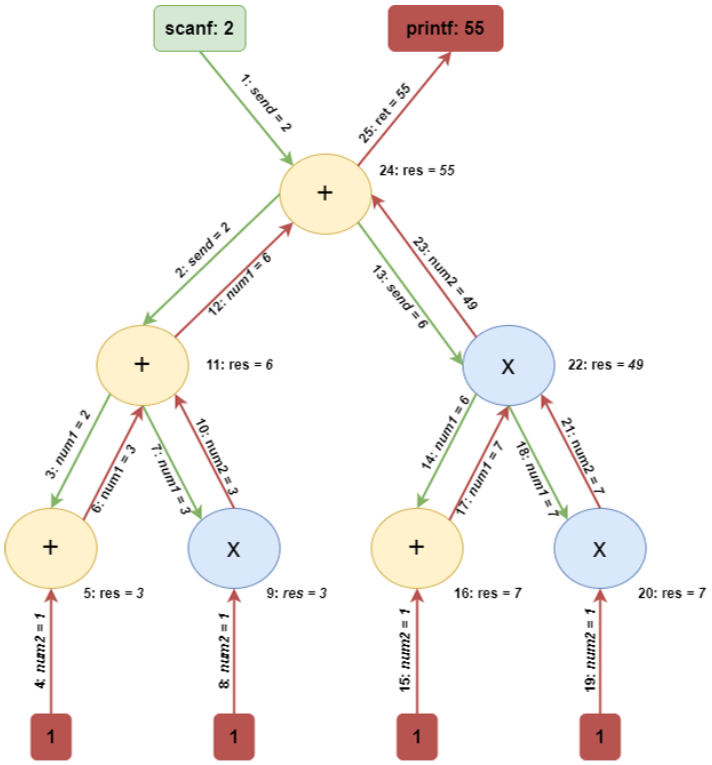
treepipe.c Man document

Goal

The goal for this project is to build a system that can get input from user specifying key attributes for a binary tree. Following these numbers the code is supposed to build a binary tree following the attributes, create child programs to simulate nodes of the tree and establish a pipe line(low hanging fruit but inevitable to miss this pun) such that it follows the given scheme.



Also every node will need to execvp the command corresponding to their position, namely left or right.

Utility functions

Let's get these out of the way, so that we can only focus on the meaty part that will follow. Firstly sum non-function utilities.

line9: const int MAX_MSG = 11;

line10: enum LeftRight { LEFT, RIGHT };

I globally introduced a MAX_MSG according to my prior knowledge that this is the byte restriction. And an enumeration for left and right that makes LEFT = 0 and RIGHT = 1. While there is no actual need for this enumeration it makes it easier to read and helped me not get confused on lr variable.

line11: outputLongExplanation()

line19: outputMyResult()

line27: outputNum1()

line35: outputCurDepht()

These functions only write according to specifications from the homework file.

line43: ConcatString(char* answer, int num1, int num2)

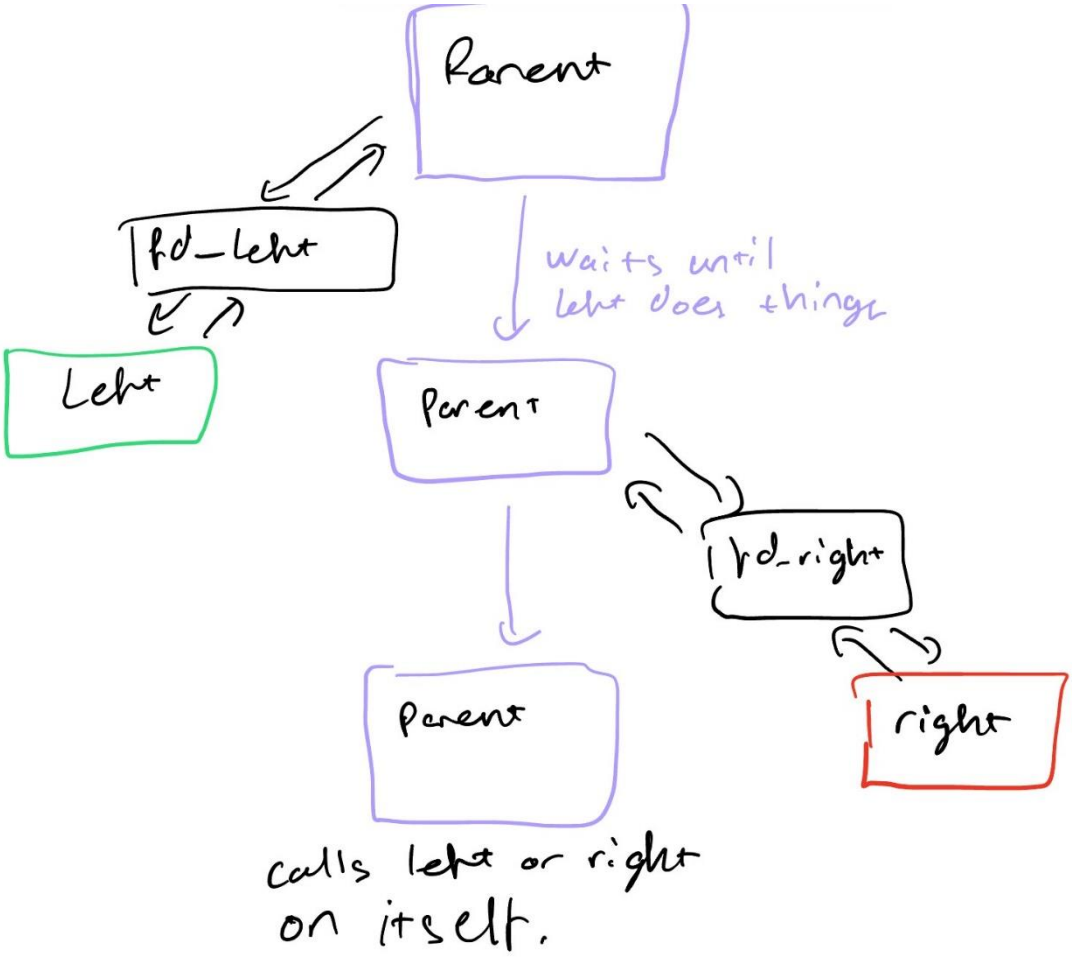
```
d ConcatString(char* answer, int num1, int num2)

char num1_str[MAX_MSG], num2_str[MAX_MSG];
sprintf(num1_str, "%d", num1);
sprintf(num2_str, "%d", num2);
int i = 0;

// Copy num1_str to answer
while(i < strlen(num1_str))
{
    answer[i] = num1_str[i];
    i++;
}

// Add a space between numbers
answer[i] = ' ';
i++;

// Copy num2_str to answer
int j = 0;
while (j < strlen(num2_str))
{
    answer[i + j] = num2_str[j];
    j++;
}

// Null-terminate the string
answer[i + j] = '\0';
```

This function gets a char pointer, and two numbers as parameters. The function writes num1 followed by space num2 followed by NULL. This is a recurring and highly error prone operation so this is the function implementation.

Process

My first intuition for this project was that if I could build a one depth tree implementation I could just make it recursively call itself to solve any depth solutions.

My second intuition was about the 1 depht solution. I knew I couldn't produce two childs at the same time, hence I would first call left child, let parent process get the result from left and then call right. And one that was done turn itself into left or right executable.



Main

Now I will move on to the real flow. Firstly main takes info from terminal. Checks the validity and then forks itself. The child calls for root function that deals with the tree and passes necessary arguments and a piped file descriptor for it to write on. Main then writes to the file descriptor of root and then reads from it.

Root Function

Root solves the problem recursively. After reading number from its file descriptor, root first checks for the base scenario where it is the leaf node. This is done by checking if current depth is equal to maximum depth. If it is, then this node is a leaf. It only runs itself according to lr passed as parameter. For running left and right leaves I have two functions.

```c
void leftLeafProcess(int fd[])
{
        // Redirect stdin to the read end of the pipe
        if (dup2(fd[0], STDIN_FILENO) == -1) {
            dprintf(STDERR_FILENO,"dup2 failed");
            exit(1);
        }
        if (dup2(fd[1], STDOUT_FILENO) == -1) {
            dprintf(STDERR_FILENO,"dup2 failed");
            exit(1);
        }
        // Execute the `./left` program
        char* command_arr[] = {"./left", NULL};

        // Attempt to run `./left`
        if (execvp(command_arr[0], command_arr) == -1) {
            dprintf(STDERR_FILENO,"execvp failed");
            exit(1);  // Ensure the child process exits on failure
        }
        // In case execvp fails, terminate the child
        exit(1);
}
```

The functions are super similar. All they do is dup2 the given file descriptor to standard input and output so that left or right program will reach for the correct file descriptor for my pipeline. After that the process is turned to left or right executable according to function.

```c
void rightLeafProcess(int fd_right[])
{
    //right child
            // Redirect stdin to the read end of the pipe
            if (dup2(fd_right[0], STDIN_FILENO) == -1) {
                dprintf(STDERR_FILENO,"dup2 failed");
                exit(1);
            }
            if (dup2(fd_right[1], STDOUT_FILENO) == -1) {
                dprintf(STDERR_FILENO,"dup2 failed");
                exit(1);
            }
            // Execute the `./left` program
            char* command_arr[] = {"./right", NULL};

            // Attempt to run `./left`
            if (execvp(command_arr[0], command_arr) == -1) {
                dprintf(STDERR_FILENO,"execvp failed");
                exit(1);  // Ensure the child process exits on failure
            }

            // In case execvp fails, terminate the child
            exit(1);
}
```

The leaf then reads the file descriptor and then writes to the file descriptor it got from its parameter.

If the process is not a leaf, meaning current depth is smaller than maximum depth, parent forks itself for a left child. The child calls root function but increments current depth by one. The original caller passes a piped file descriptor to this new recursion. After recursion does its job the original caller reads the given file descriptor. Then forks again to call a right child. The right child also calls the root function and increments the current depth by one. After that process finishes the original gets its second number from the right child. Then writes first and second numbers to its own file descriptor. After dup2'ing original file descriptor parameter to STDIN and STDOUT, the program continues and calls left or right on itself according to lr variable.

This recursive approach ensures three nice things. Firstly post-order traversal is ensured with this approach. Secondly this solution makes it so every depth imaginable can be run. Since I haven't hand coded every maximum depth. Third since BST's are mainly good for recursive approach I get to grasp its advantages and also boast myself as a programmer a bit.