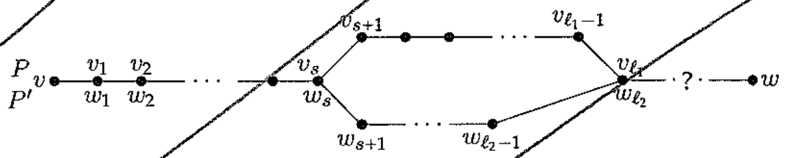
Suppose there is another path P' from v to w in T. Let

$$P' = vw_1 \dots w_n w.$$

P and P' share the first vertex and perhaps others, but they are different, so at some point they will differ. Let $v_s = w_s$ be the last vertex before P and P' differ for the *first* time.

P and P' share the last vertex, so they will meet again after splitting. Suppose this happens for the *first* time in $v_{\ell_1} = w_{\ell_2}$ such that ℓ_1, ℓ_2 are the smallest indices for which this occurs.



Note that in the place of the '?' the two paths are not necessarily the same.

Then we have a cycle $v_s v_{s+1} \dots v_{\ell_1} [= w_{\ell_2}] w_{\ell_2 - 1} \dots w_s$. But this is impossible as T is a tree and thus has no cycles. Hence there cannot be a second path P' from v to w in T. ∎

3.1.3 Spanning trees

We say that a graph H is a **subgraph** of a graph G if its vertices are a subset of the vertex set of G, its edges are a subset of the edge set of G, and each edge of H has the same end-vertices in G and H.

Definition 3.3 *If H is a subgraph of G such that $V(H) = V(G)$, then H is called a* ***spanning subgraph*** *of G. If H is a spanning subgraph which is also a tree, then H is said to be a* ***spanning tree*** *of G.*

Example 3.4 In Figure 3.2, the graphs T_1 and T_2 are both spanning trees of the graph G.

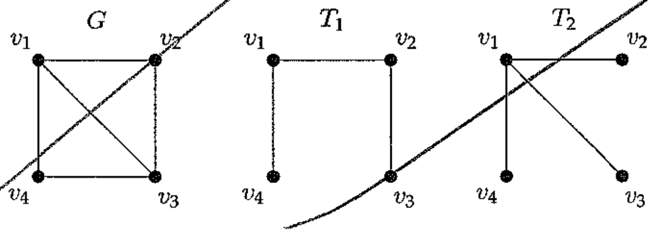


Figure 3.2: A graph G and two of its spanning trees

OPERATIONS RESEARCH 2

STARTS HERE

3.2 Rooted trees

Learning objectives for this section

- The definition of a rooted tree
- The terminology associated with rooted trees: Node, internal/external node, leaf, ancestor, descendant, parents, child, etc.
- The height of a rooted tree
- The definition of a binary tree, and in particular a binary search tree

- A balanced binary tree has 2^i nodes on all levels i apart from the highest level

---

In a number of applications, trees are used to model procedures that can be divided up into a sequence of stages, such as counting problems or searching or sorting algorithms. In this type of application, one vertex is singled out to represent the *start* of the process. A tree in which one vertex has been singled out in this way is called a **rooted tree** and the chosen vertex is called the **root** of the tree.

The topic of rooted trees has its own special terminology, which is used particularly in Computer Science. First, you will find that the vertices of a rooted tree are often referred to as **nodes**. The other terms are mainly derived from the similarity between a rooted tree and a *family tree*, depicting the descendants of a single ancestor.

We arrange the vertices of a rooted tree T in *levels*. We first put the root on level 0. For any positive integer i, the set of vertices in level i is the set of all vertices of the tree which are joined to the root by a path of length i. (The levels in the tree then correspond to *generations* in a family tree.) We know, from Lemma 3.4, that there is a unique path in the tree from the root r to any other vertex. Thus each vertex belongs to exactly one level.

The **height** of T is the length of a longest path in T starting at the root. Thus, if T has height h, then its vertices lie on levels $0, 1, 2, \dots, h$.

Let x and y be vertices of T. If the unique path from the root r to x in T passes through y, then y is called an **ancestor** of x and x is called a **descendant** of y. If the vertices y and x are adjacent on the path from r to x, then y is called the **parent** of x and x is called the **child** of y.

Example 3.5 The terminology of rooted trees

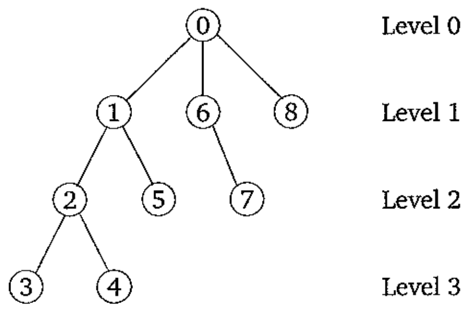


Figure 3.3: A rooted tree

In the tree shown in Figure 3.3,

- the vertex 0 is the root;
- the vertices $1, 6, 8$ are the children of 0;
- the vertex 6 is the parent of 7;
- the vertex 1 is an ancestor of $2, 3, 4$ and 5;
- vertex 1 is *not* an ancestor of any of the vertices $0, 6, 7, 8$;

- the vertex 4 is a descendant of 0, 1 and 2.

Theorem 3.5 *In a rooted tree, every vertex other than the root has exactly one parent.*

Proof. Let T be a tree rooted at r and let x be any other vertex of T. Then there is a unique path in T starting at r and ending at x, by Lemma 3.4. The parent of x must be a vertex of this path (because a parent is an ancestor) and it must be adjacent to x. Thus because the path is unique, x has a unique parent. ∎

It is not necessary for every vertex in the tree to have a child. The vertices with no children are called **end vertices** or **external nodes** of the tree. The other vertices are called **internal nodes.**

Example 3.6 The terminology of rooted trees (cont.)
The vertices $3, 4, 5, 7, 8$ are the external nodes of the rooted tree shown in Figure 3.3 and $0, 1, 2, 6$ are the internal nodes.

We can use the parent-child relationship to define the idea of *levels* in a rooted tree more precisely. Starting from the root at level 0, we obtain the **level** of each vertex x by adding 1 to the level of its parent. The **height** of the tree is the greatest of the levels, so for example the height of the rooted tree shown in Figure 3.3 is 3.

3.2.1 Binary trees

A **binary tree** is a rooted tree in which each internal node has exactly two children, one of which we call the **left child** and the other we call the **right child.** Binary trees arise as models of procedures in which two possibilities occur at each stage.

Example 3.7 A model using binary trees

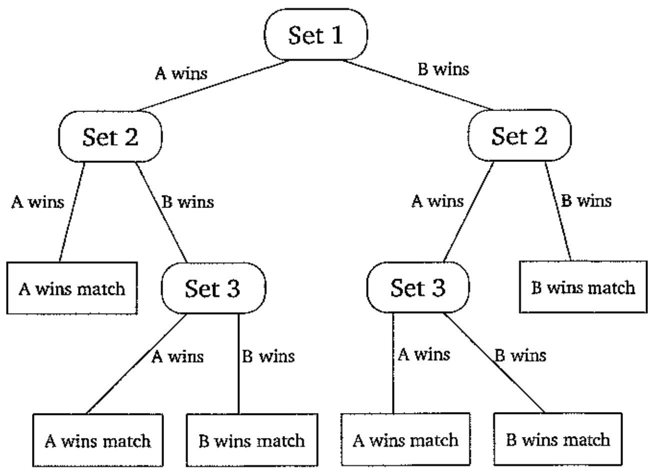


Figure 3.4: A tennis match

In a tennis match, two players A and B play up to three sets and the winner is the first player to win two sets. A binary tree to model the possible outcomes is shown in Figure 3.4.

The internal nodes in the binary tree of Figure 3.4, depicted in oval boxes, each represent the start of a set and the edges joining this node to its left and right child represent respectively the two possible outcomes for that set, a win for A or a win for B. The external vertices, depicted in rectangular boxes, represent the final outcome of the match as a result of the wins recorded on the unique path that leads from the root to that box.

The binary tree illustrated in Figure 3.4 has the property that its external nodes are *all on the highest level or the highest two levels.* Such a binary tree is called **balanced.**

If T is a balanced binary tree of height h, then we can compute the exact number of vertices on all levels apart from the highest level, that is on all levels i, for $0 \le i \le h-1$.

Theorem 3.6 *Let T be a balanced binary tree of height h. Then the number of vertices of T on level i is equal to 2^i, for all integers i, $0 \le i \le h-1$. Furthermore*

$$2^h + 1 \le |V(T)| \le 2^{h+1} - 1.$$

Proof. Since each vertex at level i has exactly two children at level $i+1$ for all levels $i = 0 \le i \le h-2$, it follows that the number of vertices at level $i+1$ is exactly twice the number of vertices at level i.

The only vertex at level 0 is the root, so the number of vertices at level 0 is $1 = 2^0$, hence we have exactly 2^i vertices at level i for all i where $0 \le i \le h-1$.

The number of vertices at level h is not fixed, but we know it must be at least[5] two and at most[6] 2^h. The number of vertices $|V(T)|$ in the tree is the sum of the number of vertices at all levels, thus

$$1+2+2^2+\ldots+2^{h-1}+2 \le |V(T)| \le 1+2+2^2+\ldots+2^{h-1}+2^h. \quad (3.1)$$

[5]If only one node at level $h-1$ has children.

[6]If all nodes at level $h-1$ have children.

From Theorem 2.1(d) we know that $1+2+2^2+\ldots+2^{h-1} = 2^h - 1$. Thus equation (3.1) becomes

$$(2^h - 1) + 2 \le |V(T)| \le (2^h - 1) + 2^h,$$

which in turn gives the required result that

$$2^h + 1 \le |V(T)| \le 2^{h+1} - 1. \blacksquare$$

3.3 Binary search trees

Learning objectives for this section

- How to store and find records in a binary search tree
- How to compute the height of a binary search tree with n records
- The maximum number of comparisons needed in order to find a record in a binary search tree with n records

We conclude the chapter by giving a description of one way in which binary trees can be used to construct an efficient storage and retrieval solution for a list of records which is to be kept on a computer.

Example 3.8 Storing a list of records
Suppose that a mail order company has a list of past customers stored on its computer in alphabetical order. When it receives an order, it wants to be able to check quickly whether this is from an existing customer, or whether this is a new customer that needs to be added to the database.

Let us assume we have N past customer records numbered $1, 2, 3, \ldots, N$, where record 1 holds the information of the customer first in alphabetical order, record 2 holds the information for the customer second in alphabetical order, etc.

One traditional way of storing the customer records would be in a line of N records as in Figure 3.5, and a simple check whether a new order is from a past customer would be to ask the computer to begin at the beginning of the list and compare the customer's name and address with each record on the existing database in turn until a match is found.

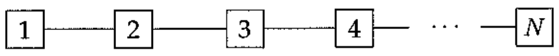


Figure 3.5: A simple linear list

This search method is not very efficient though, for if the order is from a new customer, then a computer using this simple method would have to check the name against every single record to verify that the name is not already on the database. Even in the case of an existing record, depending on the size of the database, the computer might have to make several thousand comparisons before the matching record is located. We shall see next how a binary tree can be used to design a much better storage solution for the company.

3.3.1 Subtrees of a binary tree

Binary trees have the important 'recursive' structural property that *if you take any internal node and all its descendants*, then these vertices and the edges joining them also form a binary tree, because the property that each internal node has exactly two children is preserved.

Suppose x is an internal node. Then the binary subtree that is formed by taking the *left child of x and all its descendants* is called the **left subtree** of x; similarly, the binary subtree formed by taking the *right child of x and all its descendants* is called the **right subtree** of x.

Example 3.9 Figure 3.6 shows the left subtree of the root "Set 1" of the binary tree in Figure 3.4.

We shall now show how this recursive property of binary trees can be used to effectively store an ordered list of records in a binary tree. The resulting binary tree is going to be balanced and retrieval

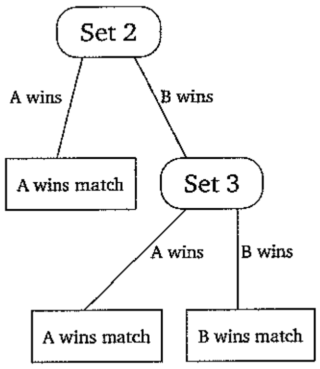


Figure 3.6: The left subtree of the root in the 'tennis match'-tree of Figure 3.4

of records will generally be much more efficient than the simple method we used in Example 3.8.

3.3.2 Storing data in a binary search tree

We first explain how data can be stored efficiently at the internal nodes of a binary tree. An ordered list of records is stored by a recursive procedure in which we first store the 'middle' record at the root and then recursively store the list of all smaller records in the left subtree and the list of all larger records in the right subtree. The resulting binary tree structure is known as a **binary search tree** or simply a **BST**.

Algorithm for storing data in a binary search tree
Suppose we have to store N records and that the first record to be stored is #1 and the last record is #N:

1. **The root** r is the record # $\lfloor (1+N)/2 \rfloor$.
2. We then store all the records that come *before* the root in the *left* subtree of r and all records that come *after* the root in the *right* subtree of r in the following way:
   If the first record in the subtree is #a and the last record is #b, then the **root of the subtree** is
   $$\# \lfloor (a+b)/2 \rfloor .$$
3. For each of the two subtrees we then divide the remaining records into two halves again, those that come before the root of the subtree and those that come after the root of the subtree. These are then stored in left and right subtrees to the roots of the subtrees by the process in step 2 again.
4. We continue dividing and adding new subtrees until each subtree consists of only one record. This completes the internal vertices of the binary tree.
5. The external nodes are empty boxes representing positions in the list between each existing pair of records (and before and after the present first and last record) in which a new record could be entered in correct alphabetical order.

Example 3.10 Storing data in a binary search tree

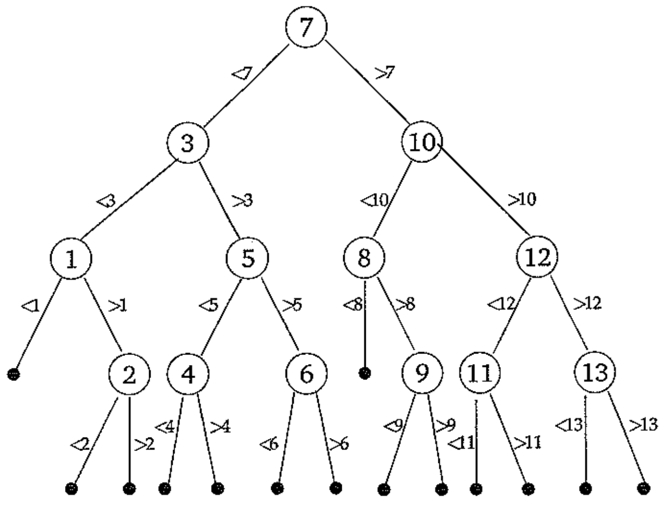


Figure 3.7: A binary search tree storing 13 records numbered $1, 2, \ldots, 13$

Figure 3.7 shows how we would store 13 records in a binary tree constructed as described above.

We imagine that the records have been put in alphabetical order and then numbered from 1 to 13. The root is at # $\lfloor (1 + 13)/2 \rfloor = 7$.

Records $\{1, 2, 3, 4, 5, 6\}$ are stored in the left subtree and $\{8, 9, 10, 11, 12, 13\}$ in the right subtree of #7. The roots of the left and right subtrees are records # $\lfloor (1 + 6)/2 \rfloor = 3$ and # $\lfloor (8 + 13)/2 \rfloor = 10$ respectively.

The left subtree of #3 contains the records $\{1, 2\}$ and its right subtree contains the records $\{4, 5, 6\}$. These subtrees are therefore rooted at #1 and #5 respectively. Similarly, the left subtree of #10 contains the records $\{8, 9\}$ and its right subtree contains the records $\{11, 12, 13\}$. These subtrees are therefore rooted at #8 and #12 respectively.

The left subtree of #1 is empty and is therefore an external node representing a position in which a data item that precedes #1 in alphabetical order could be placed. The right subtree of #1 contains just the record #2. Its subtrees consist of two external nodes representing the positions in which a new data item could be placed between #1 and #2 or between #2 and #3. The remaining subtrees are constructed in a similar way.

3.3.3 The height of a binary search tree

When storing the records at the internal nodes of a binary search tree, at any given node we always divide into two parts, of as equal size as possible, the list of records to be stored in the subtree rooted at this node.

Example 3.11 Consider the node 12 in the BST of Figure 3.7. The subtree rooted at 12 is to contain an odd number of records, namely the sublist $\{11, 12, 13\}$. 12 becomes the root and the two others are divided equally among the two subtrees of 12 which therefore have the same height.

Consider the node 8 in the BST of Figure 3.7. The subtree rooted at 8 is to contain an even number of records, namely the sublist $\{8, 9\}$. 8 becomes the root and the other node is to be divided equally among the two subtrees of 8, which is only possible by putting it in one of the subtrees and none in the other. The right subtree of 8 will thus get one more internal node than the left and the right subtree has height 1 while the left subtree has height 0.

Consider also the node 3 in the BST of Figure 3.7. The subtree rooted at 3 is to contain an even number of records, namely the sublist $\{1, 2, 3, 4, 5, 6\}$. 3 becomes the root and the other five are to be divided equally among the two subtrees of 3, i.e. the right subtree will thus get one more internal node than the left subtree. We note that the right and left subtrees of 3 have the same height though.

We note that when the list to be stored in a subtree has an odd number of records, this will make the right and left subtree of the node have the same number of internal nodes and thus the same height. When the list to be stored in a subtree has an even number of records, this will make the right subtree have one more internal node than the left subtree of the node and the right subtree may therefore have one more level than the left subtree. We may thus conclude that in a binary search tree all the external nodes occur either on just the highest level or on the highest two levels. In other words:

Lemma 3.7 *A binary search tree is balanced.*

This property enables us to calculate from the size of the data set to be stored at the internal nodes of the BST, the number of levels that the binary search tree will have.

Suppose the database contains N records, which we must store in the internal nodes of a binary tree of height h. The BST has thus got levels $0, 1, 2, \ldots, h$.

We start by estimating the number of internal nodes in a binary search tree of height h. All internal nodes occur at levels $0, 1, 2, \ldots, h-1$. Hence the *maximum* number of internal nodes in the tree is

$$1 + 2^1 + 2^2 + \ldots + 2^{h-1} = \tfrac{2^h - 1}{2-1} = 2^h - 1.$$

This maximum occurs when *all* the vertices at level $h-1$ are internal nodes.

Some of the vertices on level $h-1$ may be external nodes, as we had in Figure 3.7 for example. However, because a binary search tree is always balanced, we know that no external node will occur at any level before $h-1$. Hence all nodes on levels $0, 1, \ldots, h-2$ are internal, so there are always a minimum of

$$1 + 2^1 + 2^2 + \ldots + 2^{h-2} = 2^{h-1} - 1$$

internal nodes.

We must store N records in the BST and all of these must be at internal nodes. Thus using the lower limit on the number of internal nodes in a tree of height h and the upper limit found in the previous two paragraphs, we must have that

$$2^{h-1} - 1 < N \leq 2^h - 1$$

in order to be able to store the N records in the tree. This inequality enables us to find the height of tree needed. Adding 1 to each part of the inequality, gives

$$2^{h-1} < N+1 \leq 2^h.$$

Recall that if $y = 2^x$, then $x = \log_2 y$. Hence the height h of the tree is the positive integer h such that

$$h-1 < \log_2 (N+1) \leq h.$$

Using the ceiling function, we can express this as

Theorem 3.8 *The height h of a binary search tree with N records stored at internal nodes is*

$$h = \lceil \log_2 (N+1) \rceil .$$

important !

3.3.4 Finding a record in a binary search tree

The algorithm for retrieving a record from a BST is recursive.

Algorithm for retrieving a record from a BST

We start the search by comparing our target name with the root of the BST. There are three possible conclusions: (1) the computer tells us that the target matches the root, in which case the search is concluded; or (2) that the target comes alphabetically before the root, in which case we search for the record in the left subtree; or (3) that it comes alphabetically after the root, in which case we search for the record in the right subtree.

If we do not get a match with the root, then we know which subtree should contain the target and we search that subtree only. Thus after just one comparison, we reduce the total number of records to be searched by 50%. We now repeat this procedure in the indicated subtree, comparing our target with its root and then, if we don't get a match, moving to its left or right subtree as indicated.

Example 3.10 (cont.) Retrieving a record from a binary search tree
Suppose our target matches record 9 in Figure 3.7. The search is conducted as follows.

The computer compares the target with the root 7. Since $9 > 7$, it goes to the right subtree[7] of 7.

Now the computer compares the target with 10. Since $9 < 10$, it moves to the left subtree[8] of 10 and compares 9 with 8. Since $9 > 8$, it moves to the right subtree of 8 and achieves a match with 9. Thus the record is identified in 4 comparisons.

[7] Note that the left subtree of 7 can be completely disregarded after this comparison, so with just one comparison, we have halved the number of records to be searched.

[8] The number of records to be searched has been halved once more.

Now suppose we have a target that is not on our existing database and comes alphabetically between the records entered at 3 and 4. The computer compares it with 7 and moves to the left subtree of 7; it compares it with 3 and moves to the right subtree of 3; it compares it with 5 and moves to the left subtree of 5; it compares it with 4 and moves to the left child of 4. But this is an empty box. This tells us that the target is not on the current list and should be inserted immediately before the record 4.

Notice that only 4 comparisons[9] were needed to establish that a record was not in the list. This is considerably better than the 13 comparisons which were needed in the simple linear search method of checking all records which we considered in Example 3.8.

[9]With records 7, 3, 5 and 4.

The maximum number of comparisons needed to find a record in a BST

The computer makes its first comparison with the root at level 0 and then makes one comparison at each level until either the target is matched or it is verified that it is not in the data set. Hence at worst, the computer makes a comparison on each of the levels $0, 1, \ldots, h-1$, giving h comparisons altogether. Thus combining this with Theorem 3.8 we have the following Theorem.

Theorem 3.9 *Suppose we store an ordered list of N records labelled $1, 2, 3, \ldots, N$ at the internal nodes of a binary search tree of height h. The maximum number of comparisons needed to retrieve a record from the binary search tree is*

$$h = \lceil \log_2 (N+1) \rceil .$$

important

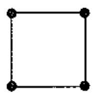
Example 3.10 (cont.) Consider again the binary search tree of Figure 3.7. We have $N = 13$ here. We need a binary tree of height 4 to store this data because $2^3 < 13 + 1 \leq 2^4$. The internal nodes of the binary search tree are on levels $0, 1, 2, 3$ and the height of the tree is $h = 4$. As we saw, a worst case requires four comparisons to match the target.

3.4 Exercises on Chapter 3

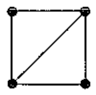
3.4.1 True/False questions

In each of the following questions, decide whether the given statements are true or false.

1. A tree is a connected graph without loops.
2. A connected graph without loops is a tree.
3. A connected graph without cycles is a tree.
4. If a graph G is a tree, then G is simple.
5. A path graph of length n is a tree with n vertices.
6. A tree with n vertices has $n - 1$ edges.
7. The following graph has two non-isomorphic spanning trees.



8. The following graph has two non-isomorphic spanning trees.



ignore this

9. Suppose that a binary search tree is designed to store an ordered list of 32 records at its internal nodes.
   (i) The height of the binary search tree is 5.
   (ii) The root of the binary search tree is 17.
   (iii) The records at level 1 of the binary search tree are 8 and 24.
   (iv) Record 10 is an ancestor of record 20 in the tree.
10. A binary search tree of height h is balanced; this means that it has 2^i records at all levels.

3.4.2 Longer exercises

ignore

Question 1
Let G be a graph.

(a) What two properties must G satisfy in order to be a *tree*?

(b) Suppose that every pair of vertices of G are joined by a unique path in G. Must G be a tree? Justify your answer.

ignore

Question 2

(a) Draw the tree T with $V(T) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E(T) = \{v_1v_2, v_2v_3, v_2v_4, v_4v_5, v_4v_6\}$.

(b) Construct all the non-isomorphic trees on seven vertices which can be obtained by attaching a new vertex of degree one to T.

(c) Explain briefly why the trees you obtain in (b) are not isomorphic to each other.

ignore

Question 3
Let G be the connected graph with $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{v_1v_2, v_2v_3, v_3v_4, v_4v_2\}$.

(a) Find all spanning trees of G.

(b) How many non-isomorphic spanning trees does G have?

Question 4
Let T be a rooted tree with root r.

(a) Explain how the nodes of T are partitioned into *levels*.

(b) What does it mean to say that T has *height* h?

(c) What does it mean to say that a node of v is an *external node*?

Question 5
A *ternary tree* is a rooted tree in which each internal node has exactly three children.

(a) Draw a ternary tree of height 2 in which each external node lies on level 2.

(b) Let T be a ternary tree of height h in which all external nodes lie on level h. Determine the number of nodes on level i for all integers i, $0 \leq i \leq h$.

Question 6
A restaurant offers a set meal consisting of a choice of one of two starters S_1, S_2; followed by a choice of one of three main courses M_1, M_2, M_3; followed by a choice of one of two desserts D_1, D_2.

(a) Construct a rooted tree to model the outcomes of ordering a meal. (Make the first three levels represent the three different courses.)

(b) How many different meals can be chosen?

Question 7

(a) Design a binary search tree for an ordered list of 11 records.

(b) What is the maximum number of comparisons that the computer would have to make to match any existing record?

(c) Which existing records would require the maximum number of comparisons?

Question 8
A mail order company has 5,000,000 records on its database. Calculate the maximum number of comparisons that would need to be made to match a target with any record in the database.