Linked lists

Introduction

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you've skipped the pointers tutorial, you should go back and redo it. You must also be familiar with dynamic memory allocation and structures.

Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Linked lists have a few advantages over arrays:

1. Items can be added or removed from the middle of the list
2. There is no need to define an initial size

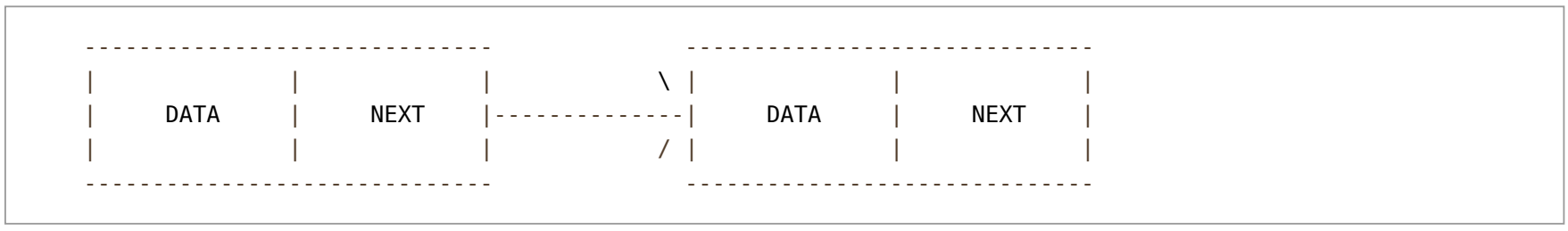
However, linked lists also have a few disadvantages:

1. There is no "random" access – it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
2. Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
3. Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

What is a linked list?

A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.



Let's define a linked list node:

```
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

Notice that we are defining the struct in a recursive manner, which is possible in C. Let's name our node type `node_t`.

Now we can use the nodes. Let's create a local variable which points to the first item of the list (called `head`).

Code

Run Reset Solution

Output

Expected Output

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
if (head == NULL) {
    return 1;
}

head->val = 1;
head->next = NULL;
```

We've just created the first variable in the list. We must set the value, and the next item to be empty, if we want to finish populating the list. Notice that we should always check if malloc returned a NULL value or not.

To add a variable to the end of the list, we can just continue advancing to the next pointer:

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
head->val = 1;
head->next = (node_t *) malloc(sizeof(node_t));
head->next->val = 2;
head->next->next = NULL;
```

This can go on and on, but what we should actually do is advance to the last item of the list, until the `next` variable will be `NULL`.

Iterating over a list

Let's build a function that prints out all the items of a list. To do this, we need to use a `current` pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the `current` pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

```
void print_list(node_t * head) {
    node_t * current = head;

    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

Adding an item to the end of the list

To iterate over all the members of the linked list, we use a pointer called `current`. We set it to start from the head and then in each step, we advance the pointer to the next item in the list, until we reach the last item.

```
void push(node_t * head, int val) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    /* now we can add a new variable */
    current->next = (node_t *) malloc(sizeof(node_t));
    current->next->val = val;
    current->next->next = NULL;
}
```

The best use cases for linked lists are stacks and queues, which we will now implement:

Adding an item to the beginning of the list (pushing to the list)

To add to the beginning of the list, we will need to do the following:

1. Create a new item and set its value
2. Link the new item to point to the head of the list
3. Set the head of the list to be our new item

This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it.

Since we use a function to do this operation, we want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself.

Code

Run Reset Solution

Output

Expected Output

```
void push(node_t ** head, int val) {
    node_t * new_node;
    new_node = (node_t *) malloc(sizeof(node_t));

    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
}
```

Removing the first item (popping from the list)

To pop a variable, we will need to reverse this action:

1. Take the next item that the head points to and save it
2. Free the head item
3. Set the head to be the next item that we've stored on the side

Here is the code:

```
int pop(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;

    return retval;
}
```

Removing the last item of the list

Removing the last item from a list is very similar to adding it to the end of the list, but with one big exception - since we have to change one item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

```
int remove_last(node_t * head) {
    int retval = 0;
    /* if there is only one item in the list, remove it */
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }

    /* get to the second to last node in the list */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    /* now current points to the second to last item of the list, so let's remove current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;

}
```

Removing a specific item

To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.

Here is the algorithm:

Code

Run Reset Solution

Output

Expected Output

There are a few edge cases we need to take care of, so make sure you understand the code.

```
int remove_by_index(node_t ** head, int n) {
    int i = 0;
    int retval = -1;
    node_t * current = *head;
    node_t * temp_node = NULL;

    if (n == 0) {
        return pop(head);
    }

    for (i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next;
    }

    if (current->next == NULL) {
        return -1;
    }

    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);

    return retval;

}
```

Exercise

You must implement the function `remove_by_value` which receives a double pointer to the head and removes the first item in the list which has the value `val`.

Start Exercise

« Previous Tutorial Next Tutorial »

Code Run Reset Solution

Output Expected Output