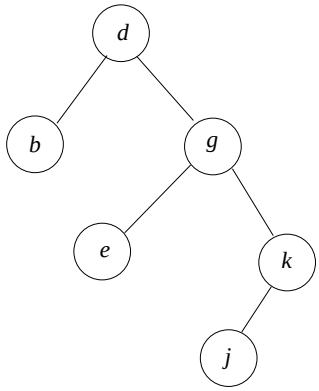
8 Binary search trees

A *binary search tree* is a binary tree whose nodes are labelled with ('carry') *sortable* keys, such that the keys are sorted with respect to inorder.

Suppose that the node has the following description

```
typedef struct BST_NODE {
  struct BST_NODE *lchild, *rchild, *parent;
  SORTABLE key;
}
```

For example



To search for a key x

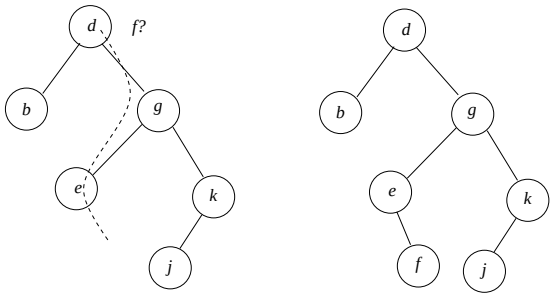
- Start with pointer p pointing to the root.
- If p is not null and x equals the key, return p.
- If p is not null and x is less than the key, replace p by its left child.
- If p is not null and x is greater than the key, replace p by its right child.
- If p is null, return null.

This closely resembles binary search. The difference is that binary search requires a sorted table of keys, whereas keys can be added and deleted much more easily in binary trees.

To add a key x,

- First search for x.
- If x is not found, then either the tree was empty, or the search reached a node z and went to a null left child or a null right child. Create a new node with key x and make it left or right child of z as appropriate.

For example, searching for f will 'fall off' e by going to the right child. Create a node with key f and attach it as right child of e.



To delete a key k**.** Search the tree for k; assuming it is found, at a node v, there are several cases:

The case where v has two children we shall deal with last. Given that v has at most one child,

Case: v is the root.

- If v has no children, make the root null; the tree is now empty.
- If v has a left or right child u, make u the root and make its parent null.

Case: v is the left child of another node x.

- If v has no children, make the left child of x null.
- If v has a left or right child u, make it the left child of x and make x its parent.

Case: v is the right child of another node x. Similar.

This leaves the case of where v has two children. There is a simple trick. Let w be the inorder successor of v. Then w has no left children. Perform the operation of removing w from the tree, but then re-install w by copying the links from v so that w occupies the place in the tree previously occupied by v.

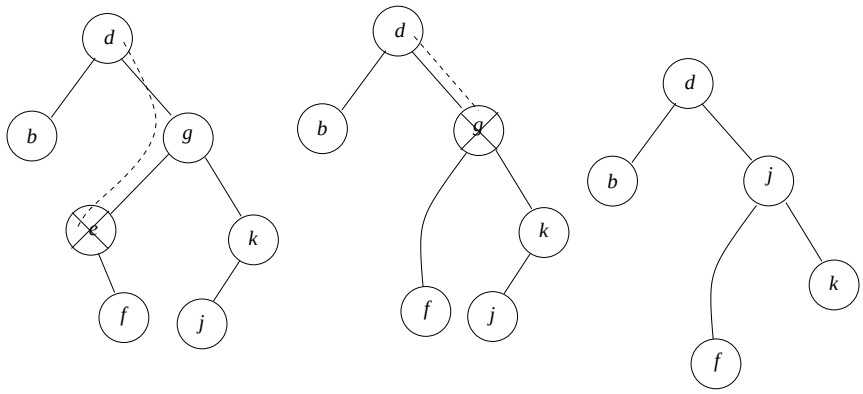


Figure 1: Deleting e, and then g.

Cost of building a binary search tree by repeated key insertion with n keys. If, for example, the keys were added in ascending order, then the tree will be a chain of right links and the total cost of all insertions is $\Omega(n^2)$.

But on average the performance is much better. The cost of inserting a key is proportional to its depth after insertion, and the total cost is proportional to the total depth of all nodes after all insertions.

We assume 'random' key input orders. In particular, for $0 \le i \le n-1$, the root of the tree — the first key inserted — has the value i with uniform probability $1/n$.

Let $T(n)$ be the average node depth in a tree with n nodes. Suppose that a particular 'random' n-node tree has left and right subtrees with i and j nodes respectively, where $i+j = n-1$ (the root is counted separately). The depth of a node in a left or right subtree is 1 less than its depth in the tree. Therefore

$$T(n) = n - 1 + \frac{1}{n}\sum_{0}^{n-1}(T(i) + T(n-1-i)).$$

As i goes from 0 to $n-1$, $n-1-i$ goes from $n-1$ to 0. Therefore

$$T(n) = n - 1 + \frac{2}{n}\sum_{0}^{n-1} T(i).$$

Some algebra:

$$nT(n) = n(n-1) + 2\sum_{0}^{n-1} T(i)$$

$$(n+1)T(n+1) = (n+1)n + 2\sum_{0}^{n} T(i)$$

subtract first from second

$$(n+1)T(n+1) - nT(n) = 2n + 2T(n)$$

$$(n+1)T(n+1) - (n+2)T(n) = 2n$$

divide by $(n+1)(n+2)$

$$U_{n+1} - U_n = \frac{2n}{(n+1)(n+2)}$$

$$= \frac{2n+4}{(n+1)(n+2)} - \frac{4}{(n+1)(n+2)} = \frac{2}{n+1} - \frac{4}{(n+1)(n+2)}$$

$$\text{where} \quad U_n = \frac{T(n)}{n+1}.$$

From this, U_n can be represented as a sum. To cut a long story short, the sum of $2/(n+1)$ is very close to $\log_e n$ and the sum of $4/((n+1)(n+2))$ converges. Hence U_n is $O(\log n)$ and $T(n)$ is $O(n \log n)$.

8.1 Average performance with deletions

The average pathlength of a binary tree is $O(n\sqrt{n})$; assuming all trees equally likely.

The average $O(n \log n)$ cost of n insertions does not account for deletions, and for deletions the average cost can be like $n\sqrt{n}$. I think Guibas, Andrew Yao, possibly Szemerédi, and

Culberson worked on this. Michaela Heyer, in UCC, gave a different deletion strategy which does achieve $O(n \log n)$ average cost for insertions and deletions.