CMSC 420:Fall 2020 Dave Mount

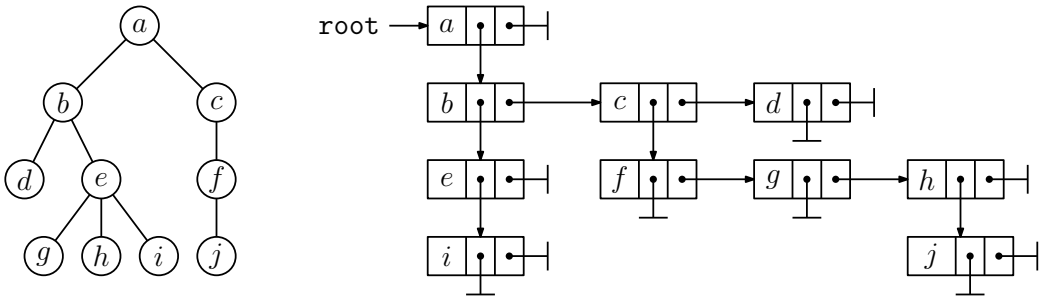
Homework 1: Basic Data Structures and Trees

Handed out Thursday, Sep 10. Due at **11:59pm, Thursday, Sep 17**. (See submission instructions below.) Before writing your answers, please see the notes at the end about submission instructions.

Problem 1. (10 points)

(a) (5 points) Consider the rooted tree of Fig. 1(a). Draw a figure showing its representation in the "first-child/next-sibling" form.

(b) (5 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.



(a) (b)

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

Problem 2. (10 points) We can reinterpret the first-child/next-sibling tree representation of a general rooted (multiway) tree as a binary tree. The first-child link is interpreted as the node's left child and the next-sibling link is interpreted as the node's right child. For example, in Fig. 2(a) we show a rooted (multiway) tree T, in Fig. 2(b), we show its first-child/next-sibling representation, and in Fig. 2(c), we show the corresponding *binary-equivalent tree*, call it T'.

An interesting question to explore involves the relationships between the original tree and its binary-equivalent tree. Answer each of the following questions. Your answer should hold not only for the above example, but any nonempty rooted tree T and its binary equivalent tree T'. *Provide a brief explanation of your answers (perhaps with an adjoining figure). A formal proof is not required.*

(i) (5 points) A *preorder traversal* of the original tree T is the same as:

(a) a *preorder traversal* of the binary-equivalent tree T'

(b) a *postorder traversal* of the binary-equivalent tree T'

(c) an *inorder traversal* of the binary-equivalent tree T'

(d) none of the above

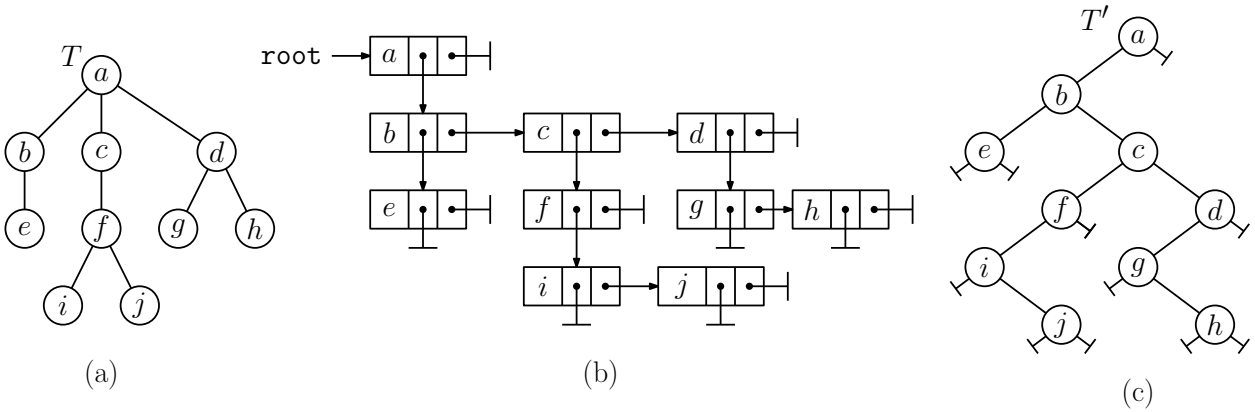


Figure 2: A rooted tree and the binary equivalent.

(ii) (5 points) A *postorder traversal* of the original tree T is the same as:

(a) a *preorder traversal* of the binary-equivalent tree T'

(b) a *postorder traversal* of the binary-equivalent tree T'

(c) an *inorder traversal* of the binary-equivalent tree T'

(d) none of the above

Problem 3. (6 points) Consider the binary tree shown in Fig. 3. Draw a figure showing the inorder threads (analogous to Fig. 7 in Lecture 3).

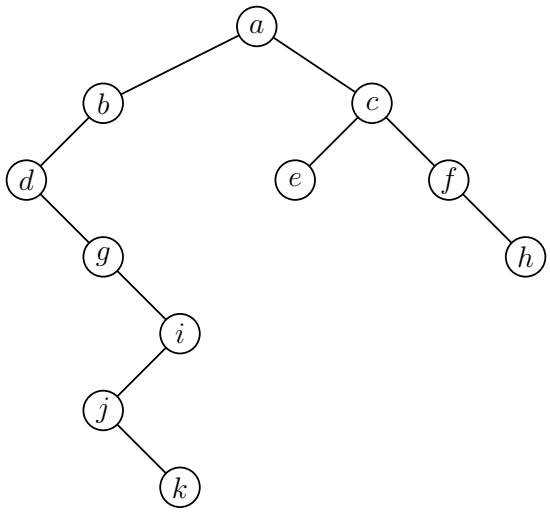


Figure 3: Add inorder threads to this tree.

Problem 4. (12 points) In our implementation of binary search trees, we assumed that each node stored a pointer to its left and right children. Suppose that, in addition, we add a pointer to the node's parent in the tree. The new node structure is as follows, and the node constructor takes an additional argument specifying the parent:

```
class BSTNode {
    Key key;
    Value value;
    BSTNode left;
    BSTNode right;
    BSTNode parent;

    BSTNode(Key x, Value v, BSTNode l, BSTNode r, BSTNode p) {
        // constructor - details omitted
    }
}
```

Assume that the root of the tree has a parent pointer of `null`.

(a) (6 points) Present pseudocode for an `insert` function that inserts a new key and updates the parent pointers appropriately. (Hint: Modify the `insert` function from Lecture 4.) We will need to change the function's signature, now taking four arguments: the key x, the value v, the current node p, and the current node's parent q:

```
BSTNode insert(Key x, Value v, BSTNode p, BSTNode q) {
    // ... fill this in
}
```

In order to insert a key-value pair (x, v) into your tree, the initial call is

```
root = insert(x, v, root, null)
```

Your function should run in time proportional to the height of the tree.

(b) (6 points) Assuming we have parent links, present pseudocode for a function that computes the *inorder successor* of an arbitrary node p in the tree. If p is the last inorder node of the tree, this function returns `null`. The function signature is as follows:

```
BSTNode inorderSuccessor(BSTNode p) {
    // ... fill this in
}
```

Briefly explain how your function works. (*Don't just give code!*) Your function should run in time proportional to the height of the tree.

Problem 5. (12 points) In this problem, we will generalize the amortized analysis of the expandable stack from Lecture 2. Recall that we are given a stack that is stored in an array. Initially, the array has 1 element. Each time that a push or pop operation does not cause an overflow, it costs us +1 unit. If a push operation causes the array to overflow, we doubled the array size, growing, say, from m to $2m$, copy the elements over to this new array, and then perform the push. It costs $+2m$ units for the reallocation and an additional +1 unit to perform the actual operation. In class, we showed that the *amortized cost* of such a stack is at most 5. Formally, this means that if we start from an empty stack and an array of size 1, and perform any sequence of n push/pop operations, the total cost is at most $5n$.

Please answer the following problems. (Part (a) is a special case of part (b). If you are confident of your answer, you can answer just part (b), and we will apply the same score to part (a).)

(a) (4 points) Suppose that we modify our expandable stack example so that, whenever the stack overflows, we allocate a new array of *three times* the size, growing, say, from m to $3m$. So, successive overflows would result in arrays of sizes $3, 9, 27, \ldots, 3^k, \ldots$. Let us assume a similar cost model. Each non-overflowing push/pop costs $+1$ unit. Whenever we overflow the array, we will charge $+2m$ for reallocation and an additional $+1$ for the actual push itself (see Fig. 4). (Why $+2m$ and not $+3m$? The choice is somewhat arbitrary, but my logic is that I'll charge $+1$ for reading each item from the old array and $+1$ for writing it in the new array.)

Using the "token-based" method given in Lecture 2, prove that this version of the data structure has an amortized cost of at most 4. (I believe that this is the best possible upper bound as n becomes large. If you get a smaller number, better check with us, because it is probably wrong.)

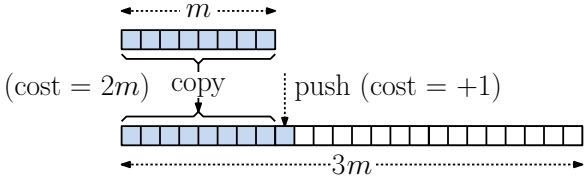


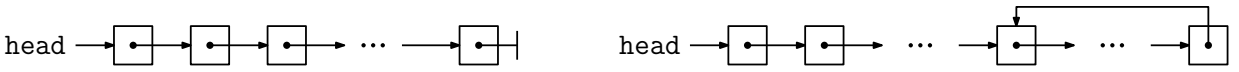
Figure 4: Generalizing the expandable stack (for $c = 3$).

(b) (8 points) Let's generalize this further. Given a positive integer constant $c \geq 2$, our expandable stack allocates a new array of c *times* the size, growing, say, from m to cm. So, successive overflows would result in arrays of sizes $c, c^2, c^3, \ldots, c^k, \ldots$. We will charge $+1$ for each individual operation and $+2m$ for copying the m elements over into the new array (see Fig. 4 for the case $c = 3$). Derive the amortized cost of this version of the stack as a function of c. (For full credit your answer should be tight. In particular, for $c = 2$ and $c = 3$, your formula should give you 5 and 4, respectively.)

Note: Challenge problems are just for fun. We grade them, but the grade is not used to determine your final grade in the class. Sometimes when assigning cutoffs, I consider whether you attempted some of the challenge problems, and I may "bump-up" a grade that is slightly below a cutoff threshold based on these points. (But there is no formal rule for this.)

Challenge Problem: You are given a rather trivial linked list, where each node contains a single member, `next`, which points to the next element in the linked list. The variable `head` points to the head of the linked list. There are two possible forms that the list might take. First, it might *end* in a `null` pointer (see Fig. 5(a)), or second, it might *loop* around on iteself, where for some node `p`, `p.next` points to an earlier node within the list, possibly `p` itself (see Fig. 5(b)).

Give pseudocode for a function that determines whether the list ends in `null` or wraps around on itself. Here is the catch: You cannot modify the nodes of the list, and your function cannot use more than a constant amount of working storage. (This latter requirement implies that you cannot make recursive calls of more than a constant recursion depth, since this would use more than a constant amount of system storage.) Your algorithm must run in time proportional to the total number of elements in the list.



(a): List ends (b): Loops around

Figure 5: Generalizing the expandable stack (for $c = 3$).

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:

- 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.
- Most scanners (including your phone) do not take very good pictures of handwritten text. (Yes, it might be "readable," but when you are grading 150 assignments, the eye strain caused by reading a low-contrast image can put a grader in a bad mood, which you don't want!) For this reason, we urge you to use a scanning app such as CamScanner or Genius Scan or Genius Scan.
- Writing can bleed through to the other side. To be safe, write on one side of the paper.
- Since Gradescope displays just one page at a time, it makes our job easiest if you start each problem answer at the top of a new page, and keep as much of the answer as possible on the same page. If your answer spans multiple pages, it is a good idea to indicate this in a note at the bottom of the page (e.g., "Continued on the next page" or the old British acronym "PTO" (for "Please Turn Over").
- You may typeset or handwrite your answers. You can also use a combination—typesetting text and handwriting figures. If you typeset, please be sure to include figures as appropriate. In my experience, students often rely too heavily on text, when a simple figure would make the point clearly. A good figure often illustrates your intent very succinctly. Also, when you provide a figure together with text, the grader can perform "error correction," when either text or figure contains a minor error. In case you are curious, when preparing my lecture notes I use Latex for text in conjunction with a figure editor called IPE for drawing figures.