CMSC 420: Spring 2021

Homework 1: Basic Data Structures and Trees

Handed out Fri, Feb 12. Due at **11:00pm, Mon, Feb 22**. Indicated point values are approximate. Before writing your answers, please see the notes at the end about submission instructions.

Problem 1. (8 points)

(1.1) (4 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the "first-child/next-sibling" form.

(1.2) (4 points) Consider the rooted tree of Fig. 1(b) represented in the "first-child/next-sibling" form. Draw a figure showing the equivalent rooted tree.

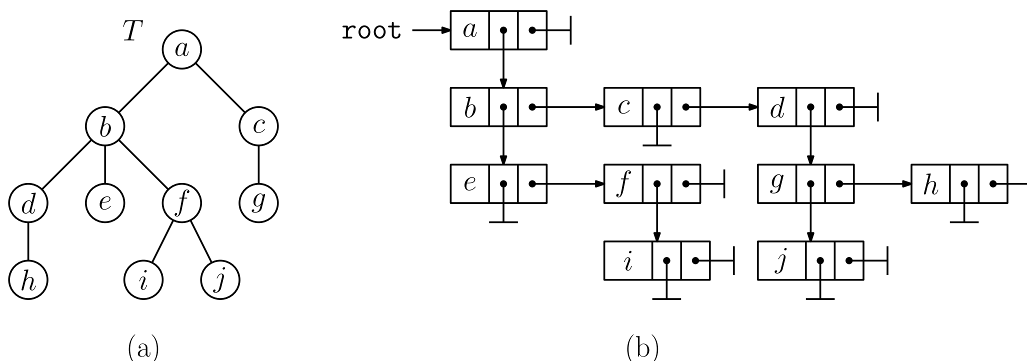

Figure 1: Rooted tree to first-child/next-sibling form and vice versa.

Problem 2. (3 points) Draw the binary tree of Fig. 2(a) with inorder threads added.

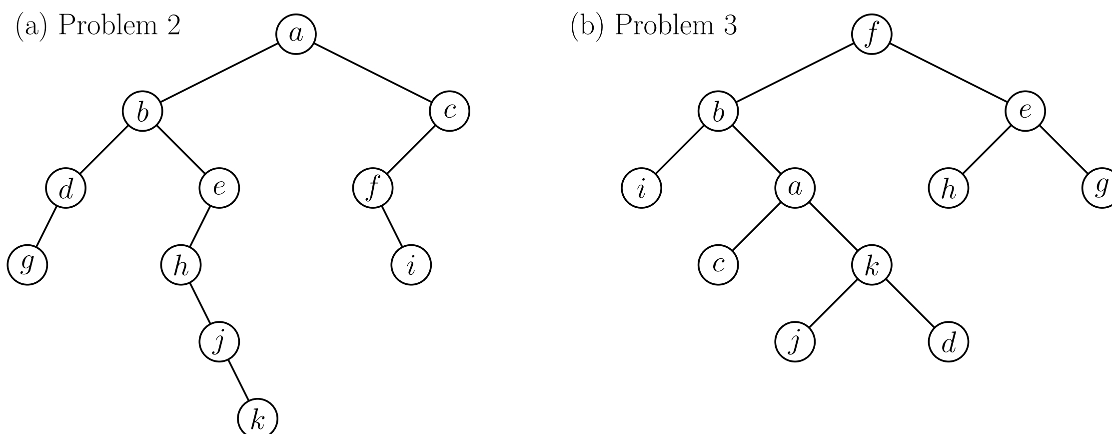

Figure 2: (a) Adding inorder threads to a binary tree and (b) a full binary tree.

Problem 3. (15 points) You have a full binary tree, where each node is labeled with a distinct letter. (Recall that a binary tree is *full* if each non-leaf node has exactly two children.) Throughout this problem, we restrict attention to full binary trees.

(3.1) (3 points) Someone has performed a *postorder* traversal, and gave you a list of the node labels. (For example, in the tree shown in Fig. 2(b), this is $\langle i, c, j, d, k, a, b, h, g, e, f \rangle$.) Is it generally possible to uniquely recover a full binary tree from its postorder sequence? If yes, explain how by presenting an algorithm for doing so. If no, draw two different labeled full binary trees where the postorder lists are the same.

(3.2) (3 points) Repeat question (3.1), but this time list has been modified so that each leaf node has been "flagged" to distinguish leaves from internal nodes. (For example, in the tree shown in Fig. 2(b), if we use "$*$" to indicate a leaf, this would be $\langle i^*, c^*, j^*, d^*, k, a, b, h^*, g^*, e, f \rangle$.)

(3.3) (3 points) Repeat (3.1), but this time for an *inorder* traversal of a full binary tree. (For example, in the tree shown in Fig. 2(b), this would be $\langle i, b, c, a, j, k, d, f, h, e, g \rangle$.)

(3.4) (3 points) Repeat (3.2), but this time for an *inorder* traversal of a full binary tree. (For example, in the tree shown in Fig. 2(b), this would be $\langle i^*, b, c^*, a, j^*, k, d^*, f, h^*, e, g^* \rangle$.)

(3.5) (3 points) We won't ask you to solve the remaining case (of a *preorder* traversal), but you suppose you discuss the unflagged case (3.1) with your best friend. (You both suspect that the evil Prof. Mount may put this question on a future exam.) This friend announces that the answer is "no" and tells you that there is a simple 6-node counterexample. Without even seeing the counterexample, you tell your friend this is wrong! How is this possible? (Assume for the sake of this problem that you are not a psychic.)

Problem 4. (8 points) You are given two $n \times n$ matrices A and B, where (following Java's convention) the rows and columns are indexed from 0 to $n-1$. Their product $A \cdot B$ is an $n \times n$ matrix C where for $0 \le i, j \le n-1$, $C[i,j] = \sum_{k=0}^{n-1} A[i,k] \cdot B[k,j]$.

(4.1) (5 points) Assume that A and B are represented sparse matrices (see Lecture 2 and Fig. 3). Present an efficient algorithm for computing the product $A \cdot B$.

To simplify things, you may assume that the output matrix C is represented as a standard $n \times n$ 2-dimensional array, which has been initialized to zero. To make it possible to generalize your solution to the sparse case, you should fill in the nonzero entries of C in sequential order (e.g., top to bottom and left to right).

(4.2) (3 points) Derive the running time of your algorithm in terms of the following quantities: n, N_A and N_B, where N_A and N_B are the numbers of nonzero entries in the matrices A and B, respectively. (That is, state what the asymptotic running time is and present a proof or convincing explanation of you bound. Hint: In the special case when the matrices are dense, that is $N_A = N_B = n^2$, the running time should be $O(n^3)$.)

Note: At the end of the handout, I present a sample solution for matrix addition, to give you some idea of the amount of detail I expect.

Problem 5. (10 points) This problem involves the AVL tree shown in Fig. 4.

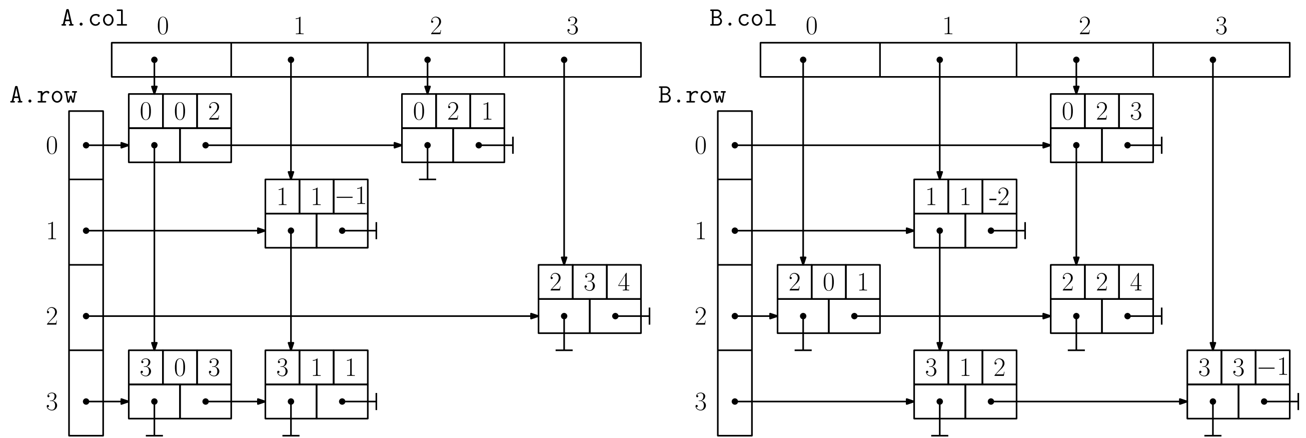

Figure 3: Sparse matrix multiplication.

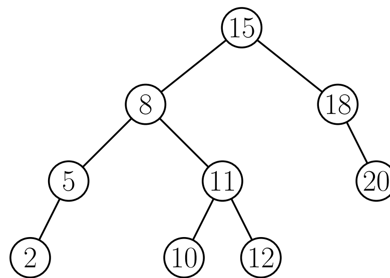

Figure 4: AVL Trees.

(5.1) (2 points) Redraw the tree, but label each node with the height of its subtree and its balance factor. Suggestion: For uniformity, use the convention from the Lecture 5 slides, where the height is written on the left of each node and the balance factor is written on the right.

(5.2) (4 points) Show the result of inserting the key 1 into this tree. First, show the result of inserting the new key into the tree (without any rebalancing) and show the updated balance factors working up from the inserted node to the first node where a rotation is needed. Second, show the final tree after rebalancing is done. Also show the final balance factors.

(5.3) (4 points) Repeat (5.2), but this time insert the key 13. (Do the insertion on the *original* tree from Fig. 4.)

Problem 6. (6 points) In our implementation of AVL search trees, we assumed that, in addition to the key and value, each node stored a pointer to its left and right child as well as the height of its subtree. Suppose that, in addition, we add a pointer to the node's *parent* in the tree. (The root node's parent pointer is set to `null`.) The new node structure is as follows, and the node constructor takes an additional argument specifying the parent:

```
class AVLNode {
   Key key;
   Value value;
   int height;
   AVLNode left, right, parent;

   AVLNode(Key x, Value val, int hgt, AVLNode lft, AVLNode rgt, AVLNode par) {
       // constructor - details omitted
   }
}
```

Present pseudocode for an `insert` function that inserts a new key, applies the appropriate rebalancing, and updates the parent pointers appropriately. As with standard AVL insertion, your function should run in time $O(\log n)$.

Hint: This can be messy if you don't approach it carefully. I believe that the cleanest solution is to find all places where a left or right child is changed (e.g., `p.left = q`) and fix the parent link right away. This works for *almost all* the cases where parent links need to be updated. Beware that `q` may be null, and you must never dereference `null`.

Note: Challenge problems are for fun. We grade them, but the grade is not used when grade cutoffs are determined. After final grades have been computed, I may "bump-up" a grade that is slightly below a cutoff threshold based on these extra points. (But there is no formal rule for this.)

Challenge Problem: You are given an array `A[1..n]` of real numbers (negative, positive, and zero). Design an efficient data structure to perform any sequence of the following two operations:

`void add(int i, float x)`: Given $1 \le i \le n$, and a strictly positive number x, this adds the value x to $A[i]$.

`float max(int m)`: Given $1 \leq m \leq n$, this returns the *maximum* value of the first m elements of A.

Notice that the number of elements remains fixed throughout (there are no insertions or deletions). Only the values may change. Each operation should take $O(\log n)$ time. (Hint: For full credit, as working storage, you can use an additional array `B[1..n]` of floats. If you don't see how to do this with just a single array `B`, for partial credit you can use any data structure of total space is $O(n)$. You shouldn't need any data structures beyond modifications of the ones we have seen so far in class.)

General note regarding coding in homeworks: When asked to present an algorithm or data structure, do **not** give complete Java code. Instead give a short, clean pseudocode description containing only the functionally important elements, along with an English description and a short example. (For example, I would prefer to see "$\lceil n/2 \rceil$" over "`(int) Math.ceil((double) n / 2.0)`".)

Submission Instructions: Please submit your assignment as a pdf file through Gradescope. Here are a few instructions/suggestions:

- You can typeset, hand-write, or use a tablet or any combination. We just need a readable pdf file with all the answers. Be generous with figures and examples. It is much easier to get an idea quickly from a figure than a segment of pseudo-code. I use Latex for text in conjunction with a figure editor called IPE for drawing figures.
- When you submit, Gradescope will ask you to indicate which page each solution appears on. Please be careful in doing this! It greatly simplifies the grading process. This takes a few minutes, so give yourself enough time if you are working close to the deadline.
- Try to keep the answer to each subproblem (e.g. 5.2) on a single page. You can have multiple subproblems on the same page, but Gradescope displays one one page at a time, and it is easiest to grade when everything needed is visible on the same page. If your answer spans multiple pages, it is a good idea to indicate this to alert the grader. (E.g., write "Continued" or "PTO" at the bottom of the page.)
- Most scanners (including your phone) do not take very good pictures of handwritten text. For this reason, write with dark ink on white paper. Use a scan enhancing app such as CamScanner or Genius Scan to improve the contrast.
- Writing can bleed through to the other side. To be safe, write on just one side of the paper.

Algorithm Solutions: Students often ask me how much detail am I expecting for questions that involve giving an algorithm. Whenever you are asked to present an "algorithm" your should present the following (even if I don't explicitly ask for all of this):

- A short English explanation
- Present the algorithm itself, typically in pseudocode
- If it is not obvious, briefly justify the algorithm's correctness
- Give a brief analysis of the running time

Below, I present a sample solution for the problem of matrix addition with the sparse-matrix representation. Let's assume that we are given two sparse matrices A and B (see Lecture 2), and our objective is to compute their matrix sum C. For simplicity, we will assume that C is given as a standard $n \times n$ matrix, which is initialized to zero, and all we need to do is to fill its the nonzero entries. Following Java's conventions, we assume that rows and columns are indexed from 0 to $n - 1$. Recall that in the sparse-matrix representation, we are given two n-element arrays, call them `row[]` and `col[]`, where `row[i]` is the head of a linked list of the entries on row i, and `col[j]` is the head of a linked list of nodes in column j. These linked lists are sorted by increasing order of indices. Each nonzero matrix entry is represented by a node containing the following information:

```
class Node {
  int row      // this entry' row index
  int col      // this entry's column index
  float value  // the value of this matrix entry
  Node rowNext // the next entry (from left to right) in this row
  Node colNext // the next entry (from top to bottom) in this column
}
```

Sparse Matrix Addition: The algorithm iterates through each row $0 \leq i \leq n - 1$, and then iterates in a coordinated manner through the two linked lists `A.row[i]` and `B.row[i]`. If both entries are in the same column, we compute their sum and store it in C. Otherwise, we copy the value that lies in the smaller column index. After processing an entry, we advance to the next element in the linked list. When we reach the end of either list, we simply copy the remaining entries of the other list to the appropriate entries in C.

Here is the pseudo-code. We'll omit braces and type specifications as much as possible.

```
void sparseAddition(SparseMatrix A, SparseMatrix B, float C[][])
    for (i = 0 to n-1)                     // iterate through all the rows
        ap = A.row[i]                      // head of A's ith row
        bp = B.row[i]                      // head of B's ith row
        while (ap != null && bp != null)   // while entries remain in both rows
            if (ap.col < bp.col)           // A's entry comes next?
                C[i][ap.col] = ap.value    //copy to C and advance
                ap = ap.rowNext
            else if (bp.col < ap.col)      // B's entry comes next?
                C[i][bp.col] = bp.value    // copy to C and advance
                bp = bp.rowNext
            else // (ap.col == bp.col)     // both in the same column?
                C[i][ap.col] = ap.value + bp.value // put sum in C and advance
                ap = ap.rowNext
                bp = bp.rowNext
      // At this point, only one list has elements remaining
      while (ap != null)                   // copy any remainder from A to C
          C[i][ap.col] = ap.value
          ap = ap.rowNext
      while (bp != null)                   // copy any remainder from B to C
          C[i][bp.col] = bp.value
          bp = bp.rowNext
```

The correctness follows from the fact that the two pointers `ap` and `bp` move in coordination, so one never gets too far ahead of the other. To obtain the running time, observe that we visit each of the nodes of the sparse matrix representations for A and B exactly once (when we are processing its row). Since there are N_A nodes in A and N_B nodes in B, this takes time $O(N_A + N_B)$. However, even if these quantities are both zero, will still need to access each of the n entries of `A.row` and `B.row`. Thus, the overall running time is $O(n + N_A + N_B)$.