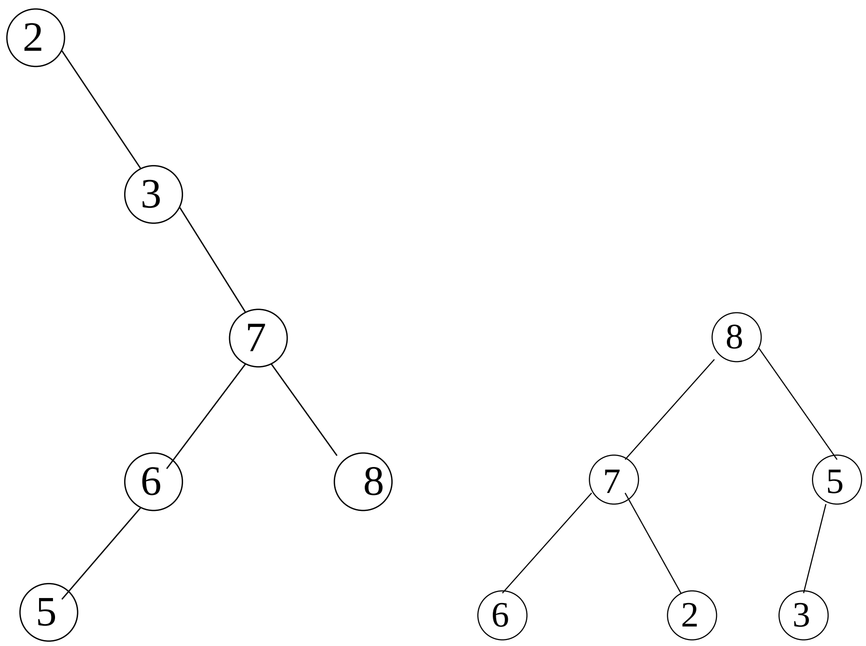
Binary Search Trees

A binary *search* tree labels each node in a binary tree with a single key such that for any node x, and nodes in the left subtree of x have keys $\leq x$ and all nodes in the right subtree of x have key's $\geq x$.



Left: A binary search tree. Right: A heap but not a binary search tree.

The search tree labeling enables us to find where any key is. Start at the root - if that is not the one we want, search either left or right depending upon whether what we want is $\leq$ or $\geq$ then the root.

Searching in a Binary Tree

Dictionary search operations are easy in binary trees ...

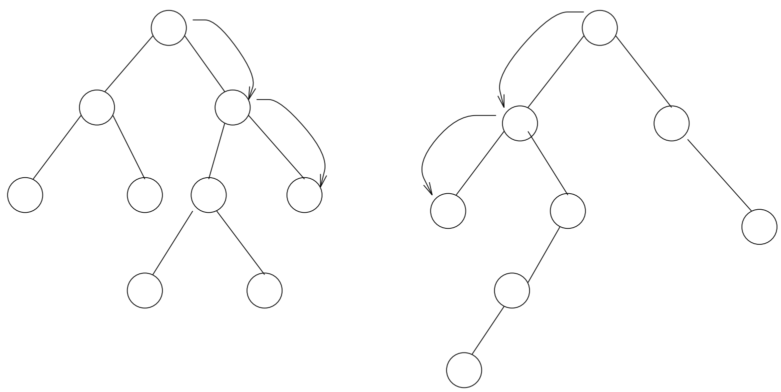
TREE-SEARCH(x, k)
    if $(x = NIL)$ OR $(k = key[x])$
        then return x
    if $(k < key[x])$
        then return TREE-SEARCH(left[x],k)
        else return TREE-SEARCH(right[x],k)

The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees – recursive structure, recursive algorithm.

This takes time proportional to the height of the tree, $O(h)$.

Maximum and Minimum

Where are the maximum and minimum elements in a binary tree?



TREE-MAXIMUM(X)

   while $right[x] \neq NIL$

      do x = right[x]

   return x

TREE-MINIMUM(x)

   while $left[x] \neq NIL$

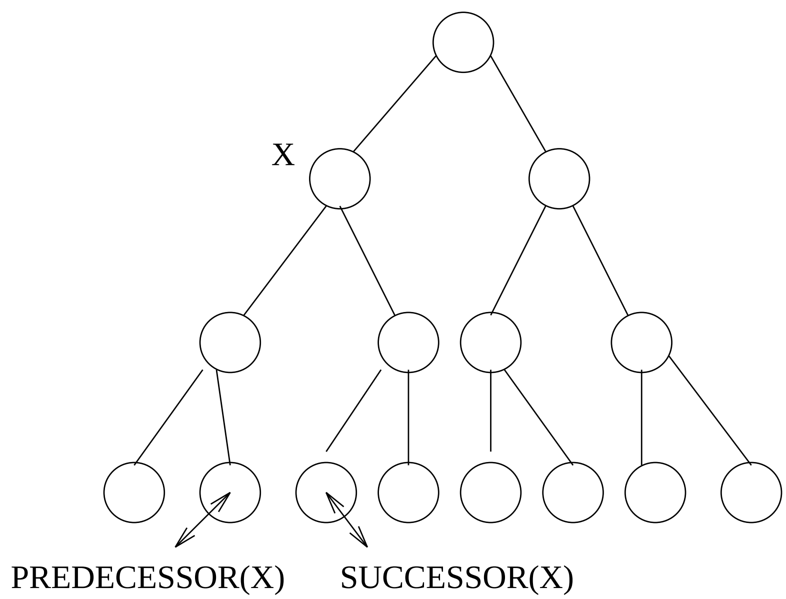
      do x = left[x]

   return x

Both take time proportional to the height of the tree, $O(h)$.

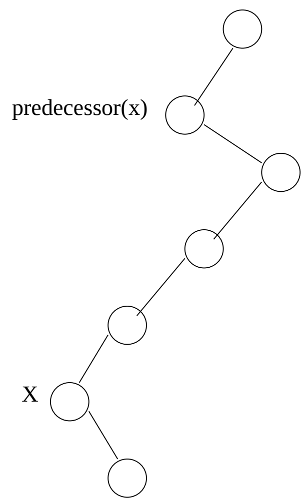
Where is the predecessor?

Where is the predecessor of a node in a tree, assuming all keys are distinct?



If X has two children, its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

What if a node doesn't have children?



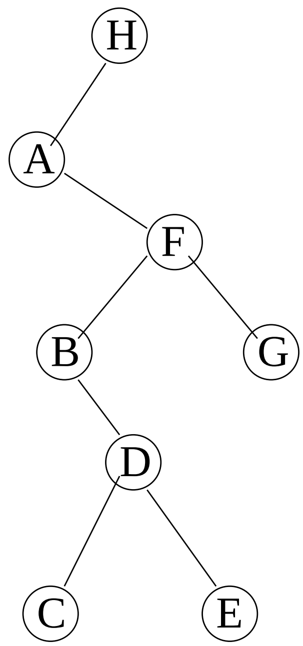
If it does not have a left child, a node's predecessor is its first left ancestor.

The proof of correctness comes from looking at the in-order traversal of the tree.

Tree-Successor(x)
    if $right[x] \neq NIL$
        then return Tree-Minimum($right[x]$)
        $y \leftarrow p[x]$
    while ($y \neq NIL$) and ($x = right[y]$)
        do $x \leftarrow y$
        $y \leftarrow p[y]$
    return y

Tree predecessor/successor both run in time proportional to the height of the tree.

In-Order Traversal

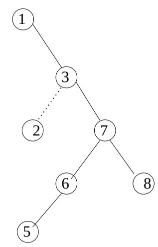


Inorder-Tree-walk(x)
    if ($x <> NIL$)
    then Inorder-Tree-Walk($left[x]$)
        print $key[x]$
        Inorder-Tree-walk($right[x]$)

A-B-C-D-E-F-G-H

Tree Insertion

Do a binary search to find where it should be, then replace the termination NIL pointer with the new item.



Tree-insert(T, z)

$y =$ NIL
$x = root[T]$
while $x \neq NIL$
  do $y = x$
    if $key[z] < key[x]$
      then $x = left[x]$
      else $x = right[x]$
$p[z] \leftarrow y$
if $y =$ NIL
  then $root[T] \leftarrow z$
  else if $key[z] < key[y]$
    then $left[y] \leftarrow z$
    else $right[y] \leftarrow z$

y is maintained as the parent of x, since x eventually becomes NIL.

The final test establishes whether the NIL was a left or right turn from y.

Insertion takes time proportional to the height of the tree, $O(h)$.

Tree Deletion

Deletion is somewhat more tricky than insertion, because the node to die may not be a leaf, and thus effect other nodes.

Case (a), where the node is a leaf, is simple - just NIL out the parents child pointer.

Case (b), where a node has one chld, the doomed node can just be cut out.

Case (c), relabel the node as its successor (which has at most one child when z has two children!) and delete the successor!

This implementation of deletion assumes parent pointers to make the code nicer, but if you had to save space they could be dispensed with by keeping the pointers on the search path stored in a stack.

Tree-Delete(T, z)
    if ($left[z] = NIL$) or ($right[z] = NIL$)
        then $y \leftarrow z$
        else $y \leftarrow$ Tree-Successor(z)
    if $left[y] \neq NIL$
        then $x \leftarrow left[y]$
        else $x \leftarrow right[y]$
    if $x \neq NIL$
        then $p[x] \leftarrow p[y]$
    if $p[y] = NIL$
        then $root[T] \leftarrow x$
        else if ($y = left[p[y]]$)
            then $left[p[y]] \leftarrow x$

```
            else $right[p[y]] \leftarrow x$
    if ($y <> z$)
        then $key[z] \leftarrow key[y]$
            /* If $y$ has other fields, copy them, too. */
    return $y$
```

Lines 1-3 determine which node y is physically removed.

Lines 4-6 identify x as the non-nil decendant, if any.

Lines 7-8 give x a new parent.

Lines 9-10 modify the root node, if necessary

Lines 11-13 reattach the subtree, if necessary.

Lines 14-16 if the removed node is deleted, copy.

Conclusion: deletion takes time proportional to the height of the tree.

Balanced Search Trees

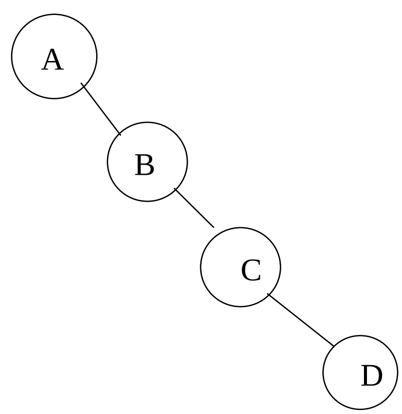
All six of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.

The best height we could hope to get is $\lg n$, if the tree was perfectly balanced, since

$\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$

But if we get unlucky with our order of insertion or deletion, we could get linear height!

insert(a)
insert(b)
insert(c)
insert(d)
$\vdots$

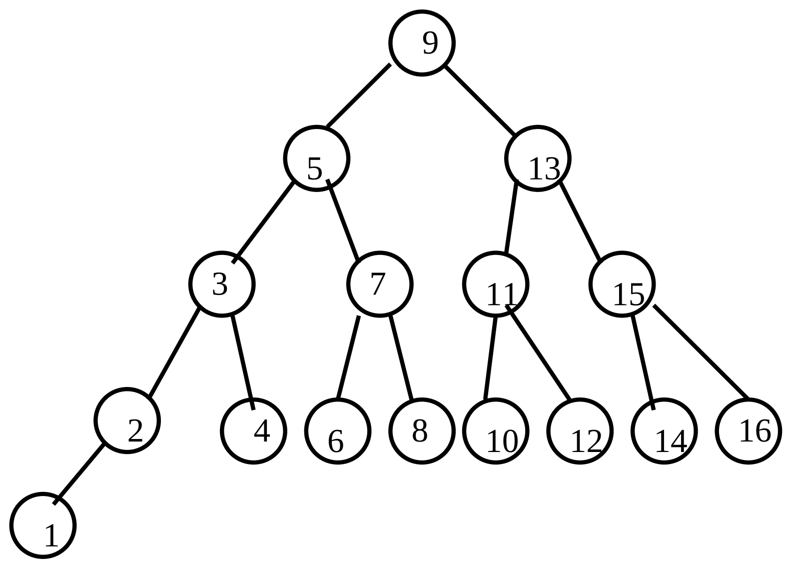


In fact, random search trees on average have $\Theta(\lg N)$ height, but we are worried about worst case height.

We can't easily use randomization - Why?

Perfectly Balanced Trees

Perfectly balanced trees require a lot of work to maintain:



If we insert the key 1, we must move every single node in the tree to rebalance it, taking $\Theta(n)$ time.

Therefore, when we talk about "balanced" trees, we mean trees whose height is $O(\lg n)$, so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take $O(\lg n)$ time.

Red-Black trees are binary search trees where each node is assigned a color, where the coloring scheme helps us maintain the height as $\Theta(\lg n)$.