Tree

Problem Description

We are going to write a seeking algorithm that takes in the file `data.bin`, which contains a binary tree which has been serialized, reads the array using pre-order notation. The flag can be printed out by printing the value when reading each node.

You have been provided `data.bin` which contains the serialized array. Each element takes the following form:

```
| Index | c | left  | right |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
```

The parts of the array entry are:

- *Index*: The index of the current array entry.
- *c*: The value at this index, a `char`.
- *left*: The index of the left child of this node.
- *right*: The index of the right child of this node.

The flag is generated by printing the value at each index when the tree is read using pre-order notation. Please remember that indices are meant for sorting and are not part of the value, and may not be in the correct location in the binary file.

The flag will be represented in ASCII.

Walkthrough

This challenge is a step above the `ArraySeek` challenge. This challenge builds a binary tree and then redads it using pre-order notation. We must read the binary, build the tree, and then read the tree using pre-order notation.

Defining the Struct

We are told each element contains four pieces: a two-byte index, a one-byte character, a two-byte index for the left child, and a two-byte index for the right. We can use this to define the following struct:

```
typedef struct ArrayItem {
    uint16_t index;
    uint8_t letter;
    uint16_t left;
    uint16_t right;
} node;
```

Since the left and the right are *indices* and not *pointers*, we use values instead of pointers. We can manage the overhead of connecting the parents to the children later.

Reading the Data

We will read the data like the rest of the challenges in this section. First, we open the file and ensure it properly opened. Then, we initialize the structure holding our nodes, then we read each note.

First, we open the file. Nothing new here.

```c
FILE* fp = fopen("data.bin", "rb");
if (fp == NULL) {
    printf("File not found.\n");
    exit(EXIT_FAILURE);
}
```

We can use a static array to hold the structure. This actually makes a lot of sense for us because we have the indices of the child nodes. Therefore, if we want to find a child, we can use a sorted copy of the array and access the child by index. This avoids having to link the children.

```c
node* nodes = malloc(100 * sizeof(node));
```

> *If you want to pre-determine the side of `nodes`, you can do so. See ArraySeek for an explanation of how this might be done.*

Now, let's read in the data. We can use an infinite while loop that breaks when we no longer read any more bytes. We only need to check when we read the index since we are told each element is complete. This section is almost the same as *ArraySeek*, so not much explanation is necessary.

```c
uint32_t num_elements = 0;
while (1) {
    node e;

    uint8_t bytes_read = fread(&e.index, sizeof(e.index), 1, fp);
    if (bytes_read == 0)
        break;

    fread(&e.letter, sizeof(e.letter), 1, fp);
    fread(&e.left, sizeof(e.left), 1, fp);
    fread(&e.right, sizeof(e.right), 1, fp);

    nodes[num_elements] = e;
    ++num_elements;
}
```

Once this is done, we need to sort the array so we can access roots and children by index. We build a `compare` function that sorts in descending order based on the index.

```
int compare(const void* a, const void* b)
{
    node* left = (node*)a;
    node* right = (node*)b;
    return left->index - right->index;
}
```

Then, we use `qsort()` to sort the nodes.

```
qsort(nodes, num_elements, sizeof(node), compare);
```

Data Processing

We need to run a pre-order read on the binary. The best way to do this is to allocate a string and then append the letters to the string as we read them. This way, we can print the string at the end as our flag.

A pre-order read is a recursive printing method for a binary tree that prints the current node, the left child, then the right. Our function needs the string storing our pre-order read, a `node*` to the list of nodes, the index of the node we're currently reading, and then a `uint*` to the number of nodes we've read thus far.

> *Why `uint*`?* We need the state of the number to be consistent across calls, so that backtracking works properly. Therefore, we pass the depth as a pointer so we can increment it across calls. As we backtrack, it automatically decrements.

Here is the signature of the method we will write:

```
void preOrder(char* string, node* nodes, uint16_t index, uint16_t* depth);
```

In this method, we need to do three things:

1. Append the current node's letter to the string.
2. Ensure we're not at a leaf node.
3. If applicable, recursively call the method on the left and right children **in that order**.

To add the current letter to the string, we can simply access by index since the string was allocated to the number of nodes. We then update the depth parameter.

```
// add current to string
string[(*depth)++] = nodes[index-1].letter;
```

Then, we check to see if we're at a leaf node. We can do this by checking to see if the left and right children are both zero. If they are, we can return.

```
// make sure we're not at the end
if (nodes[index-1].left == 0xffff && nodes[index-1].right == 0xffff)
    return;
```

Finally, we simply do the recursive travel. We do this by checking if the left child is not null, then call the `preOrder` method on the left child. Then, we do the same for the right child.

```
// perform the recursive traversal
if (nodes[index-1].left != 0xffff)
    preOrder(string, nodes, nodes[index-1].left, depth);
if (nodes[index-1].right != 0xffff)
    preOrder(string, nodes, nodes[index-1].right, depth);
```

This is all the processing we need. To start the algorithm, we allocate a string, initialize the depth to zero, and call the method on the root node.

```
char* string = malloc(num_elements * sizeof(char));
uint16_t* depth = malloc(sizeof(uint16_t));
*depth = 0;
preOrder(string, nodes, 0x1, depth);
```

Printing the Flag

Printing the flag is just printing the string:

```
printf("%s\n", string);
```

Running this gets the flag:

```
$ gcc -o solve solve.c && ./solve && rm solve
flag{Y0uR3@dTh3B!n@ryTr33C0rr3ctly}
```

Full Solution

Here is the full solution:

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <stdint.h>

typedef struct ArrayItem {
    uint16_t index;
    uint8_t letter;
    uint16_t left;
    uint16_t right;
} node;

int compare(const void* a, const void* b)
{
    node* left = (node*)a;
    node* right = (node*)b;
    return left->index - right->index;
}

void preOrder(char* string, node* nodes, uint16_t index, uint16_t* depth)
{
    // add current to string
    string[(*depth)++] = nodes[index-1].letter;

    // make sure we're not at the end
    if (nodes[index-1].left == 0xffff && nodes[index-1].right == 0xffff)
        return;

    // perform the recursive traversal
    if (nodes[index-1].left != 0xffff)
        preOrder(string, nodes, nodes[index-1].left, depth);
    if (nodes[index-1].right != 0xffff)
        preOrder(string, nodes, nodes[index-1].right, depth);
}

int main()
{
    FILE* fp = fopen("data.bin", "rb");
    if (fp == NULL) {
        printf("File not found.\n");
        exit(EXIT_FAILURE);
    }

    node* nodes = malloc(100 * sizeof(node));

    uint32_t num_elements = 0;
    while (1) {
        node e;

        uint8_t bytes_read = fread(&e.index, sizeof(e.index), 1, fp);
        if (bytes_read == 0)
            break;

        fread(&e.letter, sizeof(e.letter), 1, fp);
        fread(&e.left, sizeof(e.left), 1, fp);
        fread(&e.right, sizeof(e.right), 1, fp);
        nodes[num_elements] = e;
        ++num_elements;
    }

    qsort(nodes, num_elements, sizeof(node), compare);

    char* string = malloc(num_elements * sizeof(char));
    uint16_t* depth = malloc(sizeof(uint16_t));
    *depth = 0;
    preOrder(string, nodes, 0x1, depth);

    printf("%s\n", string);

    return 0;
}
```