Solutions to Homework 1: Basic Data Structures and Trees

Solution 1:  See Fig. 1(a) and (b).

Solution 2:  See Fig. 2.

Solution 3:  

(3.1) No, it is not possible to uniquely recover a tree from its postorder traversal. As a counterexample, consider the two trees shown Fig. 3(a). Both have the same postorder traversal \(\langle a, b, c, d, e \rangle\), but clearly the trees are different.

(3.2) Given the leaf indicators, it is possible to uniquely recover the tree from the postorder sequence. (If you have ever heard of postfix notation for arithmetic expressions, this is just a postfix parser.) Let’s give a procedure for doing this. We think of the postorder sequence
like a Java stream, and the functions `hasNext()` and `next()`. As we build subtrees, we will store them on a stack. Whenever we see a leaf, we push it on the stack as a trivial 1-node tree. Whenever we encounter a non-leaf symbol `x`, we pop the two top items off the top of the stack, call them `u` and `v`, and create a new subtree with root `x` and right and left subtrees `u` and `v`, respectively.

```java
class Tree {
    private int root;
    private Tree left, right;

    public Tree(int root) {
        this.root = root;
    }

    public void push(int value) {
        // Implementation...
    }
}
```

The proof of correctness is based on induction. For the basis case, we know each leaf is clearly its own 1-node subtree. If we see an internal node, then because the tree is full, we have already processed its two subtrees. By induction, we may assume that these subtrees have been correctly encoded and are hence on the stack. By the nature of a postorder traversal, they must be two top entries on the stack, with the right subtree at the top and the left subtree just beneath it. We then form the appropriate subtree and place it back on the stack.

(3.3) See the answer (3.4). If flagged inorder traversals are ambiguous, then clearly unflagged traversals are also ambiguous.

(3.4) No, it is not possible to uniquely recover a tree from its inorder traversal, even if the leaves are flagged. As a counterexample, consider the two trees shown Fig. 3(b). Both have the same inorder traversal `(a*, b, c*, d, e*)`, but clearly the trees are different.
(3.5) It was shown in class that a full binary tree with \( n \) internal nodes has \( n + 1 \) external (leaf) nodes. Therefore, the number of nodes in a full binary tree is always odd. Thus, your friend’s 6-node countexample cannot be a valid full binary tree.

Solution 4:

(4.1) The multiplication algorithm works as follows. For each pair \( i \) and \( j \), where \( 0 \leq i, j \leq n - 1 \), let \( A[i][\ast] \) denote \( A \)'s \( i \)th row and \( B[\ast][j] \) denote \( B \)'s \( j \)th column. We compute \( C[i][j] \) by taking the dot product of \( A[i][\ast] \) and \( B[\ast][j] \). We do this by iterating through each of these lists in parallel. When two corresponding entries are nonzero, we multiply them together to form the next entry of \( C \). Otherwise, we advance whichever is lagging behind the other. Observe that the entries of \( C \) are filled top to bottom and left to right, so this can be adapted if \( C \) is represented as a sparse matrix.

```c
void sparseMultiply(SparseMatrix A, SparseMatrix B, float C[][]) {
    // We assume that C is a standard n x n 2-dimensional array initialized to 0
    for (i = 0 to n-1)
        for (j = 0 to n-1)
            ap = A.row[i] // traverses ith row of A
            bp = B.col[j] // traverses jth column of B
            while (ap != null && bp != null) {
                if (ap.col == bp.row) // both are nonzero - include them
                    C[i][j] += ap.data * bp.data
                    ap = ap.rowNext // advance both
                    bp = bp.colNext
                else if (ap.col < bp.row) // A lags behind - advance
                    ap = ap.rowNext
                else /* bp.row < ap.col */ // B lags behind - advance
                    bp = bp.rowNext
            }
}
```

(4.2) Letting \( n \) denote the dimension of the matrices and letting \( N_A \) and \( N_B \) denote the number of nonzero entries in \( A \) and \( B \), respectively, we assert that the running time is \( O(n(N_A + N_B) + n^2) \). (We’ll also accept \( O(n(N_A + N_B)) \) for full credit since it is reasonable to assume that \( N_A + N_B \) is each at least \( \Omega(n) \).)

To see this, observe that the matrix \( C \) has \( n^2 \) entries. Each of these involves computing the dot product of a row of \( A \) with a column of \( B \). Each row of \( A \) is traversed once for each column of \( B \), and each column of \( B \) is traversed once for each row of \( A \). Therefore, each nonzero entry of \( A \) and \( B \) is visited \( n \) times, for a total time of \( O(n(N_A + N_B)) \). However, even if \( N_A = N_B = 0 \), the above algorithm spends at least \( O(n^2) \) time to consider each rows and column of the output matrix. This yields a total running time of \( O(n(N_A + N_B) + n^2) \).

Solution 5:

(5.1) See the left tree in Fig. 4, where heights are indicated to the left of each node and balance factors are indicated to the right.

(5.2) The key 1 is inserted as the left child of 2 (see Fig. 4). As we walk up the tree and update heights and balance factors, we discover that the balance factor at 5 is too small. (The
balance factors at 8 and 15 are now \(-1\) and \(-2\), respectively. We won’t bother to indicate them, because the algorithm will first fix node 5 before computing these balance factors.) Since 5 is left-left heavy, we perform a single right rotation here. This brings 2 up and pushes 5 down (see Fig. 4). Nodes 2 and 5 now both have a balance factors 0. Even though the algorithm will return to the root recomputing heights and balance factors, nothing will change.

(5.3) The key 13 will be inserted as the right child of 12 (see Fig. 5). We return back up the tree updating the balance factors. On arriving at the root, we see that its balance factor is \(-2\). Because node is left-right heavy (the insertion took place in the left-right grandchild), we perform a left-right rotation at 15 (see Fig. 5). This pulls the 11 node up to the root, and nodes 8 and 15 are its left and right children, respectively. We update the balance factors for these three nodes.

\[ \text{Solution 6: } \text{Whenever a change is made to a node’s left or right child (e.g., } p.\text{left} = q)\text{, we need to reciprocate by updating the parent pointer of the new child (e.g., } q.\text{parent} = p)\text{. We should be careful, however, since generally } q\text{ may be null, and so we need to check for this prior to doing the update. It is a straightforward manner to run through the AVL code making these} \]
changes. (This results in a couple of changes in the `insert` function itself, and three changes in the `rotateRight` and `rotateLeft` functions each.)

An alternative solution is to introduce two setter functions to the AVL node class, `setLeft` and `setRight`. These set the value of the left (or right) child of a node and update the parent pointer. Here are the relevant modifications to the `AVLNode` class:

```java
class AVLNode {
    // private data and other methods omitted

    void setLeft(AVLNode q) { // set our left child to q
        left = q;
        if (q != null) q.parent = this; // ...and update q's parent link
    }

    void setRight(AVLNode q) { // set our right child to q
        right = q;
        if (q != null) q.parent = this; // ...and update q's parent link
    }
}
```

Now, every time we change a node’s left or right child link in the original AVL code, we invoke this setter function instead. For example:

```plaintext
p.left = rotateLeft(p.left) -----> p.setLeft(rotateLeft(p.left))
p.left = insert(x, v, p.left) -----> p.setLeft(insert(x, v, p.left))
```

There is a subtlety with this approach, however. The main (public) `insert` function invokes `insert` helper on the root node (`root = insert(x, v, root)`). Since this may result in a new node becoming the root, we should update its parent pointer as well. To do this, we follow this up with `root.parent = null`.

**Solution to the Challenge Problem:** We are going to convert this into a problem on balanced trees, but (as in Heapsort) we will store this tree inside the array `B`. Let’s begin by assuming that `n` is of the form `2^\ell - 1` for some `\ell \geq 2`. (If not, round it up and store $-\infty$ in the unused elements of the array.) We can erect a complete `n`-element binary tree on top of the array `A`, where each node corresponds to an entry of `A` (see Fig. 6). This tree has height $O(\log n)$. Each subtree of this binary search tree spans a subarray of `A` that lies immediately beneath these nodes. (For example, the node labeled 12 that lies directly above `A[4]` spans the subarray `A[1..7]`). Our data structure will store the maximum value of this subarray in this node (e.g., `B[4] = \max_{1 \leq i \leq 7} A[i]`).

The questions that remain are how to answer `max` queries and perform the operation `add` efficiently.

**max(m):** Starting at the root, descend the tree looking for the node that lies immediately above `A[m]`. (We have shown this with the blue path in the figure for `max(14)`.) We need to identify a set of $O(\log n)$ values whose maximum yields `max A[1..m]`. Whenever the path visits a node at index `i`, we include `A[i]` in this set. (These are the values 0, -3, and 6 in the figure.) Also, for each node on the path that lies above `A[j]` where `j \leq m`, we include the value stored in its left child. (These are the node labels 12, 10, and 9 in the figure.) Together, these values span the entire subarray `A[1..m]`, and so their maximum is the desired result. There are two entries per level of the tree, so we can answer the `max` query in $O(\log n)$ time.
add(i, x): Starting at the root, descend the tree looking for the node that lies immediately above $A[i]$. Increment its value by $x$. Let the new value be $y$. Retrace the path back to the root, and replace each node’s entry with the maximum of its current value and $y$. Since $x > 0$, if the maximum within any of these subtrees has changed, it must now be equal to $y$.

Clearly, both operations take $O(\log n)$ time. The final part of the problem is how to embed the binary tree within the array $B$. To do this, we need to show how to compute the various tree operations in constant time. Define the levels of the tree as $0, 1, 2, \ldots$ working up from the leaves. Observe that the nodes that lie at level $\ell$ start at index $2^\ell$ and are separated by distance $2^{\ell+1}$. (For example, the nodes of level $\ell = 2$ are at indices $4 = 2^2$ and $12 = 4 + 2^3$.) We can represent each node by a pair $(i, \ell)$, where $i$ is the index of the entry and $\ell$ is its level. Given a node $(i, \ell)$, where $\ell \geq 1$, it is easy to see that its left child is $(i - 2^{\ell-1}, \ell - 1)$ and its right child $(i + 2^{\ell-1}, \ell - 1)$. Recalling our assumption that $n + 1$ is a power of two, the root of the tree is $(n+1)/2, \lg(n+1)/2)$. Using these formulas, we can store the tree inside the array $B$ and navigate it as if it were a regular binary tree.

The code is fairly simple. Here, for example, is pseudo-code for max:

```c
float max(m) {
    (i, L) = ((n+1)/2, lg((n+1)/2)) // start at the root
    answer = -infinity
    while (L >= 0) // descend until leaf level
        if (i <= m) // go right
            answer = max(answer, B[left(i,L)], A[i])
            (i,L) = right(i,L)
        else // go left
            (i,L) = left(i,L)
    return answer
}
```

Figure 6: Challenge problem.