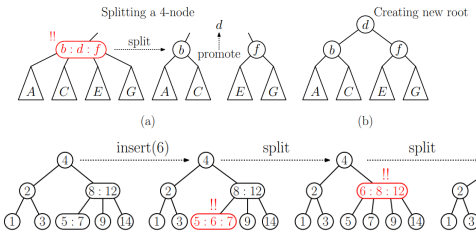
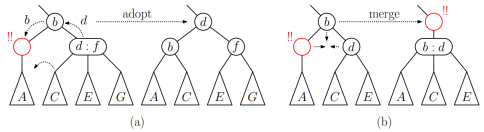
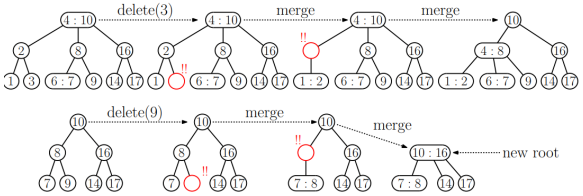
2-3 Tree

All leaves on same level



Deletion





Treaps

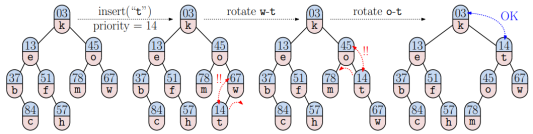
If keys are inserted in random order, expected height of BST O(logn).

Treap behave as if keys are inserted in random order

Timestamp increases from root to leaf. Assign random priority to keys

Treap has O(logn) expected height over all n! orderings

Insertion: BST insert, rotate left/right if parent priority is bigger



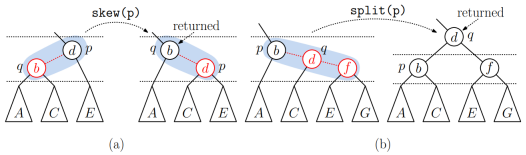
Deletion: Set node priority to ∞, rotate to leaf, delete

With standard binary search trees, the expectation was over all n! insertion orders. With treaps, the expectation was over all n! orders of the priority values. The latter is preferred, because the data structure's expected performance is not dependent on the insertion order

If just two keys have the same priority, their parent/child relationship might be affected, but the structure will be fine.

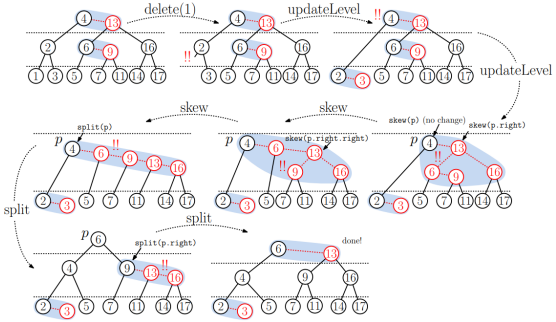
Red-Black/AA Trees

Root black, null black, red node = 2 black children, every path from node to null descendant contains same of black nodes (since all leaf node of 2-3 tree are same depth)
AA: Red only as right child of black



Insertion: split(skew(p))
UpdateLevel(): pulls p and p.right down to 1 + min level of children
Delete(): Standard BST except return fixAfterDelete(p)

```
fixAfterDelete(p) {
    updateLevel(p); // update p's level
    p = skew(p); // skew p
    p.right = skew(p.right); // ...and p's right child
    p.right.right = skew(p.right.right); // p's right-right grandchild
    p = split(p); // split p
    p.right = split(p.right); // ...and p's (new) right child
    return p;
}
```



Skip List

```
Value find(Key x)
    int i = topmostLevel // start at topmost nonempty level
    SkipNode p = head // start at head node
    while (i >= 0) { // while levels remain
        if (p.next[i].key <= x) p = p.next[i]
        else i-- // drop down a level
    }
    return (p.key == x ? p.value : null)
```

For any $c \geq 1$, the probability that the number of levels exceeds $clg(n)$ is at most $\frac{1}{n^{c-1}}$

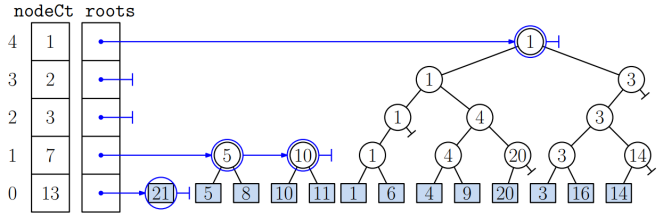
The expected storage is $O(n)$ (2n precise geometric series)

$E(i)$ = expected nodes visited at top i levels $E(i) = 1 + pE(i) + (1-p)E(i)$

$(1-p)E(i) = 1 + (1-p)E(i-1) \rightarrow E(i) = \frac{1}{1-p}E(i-1)$

Quake Heaps

Quake Heap:
- Collection of binary trees
- Nodes organized in levels
- All entries are leaves at level 0
- Internal nodes have 1 or 2 children
- Parent stores smaller of child keys



decreaseKey: Follow and change values on left path to highest ancestor, cut at end
Extract min does a lot of work → height too high, at some point we quake → height comes down we get a lot of roots, merge-trees → makes height O(logn) also removes roots which decreases potential

```
Key extract-min() { // extract the minimum key
    Node u = find-root-with-smallest-key() // find the min root
    Key result = u.key // final return result
    delete-left-path(u) // delete entire left path
    remove u from roots[u.level] // remove u as a root
    merge-trees() // merge tree pairs
    quake() // perform the quake operation
    return result
}
void delete-left-path(Node u) { // delete left path to leaf
    while (u != null) { // repeat all the way down
        cut(u) // cut off u's right child
        nodeCt[u.level] -= 1 // one less node on this level
        u = u.left // go to the left child
    }
}
void merge-trees() { // merge trees bottom-up in pairs
    for (int lev = 0; lev < nLevels-1; lev++) { // process bottom-up
        while (roots[lev] size is >= 2) { // at least two trees?
            Node u = remove any from roots[lev] // remove any two
            Node v = remove any from roots[lev]
            Node w = link(u, v) // ... and merge them
            make-root(w) // ... and make this a root
        }
    }
}
void quake() { // flatten if needed
    for (lev = 0; lev < nLevels-1; lev++) { // process bottom-up
        if (nodeCt[lev+1] > 0.75 * nodeCt[lev]) // too many?
            clear-all-above-level(lev)
    }
}
```

Trees

Binary tree with n nodes has $n+1$ null child links
Extended Binary Tree: null nodes replaced with external nodes has 2n+1 nodes in total (using above)
Complete Binary Tree: Every level filled except leaf (filled l to r)
Has between 2^h and 2^{h+1} nodes
Inorder traversal Using Threads

```
void inOrderTraversal(struct Node* root)
    struct Node* cur = leftMost(root);
    while (cur != NULL) {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right
             // subtree
            cur = leftmost(cur->right);
```

Tree with n nodes has $n-1$ edges

```
Node23 levelSuccessor(Node23 p) {
    if (p == null) return null;
    else if (rightSibling(p) != null)
        return rightSibling(p);
    else {
        q = levelSuccessor(p.parent)
        if (q == null) return null
        else return q.child[0]
    }
}
```

$2 * 2^{d-1} = 2^d$

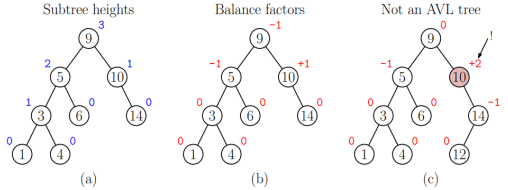
```
Node inorderSuccessor(Node p) {
    if (p.right == null) { // p has no right subtree
        Node q = p.parent
        while (q != null && p == q.right) { //r-child chain
            p = q; q = q.parent
        }
    } else { // p has a right subtree
        Node q = p.right
        while (q.left != null) { // find its leftmost node
            q = q.left
        }
    }
    return q
}

Node inorderSuccessor(Node p) {
    if (p.right == null) { // p has no right subtree
        Node q = p.parent
        while (q != null && p == q.right) //r-child chain
            p = q; q = q.parent
    } else { // p has a right subtree
        Node q = p.right
        while (q.left != null) { // find its leftmost node
            q = q.left
    return q
}
```

Trees cont.

```
Node preorderSuccessor(Node n) {
    if (n.left != null)
        return n.left;
    if (n.right != null)
          return n.right;
    Node curr = n, parent = curr.parent;
    while (parent != null && parent.right == curr) {
        curr = curr.parent;
        parent = parent.parent;
    }
    if (parent == null)
        return null;
    return parent.right;
}
```

AVL Trees

A AVL tree of height h has at least $F_{h+3} - 1$ nodes



nentially. In particular, $F_h \approx \varphi^h/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2 \approx 1.618$ is the famous *Golden Ratio.*[2]

Lemma: An AVL tree of height $h \geq 0$ has $\Omega(\varphi^h)$ nodes, where $\varphi = (1+\sqrt{5})/2$.

Proof: For $h \geq 0$, let $N(h)$ denote the minimum possible number of nodes in binary tree of height h that satisfies the AVL balance condition. We will prove that $N(h) = F_{h+3} - 1$ (see Fig. 2). The result will then follow from the fact that $F_{h+3} \approx \varphi^{h+3}/\sqrt{5}$, which is equal to φ^h up to constant factors (since φ itself is a constant).

h	0	1	2	3	4	5
F_{h+3}	2	3	5	8	13	21
$N(h)$	1	2	4	7	12	20

```
AvlNode rebalance(AvlNode p) {
    if (balanceFactor(p) < -1) { // left heavy?
        if (height(p.left.left) >= height(p.left.right)) {
            p = rotateRight(p);
        else // left-right heavy?
            p = rotateLeftRight(p);
    } else if (balanceFactor(p) > +1) { // right heavy?
        if (height(p.right.right) >= height(p.right.left)) {
            p = rotateLeft(p);
        else // right-left heavy?
            p = rotateRightLeft(p);
    }
    updateHeight(p); // update p's height
    return p; // return link to updated subtree
}
```

AVL tree of height h is formed by 1 subtree of height $h-1$ and another of height $h-1$ or $h-2$. And so both subtrees are full to height $\lfloor \frac{h-2}{2} \rfloor$.

BST

Random deletions and insertions will result in height of $O(\sqrt{n})$

This can be remedy by choosing replacement node at random
To analyze X_i, let's just focus on the first i elements and ignore the rest. Since every permutation of the numbers is equally likely, the minimum among the first i is equally likely to come at any of the positions first, second, ..., up to ith. The minimum changes only if comes last out of the first i. Thus, $\Pr(X_i = 1) = \frac{1}{i}$ and $\Pr(X_i = 0) = 1 - \frac{1}{i}$. Whenever this random event occurs ($X_i = 1$), the minimum has changed one more time. Therefore, to obtain the expected number of times that the minimum changes, we just need to sum the probabilities that $X_i = 1$, for $i = 2, \ldots, n$. Thus we have

$$D(n) = \sum_{i=2}^{n} \frac{1}{i} = \left(\sum_{i=1}^{n} \frac{1}{i}\right) - 1.$$

This summation is among the most famous in mathematics. It is called the *Harmonic Series*. Unlike the geometric series ($1/2^i$), the Harmonic Series does not converge. But it is known that when n is large, its value is very close to $\ln n$, the natural log of n. (In fact, it is not more than $1 + \ln n$.)
Therefore, we conclude that the expected depth of the leftmost node in a binary search tree under n random insertions is at most $1 + \ln n = O(\log n)$, as desired.

Misc

$n_L \leq \alpha^L n \iff \frac{1}{\alpha^L} \leq n \iff L \leq log_{1/\alpha} n = \frac{lg(n)}{lg(1/\alpha)}$
$2 = n_L \leq n^{1/2^L} \iff lg2 \leq \frac{1}{2^L} lg(n)$

Height = level - 1 $n_{min} = \sum_{i=0}^{l-1} 2^i = 2^l - 1 \iff lg(n_{min}+1)$
$n_{max} = \sum_{i=0}^{l-1} 3^i = \frac{3^l-1}{2} \iff log_3(2n_{max}+1)$