ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3435

Program: Post Graduate

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Visual Basic & Database Interface (3435) • Level: Post Graduate

Semester: Autumn 2026

Assignment No. 2

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant units of the course book, understand the concepts and the logic of every program, build and run each project on a computer in Visual Basic 6.0, and then write the assignment in their own words. Copying this guide word for word is not allowed; the printed instructions on the assignment booklet also require answers in the student's own words and warn against plagiarism. All Visual Basic programs in this guide are complete and self-contained: create a Standard EXE project, place the controls named in each answer, set the properties listed in the tables, type the code exactly, and run the project to confirm the sample run before writing the answer in the assignment.

Question 1: Modular Application with Procedures and Functions — Maintainability and Reusability (20 Marks)

Question as printed: a) Create a modular application that performs multiple mathematical operations using procedures and functions. b) Analyze how modular programming improves code maintainability and reusability. (20)

Introduction

As programs grow, the style in which code is organised matters as much as what the code does. A beginner writes one long Click procedure that reads the inputs, performs every calculation, formats every answer and clears every box; the program works but any change — a new operation, a corrected formula — means surgery inside a long, tangled block, with a real risk of breaking something unrelated. Modular programming is the disciplined alternative: the program is divided into small, named, single-purpose units — Sub procedures, which perform an action, and Functions, which compute and return a value — and the event procedures become short coordinators that call the modules in the right order. Part (a) of this question builds a modular mathematical application in exactly this style: a standard module holds pure calculation functions (addition, factorial, power, prime testing), the form holds input/output procedures, and nothing is calculated twice anywhere in the project. Part (b) then analyses why this structure improves maintainability and reusability, because the analysis and not merely the code is what earns the higher marks at postgraduate level.

Part (a): The Modular Mathematical Application

Create a project MathModular with a form frmMath and a standard module modMath (Project → Add Module). The division of labour is deliberate and should be stated in the answer: modMath contains only calculations — it never touches a control, never shows a MsgBox — so its functions can be tested and reused in any other project unchanged; frmMath contains only the interface — reading text boxes, calling the functions, and displaying results. The functions illustrate the two parameter mechanisms of Visual Basic. Arguments passed ByVal give the

procedure a copy, so the original variable cannot be altered; arguments passed ByRef give it the original, so changes are visible to the caller — used here in a Swap procedure. Factorial is written recursively, a function that calls itself with a smaller problem (n × (n−1)!), with the base case 0! = 1 stopping the recursion; IsPrime returns a Boolean and shows a function used inside a condition.

Code 1: modMath — the calculation module (no interface code)

```
'===== modMath.bas =====
Public Function AddNumbers(ByVal a As Double, ByVal b As Double) As Double
   AddNumbers = a + b
End Function

Public Function Power(ByVal x As Double, ByVal n As Integer) As Double
   Dim i As Integer, result As Double
   result = 1
   For i = 1 To n
      result = result * x
   Next i
   Power = result
End Function

Public Function Factorial(ByVal n As Integer) As Long
   If n <= 1 Then
      Factorial = 1
   Else
```

```
      Factorial = n * Factorial(n - 1)
   End If
End Function


Public Function IsPrime(ByVal n As Integer) As Boolean
   Dim i As Integer
   If n < 2 Then IsPrime = False: Exit Function
   For i = 2 To n \ 2
      If n Mod i = 0 Then IsPrime = False: Exit Function
   Next i
   IsPrime = True
End Function


Public Sub SwapValues(ByRef a As Double, ByRef b As Double)
   Dim temp As Double
   temp = a: a = b: b = temp
End Sub
```

Code 2: frmMath — short event procedures that call the module

```
Private Sub cmdCompute_Click()
   Dim x As Double, y As Double
   x = Val(txtX.Text): y = Val(txtY.Text)
   lstOut.Clear
   lstOut.AddItem "Sum = " & AddNumbers(x, y)
   lstOut.AddItem "Power (x^y) = " & Power(x, CInt(y))
```

```
    lstOut.AddItem "Factorial of x = " & Factorial(CInt(x))
    lstOut.AddItem "x is prime? " & IsPrime(CInt(x))
    SwapValues x, y
    lstOut.AddItem "After swap: x = " & x & ", y = " & y
End Sub
```

With inputs x = 5 and y = 3 the sample run below is produced. Observe how short cmdCompute_Click is: it reads, calls, and displays, and every line of real mathematics lives in modMath where it can be read, tested and corrected in isolation. Observe also the distinction the question tests: AddNumbers, Power, Factorial and IsPrime are Functions — each returns a value through its own name and is used inside an expression — while SwapValues and the event procedures are Sub procedures, which perform an action and return nothing. A useful rule of thumb follows: if the unit's job is to produce an answer, write a Function; if its job is to do something (display, swap, save, clear), write a Sub. Scope completes the picture: variables declared inside a procedure (Dim i) live only during that call; variables declared Public in a module are shared project-wide and should be rare; the parameters of a procedure are its controlled doorway, and a module that communicates only through parameters and return values is a module that can be lifted into another project intact.

Sample Run / Output

```
Inputs: x = 5, y = 3
Sum = 8
Power (x^y) = 125
Factorial of x = 120
```

```
x is prime? True
After swap: x = 3, y = 5
```

Part (b): How Modularity Improves Maintainability and Reusability

Maintainability is the ease with which a program can be understood, corrected and extended after it is written — and since real software spends most of its life being maintained, this quality dominates its true cost. Modularity improves it in four concrete ways, each visible in the application above. First, localisation of change: if the institution decides factorial must handle larger numbers, exactly one function is opened and edited, and every caller benefits at once; in the monolithic version the same formula might be copied in three buttons and one copy would inevitably be missed. Second, readability and naming: a procedure called IsPrime documents itself, so a maintainer reads intent instead of deciphering loops. Third, testability: Factorial can be verified alone in the Immediate window (?Factorial(5) → 120) without running the whole application, so faults are cornered in minutes. Fourth, controlled coupling: because modMath never references form controls, a change to the interface (renaming a text box) cannot break the mathematics, and vice versa — the two halves of the program can evolve independently.

Reusability is the companion benefit: a well-built module is an asset beyond the program it was written for. The entire modMath file can be added to next semester's statistics project, to the calculator of Assignment 1 extended, or to the database application of Question 5 (where a shared function might format dates or validate marks), without changing a line, precisely because it depends on nothing outside itself. Reuse multiplies with discipline: consistent naming, one job per procedure, parameters ByVal unless change is intended, and no hidden

reliance on global variables. The honest analysis must also note the costs and limits, which a postgraduate answer should not hide. Excessive fragmentation — a procedure for every two lines — creates a maze of calls that is harder to follow than the original block; passing many parameters everywhere is its own clutter; and Visual Basic's project-wide Public scope can quietly reintroduce coupling if modules start reading each other's globals. The professional judgement is balance: divide by responsibility (calculations, input/output, storage), keep procedures short enough to read on one screen, and reuse proven modules rather than rewriting proven logic.

Table 1: Monolithic versus modular organisation

Aspect	One long procedure (monolithic)	Modular (procedures and functions)
Change a formula	Find and edit every copied occurrence; risk of missing one	Edit one function; all callers updated together
Testing	Only by running the whole application	Each function testable alone (Immediate window)
Reuse in a new project	Copy–paste fragments and untangle them	Add the .bas module file unchanged
Team work	Two programmers cannot edit the same block safely	Different modules owned by different programmers
Reading the program	Intent buried in detail	Names state intent; detail read only when needed

Conclusion

In conclusion, the modular application separates what is calculated from how it is displayed: pure Functions and Sub procedures in a standard module perform the mathematics — sum, power, recursive factorial, prime testing and a ByRef swap — while short event procedures merely read inputs, call the modules and show results. The analysis shows why professionals insist on this structure: changes are localised to a single module, modules can be tested in isolation and read by name, and a clean module such as modMath can be reused in future projects without modification, provided fragmentation and global coupling are kept in check. Modularity is therefore not an academic decoration on a working program; it is the property that decides whether the program can be maintained and reused at all.

Question 2: Dynamic Menu-Driven System with Runtime Modifications — and the User-Experience Value of Menus (20 Marks)

Question as printed: a) Design and implement a dynamic menu-driven system with runtime modifications (adding/removing menu items). b) Evaluate how menu-based navigation enhances user experience in desktop applications. (20)

Introduction

The menu bar is the oldest and still the most complete navigation device of the desktop application: File, Edit, View and Help menus gather every command of a program into a predictable, browsable structure that costs no screen space until it is opened. A static menu, fixed at design time with Visual Basic's Menu Editor, is sufficient for many programs, but real

applications need menus that change while the program runs — a Window menu that lists the documents currently open, a Recent Files list that grows as files are used, a Favourites menu the user builds personally, or commands that appear only when they make sense. Visual Basic supports this through menu control arrays: menu items sharing one Name and differing by Index can be created at design time as a single seed item and then Loaded and Unloaded at run time, with their Caption, Checked, Enabled and Visible properties set from code. Part (a) implements such a dynamic system — a small text workbench whose File menu grows a Recent Files section at run time and whose Favourites menu is added to and removed from by the user — and part (b) evaluates, with both strengths and limitations, how menu-based navigation improves the desktop user experience.

Part (a): Designing and Implementing the Dynamic Menu System

In the Menu Editor, build the fixed skeleton first: a File menu (mnuFile) containing Open (mnuFileOpen), a separator, and a seed item mnuRecent with Index 0, Caption '(No recent files)', Enabled = False — this seed is essential, because Visual Basic can only Load further members of a control array that already exists at design time with index 0. Add an Edit menu with the usual commands, and a Favourites menu (mnuFavTop) containing Add Current File (mnuFavAdd), Remove Last Favourite (mnuFavRemove), a separator, and a seed item mnuFav with Index 0. Every menu item has a Click event; for control-array items the event procedure receives the Index of the item clicked, which is how one procedure serves a whole family of dynamically created items. A counter variable in the form tracks how many dynamic items currently exist.

Code 1: Adding and removing menu items at run time

```
Private mRecentCount As Integer
Private mFavCount As Integer


'--- Add a file name to the Recent list when a file is opened ---
Private Sub AddRecent(ByVal fileName As String)
   If mRecentCount = 0 Then
      mnuRecent(0).Caption = fileName
      mnuRecent(0).Enabled = True
   Else
      Load mnuRecent(mRecentCount)
      mnuRecent(mRecentCount).Caption = fileName
      mnuRecent(mRecentCount).Visible = True
   End If
   mRecentCount = mRecentCount + 1
End Sub


'--- One Click procedure serves the whole Recent array ---
Private Sub mnuRecent_Click(Index As Integer)
   MsgBox "Opening recent file: " & mnuRecent(Index).Caption, vbInformation
End Sub


'--- Favourites: add and remove at run time ---
Private Sub mnuFavAdd_Click()
   Dim itemName As String
   itemName = InputBox("Type the favourite to add:", "Add Favourite")
```

```
    If itemName = "" Then Exit Sub
    If mFavCount > 0 Then Load mnuFav(mFavCount)
    mnuFav(mFavCount).Caption = itemName
    mnuFav(mFavCount).Visible = True
    mFavCount = mFavCount + 1
End Sub

Private Sub mnuFavRemove_Click()
    If mFavCount > 1 Then
        mFavCount = mFavCount - 1
        Unload mnuFav(mFavCount)
    ElseIf mFavCount = 1 Then
        mFavCount = 0
        mnuFav(0).Caption = "(Empty)"
        mnuFav(0).Enabled = False
    End If
End Sub

Private Sub mnuFav_Click(Index As Integer)
    MsgBox "Favourite selected: " & mnuFav(Index).Caption, vbInformation
End Sub
```

The mechanics in this code are the heart of part (a) and should be explained item by item. Load mnuRecent(n) creates a new menu item as a copy of the design-time seed, inheriting its position in the menu; its Caption is then set and Visible made True. Unload removes an item

created with Load — attempting to Unload the original index-0 item raises an error, which is why the removal code above unloads only while more than one item exists and instead disables and relabels the seed when the list becomes empty. Three properties give menus their runtime personality without adding or removing anything: Enabled = False greys a command that is currently meaningless (Paste before anything is copied), Checked = True places a tick beside a state such as Word Wrap, and Visible = False hides advanced commands from beginners. Right-click pop-up menus complete the system: a menu designed with Visible = False at the top level is displayed at the mouse position with the PopupMenu method in a control's MouseUp event (testing Button = vbRightButton), giving context commands exactly where the user's attention already is.

Sample Run / Output

```
Start:          File menu shows '(No recent files)' (disabled)
Open a.txt:     File menu now lists 'a.txt' under Recent
Open b.txt:     Recent lists 'a.txt', 'b.txt' (items Loaded at run time)
Favourites -> Add 'Admissions Form': menu gains the item (mnuFav array grows)
Favourites -> Remove Last Favourite: item Unloaded; seed shows '(Empty)' when
list is bare
Right-click the text area: pop-up menu appears at the pointer (PopupMenu)
```

Part (b): How Menu-Based Navigation Enhances User Experience — an Evaluation

The strongest claim for menus is discoverability. A new user can learn what a program can do simply by opening each menu and reading it, because the complete command set is advertised

in one standard place; toolbars show only a cryptic subset in icons, and keyboard shortcuts show nothing at all. Menus therefore convert recall into recognition — the user recognises the wanted command when seen rather than memorising its name or key — which is the single most reliable principle of usable interface design, and it is why novice and occasional users, the majority in educational and office software, consistently succeed faster with menus. Organisation reinforces the effect: commands grouped by object and task (File for document-level actions, Edit for content, View for appearance) with separators, sub-menus for families, ellipsis (…) signalling that a dialogue will follow, and tick marks showing current state together form a grammar users transfer from program to program, so every new VB application feels partly familiar before it is even used.

Menus also serve experts better than they are usually credited for, and a balanced evaluation should say so. Access keys (the underlined letter, reached with Alt) and shortcut keys printed at the right of the caption (Ctrl+S) let keyboard users execute commands without the mouse, and the menu itself teaches those shortcuts, functioning as a built-in training ladder from browsing to speed. Dynamic behaviour strengthens this further: a Recent Files menu removes the tedious File → Open → browse journey for the commonest task in office work, disabled items prevent errors by making unavailable commands visibly unavailable instead of letting them fail, and context pop-up menus cut mouse travel by bringing commands to the object. Against this stand real limitations. Deep sub-menu nesting hides commands and demands precise mouse steering; very long menus slow visual search; menus are invisible until opened, so a first-time user may not think to look for a feature that a visible toolbar button would have advertised; and on small screens a full menu bar competes with content. The professional conclusion is therefore

combination, not substitution: menus as the complete, organised, self-teaching command structure, reinforced by a toolbar for the handful of most frequent commands, pop-up menus for context, and status-bar hints (the menu item's tag or a Description) explaining each command as it is highlighted.

Table 1: Menu navigation — strengths and limitations

Aspect	Strength for user experience	Limitation to manage
Discoverability	Whole command set browsable; recognition instead of recall	Hidden until opened; pair with a toolbar for key commands
Consistency	Same File/Edit/View grammar across programs; skills transfer	Poorly named or regrouped menus break the expectation
Efficiency	Access keys, shortcuts shown in captions, Recent lists, pop-ups	Deep nesting and long menus slow search and steering
Safety and clarity	Disabled items prevent invalid actions; ticks show state	Too many greyed items can confuse; hide rather than disable where suitable
Space economy	Costs one bar; commands appear only on demand	Competes for space on small displays

Conclusion

In conclusion, Visual Basic's menu control arrays make menus a living part of the interface: a design-time seed item plus Load and Unload, the Index-based Click event, and the Caption, Checked, Enabled and Visible properties allow Recent Files and Favourites lists to grow and

shrink while the program runs, and PopupMenu brings the same commands to the right mouse button. Evaluated as navigation, menus earn their central place by making an application's whole capability discoverable, consistently organised, keyboard-accessible and safe through disabling and state ticks; their weaknesses — hiddenness, nesting and length — are managed by combining them with toolbars and context menus rather than by abandoning them. A desktop application whose menus are complete, dynamic where the data is dynamic, and honestly organised gives both the beginner and the expert a map of the program, which is what navigation means.

Question 3: Processing Large Datasets with Arrays and Control Arrays — Sorting and Searching (20 Marks)

Question as printed: a) Develop an application that processes large datasets using arrays or control arrays. b) Implement sorting and searching techniques to efficiently manage and retrieve data. (20)

Introduction

Single variables suit single facts; real data arrives in crowds. A class has sixty students, a warehouse has ten thousand items, and a results system must hold every mark, total them, find the highest, sort them into merit order and retrieve any one record on demand. Visual Basic's answer is the array: one name with many numbered compartments (elements), each holding a value of the same type, addressed by an index — marks(1), marks(2), and so on — so that loops, not repeated typing, do the work. Its interface companion is the control array, a family of

controls sharing one Name and differing by Index, which lets a form grow rows of text boxes or labels at run time. Part (a) builds an application that loads a large set of student marks into parallel arrays (names and marks), computes the class statistics, and uses a control array of labels to display the results. Part (b) then implements the two operations that make stored data useful — sorting, with the bubble sort explained and timed honestly, and searching, contrasting linear search with binary search — because the efficiency difference between algorithms, trivial for ten records, decides whether a hundred thousand-record system answers in a blink or in minutes.

Part (a): The Dataset Application — Arrays and a Control Array

Declare the storage in the form's General section: two parallel arrays, Dim studentName(1 To 200) As String and Dim studentMarks(1 To 200) As Integer, together with a counter count recording how many elements are genuinely filled — the distinction between an array's capacity and its live size is fundamental, and every loop must run to count, not to the declared bound. A command button loads sample data (in a real system the same loop would read a file or the ADO recordset of Question 5), a second button computes total, average, maximum and minimum in one pass, and a third displays the data. Display uses a control array: one label lblRow with Index 0 is placed at design time inside a scrollable picture box or frame, and further labels are Loaded at run time, each positioned below the previous by setting its Top property — one event procedure and one formatting loop then serve any number of rows.

Code 1: Loading data, statistics, and display through a control array

```
Dim studentName(1 To 200) As String
Dim studentMarks(1 To 200) As Integer
```

```
Dim count As Integer


Private Sub cmdLoad_Click()
   Dim names As Variant, marks As Variant, i As Integer
   names = Array("Ali", "Sara", "Bilal", "Hina", "Usman", "Fatima")
   marks = Array(78, 92, 65, 88, 54, 92)
   count = UBound(names) + 1
   For i = 1 To count
      studentName(i) = names(i - 1)
      studentMarks(i) = marks(i - 1)
   Next i
   MsgBox count & " records loaded.", vbInformation
End Sub


Private Sub cmdStats_Click()
   Dim i As Integer, total As Integer, mx As Integer, mn As Integer
   total = 0: mx = studentMarks(1): mn = studentMarks(1)
   For i = 1 To count
      total = total + studentMarks(i)
      If studentMarks(i) > mx Then mx = studentMarks(i)
      If studentMarks(i) < mn Then mn = studentMarks(i)
   Next i
   lblStats.Caption = "Total = " & total & "  Average = " & _
      Format(total / count, "0.0") & "  Highest = " & mx & "  Lowest = " & mn
End Sub
```

USMAN NOTES www.usmannotes.com

```
Private Sub ShowRows()
   Dim i As Integer
   For i = 1 To count
      If i > 1 Then Load lblRow(i - 1)
      lblRow(i - 1).Top = 60 + (i - 1) * 300
      lblRow(i - 1).Caption = studentName(i) & " " & studentMarks(i)
      lblRow(i - 1).Visible = True
   Next i
End Sub
```

Three ideas in this short application generalise to all large-dataset work. First, one pass, many answers: the statistics loop touches each element once (an O(n) pass) and extracts total, maximum and minimum together, rather than looping three times. Second, arrays make whole-dataset operations expressible in a few lines — the identical For…Next skeleton serves loading, totalling, printing and saving, with only the loop body changing — which is why array fluency is the dividing line between toy programs and data programs. Third, control arrays mirror data arrays on the screen: just as marks(i) addresses data by number, lblRow(i) addresses its display by the same number, and both are created and destroyed programmatically (Load/Unload) instead of being drawn one by one at design time. For still larger or unpredictable volumes the same design upgrades naturally — ReDim Preserve to grow a dynamic array without losing its contents, and a ListBox, MSFlexGrid or database grid in place of labels — but the logic of index, count and loop remains identical.

Part (b): Sorting and Searching

Sorting rearranges the array into order so that lists read naturally and, crucially, so that fast searching becomes possible. The bubble sort is the teaching standard because its idea is visible: compare each neighbouring pair, swap them if they are out of order, and repeat passes until a pass makes no swap; after each pass the largest unsorted value has 'bubbled' to its final position. To keep the parallel arrays consistent, every swap of marks must swap the names in the same instant — forgetting the companion swap is the classic parallel-array bug, producing a merit list with the right marks attached to the wrong students. Bubble sort costs roughly $n^2/2$ comparisons (for n = 1,000, about half a million), which is acceptable for classroom sizes but slow at scale; the answer should state honestly that production systems use faster algorithms (Shell sort, or quicksort, roughly n log n) or, better, let the database sort with ORDER BY — while the bubble sort remains the clearest way to learn what sorting is.

Code 2: Bubble sort (parallel arrays) and binary search

```
Private Sub BubbleSort()
   Dim i As Integer, j As Integer, tempM As Integer, tempN As String
   Dim swapped As Boolean
   For i = 1 To count - 1
      swapped = False
      For j = 1 To count - i
         If studentMarks(j) < studentMarks(j + 1) Then
            tempM = studentMarks(j): studentMarks(j) = studentMarks(j + 1):
            studentMarks(j + 1) = tempM
```

```
            tempN = studentName(j): studentName(j) = studentName(j + 1):
            studentName(j + 1) = tempN
            swapped = True
         End If
      Next j
      If Not swapped Then Exit For
   Next i
End Sub

Private Function BinarySearch(ByVal target As String) As Integer
   Dim lo As Integer, hi As Integer, mid As Integer
   lo = 1: hi = count
   Do While lo <= hi
      mid = (lo + hi) \ 2
      If studentName(mid) = target Then
         BinarySearch = mid: Exit Function
      ElseIf studentName(mid) < target Then
         lo = mid + 1
      Else
         hi = mid - 1
      End If
   Loop
   BinarySearch = 0
End Function
```

USMAN NOTES www.usmannotes.com

Searching answers the retrieval half of the question. Linear search inspects elements one by one from the start: it needs no preparation and works on unsorted data, but its worst case inspects every element — on average n/2 — so a million-record file averages half a million inspections per query. Binary search exploits sorted order: compare the target with the middle element, discard the half that cannot contain it, and repeat, each step halves the territory, so a million records need at most 20 comparisons ($2^{20} \approx 1{,}000{,}000$) — the difference between an instant answer and a noticeable wait, multiplied by every query every day. Its precondition is absolute: the data must be sorted on the searched key, and searching an unsorted array with binary search returns wrong 'not found' results silently, the most dangerous kind of program error. The chart below quantifies the contrast that part (b) asks the student to manage: algorithm choice, not computer speed, is what makes large datasets tractable, and the professional habit is to sort once (an O(n log n) investment with a proper algorithm) and then search cheaply forever, or to keep data in a database whose indexes do the same work automatically.

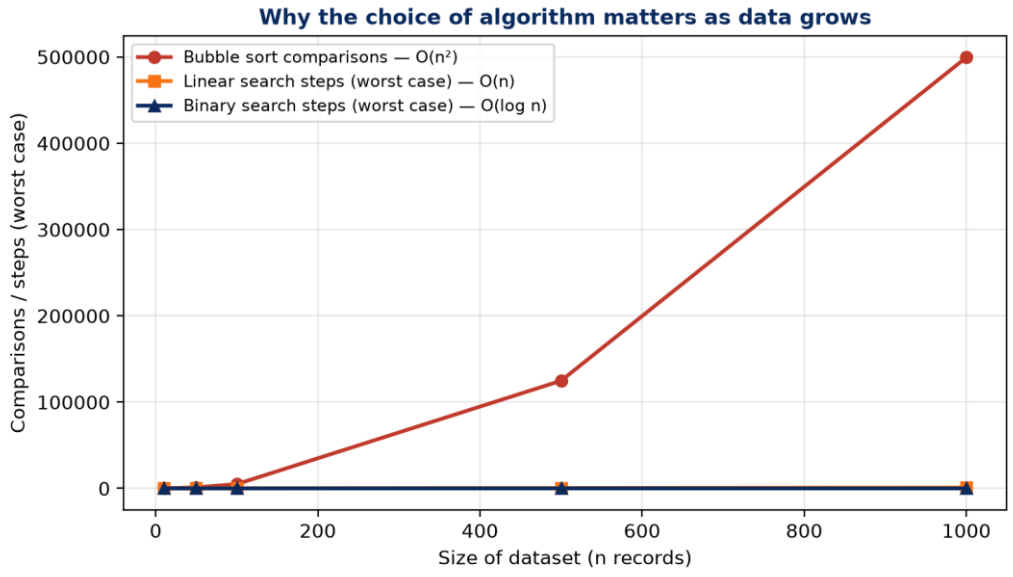


Figure 1: Growth of work with dataset size — bubble sort comparisons rise with n², linear search with n, binary search with log n.

Table 1: Sorting and searching techniques compared

Technique	Idea	Worst-case cost	Precondition / note
Bubble sort	Swap out-of-order neighbours, pass by pass	O(n²) comparisons	Simple to learn; fine for small n; swap parallel arrays together
Selection / Shell / quicksort	Select minimum each pass / gap comparisons / partition recursively	O(n²) / about O(n^1.5) / O(n log n) average	Preferred as n grows; quicksort is the usual library choice
Linear search	Inspect elements in order until found	O(n)	Works on unsorted data; only option without order
Binary search	Halve the sorted range at each step	O(log n)	Data MUST be sorted on the search key

Sample Run / Output

```
Loaded 6 records. Statistics: Total = 469  Average = 78.2  Highest = 92
Lowest = 54
After BubbleSort (descending by marks): Sara 92, Fatima 92, Hina 88, Ali 78,
Bilal 65, Usman 54
BinarySearch("Hina") after sorting by name -> position found;
BinarySearch("Zara") -> 0 (not found)
```

Conclusion

In conclusion, arrays turn Visual Basic from a calculator into a data processor: parallel arrays with a live count hold the dataset, a single loop extracts totals, averages and extremes, and a control array of labels displays any number of rows from one design-time seed. On that foundation, sorting and searching supply management and retrieval: the bubble sort — swapping names with marks, and stopping early on a swap-free pass — orders the data at $O(n^2)$ cost suitable for modest sizes, linear search retrieves from any order at O(n), and binary search, at O(log n), shows why sorted data is valuable, finding one record among a million in about twenty comparisons. The enduring lesson is that efficiency at scale comes from choosing the right structure and algorithm, a principle that leads directly to indexed databases in Question 5.

Question 4: Graphical Application with Drawing Controls — an Interactive Dashboard (20 Marks)

Question as printed: a) Design a graphical application utilizing drawing controls (lines, shapes, images) to simulate a simple interface (e.g., dashboard or diagram). b) Incorporate user interaction to dynamically update graphical elements. (20)

Introduction

Not every Visual Basic screen is boxes and buttons. Timetables need diagrams, laboratories need gauges, and managers need dashboards — screens where information is drawn: bars whose heights mean results, shapes whose colours mean status, and pictures that orient the user at a glance. Visual Basic provides two families of graphical tools, and a good answer must

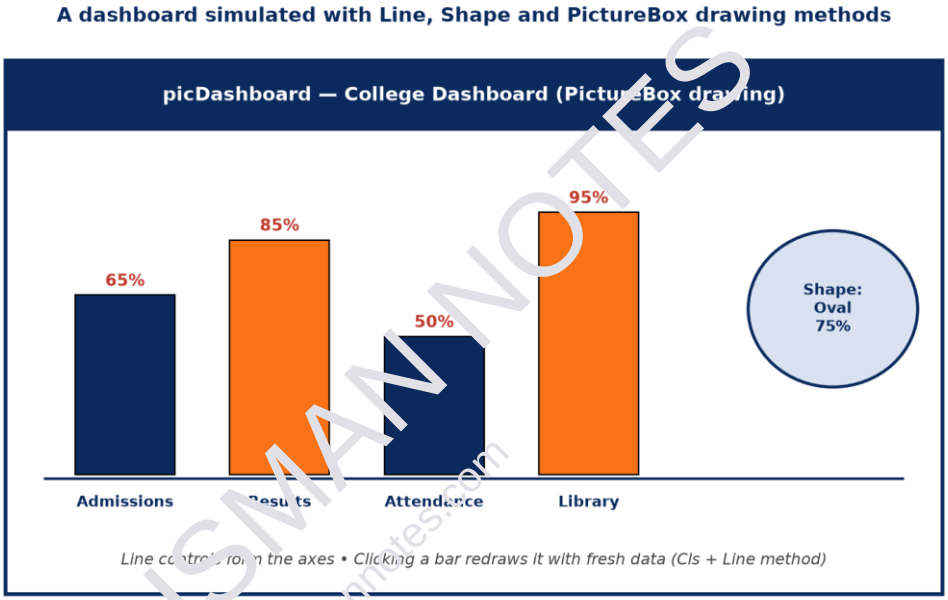
distinguish them. The graphical controls — Line, Shape and Image/PictureBox placed from the Toolbox — are objects with properties (BorderColor, FillColor, BorderWidth) that can be set at design time or changed at run time; they are easy and persistent, ideal for fixed decoration and status indicators. The graphics methods — Line, Circle, PSet and Print used in code on a form or PictureBox — draw by instruction at run time, so they can draw what cannot be known in advance: a bar chart of whatever data the user loads, a curve of whatever function is typed. Part (a) designs a college dashboard that uses both families together, and part (b) makes it interactive: sliders, buttons and a timer change the drawing dynamically, which is where the distinction between controls (moved and recoloured) and methods (cleared with Cls and redrawn) becomes a practical skill.

USMAN NOTES www.usmannotes.com

Part (a): Designing the Dashboard

Create frmDashboard with a large PictureBox, picBoard, occupying most of the form; all free drawing will happen inside it, so the rest of the interface is protected from overpainting, and its ScaleMode and coordinate system can be set once (ScaleMode = 3 – Pixel, or a custom scale with the Scale method). Around and inside the picture box, place the control-based graphics: Line controls form the axes and separators of the chart area (BorderWidth 2, BorderColor dark blue); Shape controls provide the status blocks — rectangles and ovals whose Shape property selects Rectangle, Rounded Rectangle, Oval or Circle and whose FillStyle and FillColor show green, amber or red status at a glance; an Image control displays the institution's logo (Picture property loaded at design time, Stretch = True so it fits its frame). The method-drawn element is the bar chart itself: a DrawChart procedure reads four values (Admissions, Results, Attendance, Library use) from an array and, using picBoard.Line with the BF (box, filled) option,

draws one filled bar per value with its percentage printed above it by CurrentX/CurrentY and Print.



USMAN NOTES www.usmannotes.com

Figure 1: The dashboard simulated with Line and Shape controls, an Image, and bars drawn at run time with the Line method.

Code 1: Drawing the bar chart with graphics methods

```
Dim mValues(1 To 4) As Integer
Dim mTitles(1 To 4) As String


Private Sub DrawChart()
   Dim i As Integer, barH As Single, x As Single
   picBoard.Cls
   picBoard.Line (400, 200)-(400, 3400), vbBlue
   picBoard.Line (400, 3400)-(4400, 3400), vbBlue
   For i = 1 To 4
```

```
        barH = mValues(i) * 30
        x = 400 + i * 900
        picBoard.Line (x, 3400 - barH)-Step(500, barH), _
           IIf(i Mod 2 = 0, RGB(249, 115, 22), RGB(10, 42, 94)), BF
        picBoard.CurrentX = x
        picBoard.CurrentY = 3450
        picBoard.Print mTitles(i) & " " & mValues(i) & "%"
    Next i
End Sub

Private Sub Form_Load()
    mTitles(1) = "Admissions": mValues(1) = 65
    mTitles(2) = "Results": mValues(2) = 85
    mTitles(3) = "Attendance": mValues(3) = 50
    mTitles(4) = "Library": mValues(4) = 95
    DrawChart
    UpdateStatus
End Sub

Private Sub UpdateStatus()
    Dim i As Integer
    For i = 1 To 4
        If mValues(i) >= 75 Then
            shpStatus(i - 1).FillColor = vbGreen
        ElseIf mValues(i) >= 50 Then
            shpStatus(i - 1).FillColor = RGB(255, 191, 0)
```

```
        Else
           shpStatus(i - 1).FillColor = vbRed
        End If
     Next i
  End Sub
```

Study the two techniques side by side, because the examination rewards the comparison. The Shape controls (here a control array shpStatus(0) to shpStatus(3), echoing Question 3) are never redrawn: the program simply assigns a new FillColor and Visual Basic repaints them automatically and permanently. The bar chart, by contrast, is redrawn from scratch on every change: Cls erases the PictureBox, and the Line calls rebuild it from the current data — which is why picBoard.AutoRedraw should be set to True, so the drawing also survives the form being covered and uncovered. Coordinates deserve one careful sentence: in the default Twips scale (1440 twips to the inch) positions are fine-grained but unintuitive, so dashboards usually switch the container to pixels or define a custom scale — for a chart, picBoard.Scale (0, 110)-(100, 0) makes the Y axis read directly in percentage, turning data into coordinates with no arithmetic at all. Colours are specified with the built-in constants (vbRed, vbBlue) or composed with RGB(r, g, b), as the orange bar in the code shows.

Part (b): User Interaction that Updates the Graphics Dynamically

A dashboard that cannot change is a poster. Three interaction mechanisms, in increasing liveliness, make this one dynamic. First, direct manipulation: two scroll/slider controls (or simple + and − command buttons) beside the chart let the user select a category and adjust its

value; the Change event updates the data array, calls DrawChart to redraw the affected display, and calls UpdateStatus so the traffic-light shape recolours in the same instant — the user sees cause and effect together, which is the essence of a simulation. Second, mouse interaction with the drawing itself: the PictureBox MouseDown event reports the X and Y of the click in the box's own scale, so the program can test which bar was clicked and show its exact figures in a label or MsgBox; MouseMove can similarly drive a highlight. Third, time-based animation: a Timer control with Interval 1000 (one second) fires its Timer event repeatedly, and in a 'live' mode each tick nudges the values with a small random change (Rnd) and redraws, simulating a real-time monitoring screen; a Start/Stop button toggles Timer.Enabled, teaching the essential discipline that animation is a loop the user must be able to stop.

Code 2: Interaction – slider change, bar click, and timer refresh

```
Private Sub hsbValue_Change()
   Dim cat As Integer
   cat = cboCategory.ListIndex + 1
   If cat < 1 Then Exit Sub
   mValues(cat) = hsbValue.Value
   DrawChart
   UpdateStatus
End Sub


Private Sub picBoard_MouseDown(Button As Integer, Shift As Integer, X As
Single, Y As Single)
   lblInfo.Caption = "Clicked at X = " & X & ", Y = " & Y & " (PictureBox
   scale units)"
```

```
End Sub


Private Sub cmdLive_Click()
   tmrLive.Enabled = Not tmrLive.Enabled
   cmdLive.Caption = IIf(tmrLive.Enabled, "Stop Live", "Start Live")
End Sub


Private Sub tmrLive_Timer()
   Dim i As Integer
   For i = 1 To 4
      mValues(i) = mValues(i) + Int(Rnd * 7) - 3
      If mValues(i) < 0 Then mValues(i) = 0
      If mValues(i) > 100 Then mValues(i) = 100
   Next i
   DrawChart
   UpdateStatus
End Sub
```

Table 1: Graphical controls and methods compared

Tool	Kind	How changed at run time	Best for
Line control	Control (Toolbox)	Move endpoints; set BorderColor/Width	Fixed axes, separators, diagram lines
Shape control	Control (Toolbox)	Set Shape, FillColor,	Status blocks, traffic

		FillStyle	lights, diagram nodes
Image / PictureBox	Control (container)	LoadPicture; Stretch (Image)	Logos, photos; PictureBox also hosts drawing
Line / Circle / PSet methods	Code on Form/PictureBox	Cls then redraw from data	Charts and drawings driven by changing data
Print method	Code	CurrentX/CurrentY then Print	Labels and values inside a drawn area

USMAN NOTES www.usmannotes.com

Interaction design carries obligations as well as possibilities, and the answer should name them. Redrawing must be complete and consistent — data, bars, printed percentages and status colours all derive from the same array in the same refresh procedures, so no element can contradict another. Random live data must be clamped (the 0–100 guards in the timer code) so the simulation never draws a bar off the chart. And graphics must remain honest: bar heights proportional to values from a zero baseline, colours used consistently (red always means attention), because a dashboard's whole purpose is truthful communication at speed. With those disciplines observed, extensions are natural — clicking a bar could open the detail form of Assignment 1's Question 4 pattern, and the chart data could be read from the database of Question 5, turning the exercise into the front screen of a real management system.

Sample Run / Output

```
Load: bars 65 / 85 / 50 / 95; status shapes amber, green, amber, green
```

```
Slider: Attendance raised 50 -> 78; bar redraws taller; its status shape
turns green
Live mode ON: values drift each second (e.g. 66 / 84 / 79 / 93); chart and
shapes refresh together
Click on chart: label reports the click coordinates in the PictureBox scale
```

Conclusion

In conclusion, Visual Basic graphics divide into placed controls and drawn methods, and a convincing dashboard uses both: Line and Shape controls and an Image give the fixed frame, axes, status lights and branding, while the Line method with BF, Circle, PSet and Print draw the data-driven chart inside a PictureBox, erased with Cls and rebuilt from the data array on every change. Interaction then closes the loop — sliders and buttons edit the data, MouseDown interrogates the drawing, and a Timer animates it as a live display — with a single DrawChart and UpdateStatus pair guaranteeing that every visual element always agrees with the same underlying numbers. These are exactly the techniques behind timetable diagrams, laboratory simulations and management dashboards in real VB applications.

Question 5: Database-Driven Application with ADO — Full CRUD Operations and SQL (20 Marks)

Question as printed: a) Develop a database-driven application using ADO to manage records (e.g., student or employee system). b) Implement full CRUD operations and integrate SQL queries for efficient data handling. (20)

Introduction

Everything in this course converges on this question. Forms (Assignment 1), modules, arrays and menus (Assignment 2) are all, in professional practice, servants of one master: persistent data. A program that forgets everything when it closes is a demonstration; a program connected to a database is a system. Visual Basic 6 reaches databases through ADO — ActiveX Data Objects — Microsoft's object layer that lets a VB program open a connection to Access, SQL Server or almost any other database through an OLE DB provider, run SQL statements, and work with the returned rows as a Recordset object. This answer builds a complete student record system on an Access database: a Students table, a form carrying the familiar fields of Assignment 1 (now bound to real storage), and the four operations that define record management — Create, Read, Update and Delete, universally abbreviated CRUD — each implemented twice over, once through the Recordset's own methods and once through SQL executed on the Connection, because part (b) explicitly demands SQL integration. Search, navigation and the efficiency and safety rules (parameter thinking, closing objects, handling the empty table) complete the answer.

Part (a): The ADO Application — Design and Connection

First create the database. In Microsoft Access create StudentsDB.mdb with one table, Students, whose fields are: StudentID (AutoNumber, primary key), StudentName (Text, required), RollNo (Text), Programme (Text), Marks (Number). The primary key matters conceptually: it gives every record a unique identity, which is what allows Update and Delete to strike exactly one record and never a namesake. In Visual Basic, add the ADO library through Project →

References → 'Microsoft ActiveX Data Objects 2.8 Library'; without this reference the ADODB object names are unknown to the compiler. Declare the two working objects in the form's General section — a Connection (cn) representing the open channel to the database, and a Recordset (rs) representing the current set of rows — and open them in Form_Load with a connection string naming the Jet OLE DB provider for Access (for SQL Server the provider becomes SQLOLEDB and the string adds server, database and credentials). Finally, a DisplayRecord procedure copies the current row's fields into the form's text boxes, and navigation buttons move through the recordset with MoveFirst, MovePrevious, MoveNext and MoveLast, guarding the ends with BOF and EOF tests.

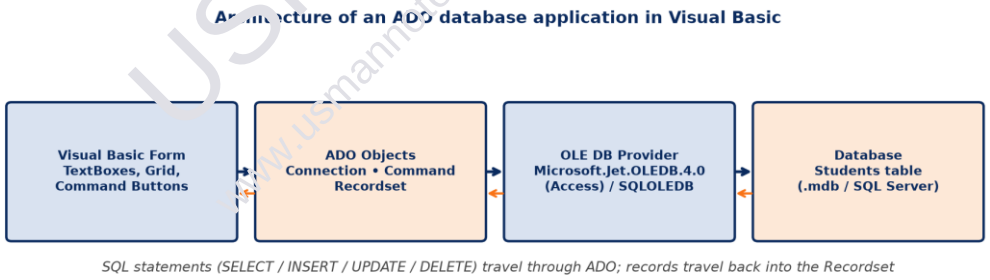


Figure 1: Architecture of the ADO application — form, ADO objects, OLE DB provider and database.

Table 1: The ADO object model used in this application

Object	Role	Key members used
Connection (ADODB.Connection)	Opens and owns the channel to the database; executes SQL directly	ConnectionString, Open, Execute, Close
Recordset (ADODB.Recordset)	Holds the rows returned by a query; the program reads and	Open, Fields, MoveNext/MovePrevious,

	edits through it	AddNew, Update, Delete, EOF/BOF
Command (ADODB.Command)	Packages a SQL statement, especially with parameters, for repeated or safer execution	CommandText, Parameters, Execute

Code 1: Connection, opening the recordset, display and navigation

```
Dim cn As ADODB.Connection
Dim rs As ADODB.Recordset

Private Sub Form_Load()
   Set cn = New ADODB.Connection
   cn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Data Source=" & App.Path & "\StudentsDB.mdb"
   cn.Open
   Set rs = New ADODB.Recordset
   rs.Open "SELECT * FROM Students ORDER BY StudentName", cn, _
      adOpenDynamic, adLockOptimistic
   If Not rs.EOF Then DisplayRecord
End Sub

Private Sub DisplayRecord()
   txtID.Text = rs.Fields("StudentID").Value & ""
   txtName.Text = rs.Fields("StudentName").Value & ""
   txtRoll.Text = rs.Fields("RollNo").Value & ""
```

```
    txtProgramme.Text = rs.Fields("Programme").Value & ""
    txtMarks.Text = rs.Fields("Marks").Value & ""
End Sub

Private Sub cmdNext_Click()
    If rs.EOF Then Exit Sub
    rs.MoveNext
    If rs.EOF Then rs.MoveLast
    DisplayRecord
End Sub

Private Sub Form_Unload(Cancel As Integer)
    If Not rs Is Nothing Then rs.Close
    If Not cn Is Nothing Then cn.Close
End Sub
```

USMAN NOTES www.usmannotes.com

Two arguments of rs.Open repay explanation because they decide whether editing is even possible. The cursor type adOpenDynamic lets the recordset reflect a live, movable, updatable view of the data; the lock type adLockOptimistic permits edits, locking a row only at the moment Update writes it — the right choice for a single-user or lightly shared student system (adLockPessimistic and adLockReadOnly serve other situations, and opening with a read-only lock is the classic cause of the beginner's complaint that 'the record will not save'). The trailing & "" in DisplayRecord is a deliberate guard from Assignment 1's error table: it converts a Null database value into an empty string instead of raising error 94 when it is assigned to a text box.

Closing rs and cn in Form_Unload is equally deliberate housekeeping — an application that leaks open connections will eventually be refused by the database.

Part (b): CRUD Operations with SQL

CRUD maps one-to-one onto both the Recordset methods and the SQL verbs, and the table below states the correspondence that this whole question turns on. Create inserts a new student: with the Recordset, AddNew blanks the fields for entry and Update commits them after the text boxes are copied in; with SQL, the equivalent is an INSERT INTO statement executed on the Connection. Read is the SELECT query — the recordset itself was created by one — refined with WHERE for search (for example, finding a roll number or all students above 80 marks) and ORDER BY for presentation. Update writes changed values for one identified record (UPDATE … WHERE StudentID = …), the WHERE clause being the safety-critical part: omit it and the statement updates every student in the institution. Delete removes one identified record (DELETE FROM … WHERE StudentID = …), and professional practice surrounds it with a confirmation MsgBox, because unlike a text box there is no undo for a deleted database row.

Table 2: CRUD — Recordset methods and their SQL equivalents

Operation	Recordset method	SQL statement (executed via cn.Execute)	Purpose
Create	rs.AddNew … rs.Update	INSERT INTO Students (StudentName, RollNo, Programme, Marks)	Add a new student

		VALUES ('Sara Khan','2026-CS-021','BSCS', 88)	
Read	rs.Open with a query; Fields; Move…	SELECT * FROM Students WHERE Marks >= 80 ORDER BY Marks DESC	Retrieve and display records
Update	Edit fields … rs.Update	UPDATE Students SET Marks = 91 WHERE StudentID = 7	Change one identified record
Delete	rs.Delete	DELETE FROM Students WHERE StudentID = 7	Remove one identified record

USMAN NOTES www.usmannotes.com

Code 2: Add, save (SQL INSERT), update, delete and search

```
Private Sub cmdAddSQL_Click()
    Dim sql As String
    If Trim(txtName.Text) = "" Or Not IsNumeric(txtMarks.Text) Then
        MsgBox "Name is required and Marks must be numeric.", vbExclamation
        Exit Sub
    End If
    sql = "INSERT INTO Students (StudentName, RollNo, Programme, Marks) VALUES
    ('" & _
        txtName.Text & "', '" & txtRoll.Text & "', '" & txtProgramme.Text & "',
        " & _
```

```
      Val(txtMarks.Text) & ")"
   cn.Execute sql
   rs.Requery
   MsgBox "Record added.", vbInformation
End Sub

Private Sub cmdUpdateSQL_Click()
   cn.Execute "UPDATE Students SET Marks = " & Val(txtMarks.Text) & _
      " WHERE StudentID = " & Val(txtID.Text)
   rs.Requery
   MsgBox "Record updated.", vbInformation
End Sub

Private Sub cmdDeleteSQL_Click()
   If MsgBox("Delete this student permanently?", vbYesNo + vbQuestion) = vbYes
   Then
      cn.Execute "DELETE FROM Students WHERE StudentID = " & Val(txtID.Text)
      rs.Requery
      If Not rs.EOF Then DisplayRecord
   End If
End Sub

Private Sub cmdSearch_Click()
   Dim rsFind As ADODB.Recordset
   Set rsFind = New ADODB.Recordset
```

USMAN NOTES www.usmannotes.com

```
    rsFind.Open "SELECT * FROM Students WHERE RollNo = '" & txtRoll.Text & "'",
    _
       cn, adOpenStatic, adLockReadOnly
    If rsFind.EOF Then
       MsgBox "No student found with that roll number.", vbInformation
    Else
       MsgBox "Found: " & rsFind.Fields("StudentName").Value & _
          " — Marks: " & rsFind.Fields("Marks").Value, vbInformation
    End If
    rsFind.Close
End Sub
```

Efficiency and safety in SQL handling follow from how these statements are used, and the answer should state the four governing rules. First, ask the database for only what is needed: name the required columns instead of SELECT * on wide tables, and filter with WHERE on the server rather than fetching ten thousand rows and discarding them in VB — the database's indexes (Question 3's binary-search principle, industrialised) then do the retrieval work in milliseconds. Second, after any change executed through the Connection, call rs.Requery so the form's recordset reflects the new reality; a stale display after a successful save is a state-management failure of exactly the kind Assignment 1 examined. Third, validate before constructing SQL (the IsNumeric guard in the Add code) and prefer ADODB.Command with parameters for production code: parameters treat user text strictly as data, which both prevents broken statements when a name contains an apostrophe (O'Brien) and blocks SQL injection, the attack in which typed text is crafted to alter the statement itself — string-built

SQL, shown here because its mechanics must be understood, should be hardened in real systems. Fourth, wrap database operations in the error handling of Assignment 1, Question 5 (On Error GoTo with Err.Number and Description), because connections fail, files move and servers stop, and a database program without a handler crashes at exactly the moment the user most needs reassurance that their data is safe.

Sample Run / Output

```
Add 'Sara Khan / 2026-CS-021 / BS CS / 88' -> INSERT executed; Requery; grid
shows the new row
Search RollNo 2026-CS-021 -> "Found: Sara Khan — Marks: 88"
Update Marks to 91 -> UPDATE ... WHERE StudentID = 7; only Sara's row changes
Delete with confirmation -> DELETE ... WHERE StudentID = 7; record gone after
Requery
Read report -> SELECT ... WHERE Marks >= 80 ORDER BY Marks DESC lists merit
students
```

Conclusion

In conclusion, the ADO student system completes the course's journey from screens to systems: a Connection opened through an OLE DB provider, a Recordset opened with a dynamic cursor and optimistic locking, fields displayed and navigated with BOF/EOF discipline, and the full CRUD cycle implemented in both its forms — Recordset methods (AddNew, Update, Delete) and SQL verbs (INSERT, SELECT, UPDATE, DELETE) executed on the Connection, always with a WHERE clause guarding Update and Delete, a confirmation before deletion, Requery after every change, and validation, parameter-awareness and error handling carried over from the earlier

questions. A student who can build this application, and explain why each object, argument and clause is present, possesses the working core of every record-keeping system — admissions, payroll, inventory or library — that Visual Basic was used to build.

USMAN NOTES
www.usmannotes.com