CMSC 420: Fall 2022

Homework 1: Trees and More

Handed out Tue, Sep 20. Due at **11:59pm, Thu, Sep 29**. Point values given with each problem may vary. **Please see the notes at the end about submission instructions**.

Problem 1. (15 points) Answer the following questions involving 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.

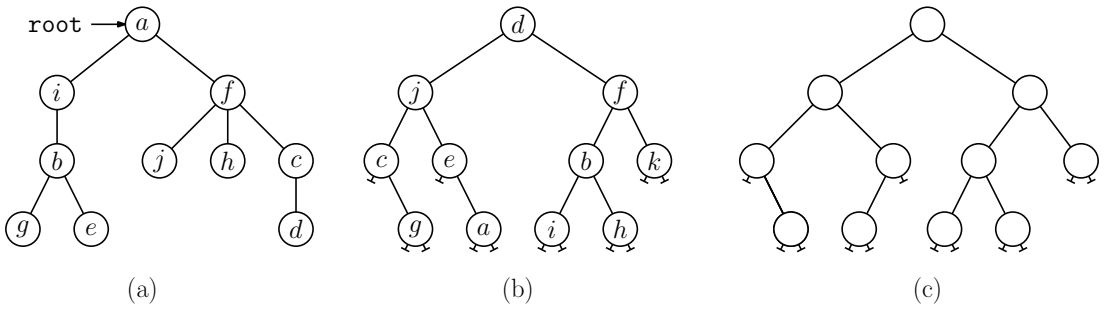


Figure 1: Rooted trees.

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

(c) (2 points) Repeat (b) but for *inorder*.

(d) (2 points) Repeat (b) but for *postorder*.

(e) (3 points) Draw a figure showing the tree of Fig. 1(c) with inorder threads. (As an example, see Fig. 7 from the Lecture 3 notes. Be sure to include any `null` threads.)

(f) (3 points) Draw a figure showing the tree of Fig. 1(c) where each node is labeled with its null path length value. (As an example, see Fig. 5(a) from the Lecture 5 notes).

Problem 2. (5 points) An alternative to using rank in the disjoint-set union-find data structure is to use the size of the tree. Suppose that we modify the union algorithm as follows. When we union two trees T' and T'' having n' and n'' nodes respectively, if $n' \leq n''$, then we link T' as a child of T'', and otherwise we link T'' as a child of T' (see Fig. 2).

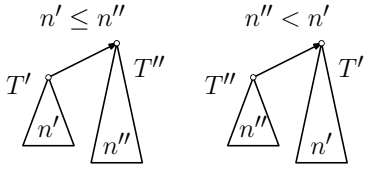


Figure 2: Size-based union.

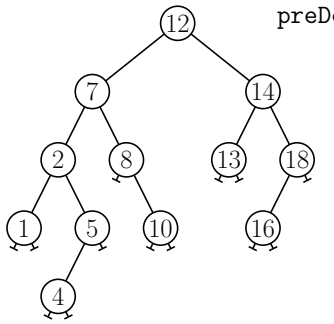
Prove the following lemma, which shows that this also yields height-balanced trees.

Lemma: If the above size-based merging process is used, a tree of height h has at least 2^h elements.

Give a formal proof of this lemma. (Hint: The proof is similar to the one given in the lecture notes.)

Problem 3. (12 points) You are given a binary search tree as given in the class `BinarySearchTree` from Lecture 6 (page 9). (In this problem, we will not be using the `value` fields of the nodes.) In answering the following you may add additional utility functions to this class, but you should not modify the node class, `BSTNode`, nor add additional data to the `BinarySearchTree` class. In addition to your pseudocode, provide a short description of how your function works.

(a) (4 points) Present pseudocode for a member function `preDepth()`, which traverses the nodes in preorder and prints each node's key and its depth (see Fig. 3). (Hint: I would suggest creating a recursive helper function `preDepthHelper(BSTNode u, ...)`, which is given a specific node `u` and, optionally, additional parameters of your choosing. The initial call is `preDepthHelper(root, ...)`. You may create additional helper functions if you like.)




```
preDepth: 12, 0   postNPL: 1, 0   postStats: 1, 3, 0
          7, 1              4, 0              4, 4, 0
          2, 2              5, 0              5, 3, 1
          1, 3              2, 1              2, 2, 2
          5, 3              10, 0             10, 3, 0
          4, 4              8, 0              8, 2, 1
          8, 2              7, 1              7, 1, 3
          10, 3             13, 0             13, 2, 0
          14, 1             16, 0             16, 3, 0
          13, 2             18, 0             18, 2, 1
          18, 2             14, 1             14, 1, 2
          16, 3             12, 2             12, 0, 4
```

Figure 3: Depth and null path lengths in postorder.

(b) (4 points) Present pseudocode for a member function `postNPL()`, which traverses the nodes in postorder and prints each node's key and its null path length (see Fig. 3). (Hint: As in part (a), create a recursive helper function.)

(c) (4 points) Present pseudocode for a member function `postStats()`, which traverses the nodes in postorder and prints each node's key, its depth, and its height (see Fig. 3).

Problem 4. (10 points) You have just invented a new data structure, called a *dual stack*, which stores two stacks in a single array. It works as follows. Given an array `A` of length m, one of the stacks starts at index 0 and grows upwards and the other starts at index $m-1$ and grows downwards (see Fig. 4). More formally, there are two stack tops `top1` and `top2`. Initially, `top1 = -1` and `top2 = m`. Assuming that `top1` $<$ `top2`, when an object is pushed on the first stack, we store it in `A[++top1]`. When an object is pushed on the second stack, we store it in `A[--top2]`. Each operation costs $+1$ unit.

If a push occurs where `top1` $=$ `top2`, we need to expand the arrays (see Fig. 4). Let $n_1 =$ `top1` $+1$ denote the number of elements in stack 1, and let $n_2 = m -$ `top2` denote the number

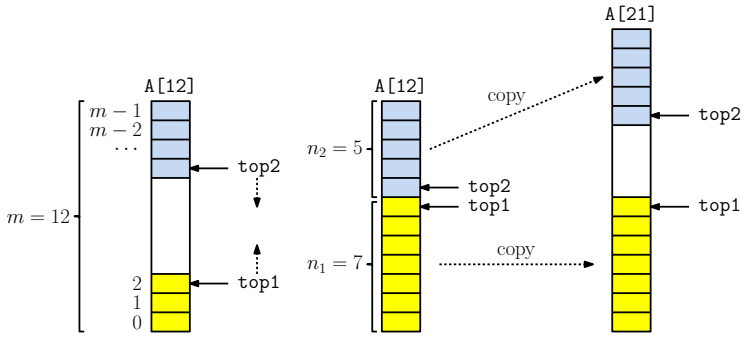


Figure 4: Expanding dual stack.

of elements in the second stack. We allocate a new array of size $m' = 3\max(n_1, n_2)$, and then we copy the elements from the current array to the new array. (The stack-2 elements are copied to the top of the new array.) The actual cost of the expansion is equal to the total number of elements copied, that is, $n_1 + n_2$. After the expansion is finished, we have sufficient space to insert the new element, and we perform the push, which costs +1 unit of work.

For example, in Fig. 4, the current array has 12 elements, and when $n_1 = 7$ and $n_2 = 5$ and someone attempts to push an additional element on one of the two stacks we allocate a new array of size $m' = 3\max(n_1, n_2) = 21$ and we copy the elements of the two arrays into the new array. The cost of the expansion is $n_1 + n_2 = 12$ plus and addition +1 unit for the actual push.

In this problem, we will prove that the dual stack has constant amortized cost. Initially, the array has space for two entries ($m = 2$), and both stacks are empty.

(a) (4 points) Suppose that we have just performed a reallocation. Our current array of size $n_1 + n_2 = m$, and our new array is of size $m' = 3\max(n_1, n_2)$. Explain why at least $m'/3$ pushes can be performed until the new array needs to expand again. (Hint: The actual number depends on the relative values of n_1 and n_2. What is the worst case?)

(b) (2 points) Explain why the cost of the next expansion is m'.

(c) (4 points) Using parts (a) and (b) and the fact that cheap stack operations (which do not cause an expansion) cost +1 unit, derive the smallest constant c such that the amortized cost of our expanding dual stack is at most c and prove the correctness of your answer.

Problem 5. (8 points) Consider the two leftist heaps with roots u and v shown in Fig. 5. In this problem, you will trace the execution of the merge function `merge(u, v)` on this input.

(a) (4 points) List all the recursive calls made in the order in which they are made. (Use the key value in each node to identify it. For example, the first recursive call made is `merge(6,3)`. You should follow the code rigorously. Do not rely on the intuitive "two-phase" approach used to illustrate the algorithm.)

(b) (4 points) Draw the final merged tree structure (after all the recursive calls terminate) and indicate the NPL values for each node.

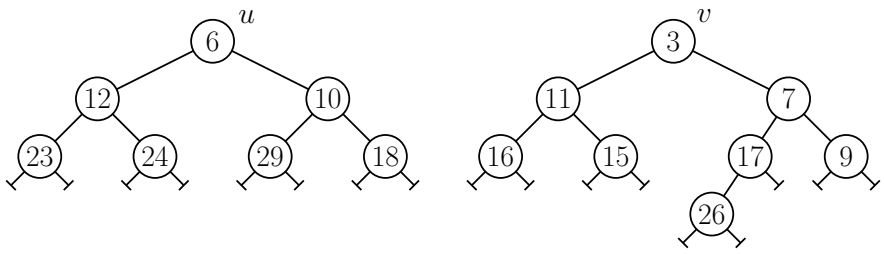


Figure 5: Leftist heap merge.

Note: Challenge problems are not graded as part of the homework. The grades are recorded separately. 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 1: In class, we showed that it is possible to merge two leftist heaps of sizes n' and n'' in time $O(\log n)$, where $n = n' + n''$. We never explained how to perform the operations insert and extract-min, however. Present pseudocode to implement these two operations in $O(\log n)$ time. (Hint: Use the merge helper function.)

Challenge Problem 2: In class, we showed that the rightmost path in any leftist heap has length $O(\log n)$. The analysis presented in the Lecture 5 notes (page 6) shows that if a leftist heap has n nodes, it has at most $\lg(n+1)$ nodes along its rightmost path. Prove that it is generally the case that in any binary tree with $n \geq 1$ nodes, there exists a path from the root to a node with a null child, such that this path has at most $\lg(n+1)$ nodes.

General note regarding coding in homeworks: A common question at the start of the semester is "how much detail are you expecting?" You will figure this out as the semester goes on, but here are some basic guidelines.

Prove vs. Show: If we ask you to "prove" something, we are looking for a well structured proof. If you are applying induction, please be careful to distinguish your basis case(s) and indicate what your induction hypothesis is. If we ask you to "show," "explain," or "justify", we are usually just expecting a brief English explanation. If you are unsure, please check.

Algorithm vs. Pseudocode: When we ask for an "algorithm" we are expecting a high-level description of some computational process, usually in a combination of English and mathematical notation (e.g., "sort the n keys and locate x using binary search"). For the latter, we are expecting a more detailed step-by-step description that look much more like Java (e.g., "`Node q = p.left`").

Remember that you are writing your code to be read by a human, and not a Java compiler. Please omit extraneous details that are easily converted into Java. For example, it is easier to understand "`i =` $\lceil$`n/m`$\rceil$" than "`int i = (int) Math.ceil((double) n / (double) m))`".

Even if we do not explicitly ask for it, whenever you give an algorithm or pseudocode, **you should always provide a brief English explanation.** This helps the grader understand

what your intentions are, and if there is a small error in your code, we can often use your explanation to understand what your actual intentions were. **Even if your solution is technically correct, we reserve the right to deduct points if it is not clear to us why it is correct.**

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. If there is a minor error in your pseudo-code, but the figure illustrates that you understood the answer, we can give partial credit.
- 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 for the graders, since Gradescope takes them right to the page where your solution starts. If done incorrectly, the grader may miss your answer, and you may receive a score of zero. (If so, you can appeal. But hunting around for your answer is troublesome, and it is always best to keep the grader in a good mood!) 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. 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 "See next page" 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 an image-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.
- Students often ask me what typesetting system I use. I use LaTeX for text. This is commonly used by academicians, especially in math, CS, and physics, and is worth taking the time to learn if you are thinking about doing research. If you use LaTeX, I would suggest downloading an IDE, such as TeXnicCenter or TeXstudio. I draw my figures using a figure editor called IPE for drawing figures.