

Abstract data types (HL only)

International Baccalaureate · IB Diploma · Computer Science · HL

Scope and prerequisites

This handout teaches Theme B4, Abstract data types, for the course first assessed in 2027, using Python. The whole theme is Higher Level only (AHL).

Prerequisites: B2 Programming (arrays, stacks, queues, recursion) and B3 Object-oriented programming (classes and references).

Abstract data types

■ Why does the structure matter?

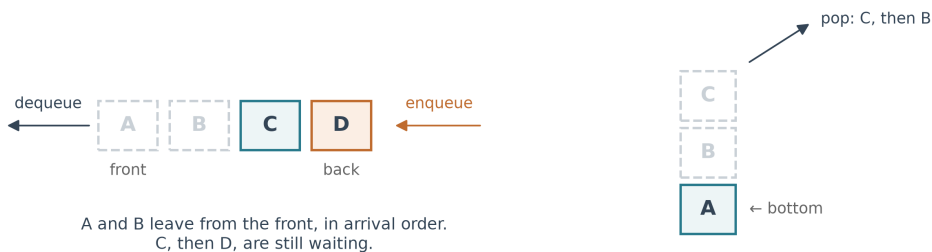
A music app holds a playlist of 10 000 songs. Adding a song in the middle of an array means shifting thousands of items along; searching an unsorted list for a title checks them one by one. Choosing a different structure makes both jobs fast. This theme is about those structures.

■ Interface and implementation

An **abstract data type** (ADT) is defined by the values it holds and the operations on them, not by how it is stored. The stack from B2 is an ADT: **push**, **pop**, **peek** and **isEmpty**, last in first out. It can be **implemented** with an array or with a linked list; code that uses the stack does not need to know which.

Queue (FIFO): enqueue A, B, C; dequeue twice; enqueue D

Stack (LIFO): push A, B, C; pop twice



Same items, two ADTs: the operations, not the storage, decide the order out

ADT	Key operations	Order or rule
stack	push, pop, peek	LIFO
queue	enqueue, dequeue	FIFO
linked list	insert, delete, traverse	follows the pointers
binary search tree	insert, search, traverse	left smaller, right larger
set	add, remove, contains	no duplicates, no order
hash table	put, get by key	position from a hash function

Linked lists

■ Nodes and pointers

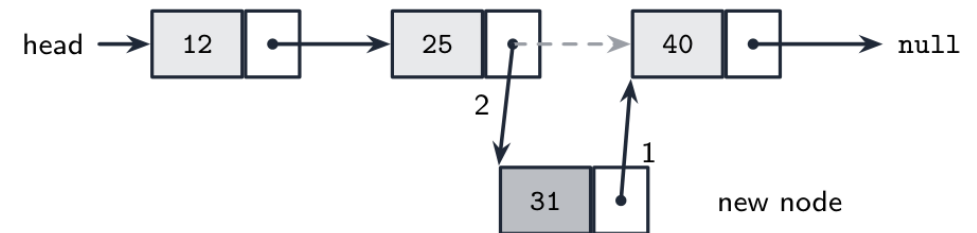
A **linked list** is a chain of **nodes**. Each node holds data and a **pointer** (a reference) to the next node. The list is reached through its **head**; the last node points to **None** (null).

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

To **traverse** the list, follow the pointers from the head until None:

```
def show(head):
    current = head
    while current is not None:
        print(current.data)
        current = current.next
```

■ Inserting and deleting



Insert after a node: first point the new node onward, then redirect the old pointer

To insert 31 after the node holding 25 (called `prev`):

```
new = Node(31)
new.next = prev.next      # step 1: new node points to 40
prev.next = new           # step 2: 25 now points to the new node
```

The order matters: doing step 2 first would lose the pointer to 40 and the rest of the list.

To delete the node after `prev`: `prev.next = prev.next.next`. The removed node is no longer reachable, and Python’s garbage collector frees its memory.

No items are shifted, so insertion and deletion are $O(1)$ once the position is found; but finding a position means traversing from the head, $O(n)$. There is no index: reading the 500th item takes 500 steps.

■ Variants

Variant	Each node points to	Advantage
singly linked	the next node	least memory
doubly linked	the next and the previous node	traverse both ways; delete a node without its predecessor
circular	the next; the last points back to the first	round-robin tasks, such as taking turns

■ Linked list or array?

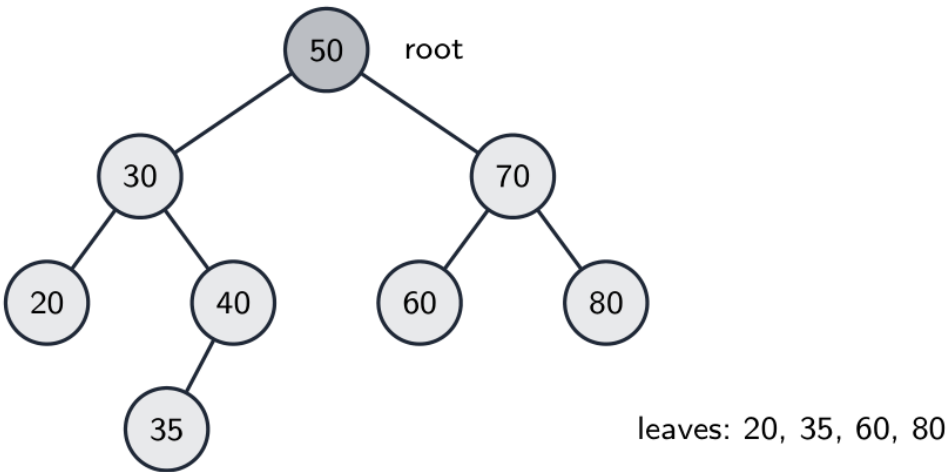
	Array	Linked list
access by index	$O(1)$	$O(n)$
insert or delete in the middle	$O(n)$: shift items	$O(1)$ at a known node
size	fixed (static) or resized	grows one node at a time
memory	data only	data plus a pointer per node

Binary search trees

■ Structure

A **binary tree** is made of nodes that each have at most two **children**, left and right. The top node is the **root**; a node with no children is a **leaf**.

. In a **binary search tree** (BST), every key in a node’s left subtree is smaller than the node’s key, and every key in its right subtree is larger.



Built by inserting 50, 30, 70, 20, 40, 60, 80, 35 in that order

■ Inserting and searching

To insert or find a key, start at the root and go left if the key is smaller, right if larger, until a match or an empty place.

Inserting 35: $35 < 50$, go left; $35 > 30$, go right; $35 < 40$, go left; empty, so 35 becomes the left child of 40.

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def contains(node, key):
    if node is None:
        return False
    if key == node.key:
        return True
    if key < node.key:
        return contains(node.left, key)
    return contains(node.right, key)
```

Each comparison drops one subtree, so a **balanced** tree of n keys is searched in $O(\log n)$. Keys inserted already sorted (10, 20, 30, ...) make a tree that is a single

chain: search becomes $O(n)$, no better than a list.

■ Traversals

A **traversal** visits every node once. The three depth-first orders differ in when the node itself is visited:

Traversal	Order	For the tree above
pre-order	node, left, right	50, 30, 20, 40, 35, 70, 60, 80
in-order	left, node, right	20, 30, 35, 40, 50, 60, 70, 80
post-order	left, right, node	20, 35, 40, 30, 60, 80, 70, 50

In-order traversal of a BST gives the keys in **sorted order**. Pre-order copies a tree (insert in that order to rebuild it); post-order deletes one (children before parents).

```
def in_order(node):
    if node is not None:
        in_order(node.left)
        print(node.key)
        in_order(node.right)
```

Sets

■ No duplicates, no order

A **set** holds distinct values with no order: adding a value already present changes nothing. Its main operation is a fast membership test.

```
a = {"maths", "physics", "cs"}
b = {"cs", "art"}
a | b      # union: {"maths", "physics", "cs", "art"}
a & b      # intersection: {"cs"}
a - b      # difference: {"maths", "physics"}
"art" in a  # False
```

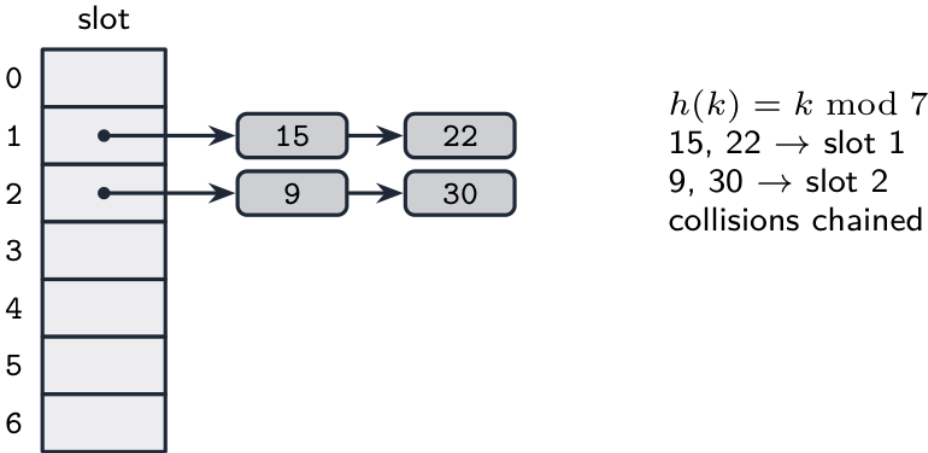
Operation	Result
union	in either set
intersection	in both sets
difference	in the first but not the second
subset	every item of one is in the other

Python sets are built on hash tables, so `in` is $O(1)$ on average, against $O(n)$ for a list.

Hash tables

■ Position from the key

A **hash table** stores key-value pairs in an array. A **hash function** turns each key into a slot index, so lookup is $O(1)$ on average under suitable hashing and load assumptions; collisions still require comparisons and worst-case lookup can be $O(n)$. A Python `dict` is a hash table.



Two keys in one slot is a collision; chaining keeps a list in the slot

With 7 slots and $h(k) = k \bmod 7$: $15 \bmod 7 = 1$, $22 \bmod 7 = 1$, $9 \bmod 7 = 2$, $30 \bmod 7 = 2$.

■ Collisions

A **collision** happens when two keys hash to the same slot. Two common fixes:

Method	How it works	15, 22, 9, 30 with 7 slots
chaining	each slot holds a linked list of items	slot 1: 15 $\rightarrow$ 22; slot 2: 9 $\rightarrow$ 30
linear probing	try the next slot until an empty one	15 $\rightarrow$ 1; 22 $\rightarrow$ 2; 9 $\rightarrow$ 3; 30 $\rightarrow$ 4

With linear probing, a search for 30 starts at slot 2 and steps on until it finds 30 or an empty slot.

A good hash function is fast and spreads keys evenly. The **load factor** $\frac{\text{items}}{\text{slots}}$ is as it rises, collisions grow, so the table is enlarged and every key rehashed.

Practice

1. Explain what is meant by an abstract data type, using a queue as the example.
2. A linked list holds $8 \rightarrow 14 \rightarrow 21 \rightarrow \text{None}$. Write the two statements that insert a node holding 17 after the node **p** that holds 14, in the correct order.
3. Explain why reading the k th item is slower in a linked list than in an array.
4. Give **one** use of a circular linked list.
5. Draw the BST made by inserting 40, 20, 60, 10, 30, 50, 70, 25.
6. For your tree in question 5, give the pre-order, in-order and post-order traversals.
7. State the number of comparisons needed to find 25 in your tree.
8. Explain what happens to search time if the keys 10, 20, 30, 40, 50 are inserted into a BST in that order.
9. $A = \{1, 3, 5, 7\}$ and $B = \{3, 4, 5\}$. State $A \cup B$, $A \cap B$ and $A - B$.
10. Insert the keys 12, 19, 5, 26 into a hash table of 7 slots with $h(k) = k \bmod 7$, using linear probing. State the slot of each key.
11. Explain why a hash table becomes slower as its load factor rises.

Answers.

1. An ADT is defined by its values and operations, not by its storage. A queue is defined by enqueue at the rear and dequeue from the front (FIFO), and can be built from an array or a linked list without changing the code that uses it.
2. **new.next** = **p.next** then **p.next** = **new**, where **new** = **Node(17)**.
3. A linked list has no index: the program must follow the pointers from the head, k steps. An array computes the item's address directly.
4. E.g. giving each player a turn in a game, or round-robin scheduling of processes.
5. Root 40; 20 left, 60 right; 10 and 30 under 20; 50 and 70 under 60; 25 the left child of 30.
6. Pre-order 40, 20, 10, 30, 25, 60, 50, 70. In-order 10, 20, 25, 30, 40, 50, 60, 70. Post-order 10, 25, 30, 20, 50, 70, 60, 40.
7. 40, 20, 30, 25: **4** comparisons.
8. Each key is larger than the last, so each goes to the right: the tree is one chain. A search may visit every node, $O(n)$, not $O(\log n)$.
9. $A \cup B = \{1, 3, 4, 5, 7\}$; $A \cap B = \{3, 5\}$; $A - B = \{1, 7\}$.
10. $12 \bmod 7 = 5$: slot 5. $19 \bmod 7 = 5$: taken, slot 6. $5 \bmod 7 = 5$: 5 and 6 taken, wrap round to slot 0. $26 \bmod 7 = 5$: 5, 6, 0 taken, slot 1.
11. More slots are full, so more keys collide; chains grow longer or probing visits more slots, and a search tends towards $O(n)$.

Exam tips

- When asked to “describe” an ADT, give its operations and its rule (LIFO, FIFO, ordered, no duplicates), not code.
- Linked-list insertion answers must give the pointer changes **in order**: new node first, then the old pointer.
- Draw BSTs neatly with smaller keys on the left; check by reading the in-order traversal, which must come out sorted.
- Learn the three traversals as node position: **pre** before the children, **in** between them, **post** after them.
- For hash tables, show the arithmetic ($k \bmod n$) for every key, and wrap round from the last slot to slot 0 when probing.
- Compare structures by Big O of the operation the question names: access, search, insert or delete.