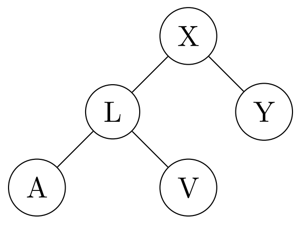
COMP20007 Design of Algorithms

Week 9 Workshop

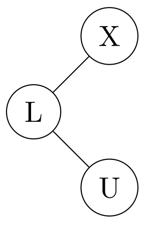
Tutorial

1. Rotations In the following binary trees, rotate the 'X' node to the right (that is, rotate it and its left child). Do these rotations make the tree more balanced, or less balanced?

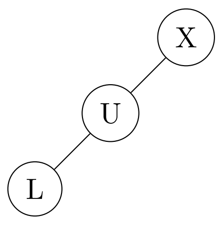
(a)



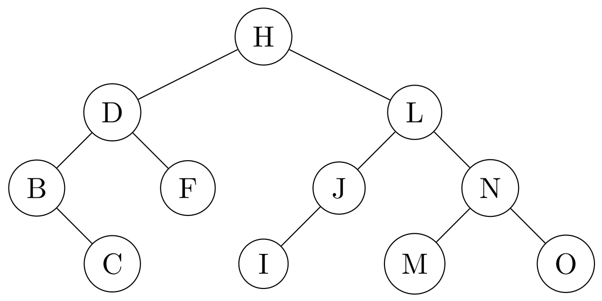
(b)



(c)



2. Balance factor A node's 'balance factor' is defined as the height of its right subtree minus the height of its left subtree. Calculate the balance factor of each node in the following binary search tree.



3. AVL Tree Insertion Insert the following letters into an initially-empty AVL Tree.

A V L T R E X M P

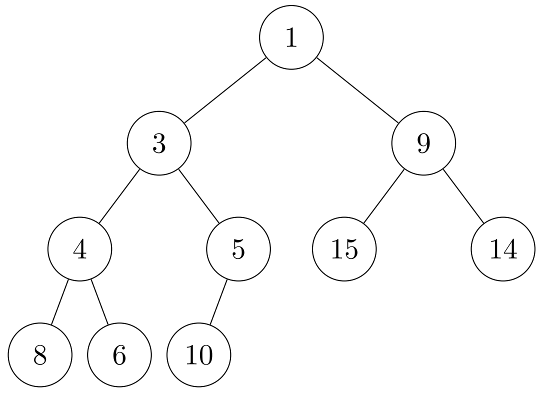
4. 2-3 Tree Insertion Insert the following letters into an initially-empty 2-3 Tree.

A L G O R I T H M

5. Heaps A *Heap* (for this question we will discuss a min-heap) is a complete binary tree which satisfies the heap property:

Heap Property: each node in the tree is no larger than its children.

Suppose we start out with the following heap:



(a) Show how this heap would be stored in an array as discussed in lectures, *i.e.*, the root is at index 1 and a node at index i has children in indices $2i$ and $2i+1$.

(b) Run the RemoveRootFromHeap algorithm from lectures on this heap by hand (*i.e.*, swap the root and the "last"element and remove it. To maintain the heap property we then SiftDown from the root).

(c) Run the InsertIntoHeap algorithm and insert the value 2 into the heap (*i.e.*, add the new value to the end of the heap, and compare it with its parent, swapping the new value and its parent until its parent is smaller than it, or it is the root).

6. (Optional) Using a min-heap for kth-smallest element The *kth-smallest element problem* takes as input an array A of length n and an index $k \in \{0, \dots, n-1\}$ and returns as output the kth-smallest element in A (with the 0th-smallest being the smallest, the 1th-smallest being the second smallest and so on).

How can we use a the min-heap data structure to solve the kth-smallest element problem? What is the time-complexity of this algorithm?

7. (Optional) Quickselect Quicksort uses a Partition(A, *pivot*) function which partitions the array A to satisfy the constraint that all elements smaller than the *pivot* occur before it in the array, and those greater than the *pivot* occur after in. In quicksort we call Partition, keep track of the final index of the *pivot*, p and call Partition recursively on $A[0 \dots p-1]$ and $A[p+1 \dots n-1]$.

(a) Design an algorithm based on Quicksort which uses the Partition algorithm to find the kth-smallest element in an array A.

Hint: We know that once we partition the array, the index of the *pivot*, p, must be the correct position of *pivot* in the final sorted array. Can we use this fact to deduce where the kth smallest element must be?

(b) Show how you can run your algorithm to find the kth-smallest element where $k = 4$ and $A = \left[9, 3, 2, 15, 10, 29, 7\right]$.

(c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

(d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

(e) What is the expected-case (*i.e.*, average) time-complexity of your algorithm?

(f) When would we use this algorithm instead of the heap based algorithm from Question 5?

Computer Lab

In this weeks lab we're going to experiment with different sorting functions, and different implementations of sorting functions.

In the lab files for this week we have provided,

- A function for timing a sorting algorithm: `time_sorting_function()`
- A function for generating random arrays of different sizes: `generate_array()`. This function can generate random arrays, sorted arrays, reversed arrays and almost sorted arrays.
- An implementation of `insertion_sort()` and `quicksort()`, however the `quicksort()` algorithm uses a very naive partitioning and pivoting strategy.

In this lab you'll be using the timing function to compare the performance of different sorting algorithms.

1. Lomuto and Hoare Partitioning In `sorting.c` we have implemented a function `partition_first_pivot()` which uses our naive partitioning function to partition the array based on the pivot being the first element in the array.

We can do better than the partitioning function we've created! Our function first allocates additional memory (an expensive operation) and does three passes over the array (to copy over elements smaller than, equal to and greater than the pivot respectively) to construct this intermediate array. It then copies this over to the original array.

Your task is to write a partitioning function which uses the following partition strategies: (a) **Lomuto** partitioning, and (b) **Hoare** partitioning.

Have a look at the last function in `main.c`, `quicksort_first()` to see how we can time these different pivot selection strategies.

1. Quicksort–Insertion Sort Hybrid Although the asymptotic time-complexity of quicksort is better than that of insertion sort, there is a high overhead associated with quicksort, so it may not always be the best choice of sorting algorithm in practice.

Time both quicksort and insertion sort on arrays of varying types and sizes to determine when each algorithm works better.

Use your results to write a hybrid sorting function `hybrid_sort()` which chooses between insertion sort and quicksort depending on the size of the input.

2. Pivot Selection Strategies Experiment with different pivot selection strategies by writing different partition functions in `sort.c` to pivot based on the last element, and a random element.

Use the functions discussed above to time your different pivot selection strategies. Which one works best for which types of arrays?

3. Better Partitioning Read about the *median-of-three* partitioning strategy which can lead to large performance improvements. Implement this median of three partitioning strategy and see if you see any performance improvements.

4. (Optional) Bottom-Up Mergesort In lectures we discussed both the *top-down* and the *bottom-up* strategies for implementing mergesort.

The *bottom-up* strategy involves starting with each pair of successive elements, sorting them with respect to each other, then moving to merge successive chunks of 2 sorted elements, continuing until we're merging the left and right half of the array. We can implement this iteratively.

Have a read of the Wikipedia's section[1] about bottom-up mergesort and implement your own function `mergesort()` which performs this algorithm. How does it compare to the other algorithms?

[1] `https://en.wikipedia.org/wiki/Merge_sort#Bottom-up_implementation`