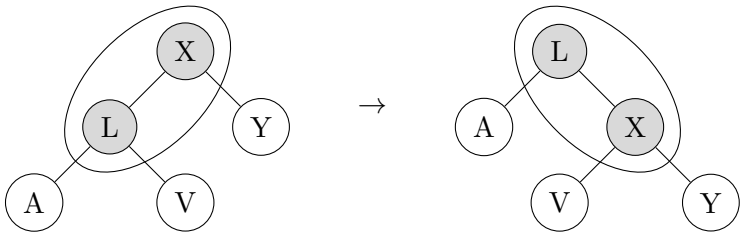
COMP20007 Design of Algorithms

Week 10 Workshop Solutions

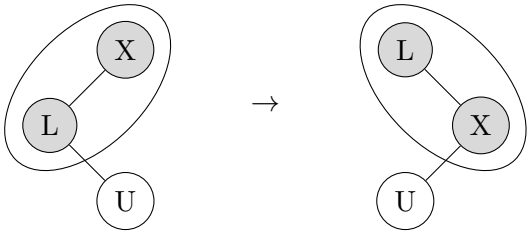
Tutorial

1. Rotations

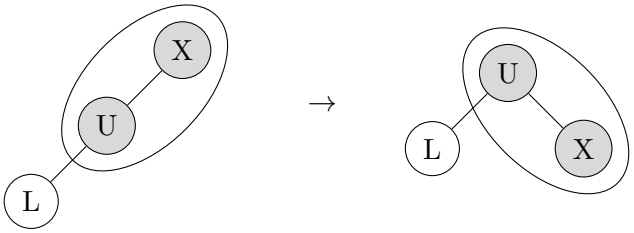
(a) doesn't improve overall balance:



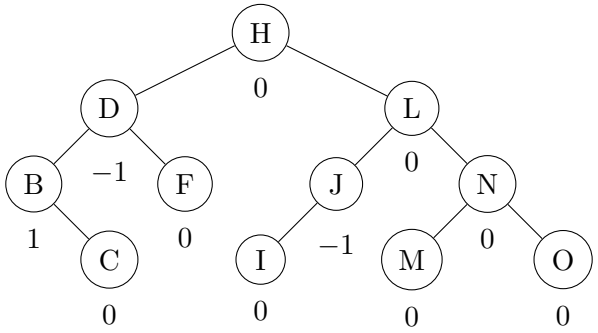
(b) doesn't improve overall balance:



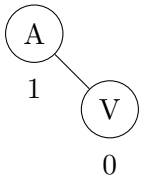
(c) *does* improve overall balance:



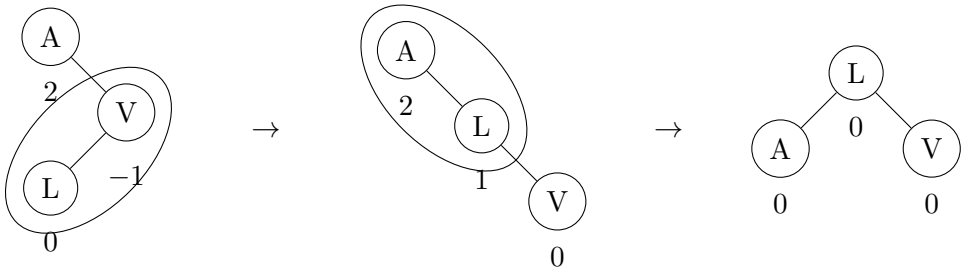
2. Balance factor Balance factor listed below each node. Calculated by subtracting the height of the node's left subtree from the height of its right subtree.



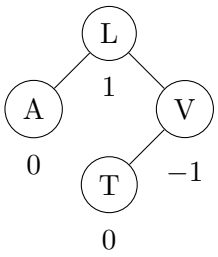
3. AVL Tree Insertion Insertion of A and V is fine, all balanced (balance factor below node):



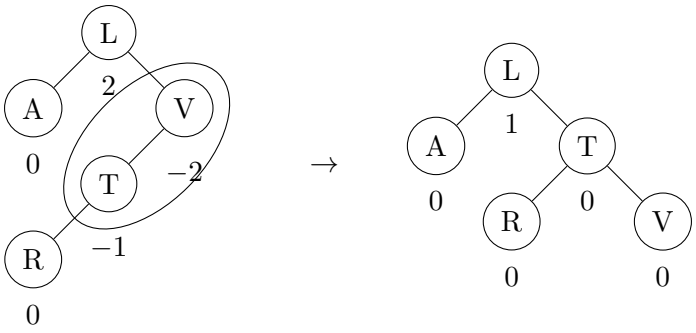
Inserting L causes an imbalance at A, and it's a zig-zag case so we need two rotations to fix it:



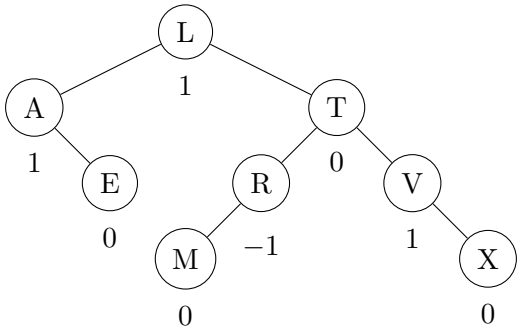
Then, inserting T is fine:



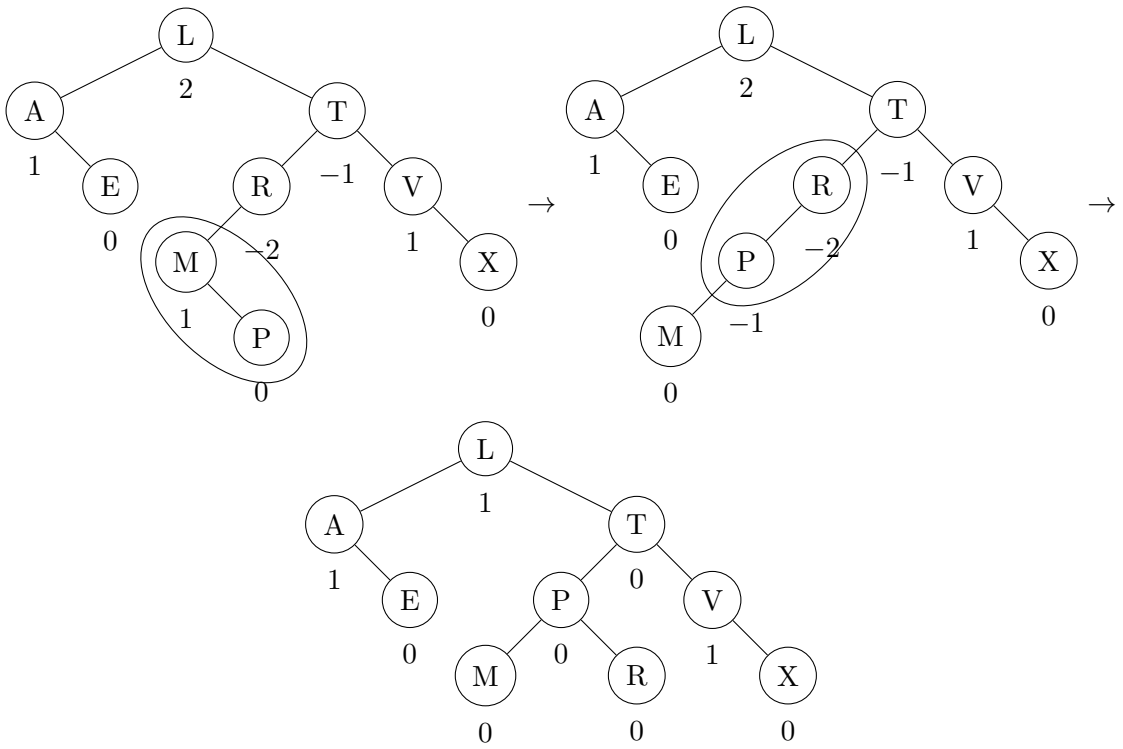
But inserting R gets us into trouble! It causes an imbalance at V. This time it's a zag-zag case, so we only need one rotation:



Next, insertion of E, X, and M cause no imbalances:



The final insertion, P, causes a zag-zig problem at R, once again fixed with two rotations:

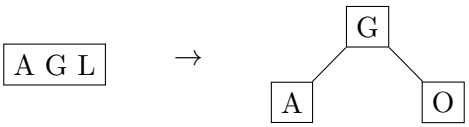


Notes:

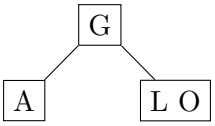
- The terms ‘zig’ and ‘zag’ are used to describe imbalances. A ‘zig-zag’ case, for example, refers to a node with an imbalance because of its right (‘zig’) subtree, and the imbalance in that subtree is coming from the left side (‘zag’).
- To determine which type of imbalance we are dealing with, we can look at the sign of the balance factor. If it’s a +2, that means the problem is in the right child. If the right child has a -1 (different sign), then the problem is with it’s left child, and it’s a zig-zag case. Two rotations are needed. If the right child has a +1 (same sign), then the problem is with its right child, and it’s a zig-zig case. One rotation is needed. Likewise, if the sign at the unbalanced node is -2, then the problem is with the left child. If the left child has a balance factor of -1 (same sign), then the problem is with its left child, i.e. it’s a zag-zag case. One rotation is needed. If the left child has a balance factor of +1 (different sign), then the problem is with its right child, i.e. it’s a zag-zig case. Two rotations are needed.
- Always deal with an imbalance at the deepest available point. For example, when inserting P, both R and L became unbalanced, but we dealt with the problem at R. This automatically fixed the problem at L. In reality, we only calculate balance factors on our way back up the tree from the insertion, so we can simply deal with the first imbalance (+2 or -2 balance factor) we encounter. All balance factors have been shown at all stages in the above diagrams, but in reality only the heights would be stored with each node — balance factors would only be calculated on the way back up the tree after an insertion, and only for the nodes on this direct path.

4. 2-3 Tree Insertion Inserting the first two elemens results in a leaf node:

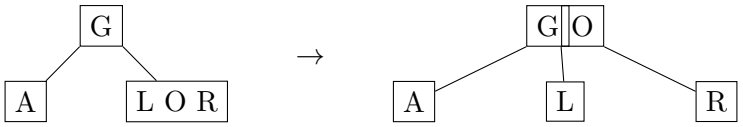
With the insertion of G, our leaf ends up with 3 elements, so it must be split up – we promote the middle element like so:



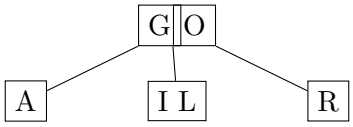
Inserting O adds to the right most leaf node:



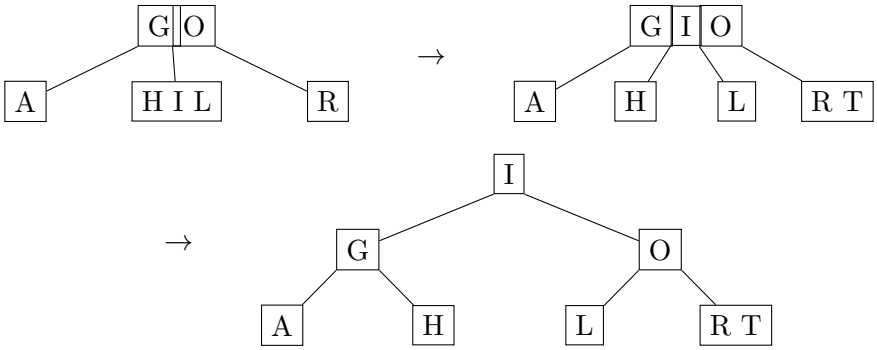
When we add R we get a node with 3 elements and must promote the middle one.



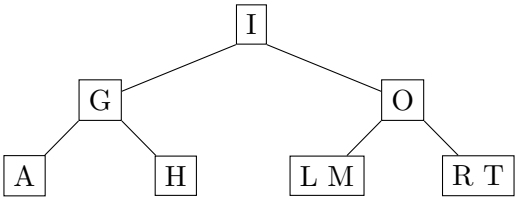
Adding I just adds to a leaf node.



Adding H leaves a node with 3 elements, we will fix this my promoting I twice:



Finally, we can insert M into a leaf node without having to do any promotions, so the final 2-3 tree looks like:

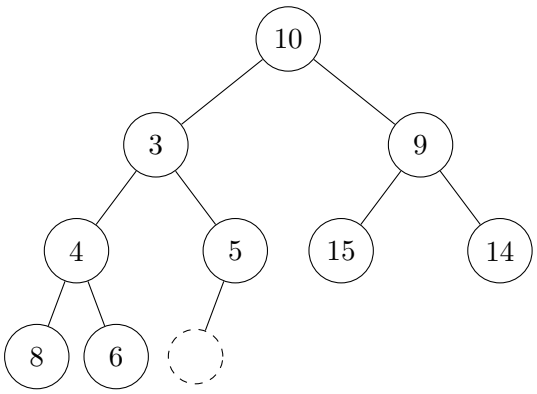


5. Heaps

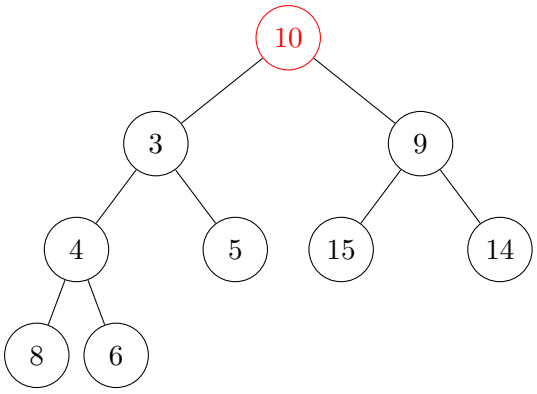
(a) As in the lectures we leave the first index empty, and then populate the array with the elements of the heap in level-order:

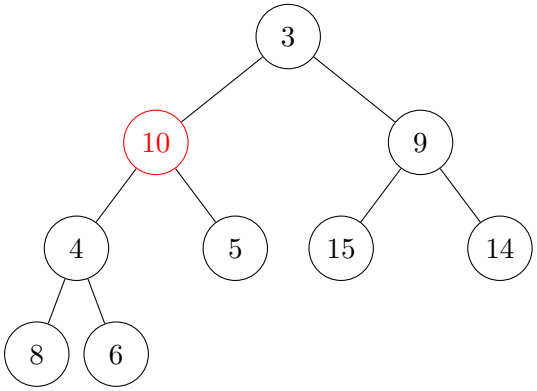
$$\Big[-, 1, 3, 9, 4, 5, 15, 14, 8, 6, 10\Big]$$

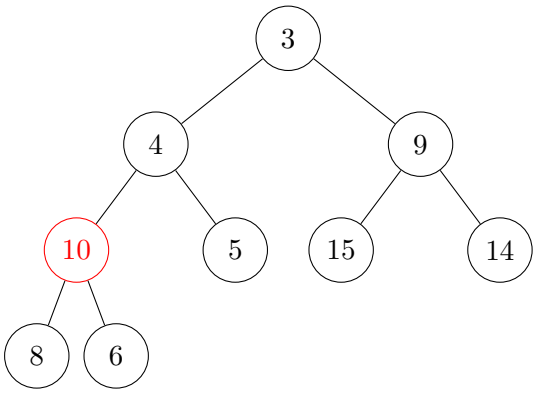
(b) First we store the value of the root of the heap, in this case it is 1, then we swap the last element in the heap to the root:

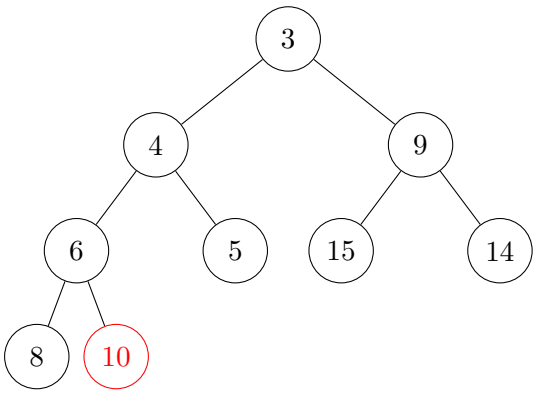


We then SiftDown the root, by comparing it to each of its children, and if it's larger than either swap it with the smaller child and repeat, if it's smaller than both children then stop.

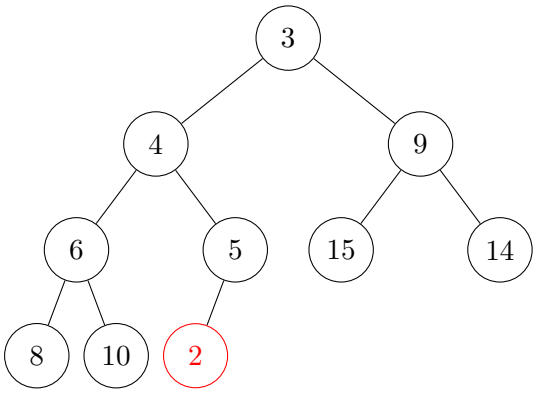


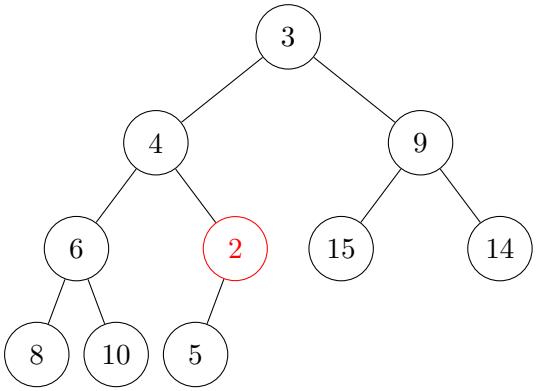


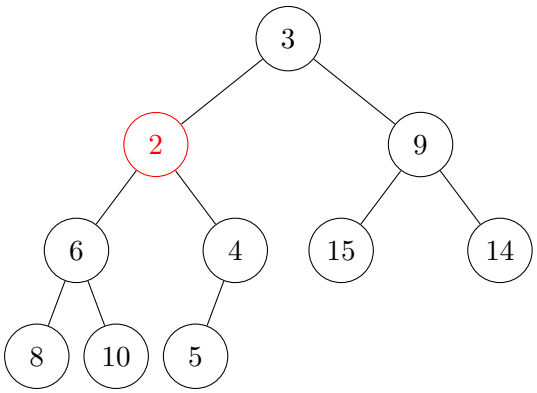


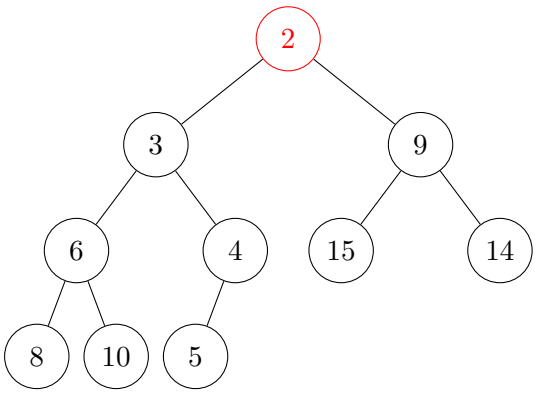


(c) We'll add the new value in a new node at the next spot in the tree, and then SiftUp. Sifting up consists of swapping the node with its parent as long its parent is larger than it.









6. (Optional) Using a min-heap for kth-smallest element Consider the following algorithm which finds the kth-smallest element using a min-heap.

function HEAPkTHSMALLEST($A[0 \ldots n-1], k$)
  $Heap \leftarrow$ HEAPIFY($A[0 \ldots n-1], k$)
  `// remove the smallest` $k-1$ `elements`
  for $i \leftarrow 0 \ldots k-2$ **do**
    REMOVEMIN($Heap$)
  return REMOVEMIN($Heap$)

The time complexity of HEAPIFY is $\Theta(n)$ and each REMOVEMIN is $\Theta(\log n)$, so the total time complexity of this algorithm is $\Theta(n + k \log n)$.

Notice that this algorithm is not input-sensitive on the order of the array, it is only input sensitive with respect to k. This may be a desirable property for a variety of reasons.

7. (Optional) Quickselect *Quickselect* is an algorithm for solving the kth-smallest element problem using the PARTITION algorithm from quicksort.

(a) We know that the *pivot* must occupy index p in the sorted array. We can make use of this fact and only search the section of the array which must contain the element which has the kth index in the sorted array. The pseudocode for quickselect is as follows.

function QUICKSELECT($A[0 \ldots n-1], k$)
  $pivot \leftarrow$ SELECTPIVOT($A[0 \ldots n-1]$)
  `// p is the index of the pivot in the partitioned array`
  $p \leftarrow$ PARTITION($A[0 \ldots n-1], pivot$)
  if $p == k$ **then**
    return $A[k]$
  else if $p > k$ **then**
    return QUICKSELECT($A[0 \ldots p-1], k$)
  else
    return QUICKSELECT($A[p+1 \ldots n-1], k-(p+1)$)

We will assume for this question that we are selecting the first element to be the pivot, but as we have seen in lectures this is not necessarily the best strategy.

(b)

$$\begin{aligned}
\Big[9,3,2,15,10,29,7\Big] &\xrightarrow{\text{PARTITION}} \Big[3,2,7,9,15,10,29\Big],\ p=3<k\\
k &\leftarrow k-(p+1)=0\\
\Big[15,10,29\Big] &\xrightarrow{\text{PARTITION}} \Big[10,15,29\Big],\ p=1>k\\
k &\leftarrow k=0\\
\Big[10\Big] &\xrightarrow{\text{PARTITION}} \Big[10\Big],\ p=0==k\\
&\implies \textbf{return } 10
\end{aligned}$$

(c) The best-case for QUICKSELECT is when $p = k$ after the first partition. With our pivot strategy of selecting $pivot = A[0]$ this corresponds to having the kth-smallest element at the start of the array.

The single PARTITION call takes $\Theta(n)$ time, so the best-case time complexity for QUICKSELECT is $\Theta(n)$.

(d) The worst-case for QUICKSELECT is when each call to PARTITION only rules out one element, and thus we have to PARTITION on n elements, $n-1$ elements, $n-2$ elements and so on.

An example of this is finding $k = 0$ when the input is reverse sorted order.

The time taken for each partition is linear, so the total time for QUICKSELECT in the worst case is,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2).$$

(e) To compute the expected time-complexity of this algorithm we will make the assumption that the order of the array is uniformly random, and that after each PARTITION, the pivot ends up in the middle of the array, *i.e.*, $p = \frac{n}{2}$.

In this case, the recurrence relation for QUICKSELECT will be:

$$T(n) = \Theta(n) + T\left(\frac{n}{2}\right),\ T(1) = 1$$

We see that this first the format required by the Master theorem with $a = 1, b = 2, c = 1$. Since $log_b(1) = 0 < 1 = c$ we have that $T(n) \in \Theta(n^c) = \Theta(n)$.

Thus QUICKSELECT has an expected time-complexity of $\Theta(n)$.

It turns out that even if the array is split into very unevenly sized components after partitioning (*e.g.,* $0.99n$ and $0.01n$ length sub-arrays) and the kth smallest element *always ends up in the larger array* then the runtime of QUICKSELECT is still $\Theta(n)$. Can you use the Master Theorem to show why this is the case?

(f) Well the worst case for QUICKSELECT is much worse than the worst case for the heap based algorithm ($\Theta(n^2)$ vs. $\Theta(n \log n)$).

Also, if we make the assumption that $k \ll n$ (*i.e.*, k is very small compared to n) then we can treat $\Theta(n + k \log n)$ as $\Theta(n)$ meaning the heap-based approach is comparable to the best case for QUICKSELECT.

In general for unknown k we can expect QUICKSORT to be faster in the expected case.