The three basic operations on binary search trees are *finding*, *inserting*, and *removing* a node with a specified key. As we know, the time required for these operations is proportional to the height of the tree. The purpose of this assignment is to compare the heights of non-rebalancing BSTs, AVL trees, and AVLX trees. An AVLX tree is a new experimental data structure that is similar to an AVL tree but enforces a weaker balance property resulting fewer restructuring operations. Homework 3a deals with BSTs and AVL trees. AVLX trees enter the picture in 3b.

The worst-case heights of non-rebalancing BSTs and AVL trees are dramatically different: linear in the number of nodes for non-rebalancing BSTs and logarithmic for AVL trees. For randomly generated keys there is no difference asymptotically: it can be shown that the expected height of a randomly built BST or AVL tree is $O(\log n)$. It should be noted, however, that in real-world applications the data to be stored in a search tree may be far from random. In this project, we will investigate the heights of trees constructed with *partially sorted* sequences.

We can define the extent to which a sequence is partially sorted in terms of the well-known *Kendall tau distance*, also called *bubble-sort distance*. Let $X = (1, 2, 3, …, n)$ and suppose that Y is a permutation of X. The bubble-sort distance between X and Y is the number of swaps of adjacent elements that the bubble-sort algorithm would make when transforming Y to X (that is, sorting Y). Equivalent definition: it is the number of *inversions* in Y. An inversion is simply a pair of elements that are out of order. We will say that a sequence is k-sorted if it has k inversions. It is easy to compute the number of inversions in a sequence: iterate over all pairs of indices and increment a counter whenever $i < j$ and $Y(i) > Y(j)$.

It is also easy to randomly generate *partially sorted* sequences—that is, sequences having a particular (relatively small) number k of inversions. Start with $(1, 2, …, n)$ and choose a pair of consecutive indices; that is, randomly choose i from $\{1, 2, …, n\text{-}1\}$ and compare the elements at indices i and i +1. If they are in order, swap them. Repeat until you have performed k swaps.

You are given the header files for two template classes, BST and AvlTree. BST represents a non-rebalancing binary search tree and AvlTree is a subclass of BST. It overrides the *insert* method to first perform an ordinary BST insertion and then perform trinode restructuring if the resulting tree is unbalanced. Your job is to implement these classes. You may add, remove, or modify private or protected members of either class, but **the public interface for both classes must not be changed. Also, do not change struct Node**. This is necessary for my test programs to compile with your implementations.

Important note: for this assignment, you are free to use any code that you may find from other sources provided only that you clearly acknowledge them in your documentation. If you end up merely tweaking code that you obtained from other sources, that is perfectly fine. The purpose of this assignment is not to learn how to implement non-rebalancing BSTs or AVL trees, it is to set the stage for the next assignment, and therefore it will be much easier than if you were expected to implement the two classes on your own. It should be easy to find working code for non-rebalancing BSTs and AVL trees. If you find a particularly good source for such code, let me know and I will share a link with the class. I would use ChatGPT only as a last resort (or for debugging help) since it can be unreliable.

The most substantial part of this assignment by far is the implementation of *AvlTree::trinode_restructure*. My own solution includes ASCII tree art in the documentation to illustrate the four cases to be handled. These diagrams make the code easy to understand and they helped me to organize my own thoughts before starting to code. You will find the diagrams in *notes.txt*. That file also directs your attention to a few other implementation details.

You are given a test program (*tree_test.cpp*) that constructs and outputs a tree of each type built with a fixed set of 12 keys. If the program produces correct output using your BST and AvlTree implementations then the basic logic is probably correct (although in this case the input is too small to reveal memory leaks). I will also test your code more rigorously with a program that creates a large AVL tree with random keys and then traverses it to verify that each node is balanced (i.e., its children differ in height by at most 1). I recommend that you test your code in the same way. It sometimes happens that a buggy implementation produces correct results for a small number of keys, so a successful run of *tree_test.cpp* by itself is not sufficient for ensuring correctness.

Write a program called *main.cpp* that compares the expected heights of trees of both types built with partially sorted inputs for a range of sizes (2^{10}, 2^{11}, 2^{12}, ..., 2^{20}). The inputs are random permutations of (1, 2, 3, …, *n*) having a particular number of inversions (see below). The expected height is the average height over 100 independent trials.

The program's output should have the following form:

```
2^10 keys:
2^11 keys:
2^12 keys:
 ⋮
2^19 keys:
2^20 keys:
```

In your program's actual output, the text on each line shown above will be followed by four numbers:

- the expected height of a non-rebalancing BST with *n* inversions, where *n* is the number of keys.
- the expected height of an AVL tree constructed with the same sequence of keys.
- the expected height of a non-rebalancing BST with 100*n* inversions, where *n* is the number of keys.
- the expected height of an AVL tree constructed with the same sequence of keys.

All columns of numbers should line up neatly.

What to submit: *bst.cpp*, *avl_tree.cpp*, *main.cpp*