4. Static Data Structures

Aims

- to describe what a data structure is;
- to review the array;
- to introduce the notation for describing algorithm complexity;
- to describe the static list data structure.

4.1 Data Structures

A data structure is a logical collection of related data values. The values may be of a single type or the values may be of different types. Each data item in a structure can be dealt with individually or occasionally the whole structure may be processed at once. A data structure may be linear (lists, stacks, queues) or non-linear (trees, graphs).

4.1.1 Static Data Structures

- Size of structure is predefined.
- Cannot grow or shrink.
- Memory is allocated at compile time.
- Memory is contiguous.
- Random access may be faster.

4.1.2 Arrays

An array is a static data structure for the storage of a collection of data items that are of the same type. An array is:

1. Homogeneous: every value stored in the array must be of the same type.
2. Ordered: the individual components of the array can be listed one by one.

All the data stored in the array is referenced using a single variable name. Each element of the array can hold one data item, and is individually referenced by using the array name and an index number.

An array has two fundamental properties that must be specified during creation:

1. Size: the number of elements the array contains.
2. Type: the type of data item that can be stored in the elements of the array.

As with all variables in C, an array must be declared before use. The syntax for an array declaration is:

```
type name[size];
```
where type is the type of data item that can be stored in the elements of the array, name is the name of the array variable, and size is a constant indicating the number of elements allocated to the array. e.g.

```c
int exam_results[10];
```
declares an array named exam_results with 10 elements, each of which is of type int.

It is good programming practice to specify the size of the array using a symbolic constant. e.g.

```c
#define MAXITEMS 10
int exam_results[MAXITEMS];
```

Each element of the array is identified by an integer value called its index. In C, the index numbers always begin with 0 and run up to the array size minus one.

### Example

```c
#define MAXITEMS 10
int exam_results[MAXITEMS];
int index;
int attendance_mark;
exam_results[2] = 50;
exam_results[4] = 15 + attendance_mark;
index = 0;
exam_results[index] = 75;
exam_results[index++] = 10;
```

**SAQ 4.1**

1. Write the code to declare an array of floats named temperature, to store twenty four values.
2. In which element of the array would you store the (a) 1st value, (b) 24th value?
3. Write the code to store the value 15.6 in element 7 of the array.
4. In which array element does the following code store the value 30.2? Why?
4.2 Algorithm Complexity

An algorithm is a sequence of steps to solve a given problem. Complexity is a measure of the efficiency of an algorithm, allowing two algorithms for the same problem to be compared. There are two measures of complexity:

1. **Time complexity**: how long does the algorithm take to execute as the data set increases in size?
2. **Space complexity**: how does the execution space vary as the data set increases in size?

Given a problem such as finding an item of data in a particular data structure, there are other factors that influence the time taken. e.g.

1. **Best case**: the record is first in the data structure.
2. **Average case**: records are evenly distributed.
3. **Worst case**: the record is last in the data structure.

We usually compute **worst-case** complexity.

We need a method that relates computation time to problem size. Then, if the time for a given problem size is known, we can estimate the time for a different data set.

4.2.1 Big-Ω Notation

A notation called big-Ω notation is used to denote the computational complexity of algorithms. The notation is simple and consists of the letter Ω, followed by a formula enclosed in parentheses. The notation ignores differences in hardware, compilers etc. and indicates the rate of growth of time as a data set increases in size.
A single operation that is independent of the number of data items is said to have a complexity of $O(1)$.

Consider the case where a loop executes once for each item of data in a data set. The complexity is dependent upon the size of the data set and such an algorithm has complexity $O(n)$. The time taken is directly proportional to the size of the data set.

### 4.3 Static List

The list is the simplest data structure and the basis of most linear data structures. The rules governing a list may depend on its intended use. e.g.

- are duplicated items allowed?
- is order important?

An array can be used to implement the list.

```c
#define MAXLEN 100
int list[MAXLEN];
int length;
length = 0; /* signifies an empty list */
```

What value should `length` hold?

#### 4.3.1 Adding an Item

Each new list entry should be placed at the end of the list.

`length` records the position of the first element of free space.

What is the time complexity of `addItem`?

How do we know when the list is full?

Should we allow duplicates to be added to the list?
4.3.2 Removing an Item

removeItem(67);

What should the list look like afterwards?

All the elements after the removed item have moved.

4.3.2.1 removeItem Implementation

1. Find the item to be removed.

```cpp
void removeItem(int item)
{
    int x, found;

    for (x = 0; x < length; x++)
    {
        if (item == list[x])
            found = x;
    }
```
Element number 2, found = 2;

What is the time complexity of this search?

2. Copy 98 into element 2.

<table>
<thead>
<tr>
<th>BusList</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>…</th>
<th>…</th>
<th>MAXLEN-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Copy other elements one at a time, until the end of the list.

<table>
<thead>
<tr>
<th>BusList</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>…</th>
<th>…</th>
<th>MAXLEN-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is there a simpler method?

<table>
<thead>
<tr>
<th>BusList</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>…</th>
<th>…</th>
<th>MAXLEN-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy the last element into position found (provided that order is not important).

4.3.2.2 `removeItem` Implementation

What is the complexity of this search?