Illinois Institute of Technology
Department of Computer Science

Homework 5 Solutions

CS 430 Introduction to Algorithms
Spring Semester, 2014

1. **Problem 1 Solution:**

   (a) The proof is inductive and constructive. The induction is on the number of elements in the treap. The base case of 0 nodes is the empty treap. We now suppose that the claim is true for $0 \leq k < n$ nodes, and show it is true for n. Given a set of n nodes with distinct keys and priorities, the root is uniquely determined - it must be the node with the smallest priority in order for the min-heap property to hold. Now suppose this root node has key r. All elements with $key < r$ must be in the left subtree, and all elements with $key > r$ must be in the right subtree. Notice that each of these sets has at most $n - 1$ keys, and therefore has a unique treap by the inductive assumption. Thus, for the n-node treap, the root is uniquely determined, and so are its subtrees. Therefore, the treap exists and is unique.

   Notice this suggests an algorithm for finding the unique treap: choose the element with the smallest priority, partition the remaining elements around its key, and recursively build the unique treaps from the elements smaller than the root key (these will go in the left subtree) and the elements larger than the root key (right subtree).

   (b) The expected height of a randomly built binary search tree is $O(\log n)$. We show that constructing the unique treap is equivalent to inserting the elements into a binary search tree in random order.

   Suppose we insert the elements into a BST in order of increasing priority. Normal BST insertion will maintain the binary search tree property, so we only need to show that the min-heap property is maintained on the priorities. Suppose v is a child of u. The insertion procedure always inserts each element at the bottom of the tree, therefore v must have been inserted after u. Thus, the priority of v is greater than the priority of u, and we have built a treap. The priorites are assigned randomly, which means any permutation of the priorities is equally likely. When considering this as BST insertion, it translates into any ordering being equally likely, and therefore the expected height of a treap is equal to the expected height of a randomly built BST, which is $O(\log n)$.

   (c) Treap-Insert works by first inserting the node according to its key and the normal BST insertion procedure. This is guaranteed to maintain the first two conditions of the treap, since those correspond to BST properties. However, we may have violated the third condition, the min-heap ordering on the priorities. Note that (initially) this violation is localized to the node were inserting and its parent, since we always insert at the leaves. Rotations preserve BST properties, so we will use rotations to move our node x up as long as the min-heap ordering is violated, i.e. as long as priority[x] ¡ priority[parent[x]]. If x is a left child we will right-rotate it, and if its a right child we will left-rotate it. Notice that this preserves the heap property elsewhere in the treap, assuming that the children of x had higher priority than parent[x] prior to the rotation (to be completely formal we would have to prove this using a loop invariant). The pseudocode is as follows:

```
TREAP-INSERT(T, x, priority)
 TREE-INSERT(T, x)
while parent[x] != NIL  priority[x] < priority[parent[x]]
if left[parent[x]] == x
Right-Rotate(T, parent[x])
else
Left-Rotate(T, parent[x])
```

(d) TREAP-INSERT first performs a BST insert procedure which runs in time proportional to the height of the treap. Then, it rotates the node up one level until the min-heap property is satisfied. Thus, the number of rotations we perform is bounded by the height of the treap. Each rotation takes constant time, so the total running time is proportional to the height of the treap, which we showed in (b) to be expected $\theta(\log n)$.

(e) proof: using a loop invariant: after having performed k rotations during the insertion of x, $C_k + D_k = k$.
Initialization: After we insert x using BST-INSERT but before performing any rotations, x is a leaf and its subtrees are empty, so $C_0 = D_0 = 0$.
Maintenance: Assume that we have performed k rotations on x and that $C_k + D_k = k$. Now, if we perform a right-rotation on x, the left child of x remains the same, so $C_{k+1} = C_k$. The right child of x changes from to y with subtrees (left) and Γ (right) using the notation from the book. The left spine of the right child used to be the left spine of and now it is y plus the left spine of . Therefore, $D_{k+1} = D_k + 1$. Thus, $C_{k+1} + D_{k+1} = C_k + D_k + 1 = k + 1$, which is precisely the number of rotations performed. The same holds for left-rotations, where we can show that $D_{k+1} = D_k$ and $C_{k+1} = C_k + 1$.
Termination: After TREAP-INSERT is finished, the number of rotations we performed is equal to $C + D$, precisely the condition that needed to be shown.

(f) First, assume $X_{i,k} = 1$. We will prove $priority[y] > priority[x]$, $key[y] < key[x]$, and $\forall z$ such that $key[y] < key[z] < key[x]$, we have $priority[y] < priority[z]$. The first property follows from the min-heap property on priorities, since y is a descendant of x. The second follows from the BST property, since y is in the left subtree of x. Finally, consider any node z satisfying $key[y] < key[z] < key[x]$, and imagine an inorder tree walk on the treap. After y is printed, we will print the right subtree of y, at which point the entire left subtree of x will have been printed since y is in the right spine of that subtree. Thus, the only nodes printed after y but before x are in the right subtree of y; therefore, z must be in the right subtree of y and by the min-heap property $priority[y] < priority[z]$.

Now, we will assume the three properties and prove that $X_{i,k} = 1$. First consider the possibility that y is in the left subtree of x but not in the right spine of that subtree. Then there exists some node z in the spine such that going left from z will lead to y. Note that this z satisfies $key[y] < key[z] < key[x]$, but $priority[z] < priority[y]$ which violates the third property. Clearly y cannot be in the right subtree of x without violating the second property, and x cannot be a descendant of y without violating the first property. Suppose that x and y are not descendants of each other, and let z be their common ancestor. Again we have $key[y] < key[z] < key[x]$ but $priority[z] < priority[y]$, violating the third property. The only remaining possibility is that y is in the right spine of the left subtree of x, i.e. $X_{i,k} = 1$.

(g) Assume that $k > i$. $PrX_{i,k} = 1$ is the probability that all the conditions in part (f) hold. Consider all the elements with keys $i, i+1, ..., k$. There are $k - i + 1$ such elements. The values of their priorities can be in any order, so there are $(k - i + 1)!$ possible (and equally likely) permutations. In order to satisfy our conditions, we need $priority[z] > priority[y] > priority[x]$ for all $z \in \{i+1, i+2, ..., k-1\}$. This fixes the priorities of x and y to be the lowest two priorities, allowing $(k-i-1)!$ permutations of the remaining priorities among the elements with keys in $i + 1, ..., k - 1$. The probability of $X_{i,k} = 1$ is the ratio of these two, which gives $Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!}$. Most of the terms in the factorials will cancel, except for the first two terms in the denominator. This leaves us with $PrX_{i,k} = 1 = \frac{1}{(k-i+1)(k-i)}$

(h) The expected value for C, which is the number of elements in the right spine of the left subtree of x is simply the expectation of the sum of $X_{i,k}$ over all $i < k$. This is
$E[C] = E[\sum_{i=1}^{k-1} X_{i,k}]$
$= \sum_{i=1}^{k-1} E[X_{i,k}]$
$= \sum_{i=1}^{k-1} Pr[X_{i,k} = 1]$
$= \sum_{i=1}^{k-1} \frac{1}{(k-i+1)(k-i)}$

$= \frac{1}{(k)(k-1)} + \frac{1}{(k-1)(k-2)} + ... + \frac{1}{(2)(1)}$
$= \sum_{j=1}^{k-1} \frac{1}{(j+1)(j)}$
$= \sum_{i=1}^{k-1} (1/j - 1/(j+1))$
$= 1 - 1/k$

(i) Note that the expected length of the right spine of the left subtree of x depends only on the rank k of the element x. The expected length of the left spine of the right subtree will have the same expected value with respect to the rank of x in the reverse ordering, which is $n - k + 1$. Thus, $E[D] = 1 - 1/(n - k + 1)$.

(j) Since the number of rotations equals $C + D$, the expected number of rotations equals $E[C + D] = E[C] + E[D] = 1 - 1/k + 1 - 1/(n - k + 1) \leq 2$. So, the expected number of rotations is less than 2.

2. **Problem 2**. **Solution:** Suppose that

$$h_1(k) + jh_2(k) = h_1(k) + ih_2(k) (\text{mod } m)$$
$$(j - i)h_2(k) = 0 (\text{mod } m)$$
$$(j - i)h_2(k) = c \cdot m \text{ where } c \text{ is some integer}$$

If $h_2(k)$ and m are not relatively prime, then the above equation holds. Thus the hash function repeats at the i-th and the j-th step, and hence the hashing function will not generate m different locations.

3. **Problem 3**. **Solution:**

(a) From how the probe-sequence computation is specified, it is easy to see that the probe sequence is $< h(k), h(k)+1, h(k)+1+2, ..., h(k)+1+2+3+..+i, ... >$, where all the arithmetic is modulo m. Starting the probe numbers from 0, the ith probe is offset (modulo m) from $h(k)$
by
$\sum_{j=0}^{i} j = \frac{i(i+1)}{2} = \frac{1}{2}i^2 + \frac{1}{2}i$
Thus, we can write the probe sequence as $h'(k, i) = (h(k) + \frac{1}{2}i^2 + \frac{1}{2}i) mod m$, which demonstrates that this scheme is a special case of quadratic probing.

(b) Let $h'(k, i)$ denote the ith probe of our scheme. We saw in part (a) that $h'(k, i) = (h(k) + \frac{i(i+1)}{2}) mod m$ To show that our algorithm examines every table position in the worst case, we show that for a given key, each of the first m probes hashes to a distinct value. That is, for any key k and for any probe numbers i and j such that $0 \leq i < j < m$, we have $h'(k, i) \neq h'(k, j)$. We do so by showing that $h'(k, i) = h'(k, j)$ yields a contradiction.

Let us assume that there exists a key k and probe numbers i and j satsifying $0 \leq i < j < m$ for which $h'(k, i) = h'(k, j)$. Then
$(h(k) + \frac{i(i+1)}{2}) \equiv (h(k) + \frac{j(j+1)}{2}) (mod m)$ which in turn implies that $\frac{i(i+1)}{2} \equiv \frac{j(j+1)}{2} (mod m)$,
$= \frac{j(j+1)}{2} - \frac{i(i+1)}{2} \equiv 0 (mod m)$,
$= (j - i)(j + i + 1)/2 \equiv 0 (mod m)$
The factors $j - i$ and $j + i + 1$ must have different parities, i.e., $j - i$ is even if and only if $j + i + 1$ is odd. Since $(j - i)(j + i + 1)/2 \equiv 0 (mod m)$, we have $(j - i)(j + i + 1)/2 = rm$ for some integer r or, equivalently, $(j - i)(j + i + 1)/2 = r.2m$. Using the assumption that m is a power of 2, let $m = 2^p$ for some nonnegative integer p, so that now we have $(j - i)(j + i + 1) = r.2^{p+1}$. Because exactly one of the factors $j - i$ and $j + i + 1$ is even, 2^{p+1} must divide one of the factors. It cannot be $j - i$, since $j - i < m < 2^{p+1}$. But it also cannot be $j + i + 1$, since $j + i + 1 \leq (m - 1) + (m - 2) + 1 = 2m - 2 < 2^{p+1}$. Thus we have derived the contradiction that 2^{p+1} divides neither of the factors $j - i$ and $j + i + 1$. We conclude that $h'(k, i) \neq h'(k, j)$.

4. **Problem 4**. **Solution:**

(a) If we execute a long process first, all the processes excuted after it will have longer completion times. So it is wiser to have shorter jobs executed first.
Greedy Algorithm:
Sort the processes in the ascending order of their processing times and execute them in that order. Sorting takes $O(n \log n)$ time and executing them takes $O(n)$ time so total running time is $O(n \log n)$
Proof of correctness:
BY contradiction: say a_m is the process with the shortest process time. Let the optimal process schedule be $a_1, a_2, ..a_m, .., a_n$ where the first job has larger process time.
Now if we exchange the a_m with a_1 the completion times improve. As the completion times for the rest of the elements remains the same we can conclude that having a_m first gives a better schedule so this gives a contradiction. Optimal Substructure: Once the shortest job is scheduled, the rest of the jobs form a sub-problem which again have to scheduled optimally in order for the big problem to have an optimal solution.

(b) At any time t, select the one task with least p_i from the tasks whose release time $r_i \leq t$, and assign this task to run. Because preemption is allowed we can do this assignment freely. The running time is $O(T * n)$, T is the total time for running all n tasks.

5. **Problem 5**. **Solution:**

(a) Say a_m is the process with the largest penalty and has . Let the optimal process schedule be $a_1, a_2, .., a_i, ..a_m, .., a_n$ where the ith job has smaller penalty and is the deadline d_m.
Now if we exchange the a_m with a_i the penalties improve. As the completion times for the rest of the elements remains the same we can conclude that having a_m instead of a_i gives a better schedule so this gives a contradiction. Now suppose a_m is replaced with the first job a_1 then there is no improvement in penalties but there might be additional penalty depending on the deadline of a_1. So, the current solution still holds. If a job a_j has no slot left before its deadline then it does not matter where we place it after, so putting it at the end might give another job a chance to complete its deadline. Optimal Substructure: Once the highest penalty job is scheduled, the rest of the jobs form a sub-problem which again have to scheduled optimally in order for the big problem to have an optimal solution.

(b) Consider the processes to be nodes of graph each of which has a marker variable say $m_i = white$, after each of the process is inserted in the right position it we mark the node black and it points to a root node which has the leftmost node in its memory. So, we first do a MAKE-SET operation, then we do a UNION operation when each new node is added to the tree. We use FIND-SET operation to find the right node.

6. **Problem 6**. **Solution:**

(a) We how that MST is unique when all edges have distinct weight by contradiction:
Let us assume a Minimum Spanning Tree say T is not unique. Then there is another spanning tree S of equal weight as T. Let an edge e be in Tree T but not in S. Adding the edge e in MST S will make a cycle C. Cycle C in S will contain at least one edge e' which will not be in MST T. Let us assume that the weight of edge e is less than edge e'. So we can replace e' with e in MST S now giving an another tree say S' with smaller weight. Above contradicts our assumption that S is a MST.

(b) In the prrof given in the previous example assume that e and e' have the same cost and that they are the minimum weighted edges, so now we have 2 MSTs T and S both with the same weight now depending on the ordering of the algorithm we get either T or S.

7. **Problem 7**. **Solution:**

```
ChangeCoin(V,d[])
{
```

```
Declare an array change[] of size equal to the number of different denominations i.e. 4
for (i =1 to change.size)
change[i] = 0; // Initialize the change array to zero
for (i=1 to d.size)
{
if (V > 0)
{
change[i] = V / d[i];
V = V%d[i];
}
else
break;
}
return change[];
}
```