DMA1 Hand-in 7: The principle behind a Binary Search Tree solution to the **inversion** task

Let's assume we need to count the number of inversions in this sequence {2, 14, 6, 4, 15, 3, 7, 9}

From which we can identify 11 inversions, being:

{14,6}, {14,4}, {14,3}, {14,7}, {14,9}, {6,4}, {6,3}, {4,3}, {15,3}, {15,7}, {15,9}

An inversion is defined as
If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A

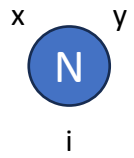
Using the principle behind Binary Search Trees, we can utilize an unbalanced binary search tree as a means of excluding parts of the tree and saving massive amounts of time doing CPU calculations while executing other more linear (or slower) implementations.

The first number in the sequence becomes the root value around which all other sequence numbers are evaluated against.

On the following pages I will illustrate how this process works:

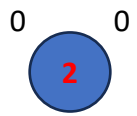
Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 0:

Evaluating index pos. 0: {2, 14, 6, 4, 15, 3, 7, 9}

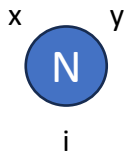


Step explanation:
1. Insert {2}, which becomes the root node for the tree.

No inversions yet.

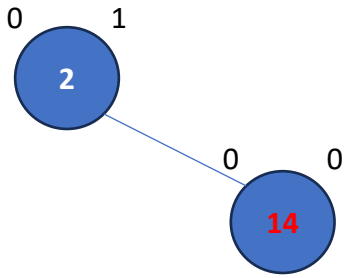
Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 1:

Evaluating index pos. 1: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

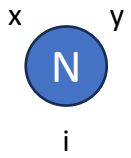
1. Insert {14}. Since it **IS NOT SMALLER** than the root, we traverse to the right.

2. {2} has no right child, so {14} becomes the right child.

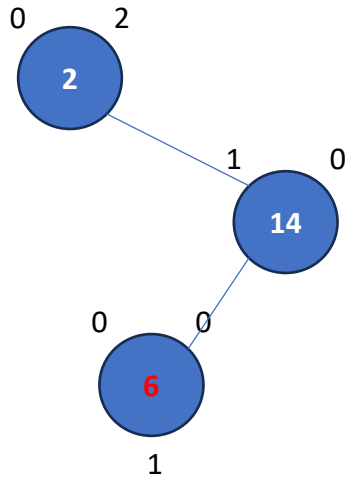
No inversions yet.

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 2:
Evaluating index pos. 2: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

1. Insert {6}. Since it **IS NOT SMALLER** than the root, we traverse to the right.

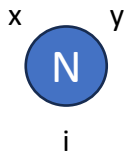
2. At node {14} we identify that {6} **is smaller** than {14}. Since {14} already exists in the tree, we thus conclude we now have an inversion.

3. We insert {6} as the left child to {14}, and note the inversion.

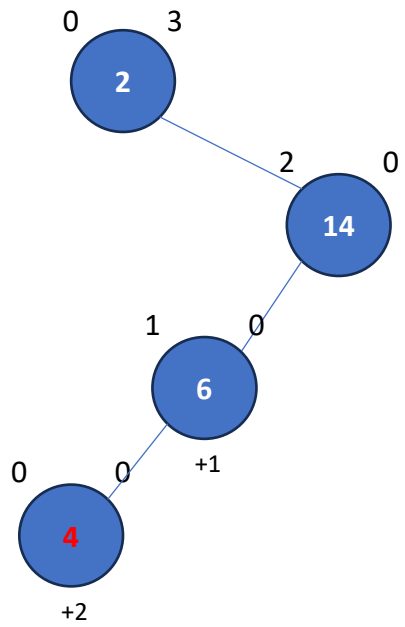
Inversions identified so far:
{6, 14}

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 3:
Evaluating index pos. 3: {2, 14, 6, **4**, 15, 3, 7, 9}



Step explanation:

1. Insert {4}. Since it **IS NOT SMALLER** than the root, we traverse to the right.

2. At node {14} we identify that {4} **is smaller** than {14}. Since {14} already exists in the tree, we thus conclude we now have an inversion.

Inversions +1

3. We traverse left. We see that {4} is also smaller than {6}. Thus we have another inversions.

Inversions +2

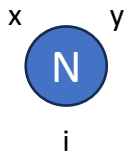
4. We insert {4} as the left child to {6}, and note the inversions.

Inversions identified so far:

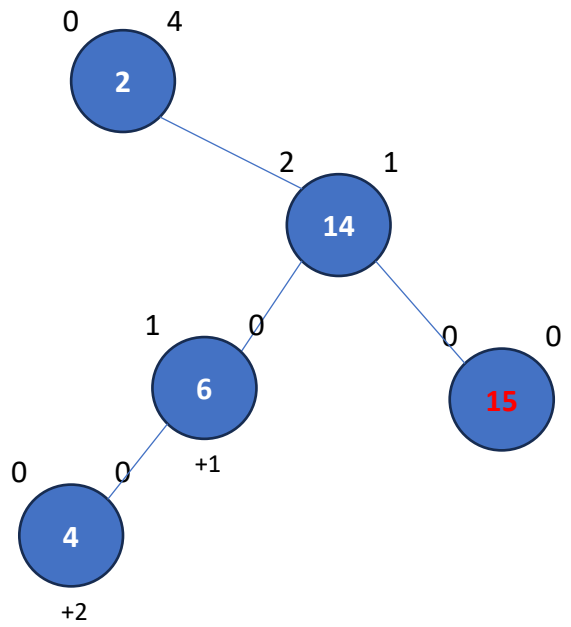
{6, 14}, {4, 14}, {4, 6}

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 4:
Evaluating index pos. 4: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

1. Insert {15}. Since it **IS NOT SMALLER** than the root, we traverse to the right.

2. At node {14} we identify that {15} **is NOT smaller** than {14}.

3. Also {14} has no right child, so we insert {15} as {14}'s right child

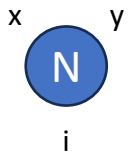
No inversions identified.

Inversions identified so far:

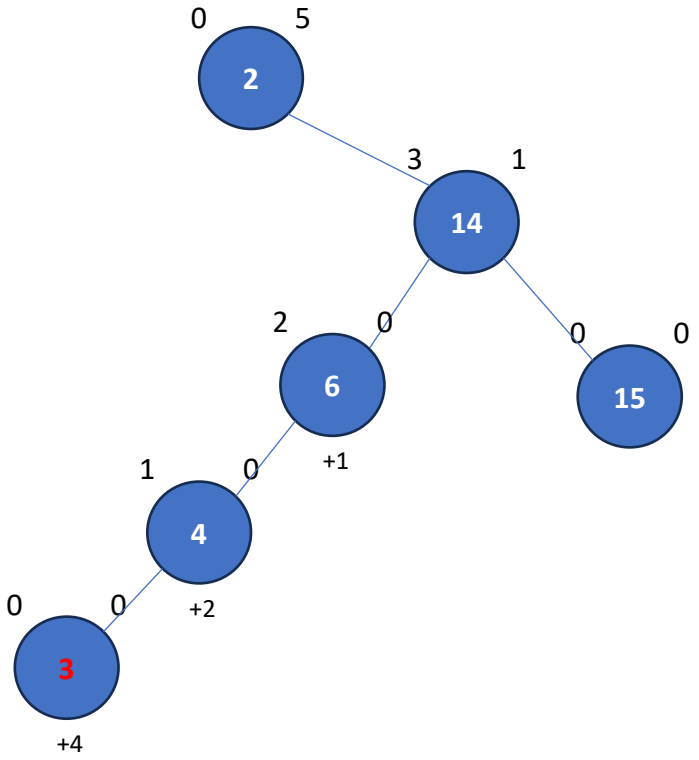
{6, 14}, {4, 14}, {4, 6}

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 5:
Evaluating index pos. 5: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

1. Insert {3}. Since it **IS NOT SMALLER** than the root, we traverse to the right.

2. At node {14} we identify that {3} **is smaller** than {14}.

We know that {14} already has sub-elements. We also know that all these sub-elements must **also** be larger than {3}, since elements to the right are larger than the current node {14}.

Inversions can be calculated as:
+1 from {14} and +1 from {14}'s **y**.

Inversions +2

3. We traverse left to {6}. Since {3} **is smaller**, we continue traversing left to {4} – and note an inversion from {6}.

Inversions +3

4. Since {3} **is smaller** and {4} has no left child, we insert {3} as the left child – and note another inversion from {4} (since {3} appears later in the sequence, and is smaller than a previous number in the same sequence)

Inversions +4

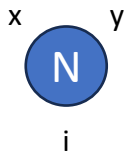
Note identified inversions.

Inversions identified so far:

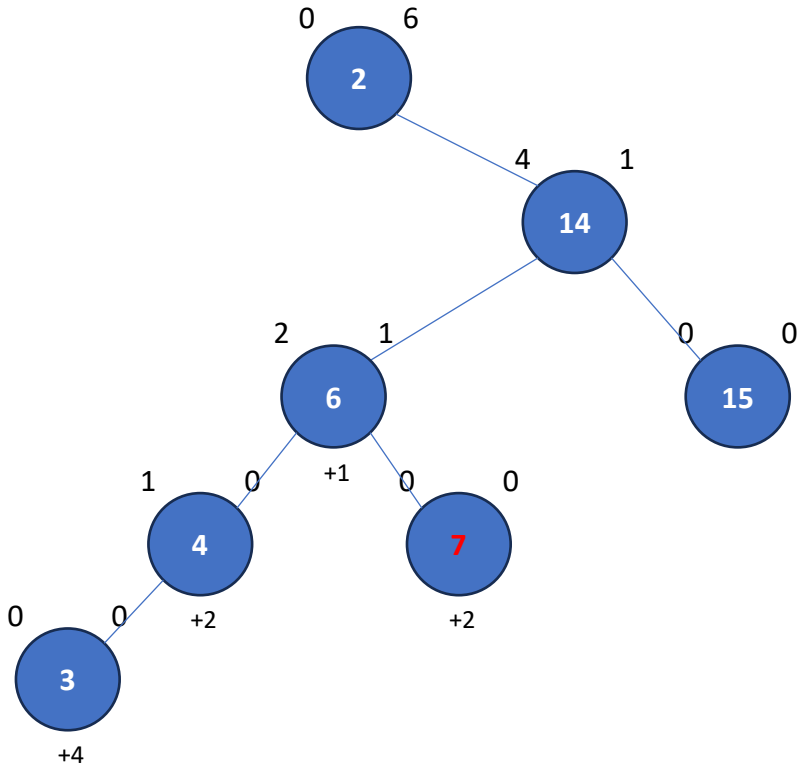
{6, 14}, {4, 14}, {4, 6}, ({3, 14},{3, 15}, {3, 6} & {3, 4})

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 6:
Evaluating index pos. 6: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

1. Insert {7}.

2. Since it **IS NOT SMALLER** than the root, we traverse to the right.

3. At node {14} we identify that {7} **is smaller** than {14}.

We know that {14} already has sub-elements. We also know that all these sub-elements must **also** be larger than {7}, since elements to the right are larger than the current node {14}.

Inversions can be calculated as:
+1 from {14} and +1 from {14}'s **y**.

Inversions +2

4. We traverse left to {6}. Since {7} **is larger**, we traverse right and see that there is no right child. We insert {7} as the right child..

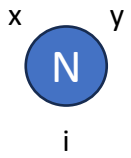
Note identified inversions.

Inversions identified so far:

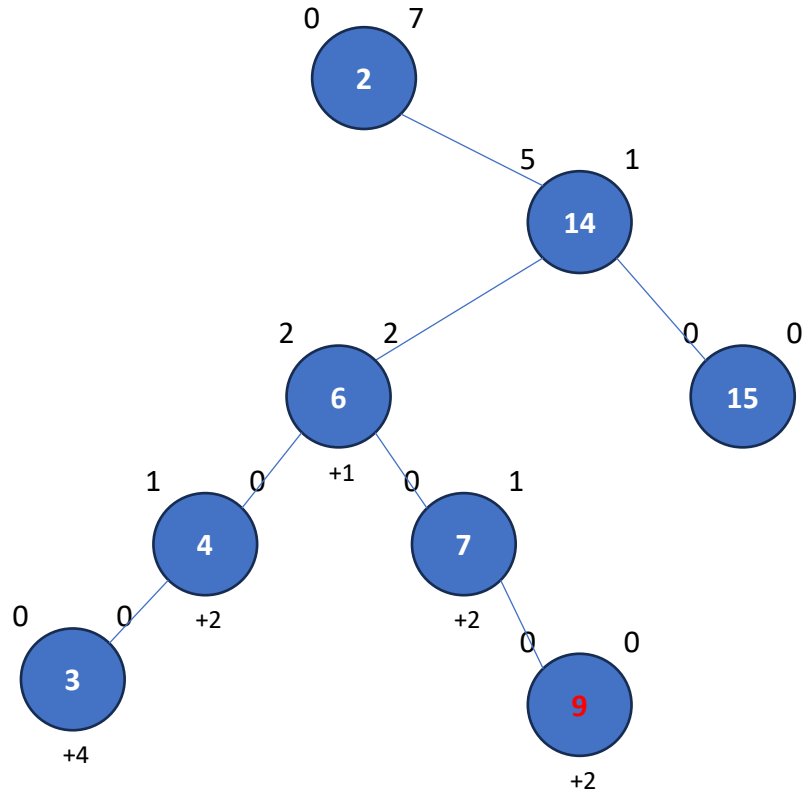
{6, 14}, {4, 14}, {4, 6}, ({3, 14},{3, 15}, {3, 6} & {3, 4}), ({7, 14} & {7, 15}).

Legend:

x *is # of elements in left subtree*
y *is # of elements in right subtree*
N *is number from the sequence*
i *is the number of inversions, this specific node encountered.*



Step 7:
Evaluating index pos. 7: {2, 14, 6, 4, 15, 3, 7, 9}



Step explanation:

1. Insert {9}.

2. Since it **IS NOT SMALLER** than the root, we traverse to the right.

3. At node {14} we identify that {9} **is smaller** than {14}.

We know that {14} already has sub-elements. We also know that all these sub-elements must **also** be larger than {7}, since elements to the right are larger than the current node {14}.

Inversions can be calculated as:
+1 from {14} and +1 from {14}'s **y**.

Inversions +2

4. We traverse left to {6}. Since {9} **is larger than** {6}, we traverse right to {7}.

5. Since {9} **is larger** than {7}, we continue traversing right and see that there is no child.

6. We insert {9} as the right child..

Note identified inversions.

Inversions identified so far:

{6, 14}, {4, 14}, {4, 6}, ({3, 14},{3, 15}, {3, 6} & {3, 4}), ({7, 14} & {7, 15}), ({9, 14} & {9, 15})

Final comments: {2, 14, 6, 4, 15, 3, 7, 9}

We obserbed from the beginning, that the given number sequence had these inversions:

11: {14,6}, {14,4}, {14,3}, {14,7}, {14,9}, {6,4}, {6,3}, {4,3}, {15,3}, {15,7}, {15,9}

Using our algorithm, we identified these:

11: {6, 14}, {4, 14}, {4, 6}, ({3, 14},{3, 15}, {3, 6} & {3, 4}), ({7, 14} & {7, 15}), ({9, 14} & {9, 15})

We can see that some of the numbers are inverted, but in actuality that doesn't matter, since for instance {14,6} and {6,14} both refer to the exact same inversion.

We see that this method indeed identifies the correct inversions.

How about the algorithm speed?

Since we are working with a unbalanced binary tree, we risk having to visit each number twice. (ie. In a reverse sorted sequence… 5,4,3,2,1). We could use a balanced search tree instead (which would cost on time complexity during balancing operations).

Time complexity in the **absolute worst** case would be O((n+1) * n/2), but more likely it would be somewhere closer to O(n * log(n)), although somewhat slower than a true O(n * log(n)) algorithm.

Why would absolute worst case be O((n+1) * n/2)?

Lets assume we have these 10 numbers {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}.

We want to find the inversions using this method. (below I am **only** evaluating time, not actual inversions)

We visit 10 once, tree depth being set to 0 (1st operation)

We visit 9 once, tree depth being set to n-9 (2nd operation)

We visit 8 once, tree depth being set to n-8 (3rd operation)

We visit 7 once, tree depth being set to n-7 (4th operation)

etc.

Basically we get this formula for the time: 10 * (1+2+3+4+5+6+7+8+9+10) = 55 time operations.

This can be expressed with the formula O((n+1) * n/2) = (10+1) * 10/2 = 55 time operations.

In comparison a true n * log2(n) time would be: 10 * LOG2(10) = 33 time operations.

And an n^2 time would be: 10*10 = 100 time operations.

So we can see that **in the absolute worst case** this algorithm is still significantly faster than an n^2 time algorithm,

but still somewhat slower than a n * log(n) algorithm.