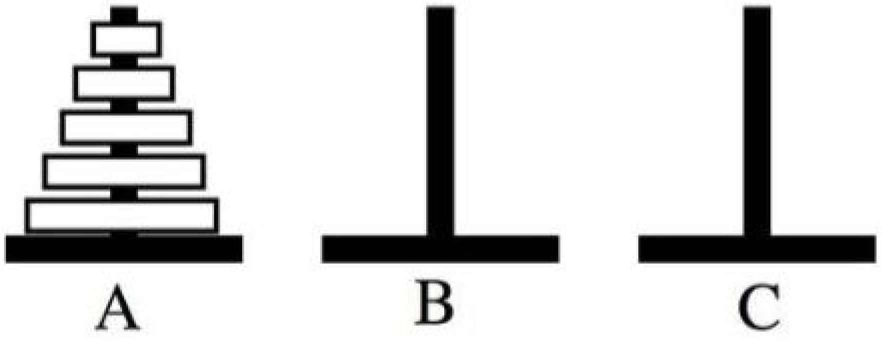
Professor Maxwell Dunne

CSE 13S - Computer Systems and C Programming

Assignment 3: The Tower of Brahma

In this lab, I will be re-creating the "End of the World Puzzle", which is most popularly known as "The Tower of Hanoi". The game is plays like this:

1. There are 3 pegs that are able to hold rings.



2. In order to finish the game, the player must move all the disks from peg A to peg B, but there are a couple of rules:
    a. You may only move one disk at a time
    b. You are not able to place a larger disk on top of a smaller disk.

The game progressively gets harder with a greater amount of disks. Coincidentally, the minimum amount of moves a player can make is related to to the formula:

$2^n - 1$, where n is the number of disks.

I will be implementing this game using two ways: **Recursion** and using a **Stack**.

IMPLEMENTATION

Main

The main function will be primarily used to run getopt() to collect commands from the user via the command line. These commands are:

1. **- n x**
    a. Parameters: x is some integer

    b. This command assigns the number of disks to be used to play the game

2. **- r**
    a. This command will run the game using the recursive implementation
3. **- s**
    a. This command will run the game using the stack implementation

Thus,

1. For each command that the user passes through getopt(), these commands will be stored into an array of 10 characters
    a. char tasks[10]
    b. Using an array keeps the input of commands in order and also limits the amount of commands the user is able to input
2. After the user issues commands, a for loop will parse through all of the characters in the tasks[10] array
3. Each character in the array will represent what function will be run
    a. if the character is “r”, it will run the recursive function once
    b. if the character is “s”, it will run the stack function once
4. At the end of the function call, the program will print the amount of minimum moves the program took, in this case, following the formula $2^n - 1$

<u>**Recursive Function**</u>

void recursiveTower(int disks, char column_A, char column_B, char column_C)

This function represents the recursive function that main will call. The bulk of the recursive process in solving the game is in this function.

Parameters:

- int disks - represents the amount of disks to run the game with
- char column_A - represents peg A
- char column_B - represents peg B
- char column_C - represents peg C

Recursive algorithm:

1. Start of the function by checking **if (disks == 1)** to check if it is the largest disk. If it is, print the disk moving from column A to B and end the recursive function.

    a. The recursive algorithm will also check **if (disks == 0)**, which is primarily used as an error check. The function will end if this condition is met.
2. The next line begins calling the function again, but this time it will:
    a. Subtract the number of disks by 1
    b. Leave column_A as peg A
    c. Change the parameter column_B into peg C
    d. Change the parameter column_C into peg B
3. I will then again print the disk that is moving from the parameter of column_A to column_B.
4. The next line will call the function again, but this time:
    a. Subtract the number of disks by 1
    b. Change the parameter column_A into peg C
    c. Change the parameter column_B into peg B
    d. Change the parameter column_C into peg A
5. The function will recursively call itself until the condition of **if (disks == 1)** is met.

Stack Function

void stackTower(int disks)

This function represents the stack function that main will also call. This function plays the game using a stack, rather than using a recursive function.

Parameters:

- int disks - represents the amount of disks to run the game with

This function will also be using the source file **stack.c** to call functions for manipulating the stack. These functions are as followed:

1. Stack *stack_create(int capacity, char name) - which is a constructor for a new stack
2. void stack_delete(Stack *s) - which is a destructor for a stack
3. int stack_pop - which pops an item off of a stack if it isn’t empty
4. void stack_push(Stack *s, int item) - which pushes an item into a stack if it isn’t full
5. bool stack_empty(Stack *s) - which returns true if a stack is empty, else false
6. int stack_peek(Stack *s) - which returns the current top value of the stack

Stack implementation:

1. Begin with creating a Stack for all 3 pegs
    a. struct Stack *column_A, struct Stack *column_B, struct Stack *column_C
    b. Give each stack the number of maximum number of disks they can hold and their char name.
2. Push the amount of rings into peg A using a for loop
    a. for (int x = disks; x >= 1; x = x - 1)
        i. stack_push(column_A, x);
3. Create a while loop that will continuously iterate through all disks and pegs until the top of column_B is the number of disks, which signal the end of the game.
    a. Have an i iterator, which will reset back to i = 1 if the maximum number of rings is called.
4. Check if the current i (disk) value is either odd or even and check if the total number of disks is odd or even
    a. The program works differently if the total number of disks is odd or even, so it is imperative that there is this check
        i. if the current disk (i) is odd
            1. legalMovementOdd(i, column_A, column_B, column_C) if total disks is odd
            2. legalMovementEven(i, column_A, column_B, column_C) if total disks is even
        ii. if the current disk (i) is even
            1. legalMovementEven(i, column_A, column_B, column_C) if total disks is odd
            2. legalMovementOdd(i, column_A, column_B, column_C) if total disks is even
5. After the loop, delete the stacks that we created to re-allocate memory used

Helper functions:

*void legalMovementOdd(int disk, struct Stack *column _A, struct Stack *column_B, struct Stack *column_C)*

and

*void legalMovementEven(int disk, struct Stack *column _A, struct Stack *column_B, struct Stack *column_C)*

These helper functions will run if the current disk is odd or even. They are quite similar,

There are only 3 moves that a disk can move if the total number of disks is odd:

- if current disk is odd, it can only move from peg C-A, B-C, and A-B
- if current disk is even, it can only move from peg A-C, C-B, and B-A

There are only 3 moves that a disk can move if the total number of disks is even:

- if current disk is odd, it can only move from peg A-C, C-B, and B-A
- if current disk is even, it can only move from peg C-A, B-C, and A-B

Notice the differences when the total amount of disks are flipped.

With this pattern:

```
1. Create 3 conditions to see where to push and pop (ODD function)
    a. if stack_peek(column_A) == disk)
        i. if stack_empty(column_B) == 1 or stack_peek(column_A) <
           stack_peek(column_B)
            1. Declare x as a variable to hold stack_pop(column_A)
            2. Push x to column_B
            3. Print the move
    b. if stack_peek(column_B) == disk)
        i. if stack_empty(column_C) == 1 or stack_peek(column_B) <
           stack_peek(column_C)
            1. Declare x as a variable to hold stack_pop(column_B)
            2. Push x to column_C
            3. Print the move
    c. if stack_peek(column_C) == disk)
        i. if stack_empty(column_A) == 1 or stack_peek(column_C) <
           stack_peek(column_A)
            1. Declare x as a variable to hold stack_pop(column_C)
            2. Push x to column_A
            3. Print the move
2. Create 3 conditions to see where to push and pop (EVEN function)
```

```
a. if stack_peek(column_A) == disk)
    i. if stack_empty(column_C) == 1 or stack_peek(column_A) <
       stack_peek(column_C)
        1. Declare x as a variable to hold stack_pop(column_A)
        2. Push x to column_C
        3. Print the move
b. if stack_peek(column_B) == disk)
    i. if stack_empty(column_A) == 1 or stack_peek(column_B) <
       stack_peek(column_A)
        1. Declare x as a variable to hold stack_pop(column_B)
        2. Push x to column_A
        3. Print the move
c. if stack_peek(column_C) == disk)
    i. if stack_empty(column_B) == 1 or stack_peek(column_C) <
       stack_peek(column_B)
        1. Declare x as a variable to hold stack_pop(column_C)
        2. Push x to column_B
        3. Print the move
```

stack.c Functions

The stack.c functions are crucial when solving this game using the stack iterative method. We have seen and implemented many of these in class (See Lecture 14 Slides) These functions and implementations are as followed:

1. Stack *stack_create(int capacity, char name)
    - a. Parameters
        - i. int capacity - which takes in an integer for the amount of space to allocate
        - ii. char name - which takes in the name of the stack (e.g "A", "B", or "C")
    - b. To implement the stack correctly to the heap, I used malloc to allocate the size of bytes of memory.
    - c. Create a conditional statement to return if the stack is not able to be created
        - i. if (!s)

    d. Create pointers using the arrow operator which point to certain variables in the struct function. These include
        i. s->name - which stores the name of the stack
        ii. s->capacity - which stores the capacity of the stack
        iii. s->items - which stores the items (disks) in the stack
            1. Using (int *)calloc(capacity, sizeof(int)) to allocate the bytes to memory

2. void stack_delete(Stack *s)
    a. Parameters
        i. Stack *s - which takes in a stack
    b. This function will properly delete the stack
        i. Call free(s->items) to free the memory that was stored in the stacks
        ii. Then call free(s) to free the stack from memory
    c. It is imperative that it is done in this order, as some errors due to memory dereferencing may occur
3. int stack_pop(Stack *s)
    a. Parameters
        i. Stack *s - which takes in a stack
    b. This function will pop the top most value of the stack and return it
        i. Return s->items[s->top] if certain conditions are met, such as (s->top > 0)
4. void stack_push(Stack *s, int item)
    a. Parameters
        i. Stack *s - which takes in a stack
        ii. int item - which takes in the item to push into the stack
    b. This function will push an item onto its stack.
    c. Create a conditional to check if the stack is valid
    d. Create a conditional which will check if(s->top == s->capacity)
        i. This will reallocate the memory from this push
    e. Create a conditional that will update the current s->top item
        i. s->items[s->top] = item
        ii. s->top += 1

5. bool stack_empty(Stack *s)
    a. Parameters
        i. Stack *s - which takes in a stack
    b. This function will return True if the stack is empty, otherwise it will return False
        i. return s->top == 0
6. int stack_peek(Stack *s)
    a. Parameters
        i. Stack *s - which takes in a stack
    b. This function will return the current value at the top of the stack. This is very helpful in looking at the top most value for comparisons
        i. if (s->top ==0) return s->top
            1. else return s->items[s->top - 1]