

Problem 1.1 (createListIndex function) is located in the LDictionnary class, it is the second to last method. It uses the methods created in the sort package (mergeSort).

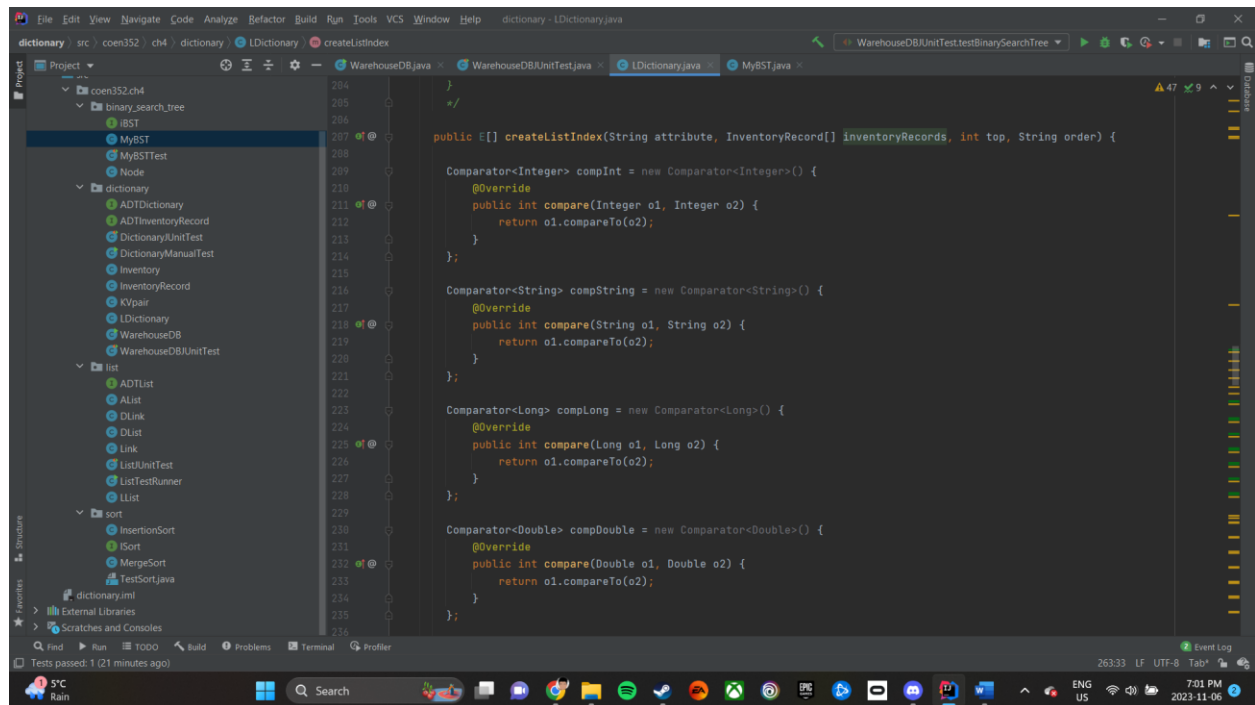
Problem 1.2 (createTreeIndex function) is also located in the LDictionnary class, it is the last method implemented in that class. It is based on the implementation of BST created in the binary\_search\_tree package.

Problem 2 (query) is located at the bottom of the WarehouseDB class.

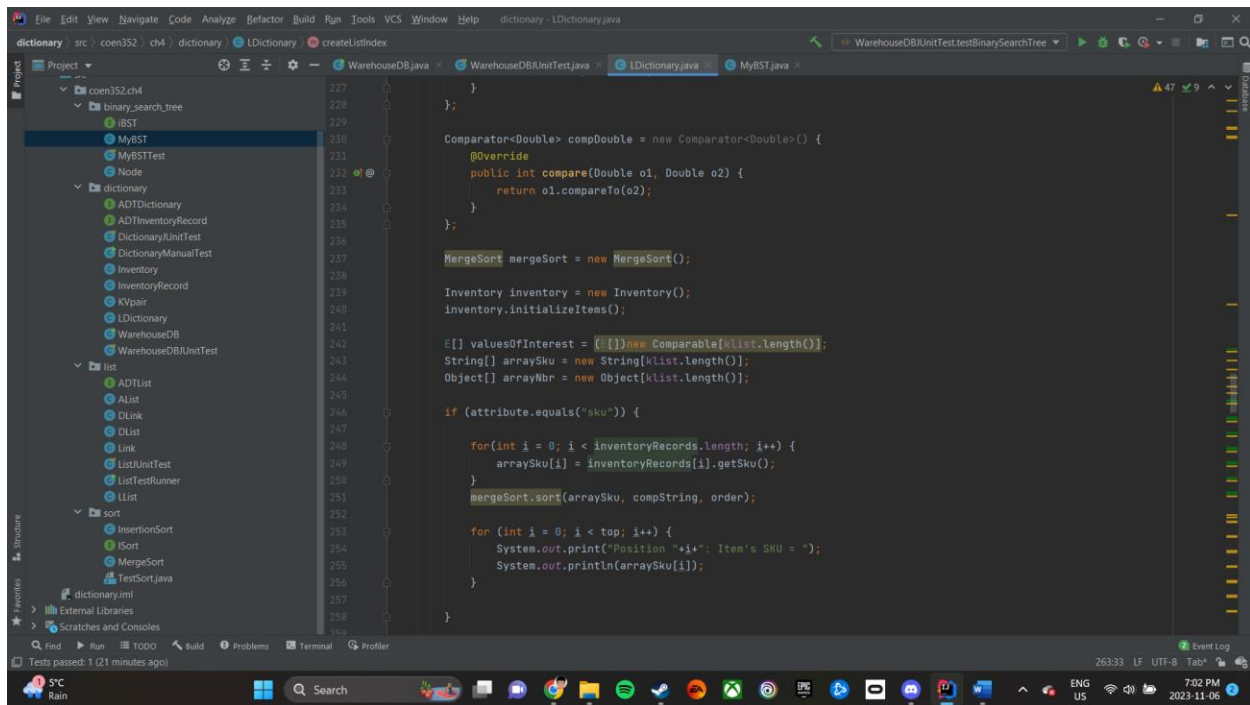
Problem 3 (both test cases for query) is located at the bottom of the WarehouseDBJUnitTest class.

I will now be explaining the asymptotic analysis for problems 1.1 and 1.2.

### Problem 1.1



The time complexity for these lines of code is a simple  $O(1)$ , they will always initialize these comparators at the same time (so  $O(1)$  for this part).



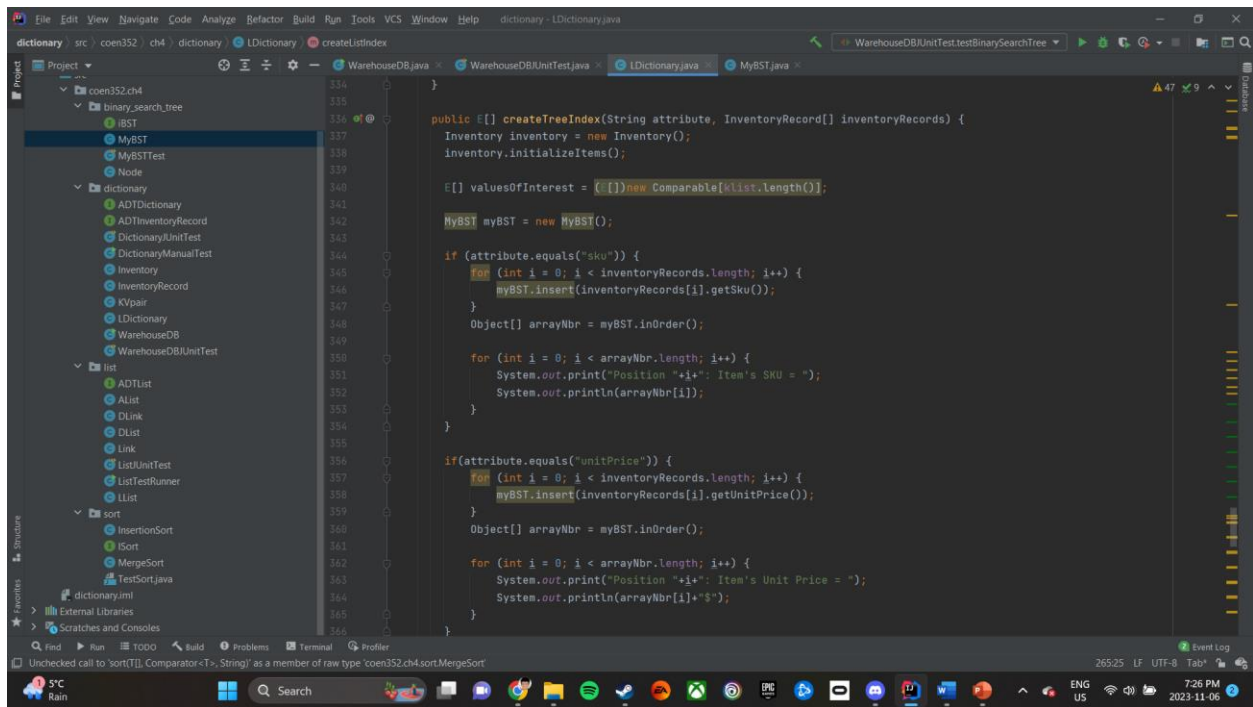
```
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```

The next lines of code initialize a bunch of variables ( $O(1)$ ), merge sorts ( $O(n \log n)$ ) the array of sku (or any chosen attribute, in this case we have chosen the sku) and uses 2 for loops that depend on the top integer: one to put all of the sku's in one array and the other to display the sorted result ( $O(\text{top})$  and  $O(\text{top})$ ).

It is unnecessary to show the next parts of the code, since they are all variations of the 2 for loops + mergeSort, except they have different attributes that need to be inserted (inventory value, quantity, unit price etc.). Only one of these variations will be used because we only call a single attribute, meaning that the time complexity for all of the variations is equal to the time complexity of a single variation.

The final time complexity is equal to the time complexity of merge sort because it is by far the most complicated and time-consuming part of the algorithm, rendering the initialization of the variables and the for loop's time complexity as unnecessary to the final calculation. Thus, the time complexity for my implementation of the createListIndex function is  $O(n \log n)$ .

## Problem 1.2



```
public E[] createTreeIndex(String attribute, InventoryRecord[] inventoryRecords) {
    Inventory inventory = new Inventory();
    inventory.initializeItems();

    E[] valuesOfInterest = {};

    MyBST myBST = new MyBST();

    if (attribute.equals("sku")) {
        for (int i = 0; i < inventoryRecords.length; i++) {
            myBST.insert(inventoryRecords[i].getSku());
        }
        Object[] arrayNbr = myBST.inOrder();

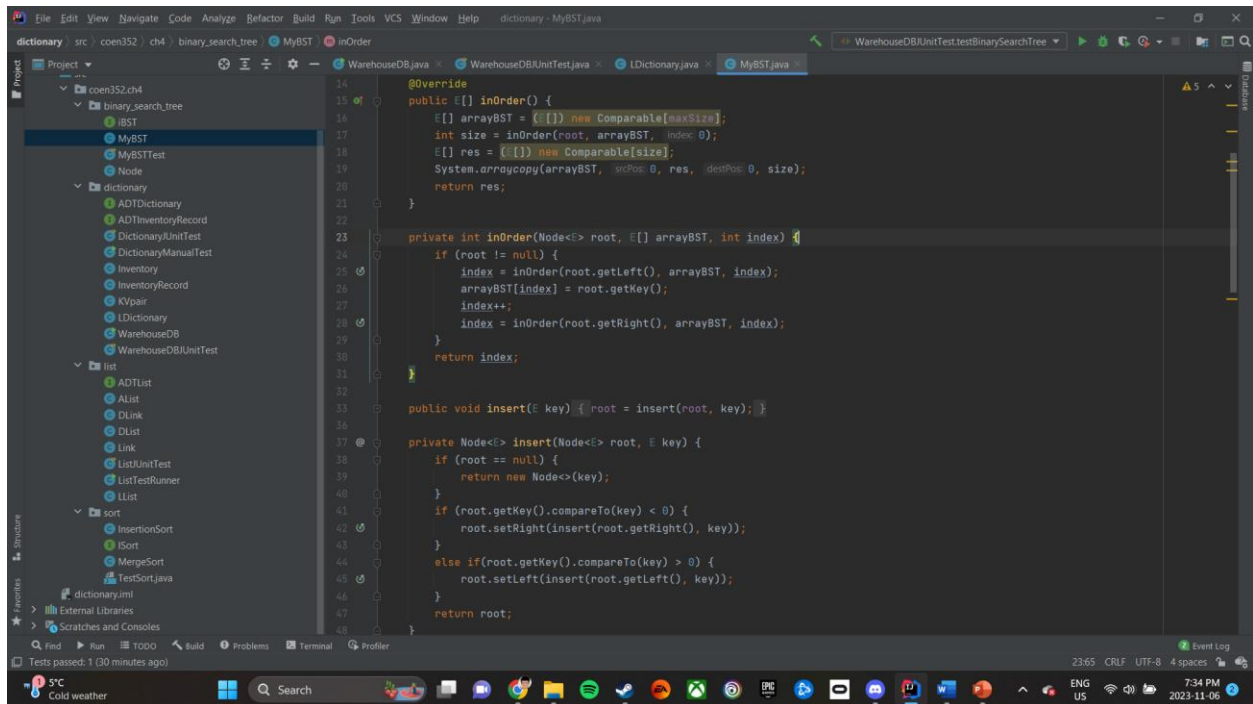
        for (int i = 0; i < arrayNbr.length; i++) {
            System.out.print("Position " + i + ": Item's SKU = ");
            System.out.println(arrayNbr[i]);
        }
    }

    if (attribute.equals("unitPrice")) {
        for (int i = 0; i < inventoryRecords.length; i++) {
            myBST.insert(inventoryRecords[i].getUnitPrice());
        }
        Object[] arrayNbr = myBST.inOrder();

        for (int i = 0; i < arrayNbr.length; i++) {
            System.out.print("Position " + i + ": Item's Unit Price = ");
            System.out.println(arrayNbr[i] + "$");
        }
    }
}
```

All of the values initialized are  $O(1)$ .

The primary time-consuming parts of this code are the 2 for loops for each attribute case ( $O(n)$  and  $O(n)$ ) along with the `myBST.inOrder()` and `myBST.insert()` functions. ( $n$  = length of array).



```
@Override
public E[] inOrder() {
    E[] arrayBST = (E[]) new Comparable[maxSize];
    int size = inOrder(root, arrayBST, index);
    E[] res = (E[]) new Comparable[size];
    System.arraycopy(arrayBST, index, res, 0, size);
    return res;
}

private int inOrder(Node<E> root, E[] arrayBST, int index) {
    if (root != null) {
        index = inOrder(root.getLeft(), arrayBST, index);
        arrayBST[index] = root.getKey();
        index++;
        index = inOrder(root.getRight(), arrayBST, index);
    }
    return index;
}

public void insert(E key) { root = insert(root, key); }

private Node<E> insert(Node<E> root, E key) {
    if (root == null) {
        return new Node<E>(key);
    }
    if (root.getKey().compareTo(key) < 0) {
        root.setRight(insert(root.getRight(), key));
    }
    else if (root.getKey().compareTo(key) > 0) {
        root.setLeft(insert(root.getLeft(), key));
    }
    return root;
}
```

The previous screenshot presents those two functions.

We have learned in class that the insert function has a time complexity of  $O(n)$ , with  $n$  being the depth of the tree. The `myBST.inOrder()` function is also  $O(n)$  since it traverses the whole tree. The overall time complexity of both these functions is  $O(n)$ .

In the first for loop, we use `myBST.insert()` with every incrementation of  $i$ . This means that the time complexity of that segment is  $O(n*n)$  which is  $O(n^2)$ .

Since the most complicated segment has a time complexity of  $O(n^2)$ , we can conclude the problem by saying that the time complexity of my `createTreeIndex` function is  $O(n^2)$ .