

Algorithms and Data Structures

B5. Binary Search Trees

Gabriele Röger and Patrick Schneider

University of Basel

April 15, 2026

1 / 29

Algorithms and Data Structures

April 15, 2026 — B5. Binary Search Trees

B5.1 Binary Search Trees

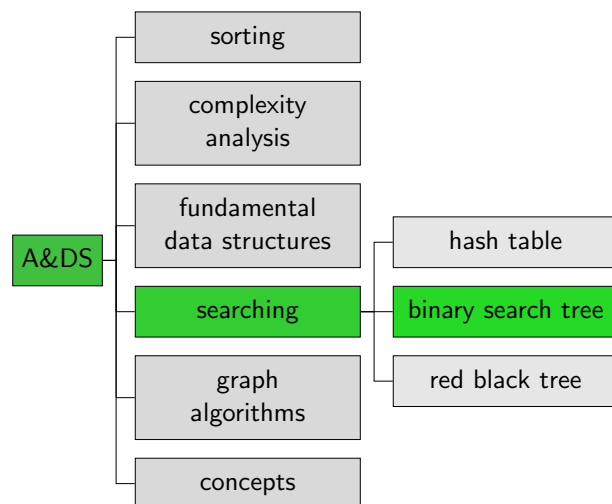
B5.2 Queries

B5.3 Insertion and Deletion

B5.4 Summary

2 / 29

Content of the Course



3 / 29

B5. Binary Search Trees

Binary Search Trees

B5.1 Binary Search Trees

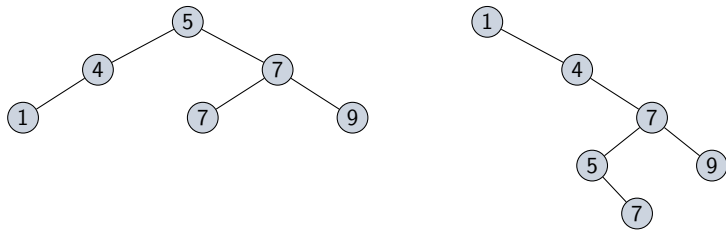
4 / 29

Binary Search Tree

Definition (Binary Search Tree)

A **binary search tree** T is a **binary tree** that satisfies the **binary search tree property**: For every node x in T

- ▶ all nodes y in the **left** subtree of x have a **key smaller than x** ($y.key \leq x.key$), and
- ▶ all nodes y in the **right** subtree of x have a **key larger than x** ($y.key \geq x.key$).



5 / 29

Binary Search Trees: Operations

We will support the following operations:

- ▶ **search**(n , k) given node n and key k , returns pointer to element with key k in the tree rooted by n , or `None` if there is no such element in the tree.
- ▶ **insert**(n , k , v) adds a node with key k and value v to tree rooted in node n .
- ▶ **delete**(n) given a pointer n to a node in the tree, removes n .
- ▶ **minimum**(n) and **maximum**(n) return the element with the smallest and largest key, respectively, from the tree rooted in node n .
- ▶ **successor**(n) given node n whose key is from a totally ordered set, returns a pointer to the next larger element in the tree, or `None` if n holds the maximum element.
- ▶ **predecessor**(n) given node n whose key is from a totally ordered set, returns a pointer to the next smaller element in the tree, or `None` if n holds the minimum element.

6 / 29

Binary Search Tree: Representation

We use a class `Node` for the nodes of the tree:

```

1 class Node:
2     def __init__(self, key, value):
3         self.key = key
4         self.value = value
5         self.parent = None # will be set to parent node
6         self.left = None # will be set to left child node
7         self.right = None # will be set to right child node

```

7 / 29

Binary Tree: Inorder Tree Walk

An **inorder tree walk** prints the key of a root of a subtree between the values of the left subtree and those in the right subtree:

```

1 def inorder_tree_walk(node):
2     if node is not None:
3         inorder_tree_walk(node.left)
4         print(node.key, end=" ")
5         inorder_tree_walk(node.right)

```

An **inorder tree walk** from the root of a **binary search tree** prints all keys in **sorted order**.

Analogously:

- ▶ **preorder tree walk**: root, then left subtree, then right subtree
- ▶ **postorder tree walk**: left subtree, then right subtree, then root

8 / 29

Jupyter Notebook



Jupyter notebook: bst.ipynb

Inorder Tree Walk: Running Time

Theorem

If the subtree rooted at node has n nodes then $\text{inorder_tree_walk}(\text{node})$ has running time $\Theta(n)$.

- ▶ Every node gets printed $\rightarrow \Omega(n)$.
- ▶ Let d be an upper bound on the (constant) running time of everything except for the recursive calls.
- ▶ Let $k < n$ be the number of nodes in the left subtree (and thus $n - k - 1$ be the number of nodes in the right subtree).
- ▶ We prove by induction that $T(n) < 2dn + d$.
- ▶ Base case ($n = 0$, empty tree): $T(0) \leq d = 2d \cdot 0 + d$
- ▶ Ind. hypothesis: for all $0 \leq m < n$: $T(m) < 2dm + d$
- ▶ Ind. step: $n - 1 \rightarrow n$

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &\leq 2dk + d + 2d(n - k - 1) + d + d = 2dn + d \end{aligned}$$

B5.2 Queries

Search

Find an entry with the given key k or return `None` if there is no such entry in the tree with the given root:

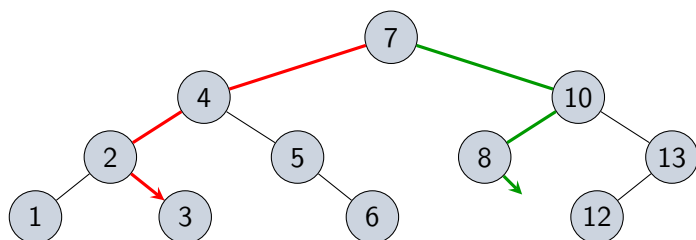
```

1 def search(root, k):
2     node = root
3     while node is not None:
4         if node.key == k:
5             return node
6         elif node.key > k:
7             node = node.left
8         else:
9             node = node.right
10    return None # no node with key k in tree

```

The nodes encountered during the search form a simple path downward from the root, so the running time is in $O(h)$, where h is the height of the tree.

Search: Illustration



Search for $k = 3$ (red) and for $k = 9$ (green).

Minimum and Maximum

Find an entry with the smallest among all keys in the tree rooted by node:

```

1 def minimum(node):
2     while node.left is not None:
3         node = node.left
4     return node

```

Running time: $O(h)$ with h height of tree.

Maximum: Find an entry with a largest key in the tree.

↔ exercise in notebook

Successor

Given element x , return a pointer to the successor in an inorder tree walk or `None` if x is the maximum node.

If keys are distinct, this is the next larger element in the tree (otherwise?).

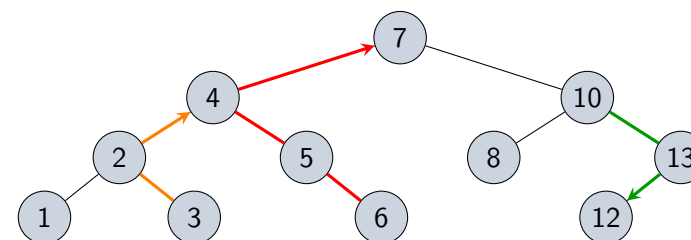
We can determine the successor without inspecting the keys.

```

1 def successor(node):
2     if node.right is not None:
3         # return left-most node in the right subtree
4         return minimum(node.right)
5     # otherwise, we must go upwards in the tree
6     parent = node.parent
7     while parent is not None and node == parent.right:
8         node = parent
9         parent = node.parent
10    return parent

```

Successor: Illustration and Running Time



Successor of node with $k = 6$ (red), $k = 3$ (orange) and $k = 10$ (green).

We either follow a simple path up the tree or down the tree.
→ Running time $O(h)$

Predecessor

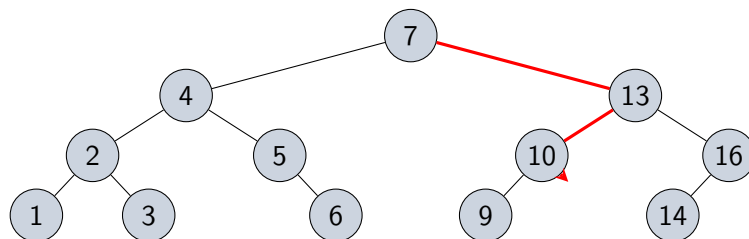
Given element x , return a pointer to the predecessor in an inorder tree walk or `None` if x is the minimum node.

- ▶ Implementation is symmetric to `successor`.
[Exercise in Jupyter notebook](#)
- ▶ The resulting running time is $O(h)$.

B5.3 Insertion and Deletion

Insertion

- ▶ Descend from root similar as in a search for the key (tracking also the parent of the current node). $\rightarrow O(h)$
- ▶ Insert the new node at the identified position. $\rightarrow O(h)$
- ▶ Overall running time $O(h)$.



Insert $k = 12$

Insertion: Implementation

```

1 def insert(root, key, value):
2     current = root
3     parent = None
4     # search for the right position
5     while current is not None:
6         parent = current
7         if current.key > key:
8             current = current.left
9         else:
10            current = current.right
11    # insert node
12    node = Node(key, value)
13    node.parent = parent
14    if parent is None: # tree was empty
15        self.root = node
16    elif key < parent.key:
17        parent.left = node
18    else:
19        parent.right = node

```

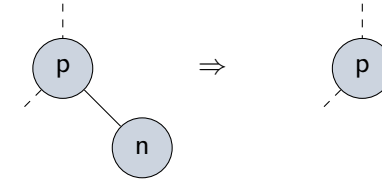
Deletion

Deleting a node n is somewhat more complicated:

- ▶ Conceptually, we distinguish three cases, that we treat differently.
- ▶ In the implementation, we organize the code a bit differently.

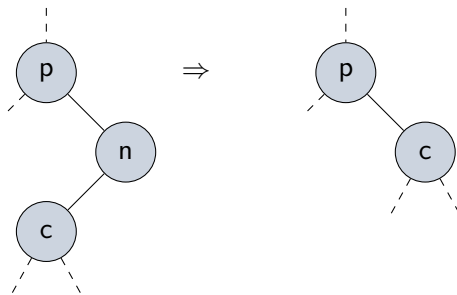
Deletion Conceptually: Case 1

- ▶ If node n has no children, replace the child reference of the parent with None.



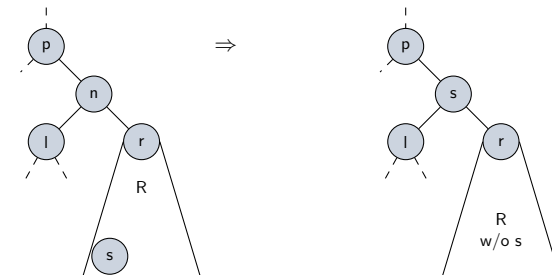
Deletion Conceptually: Case 2

- ▶ If node n has one child c , this child becomes the new child of n 's parent node.



Deletion Conceptually: Case 3

- ▶ If node n has two children, the successor s of n takes over n 's position.
- ▶ The rest of n 's original right subtree becomes the right subtree of s .
- ▶ The left subtree of n becomes the left subtree of s .



Helper Function transplant

Replace subtree rooted at node u with subtree rooted at node v .

```

1 def transplant(u, v):
2     # Also works if v is None.
3     if u.parent is None:
4         T.root = v
5         # v is new root of tree (cf. notebook)
6     elif u == u.parent.left:
7         u.parent.left = v
8     else:
9         u.parent.right = v
10    if v is not None:
11        v.parent = u.parent

```

Running time: $O(1)$

Deletion: Implementation

```

1 def delete(node):
2     if node.left is None:
3         # Case 1 and case 2, where single child is right child.
4         transplant(node, node.right)
5     elif node.right is None:
6         # Case 2, where single child is right child.
7         transplant(node, node.left)
8     else: # Case 3
9         ... # next slide

```

Deletion: Implementation (Continued)

```

8     else: # Case 3
9         s = minimum(node.right)
10        if node.right != s:
11            # remove s from right subtree
12            # (replacing it by its right # child), and
13            # make this subtree the right child of s.
14            transplant(s, s.right)
15            s.right = node.right
16            node.right.parent = s
17        # s takes over place of node with
18        # left subtree of node as left subtree
19        transplant(node, s)
20        s.left = node.left
21        s.left.parent = s

```

Running time: $O(h)$ with h height of tree
(everything constant except for minimum).

B5.4 Summary

Summary

- ▶ In a **binary search tree** the left subtree of every node n with key k only contains keys at most as large as k and the right subtree only keys at least as large as k .
- ▶ The queries **search**, **minimum**, **maximum**, **predecessor** and **successor** and the modifying operations **insert** and **delete** have **running time $O(h)$** , where h is the height of the tree.
- ▶ Binary search trees **can degenerate to chains of nodes**, in which case these operations take linear time in the number of entries.