COMS20010 — Problem sheet 6

You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and videos instead. (Then come back to the sheet when you need to revise for the exam!) I'll make solutions available shortly after the problem class. As with the Blackboard quizzes, question difficulty is denoted as follows:

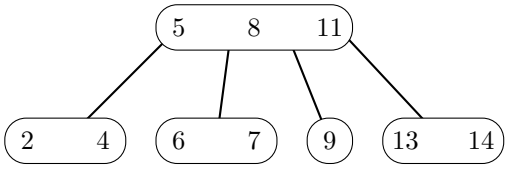
- ⋆ You'll need to understand facts from the lecture notes.
- ⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in unfamiliar situations.
- ⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes and also move a bit beyond them, e.g. by seeing how to modify an algorithm.
- ⋆⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in a way that requires significant creativity. You should expect to spend at least 5–10 minutes thinking about the question before you see how to answer it, and maybe far longer. At most 10% of marks in the exam will be from questions set at this level.
- ⋆⋆⋆⋆⋆ These questions are harder than anything that will appear on the exam, and are intended for the strongest students to test themselves. It's worth trying them if you're interested in doing an algorithms-based project next year — whether you manage them or not, if you enjoy thinking about them then it would be a good fit.

If you think there is an error in the sheet, please post to the unit Team — if you're right then it's much better for everyone to know about the problem, and if you're wrong then there are probably other people confused about the same thing.

This problem sheet covers week 6, focusing on spanning trees, the union-find data structure and 2-3-4 trees.

1. (a) [⋆⋆] Consider the list of numbers $5, 2, 7, 9, 8, 11, 13, 4, 6, 14$. Add these to a 2-3-4 tree in this order and draw what the final tree looks like.

Solution:

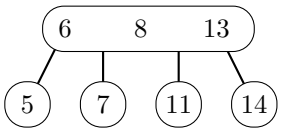


If this solution is unclear, try drawing the tree after every insertion.

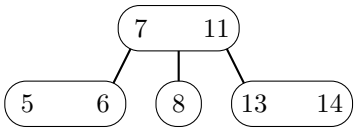
(b) [⋆⋆] Now remove the elements $2, 9, 4$. What does the final tree look like?

Solution:

There are two possible solutions depending on which direction you choose to fuse siblings. These are:

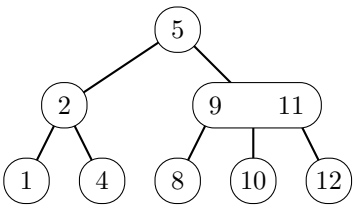


or



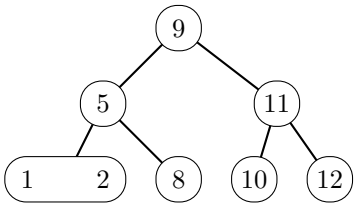
If these solutions are unclear, try drawing the tree after every deletion.

(c) [⋆⋆] Consider the 2-3-4 tree



What does this tree look like after running `Delete(4)`?

Solution:



2. [⋆⋆] In database design, it is very common for each record to have a unique identifier, and for records to be added and removed while the database is running. Explain how 2-3-4 trees could be used to support a database.

Solution: We store the identifier of each record inside a 2-3-4 tree, alongside a pointer to the record itself. If we have n records in total, this allows us to find, insert or delete a new record in $O(\log n)$ time. We can also quickly iterate over all records in the database, or records whose IDs fall into a specified range.

(Actually, databases tend to use "B-trees" instead. These are very similar to 2-3-4 trees, but instead of each node having between 2 and 4 children, each node has between d and $2d$ children for some large value of d. Search, insertion and deletion are then implemented in essentially the same way. This makes the tree shallower, which improves the constant factors in the running speed since passing to a new node in the tree often requires a slow disk or cache access.)

3. [⋆⋆] Prove, using the perfect balance property, that a 2-3-4 tree containing n values has depth $\Theta(\log n)$.

Solution: Let T be a 2-3-4 tree with depth d, and let $L_0, \ldots, L_d$ be the layers of T from top to bottom. By perfect balance, each node in $L_0, \ldots, L_{d-1}$ has at least 2 children, so $|L_i| \geq 2|L_{i-1}|$ for all $1 \leq i \leq d$, and so T contains at least $\sum_{i=0}^{d} 2^i = 2^{d+1} - 1$ nodes. Each node contains at least one value, so

$$n \geq 2^{d+1} - 1 \geq 2^d.$$

Taking logarithms of both sides yields $d \leq \log n$ and hence $d \in O(\log n)$. For the other direction, observe that each node in $L_0, \ldots, L_{d-1}$ has at most 4 children, so T contains at most

$$\sum_{i=0}^{d} 4^i = \frac{4^{d+1} - 1}{3} \leq 4^{d+1}$$

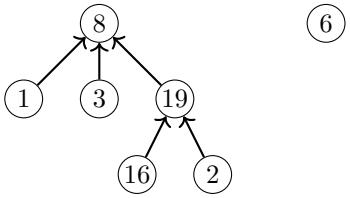
nodes. Each node contains at most three values, so $n \leq 3 \cdot 4^{d+1} \leq 4^{d+2}$. Taking logarithms of both sides yields $2(d+2) \geq \log n$ and hence $d \in \Omega(\log n)$. Since $d \in O(\log n)$ and $d \in \Omega(\log n)$, it follows that $d \in \Theta(\log n)$.

4. (a) [⋆⋆] Consider the list $8, 3, 16, 1, 6, 19, 2$. Add each of these to a union-find data structure (i.e. run `MakeUnionFind`). What does the data structure look like after running this operation?

Solution:
(8) (3) (16) (1) (6) (19) (2)

(b) [⋆⋆] Run `Union(8,1)`, `Union(19,16)`, `Union(2,16)`, `Union(1,3)`, `Union(16,8)`. What does the data structure look like after running these operations?

Solution:



Note that this solution is not unique since there are different ways of merging components - for example `Union(8,1)` might put 8 under 1 rather than 1 under 8. Any valid solution will have the same structure as above, but potentially with some labels swapped around. i.e. any solutions will be unique up to isomorphism.

(c) [⋆⋆] What is the result of running `FindSet(1)`, `FindSet(2)`, `FindSet(19)`?

Solution: The solution here depends on the exact structure of the tree formed in part (b). These answers correspond to the example shown above.
`FindSet(1)` $= 8$, `FindSet(2)` $= 8$, `FindSet(19)` $= 8$.

5. [⋆⋆⋆] Let G be an m-edge connected weighted graph, given in adjacency list form, whose edge weights are all either 1 or 2. Adapt Kruskal's algorithm into an $O(m\alpha(m))$-time algorithm to find a minimum spanning tree of G, where α is the inverse Ackermann function.

Solution: Recall that Kruskal's algorithm first sorts the edges of G in $O(m \log m)$ time, and then builds the minimum spanning tree using $O(m)$ operations on a union-find data structure. In this case, we can sort the edges of G by weight in $O(m)$ time with counting sort (i.e. by making two passes through the list of edges to extract first the weight-1 edges and then the weight-2 edges). The union-find operations then take $O(m\alpha(m))$ time using the optimised version of the data structure, so we're done.

6. [⋆⋆] Let T be a 2-3-4 tree.

   (a) Prove by induction that performing an in-order traversal of T will output the values of T in increasing order. (For example, on encountering a 3-node with values 5 and 7, and children c_0, c_1 and c_2 from left to right, in-order traversal first recursively processes c_0, then outputs 5, then processes c_1, then outputs 7, then processes c_2.)

   Solution: We proceed by induction on the depth of T. If T has depth 0, then in-order traversal simply prints the values contained in T's root in sorted order, so we're done. Suppose T has depth $d > 0$, and the result holds for trees of depth at most $d - 1$. Let $x_1 < \cdots < x_{k-1}$ be the values stored in T's root, and let $T_1, \ldots, T_k$ be the subtrees rooted in the first layer of T (from left to right). Then by induction, in-order traversal will output the vertices of T_1 in increasing order, followed by x_1, followed by the vertices of T_2 in increasing order, followed by x_2, and so on. By the definition of a 2-3-4 tree, this is in increasing order as a whole, and we're done.

   (b) Prove that if v is a value stored in a non-leaf node N of T, then the predecessor of v is stored in a leaf.

   Solution: By (a), the predecessor of v is the previous value output by in-order traversal. Since N is not a leaf, it has a child immediately to the left of v, and in-order traversal will have finished processing this child immediately before outputting v. The only possible base case of in-order traversal is a leaf, so it follows that v's predecessor is a leaf.

7. [⋆⋆⋆] Let G be a connected weighted graph, and suppose that no two edges in G have the same weight. Prove that G has a **unique** minimum spanning tree. (**Hint:** Look back at the correctness proof for Kruskal's algorithm.)

   Solution: Suppose for a contradiction that G has at least two distinct minimum spanning trees. Let S be an arbitrary minimum spanning tree, and let T be the output of Kruskal's algorithm. Recall that when we proved in lectures that the output of Kruskal's algorithm was optimal, we did it by constructed a sequence of spanning trees $T_0, \ldots, T_k$ where:

   (i) $T_0 = S$ and $T_k = T$;

   (ii) For all $0 \leq i \leq k$, T_{i+1} is formed from T_i by removing a single edge e_i and adding a single edge f_i;

   (iii) For all $0 \leq i \leq k$, $w(T_{i+1}) \leq w(T_i)$.

   Since S is minimum, no spanning tree can have weight less than S, so (iii) actually implies $w(T_{i+1}) = w(T_i)$. It follows from (ii) that $w(e_i) = w(f_i)$, and in particular e_i and f_i are two edges with distinct weights — which can't happen! So we have our contradiction. Intuitively, what this argument shows is that Kruskal's algorithm can generate any possible spanning tree of the graph, depending on how it breaks ties between equal-weight edges.

8. (a) [⋆⋆] Let T_1 and T_2 be 2-3-4 trees, containing n_1 and n_2 values respectively, where $n_1 \leq n_2$ and the depth of T_1 is at most the depth of T_2. Give an algorithm to merge them into a single tree in $O(n_1 \log(n_2))$ time. (You may assume all values in T_1 and T_2 are distinct from each other.)

Solution: Iterate over T_1 in any order, and use the insertion operation to add each element of T_1 to T_2. Each insertion takes $O(\log(n_1 + n_2)) \subseteq O(\log(2n_2)) = O(\log n_2)$ time, and there are n_1 insertions in total, so the running time follows.

(b) [$\star\star$] Explain intuitively why we should expect that any algorithm to do this will require $\Omega(n_1)$ time.

Solution: If the values of T_1 and T_2 are interleaved with each other, so that the nodes of T_1 have to be scattered throughout T_2, then we should expect to have to process each node of T_1 individually, which must take $\Omega(n_1)$ time.

(c) [$\star\star\star\star\star$] Suppose now that we are given a single element x, that every value in T_1 is strictly less than x, and that every value in T_2 is strictly greater than x. (This operation is called a *join*.) Give an algorithm to merge T_1, T_2 and x into a single 2-3-4 tree in $O(\log n_2)$ time. You do not need to prove it works.

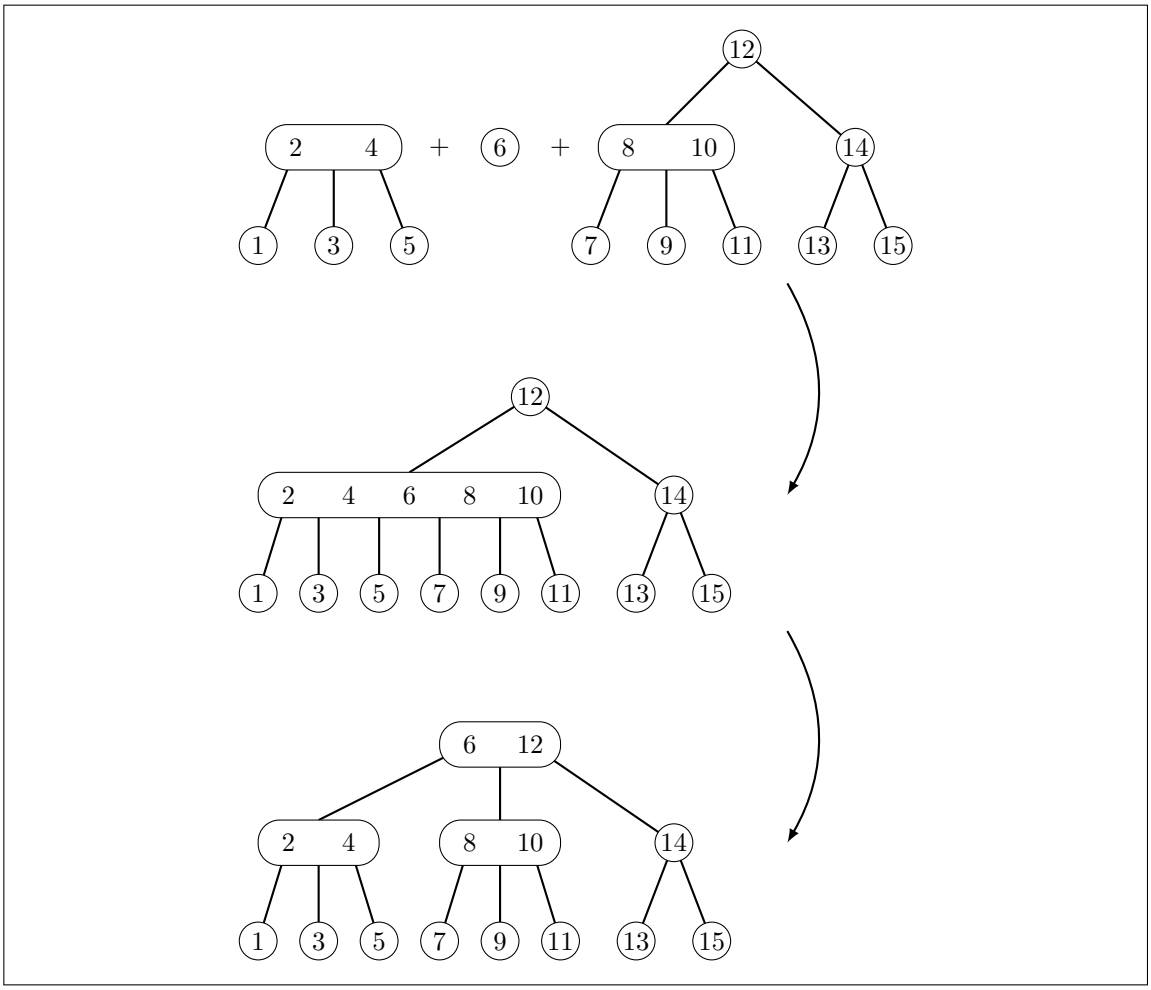
Solution: First find the leftmost node N of T_2 such that the subtree rooted at v has depth d. Split any 4-nodes encountered on the way down, exactly as in the insertion operation. This takes $O(\log n_2)$ time. Write:

- $c_1, \ldots, c_k$ for the children of the root of T_1 from left to right;
- $v_1, \ldots, v_{k-1}$ for the values of the root of T_1 from left to right;
- $d_1, \ldots, d_\ell$ for the children of N from left to right;
- $w_1, \ldots, w_{\ell-1}$ for the values N from left to right.

Now replace N by a node N' with values $v_1, \ldots, v_{k-1}, x, w_1, \ldots, w_{\ell-1}$ from left to right, and children $c_1, \ldots, c_k, d_1, \ldots, d_\ell$ from left to right; thus N' has $k + \ell$ children and $k + \ell - 1$ values, and all descendants of the ith child are between the $i - 1$st and the ith value. (This follows because T_1 and T_2 are 2-3-4 trees where every value in T_1 is less than x, which in turn is less than every value in T_2.) If $k + \ell - 1 = 3$, then N' is a valid 4-node and we are done. Otherwise, we at least have $k + \ell - 1 \leq 4 + 4 - 1 = 7$, so we can split N the same way we would in a standard insertion operation:

- Let x be the $\lfloor (k + \ell)/2 \rfloor$'th value in N', i.e. a value as close as possible to the middle of the node.
- Move x upwards to N''s parent, or to form a new 2-node if N' is the root.
- Split N' into two nodes, one of which becomes the left child of x and contains all values to the left of x, and the other of which becomes the right child of x and contains all values to the right of x.

Since we removed a value from the middle of the node and split the remaining values into two nodes of roughly equal size, each new nodes will have at most $\lceil (k + \ell - 2)/2 \rceil \leq 3$ values, so the result is a valid 2-3-4 tree. An example is shown below.



(d) [⋆⋆⋆] Now give an algorithm to merge only T_1 and T_2 into a single 2-3-4 tree in $O(\log n_2)$ time. (**Hint:** Try reducing to part (c).)

Solution: Create a dummy element x whose weight is between the rightmost element of T_1 and the leftmost element of T_2; this will take $O(\log n_2)$ time. Apply the join operation of part (c) to join T_1, T_2 and x in $O(\log n_2)$ time. Finally, delete x from the resulting tree in $O(\log(n_1 + n_2 + 1)) = O(\log n_2)$ time.

9. [⋆⋆⋆⋆] You are given an $n \times n$ bitmap image, and you wish to divide the pixels into contiguous regions of similar colours. (This is a common problem in computer vision.) We model this by a grid graph G with vertex set $[n] \times [n]$. We say a grid square (i, j) is adjacent to the grid square to its left, right, top or bottom if their corresponding pixels have similar RGB values. Suppose your computer has $C \in O(n)$ cores working in parallel, all of which have read and write access to a common union-find data structure initialised with all points in $[n] \times [n]$. Sketch an algorithm to find the components of G in $O(n^2\alpha(n^2)/C)$ time.

Solution: Assume for simplicity that n is divisible by C. Divide the vertices of G into equal horizontal bands $B_1, \ldots, B_C$, where $B_i = [n] \times \{(i-1)n/C + 1, (i-1)n/C + 2, \ldots, in/C\}$. The i'th core should scan through B_i in raster order, i.e. line-by-line from left to right and top to bottom.

When it encounters a vertex (a, b), it should check whether it is adjacent to the vertex $(a-1, b)$ to its left in G; if so, it should merge the set containing (a, b) and the set containing $(a-1, b)$. Likewise, if it is adjacent to the vertex $(a, b+1)$ above it, it should merge the set containing (a, b) and the set containing $(a+1, b)$. This takes the core $O(n^2\alpha(n^2)/C)$ time. We claim that when every core has finished, the sets in the union-find data structure will be precisely the components of G, so we're done. Indeed, a union operation will have been carried out along every edge of the graph at some point in the algorithm's operation, so its two endpoints must lie in the same set. Thus the endpoints of any path must lie in the same set, and so any two vertices in the same component of G must lie in the same set. Conversely, if two vertices lie in the same set, there must be some path of edges joining them and so they must lie in the same component. This is a slightly simplified version of the Hoshen-Kopelman algorithm.