Solutions

Each answer shows the steps, then the **result**.

2.1

- start at the node given by the **start pointer**
- compare the data at the current node with the item being searched for
- if it does not match, move to the node given by that node's **pointer**, and repeat until it matches or the pointer is **null**

2.2

- $45 < 50$, so go **left** to 30
- $45 > 30$, so go **right** to 40; $45 > 40$, so go right again, and that pointer is **null**
- so 45 becomes the **right child of 40**

2.3

- 20, 30, 40, 50, 60, 70, 80
- an in-order traversal of a binary search tree gives the values in **ascending order**

2.4

- any two: it is a set of **vertices** joined by **edges**; an edge may be **directed** or undirected; an edge may be **weighted**; a vertex may have any number of edges; the graph may contain **cycles**

2.5

- linear search $\mathbf{O(n)}$
- binary search $\mathbf{O(\log n)}$

3.1

- initialise the current pointer to `StartPointer` and a `Found` flag to `FALSE`
- a loop that continues while the pointer is not -1 **and** the item has not been found
- compare `Data[CurrentPointer]` with `SearchItem`
- on a match, set `Found` to `TRUE`
- otherwise follow the pointer: `CurrentPointer <- Pointer[CurrentPointer]`

```
CurrentPointer <- StartPointer
Found <- FALSE
WHILE CurrentPointer <> -1 AND Found = FALSE
    IF Data[CurrentPointer] = SearchItem THEN
        Found <- TRUE
    ELSE
        CurrentPointer <- Pointer[CurrentPointer]
    ENDIF
ENDWHILE
```


- a loop that tests only for the item, and not for the null pointer, runs off the end of the list when the item is absent

3.2

(a) pre-order: 50, 30, 20, 40, 70, 60, 80

- post-order: 20, 40, 30, 60, 80, 70, 50

(b) **pre-order** visits the node before its subtrees, so it is used to **copy** a tree, or to write it out in a form that rebuilds the same shape

- **in-order** gives a binary search tree's values in **ascending order**, so it is used to output them sorted

3.3

- declare a 1-D array of a fixed size, and an integer pointer `Top`, initialised to show an empty stack
- to **push**: first check the stack is not **full**; increment `Top`, then store the new value at that index
- to **pop**: first check the stack is not **empty**; read the value at `Top`, then decrement `Top`
- the value popped is the one pushed most recently, which is what makes the structure **LIFO**; the data is not erased, it is simply no longer within the pointer

3.4

- a **graph**
- the towns are **vertices** and the roads are **edges**, the one-way roads are **directed** edges and the lengths are **weights** on the edges
- a tree could not represent it, because the loops in the network are **cycles** and a tree has none; and a road may connect any two towns, whereas a tree allows each node only one parent

3.5

(a) every value inserted is larger than all those before it, so it goes to the **right** of every existing node

- the tree has **one branch**: every node has only a right child, so it is effectively a **linked list**

(b) $\mathbf{O(n)}$

- a balanced tree halves the number of remaining nodes at each step, giving $O(\log n)$; this tree removes only **one** node per step, so the search has to walk the whole chain