Course
ELEC-A7100
Course materials
Your points
Microsoft Teams
Code Vault

This course has already ended.

« 2 Abstract data types Course materials 4 Round feedback »
ELEC-A7100 / 6. Structured Data Types / 3 Linked list

Sixth round tasks as a `ZIP file.`

Structured data types (*struct*) can be used to combine variables related to each other and aggregate it into a single data type, which after structure definition can be used much like other types of data. It is common to use structured data types in simple to complex C programs.

In this section, you are going to learn how to use structured data types through pointers and array of structures. The section ends with **linked list**, which is a commonly used dynamic data structure in C programs. Implementation of a **linked list** is also a good exercise to learn structured data types.

Linked list¶

Linked list is a common data structure based on dynamically allocated nodes linked with pointers between the nodes. It is arranged in such a way that each node contains values 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.

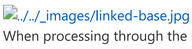
A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.

There are different variants of linked lists,but here we discuss a basic list with a pointer to the beginning of the list, and each member of the list containing a pointer to the next member of the list.

Let's assume a linked list that stores integer values. Each list member is defined by the following structure:

```
struct intlist {
    int value;
    struct intlist *next;
};
```

Linked list consisting of the above kind of nodes looks like below:



When processing through the list, an implementation has to have a pointer to the first member of the list, and then walk through the list by following the next pointers.

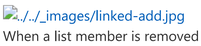
For example, the following code prints all values in the linked list:

```
struct intlist *first;

// Link to the beginning of the list in some way ...

struct intlist *current = first;
while (current) {  // Loop until a NULL pointer is reached
        printf("value: %d\n", current->value);
        current = current->next;
}
```

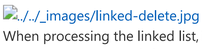
When adding a new member to the end of the list, the following steps need to be taken:

1. Allocate memory for a new list node: `struct intlist *new = malloc(sizeof(struct intlist);`. Initialize the members of the allocated structure. Because the new member will be the last element of the list, the `next` pointer will be NULL.
2. Find the last member of the list. The last member is the one that has `current->next == NULL`.
3. Modify the next link of the last member to point to the newly allocated member. (`current->next = new` or something like that). Now the new member is at the end of the list.

The same illustrated as diagram:



When a list member is removed from the list, the next pointer of the previous member needs to be modified to bypass the member to be removed, and point to the following list member. Naturally, the memory allocated for the removed member needs to be released.



When processing the linked list, it is good to pay attention to the special cases at the first member of the list (or when adding an element to empty list) and the last member of the list. Sometimes they may need special attention.

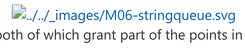
It is now good to compare arrays and linked lists.

- Linked lists have several advantages over arrays. Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may become wastefully empty or need to be made smaller.
- On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays is also faster than on linked lists on many machines due to locality of reference and data caches. Linked lists receive almost no benefit from the cache.
- Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Task 6.5: Exercise queue¶

In this task you will implement a queueing system using a linked list. The queue holds names (i.e., strings) of the members in the queue. The length of the strings is not limited. The items in linked list are stored in following data structures:

As mentioned above, the names of queue members can be arbitrarily long, so the name should be stored in dynamically allocated memory. To ease the handling of the list, the the first element is always empty, and should be passed. In the first element, the *name* field should always be NULL pointer. In the last element of the list, the *next* pointer should be NULL. When the list is empty, both fields are NULL in the first element. Below is a picture the illustrates a queue with two members.

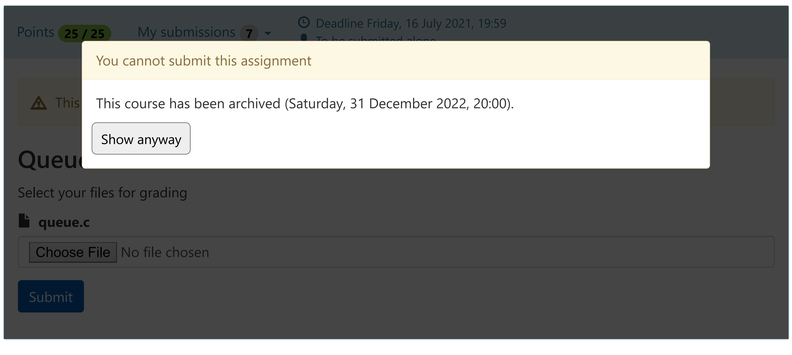


You will need to implement two functions, both of which grant part of the points in this task:

(a): Function `int enqueue(struct studentqueue *q, const char *name)` that adds given *name* as the last item of the list (that has NULL name in the beginning). The function makes two memory allocations: one for the data structure, and other for the name. The function returns value 1, if addition was successful, and value 0 if it failed (for example because of failed memory allocation).

(b): Function `int dequeue(struct studentqueue *q, char *buffer, unsigned int size)` that removes the first item from the queue. Before removing the item, the name it contains should be copied to address *buffer*. In this address there is space for *size* characters, so you can copy at most *size-1* characters. You should also release the memory allocated for this item. The function returns value 1, if there was an item removed from the list, and the related name is copied to *buffer*. The function returns 0, if there was nothing removed from the list, for example because the list was already empty.

In the implementation and testing, you should pay attention to the special cases, such as the case of an empty list.



« 2 Abstract data types Course materials 4 Round feedback »