Asgn 3 Tower of Hanoi Design Document

Despina Patronas

Stack header file

//defines struct Stack and its function prototypes

Start Stack file header file

Define struct Stack

//fields

int capacity //number of items the stack is holding

int itempointer //array of what the stack item is holding

int top //where our 'cursor' is

char name //identifier character

End struct Stack

//Remember our stack is a 'Last in First Out' structure

stack_create (Stack stck, char name) //create a new stack structure

push_stack (Stack stck, int item) //add an item element to the top of specified stack

pop_stack(Stack stck,) //remove an item element from the top of specified stack

is_empty(Stack stck,) //check is the specified stack has any items in it

peek_stack (Stack stck,) //what is the at the top of the specified stack? (item)

delete_stack (Stack stck,) //deletes the space we allocated in heap for the stack
// don't forget to delete the pointer for the items within
// the stack as well!

End definition Stack header file

Stack source file

Includes header file for stack

Start Define Stack Functions File

Stack stack_create (Stack stck, char name_input)

Define new stack on heap

//check if it was instantiated

If (!stck)

Return 0

End if

Set capacity field = input_discs

Set name field = name_input

Set items field as a heap location

//check if items field was created

If (!stck items)

Return 0

Endif

Return stck

End stack_create

Void push_stack(Stack stck, item)

if (!stck)
return
endif

if (stck top == stck capacity – 1)
double stck capacity
reallocate stck items into new stck capacity
endif

if (stck items)
set the stck items top element = item
increment the stck top to reflect new item added
endif
return
end push_stack

int pop_stack(Stack stck)

```
        if (!stck)
                return
        endif

        //if theres some item in stack to pop

        if (!is_empty)

                decrement stck top
                return the new top of stack
        endif

        otherwise return -1                 //signifies an empty stack
end pop_stack
```

Boolean is_empty(Stack stck)

```
        return if top of stck is 0          //if top  = 0 = stack is empty = true

end is_empty
```

int peek_stack (Stack stck)

```
        // if no stack
        (!stck)
                return -1
        endif
        otherwise
                return item at the  top of stck

end peek_stack
```

void delete_stack (Stack stck)

```
        // if no stack
        (!stck)
                return -1
        endif

        remove the pointer to stcks items
        remove the stck itself
end delete_stack
```

Tower source file

Includes header file for stack

//This implementation has been inspired by the visual representation provided by content maker "Reducible" on YouTube: "Towers of Hanoi: A complete visualization"
https://www.youtube.com/watch?v=rf6uf3jNjbo

// goal : move the user defined # discs from 'A' position into the 'B' position using recursion
// there will be a huge call stack here! Not the most efficient approach

```
void tower_recursion( int choice, char from_peg, char to_peg, char extra_peg)

    // If the current disk is the top disk move directly and then exit the current function call

    If ( n == 1 )

        Display the move taking place "Moving disk n from from_peg to to_peg"
        return

    Endif

    // If we are looking at the n-1 disk (second to bottom n disk)
    // Queues up our function calls onto stack based on the n-1 disk were looking at

    Else

        //The goal here is to move the stack of disks n-1 to the extra peg freeing up the  destination peg,
        allowing us to move that last n disk directly into the source

        Call tower_recursion(n-1, from_peg, extra-Peg, to_peg)

        //this displays the moves taking place between the moves

        Display "move disk n from from_peg to to_peg"

        //the goal here is to make the final move of the stack of disks into the destination peg.

        Call tower_recursion(n-1, from_peg, extra-Peg, to_peg)

        //note: alternating call patterns that occur

        //depending on if we start with even or odd this will determine the call sequence and
        thus sequence of the disks move. This will have more affect on the stack implementation

         //Ex:   if n = even moves the initial disk into the extra peg ( A -> C)

         //     if n = odd move into the goal peg ( A -> B )

    Endelse

End tower_recursion
```

//In the implementation I used my version of a power function in order to keep from linking math.h library in the makefile

```
Real Pow( base, power)


        Real result = 1
        iterate through [0,power)
                result = result * base
                iteration++

        End iteration
        return result
End Pow

//from here on out * represents a pass by reference

//Helper function to print moves for stack implementation
Display( Stack s1, Stack s2)

        //Display the disk exchange between s1 and s2
        "Move disk (value of disk being moved) from peg (name of s1) to peg (name of s2)"

End Display
```

//Helper function to compare the values between two pegs top disks (if any) for stack implementation

//lets assume the peg is empty, if we compare the values of disks between pegs ex:(peg1 < emptypeg),
//then we will not branch into the block because ex:(diskn < 0) will never be true
//empty variable assumes value of n+1 to compensate for this logic hazard
// c1 and c2 are the values of disks from the s1 and s2 pegs

```
Void Compare (Stack *s1, Stack *s2, *c1, *c2)

        Set Empty = disks+1
        If (is_empty(s1)
                set value c1 = empty

        or if (is_empty(s2)
                set value c2 = empty

        otherwise
                set value c1 = s_peek(s1)
                set value c2= s_peek(s2)

        end if

End Compare
```

//the stack implementation has been influenced by the visual representation from Pooya Taheri on YoutTube
"Recursive and Non-Recursive Hanoi Tower"
https://www.youtube.com/watch?v=ZWNK34T0YKM

```
Void Stack_tower(int disks)

    Declare int n = disks      //n is a temp variable to hold disks for setting our source peg values
    declare int comps = 0      //where s is source peg  // will be used to hold the values of the disks at top of stack
    declare int compg = 0      //where g is goal peg
    declare int compe = 0      //where e is extra peg

    Declare int num_moves = (Pow(2,disks)) -1      //the formula for finding the number of moves based on disks

    Instantiate Stack * source = stack_create disks, 'A')

    //fill the newly created stack for peg source

    Iterate from [0,disks)
        stack_push(source, n)
        decrement n
    end iteration

    Instantiate Stack * goal = stack_create disks, 'B')

    Instantiate Stack * extra = stack_create disks, 'C')

    //check if disks is odd
    //even v odd determines our sequence of steps

    //odd :  case 0:  1) source<->goal
    //       case 1:  2) source<->extra      following the game rules: smaller disk cannot be placed on a larger disk
    //       case 2:  3) goal<->extra


    if (disks & 1 )

        Iterate from [i=0, num_moves ) incrementing by 1
            declare int move = i mod 3      //this tells us which step were doing ex: 0 mod 3 = 0

            Case (move)
                case (0)                    // compare goal and source peg disks value for approp. move
                    compare (goal, source, compg, comps

                    If ( comps < compg )                    // if this is a valid move

                        Display (source, goal)
                        push(goal, stack_pop(source))   //remove disk from source add to goal

                    Otherwise                               //if not just reverse the operation

                        Display(goal, source)
                        push(source, stack_pop(source))
                    end if
                break and End case (0)
```

```
                    Case(1)

                        Compare (source, extra, comps, compe)

                        If ( comps < compe)

                            Display(source, extra)
                            Push (extra, stack_pop(source))

                        Otherwise
                            display(source, extra)
                            Push(source, stack_pop(extra)

                        End if
                    break and End case(1)

                    Case(2)

                        Compare(extra, goal, compe, compg)

                        If ( compe < compg)
                            Display(extra, goal)
                            Push (goal, stack_pop(extra))

                        Otherwise
                            Display(goal, extra)
                            Push (extra, stack_pop(goal))

                        Endif

                    Break and end case(2)

                End Case (Moves)

            //if disks is not odd then its even
            //even : case 0: 1) source <->extra
            //       case 1:  2) source <->goal        rules: smaller disk cannot be placed on a larger disk
            //       case 2:  3) extra  <->goal

            Otherwise

            //Perform the steps for even sequence following the same logic as the odd
            // since it is rather redundant I will omit the even disks implementation

            End If

        End Iteration

        //take care of the stacks that are floating in the heap (and the items within the stack)

        Stack_delete(source)
        Stack_delete(goal)
        Stack_delete(extra)


End stack_tower
```

```
//Finally the main function

Int Main (int argc, char **argv)

        Define OPTIONS “nsr :”          //should be a static global variable
        declare input_discs = 5         //this is our default value ideally this should be static global variable)
        declare int c = 0               //for optcode
        boolean recursion = false
        Boolean call_stack = false
        char *input  = NULL             //holds the optarg value

        While (  (c = getopt(argc, argv, OPTIONS) ) != -1 )

                Case based on choice

                //user can choose to play using recursive function call implementation with input_discs
                        Case ‘r’

                                Recursion = true
                                Break

                //user can choose to play using stack implementation with current input_discs value
                        Case ‘s’

                                Call_stack = true
                                break

                //user may define the amount of discs to play
                        Case ‘n x’
                                If ( (atoi (x) > 0) )           //should only allow positive integers
                                        input_discs = atoi(x)

                                Otherwise                       //ignore the input
                                        break

                End case
        End while

        If (argc == 1)
                Display error message
                Return -1
        End if

        If (recursion)                                  //if ‘r’ was caught and triggered to true
                call Void recursive(input_discs, ‘A’, ‘B’, ‘C’)
        End if

        If (call_stack)                                 if ‘s’ was caught and triggered to true
                stack_tower(dinput_disks)
        End if
        Return 0

End main
```

End Tower File