Solutions to Homework 1: Basic Data Structures and Trees

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

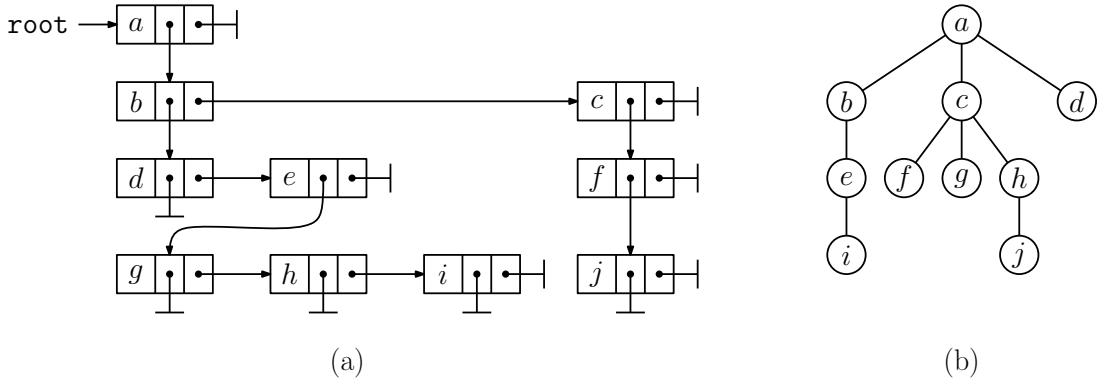

Figure 1: Solution to Problem 1, converting between tree representations.

Solution 2:

(i) (a): A *preorder traversal* of the original tree T is the same as a *preorder traversal* of the binary-equivalent tree. (In the example given in the homework, this is $\langle a, b, e, c, f, i, j, d, g, h\rangle$ for both trees.)

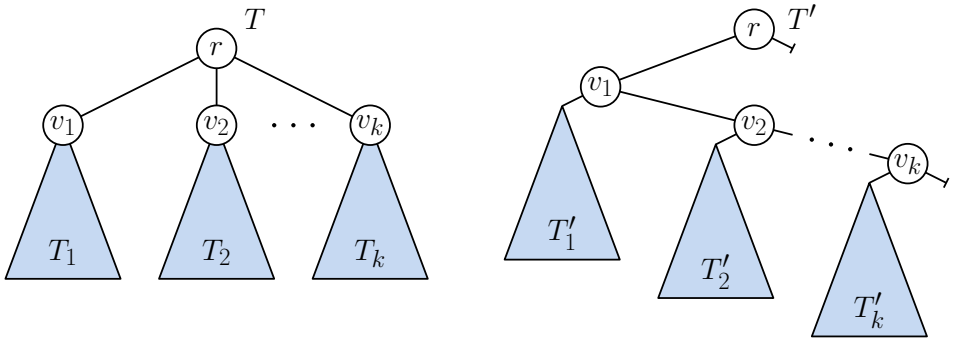

Figure 2: Solution to Problem 2.

This can be seen intuitively as follows. Suppose we have a rooted tree T with root r and subtrees $T_1, T_2, \ldots, T_k$. Let $v_1, v_2, \ldots, v_k$ denote the root nodes of these subtrees. Let T' denote the associated binary-equivalent tree. In this tree r is the root, and it has no right child. If we go to r's left subtree and following right-child links as far as possible, we encounter the roots, $v_1, v_2, \ldots, v_k$, of the associated binary-equivalent subtrees $T'_1, T'_2, \ldots, T'_k$ (see Fig. 2). By definition, a preorder traversal of T visits r followed by preorder traversals of T_1 through T_k. Similarly, a preorder traversal of the binary-equivalent tree visits r followed by preorder

traversals of T'_1 through T'_k. (This can be seen by unraveling the recursive calls along the right-child links.)

(ii) (c): A *postorder traversal* of the original tree T is the same as an *inorder traversal* of the binary-equivalent tree T'. (In the example given in the homework, this is $\langle e, b, i, j, f, c, g, h, d, a \rangle$ for both trees.)

Here again is an intuitive explanation. Consider the same trees T and T' as in (a). A postorder traversal of T applies a postorder traversal to $T_1, \ldots, T_k$, and finally visits r. Observe that the last node visited in each subtree T_i is its root v_i. It is easy to see that an inorder traversal of the binary-equivalent tree T' first performs an inorder traversal of the left subtree and then returns to the root, so again, r is the last node visited. In the process of visiting each subtree T'_i, we first visit all the nodes of this subtree, ending with the root v_i, and then we proceed to the next subtree. Thus, the order in which nodes are visited for both trees is the same.

Solution 3: The inorder threads are shown as dotted lines in Fig. 3. (The inorder traversal is $\langle d, g, j, k, i, b, a, e, c, f, h \rangle$.)

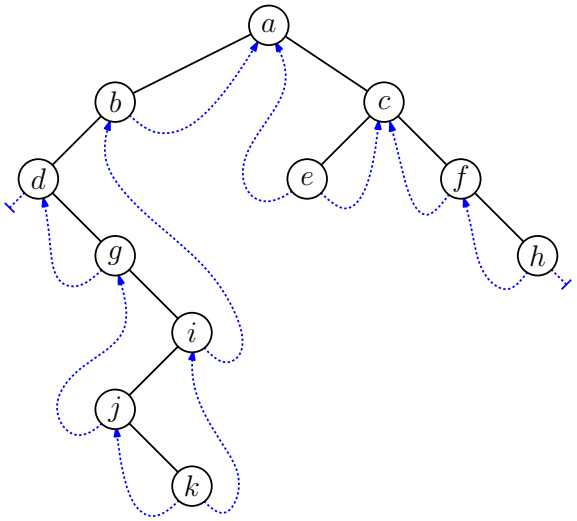

Figure 3: Solution to Problem 3: Inorder threads.

Solution 4:

(a) The insert function for a tree with parent links is a very minor extension to the previous insertion function. The only real difference is that we pass a reference to the parent node as we descend through the tree. Let's assume that the node constructor takes five arguments, the key, the value, the left child, the right child, and the parent. When the new node is added, we set its parent link appropriately.

```
BSTNode insert(Key x, Value v, BSTNode p, BSTNode q) {
    if (p == null)                              // fell out of the tree?
        p = new BSTNode(x, v, null, null, q);  // ... create a new leaf node
    else if (x < p.key)                         // x is smaller?
```

```
            p.left  = insert(x, v, p.left, p);     // ...insert left
        else if (x > p.key)                         // x is larger?
            p.right = insert(x, v, p.right, p);     // ...insert right
        else throw DuplicateKeyException;           // x is equal ...duplicate key!
        return p                                    // return ref to current node
    }
```

We should be careful that we handle the boundary case correctly. When the tree is empty (`root == null`) the initial call will be `root = insert(x, v, null, null)`. This will trigger the `p == null` case, which creates a new (root) node in which all the links are `null`, and the root will be assigned to this new node. Which is exactly what we want.

(b) To compute the inorder successor of a node, we first check whether its right child is not `null`. If so, we apply the same process as in the `findReplacement` function given in Lecture 4 (see Fig. 4(a)). Otherwise, we iteratively follow parent links until we first find an ancestor where we lie in the left subtree of this ancestor (see Fig. 4(b)). If no such ancestor is found, p must be the last inorder node of the tree, and we return `null`.

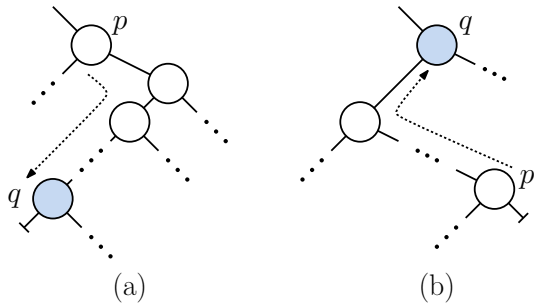

Figure 4: Solution to Problem 4(b): Inorder successor.

```
BSTNode inorderSuccessor(BSTNode p) {          // find p's inorder successor
    if (p.right != null) {                     // p has a right subtree?
        BSTNode q = p.right;
        while (q.left != null) q = q.left;     // find its leftmost node
        return q;
    }
    else {
        BSTNode q = p.parent;                 // follow p's ancestor chain
        while (q != null && p == q.right) {   // until we are in a left child
          p = q;
          q = q.parent;
        }
        return q;
    }
}
```

Solution 5: We will just answer part (b) and point out how part (a) arises as a special case. We will show that the amortized cost in the c-expanding case is $1 + 2\frac{c}{c-1}$. Note that this approaches 3

in the limit as $c \to \infty$. I'll give two proofs. The first is "token-based," and the second is based on straightforward counting all the costs.

Token-based proof: We will prove that the amortized cost of the c-exanding array is $f(c)$ for some function f, and as the analysis proceeds, we will see what function f does the job. Consider any run of n push/pop operations, starting from an empty stack.

First, let's partition the sequence of operations into *runs*, where each run ends whenever we perform a reallocation. At the start of a new run, we have just overflown an array of some size size m and allocated a new array of size cm (see Fig. 5). We copied the existing m elements to this new array, which leaves $cm - m = (c-1)m$ empty slots for expansion. We know, therefore, that the length of the run will be at least $(c-1)m$ (and possibly longer if there are many pops). For each of these operations, we charge the user $f(c)$ tokens. One token will go to pay for the push/pop operation and the remaining $f(c) - 1$ tokens will go into our bank account. When this run ends, we have banked at least $(f(c)-1)(c-1)m$ tokens. We allocate a new array of size c^2m and copy the cm elements over to this array, at a cost of $+2cm$. We want to select $f(c)$ so that we have enough tokens in our bank account. Thus, we require that

$$\text{Number of tokens banked} \;=\; (f(c)-1)\cdot(c-1)m \;\geq\; 2cm.$$

By simple manipulations, we see that $f(c) - 1 = 2\frac{c}{c-1}$ works, that is $f(c) = 1 + 2\frac{c}{c-1}$. This is our final amortized cost. Notice that in the case where $c = 2$, this yields $f(c) = 5$ and when $c = 3$, this yields $f(c) = 4$.

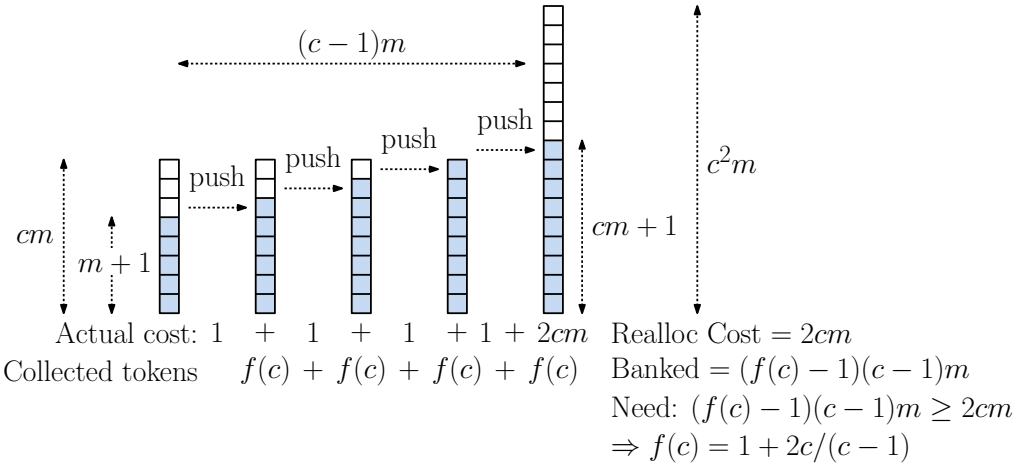

Figure 5: Solution to Problem 5.

Direct proof: Consider any sequence of n operations on the c-expanding stack. Suppose that the array has been reallocated k times. Starting with the initial array of size 1, we have the array sizes $c, c^2, \ldots, c^k$ after each reallocation. Each reallocation to an array of size c^i incurs a cost of twice the size of the previous array, that is, $2c^{i-1}$. This yields a total reallocation cost of:

$$2\sum_{i=1}^{k} c^{i-1} \;=\; 2\sum_{i=0}^{k-1} c^i \;=\; 2\frac{c^k - 1}{c-1} \;\leq\; 2\frac{c^k}{c-1}.$$

(We applied the formula for the geometric series, $\sum_{i=0}^{k-1} c^i = (c^k - 1)/(c-1)$.)

In order to perform the final reallocation, we must have overflown the array of next-to-last size of c^{k-1}. Therefore, the number of operations n is at least $c^{k-1} + 1 > c^{k-1}$. (This is achieved if we do all pushes and no pops.) Therefore, if we consider just the reallocation costs alone, we have a total cost of $T(n) \leq 2\frac{c^{k+1}}{c-1}$ distributed over a total of $n > c^{k-1}$ operations. This implies that the amortized cost for just the reallocations satisfies

$$\frac{T(n)}{n} \leq \frac{2\frac{c^k}{c-1}}{c^{k-1}} = \frac{2c}{c-1}.$$

If we include the addition +1 cost for the actual pushes and pops, the total amortized cost is at most $1 + 2\frac{c}{c-1}$.

Solution to the Challenge Problem: We create two pointers that "chase" each other around the list at different speeds. (I've heard this called the "tortoise and hare" approach.) The lead pointer advances once with every iteration, and the trailing pointer advances once with every other iteration, that is, at half the speed. If the lead pointer ever hits `null`, we report that the list "ends." If the lead pointer ever equals the trailing pointer, we report that the list "loops" around.

We claim that the running time is $O(n)$, where n is the number of nodes. If the list ends, then clearly we discover this after n iterations. If the list loops around, let m denote the number of nodes in the looped part. After $2(n-m)$ iterations, both pointers have made it into the looped section. After an additional $\lceil 3m/2 \rceil$ iterations, the lead pointer is guaranteed to pass the trailing pointer. So, if the list is looped, we will discover this after

$$2(n-m) + \frac{3m}{2} = 2n - \frac{m}{2} \leq 2n - \frac{n}{2} = \frac{3n}{2} = O(n),$$

iterations, as desired.

There is an alternative (less elegant) solution based on guessing a value n' that is greater than or equal to the actual list length. Go n' positions into the list, save a pointer to this node, and then travel an additional n' entries. If you hit the `null` pointer, report that the list "ends". If you hit your saved pointer, report that the list "loops." This algorithm will succeed if $n' \geq n$, where n is the actual number of elements in the list. To make this efficient, try $n' = 2, 4, 8, \ldots, 2^k$. By a geometric series, it can be shown that this runs in $O(n)$ time.