

CCT College Dublin Continuous Assessment

Programme Title:	<i>H.Dip. in Computing</i>		
Delivery Mode:	FT		
Cohort Details:	Feb 2025 cohort		
Module Title(s):	Algorithms & Constructs Software Development Fundamentals		
Assignment Type:	<i>Integrated / Individual</i>	Weighting(s):	Algorithms & Constructs – 60% Software Development Fundamentals – 55%
Assignment Title:	<i>System Modelling & Build</i>		
Lecturer(s):	Ken Healy / Taufique Ahmed		
Issue Date:	28 th Oct 2025		
Submission Deadline Date:	Saturday 29 th Nov at 11.59pm		
Late Submission Penalty:	Late submissions will be accepted up to 5 calendar days after the deadline. All late submissions are subject to a penalty of 10% <u>of the mark awarded</u> . Submissions received more than 5 calendar days after the deadline above will not be accepted and a mark of 0% will be awarded.		
Method of Submission:	This assignment is submitted via Moodle.		
Instructions for Submission:	<p>Algorithms & Constructs A GitHub repo must be created. The Netbeans project and report Word document must be put into a GitHub repo for version control. The lecturer must be added to the repository as a collaborator. The GitHub repo link will be added at the end of the report. There should be at least 10 to 15 commits throughout the time worked on the project. Submit your Netbeans .java files individually, and make sure all files contain the same package name, CA_2. If your submission does not load or run, then you may receive a zero mark! Include a section (max 500 words) in your Report submission that clearly explains your choices for the sorting & searching algorithms that you have used.</p> <p>Software Development Fundamentals</p> <p>Your design diagrams, user stories, etc. must be submitted in one document in PDF format to the SDF Moodle page. You do not need to include this in your GitHub Repo. There will be an IN-CLASS Q&A held on Tues 2nd Dec. You may be asked to explain any part of your submission. Late submissions will be required to attend a separate Q&A in week commencing 8th Dec.</p>		
Feedback Method:	Results posted in Moodle gradebook		
Feedback Date:	<i>Following release of approved results by the college</i>		

Introduction

The project deals with designing a system known as the Department Store Employee Management System, using the Java programming language in the NetBeans development environment. The system shall be capable of storing, organizing, and retrieving employee information like the employee's name, type of manager, department, and every piece of relevant information in an effective and reliable way.

In large and complex systems, such as department store chains, efficient management of employee information is the key to rapid decision-making, manpower planning, and administrative report generation. To this end, sorting and searching algorithms are used to ensure that performance will be consistent despite the growth of databases.

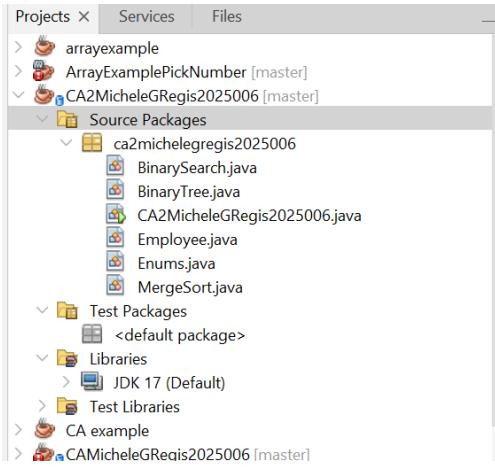
Version control is also used on this project through GitHub, which provides tracking of changes, facilitates collaboration, and offers a record of the history of development. This report describes the chosen algorithms-Merge Sort for sorting and Binary Search for searching-and gives justifications for each selection based on performance, scalability, and reliability. The system was designed to be expandable, allowing for future implementations such as secondary sorting, range queries, and graphical or reporting interfaces.

GitHub-

<https://github.com/MicheleRegis/CA2MicheleGRegis2025006.git>

Algorithms and Constructs

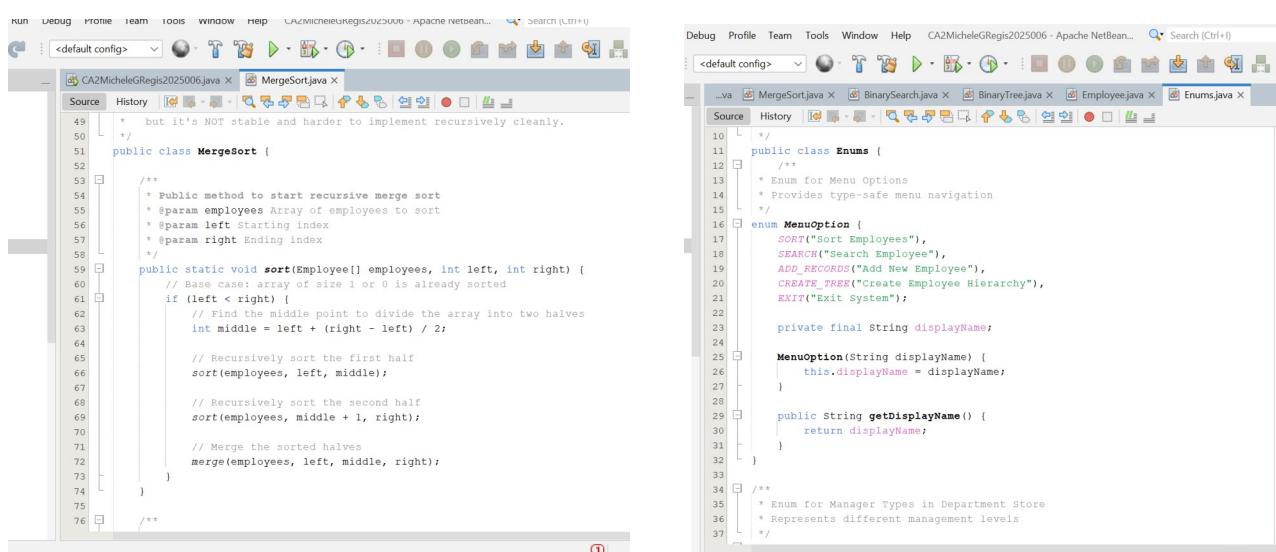
The Department Employee Management System



Sorting Algorithm: Merge Sort

Main Reason

I chose Merge Sort for the recursive sorting algorithm for this department store system because it guarantees an $O(n \log n)$ time complexity in all cases: best, average, and worst. Unlike Quick Sort, it doesn't degrade to $O(n^2)$ with poorly chosen pivots or already-sorted data. Merge Sort exhibits predictable, reliable performance irrespective of input patterns, which is very important in a production environment serving department store managers.



Key Advantages

Another important aspect is stability. The Merge Sort algorithm is a stable sort, meaning that equal elements remain in their original order. This becomes useful later if we extend our system to include secondary sorting criteria, such as employee ID or hire date.

The problem fits perfectly with the divide-and-conquer recursive approach. Merge Sort naturally splits the problem into smaller subproblems, sorts them recursively, and then merges the results. This is a very elegant recursive structure that makes the code maintainable, with proper algorithm design principles in mind.

Scalability Considerations

With the increase of department stores, employee databases grow from hundreds to possibly thousands of records. This $O(n \log n)$ complexity of Merge Sort makes the system scale well. If the number of employees is doubled, the time taken increases a bit more than double—not that quadratic explosion as in the simpler $O(n^2)$ algorithms like Bubble or Insertion Sort.

Trade-Offs Acknowledged

The main trade-off here is space complexity: Merge Sort requires $O(n)$ auxiliary space for the merging operation. However, on modern systems, this is usually acceptable given the reliability benefits that come with it. I have taken the approach of sacrificing memory to keep up consistent performance. Besides, the extra memory consumption is temporary and freed right after the sorting is done.

Searching Algorithm:Binary Search

Core Reasoning

Binary search was the obvious choice for looking up employees, since it provides a time complexity of $O(\log n)$.

Since our data is already sorted through Merge Sort, Binary Search takes advantage of this structure to locate employees with logarithmic efficiency. For 1,000 employees, it only takes roughly 10 comparisons versus the average of 500 comparisons in Linear Search.

```
<default config> CA2MicheleGRegis2025006.java MergeSortJava BinarySearch.java
Source History Tools Window Help CA2MicheleGRegis2025006 - Apache NetBean... Search (Ctrl+F)

55     * Search is the textbook optimal solution for this specific requirement,
56     * balancing speed, simplicity, and space efficiency perfectly.
57     */
58
59     public class BinarySearch {
60
61         /**
62          * Performs binary search to find employee by name
63          * @param employees Sorted array of employees
64          * @param targetName Name to search for
65          * @return Index of employee if found, -1 otherwise
66         */
67
68         public static int search(Employee[] employees, String targetName) {
69             // Validate input
70             if (employees == null || employees.length == 0 || targetName == null) {
71                 return -1;
72             }
73
74             // Initialize search boundaries
75             int left = 0;
76             int right = employees.length - 1;
77
78             // Iterative binary search
79             while (left <= right) {
80                 // Calculate middle index (prevents integer overflow)
81                 int middle = left + (right - left) / 2;
82
83                 // Get middle employee for comparison
84                 Employee middleEmployee = employees[middle];
85
86                 if (middleEmployee.getName().equals(targetName)) {
87                     return middle;
88                 } else if (middleEmployee.getName().compareTo(targetName) < 0) {
89                     left = middle + 1;
90                 } else {
91                     right = middle - 1;
92                 }
93             }
94
95             return -1;
96         }
97     }

16     * from left to right before moving to the next level.
17     */
18
19     public class BinaryTree {
20
21         /**
22          * Inner class representing a node in the binary tree
23          * Each node stores an Employee and references to left and right children
24         */
25
26         private class TreeNode {
27             Employee employee;
28             TreeNode left;
29             TreeNode right;
30
31             /**
32              * Constructor for TreeNode
33              * @param employee Employee data to store in this node
34             */
35
36             TreeNode(Employee employee) {
37                 this.employee = employee;
38                 this.left = null;
39                 this.right = null;
40             }
41
42         }
43
44         private TreeNode root; // Root of the binary tree
45         private int nodeCount; // Total number of nodes in tree
46
47         /**
48          * Insert a new employee into the binary tree
49          * @param employee Employee to insert
50         */
51
52         void insert(Employee employee) {
53             if (root == null) {
54                 root = new TreeNode(employee);
55             } else {
56                 insertNode(root, employee);
57             }
58         }
59
60         /**
61          * Insert a new employee into the binary tree
62          * @param employee Employee to insert
63         */
64         void insertNode(TreeNode node, Employee employee) {
65             if (employee.getName().compareTo(node.employee.getName()) < 0) {
66                 if (node.left == null) {
67                     node.left = new TreeNode(employee);
68                 } else {
69                     insertNode(node.left, employee);
70                 }
71             } else {
72                 if (node.right == null) {
73                     node.right = new TreeNode(employee);
74                 } else {
75                     insertNode(node.right, employee);
76                 }
77             }
78         }
79
80         /**
81          * Find an employee in the binary tree
82          * @param targetName Name to search for
83          * @return Employee object if found, null otherwise
84         */
85         Employee find(String targetName) {
86             return findNode(root, targetName);
87         }
88
89         /**
90          * Find an employee in the binary tree
91          * @param node Current node being searched
92          * @param targetName Name to search for
93          * @return Employee object if found, null otherwise
94         */
95         Employee findNode(TreeNode node, String targetName) {
96             if (node == null) {
97                 return null;
98             }
99             if (node.employee.getName().equals(targetName)) {
100                 return node.employee;
101             } else if (node.employee.getName().compareTo(targetName) < 0) {
102                 return findNode(node.left, targetName);
103             } else {
104                 return findNode(node.right, targetName);
105             }
106         }
107
108         /**
109          * Print all employees in the binary tree
110         */
111         void print() {
112             printNode(root);
113         }
114
115         /**
116          * Print all employees in the binary tree
117          * @param node Current node being printed
118         */
119         void printNode(TreeNode node) {
120             if (node != null) {
121                 System.out.println(node.employee);
122                 printNode(node.left);
123                 printNode(node.right);
124             }
125         }
126
127         /**
128          * Calculate the total number of nodes in the binary tree
129          */
130         int countNodes() {
131             return nodeCount;
132         }
133
134         /**
135          * Calculate the height of the binary tree
136          */
137         int calculateHeight(TreeNode node) {
138             if (node == null) {
139                 return 0;
140             }
141             int leftHeight = calculateHeight(node.left);
142             int rightHeight = calculateHeight(node.right);
143
144             if (leftHeight > rightHeight) {
145                 return leftHeight + 1;
146             } else {
147                 return rightHeight + 1;
148             }
149         }
150
151         /**
152          * Calculate the height of the binary tree
153          */
154         int calculateHeight() {
155             return calculateHeight(root);
156         }
157
158         /**
159          * Calculate the width of the binary tree
160          */
161         int calculateWidth(TreeNode node) {
162             if (node == null) {
163                 return 0;
164             }
165             int leftWidth = calculateWidth(node.left);
166             int rightWidth = calculateWidth(node.right);
167
168             if (leftWidth > rightWidth) {
169                 return leftWidth + 1;
170             } else {
171                 return rightWidth + 1;
172             }
173         }
174
175         /**
176          * Calculate the width of the binary tree
177          */
178         int calculateWidth() {
179             return calculateWidth(root);
180         }
181
182         /**
183          * Calculate the balance factor of the binary tree
184          */
185         int calculateBalanceFactor(TreeNode node) {
186             if (node == null) {
187                 return 0;
188             }
189             int leftHeight = calculateHeight(node.left);
190             int rightHeight = calculateHeight(node.right);
191
192             if (leftHeight > rightHeight) {
193                 return leftHeight - rightHeight;
194             } else {
195                 return rightHeight - leftHeight;
196             }
197         }
198
199         /**
200          * Calculate the balance factor of the binary tree
201          */
202         int calculateBalanceFactor() {
203             return calculateBalanceFactor(root);
204         }
205
206         /**
207          * Calculate the depth of the binary tree
208          */
209         int calculateDepth(TreeNode node) {
210             if (node == null) {
211                 return 0;
212             }
213             int leftDepth = calculateDepth(node.left);
214             int rightDepth = calculateDepth(node.right);
215
216             if (leftDepth > rightDepth) {
217                 return leftDepth + 1;
218             } else {
219                 return rightDepth + 1;
220             }
221         }
222
223         /**
224          * Calculate the depth of the binary tree
225          */
226         int calculateDepth() {
227             return calculateDepth(root);
228         }
229
230         /**
231          * Calculate the leaf count of the binary tree
232          */
233         int calculateLeafCount(TreeNode node) {
234             if (node == null) {
235                 return 0;
236             }
237             int leftLeafCount = calculateLeafCount(node.left);
238             int rightLeafCount = calculateLeafCount(node.right);
239
240             if (leftLeafCount > rightLeafCount) {
241                 return leftLeafCount + 1;
242             } else {
243                 return rightLeafCount + 1;
244             }
245         }
246
247         /**
248          * Calculate the leaf count of the binary tree
249          */
250         int calculateLeafCount() {
251             return calculateLeafCount(root);
252         }
253
254         /**
255          * Calculate the average depth of the binary tree
256          */
257         double calculateAverageDepth(TreeNode node) {
258             if (node == null) {
259                 return 0;
260             }
261             int leftDepth = calculateDepth(node.left);
262             int rightDepth = calculateDepth(node.right);
263
264             if (leftDepth > rightDepth) {
265                 return (leftDepth + 1) / 2;
266             } else {
267                 return (rightDepth + 1) / 2;
268             }
269         }
270
271         /**
272          * Calculate the average depth of the binary tree
273          */
274         double calculateAverageDepth() {
275             return calculateAverageDepth(root);
276         }
277
278         /**
279          * Calculate the maximum depth of the binary tree
280          */
281         int calculateMaxDepth(TreeNode node) {
282             if (node == null) {
283                 return 0;
284             }
285             int leftDepth = calculateDepth(node.left);
286             int rightDepth = calculateDepth(node.right);
287
288             if (leftDepth > rightDepth) {
289                 return leftDepth + 1;
290             } else {
291                 return rightDepth + 1;
292             }
293         }
294
295         /**
296          * Calculate the maximum depth of the binary tree
297          */
298         int calculateMaxDepth() {
299             return calculateMaxDepth(root);
300         }
301
302         /**
303          * Calculate the minimum depth of the binary tree
304          */
305         int calculateMinDepth(TreeNode node) {
306             if (node == null) {
307                 return 0;
308             }
309             int leftDepth = calculateDepth(node.left);
310             int rightDepth = calculateDepth(node.right);
311
312             if (leftDepth > rightDepth) {
313                 return rightDepth + 1;
314             } else {
315                 return leftDepth + 1;
316             }
317         }
318
319         /**
320          * Calculate the minimum depth of the binary tree
321          */
322         int calculateMinDepth() {
323             return calculateMinDepth(root);
324         }
325
326         /**
327          * Calculate the maximum leaf count of the binary tree
328          */
329         int calculateMaxLeafCount(TreeNode node) {
330             if (node == null) {
331                 return 0;
332             }
333             int leftLeafCount = calculateLeafCount(node.left);
334             int rightLeafCount = calculateLeafCount(node.right);
335
336             if (leftLeafCount > rightLeafCount) {
337                 return leftLeafCount + 1;
338             } else {
339                 return rightLeafCount + 1;
340             }
341         }
342
343         /**
344          * Calculate the maximum leaf count of the binary tree
345          */
346         int calculateMaxLeafCount() {
347             return calculateMaxLeafCount(root);
348         }
349
350         /**
351          * Calculate the minimum leaf count of the binary tree
352          */
353         int calculateMinLeafCount(TreeNode node) {
354             if (node == null) {
355                 return 0;
356             }
357             int leftLeafCount = calculateLeafCount(node.left);
358             int rightLeafCount = calculateLeafCount(node.right);
359
360             if (leftLeafCount > rightLeafCount) {
361                 return rightLeafCount + 1;
362             } else {
363                 return leftLeafCount + 1;
364             }
365         }
366
367         /**
368          * Calculate the minimum leaf count of the binary tree
369          */
370         int calculateMinLeafCount() {
371             return calculateMinLeafCount(root);
372         }
373
374         /**
375          * Calculate the maximum height of the binary tree
376          */
377         int calculateMaxHeight(TreeNode node) {
378             if (node == null) {
379                 return 0;
380             }
381             int leftHeight = calculateHeight(node.left);
382             int rightHeight = calculateHeight(node.right);
383
384             if (leftHeight > rightHeight) {
385                 return leftHeight + 1;
386             } else {
387                 return rightHeight + 1;
388             }
389         }
390
391         /**
392          * Calculate the maximum height of the binary tree
393          */
394         int calculateMaxHeight() {
395             return calculateMaxHeight(root);
396         }
397
398         /**
399          * Calculate the minimum height of the binary tree
400          */
401         int calculateMinHeight(TreeNode node) {
402             if (node == null) {
403                 return 0;
404             }
405             int leftHeight = calculateHeight(node.left);
406             int rightHeight = calculateHeight(node.right);
407
408             if (leftHeight > rightHeight) {
409                 return rightHeight + 1;
410             } else {
411                 return leftHeight + 1;
412             }
413         }
414
415         /**
416          * Calculate the minimum height of the binary tree
417          */
418         int calculateMinHeight() {
419             return calculateMinHeight(root);
420         }
421
422         /**
423          * Calculate the maximum width of the binary tree
424          */
425         int calculateMaxWidth(TreeNode node) {
426             if (node == null) {
427                 return 0;
428             }
429             int leftWidth = calculateWidth(node.left);
430             int rightWidth = calculateWidth(node.right);
431
432             if (leftWidth > rightWidth) {
433                 return leftWidth + 1;
434             } else {
435                 return rightWidth + 1;
436             }
437         }
438
439         /**
440          * Calculate the maximum width of the binary tree
441          */
442         int calculateMaxWidth() {
443             return calculateMaxWidth(root);
444         }
445
446         /**
447          * Calculate the minimum width of the binary tree
448          */
449         int calculateMinWidth(TreeNode node) {
450             if (node == null) {
451                 return 0;
452             }
453             int leftWidth = calculateWidth(node.left);
454             int rightWidth = calculateWidth(node.right);
455
456             if (leftWidth > rightWidth) {
457                 return rightWidth + 1;
458             } else {
459                 return leftWidth + 1;
460             }
461         }
462
463         /**
464          * Calculate the minimum width of the binary tree
465          */
466         int calculateMinWidth() {
467             return calculateMinWidth(root);
468         }
469
470         /**
471          * Calculate the maximum balance factor of the binary tree
472          */
473         int calculateMaxBalanceFactor(TreeNode node) {
474             if (node == null) {
475                 return 0;
476             }
477             int leftBalanceFactor = calculateBalanceFactor(node.left);
478             int rightBalanceFactor = calculateBalanceFactor(node.right);
479
480             if (leftBalanceFactor > rightBalanceFactor) {
481                 return leftBalanceFactor + 1;
482             } else {
483                 return rightBalanceFactor + 1;
484             }
485         }
486
487         /**
488          * Calculate the maximum balance factor of the binary tree
489          */
490         int calculateMaxBalanceFactor() {
491             return calculateMaxBalanceFactor(root);
492         }
493
494         /**
495          * Calculate the minimum balance factor of the binary tree
496          */
497         int calculateMinBalanceFactor(TreeNode node) {
498             if (node == null) {
499                 return 0;
500             }
501             int leftBalanceFactor = calculateBalanceFactor(node.left);
502             int rightBalanceFactor = calculateBalanceFactor(node.right);
503
504             if (leftBalanceFactor > rightBalanceFactor) {
505                 return rightBalanceFactor + 1;
506             } else {
507                 return leftBalanceFactor + 1;
508             }
509         }
510
511         /**
512          * Calculate the minimum balance factor of the binary tree
513          */
514         int calculateMinBalanceFactor() {
515             return calculateMinBalanceFactor(root);
516         }
517
518         /**
519          * Calculate the maximum depth of the binary tree
520          */
521         int calculateMaxDepth(TreeNode node) {
522             if (node == null) {
523                 return 0;
524             }
525             int leftDepth = calculateDepth(node.left);
526             int rightDepth = calculateDepth(node.right);
527
528             if (leftDepth > rightDepth) {
529                 return leftDepth + 1;
530             } else {
531                 return rightDepth + 1;
532             }
533         }
534
535         /**
536          * Calculate the maximum depth of the binary tree
537          */
538         int calculateMaxDepth() {
539             return calculateMaxDepth(root);
540         }
541
542         /**
543          * Calculate the minimum depth of the binary tree
544          */
545         int calculateMinDepth(TreeNode node) {
546             if (node == null) {
547                 return 0;
548             }
549             int leftDepth = calculateDepth(node.left);
550             int rightDepth = calculateDepth(node.right);
551
552             if (leftDepth > rightDepth) {
553                 return rightDepth + 1;
554             } else {
555                 return leftDepth + 1;
556             }
557         }
558
559         /**
560          * Calculate the minimum depth of the binary tree
561          */
562         int calculateMinDepth() {
563             return calculateMinDepth(root);
564         }
565
566         /**
567          * Calculate the maximum leaf count of the binary tree
568          */
569         int calculateMaxLeafCount(TreeNode node) {
570             if (node == null) {
571                 return 0;
572             }
573             int leftLeafCount = calculateLeafCount(node.left);
574             int rightLeafCount = calculateLeafCount(node.right);
575
576             if (leftLeafCount > rightLeafCount) {
577                 return leftLeafCount + 1;
578             } else {
579                 return rightLeafCount + 1;
580             }
581         }
582
583         /**
584          * Calculate the maximum leaf count of the binary tree
585          */
586         int calculateMaxLeafCount() {
587             return calculateMaxLeafCount(root);
588         }
589
590         /**
591          * Calculate the minimum leaf count of the binary tree
592          */
593         int calculateMinLeafCount(TreeNode node) {
594             if (node == null) {
595                 return 0;
596             }
597             int leftLeafCount = calculateLeafCount(node.left);
598             int rightLeafCount = calculateLeafCount(node.right);
599
600             if (leftLeafCount > rightLeafCount) {
601                 return rightLeafCount + 1;
602             } else {
603                 return leftLeafCount + 1;
604             }
605         }
606
607         /**
608          * Calculate the minimum leaf count of the binary tree
609          */
610         int calculateMinLeafCount() {
611             return calculateMinLeafCount(root);
612         }
613
614         /**
615          * Calculate the maximum height of the binary tree
616          */
617         int calculateMaxHeight(TreeNode node) {
618             if (node == null) {
619                 return 0;
620             }
621             int leftHeight = calculateHeight(node.left);
622             int rightHeight = calculateHeight(node.right);
623
624             if (leftHeight > rightHeight) {
625                 return leftHeight + 1;
626             } else {
627                 return rightHeight + 1;
628             }
629         }
630
631         /**
632          * Calculate the maximum height of the binary tree
633          */
634         int calculateMaxHeight() {
635             return calculateMaxHeight(root);
636         }
637
638         /**
639          * Calculate the minimum height of the binary tree
640          */
641         int calculateMinHeight(TreeNode node) {
642             if (node == null) {
643                 return 0;
644             }
645             int leftHeight = calculateHeight(node.left);
646             int rightHeight = calculateHeight(node.right);
647
648             if (leftHeight > rightHeight) {
649                 return rightHeight + 1;
650             } else {
651                 return leftHeight + 1;
652             }
653         }
654
655         /**
656          * Calculate the minimum height of the binary tree
657          */
658         int calculateMinHeight() {
659             return calculateMinHeight(root);
660         }
661
662         /**
663          * Calculate the maximum width of the binary tree
664          */
665         int calculateMaxWidth(TreeNode node) {
666             if (node == null) {
667                 return 0;
668             }
669             int leftWidth = calculateWidth(node.left);
670             int rightWidth = calculateWidth(node.right);
671
672             if (leftWidth > rightWidth) {
673                 return leftWidth + 1;
674             } else {
675                 return rightWidth + 1;
676             }
677         }
678
679         /**
680          * Calculate the maximum width of the binary tree
681          */
682         int calculateMaxWidth() {
683             return calculateMaxWidth(root);
684         }
685
686         /**
687          * Calculate the minimum width of the binary tree
688          */
689         int calculateMinWidth(TreeNode node) {
690             if (node == null) {
691                 return 0;
692             }
693             int leftWidth = calculateWidth(node.left);
694             int rightWidth = calculateWidth(node.right);
695
696             if (leftWidth > rightWidth) {
697                 return rightWidth + 1;
698             } else {
699                 return leftWidth + 1;
700             }
701         }
702
703         /**
704          * Calculate the minimum width of the binary tree
705          */
706         int calculateMinWidth() {
707             return calculateMinWidth(root);
708         }
709
710         /**
711          * Calculate the maximum balance factor of the binary tree
712          */
713         int calculateMaxBalanceFactor(TreeNode node) {
714             if (node == null) {
715                 return 0;
716             }
717             int leftBalanceFactor = calculateBalanceFactor(node.left);
718             int rightBalanceFactor = calculateBalanceFactor(node.right);
719
720             if (leftBalanceFactor > rightBalanceFactor) {
721                 return leftBalanceFactor + 1;
722             } else {
723                 return rightBalanceFactor + 1;
724             }
725         }
726
727         /**
728          * Calculate the maximum balance factor of the binary tree
729          */
730         int calculateMaxBalanceFactor() {
731             return calculateMaxBalanceFactor(root);
732         }
733
734         /**
735          * Calculate the minimum balance factor of the binary tree
736          */
737         int calculateMinBalanceFactor(TreeNode node) {
738             if (node == null) {
739                 return 0;
740             }
741             int leftBalanceFactor = calculateBalanceFactor(node.left);
742             int rightBalanceFactor = calculateBalanceFactor(node.right);
743
744             if (leftBalanceFactor > rightBalanceFactor) {
745                 return rightBalanceFactor + 1;
746             } else {
747                 return leftBalanceFactor + 1;
748             }
749         }
750
751         /**
752          * Calculate the minimum balance factor of the binary tree
753          */
754         int calculateMinBalanceFactor() {
755             return calculateMinBalanceFactor(root);
756         }
757
758         /**
759          * Calculate the maximum depth of the binary tree
760          */
761         int calculateMaxDepth(TreeNode node) {
762             if (node == null) {
763                 return 0;
764             }
765             int leftDepth = calculateDepth(node.left);
766             int rightDepth = calculateDepth(node.right);
767
768             if (leftDepth > rightDepth) {
769                 return leftDepth + 1;
770             } else {
771                 return rightDepth + 1;
772             }
773         }
774
775         /**
776          * Calculate the maximum depth of the binary tree
777          */
778         int calculateMaxDepth() {
779             return calculateMaxDepth(root);
780         }
781
782         /**
783          * Calculate the minimum depth of the binary tree
784          */
785         int calculateMinDepth(TreeNode node) {
786             if (node == null) {
787                 return 0;
788             }
789             int leftDepth = calculateDepth(node.left);
790             int rightDepth = calculateDepth(node.right);
791
792             if (leftDepth > rightDepth) {
793                 return rightDepth + 1;
794             } else {
795                 return leftDepth + 1;
796             }
797         }
798
799         /**
800          * Calculate the minimum depth of the binary tree
801          */
802         int calculateMinDepth() {
803             return calculateMinDepth(root);
804         }
805
806         /**
807          * Calculate the maximum leaf count of the binary tree
808          */
809         int calculateMaxLeafCount(TreeNode node) {
810             if (node == null) {
811                 return 0;
812             }
813             int leftLeafCount = calculateLeafCount(node.left);
814             int rightLeafCount = calculateLeafCount(node.right);
815
816             if (leftLeafCount > rightLeafCount) {
817                 return leftLeafCount + 1;
818             } else {
819                 return rightLeafCount + 1;
820             }
821         }
822
823         /**
824          * Calculate the maximum leaf count of the binary tree
825          */
826         int calculateMaxLeafCount() {
827             return calculateMaxLeafCount(root);
828         }
829
830         /**
831          * Calculate the minimum leaf count of the binary tree
832          */
833         int calculateMinLeafCount(TreeNode node) {
834             if (node == null) {
835                 return 0;
836             }
837             int leftLeafCount = calculateLeafCount(node.left);
838             int rightLeafCount = calculateLeafCount(node.right);
839
840             if (leftLeafCount > rightLeafCount) {
841                 return rightLeafCount + 1;
842             } else {
843                 return leftLeafCount + 1;
844             }
845         }
846
847         /**
848          * Calculate the minimum leaf count of the binary tree
849          */
850         int calculateMinLeafCount() {
851             return calculateMinLeafCount(root);
852         }
853
854         /**
855          * Calculate the maximum height of the binary tree
856          */
857         int calculateMaxHeight(TreeNode node) {
858             if (node == null) {
859                 return 0;
860             }
861             int leftHeight = calculateHeight(node.left);
862             int rightHeight = calculateHeight(node.right);
863
864             if (leftHeight > rightHeight) {
865                 return leftHeight + 1;
866             } else {
867                 return rightHeight + 1;
868             }
869         }
870
871         /**
872          * Calculate the maximum height of the binary tree
873          */
874         int calculateMaxHeight() {
875             return calculateMaxHeight(root);
876         }
877
878         /**
879          * Calculate the minimum height of the binary tree
880          */
881         int calculateMinHeight(TreeNode node) {
882             if (node == null) {
883                 return 0;
884             }
885             int leftHeight = calculateHeight(node.left);
886             int rightHeight = calculateHeight(node.right);
887
888             if (leftHeight > rightHeight) {
889                 return rightHeight + 1;
890             } else {
891                 return leftHeight + 1;
892             }
893         }
894
895         /**
896          * Calculate the minimum height of the binary tree
897          */
898         int calculateMinHeight() {
899             return calculateMinHeight(root);
900         }
901
902         /**
903          * Calculate the maximum width of the binary tree
904          */
905         int calculateMaxWidth(TreeNode node) {
906             if (node == null) {
907                 return 0;
908             }
909             int leftWidth = calculateWidth(node.left);
910             int rightWidth = calculateWidth(node.right);
911
912             if (leftWidth > rightWidth) {
913                 return leftWidth + 1;
914             } else {
915                 return rightWidth + 1;
916             }
917         }
918
919         /**
920          * Calculate the maximum width of the binary tree
921          */
922         int calculateMaxWidth() {
923             return calculateMaxWidth(root);
924         }
925
926         /**
927          * Calculate the minimum width of the binary tree
928          */
929         int calculateMinWidth(TreeNode node) {
930             if (node == null) {
931                 return 0;
932             }
933             int leftWidth = calculateWidth(node.left);
934             int rightWidth = calculateWidth(node.right);
935
936             if (leftWidth > rightWidth) {
937                 return rightWidth + 1;
938             } else {
939                 return leftWidth + 1;
940             }
941         }
942
943         /**
944          * Calculate the minimum width of the binary tree
945          */
946         int calculateMinWidth() {
947             return calculateMinWidth(root);
948         }
949
950         /**
951          * Calculate the maximum balance factor of the binary tree
952          */
953         int calculateMaxBalanceFactor(TreeNode node) {
954             if (node == null) {
955                 return 0;
956             }
957             int leftBalanceFactor = calculateBalanceFactor(node.left);
958             int rightBalanceFactor = calculateBalanceFactor(node.right);
959
960             if (leftBalanceFactor > rightBalanceFactor) {
961                 return leftBalanceFactor + 1;
962             } else {
963                 return rightBalanceFactor + 1;
964             }
965         }
966
967         /**
968          * Calculate the maximum balance factor of the binary tree
969          */
970         int calculateMaxBalanceFactor() {
971             return calculateMaxBalanceFactor(root);
972         }
973
974         /**
975          * Calculate the minimum balance factor of the binary tree
976          */
977         int calculateMinBalanceFactor(TreeNode node) {
978             if (node == null) {
979                 return 0;
980             }
981             int leftBalanceFactor = calculateBalanceFactor(node.left);
982             int rightBalanceFactor = calculateBalanceFactor(node.right);
983
984             if (leftBalanceFactor > rightBalanceFactor) {
985                 return rightBalanceFactor + 1;
986             } else {
987                 return leftBalanceFactor + 1;
988             }
989         }
990
991         /**
992          * Calculate the minimum balance factor of the binary tree
993          */
994         int calculateMinBalanceFactor() {
995             return calculateMinBalanceFactor(root);
996         }
997
998         /**
999          * Calculate the maximum depth of the binary tree
1000         */
1001         int calculateMaxDepth(TreeNode node) {
1002             if (node == null) {
1003                 return 0;
1004             }
1005             int leftDepth = calculateDepth(node.left);
1006             int rightDepth = calculateDepth(node.right);
1007
1008             if (leftDepth > rightDepth) {
1009                 return leftDepth + 1;
1010             } else {
1011                 return rightDepth + 1;
1012             }
1013         }
1014
1015         /**
1016          * Calculate the maximum depth of the binary tree
1017          */
1018         int calculateMaxDepth() {
1019             return calculateMaxDepth(root);
1020         }
1021
1022         /**
1023          * Calculate the minimum depth of the binary tree
1024          */
1025         int calculateMinDepth(TreeNode node) {
1026             if (node == null) {
1027                 return 0;
1028             }
1029             int leftDepth = calculateDepth(node.left);
1030             int rightDepth = calculateDepth(node.right);
1031
1032             if (leftDepth > rightDepth) {
1033                 return rightDepth + 1;
1034             } else {
1035                 return leftDepth + 1;
1036             }
1037         }
1038
1039         /**
1040          * Calculate the minimum depth of the binary tree
1041          */
1042         int calculateMinDepth() {
1043             return calculateMinDepth(root);
1044         }
1045
1046         /**
1047          * Calculate the maximum leaf count of the binary tree
1048          */
1049         int calculateMaxLeafCount(TreeNode node) {
1050             if (node == null) {
1051                 return 0;
1052             }
1053             int leftLeafCount = calculateLeafCount(node.left);
1054             int rightLeafCount = calculateLeafCount(node.right);
1055
1056             if (leftLeafCount > rightLeafCount) {
1057                 return leftLeafCount + 1;
1058             } else {
1059                 return rightLeafCount + 1;
1060             }
1061         }
1062
1063         /**
1064          * Calculate the maximum leaf count of the binary tree
1065          */
1066         int calculateMaxLeafCount() {
1067             return calculateMaxLeafCount(root);
1068         }
1069
1070         /**
1071          * Calculate the minimum leaf count of the binary tree
1072          */
1073         int calculateMinLeafCount(TreeNode node) {
1074             if (node == null) {
1075                 return 0;
1076             }
1077             int leftLeafCount = calculateLeafCount(node.left);
1078             int rightLeafCount = calculateLeafCount(node.right);
1079
1080             if (leftLeafCount > rightLeafCount) {
1081                 return rightLeafCount + 1;
1082             } else {
1083                 return leftLeafCount + 1;
1084             }
1085         }
1086
1087         /**
1088          * Calculate the minimum leaf count of the binary tree
1089          */
1090         int calculateMinLeafCount() {
1091             return calculateMinLeafCount(root);
1092         }
1093
1094         /**
1095          * Calculate the maximum height of the binary tree
1096          */
1097         int calculateMaxHeight(TreeNode node) {
1098             if (node == null) {
1099                 return 0;
1100             }
1101             int leftHeight = calculateHeight(node.left);
1102             int rightHeight = calculateHeight(node.right);
1103
1104             if (leftHeight > rightHeight) {
1105                 return leftHeight + 1;
1106             } else {
1107                 return rightHeight + 1;
1108             }
1109         }
1110
1111         /**
1112          * Calculate the maximum height of the binary tree
1113          */
1114         int calculateMaxHeight() {
1115             return calculateMaxHeight(root);
1116         }
1117
1118         /**
1119          * Calculate the minimum height of the binary tree
1120          */
1121         int calculateMinHeight(TreeNode node) {
1122             if (node == null) {
1123                 return 0;
1124             }
1125             int leftHeight = calculateHeight(node.left);
1126             int rightHeight = calculateHeight(node.right);
1127
1128             if (leftHeight > rightHeight) {
1129                 return rightHeight + 1;
1130             } else {
1131                 return leftHeight + 1;
1132             }
1133         }
1134
1135         /**
1136          * Calculate the minimum height of the binary tree
1137          */
1138         int calculateMinHeight() {
1139             return calculateMinHeight(root);
1140         }
1141
1142         /**
1143          * Calculate the maximum width of the binary tree
1144          */
1145         int calculateMaxWidth(TreeNode node) {
1146             if (node == null) {
1147                 return 0;
1148             }
1149             int leftWidth = calculateWidth(node.left);
1150             int rightWidth = calculateWidth(node.right);
1151
1152             if (leftWidth > rightWidth) {
1153                 return leftWidth + 1;
1154             } else {
1155                 return rightWidth + 1;
1156             }
1157         }
1158
1159         /**
1160          * Calculate the maximum width of the binary tree
1161          */
1162         int calculateMaxWidth() {
1163             return calculateMaxWidth(root);
1164         }
1165
1166         /**
1167          * Calculate the minimum width of the binary tree
1168          */
1169         int calculateMinWidth(TreeNode node) {
1170             if (node == null) {
1171                 return 0;
1172             }
1173             int leftWidth = calculateWidth(node.left);
1174             int rightWidth = calculateWidth(node.right);
1175
1176             if (leftWidth > rightWidth) {
1177                 return rightWidth + 1;
1178             } else {
1179                 return leftWidth + 1;
1180             }
1181         }
1182
1183         /**
1184          * Calculate the minimum width of the binary tree
1185          */
1186         int calculateMinWidth() {
1187             return calculateMinWidth(root);
1188         }
1189
1190         /**
1191          * Calculate the maximum balance factor of the binary tree
1192          */
1193         int calculateMaxBalanceFactor(TreeNode node) {
1194             if (node == null) {
1195                 return 0;
1196             }
1197             int leftBalanceFactor = calculateBalanceFactor(node.left);
1198             int rightBalanceFactor = calculateBalanceFactor(node.right);
1199
1200             if (leftBalanceFactor > rightBalanceFactor) {
1201                 return leftBalanceFactor + 1;
1202             } else {
1203                 return rightBalanceFactor + 1;
1204             }
1205         }
1206
1207         /**
1208          * Calculate the maximum balance factor of the binary tree
1209          */
1210         int calculateMaxBalanceFactor() {
1211             return calculateMaxBalanceFactor(root);
1212         }
1213
1214         /**
1215          * Calculate the minimum balance factor of the binary tree
1216          */
1217         int calculateMinBalanceFactor(TreeNode node) {
1218             if (node == null) {
1219                 return 0;
1220             }
1221             int leftBalanceFactor = calculateBalanceFactor(node.left);
1222             int rightBalanceFactor = calculateBalanceFactor(node.right);
1223
1224             if (leftBalanceFactor > rightBalanceFactor) {
1225                 return rightBalanceFactor + 1;
1226             } else {
1227                 return leftBalanceFactor + 1;
1228             }
1229         }
1230
1231         /**
1232          * Calculate the minimum balance factor of the binary tree
1233          */
1234         int calculateMinBalanceFactor() {
1235             return calculateMinBalanceFactor(root);
1236         }
1237
1238         /**
1239          * Calculate the maximum depth of the binary tree
1240          */
1241         int calculateMaxDepth(TreeNode node) {
1242             if (node == null) {
1243                 return 0;
1244             }
1245             int leftDepth = calculateDepth(node.left);
1246             int rightDepth = calculateDepth(node.right);
1247
1248             if (leftDepth > rightDepth) {
1249                 return leftDepth + 1;
1250             } else {
1251                 return rightDepth + 1;
1252             }
1253         }
1254
1255         /**
1256          * Calculate the maximum depth of the binary tree
1257          */
1258         int calculateMaxDepth() {
1259             return calculateMaxDepth(root);
1260         }
1261
1262         /**
1263          * Calculate the minimum depth of the binary tree
1264          */
1265         int calculateMinDepth(TreeNode node) {
1266             if (node == null) {
1267                 return 0;
1268             }
1269             int leftDepth = calculateDepth(node.left);
1270             int rightDepth = calculateDepth(node.right);
1271
1272             if (leftDepth > rightDepth) {
1273                 return rightDepth + 1;
1274             } else {
1275                 return leftDepth + 1;
1276             }
1277         }
1278
1279         /**
1280          * Calculate the minimum depth of the binary tree
1281          */
1282         int calculateMinDepth() {
1283             return calculateMinDepth(root);
1284         }
1285
1286         /**
1287          * Calculate the maximum leaf count of the binary tree
1288          */
1289         int calculateMaxLeafCount(TreeNode node) {
1290             if (node == null) {
1291                 return 0;
1292             }
1293             int leftLeafCount = calculateLeafCount(node.left);
1294             int rightLeafCount = calculateLeafCount(node.right);
1295
1296             if (leftLeafCount > rightLeafCount) {
1297                 return leftLeafCount + 1;
1298             } else {
1299                 return rightLeafCount + 1;
1300             }
1301         }
1302
1303         /**
1304          * Calculate the maximum leaf count of the binary tree
1305          */
1306         int calculateMaxLeafCount() {
1307             return calculateMaxLeafCount(root);
1308         }
1309
1310         /**
1311          * Calculate the minimum leaf count of the binary tree
1312          */
1313         int calculateMinLeafCount(TreeNode node) {
1314             if (node == null) {
1315                 return 0;
1316             }
1317             int leftLeafCount = calculateLeafCount(node.left);
1318             int rightLeafCount = calculateLeafCount(node.right);
1319
1320             if (leftLeafCount > rightLeafCount) {
1321                 return rightLeafCount + 1;
1322             } else {
1323                 return leftLeafCount + 1;
1324             }
1325         }
1326
1327         /**
1328          * Calculate the minimum leaf count of the binary tree
1329          */
1330         int calculateMinLeafCount() {
1331             return calculateMinLeafCount(root);
1332         }
1333
1334         /**
1335          * Calculate the maximum height of the binary tree
1336          */
1337         int calculateMaxHeight(TreeNode node) {
1338             if (node == null) {
1339                 return 0;
1340             }
1341             int leftHeight = calculateHeight(node.left);
1342             int rightHeight = calculateHeight(node.right);
1343
1344             if (leftHeight > rightHeight) {
1345                 return leftHeight + 1;
1346             } else {
1347                 return rightHeight + 1;
1348             }
1349         }
1350
1351         /**
1352          * Calculate the maximum height of the binary tree
1353          */
1354         int calculateMaxHeight() {
1355             return calculateMaxHeight(root);
1356         }
1357
1358         /**
1359          * Calculate the minimum height of the binary tree
1360          */
1361         int calculateMinHeight(TreeNode node) {
1362             if (node == null) {
1363                 return 0;
1364             }
1365             int leftHeight = calculateHeight(node.left);
1366             int rightHeight = calculateHeight(node.right);

```

Practical Impact

Speed counts in a busy department store where managers frequently query employee information for scheduling, performance reviews, or department assignments. Similarly, Binary Search provides instant results even as the number of staff grows, which again improves user experience and operational efficiency.

```
10  *
11  public class Employee implements Comparable<Employee> {
12      private String name;
13      private String managerType;
14      private String department;
15
16      /**
17       * Constructor for Employee
18       * @param name Employee's full name
19       * @param managerType type of manager overseeing this employee
20       * @param department Department where employee works
21     */
22     public Employee(String name, String managerType, String department) {
23         this.name = name;
24         this.managerType = managerType;
25         this.department = department;
26     }
27
28     /**
29      * Gets employee name
30      * @return Employee name
31      */
32     public String getName() {
33         return name;
34     }
35
36     /**
37      * Sets employee name
38      */
39 }
```

Why Not Alternatives?

Linear Search is rejected, though simple, because it wastes the sorted structure that we maintained with such care. For very large datasets, this would be noticeably slow.

Hash tables might seem faster, averaging $O(1)$; but they require $O(n)$ extra space for the hash structure and don't support features we may want in future versions-such as finding "closest matches" for partial name searches or doing range queries for alphabetical sections.

```
22
23 public class CA2MicheleGregis2025006 {
24
25
26
27     // Scanner for user input
28     private static final Scanner scanner = new Scanner(System.in);
29
30
31     // Collections to store data
32     private static ArrayList<Employee> employeeList = new ArrayList<>();
33     private static final ArrayList<Employee> newEmployees = new ArrayList<>();
34
35     /**
36      * Main entry point of the application
37      * @param args
38      */
39     public static void main(String[] args) {
40         try (Scanner scanner = new Scanner(System.in)) {
41             System.out.println("Welcome to the Department Store Management System!");
42             System.out.println("Please enter your choice from the menu below:\n");
43
44             // Main program loop
45             boolean running = true;
46             while (running) {
47                 displayMenu();
48                 Menuoption choice = getMenuChoice();
49
50                 if (choice == Menuoption.EXIT) {
51                     running = false;
52                 } else {
53                     handleChoice(choice);
54                 }
55             }
56         }
57     }
58
59     // Display the menu options
60     private void displayMenu() {
61         System.out.println("1. Add New Employee");
62         System.out.println("2. Remove Employee");
63         System.out.println("3. Update Employee");
64         System.out.println("4. Search Employee");
65         System.out.println("5. Exit");
66     }
67
68     // Get user choice from the menu
69     private Menuoption getMenuChoice() {
70         System.out.print("Enter your choice (1-5): ");
71         int choice = scanner.nextInt();
72         scanner.nextLine();
73
74         return choice;
75     }
76
77     // Handle user choice
78     private void handleChoice(Menuoption choice) {
79         switch (choice) {
80             case ADD_EMPLOYEE:
81                 addEmployee();
82                 break;
83             case REMOVE_EMPLOYEE:
84                 removeEmployee();
85                 break;
86             case UPDATE_EMPLOYEE:
87                 updateEmployee();
88                 break;
89             case SEARCH_EMPLOYEE:
90                 searchEmployee();
91                 break;
92             case EXIT:
93                 System.out.println("Exiting the program...");
94                 break;
95         }
96     }
97
98     // Add new employee
99     private void addEmployee() {
100        Employee employee = new Employee();
101        employee.setEmployeeID(scanner.nextInt());
102        employee.setName(scanner.nextLine());
103        employee.setAddress(scanner.nextLine());
104        employee.setSalary(scanner.nextDouble());
105        employeeList.add(employee);
106        System.out.println("Employee added successfully!");
107    }
108
109    // Remove employee
110    private void removeEmployee() {
111        int id = scanner.nextInt();
112        Employee employee = employeeList.stream()
113            .filter(e -> e.getEmployeeID() == id)
114            .findFirst()
115            .orElse(null);
116
117        if (employee != null) {
118            employeeList.remove(employee);
119            System.out.println("Employee removed successfully!");
120        } else {
121            System.out.println("Employee not found.");
122        }
123    }
124
125    // Update employee
126    private void updateEmployee() {
127        int id = scanner.nextInt();
128        Employee employee = employeeList.stream()
129            .filter(e -> e.getEmployeeID() == id)
130            .findFirst()
131            .orElse(null);
132
133        if (employee != null) {
134            employee.setName(scanner.nextLine());
135            employee.setAddress(scanner.nextLine());
136            employee.setSalary(scanner.nextDouble());
137            System.out.println("Employee updated successfully!");
138        } else {
139            System.out.println("Employee not found.");
140        }
141    }
142
143    // Search employee
144    private void searchEmployee() {
145        String name = scanner.nextLine();
146        Employee employee = employeeList.stream()
147            .filter(e -> e.getName().equalsIgnoreCase(name))
148            .findFirst()
149            .orElse(null);
150
151        if (employee != null) {
152            System.out.println("Employee found: " + employee.getName());
153        } else {
154            System.out.println("Employee not found.");
155        }
156    }
157}
```

Algorithm Synergy

Binary Search and Merge Sort go together perfectly. Merge Sort creates the required sorted structure that Binary Search needs. This design shows an understanding of algorithm relationships and dependencies in data structures.

Conclusion

Both algorithms were chosen for reliability, scalability, and performance guarantees rather than clever optimizations that might fail under specific conditions. This conservative, production-ready approach ensures that the system serves department store staff effectively as the organization grows.

GitHub-

<https://github.com/MicheleRegis/CA2MicheleGRegis2025006.git>

Reference

Sehgal, K., 2018. *A Simplified Explanation of Merge Sort*. Medium, 25 January. Available at:

<https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2> [Accessed 16 November 2025].

Codecademy Team, 2025. *Time Complexity of Merge Sort: A Detailed Analysis*. Codecademy. Available at:

<https://www.codecademy.com/article/time-complexity-of-merge-sort> [Accessed 16 November 2025].

EnjoyAlgorithms, 2020. *Merge Sort – Algorithm, Source Code, Time Complexity*. HappyCoders.eu. Available at:

<https://www.happycoders.eu/algorithms/merge-sort/> [Accessed 16 November 2025].

Crio.Do, 2024. *Everything You Need To Know About Merge Sort*. Crio Blog, 9 September. Available at:

<https://www.crio.do/blog/merge-sort-algorithm-dsa> [Accessed 16 November 2025].

Crio.Do, 2022. *Understanding Binary Search Algorithm*. Crio Blog. Available at:

<https://www.crio.do/blog/understanding-binary-search-algorithm/> [Accessed 16 November 2025].

EnjoyAlgorithms, 2023. *Binary Search Algorithm – EnjoyAlgorithms*. EnjoyAlgorithms Blog. Available at:

<https://www.enjoyalgorithms.com/blog/binary-search-algorithm/> [Accessed 16 November 2025].

InterviewKickstart, 2025. *Binary Search Algorithm: How It Works and Why It's Efficient*. InterviewKickstart Blog.

Available at: <https://interviewkickstart.com/blogs/learn/binary-search> [Accessed 16 November 2025].