18 B-Trees

18.1 Definition of B-trees

18.1-1

> Why don't we allow a minimum degree of $t = 1$?

According to the definition, minimum degree t means every node other than the root must have at least $t - 1$ keys, and every internal node other than the root thus has at least t children. So, when $t = 1$, it means every node other than the root must have at least $t - 1 = 0$ key, and every internal node other than the root thus has at least $t = 1$ child.

Thus, we can see that the minimum case doesn't exist, because no node exists with 0 key, and no node exists with only 1 child in a B-tree.

18.1-2

> For what values of t is the tree of Figure 18.1 a legal B-tree?

According to property 5 of B-tree, every node other than the root must have at least $t - 1$ keys and may contain at most $2t - 1$ keys. In Figure 18.1, the number of keys of each node (except the root) is either 2 or 3. So to make it a legal B-tree, we need to guarantee that $t - 1 \leq 2$ and $2t - 1 \geq 3$, which yields $2 \leq t \leq 3$. So t can be 2 or 3.

18.1-3

> Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

We know that every node except the root must have at least $t - 1 = 1$ keys, and at most $2t - 1 = 3$ keys. Also remember that the leaves stay in the same depth. Thus, there are 2 possible legal B-trees:

- $$|1, 2, 3, 4, 5|$$
- $$\begin{array}{c} |3| \\ \swarrow \quad \searrow \\ |1, 2| \qquad |4, 5| \end{array}$$

18.1-4

> As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

$$\begin{aligned} n &= (1 + 2t + (2t)^2 + \cdots + (2t)^h) \cdot (2t - 1) \\ &= (2t)^{h+1} - 1. \end{aligned}$$

18.1-5

> Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

After absorbing each red node into its black parent, each black node may contain $1, 2$ (1 red child), or 3 (2 red children) keys, and all leaves of the resulting tree have the same depth, according to property 5 of red-black tree (For each node, all paths from the node to descendant leaves contain the same number of black nodes). Therefore, a red-black tree will become a Btree with minimum degree $t = 2$, i.e., a 2-3-4 tree.

18.2 Basic operations on B-trees

18.2-1

> Show the results of inserting the keys
>
> $$\mathrm{F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E}$$
>
> in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

(Omit!)

18.2-2

> Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

In order to insert the key into a full child node but without its parent being full, we need the following operations:

- DISK-READ: Key placement
- DISK-WRITE: Split nodes
- DISK-READ: Get to the parent
- DISK-WRITE: Fill parent

If both were full, we'd have to do the same, but instead of the final step, repeat the above to split the parent node and write into the child nodes. With both considerations in mind, there should never be a redundant DISK-READ or DISK-WRITE on a B-TREE-INSERT.

18.2-3

> Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

- Finding the minimum in a B-tree is quite similar to finding a minimum in a binary search tree. We need to find the left most leaf for the given root, and return the first key.
  - **PRE:** x is a node on the B-tree T. The top level call is B-TREE-FIND-MIN(T.root).
  - **POST:** FCTVAL is the minimum key stored in the subtree rooted at x.

```
B-TREE-FIND-MIN(x)
    if x == NIL           // T is empty
        return NIL
    else if x.leaf        // x is leaf
        return x.key[1]   // return the minimum key of x
    else
        DISK-READ(x.c[1])
        return B-TREE-FIND-MIN(x.c[1])
```

- Finding the predecessor of a given key $x.\mathrm{key}_i$ is according to the following rules:
  1. If x is not a leaf, return the maximum key in the i-th child of x, which is also the maximum key of the subtree rooted at $x.\mathrm{c}_i$.
  2. If x is a leaf and $i > 1$, return the $(i-1)$st key of x, i.e., $x.\mathrm{key}_{i-1}$.
  3. Otherwise, look for the last node y (from the bottom up) and $j > 0$, such that $x.\mathrm{key}_i$ is the leftmost key in $y.\mathrm{c}_j$; if $j = 1$, return NIL since $x.\mathrm{key}_i$ is the minimum key in the tree; otherwise we return $y.\mathrm{key}_{j-1}$.
     - **PRE:** x is a node on the B-tree T. i is the index of the key.
     - **POST:** FCTVAL is the predecessor of $x.\mathrm{key}_i$.

```
B-TREE-FIND-PREDECESSOR(x, i)
    if !x.leaf
        DISK-READ(x.c[i])
        return B-TREE-FIND-MAX(x.c[i])
    else if i > 1      // x is a leaf and i > 1
        return x.key[i - 1]
    else
        z = x
        while true
            if z.p == NIL    // z is root
                return NIL   // z.key[i] is the minimum key in T; no
predecessor
            y = z.p
            j = 1
```

```
            DISK-READ(y.c[1])
            while y.c[j] != x
                j = j + 1
                DISK-READ(y.c[j])
            if j == 1
                z = y
            else
                return y.key[j - 1]
```

     - **PRE:** x is a node on the B-tree T. The top level call is B-TREE-FIND-MAX(T.root).
     - **POST:** FCTVAL is the maximum key stored in the subtree rooted at x.

```
B-TREE-FIND-MAX(x)
    if x == NIL           // T is empty
        return NIL
    else if x.leaf        // x is leaf
        return x.[x.n]    // return the maximum key of x
    else
        DISK-READ(x.c[x.n + 1])
        return B-TREE-FIND-MIN(x.c[x.n + 1])
```

18.2-4 ⋆

> Suppose that we insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

The final tree can have as many as $n - 1$ nodes. Unless $n = 1$ there cannot ever be n nodes since we only ever insert a key into a non-empty node, so there will always be at least one node with 2 keys.

Next observe that we will never have more than one key in a node which is not a right spine of our B-tree. This is because every key we insert is larger than all keys stored in the tree, so it will be inserted into the right spine of the tree. Nodes not in the right spine are a result of splits, and since $t = 2$, every split results in child nodes with one key each. The fewest possible number of nodes occurs when every node in the right spine has 3 keys. In this case, $n = 2h + 2^{h+1} - 1$ where h is the height of the B-tree, and the number of nodes is $2^{h+1} - 1$. Asymptotically these are the same, so the number of nodes is $\Theta(n)$.

18.2-5

> Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

You would modify the insertion procedure by, in B-TREE-INSERT, check if the node is a leaf, and if it is, only split it if there twice as many keys stored as expected. Also, if an element needs to be inserted into a full leaf, we would split the leaf into two separate leaves, each of which doesn't have too many keys stored in it.

18.2-6

> Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. Thus, the B-TREE-SEARCH procedure needs $O(h) = O(\log_t n)$ CPU time to search along the path, where h is the height of the B-tree and n is the number of keys in the B-tree, and we know that $h \leq \log_t \frac{n+1}{2}$. Since the number of keys in each nodeis less than $2t - 1$, a binary search within each node is $O(\lg t)$. So the total time is:

$$\begin{aligned} O(\lg t \cdot \log_t n) &= O(\lg t \cdot \frac{\lg n}{\lg t}) \qquad \text{by changing the base of the logarithm.} \\ &= O(\lg n). \end{aligned}$$

Thus, the CPU time required is $O(\lg n)$.

18.2-7

> Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

$$\begin{aligned} \min \log_t n \cdot (a + bt) &= \min \frac{a + bt}{\ln t} \\ \frac{\partial}{\partial t}(\frac{a + bt}{\ln t}) &= -\frac{a + bt - bt \ln t}{t \ln^2 t} \\ a + bt &= bt \ln t \\ 5 + 10t &= 10t \ln t \\ t &= e^{W(\frac{1}{2e})+1}, \end{aligned}$$

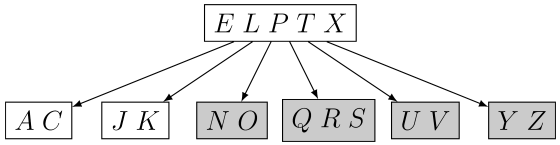
where W is the LambertW function, and we should choose $t = 3$.

18.3 Deleting a key from a B-tree

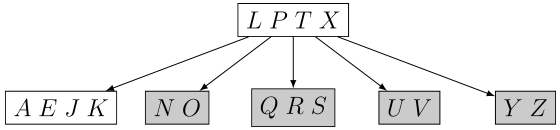
18.3-1

> Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

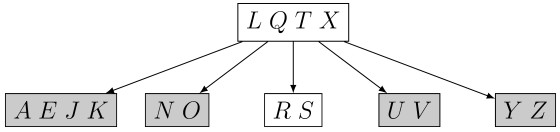
- Figure 18.8(f)



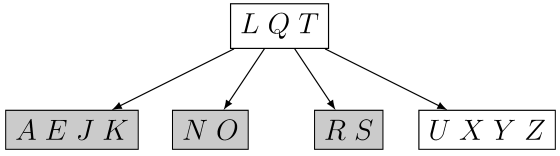
- delete C



- delete P



- delete V



18.3-2

> Write pseudocode for B-TREE-DELETE.

The algorithm B-TREE-DELETE(x, k) is a recursive procedure which deletes key k from the B-tree rooted at node x.

The functions $\text{PREDECESSOR}(k, x)$ and $\text{SUCCESSOR}(k, x)$ return the predecessor and successor of k in the B-tree rooted at x respectively.

The cases where k is the last key in a node have been omitted because the pseudocode is already unwieldy. For these, we simply use the left sibling as opposed to the right sibling, making the appropriate modifications to the indexing in the **for** loops.

Problem 18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in - memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p, the top element is the $(p \mod m)$th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

a. We will have to make a disk access for each stack operation. Since each of these disk operations takes time $\Theta(m)$, the CPU time is $\Theta(mn)$.

b. Since only every mth push starts a new page, the number of disk operations is approximately n/m, and the CPU runtime is $\Theta(n)$, since both the contribution from the cost of the disk access and the actual running of the push operations are both $\Theta(n)$.

c. If we make a sequence of pushes until it just spills over onto the second page, then alternate popping and pulling many times, the asymptotic number of disk accesses and CPU time is of the same order as in part a. This is because when we are doing that alternating of pops and pushes, each one triggers a disk access.

d. We define the potential of the stack to be the absolute value of the difference between the current size of the stack and the most recently passed multiple of m. This potential function means that the initial stack which has size 0, is also a multiple of m, so the potential is zero. Also, as we do a stack operation we either increase or decrease the potential by one. For us to have to load a new page from disk and write an old one to disk, we would need to be at least m positions away from the most recently visited multiple of m, because we would have had to just cross a page boundary. This cost of loading and storing a page takes (real) cpu time of $\Theta(m)$. However, we just had a drop in the potential function of order $\Theta(m)$. So, the amortized cost of this operation is $O(1)$.

Problem 18-2 Joining and splitting 2-3-4 trees

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.\text{key} < x.\text{key} < x''.\text{key}$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.\text{key}$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.\text{key}$. In this problem, we investigate how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.\text{height}$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$, where, for $i = 1, 2, \ldots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (Hint: The costs for joining should telescope.)

a. For insertion it will suffice to explain how to update height when we split a node. Suppose node x is split into nodes y and z, and the median of x is merged into node w. The height of w remains unchanged unless x was the root (in which case $w.\text{height} = x.\text{height} + 1$).

The height of y or z will often change. We set

$$y.\text{height} = \max_i y.c_i.\text{height} + 1$$

and

$$z.\text{height} = \max_i z.c_i.\text{height} + 1.$$

Each update takes $O(t)$. Since a call to B-TREE-INSERT makes at most h splits where h is the height of the tree, the total time it takes to update heights is $O(th)$, preserving the asymptotic running time of insert. For deletion the situation is even simple. The only time the height changes is when the root has a single node and it is merged with its subtree nodes, leaving an empty root node to be deleted. In this case, we update the height of the new node to be the (old) height of the root minus 1.

b. Without loss of generality, assume $h' \geq h''$. We essentially wish to merge T'' into T' at a node of height h'' using node x. To do this, find the node at depth $h' - h''$ on the right spine of T'. Add x as a key to this node, and T'' as the additional child. If it should happen that the node was already full, perform a split operation.

c. Let x_i be the node encountered after i steps on path p. Let l_i be the index of the largest key stored in x_i which is less than or equal to k. We take $k'_i = x_i.\text{key}_{l_i}$ and T'_{i-1} to be the tree whose root node consists of the keys in x_i which are less than $x_i.\text{key}_{l_i}$, and all of their children. In general, $T'_{i-1}.\text{height} \geq T'_i.\text{height}$.

For S'', we take a similar approach. They keys will be those in nodes passed on p which are immediately greater than k, and the trees will be rooted at a node consisting of the larger keys, with the associated subtrees. When we reach the node which contains k, we don't assign a key, but we do assign a tree.

d. Let T_1 and T_2 be empty trees. Consider the path p from the root of T to k. Suppose we have reached node x_i. We join tree T'_{i-1} to T_1, then insert $k'i$ into T_1. We join T''_{i-1} to T_2 and insert k''_i into T_2. Once we have encountered the node which contains k at $x_m.\text{key}_k$, join $x_m.c_k$ with T_1 and $x_m.c_{k+1}$ with T_2.

We will perform at most 2 join operations and 1 insert operation for each level of the tree. Using the runtime determined in part (b), and the fact that when we join a tree T' to T_1 (or T'' to T_2 respectively) the height difference is

$$T'.\text{height} - T_1.\text{height}.$$

Since the heights are nondecreasing of successive tree that are joined, we get a telescoping sum of heights. The first tree has height h, where h is the height of T, and the last tree has height 0. Thus, the runtime is

$$O(2(h + h)) = O(\lg n).$$

19 Fibonacci Heaps

19.1 Structure of Fibonacci heaps

There is no exercise in this section.

19.2 Mergeable-heap operations

19.2-1

Show the Fibonacci heap that results from calling $\text{FIB-HEAP-EXTRACT-MIN}$ on the Fibonacci heap shown in Figure 19.4(m).

(Omit!)

19.3 Decreasing a key and deleting a node

19.3-1

Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn't matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

x came to be a marked root because at some point it had been a marked child of $H.\min$ which had been removed in $\text{FIB-HEAP-EXTRACT-MIN}$ operation. See figure 19.4 for an example, where the node with key 18 became a marked root. It doesn't add the potential for having to do any more actual work for it to be marked. This is because the only time that markedness is checked is in line 3 of cascading cut. This however is only ever run on nodes whose parent is non NIL. Since every root has NIL as it parent, line 3 of cascading cut will never be run on this marked root. It will still cause the potential function to be larger than needed, but that extra computation that was paid in to get the potential function higher will never be used up later.

19.3-2

Justify the $O(1)$ amortized time of $\text{FIB-HEAP-DECREASE-KEY}$ as an average cost per operation by using aggregate analysis.

Recall that the actual cost of $\text{FIB-HEAP-DECREASE-KEY}$ is $O(c)$, where c is the number of calls made to CASCADING-CUT. If c_i is the number of calls made on the ith key decrease, then the total time of n calls to $\text{FIB-HEAPDECREASE-KEY}$ is $\sum_{i=1}^{n} O(c_i)$.

Next observe that every call to CASCADING-CUT moves a node to the root, and every call to a root node takes $O(1)$. Since no roots ever become children during the course of these calls, we must have that

$\sum_{i=1}^{n} c_i = O(n)$. Therefore the aggregate cost is $O(n)$, so the average, or amortized, cost is $O(1)$.

19.4 Bounding the maximum degree

19.4-1

Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

- **Initialize:** insert 3 numbers then extract-min.
- **Iteration:** insert 3 numbers, in which at least two numbers are less than the root of chain, then extract-min. The smallest newly inserted number will be extracted and the remaining two numbers will form a heap whose degree of root is 1, and since the root of the heap is less than the old chain, the chain will be merged into the newly created heap. Finally we should delete the node which contains the largest number of the 3 inserted numbers.

19.4-2

Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. (The rule in Section 19.3 uses $k = 2$.) For what values of k is $D(n) = O(\lg n)$?

Following the proof of lemma 19.1, if x is any node if a Fibonacci heap, $x.\text{degree} = m$, and x has children $y_1, y_2, \ldots, y_m$, then $y_1.\text{degree} \geq 0$ and $y_i.\text{degree} \geq i - k$. Thus, if s_m denotes the fewest nodes possible in a node of degree m, then we have $s_0 = 1, s_1 = 2, \ldots, s_{k-1} = k$ and in general, $s_m = k + \sum_{i=0}^{m-k} s_i$. Thus, the difference between s_m and s_{m-1} is s_{m-k}.

Let $\{f_m\}$ be the sequence such that $f_m = m + 1$ for $0 \leq m < k$ and $f_m = f_{m-1} + f_{m-k}$ for $m \geq k$.

If $F(x)$ is the generating function for f_m then we have $F(x) = \frac{1-x^k}{(1-x)(1-x-x^k)}$. Let α be a root of $x^k = x^{k-1} + 1$. We'll show by induction that $f_{m+k} \geq \alpha^m$. For the base cases:

$$\begin{aligned} f_k &= k + 1 \geq 1 = \alpha^0 \\ f_{k+1} &= k + 3 \geq \alpha^1 \\ &\vdots \\ f_{k+k} &= k + \frac{(k+1)(k+2)}{2} = k + k + 1 + \frac{k(k+1)}{2} \geq 2k + 1 + \alpha^{k-1} \geq \alpha^k. \end{aligned}$$

In general, we have

$$f_{m+k} = f_{m+k-1} + f_m \geq \alpha^{m-1} + \alpha^{m-k} = \alpha^{m-k}(\alpha^{k-1} + 1) = \alpha^m.$$

Next we show that $f_{m+k} = k + \sum_{i=0}^{m} f_i$. The base case is clear, since $f_k = f_0 + k = k + 1$. For the induction step, we have

$$f_{m+k} = f_{m-1-k} + f_m = k\sum_{i=0}^{m-1} f_i + f_m = k + \sum_{i=0}^{m} f_i.$$

Observe that $s_i \geq f_{i+k}$ for $0 \leq i < k$. Again, by induction, for $m \geq k$ we have

$$s_m = k + \sum_{i=0}^{m-k} s_i \geq k + \sum_{i=0}^{m-k} f_{i+k} \geq k + \sum_{i=0}^{m} f_i = f_{m+k}.$$

So in general, $s_m \geq f_{m+k}$. Putting it all together, we have

$$\begin{aligned} \text{size}(x) &\geq s_m \\ &\geq k + \sum_{i=k}^{m} s_{i-k} \\ &\geq k + \sum_{i=k}^{m} f_i \\ &\geq f_{m+k} \\ &\geq \alpha^m. \end{aligned}$$

Taking logs on both sides, we have

$$\log_\alpha n \geq m.$$

In other words, provided that α is a constant, we have a logarithmic bound on the maximum degree.

Problem 19-1 Alternative implementation of deletion

Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.\min$.

```
PISANO-DELETE(H, x)
    if x == H.min
        FIB-HEAP-EXTRACT-MIN(H)
    else y = x.p
        if y != NIL
            CUT(H, x, y)
            CASCADING-CUT(H, y)
        add x's child list to the root list of H
        remove x from the root list of H
```

a. The professor's claim that this procedure runs faster is based partly on the assumption that line 7 can be performed in $O(1)$ actual time. What is wrong with this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.\min$. Your bound should be in terms of $x.\text{degree}$ and the number c of calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE(H, x), and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.\text{degree}$, c, $t(H)$, and $m(H)$.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, evenwhen $x \neq H.\min$.

a. It can take actual time proportional to the number of children that x had because for each child, when placing it in the root list, their parent pointer needs to be updated to be NIL instead of x.

b. Line 7 takes actual time bounded by $x.\text{degree}$ since updating each of the children of x only takes constant time. So, if c is the number of cascading cuts that are done, the actual cost is $O(c + x.\text{degree})$.

c. We examine the number of trees in the root list $t(H)$ and the number of marked nodes $m(H)$ of the resulting Fibonacci heap H' to upper-bound its potential. The number of trees $t(H)$ increases by the number of children x had (= $x.\text{degree}$), due to line 7 of PISANO-DELETE(H, x). The number of marked nodes in H' is calculated as follows. The first $c - 1$ recursive calls out of the c calls to CASCADING-CUT unmarks a marked node (line 4 of CUT invoked by line 5 of CASCADING-CUT). The final cth call to CASCADING-CUT marks an unmarked node (line 4 of CASCADING-CUT), and therefore, the total change in marked nodes is $-(c - 1) + 1 = -c + 2$. Therefore, the potential of H' is

$$\Phi(H') \leq t(H) + x.\text{degree} + 2(m(H) - c + 2).$$

d. The asymptotic time is

$$\Theta(x.\text{degree}) = \Theta(\lg(n)),$$

which is the same asyptotic time that was required for the original deletion method.

Problem 19-2 Binomial trees and binomial heaps

The ***binomial tree*** B_k is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.6(a), the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the leftmost child of the root of the other. Figure 19.6(b) shows the binomial trees B_0 through B_4.

a. Show that for the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover, as Figure 19.6(c) shows, if we number the children of the root from left to right by $k - 1, k - 2, \ldots, 0$, then child i is the root of a subtree B_i.

A ***binomial heap*** H is a set of binomial trees that satisfies the following properties:

1. Each node has a key (like a Fibonacci heap).
2. Each binomial tree in H obeys the min-heap property.
3. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

b. Suppose that a binomial heap H has a total of n nodes. Discuss the relationship between the binomial trees that H contains and the binary representation of n. Conclude that H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Suppose that we represent a binomial heap as follows. The left-child, right-sibling scheme of Section 10.4 represents each binomial tree within a binomial heap. Each node contains its key; pointers to its parent, to its leftmost child, and to the sibling immediately to its right (these pointers are NIL when appropriate); and its degree (as in Fibonacci heaps, how many children it has). The roots form a singly linked root list, ordered by the degrees of the roots (from low to high), and we access the binomial heap by a pointer to the first node on the root list.

c. Complete the description of how to represent a binomial heap (i.e., name the attributes, describe when attributes have the value NIL, and define how the root list is organized), and show how to implement the same seven operations on binomial heaps as this chapter implemented on Fibonacci heaps. Each operation should run in $O(\lg n)$ worst-case time, where n is the number of nodes in the binomial heap (or in the case of the UNION operation, in the two binomial heaps that are being united). The MAKE-HEAP operation should take constant time.

d. Suppose that we were to implement only the mergeable-heap operations on a Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE operations). How would the trees in a Fibonacci heap resemble those in a binomial heap? How would they differ? Show that the maximum degree in an n-node Fibonacci heap would be at most $\lfloor \lg n \rfloor$.

e. Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports just the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

a.

1. B_k consists of two binomial trees B_{k-1}.
2. The height of one B_{k-1} is increased by 1.
3. For $i = 0$, $\binom{k}{0} = 1$ and only root is at depth 0. Suppose in B_{k-1}, the number of nodes at depth i is $\binom{k-1}{i}$, in B_k, the number of nodes at depth i is $\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$.
4. The degree of the root increase by 1.

b. Let $n.b$ denote the binary expansion of n. The fact that we can have at most one of each binomial tree corresponds to the fact that we can have at most 1 as any digit of $n.b$. Since each binomial tree has a size

which is a power of 2, the binomial trees required to represent n nodes are uniquely determined. We include B_k if and only if the kth position of $n.b$ is 1. Since the binary representation of n has at most $\lfloor \lg n \rfloor + 1$ digits, this also bounds the number of trees which can be used to represent n nodes.

c. Given a node x, let $x.\text{key}$, $x.p$, $x.c$, and $x.s$ represent the attributes key, parent, left-most child, and sibling to the right, respectively. The pointer attributes have value NIL when no such node exists. The root list will be stored in a singly linked list.

- **MAKE-HEAP** initialize an empty list for the root list and return a pointer to the head of the list, which contains NIL. This takes constant time. To insert: Let x be a node with key k, to be inserted. Scan the root list to find the first m such that B_m is not one of the trees in the binomial heap. If there is no B_0, simply create a single root node x. Otherwise, union $x, B_0, B_1, \ldots, B_{m-1}$ into a B_m tree. Remove all root nodes of the unioned trees from the root list, and update it with the new root. Since each join operation is logarithmic in the height of the tree, the total time is $O(\lg n)$. MINIMUM just scans the root list and returns the minimum in $O(\lg n)$, since the root list has size at most $O(\lg n)$.
- **EXTRACT-MIN:** finds and deletes the minimum, then splits the tree Bm which contained the minimum into its component binomial trees $B_0, B_1, \ldots, B_{m-1}$ in $O(\lg n)$ time. Finally, it unions each of these with any existing trees of the same size in $O(\lg n)$ time.
- **UNION:** suppose we have two binomial heaps consisting of trees $B_{i_1}, B_{i_2}, \ldots, B_{i_k}$ and $B_{j_1}, B_{j_2}, \ldots, B_{j_m}$ respectively. Simply union orresponding trees of the same size between the two heaps, then do another check and join any newly created trees which have caused additional duplicates. Note: we will perform at most one union on any fixed size of binomial tree so the total running time is still logarithmic in n, where we assume that n is sum of the sizes of the trees which we are unioning.
- **DECREASE-KEY:** simply swap the node whose key was decreased up the tree until it satisfies the min-heap property. This method requires that we swap the node with its parent along with all their satellite data in a brute-force manner to avoid updating p attributes of the siblings of the node. When the data stored in each node is large, we may want to update p instead, which, however, will increase the running time bound to $O(\lg^2 n)$.
- **DELETE:** note that every binomial tree consists of two copies of a smaller binomial tree, so we can write the procedure recursively. If the tree is a single node, simply delete it. If we wish to delete from B_k, first split the tree into its constituent copies of B_{k-1}, and recursively call delete on the copy of B_{k-1} which contains x. If this results in two binomial trees of the same size, simply union them.

d. The Fibonacci heap will look like a binomial heap, except that multiple copies of a given binomial tree will be allowed. Since the only trees which will appear are binomial trees and B_k has $2k$ nodes, we must have $2k \leq n$, which implies $k \leq \lfloor \lg n \rfloor$. Since the largest root of any binomial tree occurs at the root, and on B_k it is degree k, this also bounds the largest degree of a node.

e. INSERT and UNION will no longer have amortized $O(1)$ running time because CONSOLIDATE has runtime $O(\lg n)$. Even if no nodes are consolidated, the runtime is dominated by the check that all degrees are distinct.

Since calling UNION on a heap and a single node is the same as insertion, it must also have runtime $O(\lg n)$. The other operations remain unchanged.

Problem 19-3 More Fibonacci-heap operations

We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

a. The operation FIB-HEAP-CHANGE-KEY(H, x, k) changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to $x.\text{key}$.

b. Give an efficient implementation of FIB-HEAP-PRUNE(H, r), which deletes $q = \min(r, H.n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (Hint: You may need to modify the data structure and potential function.)

a. If $k < x.\text{key}$ just run the decrease key procedure. If $k > x.\text{key}$, delete the current value x and insert x again with a new key. For the first case, the amortized time is $O(1)$, and for the last case the amortized time is $O(\lg n)$.

b. Suppose that we also had an additional cost to the potential function that was proportional to the size of the structure. This would only increase when we do an insertion, and then only by a constant amount, so there aren't any worries concerning this increased potential function raising the amortized cost of any operations. Once we've made this modification, to the potential function, we also modify the heap itself by having a doubly linked list along all of the leaf nodes in the heap.

To prune we then pick any leaf node, remove it from it's parent's child list, and remove it from the list of leaves. We repeat this $\min(r, H.n)$ times. This causes the potential to drop by an amount proportional to r which is on the order of the actual cost of what just happened since the deletions from the linked list take only constant amounts of time each. So, the amortized time is constant.

Problem 19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement ***2-3-4 heaps***, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf x stores exactly one key in the attribute $x.\text{key}$. The keys in the leaves may appear in any order. Each internal node x contains a value $x.\text{small}$ that is equal to the smallest key stored in any leaf in the subtree rooted at x. The root r contains an attribute $r.\text{height}$ that gives the height of the tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation should run in $O(\lg n)$ time on a 2-3-4 heap with n elements. The UNION operation in part (f) should run in $O(\lg n)$ time, where n is the number of elements in the two input heaps.

a. MINIMUM, which returns a pointer to the leaf with the smallest key.

b. DECREASE-KEY, which decreases the key of a given leaf x to a given value $k \leq x.\text{key}$.

c. INSERT, which inserts leaf x with key k.

d. DELETE, which deletes a given leaf x.

e. EXTRACT-MIN, which extracts the leaf with the smallest key.

f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

a. Traverse a path from root to leaf as follows: At a given node, examine the attribute $x.\text{small}$ in each child-node of the current node. Proceed to the child node which minimizes this attribute. If the children of the current node are leaves, then simply return a pointer to the child node with smallest key. Since the height of the tree is $O(\lg n)$ and the number of children of any node is at most 4, this has runtime $O(\lg n)$.

b. Decrease the key of x, then traverse the simple path from x to the root by following the parent pointers. At each node y encountered, check the attribute $y.\text{small}$. If $k < y.\text{small}$, set $y.\text{small} = k$. Otherwise do nothing and continue on the path.

c. Insert works the same as in a B-tree, except that at each node it is assumed that the node to be inserted is 'smaller' than every key stored at that node, so the runtime is inherited. If the root is split, we update the height of the tree. When we reach the final node before the leaves, simply insert the new node as the leftmost child of that node.

d. As with B-TREE-DELETE, we'll want to ensure that the tree satisfies the properties of being a 2-3-4 tree after deletion, so we'll need to check that we're never deleting a leaf which only has a single sibling. This is handled in much the same way as in chapter 18. We can imagine that dummy keys are stored in all the internal nodes, and carry out the deletion process in exactly the same way as done in exercise 18.3-2, with the added requirement that we update the height stored in the root if we merge the root with its child nodes.

e. EXTRACT-MIN simply locates the minimum as done in part (a), then deletes it as in part (d).

f. This can be done by implementing the join operation, as in Problem 18-2 (b).

20 van Emde Boas Trees

20.1 Preliminary approaches

20.1-1

> Modify the data structures in this section to support duplicate keys.

To modify these structure to allow for multiple elements, instead of just storing a bit in each of the entries, we can store the head of a linked list representing how many elements of that value that are contained in the structure, with a NIL value to represent having no elements of that value.

20.1-2

> Modify the data structures in this section to support keys that have associated satellite data.

All operations will remain the same, except instead of the leaves of the tree being an array of integers, they will be an array of nodes, each of which stores $x.key$ in addition to whatever additional satellite data you wish.

20.1-3

> Observe that, using the structures in this section, the way we find the successor and predecessor of a value x does not depend on whether x is in the set at the time. Show how to find the successor of x in a binary search tree when x is not stored in the tree.

To find the successor of a given key k from a binary tree, call the procedure $\text{SUCC}(x, T.root)$. Note that this will return NIL if there is no entry in the tree with a larger key.

20.1-4

> Suppose that instead of superimposing a tree of degree $\sqrt{u}$, we were to superimpose a tree of degree $u^{1/k}$, where $k > 1$ is a constant. What would be the height of such a tree, and how long would each of the operations take?

The new tree would have height k. INSERT would take $O(k)$, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE would take $O(ku^{1/k})$.

20.2 A recursive structure

20.2-1

> Write pseudocode for the procedures PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

See the two algorithms, PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

20.2-2

> Write pseudocode for PROTO-vEB-DELETE. It should update the appropriate summary bit by scanning the related bits within the cluster. What is the worst-case running time of your procedure?

```
PROTO-vEB-DELETE(V, x)
    if V.u == 2
        V.A[x] = 0
    else
        PROTO-vEB-DELETE(V.cluster[high(x)], low(x))
        inCluster = false
        for i = 0 to sqrt(u) - 1
            if PROTO-vEB-MEMBER(V.cluster[high(x)], low(i))
                inCluster = true
                break
        if inCluster == false
            PROTO-vEB-DELETE(V.summary, high(x))
```

When we delete a key, we need to check membership of all keys of that cluster to know how to update the summary structure. There are $\sqrt{u}$ of these, and each membership takes $O(\lg \lg u)$ time to check. With the recursive calls, recurrence for running time is

$$T(u) = T(\sqrt{u}) + O(\sqrt{u} \lg \lg u).$$

We make the substitution $m = \lg u$ and $S(m) = T(2^m)$. Then we apply the Master Theorem, using case 3, to solve the recurrence. Substituting back, we find that the runtime is $T(u) = O(\sqrt{u} \lg \lg u)$.

20.2-3

> Add the attribute n to each proto-vEB structure, giving the number of elements currently in the set it represents, and write pseudocode for PROTO-vEB-DELETE that uses the attribute n to decide when to reset summary bits to 0. What is the worst-case running time of your procedure? What other procedures need to change because of the new attribute? Do these changes affect their running times?

We would keep the same as before, but insert immediately after the else, a check of whether $n = 1$. If it doesn't continue as usual, but if it does, then we can just immediately set the summary bit to 0, null out the pointer in the table, and be done immediately. This has the upside that it can sometimes save up to $\lg \lg u$. The procedure has the big downside that the number of elements that are in the set could be as high as $\lg(\lg u)$, in which case $\lg u$ many bits are needed to store n.

20.2-4

> Modify the proto-vEB structure to support duplicate keys.

The array A found in a proto van Emde Boas structure of size 2 should now support integers, instead of just bits. All other pats of the structure will remain the same. The integer will store the number of duplicates at that position. The modifications to insert, delete, minimum, successor, etc will be minor. Only the base cases will need to be updated.

20.2-5

> Modify the proto-vEB structure to support keys that have associated satellite data.

The only modification necessary would be for the $u = 2$ trees. They would need to also include a length two array that had pointers to the corresponding satellite data which would be populated in case the corresponding entry in A were 1.

20.2-6

> Write pseudocode for a procedure that creates a proto-vEB(u) structure.

This algorithm recursively allocates proper space and appropriately initializes attributes for a proto van Emde Boas structure of size u.

```
MAKE-PROTO-vEB(u)
    allocate a new vEB tree V
    V.u = u
    if u == 2
        let A be an array of size 2
        V.A[1] = V.A[0] = 0
    else
        V.summary = MAKE-PROTO-vEB(sqrt(u))
        for i = 0 to sqrt(u) - 1
            V.cluster[i] = MAKE-PROTO-vEB(sqrt(u))
```

20.2-7

> Argue that if line 9 of PROTO-vEB-MINIMUM is executed, then the proto-vEB structure is empty.

For line 9 to be executed, we would need that in the summary data, we also had a NIL returned. This could of either happened through line 9, or 6. Eventually though, it would need to happen in line 6, so, there must be some number of summarizations that happened of V that caused us to get an empty $u = 2$ vEB. However, a summarization has an entry of one if any of the corresponding entries in the data structure are one. This means that there are no entries in V, and so, we have that V is empty.

20.2-8

> Suppose that we designed a proto-vEB structure in which each *cluster* array had only $u^{1/4}$ elements. What would the running times of each operation be?

There are $u^{3/4}$ clusters in each proto-vEB.

- **MEMBER/INSERT:**

$$T(u) = T(u^{1/4}) + O(1) = \Theta(\lg \log_4 u) = \Theta(\lg \lg u).$$

- **MINIMUM/MAXIMUM:**

$$T(u) = T(u^{1/4}) + T(u^{3/4}) + O(1) = \Theta(\lg u).$$

- **SUCCESSOR/PREDECESSOR/DELETE:**

$$T(u) = T(u^{1/4}) + T(u^{3/4}) + \Theta(\lg u^{1/4}) = \Theta(\lg u \lg \lg u).$$

20.3 The van Emde Boas tree

20.3-1

> Modify vEB trees to support duplicate keys.

To support duplicate keys, for each $u = 2$ vEB tree, instead of storing just a bit in each of the entries of its array, it should store an integer representing how many elements of that value the vEB contains.

20.3-2

> Modify vEB trees to support keys that have associated satellite data.

For any key which is a minimum on some vEB, we'll need to store its satellite data with the min value since the key doesn't appear in the subtree. The rest of the satellite data will be stored alongside the keys of the vEB trees of size 2. Explicitly, for each non-summary vEB tree, store a pointer in addition to min. If min is NIL, the pointer should also point to NIL. Otherwise, the pointer should point to the satellite data associated with that minimum. In a size 2 vEB tree, we'll have two additional pointers, which will each point to the minimum's and maximum's satellite data, or NIL if these don't exist. In the case where $\min = \max$, the pointers will point to the same data.

20.3-3

> Write pseudocode for a procedure that creates an empty van Emde Boas tree.

We define the procedure for any u that is a power of 2. If $u = 2$, then, just slap that fact together with an array of length 2 that contains 0 in both entries.

If $u = 2k > 2$, then, we create an empty vEB tree called Summary with $u = 2^{\lceil k/2 \rceil}$. We also make an array called cluster of length $2^{\lceil k/2 \rceil}$ with each entry initialized to an empty vEB tree with $u = 2^{\lfloor k/2 \rfloor}$. Lastly, we create a min and max element, both initialized to NIL.

20.3-4

> What happens if you call VEB-TREE-INSERT with an element that is already in the vEB tree? What happens if you call VEB-TREE-DELETE with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.

= 0. If it's not, simply return. If it is, set A[x] = 1 and proceed with insert as usual. When deleting x, check if A[x] = 1. If it isn't, simply return. If it is, set A[x] = 0$ and proceed with delete as usual.

20.3-5

> Suppose that instead of $\sqrt{u}$ clusters, each with universe size $\sqrt{u}$, we constructed vEB trees to have $u^{1/k}$ clusters, each with universe size $u^{1-1/k}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{1/k}$ and $u^{1-1/k}$ are always integers.

Similar to the analysis of (20.4), we will analyze

$$T(u) \le T(u^{1-1/k}) + T(u^{1/k}) + O(1).$$

This is a good choice for analysis because for many operations we first check the summary vEB tree, which will have size $u^{1/k}$ (the second term). And then possible have to check a vEB tree somewhere in cluster, which will have size $u^{1-1/k}$ (the first term). We let $T(2^m) = S(m)$, so the equation becomes

$$S(m) \le S(m(1 - 1/k)) + S(m/k) + O(1).$$

If $k > 2$ the first term dominates, so by master theorem, we'll have that $S(m)$ is $O(\lg m)$, this means that T will be $O(\lg(\lg u))$ just as in the original case where we took squareroots.

20.3-6

> Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg \lg u)$?

Set $n = u/\lg \lg u$. Then performing n operations takes $c(u + n \lg \lg u)$ time for some constant c. Using the aggregate amortized analysis, we divide by n to see that the amortized cost of each operations is $c(\lg \lg u + \lg \lg u) = O(\lg \lg u)$ per operation. Thus we need $n \ge u/\lg \lg u$.

Problem 20-1 Space requirements for van Emde Boas trees

> This problem explores the space requirements for van Emde Boas trees and suggests a way to modify the data structure to make its space requirement depend on the number n of elements actually stored in the tree, rather than on the universe size u. For simplicity, assume that $\sqrt{u}$ is always an integer.

> **a.** Explain why the following recurrence characterizes the space requirement $P(u)$ of a van Emde Boas tree with universe size u:
>
> $$P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u}). \qquad (20.5)$$
>
> **b.** Prove that recurrence (20.5) has the solution $P(u) = O(u)$.
>
> In order to reduce the space requirements, let us define a ***reduced-space van Emde Boas tree***, or ***RS-vEB tree***, as a vEB tree V but with the following changes:
>
> - The attribute $V.cluster$, rather than being stored as a simple array of pointers to vEB trees with universe size $\sqrt{u}$, is a hash table (see Chapter 11) stored as a dynamic table (see Section 17.4). Corresponding to the array version of $V.cluster$, the hash table stores pointers to RS-vEB trees with universe size $\sqrt{u}$. To find the ith cluster, we look up the key i in the hash table, so that we can find the ith cluster by a single search in the hash table.
> - The hash table stores only pointers to nonempty clusters. A search in the hash table for an empty cluster returns NIL, indicating that the cluster is empty.
> - The attribute $V.summary$ is NIL if all clusters are empty. Otherwise, $V.summary$ points to an RS-vEB tree with universe size $\sqrt{u}$.
>
> Because the hash table is implemented with a dynamic table, the space it requires is proportional to the number of nonempty clusters.
>
> When we need to insert an element into an empty RS-vEB tree, we create the RS-vEB tree by calling the following procedure, where the parameter u is the universe size of the RS-vEB tree:
>
> ```
> CREATE-NEW-RS-vEB-TREE(u)
>     allocate a new vEB tree V
>     V.u = u
>     V.min = NIL
>     V.max = NIL
>     V.summary = NIL
>     create V.cluster as an empty dynamic hash table
>     return V
> ```
>
> **c.** Modify the VEB-TREE-INSERT procedure to produce pseudocode for the procedure RS-VEB-TREE-INSERT(V, x), which inserts x into the RS-vEB tree V, calling CREATE-NEW-RS-VEB-TREE as appropriate.
>
> **d.** Modify the VEB-TREE-SUCCESSOR procedure to produce pseudocode for the procedure RS-VEB-TREE-SUCCESSOR(V, x), which returns the successor of x in RS-vEB tree V, or NIL if x has no successor in V.
>
> **e.** Prove that, under the assumption of simple uniform hashing, your RS-VEBTREE-INSERT and RS-VEB-TREE-SUCCESSOR procedures run in $O(\lg \lg u)$ expected time.

f. Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.

g. RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

a. Lets look at what has to be stored for a vEB tree. Each vEB tree contains one vEB tree of size $\sqrt{u}$ and $\sqrt{u}$ vEB trees of size $\sqrt{u}$. It also is storing three numbers each of order $O(u)$, so they need $\Theta(\lg(u))$ space each. Lastly, it needs to store $\sqrt{u}$ many pointers to the cluster vEB trees. We'll combine these last two contributions which are $\Theta(\lg(u))$ and $\Theta(\sqrt{u})$ respectively into a single term that is $\Theta(\sqrt{u})$. This gets us the recurrence

$$P(u) = P(\sqrt{u}) + \sqrt{u}P(\sqrt{u}) + \Theta(\sqrt{u}).$$

Then, we have that $u = 2^{2m}$ (which follows from the assumption that $\sqrt{u}$ was an integer), this equation becomes

$$\begin{aligned} P(u) &= (1 + 2^m)P(2^m) + \Theta(\sqrt{u}) \\ &= (1 + \sqrt{u})P(\sqrt{u}) + \Theta(\sqrt{u}) \end{aligned}$$

as desired.

b. We recall from our solution to problem 3-6.e (it seems like so long ago now) that given a number n, a bound on the number of times that we need to take the squareroot of a number before it falls below 2 is $\lg \lg n$. So, if we just unroll out recurrence, we get that

$$P(u) \leq \left(\prod_{i=1}^{\lg \lg u} (u^{1/2^i} + 1) \right) P(2) + \sum_{i=1}^{\lg \lg u} \Theta(u^{1/2^i})(u^{1/2i} + 1).$$

The first product has a highest power of u corresponding to always multiplying the first terms of each binomial. The power in this term is equal to $\sum_{i=1}^{\lg \lg u}$ which is a partial sum of a geometric series whose sum is 1. This means that the first term is $o(u)$. The order of the ith term in the summation appearing in the formula is $u^{2/2^i}$. In particular, for $i = 1$ is it $O(u)$, and for any $i > 1$, we have that $2/2^i < 1$, so those terms will be $o(u)$. Putting it all together, the largest term appearing is $O(u)$, and so, $P(u)$ is $O(u)$.

c. For this problem we just use the version written for normal vEB trees, with minor modifications. That is, since there are entries in cluster that may not exist, and summary may of not yet been initialized, just before we try to access either, we check to see if it's initialized. If it isn't, we do so then.

d. As in the previous problem, we just wait until just before either of the two things that may of not been allocated try to get used then allocate them if need be.

e. Since the initialization performed only take constant time, those modifications don't ruin the the desired runtime bound for the original algorithms already had. So, our responses to parts (c) and (d) are $O(\lg \lg n)$.

f. As mentioned in the errata, this part should instead be changed to $O(n \lg n)$ space. When we are adding an element, we may have to add an entry to a dynamic hash table, which means that a constant amount of extra space would be needed. If we are adding an element to that table, we also have to add an element to the RS-

vEB tree in the summary, but the entry that we add in the cluster will be a constant size RS-vEB tree. We can charge the cost of that addition to the summary table to the making the minimum element entry that we added in the cluster table. Since we are always making at least one element be added as a new min entry somewhere, this amortization will mean that it is only a constant amount of time in order to store the new entry.

g. It only takes a constant amount of time to create an empty RS-vEB tree. This is immediate since the only dependence on u in CREATE-NEW-RSvEB-TREE(u) is on line 2 when $V.u$ is initialized, but this only takes a constant amount of time. Since nothing else in the procedure depends on u, it must take a constant amount of time.

Problem 20-2 y-fast tries

This problem investigates D. Willard's "y-fast tries" which, like van Emde Boas trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on elements drawn from a universe with size u in $O(\lg \lg u)$ worst-case time. The INSERT and DELETE operations take $O(\lg \lg u)$ amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), y-fast tries use only $O(n)$ space to store n elements. The design of y-fast tries relies on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing not only every element in the dynamic set, but every prefix of the binary representation of every element in the set. For example, if $u = 16$, so that $\lg u = 4$, and $x = 13$ is in the set, then because the binary representation of 13 is 1101, the perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to the hash table, we create a doubly linked list of the elements currently in the set, in increasing order.

a. How much space does this structure require?

b. Show how to perform the MINIMUM and MAXIMUM operations in $O(1)$ time; the MEMBER, PREDECESSOR, and SUCCESSOR operations in $O(\lg \lg u)$ time; and the INSERT and DELETE operations in $O(\lg u)$ time.

To reduce the space requirement to $O(n)$, we make the following changes to the data structure:

- We cluster the n elements into $n/\lg u$ groups of size $\lg u$. (Assume for now that $\lg u$ divides n.) The first group consists of the $\lg u$ smallest elements in the set, the second group consists of the next $\lg u$ smallest elements, and so on.
- We designate a "representative" value for each group. The representative of the ith group is at least as large as the largest element in the ith group, and it is smaller than every element of the $(i + 1)$st group. (The representative of the last group can be the maximum possible element $u - 1$.) Note that a representative might be a value not currently in the set.
- We store the $\lg u$ elements of each group in a balanced binary search tree, such as a red-black tree. Each representative points to the balanced binary search tree for its group, and each balanced binary search tree points to its group's representative.

The perfect hash table stores only the representatives, which are also stored in a doubly linked list in increasing order.

We call this structure a ***y-fast trie***.

c. Show that a y-fast trie requires only $O(n)$ space to store n elements.

d. Show how to perform the MINIMUM and MAXIMUM operations in $O(\lg \lg u)$ time with a y-fast trie.

e. Show how to perform the MEMBER operation in $O(\lg \lg u)$ time.

f. Show how to perform the PREDECESSOR and SUCCESSOR operations in $O(\lg \lg u)$ time.

g. Explain why the INSERT and DELETE operations take $\Omega(\lg \lg u)$ time.

h. Show how to relax the requirement that each group in a y-fast trie has exactly $\lg u$ elements to allow INSERT and DELETE to run in $O(\lg \lg u)$ amortized time without affecting the asymptotic running times of the other operations.

a. By 11.5, the perfect hash table uses $O(m)$ space to store m elements. In a universe of size u, each element contributes $\lg u$ entries to the hash table, so the requirement is $O(n \lg u)$. Since the linked list requires $O(n)$, the total space requirement is $O(n \lg u)$.

b. MINIMUM and MAXIMUM are easy. We just examine the first and last elements of the associated doubly linked list. MEMBER can actually be performed in $O(1)$, since we are simply checking membership in a perfect hash table. PREDECESSOR and SUCCESSOR are a bit more complicated.

Assume that we have a binary tree in which we store all the elements and their prefixes. When we query the hash table for an element, we get a pointer to that element's location in the binary search tree, if the element is in the tree, and NIL otherwise. Moreover, assume that every leaf node comes with a pointer to its position in the doubly linked list. Let x be the number whose successor we seek. Begin by performing a binary search of the prefixes in the hash table to find the longest hashed prefix y which matches a prefix of x. This takes $O(\lg \lg u)$ since we can check if any prefix is in the hash table in $O(1)$.

Observe that y can have at most one child in the BST, because if it had both children then one of these would share a longer prefix with x. If the left child is missing, have the left child pointer point to the largest labeled leaf node in the BST which is less than y. If the right child is missing, use its pointer to point to the successor of y. If y is a leaf node then $y = x$, so we simply follow the pointer to x in the doubly linked list, in $O(1)$, and its successor is the next element on the list. If y is not a leaf node, we follow its predecessor or successor node, depending on which we need. This gives us $O(1)$ access to the proper element, so the total runtime is $O(\lg \lg u)$. INSERT and DELETE must take $O(\lg u)$ since we need to insert one entry into the hash table for each of their bits and update the pointers.

c. The doubly linked list has less than n elements, while the binary search trees contains n nodes, thus a y-fast trie requires $O(n)$ space.

d. MINIMUM: Find the minimum representative in the doubly linked list in $\Theta(1)$, then find the minimum element in the binary search tree in $O(\lg \lg u)$.

e. Find the smallest representative greater than k with binary searching in $\Theta(\lg \lg u)$, find the element in the binary search tree in $O(\lg \lg u)$.

f. If we can find the largest representative greater than or equal to x, we can determine which binary tree contains the predecessor or successor of x. To do this, just call PREDECESSOR or SUCCESSOR on x to locate the appropriate tree in $O(\lg \lg u)$. Since the tree has height $\lg u$, we can find the predecessor or successor in $O(\lg \lg u)$.

g. Same as **e**, we need to find the cluster in $\Theta(\lg \lg u)$, then the operations in the binary search tree takes $O(\lg \lg u)$.

h. We can relax the requirements and only impose the condition that each group has at least $\frac{1}{2} \lg u$ elements and at most $2 \lg u$ elements.

- If a red-black tree is too big, we split it in half at the median.
- If a red-black tree is too small, we merge it with a neighboring tree.
- If this causes the merged tree to become too large, we split it at the median.
- If a tree splits, we create a new representative.
- If two trees merge, we delete the lost representative.

Any split or merge takes $O(\lg u)$ since we have to insert or delete an element in the data structure storing our representatives, which by part (b) takes $O(\lg u)$.

However, we only split a tree after at least $\lg u$ insertions, since the size of one of the red-black trees needs to increase from $\lg u$ to $2 \lg u$ and we only merge two trees after at least $(1/2) \lg u$ deletions, because the size of the merging tree needs to have decreased from $\lg u$ to $(1/2) \lg u$. Thus, the amortized cost of the merges, splits, and updates to representatives is $O(1)$ per insertion or deletion, so the amortized cost is $O(\lg \lg u)$ as desired.

21 Data Structures for Disjoint Sets

21.1 Disjoint-set operations

21.1-1

Suppose that CONNECTED-COMPONENTS is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ and the edges of E are processed in the order (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e). List the vertices in each connected component after each iteration of lines 3–5.

Edge processed											
initial	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	{k}
(d, i)	{a}	{b}	{c}	{d, i}	{e}	{f}	{g}	{h}		{j}	{k}
(f, k)	{a}	{b}	{c}	{d, i}	{e}	{f, k}	{g}	{h}		{j}	
(g, i)	{a}	{b}	{c}	{d, i, g}	{e}	{f, k}		{h}		{j}	
(b, g)	{a}	{b, d, i, g}	{c}		{e}	{f, k}		{h}		{j}	
(a, h)	{a, h}	{b, d, i, g}	{c}		{e}	{f, k}				{j}	
(i, j)	{a, h}	{b, d, i, g, j}	{c}		{e}	{f, k}					
(d, k)	{a, h}	{b, d, i, g, j, f, k}	{c}		{e}						
(b, j)	{a, h}	{b, d, i, g, j, f, k}	{c}		{e}						
(d, f)	{a, h}	{b, d, i, g, j, f, k}	{c}		{e}						
(g, j)	{a, h}	{b, d, i, g, j, f, k}	{c}		{e}						
(a, e)	{a, h, e}	{b, d, i, g, j, f, k}	{c}								

So, the connected components that we are left with are $\{a, h, e\}$, $\{b, d, i, g, j, f, k\}$, and $\{c\}$.

21.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected component if and only if they are in the same set.

First suppose that two vertices are in the same connected component. Then there exists a path of edges connecting them. If two vertices are connected by a single edge, then they are put into the same set when that edge is processed. At some point during the algorithm every edge of the path will be processed, so all vertices on the path will be in the same set, including the endpoints. Now suppose two vertices u and v wind up in the same set. Since every vertex starts off in its own set, some sequence of edges in G must have resulted in eventually combining the sets containing u and v. From among these, there must be a path of edges from u to v, implying that u and v are in the same connected component.

21.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express

your answers in terms of $|V|$, $|E|$, and k.

Find set is called twice on line 4, this is run once per edge in the graph, so, we have that find set is run $2|E|$ times. Since we start with $|V|$ sets, at the end only have k, and each call to UNION reduces the number of sets by one, we have that we have to made $|V| - k$ calls to UNION.

21.2 Linked-list representation of disjoint sets

21.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

The three algorithms follow the english description and are provided here. There are alternate versions using the weighted union heuristic, suffixed with WU.

```
MAKE-SET(x)
// Assume x is a pointer to a node contains .key .set .next
    Create a node S contains .head .tail .size
    x.set = S
    x.next = NIL
    S.head = x
    S.tail = x
    S.size = 1
    return S
```

```
FIND-SET(x)
    return x.set.head
```

```
UNION(x, y)
    S1 = x.set
    S2 = y.set
    if S1.size >= S2.size
        S1.tail.next = S2.head
        z = S2.head
        while z != NIL // z.next is incorrect, so S2.tail node should not be
updated
            z.set = S1
            z = z.next
        S1.tail = S2.tail
        S1.size = S1.size + S2.size // Update the size of set
        return S1
    else
        same procedure as above
```

```
        change x to y
        change S1 to S2
```

21.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic.

```
for i = 1 to 16
    MAKE-SET(x[i])
for i = 1 to 15 by 2
    UNION(x[i], x[i + 1])
for i = 1 to 13 by 4
   UNION(x[i], x[i + 2])
UNION(x[1], x[5])
UNION(x[11], x[13])
UNION(x[1], x[10])
FIND-SET(x[2])
FIND-SET(x[9])
```

Assume that if the sets containing x_i and x_j have the same size, then the operation $\text{UNION}(x_i, x_j)$ appends x_j's list onto x_i's list.

Originally we have 16 sets, each containing x_i. In the following, we'll replace x_i by i. After the **for** loop in line 3 we have:

$$\{1,2\},\{3,4\},\{5,6\},\{7,8\},\{9,10\},\{11,12\},\{13,14\},\{15,16\}.$$

After the **for** loop on line 5 we have

$$\{1,2,3,4\},\{5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 7 results in:

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 8 results in:

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12,13,14,15,16\}.$$

Line 9 results in:

$$\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\}.$$

$\text{FIND-SET}(x_2)$ and $\text{FIND-SET}(x_9)$ each return pointers to x_1.

```
MAKE-SET-WU(x)
    L = MAKE-SET(x)
    L.size = 1
    return L

UNION-WU(x, y)
    L1 = x.set
    L2 = y.set
    if L1.size ≥ L2.size
        L = UNION(x, y)
    else L = UNION(y, x)
    L.size = L1.size + L2.size
    return L
```

21.2-3

Adapt the aggregate proof of Theorem 21.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

During the proof of theorem 21.1, we concluded that the time for the n UNION operations to run was at most $O(n \lg n)$. This means that each of them took an amortized time of at most $O(\lg n)$. Also, since there is only a constant actual amount of work in performing MAKE-SET and FIND-SET operations, and none of that ease is used to offset costs of UNION operations, they both have $O(1)$ runtime.

21.2-4

Give a tight asymptotic bound on the running time of the sequence of operations in Figure 21.3 assuming the linked-list representation and the weighted-union heuristic.

We call MAKE-SET n times, which contributes $\Theta(n)$. In each union, the smaller set is of size 1, so each of these takes $\Theta(1)$ time. Since we union $n-1$ times, the runtime is $\Theta(n)$.

21.2-5

Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (head and tail), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (Hint: Use the tail of a linked list as its set's representative.)

For each member of the set, we will make its first field which used to point back to the set object point instead to the last element of the linked list. Then, given any set, we can find its last element by going ot the head and following the pointer that that object maintains to the last element of the linked list. This only requires following exactly two pointers, so it takes a constant amount of time. Some care must be taken when unioning these modified sets. Since the set representative is the last element in the set, when we combine two linked lists, we place the smaller of the two sets before the larger, since we need to update their set representative pointers, unlike the original situation, where we update the representative of the objects that are placed on to the end of the linked list.

21.2-6

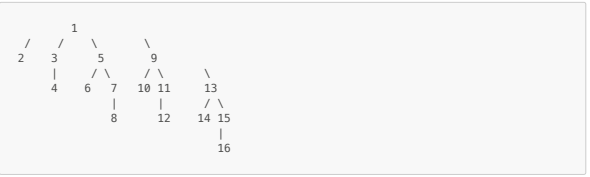
Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the tail pointer to the last object in each list. Whether or not the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (Hint: Rather than appending one list to another, splice them together.)

Instead of appending the second list to the end of the first, we can imagine splicing it into the first list, in between the head and the elements. Store a pointer to the first element in S_1. Then for each element x in S_2, set $x.\text{head} = S_1.\text{head}$. When the last element of S_2 is reached, set its next pointer to the first element of S_1. If we always let S_2 play the role of the smaller set, this works well with the weighted-union heuristic and don't affect the asymptotic running time of UNION.

21.3 Disjoint-set forests

21.3-1

Redo Exercise 21.2-2 using a disjoint-set forest with union by rank and path compression.



21.3-2

Write a nonrecursive version of FIND-SET with path compression.

To implement FIND-SET nonrecursively, let x be the element we call the function on. Create a linked list A which contains a pointer to x. Each time we most one element up the tree, insert a pointer to that element into A. Once the root r has been found, use the linked list to find each node on the path from the root to x and update its parent to r.

21.3-3

Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.

Suppose that $n' = 2k$ is the smallest power of two less than n. To see that this sequences of operations does take the required amount of time, we'll first note that after each iteration of the **for** loop indexed by j, we have that the elements $x_1, \ldots, x_{n'}$ are in trees of depth i. So, after we finish the outer **for** loop, we have that $x_1, \ldots, x_{n'}$ all lie in the same set, but are represented by a tree of depth $k \in \Omega(\lg n)$. Then, since we repeatedly call FIND-SET on an item that is $\lg n$ away from its set representative, we have that each one takes time $\lg n$. So, the last **for** loop alltogther takes time $\Omega(m \lg n)$.

```
for i = 1 to n
    MAKE-SET(x[i])
for i = 1 to k
    for j = 1..n' − 2^{i = 1} by 2^i
        UNION(x[i], x[i + 2^{j − 1}])
for i = 1 to m
    FIND-SET(x[1])
```

21.3-4

Suppose that we wish to add the operation PRINT-SET(x), which is given a node x and prints all the members of x's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that PRINT-SET(x) takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

In addition to each tree, we'll store a linked list (whose set object contains a single tail pointer) with which keeps track of all the names of elements in the tree. The only additional information we'll store in each node is a pointer $x.l$ to that element's position in the list.

- When we call MAKE-SET(x), we'll also create a new linked list, insert the label of x into the list, and set $x.l$ to a pointer to that label. This is all done in $O(1)$.
- FIND-SET will remain unchanged.
- UNION(x, y) will work as usual, with the additional requirement that we union the linked lists of x and y, since we don't need to update pointers to the head, we can link up the lists in constant time, thus preserving the runtime of UNION.
- Finally, PRINT-SET(x) works as follows: first, set $s = \text{FIND-SET}(x)$. Then print the elements in the linked list, starting with the element pointed to by x. (This will be the first element in the list). Since the list contains the same number of elements as the set and printing takes $O(1)$, this operation takes linear time in the number of set members.

21.3-5 ⋆

Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time if we use both path compression and union by rank. What happens in the same situation if we use only the path-compression heuristic?

Clearly each MAKE-SET and LINK operation only takes time $O(1)$, so, supposing that n is the number of FIND-SET operations occuring after the making and linking, we need to show that all the FIND-SET operations only take time $O(n)$.

To do this, we will ammortize some of the cost of the FIND-SET operations into the cost of the MAKE-SET operations. Imagine paying some constant amount extra for each MAKE-SET operation. Then, when doing a FIND-SET(x) operation, we have three possibilities:

- First, we could have that x is the representative of its own set. In this case, it clearly only takes constant time to run.
- Second, we could have that the path from x to its set's representative is already compressed, so it only takes a single step to find the set representative. In this case also, the time required is constant.
- Third, we could have that x is not the representative and it's path has not been compressed. Then, suppose that there are k nodes between x and its representative. The time of this FIND-SET operation is $O(k)$, but it also ends up compressing the paths of k nodes, so we use that extra amount that we paid during the MAKE-SET operations for these k nodes whose paths were compressed. Any subsequent call to find set for these nodes will take only a constant amount of time, so we would never try to use the work that amortization amount twice for a given node.

21.4 Analysis of union by rank with path compression

21.4-1

Prove Lemma 21.4.

The lemma states:

> For all nodes x, we have $x.\text{rank} \le x.p.\text{rank}$, with strict inequality if $x \ne x.p$. The value of $x.\text{rank}$ is initially 0 and increases through time until $x \ne x.p$; from then on, $x.\text{rank}$ does not change. The value of $x.p.\text{rank}$ monotonically increases over time.

The initial value of $x.\text{rank}$ is 0, as it is initialized in line 2 of the MAKE-SET(x) procedure. When we run LINK(x, y), whichever one has the larger rank is placed as the parent of the other, and if there is a tie, the parent's rank is incremented. This means that after any LINK(y, x), the two nodes being linked satisfy this strict inequality of ranks.

Also, if we have that $x \ne x.p$, then, we have that x is not its own set representative, so, any linking together of sets that would occur would not involve x, but that's the only way for ranks to increase, so, we have that $x.\text{rank}$ must remain constant after that point.

21.4-2

Prove that every node has rank at most $\lfloor \lg n \rfloor$.

We'll prove the claim by strong induction on the number of nodes. If $n = 1$, then that node has rank equal to $0 = \lfloor \lg 1 \rfloor$. Now suppose that the claim holds for $1, 2, \ldots, n$ nodes.

Given $n+1$ nodes, suppose we perform a UNION operation on two disjoint sets with a and b nodes respectively, where $a, b \le n$. Then the root of the first set has rank at most $\lfloor \lg a \rfloor$ and the root of the second set has rank at most $\lfloor \lg b \rfloor$.

If the ranks are unequal, then the UNION operation preserves rank and we are done, so suppose the ranks are equal. Then the rank of the union increases by 1, and the resulting set has rank $\lfloor \lg a \rfloor + 1 \le \lfloor \lg(n+1)/2 \rfloor + 1 = \lfloor \lg(n+1) \rfloor$.

21.4-3

In light of Exercise 21.4-2, how many bits are necessary to store $x.\text{rank}$ for each node x?

Since their value is at most $\lfloor \lg n \rfloor$, we can represent them using $\Theta(\lg(\lg(n)))$ bits, and may need to use that many bits to represent a number that can take that many values.

21.4-4

Using Exercise 21.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

MAKE-SET takes constant time and both FIND-SET and UNION are bounded by the largest rank among all the sets. Exercise 21.4-2 bounds this from about by $\lceil \lg n \rceil$, so the actual cost of each operation is $O(\lg n)$. Therefore the actual cost of m operations is $O(m \lg n)$.

21.4-5

Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.\text{rank} > 0$ and $x.p$ is not a root, then $\text{level}(x) \le \text{level}(x.p)$. Is the professor correct?

Professor Dante is not correct.

Suppose that we had that $x.p.\text{rank} > A_2(x.\text{rank})$ but that $x.p.p.\text{rank} = 1 + x.p.\text{rank}$, then we would have that $\text{level}(x.p) = 0$, but $\text{level}(x) \ge 2$. So, we don't have that $\text{level}(x) \le \text{level}(x.p)$ even though we have that the ranks are monotonically increasing as we go up in the tree. Put another way, even though the ranks are monotonically increasing, the rate at which they are increasing (roughly captured by the level values) doesn't have to be increasing.

21.4-6 ⋆

Consider the function $\alpha'(n) = \min\{k : A_k(1) \ge \lg(n+1)\}$. Show that $\alpha'(n) \le 3$ for all practical values of n and, using Exercise 21.4-2, show how to modify the potential-function argument to prove that we can perform a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\alpha'(n))$.

First, observe that by a change of variables, $\alpha'(2^{n-1}) = \alpha(n)$. Earlier in the section we saw that $\alpha(n) \le 3$ for $0 \le n \le 2047$. This means that $\alpha'(n) \le 2$ for $0 \le n \le 2^{2046}$, which is larger than the estimated number of atoms in the observable universe.

To prove the improved bound $O(m\alpha'(n))$ on the operations, the general structure will be essentially the same as that given in the section.

First, modify bound 21.2 by observing that $A_{\alpha'(n)}(x.\text{rank}) \ge A_{\alpha'(n)}(1) \ge \lg(n+1) > x.p.\text{rank}$ which implies $\text{level}(x) \le \alpha'(n)$.

Next, redefine the potential replacing $\alpha(n)$ by $\alpha'(n)$. Lemma 21.8 now goes through just as before. All subsequent lemmas rely on these previous observations, and their proofs go through exactly as in the section, yielding the bound.

Problem 21-1 Off-line minimum

The ***off-line minimum problem*** asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array extracted$[1..m]$, where for $i = 1, 2, \ldots, m$, extracted$[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "off-line" in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

a. In the following instance of the off-line minimum problem, each operation INSERT(i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

$$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.$$

Fill in the correct values in the *extracted* array.

To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by

$$I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1},$$

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j , we initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. We then do the following:

```
OFF-LINE-MINIMUM(m, n)
    for i = 1 to n
        determine j such that i ∈ K[j]
        if j != m + 1
            extracted[j] = i
            let l be the smallest value greater than j for which set K[l]
exists
            K[l] = K[j] ∪ K[l], destroying K[j]
    return extracted
```

b. Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.

c. Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

a.

index	value
1	4
2	3
3	2
4	6
5	8
6	1

b. As we run the **for** loop, we are picking off the smallest of the possible elements to be removed, knowing for sure that it will be removed by the next unused EXTRACT-MIN operation. Then, since that EXTRACT-MIN operation is used up, we can pretend that it no longer exists, and combine the set of things that were inserted by that segment with those inserted by the next, since we know that the EXTRACT-MIN operation that had separated the two is now used up. Since we proceed to figure out what the various extract operations do one at a time, by the time we are done, we have figured them all out.

c. We let each of the sets be represented by a disjoint set structure. To union them (as on line 6) just call UNION. Checking that they exist is just a matter of keeping track of a linked list of which ones exist(needed for line 5), initially containing all of them, but then, when deleting the set on line 6, we delete it from the linked list that we were maintaining. The only other interaction with the sets that we have to worry about is on line 2, which just amounts to a call of FIND-SET(j). Since line 2 takes amortized time $\alpha(n)$ and we call it exactly n times, then, since the rest of the **for** loop only takes constant time, the total runtime is $O(n\alpha(n))$.

Problem 21-2 Depth determination

In the ***depth-determination problem***, we maintain a forest $\mathcal{D} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.

FIND-DEPTH(v) returns the depth of node v within its tree.

GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree than r but may or may not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that we implement GRAFT(r, v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, we can reduce the worst-case running time. We use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{D}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

d. Show how to implement GRAFT(r, v), which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

a. MAKE-TREE and GRAFT are both constant time operations. FINDDEPTH is linear in the depth of the node. In a sequence of m operations the maximal depth which can be achieved is $m/2$, so FIND-DEPTH takes at most $O(m)$. Thus, m operations take at most $O(m^2)$. This is achieved as follows: Create $m/3$ new trees. Graft them together into a chain using $m/3$ calls to GRAFT. Now call FIND-DEPTH on the deepest node $m/3$ times. Each call takes time at least $m/3$, so the total runtime is $\Omega((m/3)^2) = \Omega(m^2)$. Thus the worst-case runtime of the m operations is $\Theta(m^2)$.

b. Since the new set will contain only a single node, its depth must be zero and its parent is itself. In this case, the set and its corresponding tree are indistinguishable.

```
MAKE-TREE(v)
    v = ALLOCATE-NODE()
    v.d = 0
    v.p = v
    return v
```

c. In addition to returning the set object, modify FIND-SET to also return the depth of the parent node. Update the pseudodistance of the current node v to be $v.d$ plus the returned pseudodistance. Since this is done recursively, the running time is unchanged. It is still linear in the length of the find path. To implement FIND-DEPTH, simply recurse up the tree containing v, keeping a running total of pseudodistances.

```
FIND-SET(v)
    if v != v.p
        (v.p, d) = FIND-SET(v.p)
        v.d = v.d + d
        return (v.p, v.d)
    return (v, 0)
```

d. To implement GRAFT we need to find v's actual depth and add it to the pseudodistance of the root of the tree S_i which contains r.

```
GRAFT(r, v)
    (x, d_1) = FIND-SET(r)
    (y, d_2) = FIND-SET(v)
    if x.rank > y.rank
        y.p = x
        x.d = x.d + d_2 + y.d
    else
        x.p = y
        x.d = x.d + d_2
        if x.rank == y.rank
            y.rank = y.rank + 1
```

e. The three implemented operations have the same asymptotic running time as MAKE, FIND, and UNION for disjoint sets, so the worst-case runtime of m such operations, n of which are MAKE-TREE operations, is $O(m\alpha(n))$.

Problem 21-3 Tarjan's off-line least-common-ancestors algorithm

The ***least common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***off-line least-common-ancestors problem***, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and we wish to determine the least common ancestor of each pair in P.

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call LCA$(T.root)$. We assume that each node is colored WHITE prior to the walk.

```
LCA(u)
    MAKE-SET(u)
    FIND-SET(u).ancestor = u
    for each child v of u in T
        LCA(v)
        UNION(u, v)
        FIND-SET(u).ancestor = u
    u.color = BLACK
    for each node v such that {u, v} ∈ P
        if v.color == BLACK
            print "The least common ancestor of" u "and" v "is" FIND-
SET(v).ancestor
```

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

b. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of u in T.

c. Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

d. Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure in Section 21.3.

a. Suppose that we let $\preceq_{LCA}$ to be an ordering on the vertices so that $u \preceq_{LCA} v$ if we run line 7 of LCA(u) before line 7 of LCA(v). Then, when we are running line 7 of LCA(u), we immediately go on to the **for** loop on line 8.

So, while we are doing this **for** loop, we still haven't called line 7 of LCA(v). This means that $v.color$ is white, and so, the pair $\{u, v\}$ is not considered during the run of LCA(u). However, during the **for** loop of LCA(v), since line 7 of LCA(u) has already run, $u.color = black$. This means that we will consider the pair $\{v, u\}$ during the running of LCA(v).

It is not obvious what the ordering $\preceq_{LCA}$ is, as it will be implementation dependent. It depends on the order in which child vertices are iterated in the **for** loop on line 3. That is, it doesn't just depend on the graph structure.

b. We suppose that it is true prior to a given call of LCA, and show that this property is preserved throughout a run of the procedure, increasing the number of disjoint sets by one by the end of the procedure. So, supposing that u has depth d and there are d items in the disjoint set data structure before it runs, it increases to $d + 1$ disjoint sets on line 1. So, by the time we get to line 4, and call LCA of a child of u, there are $d + 1$ disjoint sets, this is exactly the depth of the child. After line 4, there are now $d + 2$ disjoint sets, so, line 5 brings it back down to $d + 1$ disjoint sets for the subsequent times through the loop. After the loop, there are no more changes to the number of disjoint sets, so, the algorithm terminates with $d + 1$ disjoint sets, as desired. Since this holds for any arbitrary run of LCA, it holds for all runs of LCA.

c. Suppose that the pair u and v have the least common ancestor w. Then, when running LCA(w), u will be in the subtree rooted at one of w's children, and v will be in another. WLOG, suppose that the subtree containing u runs first.

So, when we are done with running that subtree, all of their ancestor values will point to w and their colors will be black, and their ancestor values will not change until LCA(w) returns. However, we run LCA(v) before LCA(w) returns, so in the **for** loop on line 8 of LCA(v), we will be considering the pair $\{u, v\}$, since $u.color = black$. Since $u.ancestor$ is still w, that is what will be output, which is the correct answer for their LCA.

d. The time complexity of lines 1 and 2 are just constant. Then, for each child, we have a call to the same procedure, a UNION operation which only takes constant time, and a FIND-SET operation which can take at most amortized inverse Ackerman's time. Since we check each and every thing that is adjacent to u for being black, we are only checking each pair in P at most twice in lines 8-10, among all the runs of LCA. This means that the total runtime is $O(|T|\alpha(|T|) + |P|)$.