Edinburgh Napier University

SET08122 Algorithms & Data Structures

Lab 4 - Data Structures #2

Dr Simon Wells

1 Aims

Our goal this week is to investigate one of the most useful data structures, the Linked List.

At the end of the practical portion of this topic you will be able to:

- Implement a Linked List data structure
- Design an API for interacting with a Linked List
- Implement a Doubly Linked List data structure
- Design an API for interacting with a Doubly Linked List
- Implement a stack using a list data structure instead of an Array
- Implement a queue using a list data structure instead of an Array

2 Structured Activities

Data structures that we can use in a language are concrete implementations of abstract concepts. Often they are constructed using simpler, primitive data-types such as ints, floats, chars, and strings.

2.1 Linked Lists

Whilst until now we have used arrays because they are basic and immediately useful, they have very obvious limitations. For example, because Arrays require the allocation of contiguous sections of memory they will have issues when memory is limited or fragmented. If there isn't a contiguous section of memory to allocate, perhaps because of lots of other memory allocations, then the new allocation will fail.

However, Arrays are structured very closely to the way that computer memory, e.g. RAM, is organised in both physical and logical senses. Whilst more complex implementations of arrays could be implemented, as is found in many higher level languages such as Java with Vectors and ArrayLists, and Python with Lists, the C implementation is straightforward and pragmatic. If an Array is insufficient, then there are other data structures that can be used which, at the expense of an increase in complexity, lead to perhaps more flexible data structures.

The linked list is a data structure that is fundamental and basic and gives us another method for building a collection of data. Note that until now, whilst we have built some different data structures, e.g. the Stack, Queue and Deque, the underlying collection has been the Array. The Linked List provides an alternative to the Array for collecting data, consituting both a basic collection type as well as a type of data structure.

Just like the Array, the linked list is inspired by the way that memory is organised. However, instead of requiring all of the memory for a collection of data to be allocated together, the linked list instead stores just one element of the collection along with the location of the next element in the collection. This means that the individual elements of the collection can be spread throughout the available memory rather than concentrated in one place. This feels like it should be a more efficient use of memory because we can keep adding elements to the Linked List until we run out of space. There is no need to increase the size of arrays when our collection exceeds the size of it's container.

2.1.1 Implementing the Linked List

Let's create our own linked list. We are going to create an API for doing the following with a linked list:

1. prepend
2. append

3. count
4. delete
5. display
6. insert after

This should be enough to give you the idea of how a linked list works in practise. It will give us some utility functions for counting the number of elements in the linked list and displaying the contents of the list. It also give us API functions for appending (adding to the end), prepending (adding to the start), inserting (after a given location), and deleting elements from the linked list.

```c
struct node
{
    int data;
    struct node * link;
};
```

As before, for each of these functions you'll need to add your own function prototypes to your source file. Note that a complete listing of this program is available in Appendix A.

If we step through our list, incrementing a counter at each step, we should get an accurate count of the number of nodes in our list, and hence it's length[1]

```c
int count(struct node * list)
{
    int count = 0;
    while(list != NULL)
    {
        list = list -> link;
        count++;
    }
    return count;
}
```

We can now count our (currently empty) list

Before we add any contents to our list, let's do something as equally useless as counting the number of nodes in an empty list, let's print the contents of our (currently empty) list.

```c
void display(struct node * list)
{
    while(list != NULL)
    {
        printf("%d ", list -> data);
        list = list -> link;
    }
    printf("\n");
}
```

Note that in both the count and display functions we are passing in our list, checking whether it is empty, and if not, stepping through the list until we reach the end (indicated by NULL in both the end of list and empty list cases because an empty list can also be interpreted as also being at the end of a very short list. Notice the pattern for traversing the list to process each node.

[1] "length" would be an equally good function name here but we can stick with count. Either would be sufficiently descriptive to indicate what we are doing.

Now we can display the contents of our empty linked list, we really need a way to add some content. How adding a new node to the end of the list?

```c
void append(struct node **list, int num)
{
    struct node *temp, *r;
    if(*list == NULL)
    {
        temp = (struct node *) malloc (sizeof(struct node));
        temp -> data = num;
        temp -> link = NULL;
        *list = temp;
    }
    else
    {
        temp = *list;
        while(temp -> link != NULL)
            temp = temp -> link;

        r = (struct node *) malloc(sizeof(struct node));
        r -> data = num;
        r -> link = NULL;
        temp -> link = r;
    }
}
```

Now perhaps we shold implement the inverse of append, let's add a function to prepend a new node to the list.

```c
void prepend(struct node ** list, int num)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp -> data = num;
    temp -> link = *list;
    *list = temp;
}
```

We might want to add a node to the middle of our list rather than just the start or end of the list. Let's implement a function to insert a node into a particular location.

```c
void insert_after(struct node * list, int location, int num)
{
    struct node *temp, *r;
    int i;
    temp = list;
    for(i=0; i<location; i++)
    {
        temp = temp -> link;
        if(temp == NULL)
        {
            printf("Length of list is %d but supplied location is %d\n", i,
                location);
            return;
        }
    }
    r = (struct node *) malloc (sizeof(struct node));
    r -> data = num;
    r -> link = temp -> link;
    temp -> link = r;
}
```

Most lists don't just grow though, they also need to shrink, we need to be able to remove elements from the list. How about a delete function?

```c
void delete(struct node ** list, int num)
{
    struct node *old, *temp;
    temp = *list;

    while(temp != NULL)
    {
        if(temp -> data == num)
        {
            if (temp == *list)
            {
                *list = temp -> link;
            }
            else
                old -> link = temp ->link;
                free(temp);
                return;
        }
        else
        {
            old = temp;
            temp = temp -> link;
        }
    }
    printf("Element %d not found in the supplied list\n", num);
}
```

That's all of the functions we set out for API at the beginning. Now we need to test out our API by creating a new empty list, then adding, traversing, diplaying, and deleting the contents.

```c
int main(void)
{
    struct node *list;
    list = NULL;

    printf("No of elements in linked list = %d\n", count(list) );

    append(&list, 14);
    append(&list, 30);
    append(&list, 25);
    append(&list, 42);
    append(&list, 17);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    prepend(&list, 999);
    prepend(&list, 888);
    prepend(&list, 777);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 1, 0);
    insert_after(list, 2, 1);
    insert_after(list, 5, 99);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 99, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 99);
    delete(&list, 1);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    return 0;
}
```

Run the program and play around with it. Look carefull at the nodes and their order in the output and consider how they relate to the function calls we have made to manipulate our list.

Now try implementing your own *insert_before* function to partner with the insert_after function. This should test whether you really understand your implementation. Similarly, consider how we've implemented the delete function, removing a specific element based on it's value, rather than the element in a particular position. Perhaps you should implement this second form of delete. As with any data structure, the more you think about it, the more functions you might come up with that make it more useful in specific situations, for example, functions to empty all of the contents of the list by de-linking them and freeing up the memory, or functions to concatenate one list on to another.

What we should take from this section is the idea that there are multiple ways to implement any given structure. Just like last week when we saw that there are multiple ways to describe the API for a data structure, the basic underlying implementation can also vary. This is part of the reason that data structures are often referred to as *asbtract* data structures, because they represent an ideal data structure, but there are many design decisions and constraints imposed along the path from the idea of the ideal abstract data structure to the implementation of any given concrete data structure. Theorising about data structures, and their characteristics can be very different to the experience of implementing and using them in real world contexts to solve actual programming problems.

2.2 Doubly Linked Lists

A drawback of the linked list is that if we have traversed our list, say we are part way through, perhaps at node 24 but we now need to access node 22, there is no way to backtrack to node 22. Instead, in the linked list that we've implemented, we have to traverse the entire list again right from the start until we get to node 22. If we then need to access node 18 we have to traverse everything again. This seems quite wasteful, to repeatedly traverse the list until we get to the place we need. Wouldn't it be better if we could just move backwards and forwards between locations in the list without needing to start at the beginning every time? Well, this is one limitation of the kind of list that we've just implemented, the singly linked list. In this list we only have one link, which always points forwards, to the next element. Wouldn't it be great if we added a second list? One which points back to the previous node which our original link continues to point forward to the next node? This is what the doubly linked list does, it merely has a second link in each node which points to the previous node so that we can travers our list in either direction, from front to back, or vice versa.

Let implement a doubly linked list. You'll notice that our implementation of the doubly linked list is very similar to our earlier linked list. For now we'll implement the exact same API, but you should now consider also any new API functions that you could implement to allow you to make use of the additional direction of travel that the link to previous elements gives us.

First we need a struct to hold the variables associated with each node in our doubly linked list.

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

Notice that we are now storing links to both the previous and the next node rather than just the next node. This is the key difference from our singly-linked list implementation. Obviously there is a knock on effect because we now have to manage two links for every node instead of just one link. So our job just got twice as difficult :D

As before, for each of these functions you'll need to add your own function prototypes to your source file. Note that a complete listing of this program is available in Appendix B.

Let's start, as before, with the count function.

```
int count(struct node *list)
{
    int count = 0;
    while(list != NULL)
    {
        list = list -> next;
        count++;;
    }
    return count;
}
```

Note that we don't really do anything different. We are just travelling (*traversing*) our list, starting at the beginning, processing the current node, before moving on to the next. Our first difference with the singly linked list though is that instead of using "link" we use "next" in the traversal. This because we are now keeping track of two links between nodes.

Now displaying our list.

```
void display(struct node * list)
{
    while(list != NULL)
    {
        printf("%2d\t", list -> data);
        list = list -> next;
    }
    printf("\n");
}
```

Again, very similar to before, travelling through our list, getting our navigational cue for the traversal from the next link. Note that in both the count and display functions, our prev links isn't used at all. We are traversing in one direction through the list, and for our purposes right now, it doesn't matter that we can also reverse direction.

Now on to appending and prepending, adding a node to the end of the list and adding a node to the beginning of the list respectively. First the append:

```
void append(struct node **list, int num)
{
    struct node *temp, *current = *list;
    if(*list == NULL)
    {
        *list = (struct node *) malloc(sizeof(struct node));
        (*list) -> prev = NULL;
        (*list) -> data = num;
        (*list) -> next = NULL;
    }
    else
    {
        while(current -> next != NULL)
            current = current -> next;

        temp = (struct node *) malloc(sizeof(struct node));
        temp -> data = num;
        temp -> next = NULL;
        temp -> prev = current;
        current -> next = temp;
    }
}
```

Now the prepend:

```c
void prepend(struct node ** list, int num)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp -> prev = NULL;
    temp -> data = num;
    temp -> next = *list;

    (*list) -> prev = temp;
    *list = temp;
}
```

Notice that we are now having to maintain links both next and previous nodes. Notice also that the append function is slightly more complex because we are having to traverse the list, to get to the end, before we can add our new node. Consider what steps you coult take to mitigate this and simplify the append function. Also take note of how there are similarities and differences between each case of creating a new node. In both cases we need to allocate enough storage for our struct, and to make our links point to the right places. However, if the node is at the beginning of the list then there is no previous node so this is set to NULL. The opposite is done for appending, when the new node is at the end of the list and obvious doesn't have a next node, yet, so the next link must be set to NULL.

A lot of the implementation of any list is merely managing which node points to which other node(s) depending upon the type of list you're handling. We'll revisit this later when we look at trees because these, at least from an implementational perspective, are a special type of list, one in which the organisation of the links between nodes point to parent and child nodes rather then previous and next, and, importantly, in which there might be multiple child nodes[2]. In fact, we'll see that graphs are also very similar in implementation, because trees are really just a special case of a graph. Practically speaking, a graph is a set of nodes that can point to any other node, including in some cases, themselves[3].

The insert_after and delete functions are both as for the singly-linked list, but now taking into account that we have an extra node link to manage.

```c
void insert_after(struct node * list, int location, int num)
{
    struct node *temp;
    int i;

    for(i=0; i<location; i++)
    {
        list = list -> next;
        if(list == NULL)
        {
            printf("Length of list is %d but supplied location is %d\n", i,
                location);
            return;
        }
    }
    list = list -> prev;
    temp = (struct node *) malloc (sizeof(struct node));

    temp -> data = num;
    temp -> prev = list;
    temp -> next = list -> next;
    temp -> next -> prev = temp;
    list -> next = temp;
}
```

[2]Depending upon the type of tree that you're implementing

[3]Again, depending upon the type of graph you are implementing (some have restrictions over the nature of links between nodes that they can support)

Again, managing our extra links in the forward and backward directions....

```c
void delete(struct node ** list, int num)
{
    struct node *temp = *list;
    while (temp != NULL)
    {
        if(temp -> data == num)
        {
            if(temp == *list)
            {
                *list = (*list) -> next;
                (*list) -> prev = NULL;
            }
            else
            {
                if(temp -> next == NULL)
                    temp -> prev -> next = NULL;
                else
                {
                    temp -> prev -> next = temp -> next;
                    temp -> next -> prev = temp -> prev;
                }
                free(temp);
            }
            return;
        }
        temp = temp -> next;
    }
    printf("Element %d not found in the supplied list\n", num);
}
```

Finally, here's a main function that exercises all of the API functions we just implemented.

```c
int main(void)
{
    struct node *list;
    list = NULL;

    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    append(&list, 14);
    append(&list, 30);
    append(&list, 25);
    append(&list, 42);
    append(&list, 17);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    prepend(&list, 999);
    prepend(&list, 888);
    prepend(&list, 777);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 1, 0);
    insert_after(list, 2, 1);
    insert_after(list, 5, 99);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 99, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 99);
    delete(&list, 1);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    return 0;
}
```

Whilst we have a display function that prints out our list, starting at the beginning of the list and completing at the end, this only takes us one way through the list and doesn't take advantage of the fact that we can now navigate in either direction. Perhaps it is worth implementing two new functions, traverse_forwards() and traverse_backwards() to cover the two cases of displaying in each direction. Alternatively, you could consider modifying your display function to take an argument indicating the direction of travel. In all cases, if we want to start at the end of the list then we need to know where that is. We don't currently store the end node, only the start node, our list variable in main(), so it might be worth considering storing this. Perhaps having a list struct that encapsulates node structs for the start and end of the list. In whichever case, you will likely have to edit every function that manipulates your list to track the end node, for example, when appending a node the end node will be replaced with a new one.

Again, this is just more evidence that no matter how simple our idea for a data structure, they can quickly become more complex, sometimes because we need more functionality, and sometime because we just want our structure to be as optimised as possible, however this can often be a trade-off.

3 summary

List are a fundamental data structure, just like arrays, they are merely a slightly more complex, but fantastically more flexible, way to exploit the physical and logical layout of computer memory. As such they give us really powerful tools on which we can base more complex ways to structure, store, and manipulate data. That said, they're pretty great in their own right. Many programmers, for example, in Python, can happily solve most of their problems by just processing lists of data[4]. There are even some programming languages, like Lisp, which are heavily dependent upon processing lists[5].

Comparing your code for the various array basd and list based data structure that we've discovered so far, you should be starting to see some implementational patterns. Perhaps in the way that we structure the data that a structure holds, perhaps in the characteristics of the API that is necessary for making a given structure useful, or else for implementing the key functionality expected for a given data structure, e.g. push and pop are charateristic of the stack although they do occur in the APIs of other data structures. It is alll down to getting the right syntax and semantics for how you want to interact with your data.

4 Challenges

1. Write a short program that stores the names and ages of a group of people and prints them out in order of age. Getting the order correct is possibly best done at construction time, as otherwise you will need a sort function[6]. You will also need to consider the data that each node of your list stores, our node struct so far has stored simple int values as their sole data, but now it looks like we need something a little more complex.

2. Revisit your stack implementation from the last lab. Reimplement your stack API using a linked list as the storage collection instead of an Array. At the very least, all you really need is a push and pop function for your list to enable you to (mis)use your list as a stack. What are the advantages and disadvantages of this approach?

[4]Although each node of that data might be more complex.

[5]Some people even say that LISP is a portmanteau of *LIS*t *P*rocessing, althought that language has many other aspects that make it great

[6]and we aren't going to look at sorting things just yet (although you can give it a go if you like, start simply, with the most naive sorting algorithm you can think of).

3. Revisit your queue implementation from the last lab. Reimplement your queue API using a linked list as the storage collection instead of an Array. At the very least, all you really need are enqueue and dequeue functions for your list to enable you to (mis)use your list as a queue What are the advantages and disadvantages of this approach?

4. Use a list structure to write the simplest of text-based dungeon crawler games. Each room in the dungeon should be a node in the list and your player must be able to navigate from room to room, having each room described to them as they progress. No need to consider monsters and treasure at this point, unless you want to. Just navigating between rooms can sometimes be a challenge at this point. Obviously this won't be a great dungeon crawler as the rooms will be laid out in a linear pattern, but it should be enough to get you thinking about how you might use lists to represent features of your own programs.

Part I
Appendices

A Linked List Source Code Listing

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

struct node
{
    int data;
    struct node * link;
};

void prepend(struct node **, int);
void append(struct node **, int);
int count(struct node *);
void delete(struct node **, int);
void display(struct node *);
void insert_after(struct node *, int , int);

int main(void)
{
    struct node *list;
    list = NULL;

    printf("No of elements in linked list = %d\n", count(list) );

    append(&list , 14);
    append(&list , 30);
    append(&list , 25);
    append(&list , 42);
    append(&list , 17);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    prepend(&list , 999);
    prepend(&list , 888);
    prepend(&list , 777);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list , 1, 0);
    insert_after(list , 2, 1);
    insert_after(list , 5, 99);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list , 99, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list , 99);
    delete(&list, 1);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    return 0;
}

void prepend(struct node ** list, int num)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp -> data = num;
    temp -> link = *list;
    *list = temp;
}

void append(struct node **list, int num)
{
    struct node *temp, *r;
    if(*list == NULL)
    {
        temp = (struct node *) malloc (sizeof(struct node));
        temp -> data = num;
        temp -> link = NULL;
        *list = temp;
    }
    else
    {
        temp = *list;
        while(temp -> link != NULL)
            temp = temp -> link;

        r = (struct node *) malloc(sizeof(struct node));
        r -> data = num;
        r -> link = NULL;
        temp -> link = r;
    }
}

int count(struct node * list)
{
    int count = 0;
    while(list != NULL)
    {
        list = list -> link;
        count++;
    }
    return count;
}

void delete(struct node ** list, int num)
{
    struct node *old, *temp;
    temp = *list;

    while(temp != NULL)
    {
        if(temp -> data == num)
        {
            if (temp == *list)
            {
                *list = temp -> link;
            }
            else
                old -> link = temp ->link;
                free(temp);
                return;
        }
        else
        {
            old = temp;
            temp = temp -> link;
        }
    }
    printf("Element %d not found in the supplied list\n", num);
}

void display(struct node * list)
{
    while(list != NULL)
    {
        printf("%d ", list -> data);
        list = list -> link;
    }
    printf("\n");
}

void insert_after(struct node * list, int location, int num)
{
    struct node *temp, *r;
    int i;
    temp = list;
    for(i=0; i<location; i++)
    {
        temp = temp -> link;
        if(temp == NULL)
        {
            printf("Length of list is %d but supplied location is %d\n", i,
                location);
            return;
        }
    }
    r = (struct node *) malloc (sizeof(struct node));
    r -> data = num;
    r -> link = temp -> link;
    temp -> link = r;
}
```

B Doubly Linked List Source Code Listing

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

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

void prepend(struct node **, int);
void append(struct node **, int);
int count(struct node *);
void delete(struct node **, int);
void display(struct node *);
void insert_after(struct node *, int, int);

int main(void)
{
    struct node *list;
    list = NULL;

    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    append(&list, 14);
    append(&list, 30);
    append(&list, 25);
    append(&list, 42);
    append(&list, 17);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    prepend(&list, 999);
    prepend(&list, 888);
    prepend(&list, 777);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 1, 0);
    insert_after(list, 2, 1);
    insert_after(list, 5, 99);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    insert_after(list, 99, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 99);
    delete(&list, 1);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    delete(&list, 10);
    printf("No of elements in linked list = %d\n", count(list) );
    display(list);

    return 0;
}

void prepend(struct node ** list, int num)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp -> prev = NULL;
    temp -> data = num;
    temp -> next = *list;

    (*list) -> prev = temp;
    *list = temp;
}

void append(struct node **list, int num)
{
    struct node *temp, *current = *list;
    if(*list == NULL)
    {
        *list = (struct node *) malloc(sizeof(struct node));
        (*list) -> prev = NULL;
        (*list) -> data = num;
        (*list) -> next = NULL;
    }
    else
    {
        while(current -> next != NULL)
            current = current -> next;

        temp = (struct node *) malloc(sizeof(struct node));
        temp -> data = num;
        temp -> next = NULL;
        temp -> prev = current;
        current -> next = temp;
    }
}

int count(struct node *list)
{
    int count = 0;
    while(list != NULL)
    {
        list = list -> next;
        count++;;
    }
    return count;
}

void delete(struct node ** list, int num)
{
    struct node *temp = *list;
    while (temp != NULL)
    {
        if(temp -> data == num)
        {
            if(temp == *list)
            {
                *list = (*list) -> next;
                (*list) -> prev = NULL;
            }
            else
            {
                if(temp -> next == NULL)
                    temp -> prev -> next = NULL;
                else
                {
                    temp -> prev -> next = temp -> next;
                    temp -> next -> prev = temp -> prev;
                }
                free(temp);
            }
            return;
        }
        temp = temp -> next;
    }
    printf("Element %d not found in the supplied list\n", num);
}

void display(struct node * list)
{
    while(list != NULL)
    {
        printf("%2d\t", list -> data);
        list = list -> next;
    }
    printf("\n");
}

void insert_after(struct node * list, int location, int num)
{
    struct node *temp;
    int i;

    for(i=0; i<location; i++)
    {
        list = list -> next;
        if(list == NULL)
        {
            printf("Length of list is %d but supplied location is %d\n", i,
                location);
            return;
        }
    }
    list = list -> prev;
    temp = (struct node *) malloc (sizeof(struct node));

    temp -> data = num;
    temp -> prev = list;
    temp -> next = list -> next;
    temp -> next -> prev = temp;
    list -> next = temp;
}
```