Chapter 16 - Storing data and records

16.1 Introduction

When a computer stores data in a program or an application such as a database, it needs to know what type of data it is. This is because different data types require different amounts of storage space. Before we look at storing data in both fixed-size records and variable-sized records, we need to look in a little more detail at the most common data types.

16.1.1 Integers

An integer is a positive or negative whole number. It has no fractional part. 5, -23 and 800 are examples of integers while 5.7, -23.98 and 4.0 are all real numbers because they have a fractional part (even if the fractional part equals zero).

We have already seen some methods of storing integers. We have seen the pure binary method of representing integers, the sign and magnitude system and the two's complement system. Using the pure binary method, for example, we can represent whole positive numbers between 0 and 255 if we use one byte. That's because there are 256 unique combinations of 8 bits. (0000 0000, 0000 0001, 0000 0010, 0000 0011, up to 1111 1111). One byte per integer is of little use, however, if you need to store numbers between 0 and 1000, for example. In this case, two bytes would be needed. If we use two bytes, we can represent integers between 0 and 65535.

We know that pure binary is of no use if we want to represent negative numbers. In this case, we could use sign and magnitude if we just want to hold a positive or negative number but don't need to do any calculations on the numbers. If calculations on positive or negative numbers are needed, then the two's complement system should be selected. The key questions to ask, then, when deciding on what binary system to use to represent integers are:

- What range of numbers do I need to store?
- Do I need to hold negative as well as positive numbers?
- Do I need to do calculations on the numbers?

16.1.2 Real numbers

Real numbers are numbers that have a fractional part. For example 6.7 and -23.65 are real numbers because they both have a fractional part. So far, we have only looked at using a byte or two bytes to represent integers. To represent real numbers in a computer, we use a system called the 'floating-point' system. How is this used to store a real number?

Consider this. The number 3547.54 is a real number. It's quite complex in that it has a whole part and a fractional part. This number can also be represented as 0.354754×10^4. The 0.354754 is known as the 'mantissa' and the 4 is known as the 'exponent'. If I now store both the mantissa and the exponent, I can store all the information I need to get back to the original number. You may have come across this way of representing numbers before in your maths lessons. It's known as 'standard form'. If you want to represent a real number then, you need to store both the mantissa and the exponent. If you only used one byte, the range and the accuracy of the numbers you could store would be very limited. Real numbers are therefore usually represented using either 2 bytes or 4 bytes. We will look in much more detail at floating-point numbers and become expert in their use later in the book!

16.1.3 String

Data type string is used to hold one or more characters (letters, numbers, special characters or control characters) from the keyboard. We have already seen that PCs use ASCII. Every character on the keyboard has a unique binary code. To represent each character requires 1 byte. If you were setting up a database, for example, and you needed to store the NAME of someone, you would define NAME to be a data type 'string' because each name is made up of a string of characters from the keyboard. But how many characters should you allow? You need to think what the longest name possible is and then add a bit, just in case! For example, you might decide the longest name you can think of is 20 characters long. You need to allow 20 bytes of storage space for every person's name, plus say an extra 5 for 'insurance' - just in case there is a slightly longer name than you thought of! The amount of storage space will quickly increase as the number of NAMEs you store increases. There is a trick you can use, however, to reduce the amount of space you need.

16.1.3.1 Coding text values to save on storage space

Suppose you had a university database that stored each NAME OF STUDENT at university and the TITLE OF DEGREE they were doing. There are 100 different degrees. If you had allowed 100 bytes for each TITLE OF DEGREE and you had 10 000

students, you would need 1 000 000 bytes to store their degrees. However, if we coded up the titles of degrees (Maths could be code MA, History of Art could be HA and so on) then we would only need to store two bytes per student for the title of degree. This method of coding is a very useful way of reducing the amount of storage space needed so long as there are a fixed number of choices. It enables you to store lots of pieces of information as a short code. It wouldn't be a suitable method for NAME OF STUDENT, for example, because there are an endless number of possible names!

There are some other benefits of coding data. It helps reduce the amount of time needed to enter in a particular piece of data into a database. If the codes are selected in a certain way, only authorised people will know what they mean. Therefore the data is slightly more secure. In addition, you can easily apply validation rules to codes whereas applying them to long text values is difficult.

16.1.4 Character

Data type 'character' is used to store just one character from the keyboard. It is usually a letter that is stored. For example, you might have in a university database the LOCATION of a degree, what campus it is taught on. You might have a choice of four locations, which have been coded up into A, B, C or D. However, any character, whether text, number, special or control, can be stored in this data type.

16.1.5 Boolean

Any piece of information that can have two and only two choices should be declared as data type Boolean. For example, GENDER should be Boolean because there are only two choices. 'UK STUDENT?' should be data type Boolean because every student is either a UK student, or not. Each Boolean data item needs one bit to hold its value. For example, 'bit three' could be set if the gender of the student is male and reset if they are female. One byte can be used to hold 8 Boolean data items.

16.1.6 Date

Date fields hold calendar dates! You need to decide, however, what format you want the date to be in because this affects how many bytes you need to store the date. For example, if you want the date to be DD/MM/YY (for example, 21/08/86) then you need 6 bytes. (You don't need to store the / symbol between the numbers). If, however, you want the date in the form DD/MM/YYYY (for example, 21/08/1986) then you need 8 bytes.

16.1.7 Calculated fields

It is worth mentioning that some pieces of data aren't stored at all and so may not need to be declared in the same way as pieces of data that are stored. For example, calculated fields are used in databases. These pieces of information are not stored in a database or a program, but are calculated at the time they are needed from pieces of information that you have stored!

You can show how clever you are in any database project by using a calculated field. A classic example is where you store someone's DATE_OF_BIRTH (data type DATE, 6 or 8 bytes) but you don't store their age. (If you did store their age rather than their date of birth, you would have to change each record every single year!!) If you needed to display someone's age, perhaps in a report, then you would calculate it in the background with programming code, using their DATE_OF_BIRTH information and the computer's internal date. The code to do this for any programming language you are familiar with can be found by searching the Internet.

16.1.8 Telephone numbers and other non-integer codes

When we looked at BCD numbers, we saw that just because a number is made up of numbers doesn't mean it is a number!! A number should only be declared a data type number if you may need to do calculations with it. If you don't ever need to do calculations with it then it is safer to declare it as text or as a BCD (Binary Coded Decimal) number. This is because leading zeros in integers are removed but are not removed if the piece of data is a data type text or BCD! Try, for example, typing in 00243 into a calculator. Either it won't enter the leading zeros or it removes them after you have entered the number in. Telephone numbers sometimes begin with a leading zero and so should be declared as data type text or BCD. If you had a phone number 764880, you could use 6 bytes if the data type was declared as a text. One byte would be used to represent the 7. The second byte would be used to represent the 6, and so on. You could, however, use 3 bytes to store the number as a BCD number in the following way:

1) You would store 0111 0110 in the first byte. (The 7 would occupy the 4 bits on the left of the first byte and the 6 would occupy the 4 bits on the right of the first byte.
2) You would store 0100 1000 in the second byte. (The 4 would occupy the 4 bits on the left of this byte and the 8 would occupy the 4 bits on the right of this byte).
3) You would store 1000 0000 in the second byte. (The 8 would occupy the 4 bits on the left of this byte and the 0 would occupy the 4 bits on the right of this byte).

There are other examples of 'numbers' that are really codes - they are never are added', for example, to other 'numbers'. Consider a 6-digit product code made up only of numbers. The numbers in the code mean something - they are codes rather than numbers. For example, the first number might show a type of product, the next 2 numbers might be a 'country of origin' code and the last three numbers might be the manufacturer's ID. You may need to strip out these numbers from a code, and the first number might also be a zero. You will never need to add one code to another, however, because this type of calculation for this piece of data is a nonsense. You don't ever add one person's telephone number to another person's telephone number for any reason or add the product code for one item to the product code for another item! Codes shouldn't be stored as data type number because arithmetic is never performed on them and because the code may begin with a zero. They should be stored as data type text or data type BCD.

16.2 An example of sizing a fixed-length file

Now that we know some details and typical uses of the most common data types, we can look in more detail at how a computerised database might store data using fixed-length records. We will use an example to see how to work out the size of a file that holds fixed-length records and what we need to consider. Information in a computer database is stored in an organised manner. Consider the following database of information, about dogs. There are seven 'fields' in each 'record' in this 'file'.

ID	Name	Type	Date of Birth	Gender	Weight (Kg)	Owner's phone number
0001	Prince	Poodle	21/08/02	M	15.7	234232
0002	Gnash	Bulldog	03/04/99	F	16.5	554543
0003	Sammy	Terrier	10/09/01	F	13.9	654455
0004	Freddy	Alsatian	12/12/00	M	20.0	712323
0005	Sammy	Greyhound	07/01/97	F	18.5	665643

A file of records about dogs.

There are five database terms you need to be familiar with.

16.2.1 Field

A field is an individual piece of information. In the above example, ID is a field, Name is a field, Type is a field and so on. Fields are also known as 'attributes'. Note that fields correspond to columns in a table.

16.2.2 Record

When you put some data into fields, all the data about one person, or one pet, or one subject and so on, make up a 'record'. Records correspond to rows in a table. For example, all the information about Prince is referred to as 'Prince's record'.

16.2.3 File

All of the records together make up a file of information.

16.2.4 Database

Whether you have just one file, or whether you have lots of files in a system, together they are known as a 'database'. If your files are interconnected in some way, then they are known as a 'relational database'. You will learn to love relational databases later in the book!

16.2.5 Primary key

Every record needs a primary key. It is a piece of data that is **unique** for that record. It can be used to pick out one and only one record. Looking at the small example above, there are 2 dogs called Sammy. Imagine if there were hundreds of dogs called Sammy. We will need to pick out from time to time individual records. Often, the only way we can do this is to add a new field to our database. This field is called the 'Primary Key'. You may know lots of examples of primary keys - a pupil ID number on the pupil database, a National Insurance number, a driving licence number, a club membership number and so on. Without primary keys, you have no way of picking out ***only*** one particular record. If you are making a database and you don't have a field whose data is unique for each record, then you need to add one, perhaps an ID field, for example.

16.2.6 Decide on the data type and size to allow

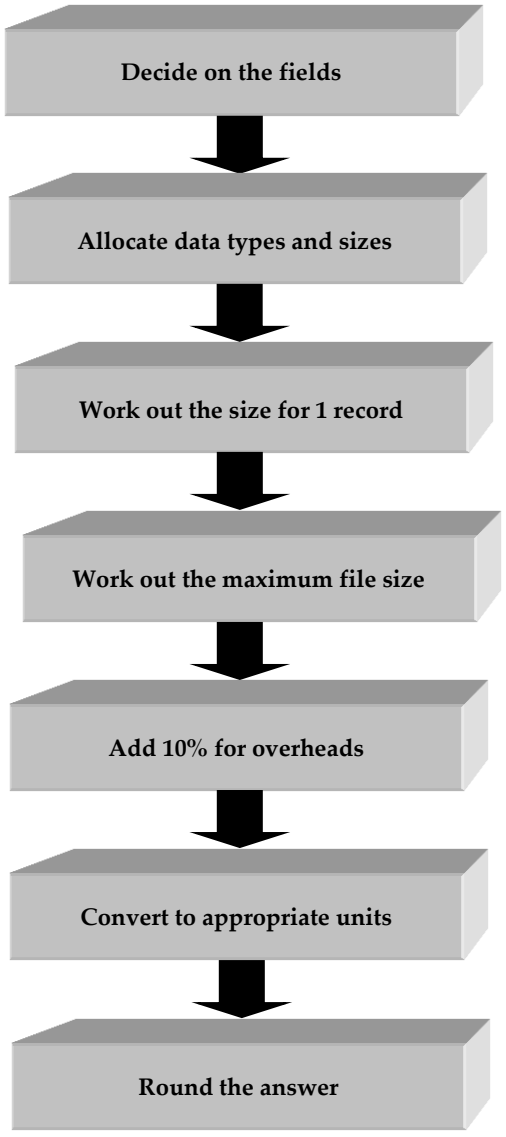
If you have ever set up a database, then you will have told your application what data type you want to use for each of your fields in a file, and also some information about how much space to allow for each field. This information would be brought together in a document called a '**Data Dictionary**'. For example, in the dog file, you may have decided on the following:

1) **ID.** This field is ***not*** a number but is an ID code. Therefore we will not use data type Integer but will use data type text instead. We will assume that the maximum number of dogs that will ever be in this file is 5000 so that an ID code of 4 characters long will be fine. We will allow 4 bytes for the ID code, one byte for each character.
2) **Name.** We know that some people give their dogs very long names. It is difficult to judge what to allow so we will allow plenty of room for error. We will allow 50 bytes, data type text (string).
3) **Type.** Let us assume that we have identified 203 different breeds of dog. The longest breed name is 28 characters long. If we allow that for each breed, there will be a lot of waste. For example, 'Poodle' only needs six bytes not 28. Because there are a **fixed number of choices**, we will code up the breeds. If we use one character, we can represent 26 breeds. If we use 2 characters, we can represent 26 x 26 = 676 breeds. This is more than enough. We could give Poodle the code PO, Alsatian AL and so on. We will therefore make this field data type text and allow 2 bytes.
4) **Date of birth.** This needs to be in the format DD/MM/YY. This is therefore data type Date and requires 6 bytes.
5) **Gender.** This is data type Boolean because a dog can only be either male or female. Allow 1 bit.
6) **Weight.** This is a real number. We only need 1 decimal place and the range of numbers is small therefore allow 2 bytes. (Real numbers will be represented using the floating-point system).
7) **Owner's telephone number.** This is data type text (because telephone numbers may include leading zeros and spaces). We will allow 6 bytes, to represent 12 digit numbers in BCD format.

We now have the following:

ID	Text	4 bytes
Name	Text	50 bytes
Type	Text	2 bytes
DofB	Date	6 bytes
Gender	Bool	1 bit
Weight	Real	2 bytes
Phone	BCD	6 bytes

Sizing a fixed-length file.



We need to remember when sizing a file that we are **estimating** the file size, not working out an exact size. Therefore, we can round numbers as we see fit. Let's now look at the five steps in estimating the file size.

STEP 1 - Decide on the fields.
Write down the fields you need in your database.

STEP 2 - Allocate data types and sizes.
For each field, state what data type it is and then state the number of bytes you are going to allow for that field.

STEP 3 - Work out the total size of one record.
The total size for one record is 4 + 50 + 2 + 6 +1bit + 2 + 6 = 70 bytes and 1 bit exactly, or 70 bytes approximately.

STEP 4 - Work out the maximum size the file could ever be.
Let's assume that the maximum number of dogs we are ever going to need to store is 5000. Therefore, the storage space required for the file will be up to 5000 x 70 = 350000 bytes.

STEP 5 - Work out the overheads.
When you store files, however, you don't just store the data! For example, information about when the file was created, last accessed, last changed and its size are stored. We refer to these items as '**overheads**'. We need to allow some extra storage for these overheads. The rule is to allow 10% of our estimated file size. The size of the overheads in this example can be worked out therefore as 10% of 350000, or 35000 bytes.

STEP 6 - Work out the total maximum file size, including overheads.
The total file size is now 350000 + (10% of 350000) = 350000 + 35000 = 385000 bytes

STEP 7 - Put the answer into appropriate units and round the answer up or down as appropriate.
Our final answer is 385000 bytes. The problem is that the units are not the best ones to select. We can do better than this. If we divide our answer by 1000, we will get the answer in Kbytes (Kilobytes) approximately. If we divide an answer by 1000000, we will get the answer in Mbytes (Megabytes). If we divide by 1000000000 we will get the answer in Gbytes (Gigabytes). The most sensible thing to do in this example is to divide by 1000. This gives us 385000 / 1000 = 385 Kbytes. The answer to this problem, then, is that the file size is approximately 400 Kbytes long. Don't be afraid to round up or round down numbers! Just remember to put 'approximately' with the answer. If you round an answer, it demonstrates that you understand what you are doing, that you are **estimating** and not working out exact numbers.

16.3 A summary of how to size a fixed-length file of records

1) Decide what fields you need.
2) Decide how many bytes you need for each field, justifying your decisions.
3) Add up the total numbers of bytes for 1 record. Round the answer if it helps.
4) Multiply the size of one record by the maximum number of records you are likely to store.
5) Add 10% for overheads.
6) Put your answer in appropriate units. Round the answer if it helps.
7) Put approximately next to your answer.

16.4 Variable-length records

Whilst fixed-length records are relatively easy to process, they can waste a lot of storage space on disk, especially when a short length data item of, for example, five bytes has been entered into a text field that has been defined as a long fixed length of perhaps 30 characters long. 25 bytes of valuable storage space are going to be wasted each time we do this. One way to address this problem is to use variable-length records. This system is **more complicated** than fixed-length systems because calculations have to be done but this system will **save storage space**. Suppose you had to store the following data:

First name: Fred Second name: Jones Name of course: Art	First name: Mandy Second name: Shu Name of course: English	First name: Ali Second name: Patel Name of course: Mathematics

Some records we want to store as variable-length records.

In a fixed-length system, you might have defined the following:

- First name: data type text, allow 30 characters.
- Second name: data type text, allow 30 characters.
- Name of course: data type text, allow 4 characters, coded.

If we stored Fred Jones who is on an Art course, we should actually need only 4+5+3 = 12 bytes but would in fact use 64 bytes. This is because all of the fields in the record are a fixed size. If we stored the records using a variable-length system, we could diagrammatically represent what we are storing as:

Fred$Jones$Art#MandyShuEnglish#Ali$Patel$Mathematics#

Note two points about this system:

1) The end of a field is signalled by a dollar sign (EndOfField marker).
2) A hash sign signals the end of a record (EndOfRecord marker).

Imagine a user wants to enter some data into their database. The user starts to enter data. They enter the first piece of data into the data input form where it says "First name". When the user presses ENTER and moves to the next data entry field, a dollar sign is automatically inserted after the field by the database software. The second name is then entered. As soon as the user presses ENTER again to go to the 'Name of course' field, another dollar sign is automatically entered after that field. After the user presses ENTER again (to enter the name of the course and to get a new blank data input form up on the screen) a hash symbol is entered. This signals the end of the record. The whole process is repeated for the next record, and the next, and so on.

Now there is no space wasted to store each record. If we stored Fred Jones who is on an Art course, we would need only 5+6+4 = 15 bytes (including two EndOfField markers and an EndOfRecord markers). These markers can easily be used by a program when reading records back from a file. All it has to do is keep looking for EndOfField markers and EndOfRecord markers. These will signal the end of a particular piece of data or a particular record.

16.5 Summary

- Variable-length records save space and produce a smaller overall file of records.
- A smaller file of records is quicker to work with (for example, to search and sort) than a larger file.
- Variable-length systems are more complex to program compared to fixed-record systems.
- It is easier to estimate the overall file size with fixed-length records than variable-length records. You have to make assumptions about the average size of fields, which may not be easy to do or correct.

Q1. What is the difference between an integer and a real number?
Q2. What is a string?
Q3. Describe with an example how to convert a number into BCD.
Q4. Define a field and a record.
Q5. Define a file and a database.
Q6. Explain what a primary key is.
Q7. What is an 'overhead'? Give examples to illustrate your answer.
Q8. Describe the seven steps needed to calculate the size of a file of fixed size records.
Q9. Explain how a file of variable length records is constructed.
Q10. State one advantage and one disadvantage of variable length records over fixed length records.

Chapter 17 - Arrays

17.1 Introduction

According to the BCS, an array is "a set of data items of the same type grouped together using a single identifier". A one-dimensional array can be thought of as a list of data. Two-dimensional arrays can be thought of as tables of data. Arrays are important structures in programming. They are set up, or initialised, by a programmer to force a computer to store data in blocks. When you write a program and then compile it, your computer (not you) decides exactly where in your RAM to put the executable code. If your computer has 64MB of RAM, then it has 64 x 1024 x 1024 memory addresses where it could store data. Your computer decides what actual memory addresses to use for the program and also for the data your program uses. The programmer doesn't! The programmer refers to memory addresses in a program in an indirect way, using variable names.

17.2 Why do we need arrays?

If you store 10 pieces of data, perhaps 10 exam results, for example, your computer decides which addresses to use to store the data in. It could decide to use 10 memory addresses that are spread out from each other. As far as the programmer is concerned, they don't know what actual addresses have been selected and they don't need to know. They will need to be able to fetch each of the pieces of data from time to time, however. They do this by giving each piece of data a name. If they then need to display the third exam result, for example, they might have an instruction in the program:

Write ("The third exam result was: ", Third_result)

Suppose you had a thousand pupils and you had to store their names. You would need a thousand variable names so that you can get back each one when you need it. You would also need variable names for each of their addresses, the dates of births and so on. You would need a variable name for each individual piece of data so that you can always get back each individual piece of data. Programs could end up with thousands of variable names. There is a better way! It might be better to group similar types of data together in a table (otherwise known as an **array**), giving the array a single name. Then, if you wanted any particular piece of data (an **element**), you can refer to the name of the array along with the co-ordinates of that element.

17.3 How do you initialise an array?

If you want your program to use an array, then you need to tell the program to use one! Every array in a program, and you may have several, must be initialised in the program itself. This forces the computer to group together the array data in a block of successive memory locations. A programmer sets up an array in a program by stating:

- The name (the identifier) of the array. (You might have more than one array.)
- The size of the array
- The data type the array will hold - what kind of data will be stored in the array.

17.4 Addressing elements in a one-dimensional array

If you have an array, then you refer to a particular place in the array by using the array's identifier (its name) followed by an 'address'. For example, suppose we define a one-dimensional array called SCORES that can hold up to 10 integers, and then fill it with some numbers. You would use something like this:

SCORES: array[1..10] of integer

This sets up an array called SCORES that is 10 elements (memory locations) long, each location holding an integer.

88
65
87
67
90
39
50
68
71
34

You would refer to the score held in the n^{th} position from the top of the array by using the name of the array and stating the position.

For example, you would refer to the score held in the sixth position using SCORES [6]. SCORES [6] would return the value 39, assuming that the first element was SCORE [1]. Many languages start counting at zero, so the first element in that case would be SCORE [0] and 39 would therefore be in SCORE [5].

17.5 Addressing elements in a two-dimensional array

Suppose you now define a two-dimensional array called RECORD_OF_SCORES. It stores 10 pupils' last three scores. You would use something like this:

RECORD_OF_SCORES : array[1..10,1..3] of integer

This sets up an integer array called RECORD_OF_SCORES, which is 10 rows by 3 columns, as shown here:

67	78	97
80	75	65
76	67	65
67	68	52
65	56	65
60	59	72
70	80	54
60	57	77
80	76	67
56	34	34

You would refer to the score held in any position from the top left hand corner of the array by using the name of the array and stating the row followed by the column for that element.

For example, the 8th pupil's 3rd score (held in row 8 column 3) would be referred to using RECORD_OF_SCORES [8, 3]. This would return the value 77.

17.6 Some good and bad points about arrays

The following points can be made:

- Arrays are very **straightforward** to set up in a programming language. You simply declare that you are going to use one by giving the name of the array, its size and the type of data it will store. This is an advantage of an array compared to some other data structures that are more complex to define and use.
- As has already been said above, when a programmer sets up an array, they have to state the size of the array. That means they must estimate the size. But what if they make the maximum size too small? Some data will get lost. What if they estimate too big a size? This would waste some space in memory. Having to **define the size in advance** is a disadvantage.
- Suppose you define a one-dimensional array that is 1000 elements long. When you compile it, a thousand places will be reserved in memory for the array. Now suppose it you leave it empty! You have reserved all of this space, but not used it. Another disadvantage of arrays is that they can lead to **a waste of RAM**.
- Turning the above disadvantage around, however, you can argue that because the space for an array is defined when a program is compiled, it means that you will always have the space available to store data. Some data structures grow and shrink as you store and remove data. There is no guarantee, however, that space will be available for new data if one of these dynamic data structures is used. Compared to these 'dynamic' data structures, arrays' structure is fixed and reserved in advance and having this **guaranteed storage space** is an advantage. We refer to arrays as 'static data structures' for this reason.
- With some data structures, you cannot get to a data item without going through other data items. Think of a music tape for a moment. You can only get to a particular song in the middle of the tape by winding through all of the ones at the front first! The good news with an array is that you don't have to go through other data items to get to a particular one. You can access a particular 'element' directly. This kind of access is known as **direct access** and it allows fast retrieval of data. Not all data structures allow fast access to data, as we will see.
- It is **more efficient** for a programmer to store data together in a block if the data has to be stepped through and worked on. An index can be set up that points to the beginning of an array, and then it can simply be incremented within a loop. Each data item in the array can then be processed in each pass of the loop.
- If an array stores some data in an organised way, perhaps storing names of pupils alphabetically, then **inserting and deleting items can become a little complicated** compared to other data structures. For example, if you added Jones to an alphabetical list of pupils, then to keep the alphabetical order correct, you would have to change the array locations of pupils above Jones - you would have to move them up to make space for Jones!

17.7 Languages that do not use arrays

Some languages do not use the structure or term 'array'. They use data structures that are similar but different to arrays and are called different names.

For example, Python doesn't use arrays as a main data structure but uses tuples, lists and dictionaries instead. (It is actually possible to use an array in Python by importing a specialist library so arrays which have classic array characteristics can be employed in Python). A list in Python, for example, is similar to an array and can be used in similar ways but you can hold different datatypes in a Python list whereas you have to specify what datatype an array will hold when you declare it. This is because an array can hold only one type of data. In Python, you can also just use a list. You don't have to declare it first.

In Python, you can have a list of lists, which can be manipulated in the same way as a 2 dimensional array in a language that uses arrays. Here is an example in Python that sets up a list of lists, prints out the elements, then sorts the list and finally, prints out a particular part of one of the elements in myList.

```
myList = [["Dave", 32],["Mary", 45],["John", 9],["Fred", 55]] #create the list.

#print the whole list. There are 4 elements in myList and each element is a list.
print(myList[:])

for item in myList: #print the list element by element.
   print(item)

myList.sort() #sort the list and print it out again.
   print(myList[:])

#print the first part of the third element.
#remember - Python always starts counting from zero.
print(myList[2][0])

input("Press <ENTER> to quit >>> ")
```

Q1. Define an array.
Q2. Why do all arrays need a name?
Q3. How many different datatypes can an array hold?
Q4. What is meant by a one dimensional array?
Q5. What is meant by a two dimensional array?
Q6. What is an 'element' when referring to arrays?
Q7. Using an example, show how you would read and write to an element in a one dimensional array.
Q8. Using an example, show how you would read and write to an element in a two dimensional array.
Q9. Describe one good point and one bad point about an array.
Q10. State a language that doesn't use arrays and state what data structures it typically uses instead.

Chapter 18 - Linked lists

18.1 Introduction

A linked list is another kind of structure used to store data. However, this kind of structure is known as a **dynamic data structure**. It is 'dynamic' because this data structure changes its size, unlike arrays, which do not change their size - arrays are static structures. As you store more data in a linked list, the structure expands to take it. As data is removed, it shrinks.

When the programmer wants to use an array, they set it up in the program using keywords reserved for this purpose. Remember, a programmer sets up an array in their program but they have no idea what actual memory addresses the computer will put the data in when the program is translated. They don't need to know - to refer to a particular location in the array they simply refer to the name of the array along with some co-ordinates. It is a similar situation with linked lists. The programmer uses keywords in the program to initialise a linked list. They then use keywords to create 'nodes' to store data.

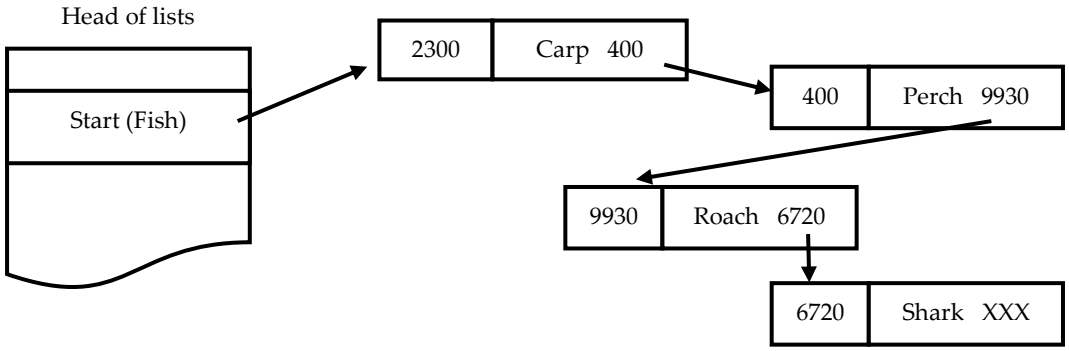
18.2 How linked lists work

Read this description and study the diagram. A programmer wants to store a list of fish alphabetically in a linked list data structure. We want to store Carp, Perch, Roach and Shark. Below is an illustration of how the computer works with a linked list.

The programmer will initialise the data structure in the program (in pseudo-code below) but they will have no idea what actual memory locations are selected by the computer when the program is translated. For illustration purposes, however, we will select a range of memory addresses. We will select a range of memory addresses only to illustrate that, in a linked list, the individual pieces of data can be all over the available RAM!

- The computer will store the beginning of the list in an area of RAM we will call the Head of Lists.
- We will call the memory location that holds where to find the first piece of data the START(FISH).
- If the computer looks at the contents of the START memory location, it will show it the memory location of the first **node**. A node is a piece of data plus the address of the next node. It's memory location 2300!
- The computer can then go to memory location 2300, and it will find the data item Carp!
- Not only that, however. It will also find the memory location of the next piece of data, 400.
- If the computer goes to memory location 400, it will find the next **node**; it will get the next piece of data, Perch, as well as the memory location of the next piece of data, 9930, and so on.
- At some point the computer will get to the end of the list. When it gets to the last piece of data, the pointer at that node is given a null value - it is given a value that is not a valid memory address e.g. XXX

If you are having trouble working out what is going on, then a diagram might help!

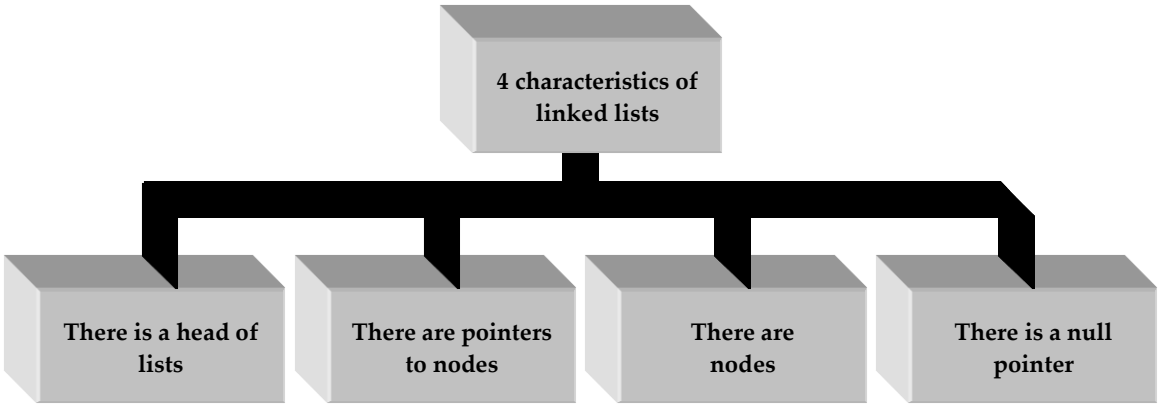


A diagram of a linked list.

A diagram showing a linked list is characterised by 4 things:

1) The head of the list. (This points to the first node).
2) Pointers to nodes. (They either point at the next place to look for a node or contain a null value).
3) Nodes (A node is a memory location that contains a piece of data and another memory location).
4) A null pointer. (This indicates the end of the list. We have used XXX).

Note also that linked lists can make use of any spare memory addresses going. If, for example, the computer had two applications running in RAM, separated by a couple of empty RAM addresses, it could make use of that little bit of memory in a linked list. Linked lists can help a computer ensure that memory is used wisely!



18.3 Algorithms for linked lists

When you store sets of data, whether the data structure is an array, a linked list or any other structure, there will be times when you need to do things with it! For example, you might want to:

- Search for a data item.
- Add a data item.
- Delete a data item.

It would be useful to write algorithms to describe how we could do the above. We have already seen that an algorithm is simply a set of step-by-step instructions that describe how to do a particular task. Algorithms can be written in semi-English, or pseudo-code! When writing algorithms, you should concentrate on getting the logic correct and dealing with any special cases.

18.3.1 Searching for a data item

To search for a data item ALPHA in a linked list, we could write out a description.

> **Start by getting the pointer in the Head of Lists. Then we need to test the pointer to see if it is the null pointer – it might be an empty list. If it is, we need to report that the list is empty and stop. If it isn't, we need to follow the pointer to the first node. (Remember, a node contains a data item and another pointer). We need to look at the data item at that node. Is it the one that we are looking for? If it is, then we can report that we have found the data item and stop. If it isn't we need to get the pointer at the node. We then check it because if that pointer is the null pointer we need to report that the data item we are looking for is not in the list and stop. If it isn't the null pointer, we need to go to the node pointed at by this pointer and repeat. We keep doing this until either we find the data item or we come to the end of the list.**

There is nothing wrong with the above description, although it is not so easy to code up into a programming language because there is no indication of programming structure in the description. It is written as an English paragraph. If we use pseudo-code, however, we can use programming structures, which makes converting the design into real code easy! On top of that, pseudo-code is language-independent. That means a programmer expert in *any* language should easily be able to convert pseudo-code into their particular language. With pseudo-code, we can make up any keywords we like so long as they broadly describe what we want to do!! An algorithm written in pseudo-code to search for an item in a list might look like the following. As you study the algorithm, there are a number of things to note about it:

- Some Boolean variables have been used to store whether some events have happened or not.
- All IF and WHILE statements have a corresponding ENDIF or ENDWHILE statement to aid clarity.
- Programming constructs were used to aid the step from pseudo-code to writing in actual code.
- Indentation, meaningful variable names and comments help the reader follow the logic of the code.
- Special cases e.g. if the list is empty in the first place, if data isn't present in the list.

```
BEGIN
FOUND=FALSE  //2 Boolean variables, to hold whether the data has been found and to indicate the end of list.
NULLPOINTER=FALSE
GET POINTER to first node from Head of Lists

IF POINTER=XXX          //test to see if the list is empty i.e. the head of list contains the null pointer.
        NULLPOINTER=TRUE
        REPORT "LIST EMPTY"
ELSE    //if there are data items in the list do this loop until you either find the data or get to the end of the list.

BEGIN
        WHILE (NULLPOINTER=FALSE) AND (FOUND=false) DO
        BEGIN
                GET (data at node) pointed to by POINTER
                IF data=ALPHA
                        FOUND=TRUE
                        REPORT "Data found"
                ELSE
                        Get (pointer at node) pointed to by POINTER
                        IF (pointer at node) =XXX
                                NULLPOINTER=TRUE
                                REPORT "Data item not found in list"
                        ENDIF
                ENDIF
        ENDWHILE
ENDIF
END.
```

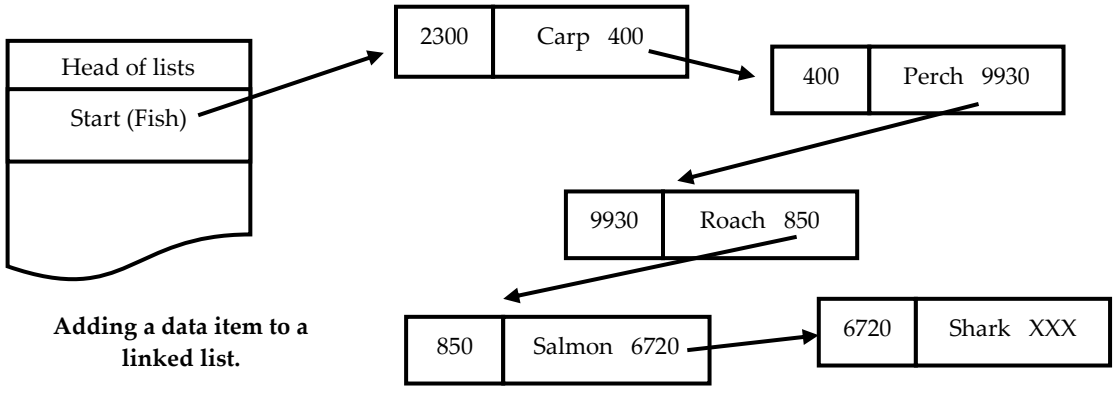
An algorithm written in pseudo-code to search for an item.

18.3.2 Adding a data item

The following algorithm describes how to add a new data item BETA to a linked list.

```
BEGIN
PUT BETA in a memory location
GET POINTER to first node from Head of Lists
IF POINTER=Null THEN
        SET Head of Lists point to BETA's memory location
        SET BETA's pointer to Null and finish
ELSE
        FOLLOW pointers until correct position for data located
        ADJUST pointers as necessary and finish
ENDIF
END.
```

For example, suppose the computer needed to add Salmon to the linked list of Fish. The computer inserts Salmon in memory location 850. A diagrammatic representation of the result would look like this:



Adding a data item to a linked list.

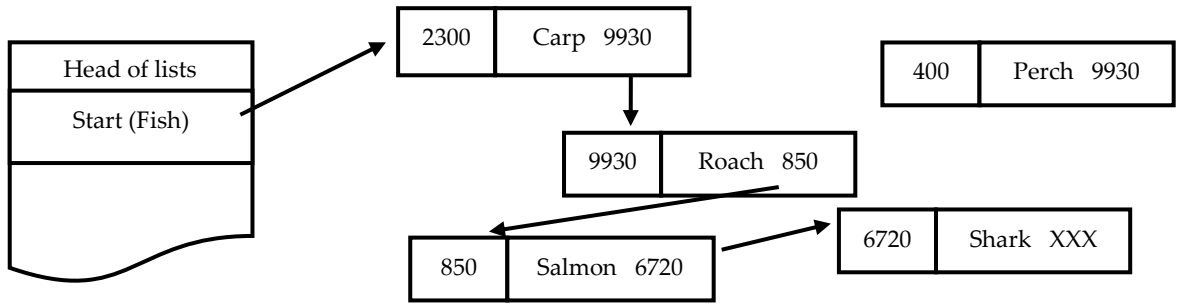
Notice how the pointer at Roach's node has been changed to point to Salmon's location, and the pointer for Salmon has been set to the memory location that was pointed to by Roach's node.

18.3.3 Deleting a data item

The following is an algorithm for deleting an item in a linked list.

BEGIN
GET pointer to first node from Head of Lists
FOLLOW pointers until the data to be deleted is found
ADJUST pointers as necessary
END.

For example, suppose the computer needed to delete Perch from the linked list of Fish. This diagram describes the process.



Deleting a data item from a linked list.

Notice that Perch is still in memory location 400. It's just that no pointer points to it anymore so it is not part of the linked list. The computer will keep a record of memory locations that have been freed up, and may at some point overwrite the contents of those memory locations when they are needed again. Also notice that the pointer in the node Carp has now changed. Some further points about linked lists.

- Inserting data and deleting data is straightforward compared to arrays. If you are storing data in an array in an organised manner and you want to insert or delete an item, then you need to actually move the data items in the array to new locations to maintain the proper order. You don't have to do this with linked lists. You just find the appropriate place and then adjust appropriate pointers.
- To find a piece of data in a linked list, you have to traverse through all the nodes in a serial fashion. This can be **slow** compared to a direct access data structure, such as an array.
- Linked lists grow and shrink depending on how much data is stored. There are no blocks of memory that have been reserved in RAM as there are with arrays. Linked lists therefore use RAM more efficiently.
- With arrays, you reserve space when the program is compiled. This doesn't happen with linked lists. It's possible you might not be able to store what you want to with a linked list data structure!

Q1. Why are linked lists described as 'dynamic' data structures and arrays 'static data structures'?
Q2. What is a 'node' in a linked list?
Q3. What does the 'Head of lists' hold?
Q4. What does a pointer do in a linked list?
Q5. What is a null pointer?
Q6. Describe a special case that should be tested for when searching for data in a linked list.
Q7. Describe an algorithm for finding a piece of data in a linked list.
Q8. Describe an algorithm for adding a piece of data to a linked list.
Q9. Describe an algorithm for deleting an item from a linked list.
Q10. What is meant by 'serial access to data' in a linked list?

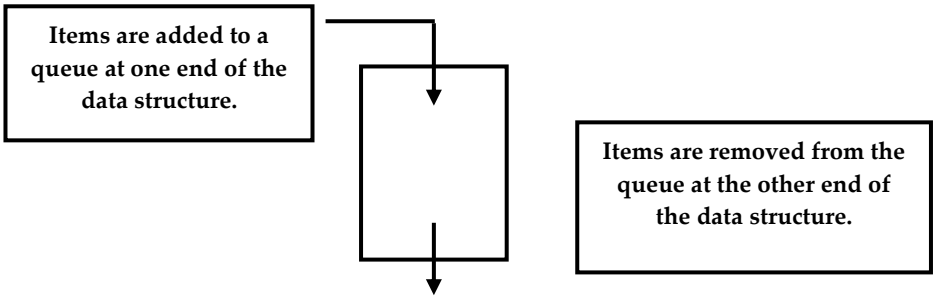
Chapter 19 - Queues and stacks

19.1 Introduction

So far, we have looked at one static data structure (arrays) and one dynamic data structure (Linked Lists). They are not the only data structures available. We need to look at two more, the queue and the stack. They have their own particular way of working and each can make certain data handling tasks more efficient.

19.2 How a queue works

Consider a supermarket. The first person to arrive at a checkout is first in the queue! Then a second person arrives and they are second in the queue. Then a third person arrives and they are third in the queue and so on. Of course, people are also leaving the queue as well. When someone leaves, they leave from the front of the queue. The first person into the queue is the first person out of the queue. This kind of data structure is known as a FIFO, or First In First Out structure.



A diagram showing how a queue works.

If the first person in the queue leaves, then the front of the queue is now at the second person! You can find out where the front of the queue is by using a pointer to the front of the queue. You can also keep track of where the back of the queue is by using a pointer there, too.

Queue structures are very useful in computing. They are dynamic data structures, growing and shrinking in size as necessary. They might typically be used, for example, to store the order of jobs sent to a printer. They could be used to order the calls made to a call centre so they are answered fairly. They can be used to store MP3 songs so they are played in order.

19.3 An example of a queue

We will set up an array with 6 elements. Within this array, we will set up our queue. We will have a pointer called FRONT to point to the memory location where the front of the queue can be found, and a pointer called END, to point to the END of the queue. These will in fact simply be variables called FRONT and END. Note that the queue itself is a dynamic data structure because it grows and shrinks in size, but it is confined within an array - a static data structure, a fixed size! This may seem a contradiction because you are losing all the benefits of a dynamic structure by placing it within a static structure. However, it is the processing of the data itself in a queue that is most important so we will conveniently 'overlook' this anomaly for now! We will visualise the queue with the front of the queue at the bottom of the following diagrams.

Step 1: The queue is empty. FRONT=0, END=0

6	
5	
4	
3	
2	
1	

Step 2: John joins the queue, then Zubair comes along and joins the queue. Finally, Mandy joins the queue. FRONT=1, END=3

6	
5	
4	
3	**Mandy**
2	**Zubair**
1	**John**

Step 3: Sam joins the queue (at END+1 memory location) and John leaves the queue. FRONT=2, END=4

6	
5	
4	**Sam**
3	**Mandy**
2	**Zubair**
1	

Step 4: Larry joins the queue and Zubair then Mandy leave the queue. FRONT=4, END=5

6	
5	**Larry**
4	**Sam**
3	
2	
1	

Step 5: Fred joins the queue. FRONT=4, END=6

6	**Fred**
5	**Larry**
4	**Sam**
3	
2	
1	

When the END pointer points to the very end of the array, there appears to be a problem - how can we add another piece of data, Emma? There is free space in the array, just not at the end! Actually, it presents no problem! When the END pointer points to the end of the array AND the FRONT pointer isn't at the front of the array, it means there is space. We put the new data in to the front of the array, and adjust the END pointer accordingly. It doesn't matter that the END is in front of the FRONT pointer!

Step 6: Emma joins the queue. FRONT=4, END=1

6	**Fred**
5	**Larry**
4	**Sam**
3	
2	
1	**Emma**

Step 7: Sam leaves the queue and Ali joins the queue. FRONT=5, END=2

6	**Fred**
5	**Larry**
4	
3	
2	**Ali**
1	**Emma**

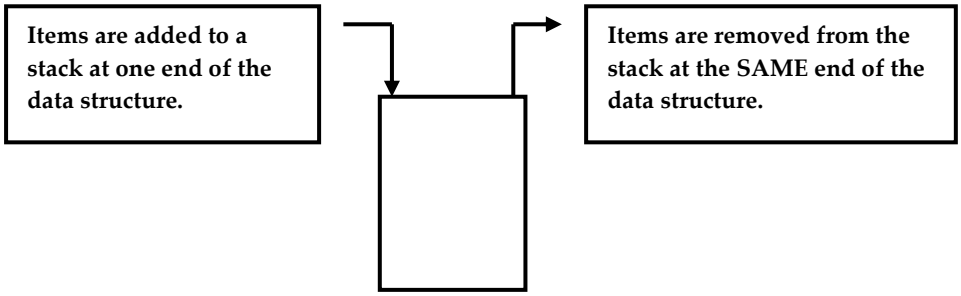
This kind of queue is known as a 'circular queue' because the FRONT and END pointers go around in circles! It ensures that free space made available at the front of the queue when a data item is removed can be re-used.

19.4 A classic application of a queue structure

A classic application of a queue structure is in the management of print jobs on a network. If 6 people on a network all send their work to be printed, their print jobs are intercepted in the order they are sent by a special program called a 'spooler'. This program's function is to store each of the data in each print job in a queue. It then sends each job to the printer in a 'First In First Out' manner. (Using a spooler program ensures that a user can use their computer quickly after they have sent a file to be printed because the spooler is in charge of printing rather than the individual's computer).

19.5 How the stack works

Stacks are another example of a dynamic data structure, and like queues, they can be implemented within an array. Unlike queues, however, data is put into the structure and removed from the structure at the **same** end of the structure.



A diagram showing how a stack works.

Like queues, the data cannot be accessed directly in a stack - you cannot go straight to the middle of a set of data items and retrieve it. You have to remove other data items first. You can think of a stack as lots of CDs stored on a spindle. You put CDs on the spindle, and to remove them you take them off in reverse order. In fact, this kind of device is known as a Last In First Out (LIFO) device because unlike a queue, the last item put into the stack is the first item to be removed. To implement a stack in an array, you need a TOP pointer to point to the top of the stack. An empty stack has TOP = 0. When you put a data item on to the stack, you talk about it being 'pushed' onto the stack. When you remove an item, you say it is 'popped' from the stack. We will use an array that can hold up to 6 data items as before.

Step 1: The stack is empty. TOP = 0

6	
5	
4	
3	
2	
1	

Step 2: Firstly, The data item Sheep is pushed onto the stack. Then Cow is pushed onto the stack. TOP = 2

6	
5	
4	
3	
2	**Cow**
1	**Sheep**

Step 3: The data item Goat is then pushed onto the stack. Then Chicken is pushed onto the stack. TOP = 4

6	
5	
4	**Chicken**
3	**Goat**
2	**Cow**
1	**Sheep**

Step 4: The data item Chicken is then popped from the stack. TOP = 3

6	
5	
4	
3	**Goat**
2	**Cow**
1	**Sheep**

Step 5: The data item Horse is pushed onto the stack. Then Duck is pushed onto the stack. TOP = 5

6	
5	**Duck**
4	**Horse**
3	**Goat**
2	**Cow**
1	**Sheep**

Step 6: We want to now remove three items. Remember that items are removed from the stack in the reverse order that they were put onto the stack. So Duck comes off first, then Horse, and finally Goat. TOP = 2

6	
5	
4	
3	
2	**Cow**
1	**Sheep**

Of course, if the stack becomes full, you can't store any more data in it! If you tried to store another data item in a structure that was full you would get an overflow error message. An overflow happens when you try to store data in a data structure that is full. It can also be used to describe the error that occurs when you try to store a number that is too big for the space reserved for it. For example, if you try to store 452 in one byte you will get an overflow error message because the maximum number that can be stored in one byte is 255.

19.6 Algorithms for adding and removing items to and from a stack

So far, we have simply added and removed items from our queue and stack. If we wanted to use a programming language to actually program these operations, we would first want to describe what needs to happen using an algorithm. We will use some pseudo-code to describe how to push an item onto a stack, and how to remove an item from the stack.

19.6.1 Algorithm for pushing an item on to the stack

```
Check if the stack is full (pointer = maxSize)
IF the stack is full, THEN
        REPORT full and STOP
ELSE
        INCREMENT pointer
        INSERT Data into position pointed to by the pointer
ENDIF
STOP
```

19.6.2 Algorithm for popping an item from the stack

```
Check if stack is empty (pointer=0)
IF empty, THEN
        REPORT empty and STOP
ELSE
        Temp := data pointed to by pointer {Temp is a variable to hold the popped data}
        DECREMENT pointer
ENDIF
STOP
```

19.7 The use of the stack when an interrupt happens

A CPU might be working on a program when an interrupt happens. An interrupt is a signal from a piece of software or hardware to the CPU that tells it that it needs some processing time. For example, if a DVD drive has a problem, an interrupt will be sent to the CPU to say "There is a problem - can you run some software to deal with the problem?"

The CPU, then, is in the middle of something when it receives an interrupt. If it decides to deal with this interrupt immediately, it needs to stop processing its current program (a CPU can only do one thing at a time) but does need to remember where it left off. This is so that it can return to the correct place in the program after it has serviced the interrupt. The way it does this is to push all of its important information (held in 'registers') onto the stack. It then jumps to the interrupt handling routine and runs the software to deal with the interrupt. When it has finished dealing with the interrupt, it pops all of the information it stored in the stack back into the CPU's registers, and the CPU carries on from where it was before.

19.8 The use of a stack to reverse a queue

All new items join the queue from the back.

Imagine you had a queue with these elements in!

Front of queue.

Banana	Orange	Apple	Coconut	Kiwi fruit	Mango

Mango is at the front of the queue and banana is at the back. In a queue, new items must join the back of the queue and must leave from the front (like a supermarket queue). Suppose you wanted to reverse the order of the queue, so that banana was at the front and mango at the back. You could use a stack to do this.

1) Push the items onto the stack. Start with the front of the queue, mango, then kiwi fruit etc until you push banana. This gives you a stack that looks like this:

Banana
Orange
Apple
Coconut
Kiwi fruit
Mango

2) Now pop all of the items out of the stack back into the queue again! Banana is popped first and is now at the head of the queue. Then orange is popped and that joins the queue at the back, behind banana. Then apple, coconut, kiwi fruit and mango are popped in turn. The queue now looks like this:

All new items join the queue from the back.

Front of queue.

Mango	Kiwi fruit	Coconut	Apple	Orange	Banana

We have now completely reversed the order of the queue.

19.9 The stack to carry out calculations

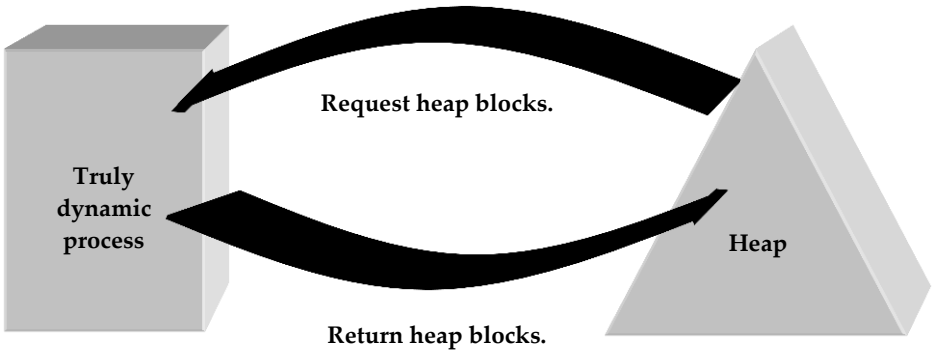
Stacks are also used to carry out calculations on numbers. If you look up 'reverse Polish notation', you can get examples of how the stack is used for this purpose.

19.10 The heap

It is possible to have completely dynamic data structures without placing them inside an array, although it is often convenient to use an array if it is the processing on the data that is important and an idea of the amount of data that will be processed is predictable.

To use a data structure completely dynamically requires the structure to use an area of memory known as the **heap**. This is an area in RAM that has been reserved for processes that don't quite know how much RAM they will need during their processing, as is the case with truly dynamic data structures. The heap is available for any process to use. It is a lot quicker for a process to access some RAM from the heap than it is for it to ask the operating system for some RAM each time it needs some.

If a process needs a block of RAM from the heap then it will request a **heap block**. When a structure has finished with some heap blocks, they can be returned to the heap, ready for use by another process.



Using the heap.

Q1. State a LIFO data structure.
Q2. State a FIFO data structure.
Q3. Are stacks and queues dynamic or static data structures?
Q4. State two uses for a queue.
Q5. State two uses for a stack.
Q6. What does 'pushing' a data item onto a stack mean?
Q7. What does 'popping' a data item from a stack mean?
Q8. Write an algorithm for adding a new data item to a stack.
Q9. What is RAM used for?
Q10. What is a 'heap'?

Chapter 20 - Binary trees

20.1 Tree structures

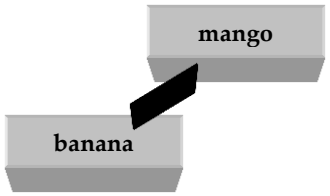
Another dynamic data structure is known as a 'tree structure'. Data items are organised according to some rules. **Once in this structure, the data can then be sorted.** Suppose, for example, we wanted to organise the following list of fruit into a **binary tree** structure so that they can then be sorted into alphabetical order.

mango, banana, pineapple, apple, coconut, pear, grape, strawberry, raspberry.

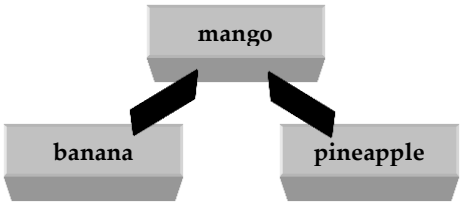
How would we go about this?

Step 1. The tree doesn't exist when we want to 'insert' the first item, mango, into the tree. The first item is therefore placed at the top of the diagram. This position, or node, is known as the 'root node'.

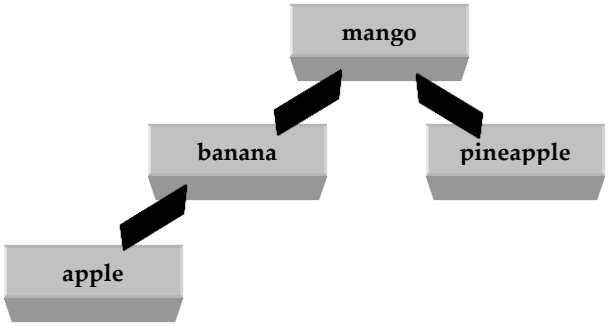
Step 2. Alphabetically, the b of banana is less than the m of mango, so we branch left. In binary trees, if a piece of data is less than the current node, we will branch left. If it is greater than the node, we will branch right. (Note that we could reverse the left and right branching if we wanted to. We would get the same tree but as a mirror image). A node will only ever have a maximum of two branches underneath it in a binary tree, one branching left and the other branching right. (You can have trees that are not binary, but that is beyond the scope of our discussion).



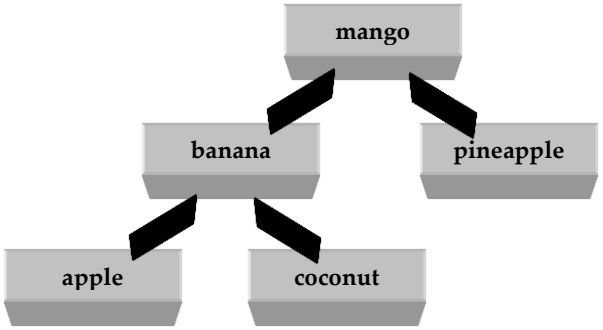
Step 3. The p of pineapple is greater than the m of mango, so we branch right.



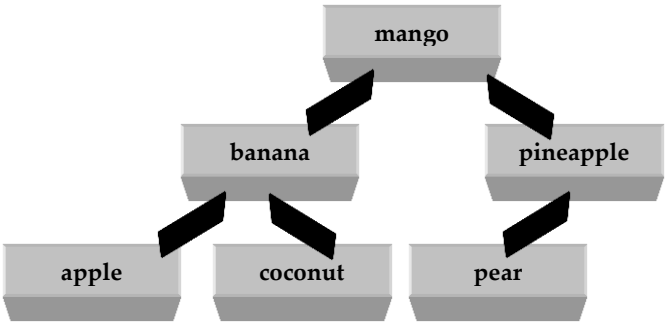
Step 4. The a of apple is less than m of mango so we branch left. Then it is less than the b of banana so we branch left again. Now there is no node in that position so our new data goes in that position. Apple goes under banana and to the left.



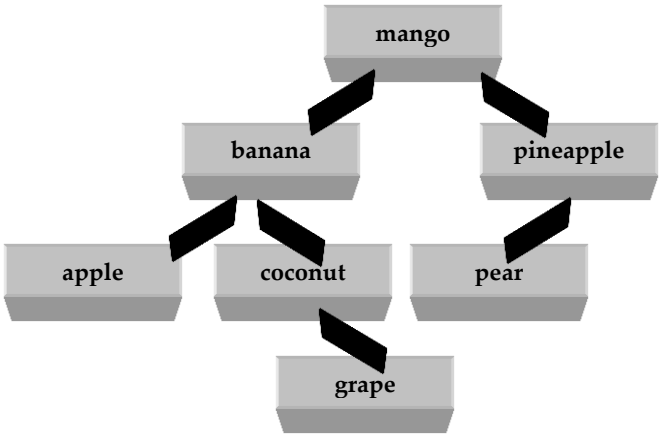
Step 5. The c of coconut is less than mango so branch left. Coconut is greater than banana so branch right.



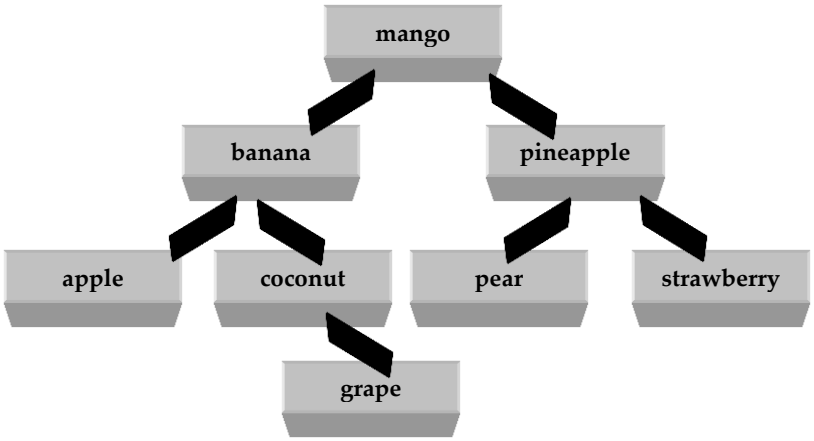
Step 6. Pear is alphabetically greater than mango, so branch right. It is less than pineapple so branch left.



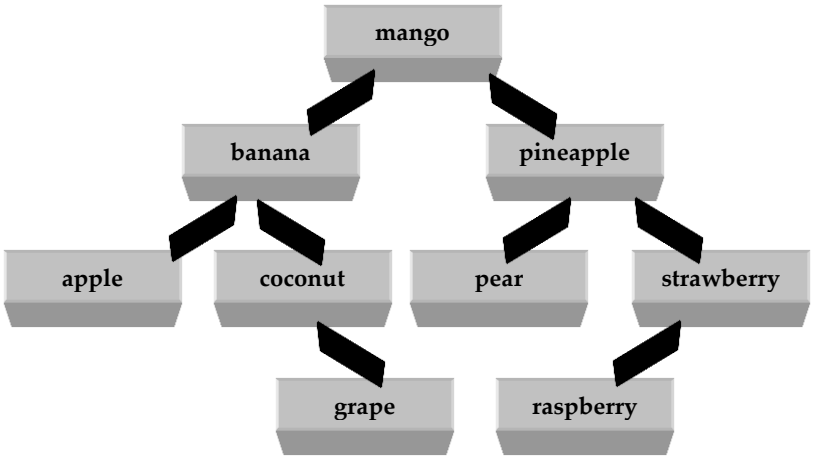
Step 7. Grape is less than mango so branch left. It is greater than banana, so branch right. It is greater than coconut so branch right again.



Step 8. Strawberry is greater than mango so branch right. It is also greater than pineapple, so branch right again.



Step 9. Raspberry is greater than mango, so branch right. It is greater than pineapple, so branch right again and it is less than strawberry, so now branch left.



Mango is known as the '**root node'**. The nodes with no nodes underneath them are '**terminal nodes'**. The terminal nodes are apple, grape, pear and raspberry. A tree that can have only two nodes under each node is known as a '**binary tree'**.

20.2 Insertion algorithm

To inset an item into a tree structure, we follow this algorithm for each item in turn. The CURRENT NODE is the position we are currently at. The DATA ITEM is the new item we want to insert into the tree.

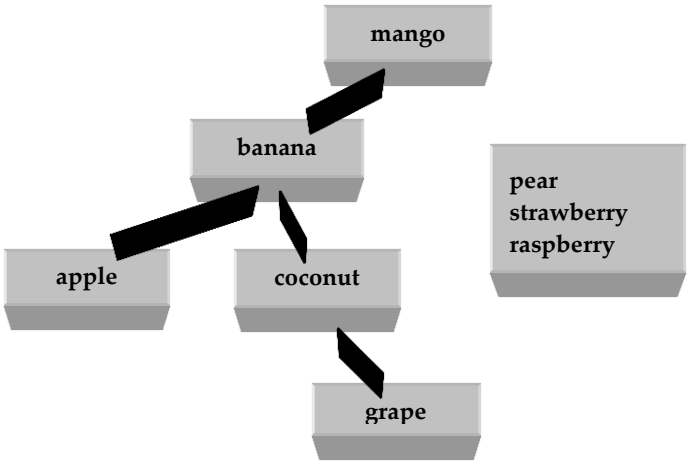
If the tree doesn't exist yet, make the DATA ITEM = ROOT NODE and finish.
Let CURRENT NODE = ROOT NODE.
Repeat until CURRENT NODE is null.
    If DATA ITEM is less than CURRENT NODE, go to the left.
    Else go to the right.
    CURRENT NODE = value of node reached so far. (Node reached so far = null if there is no node at that position).
Create new node and add DATA ITEM to that position.

20.3 Deletion algorithm

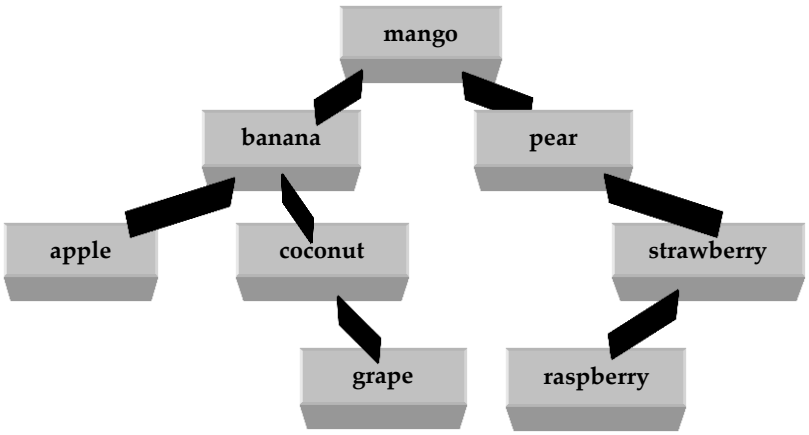
Deleting items from a tree structure is quite complicated. There are a number of ways to do it. Here's one way:

- Traverse the tree until you find the item that you want to delete.
- Call that item the root node.
- Copy all of the items underneath the root node to another data structure, e.g. using a stack.
- Delete the root node.
- Use the insertion algorithm to re-enter each data item in turn from the stack into the tree.

Suppose we wanted to delete 'pineapple' from the fruit binary tree we used above. We find pineapple, copy all the items underneath it to a stack and then delete the pineapple node. The tree and stack data structures would look like this.



We then pop the data items from the stack and use the insertion algorithm to re-enter data into the tress structure. The tree will then look like this.



20.4 Amending data

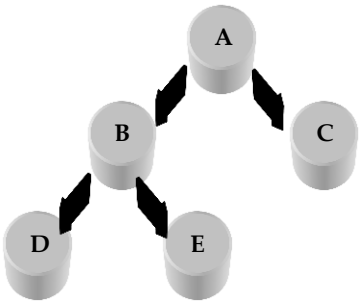
Trees are used because they maintain an order between a number of data items. Amending the data would result in the order of the data in the tree changing. Algorithms for amending the data in trees are therefore beyond the scope of this book.

20.5 Traversing binary trees

When we want to visit a binary tree and look at its nodes, there are three different ways we can do this. These are known as

1) IN-ORDER
2) PRE-ORDER
3) POST-ORDER

Consider the following binary tree.



If you look at this tree, you should notice that there *may* be a tree structure under every node. For example, look at the node B. There is a tree structure with B as the root node and nodes underneath it. We can say that a tree may be made up of sub-trees and we can use this property (recursion) to traverse the tree in different ways.

20.5.1 Traversing a tree: IN-ORDER

Using this method, we must visit the tree in this order:

1) Visit the left sub-tree.
2) Visit the root node.
3) Visit the right sub-tree.

How does this work? We visit the **left sub-tree**, then the **node**, then the **right sub-tree**.

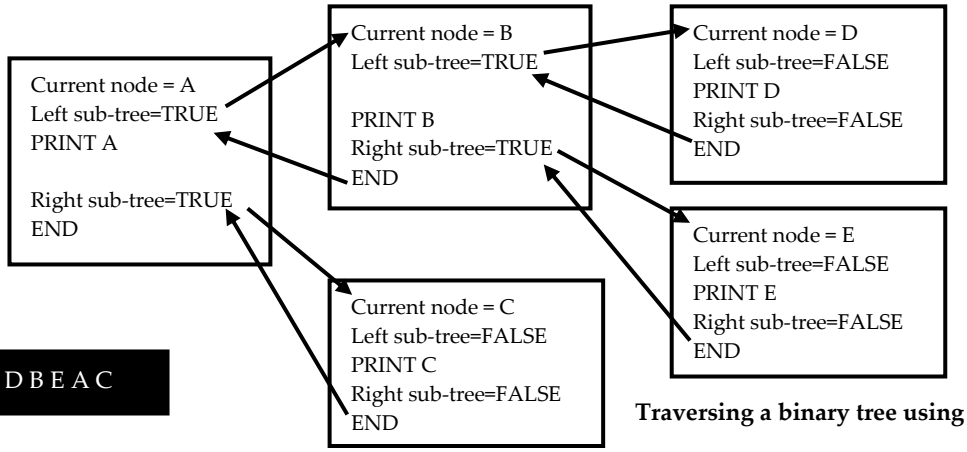
- We start at the root, node A.
- Underneath node A is the left sub-tree (with root node B) and the right sub-tree (with root node C).
- We must check the left sub-tree first according to our INORDER rules. Move to B.
- But B has a left sub-tree (with a root at D) and a right sub-tree (with a root at E).
- We must check the left sub-tree first according to our INORDER rules. Move to D.
- D does not have a left sub-tree, so visit the node D.
- Now check for D's right sub-tree. It doesn't have one.
- We have now done the left sub-tree for the tree that has a root node at B. Now visit node B.

- Now visit the right sub-tree of B. We move to E.
- E doesn't have a left sub-tree so visit E.
- E doesn't have a right sub-tree so move to B and because we have now completely visited the tree with the root node at B, we move up to node A. Visit node A.
- Now visit the right sub-tree of A. We move to C.
- C doesn't have a left sub-tree so visit C.
- C doesn't have a right sub-tree so move back up to A.
- We have now visited every node.

The order that we visited the nodes was **DBEAC.** We can write an algorithm to print out all of the data at the nodes, like this:

1) For the current node, check if there is a left sub-tree. If there is, go to the root node for this sub-tree and then go to 2). If there isn't, go to 3).
2) Repeat 1).
3) Print the current node.
4) For the current node, check whether it has a right sub-tree. If it has, go to 5) else go to 6).
5) Repeat 1).
6) END

This algorithm will take a little bit of thinking about because it is a **recursive algorithm**. Refer back to chapter 10 on recursion. To understand this algorithm, you need to draw a box for each recursive call as we did in chapter 10.



OUTPUT SCREEN: D B E A C

Traversing a binary tree using IN-ORDER.

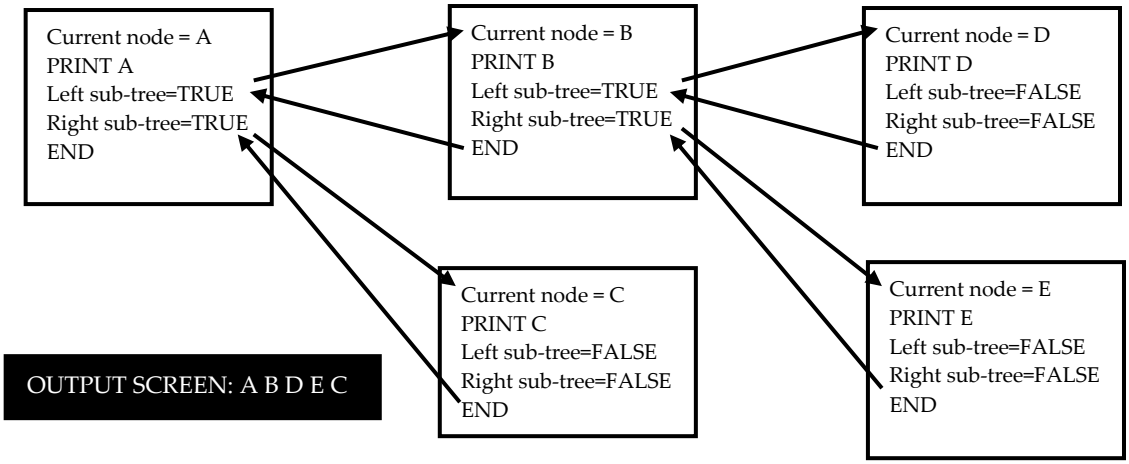
20.5.2 Traversing a tree: PRE-ORDER

Using this method, we need to

1) Visit the root node.
2) Visit the left sub-tree.
3) Visit the right sub-tree.

Using our previous binary tree, we would visit the nodes in the order: **A B D E C**. We can write an algorithm that would print out all of the data at the nodes, like this:

1) Print the current node.
2) For the current node, check if there is a left sub-tree. If there is, go to the root node for this sub-tree and then go to 1). If there isn't, go to 3).
3) For the current node, check whether it has a right sub-tree. If it has go to 4) else go to 5).
4) Repeat 1).
5) END.



Traversing a binary tree using PRE-ORDER.

20.5.3 Traversing a tree: POST-ORDER

Using this method, we need to

1) Visit the left sub-tree.
2) Visit the right sub-tree.
3) Visit the root node.

Using our example binary tree, the order that we would visit is: D E B C A

Q1. Put the following list of names into a binary tree: Mandy, John, Ali, Susan, Robert, Emma, Tim, William, Tom.
Q2. Write down the algorithm for inserting a data item into a binary tree.
Q3. Using the algorithm for insertion, insert the name 'Peter' to your binary tree from Q1 above.
Q4. Put the following primary keys into a binary tree: 50, 25, 74, 90, 94, 80, 17, 35.
Q5. Describe how you would delete an item from a binary tree.
Q6. In what sense is a binry tree 'binary'?
Q7. State three methods of traversing a binary tree.
Q8. What are the rules for In-Order traversal of a binary tree?
Q9. What are the rules for Pre-Order traversal of a binary tree?
Q10. What are the rules for Post-Order traversal of a binary tree?