COMP 3804 — Solutions Assignment 2

Question 1: Write your name and student number.

Solution: Uriah Heep, 1969

Question 2: You are given k sorted lists $L_1, L_2, \ldots, L_k$ of numbers. Let n denote the total length of all these lists.

Describe an algorithm that returns one list containing all these n numbers in sorted order. The running time of your algorithm must be $O(n \log k)$.

Explain why your algorithm is correct and why the running time is $O(n \log k)$.
Hint: If $k = 2$, this should look familiar.

Solution: The basic approach is as follows: We are going to traverse all lists simultaneously. The k numbers that we are currently visiting (one per list) are stored in a min-heap. In one iteration, we take the smallest number, say x, in the heap and remove it from the min-heap. This number x is added at the end of the output list (which, at the end, will contain all elements in sorted order). Let i be the index such that x is in L_i. Then, in L_i, we delete x, take the new first number, say y, and insert it into the heap. More formally:

1. For each $i = 1, 2, \ldots, k$, let x_i be the smallest number in L_i; this is the first number in L_i. Build a min-heap H for the numbers $x_1, x_2, \ldots, x_k$.
2. Initialize an empty list R.
3. While the min-heap is not empty:
   (a) Find the smallest number, say x, in H and delete it. Note that this is the ExtractMin operation.
   (b) Add x at the end of the list R.
   (c) Let i be such that x is in L_i. (Using indices/pointers, x "knows" the value of i.)
   (d) Delete x from L_i, and let y be the new first number in L_i. Insert y into the min-heap H.
4. Return the list R.

The reason this is correct is the same as for the merge function in merge-sort (in which case $k = 2$). Since all lists are sorted, the overall smallest number, say x, must be the smallest among the first elements in all lists. The algorithm finds x in the first iteration of the while-loop. It then removes x, and repeats the same process. Thus, in the second iteration of the while-loop, the algorithm finds the overall second smallest element, in the third iteration, the algorithm finds the overall third smallest element, in the fourth iteration, the algorithm finds the overall fourth element, etc.

What is the running time: Step 1 takes $O(k)$ time. Step 2 takes $O(1)$ time. One iteration of the while-loop in Step 3 takes $O(\log k)$ time. Since there are n iterations, the total time for Step 3 is $O(n \log k)$. Step 4 takes $O(k)$ time. Thus, the total running time is

$$O(k) + O(1) + O(n \log k) + O(k).$$

Since $k \leq n$, this is $O(n \log k)$.

Question 3: This is a long question. Don't be intimidated! As always, for each part in this question, you must justify your answer.

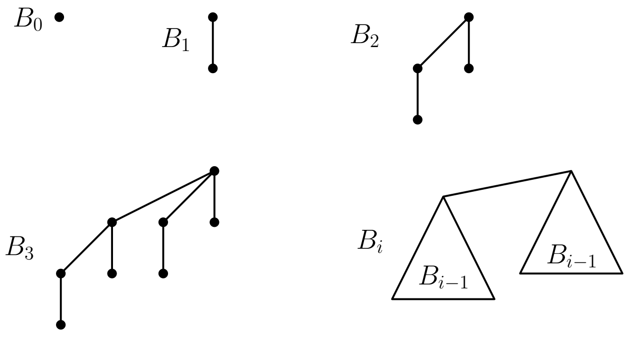
Professor Justin Bieber needs a data structure that maintains a collection $A, B, C, \ldots$ of sets under the following operations:

1. Maximum(X): return the largest element in the set X.
2. Insert(X, y): add the number y to the set X.
3. ExtractMax(X): delete and return the largest element in the set X.
4. Combine(X, Y): take the union $X \cup Y$ of the sets X and Y, and call the resulting set X.

Professor Bieber knows how to support the first three operations: Store each set X in a max-heap. The fourth operation seems to be more problematic, because we have to take two max-heaps and combine them into one max-heap.

To support all four operations, Professor Bieber has invented the following sequence $B_0, B_1, B_2, \ldots$ of trees, which are now universally known as *Bieber trees*:

1. B_0 is a tree with one node.
2. For each $i \geq 1$, the tree B_i is obtained as follows: Take two copies of B_{i-1} and make the root of one copy a child of the root of the other copy.



Question 3.1: Let $i \geq 0$. How many nodes does the tree B_i have?
Solution: Let n_i denote the number of nodes in B_i. Then $n_0 = 1$ and, for $i \geq 1$, $n_i = 2 \cdot n_{i-1}$. A straightforward induction shows that $n_i = 2^i$ for all $i \geq 0$.

Question 3.2: Let $i \geq 0$. What is the height of the tree B_i?
Solution: Let h_i denote the height of B_i. Then $h_0 = 0$ and, for $i \geq 1$, $h_i = 1 + h_{i-1}$. A straightforward induction shows that $h_i = i$ for all $i \geq 0$.

Question 3.3: Let $i \geq 1$. Prove that the subtrees of the root of B_i are the Bieber trees $B_0, B_1, \ldots, B_{i-1}$.
Solution: The proof is by induction. For $i = 1$, the subtree of the root of B_1 is the tree B_0.

Let $i \geq 2$, and assume the claim is true for $i - 1$. Consider the root, say r, of the tree B_i. By the definition of B_i, one subtree of r is B_{i-1}. The other subtrees of r are the subtrees of the root of B_{i-1}; by induction, these are $B_0, B_1, \ldots, B_{i-2}$.

Let X be a set of n numbers, assume that $n \geq 1$, and let

$$n = (b_m, b_{m-1}, \ldots, b_1, b_0)$$

be the binary representation of n. Note that $b_m = 1$ and

$$n = \sum_{i=0}^{m} b_i \cdot 2^i.$$

The *Bieber max-heap* for the set X is obtained as follows:

1. Partition the set X, arbitrarily, into subsets such that for each i for which $b_i = 1$, there is exactly one subset of size 2^i.

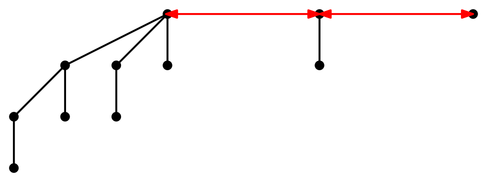
   For example, if $n = 11 = 2^3 + 2^1 + 2^0$, the set X is partitioned into three subsets: one of size 2^3, one of size 2^1, and one of size 2^0.

2. Each subset of size 2^i is stored in a Bieber tree B_i. Each node in B_i stores one element of the subset. Each node in B_i has pointers to its parent and all its children. There is a pointer to the root of B_i.

3. Each Bieber tree has the property that the value stored at a node is larger than the values stored at any of its children.

4. The roots of all these Bieber trees are connected using a doubly-linked list.

The figure below gives an example when $n = 11$.



Question 3.4: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation $\textsc{Maximum}(X)$ in $O(\log |X|)$ time.
Solution: To find the largest number in the Bieber max-heap, traverse the list that connects the roots of all Bieber trees. The largest number in this list is the largest number in the set X.

How much time does this take: The size of the list is equal to the number of 1's in the binary representation of $|X|$, which is $O(\log |X|)$. Thus, the entire operation takes $O(\log |X|)$ time.

Question 3.5: Let X and Y be two sets of numbers, and assume that both sets have the same size 2^i. A Bieber max-heap for X consists of one single Bieber tree B_i. Similarly, a Bieber max-heap for Y consists of one single Bieber tree B_i. Describe an algorithm that implements the operation $\textsc{Combine}(X, Y)$ in $O(1)$ time.
Solution: The Bieber max-heap for $X \cup Y$ consists of one tree B_{i+1}. To obtain this tree, we do the following:

1. Let x be the number stored at the root of the tree B_i^X for X.
2. Let y be the number stored at the root of the tree B_i^Y for Y.
3. If $x < y$, then we make the root of B_i^X a child of the root of B_i^Y.
4. If $x \geq y$, then we make the root of B_i^Y a child of the root of B_i^X.

This takes $O(1)$ time.

Question 3.6: Let X and Y be two non-empty sets of numbers, and assume that X is stored in a Bieber max-heap and Y is stored in a Bieber max-heap. Describe an algorithm that implements the operation $\textsc{Combine}(X, Y)$ in $O(\log |X| + \log |Y|)$ time.
Hint: This operation computes one Bieber max-heap storing the union $X \cup Y$. If you take the sum of two integers, both given in binary, then you go through the bits from right to left and keep track of a carry bit.
Solution: Consider the binary representations of the sizes of X and Y:

$$|X| = (a_k, a_{k-1}, \ldots, a_1, a_0)$$

and

$$|Y| = (b_\ell, b_{\ell-1}, \ldots, b_1, b_0),$$

where $a_k = 1$ and $b_\ell = 1$. We may assume that $k \geq \ell$; if this is not the case, then we swap X and Y. We define

$$a_{k+1} = b_{k+1} = b_k = \cdots = b_{\ell+1} = 0.$$

The Bieber max-heap BMH^X for X has one tree B_i^X for every i for which $a_i = 1$. The Bieber max-heap BMH^Y for Y has one tree B_i^Y for every i for which $b_i = 1$.

Let $Z = X \cup Y$. We show below how to obtain a Bieber max-heap BMH^Z for the set Z. This max-heap will have one tree B_i^Z for every i such that the i-th bit (from the right) in the binary representation of $|Z|$ is equal to one.

1. Initialize $c = null$. Note: c will be the "carry Bieber tree".
2. For each $i = 0, 1, 2, \ldots, k+1$, do the following:
   (a) If $c = null$:
      i. If $a_i = 0$ and $b_i = 0$: Do nothing (there is no B_i^Z in BMH^Z).
      ii. If $a_i = 1$ and $b_i = 0$: Set $B_i^Z = B_i^X$.
      iii. If $a_i = 0$ and $b_i = 1$: Set $B_i^Z = B_i^Y$.
      iv. If $a_i = 1$ and $b_i = 1$: Set $c = \textsc{Combine}(B_i^X, B_i^Y)$.
   (b) Else (i.e., $c \neq null$):
      i. If $a_i = 0$ and $b_i = 0$: Set $B_i^Z = c$.
      ii. If $a_i = 1$ and $b_i = 0$: Set $c = \textsc{Combine}(c, B_i^X)$.
      iii. If $a_i = 0$ and $b_i = 1$: Set $c = \textsc{Combine}(c, B_i^Y)$.
      iv. If $a_i = 1$ and $b_i = 1$: Set $B_i^Z = B_i^X$ and $c = \textsc{Combine}(c, B_i^Y)$.

What is the running time: First note that $k = O(\log|X|)$ and $\ell = O(\log|Y|)$. Recall that we assumed that $k \geq \ell$. Using **3.5**, one iteration of the for-loop takes $O(1)$ time. Thus, the entire for-loop takes time $O(\log k)$, which is $O(\log|X| + \log|Y|)$.

Question 3.7: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation $\textsc{Insert}(X, y)$ in $O(\log|X|)$ time.

Note that this operation computes a Bieber max-heap for the set $X \cup \{y\}$.

Solution: Let $Y = \{y\}$. We construct, in $O(1)$ time, a Bieber max-heap for the set Y; it consists of one single tree B_0^Y storing the number y. Then we call $\textsc{Combine}(X, Y)$. By **3.6**, this takes $O(\log|X|)$ time.

Question 3.8: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. Describe an algorithm that implements the operation $\textsc{ExtractMax}(X)$ in $O(\log|X|)$ time.

Note that this operation computes a Bieber max-heap for the set $X \setminus \{y\}$, where y is the largest number in X.

Solution: First we call $\textsc{Maximum}(X)$, which, by **3.4**, takes $O(\log|X|)$ time. The largest number, say y, in X will be returned at the end of the operation. Let i be the index such that the largest element in X is the root of the tree B_i^X. Let X' be the set obtained by removing all numbers stored in B_i^X from the set X, and let Y' be the set obtained by removing y from the set of all numbers that are stored in B_i^X. Our goal is to compute a Bieber max-heap for $X' \cup Y'$.

We remove B_i^X from the Bieber max-heap for X. This gives a Bieber max-heap for X'. Next we remove the root from B_i^X. By **3.3**, the subtrees of the (now removed) root are trees $B_0, B_1, \ldots, B_{i-1}$; these form a Bieber max-heap for Y'. Thus, we can call COMBINE(X', Y') to obtain a Bieber max-heap for $X' \cup Y'$. From the previous results, all of this takes $O(\log |X|)$ time.

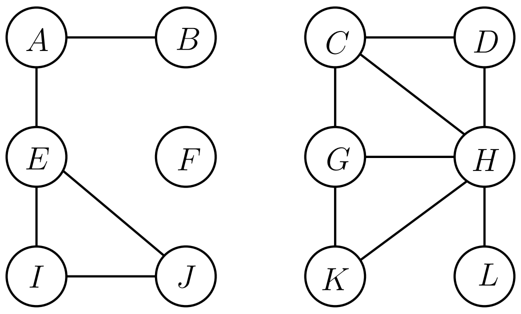
Question 3.9: Let X be a non-empty set of numbers, and assume that this set is stored in a Bieber max-heap. How would you extend this data structure such that the operation MAXIMUM(X) only takes $O(1)$ time, whereas the running times for the other operations COMBINE, INSERT, and EXTRACTMAX remain as above?

Solution:

1. We store a variable max^X, whose value is the largest number in X.
2. Obviously, the operation MAXIMUM(X) takes $O(1)$ time.
3. At the end of the operation COMBINE(X, Y): Let $Z = X \cup Y$. We set max^Z to the larger of max^X and max^Y.
4. At the end of the operation INSERT(X, y), we set max^X to the larger of max^X and y.
5. At the end of the operation EXTRACTMAX(X), we set max^X to the value obtained by running MAXIMUM on the resulting Bieber max-heap.

Remark: As you can imagine, Justin Bieber did not invent this data structure. The "mergeable" heap in this question is called *binomial heap*. See wikipedia.

Question 4: Consider the following undirected graph:

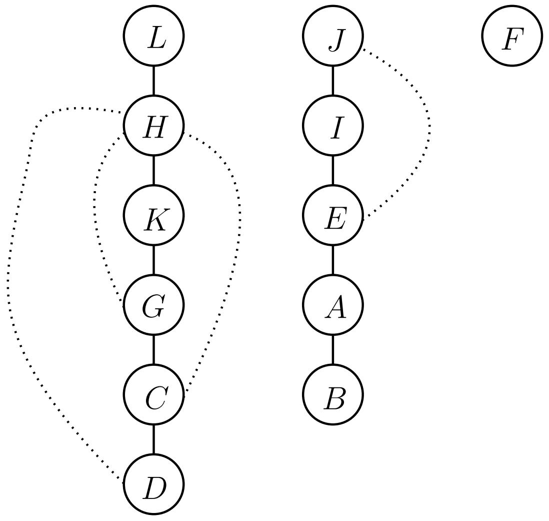


Draw the DFS-forest obtained by running algorithm DFS on this graph. The pseudocode is given at the end of this assignment. Algorithm DFS uses algorithm EXPLORE as a subroutine; the pseudocode for this subroutine is also given at the end of this assignment.

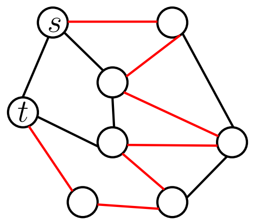
In the forest, draw each tree edge as a solid edge, and draw each back edge as a dotted edge.

Whenever there is a choice of vertices, pick the one that is alphabetically last.

Solution: In case there is more than choice, we pick the alphabetically largest one. Thus, the for-loop in algorithm DFS(G) goes through the vertices in the order $L, K, J, \ldots, A$. In the for-loop of algorithm EXPLORE, we also go in reverse alphabetical order. For example, the for-loop of EXPLORE(H) goes through H's neighbors in the order K, G, D, C. Here is the resulting DFS-forest:



Question 5: Tyler is not only your friendly TA, he is also the inventor of Tyler paths and Tyler cycles in graphs: A *Tyler path* in an undirected graph is a path that contains every vertex exactly once. In the figure below, you see a Tyler path in red. A *Tyler cycle* is a cycle that contains every vertex exactly once. In the figure below, if you add the black edge $\{s, t\}$ to the red Tyler path, then you obtain a Tyler cycle.



If $G = (V, E)$ is an undirected graph, then the graph G^3 is defined as follows:

1. The vertex set of G^3 is equal to V.
2. For any two distinct vertices u and v in V, $\{u, v\}$ is an edge in G^3 if and only if there is a path in G between u and v consisting of at most three edges.

Question 5.1: Describe a *recursive* algorithm TYLERPATH that has the following specification:

Algorithm TylerPath(T,u,v):
Input: A tree T with at least two vertices; two distinct vertices u and v in T such that $\{u,v\}$ is an edge in T.
Output: A Tyler path in T^3 that starts at vertex u and ends at vertex v.

Hint: You do not have to analyze the running time. The base case is easy. Now assume that T has at least three vertices. If you remove the edge $\{u,v\}$ from T, then you obtain two trees T_u (containing u) and T_v (containing v).

1. One of these two trees, say, T_u, may consist of the single vertex u. How does your recursive algorithm proceed?

2. If each of T_u and T_v has at least two vertices, how does your recursive algorithm proceed?

Solution: Algorithm TylerPath(T,u,v) does the following:

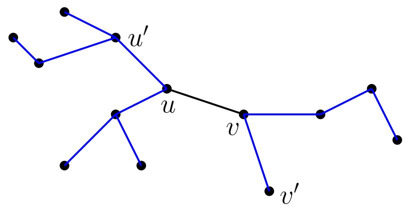
1. If T consists of two vertices: Return the path consisting of the single edge $\{u,v\}$.

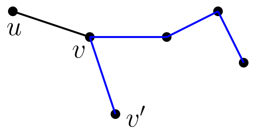
2. If T has at least three vertices: Let T_u and T_v be the two trees obtained by removing the edge $\{u,v\}$ from T.

   (a) If each of T_u and T_v has at least two vertices (see the left figure below): Let u' be a neighbor of u in T_u, and let v' be a neighbor of v in T_v. Run algorithm TylerPath(T_u,u,u') and let P be the path returned; note that P is a Tyler path in T_u^3 that starts at u and ends at u'. Run algorithm TylerPath(T_v,v',v) and let Q be the path returned; note that Q is a Tyler path in T_v^3 that starts at v' and ends at v. Note that, since u' and v' have distance three in T, the edge $\{u',v'\}$ is in T^3. Thus, we return the path that starts by following P, then takes the edge $\{u',v'\}$, and then follows Q. This is a Tyler path in T^3 that starts at u and ends at v.

   (b) If T_u consists of the single vertex u and T_v has at least two vertices (see the right figure below): Let v' be a neighbor of v in T_v. Run algorithm TylerPath(T_v,v',v) and let Q be the path returned; note that Q is a Tyler path in T_v^3 that starts at v' and ends at v. Note that, since u and v' have distance two in T, the edge $\{u,v'\}$ is in T^3. Thus, we return the path that starts with the edge $\{u,v'\}$ and then follows Q. This is a Tyler path in T^3 that starts at u and ends at v.

   (c) If T_u has at least two vertices and T_v consists of the single vertex v: Swap u and v and proceed as in the previous case.





Question 5.2: Prove the following lemma:

Tuttle's Lemma: For every tree T that has at least three vertices, the graph T^3 contains a Tyler cycle.

Solution: Take an arbitrary edge $\{u, v\}$ in T. Algorithm TylerPath(T, u, v) gives us a Tyler path in T^3 that starts at u and ends at v. This path does not contain the edge $\{u, v\}$: This is because T has at least three vertices. If we connect the end-vertices u and v of this path using the edge $\{u, v\}$, then we obtain a Tyler cycle in T^3.

Question 5.3: Prove the following theorem:

Tuttle's Theorem: For every connected undirected graph G that has at least three vertices, the graph G^3 contains a Tyler cycle.

Solution: We run algorithm DFS(G). Since G is connected, this gives us a spanning tree, say T, of G. We have seen above that T^3 contains a Tyler cycle. Since T^3 is a subgraph of G^3, this is also a Tyler cycle in G^3.

Remark: As you can imagine, Tyler did not invent the paths and cycles in this question. They are known as *Hamiltonian* paths and cycles. See wikipedia.

```
Algorithm DFS($G$):
for each vertex $u$
do $visited(u) = false$
endfor;
$cc = 0$;
for each vertex $v$
do if $visited(v) = false$
    then $cc = cc + 1$
         EXPLORE($v$)
    endif
endfor
```

```
Algorithm EXPLORE($v$):
$visited(v) = true$;
$ccnumber(v) = cc$;
for each edge $\{v, u\}$
do if $visited(u) = false$
    then EXPLORE($u$)
    endif
endfor
```