Difference between binary search and binary search tree?

Asked 8 years, 10 months ago Modified 2 years, 9 months ago Viewed 28k times

39

What is the difference between binary search and binary search tree?

Are they the same? Reading the internet it seems the second is only for trees (up to 2 children nodes) and binary search doesn't follow this rule. I didn't quite get it.

algorithm data-structures binary-search-tree binary-search

Share Follow

edited Mar 12, 2018 at 1:04
sg7
5,848 2 31 39

asked Feb 5, 2014 at 18:57
RollRoll
7,966 18 73 129

5 A binary search is an algorithm, a binary search tree is a data structure. See binary search tree and binary search algorithm. – Ken White Feb 5, 2014 at 19:04

1 @KenWhite I agree with your comment about the difference; I think it's worth pointing out, though, that when you perform a binary search *on* something, you're *implicitly* treating that something as a binary search tree. I.e., the difference is largely one of the interface. – Joshua Taylor Feb 5, 2014 at 20:43

4 Answers

Sorted by: Highest score (default)

63

Binary Search Trees

A node in a binary tree is a data structure that has an element, and a reference to two other binary trees, typically called the left and right subtrees. I.e., a node presents an interface like this:

```
Node:
  element  (an element of some type)
  left     (a binary tree, or NULL)
  right    (a binary tree, or NULL)
```

A binary *search* tree is a binary tree (i.e., a node, typically called the root) with the property that the left and right subtrees are also binary search trees, and that all the elements of all the nodes in the left subtree are less than the root's element, and all the elements of all the nodes in the right subtree are greater than the root's element. E.g.,

```
      5
     / \
    /   \
```

```
  2     8
 / \   / \
1   3 6   9
```

Binary Search

Binary search is an algorithm for finding an element in binary search tree. (It's often expressed as a way of searching an ordered collection, and this is an equivalent description. I'll describe the equivalence afterward.) It's this:

```
search( element, tree ) {
  if ( tree == NULL ) {
    return NOT_FOUND
  }
  else if ( element == tree.element ) {
    return FOUND_IT
  }
  else if ( element < tree.element ) {
    return search( element, tree.left )
  }
  else {
    return search( element, tree.right )
  }
}
```

This is typically an efficient method of search because at each step, you can remove half the search space. Of course, if you have a poorly balanced binary search tree, it can be inefficient (it can degrade to linear search). For instance, it has poor performance in a tree like:

```
3
 \
  4
   \
    5
     \
      6
```

Binary Search on Arrays

Binary search is often presented as a search method for sorted arrays. This does not contradict the description above. In fact, it highlights the fact that we don't actually care how a binary search tree is implemented; we just care that we can take an object and do three things with it: get a element, get a left sub-object, and get a right sub-object (subject, of course, to the constraints about the elements in the left being less than the element, and the elements in the right being greater, etc.).

We can do all three things with a sorted array. With a sorted array, the "element" is the middle element of the array, the left sub-object is the subarray to the left of it, and the right sub-object is the subarray to the right of it. E.g., the array

```
[1 3 4 5 7 8 11]
```

corresponds to the tree:

```
      5
     / \
    /   \
   3     8
  / \   / \
 1  4  7   11
```

Thus, we can write a binary search method for arrays like this:

```
search( element, array, begin, end ) {
  if ( end <= begin ) {
    return NOT_FOUND
  }
  else {
    midpoint = begin+(end-begin)/2
    a_element = array[midpoint]
    if ( element == midpoint ) {
      return FOUND_IT
    }
    else if ( element < midpoint ) {
      return search( element, array, begin, midpoint )
    }
    else {
      return search( element, array, midpoint, end )
    }
  }
}
```

Conclusion

As often presented, binary search refers to the array based algorithm presented here, and binary search tree refers to a tree based data structure with certain properties. However, the properties that binary search requires and the properties that binary search trees have make these two sides of the same coin. Being a binary search tree often implies a particular implementation, but really it's a matter of providing certain operations and satisfying certain constraints. Binary search is an algorithm that functions on data structures that have these operations and meet these constraints.

Share Follow

edited Jul 5, 2019 at 11:42

answered Feb 5, 2014 at 20:52

Joshua Taylor
84.1k 9 150 346

binary trees can be stored in an array, in which case the nodes just consist of the data. – Deduplicator May 26, 2018 at 9:42

16

No, they're not the same.

Binary search tree:

- A tree **data structure**
- Each node has up to 2 children
- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key

Binary search:

- An **algorithm** that works on a sorted data structure (usually, but not necessarily, an array) and, at each step, looking at the value in the middle and recursing to either the left or the right, depending on whether the target value is smaller or greater than the value in the middle (or stopping if it's equal).

And, of course, a data structure is:

> A particular way of storing and organizing data in a computer so that it can be used efficiently.

While an algorithm is:

> A step-by-step procedure for calculations.

The *search* process in a binary search tree (where we look for a specific value in the tree) can be thought of as similar to (or an instance of, depending on your definitions and whether you're using a balanced BST) binary search, since this also looks at the 'middle' element and recurses either left or right, depending on the result of the comparison between that value and the target value.

Share Follow

edited Feb 5, 2014 at 20:38

answered Feb 5, 2014 at 20:33

Bernhard Barker
53.8k 14 102 135

1 Binary search is an algorithm, but notice that you can easily phrase it in such a way that it operates on binary trees: "an algorithm that works on a binary search tree and, at each step looks at the value at the root and recursing either to the left or the right, depending on whether the target value is smaller or greater than the value at the root, or stopping it it's equal)." Binary search tree, then, is really just a sort of *interface* or *presentation* of some data. A sorted array can be presented as a binary search tree: the "value at the root" is simply the middle element of the... – Joshua Taylor Feb 5, 2014 at 20:55

1 ...array, the left subtree is the sub-array to the left of the root, and the right subtree is the sub-array to the right of the root. – Joshua Taylor Feb 5, 2014 at 20:55

@JoshuaTaylor The first part of your comment is exactly what I was trying to say with the last paragraph. – Bernhard Barker Feb 5, 2014 at 22:15

10

For those who came here to quickly check which one to use. In addition to the answers,posted above, I would like to add complexities with respect to the operations for both of these techniques.

Binary search Tree:

Search: **θ(log(n))**, Worst case **(O(n))** for *unbalanced* BST,

Insert of node: **θ(log(n))** , Worst case **(O(n))** for *unbalanced* BST

Deletion of node: **θ(log(n))**, , Worst case **(O(n))** for unbalanced BST

Balanced Binary search Tree:

Search: **log(n)**,

Insert of node: **O(log(n))**

Deletion of node: **O(log(n))**

Binary Search on sorted array:

Search: **O(log(n))** But,

Insertion of node: Not possible if array is statically allocated and already full. Otherwise **O(n)** (**O(n)** for moving larger items of array to their adjacent right)

Deletion of node: **O(log(n)) + O(n)**. (So it would be O(log(n)) for finding position of deletion **+ O(n)** for moving larger items of array to their adjacent left)

So based on your requirements, you can choose whether you need quick inserts and deletes. If you don't need these, then keeping things in sorted array will work for you, as array will take less memory compared to tree

Share Follow

edited Aug 6, 2018 at 12:11

answered Apr 21, 2018 at 14:38

Alok Nayak
2,322 22 28

Please add for BST: log n, given the tree is balanced. – Israel May 26, 2018 at 9:25

Why log(n) + O(n) (O(log(n)) for finding position of deletion how did you calculate it ? – shareef Aug 3, 2018 at 17:29

I have reformatted the answer. For finding position of deletion of a value, it can do a binary search on this sorted array which would be O(log(n)). – Alok Nayak Aug 3, 2018 at 19:36

0

1. to search for an element using binary search the elements should be represented in sequential memory locations like using an array where u need to know the size of the array to find the middle element to search using binary search tree we need not have the data in sequential locations .. we need to add elements into a BST node ... each BST node contains its right and left child so knowing the root is enough to conduct a search on the tree
2. since u use an array for binary search the insertion and deletion will be easy but in BST we need to traverse through the height of tree even in worst case
3. To do binary search we need the elements to be in sorted order but BST doesn't follow this rule
4. Now coming to the searching time complexities ..

   a) using binary search the worst case complexity is O(logn)

   b) using BST the worst case complexity is O(n) i.e., if the tree is skewed then it just works like a linked list and we end up searching all the elements(thats why we need to implement balanced BSTs)
5. Binary search needs O(1) space complexity since the locations are consecutive .. BST needs O(n) space for each node we need extra space to store the pointer of its child nodes

Share Follow

answered Mar 31, 2020 at 20:27

sushmitha

69 1 2