L12 - Linked Lists

November 13, 2019

0.1 Linked Lists

0.1.1 Limits of Arrays

The size of an array is determined at compile time - `arr[100]` will always give you an array of size 100. We can either allocate it too big (and waste memory) - or we can make it small (and risk going off the end).

The solution is to use *dynamic memory allocation* (that is, the heap) for elements as needed.

In a linked list, each *node* stores: - the value of the list at that node; - a pointer to the next value in the list.

The *head* of the list stores a pointer to the first node. The *tail* stores a special link/pointer that indicates reaching the end of the list.

Pros

- allocate nodes only as needed (can grow)
- can deallocate nodes when not needed (can shrink)
- efficient memory use

Cons

- more complicated than arrays
- no operator for indexing the list
  - must follow the entire list through to find each value of the list
  - time to access depends on its position in the list

0.1.2 Structure of a list

```
typedef struct node {
    int data;              // list payload
    struct node *next;     // pointer to the next node
} node_t;
```

We cannot declare `next` as `node_t next` - because type `node_t` is not done being defined yet!

```
In [19]: #include <stdio.h>
         #include <stdlib.h>
         #include <assert.h>
```

```c
// node structure
struct node {
    int value;             // list payload
    struct node* next;     // pointer to the next node
};
typedef struct node node_t;

// node constructor function
node_t* node_construct( int value, node_t* next )
{
    node_t* p = malloc(sizeof(node_t));  // A: alocate memory
    assert ( p != NULL );
    p->value = value;                    // B: build structure (initialize values)
    p->next = next;                      // C: connect to next node
    return p;
}

int main()
{
    node_t* first;
    node_t* second;
    node_t* third;

    // initialize nodes
    third = node_construct(3, NULL);
    second = node_construct(2, third);
    first = node_construct(1, second);

    // get a pointer to the front of the list
    node_t* current = first;

    // iterate over the linked list
    while(1)
    {
        printf("%d\n", current->value);

        if (current->next == NULL)
        {
            break; // stop iterating down the list after reaching the end
        }
        else
        {
            current = current->next; // walk the pointer down the list
        }
    }

    return EXIT_SUCCESS;
}
```

0.1.3 Designing Linked List Operations

When designing an operation on a linked list, ensure the algorithm works for:

- an empty list
- operations at (or ahead of) the first node
- operations at (or behind) the last node
- operations somewhere in the middle

To make a linked list ... Just follow the ABCs:

- *Allocate*: allocate memory on the heap to store the node structure
- *Build*: fill in the struct with values
- *Connect*: set up the pointers to connect the node into the list

Generally, we want to *make* new pointers before we *break* old pointers, to avoid accidentally losing nodes because they aren't linked anymore. This is called **make before break**.

0.1.4 `push` - insert a node at the front of a list

- list is either empty or not empty
- `head` pointer will change
  - from old first node to new node

```
node_t* push(node_t* head, int value)
{
    node_t* newnode = malloc(sizeof(node_t));
    assert(newnode != NULL);
    newnode -> data = some_value_to_be_stored;

    // set up the link
    newnode->next = head;

    // update head pointer
    head = newnode;

    return head;
}
```

0.1.5 pop - remove a node from the front of a list

Pop takes the head pointer of a list, and: * stores the data in a variable popped;

- removes the head node from the list;
- return a pointer to the list's *new* head node

In [7]:
```
// from the C-Tutor link on CULearn
#include <stdlib.h>
#include <stdio.h>

typedef struct node {
  int data;
  struct node *next;
} node_t;

/* Insert a new node containing data at the front of the
 * linked list pointed to by head.
 * Return a pointer to the new head node.
 */
node_t *push(node_t *head, int data)
{
  node_t *newnode = malloc(sizeof(node_t));
  // assert(newnode != NULL);
  newnode->data = data; // make the connection
  newnode->next = head; // break/remake connections
  return newnode;
}

/* Remove the node at the front of the the linked list
 * pointed to by head.
 * Store the data from that node in the variable pointed
 * to by popped.
 * Return a pointer to the head of the modified list.
 * Terminate (via assert) if the list is empty.
 */
node_t *pop(node_t *head, int *popped)
{
  // assert(head != NULL);
  *popped = head->data;
  node_t *node_to_free = head; // without this we'll free the *new* head
  head = head->next; // move head down the list by one
  free(node_to_free);
  return head;
}

// Make sure you understand why this code is wrong.
node_t *wrongpop(node_t *head, int *popped)
```

```
{
  // assert(head != NULL);
  *popped = head->data;
  free(head);
  head = head->next;
  return head;
}

int main()
{
  node_t *my_list = NULL;  // empty list
  my_list = push(my_list, 3);
  my_list = push(my_list, 2);
  my_list = push(my_list, 1);

  int val;
  my_list = pop(my_list, &val);
  printf("%d\n", val);
  my_list = pop(my_list, &val);
  printf("%d\n", val);
  my_list = pop(my_list, &val);
  printf("%d\n", val);

  return EXIT_SUCCESS;
}
```

```
1
2
3
```

Since `push` adds new entries to the beginning of the list, we need to break and remake the head pointer each time.

When `pop`ing the nodes back off the list, we need to store the node we wish to delete/free in a temporary pointer so that we still know where it is after changing the head pointer. This is the error in `wrongpop()` - it will free head and deallocate the structure, which means it *cannot* then follow the `node.next` pointer.

0.2 Traversing A Linked List

What do these snippets do?

0.2.1 Snippet 1

```
current = head;

while ( current != NULL )
{
```

```
    do_something();
    current = current -> next;
}
```

0.2.2 Snippet 2

```
current = head;

while ( current->next != NULL)
{
    do_something();
    current = current->next;
}
```

Snippet 1 will fetch the values from each node. Snippet 2 will *not* retrieve the value from the last node in the list - note that `current->next` will be null when the *second last node* in the list is reached. Snippet 1 could be used to sum all the values in a list while Snippet 2 could be used to count values in a list.

0.2.3 Revisiting `push()`

Can you use `push()` to add a new node that *isn't* at the beginning of the list? Sure - just pass the function a `*head` pointer that is somewhere down the list.

0.3 Swap Two Variables in a Linked List

When swapping two variables, recall we need a third variable `temp` to store one value in so no variables are lost in the swap.

Assume we have two pointers, `front`, `rear` and `curr`. To swap two nodes in a linked list: - Add a node at the end of the queue:

```
p = malloc(sizeof(node_t));
p->data = 1;

- What connections need to be changed?
```
front = p;
p->next = front;
```
```

- Insert node 25 after the current node?

  ```
  p = malloc(...);
  p->next = curr->next;
  curr->next = p;
  ```