Computer Science CSC263H | February 1, 2018
St. George Campus | University of Toronto

Solutions for Homework Assignment #2

Answer to Question 1.

a. A binomial heap H with n vertices consists of $\alpha(n)$ trees. Let T_i, $1 \le i \le \alpha(n)$, denote the trees of H. A tree T_i with n_i vertices has $n_i - 1$ edges. So the total number of edges in H is $\sum_{i=1}^{i=\alpha(n)} (n_i - 1) = (\sum_{i=1}^{i=\alpha(n)} n_i) - \alpha(n) = n - \alpha(n)$

b. Binomial heap H has n nodes before the insertions. Thus, by Part (a), H has $n - \alpha(n)$ edges before the insertions. After the k consecutive insertions, H has $n + k$ nodes, hence it now has $(n+k) - \alpha(n+k)$ edges. So the number of new edges created by the k consecutive insertions is:
$[(n+k) - \alpha(n+k)] - [n - \alpha(n)] = k + \alpha(n) - \alpha(n+k) \le k + \alpha(n)$ edges.

As we explained in class, the number of pairwise comparisons between the keys of H needed to execute k consecutive insertions is equal to the number of new edges created by these insertions: each key comparison creates a new edge in H and each new edge in H comes from a key comparison. So k consecutive insertions require at most $k + \alpha(n)$ key comparisons.

By definition $\alpha(n)$ is the number of 1's in the binary representation of n, therefore, $\alpha(n) \le \lfloor \log_2 n \rfloor + 1$. So k consecutive insertions require at most $k + \lfloor \log_2 n \rfloor + 1$ comparisons, and the average cost per insertion is at most $(k + \lfloor \log_2 n \rfloor + 1)/k = 1 + \lfloor \log_2 n \rfloor / k + 1/k$. Thus, for $k > \log n$, this average cost is less than 3.

Answer to Question 2. We define a procedure Check which takes a pointer u to a tree node and returns the following:

- (NIL, NIL) if the tree rooted at u is not 2-balanced;
- a pair of numbers (r, h), where $r = \text{radius}(u)$ and $h = \text{height}(u)$, if the tree rooted at u is 2-balanced.

The procedure follows:

```
Check(u)
if u == NIL
    return (−1, −1)
elseif u.lchild == NIL
    (r_R, h_R) = Check(u.rchild)
    if r_R == NIL
        return (NIL, NIL)
    else return (r_R + 1, h_R + 1)
elseif u.rchild == NIL
    (r_L, h_L) = Check(u.lchild)
    if r_L == NIL
        return (NIL, NIL)
    else return (r_L + 1, h_L + 1)
else (r_L, h_L) = Check(u.lchild)
    (r_R, h_R) = Check(u.rchild)
    if r_L == NIL or r_R == NIL
        return (NIL, NIL)
    r = min(r_L, r_R) + 1
    h = max(h_L, h_R) + 1
    if h ≤ 2r
        return (r, h)
    else return (NIL, NIL)
```

To check that T is 2-balanced, we call Check with a pointer to the root of T. The procedure takes $\Theta(n)$ time since it visits each vertex of T exactly once.