Algorithms and Data Structures

B5. Binary Search Trees

Gabriele Röger

University of Basel

April 17, 2024

Algorithms and Data Structures

April 17, 2024 — B5. Binary Search Trees

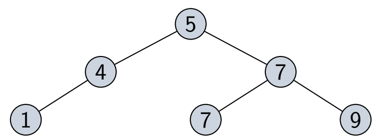
B5.1 Binary Search Trees

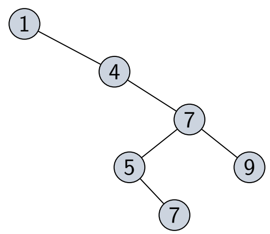
Binary Search Tree

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 ($y.key \geq x.key$).





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.

Binary Search Tree: Representation

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

```python
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.parent = None  # will be set to parent node
        self.left = None   # will be set to left child node
        self.right = None  # will be set to right child node
```


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:

```python
def inorder_tree_walk(node):
    if node is not None:
        inorder_tree_walk(node.left)
        print(node.key, end=" ")
        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

Jupyter Notebook

Jupyter notebook: `bst.ipynb`

Inorder Tree Walk: Running Time

Theorem
If the subtree rooted at `node` has n nodes then `inorder_tree_walk(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

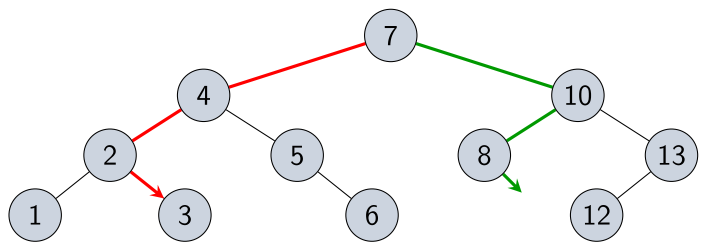
Search

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

```
def search(root, k):
    node = root
    while node is not None:
        if node.key == k:
            return node
        elif node.key > k:
            node = node.left
        else:
            node = node.right
    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`:

```
def minimum(node):
    while node.left is not None:
        node = node.left
    return node
```

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

Maximum: Find an entry with a largest key in the tree.
$\rightsquigarrow$ exercise in notebook

Successor

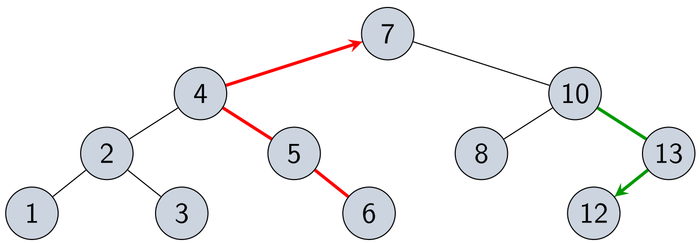
Given element `x`, return a pointer to the successor in an inorder tree walk or `None` if `x` is the maximum node.

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

We can determine the successor without inspecting the keys.

```
def successor(node):
    if node.right is not None:
        # return left-most node in the right subtree
        return minimum(node.right)
    # otherwise, we must go upwards in the tree
    parent = node.parent
    while parent is not None and node == parent.right:
        node = parent
        parent = node.parent
    return parent
```


Successor: Illustration and Running Time



Successor of node with $k = 6$ (red) and for $k = 10$ (green).

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

Predecessor

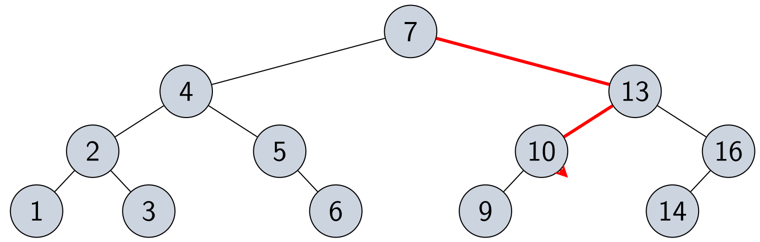
Given element `x`, return a pointer to the predecessor in an inorder tree walk or `None` if `x` is the minimum node.

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

B5.3 Insertion and Deletion

Insertion

- Decent 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

```python
def insert(root, key, value):
    current = root
    parent = None
    # search for the right position
    while current is not None:
        parent = current
        if current.key > key:
            current = current.left
        else:
            current = current.right
    # insert node
    node = Node(key, value)
    node.parent = parent
    if parent is None: # tree was empty
        self.root = node
    elif key < parent.key:
        parent.left = node
    else:
        parent.right = node
```

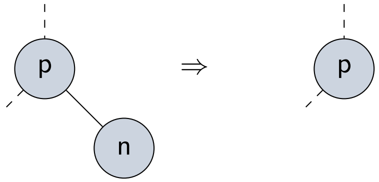

Deletion

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.

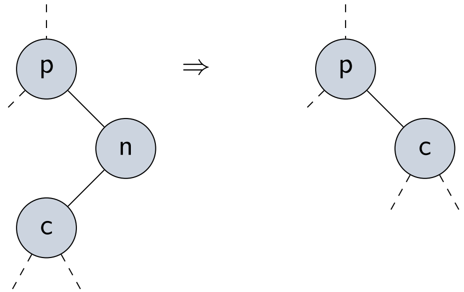
Deletion Conceptually: Case 1

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



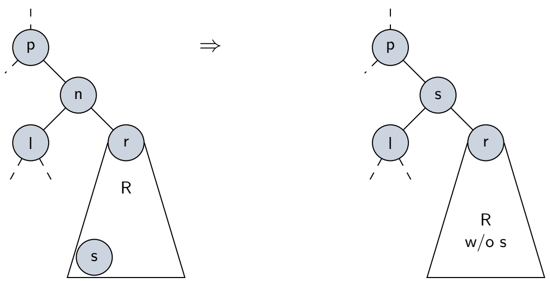
Deletion Conceptually: Case 2

- 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.

```
def transplant(u, v):
    # Also works if v is None.
    if u.parent is None:
        T.root = v
        # v is new root of tree (cf. notebook)
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    if v is not None:
        v.parent = u.parent
```

Running time: $O(1)$

Deletion: Implementation

```python
def delete(node):
    if node.left is None:
        # Case 1 and case 2, where single child is right child.
        transplant(node, node.right)
    elif node.right is None:
        # Case 2, where single child is right child.
        transplant(node, node.left)
    else: # Case 3
        ... # next slide
```


Deletion: Implementation (Continued)

```python
    else: # Case 3
        s = minimum(node.right)
        if node.right != s:
            # remove s from right subtree
            # (replacing it by its right # child), and
            # make this subtree the right child of s.
            transplant(s, s.right)
            s.right = node.right
            node.right.parent = s
        # s takes over place of node with
        # left subtree of node as left subtree
        transplant(node, s)
        s.left = node.left
        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.