Sample Solution to Problem Set 4

1. (4 points) Insertion and deletion in binary search trees

Is the operation of insertion in binary search trees "commutative" in the sense that inserting x and then y into a binary search tree leaves the same tree as inserting y and then x. Argue why it is or give a counterexample. Similarly, determine whether the operation of deletion is commutative.

Answer: The operation of insertion is not commutative. Consider inserting x then y. We get x as the root, and y as its child. Now consider inserting y then inserting x. We get y as the root and x as its child.

The operation of delete is also not commutative.

2. (8 points) Radix trees

Problem 12-2, pages 269–270.

Answer: We use a radix tree to maintain the strings in S. For this, we define two operations: INSERT and TREE-WALK. Both of these are analogous to the operations defined for binary search trees.

For inserting string $a = a_0 a_1 \ldots a_p$, we traverse the radix tree using the bits of a in the order a_0, a_1, ..., a_p. As in the search procedure, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. If we reach a NIL node, then we create a new node and continue. Finally, we store the string a at a leaf node or at an unoccupied node, when we have considered the bit a_p. The insertion of a string takes time proportional to its length. Thus, the total time taken for inserting all the strings is $\Theta(n)$. We also note that the total number of edges in the radix tree after all of the strings have been inserted exactly equals n.

For the TREE-WALK procedure, we proceed exactly as the procedure PREORDER-TREE-WALK, the only difference being that we only print those nodes which are occupied by the strings. This traversal also takes $\Theta(n)$ time since the number of edges in the tree is $\Theta(n)$.

3. (6 points) Hashing techniques

Exercise 11.4–1, page 244.

Answer: The three hashing techniques yield the following results:

- **Linear probing:** The key 10 goes into location 10, 22 into 0, 31 into 9, 4 into 4, 15 after colliding with 4 goes into location 5, and so on. The final result is:

$$[22, 88, , 4, 15, 28, 17, 59, 31, 10,].$$

- **Quadratic probing:**

$$[22, , , 4, 88, 59, 28, 15, 31, 10, 17].$$

- **Double hashing:**

$$[22, 59, 17, 4, 15, 28, 88, , 31, 10,].$$

4. (5 + 5 = 10 points) Quadratic probing

Problem 11-3, pages 250-251.

Answer:

- For a given key k, the ith probe is into index $h(k) + 1 + 2 + \ldots + i \bmod m$, which equals $h(k) + i^2/2 + i/2 \bmod m$. Thus, the probing technique is quadratic probing with coefficients $1/2$ and $1/2$, respectively.
- We argue that given any two distinct integers $0 \leq i, j < m$, the probes are distinct. Note that this is sufficient to prove the desired claim since the m distinct values of i in the probe sequence will then yield the m distinct indices of the hash table. For $h(k, i)$ to be equal to $h(k, j)$, we need to have $i^2/2 + i/2 = j^2/2 + j/2 \bmod m$. In other words, m must divide the difference in these two terms, which is $(i - j)(i + j + 1)/2$. Can this ever happen?

  We claim not. We are given that m is a power of 2. One of the two terms $i - j$ or $i + j + 1$ has to be odd – if one of i and j is odd, then $i - j$ is odd, otherwise $i + j + 1$ is odd. So m can share common factors with only one of the two terms $i - j$ or $i + j + 1$. Thus, in order for m to divide $(i - j)(i + j + 1)/2$, m has to divide either $i - j$ or $(i + j + 1)/2$. Now, $|i - j|$ is smaller than m – because i and j are both smaller than m – so m cannot divide $i - j$. Similarly $(i + j + 1)/2$ is smaller than m, so m cannot divide $(i + j + 2)/2$ either. This completes the proof.

5. (5 + 6 + 1 = 12 points) Coin changing

Problem 16-1, parts (a), (b), and (c), page 402.

Answer:

(a) and (b) The greedy algorithm is the following.

1. *remaining* $\leftarrow n$; $S \leftarrow \emptyset$
2. while *remaining* $\neq 0$ do
3. Let c be the largest coin less than or equal to n
4. Add c to S
5. *remaining* $\leftarrow$ *remaining* $- c$

We note step 3 is always defined since 1 is one of the denominations.

(In part (a), for example, if our solution consists of m_1 quarters, m_2 dimes, and m_3 pennies, it can be easily seen that $m_1 = \lfloor n/25 \rfloor$, $m_2 = \lfloor (n - 25m_1)/10 \rfloor$, and $m_3 = n - 25m_1 - 10m_2$.)

We prove the correctness of the greedy algorithm by establishing the optimal substructure property and the greedy choice property. For completeness, we provide a complete formal argument here.

Optimal substructure property: Here we need to prove that if S is an optimal solution for making change for n cents and c be any coin in S, then $S - \{c\}$ is an optimal solution for making change for $n - c$ cents. The proof is by contradiction. Let S' be an optimal solution for $n - c$ cents. Thus, $S' + \{c\}$ is a valid solution for n cents. If $S - \{c\}$ is not an optimal solution for $n - c$ cents, then the number of coins in $S - \{c\}$ is at least one more than that in S'. This implies $S' + \{c\}$ has fewer coins than S, hence contradicting the fact that S is an optimal solution for n cents. (Note that this proof holds no matter what the coin denominations are.)

Greedy choice property: Here we need to prove that if c is the largest coin denomination less than or equal to n, then there exists an optimal solution for n that contains a c coin. The proof proceeds as follows. Let S be any optimal solution for making change for n cents. If S contains a c coin, then there is nothing left to proof. If not, then let S' be the smallest subset of coins in S (smallest in terms of the total value of the coins) that add up to at least c. Since $n \geq c$, S itself is a candidate for S'; therefore, S' exists. By our assumption, S' does not have c. Let c' be the smallest coin denomination in S'. We have $c' < c$. Let the total value of S' be n'. Then, it follows that $n' - c'$, which is the total value of $S' - \{c'\}$, is less than c. This is because otherwise $S' - \{c'\}$ is a subset of S whose coins add up to at least c, thus contradicting the assumption that S is the smallest such set. We thus have $n' \geq c$ and $n' - c' < c$

We first consider the case in which all coins larger than c' are a multiple of c'. In this case, n' is a multiple of c'. Since $c \leq n' < c + c'$, the only multiple of c' that lies in $[c, c + c')$ is c. Therefore, n' equals c. We can now replace the set S' in the solution S by a single coin of denomination c and obtain a solution for n that has no more coins than S. Thus, the desired claim holds.

The above argument takes care of part (b) since it is true for each coin c^i that the coins bigger than c^i are all multiple of c^i.

The above argument also takes care of the situation when c' equals 1 or 5 for part (a). The only case that remains is when c' equals 10. In this case, c has to be 25, since $c > c'$. If c' equals 10, then $25 \leq n' < 35$. Since n' in this case is a multiple of 10, the only possibility for S' is 3 dimes, which we can replace by a quarter and a nickel and contradict the fact that S was on optimal solution for n cents.

The optimality of the greedy algorithm can now be shown by induction on n. The base case, $n = 0$, is trivial. For the induction hypothesis, we assume that the greedy algorithm is optimal for $n < m$. We now consider the case $n = m$. The greedy algorithm proceeds by picking the largest coin less than or equal to m and adding it to the solution for $m - c$. By the greedy choice property, there exists an optimal solution for m that contains c. Let us say this solution has x coins. By the optimal substructure property, the number of coins in the optimal solution for $m - c$ is $x - 1$. Since the greedy algorithm obtains the optimal solution for $m - c$ (by the induction hypothesis), it follows that the number of coins in the solution returned by the greedy algorithm equals 1 more than the number of coins in the optimal solution for $m - c$, i.e. $1 + x - 1 = x$. This completes the proof.

(c) A set of coin denominations for which the greedy algorithm does not yield an optimal solution

is pennies, dimes, and quarters. While making change for 30 cents, the greedy algorithm will yield 1 quarter and 5 pennies, that is, a total of 6 coins. The optimal solution, however, is 3 dimes.

6. (10 points) Human resource allocation

Suppose you are a manager in a construction firm and you are managing n building projects. You are asked to assign m of the engineers in your firm among these n projects. Assume for simplicity that all of the engineers are equally competent.

After some careful thought, you have figured out how much benefit i engineers will bring to project j. View this benefit as a number. Formally put, for each project j, you have computed an array $A_j[0..m]$ where $A_j[i]$ is the benefit obtained by assigning i engineers to project j. Assume that $A_j[i]$ is nondecreasing with increasing i. Further make the (plausibly sound) assumption that the marginal benefit obtained by assigning an ith engineer to a project is nonincreasing as i increases. Thus, for all j and $i \geq 1$, $A_j[i+1] - A_j[i] \leq A_j[i] - A_j[i-1]$.

Design a greedy algorithm to determine how many engineers you will assign to each project such that the total benefit obtained over all projects is maximized. Justify the correctness of your algorithm and analyze its running time.

Answer: We assign engineers one at a time, from 1 to m. At the ith step, we greedily assign the ith engineer to the project to which the engineer will bring the most marginal benefit. Note that the marginal benefit that the ith engineer brings to a project j depends on the particular assignment of the previous $i-1$ engineers; it may not be the same as $A_j[i]$.

Correctness proof: We now prove that the above greedy choice leads to an optimal solution using a standard "swapping argument". Our main claim is this: for all i, the assignment obtained after the ith step is the subset of an optimal assignment. We prove the claim by induction.

The induction basis is trivial since we start with an empty assignment which is clearly a subset of an optimal assignment. As the induction hypothesis, we assume that the claim holds after step $i-1$. That is, the assignment obtained after step $i-1$ is a subset of an optimal assignment, say S. We now consider the state after step i. Let us suppose that the greedy choice assigns the ith engineer to project j, thus resulting in k engineers for project j. If S has assigned at least k engineers to project j, then there is nothing to prove. (Note that by the induction hypothesis, S has assigned at least $k-1$ engineers to project j.) Consider the case in which S has assigned only $k-1$ engineers. This implies that there must be some other project j' such that S has assigned more engineers than the greedy assignment has assigned thus far. By the definition of our greedy choice and the assumption about marginal benefits, it follows that if we modify S by removing one engineer from project j' and move it to j, the change in total benefit cannot decrease. Therefore, the new assignment thus obtained by modifying S is optimal. It also contains the current greedy assignment, thus establishing the induction step.

Running time: We now consider the running time of the greedy algorithm. We have m iterations and in each iteration we are determining the project for which the marginal benefit will be maximum. Thus, a naive n-element comparison to determine the maximum will take $O(n)$ time, leading to a runtime of $\Theta(nm)$. One can do better by maintaining a priority queue in which we store an element for each project, representing the marginal benefit the next engineer will bring to the

project. Initially, we add n entries to a priority queue, one for each of the projects, representing the marginal benefit that the first engineer brings to the project. As we have seen in class (and Chapter 6), we can implement this in $O(n)$ time using BUILD-HEAP. Each iteration then translates to one Extract-Max operation (to find the project to which the engineer will be assigned) and an insert operation to insert a new element giving the marginal benefit the next engineer will bring to the chosen project. Each iteration thus takes $O(\log n)$ time, thus leading to a total runtime of $O(n + m \log n)$.