LinkedOps

Problem Description

We are going to write a seeking algorithm that takes in the file `data.bin`, which contains a series of instructions that need to be performed on a Linked List. The flag can be printed out by reading the list after performing the operations.

You have been provided `data.bin` which contains the serialized array. Each element contains an operational code (OP) that tells what operation to perform, which in turn has the following form:

- `0`: Add a node to the end of the list. Following this sequence is a `char` value to add.

```
| OP | c |
| 0  | 1 |
```

- `1`: Add a node to the front of the list. Following this sequence is a `char` value to add.

```
| OP | c |
| 0  | 1 |
```

- `2`: Add a node to a specific index. Following the sequence is the index to add at, and then a `char` value to add.

```
| OP | index | c |
| 0  | 1 | 2 | 3 |
```

- `3`: Remove a node from the end of the list.

```
| OP |
| 0  |
```

- `4`: Remove a node from the front of the list.

```
| OP |
| 0  |
```

- `5`: Remove a node from a specific index. Following the sequence is the index to remove from.

```
| OP | index |
| 0  | 1 | 2 |
```

The flag is generated by performing the sequence of operations provided in the `data.bin` file and printing out the Linked List once the operations have been completed.

The flag will be represented in ASCII. Surround the data that you get in `flag{}` braces.

Walkthrough

This challenge is a bit different than the previous. Rather than the serialization representing a raw data structure, we are provided *raw operations to generate our own*.

For this challenge, we must define how we will build the linked list, then build the linked list, then print the list once it's fully initialized.

Defining the Struct

Despite the serialized data being operations on a linked list, we still must define a linked list. The linked list holds characters. Each item must contain a pointer to the next node. We can represent this as a struct containing these two values, like so:

```
typedef struct ArrayItem {
    uint8_t letter;
    struct ArrayItem* next;
} node;
```

From now on, we can simply refer to the `struct ArrayItem` as `node`.

Reading the Data

We will open the file as always and ensure the file has been opened.

```
FILE* fp = fopen("data.bin", "rb");
if (fp == NULL) {
    printf("Error opening file.\n");
    exit(EXIT_FAILURE);
}
```

After this, we must initialize the data structure. We define the head node as `NULL` to indicate the list is empty.

```
node* head = NULL;
```

Now, we can begin the `while` loop. It's not practical to count the number of elements before the loop because each element has a different size. It's easier to simply run an infinite loop that breaks when no more elements are read. We can check when reading the `opcode`.

> *If you're confused why we only need to check when we read in the `opcode`*, please consult the explanation in the ArraySeek challenge.

```
uint16_t num_elements = 0;
while (1)
{
    /** Read and process data **/
}
```

For each entry, we first need to find the `opcode`. If we don't read any bytes at this step, we can break the loop.

```
uint8_t opcode;
uint32_t bytes = fread(&opcode, sizeof(opcode), 1, fp);
if (bytes == 0)
    break;
```

Our next step depends on the `opcode`. You can either use a series of `if` statements or a `switch` to handle the `opcode`. I personally chose `if` statements because that's the first thing that comes to mind when I build solutions.

Let's handle each case:

- Case `0`: Add a node to the end of the list. We need to read one more byte (the character to insert) and then insert this node at the end of the list. We also must increment the number of elements.

  ```
  if (opcode == 0) {
      fread(&chr, sizeof(chr), 1, fp);
      insertNode(&head, num_elements, chr);
      ++num_elements;
  }
  ```

- Case `1`: Insert at the front of the list. We need to read one more byte, being the character to insert.

  ```
  if (opcode == 1) {
      fread(&chr, sizeof(chr), 1, fp);
      insertNode(&head, 0, chr);
      ++num_elements;
  }
  ```

- Case 2: Insert at an index. We need to read one byte for the index and one byte for the character. Then, we can insert.

```
if (opcode == 2) {
    fread(&index, sizeof(index), 1, fp);
    fread(&chr, sizeof(chr), 1, fp);
    insertNode(&head, index, chr);
    ++num_elements;
}
```

- Case 3: Remove a node from the end of the list. We don't need any more data. We remove the node and then decrement the number of elements.

```
if (opcode == 3) {
    removeNode(&head, num_elements-1);
    --num_elements;
}
```

- Case 4: Remove a node from the front of the list. We don't need any more data. We remove the node and then decrement the number of elements.

```
if (opcode == 4) {
    removeNode(&head, 0);
    --num_elements;
}
```

- Case 5: Remove a node from an index. We need to read one byte for the index. We remove the node and then decrement the number of elements.

```
if (opcode == 5) {
    fread(&index, sizeof(index), 1, fp);
    removeNode(&head, index);
    --num_elements;
}
```

- Default Case: We shouldn't ever reach here. If we read this, we should print it out and exit. This means we read something wrong.

```
else {
    printf("Bad read: %01x", opcode);
    exit(EXIT_FAILURE);
}
```

Data Processing

Let's handle the `insertNode` and `removeNode` methods. These methods are standard linked list operations. You could easily look online for implementations of linked lists, but I'll explain my own. Rather than making separate functions for inserting and removing at the start, end, and at an index, it's easier to each use the same method. It makes the data reading code a lot simpler.

Here are the declarations for both methods:

```
void insertNode(node** head, uint16_t loc, char letter);
void removeNode(node** head, uint16_t loc);
```

Inserting a Node

To insert a node, we must make a new node for it. I'll dynamically allocate it to ensure it doesn't lose scope and get deleted.

```
// make a new node
node* new = malloc(sizeof(node));
new->letter = letter;
new->next = NULL;
```

We must handle the edge cases. In the case that we're inserting to an empty list (meaning `head == NULL`), we can simply set the head to the new node.

```
// handle empty list
if (*head == NULL) {
    *head = new;
    return;
}
```

We can actually combine this with the other edge case, if we insert to the front. Both these have the same methodology.

```
// edge case: empty list or inserting beginning
if (*head == NULL || loc == 0) {
    new->next = *head;
    *head = new;
    return;
}
```

Now, we can handle the main case: inserting in the center or end. We first must iterate to the location we want to insert. We want to stop at the node before the location we want to insert. At the same time, we

should ensure that we aren't inserting beyond the length of the list.

```
// main case: inserting center
node* current = *head;
for (uint16_t index = 0; index < loc - 1 && current != NULL; ++index)
    current = current->next;

if (current == NULL) { // error: beyond edge length
    free(new);
    return;
}
```

Finally, once we have the location, we link in `new` after `next` .

```
// link the nodes
new->next = current->next;
current->next = new;
```

Removing a Node

Removing a node is a similar process. We first handle the two edge cases: (1) when the list is already empty, or (2) when we're removing from the start.

- If the list is already empty, we do nothing.
- If we're removing from the start, we simply set the head to the next node and free the old head.

```
if (*head == NULL) return; // list is empty

if (loc == 0) { // remove from start
    node* toDelete = *head;
    *head = (*head)->next;
    free(toDelete);
    return;
}
```

Then, we cover the main case: removing from the center. We navigate to the node before the one we want to remove using the same logic as inserting a node.

```
// main case: remove from center
node* current = *head;
for (uint16_t index = 0; index < loc - 1 && current != NULL; ++index)
    current = current->next;

if (current == NULL || current->next == NULL) {
    // error: beyond edge length
```

```
    return;
}
```

Once we're here, we can remove the node by linking the surrounding nodes.

```
node* toDelete = current->next;
current->next = toDelete->next;
free(toDelete);
```

This is it! Now we can print the flag.

Printing the Flag

We can make a function that prints a linked list for us. We'll encapsulate the flag braces at the same time.

```
void printNode(node** head)
{
    printf("flag{");
    for (node* c = *head; c != NULL; c=c->next)
        printf("%c", c->letter);
    printf("}\n");
}
```

Running this prints the flag:

```
$ gcc -o solve solve.c && ./solve && rm solve
flag{TjWc9AY)')G\A.Bf=N3pHP9Ej=3@M|ly4M;8KJOJgIT/ch8e}
```

Full Solution

Here is the full solution:

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

typedef struct ArrayItem {
    uint8_t letter;
    struct ArrayItem* next;
} node;

void insertNode(node** head, uint16_t loc, char letter) {
    // make a new node
    node* new = malloc(sizeof(node));
    new->letter = letter;
```

```
    new->next = NULL;

    // edge case: empty list or inserting beginning
    if (*head == NULL || loc == 0) {
        new->next = *head;
        *head = new;
        return;
    }

    // main case: inserting center
    node* current = *head;
    for (uint16_t index = 0; index < loc - 1 && current != NULL; ++index)
        current = current->next;

    if (current == NULL) { // error: beyond edge length
        free(new);
        return;
    }

    // link the nodes
    new->next = current->next;
    current->next = new;
}

void removeNode(node** head, uint16_t loc) {
    if (*head == NULL) return; // list is empty

    if (loc == 0) { // remove from start
        node* toDelete = *head;
        *head = (*head)->next;
        free(toDelete);
        return;
    }

    // main case: remove from center
    node* current = *head;
    for (uint16_t index = 0; index < loc - 1 && current != NULL; ++index)
        current = current->next;

    if (current == NULL || current->next == NULL) {
        // error: beyond edge length
        return;
    }

    node* toDelete = current->next;
    current->next = toDelete->next;
    free(toDelete);
}


void printNode(node** head)
{
    printf("flag{");
    for (node* c = *head; c != NULL; c=c->next)
```

```
            printf("%c", c->letter);
        printf("}\n");
    }

    int main()
    {
        FILE* fp = fopen("data.bin", "rb");
        if (fp == NULL) {
            printf("Error opening file.\n");
            exit(EXIT_FAILURE);
        }

        node* head = NULL;

        uint16_t num_elements = 0;
        while (1) {
            uint8_t opcode = 0;
            uint16_t bytes_read = fread(&opcode, sizeof(opcode), 1, fp);
            if (bytes_read == 0)
                break;

            uint8_t chr;
            uint16_t index;
            if (opcode == 0) { // insert end
                fread(&chr, sizeof(chr), 1, fp);
                insertNode(&head, num_elements, chr);
                ++num_elements;
            } else if (opcode == 1) { // insert front
                fread(&chr, sizeof(chr), 1, fp);
                insertNode(&head, 0, chr);
                ++num_elements;
            } else if (opcode == 2) { // insert index
                fread(&index, sizeof(index), 1, fp);
                fread(&chr, sizeof(chr), 1, fp);
                insertNode(&head, index, chr);
                ++num_elements;
            } else if (opcode == 3) { // remove end
                removeNode(&head, num_elements - 1);
                --num_elements;
            } else if (opcode == 4) { // remove front
                removeNode(&head, 0);
                --num_elements;
            } else if (opcode == 5) { // remove index
                fread(&index, sizeof(index), 1, fp);
                removeNode(&head, index);
                --num_elements;
            } else {
                printf("Bad read: %01x", opcode);
                exit(EXIT_FAILURE);
            }
        }

        printNode(&head);
```

```
    return 0;
}
```