CPS 130 Homework 11-15

Hashing, red-black trees, augmented search trees, dynamic programming

Write and justify your answers in the space provided.[1]

Hashing

1. (CLRS 11.1-4) We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT and DELETE should take $O(1)$ time each; and the initialization of the data structure should take $O(1)$ time.

   (Hint: Use an additional stack, whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

---

[1] Collaboration is allowed, even encouraged, provided that the names of the collaborators are listed along with the solutions. Students must write up the solutions on their own.

Red-Black Trees

2. (CLRS 13.1-5) Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf

3. (CLRS 13.1-6) What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

4. (CLRS 13-2) The **join** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $key[x_1] \leq key[x] \leq key[x_2]$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

   (a) Given a red-black tree T, we store its black-height as the field $bh[T]$. Argue that this field can be maintained by RB-INSERT and RB-DELETE without requiring extra storage in the tree and without increasing the asymptotic running times. Show while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

   We wish to implement the operation RB-JOIN(T_1, x, T_2) which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

   (b) Assume without loss of generality that $bh[T_1] \geq bh[T_2]$. Describe an $O(\lg n)$ time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $bh[T_2]$.

   (c) Let T_y be the subtree rooted at y. Describe how T_y can be replaced by $T_y \cup \{x\} \cup T_2$ in $O(1)$ time without destroying the binary-search-tree property.

   (d) What color should we make x so that red-black properties $1, 2,$ and 4 are maintained? Describe how property 3 can be enforced in $O(\lg n)$ time.

   (e) Argue that the running time of RB-JOIN is $O(\lg n)$

Augmented Search Trees

5. (CLRS 14.1-5) Given an element x in an n-node order-statistic tree and a natural number i, how can the ith successor of x in the linear order of the tree be determined in $O(\log n)$ time?

6. In this problem we consider a data structure for maintaining a multi-set M. We want to support the following operations:

   - $\mathit{Init}(M)$: create an empty data structure M.
   - $\mathit{Insert}(M, i)$: insert (one copy of) i in M.
   - $\mathit{Remove}(M, i)$: remove (one copy of) i from M.
   - $\mathit{Frequency}(M, i)$: return the number of copies of i in M.
   - $\mathit{Select}(M, k)$: return the k'th element in the sorted order of elements in M.

   If for example M consists of the elements

   $$< 0, 3, 3, 4, 4, 7, 8, 8, 8, 8, 9, 11, 11, 11, 11, 13 >$$

   then $\mathit{Frequency}(M, 4)$ will return 2 and $\mathit{Select}(M, 6)$ will return 7.

   Let $|M|$ and $\|M\|$ denote the number of elements and the number of *different* elements in M, respectively.
   a) Describe an implementation of the data structure such that $\mathit{Init}(M)$ takes $O(1)$ time and all other operations take $O(\log \|M\|)$ time.

   b) Design an algorithm for sorting a list L in $O(|L| \log \|L\|)$ time using this data structure.

Dynamic Programming

7. A game-board consists of a row of n fields, each consisting of two numbers. The first number can be any positive integer, while the second is 1, 2, or 3. An example of a board with $n = 6$ could be the following:

17	2	100	87	33	14
1	2	3	1	1	1

The object of the game is to jump from the first to the last field in the row. The top number of a field is the cost of visiting that field. The bottom number is the maximal number of fields one is allowed to jump to the right from the field. The cost of a game is the sum of the costs of the visited fields.

Let the board be represented in a two-dimensional array $B[n, 2]$. The following recursive procedure (when called with argument 1) computes the cost of the cheapest game:

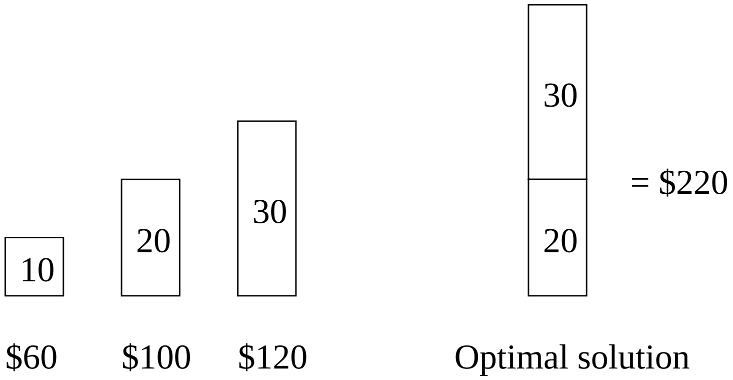
```
Cheap(i)

     IF i>n THEN return 0
     x=B[i,1]+Cheap(i+1)
     y=B[i,1]+Cheap(i+2)
     z=B[i,1]+Cheap(i+3)
     IF B[i,2]=1 THEN return x
     IF B[i,2]=2 THEN return min(x,y)
     IF B[i,2]=3 THEN return min(x,y,z)

END Cheap
```

(a) Analyze the asymptotic running time of the procedure.

(b) Describe and analyze a more efficient algorithm for finding the cheapest game.

8. In this problem we consider the 0-1 KNAPSACK PROBLEM: Given n items, with item i being worth $v[i]$ dollars and having weight $w[i]$ pounds, fill a knapsack of capacity m pounds with the maximal possible value.

Example: Given a knapsack of capacity 50, the maximal value obtainable with three items of value \$60, \$100, and \$120 and weights 10, 20, and 30, respectively, is \$220.



The algorithm `Knapsack(i,j)` below returns the maximal value obtainable when filling a knapsack of capacity j using items among items 1 through i (`Knapsack(n,m)` solves our problem). The algorithm works by recursively computing the best solution obtainable *with* the last item and the best solution obtainable *without* the last item, and returning the best of them.

```
Knapsack(i,j)

  IF w[i] <= j THEN
    with = v[i] + Knapsack(i-1, j-w[i])
  ELSE
    with = 0
  END IF
  without = Knapsack(i-1,j)
  RETURN max{with, without}

END Knapsack
```

(a) Show that the running time T of `Knapsack`(n, m) is exponential in n or m. (*Hint:* look at the case where $w[i] = 1$ for all $1 \leq i \leq n$ and show that $T(n, m) = \Omega(2^{\min(m,n)})$).

(b) Describe an $O(n \cdot m)$ algorithm for computing the value of the optimal solution.