CMSC 420: Spring 2023

Homework 1: Basics, Union-Find, and Heaps

Handed out Thu, Feb 9. Due at the **start of class, Tue, Feb 21**. Solutions will be discussed in class, and so **no late submissions will be accepted**. Submissions will be through Gradescope.

Problem 1. (10 points) This question involves the rooted trees shown in Fig. 1.

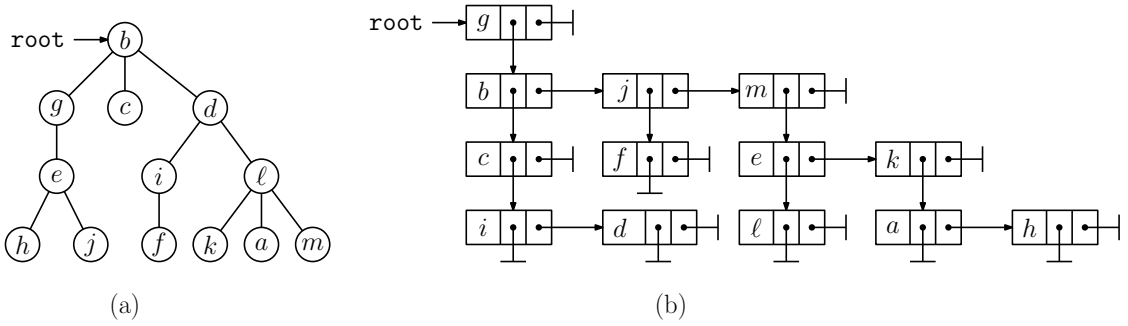

Figure 1: Rooted trees.

(a) (3 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the *first-child/next-sibling form*.

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

(c) (2 points) List the nodes Fig. 1(a) in *postorder*.

(d) (3 points) Consider the rooted tree of Fig. 1(b) given in its first-child/next-sibling form. Draw a figure in its *standard form*.

Problem 2. (10 points) Consider the union-find trees shown in Fig. 2. The `rank` values for each tree are indicated in blue next to each root.

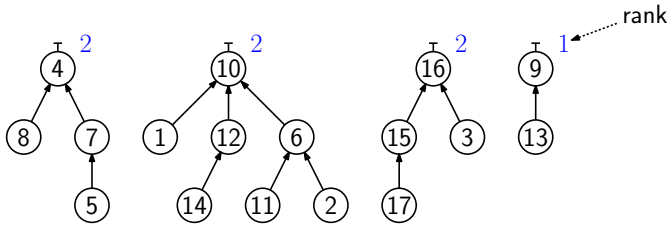

Figure 2: Rooted trees.

(a) (3 points) Show the array of `parent` indices for this set of trees and indicate which elements of this array are *set identifiers*. (See Fig. 1(b) of the latex notes for Lecture 4 as an example of what we are looking for.)

(b) (5 points) Show results after of performing the operations `union(4,10)` and `union(16,9)`. (For the sake of uniformity, use the same convention given in the lecture notes. When performing `union(s,t)`, if both trees have the same rank, link `s` as a subtree of `t`.)

(c) (2 points) Show the result of performing the operation `find(5)` on the data structure *after* the union operations of part (b). In particular, indicate the resulting set identifier returned by the find operation and show the final tree that results after path compression is applied.

Problem 3. (10 points) Consider the two leftist heaps, H_1 and H_2, shown in Fig. 3.

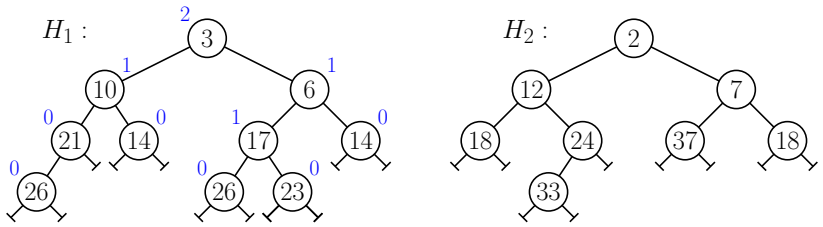

Figure 3: Leftist Heaps.

(a) (3 points) We have labeled each node of H_1 with its `npl` value. Redraw H_2 indicating the `npl` values for each node.

(b) (7 points) Show the result of merging these two heaps together. Indicate the `npl` values for each node. (You need only show the final tree, but the intermediate tree may be given for partial credit.)

Problem 4. (10 points) A useful operation on trees is that of changing the root of the tree, while maintaining all the existing connections. Suppose that you are given a rooted, multiway tree presented using the first-child/next-sibling representation as shown below. The member `root` points to the root node of the tree. For this problem, you may assume that the root is non-null and it has at least one child.

```
class Tree {                    // a rooted multiway tree
  class Node {                  // a node of the tree
    String data;
    Node firstChild;
    Node nextSibling;
  }
  Node root;                    // the root node
  ...
}
```

You are asked to implement a function `promote(Node v)`. This function is given a reference to a node `v`, which is one of the children of the root node. Let `r` denote the root. This function makes `v` the new root of the tree by (1) removing `v` as child of `r` and (2) making `r` the first child of `v` (see Fig. 1). All the other children of `v` and all the other children of `r` remain unchanged. If we think of the tree as an undirected graph, the graph structure is unchanged. We just have a different node as the root.

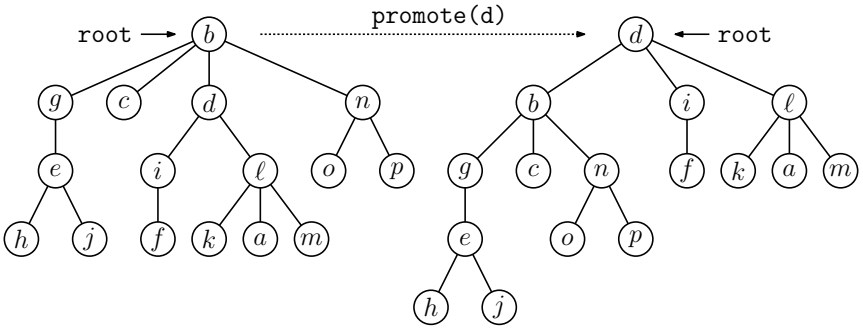

Figure 4: Promoting a child of the root to be the new root.

Present pseudocode for this function. Briefly explain how it works. For full credit, your algorithm should run in time proportional to the number of children the root has. You do not need to perform any error checking. You may assume that `v` is indeed one of the children of the root node.

Problem 5. (10 points) In this problem we will analyze the amortized complexity of a new data structure, called a *double stack*. The idea is to store two separate stacks within a single array `A`. Let `m` denote the current size of this array. We split the array into two subarrays, one of size `s` and the other of size $\texttt{m} - \texttt{s}$. The *lower stack* occupies entries `A[0..s-1]` and the *upper stack* occupies entries `A[s..m-1]` (see Fig. 5(a)). Initially $\texttt{m} = 4$ and $\texttt{s} = 2$, but whenever we expand the array, we will adjust these two values.

Both stacks grow (through pushes) and shrinks (through pops) within their respective subarrays. Each push or pop costs +1 units. We continue until we encounter one of two overflow events:

Lower-stack overflow: A push to the lower stack, when it contains s entries (see Fig. 5(b)).

Upper-stack overflow: A push to the upper stack, when it contains $m - s$ entries (see Fig. 5(c)).

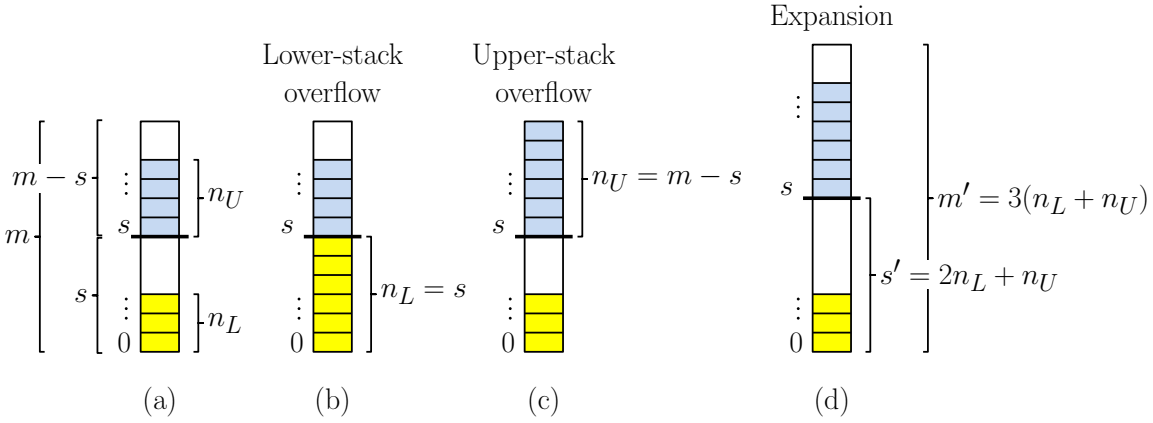

Figure 5: The double stack.

When either of these events occur, we trigger an *expansion*, which works as follows. Let n_L and n_U denote the number of elements currently in the lower and upper stacks, respectively, just before the push. We allocate a new array of size $m' = 3(n_L+n_U)$, and we set $s' = (2n_L)+n_U$. We copy the elements of the lower stack into entries $[0..n_L - 1]$, and we copy the elements of the upper stack into entries $[s'..s' + n_U - 1]$. Note that the number of available entries in the lower stack is $s' - n_L = n_L + n_U$, and the number of available entries in the upper stack is $m' - (s' + n_U) = n_L + n_U$. Thus, both stacks have an equal amount of space in which to expand. After this, we push the new element onto the specified stack.

The cost of the expansion is $n_L + n_U$, accounting for the time to copy the elements from the old array into the new array. To this we add +1 for the actual push operation itself.s

As an example, suppose that $m = 12$, $s = 4$, $n_L = 4$ and $n_U = 2$, and suppose that we perform a push into the lower stack (see Fig. 6). This push will cause the lower stack to overflow. We allocate a new array of size $m' = 3(n_L + n_U) = 18$, and we set $s' = (2n_L) + n_U = 10$. We then copy the four elements from the lower stack to entries [0..3] and we copy the two elements of the upper stack to the entries [10..11]. At this point, both stacks can add another $n_L + n_U = 6$ elements before they overflow. The cost for the expansion is $n_L + n_U = 6$ (since this is the number of elements copied). Finally, we push the new item into entry [4] of the lower stack at a cost of +1. Thus, the entire expansion has cost us $6 + 1 = 7$ units.

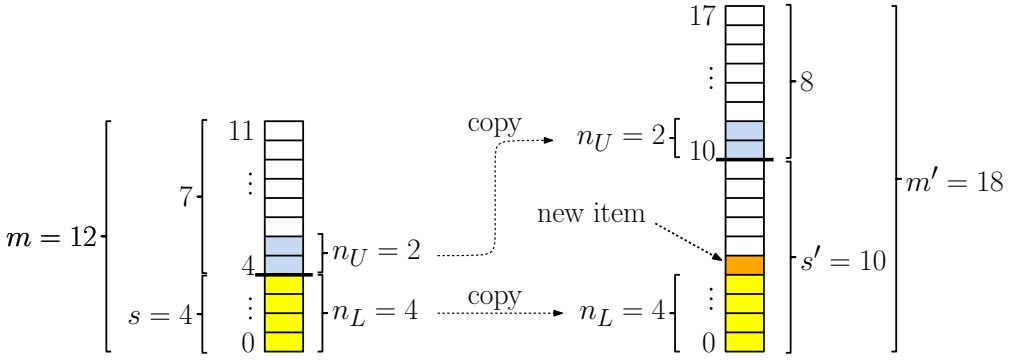

Figure 6: Example of double-stack expansion.

In this problem, we will derive the amortized running time of the double stack. As in class, we will use a token-based argument. We start with the initial configuration, with $m = 4$, $s = 2$, and an empty stack, so $n_L = n_U = 0$. We break the sequence of operations (pushes and pops) into into subsequences called *runs*. The first run starts with the initialization and ends with the first expansion. In general, each run starts just after the prior expansion and runs until the next expansion. (The final run may not end in an expansion, but this is good, since all the operations of this run cost just +1 each.)

Answer the following questions:

(a) (2 points) Suppose that we have just performed an expansion. Prior to the expansion we had an array of size m, and our new array is of size $m' = 3(n_L + n_U)$. As a function of m' (or if you prefer, n_L and n_U), what is the minimum number of operations needed until the next expansion occurs? (Briefly explain.)

(b) (2 points) As a function of m' (or if you prefer, n_L and n_U), what is the worst-case (maximum) cost for the next expansion? (**Hint:** This may depend on the relative sizes of the two stacks.)

(c) (6 points) Using parts (a) and (b), derive a constant τ such that the amortized cost of our expanding dual stack is at most τ. (**Hint:** Note that the worst-cases for (a) and (b) may arise from different scenarios. For the sake of obtaining an upper bound on the amortized cost, it is okay to simply use the worst-case for each, without regard for whether they can both happen simultaneously. See Challenge Problem 2.)

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: You have two friends, Alice and Bob. They have just implemented the `merge` function for leftists heaps, but they each made one mistake. For each of the following, indicate what the consequences are of their error.

(a) When computing the `npl` values for node, Alice consistently used the *maximum* of the `npl` values of the two children (rather than the minimum). That is, she defined

$$\texttt{npl(v)} = \begin{cases} \texttt{-1} & \text{if } \texttt{v} = \texttt{null}, \\ \texttt{1 + max(npl(v.left), npl(v.right))} & \text{otherwise.} \end{cases}$$

Otherwise, her code is exactly the same as given in class.

Which of the following could be said of Alice's program. Select one and justify your answer.

(i) No difference at all. The tree structure is identical, the results are identical, and the running time to perform the `merge` operation is $O(\log n)$.

(ii) The tree structure may differ, but it is still in valid heap order, and the running time to perform the `merge` operation is $O(\log n)$.

(iii) The tree structure may differ, but it is still in valid heap order. However, the running time to perform the `merge` operation may be worse than $O(\log n)$.

(iv) The tree structure may fail to be in heap order.

(b) Bob made a different mistake. Whenever he checks the `npl` values of two subtrees, he always puts the subtree with the *larger* `npl` value on the right (rather than on the left). Otherwise, his code is identical. (In particular, he merges trees along their rightmost path.) Repeat part (a), but with this variant. Again, justify your answer.

Challenge Problem 2: It was noted in Problem 5(c) that the simple analysis based on the worst-case scenarios for parts (a) and (b) cannot both occur. Present a more accurate analysis of the amortized running time that corrects this shortcoming.

General note regarding coding in homework assignments: 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 code to be read by a human, and not a compiler. Please omit extraneous details, as long as the meaning is clear. 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:

- Please read these instructions carefully. If your submitted work is difficult to read, you will receive a grade of 0 for that problem, even if it is technically correct.
- 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.