12. Binary search tree

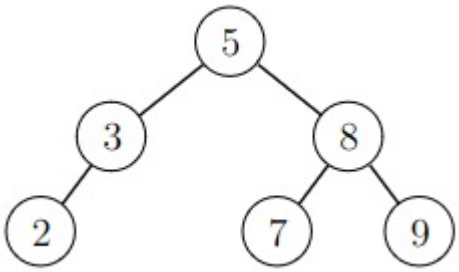
Binary search tree is a data structure that maintains a set of elements. The basic operations are the same as with hashing: elements can be added, searched and removed efficiently.

Binary search tree differs from hashing in that it maintains the elements in order. Because of this, the smallest and the largest element in the set can be found efficiently, which is not possible with hashing.

The Python standard library does not have an implementation of a binary search tree, which makes using them a little bit more difficult in Python. In this chapter, we develop our *own* implementation of the binary search tree.

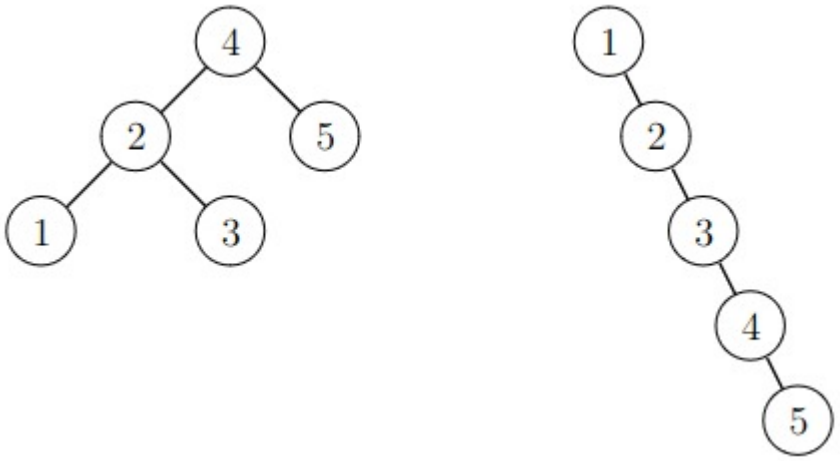
Set as a binary tree

A binary search tree is a binary tree, where each node stores one element of the set. For example, the following binary search tree corresponds to the set $\{2, 3, 5, 7, 8, 9\}$:



A binary search tree is organized so that for every node the elements in the left subtree are smaller than the element in the node, and symmetrically the elements in the right subtree are larger than the element in the node. For example in the tree above, the elements in the left subtree of the root are 2 and 3, which are smaller than the root element 5. Similarly, the elements in the right subtree are 7, 8 and 9, all of which are larger than the root element 5.

The element locations can be chosen freely in the binary tree as long as the above ordering condition is satisfied by every node. Thus the same set can be represented by different binary search trees. For example, the following two binary search trees both represent the set $\{1, 2, 3, 4, 5\}$:

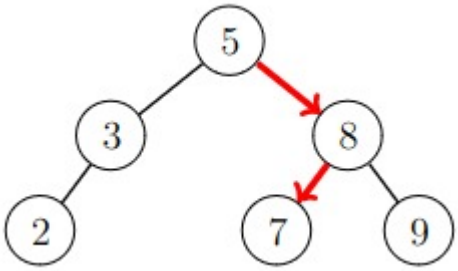


Let us next consider how set operations can be implemented using a binary search tree.

Finding an element

When searching for an element, we start at the root. If the element in the node is smaller than the query element, the search continues in the right subtree. If the element in the node is larger than the query element, the search continues in the left subtree. This continues until we find the element, or until there is no child in the direction where the search should continue. The latter case means that the query element is not in the set.

For example, the search route for an element 7 is illustrated in the following image:



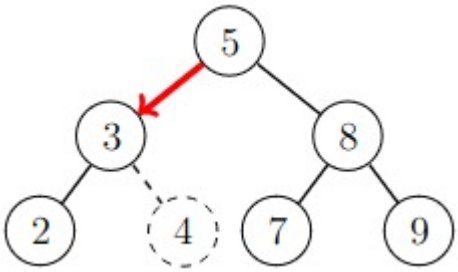
The search starts at the root, where the element is 5. This is smaller than the element 7 and the search continues in the right child. The element there is 8, which is larger than the element 7. Thus the search continues to the left child, where the element 7 is found.

Adding an element

When adding an element, the first stage is to search for the element in the set. If the element is found, no addition is performed, because a set can contain an

element only once. If the element is not found, a new node is inserted where the seach would have continued next, and the new element is stored in the new node.

For example, the following image illustrates the addition of an element 4:



The search for the element 4 starts at the root, where the element is 5. Next the search goes to the left child, where the element is 3. Here the search should continue to the right child but the node has no right child. This means that the element 4 is not in the tree, and a new node storing the element 4 is added as that missing right child. This ensures that a later search for the element 4 will reach the node.

Smallest element

A search for the smallest element of the set starts at the root and always continues to the left child as long as possible. When going left is no more possible, the smallest element has been found.

Largest element

A search for the largest element follows a similar procedure but always goes to the right instead of the left.

Successor

The successor of an element x in the set is the smallest element that is larger than x. A search for the successor starts at the root, goes left when the element in the node is larger than x, and otherwise right. The search continues until there is nowhere to go. The desired element is among those encountered during the search, i.e., the smallest of them that is larger than x.

Predecessor

The predecessor of an element x in the set is the largest element that is smaller than x. A predecessor search is symmetric to a successor search: go right when

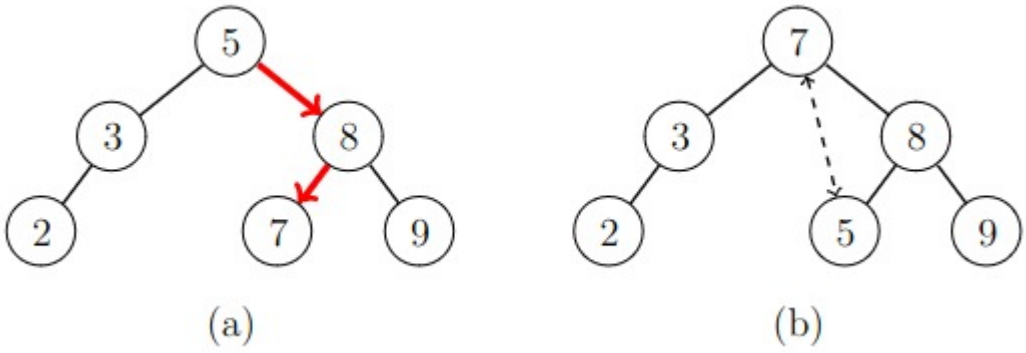
the element is smaller than x, and left oterwise. The predecessor is the largest element smaller than x among the encountered elements.

Removing an element

When removing and element, the first step is to find the node containing the element. The next step depends on the situation of the node in the tree. There are three cases:

- The node has no children. Then the node can just be removed.
- The node has one child. Then the node can be removed and replaced by the its only child.
- The node has two children. Then find the successor of the element, and swap the elements in the two nodes. Then the new node of the element can be removed, because it has at most one child.

The following image shows an example, where we want to remove the element 5. Since the node with the element 5 has two children, the element 5 is swapped with its successor 7. Then the new node of the element 5 is easy to remove, because it has no children.



Notice that when the node has two children, the successor node cannot have a left child and can always be removed easily. Also, the swap of the two elements does not violate the ordering conditions relative to any other elements.

Implementation in Python

Next we will begin developing an implementation of a binary search tree in Python. The goal here is a class `TreeSet` that can be used as follows:

```
s = TreeSet()

s.add(1)
```

```
s.add(2)
s.add(3)

print(2 in s) # True
print(4 in s) # False

print(s) # [1, 2, 3]
```

The method `add` adds an element to the set, the operator `in` reports if an element is in the set, and the string representation of the set is a list of its elements.

The following class `Node` stores the information related to a node in the tree:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

Each node contains three attributes: the element in the node (`value`) and references to the children (`left` and `right`).

The class `TreeSet` implements the binary search tree. Here is an initial template:

```
class TreeSet:
    def __init__(self):
        self.root = None
```

The method `__init__` defines a reference to the root of the tree. Initially, there are no nodes, which is why `root` is `None`.

Adding an element

The following method `add` adds an element to the set:

```
    def add(self, value):
        if not self.root:
            self.root = Node(value)
            return
```

```
        node = self.root
        while True:
            if node.value == value:
                return
            if node.value > value:
                if not node.left:
                    node.left = Node(value)
                    return
                node = node.left
            else:
                if not node.right:
                    node.right = Node(value)
                    return
                node = node.right
```

If the tree is empty, the element is stored in a new node that becomes the root of the tree. Otherwise, the method performs a search starting from the root.

If a node containing the element is found, the method ends, because the element is already in the set. If the element in the node is larger than the element to be added, the search goes left, and if the element in the node is smaller than the element to be added, the search goes right. If there is no child where the search should continue, the element is stored in a new node that takes the place of the missing child, and the method ends.

Finding an element

The following method `__contains__` checks if a given element is in the set. The method `__contains__` is called by the operator `in`.

```
    def __contains__(self, value):
        if not self.root:
            return False

        node = self.root
        while node:
            if node.value == value:
                return True
            if node.value > value:
                node = node.left
            else:
```

```
                node = node.right

        return False
```

If the tree is empty, the element is not in the tree and the method returns `False`. Otherwise the method searches for the element similarly as with addition. If the element is found, the method returns `True`. If the element is not in the tree, the search will eventually go outside the tree (`node` becomes `None`), and the method ends and returns `False`.

String representation

The following method `__repr__` constructs a string representation of the set, which contains the elements of the set as a list.

```
    def __repr__(self):
        items = []
        self.traverse(self.root, items)
        return str(items)

    def traverse(self, node, items):
        if not node:
            return
        self.traverse(node.left, items)
        items.append(node.value)
        self.traverse(node.right, items)
```

The method `__repr__` uses the method `traverse` that visits all nodes of the tree and adds their elements to the list. The traversal visits first all nodes in the left subtree, then adds the element in the node to the list, and then visits the nodes in the right subtree. This ensures that the elements are added to the list in the order of their value.

Other operations

This implementation does not yet have methods for finding the smallest and the largest elements or for removing an element. The implementation of these methods is a part of the exercises for this week.

Balanced trees

The operations on binary search trees follow a route from the root of the tree down towards the leafs of the tree. The efficiency of the tree depends on how long these routes can be. The length of the longest possible route is equal to the height of the tree h. Thus the time complexity of the operations can be stated to be $O(h)$.

An arbitrary binary search tree is not necessarily an efficient data structure, because the tree can be tall. For example, if we add n elements into the tree in the order $1, 2, \ldots, n$, all element go into a single chain and the height of the tree is $n-1$. Then the time complexity of the tree operations is $O(n)$.

However, it is possible to implement a binary search tree so that the elements are distributed evenly across the tree and the height of the tree is always of order $\log n$. Then the tree operations are efficient with time complexity $O(\log n)$. Such a binary search tree is called *balanced*.

A balanced binary search tree is implemented so that the height of the tree never grows too big. For example, an AVL tree is a balanced binary search tree, where the heights of the left subtree and the right subtree of the same node cannot differ by more than 1. This condition ensures that the height of the tree is always of order $\log n$. To maintain that condition, the structure of the tree is modified with rotations when necessary.

Example: Hotel

The following is an example of a problem that can be solved efficiently using a binary search tree:

Task

A hotel has n rooms, each of which has a certain capacity (number of people). The hotel receives m groups of visitors. Your task is to process the groups in order from left to right, and assign to each group the smallest room with sufficient capacity for the whole group, or report that no suitable room is available.

For example, suppose the room capacities are $[2, 4, 4, 8]$ and the group sizes are $[4, 6, 2, 5, 2]$. Then the desired answer is $[4, 8, 2, 0, 4]$ (0 means that no room was assigned):

- The first group gets a room with capacity 4.

- The second group gets a room with capacity 8.
- The third group gets a room with capacity 2.
- The fourth group does not get a room.
- The fifth group gets a room with capacity 4.

Assume that we can use a class `TreeSet` with the following methods:

- `add(x)`: adds an element `x` to the set
- `next(x)`: returns the smallest element larger than `x` (or `None` if there is no such element)
- `remove(x)`: removes the element `x` from the set

The task can be solved using these methods as follows:

```
def find_rooms(sizes, requests):
    rooms = TreeSet()
    counter = 0
    for size in sizes:
        counter += 1
        rooms.add((size, counter))

    result = []
    for request in requests:
        room = rooms.next((request, 0))
        if room == None:
            result.append(0)
        else:
            rooms.remove(room)
            result.append(room[0])

    return result
```

The function creates a set `rooms` and adds all available rooms to the set. Since multiple rooms can have the same capacity but the set cannot contain the same element multiple times, each room is represented as a pair, where the first part is the room capacity and the second is a unique room number. In the example case, the pairs `(2, 1)`, `(4, 2)`, `(4, 3)` and `(8, 4)` are added to the set.

Then the function goes through the groups and searches for a suitable room using the method `next`. The parameter for the method `next` is the pair `(request, 0)`, where `request` is the size of the group. Since `0` is smaller than any room number, the method finds the smallest room with a capacity at least

`request`.

The time complexity of the function is $O(n \log n + m \log n)$ assuming that the binary search tree is balanced so that the tree operations take $O(\log n)$ time.

Why not in Python?

Many programming languages offer an implementation of a binary search tree, but the Python standard library does not. Why is this?

The likely explanation is that the Python developers did not consider the binary search tree to be so frequently needed that it should be in the standard library. Python relies on hashing-based data structures (`set` and `dict`), which are sufficient for most purposes.

Indeed, hashing as well as sorting and a heap are alternatives to a binary search tree. In many tasks, one of the efficient solutions is to use a binary search tree, but often there are other efficient solutions without binary search trees.

If a binary search tree is needed, there are many implementation outside the standard library. Using one of these might be a better alternative than implementing your own binary search tree.

Other programming languages

In C++, the data structures `std::set` and `std::map` implement a set and a dictionary using a binary search tree. For example, the following code creates a set, adds elements with the function `insert` and searches for a successor using the function `upper_bound`.

```
std::set<int> items;

items.insert(1);
items.insert(3);
items.insert(6);
items.insert(8);

auto it = items.upper_bound(4);
std::cout << *it << "\n"; // 6
```

Java has similar data structures `TreeSet` and `TreeMap`. For example, the

following code corresponds to the above C++ code.

```
TreeSet<Integer> items = new TreeSet<>();

items.add(1);
items.add(3);
items.add(6);
items.add(8);

int item = items.higher(4);
System.out.println(item); // 6
```

JavaScript, similarly to Python, does not have a standard library implementation of binary search trees.