1. a) Written by: Crystal Yip Read by: Anna Lieu
Define predicate P(n) for all $n \in \mathbb{N}$ such that -

P(n): For all binomial heaps, S_n, with n elements, there are exactly n - α(n) edges, where α(n) is the number of 1's in binary representation of n.

Note: There is an edge iff there is a comparison. When merging, the # of comparisons = # of carry bits in binary representation [taken from lecture] (*)

Base Case: n = 1

Consider a binary tree with 1 single element.

1 - α(1) = 1 - 1 = 0 as wanted [since there are no edges in a heap with a single element]

Inductive Step: $1 \leq$ i < n

Suppose P(i) holds [IH]
w.t.s P(n) holds.

2 cases:
- *1)* The heap with n-1 elements is an even number
- *2)* The heap with n-1 elements is an odd number

Consider case 1):

Assume binomial heap with n-1 nodes has an even number of elements. Thus we know that adding an extra node means that an S_0 binomial tree is added (a tree with a single node). Inserting a single S_0 binomial tree does not create any new edges. (#)

We also know that n-1 has a 0 at the right most bit in binary representation. So a heap with n nodes has one more 1 than n-1 in binary representation. i.e. if the binary representation of a heap with n-1 nodes is $<S_k...S_{k-n}\ 0>_2$, then a heap would n nodes would be $<S_k...S_{k-n}\ 1>_2$. (##)

A binomial heap with n-1 elements has n - 1 - α(n - 1) edges [IH, since n-1 < n, then P(n-1) holds]

From (#), we have that P(n): *# edges of heapsize n* = n - 1 - α(n - 1) [no new edges were formed]
From (##), we have α(n) = α(n - 1) + 1
Thus, P(n): *# edges of heapsize n*
= n - 1 - α(n - 1)
= n - 1 - (α(n) - 1)
= n - α(n) as wanted.

Consider case 2):

Assume binomial heap n-1 nodes has an odd number of elements. Thus we know that adding an extra node means that there will be merges and edges will be created.

Suppose we add 1 to a binary number m. α(m+1) = α(m) - # of carries during the binary addition + 1. This is because every time every time there is a carry bit during addition of 1 to m, the carry bit 1 is added to a digit with value 1 in m which produces a sum of bit 0 with a carry of 1. Therefore, in order to get α(m+1), we subtract α(m) by the number of carries. However, if the carry bit 1 is added to a digit with value 0 in m, this carry bit would be the last carry bit of the addition. Adding 1 + 0 produces a 1 bit digit in α(m+1), so we add a 1 back to the total. (***)
Ex.

$1^{1}0^{1}1...^{1}11$	With every carry, the 1 cancels out to a 0 except for
1	the last carry bit which results in bit 1.
-------------	The *1* is resulted from the last carry bit.
1 *1* 0... 0 0	Thus, we **add 1** back to the difference of:
	α(m) - *# of carry bits in the binary addition of m+1*

Thus, following (***), α(n) = α(n-1) - *# of carry bits in the binary addition of (n-1)+1* + 1
Rearranging equation above, *# of carry bits in binary addition of (n-1)+1* = α(n-1) - α(n) + 1 (**)
From (*), a new edge is created when a comparison is made during a merge. So,
P(n): *# edges of heapsize n* (S_n)
= *# edges of* S_{n-1} + *# of edges formed in adding 1 node to* S_{n-1}
= *# edges of heapsize n-1* + *# of carry bits in the binary addition of (n-1)+1*
= n - 1 - α(n - 1) + *# of carry bits in the binary addition of (n-1)+1* [IH, P(n-1) holds]
= n - 1 - α(n - 1) + α(n - 1) - α(n) + 1 [from (**)]
= n - α(n) as wanted.

1. b) Written by: Xingyu Wang, Read by: Anna Lieu
Let $n \in \mathbb{N}, k \in \mathbb{N}$ such that $k > \log n$.
Let H_n be a Binomial Heap of size n.

When inserting an element into H, we add the element to the Binomial Heap as a B_0, then update it. During updating, if there is another B_0 in the Heap, we compare the key of the roots and merge the two trees into one. This goes on for any encounter of a tree with **same size** (e.g., if there is an existing B_1 in addition to the new B_1 merged from the two B_0s), because there can be only one binomial tree of a particular size in the forest. Each merge takes 1 comparison of the roots, and creates 1 new edge between the two existing same-size trees to combine them.
Therefore, the number of new edges are the number of comparisons occurred.
Now, suppose we are adding k elements into H_n. Denote the new result Binomial Heap as H_{n+k}.
Assume each comparison takes linear time c_0.
Denote the total cost of adding k elements as C.

$$\begin{aligned}
C &= c_0(\text{Extra edges created}) \\
&= c_0(\text{edges of } H_{n+k} - \text{edges of } H_n) \\
&= c_0(n + k - \alpha(n + k) - (n - \alpha(n))) \qquad \text{from 1(a)} \\
&= c_0(n + k - \alpha(n + k) - n + \alpha(n)) \\
&= c_0(k + \alpha(n) - \alpha(n + k))
\end{aligned}$$

By definition from 1(a),

$$\begin{aligned}
\alpha(n) &= \text{number of 1s in binary form of } n \\
&\leq \text{number of all bits in binary form of } n \\
&\leq \log n
\end{aligned}$$

Since we are analyzing the run-time with respect to the input size k, everything related to n is a constant in our analysis. Denote $c_0\alpha(n)$ as c.
$\alpha(n + k)$ denotes the number of 1s in binary $n + k$, so $\alpha(n + k) \geq 0 \Longrightarrow -\alpha(n + k) \leq 0$.
n and k are natural numbers. If $n = 0$, since $k > \log n$ and $k \in \mathbb{N}$, $k \geq 1$. So $n + k \geq 1$. If $n > 0, \log n \geq 0 \longrightarrow k > 0 \Longrightarrow n + k \geq 1$. In both cases, $n + k \geq 1 \Longrightarrow \log(n + k) \geq 0$. So $-\alpha(n + k) \leq \log(n + k)$.
So

$$\begin{aligned}
C &= c + c_0(k - \alpha(n + k)) \\
&\leq c + c_0(k + \log(n + k))
\end{aligned}$$

Because we treat n as a constant, $n + k \in \mathcal{O}(k)$. So eventually $C \leq c + c_0(k + \log k)$.
Then because $\log k \in \mathcal{O}(k)$, eventually $C = c + c_0k \in \mathcal{O}(k)$.
This is the worst-case total cost of k insertions. Therefore, the average cost of inserting k elements would be $\in \mathcal{O}(\frac{k}{k}) = \mathcal{O}(1)$.

2. Written by: Crystal Yip, Anna Lieu Read by: Xingyu Wang

```
check2balance(node):
    if traverse(node)[0] > -2: return true
    else: return false

/* traverse returns a tuple of 2 integers (int radius, int height) */
traverse(node):
    if (node == None): return (-1, -1)
    left_tree     = traverse(node.lchild)
    right_tree    = traverse(node.rchild)
    left_radius   = left_tree[0] + 1   // keep track of the radius of left subtree
    right_radius  = right_tree[0] + 1  // keep track of radius of right subtree
    left_height   = left_tree[1] + 1   // keep track of height of left subtree
    right_height  = right_tree[1] + 1  // keep track of height of right subtree
   // traverse returned a -2, i.e. one of the nodes failed 2-balanced check
    if (left_radius == -1 or right_radius == -1): return (-2, -2)
    // if the node is a single leaf, return a radius and height of 0
    if (left_height == 0 and right_height == 0): return (0, 0)
    // if node has only one subtree, it passes 2-balance check - return radius and height of that subtree
    if (left_height == 0): return (right_radius, right_height)
    if (right_height == 0): return (left_radius, left_height)
    // if the node has two children/subtrees, must check if the 2-balanced property is satisfied.
    // set radius to smaller of the two subtree radii
    if (left_radius ≥ right_radius): radius = right_radius; else: radius = left_radius
    // set height to larger of the two subtree heights
    if (left_height ≥ right_height): height = left_height; else: height = right_height
    // check if the current node passes the 2-balanced property -> return (-2,-2) if it fails
    if (2*radius ≥ height): return (radius, height); else: return (-2, -2)
```

Upper bound: For node u that is the root of binary search tree T, traverse(u) traverses down all descendants of T, where it visits each node once. Say T(n) is the number of steps when traverse(u) executes, where u is the root of a binary search tree T and n is the number of nodes in T. Case 1 is when u is nil and returns on the first line. This takes a constant time a. Case 2 is when u is not nil (u is a leaf, or has 1-2 children), then one recursive call each is made on left child and right child of u. Since each recursive call is made on u's left and right subtree (assume each subtree contains one half of the elements of tree T), then each call takes T(n/2) steps. The rest of the lines take, say, constant time b. We get the closed form formula:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + b & \text{if } n > 1 \end{cases}$$

This takes same form as T(n) = cT(n/d) + f(n), where constants $c, d \in \mathbb{Z}^+, d > 1$ and $f : \mathbb{N} \to \mathbb{R}^+$, and $f(n) \in \theta(n^k)$, for $k \in \mathbb{R}, k \geq 0$. Thus, we can use Master Theorem.
c = 2; d = 2; f(n) = b = bn^0 = bn^k, where k = 0. Then $log_d c = log_2 2 = 1 > 0 = k$
Then $T(n) \in \mathcal{O}(n^{log_d c}) = T(n) \in \mathcal{O}(n^{log_2 2}) = T(n) \in \mathcal{O}(n)$
The algorithm check2balance(u) calls traverse(u) on the first line which takes $\mathcal{O}(n)$ time and the rest of the lines take constant running time $\mathcal{O}(1)$. Thus, check2balance(u) worst case running time $\mathcal{O}(n)$.

Lower bound: Consider an arbitrary binary search tree T with n elements and node u as root. When traverse(u) is called on tree T, then the algorithm recursively calls both children of u, and their children are recursively called, down to the leaves of T. The algorithm stops when the children of the leaves are called. Since leaves have nil children, then the algorithm returns on the nil nodes on the first line, taking constant time. Since a single call of traverse(u) takes constant running time, and it is called at least n times, then $T(n) \in \Omega(n)$. Since check2balance(u) calls traverse(u) once and the remaining lines take constant time, then worst case $\in \Omega(n)$
Thus worst case running time $\in \theta(n)$