

19.1.1 ADT algorithms - linked lists, binary trees, graphs, and Big O

Name: _____ Class: _____ Date: _____

Total: 32 marks

KEY WORDS binary tree 二叉树 • binary search 二分查找 • bubble sort 冒泡排序 • queue 队列
• linked list 链表

Objective

Build the skills to answer the **Abstract Data Type** 抽象数据类型 questions in 19.1. Sheet 19.1 covers the searches, the sorts and the comparison criteria well. The other half of the unit is missing from it entirely: the words binary tree, linked list, graph, dictionary and traversal occur **zero** times in it, although two of the five statements are about writing algorithms for exactly those structures.

You must be able to:

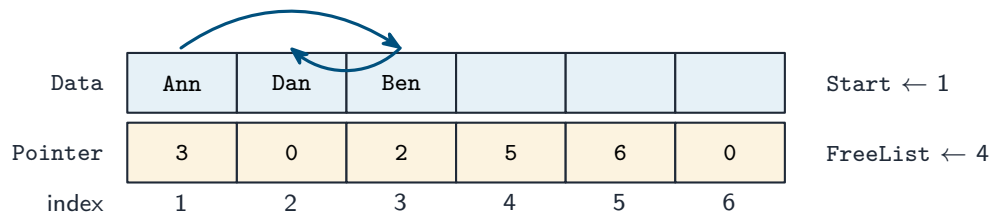
- write algorithms to **find** an item in a **linked list** 链表 and in a **binary tree** 二叉树
- write algorithms to **insert** an item into a stack, a queue, a linked list and a binary tree, and to **delete** from the first three
- show that a **graph** 图 is an ADT, describe its key features and justify its use for a given situation
- show how one ADT can be **implemented from another**, or from a built-in type
- compare algorithms using **Big O** notation for time and space complexity

1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

- Finding an item in a linked list

follow the pointers: $1 \rightarrow 3 \rightarrow 2 \rightarrow 0$ gives Ann, Ben, Dan



a pointer of 0 marks the end of a list; unused cells are chained into the free list $4 \rightarrow 5 \rightarrow 6$
to insert Cal: take cell 4 from the free list, store Cal, set $\text{Pointer}[4] \leftarrow 2$ and $\text{Pointer}[3] \leftarrow 4$

A linked list has no index, so there is exactly one way in: start at the head and follow the pointers.

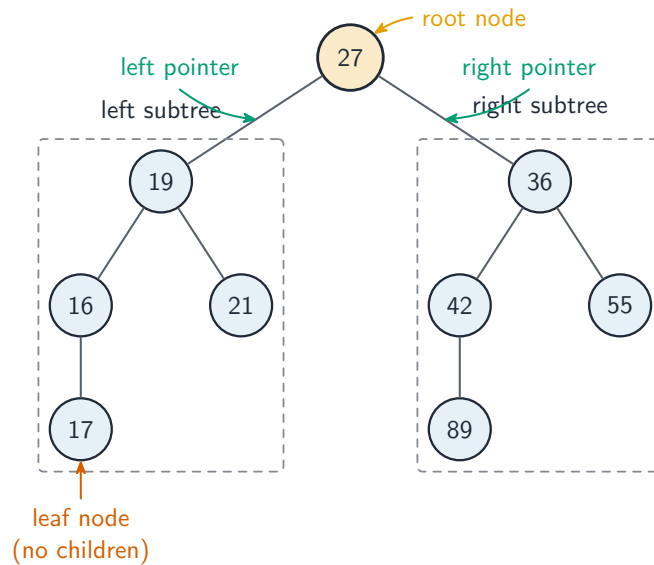
```

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

```

- The loop condition needs **both** tests: the value may be found, or the list may run out. Leaving out the null test walks off the end of the list.
- Because every node must be visited in the worst case, a linked-list search is $O(n)$ – there is no way to jump to the middle, which is exactly what an array allows and a list does not.

■ A binary tree: inserting, searching, traversing



In a **binary search tree** every value in the left subtree is **smaller** than the node and every value in the right subtree is **larger**. That one rule drives both algorithms.

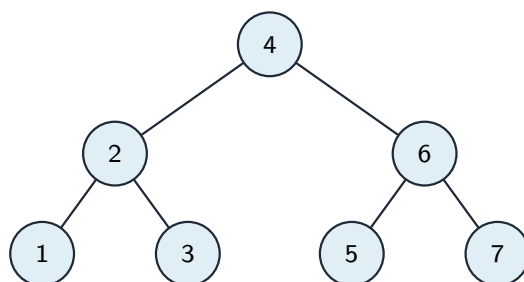
To insert: start at the root; go **left** if the new value is smaller and **right** if it is larger; repeat until the pointer you would follow is **null**; put the new node there.

To search: the same walk. It succeeds when the values match, and fails when the pointer to follow is null.

Worked example. Insert 50, 30, 70, 20, 40, 60, 80 in that order, then insert 45.

50 becomes the root. 30 goes left of it, 70 right. 20 goes left of 30, 40 right of 30; 60 left of 70, 80 right of 70.

For 45: $45 < 50$ so go left to 30; $45 > 30$ so go right to 40; $45 > 40$ and 40's right pointer is null, so **45 becomes the right child of 40**.



pre-order: 4 2 1 3 6 5 7

in-order: 1 2 3 4 5 6 7

post-order: 1 3 2 5 7 6 4

The three **traversals** differ only in *when the node itself is visited*:

Traversal	Order	On the tree above
in-order	left, node , right	20, 30, 40, 50, 60, 70, 80
pre-order	node , left, right	50, 30, 20, 40, 70, 60, 80
post-order	left, right, node	20, 40, 30, 60, 80, 70, 50

- An **in-order traversal of a binary search tree gives the values in ascending order**. That is the single most useful fact about the structure, and it is why a BST is a sorted structure that is still cheap to insert into.

■ A graph, and why it is not a tree

A **graph** is a set of **vertices** 顶点 (nodes) joined by **edges** 边. Its key features:

- an edge may be **directed** (one way) or **undirected**, and may carry a **weight** – a distance, a cost, a time;
- a vertex may have **any number** of edges, and there may be **cycles**;
- there is no root and no parent-child rule.

A tree is a special case of a graph: connected, with no cycles, and every node having exactly one parent. So use a **graph** whenever the data is about **relationships between pairs** that those restrictions would break – a road network where routes loop back, a social network where friendship is mutual, web pages linking to each other.

It is stored either as an **adjacency matrix** (a grid of every vertex against every other, simple but wasteful when there are few edges) or as an **adjacency list** (each vertex holding a list of its neighbours, compact when the graph is sparse).

■ One ADT built from another

An ADT is defined by its **operations**, not by how it is stored, so the same behaviour can be built several ways:

ADT	Built from	How
stack	a 1-D array	plus one integer Top ; push increments it and writes, pop reads and decrements
queue	a 1-D array	plus Front , Rear and a count, with the pointers wrapping round the end
linked list	two 1-D arrays	one for the data, one for the index of the next node, plus a start pointer and a free list
binary tree	three 1-D arrays	data, left-pointer and right-pointer, plus a root pointer
dictionary	a hash table , or two parallel arrays	a key is hashed to an index, and the value stored there

■ Comparing algorithms with Big O

Big O states how the work grows as the number of items n grows, ignoring constants.

Operation	Time complexity
linear search	$O(n)$
binary search	$O(\log n)$
bubble sort, insertion sort	$O(n^2)$ worst case
insertion sort on nearly sorted data	$O(n)$ best case
search a balanced binary search tree	$O(\log n)$
search a degenerate binary search tree	$O(n)$
push or pop a stack	$O(1)$

- That last pair is worth stating carefully: a BST is only $O(\log n)$ while it stays **balanced**. Inserting already-sorted data gives every node one child, and the tree degenerates into a linked list.

2 Practice

Now use the methods above.

2.1 Describe how an item is found in a linked list. [3]

2.2 A binary search tree contains 50, 30, 70, 20, 40, 60 and 80, inserted in that order. State where the value 45 is placed, and explain how you decided. [3]

2.3 Give the **in-order** traversal of that tree, and state what is true of the order it produces. [2]

2.4 State **two** key features of a **graph**. [2]

2.5 State the time complexity, in Big O notation, of a linear search and of a binary search. [2]

3 Exam-style questions

3.1 Write an algorithm, using pseudocode, that searches a **linked list** for a value **SearchItem**.

The list is held in the arrays **Data** and **Pointer**, with **StartPointer** giving the index of the first node and a null pointer written as -1 . [5]

3.2 (a) Give the **pre-order** and **post-order** traversals of the tree in 2.2. [2]

(b) State **one** use of a pre-order traversal and **one** use of an in-order traversal. [2]

3.3 Describe how a **stack** can be implemented using a 1-D array, including what happens on a push and on a pop. [4]

3.4 A company stores the roads between towns. Some roads are one-way, and each road has a length. There are loops in the network.

Name a suitable ADT and **justify** your choice. [3]

3.5 A binary search tree is built by inserting values that are already in ascending order.

(a) Describe the shape of the resulting tree. [2]

(b) State the time complexity of searching it, and explain why it differs from that of a balanced tree. [2]
