Balance factor: 
\[ \text{bal}(v) = \text{hgt}(v \text{.right}) - \text{hgt}(v \text{.left}) \]

AVL Height Balance
- for each node \( v \), the heights of its subtrees differ by \( \leq 1 \).

AVL tree: A binary search tree that satisfies this condition.

AVL Trees I
- Basic defs
- Height props
- Rotations

Theorem: An AVL tree of height \( h \) has at least \( F_{h+3} - 1 \) nodes.

Proof: (Induct. on \( h \))
- \( h = 0 \): \( n(h) = 1 = F_3 - 1 \)
- \( h = 1 \): \( n(h) = 2 = F_4 - 1 \)
- \( h \geq 2 \):
  \[ n(h) = 1 + n(h-1) + n(h-2) = 1 + (F_{h-2} - 1) + (F_{h-3} - 1) = F_{h+2} - 1 \]

Corollary: An AVL tree with \( n \) nodes has height \( O(\log n) \).

Proof: Fact: \( F_n \approx \phi^n / \sqrt{5} \) where \( \phi = (1 + \sqrt{5}) / 2 \) "Golden ratio" 

\[ n \geq \phi^h \Rightarrow \log n \geq \log \phi^h = h \log \phi \]

\[ h \leq \log_n n / \log_\phi \phi = O(\log n) \]

Conjecture: Min no. of nodes in AVL tree of height \( h \) is \( F_{h+3} - 1 \)

Recall: \( F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \)

Does this imply \( O(\log n) \) height?

Worst cases:
- height: \( h = 0, 1, 2, 3, 4, 5, 8, 13, 21, ... \)
- nodes: \( n(h) = 1, 2, 3, 5, 8, 13, 21, ... \)

How to maintain the AVL property?
- Right rotation
- Left rotation

AVL Node rotateRight(BSTNode p) 
\[ \text{BSTNode } q = p \text{.left} \]
\[ p \text{.left} = q \text{.right} \]
\[ q \text{.right} = \text{rotateRight}(r \text{.right}) \]

Worst cases:
- height: \( h = 0, 1, 2, 3, 4, 5, 8, 13, 21, ... \)
- nodes: \( n(h) = 1, 2, 3, 5, 8, 13, 21, ... \)

Theorem: An AVL tree of height \( h \) has at least \( F_{h+3} - 1 \) nodes.

Proof: (Induct. on \( h \))
- \( h = 0 \): \( n(h) = 1 = F_3 - 1 \)
- \( h = 1 \): \( n(h) = 2 = F_4 - 1 \)
- \( h \geq 2 \):
  \[ n(h) = 1 + n(h-1) + n(h-2) = 1 + (F_{h-2} - 1) + (F_{h-3} - 1) = F_{h+2} - 1 \]

Corollary: An AVL tree with \( n \) nodes has height \( O(\log n) \).

Proof: Fact: \( F_n \approx \phi^n / \sqrt{5} \) where \( \phi = (1 + \sqrt{5}) / 2 \) "Golden ratio" 

\[ n \geq \phi^h \Rightarrow \log n \geq \log \phi^h = h \log \phi \]

\[ h \leq \log_n n / \log_\phi \phi = O(\log n) \]
AVL Node: Same as BSTNode (from Lect 4) but add: int height

Utilities:

int height (AVLNode p)
return \{p == null \rightarrow -1
\phantom{=} \text{else} \rightarrow p.height\}

void updateHeight (AVLNode p)
p.height = 1 + \max (height (p.left), height (p.right))

int balanceFactor (AVLNode p)
return height (p.right) - height (p.left)

Find: Same as BST.
Insert: Same as BST but as we "back out" rebalance

How to rebalance?Bal = -2

Left-right heavy:

- Double rotations:
  - left-right
  - right-left

AVL Trees II
- double rotations
- insertion

AVLNode rebalance (AVLNode p)
if (p == null) return p
if (balanceFactor (p) < -1)
  if (ht (p.left.left) > ht (p.left.right))
    p = rotateRight (p)
  else p = rotateLeftRight (p)
else if (balanceFactor (p) > +1)
  \[ \text{[symmetrical]} \]
  updateHeight (p); return p

AVLNode insert (Key x, Value v, AVLNode p)\{if (p == null) p = new AVLNode (x, v)
else if (x < p.key)
  p.left = insert (x, v, p.left)
else if (x > p.key)
  p.right = insert (x, v, p.right)
else throw - Error - Duplicate! \}
return rebalance (p)
Announcements - 02/21

- Programming Assignment 1
  → Due Wed of next week 11:59pm (3/1)
  → Handout, skeleton, + test data available

- HW 2 - Coming soon
  → Due Tue, Mar 7

- Midterm 1 - Thu, Mar 9, in class
Problem 1. (10 points) This question involves the rooted trees shown in Fig. 1.

(a) (3 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the first-child/next-sibling form.

(b) (2 points) List the nodes Fig. 1(a) in preorder.

(c) (2 points) List the nodes Fig. 1(a) in postorder.

(d) (3 points) Consider the rooted tree of Fig. 1(b) given in its first-child/next-sibling form. Draw a figure in its standard form.

Solution 1:

(a) See Fig. 2(a)

(b) Preorder: (b, g, e, h, j, c, d, i, f, k, a, m)

(c) Postorder: (h, j, e, g, c, f, i, k, a, m, f, d, b)

(d) See Fig. 2(b)
Problem 2. (10 points) Consider the union-find trees shown in Fig. 3. The rank values for each tree are indicated in blue next to each root.

(a) (3 points) Show the array of parent indices for this set of trees and indicate which elements of this array are set identifiers. (See Fig. 1(b) of the lecture notes for Lecture 4 as an example of what we are looking for.)

(b) (5 points) Show results after of performing the operations union(4,10) and union(16,9). (For the sake of uniformity, use the same convention given in the lecture notes. When performing union(a,t), if both trees have the same rank, link a as a subtree of t.)

(c) (2 points) Show the result of performing the operation find(5) on the data structure after the union operations of part (b). In particular, indicate the resulting set identifier returned by the find operation and show the final tree that results after path compression is applied.

Solution 2:

(a) See Fig. 4 (bottom left).

(b) See Fig. 4. Since trees 4 and 10 have the same ranks, we follow the convention of the code given in class and link the first tree under the second, and the rank of the resulting tree is higher by 1. Since 9 is of lower rank compared to 16, so we link 9 as a child of 16, and the rank of 16 does not change.

(c) See Fig. 4. The path from 5 leads through 7, 4, before arriving at the root 10. We compress the path by linking 5 and 7 directly to 10. The rank of the tree does not change (even though its height decreases.) The final answer is 10.
Problem 3. (10 points) Consider the two leftist heaps, $H_1$ and $H_2$, shown in Fig. 5.

Figure 5: Leftist Heaps.

(a) (3 points) We have labeled each node of $H_1$ with its npl value. Redraw $H_2$ indicating the npl values for each node.

(b) (7 points) Show the result of merging these two heaps together. Indicate the npl values for each node. (You need only show the final tree, but the intermediate tree may be given for partial credit.)

Solution 3:

(a) The NPL values were already shown for heap $H_1$. For heap $H_2$, see Fig. 6.

Figure 6: NPL values in a leftist heap.

(b) We’ll follow the intuitive approach given in the lecture notes, rather than tracing the formal algorithm. We begin by merging along their rightmost paths (see Fig. 7(a)). Next, we update the NPL values, but only node 6 changes its NPL value, which is now 2. Finally, we check whether the nodes on rightmost path satisfy the leftist property by comparing the NPLs of their children. Nodes 2, 3, and 14 violate this (right NPL exceeds left NPL), and we swap their left and right subtrees (see Fig. 7(b)).

Figure 7: Merging two leftist heaps.
Problem 4. (10 points) You are asked to implement a function `promote(Node v)`. This function is given a reference to a node `v`, which is one of the children of the root node. Let `r` denote the root. This function makes `v` the new root of the tree by (1) removing `v` as child of `r` and (2) making `r` the first child of `v` (see Fig. 8). All the other children of `v` and all the other children of `r` remain unchanged. If we think of the tree as an undirected graph, the graph structure is unchanged. We just have a different node as the root.

Figure 8: Promoting a child of the root to be the new root.

Present pseudocode for this function. Briefly explain how it works. For full credit, your algorithm should run in time proportional to the number of children the root has. You do not need to perform any error checking. You may assume that `v` is indeed one of the children of the root node.

Solution 4: The process consists of the following steps. First, we locate `v` node by searching through the children of the root (see Fig. 9(a)).

Next, we unlink `v` from the tree. There are two cases depending on whether it is the first child of the root or one of the other children. If it the first child of the root, we just advance the root’s first child link to point to its second child. Otherwise, we find the node `c` that immediately precedes `v` in the child list, and adjust `c`’s next-sibling link to skip over `v` (see Fig. 9(b)). Once `v` is removed, we insert the old root node as `v`’s first child, and copy `v`’s old first-child link to become the old root’s next-sibling link. Finally, we update the root pointer to point to `v` (see Fig. 9(c)). The pseudocode is presented below.

```java
void promote(Node v) {
    Node c = root.firstChild;
    if (c == v) {
        root.firstChild = v.nextSibling // v is the first child of root
        remove v from root child list
    } else {
        while (c.nextSibling != v) c = c.nextSibling // find v’s predecessor sibling
        c.nextSibling = v.nextSibling // unlink v
        root.nextSibling = v.firstChild
        v.firstChild = root
        v.nextSibling = null
        root = v // link root into v’s child list
        make old root v’s first child
        v now has no sibling
        make v the new root
    }
}
```
Problem 5. (10 points)

Answer the following questions:

(a) (2 points) Suppose that we have just performed an expansion. Prior to the expansion we had an array of size \(m\), and our new array is of size \(m' = 3(n_L + n_U)\). As a function of \(m'\) (or if you prefer, \(n_L\) and \(n_U\)), what is the minimum number of operations needed until the next expansion occurs? (Briefly explain.)

(b) (2 points) As a function of \(m'\) (or if you prefer, \(n_L\) and \(n_U\)), what is the worst-case (maximum) cost for the next expansion? (Hint: This may depend on the relative sizes of the two stacks.)

(c) (6 points) Using parts (a) and (b), derive a constant \(\tau\) such that the amortized cost of our expanding dual stack is at most \(\tau\). (Hint: Note that the worst-cases for (a) and (b) may arise from different scenarios. For the sake of obtaining an upper bound on the amortized cost, it is okay to simply use the worst-case for each, without regard for whether they can both happen simultaneously. See Challenge Problem 2.)

Solution 5:

(a) As remarked in the problem description, when the expansion takes place, each of the arrays has \(n_L + n_U = m'/3\) available empty positions (see Fig. 11(a)).

We have just inserted a new element into one of these stacks, implying that this stack has \(m'/3 - 1\) available entries remaining (and the other has one more). The fastest we can induce the next overflow is to push elements into this stack (see Fig. 11(b)). This happens after \(m'/3\) pushes (\(m'/3 - 1\) to fill the stack, and one more to induce the overflow).

(b) Because the expansion cost depends on the number of entries that are copied, the worst-case occurs when both stacks are as full as possible, and we perform one additional push. The worst case occurs when we are using every element of the current array, implying that there are \(m'/3\) elements in the current stack, leading to an expansion cost of \(m'\) (see Fig. 11(c)).

Figure 11: Double stack analysis.
We will show that the amortized cost is at most $4$. Our proof will involve a token-based approach. To avoid begging the question of why $4$ is the correct answer, let’s start by assuming that the amortized cost is some value $\tau$, and we will show that $\tau = 4$ does the job. We will show that for any sequence of $m$ operations, the total cost, denoted $T(m)$, is at most $7m$.

To make the analysis simpler, we break the sequence of $m$ operations into runs. Each run starts just after the prior expansion and ends with the next expansion. We ignore the first and last runs, since they don’t follow the pattern, but they are easy to account for. (The first run involves only a constant number of operations, and the final run may not end in an expansion, which only makes the total cost smaller.)

Each time we perform an operation, we will use one token to pay for the operation, leaving $\tau - 1$ tokens to put in our bank account. In (a) we argued that there are at least $m'3$ operations until the next expansion, implying that we have banked at least $(\tau - 1)m'/3$ tokens by the end of the run. From (b), we know that the worst-case expansion cost is $m'$. In order to pay for this, it suffices to set $\tau$ large enough so that

\[(\tau - 1)\frac{m'}{3} \geq m' \quad \text{or equivalently} \quad \tau \geq \frac{m'3}{m'} + 1 = 4.
\]

Therefore, setting $\tau = 4$ is sufficient, which implies that the amortized cost is at most $4$. Is this tight? No. To see why, see Challenge Problem 2.
Challenge Problem 1: You have two friends, Alice and Bob. They have just implemented the merge function for leftist heaps, but they each made one mistake. For each of the following, indicate what the consequences are of their error.

(a) When computing the npl values for node, Alice consistently used the maximum of the npl values of the two children (rather than the minimum). That is, she defined

\[
\text{npl}(v) = \begin{cases} 
-1 & \text{if } v = \text{null}, \\
1 + \max(\text{npl}(v.\text{left}), \text{npl}(v.\right)) & \text{otherwise.}
\end{cases}
\]

Otherwise, her code is exactly the same as given in class. Which of the following could be said of Alice’s program. Select one and justify your answer.

(i) No difference at all. The tree structure is identical, the results are identical, and the running time to perform the merge operation is \(O(\log n)\).

(ii) The tree structure may differ, but it is still in valid heap order, and the running time to perform the merge operation is \(O(\log n)\).

(iii) The tree structure may differ, but it is still in valid heap order. However, the running time to perform the merge operation may be worse than \(O(\log n)\).

(iv) The tree structure may fail to be in heap order.

(b) Bob made a different mistake. Whenever he checks the npl values of two subtrees, he always puts the subtree with the larger npl value on the right (rather than on the left). Otherwise, his code is identical. (In particular, he merges trees along their rightmost path.) Repeat part (a), but with this variant. Again, justify your answer.

Solution to Challenge Problem 1:

(a) (iii): The results are correct, but the running time of merge may be not be \(O(\log n)\). Alice used “max” rather than “min” in her NPL computation. Notice that this is equivalent to using height rather than NPL to decide which subtree goes on the right. Unfortunately for Alice, having the lower height tree on the right side does not guarantee that the number of nodes in the rightmost chain is \(O(\log n)\). To see why, consider the example shown in Fig.12.

(b) (iii): The results are correct, but the running time of merge may be not be \(O(\log n)\). Bob’s mistake is more serious than Alice’s. If the right subtrees have higher NPL values, then in the worst case, the tree may degenerate to a chain of \(n\) nodes along the right side. In this case, the merge process will take \(\Omega(n)\) time. The valid heap order will still be maintained by the merging procedure.
Challenge Problem 2: It was noted in Problem 5(c) that the simple analysis based on the worst-case scenarios for parts (a) and (b) cannot both occur. Present a more accurate analysis of the amortized running time that corrects this shortcoming.

Solution to Challenge Problem 2: Recalling the analysis of Problem 5, we know that at the start of the run there are $m'/3$ elements in the stack and each stack has $m'/3$ available entries. We need at least $m'/3$ pushes to induce another expansion, but we may push as many as $2m'/3$ entries. To determine where the worst-case resides, let the actual number of pushes be $(1+\alpha)m'/3$, for $0 \leq \alpha \leq 1$. Let $\tau$ denote the amortized cost. As before, each cheap operation costs $1+\alpha$ unit, and the remaining $\tau-1$ are banked with each operation. The total number of banked tokens until the next rebuild is $(1+\alpha)m'/3$. The expansion involves copying these elements plus the original $m'/3$, for a total cost of $(1+\alpha)m'/3 + m'/3 = (2+\alpha)m'/3$. In order to pay for this, it suffices to set $\tau$ large enough so that

\[
(\tau-1)(1+\alpha)m'/3 \geq (2+\alpha)m'/3 \quad \text{or equivalently} \quad \tau \geq \frac{2+\alpha}{1+\alpha} + 1 = \left(1 + \frac{1}{1+\alpha}\right) + 1.
\]

This is maximized when $\alpha$ is minimized. Setting $\alpha = 0$, it follows that $\tau \geq (1+1) + 1 = 3$. Setting $\tau = 3$ satisfies this, implying that the amortized cost is at most $2m'/3$. 

\[
\text{How many?} \quad \frac{m'}{3} + \alpha \frac{m'}{3} = (2+\alpha) \frac{m'}{3}
\]