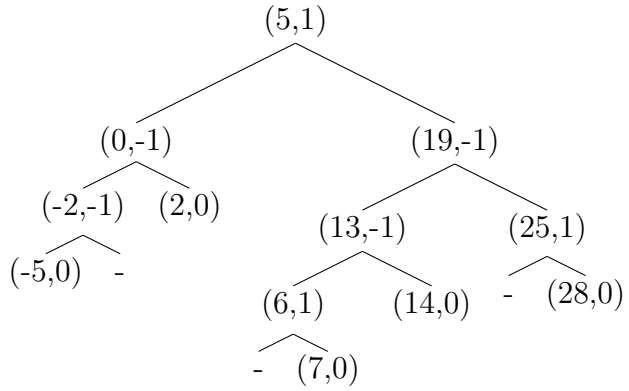
CSC263, Assignment #2

Brendan Neal[1], Filip Tomin[2]

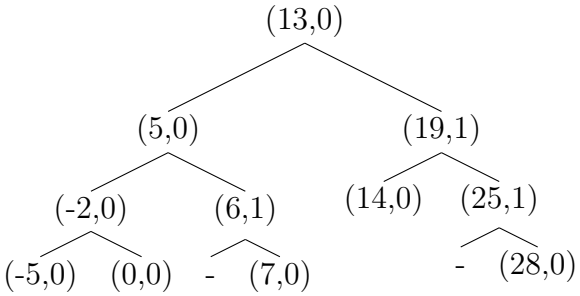
[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. Tuples are in format (key, balance factor).
   AVL Tree after insertions, before deleting 2:



   After deleting 2:



2. Algorithm for merging binary search trees B_1 and B_2 into binary search tree T.

Algorithm Description:

Recurse down the right hand side of B_1 in the direction of its maximum key. At the same time, recurse down the left hand side of B_2 in the direction of B_2's minimum key. Stop when either one of the traversals is unable to keep going (i.e. there are no more children).

We will refer to whatever node causes the recursion to stop as R. R will be removed from its respective tree and any children of R will be "moved up" to be the children of R's parent (through pointer reassignment). R will become the root of T.

If the recursion stops first in B_1, $R = max(B_1)$. In this case, every node in B_1 will be less than R (by definition of max) and every node in B_2 is greater than R (by the specification given in the question).

If the recursion stops first in B_2, $R = min(B_2)$. In this case, every node in B_2 will be greater than R (by definition of min) and every node in B_1 will be less than R (by the specification given in the question).

In either case, the value of R will be greater than B_1 and less than B_2, since $B_1 < max(B_1) < min(B_2) < B_2$. We can then make R the new root of the tree T and make its left child B_1 and its right child B_2.

Obviously, since T is constructed with R as its root and B_1 and B_2 as its children, its height must be less than or equal to $max\{h_1, h_2\} + 1$.

Runtime Analysis:

Since the algorithm recurses down one side of a binary tree, it's running time should be proportional to the length of the side it is traversing. The length of this side is bounded above by the height of the tree (since the height of a tree is the longest length between the root and any one of its leaves). In the worst case, the length of the path between the root of a tree and its left/rightmost child would be equal to the height of the tree. However, since the algorithm stops recursing when it cannot find any more children on one side of a tree, it will obviously stop when the "end" of the shorter tree is reached. Thus, the algorithm makes at most $min\{h_1, h_2\}$ traversals to get from the root to the "end" of a tree.

When R is found, it will take constant time to remove R from its respective tree, as there are two possible cases: R has no children, R has a child in the opposite direction of traversal. In the first case, removal will take constant time (simple reassignment of one of its parent's pointers to null). In the second case, removal will again take constant time (simple reassignment of one of its parent's pointers to R's child).

Building the tree will take constant time because all that needs to be done is set R's child pointers to point to B_1 and B_2.

Since traversing from a node to one of its children takes constant time and the algorithm executes at most $min\{h_1, h_2\}$ of these traversals and does some constant amount of steps to remove R and construct T the algorithm must run in $O(min\{h_1, h_2\}+c), c \in \mathbb{R}^+$ time, which is in $O(min\{h_1, h_2\})$ time.

3. (a) We are going to be using two heaps (one min-heap and one max-heap) to accomplish this, with the combined data structure being called S. We will refer to these heaps as MAX and MIN, respectively. Every value in S that is less than or equal to the median of S will be stored in MAX. Likewise, every value in S that is greater than or equal to the median of S will be stored in MIN. Both MAX and MIN will also have respective variables that keep track of their size, referred to as MAX.size and MIN.size.

   By the definition of median, approximately half of the values in a set will be greater than or equal to the median. The remaining half of the values will be less than or equal to the median. Thus, MAX.size $\approx$ MIN.size $\approx \frac{\text{MAX.size + MIN.size}}{2}$.

   When the total number of elements in both heaps is odd, either MAX or MIN has one more element than its counterpart. In this case, S has a definitive median. The root node of the heap that has one extra node will be the median in this case.

   When the total number of elements is even, MAX and MIN have the same number of elements. In this case, it is true that S has no definitive median - it must be calculated by averaging the heads of MAX and MIN.

   The following will consist of pseudocode descriptions of how this data structure will handle insertion and median calculation.

   Defining some helper functions (algorithms that pertain to heap extraction and insertion are recalled from CLRS chapter 6):

   Bigger():

   ```
   If (MAX.size > MIN.size):
       return MAX
   If (MIN.size > MAX.size):
       return MIN
   ```

   Smaller():

   ```
   If (MAX.size > MIN.size):
       return MIN
   If (MIN.size > MAX.size):
       return MAX
   ```

Heap-Extract-Head(H):

```
If H == MAX:
    Heap-Extract-Max(H)
    MAX.size -= 1
If H == MIN:
    Heap-Extract-Min(H)
    MIN.size -= 1
return
```

Heap-Insert(H, x):

```
If H == MAX:
    Max-Heap-Insert(H, x)
    MAX.size += 1
If H == MIN:
    Min-Heap-Insert(H, x)
    MIN.size += 1
return
```

Inserting some integer x into this dual-heap structure S will work as follows:

Insert(S, x):

```
If (MAX.size == 0 and MIN.size == 0):
    MAX[1] = x
    MAX.size += 1
Else:
    If x < Median(S):
        Heap-Insert(MAX, x)
        MAX.size += 1
    If x > Median(S):
        Heap-Insert(MIN, x)
        MIN.size += 1
If |MAX.size − MIN.size| > 1:
    val = Heap-Extract-Head(Bigger())
    Heap-Insert(Smaller(), val)
return
```

Calculating the median of the dual-heaps will work as follows:

Median(S):

```
If (MAX.size == 0 and MIN.size == 0):
    return Null
If (MAX.size == MIN.size):
    return (MAX[1] + MIN[1]) / 2
If (MAX.size != MIN.size):
    return (Bigger())[1]
```

Using the above algorithms to form array M, where $M[i] = \text{median}(A[1:i])$:
M(A):

```
M = []
for i = 1 to n:
    Insert(S, A[i])
    M.append(Median(S))
return M
```

Correctness of Algorithm:
The general idea of the algorithm is to use the dual-heap data structure to keep the median or values approximately equal to the median of the array prefix at the head of the heap(s), allowing for quick computation of the median by simply inspecting the value(s) at the head of the heap(s) and doing the appropriate computations.

By iterating over each element of A and inserting it into S, we will ensure that the prefix of A ($A[1 \ldots i]$) will always have its median or values close to its median at the head(s) of MAX and MIN. We can use this fact to quickly compute the median based on the head(s) of MAX and MIN.

Correctness of Insert:
In the case when MAX and MIN are both empty heaps, Insert(S, x) inserts x into MAX and increments the size of MAX by one. In this case, x is now the median of S. The property that every value in S that is less than median(S) is stored in MAX has been preserved. As well, MAX.size has been incremented by one to the correct amount (1) from it's initial value (0) to account for this insertion.

In the case where MAX and MIN are non-empty heaps, Insert(S, x) inserts x into MAX if x is less than the head of MAX, and into MIN if x is greater than the head of MIN. If the difference in size between MAX and MIN is greater than 1, we rotate the bigger tree onto the smaller tree by removing the head of the bigger

tree and inserting it into the smaller tree. Keeping the trees balanced will keep the heads of both trees approximately equal to the median, depending on the trees' sizes, since approximately half the total number of values go in each tree and the head of each approaches the same value.

Correctness of Median:
Thanks to Insert(S,x), the median of all the elements can be found via the heads of the min and max heaps. If either heaps' size is greater than the other, we have an odd number of values and the median is well-defined as the root node of the bigger tree, which is returned with the 3rd if statement. If they're equal in size, then the median is defined as the average of the two root nodes, handled by the second if statement. The first if statement handles the situation where S is empty and returns NULL.

(b) **Runtime Analysis:**
We observe that $M(A)$'s first and last lines take constant time to run. The two lines in the for-loop (that iterates over 1 to n) make calls to Insert(S, A[i]) and M.append(Median(S)).

Fully tracing the code of Insert(S, A[i]), we see that it makes calls to Max/Min-Heap-Insert and Heap-Extract-Max/Min as well as running some constant time steps (e.g. incrementation). Since CLRS declares these heap insertion and extraction algorithms to run in $O(\log n)$ time for heaps with $O(n)$ elements, Insert(S, A[i]) $\in O(\log n + d), d \in \mathbb{R}^+$. Since constant running time is in $O(\log n)$, it is true that $d \in O(\log n)$, so $O(\log n + d) \in O(\log n)$. Thus, the running time of Insert(S, A[i]) is $O(\log m)$, where m is the number of elements in S.

The call to M.append(Median(S)) should take constant time, as appending to an array takes constant time and determining the median of S takes constant time. Median(S) takes constant time because Median(S) is simply a collection of if-statements that immediately return some value (they do not lead to any sort of iteration).

Thus, M(A) runs a for-loop that has $O(n)$ iterations, and each iteration executes steps that run in $O(\log m)$ (m is the number of elements in S). Therefore, $T(n)$ of M(A) is as follows:
$T(n) = \sum_{i=1}^{n} \log m = \sum_{i=1}^{n} \log i$ | Because the number of elements in S is equal to i and there are n calls to insert, which runs in $O(\log n)$ time
$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n$ | Because n is greater or equal to all values of i in the summation

$\sum_{i=1}^{n} \log n = n \log n$ | Value of the sum
$T(n) \leq n \log n$ | Inequality from before
Therefore, an upper bound for $M(A)$'s running time $T(n)$ is $n \log n$. Thus, $M(A)$'s running time $T(n)$ is $O(n \log n)$, so constructing M from A will take $O(n \log n)$ time.