

Linked Lists

Linked list store lists of data elements linked together with pointers.

Linked Lists are important because of their simplicity. They're usually slower to access than arrays (because of cache issues), but are easier to shrink and grow as data is added or removed from them. They are one of the easiest data structures to code up (especially in a garbage-collected programming language). They're recursive structure makes them a natural fit for being built and operated on by recursive functions, and they are a primitive construct in many functional programming languages such as Lisp and ML.

They are really easy to implement in Java, but there is a generic version called `java.util.LinkedList`. In C++, the standard template library implementation is called `std::list`.

What is a Linked List?

In its simplest form, a linked list is a connection of nodes, each of which contains some data and a pointer to the next node in a chain.

Since all of the other nodes can be reached by following pointers from the first one, a pointer to the first node can be taken to represent the entire list. The last node of course does not need a pointer to a "next node", so some special value (usually `NULL`) in its place.

The code for such a linked list structure is quite simple. If the data being stored is single character, the following declaration is sufficient:

```
struct node {
    char data;
    struct node *next;
}
```

Each node is a (dynamically allocated) instance of this data structure.

The Recursive Structure

Note that there is a reference to the node structure within the definition of the node structure. This is what makes the data structure recursive. It is not a problem, because it is not a whole node structure but rather just a pointer to a node structure that is contained in each node structure.

If you look back at our example linked list, the `next` field of the first node (which contains 'A') points to the second node (which contains 'B'). And, as we said that we can represent the whole list ('A', 'B', 'C') by a pointer to the first node, this pointer to the second node can be considered to represent the list ('B', 'C'). And the `next` field of the 'B' node, which points to the 'C' node can be considered to represent the list 'C'. For consistency we can also consider the `NULL` value for the `next` field of the 'C' node to represent an empty list ().

In this way, every list can be considered to be either `NULL`, or `Node` containing the data for the *first* element and a pointer to a sublist containing the *rest* of the list. ML and Lisp lists are constructed and deconstructed in exactly this manner, and the operations provided are:

- construct a new list by adding a new element to the front of an old list:

```
struct node *cons(char first, struct node *rest)
{
    struct node *linked_list;
    linked_list = malloc(sizeof(struct node));
    linked_list->data = first;
    linked_list->next = rest;
}
```

- access the first data element of a list:

```
char first(struct node *linked_list)
{
    return linked_list->data;
}
```

- access the rest of the list

```
struct node *rest(struct node linked_list)
{
    return old_list->next;
}
```

Iterating over the elements of the list can be done pretty easily with something like:

```
struct node *node_ptr = linked_list;
while ((node_ptr = node_ptr->next) != NULL) {
    sum += node_ptr->data
}
```

You can also get the same effect with a recursive function:

```
int sum(struct node *linked_list) {
    return linked_list->data + sum(linked_list->next);
}
```

This is a minimal linked list. You can add a `next` field to any structure, and initialize it to point to another instance of the same structure, and so on, eventually terminating in a `NULL` pointer and you have a linked list. The basic operations are so simple that they're often embedded directly in the code (or provided as language primitives in Lisp and ML).

Comparison to Dynamic Arrays

Dynamic arrays support random access (e.g., what is the n th element?) in constant time, whereas linked lists require iteration over n nodes to get to the n th. In addition, on modern computers, dynamic arrays are much more cache friendly, so iteration will be much faster over dynamic arrays.

On the other hand, linked lists are more flexible, one can generally insert or delete an element in the middle or end of a linked list (assuming one has a pointer/iterator to it) in constant time. In addition, linked lists are generally easier to implement especially in language that have garbage collection but not `realloc` (e.g., Java).

Variations and Construction of Wrapped Linked List

For a minimal linked list, you can add a *next* field to any structure (making the structure a node) and initialize the next pointer to point to another instance of the same structure, and so on, eventually terminating in a NULL pointer and coupled with a first (head) pointer you have a linked list.

Some common variations on the linked-list are:

1. to include a pointer to the *last* (a.k.a. tail) node of the linked list (in addition to a pointer to the *first*, a.k.a. head),
2. to include in each node a pointer to the *previous* node (in addition to the *next* node), and/or
3. the use of 'sentinel' nodes before the first and last list

A linked list with both *previous* and *next* nodes is known as a "doubly-linked list".

In some cases, it is sufficient to use a pointer to the first node to represent the entire list, but this can be awkward if one list is to be accessed/modified in more than one location, especially if you are going to be adding nodes to the front of the list or back of the list at various points in the program. In the former case, the pointer to the head of the list changes and changes with each insertion and it is necessary to maintain consistency among all of the users of the list. In the latter case, in order to avoid traversing the whole list, it is desirable to maintain a 'last' pointer pointing to the last node of the list. This tail pointer needs to be available at any program point might remove from or add to the end of the list. For simple cases, just having a 'first' variable and a 'last' variable may be sufficient, but if you need to access the list from lots of different functions, copying around both variables can get old.

An alternative is to 'wrap' the linked-list in some kind of structure and of the list. This allows one to use a single unchanging pointer to represent a particular linked list as it evolves. A pointer to that outer linked list structure can then be used to represent the list, and operations can modify the outer structure and the internal nodes it points to without invalidating that pointer to the outer structure. An additional step towards modularity can be made by putting the operations into a separate functions. For a simple linked-list that is only constructed in a single location at a single time and then accessed sequentially, this may be overkill compared to just embedding the pointer operations directly, but for the more complex variants of linked-lists or for more complex usage patterns it can help make the program more modular.

The C structures for such a linked list containing an integer in each node might be declared as:

```
struct node {
    int data;
    struct node *prev;
    struct node *next;
};

struct linked_list {
    struct node *first;
    struct node *last;
};
```

Operations

The following outlines the steps to inserting and removing nodes from a doubly-linked list.

inserting a node

1. malloc space for a struct node
2. initialize the data to be stored in the node
3. link the node's next and prev links to its neighbors (NULL if it is at the beginning or end)
4. link the preceding neighbor's next link to the node (use the list's first link instead, if the node is being inserted at the beginning of the list)
5. link the succeeding neighbor's prev link to the node (use the list's last link instead, if the node is being inserted at the end of the list)

removing a node

1. connect the preceding neighbor's next link to the succeeding neighbor (use the list's first link instead, if the node is being removed from the beginning of the list)
2. connect the succeeding neighbor's prev link to the preceding neighbor (use the list's last link instead, if the node is being removed from the end of the list)
3. deallocate the node

In a singly linked list without last links, it is, of course, unnecessary to update the previous/last links. It is still, however, necessary to update the preceding neighbor's next link. Unfortunately, in a singly-linked list, there is no way, given just a pointer to a node to get access to the preceding neighbor. So although a singly-linked list is itself simpler, the code for looping over its nodes and acting upon them is often more complex, since it may need to be able to access both the "preceding" and "current" node.

In general, linked lists operations often need special cases if the node being manipulated is the first or last in the list, or if the list is or is becoming empty. So when writing linked-list operations pay special attention (e.g., think through and imagine or draw diagrams) to what should happen in each of those cases. Sometimes special cases in the code can be eliminated by either using a pointer to the previous node's next pointer in place of a pointer to a node, or by adding extra dummy 'sentinel' nodes at the beginning and end of the linked list.

An Example Removal of Matching Nodes from Singly- and Doubly-Linked Lists

For example, if one wants to remove all node whose data equals `x`, for a doubly-linked list (pointed to by the variable `ll`), the code might look like:

```
struct node *p = ll->first;

// iterate over the nodes
while (p != NULL) {
    // if the node matches x, delete the node
    if (p->data == x) {
        struct node *n = p;
        p = p->next;

        // connect predecessor to its new successor
        if (n->prev != NULL)
            n->prev->next = n->next;
        else
```

```

        ll->first = n->next;

        // connect successor to its new predecessor
        if (n->next != NULL)
            n->next->prev = n->prev;
        else
            ll->last = n->prev;

        // delete the node
        free(n);
    } else {
        p = p->next;
    }
}

```

While for a singly-linked list, it might look like:

```

struct node *p, *n;

while (ll->first != NULL && ll->first->data == x) {
    n = ll->first;
    ll->first = n->next;
    free(n);
}

p = ll->first;
// iterate over the nodes p->next, and find those which match x
while (p != NULL && p->next != NULL) {
    struct node *n = p->next;

    if(n->data == x) {
        // connect the predecessor to its new successor
        p->next = n->next;

        // update the last link, if necessary
        if (n->next == NULL)
            ll->last = p;

        // deallocate the memory for the node
        free(n);
    } else {
        p = p->next;
    }
}

```