Edinburgh Napier University

SET08122 Algorithms & Data Structures

Lab 3 - Data Structures #1

Dr Simon Wells

1 Aims

Our goal this week is to build upon the work from last week. Firstly by extending our knowledge of Arrays, then using arrays as a building block for implementing and exploring some new data structures, the stack, queue, deque, and list.

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

- Understand and implement the API for Array, Stack, Queue, Deque, and List data structures.

If you are a confident or experienced programmer then you might have met all of these structures before. However others in the class may have had a different background or route to this class that did not include your experiences. We are starting however to go beyond the basic level, to use primitive datatypes and built in structure to define our own more complex structures which have specific properties.

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. So today, having refreshed our knowledge of C primitives, structs, and array usage last week, we can now build on this to implement a few structures that don't come as standard in the C programming language.

2.1 Arrays

Last week we used arrays in a pretty raw state, manipulating them directly, inserting and removing data from specific locations by directly accessing the relevant slots. However, very often we will create functions to manage this process for us. The advantage of this is that these functions can incorporate checks to catch errors and handle them appropriately. This set of functions canbe said to provide the API for our data structure. C implementations of data structures will often comprise the data storage and functions for safely manipulating it. Because we are familiar with arrays, let's try this out before applying the same principle to other data structures.

We are going to create an API (functions) for doing the following with an array:

1. insert
2. delete
3. init
4. reverse
5. display
6. search

Let's start with a framework for our program.

```
1  #include <stdio.h>
2
3  #define MAX 5
4
5  int main(void)
6  {
7      int array[MAX];
8
9      return 0;
10 }
```

We've included the necessary library, defined how big our array should be, then declared an array, called *array* of size MAX, within a normal C program scaffolding. Make sure that you understand what is happening and also that you have a program that compiles and runs (although it won't do too much right now).

The first thing we want to do is to add a function to display the contents of our array data structure. Let's do that. We'll need a function prototype:

```
1 void display(int*);
```

and a function implementation:

```
1  void display(int* array)
2  {
3      int idx;
4      for(idx=0; idx < MAX; idx++)
5      {
6          printf("%d\t", idx);
7      }
8      printf("\n");
9
10     for(idx=0; idx < MAX; idx++)
11     {
12         printf("%d\t", array[idx]);
13     }
14     printf("\n");
15 }
```

We also need to add a line to actually use our new function in main, so let's do that too:

```
1 int main(void)
2 {
3     int array[MAX];
4
5     display(array);
6
7     return 0;
8 }
```

Compile and run your program. You'll likely see some weird values printed out because we haven't actually initialised our array before we printed out it's contents. Let's add an initialisation function to our API now with the prototype:

```
1 void init(int*);
```

and the implementation:

```
1 void init(int* array)
2 {
3     int idx;
4     for(idx=0; idx < MAX; idx++)
5     {
6         array[idx] = 0;
7     }
8 }
```

and our updated main function to use it:

```
1 int main(void)
2 {
3     int array[MAX];
4
5     init(array);
6     display(array);
7
8     return 0;
9 }
```

Let's repeat the process for the remaining functions of our Array API. Starting with the insert function. All this is meant to do is to make space in our array by moving the content up one space so that there is now a vacant slot at the nominated position. Notice that it isn't merely replacing the value in a given array position, but is inserting into a given position, moving everything else around to make space, and perhaps losing the value at the end of the array if there are no free slots. Semantics, or meaning, is important. In this case the semantics of the word insert instead of replace (or update). Anyhow, let's implement it, starting with a function prototype:

```
1 void insert(int*, int pos, int num);
```

and a function implementation:

```
1  void insert(int* array, int pos, int num)
2  {
3      int idx;
4
5      for(idx = MAX-1; idx>= pos; idx--)
6      {
7          array[idx] = array[idx-1];
8      }
9      array[idx] = num;
10 }
```

and finally, some code that uses the function within main:

```
1  int main(void)
2  {
3      int array[MAX];
4      init(array);
5
6      insert(array, 1, 11);
7      insert(array, 2, 12);
8      insert(array, 3, 13);
9      insert(array, 4, 14);
10     insert(array, 5, 15);
11
12     printf("Array Contents: \n");
13     display(array);
14
15     return 0;
16 }
```

Check the output and make sure that you are understanding exactly what is happening.

For completions sake you might want to add another function that updates or replaces a value in a specified slot.

Let's now delete some stuff... First add the function definition:

```
1 void delete(int *, int pos);
```

and the implementation:

```
1  void delete(int * array, int pos)
2  {
3      int idx;
4      for(idx = pos; idx<MAX; idx++)
5      {
6          array[idx-1] = array[idx];
7      }
8      array[idx-1] = 0;
9  }
```

and our updated main function:

```
int main(void)
{
    int array[MAX];

    insert(array, 1, 11);
    insert(array, 2, 12);
    insert(array, 3, 13);
    insert(array, 4, 14);
    insert(array, 5, 15);

    printf("Array Contents: \n");
    display(array);

    delete(array, 5);
    delete(array, 2);

    printf("After Deletion: \n");
    display(array);

    insert(array, 2, 222);
    insert(array, 5, 555);

    printf("Array Insertion: \n");
    display(array);

    return 0;
}
```

Notice that we've inserted some values, the deleted some, then added some others. You should be able to trace the printed output when you run your program and match it to your code.

Reversing our array might be useful. Let's try that out, first our declaration:

```
void reverse(int*);
```

then our function implementation:

```
void reverse(int* array)
{
    int idx;
    for(idx=0; idx<MAX/2; idx++)
    {
        int temp = array[idx];
        array[idx] = array[MAX-1-idx];
        array[MAX-1-idx] = temp;
    }
}
```

and finally let's include an example of using the reverse function within main:

```
int main(void)
{
    int array[MAX];

    insert(array, 1, 11);
    insert(array, 2, 12);
    insert(array, 3, 13);
    insert(array, 4, 14);
    insert(array, 5, 15);

    printf("Array Contents: \n");
    display(array);

    delete(array, 5);
    delete(array, 2);

    printf("After Deletion: \n");
    display(array);

    insert(array, 2, 222);
```

```
21      insert(array, 5, 555);
22
23      printf("Array Insertion: \n");
24      display(array);
25
26      reverse(array);
27
28      printf("After Reversal: \n");
29      display(array);
30
31      return 0;
32  }
```

Finally, let's search our array for a particular value. We'll return to this again later to evaluate the complexity of our search function. First the prototype:

```
1  void search(int*, int num);
```

then the implementation:

```
1  void search(int* array, int num)
2  {
3      int idx;
4      for(idx=0; idx<MAX; idx++)
5      {
6          if(array[idx] == num)
7          {
8              printf("%d found in position %d\n", num, idx+1);
9              return;
10         }
11     }
12     if(idx == MAX)
13         printf("%d not found in array\n", num);
14 }
```

Finally, our completed program. If you're unsure of anything, compare what you've implemetned to the complete example in Appendix A

```
1  int main(void)
2  {
3      int array[MAX];
4
5      insert(array, 1, 11);
6      insert(array, 2, 12);
7      insert(array, 3, 13);
8      insert(array, 4, 14);
9      insert(array, 5, 15);
10
11     printf("Array Contents: \n");
12     display(array);
13
14     delete(array, 5);
15     delete(array, 2);
16
17     printf("After Deletion: \n");
18     display(array);
19
20     insert(array, 2, 222);
21     insert(array, 5, 555);
22
23     printf("Array Insertion: \n");
24     display(array);
25
26     reverse(array);
27
28     printf("After Reversal: \n");
29     display(array);
30
31     search(array, 222);
32     search(array, 666);
33
34     return 0;
```

```
35 }
```

As promised, we've now got a set of functions to manage an array, or really, any array that we pass into our array functions. We've got the implementations for the following:

1. insert
2. delete
3. init
4. reverse
5. display
6. search

Notice that in C, rather than encapsulating some data with the methods that operate upon it, instead we create some stored data, in this case our array variable, which we then pass into our functions. This means that our functions are quite generic, they give us a generic API for operating upon arrays of integers. Obviously more work would be required to operate on other primitives, but this is a good start. Our investigations into other data structures will follow a similar pattern, identifying an API, implementing some functions, then testing out those functions in our program's main function.

Don't just copy the code however, once you have something implemented, play around with it and make sure that you understand what is happening.

Using your code from last week, create an additional API function that will increase the size of your array for when it isn't big enough to store all of your data. One of the things that you should consider here is by how much to increase the size, too much and you are allocating space for data that you might never need, too little and you might have to increase the size of your array too often with a resulting effect on the performance of your software.

2.2 Stacks

Quite often we want our data structures to have specific behaviours which enable us to interact with them, or reason about their contents in particular ways. To achieve this we often think of a structure like the array and restrict the interface so that we can only interact with the structure in a particular way. The stack is one such structure. NB. Usually you *can* interact with a data structure in many other ways, it depends upon how strictly encapsulation is enforced, for example, in Python the language trusts the programmer to know what they are doing, so you can often still manipulate a stack as thought it was a standard list, whereas in Java the interface will usually restrict you from accessing the encapsulated data in a way that is not supported by the interface.

Stacks are known as a "Last In First Out", or *LIFO*, structures. Stacks are last-in-first-out data structures. This means that the last item added to the stack, or in data structures parlance "*pushed*" onto the stack, is the first item that is removed. Prove this to yourself by piling up some objects into a stack. To remove things from the stack you then take an item from the top, known as "popping" from the stack, which gives you an item and leaves the next item on the top of the stack. You can continue popping items off the stack until it is empty, or pushing items onto the stack until it is full, or some combination of the two.

Think of a stack of items in the real world, for example, a stack of books. Usually we would start a stack by placing an item, then putting another item on top, then continuing to do so until we run out of items. If you are a book lover then no stack of books is too big, even if it risks crushing you to death (which some bibliophiles might think is one of the better ways to go). To process the stack we take the item from the top and process it, in the case of a book we read it, then we get the next item, process (read) that, and then the next, until the stack is empty. Notice that the first item onto the stack, at the bottom of the stack is the last item to be processed (if you don't believe me then make a stack of things right now and try it out). Conversely the last item onto the stack is the first item to be processed. This what we mean by a LIFO data structure.

One way to think of a stack is as an array in which one end is the top and the other the bottom. We then only add and remove items from the top of the stack. It is simple to implement a stack in C by using an array and some functions to restrict how we interact with it. Whilst we implement a stack using an array, we actually restrict how we interact with the underlying array. Instead of inserting and deleting like we did earlier, with a stack we push and pop, two functions that define the API for a stack data structure, and which should be the only way that we interact with it.

The API for our Stack implementation is as follows:

1. init
2. push
3. pop

Obviously this API can be made more complicated and functional, with the addition of more functions, but these three cover all that we need, and only push and pop are really necessary to capture the essence of the Stack.

The full listing for our implementation is in Appendix B. But let's start constructing it from its parts now. First we'll make a structure to store our stack's variables. We will need an array to store the stacks contents themselves, and a second variable to point to the top of the stack.

```
1 struct stack
2 {
3     int array[MAX];
4     int top;
5 };
```

Bow let's deal with the init function. Notice that you will have to write the function declaration and test code from main yourself this time around. If you get stuck check the full listing in the appendices.

```
1 void init_stack(struct stack *s)
2 {
3     s->top = -1;
4 }
```

All we are doing here is accessing the variable that points to the top of the stack and setting it to -1 to indicate that the stack is empty. Our other functions will inspect and alter this as we use the stack, so getting it in the right place is important.

```
1  void push(struct stack *s, int item)
2  {
3      if(s->top == MAX-1)
4      {
5          printf("Stack is full. Couldn't push '%d' onto stack\n", item);
6          return;
7      }
8      s->top++;
9      s->array[s->top] = item;
10 }
```

All we are doing here is checking the position of the variable that indicates the top of the stack. If the top is equal to the maximum size of our underlying array, taking into account zero indexing, then our stack is full and we can't push anything on. Otherwise we alter where the variable pointing to top points to, then place our supplied item into the stack.

We are just putting new things into the array in the order in which they arrive and keeping track of which was the last item to be added so that we can access it when needed.

At this point we probably want to actually access our stack, so let's go ahead and implement the pop function:

```
1  int *pop(struct stack *s)
2  {
3      int *data;
4      if(s->top == -1)
5      {
6          printf("Stack is empty\n");
7          return NULL;
8      }
9      data = &s->array[s->top];
10     s->top--;
11     return data;
12 }
```

In this case we are creating some temporary storage for the item we are popping off the stakc. We are then checking the position of the top indicator variable. If this points to the bottom of the array then the stack is empty and we need to tell our user that there is nothing to pop. There are a number of ways to do that. in this case we are returning NULL so that the user, when calling this function can check for NULL. If they don't get a NULL then there is a usable item returned fomr the stack. This approach is a pragmatic decision. We could have taken a different approach, perhaps returned a struct that contains both the item and a status indicator, or else we could have created a separate function to perform the empty check. Note that we could have also crteated a function to perform the full check for push as well. There are a lot of ways to implement even a simple data structure and we have made a pragmatic choice here. We'll see a different approach in the Queue data structure later. Onwards for now. ONce we know that the stack is not empty we need to access the value stored in our current location in the stack and return it. Because the semantics of popping an item from the stack alters the stack. This is not peeking at that position to see what is there, but the equvalent of getting the next item from the stack so that the item is no longer there. As a result we also need to alter our pointer to the top of the stack to point to the item below our current item so that next time pop is called it will retrieve the next item rather than returning the same one repeatedly.

That is all the functions for a basic Stack API. Let's see how we would use them to creat a stack and manipulate it:

```
1  int main(void)
2  {
3      struct stack s;
4
5      init_stack(&s);
6
7      push(&s, 11);
8      push(&s, 23);
9      push(&s, -8);
10     push(&s, 16);
11     push(&s, 27);
12     push(&s, 14);
13     push(&s, 20);
14     push(&s, 39);
15     push(&s, 2);
16     push(&s, 15);
17
18     push(&s, 7);
19
20     int *i = NULL;
21
22     i =  pop(&s);
23     if(i) { printf("Item popped: %d\n", *i); }
24
25     i = pop(&s);
26     if(i) { printf("Item popped: %d\n", *i); }
27
28     i = pop(&s);
29     if(i) { printf("Item popped: %d\n", *i); }
30
31     i = pop(&s);
32     if(i) { printf("Item popped: %d\n", *i); }
33
```

```
34      i = pop(&s);
35      if(i) { printf("Item popped: %d\n", *i); }
36
37      i = pop(&s);
38      if(i) { printf("Item popped: %d\n", *i); }
39
40      i = pop(&s);
41      if(i) { printf("Item popped: %d\n", *i); }
42
43      i = pop(&s);
44      if(i) { printf("Item popped: %d\n", *i); }
45
46      i = pop(&s);
47      if(i) { printf("Item popped: %d\n", *i); }
48
49      i = pop(&s);
50      if(i) { printf("Item popped: %d\n", *i); }
51
52      i = pop(&s);
53      if(i) { printf("Item popped: %d\n", *i); }
54
55      return 0;
56  }
```

As I mentioned, there are other functions that a Stack API might implement, peeking at particular places in the stack might be useful in some circumstances. Similarly poking new items into specific positions in th Stack might also be useful. Searching for specific values without removing them from the stack can also be useful. It really depends upon how you need to use your stack, how feature rich you want, or need, your API to be, and how far from the *iconic* Stack API you want to deviate.

Consider why you might want to use a stack data structure. Do some research into how queues are used in computer programming. What are they most commonly used for? What is the most interesting usage you can find for this data structure? Find at least three examples of uses of the stack data structure.

2.3 Queues

Everybody naturally knows how a queue works (especially the British members of the class). Somebody waits for something, and somebody else stands behind the first person because they want the same thing but they are waiting their turn, a third person stands behind the second, and so on.

The basic idea with a queue is that you have data that should normally be processed in the order in which it arrives, just like a queue of customers is served in the supermarket in the order in which they arrive at the checkout[1]. One way to think of this is as data waiting for a service or processing. If you want that data to be processed in the order in which it arrives, or is genereated, then you probably want a queue. For these reasons a queue is often referred to as a "First In First out", or FIFO, data structure.

The API for our Queue implementation is as follows:

1. enqueue
2. dequeue
3. empty

As before, we can make this API more complex. The enqueue and dequeue functions are the minimum that is required. The empty function is just to help us along. It also shows a slightly different way of implementing things. Whereas in the Stack implementation there is some complex maneuvering to enable us to return both the content, an int, or NULL in the case of the stack being empty, in this case we have a simpler return from dequeue, just the content, but in order to

[1] unless of course they open another till. In which case there is usually a bit of a scrum and jockeying for places before the new second queue is constructure

use this safely we need to check whether the queue is empty of not before dequeuing items. An alternative to either approach would be to return a struct instead of a primitive data type. This could then contain both a status indicator and the content if the status was successful. We'd still have to check the returned struct, but this might be a cleaner solution. As an exercise, why not try doing new versions of your pop and dequeue functions so that they return a struct. In theory this might lead to some cleaner code and a better API.

The full listing for our implementation is in Appendix C. We are going to take a similar approach to presenting the Queue implemenation as we did before for Stacks. This time we are not using a Struct to encapsulate our base array. This is just a design decision that you should consider and decide whether it is advantageous or not. Because we don't have a struct holding all of our queue data, we are going to need an array and some variables to point to the front and the rear of the queue. For example:

```
int arr[MAX];
int front = -1, rear = -1;
```

Now let's see what the enqueue function will involve so that we can add data to our queue:

```
void enqueue(int *arr, int item, int *pfront, int *prear)
{
    if(*prear == MAX-1)
    {
        printf("Queue is full\n");
        return;
    }
    else
    {
        printf("Enqueuing: %d\n", item);
        (*prear)++;
        arr[*prear] = item;

        if(*pfront == -1)
            *pfront = 0;
    }
}
```

For this we passing in the array that stores our queue, the item to add, and the variables that point to the fron and rear of the queue. We then check, using the rear variable, whether the queue is full and do something appropriate if it is. Otherwise we increment the rear indicator, add our new item where the rear indicator points to and alter the front indicator, which we use to check whether the queue is empty or not, to indicate that the queue is not empty.

Now we need to implement our function to check if the queue is empty of not. In this case we are just checking whether our front indicator is negative 1, in which case the the queue is empty, or o, in which case the queue has content.

```
int empty(int *pfront)
{
    if(*pfront == -1)
    {
        printf("Queue is empty\n");
        return 1;
    }
    else
        return 0;
}
```

We will use this function to determine whether it is safe to call the dequeue method. Which, funnily enough, is exactly what we'll look at next.

```
int dequeue(int *arr, int *pfront, int *prear)
{
    int data = arr[*pfront];
    printf("Dequeueing: %d\n", data);

    arr[*pfront] = 0;
```

```
7       if(*pfront == *prear)
8           *pfront = *prear = -1;
9       else
10          (*pfront)++;
11
12      return data;
13  }
```

When we dequeue an item we want to return the next item in the queue to our caller so we use the front indicator variable to access it directly ready to return. Notice that, because the semantics of the queue mean that the newest item is at the rear of the queue, and the oldest item is at the front, we add things using the read indicator to tell us where to put things, then remove items by using the front indicator to retrieve them from the opposite end of the array. Once we have our item we are then adjusting our indicators. As enqueing and dequeing are both functions whose semantics make changes to the underlying data structure, by retrieving an item using dequeue, we also need to adjust our indicators so that the next time dequeue is called it retrieves the next item instead of the current one again. In this case we first set our front indicator to zero, then check whether our the queue is now empty (which happens if we've just dequeued the final item) by seeing if the front and rear point to the same place, in which case we set the indicators to negative 1 just as we did when initialising our array. If the queue isn't empty we need to increment our indicator for the front so it points to the next item in the queue ready for the next call. Finally we return our data item.

Now let's look at how we could use our enqueue, dequeue, and empty functions:

```
1   int main()
2   {
3       int arr[MAX];
4       int front = -1, rear = -1;
5
6       enqueue(arr, 23, &front, &rear);
7       enqueue(arr, 9, &front, &rear);
8       enqueue(arr, 11, &front, &rear);
9       enqueue(arr, -10, &front, &rear);
10      enqueue(arr, 25, &front, &rear);
11      enqueue(arr, 16, &front, &rear);
12      enqueue(arr, 17, &front, &rear);
13      enqueue(arr, 22, &front, &rear);
14      enqueue(arr, 19, &front, &rear);
15      enqueue(arr, 30, &front, &rear);
16      enqueue(arr, 32, &front, &rear);
17
18      int i;
19      for(int idx=0; idx<MAX+1; idx++)
20      {
21          if(!empty(&front))
22          {
23              i = dequeue(arr, &front, &rear);
24              printf("Received Dequeued item: %d\n", i);
25          }
26      }
27
28      return 0;
29  }
```

Again, we've taken a slightly different approach to demonstrating as we did for the Stack example. This is mainly so that we reinforce the idea that there is an ideal abstract conceot for the stack or the queue, and that all implementations are imperfect versions of this. Beacuse the problems that we need to solve in the real world are rarely perfect problems, they have messy corner cases, and weird idiosyncracies, so similarly, our data structure implementations sometimes need to bend to the needs of the real world. Similarly implementation in different languages can lead to more complex, or simpler implementations. This doesn't necessarily indicate a better language, just one that perhaps fits with the abstract concept a little better in the specific case that you are looking at.

Consider why you might want to use a queue data structure. Do some research into how queues are used in computer programming. What are they most commonly used for? What is the most

interesting usage you can find for this data structure? Find at least three examples of uses of the queue data structure.

Note that our example is a far from ideal implementation. You could easily extend the API for your queue to include an initialisation function to set up your array. You might also want to consider a way to move elements forward in the array so that after dequeuing there is always the maximum amount of space left at the rear of the queue for additional items. There are always alternative approaches however, one to consider is making the queue circular so that front and rear indicators are always chasing each other. This can avoid the need to move items around in the array, but at the expense of slightly more complex array management.

We will return to Stacks and Queues later, once we have some other data structures under our belts, in order to see some different ways of implementing them, for example, using linked lists. Again, this can lead to a range of advantages and disadvantages, which must be balanced by pragmatism. Ultimately, in the real world, we would use a standard or library implementation of these concepts so that we knew we were using a solution that had been tested and fixed by a lot of people before we got to it.

What we should take away from this topic is the idea that very small changes in behavisectionour enable us to specialise our data structures to capture various real-world behaviours. We see this with the array, where an API that constrains of our use in one way, to pushing and popping, gives us stack-like behaviour, and another API that constrains our use of an array to just enqueueing and dequeing give us queue-like behaviour.

Now we could get similar behaviour to a queue data structure using alternative means. We could just use a list of data with timestamps and each time we want to process some data we check the timestamps of all elements and only process the oldest, then check through and get the next oldest. This would have pretty poor performance however. We might imagine a more efficient solution that involved sorting the collection (which we'll see later in the module) in order of timestamp, inserting new data in timestamp-sorted order as required. However, in many circumstance, the simplest and most robust solution is a data structure that naturally gives you the oldest element each time without sorting or comparisons purely by virtue of structure. In this case, that is what a queue does, gives us the oldest element when we ask for it.

Often when writing larger software we can get bogged down in custom designed, increasingly-asbtract, hierarchies of classes when what we often need is a simple data structure that captures the essence of the specific problem in the simplest most robust way possible[2].

2.4 Challenges

2.4.1 Implementng the Deque Data Structure

This is the double ended queue which is pronounced "Deck"[3]. As it's name suggests, the Deque can have elements added to or removed from both the head or the tail of the queue. This gives the Deque similar but slightly different behaviour and characteristics compared to the (single-ended) Queue.

Furthermore, we can specialise the Deque in a few ways to further modify it's behaviour, for example, by restricting either the input or output at a given end of the queue, e.g. an input restricted deque will support removal of elements are both ends but insertion is restricted to one end only. An output restricted deque is simlar but removal is restricted to one end only whilst either end supports insertion.

1. Research the methods that a pure deque implementation would support. Compare these to the methods that we've implemented so far for the Stack and Queue data structures.

[2]We should also always remember the YAGNI principle. When designing a piece of software and you want to add support for a feature just in case remind yourself "you aren't gonna need it", unless of course you eventually do, but that is another story

[3]Some people pronouce it "dee-queue" but this risks confusion with the method used to remove an element from a queue, i.e. 'to dequeue an element'

2. Write a simple C program that implements the functilns necessary for a deque data structure.

3. Test your implementation by writing a simple program to check whether a string is a palindrome.

4. Consider why you might want to use a deque data structure. Do some research into how deques are used in computer programming. What are they most commonly used for? What is the most interesting usage you can find for this data structure? Find at least three examples of uses of the deque data structure.

2.4.2 Implementing the List Data Structure

A list is a simple example of a sequence, or sequential, data structure. We are going to explore sequences more in the stack and queue data structures.

1. Research the methods that a pure list implementation would support. Implement this using C. You might find that a list is not really much more than our array API with a few additional functions.

2. Write a simple program that stores a shopping list (if you don't have any shopping to do then create a wishlist of some sort). Your program does not need a user interface because we don't want to get sidetracked into that right now. Your program should print the content of your list, add items to the list, remove individual items from the list, remove all items from the list.

3. (Optionally) Add a simple command line interface to your program to enable you to control the contents and behaviour of your list. Consider what things you might want to do with your shopping list to extend it's behaviour, perhaps you need to combine shopping lists from multiple people, for example, you and your flatmates. How might you achieve this?

You should use your own judgement about whether you have met the challenge posed (however that doesn't preclude discussing your solution with the lecturers, or demonstrators, or your peers.)

3 summary

We have seen that even the simplest, most basic data structures give us a language for describing solutions to programming problems. Data structures, particularly in their abstract sense, are design patterns that can help us to see how to approach a challenge or enable us to communicate our ideas to other developers. Because many data structures have been studied in their own right their behaviour and performance characteristics are often known, and particular implementations can be compared against other implementations or ideal behaviour. As such, it is very important, if you want to become a great programmer, to have a thorough grasp of at least the basic, and often more esoteric, data structures.

Additionally, these data structures are a simple place to start[4] for an exploration of algorithms, as many of the more famous, or popular algorithms, in the simplest case, operate on bog standard data-structures.

Part I
Appendices

A Array Listing

```
1 #include <stdio.h>
2
3 #define MAX 5
4
5 void init(int* array);
```

[4] We can make things as complicated as we want (or as necessary) but let's start simple

```c
void insert(int*, int pos, int num);
void delete(int *, int pos);
void reverse(int*);
void display(int*);
void search(int*, int num);

int main(void)
{
    int array[MAX];

    init(array);
    display(array);


    return 0;
}


void insert(int* array, int pos, int num)
{
    int idx;

    for(idx = MAX-1; idx>= pos; idx--)
    {
        array[idx] = array[idx-1];
    }
    array[idx] = num;
}

void delete(int * array, int pos)
{
    int idx;
    for(idx = pos; idx<MAX; idx++)
    {
        array[idx-1] = array[idx];
    }
    array[idx-1] = 0;
}

void reverse(int* array)
{
    int idx;
    for(idx=0; idx<MAX/2; idx++)
    {
        int temp = array[idx];
        array[idx] = array[MAX-1-idx];
        array[MAX-1-idx] = temp;
    }
}

void init(int* array)
{
    int idx;
    for(idx=0; idx < MAX; idx++)
    {
        array[idx] = 0;
    }
}

void display(int* array)
{
    int idx;
    for(idx=0; idx < MAX; idx++)
    {
        printf("%d\t", idx);
    }
    printf("\n");

    for(idx=0; idx < MAX; idx++)
    {
        printf("%d\t", array[idx]);
    }
    printf("\n");
}

void search(int* array, int num)
```

```
{
    int idx;
    for(idx=0; idx<MAX; idx++)
    {
        if(array[idx] == num)
        {
            printf("%d found in position %d\n", num, idx+1);
            return;
        }
    }
    if(idx == MAX)
        printf("%d not found in array\n", num);
}
```

B Stack Listing

```
#include <stdio.h>

#define MAX 10

struct stack
{
    int array[MAX];
    int top;
};

void init_stack(struct stack *);
void push(struct stack *, int item);
int *pop(struct stack *);

int main(void)
{
    struct stack s;

    init_stack(&s);

    push(&s, 11);
    push(&s, 23);
    push(&s, -8);
    push(&s, 16);
    push(&s, 27);
    push(&s, 14);
    push(&s, 20);
    push(&s, 39);
    push(&s, 2);
    push(&s, 15);

    push(&s, 7);

    int *i = NULL;

    i =  pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
    if(i) { printf("Item popped: %d\n", *i); }

    i = pop(&s);
```

```
58      if(i) { printf("Item popped: %d\n", *i); }
59
60      i = pop(&s);
61      if(i) { printf("Item popped: %d\n", *i); }
62
63      i = pop(&s);
64      if(i) { printf("Item popped: %d\n", *i); }
65
66      i = pop(&s);
67      if(i) { printf("Item popped: %d\n", *i); }
68
69      return 0;
70  }
71
72  void init_stack(struct stack *s)
73  {
74      s->top = -1;
75  }
76
77  void push(struct stack *s, int item)
78  {
79      if(s->top == MAX-1)
80      {
81          printf("Stack is full. Couldn't push '%d' onto stack\n", item);
82          return;
83      }
84      s->top++;
85      s->array[s->top] = item;
86  }
87
88
89  int *pop(struct stack *s)
90  {
91      int *data;
92      if(s->top == -1)
93      {
94          printf("Stack is empty\n");
95          return NULL;
96      }
97      data = &s->array[s->top];
98      s->top--;
99      return data;
100 }
```

C Queue Listing

```
1   #include <stdio.h>
2
3   #define MAX 10
4
5   void enqueue(int *, int, int *, int *);
6   int dequeue(int *, int *, int *);
7   int empty(int *);
8
9   int main()
10  {
11      int arr[MAX];
12      int front = -1, rear = -1;
13
14      enqueue(arr, 23, &front, &rear);
15      enqueue(arr, 9, &front, &rear);
16      enqueue(arr, 11, &front, &rear);
17      enqueue(arr, -10, &front, &rear);
18      enqueue(arr, 25, &front, &rear);
19      enqueue(arr, 16, &front, &rear);
20      enqueue(arr, 17, &front, &rear);
21      enqueue(arr, 22, &front, &rear);
22      enqueue(arr, 19, &front, &rear);
23      enqueue(arr, 30, &front, &rear);
24      enqueue(arr, 32, &front, &rear);
25
26      int i;
27      for(int idx=0; idx<MAX+1; idx++)
```

```
    {
        if(!empty(&front))
        {
            i = dequeue(arr, &front, &rear);
            printf("Received Dequeued item: %d\n", i);
        }
    }

    return 0;
}

void enqueue(int *arr, int item, int *pfront, int *prear)
{
    if(*prear == MAX-1)
    {
        printf("Queue is full\n");
        return;
    }
    else
    {
        printf("Enqueuing: %d\n", item);
        (*prear)++;
        arr[*prear] = item;

        if(*pfront == -1)
            *pfront = 0;
    }
}

int empty(int *pfront)
{
    if(*pfront == -1)
    {
        printf("Queue is empty\n");
        return 1;
    }
    else
        return 0;
}

int dequeue(int *arr, int *pfront, int *prear)
{
    int data = arr[*pfront];
    printf("Dequeueing: %d\n", data);

    arr[*pfront] = 0;
    if(*pfront == *prear)
        *pfront = *prear = -1;
    else
        (*pfront)++;

    return data;
}
```