3 Data Structures

A ‘data structure’ is an organisation of data. There are many different ways how data can be organised. Each data structure has its own operations that can be performed on the data e.g. to add data, to delete data, to sort data etc.

Data structures are very important in programming. We use them to organise data as suits the particular situation being designed and coded.

3.1 Static and Dynamic Data Structures

Some data structures after being created cannot be increased or decreased in size i.e. the maximum number of elements that the data structure can hold is defined once during the creation of the data structure. That number is fixed. Such data structures are called ‘static’. An array as defined in Java (and other programming languages) is a static data structure.

On the other hand ‘dynamic’ data structures when created can increase or decrease in size according to how many elements it contains. A file of records is one example of a dynamic data structure.

3.2 Arrays

An array is an indexed collection of data values of the same type.

3.2.1 One-Dimensional Arrays

A one-dimensional array is a sequence of elements of the same type. Each element of the one-dimensional array is referred to by an index.

	0	1	2	3	4	5	6	7	8	9
num	22	5	-47	54	8	0	-36	9	54	-8

Diagram 25. Array num

The whole data structure has a name which is ‘num’. Elements can be accessed individually e.g. num[3] refers to the element 54.

Note that all the elements are integers (whole numbers). We therefore say that the ‘type’ of the elements is ‘integer’ (or ‘int’ in Java). Other types are ‘double’ (numbers that can have a decimal part e.g. 5.26) and ‘string’ (a string is a sequence of characters e.g. “Mary Smith”).

The number 2 in the expression num[2] is called the ‘index’.

The array in diagram 25 is created in the following way in Java:

```
int num[] = new int[10];
```

Refer to the array in diagram 25. The following are some expressions (an expression is a sequence of values and operators that can be worked out to produce a result; e.g. 5-8+2) using array elements:

(a)	num[1] - 3	(this expression evaluates to 2)
(b)	2*num[7] + num[0]	(this expression evaluates to 40)
(c)	num[2+4]	(this expression evaluates to -36)
(d)	num[12]	(this expression is invalid)

The following are a few assignment statements (a statement is a sentence in a HLL that tells the computer what to do):

(a)	num[4] = num[2] + 50;	(this statement changes the value of num[4]; num[4] is now equal to 3)

(b)	i = 6;
	num[i] = num[i+1];
		(this statement changes the value of num[6]; num[6] is now equal to 9)

3.2.2 Applications of One-Dimensional Arrays

One-dimensional arrays can be used to sort elements. For example the elements from a file (on disk) can be copied in an array, then sorted, then put in a new file on disk. One-dimensional arrays can also be used to implement Queues and Stacks. Queues and Stacks are two other data structures (to be studied later on).

3.2.3 Exercise on One-Dimensional Arrays

a) Considering the array ‘num’ in diagram 25 evaluate the following expressions:

   i) num[8] + 12

   ii) 3*num[9] + 2*num[2]

   iii) 2*num[2 + 5]

b) Consider again the array ‘num’ and show how the array changes after the execution of the following assignment statements:

```
i)    num[3] = 2*num[2] + 7;

ii)   num[8] = num[1 + 5] + 2*num[4];

iii)  i = 4;
      num[i] = i*num[3] - 8*i;

iv)   i = 2;
      j = 4;
      num[i + j] = 3*num[3*i] + 2*num[j];
v)    for t = 0 to 9 do
      num[t] = num[t] + 3;
```

3.2.4 Multi-Dimensional Arrays

Let us take an example of a 2-dimensional array.

	0	1	2	3	4	5	6	7
0								
1					6			
2								
3								
4								
5			23					
6								
7								
8								
9								
10						50		
11								

Diagram 26. Array Sales

This array's name (diagram 26) is called Sales. In this example the horizontal indexes (0 to 7) represent Salespersons while the vertical indexes (0 to 11) represent codes of objects.

In Java the table would be created thus:

```
int Sales [] [] = new int [12] [8];
```

Using the Java notation, Sales [5] [2] is equal to 23. This means that salesperson 2 sold 23 items whose code is 5. The following are some expressions (referring to the array Sales):

(a)	Sales [10] [5] + 2 * Sales [1] [4]	(evaluates to 62)
(b)	i = 4; Sales [i+1] [i div 2] – 15	(evaluates to 8)

3.2.5 Applications of Multi-Dimensional Arrays

The Sales array is an example of the use of a 2-dimensional array. In this case we had two variables 'Salesperson' and 'Code of Item' that indicated how many items of a particular code were sold by a particular salesperson. A digital picture is an array where each element (pixel) represents a colour. The mathematical branch of matrices can be programmed such that each matrix is a 2-dimensional array. There can be various applications for multi-dimensional arrays depending on the situation to be programmed.

3.2.6 Exercise on Multi-Dimensional Arrays

(a) Draw an array similar to Sales above that contains 4 rows and 3 columns. Call this array 'ArNum'. This array will hold integers.
(b) Show how this array is created in Java.

(c) Fill some values in the array as shown here:

   (i) ArNum [2] [1] = 7 (assume that the first index represents the row)
   (ii) ArNum [0] [0] = -3
   (iii) For i=1, 2 and 3 ArNum [i] [0] = 2 * i.
   (iv) For the other elements ArNum [i] [j] = i + j.

(d) Evaluate the following expressions:

   (i) ArNum [2] [1] + 2 * ArNum [3] [0].
   (ii) Add all ArNum [1] [j] for j = 0, 1 and 2.
   (iii) Add all ArNum [i] [2] for i = 0, 1, 2 and 3.

3.3 Strings

A string is a sequence of characters. The following are all examples of strings:

- "computer"
- "I like data structures!"
- "" (this is called the empty string)

Note that a string is written between inverted commas (quotes). The inverted commas tell the computer where the string starts and where it ends. They are called **Delimiters** – i.e. they show the limits of the string. In Pascal the delimiters of the string are apostrophes. So in Pascal we would write 'computer'. In Java the delimiters are inverted commas.

A string is also considered to be a one-dimensional array of characters. In the first version of Pascal you could not define a variable as a string but you had to define an array of characters for each string used.

The following syntax shows how a string can be declared and initiated in Java.

```
String st;          (this line declares st as a string)
st = "ship";        (this line initializes the value of st to
                     "ship")
```

Declaration and initialization can be combined in one line as shown here.

```
String st = "ship";
```

Note that the keyword **String** starts with an uppercase S. Java is a **case-sensitive** language and if String is not written with the letter S in uppercase the compilation would not complete and you would be informed the program has a syntax error.

3.3.1 Some Operations on Strings

Java presents quite a huge number of operators (they are called 'methods') that work on strings. Let us look at a few of them.

length: This operator gives the number of characters of a string e.g.

- “Mark”.length() gives a value of 4
- if st = “Junior College” then st.length() gives a value of 14 (in the examples below st will always represent the string “Junior College”).

charAt: This operator gives the character at a given position. e.g.

- “Mark”.charAt(3) gives a value of ‘k’. Note that, like the indices in arrays, the index of the character at position n is n-1. So at index 3 we find the 4th character.
- st.charAt(3) gives ‘i’.

equals: This operator compares two strings and if they are equal (even in case) then it gives the Boolean result of True. If not it gives a result of False. e.g.

- st.equals (“junior college”) is evaluated to False.

indexOf: This operator gives you the index of the first occurrence (from the left hand side) of a given character. e.g.

- st.indexOf (‘l’) returns a value of 9.
- st.indexOf (‘a’) returns a value of -1 because the character ‘a’ does not appear in string st.

toLowerCase: This operator changes the whole string to lowercase letters. e.g.

- st.toLowerCase() returns “junior college”.

toUpperCase: This operator changes the whole string to uppercase letters. e.g.

- st.toUpperCase() returns “JUNIOR COLLEGE”.

substring: This operator returns a part of the string according to the indices given. e.g.

- st.substring (10, 13) returns “leg”. Note that the character ‘l’ is in position 10 but ‘g’ is in position 12. The second parameter has to be equal to the position of the last character of the substring plus one.

concat: This operator joins strings together. e.g.

- st.concat (“ is located in Msida”) returns the string “Junior College is located in Msida”. The operator concat can be represented by the + sign. For example “wind” + “mill” will give “windmill”. Also “ten” + “tat” + “ive” gives the word “tentative”.

Note that although all operators are applied to strings some return an integer, some return a character and some return a string.

It should also be said that some operators (methods) have more than one version for example there are two versions of indexOf. Here we give an example of each:

- "follow".indexOf ('o')) gives a result of 1 (this form of indexOf has already been seen above)
- "follow".indexOf ('o', 3)) gives a result of 4. First of all note that in the first example indexOf has one parameter (i.e. the value inside the bracket) while in this example it has two parameters. In this case the operator looks for the character 'o' from the character with index 3 onwards.

3.3.2 String Expressions

Some examples (assume st1 = "Corridor" and st2 = "Flight of steps."):

(a) st1.length() + st2.indexOf ('t') gives a result of 13.

(b) st1 + " and a " + "f" + st2.substring(1, 16) gives a result of "Corridor and a flight of steps."

(c) (st2.substring (1, 6) + " " + st1.substring (st1.length()-2, st1.length()) + " heavy?").toUpperCase() gives a result of "LIGHT OR HEAVY?"

3.3.3 Exercise on Strings

Consider these three strings:

st1 = "cruise liner"
st2 = "voyage to Bahamas"
st3 = "holiday in Valletta"

Evaluate the following string expressions:

(a) st1.length() + st2.length() + st3.substring(1, 5).length()

(b) If r = st1.indexOf(' ') what is the value of the following expression:

st1.substring(0, r) + st1.substring(r+1, st1.length())

(c) st1.substring(0, st1.indexOf(' ')).concat (st2.substring (st2.indexOf (' '), st2.length()))

3.4 Records

A **record** is a sequence of data. The various data that form a record can be of different types. Each datum is written in a **field**. The following diagram is a representation of a record. The record's name is Member and it consists of four fields. The first three fields are of type 'String' while the other is of type 'float'.

Account No	Name	Address	Amount
12-3456	Farrugia Stephan	23, St. Paul Square, Paola	€ 5,784.08

Member

Diagram 27. A record

Records can have one of two forms:

- **Fixed size** i.e. all records have the same number of fields and each corresponding field of different records has the same size.
- **Variable size** i.e. records may have a different number of fields; also each corresponding field of different records can be of different size.

A sequence of records constitutes a 'file of records' or a '**table**'. Note that a file of records is also a data structure.

Records are also called '**tuples**'.

In Pascal the keyword 'record' exists to define a type of record. In Java this keyword does not exist. Still one can create a prototype (model) of a record in a Class and then create as many records of this type as required by creating Objects of the said Class (more of this later).

Example: Class SetOfStamps

Date of Issue	Title Name of Issue	No in Set
(Type is String)	(Type is String)	(Type is int)

Diagram 28. A record defined by means of a class

Three Objects derived from class SetOfStamps may be:

Date of Issue	Title Name of Issue	No. in Set
1/9/65	IXth Centenary Of The Great Siege	7
20/3/71	De Soldanis & Dun Karm	2
12/1/74	Prominent Maltese	5

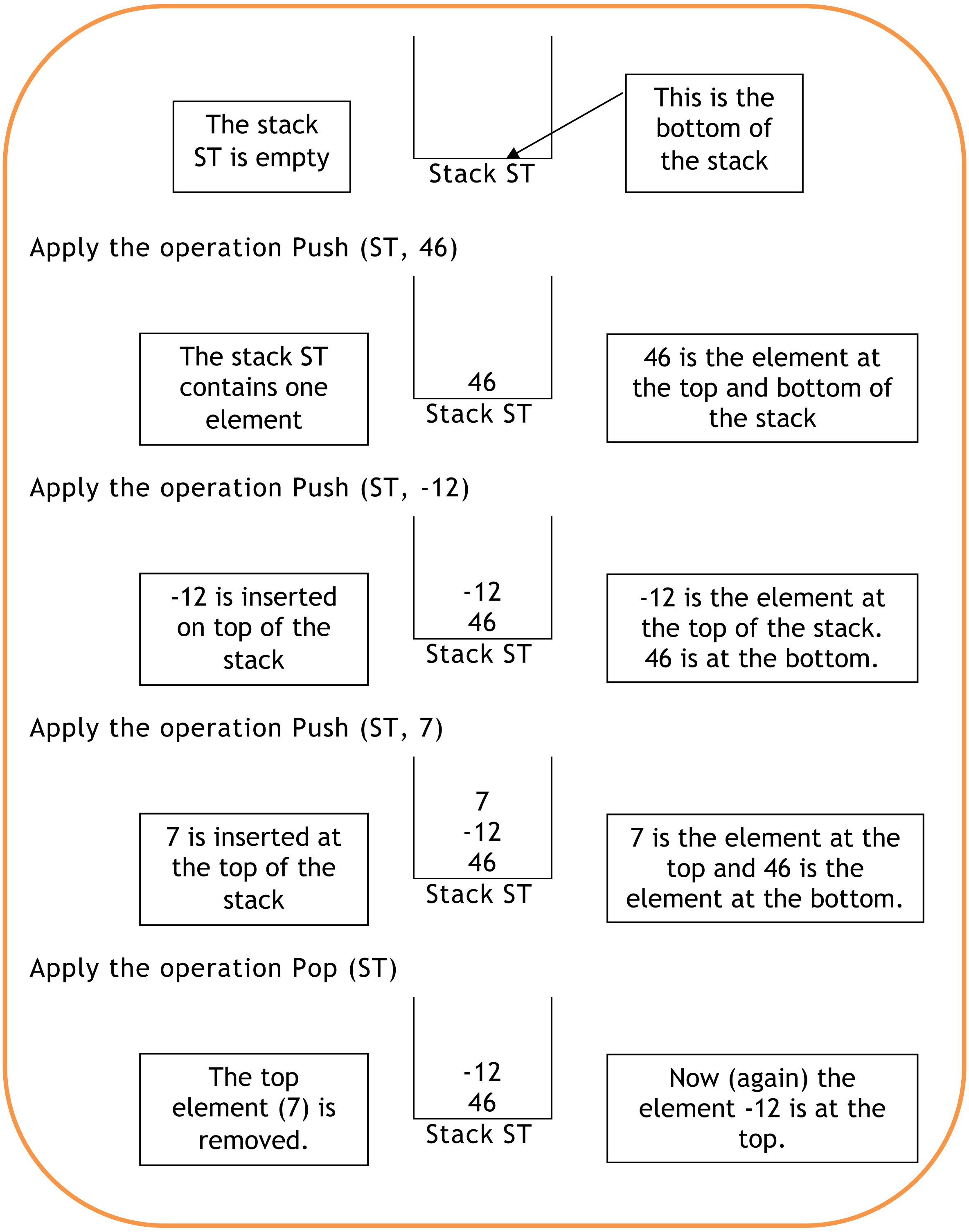
Diagram 29. A Table

3.5 Stacks

A **stack** is a data structure that simulates a stack of items put one above another in such a way that when a new item is included in the stack it is put on top and when an item is removed it is the one at the top that is removed. This is called a **LIFO** (**Last In First Out**) discipline. When an item is put on top of the stack the operation is called 'push'. When the top item is removed the operation is called 'pop'.

3.5.1 An Example of a Stack

Let us imagine a stack of integers named ST. Diagram 30 shows a number of operations on stack ST.



The stack ST is empty
Stack ST
This is the bottom of the stack
Apply the operation Push (ST, 46)
The stack ST contains one element
46
Stack ST
46 is the element at the top and bottom of the stack
Apply the operation Push (ST, -12)
-12 is inserted on top of the stack
-12
46
Stack ST
-12 is the element at the top of the stack. 46 is at the bottom.
Apply the operation Push (ST, 7)
7 is inserted at the top of the stack
7
-12
46
Stack ST
7 is the element at the top and 46 is the element at the bottom.
Apply the operation Pop (ST)
The top element (7) is removed.
-12
46
Stack ST
Now (again) the element -12 is at the top.

Diagram 30. A Stack in operation

3.5.2 Applications of Stacks

There are various applications of stacks in computer science. One of them is its use to keep track of procedure returns i.e. if a procedure A is called and inside A a procedure B is called and inside B a procedure C is called then when procedure C ends the computer has to continue executing procedure B and when this ends then procedure A has to continue its execution. To keep track where the computer must go to continue a procedure a stack is used.

3.5.3 Exercise on Stacks

A stack named STCK that holds numbers is initially empty. Show how STCK will appear after the following operations:

```
push (STCK, 5)
push (STCK, 8)
push (STCK, -3)
n = pop (STCK)
push (STCK, 7-n)
p = pop (STCK)
q = pop (STCK)
r = pop (STCK)
push (STCK, p - q + r)
```

3.6 Queues

A queue is a data structure that follows a FIFO (First In First Out) discipline. It follows the principles of a normal queue where elements arriving at the queue are placed at the end of the queue and the element that leaves the queue is the first element. The Queue has two basic operations which are 'insert' (also called 'enqueue') and 'remove' (also called 'dequeue').

3.6.1 An Example of a Queue

Let the queue name be 'Jobs' and let it hold integers.

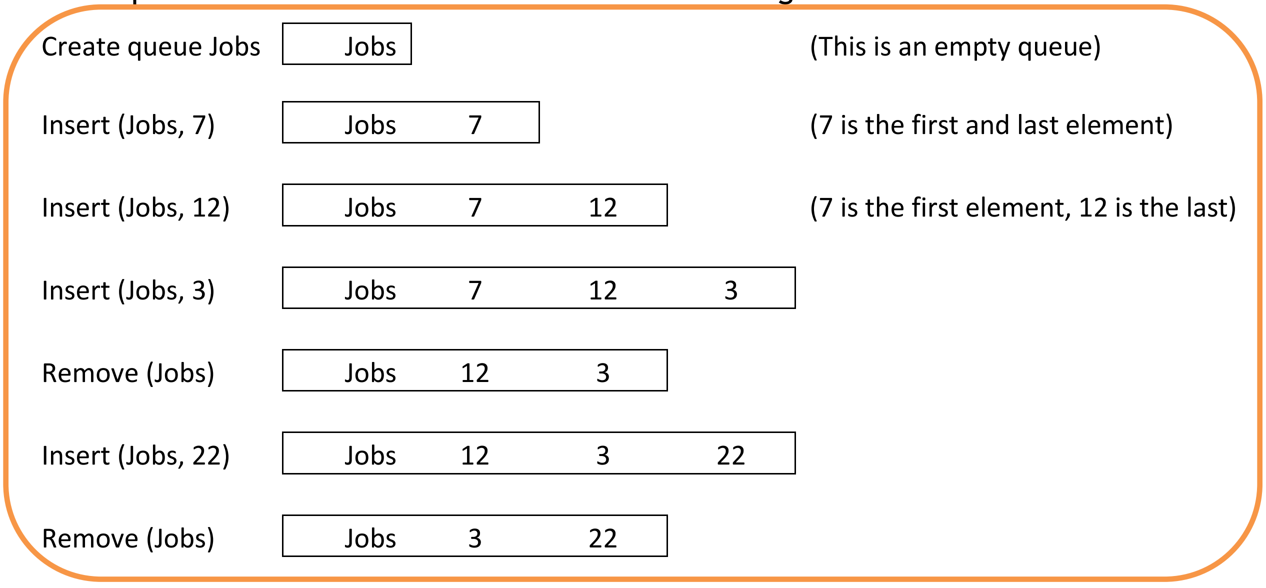


Diagram 31. A Queue in operation

3.6.2 Applications of Queues

Two applications of stacks are the Spooler, where a printer keeps a note of all the files to be printed. When a file is printed its name and location are removed from the queue and the next file in the queue is printed. A file that requires printing is put at the end of the queue if the printer is busy. Another example of the use of a queue is the keyboard buffer where characters pressed on the keyboard are kept in a queue before being passed to RAM.

3.6.3 Exercise on Queues

A queue named QU holds RAM addresses. Follow the steps below to show how QU is changing:

```
Insert (QU, AB6h)
Insert (QU, 24Fh)
Insert (QU, 109h)
Remove (QU)
Remove (QU)
Insert (QU, 10Ah)
```

3.7 Trees

A tree is a data structure in the form of a hierarchy. The diagram below shows a tree.

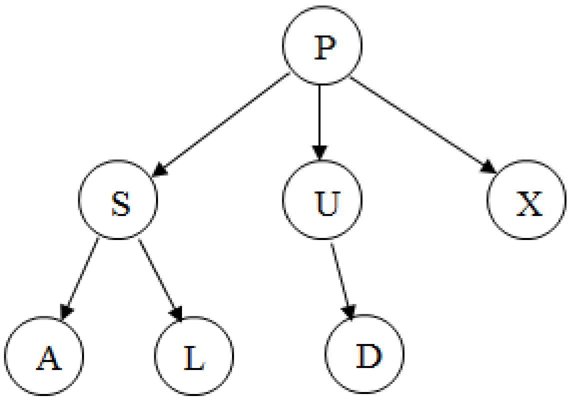


Diagram 32: A Tree

This tree has 7 'nodes'. P is called the 'root'. The terminology of trees is derived from that of family trees. So we say that node S is the 'parent' (or 'father') of nodes A and L. Thus P is the 'grandparent' ('grandfather') of A and L. Other relationships are 'child' ('son'), 'grandchild' ('grandson') and 'sibling' ('brother'). A part of a tree, for example the nodes S, A and L, is called a 'subtree'. The tree in this example is made up of three levels. Nodes that have no children (in this case A, L, D and X) are called 'terminal nodes' or 'leave nodes'.

Relations are indicated by the arrows, so P is related ('parent of') to S, U and X. A node cannot have two or more parents. P is also related to D as grandparent etc.

3.7.1 Applications of Trees

Trees have a lot of applications. One of the most important is that it can be used as an index to find data. By means of the tree-index, searching for data becomes very fast.

3.7.2 Binary Trees

A special tree is the 'binary tree'. In this tree each node cannot have more than two children. Children nodes are placed on the left or right of the parent. The diagram below shows a binary tree. The nodes in this tree are in alphabetical order. On the left subtree of each node are elements that precede (alphabetically) the node and on the right subtree are elements that follow the node.

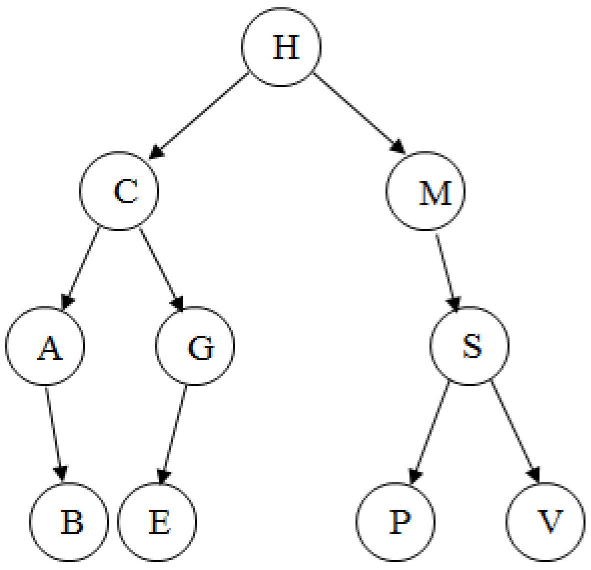


Diagram 33: A Binary Tree

The **leftmost node** is found by starting from the root and moving left until there are no more nodes on the left. The **rightmost node** is found in a similar way. "A" and "V" are respectively the leftmost and rightmost nodes. When a binary tree is organised in this way searching for an element and inserting a new element are easy tasks. Deleting an element is slightly more complex.

3.7.2.1 Searching for an Element in an Ordered Binary Tree

The searching of an element in an ordered binary tree is described by means of the algorithm in diagram 34. The algorithm simply searches the binary tree and returns either "yes" or "no".

```
Searching_in_Ordered_Binary_Tree (TR, N)
begin
    Node = root of TR
    found = no
    while (Node is not nil) and (found==no) do
        begin
            if (N == Node)
                then found = yes
                else if (N<Node) then Node=root of left subtree (Node)
                                 else Node=root of right subtree (Node)
        end
    return found
end
```

Diagram 34: Searching an Ordered Binary Tree

3.7.2.2 Insertion in an Ordered Binary Tree

Insertion follows the algorithm in diagram 35.

```
Insertion_in_an_Ordered_Binary_Tree (TR, N)
begin
    Node = root of TR
    inserted = no
    while (inserted == no) do
        begin
            if Node == nil
                then
                        begin
                                Node = N
                                inserted = yes
                        end
                else if (N < Node)
                        then Node = root of left subtree (Node)
                        else Node = root of right subtree (Node)
        end
end
```

Diagram 35: Insertion in an Ordered Binary Tree

3.7.2.3 Deleting an Element in an Ordered Binary Tree

Here we are assuming that the binary tree is alphabetically ordered and after the deletion of the desired element the tree will remain alphabetically ordered. To delete one can follow the steps in the algorithm found in diagram 36.

1) Find the element. Let us call it X.
2) If X is a terminal node then simply delete it from the tree. If X is not a terminal node then perform (a) OR (b)
    a. Find the rightmost element of the left subtree of X. Let us call this Y. Put the value of Y in place of X. Repeat the whole process (steps 1 and 2) to eliminate Y
    b. Find the leftmost element of the right subtree of X. Let us call this Y. Put the value of Y in place of X. Repeat the whole process (steps 1 and 2) to eliminate Y.

Diagram 36: How to Delete a Node from an Ordered Binary Tree

As an example consider the tree in figure 37. Suppose the node G needs to be eliminated. One solution would perform the steps (1) Go to the left subtree of G, (2) take its rightmost node (in this case it is E), (3) remove G and put E in its place. The resulting tree is ordered. A second solution would (1) consider the right subtree of G, (2) find the leftmost node (in this case this is J, (3) remove G and put J in its place.

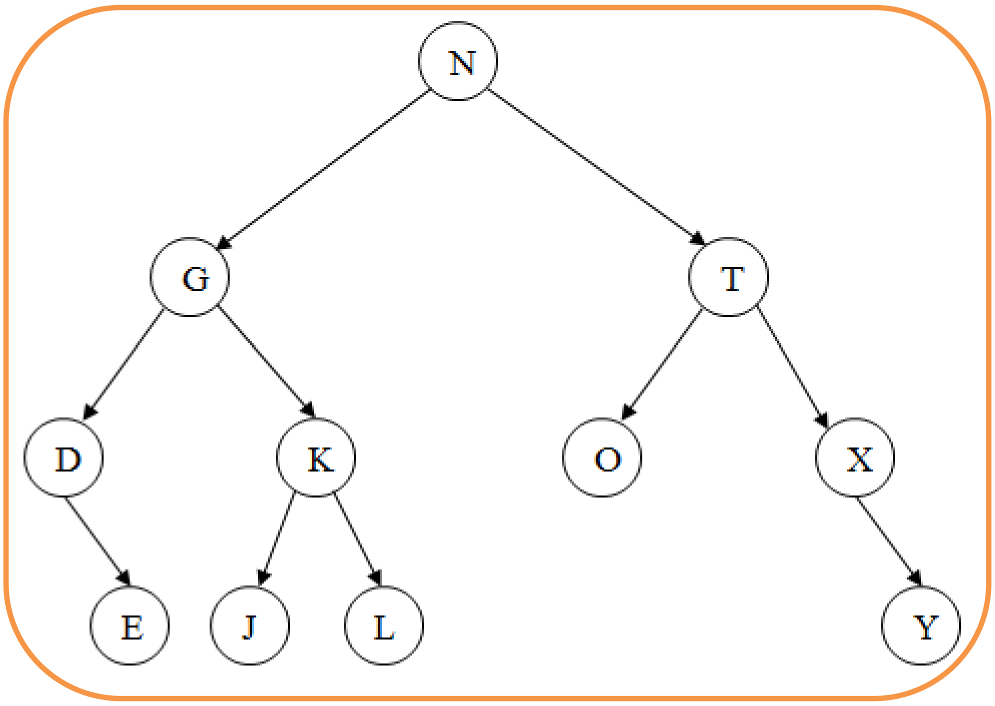


Diagram 37: An Ordered Binary Tree

3.7.2.4 Traversals

A traversal is a method by means of which all the nodes of a tree are visited and read. In the case of a binary tree (not necessarily ordered) the most three common traversals are:

- Inorder traversal
- Preorder traversal
- Postorder traversal

These traversals can be defined as shown here.

Preorder traversal

- Visit the root node
- Traverse the left subtree.
- Traverse the right subtree.

Inorder traversal

- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.

Postorder traversal

- Traverse the left subtree.
- Traverse the right subtree
- Visit the root node.

Let us apply the preorder traversal to the tree in diagram 38.

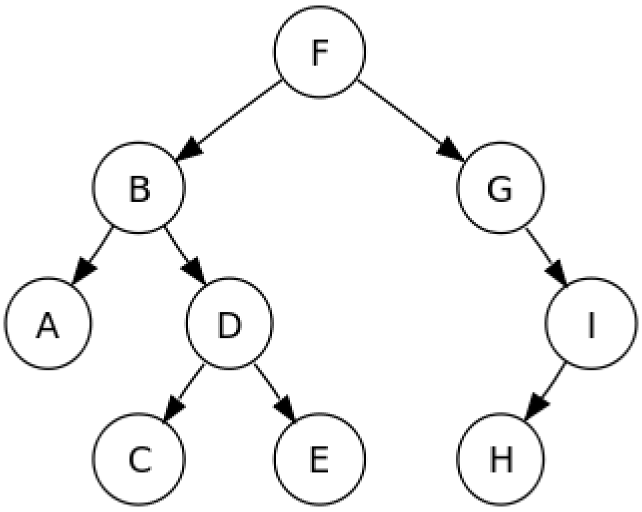
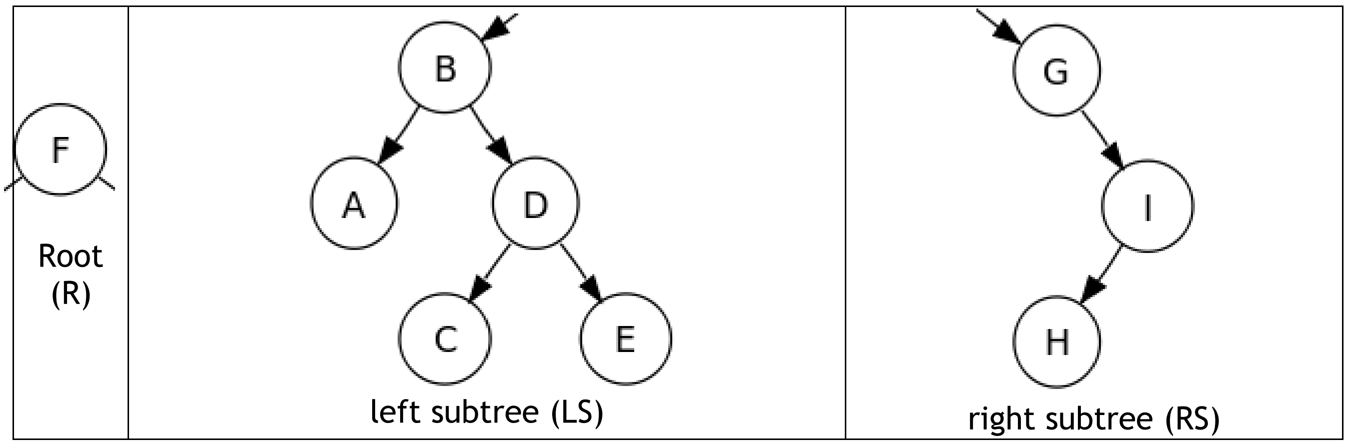


Diagram 38: An Ordered Binary Tree



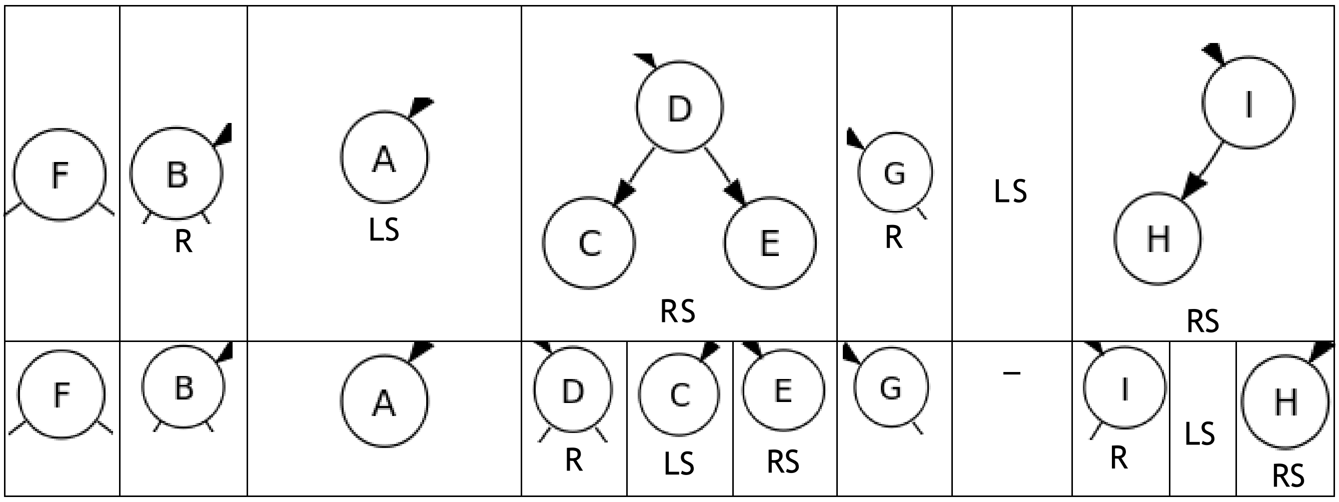


Diagram 39: Performing Preorder Traversal from First Principles

Therefore the traversal gives F, B, A, D, C, E, G, I, H

There is however a quicker way to perform this traversal. Refer to diagram 40. Follow these three steps:

1. Mark a dot at the left part of each node.
2. Mark a path from left to right that circles all the nodes as shown.
3. Write the nodes in the order as the dots are met along the path.

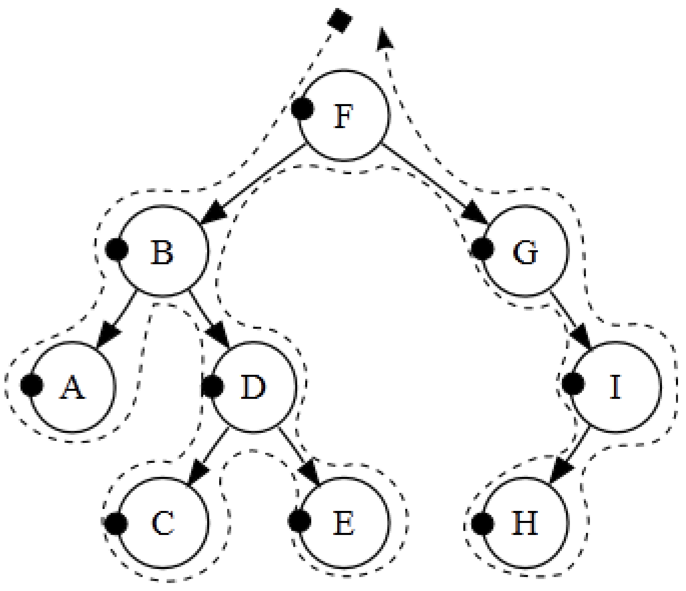


Diagram 40: A Quick Way to Perform Preorder Traversal

To perform the inorder and postorder traversals do the same but mark the dots respectively at the lower centre and right of the nodes as shown in diagram 41.

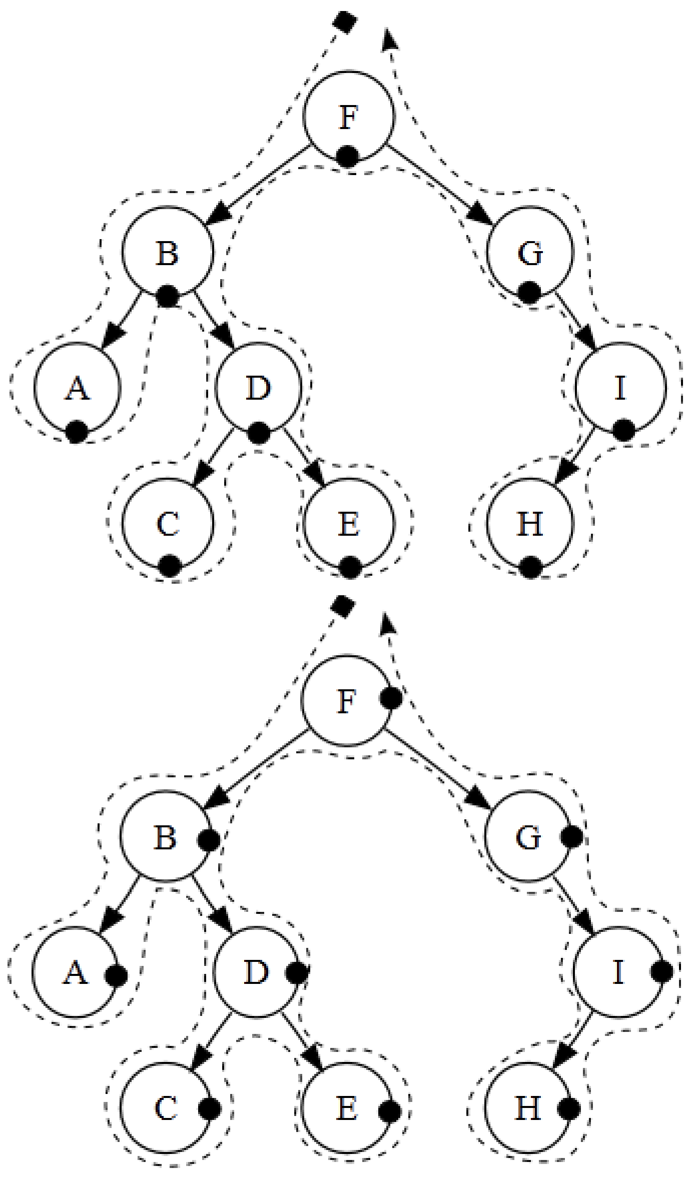


Diagram 41: The Quick Solution for the Inorder and Postorder Traversals

The result for the inorder traversal is A, B, C, D, E, F, G, H, I. As can be seen this traversal visits the nodes in alphabetical order. For this to happen the binary tree has to be ordered.

The result of the postorder traversal is A, C, E, D, B, H, I, G, F.

3.7.2.5 Binary Tree to Represent an Arithmetic Expression

We have already seen one application of binary trees i.e. that of keeping strings in alphabetical order. Another application is that of representing expressions. Consider the arithmetic expression (4 + 6) * 5 – 3 * (2 * 9 – 4). This expression is shown as a binary tree in diagram 42.

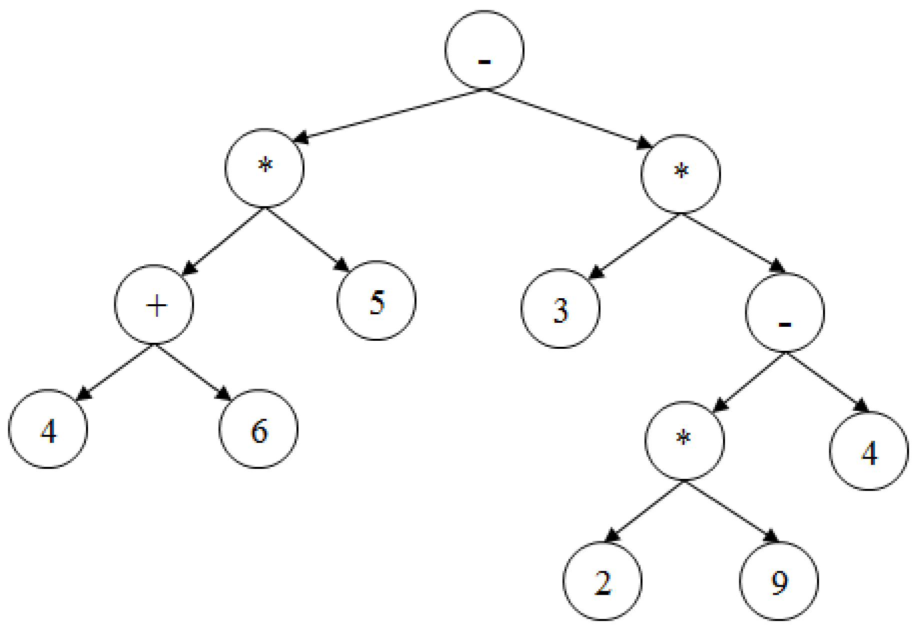


Diagram 42: An Arithmetic Expression

The expression as represented in the binary tree can be evaluated in a bottom-up manner. This means that terminal nodes are calculated with their respective operator. The terminal nodes are eliminated and the node holding the operator will now hold the solution of the operation. Finally the root node will hold the value of the expression.

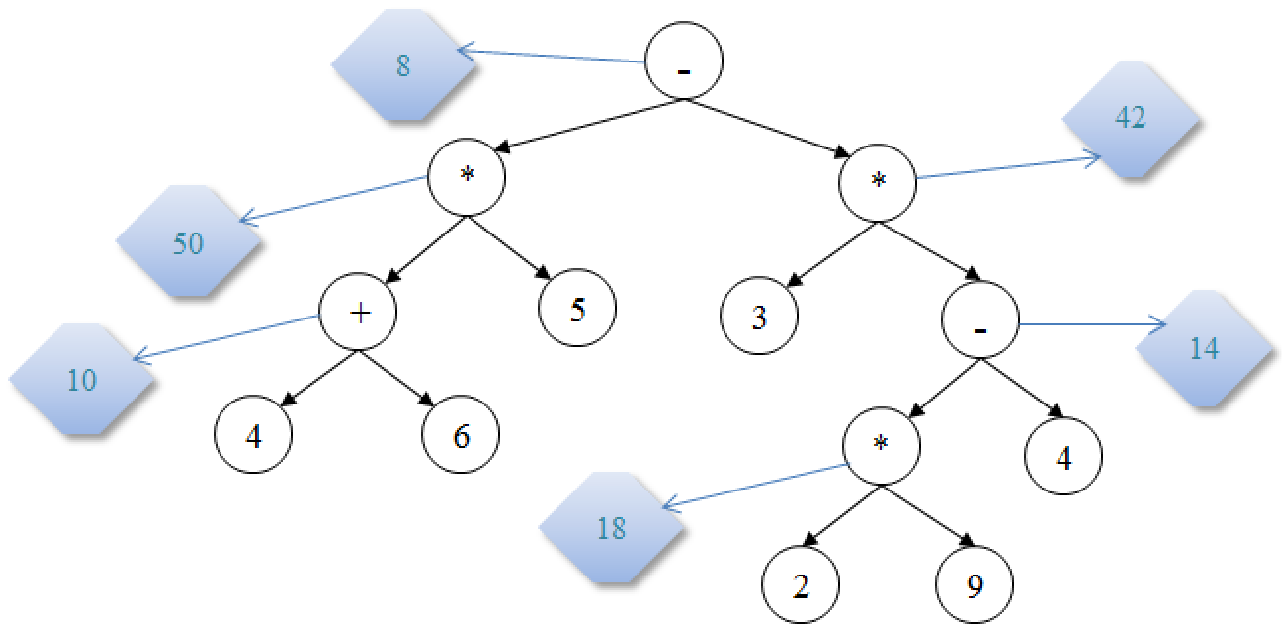


Diagram 43: Bottom-Up Evaluation of an Arithmetic Expression

3.7.2.6 Exercise on Binary Trees

(a) Perform all three traversals on the binary tree in diagram 44.
(b) Add to the tree the node T.
(c) Delete the node R.

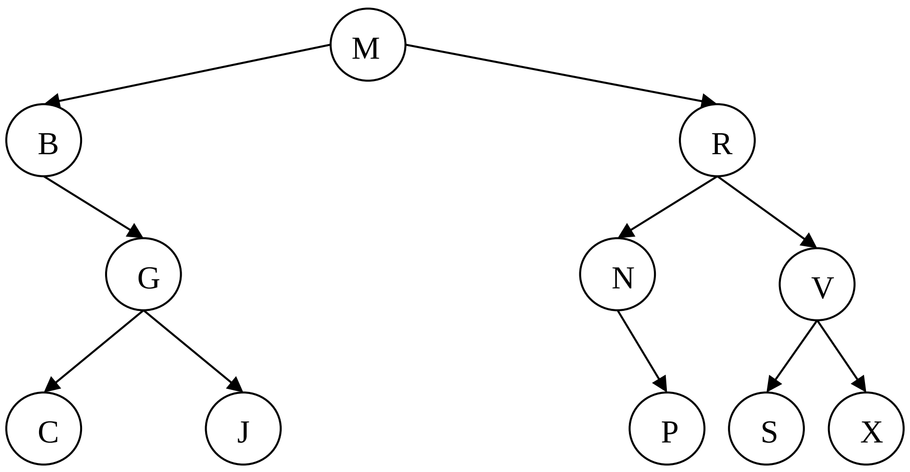


Diagram 44: Bottom-Up Evaluation of an Arithmetic Expression

(d) Express the arithmetic expression 7 – 8 * 2 / (4 – 2) as a binary tree.
(e) Evaluate this expression by using the bottom-up method on the tree.

3.8 Linked Lists

A 'linked list' holds data in such a way that the information in one element leads to the location of the successive element. Graphically, a linked list is represented as shown in diagram 45. This list particular list contains 5 elements. Each element consists of two parts; the first contains the value required and the second contains a pointer that shows where the next element is. The pointer of the last element is a value that shows that the list ends there. We also need a pointer to show us where the list starts.

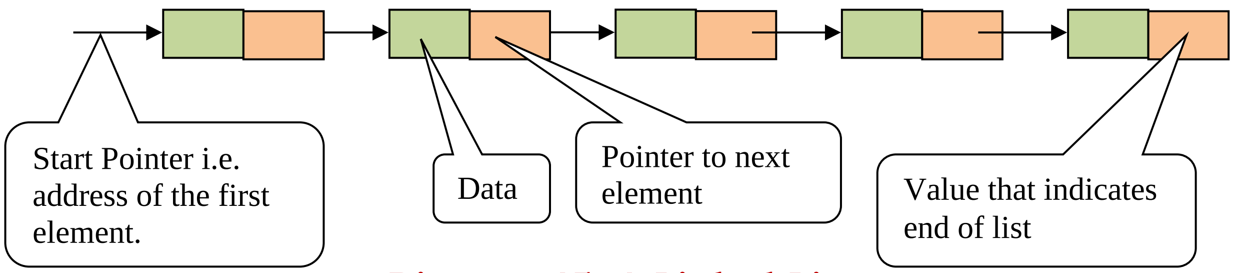

Start Pointer i.e. address of the first element.
Data
Pointer to next element
Value that indicates end of list

Diagram 45: A Linked List

3.8.1 Example of a Linked List

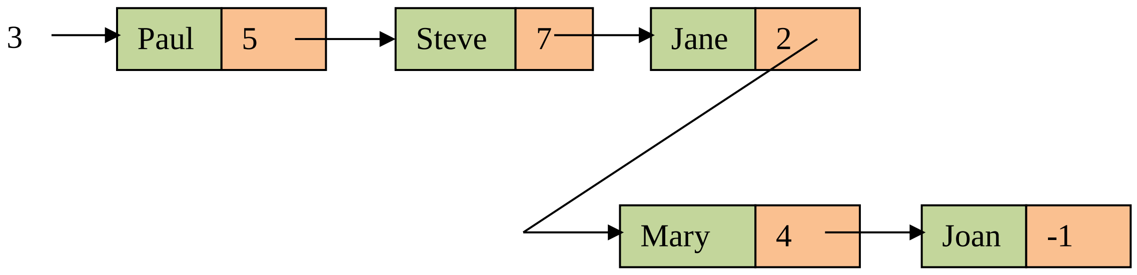


Diagram 46: A Linked List holding Names

Note that (from diagram 46):

- The address of Mary is 2.
- Paul is the first element and his address is 3.
- The end-of-list is marked here by -1.

3.8.2 Searching in a Linked List

A linked list is a sequential data structure. Therefore when a search is made the search has to start from the first element. If the first element is not the element required we go to the second element, and so on.

3.8.3 Adding an Element to a Linked List

Consider the example in diagram 46. Suppose we need to add the element 'Mark' after 'Jane'. To do this we do the following:

a) Get a list element from the free list (the free list holds elements of the list that are currently not in use). (Let us assume that the free element is location 10)
b) Write 'Mark' in the data section at location 10.
c) Make the pointer of 'Mark' point to 'Mary' (i.e. this pointer should be 2).
d) Change the pointer of 'Jane' to point to 'Mark' (the pointer part of element holding 'Jane' is changed to 10).

These steps create the linked list in diagram 47.

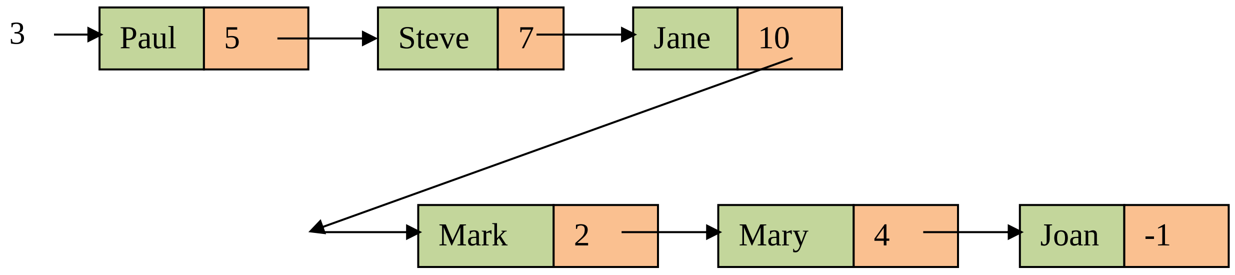


Diagram 47: Adding an Element to a List

3.8.4 Deleting an Element from a Linked List

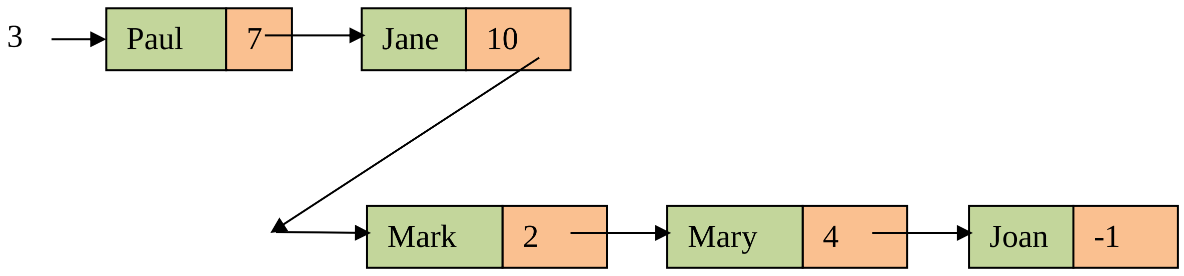


Diagram 48: Removing an Element from a List

Now 'Steve' has been removed. The following steps were taken:

(a) The pointer of the element holding 'Paul' now points to 'Jane'.
(b) The element containing 'Steve' is returned to the free list.

3.8.5 An Advantage and a Disadvantage of Linked Lists

One advantage of linear lists is that an insertion or deletion of an element causes little change in the data structure holding the linked list. However one disadvantage is that searches in linked lists are only sequential.

3.8.6 Circular Lists

A circular list is simply a list that has the pointer of the last element pointing to the start of the list. A diagram is shown below.

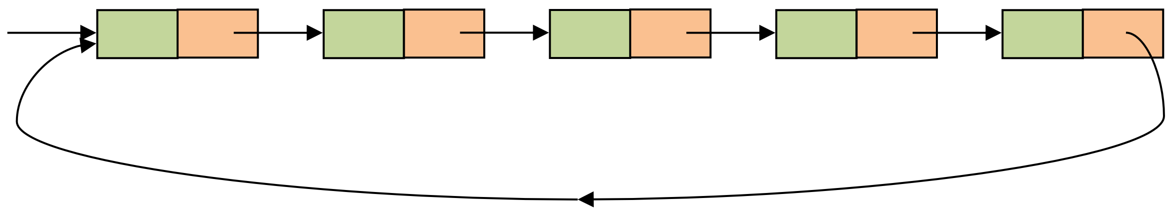


Diagram 49: A Circular List

3.8.7 Double Linked Lists

A double linked list (also called two-way linked list) gives the possibility of traversing the list in two directions. In the linked lists we have seen so an element could only take you to the next element but not to the preceding element. Double linked lists can do this.

The diagram below shows how this can be done.

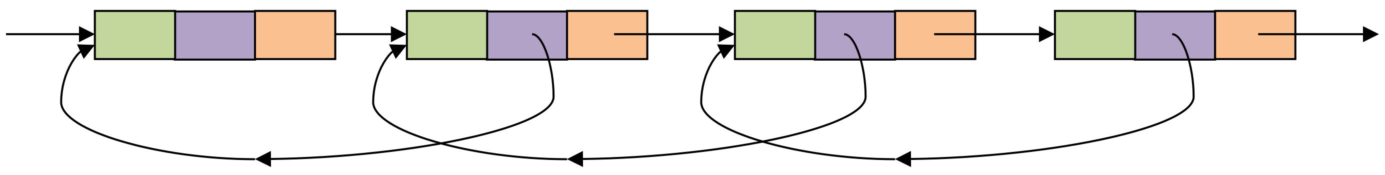
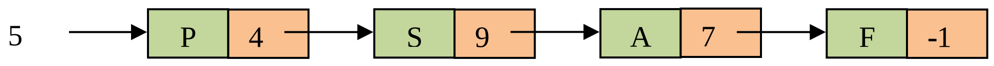


Diagram 50: A Double-Linked List

3.8.8 Exercise on Linked Lists

Consider the following linked list.



a) In which position (address) can you find letter A?

b) In which position can you find letter P?

c) A new element M is placed at location 2. This is to be inserted between S and A. Redraw the new linked list.

d) Now S is removed. Redraw the linked list again.

e) How would the linked list (after the modifications) appear if it was changed to a circular linked list?

f) How would the same linked list appear if it was changed to a two-directional linked list?

3.9 Vectors

A Vector is similar to an array of elements (although it can hold values of different types) but it is dynamic i.e. the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Java has a public class called 'Vector'. As in arrays each element is referred to by an index. The index of the first element is 0 and that of the nth element is n-1.

Here is a list of some methods (operations) that can be applied on a Vector object:

Method	**Description**
addElement(elt)	Add the element 'elt' at the end of the Vector increasing its size by 1.
insertElementAt(elt, ind)	Inserts the specified object 'elt' at the specified index 'ind'. Each component in this vector with an index greater or equal to the specified index is shifted upward to have an index one greater than the value it had previously.
firstElement()	Returns the first component (the item at index 0) of this vector.
lastElement()	Returns the last component of the vector.
elementAt (ind)	Returns the component at the specified index 'ind'.
setElementAt (elt, ind)	Sets the component at the specified index 'ind' of this vector to be the specified object 'elt'.
removeElement(elt)	Removes the first (lowest-indexed) occurrence of 'elt' from this vector. If 'elt' is found in this vector the other elements that follow 'elt' are shifted accordingly.
removeElementAt(ind)	Deletes the component at 'ind'. Corrective shifting occurs and the size of the vector is decreased by 1.

3.9.1 Exercise on Vectors

In a Java program a vector called ‘vec’ is created (it is therefore initially empty). What values will ‘vec’ hold after the execution of the following statements?

```
vec.addElement('P');
vec.addElement('D');
vec.addElement(2009);
vec.addElement('F');
vec.insertElementAt('G', 2);
vec.setElementAt ('M', 1);
vec.addElement ('G');
vec.removeElement ('G');
vec.removeElementAt (3);
```

3.10 Hash Tables

A hash table is a table that holds information according to a mathematical formula (called a ‘hash function’). The hash function calculates a value which is used as the address for entering information, searching information and deleting information.

3.10.1 An Example of a Hash Table

Records of the type shown hereunder are to be stored in an array that holds 1000 records. The indices of the elements in this array vary from 0 to 999.

ID. No	Name	Address
129655	Borg James	5, Church Square, Zurrieq

Diagram 51: A Record

3.10.2 Entering Data in a Hash Table

Now suppose the array (of records) is empty and we need to place this record in the array. The hashing system needs a mathematical function (called a ‘hash function’) to gives us the index. Let us say that the hash function is the following: index = (ID. No) MOD 1000

So to find where the above record should be placed the system works out the expression 129655 MOD 1000. This gives 655. So the above record is placed in the array cell with index 655. Now suppose the following two records are to be inserted in the hash table:

ID. No	Name	Address
402294	Micallef Alfred	23, Dun Karm Street, Qormi
354107	Busuttil Maria	76, Grapes Street, Naxxar

Diagram 52: Two Records

Applying the hash function to each would produce the indices 294 and 107. The array now would look like the diagram below.

	ID. No	Name	Address
0			
107	354107	Busuttil Maria	76, Grapes Street, Naxxar
294	402294	Micallef Alfred	23, Dun Karm Street, Qormi
655	129655	Borg James	5, Church Square, Zurrieq
999			

Diagram 53: A Hash Table

3.10.3 Collisions

Now suppose we want to insert the following record:

ID. No	Name	Address
155294	Zammit Sonia	45, Cinema Road, Fgura

Diagram 54: A Record to be Inserted

The hash function would give index 294. However at address 294 there is already a record. This situation is called a 'collision'. What happens now? There are various solutions given in case a collision occurs. Two solutions are the following:

(a) Create another table so that the records that find the location of their index occupied by another record would be put in it. This is called a 'collision table'.

(b) Create a collision table for each address.

Solution (b) is shown in diagram 55.

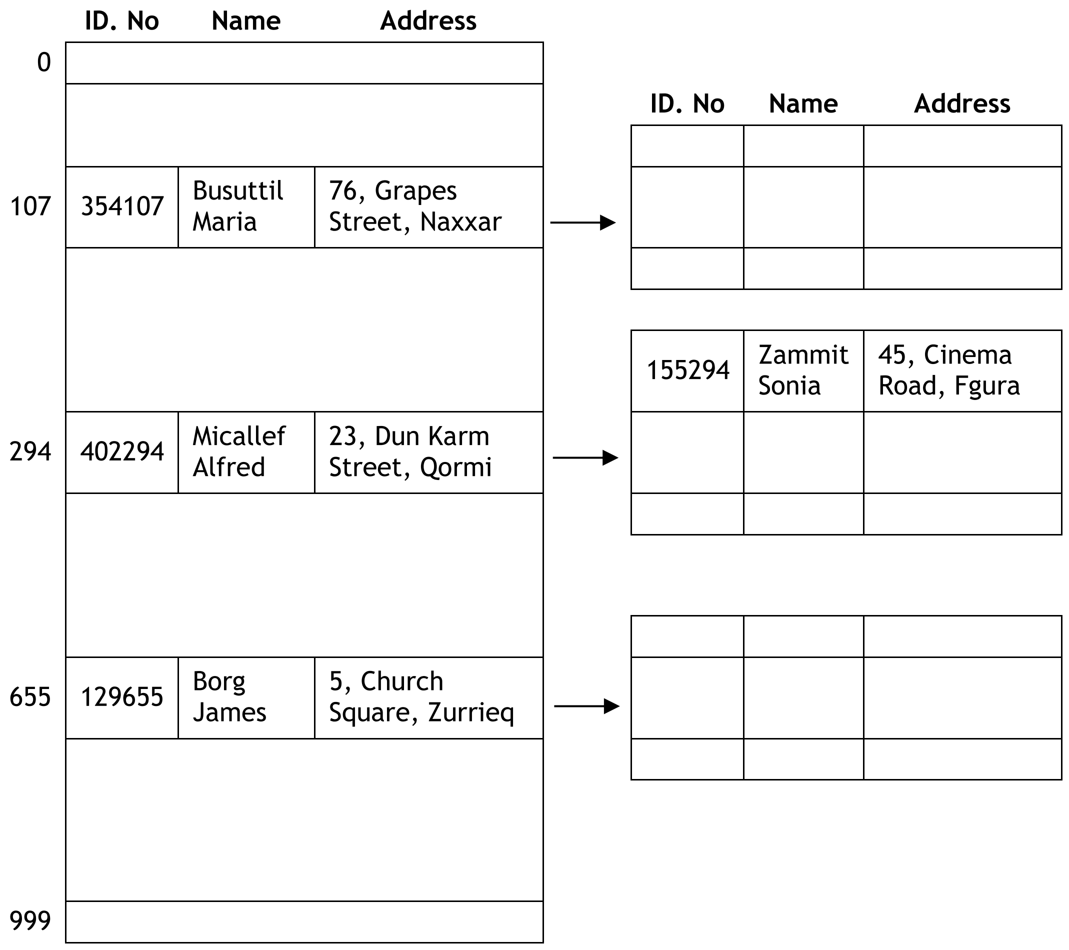


Diagram 55: A Hash Table with Collision Tables

3.10.4 Deleting an Element from a Hash Table

Let us assume the setup shown above (i.e. where each location has a collision table attached to it). The program that performs deletion has to take into consideration all the following possibilities:

- Delete a record of the like of “Borg James”. At address 655 the collision table is empty so “Borg James” is simply deleted.
- Delete a record of the like of “Micallef Alfred”. After this record is deleted “Zammit Sonia” is moved from the collision table and put at address 294.
- Delete a record that is found in the collision table.

3.10.5 Hash Functions

Some points on hash functions:

- There are thousands of hash functions.
- For an efficient use of hash tables the hash function has to possess the property of uniform distribution (i.e. when calculating addresses it gives

out numbers that are distributed evenly in the range of addresses available).

- The function should be easy to compute. This is used to have a program as efficient as possible.
- Two examples of hash functions are:
    - address = [key] MOD [size of table] (this is the one seen in the example above; the size of the table should be a prime number for the function to be uniform)
    - Mid-Square method. An example of this method is seen here:
        - key = 1210
        - square it (1210^2 = 01464100)
        - take middle four digits (4641) as address

3.10.6 Other Points on Hash Tables

Hash tables have random access. They are designed to have a very small number (almost zero) of collisions. The efficiency of the hash table depends primarily on the hash function and the size of the hash table.

Hash tables can be used for (a) storing records in a random file, and (b) searching for records with certain key values.

Each entry of the table is called a bucket. In general, one bucket may contain more than one record. In our discussions we considered only buckets with one record. This means that if a bucket contains 3 records, you can have two collisions before you require a collision table. If a record is mapped on a bucket that is full we say there is an 'overflow'.

In Java this data structure is implemented in a class called 'Hashtable'.

3.10.7 Exercise on Hash Tables

A small hash table consists of 10 locations (indexed from 0 to 9). Each location holds one record only and each location has a collision table consisting of three locations. The hash function is (identityNumber) MOD 10. Show how the table will appear after the following operations.

	ID Number	**Name**
insert	258483	Micallef Anne
insert	548761	Borg Michael
insert	589593	Zammit Stephen

insert	201456	Caruana Jane
insert	528088	Busuttil Claire
insert	801480	Portelli Christopher
insert	487677	Zahra John
insert	200488	Debono Sarah
delete	201456	Caruana Jane
delete	258483	Micallef Anne

Diagram 56: Records for Exercise

3.11 Array Lists

The Java class ArrayList is a fast and easy-to-use class representing one-dimensional arrays. ArrayList is similar to Vector. It is not synchronized i.e. using it in more than one thread may cause problems.

Simply put, a 'thread' is a program's path of execution. Most programs written in these days run as a single thread. This causes problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allow us to do this.

Multithreaded applications run many threads concurrently within a single program.

Some of the operations provided by the ArrayList class are the following:

- 'add' to insert an element in an array-list. The new element can be inserted in the middle of the list.
- 'remove' is the opposite operation of 'add'. When an element is removed, other elements shift to eliminate the space left by the parted element.
- 'clear': removes all of the elements from this list. The list will be empty after this call returns.
- 'removeRange': removes a range of elements from the array-list.

ArrayList stores only object references. That's why it's not possible to use it for primitive data types like double or int. In such a case use a wrapper class (like Integer or Double) instead. These classes wrap a value of a primitive type inside an object.

For multi-threaded (synchronized) array class use Vector (java.lang.Vector)

3.12 Linear Lists

The term ‘linear list’ is used to indicate a data structure where the elements are placed adjacent to each other in the form of a line. A one-dimensional array, a linked list, a stack and a queue are all considered as linear lists.

A tree is not a linear list.