Homework 2 Solutions

You may assume that `add, delete, search` operations take $O(1)$ time with Hash Tables. For all algorithm/data structure design problems, part of the grade depends on efficiency.

1. (25 pts) Let $S = \{44, 21, 10, 22, 12, 3, 19, 1, 4\}$.
   - Draw the Binary Search Tree obtained by inserting the elements of S in the order they appear above.
   - Draw the hash table (using chain hashing) obtained by adding elements of S into a hash table of size 5. Use $(7x + 2)\%5$ as the hash function.

2. (30 pts)

   (a) Given any set S of unique integers, and an integer $x \in S$, $succ(x)$ is defined to be the smallest element in S that is larger than x. If there is no such element, then $succ(x)$ returns a null value. Suppose that S is the set of keys of a hash table H that is implemented using chaining. Give an algorithm that, given H and x as inputs, returns $succ(x)$. Report the run-time of your algorithm.

   Ans. Recall that H is an array and $H[i]$ points to a linked list. Consider the following algorithm:

```
len = H.length;
Min = Infinite;
for i = 0 to len-1 {
   Traverse the list at H[i];
   If we find a y such that y > x and y < Min, then Min = y;
 }
 Output Min.
```

   Note we can traverse a list of size m in time $O(m)$. To traverse an empty list we need constant 1 time. Let ℓ_i be the size of the list at $H[i]$. In the ith iteration, we are traversing the list at $H[i]$, and this traversal takes $\max\{\ell_i, 1\}$ time. Thus the total time taken is

$$\sum_{i=0}^{SizeOfH-1} \max\{\ell_i, 1\}$$

   which is at most

$$\sum_{i=0}^{SizeOfH-1} [\ell_i + 1]$$

which is

$$\sum_{i=0}^{SizeOfH-1} \ell_i + \sum_{i=0}^{SizeOfH-1} 1$$

Since n elements are stored in the hash table the total size all lists (thus the sum of all ℓ_is) is n.

Thus the time is $O(n + SizeOfH)$, which is $O(n)$ since hash tables are maintained such that size of the table is $O(n)$.

(b) Using the *succ* function above, here is a strategy for constructing an ordered array of keys

```
n = number of elements in H
arr = new array of length n
x = -infinity
i = 0
while i < n
  x = succ(x)
  arr[i++] = x
```

What is the runtime of this algorithm? What would be a better way to produce an ordered array of keys?

Ans. Each iteration of the loop makes a call to *succ* which takes $O(n)$ time, and the loop is repeated n times. Thus the time taken by the algorithm is $O(n^2)$. A better algorithm would be to copy all the elements of the hash table into an array and sort the array. The following algorithm copies elements of the hash table to Array.

```
len = H.length;
A is an array of size n
counter = 0;
for i = 0 to len-1 {
   ListSize = length of the list at H[i]
  Copy the element of the list to A[counter], A[counter+1] ....A[counter+ListSize]
  counter = counter + ListSize;
    }
 Return A;
```

Time taken by the ith iteration is $O(\ell_i + 1)$. By the same analysis as before, the total time taken by the above algorithm is $O(n)$. Time taken to sort is $O(n \log n)$. Thus we can sort elements in a hash table in $O(n \log n)$ time.

3. (25 pts) Let G be a graph with vertices identified by integers $\{1, 2 \cdots, n\}$. The edges of G can be represented as a set E of ordered pairs $\langle i, j \rangle$, representing that there is a directed edge from i to j. We define a graph G to be *undirected* if $\langle i, j \rangle \in E$ always implies that $\langle j, i \rangle$ is also in E. Suppose you are given G as an array of pairs representing all edges in E. Give an algorithm to detect whether the graph is undirected or not. Report the run-time of your algorithm.

Ans. Place all edges in a hash table. For every $\langle i, j\rangle \in E$, search for $\langle j, i\rangle$ in the hash table. If the search ever returns "No", then the graph is not undirected. If the search always returns "Yes", then the graph is undirected. The time taken is $O(n)$.

4. (50 pts) This problem is about the array-based binary heap implementation of a priority queue. Let `H` denote the backing array, I.e H is a binary Min-heap Array. You should assume that the universe of possible keys is extremely large relative to the size of the heap (i.e., it is not practical to use a `Position` array as described on p. 65 of the text).

   (a) Give an algorithm for a new operation `decreaseKey(index, delta)` that subtracts `delta` from the key located at the given index in `H`, and restores the heap ordering. Report the time taken by the algorithm.

   Ans. Change $H[i]$ to $H[i] - \delta$ Now it is possible that $H[i]$ is smaller than its parent (which is at $H[\lfloor i/2 \rfloor]$. Thus make a call to $HeapifyUp(i)$. The time taken is $O(\log n)$.

   (b) Suppose we also want an operation `findKey(v)` that returns the index of a key `k` in H (if `k` appears more than once, then it does not matter which index is returned). Describe a modified implementation of the heap in which your `findKey(v)` operation is $O(1)$, finding the minimum is still $O(1)$, and the operations to insert a key or remove the minimum are still $O(\log n)$.

   Ans. Our new data structure consists of: A hash table T, a Binary Heap H, and a counter to keep track of number of elements stored in the heap. H is maintained as standard binary heap.

   The hash table is build such that it can store tuples of the form $\langle v, i\rangle$ where v is a key and i is an integer. Furthur we ensure that for every v there is a unique i such that $\langle v, i\rangle$ is in the hash table. Finally our hash table can be built using a hash function h that maps keys to integers.

   We assumed that the operations search, add and delete on hash tables take $O(1)$ time. This implies that size of the list at $T[i]$ is constant for every i. Consider the following function: $getVal(v)$, this returns unique i such that $\langle v, i\rangle$ is in the hash table. This can be implemented in $O(1)$ time as follows. Compute $h(v) = x$. Search the list at $T[x]$ for a tuple of the form $\langle v, i\rangle$ and return i. If no such tuple exists, then return Null. Since all lists are of constant size, the time taken is $O(1)$.

   We will ensure the following relationship between the hash table T and the Heap H: If $\langle v, i\rangle$ is in the table T, then $H[i] = v$.

   Recall that heap methods such as add, delete, min, extractmin depend on $HeapifyUp$ or $HeapifyDown$. We describe $HeapifyUp(i, y)$: The goal of this method is to replace $H[i]$ with y (if $y < H[i]$), and maintain the heap property. Set $H[i] = y$. If i equals 1, then $H[1]$ is y and the heap property is maintained (since $y < H[1]$). Let p be the index of the parent of i in the heap. Compute the minimum among $H[p], H[2p]$ and $H[2p+1]$. Note that i is either $2p$ or $2p+1$. Note that the minimum must be one of $H[p]$ or $H[i]$. If the minimum is $H[p]$, then the procedure stops. If the minimum is $H[i]$, then we we replace $H[i]$ with $H[p]$, delete $\langle H[p], p\rangle$ from the hash table and add $\langle H[p], i\rangle$ to the hash table. And call $HeapifyUp(p, y)$. Similarly we can define $HapifyDown(i, y)$ operation. Note that both the operations take $O(\log n)$ time. These operations can be used to add and element to the heap or extract minimum element from the heap.

We are now ready to describe the methods:

$Add(v)$. Place $\langle v, counter + 1\rangle$ in T and add v to the binary heap (by using the above described methods $HeapifyUp$ and $HeapifyDown$. Increment counter by 1. The time taken is $O(\log n)$.

$Delete(v)$: Call $getVal(v)$, if this returns i, then remove $\langle v, i\rangle$ from the hash table and delete $H[i]$ from the heap. The time taken is $O(\log n)$.

$findKey(v)$: Call $getVal(v)$. If it returns Null, then v is not in the heap/hash table. The time taken is $O(1)$.

$min()$. Returns $H[1]$. Time taken is $O(1)$.

$extractMin()$. Remove $\langle H[1], 1\rangle$ from the hash table. Decrement counter by 1. Perform extract Min operation on H (by using described methods $HeapifyUp$ and $HeapifyDown$. Time taken is $O(\log n)$.