CS21003 Algorithms I, Autumn 2013–14

Mid-Semester Test

Maximum marks: 60 Time: 24-Sep-2013 (2:00–4:00 pm) Duration: 2 hours

---

Roll no: ____________ **Name:** ______________________________

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. You are given a binary tree T in the standard pointer-based representation. Each node in the tree consists of a key and two pointers (left and right). Write a function that, upon the input of a pointer to the root node, returns a suitable value indicating whether T is *structurally* an AVL tree. You do not need to look at the keys to identify whether T satisfies BST ordering (this is already covered in the class). Do not add any extra space in the nodes of the tree. If there are n nodes in the tree, your function must run in $O(n)$ time. **(10)**

Solution The following recursive function returns an integer. If the returned value is -2, then T is structurally *not* an AVL tree. Otherwise, the height of T is returned. The outermost caller function can check whether T is an AVL tree by verifying whether the return value is $\geqslant -1$.

```
int isAVLTree ( treenode *T )
{
   int lh, rh, d;

   if (T == NULL) return -1;
   lh = isAVLTree(T -> left);
   rh = isAVLTree(T -> right);
   if ((lh == -2) || (rh == -2)) return -2;
   d = rh - lh;
   if ((d <= -2) || (d >= 2)) return -2;
   return 1 + ((lh >= rh) ? lh : rh);
}
```

2. You are given a rooted tree T. The *width* of T is the maximum number of nodes at a level in the tree. For example, consider a tree of height three on ten nodes $a, b, c, d, e, f, g, h, i, j$, where a is the root having three children b, c, d, node b has two children e, f, node d has three children g, h, i, and h has one child j. In this tree, the numbers of nodes at levels $0, 1, 2, 3$ are respectively $1, 3, 5, 1$. The width of this tree is therefore 5.

You are given T in the first-child-next-sibling representation. Design an algorithm to compute the width of T in $O(n)$ time, where n is the number of nodes in T. **(10)**

Solution In the class, we have seen how the height h of T can be computed in $O(n)$ time. We use an array C of size $h+1$ in order to store the counts of the nodes at different levels of T. This array is filled by a recursive traversal of the tree. Then, a maximum is taken over the counts.

```
void traverse ( tree T, int C[], int level )
{
   if (T == NULL) return;
   C[level]++;
   traverse(T -> left, C, level+1);
   traverse(T -> right, C, level);
}

int width ( tree T )
{
   int *C, max, h, i;

   h = height(T);
   C = (int *)malloc((h+1) * sizeof(int));
   for (i=0; i<=h; ++i) C[i] = 0;
   traverse(T,C,0);
   max = 0;
   for (i=0; i<=h; ++i) if (C[i] > max) max = C[i];
   free(C);
   return max;
}
```

Roll no: ____________________ **Name:** ____

3. You are given two arrays A and B of integers of sizes m and n. Your task is to check whether A and B are equal as *sets*. The arrays A and B need not be sorted, and may contain repeated occurrences of the same values. When we treat them as sets, all repetitions should be discarded (only one occurrence counts). For example, if $A = (5, 1, 2, 5, 1, 8, 1, 3)$ and $B = (2, 1, 8, 2, 5, 3)$, the answer is *Yes*, since both the arrays are equal to $\{1, 2, 3, 5, 8\}$ as sets. Design an algorithm to solve this problem in expected $O(m + n)$ time. **(10)**

Solution We first verify whether $A \subseteq B$ using a hash table H. We insert elements of B one by one in H. Equal values are not duplicated in H. After this insertion phase, we search whether each value stored in A can be found in H. If we choose a hash table of an appropriate size (like $2n$ cells with open addressing), each insertion and search finishes in expected $O(1)$ time. Consequently, the expected running time for checking whether $A \subseteq B$ is $O(m + n)$.

We can similarly check in $O(m + n)$ time whether $B \subseteq A$. We finally declare A and B as equal if and only if both $A \subseteq B$ and $B \subseteq A$ are true.

(A faulty strategy to decide whether $A \subseteq B$ is to insert the elements of B in H, and then delete elements of A one by one from H. The problem with this method is that a failed deletion attempt does not imply the absence of this element in B. It could very well be that this element had an earlier appearance in A and was successfully deleted from H in an earlier iteration.)

4. Let A be an unsorted array of integers $a_0, a_1, a_2, \ldots, a_{n-1}$. An *inversion* in A is a pair of indices (i, j) with $i < j$ and $a_i > a_j$. Modify the merge sort algorithm so as to count the total number of inversions in A in $O(n \log n)$ time. **(10)**

Solution We split A in two halves L (the left $\lceil n/2 \rceil$ elements) and R (the right $\lfloor n/2 \rfloor$ elements). We recursively compute the numbers n_{LL} and n_{RR} of inversions in L and R. The recursive algorithm will also sort L and R. While merging L and R to a final sorted array, we compute the number n_{LR} of inversions between L and R. We finally return the count $n_{LL} + n_{RR} + n_{LR}$. The following functions do this.

```
int merge ( int *L , int n1, int *R, int n2 )
{
   int *B, i, j, k, nLR;

   B = (int *)malloc((n1 + n2) * sizeof(int));
   nLR = 0; i = j = k = 0;
   while ((i < n1) && (j < n2)) {
      if ((j == n2) || (L[i] <= L[j])) { B[k] = L[i]; ++k; ++i; }
      else { B[k] = L[j]; nLR += n1 - i; ++k; ++j; }
   }
   for (k=0; k<n1+n2; ++k) L[i] = B[i];
   free(B);
   return nLR;
}

int ninv ( int *A, int n )
{
   int n1, n2, cnt;

   n1 = (n + 1) / 2; n2 = n - n1;
   cnt = ninv(A,n1);
   cnt += ninv(A+n1,n2);
   cnt += merge(A,n1,A+n1,n2);
   return cnt;
}
```

The computation of the inversion counts adds only a $\Theta(n)$ overhead to `merge` (and $O(1)$ overhead to `ninv`). Therefore, the running time satisfies

$$T(n) = 2T(n/2) + \Theta(n).$$

By the master theorem of divide-and-conquer recurrences, we have $T(n) = \Theta(n \log n)$. The function sorts A as a side effect. If this is undesirable, one first copies A element-by-element to another array and computes the number of inversions in the copy. This has only a $\Theta(n)$ added overhead.

Roll no: ____________________ **Name:** ____

5. You are given an array of n integers $a_0, a_1, a_2, \ldots, a_{n-1}$. Your task is to find out the count k of mutually distinct values that appear in A. Of course, if you sort A, you can obtain this count in $O(n \log n)$ time. Propose an algorithm to determine k in $O(n \log k)$ time. (If $k \ll n$, then this is a faster algorithm than the algorithm based upon sorting.) **(10)**

Solution This is an algorithm which makes a careful use of data structures. It uses a height-balanced BST (like an AVL tree) T which is initialized to the empty tree. The array elements $a_0, a_1, \ldots, a_{n-1}$ are inserted one by one (in any order) in T. We assume that T does not support repeated existence of the same item. After the n insertions (or insertion attempts), the count of nodes in T is computed and returned.

If A contains exactly k distinct values, then T never contains more than k nodes. Since T is height-balanced, its height always remains $O(\log k)$. Therefore, n insertions take a total of $O(n \log k)$ time. The number of nodes in T (after all insertions) can be computed by a $\Theta(k)$ traversal in T (done in class).

6. In this exercise, you design a max-heap H (more precisely, a max priority queue) that supports $O(1)$-time findMin, and $O(\log n)$-time deleteMin operations (where n is the number of elements stored in H). H is represented as an array with each cell storing a structure of two components: a value (val) that follows max-heap ordering, and the minimum (min) of all the values stored in the subtree rooted at that node. During makeHeap, insert and deleteMax, the minimum values can be updated suitably so that the running times of these functions increase only by constant factors (you do not have to rewrite these functions). For this array H, implement findMin to run in $O(1)$ time and deleteMin to run in $O(\log n)$ time. **(10)**

Solution FindMin should return the min value stored at the root (**`H[0].min`**).

For deleteMin, we start from the root, read the min value m stored there, and traverse a root-to-leaf path to a leaf storing this minimum value m. In each step, we choose the child for which the minimum is the same as m. We delete this leaf by copying the last leaf at that position. But then, there are two tasks remaining. First, we have to move up the deletion path (that located the leaf) in order to restore the max-heap-ordering property. Second, we have to adjust the min values on all the nodes on the deletion path. Moreover, the parent of the last leaf loses a child, so the min values on the root-to-last-leaf path should also be recalculated (unless the deleted leaf is the last leaf, in which case the root-to-last-leaf path is the same as the deletion path). No nodes outside these two paths are affected by deleteMin, so we do not need to recalculate their val and min fields. The following code snippet elaborates the deleteMin operation.

```
i = 0;
while (1) {                /* Follow the root-to-leaf path to a minimum leaf */
   l = 2*i + 1; r = 2*i + 2;                          /* Indices of two children */
   if (l >= n) break;                               /* Already reached a leaf node */
   if (r >= n) i = l; else i = (H[l].min <= H[r].min) ? l : r;
                                          /* Go to the child storing the smaller min value */
}

/* Here i stores the index of the minimum leaf to be deleted */
if (i < n-1) H[i] = H[n-1];   /* Copy the last leaf to the deletion position */
else i = (n-1)/2;      /* The last leaf is deleted, so start from its parent */
j = i;                              /* Remember the end of the deletion path */

if (i != (n-1)/2) {                            /* If the last leaf is not deleted */
   while (1) {                             /* Loop for restoring max-heap property */
      if (i == 0) break;                                /* Already reached root */
      p = (i-1)/2;                                          /* Index of parent */
      if (H[p].val >= H[i].val) break;          /* Max-heap ordering restored */
      t = H[p]; H[p] = H[i]; H[i] = t;                      /* Swap with parent */
      i = p;
   }
}

i = j;       /* Restore i to the index of the last node on the deletion path */
while (1) {    /* Loop for recalculating the min values on the deletion path */
   l = 2*i + 1; r = 2*i + 2;       /* Indices of the left and right children */
   if (l >= n) H[i].min = H[i].val;                   /* Minimum at a leaf node */
   else if (r >= n) H[i].min = H[l].min;        /* Only the left child exists */
   else H[i].min = (H[l].min <= H[r].min) ? H[l].min : H[r].min;
                        /* Smaller of the min values stored in the two children */
   if (i == 0) break;                           /* Upward journey ends at the root */
   i = (i-1)/2;                                             /* Move to the parent */
}

if (j != (n-1)/2) {       /* Adjust min values on the root-to-last-leaf path */
   ...                 /* Repeat the last loop with i initialized to (n-1)/2 */
}

--n;                               /* Finally, decrease the heap size by one */
```

For rough work and leftover answers