CSE373 – Data Structures and Algorithms
Autumn 2003
Dry assignment #4 – Solutions

Problem 1.

a. BuildHeap( );

Initial step:

-∞	4	10	3	9	2	8	1	11	5	7

Performing BuildHeap on node at index 5 – no change:

-∞	4	10	3	9	2	8	1	11	5	7

Performing BuildHeap on node at index 4 – 5 percolates up a level:

-∞	4	10	3	5	2	8	1	11	9	7

Performing BuildHeap on node at index 3 – 1 percolates up a level:

-∞	4	10	1	5	2	8	3	11	9	7

Performing BuildHeap on node at index 2 – 2 percolates up a level:

-∞	4	2	1	5	7	8	3	11	9	10

Performing BuildHeap on node at index 1 – 1 percolates up a level – gives us the final state.

-∞	1	2	3	5	7	8	4	11	9	10

b. DeleteMin( );

Initial Step:

-∞	1	2	3	5	7	8	4	11	9	10

Replacing root (minimum element) with last element in heap:

-∞	10	2	3	5	7	8	4	11	9	

Percolating down from the root to maintain heap property – gives us the final state:

-∞	2	5	3	9	7	8	4	11	10	

c. IncreaseKey (5, 7)

Initial step:

-∞	2	5	3	9	7	8	4	11	10	

Increasing value of node at index 5 by 7 (i.e., 7 + 7 = 14), and calling percolate down which has no effect – gives us the final state:

-∞	2	5	3	9	14	8	4	11	10	

d. Insert (1);

Initial step:

-∞	2	5	3	9	14	8	4	11	10	

Checking if the element to insert can be inserted at the end of the array without violating the heap property. If it does violate the heap property, percolate it up until it finds its place. The (1?) denotes that this element is not yet inserted in this position, but we are checking if it can be inserted there:

-∞	2	5	3	9	14	8	4	11	10	(1?)

Percolating up element 1 – gives us the final state:

-∞	1	2	3	9	5	8	4	11	10	14

e. Insert (6);
Initial step:

-∞	1	2	3	9	5	8	4	11	10	14	

Checking if the element to insert can be inserted at end of array without violating the heap property. If it does violate the heap property, percolate it up until it finds its place. The (6?) denotes that this element is not yet inserted in this position, but we are checking if it can be inserted there:

-∞	1	2	3	9	5	8	4	11	10	14	(6?)

Inserting element 6 in the last position (it can go there) – gives us the final state:

-∞	1	2	3	9	5	8	4	11	10	14	6

Note: Some of you were using the -∞ as an actual element of the heap, which is not correct. The heap starts at index 1.

Problem 2.

a)

We will prove this by induction.

Base case: The binomial tree B_1 has one root and one child. The child (node) can be thought of as the binomial tree B_0. This satisfies the claim.

Induction hypothesis: Let's assume that the claim is true for B_k, i.e., the binomial tree of size k has binomial trees $B_0, B_1, \ldots, B_{k-1}$ as children of its root.

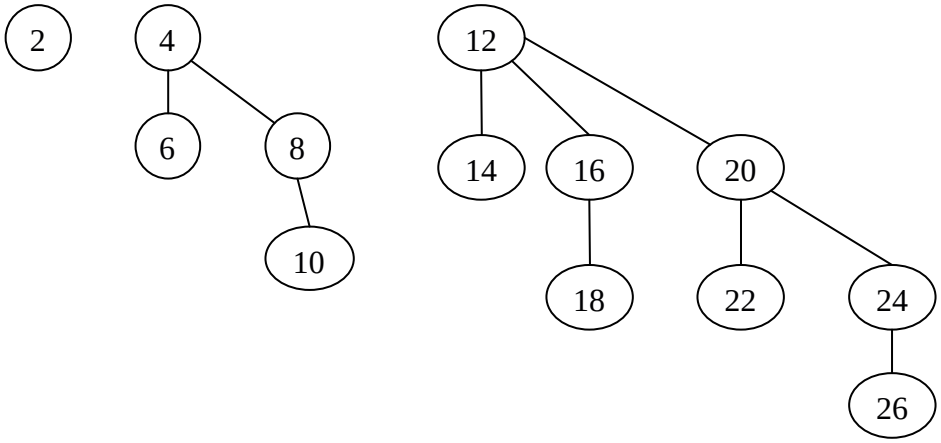
Induction step: Consider the binomial tree B_{k+1}. By definition (see p.209 in the text) it consists of two binomial trees B_k, one of which $B_{k,2}$ is attached to the root of the other, $B_{k,1}$. Therefore, by the induction hypothesis, B_{k+1} already has $B_0, B_1, \ldots, B_{k-1}$ attached to its root, since this is the same root that $B_{k,1}$ has. In addition, B_{k+1} also has $B_{k,2}$ attached to its root (by the definition). Hence B_{k+1} has $B_0, B_1, \ldots, B_{k-1}$, and $B_{k,2}$ all as children of its root. But $B_{k,2}$ is a proper binomial tree of size k, which proves the claim.

Because the claim was shown to be true for $k = k_0 = 1$ (base case) and the assumption of its truthfulness for some k led to proving it for $k+1$ (the inductive step), we can conclude that the claim is true for all natural $k \geq k_0$.

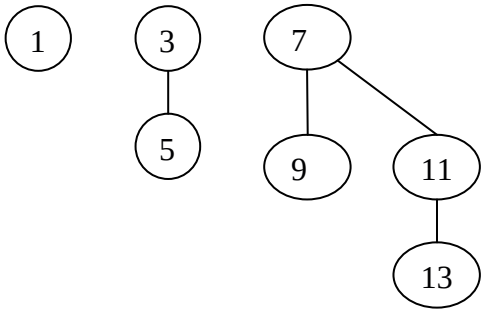
b)

There are many ways to do this problem – the following is just one way.

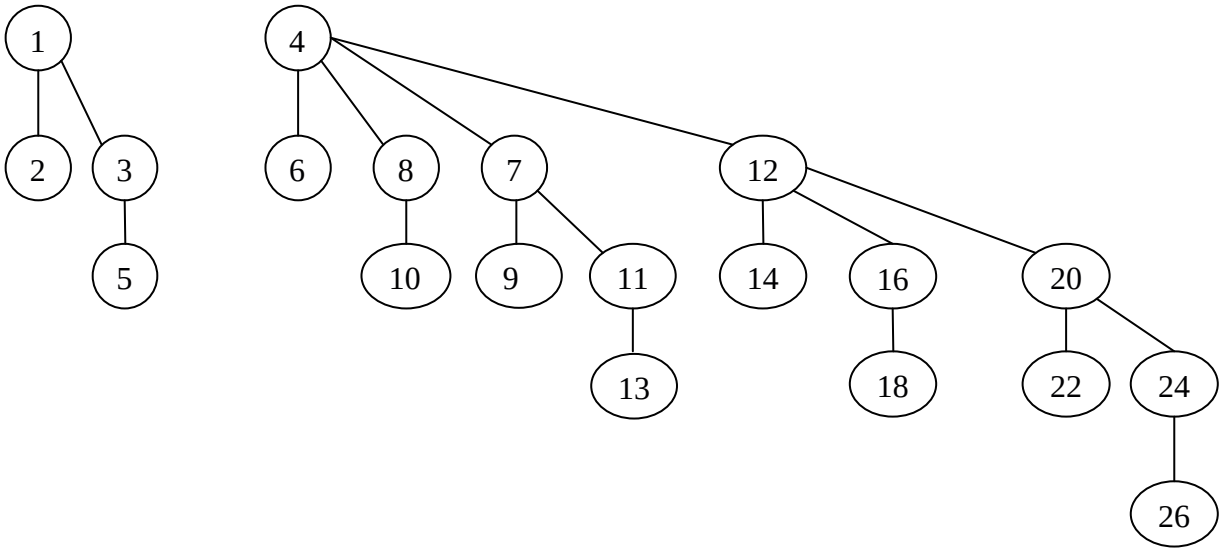
A binomial queue that holds the first 13 even numbers:



A binomial queue that holds the first 7 even numbers:



After merging the two binomial queues above, one possible result is:



Problem 3

a.

Check all pairs:

```
for i=1 to n-1
    for j = i+1 to n
       If (a[i]+a[j] =s) return 'yes'
```

Return ‘no’

Every pair is checked, the number of iterations is (n-1)+(n-2)+...+1 = $O(n^2)$. Each iteration is O(1). The space comp. is O(1) since only two variables (i,j) are needed.

b.

```
Allocate a hash-table of size m where m=O(n).
Select a hash-function and a collision-handle strategy (any can do).
for i = 1 to n do
    search for (s-a[i]) in the hash-table
    if find then return ‘yes’
            else insert a[i] into the hash table.
return ‘no’
```

Time complexity: Each element is inserted once, and causes one ‘search’ operation. So the total number of hash-operations is at most 2n, each takes on average O(1), which gives a total of O(n). Space complexity: O(n) – a reasonable size for the hash-table is about 2n.

Note: If you first insert all the elements into a hash-table, and then go through the array and search for (s-a[i]) (for each i), you have to pay attention not to say ‘yes’ if s=2*a[i]. In this case you have to search for two elements with value a[i].

Problem 4

4.1. (iii) Two possible insertion orders which result in the same contents of the hash table are 12→13→26→25→8→21→9 and 8→21→9→12→13→26→25. In one of them 25 comes last while in the other it does not, therefore the answer is (iii). To get a better idea of how we obtained those sequences see the (related) answer to 5.2.

4.2. (i) Since 25 mod 13 = 12, the original slot number 25 attempts to occupy is 12. Since 25 is not in slot 12, that slot must have been occupied already and a collision must have occurred. The specific linear probing used to resolve collisions suggests that the next slots (in that order) that 25 would attempt are 0, 1, 2, and so on, until either an empty slot is discovered or the hash table is found to be full. Since element 25 is not in slots 0 or 1 either, those two slots must also have been occupied prior to its arrival, while slot 2 must have been empty. Therefore, at least the (three) slots 12, 0, and 1 must have been occupied at the time of the arrival of 25. Hence at least three numbers were inserted prior to 25.

4.3. (iii) Two possible insertion orders are …8→12… and …12→8… Since both numbers occupy their original slots (8 and 12, respectively), there must have been no collisions on the insertion of either. Therefore, they could have come in any order with respect to each other; hence the answer is (iii).

4.4. (ii) Since 8 mod 13 = 8 and 21 mod 13 = 8 too, upon entry in the hash table both numbers 8 and 21 would first attempt to fill in slot 8. Apparently 8 succeeded, which means that it came before 21. If we assume that 9 was inserted before 8, from the fact that 8 preceded 21 it will follow (by transitivity) that 9 preceded 21. (*)

However, 9 mod 13 = 9, so the original slot that number 9 would attempt to occupy is 9. Since 21 actually occupies slot 9, it must have arrived before number 9, which leads to a contradiction with (*). The reason for the contradiction must be the incorrect assumption we made earlier – that 9 was inserted before 8. Therefore, no insertion order (that produces the current hash table’s contents) could have had 9 precede 8; hence the answer is (ii).

Note: We assume that the only operations that led from an empty hash table to its current contents were insertions from the given set of 7 elements. If there were deletions (especially of elements not from that set) too, some of the above arguments would need to be more complicated and the answers would likely differ.

Problem 5.

In a topological sort we need to start with vertices that have no incoming edges. These correspond to columns in the adjacency matrix *M* that have all values equal to 0.
In our case, both vertices *E* and *F* are appropriate starting points. Let us choose them in this order: *E*, then *F*. Removing the corresponding rows and columns for *E* and *F* we now have to solve the same problem, only on a smaller matrix (graph).
The removal of the rows *E* and *F* resulted in column *A* now consisting of all 0's. We therefore choose *A* as the next vertex and remove its corresponding row and column.
Continuing in the same fashion, we pick vertex *B* next, followed by *C* and finally by *D*.
Therefore, the topological sort yielded the following sequence of vertices: *E, F, A, B, C, D.*