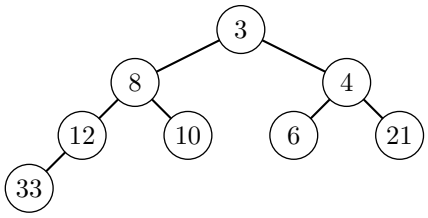
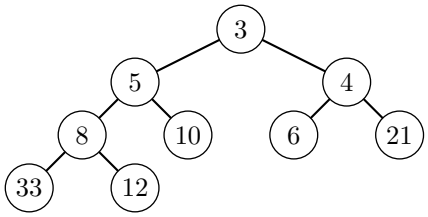
Sample Midterm 2 Solutions and Grading Guide

1. (a) The array of a heap contains the following elements (in this order) $3, 8, 4, 12, 10, 6, 21, 33$. Draw the heap as a binary tree.
(b) Insert 5 into this heap and then perform a DELETEMIN. You must show all the steps performed in these two operations.

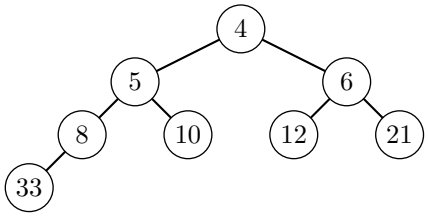
Solution:
(a)



(b) 5 will be inserted as the right child of 12 (after the last element 33 in the array). We Upheapify with 5 until its parent is smaller than it. So we will have to swap it with 12 and then 8, but not with 3. The resulting heap will look like:



The DELETEMIN will replace 3 with 12 and DOWNHEAPFIY until its children are both bigger than it. This is achieved by swapping with 4 and then 6 and 12 becomes a leaf again. Result:



Grading guide

(a) Correct drawing of tree:	**2 points**
(b) Initially 5 is placed correctly in the array:	**1 point**
Upheapify performed correctly, with a brief desciption of which swaps are made and why:	**2 points**
Correct resulting heap:	**1 point**
12 is made the root in DELETEMIN:	**1 point**
Downheapify performed correctly with reasoning for swaps:	**2 points**
Correct resulting heap:	**1 point**

2. Given an array of $n \geq 2$ distinct numbers, we want to determine if there are two numbers such that their difference is exactly 42. Give an algorithm that solves this problem with at most $O(n \log n)$ comparisons.

Solution and grading guide:

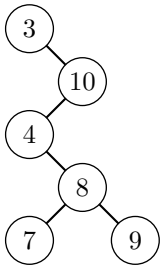
Algorithm:
Sort the array: **1 point**
Mergesort or Heapsort (if some other sort is used or nothing is mentioned then no points): **1 point**
(Solution 1 assuming integers) In the sorted array, check every $A[i]$ with the next 42 elements. If $A[i] + 42$ and that are the same then return 'yes' and if they were never the same, then return 'no'. (Solution 2 not assuming integers) Make another array B with $B[i] = A[i] + 42$. Merge these two arrays and check if any element repeats by checking every two consecutive elements. **3 points**

Correctness:
Sorted array of integers must have $A[i] + 42$ in a position that is at most $i + 42$ since the numbers are distinct. Solution2: if $A[i] + 42 = A[j]$, then $B[i] = A[j]$ and they will be adjacent in the new merged array. **3 points**
Running time:
Sorting $O(n \log n)$: **1 point**
Solution 1: checking the next 42 elements is $O(1)$ and this is done at most n times. Solution 2: making the new array is $O(n)$ and merging both is $O(n)$ as both are sorted. Finding a common element is again $O(n)$: **1 point**
So together $O(n \log n) + O(n)$, which is $O(n \log n)$. **0 points**

Any other correct solution (using Binary search or pointers) is also awarded full points if all necessary steps are included.

3. The preorder of a binary search tree is $3, 10, 4, 8, 7, 9$. Draw this binary search tree on 6 vertices and justify why this is the way to draw the tree.

Solution and grading guide:
Correct drawing of the tree: **3 points**



The root has to be 3, because that is listed first in preorder.
3 does not have a left child, because these must be smaller than 3, but all the other numbers are bigger.
Of the numbers $10, 4, 8, 7, 9$, 10 comes first in the preorder, so 10 must be the right child of 3. Also since every other number is smaller than 10, they must all be in the left subtree of 10.
Of the numbers $4, 8, 7, 9$, 4 occurs first in the preorder, so it is the root of the left subtree of 10, and since the rest of the numbers are bigger than 4, they must be in its right subtree.
Of the numbers $8, 7, 9$, we see 8 first in preorder, so it is the root of this subtree, and 7 must be its left child, 9 its right since 7 is smaller and 9 bigger than 8.

The reasoning contains properties of preorder mentioned correctly (first number is the root etc): **3 points**

The reasoning contains properties of binary search trees (all descendents in left subtree are smaller and right subtree are bigger etc): **4 points**
Just writing the properties of preorder or binary search trees without using them in the problem is not awarded any points.

4. We insert $1, 2, \ldots, n$, in this order, into an empty 23-tree. Show that in the resulting tree, the number of 3-nodes is at most $O(\log n)$.

Solution and grading guide:

A 3-node is only created when an additional value is inserted into a 2-node. This can happen during insertion at a leaf, or when a value is being propagated up. **2 points**
Because the numbers are inserted in increasing order, the new element is always inserted in the rightmost leaf. So in the bottom level, only the rightmost leaf can be a 3-node.
2 points
During up propagation of a value, since we only propagate from the rightmost leaf, at the second last level also, only the rightmost element can be a 3-node. **3 points**
We can see that similarly, at every level of the tree, only the rightmost element can be a 3-node. So the number of 3-nodes is at most the number of levels in this 23-tree. But there are only $O(\log n)$ levels in a 23-tree after insertion of n elements.
3 points
Writing the height of the tree is only awarded points if this property is used in some way in a solution.

5. Show that the following language is in coNP:
DEG4SPANNING-TREE = $\{G|G$ is a simple, connected, undirected graph whose every spanning tree contains a degree 4 vertex $\}$

Solution and grading guide:

We need to show that the complement of this language must be in NP. **1 point**

The complement (ignoring input strings which do not encode graphs, which can be checked in polynomial time) is:
$\overline{\text{DEG4SPANNING-TREE}} = \{G|G$ is a simple, connected, undirected graph with a spanning tree with no vertex of degree 4$\}$ **2 points**
A witness for this is a spanning tree T given with either the adjacency matrix or adjacency list. **2 points**
Based on the choice of matrix or list, the size of the witness is $O(n^2)$ or $O(n^2 \log n)$, but in either case, since it is a subgraph of G, its description is atmost as big as the input description. **1 point**
Verification steps: **2 points**

- every edge of T is in G: we need to compare the matrix/list of T with that of G.
- T is connected: can be solved with BFS/DFS
- T has $n - 1$ edges: we check the adjacency matrix or list of T.
- check the degrees of each vertex in T and see if it is not 4: for this we need to go through the matrix/list once and find the degrees.

In the case of an adjacency matrix, we have inspected entire matrix a constant number of times in each step, so it is $O(n^2)$ steps. With a list, comparing two lists of size $n \log n$ is $O(n^2 \log n)$ time and we do n such comparisons in the first step above. All the other steps just look at the adjacency list once, so it is atmost $O(n^3 \log n)$ time. (Here it is not important to give exact running times, but some

reasoning should be given why every step is polynomial time). **2 points**
Since the complement has a polynomial size witness that can be verified in polynomial time, this language is in NP, so the original language is in coNP. **0 points**

6. Determine if the following language is in P or is NP-complete:
kINDPDT-SET-CLIQUE = $\{(G, k)|G$ is a simple, undirected graph that contains a clique on k vertices and also a vertex independent set on k vertices $\}$

Solution and grading guide:

The problem is NP-complete. **0 points**
First we must show that it is in NP. A witness here is an independent set and a clique of size k. Size of the witness is $O(2k \log n)$. **1 point**
Verification: We must check that both these sets contain k distinct vertices. This can be checked with sorting (takes polynomial time). Then we must check if the first set of k vertices are independent, for which we check k^2 pairs of vertices if there is no edge between them. Finally we must check that every pair in the last k vertices has an edge between them in G. This requires check the adjacency list of the graph $2k^2$ times, which is a polynomial time check. **1 point**
To see that it is NP-hard, we will reduce a NP-complete language to this, which using the transitivity of Karp reductions shows that every language in NP can be reduced to this. We will reduce MAX-INDEPENDENT-SET to this. **1 point**
If $k > n$, then $G' = G$ and $k' = k$. **Otherwise:** $f(G, k) = (G', k')$ where $k' = k$ and G' is obtained by adding a disjoint new k-clique to G and connecting every vertex of this clique to every vertex of G. **2 points**
Forming (G', k') takes polynomial time because $k \leq n$ when we expand the graph G, and so we at most double the graph. **1 point**

If $k > n$ then both G and $G' = G$ have no independent set of size k. **0 points**

If $k \leq n$, then:

- if G has a k independent vertex set, it is still independent in G' and G' necessarily has a k-clique (the new added clique). **2 points**
- if G' has a k-clique and a k-independent vertex set then the independent vertex set cannot use any new added vertex as there are connected to every other vertex of the graph. So these vertices are necessarily in G, so G has a k-independent vertex set. **2 points**

If the case of $k > n$ is never mentioned, but all other steps are correct, then 9 points are awarded.

Reduction from MAXCLIQUE with k new isolated vertices is also awarded full points provided all required reasoning is present.

7. Given n numbers, we define the middle number as the $\lceil n/2 \rceil$-th number when these numbers are arranged in increasing order. We are initially given integers $a_1, a_2, \ldots, a_n$, where we know that a_1 is their middle number, but all other numbers are unordered. Design a data structure that can be constructed with these numbers with at most $O(n)$ comparisons. Also, it must support the following two operations, both with running time $O(\log k)$ when the data structure has k elements:
INSERT: insert a new element into the data structure,
DELETE-MIDDLE: deletes the current middle number in the data structure.

Solution and grading guide:

We will use two heaps. One a min-heap will contain all the numbers bigger than the middle element. The other, containing middle element and all numbers smaller than the middle, will be in a max-heap. **2 points**

Construction of the min and max heap with elements more and less than a_1 takes $O(n)$ steps. **1 point**

We will maintain the property that: the max-heap will always contain as many or exactly one more element than the min-heap. Also the middle element will be the root of the max-heap. For this we will use the sizes of the two heaps (A.heap-size attribute). **1 point**

INSERT: if the new element to be inserted is smaller than the middle number, then we insert into the max-heap, otherwise into the min-heap. **1 point**

If the size of max-heap is equal or one more than the min-heap, then there is nothing more to do. But if the max-heap has 2 more elements than the min-heap, then we DELETEMAX and insert it in the min-heap. Similarly, if the min-heap has one more element than the max-heap, then we DELETEMIN and insert in into the max-heap.

1 point

These steps take $O(\log k)$ time because with k elements, each heap has half the elements, so its size is $O(k)$. **1 point**

DELETE-MIDDLE : we DELETEMAX from the max-heap and if the size of the min-heap is bigger, then we DELETEMIN and INSERT it into the max-heap. **2 points**

All steps of this also take $O(\log k)$ time, again because the size of the heaps is $O(k)$. **1 point**