Solutions to Homework 2: Search Trees

Solution 1:

(a) We first traverse the search path and insert 5 as the left child of 8 (see Fig. 1). As we return along the recursion path, we update heights and compute balance factors. On reaching 4, we see that the balance factor is +2. Since it is right-left heavy, we do an RL double rotation at 4, which brings 8 up and 4 and 10 are its children. We continue up to the root.

(b) We first locate 10 and delete this node (see Fig. 2). We then return along the recursion path, and update the heights and recompute balance factors.

Figure 1: AVL-tree insertion.

Figure 2: AVL-tree deletion.
On reaching 11 we find that the balance factor is +2. Since it is right-left heavy, we perform a RL double rotation at 11, which brings 12 up and 11 and 13 are its children. We continue up to the root, where we find the balance factor is +2. Since it is left-right heavy, we perform an LR double rotation at 9, which brings the 6 up, and 3 and 9 are its children.

Solution 2:

(a) See Fig. 3. The new node containing 1 is inserted as the left child of 2. The operation \texttt{skew(2)} results in a right rotation at 2, which creates the right chain with \((1, 2, 4), \) and 1 becomes the current node. The operation \texttt{split(1)} performs a left rotation at 1 and promotes 2 to level two, where it becomes the current node. We return to 7, which performs \texttt{skew(7)} resulting in a right rotation at 7, which creates the right chain with \((2, 7, 12), \) and 2 becomes the current node. The operation \texttt{split(2)} performs a left rotation at 2 and promotes 7 to level 3, where it becomes the current node. We return to 17, which performs \texttt{skew(17)} resulting in a right rotation at 17, bringing 7 up to be the new root and current node. This is the final tree.

![](image)

(b) See Fig. 4. We delete 5, and return to 3. The operation \texttt{update-level(3)} causes 3 to be demoted to level 1. We then perform \texttt{skew(3)}, which performs a right rotation and makes 1 the current node. We return to 6. The operation \texttt{update-level(6)} causes 6 and its child 15 to be demoted to level 2, where we have a massive cluster of 6 nodes \((6, 9, 12, 15, 18, 22).\) The current node is 6, which we denote by \(p.\) The operation \texttt{skew(p)} does nothing, but \texttt{skew(p.right)} results in a right rotation at 15 (bringing up the 9). Now, \(p.\) points to 9, and the operation \texttt{skew(p.right.right)} results in another right rotation at 15 (bringing up the 12). We now have the right chain \((6, 9, 12, 15, 18, 22),\) and now 6 is the current node. Again, letting \(p\) refer to 6, the operation \texttt{split(p)} performs a left rotation at 6 and promotes 9 to level 3. This also makes 9 the new root, and 9 is now the current node and has 12 as its right child. The operation \texttt{split(p.right)} performs a left rotation 12 and promotes 15 up on level. This makes 15 the right child of 9. This is the final tree.

Solution 3:
Theorem: For any $h \geq 0$, if the nodes of $T_h$ are labeled according to their position in an inorder traversal of the tree (starting with 1), then the labels along the leftmost chain of tree (from leaf to root) generate the Fibonacci sequence

$$(F(2), F(3), F(4), F(5), \ldots, F(h+2)),$$

where $F(h)$ denotes the $h$th Fibonacci number.

Proof: The proof is by induction on $h$. The basis cases ($h = 0$ and $h = 1$) are trivial, since they leftmost chain form the sequences $\langle 1 \rangle$ and $\langle 1, 2 \rangle$, respectively.

To prove the induction step, let us assume that $h \geq 2$, and let us assume the induction hypothesis that for any $h' < h$, the leftmost chain of the tree $T_{h'}$ yields the Fibonacci sequence $\langle F(2), F(3), F(4), \ldots, F(h' + 2) \rangle$, and we will use this to prove the theorem for $h$ itself.

By definition, the left subtree of $T_h$ has $T_{h-1}$ as its left subtree. In an inorder traversal of $T_h$, the nodes of the left subtree will be labeled first. Since $h - 1 < h$, we may apply the induction hypothesis, which implies that the leftmost chain of $T_{h-1}$ forms the Fibonacci sequence $\langle F(2), F(3), F(4), \ldots, F(h+1) \rangle$. This subtree has height $h - 1$. Therefore, by the result proven in class, the number of nodes in this left subtree is $F((h-1)+3) - 1 = F(h+2) - 1$.

The very next node in the inorder traversal is the root, which must then be labeled with the next number, that is, $F(h+2)$. Appending this to the end of the previous sequence, we obtain the final sequence $\langle F(2), F(3), F(4), \ldots, F(h+1), F(h+2) \rangle$, as desired.

Solution 4: Before presenting the answer, it is useful to have a utility function that determines which child a node $p$ is (either 0, 1, or 2). If $p$ has no parent (that is, it is the root) then this function returns $-1$.

```c
int whichChild(Node p) {
    Node par = p.parent;
    if (par == null) return -1; // p is not a child of anyone
    else if (p == par.child[0]) return 0; // it must be one of three
    else if (p == par.child[1]) return 1
```
(a) We invoke the \texttt{whichChild} function, and return the previous child $i - 1$, provided that $i$ is 1 or larger.

```plaintext
Node leftSib(Node p) { // get p’s left sibling
    int i = whichChild(p)
    if (i < 1) return null // root or leftmost child
    return p.parent.child[i-1] // return previous child
}
```

(b) We invoke the \texttt{whichChild} function, and return the next child $i + 1$, provided that $i$ is neither $-1$ nor the last child.

```plaintext
Node rightSib(Node p) { // get p’s right sibling
    int i = whichChild(p)
    if (i < 0 || i >= p.parent.nChild-1) // root or last child
        return null
    else return p.parent.child[i+1] // return next child
}
```

(c) We use the above functions to determine which child \( p \) is and which are its two siblings. We first attempt a merge with the left sibling and otherwise we attempt a merge with the right sibling.

```plaintext
Node merge(Node p) {
    int i = whichChild(p)
    Node par = p.parent
    Node ls = leftSib(p)
    Node rs = rightSib(p)
    if (ls != null && ls.nChild == 2) { // can merge with left?
        Key x = p.parent.key[i-1] // key just before p
        return new Node(par, ls.child[0], ls.key[0], ls.child[1], x, p.child[0])
    } else if (rs != null && rs.nChild == 2) { // can merge with right?
        Key x = p.parent.key[i] // key just after p
        return new Node(par, p.child[0], x, rs.child[0], rs.key[0], rs.child[1])
    } else {
        return null
    }
}
```

\textbf{Solution 5:} Throughout this problem, we let $k = \sqrt{n}$.

(a) The final column $\langle 4, \ldots, 16 \rangle$ forms a right chain of nodes. Following this, each successive column is adds a new left child to each of the nodes from the previous column. See Fig. 5.

(b) Starting at the root, the longest path comes by following a path of length $k - 1$ to the rightmost node of the tree (corresponding to the node $n^2$) followed by a path of length $k - 1$ to its leftmost descendant. Thus, the tree’s height is $2(k - 1) = 2\sqrt{n} - 2$. 

4
Figure 5: Matrix-based insertion order.

(c) The number of nodes per level increases linearly with each level from 1 up to $\sqrt{n}$, and then it descends linearly back to 1. Since the first level is zero, we have

$$d(i) = \begin{cases} 
    i + 1 & \text{if } 0 \leq i \leq \sqrt{n} - 1, \\
    2\sqrt{n} - (i + 1) & \text{if } \sqrt{n} \leq i \leq 2\sqrt{n} - 2.
\end{cases}$$

If you want to avoid the cases, there is an elegant way to express this using absolute values as $d(i) = \sqrt{n} - |i - \sqrt{n} + 1|$.

(d) The total insertion time $T(n)$ is the sum of the product $(i + 1)d(i)$ for $i$ ranging from zero up to the height of the tree. We can split the sum between the increasing and decreasing parts, call then $I(n)$ and $D(n)$. Recalling that $k = \sqrt{n}$, we can express the increasing part as

$$I(n) = 1 \cdot 1 + 2 \cdot 2 + \ldots + k \cdot k = \sum_{i=1}^{k} i^2 = k^2 + \sum_{i=1}^{k-1} i^2.$$  

(We’ll see below why I separated out the final term.) The decreasing part can be expressed as follows:

$$D(n) = (k + 1)(k - 1) + (k + 2)(k - 2) + \ldots (2k - 1)(1).$$

Observe that this is a series of terms of the form $(k + a)(k - a)$, which can written as $(k^2 - a^2)$. Thus, we have

$$D(n) = (k^2 - 1) + (k^2 - 4) + \ldots (k^2 - (k - 1)^2) = \sum_{i=1}^{k-1} (k^2 - i^2) = \sum_{i=1}^{k-1} k^2 - \sum_{i=1}^{k-1} i^2.$$  

To get the total cost, we take $I(n) + D(n)$. Observe that $\sum_{i=1}^{k-1} i^2$ in $I(n)$ and $D(n)$ cancel each other out, so we obtain

$$T(n) = I(n) + D(n) = k^2 + \sum_{i=1}^{k-1} k^2 = k^2 + (k - 1)k^2 = k \cdot k^2 = k^3.$$  

Wow! All that work just to get $k^3$. So, in terms of $n$, the final cost is $T(n) = n^{3/2}$. Since there are $n$ insertions, we can express this more meaningfully as the amortized cost, which is $T(n)/n = \sqrt{n}$.  

5
Solution to Challenge Problem 1: There are a number of cases depending on which child $p$ is and which sibling we merge with. Suppose that $p$ is the $c$-th child of its parent. To encode the sibling we are merging with, we will let $s = -1$ denote its left sibling and $s = 0$ denote its right sibling. From the parent’s perspective, the merger now involves replacing the children at indices $c+s$ and $c+s+1$ with the newly constructed merged node, and removing key $c+s$. For example, if $p$ is the middle of three children ($c = 1$), and we are merging with its left sibling ($s = -1$), then in the parent, we replace children $c+s = 0$ and $c+s+1 = 1$ with the merged node, and we remove the key at index $c+s = 0$. On the other hand, if we merge with its right sibling, all these indices are increased by +1.

We do this with the following utility function:

```cpp
def replace(Node p, int c, int s) {  # merge utility
    Node q = p.parent
    q.child[c+s] = merge(p)
    for (int i = c+s; i < q.nChild-1; i++) {  # slide children down
        q.child[i] = q.child[i+1]  # ...to fill hole in child[c]
    }
    for (int j = c+s; j < q.nChild-2; j++) {  # slide keys down
        q.key[j] = q.key[j+1]  # ...to fill hole in key[k]
    }
    q.nChild -= 1
}

def merge(Node p) {
    # merge with left?
    if (ls != null && ls.nChild == 2) {
        Key x = p.parent.key[c]  # key just before p
        Node r = Node(ls.child[0], ls.key[0], ls.child[1], x, p.child[0])
        replace(p, r, c, -1)  # replace with r
    }
    # merge with right?
    if (rs != null && rs.nChild == 2) {
        Key x = p.parent.key[c+1]  # key just after p
        Node r = Node(p.child[0], x, rs.child[0], rs.key[0], rs.child[1])
        replace(p, r, c, 0)  # replace with r
    }
    return null
}
```

Solution to Challenge Problem 2: One approach is to employ a postorder traversal that does not use a stack or recursion (see, e.g., the Morris Traversal).

Our approach will be to perform repeated right rotations at the root, until it has no left child. Then we can delete the root and continue with its right child.

```cpp
void free(BinaryTree t) {
    Node v = t.root
    while (v is not null) {
        while (v has left child) {  // rotate right until left subtree is gone
            v = rotateRight(v)
        }
        // delete node v
        Node p = v.parent
        p.child[c] = null  # replace child[c] with v
        p.key[c] = v.key  # replace key[c] with v.key
        p.nChild -= 1  # decrease number of children
    }
}
```
Node u = v.right
delete(v) // deallocate the root
v = u // continue with its right child
}