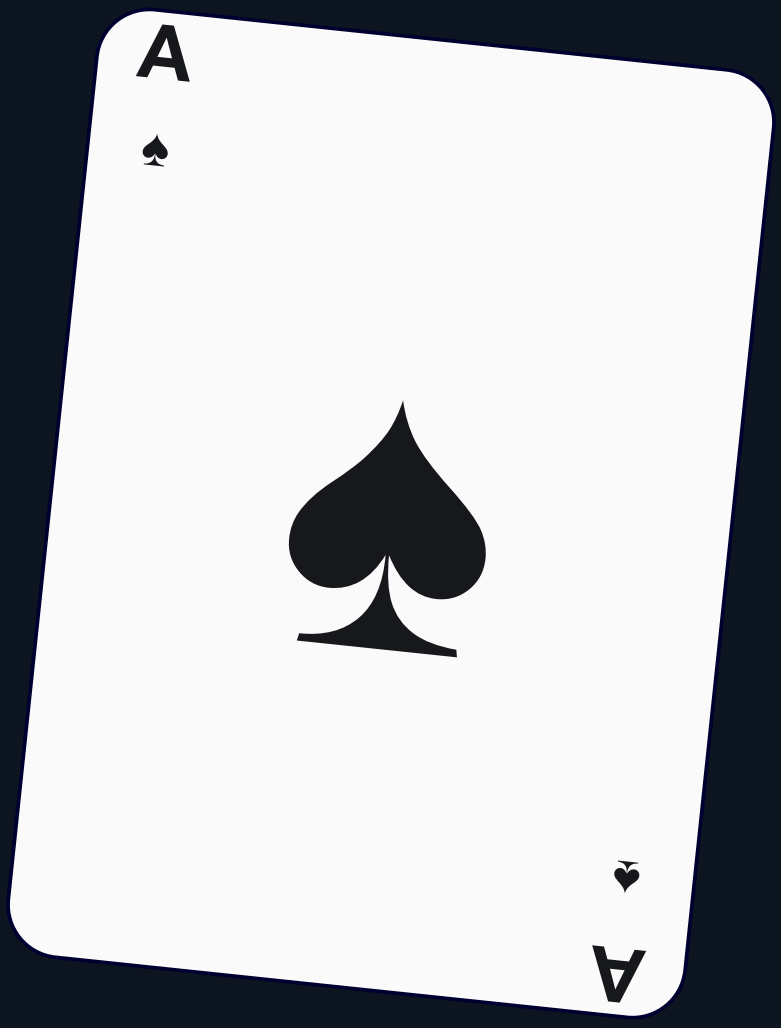


ACE MY ALGORITHMS

The Complete Reference

17 Cambridge 0478 / 9618 topics — pseudocode and Python, one page at a time

Don't just read it — try it.

Every algorithm in this book has a free, interactive, step-by-step version online — watch the code actually run as you work through it.

acemyalgorithms.com

Free to share with students and colleagues.

CONTENTS

Selection 3

FOR Loops 4

WHILE & REPEAT 5

Arrays 6

Linear Search 7

Binary Search 8

Bubble Sort 9

Insertion Sort 10

Functions and Procedures 11

Stacks — OOP (pointers) 12

Stacks — Array (exam-style) 13

Queues — OOP (pointers) 14

Queues — Array (exam-style) 15

Linked Lists — OOP (pointers) 16

Linked Lists — Array (exam-style) 17

Binary Trees — OOP (pointers) 18

Binary Trees — Array (exam-style) 19

OOP — Classes, Attributes, Methods and Objects 20

OOP with Inheritance — Building a Subclass 21

Recursion 22

File Processing 23

Selection

IF statements and CASE: branching logic — Cambridge 0478/9618

Selection is one of the three building blocks of every program, alongside sequence and iteration: it lets a program take a different path depending on a condition. IF...THEN...ELSE runs one branch or the other, never both.

Nesting an IF inside another branch lets a program check a second condition only once the first has already been decided. CASE OF offers a different shape for the same idea: a single variable tested against a list of values or ranges in order — the first match wins and every branch after it, including OTHERWISE, is skipped.

CAMBRIDGE PSEUDOCODE

Simple IF

```
DECLARE ChallengerScore : INTEGER
DECLARE ChampionScore : INTEGER
OUTPUT "Enter the challenger's score: "
INPUT ChallengerScore
OUTPUT "Enter the champion's score: "
INPUT ChampionScore
IF ChallengerScore > ChampionScore
THEN
   OUTPUT "The challenger wins"
ELSE
   OUTPUT "The champion wins"
ENDIF
```

Nested IF

```
IF ChallengerScore > ChampionScore
THEN
   IF ChallengerScore > HighestScore
   THEN
      OUTPUT "The challenger is champion and highest scorer"
   ELSE
      OUTPUT "The challenger is the new champion"
   ENDIF
ELSE
   OUTPUT "The champion is still the champion"
   IF ChampionScore > HighestScore
   THEN
      OUTPUT "The champion is also the highest scorer"
   ELSE
      OUTPUT "The champion has not beaten the highest score"
   ENDIF
ENDIF
```

CASE

```
CASE OF Score
   81 TO 100 : OUTPUT "Grade A*"
   63 TO 80  : OUTPUT "Grade A"
   47 TO 62  : OUTPUT "Grade B"
   31 TO 46  : OUTPUT "Grade C"
   25 TO 30  : OUTPUT "Grade D"
   20 TO 24  : OUTPUT "Grade E"
   15 TO 19  : OUTPUT "Grade F"
   9  TO 14  : OUTPUT "Grade G"
   OTHERWISE  : OUTPUT "Ungraded"
ENDCASE
```

PYTHON

Simple IF

```
challenger_score = int(input("Enter the challenger's score:
  "))  # DECLARE ChallengerScore : INTEGER
champion_score = int(input("Enter the champion's score: "))
  # DECLARE ChampionScore : INTEGER
if challenger_score > champion_score:
    print("The challenger wins")
else:
    print("The champion wins")
```

Nested IF

```
if challenger_score > champion_score:
    if challenger_score > highest_score:
        print("The challenger is champion and highest
          scorer")
    else:
        print("The challenger is the new champion")
else:
    print("The champion is still the champion")
    if champion_score > highest_score:
        print("The champion is also the highest scorer")
    else:
        print("The champion has not beaten the highest
          score")
```

CASE

```
if 81 <= score <= 100:
    print("Grade A*")
elif 63 <= score <= 80:
    print("Grade A")
elif 47 <= score <= 62:
    print("Grade B")
elif 31 <= score <= 46:
    print("Grade C")
elif 25 <= score <= 30:
    print("Grade D")
elif 20 <= score <= 24:
    print("Grade E")
elif 15 <= score <= 19:
    print("Grade F")
elif 9 <= score <= 14:
    print("Grade G")
else:
    print("Ungraded")
```

FOR Loops

A fixed number of iterations, counting up or down — Cambridge 0478/9618

A FOR loop repeats a fixed, known number of times — the loop variable counts through a range automatically, rather than a condition being checked each time round as WHILE and REPEAT do. Cambridge pseudocode's FOR...TO...NEXT always counts up by default.

STEP reverses the direction: TO with STEP -1 counts down instead. Python's range() takes the same idea — a start, a stop, and an optional step — in one function call.

CAMBRIDGE PSEUDOCODE

Count up

```
DECLARE N : INTEGER
DECLARE Counter : INTEGER
OUTPUT "How many times should this loop
  run? "
INPUT N
FOR Counter ← 1 TO N
   OUTPUT Counter
NEXT Counter
```

Count down

```
DECLARE N : INTEGER
DECLARE Counter : INTEGER
OUTPUT "Count down from how many? "
INPUT N
FOR Counter ← N TO 1 STEP -1
   OUTPUT Counter
NEXT Counter
```

PYTHON

Count up

```
n = int(input("How many times should this
  loop run? "))  # DECLARE N : INTEGER
for counter in range(1, n + 1):  # DECLARE
  Counter : INTEGER
    print(counter)
```

Count down

```
n = int(input("Count down from how many?
  "))  # DECLARE N : INTEGER
for counter in range(n, 0, -1):  # DECLARE
  Counter : INTEGER
    print(counter)
```

WHILE & REPEAT

Condition-controlled loops: check-first vs check-last — Cambridge 0478/9618

WHILE and REPEAT both loop until a condition changes, rather than a fixed number of times — the right choice when you don't know in advance how many repeats are needed, like validating input until it's acceptable.

The difference is when the condition is checked. WHILE checks first, so the loop body can run zero times if the condition already fails. REPEAT checks last, so the body always runs at least once — useful whenever the user has to be asked at least once before their answer can be judged.

CAMBRIDGE PSEUDOCODE

WHILE (check first)

```
DECLARE Password : STRING
OUTPUT "Please enter the password: "
INPUT Password
WHILE Password <> "Secret" DO
   OUTPUT "Incorrect password, try again"
   OUTPUT "Please enter the password: "
   INPUT Password
ENDWHILE
OUTPUT "Access granted"
```

REPEAT (check last)

```
DECLARE Password : STRING
REPEAT
   OUTPUT "Please enter the password: "
   INPUT Password
   IF Password <> "Secret" THEN
      OUTPUT "Incorrect password, try
        again"
   ENDIF
UNTIL Password = "Secret"
OUTPUT "Access granted"
```

PYTHON

WHILE (check first)

```
password = input("Please enter the
  password: ")  # DECLARE Password : STRING
while password != "Secret":
    print("Incorrect password, try again")
    password = input("Please enter the
      password: ")
print("Access granted")
```

REPEAT (check last)

```
while True:
    password = input("Please enter the
      password: ")  # DECLARE Password :
      STRING
    if password == "Secret":
        break
    print("Incorrect password, try again")
print("Access granted")
```

Arrays

Declare, fill, and read a fixed-size array — Cambridge 0478/9618

An array is a fixed number of same-type slots, each reached directly by its position (its index) rather than by searching. Cambridge pseudocode indexes arrays from 1; Python indexes from 0 — the two panels below show the same logic in both conventions.

DECLARE fixes the array's size up front. A FOR loop then visits every index in turn: Populate asks for a value at each position, Traverse reads them back in order. Because every slot has a known position, reaching any one of them directly — as Update does — never needs a search or a loop, unlike a linked list.

CAMBRIDGE PSEUDOCODE

Declare

```
DECLARE Items : ARRAY[1:5] OF STRING
```

Populate

```
FOR Index ← 1 TO 5
   OUTPUT "Enter an item: "
   INPUT Items[Index]
NEXT Index
```

Traverse

```
FOR Index ← 1 TO 5
   OUTPUT Items[Index]
NEXT Index
```

Update

```
DECLARE Index : INTEGER
DECLARE NewValue : STRING
OUTPUT "Enter the index to update: "
INPUT Index
OUTPUT "Enter the new value: "
INPUT NewValue
Items[Index] ← NewValue
```

PYTHON

Declare

```
items = [None] * 5  # DECLARE Items :
  ARRAY[1:5] OF STRING
```

Populate

```
for index in range(5):
    print("Enter an item: ")
    items[index] = input()
```

Traverse

```
for index in range(5):
    print(items[index])
```

Update

```
# DECLARE Index : INTEGER
# DECLARE NewValue : STRING
index = int(input("Enter the index to
  update: "))
new_value = input("Enter the new value: ")
items[index - 1] = new_value  # pseudocode
  is 1-based, Python is 0-based
```

Linear Search

Check every item in turn until a match is found or the list runs out — Cambridge 0478/9618

Linear search is the simplest way to find something in an array: start at the first position and check every item in turn until the target is found or there are no positions left to check. It works on any array, sorted or not.

The trade-off is speed — in the worst case it checks every single item, so it's O(n). Binary search is faster, but only works on data that's already sorted.

CAMBRIDGE PSEUDOCODE

Search

```
DECLARE Target : STRING
DECLARE Index : INTEGER
DECLARE Found : BOOLEAN
OUTPUT "Enter the item to find: "
INPUT Target
Index ← 1
Found ← FALSE
WHILE Found = FALSE AND Index <= 5 DO
   IF Items[Index] = Target
   THEN
      Found ← TRUE
   ELSE
      Index ← Index + 1
   ENDIF
ENDWHILE
IF Found = TRUE
THEN
   OUTPUT "Item found at position ",
     Index, ": ", Items[Index]
ELSE
   OUTPUT "Item not found"
ENDIF
```

PYTHON

Search

```
target = input("Enter the item to find: ")
  # DECLARE Target : STRING
index = 0  # DECLARE Index : INTEGER
found = False  # DECLARE Found : BOOLEAN
while found == False and index < 5:
    if items[index] == target:
        found = True
    else:
        index = index + 1
if found == True:
    print("Item found at position", index,
      ":", items[index])
else:
    print("Item not found")
```

Binary Search

Halve the search space each time — needs a sorted array — Cambridge 0478/9618

Binary search only works on a sorted array, but in exchange it's dramatically faster than checking every item: compare the target to the midpoint, then discard whichever half it can't be in, and repeat on the remaining half.

Each comparison halves the remaining search space, so it's O(log n) — a 1,000-item sorted array takes at most about 10 comparisons to search, not 1,000.

CAMBRIDGE PSEUDOCODE

Search

```
DECLARE Target : STRING
DECLARE First : INTEGER
DECLARE Last : INTEGER
DECLARE Midpoint : INTEGER
DECLARE Found : BOOLEAN
OUTPUT "Enter the item to find: "
INPUT Target
Found ← FALSE
First ← 1
Last ← 5
WHILE First <= Last AND Found = FALSE
   Midpoint ← (First + Last) DIV 2
   IF Items[Midpoint] = Target THEN
      Found ← TRUE
   ELSE
      IF Items[Midpoint] < Target THEN
         First ← Midpoint + 1
      ELSE
         Last ← Midpoint - 1
      ENDIF
   ENDIF
ENDWHILE
IF Found = TRUE THEN
   OUTPUT "Item found at position ",
     Midpoint, ": ", Items[Midpoint]
ELSE
   OUTPUT "Item not found"
ENDIF
```

PYTHON

Search

```
target = input("Enter the item to find: ")
  # DECLARE Target : STRING
found = False  # DECLARE Found : BOOLEAN
first = 0  # DECLARE First : INTEGER
last = 4  # DECLARE Last : INTEGER
while first <= last and found == False:
    midpoint = (first + last) // 2  #
      DECLARE Midpoint : INTEGER
    if items[midpoint] == target:
        found = True
    else:
        if items[midpoint] < target:
            first = midpoint + 1
        else:
            last = midpoint - 1
if found == True:
    print("Item found at position",
      midpoint, ":", items[midpoint])
else:
    print("Item not found")
```

Bubble Sort

Repeatedly swap adjacent out-of-order pairs until nothing moves — Cambridge 0478/9618

Bubble sort makes repeated passes through the array, comparing each pair of neighbours and swapping them if they're in the wrong order. Larger values 'bubble' toward the end with each pass.

A Swapped flag tracks whether any swap happened on the last pass — if nothing moved, the array is already sorted and the loop can stop early. Simple to trace by hand, but O(n²) in the worst case, so not the fastest choice for large arrays.

CAMBRIDGE PSEUDOCODE

Sort

```
DECLARE N : INTEGER
DECLARE Swapped : BOOLEAN
DECLARE Temp : STRING
N ← 5
Swapped ← TRUE
WHILE N > 0 AND Swapped DO
   Swapped ← FALSE
   N ← N - 1
   FOR Index ← 1 TO N
      IF Items[Index] > Items[Index + 1]
      THEN
         Temp ← Items[Index]
         Items[Index] ← Items[Index + 1]
         Items[Index + 1] ← Temp
         Swapped ← TRUE
      ENDIF
   NEXT Index
ENDWHILE
OUTPUT Items
```

PYTHON

Sort

```
n = 5
swapped = True
while n > 0 and swapped:
    swapped = False
    n = n - 1
    for index in range(n):
        if items[index] > items[index +
          1]:
            temp = items[index]
            items[index] = items[index +
              1]
            items[index + 1] = temp
            swapped = True
print(items)
```

Insertion Sort

Build a sorted section of the array one item at a time — Cambridge 0478/9618

Insertion sort grows a sorted section at the start of the array. Each pass takes the next unsorted item and shifts it backward through the sorted section until it reaches its correct place — the same way a card player sorts a hand of cards.

Like Bubble Sort it's O(n²) in the worst case, but it does less work when the array is already mostly sorted, since fewer shifts are needed.

CAMBRIDGE PSEUDOCODE

Sort

```
DECLARE N : INTEGER
DECLARE Current : STRING
DECLARE Index2 : INTEGER
N ← 5
FOR Index ← 2 TO N
   Current ← Items[Index]
   Index2 ← Index
   WHILE Index2 > 1 AND Items[Index2 - 1]
     > Current
      Items[Index2] ← Items[Index2 - 1]
      Index2 ← Index2 - 1
   ENDWHILE
   Items[Index2] ← Current
NEXT Index
OUTPUT Items
```

PYTHON

Sort

```
n = 5  # DECLARE N : INTEGER
for index in range(1, n):
    current = items[index]  # DECLARE
      Current : STRING
    index2 = index  # DECLARE Index2 :
      INTEGER
    while index2 > 0 and items[index2 - 1]
      > current:
        items[index2] = items[index2 - 1]
        index2 = index2 - 1
    items[index2] = current
print(items)
```

Functions and Procedures

Parameters, RETURN, and local vs global scope — IGCSE syllabus, 9618 pseudocode

On the IGCSE syllabus (0478), but the IGCSE pseudocode guide defines no formal PROCEDURE/FUNCTION syntax of its own — that's spelled out only in the 9618 (A Level) guide, so this page borrows it to teach genuinely IGCSE content.

A procedure groups statements under a name and runs them when called; a function does the same but also RETURNs a single value, so it can be used inside an expression. Parameters pass values in. A variable declared inside a procedure or function is LOCAL to that one call and is discarded the moment it ends — a variable declared outside, at program level, is GLOBAL and survives every call.

CAMBRIDGE PSEUDOCODE

Procedure, with a parameter

```
GLOBAL Salary : INTEGER

// CalculateBonus takes one parameter -- BYVAL by default, a copy not the
  original
PROCEDURE CalculateBonus(Amount : INTEGER)
   // CalculatedBonus exists only inside this call
   LOCAL CalculatedBonus : INTEGER
   CalculatedBonus ← Amount * 2
   // this line reaches outside the procedure -- it changes the GLOBAL
   Salary ← Salary + CalculatedBonus
ENDPROCEDURE
```

Calling the procedure

```
// Main program
Salary ← 10
OUTPUT "The Salary is: ", Salary
DECLARE Bonus : INTEGER
OUTPUT "Enter a bonus amount: "
INPUT Bonus
CALL CalculateBonus(Bonus)
OUTPUT "New Salary: ", Salary
```

Function, returning a value

```
// CalculateAverage takes two parameters -- A and B are local copies of
  whatever was passed in
FUNCTION CalculateAverage(A : INTEGER, B : INTEGER) RETURNS INTEGER
   // Sum and Average both exist only inside this call
   LOCAL Sum : INTEGER
   LOCAL Average : INTEGER
   Sum ← A + B
   Average ← Sum DIV 2
   // RETURN ends the function immediately, right here
   RETURN Average
ENDFUNCTION
```

Calling the function

```
// Main program
DECLARE A : INTEGER
OUTPUT "Enter the first number: "
INPUT A
DECLARE B : INTEGER
OUTPUT "Enter the second number: "
INPUT B
DECLARE Result : INTEGER
Result ← CalculateAverage(A, B)
OUTPUT "The Average is: ", Result
```

PYTHON

Procedure, with a parameter

```
salary = 0  # global

def calculate_bonus(amount):
    # needed because this function REASSIGNS salary below
    global salary
    calculated_bonus = amount * 2
    salary = salary + calculated_bonus
```

Calling the procedure

```
# Main program
salary = 10
print("The Salary is:", salary)
bonus = int(input("Enter a bonus amount: "))
calculate_bonus(bonus)
print("New Salary:", salary)
```

Function, returning a value

```
def calculate_average(a, b):
    # Sum and average both exist only inside this call
    sum = a + b
    average = sum // 2
    return average
```

Calling the function

```
# Main program
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
result = calculate_average(a, b)
print("The Average is:", result)
```

Stacks — OOP (pointers)

Last in, first out, pointer-based — Cambridge 9618 · reverse: Array (exam-style)

A stack only allows access at one end, the top. Push adds a new item on top; Pop removes and returns whatever's currently on top — always the most recently added item, last in first out.

This side uses 9618's pointer-based node structure (^Node), not a fixed array — each node points to the one beneath it, so the stack can grow to any size without a declared maximum. The reverse side shows the same two operations the array-based, exam-style way.

CAMBRIDGE PSEUDOCODE

Push

```
PROCEDURE Push(NewData : STRING)
   DECLARE NewNode : ^Node
   NewNode ← NEW Node
   NewNode^.Data ← NewData
   NewNode^.NextNode ← StackPointer
   StackPointer ← NewNode
ENDPROCEDURE
```

Pop

```
PROCEDURE Pop()
   DECLARE Popped : STRING
   IF StackPointer <> NULL THEN
      Popped ← StackPointer^.Data
      StackPointer ←
        StackPointer^.NextNode
      OUTPUT "Item popped: ", Popped
   ELSE
      OUTPUT "Stack underflow"
   ENDIF
ENDPROCEDURE
```

PYTHON

Push

```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.stack_pointer
    self.stack_pointer = new_node
```

Pop

```
def pop(self):
    if self.stack_pointer is not None:
        popped = self.stack_pointer.data
        self.stack_pointer =
          self.stack_pointer.next
        print('Item popped:', popped)
    else:
        print('Stack underflow')
```

Stacks — Array (exam-style)

The same Push and Pop, built on a fixed array — Cambridge 9618

The exam-style model: a fixed-size array plus a single StackPointer index tracking the top. Push and Pop move the pointer and read/write the array directly — no nodes, no NEW, just index arithmetic.

Overflow and underflow are explicit checks now, against the array's bound, rather than a NULL pointer — the reverse side's pointer-based version checks for NULL instead.

CAMBRIDGE PSEUDOCODE

Push

```
PROCEDURE Push(NewData : STRING)
   IF StackPointer = MaxSize - 1 THEN
      OUTPUT "Stack overflow"
   ELSE
      StackPointer ← StackPointer + 1
      Stack[StackPointer] ← NewData
   ENDIF
ENDPROCEDURE
```

Pop

```
PROCEDURE Pop()
   DECLARE Popped : STRING
   IF StackPointer = -1 THEN
      OUTPUT "Stack underflow"
   ELSE
      Popped ← Stack[StackPointer]
      StackPointer ← StackPointer - 1
      OUTPUT "Item popped: ", Popped
   ENDIF
ENDPROCEDURE
```

PYTHON

Push

```
def push(data):
    global stack_pointer
    if stack_pointer == max_size - 1:
        print('Stack overflow')
    else:
        stack_pointer += 1
        stack[stack_pointer] = data
```

Pop

```
def pop():
    global stack_pointer
    if stack_pointer == -1:
        print('Stack underflow')
    else:
        popped = stack[stack_pointer]
        stack_pointer -= 1
        print('Item popped:', popped)
```

Queues — OOP (pointers)

First in, first out, pointer-based — Cambridge 9618 · reverse: Array (exam-style)

A queue allows access at both ends but in fixed roles: Enqueue always adds at the back, Dequeue always removes from the front — first in, first out, the opposite discipline to a stack.

Two pointers track the ends: FrontPointer for the next item to come out, BackPointer for where the next new item attaches. Like Stacks, this side uses 9618's pointer-based nodes — the reverse side shows the same two operations as a circular array instead.

CAMBRIDGE PSEUDOCODE

Enqueue

```
PROCEDURE Enqueue(NewData : STRING)
   DECLARE NewNode : ^Node
   NewNode ← NEW Node
   NewNode^.Data ← NewData
   NewNode^.NextNode ← NULL
   IF BackPointer = NULL THEN
      FrontPointer ← NewNode
   ELSE
      BackPointer^.NextNode ← NewNode
   ENDIF
   BackPointer ← NewNode
ENDPROCEDURE
```

Dequeue

```
PROCEDURE Dequeue()
   DECLARE Dequeued : STRING
   IF FrontPointer <> NULL THEN
      Dequeued ← FrontPointer^.Data
      FrontPointer ←
        FrontPointer^.NextNode
      IF FrontPointer = NULL THEN
         BackPointer ← NULL
      ENDIF
      OUTPUT "Item dequeued: ", Dequeued
   ELSE
      OUTPUT "Queue underflow"
   ENDIF
ENDPROCEDURE
```

PYTHON

Enqueue

```
def enqueue(self, data):
    new_node = Node(data)
    if self.back_pointer is None:
        self.front_pointer = new_node
    else:
        self.back_pointer.next = new_node
    self.back_pointer = new_node
```

Dequeue

```
def dequeue(self):
    if self.front_pointer is not None:
        dequeued = self.front_pointer.data
        self.front_pointer =
          self.front_pointer.next
        if self.front_pointer is None:
            self.back_pointer = None
        print('Item dequeued:', dequeued)
    else:
        print('Queue underflow')
```

Queues — Array (exam-style)

The same Enqueue and Dequeue, as a circular array — Cambridge 9618

The exam-style model: a fixed-size array used as a circular buffer. FrontPointer and BackPointer wrap around with MOD instead of walking off the end, and a separate ItemCount tracks how full the queue is.

The reverse side's pointer-based version needs no wraparound at all — a node is simply allocated or freed — which is exactly the trade-off worth noticing between the two.

CAMBRIDGE PSEUDOCODE

Enqueue

```
PROCEDURE Enqueue(NewData : STRING)
   IF ItemCount = MaxSize THEN
      OUTPUT "Queue overflow"
   ELSE
      BackPointer ← (BackPointer + 1) MOD
        MaxSize
      Queue[BackPointer] ← NewData
      ItemCount ← ItemCount + 1
   ENDIF
ENDPROCEDURE
```

Dequeue

```
PROCEDURE Dequeue()
   DECLARE Dequeued : STRING
   IF ItemCount = 0 THEN
      OUTPUT "Queue underflow"
   ELSE
      Dequeued ← Queue[FrontPointer]
      FrontPointer ← (FrontPointer + 1)
        MOD MaxSize
      ItemCount ← ItemCount - 1
      OUTPUT "Item dequeued: ", Dequeued
   ENDIF
ENDPROCEDURE
```

PYTHON

Enqueue

```
def enqueue(data):
    global back_pointer, item_count
    if item_count == max_size:
        print('Queue overflow')
    else:
        back_pointer = (back_pointer + 1)
          % max_size
        queue[back_pointer] = data
        item_count += 1
```

Dequeue

```
def dequeue():
    global front_pointer, item_count
    if item_count == 0:
        print('Queue underflow')
    else:
        dequeued = queue[front_pointer]
        front_pointer = (front_pointer +
          1) % max_size
        item_count -= 1
        print('Item dequeued:', dequeued)
```

Linked Lists — OOP (pointers)

Insert and delete without shifting anything, pointer-based — Cambridge 9618 · reverse: Array (exam-style)

A linked list is a chain of nodes, each holding data and a pointer to the next node. Unlike an array, there's no fixed size and no shifting — inserting or deleting a node just means re-pointing its neighbours.

The trade-off against an array is access: reaching a specific node means walking the chain from Head one pointer at a time, O(n), not a direct index lookup. The reverse side shows the same two operations built on a fixed array of nodes instead of real pointers — the exam-style model real mark schemes often use.

CAMBRIDGE PSEUDOCODE

Add (in order)

```
PROCEDURE AddOrdered(NewData : STRING)
   DECLARE NewNode, Current, Previous : ^Node
   NewNode ← NEW Node
   NewNode^.Data ← NewData
   IF Head = NULL OR Head^.Data > NewData THEN
      NewNode^.NextNode ← Head
      Head ← NewNode
   ELSE
      Current ← Head
      Previous ← NULL
      WHILE Current <> NULL AND Current^.Data <
        NewData
         Previous ← Current
         Current ← Current^.NextNode
      ENDWHILE
      NewNode^.NextNode ← Current
      Previous^.NextNode ← NewNode
   ENDIF
ENDPROCEDURE
```

Delete

```
PROCEDURE DeleteNode(Target : STRING)
   DECLARE Current, Previous : ^Node
   IF Head = NULL THEN
      OUTPUT "List is empty"
   ELSE
      IF Head^.Data = Target THEN
         Head ← Head^.NextNode
      ELSE
         Current ← Head
         Previous ← NULL
         WHILE Current <> NULL AND Current^.Data <>
           Target
            Previous ← Current
            Current ← Current^.NextNode
         ENDWHILE
         IF Current <> NULL THEN
            Previous^.NextNode ← Current^.NextNode
         ELSE
            OUTPUT "Not found"
         ENDIF
      ENDIF
   ENDIF
ENDPROCEDURE
```

PYTHON

Add (in order)

```
def add_ordered(self, data):
    new_node = Node(data)
    if self.head is None or self.head.data > data:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        previous = None
        while current is not None and current.data
          < data:
            previous = current
            current = current.next
        new_node.next = current
        previous.next = new_node
```

Delete

```
def delete_node(self, target):
    if self.head is None:
        print('List is empty')
        return
    if self.head.data == target:
        self.head = self.head.next
        return
    current = self.head
    previous = None
    while current is not None and current.data !=
      target:
        previous = current
        current = current.next
    if current is not None:
        previous.next = current.next
    else:
        print('Not found')
```

Linked Lists — Array (exam-style)

The same Add and Delete, nodes stored in a fixed array — Cambridge 9618

The exam-style model real mark schemes often use: nodes live in a fixed array of records, and NextPointer is a plain INTEGER index into that array rather than a real memory reference. A FreePointer chains together the unused slots.

Adding re-uses the next free slot instead of calling NEW; deleting returns a slot to the free chain instead of leaving it to be garbage-collected.

CAMBRIDGE PSEUDOCODE

Add (in order)

```
PROCEDURE AddOrdered(NewData : STRING)
   DECLARE NewNodePointer, Current, Previous : INTEGER
   IF FreePointer = -1 THEN
      OUTPUT "No free space left"
   ELSE
      NewNodePointer ← FreePointer
      FreePointer ← List[FreePointer].NextPointer
      List[NewNodePointer].Data ← NewData
      IF HeadPointer = -1 OR List[HeadPointer].Data > NewData
        THEN
         List[NewNodePointer].NextPointer ← HeadPointer
         HeadPointer ← NewNodePointer
      ELSE
         Current ← HeadPointer
         Previous ← -1
         WHILE Current <> -1 AND List[Current].Data < NewData
            Previous ← Current
            Current ← List[Current].NextPointer
         ENDWHILE
         List[NewNodePointer].NextPointer ← Current
         List[Previous].NextPointer ← NewNodePointer
      ENDIF
   ENDIF
ENDPROCEDURE
```

Delete

```
PROCEDURE DeleteNode(Target : STRING)
   DECLARE Current, Previous, FoundPointer : INTEGER
   IF HeadPointer = -1 THEN
      OUTPUT "List is empty"
   ELSE
      IF List[HeadPointer].Data = Target THEN
         FoundPointer ← HeadPointer
         HeadPointer ← List[HeadPointer].NextPointer
         List[FoundPointer].NextPointer ← FreePointer
         FreePointer ← FoundPointer
      ELSE
         Current ← HeadPointer
         Previous ← -1
         WHILE Current <> -1 AND List[Current].Data <> Target
            Previous ← Current
            Current ← List[Current].NextPointer
         ENDWHILE
         IF Current = -1 THEN
            OUTPUT "Not found"
         ELSE
            List[Previous].NextPointer ←
              List[Current].NextPointer
            List[Current].NextPointer ← FreePointer
            FreePointer ← Current
         ENDIF
      ENDIF
   ENDIF
ENDPROCEDURE
```

PYTHON

Add (in order)

```
def add_ordered(data):
    global head_pointer, free_pointer
    if free_pointer == -1:
        print('No free space left')
    else:
        new_node_pointer = free_pointer
        free_pointer = pool[free_pointer].next_pointer
        pool[new_node_pointer].data = data
        if head_pointer == -1 or pool[head_pointer].data >
          data:
            pool[new_node_pointer].next_pointer =
              head_pointer
            head_pointer = new_node_pointer
        else:
            current = head_pointer
            previous = -1
            while current != -1 and pool[current].data <
              data:
                previous = current
                current = pool[current].next_pointer
            pool[new_node_pointer].next_pointer = current
            pool[previous].next_pointer = new_node_pointer
```

Delete

```
def delete_node(target):
    global head_pointer, free_pointer
    if head_pointer == -1:
        print('List is empty')
        return
    if pool[head_pointer].data == target:
        found_pointer = head_pointer
        head_pointer = pool[head_pointer].next_pointer
        pool[found_pointer].next_pointer = free_pointer
        free_pointer = found_pointer
        return
    current = head_pointer
    previous = -1
    while current != -1 and pool[current].data != target:
        previous = current
        current = pool[current].next_pointer
    if current == -1:
        print('Not found')
    else:
        pool[previous].next_pointer =
          pool[current].next_pointer
        pool[current].next_pointer = free_pointer
        free_pointer = current
```

Binary Trees — OOP (pointers)

Smaller left, larger-or-equal right, pointer-based — Cambridge 9618 · reverse: Array (exam-style)

A binary search tree keeps every value less than a node in its left subtree, and every value greater-or-equal in its right — so Add walks down from Root comparing at each node until it finds an empty spot.

InOrder traversal (left, visit, right) is the payoff: visiting this way always outputs every value in sorted order, using recursion rather than a loop.

CAMBRIDGE PSEUDOCODE

Add

```
PROCEDURE AddNode(NewData : STRING)
   DECLARE NewNode, Current : ^Node
   NewNode ← NEW Node
   NewNode^.Data ← NewData
   IF Root = NULL THEN
      Root ← NewNode
   ELSE
      Current ← Root
      WHILE Current <> NULL
         IF NewData < Current^.Data THEN
            IF Current^.LeftPointer = NULL THEN
               Current^.LeftPointer ← NewNode
               Current ← NULL
            ELSE
               Current ← Current^.LeftPointer
            ENDIF
         ELSE
            IF Current^.RightPointer = NULL THEN
               Current^.RightPointer ← NewNode
               Current ← NULL
            ELSE
               Current ← Current^.RightPointer
            ENDIF
         ENDIF
      ENDWHILE
   ENDIF
ENDPROCEDURE
```

InOrder traversal

```
PROCEDURE InOrder(Node : ^Node)
   IF Node <> NULL THEN
      CALL InOrder(Node^.LeftPointer)
      OUTPUT Node^.Data
      CALL InOrder(Node^.RightPointer)
   ENDIF
ENDPROCEDURE
CALL InOrder(Root)
```

PYTHON

Add

```
def add_node(data):
    global root
    new_node = Node(data)
    if root is None:
        root = new_node
    else:
        current = root
        while current is not None:
            if data < current.data:
                if current.left is None:
                    current.left = new_node
                    current = None
                else:
                    current = current.left
            else:
                if current.right is None:
                    current.right = new_node
                    current = None
                else:
                    current = current.right
```

InOrder traversal

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

Binary Trees — Array (exam-style)

The same Add, nodes stored in a fixed array — Cambridge 9618

The exam-style model: tree nodes live in a fixed array of records, with LeftPointer and RightPointer as plain INTEGER indices rather than real references, and a FreePointer chain tracking unused slots — the same scheme Linked Lists uses on its array side.

InOrder isn't repeated here — it works identically either way, only ever following whichever pointer type the tree happens to use, so the reverse side's version already covers it. Add is where the two representations actually differ.

CAMBRIDGE PSEUDOCODE

Add

```
PROCEDURE AddNode(NewData : STRING)
   DECLARE NewNodePointer, Current : INTEGER
   IF FreePointer = -1 THEN
      OUTPUT "No free space left"
   ELSE
      NewNodePointer ← FreePointer
      FreePointer ←
        Tree[FreePointer].LeftPointer
      Tree[NewNodePointer].Data ← NewData
      Tree[NewNodePointer].LeftPointer ← -1
      Tree[NewNodePointer].RightPointer ← -1
      IF Root = -1 THEN
         Root ← NewNodePointer
      ELSE
         Current ← Root
         WHILE Current <> -1
            IF NewData < Tree[Current].Data THEN
               IF Tree[Current].LeftPointer = -1
                 THEN
                  Tree[Current].LeftPointer ←
                    NewNodePointer
                  Current ← -1
               ELSE
                  Current ←
                    Tree[Current].LeftPointer
               ENDIF
            ELSE
               IF Tree[Current].RightPointer =
                 -1 THEN
                  Tree[Current].RightPointer ←
                    NewNodePointer
                  Current ← -1
               ELSE
                  Current ←
                    Tree[Current].RightPointer
               ENDIF
            ENDIF
         ENDWHILE
      ENDIF
   ENDIF
ENDPROCEDURE
```

PYTHON

Add

```
def add_node(data):
    global root, free_pointer
    if free_pointer == -1:
        print('No free space left')
    else:
        new_node_pointer = free_pointer
        free_pointer =
          pool[free_pointer].left_pointer
        pool[new_node_pointer].data = data
        pool[new_node_pointer].left_pointer = -1
        pool[new_node_pointer].right_pointer =
          -1
        if root == -1:
            root = new_node_pointer
        else:
            current = root
            while current != -1:
                if data < pool[current].data:
                    if
                      pool[current].left_pointer
                      == -1:
                        pool[current].left_pointer
                          = new_node_pointer
                        current = -1
                    else:
                        current =
                          pool[current].left_pointer
                else:
                    if
                      pool[current].right_pointer
                      == -1:
                        pool[current].right_pointer
                          = new_node_pointer
                        current = -1
                    else:
                        current =
                          pool[current].right_pointer
```

OOP — Classes, Attributes, Methods and Objects

Classes, objects, getters and setters — Cambridge 9618

A class is a blueprint for a kind of object — it defines what data each object holds (its attributes) and what it can do (its methods). PRIVATE attributes can only be changed from inside the class itself; getters and setters are the PUBLIC methods that let the rest of the program read and change them safely.

Creating an object from a class is called instantiation — NEW calls the class's constructor, always named NEW, which sets up that object's own attributes. A program can create several objects from the same class side by side, each with its own independent values, then amend and read any of them back through its getters and setters.

CAMBRIDGE PSEUDOCODE

Defining the class

```
CLASS Dog
   PRIVATE Name : STRING
   PRIVATE Breed : STRING
   PRIVATE Age : INTEGER

   PUBLIC PROCEDURE NEW(NewName : STRING, NewBreed :
     STRING, NewAge : INTEGER)
      Name ← NewName
      Breed ← NewBreed
      Age ← NewAge
   ENDPROCEDURE

   PUBLIC FUNCTION GetName() RETURNS STRING
      RETURN Name
   ENDFUNCTION

   PUBLIC FUNCTION GetBreed() RETURNS STRING
      RETURN Breed
   ENDFUNCTION

   PUBLIC FUNCTION GetAge() RETURNS INTEGER
      RETURN Age
   ENDFUNCTION

   PUBLIC PROCEDURE SetAge(NewAge : INTEGER)
      Age ← NewAge
   ENDPROCEDURE
ENDCLASS
```

Creating an object, amending, reading it back

```
MyDog ← NEW Dog("Rex", "Labrador", 3)
MyDog.SetAge(4)
OUTPUT MyDog.GetAge()
// one OUTPUT, every attribute -- the whole object
OUTPUT MyDog.GetName(), MyDog.GetBreed(),
  MyDog.GetAge()
```

PYTHON

Defining the class

```
class Dog:
    def __init__(self, new_name, new_breed,
      new_age):
        self.name = new_name
        self.breed = new_breed
        self.age = new_age

    def get_name(self):
        return self.name

    def get_breed(self):
        return self.breed

    def get_age(self):
        return self.age

    def set_age(self, new_age):
        self.age = new_age
```

Creating an object, amending, reading it back

```
my_dog = Dog("Rex", "Labrador", 3)
my_dog.set_age(4)
print(my_dog.get_age())
# one print, every attribute -- the whole object
print(my_dog.get_name(), my_dog.get_breed(),
  my_dog.get_age())
```

OOP with Inheritance — Building a Subclass

A subclass that INHERITS a superclass, with SUPER.NEW — Cambridge 9618

A subclass can INHERIT everything a superclass already has, then add its own attribute on top. Dog INHERITS Animal, so every Dog is also an Animal, with Name and Sound already built in — Dog only has to declare and manage the one attribute that's really its own, Breed.

SUPER.NEW() is how a subclass's own constructor reaches up to run its superclass's constructor first — the one call that actually sets Name and Sound — before the subclass sets up its own attribute. Reading an object back shows the same split: GetName() and GetSound() come from Animal, GetBreed() only exists because it's Dog's own.

CAMBRIDGE PSEUDOCODE

The superclass — already built

```
// Animal is already built -- the superclass for this demo
CLASS Animal
   PRIVATE Name : STRING
   PRIVATE Sound : STRING

   PUBLIC PROCEDURE NEW(NewName : STRING, NewSound :
     STRING)
      Name ← NewName
      Sound ← NewSound
   ENDPROCEDURE

   PUBLIC FUNCTION GetName() RETURNS STRING
      RETURN Name
   ENDFUNCTION

   PUBLIC FUNCTION GetSound() RETURNS STRING
      RETURN Sound
   ENDFUNCTION
ENDCLASS
```

Dog INHERITS Animal

```
CLASS Dog INHERITS Animal
   PRIVATE Breed : STRING

   PUBLIC PROCEDURE NEW(NewName : STRING, NewSound :
     STRING, NewBreed : STRING)
      SUPER.NEW(NewName, NewSound)
      Breed ← NewBreed
   ENDPROCEDURE

   PUBLIC FUNCTION GetBreed() RETURNS STRING
      RETURN Breed
   ENDFUNCTION
ENDCLASS
```

Creating an object — inherited vs own

```
MyDog ← NEW Dog("Rex", "Woof", "Labrador")
// one OUTPUT, every attribute -- the whole object
OUTPUT MyDog.GetName(), MyDog.GetSound(), MyDog.GetBreed()
// Name and Sound are inherited from Animal; Breed is
  Dog's own
```

PYTHON

The superclass — already built

```
# Animal is already built -- the superclass for this demo
class Animal:
    def __init__(self, new_name, new_sound):
        self.name = new_name
        self.sound = new_sound

    def get_name(self):
        return self.name

    def get_sound(self):
        return self.sound
```

Dog inherits Animal

```
class Dog(Animal):
    def __init__(self, new_name, new_sound, new_breed):
        super().__init__(new_name, new_sound)
        self.breed = new_breed

    def get_breed(self):
        return self.breed
```

Creating an object -- inherited vs own

```
my_dog = Dog("Rex", "Woof", "Labrador")
# one print, every attribute -- the whole object
print(my_dog.get_name(), my_dog.get_sound(),
  my_dog.get_breed())
# name and sound are inherited from Animal; breed is Dog's
  own
```

Recursion

A function that calls itself — Cambridge 9618

A recursive function solves a problem by calling itself on a smaller version of the same problem. Every recursive function needs three things: a base case that stops the recursion, a way of working towards that base case, and the self-call itself.

Each call waits on a stack frame for the one below it to return — the same call stack used elsewhere in this series (Stacks) — so a base case that's never reached eventually exhausts memory, a real RecursionError in Python.

CAMBRIDGE PSEUDOCODE

Factorial (base case N <= 1)

```
FUNCTION Factorial(N : INTEGER) RETURNS
  INTEGER
   IF N <= 1 THEN
      RETURN 1
   ELSE
      RETURN N * Factorial(N - 1)
   ENDIF
ENDFUNCTION

DECLARE N, Result : INTEGER
OUTPUT "Enter a number: "
INPUT N
Result ← Factorial(N)
OUTPUT "The Factorial of ", N, " is ",
  Result
```

Fibonacci (base case N <= 1)

```
FUNCTION Fibonacci(N : INTEGER) RETURNS
  INTEGER
   IF N <= 1 THEN
      RETURN N
   ELSE
      RETURN Fibonacci(N - 1) +
        Fibonacci(N - 2)
   ENDIF
ENDFUNCTION

DECLARE N, Result : INTEGER
OUTPUT "Enter a number: "
INPUT N
Result ← Fibonacci(N)
OUTPUT "The Fibonacci of ", N, " is ",
  Result
```

PYTHON

Factorial (base case n <= 1)

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

n = int(input("Enter a number: "))
result = factorial(n)
print("The Factorial of", n, "is", result)
```

Fibonacci (base case n <= 1)

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) +
          fibonacci(n - 2)

n = int(input("Enter a number: "))
result = fibonacci(n)
print("The Fibonacci of", n, "is", result)
```

File Processing

Reading from and writing to a text file — Cambridge 0478/9618

A program's variables disappear when it ends — a file is how data survives between runs. OPENFILE names a file and a mode (READ, WRITE, or APPEND) before anything else can happen to it; CLOSEFILE releases it once you're done.

WRITEFILE adds data to an open file; READFILE reads the next line back as a STRING. EOF (end of file) tells you when there's nothing left to read, so a WHILE loop can process every line without knowing in advance how many there are.

CAMBRIDGE PSEUDOCODE

Write to a file

```
OPENFILE "Scores.txt" FOR WRITE
FOR Index ← 1 TO 5
   OUTPUT "Enter a score: "
   INPUT Score
   WRITEFILE "Scores.txt", Score
NEXT Index
CLOSEFILE "Scores.txt"
```

Read a file back

```
DECLARE Line : STRING
OPENFILE "Scores.txt" FOR READ
WHILE NOT EOF("Scores.txt")
   READFILE "Scores.txt", Line
   OUTPUT Line
ENDWHILE
CLOSEFILE "Scores.txt"
```

PYTHON

Write to a file

```
with open("Scores.txt", "w") as f:
    for index in range(5):
        score = input("Enter a score: ")
        f.write(score + "\n")
```

Read a file back

```
with open("Scores.txt", "r") as f:
    while (line := f.readline()) != "":
        print(line.strip())
```