Due: Thu Oct 27, in class.

HOMEWORK with SOLUTION, prepared by Instructor and T.A.s

INSTRUCTIONS:

• Please write clearly, and economically. Use notations we introduce in lecture notes and in lectures.

• IMPORTANT: recall our new rules that when submitting your homework, you should write your name in the format:
  LAST-NAME, First-Name.
  You will be given two bonus points for doing this. You may even do the redundant thing of underlining your LAST-NAME.

1. (8 Points, Checking BST Property) In the previous homework, you designed a recursive subroutine \( \text{CheckBST}(u) \) to check if the binary tree \( T_u \) rooted at \( u \) represents a BST. We now explore a different solution. Design a recursive subroutine \( \text{bCheckBST}(u, \text{min}, \text{max}) \) that returns a Boolean value; the value is true iff \( T_u \) is a BST with keys lying in the range \([\text{min}, \text{max}]\). REMARK: we also define \( \text{bCheckBST}(u) \) to be \( \text{bCheckBST}(u, -\infty, +\infty) \).

SOLUTION

\[
\text{bCheckBST}(u, \text{min}, \text{max}): \\
0. \quad \text{If } (u = \text{nil}) \quad \text{Return(True).} \\
1. \quad \text{ElseIf } ((\text{min} \leq u.\text{key} \leq \text{max}) \\
\quad \quad \quad \text{and } \text{bCheckBST}(u.\text{left}, \text{min}, u.\text{key}) \\
\quad \quad \quad \text{and } \text{bCheckBST}(u.\text{right}, u.\text{key}, \text{max})) \\
\quad \quad \quad \text{Return(True).} \\
2. \quad \text{Else Return(False).}
\]

The base case simply checks. Recursively, this algorithm is correct if there are no duplicate keys. To handle duplicate keys, it is better to require the keys to lie in range \([\text{min}, \text{max}]\) (i.e., keys in the BST must satisfy \( \text{min} \leq K < \text{max} \) instead of \( \text{min} \leq K \leq \text{max} \)).

Comments: This solution seems more direct that the one in the previous homework because we do not have to (artificially) use the returned pair to encode true/false values.

2. (4 Points, \((a, b)\)-Search Tree)  

(PROBLEM IS SCRATCHED because it appeared in Spring2011)

Justify the following statements about \((a, b)\)-search trees:

(a) If we only have insertions into an \((a, b)\)-tree, then the keys in an internal node are just copies of keys of items found in the leaves.

(b) It is possible to maintain the property in part (a) even if there are both insertions and deletions.

3. (4 Points, \((a, b)\)-Search Tree)  

(PROBLEM IS SCRATCHED, as it appeared in Spring2011)

Consider the tree shown in Figure 1. Although we previously viewed it as a \((3, 4)\)-tree, we now want to view it as a \((2, 4)\)-tree. For insertion/deletion we further treat it as a \((2, 4, 1)\)-tree.

(a) Insert an item (whose key is) 14 into this tree. Draw intermediate results.

(b) Delete the item (whose key is) 4 from this tree. Draw intermediate results.
4. (6 Points, \((a, b)\)-Search Tree) Our insertion and deletion algorithms try to share (i.e., donate or borrow) children from siblings only. Suppose we now relax this condition to allow sharing among “first cousins” (i.e., nodes that share a common grandparent). Modify our insert/delete algorithms so that we try to share with direct siblings or cousins before doing the generalized split/merge.

ADDED REMARKS: We should say "immediate cousin" instead of "first cousin". You are to modify the "Enhanced Standard Insert/Delete". In the Enhanced Version, we try to denote (in case of insertion) and try to borrow (in case of delete) if at all possible. Donate and borrow can only be from an immediate sibling.
SOLUTION  It is most convenient to use the basic framework in ¶48 (Basic Mechanics of Insertion and Deletion). We only have to modify the MAIN LOOP, which is a while-loop with the current node \( u \) either overfull (for insertion) or underfull (for deletion). Here are the new steps for the MAIN LOOP:

STEP 1. First assume \( u \) is not a root. Read into main member the parent \( p = u.Parent \) of \( u \) (this assumes nodes have parent pointers).

STEP 2. Using information in \( p \), we can bring into main memory each of the immediate siblings of \( u \) (in turn). For each immediate sibling \( v \), we check if we can donate (if overfull) or borrow (if underfull) to \( v \). If so, we can modify \( u, v, p \) as described in Lecture Notes (¶46). After writing \( u \) and \( v \) back to secondary memory, and letting \( u \leftarrow p \), we are done (the MAIN LOOP will terminate).

STEP 3. Suppose we fail in borrowing/donating. We next wish to borrow/donate to a “immediate first cousin”. NOTE: “Immediate” means it is an adjacent node in the same level, and “first cousin” means it shares a common grandparent. To do this we first check that we do have an immediate first cousin. This can be done by bringing the grandparent \( g \) into main memory. If so, we can then bring into memory the uncle \( u \) and the immediate first cousin \( c \). So we have in memory these 5 nodes: \( u, p, g, u, c \). If possible to borrow or donate, we do so. This will require modifications to all five nodes. We then write them back to main memory, with the exception of \( p \) (which is redefined to be the current node \( u \)).

STEP 4. If borrowing/donating fails to siblings or cousin, we split/merge in the usual way. THAT is the full high-level description of the algorithm.

Let us now give a bit more detail (and you ought to in your solutions): Let us assume that \( p.degree \) is the number of children of \( p \), and the keys and pointers in \( p \) are stored in two arrays, \( p.Keys[1..m] \) and \( p.Pointers[0..m] \) (with \( m = p.degree \)). See ¶44.

Here is how we do the Enhanced Algorithms (the part in STEP 2 above): We search sequentially for the \( i \) such that \( p.Pointers[i] = u \). If \( i \geq 1 \) then \( u \) has a left sibling which is \( v = p.Pointers[i-1] \). We bring \( v \) into main memory. If \( v.degree = b \), then we cannot donate to \( v \), and if \( v.degree = a \), we cannot borrow from \( v \). Assume we fail to donate/borrow. Then if \( v < m \), then we can bring into memory \( p.Pointers[i+1] \) (and still call it \( v \)). Again, we test whether we can donate/borrow.

Here is how we do STEP 3 above: If \( i > 0 \) and \( i < m \), then there is no direct cousin, and we are done. Otherwise, we bring in the grandparent \( g = p.Parent \). We search for the index \( j \) such that \( g.Pointers[j] = p \). If \( i = 0 \) then we have an immediate cousin iff \( j > 0 \). If \( i = m \) then we have an immediate cousin iff \( j < g.degree \). If we have an immediate cousin \( c \), then we bring into memory the uncle \( u \) and also \( c \). These are defined to be

\[
\begin{align*}
u & := \begin{cases} 
  g.Pointers[j-1] & \text{if } i = 0, \\
  g.Pointers[j+1] & \text{if } i = m.
\end{cases} \\
c & := \begin{cases} 
  u.Pointers[u.degree] & \text{if } i = 0, \\
  u.Pointers[0] & \text{if } i = m.
\end{cases}
\end{align*}
\]

If \( c.degree < b \), we can donate to \( c \), and if \( c.degree > a \), we can borrow from \( c \). In either case, we need modify the 5 nodes. This is similar to borrow or donate from an immediate sibling (we expect you to provide some description of this...).

Comments: Clearly, one could generalize this to ”second immediate cousin”, etc. The unclear question is whether this is still productive.

5. (8 Points, Graph Reversal) The following is a basic operation for many algorithms: given a digraph \( G \) represented by adjacency lists, compute the reverse digraph \( G^{rev} \) in time \( O(m + n) \). Recall (Lecture 1, Appendix) that \( u \rightarrow v \) is an edge of \( G \) iff \( v \rightarrow u \) is an edge of \( G^{rev} \). Show that your algorithm has the stated running time.

ADDED REMARKS: Adjacency lists can be an array of lists or list-of-lists. You can use either. You can directly modify the input graph (destroying the original), or create a new reverse graph (preserving the original).
(8 Points, BFS Classification of Digraph Edges) (a) Give a classification of the edges of a digraph $G$ relative to the operations of running the BFS algorithm on $(G; s_0)$. You must prove why your classification is complete.

(b) Now turn your answer in part(a) into a “computational classification”. I.e., devise an algorithm to classify every edge of $G$ according to (a). Recall that you must use shell programming.

POSTSCRIPT REMARKS: For part (a), we hinted that there are two additional kinds of edges, in addition to the classification of bigraph edges: tree, cross, level, and unseen. For part (b), we hinted that it may be hard to do the classification in $O(n + m)$ time, but do the best you can.

**SOLUTION**

(a) We have the types of edges used in classifying bigraph edges: tree, cross, level, and unseen. NOW, it is now possible to have edges $u-v$ from level $i$ to level $j$ where $j < i$. These are the “back edges”. But there are two kinds of back edges: $u-v$ where $v$ is an ancestor of $u$ and $v$ is not an ancestor. Call them “ancestor edges” and “non-ancestor edges”, respectively.

E.g., consider the cyclic graph $C_n$ with edges of the form $i-(i+1)$ (for $i = 1, \ldots, n-1$) and $n-1$. If the root is 1, then $n-1$ is an ancestor edge. How to show that the above classification is complete? Besides the tree and unseen edges, the rest are edges that goes between two nodes in the tree. Suppose $u-v$ is an edge that goes from level level $i$ to level $j$. Logically, we have only four cases:

- If $i = j$, this is a level edge.
- If $j - i = 1$, this is a cross edge.
- If $j - i > 1$, call this a “forward edge”. But such edges cannot arise in BFS, because of the Monotone 0-1 Property.
- If $j - i < 0$. These are the back edges above.

This proves the completeness of our classification.

(b) If we maintain a parent pointer for every node in the BFS tree, then it is easy to check for each back edge whether it is an ancestor edge or non-ancestor edge. Doing this check costs $O(n)$ per back edge, and so $O(mn)$ for all the $m$ edges. So the overall complexity is $O(m + n + mn) = O(mn)$.

**Comments:** Here is a more advanced solution for (b), with complexity to $O(m + n^2)$.

Recall our goal is to classify each back-edge into ancestor or non-ancestor. We will not do this at the instant we discover the back-edge. Instead, we just store any back-edge $u-v$ in a set $A(u)$ associated with $A(u)$. After we have constructed the BFS tree (and all the sets $A(u)$), we start a new process. For each leaf in the BFS tree, we trace the root-path from that leaf back to the root. We assume that each node $u$ of the BFS tree maintains another set $B(u)$ of backedges into $u$; initially, $B(u)$ is empty. In this tracing a rootpath, when we reach a node $u$, we do two things:

(A) For each back edge $u-v$ in $A(u)$, then we append edge $u-v$ to $B(v)$.

(B) For each edge $w-u$ in $B(u)$, we mark it as an ancestor edge.

At the end of tracing one path, we can assume that $B(u)$ along the rootpath is reset to be empty. The total work done in retracing each root-paths is $O(n + m')$ where $m'$ is the number of back edges in $A$-sets along the path. Summing over all the root paths, we get $O(n^2 + m)$ because the total of all the $m'$s is $m$.
7. (12 Points, Graph Center, Radius, and Diameter)

(PROBLEM IS SCRATCHED, as the solution could be found in the web).

Let $G = (V, E)$ be a connected bigraph. For any vertex $v \in V$ define

$$\text{radius}(v, G) := \max_{u \in V} \text{distance}(u, v)$$

where distance$(u, v)$ is the length of the shortest (link-distance) path from $u$ to $v$. The center of $G$ is the vertex $v_0$ such that radius$(v_0, G)$ is minimized. We call radius$(v_0, G)$ the radius of $G$ and denote it by radius$(G)$. Define the diameter diameter$(G)$ of $G$ to be the maximum value of distance$(u, v)$ where $u, v \in V$.

(a) Prove that $2 \cdot \text{radius}(G) \geq \text{diameter}(G) \geq \text{radius}(G)$.

(b) Show that for every natural number $n$, there are graphs $G_n$ and $H_n$ such that $n = \text{radius}(G_n) = \text{diameter}(G_n)$ and diameter$(H_n) = n$ and radius$(H_n) = \lceil n/2 \rceil$. This shows that the inequalities in (a) are the best possible.

(c) Using DFS, give an efficient algorithm to compute the diameter of a undirected tree (i.e., connected acyclic undirected graph). Please use shell programming. Prove the correctness of your algorithm. What is the complexity of your algorithm? HINT: write down a recursive formula for the diameter of a tree in terms of the diameter and height of its subtrees. Store these values as fields in each node.

(d) Same as (c), but compute the radius instead of diameter.

(e,f) Same as (c) and (d) but using BFS instead of DFS.