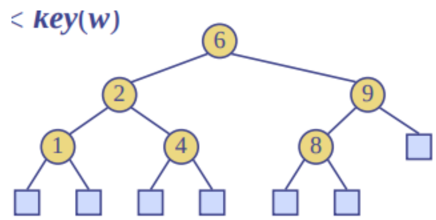
1 Binary Search Trees

A binary search tree is a binary tree storing keyvalue entries at its internal nodes and satisfying the following search tree property: Let u, v, and w be any three nodes such that u is in the **left** subtree of v and w is in the **right** subtree of v. Then we must have

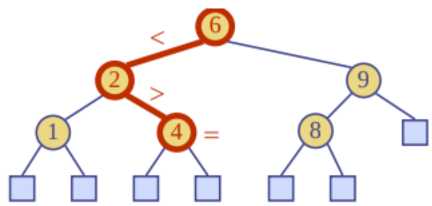
$$key(u) < key(v) < key(w)$$



External nodes **do not** store items and likely are **not actually implemented**, but are just `null` links from the parent

1.1 Search

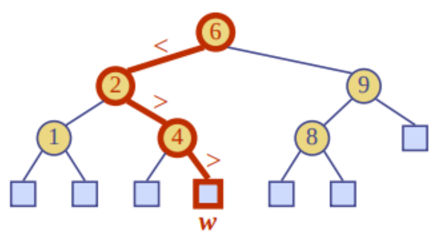
To search for key k, trace a downward path starting at the root. The next node visited depends on the outcome of the comparison of k with the key of the current node. If we reach a leaf, the key is not found and we return `null`.

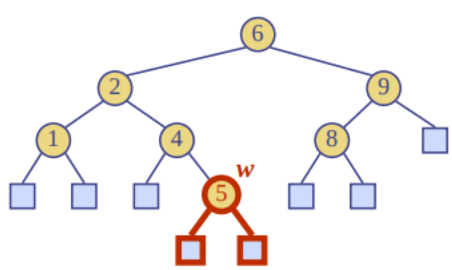


1.2 Insertion

Have to insert k where a $get(k)$ would find it!. So natural that `insert(k,v)` starts with `get(k)`. We search for key k (using *TreeSearch*).

- If k is already in the tree then just replace the value
- Otherwise, k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node



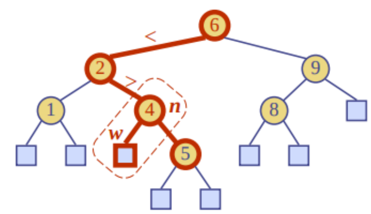


1.3 Deletion

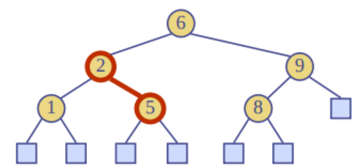
- As usual we start by trying to find(k)
- Four cases: (think of the externals as `null`)
  - k is not present. (Do nothing)
  - n has no children (straightforward)
  - n has one child,
  - n has two children

1.3.1 Dealing with one child

- `remove(4)`
- Search for key 4. Let n be the node storing 4

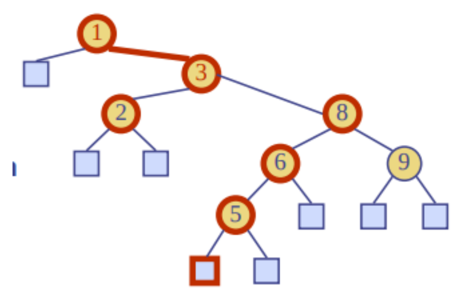


- Node n has a `null` (leaf) child w, and a real child 5
- We remove n from the tree and connect 5 back to the parent of n

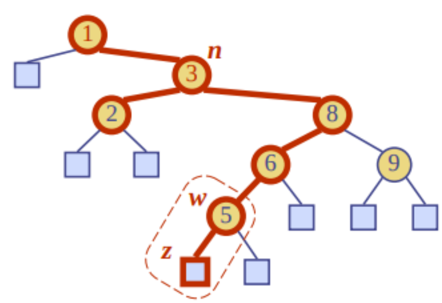


1.3.2 Dealing with two children

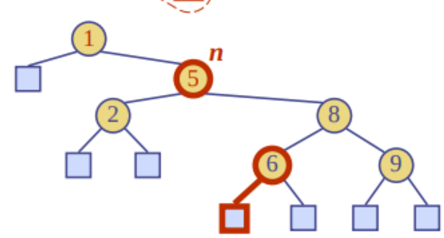
- `remove(3)`
- As a sorted list, we would have `[1,2,3,5,6,8,9]`
- If we want to remove 3 then copy a key k that is adjacent to 3 on top of 3 and then delete that key k
- Options in this case are k being 2 or 5. We will focus on the nextKey, that is, 5



- The key node n has two internal children
- Find the internal node w that follows n in an inorder traversal



- Copy `key(w)` into node n 3
- Remove node w and its left child z (which must be a leaf) by means of same procedure as before for one child



1.4 Balanced Trees

Binary search trees: if all levels filled, then search, insertion and deletion are `O(log N)`. However, performance may deteriorate to linear if nodes are inserted in order. (10 -¿ 11 -¿ 12)

1.5 Performance

Consider a binary search tree of height h with n items the space used is `O(n)`, methods find, **insert and remove** take `O(h)` time. The height h is `O(n)` in the worst case and `O(log n)` in the best case. (Worst case: all nodes only have single child. Best case all nodes have two children)

1.6 Self balancing

Constantly **re-structure** the trees: Keep the trees height **balanced** so that the height is logarithmic in the size Performance **always logarithmic**.

1.6.1 Issues

Suppose a very imbalanced search tree, there are always corresponding balanced search trees. Could make trees balanced using a *total rebuild*. But would require $O(n)$, and so very **inefficient** compared to the desired $O(log\, n)$. Re-balancing needs to be $O(log\, n)$ or $O(height)$. Suggests re-balancing needs to just look at the path to some **recently changed node**, not the entire tree. A priori, it is not at all obvious that this is possible!

1.7 AVL trees

AVL (Adelson-Velskii & Landis) trees are *binary search trees* where nodes also have additional information:

- The difference in **depth** between their right and left subtrees (*balance factor*).
- For each node, the *balance factor* of that node is $height(rightsubtree)height(leftsubtree)$
- In an AVL tree the balance of every node is allowed to be only **0,1 or -1**.
- AVL trees do **dynamic self-balancing** in $O(log\, n)$ time

1.8 Top down and bottom up insertion

Top down insertion algorithms make changes to the tree (necessary to keep the tree balanced) as they search for the place to insert the item. They make **one** pass through the tree. **Bottom up**: first insert the item, and then work **back** through the tree making changes. **Less efficient** because make **two passes** through the tree. (Need to find the item going from the top)

Reference section

load factor
a measure of how full the hash table is allowed to get before its capacity is automatically increased.