CSCB36 A2: Nena Harsch and Natalia Tabja

Question 1 (written by Nena, verified by Natalia)

Proof that binomial heap has exactly $n - \alpha(n)$ edges
We define a predicate P on integers greater than or equal to 1.
$P(n) : f(n) = n - \alpha(n)$, where α is the number of 1's in the binary representation of n (also known as the number of binomial trees in the forest).

Basis: Let n = 1
By definition, $n = 1 =< 1 >_2 = 2^0$ which leads to $F_1 =< B_0 >$, a single tree with one node and no edges.
$\alpha(n)$ is 1 from the equation above, therefore $f(n) = n - \alpha(n) = 1 - 1 = 0$.
Therefore, $P(n)$ holds as wanted.

Induction Step: Let n > 1
Suppose $P(n)$ holds whenever $j \geq 2$
WTP: $P(n+1)$ holds.
We need to consider two cases: where adding on just adds a binary 1 to the end and when it changes the other binary 1's.
Case 1: Add a binary 1 to the end
In this case, there is no B_0 tree to start with (ex/ 1010 and by adding a node we will essentially be adding the B_0 tree. There will be no additional edges added because B_0 tree has no edges, so $f(n+1) = f(n)$. We prove this by showing $f(n+1) = (n+1) - \alpha(n+1) = (n+1) - (\alpha(n)+1) = n - \alpha(n) = f(n)$. Therefore, $P(n+1)$ holds in this case.
Case 2: When there are multiple 1's trailing after the first zero.
We need to know the placement of the first 0 to know what an additional 1 will do. An additional edge is created for every 1 it turns into 0 since it takes one key comparison to make two B_n into a N_{n+1} tree. For example, if we have 11011 and we add a 1, the binary number changes to 11100, which is 2 key comparisons, two new edges, and also happens to be the number of 1's trailing the first 0.
From this pattern, we see that $\alpha(n+1) = \alpha(n) + 1 - numberof1'strailingthefirst0$. Therefore, $f(n+1) = (n+1) - \alpha(n+1) = n + 1 - (\alpha(n) + 1 - numberof1'strailingthefirst0) = n - \alpha(n) + numberof1'strailingthefirst0 = f(n) + numberof1'strailingthefirst0$. This is exactly the observation above where the number of edges added was the original number of edges plus the number of key comparisons which was the number of new edges which was the number of 1's trailing the first 0. Therefore, $P(n+1)$ holds.

Therefore, by the Principle of Simple Induction, P(n) holds for every integer greater than or equal to 1. QED

Worst-case total cost of k successive insertions
Another way that we can find the total cost of k successive inserts is by looking at the difference in the number of edges on the initial tree and the tree after k inserts. The initial tree with n nodes would have $n - \alpha(n)$ edges, and the tree after k inserts (n + k nodes) would have $(n+k) - \alpha(n+k)$ edges. x, the difference from the final number of edges and initial, would be $x = (n+k) - \alpha(n+k) - (n - \alpha(n)) = k - \alpha(n+k) + \alpha(n)$. We know that the number of trees for a given integer, i, is $\lfloor log_2(i) \rfloor$. Therefore, $x = k - \lfloor log_2(n+k) \rfloor + \lfloor log_2(n) \rfloor \leq k - log_2(n+k) + log_2(n)$. When $k > log_2(n)$, $log_2(n+k) > log_2(n)$, therefore we can say $x < k$, so the average cost of an insertion is bounded above by the constant k.

Question 2 (Written by both, verified by both)

Part I: Increase key and Delete
a) Increase Key:
A simple algorithm to increase the key of a given item in a binomial max heap H can be described by the following algorithm:

Algorithm 1 INCREASE-KEY(T, x, k)

```
1: Input: Input binomial max heap T, pointer to the key we want to increase x, value we
   want to increase the key to T
2: Output: A binomial max heap
3:
4: if x.key < k then
5:     x.key = k
6:     while x.parent.key != Null do
7:         if x.parent.key ≥ k then
8:             break
9:         end if
10:        x.key = x.parent.key
11:        x.parent.key = k
12:        x = x.parent
13:    end while
14: end if
```

This algorithm reassigns the node's value to k if k is greater than the current key of x. Then, to restore the max heap properly, it compares it's value with it's parent. If the nodes do not need to be switched, then we break since the max heap property holds. Else, if the parent's key is less than the current, we switch their keys and reassign our pointer to the parent with the value of k.
The worst case would be that we have n nodes, which means $\lfloor log_2(n) \rfloor$ trees, and we want to reassign x at the very bottom of the largest tree to k which is greater than any value in that respective tree. We can also say that the largest tree in a binomial heap of n nodes will have a depth of $\lfloor log_2(n) \rfloor$, so the maximum key comparisons needed to be made would be at most $\lfloor log_2(n) \rfloor$. Therefore, the algorithm runs in $O(log_2(n))$ time. (Note: look at lemma 19.1 in the textbook to check this)

b) Delete:

Algorithm 2 DELETE-ITEM(T, x)

```
1: Input: Input binomial max heap T, pointer to the key we want to increase x, value we
   want to increase the key to T
2: Output: A binomial max heap
3:
4: INCREASE-KEY(T,x,∞)
5: EXTRACT-MAX(T)
```

Delete Algorithm: To delete a given item x from a binomial max heap H, we first call the Increase-Key(T, x, ∞) algorithm. This increases the key of node x to ∞, ensuring that x becomes the maximum element in the heap. After increasing the key, we perform the Extract-Max

operation to remove x from the heap, which is synonymous to the Binomial-Heap-Extract-Min defined in the textbook just with max instead. It finds the root with the maximum key, removes it from the root list, removes the maximum key and make the children trees a heap, and then merges it back with the root list. The Increase-Key operation runs in $O(\log n)$, where n is the number of elements in the heap, as it may need to traverse up the tree to restore the heap property. The Extract-Max operation also takes $O(\log n)$ time, as it involves locating and removing the maximum element from the heap. Therefore, the total time complexity of the delete operation is $O(\log n)$.

Part II: Ultra-Heap

c) Ultra-Heap Design:

The Ultra-Heap we have designed is essentially a combination of two binomial heaps: one max-heap and one min-heap. The advantage of having one heap of each type is that we are able to easily access and extract both the minimum and maximum elements of the heaps in question, since these operations are already defined for their respective heap.

Every element inserted into the Ultra-Heap is inserted into both the min-heap and the max-heap. Furthermore, each node in the min-heap has a pointer to its corresponding node in the max-heap, and vice versa. These cross-pointers allow us to efficiently locate and delete an element from the other heap when it is extracted from either one of them.

d) Ultra-Heap Operations:

• **Insert(k)**: k will be inserted into the min-heap and the max-heap separately using the predefined `Insert()` method, and a cross pointer between the k in each heap will be established. This will make deletion easier. Since the binomial heap `Insert(x)` method takes $O(\log n)$ time, and we are simply inserting twice, the runtime is still $O(\log n)$.

• **ExtractMax()**: First, we use the cross-pointer to locate the corresponding node in the min-heap and remove it using algorithms defined in Part I. Then, the `ExtractMax()` method is performed on the max-heap to remove the maximum element. Since locating and removing an element from a binomial heap takes $O(\log n)$, the total runtime for this operation remains $O(\log n)$.

• **ExtractMin()**: First, we use the cross-pointer to locate the corresponding node in the max-heap and remove it using algorithms defined in Part I. Then, the `ExtractMin()` method is performed on the min-heap to remove the minimum element. Like `ExtractMax()`, both locating and removing the element from each heap take $O(\log n)$, so the total runtime for this operation is $O(\log n)$.

• **Merge(D, D')**: To merge two Ultra-Heaps D and D', we merge the min-heaps of both heaps and the max-heaps of both heaps. After the merge, cross-pointers between corresponding nodes in the min-heap and max-heap are re-established. Since merging binomial heaps takes $O(\log n)$, the total runtime for the merge operation is $O(\log n)$.

Question 3 (written by Natalia, verified by Nena)

Algorithm 3 IS-AVL(u)

```
1: Input: Root node u of a BST.
2: Output: Boolean value indicating whether T is an AVL tree.
3:
4: function isAVL(u)
5:     if u = NIL then
6:         return (-1, True)
7:     end if
8:     (leftHeight, isLeftAVL) = isAVL(u.lchild)
9:     (rightHeight, isRightAVL) = isAVL(u.rchild)
10:    balanceFactor = abs(leftHeight - rightHeight)
11:    isBalanced = (balanceFactor ≤ 1)
12:    isAVLTree = isLeftAVL and isRightAVL and isBalanced
13:    currentHeight = 1 + max(leftHeight, rightHeight)
14:    return (currentHeight, isAVLTree)
15: end function
16: function checkIfAVL(T)
17:    (height, isAVLTree) = isAVL(T)
18:    return isAVLTree
19: end function
```

Explanation of Our Code:

- **Base case**: If the node u is `NIL`, it is considered AVL by definition, and its height is -1.
- **Recursive part**: For each node u, we recursively check the left and right subtrees. If both subtrees are AVL, and the balance factor of the current node is within the acceptable range, then the subtree rooted at u is AVL.
- **Balance Factor**: The balance factor is calculated as the absolute difference between the heights of the left and right subtrees. If this value is greater than 1, the subtree rooted at u is not balanced, and hence, not AVL. We can use the absolute value because we don't need to know if the tree is left-heavy or right-heavy for the purposes of this function.
- **Output**: For each node u, the function returns two values: the height of the subtree rooted at u, and a boolean indicating whether or not the subtree is AVL.

Time Complexity: The algorithm runs in $O(n)$, where n is the number of nodes in the tree. This is because:

- Each node is visited exactly once.
- For each node, we perform a constant amount of work (calculating heights, checking balance, and returning the result).

Thus, the overall time complexity is linear, $O(n)$, which satisfies the required worst-case running time.