Given two rooted trees, with roots r_1 and r_2, respectively, we use the operation $link(r_1, r_2)$ (read as *link* r_1 *into* r_2) to make r_1 a child of r_2. Once this is done, we will be left with a single tree rooted at r_2.

A *binomial tree* is a rooted tree data structure, defined recursively as follows.

- A binomial tree of rank 0, denoted by B_0, is a single vertex.
- A binomial tree of rank k, denoted by B_k, is obtained by *linking* a B_{i-1} into another B_{i-1}.

We can alternatively define a *binomial tree* as follows.

- A binomial tree of rank 0, denoted by B_0, is a single vertex.
- A binomial tree of rank k, denoted by B_k, is obtained by successively linking the binomial trees $B_0, B_1, \ldots, B_{k-1}$ into a single node.

(In what follows, the heaps we talk about are assumed to be min-heaps, although the arguments made can be generalized to max-heaps as well).

A *binomial heap* is a collection of heap-ordered binomial trees. We define two variants of a binomial heap, as detailed below.

1. *Strict variant.* The heap contains at most 1 binomial tree of each rank. We maintain an array of pointers to the roots of each of the binomial trees such that the i^th index of the array points to B_i, if it exists.
2. *Lazy variant.* The heap can contain any number of binomial trees of a particular rank. As the heap can contain arbitrarily many binomial trees and no particular ordering between the trees need be enforced, we maintain the pointers to the roots of each of the binomial trees, in a doubly linked list instead of an array.

In both variants, we also maintain a pointer min to point to the smallest element in the heap.

The following operations are normally supported in a binomial heap.

1. $makeHeap(x)$ - Make a new heap that contains only the element x.
2. $merge(H_1, H_2)$ - Merge the elements of both the heaps h_1, h_2 into a single heap.
3. $insert(x, H)$ - Insert element x into heap H.
4. $findMin(H)$ - Return the element pointed to by the $min(H)$ pointer.
5. $deleteMin(H)$ - Delete the node pointed to by $min(H)$.
6. $decreaseKey(p, k, H)$ - Decrease by k the value of the node p points to.
7. $delete(p, H)$ - Delete the node pointed to by the pointer p.

If an operation introduces a new minimum element or violates the heap-ordering of the trees, the heap must be reorganized appropriately.

1. Prove that both the definitions for a binomial tree are equivalent; that is, using the first definition, we can prove the second definition, and vice versa.

2. Suppose that T is a binomial tree of rank k. Show that
   (a) the height of T is exactly k.
   (b) the tree has exactly 2^k nodes.
   (c) the root of the tree has exactly k children.
   (d) the degrees of nodes along a rooted path are in strictly decreasing order. That is, if u, v are nodes at levels i, j, respectively, and $0 \leq i < j$, then $\deg(u) > \deg(v)$.

3. In a binomial tree of rank k, with root at a node r, let I be the set of nodes at level i (assume that the root r is at level 0). Show that
   (a) there exist exactly $|I|$ rooted paths that end in some vertex $v \in I$.
   (b) no two rooted paths to vertices $u, v \in I$ have the same degree sequence.
   (c) for each $|I|$ element subset S of $[0, k-1]$, there exists a vertex $v \in I$, such that the rooted path to v has the degree sequence given by $descendingSort(S \cup \{k\})$.
   (d) the tree has exactly $\binom{k}{i}$ nodes at level i.

4. For some non-root vertex v in a binomial tree B_r, the $cut(v)$ operation *cuts the link* between v and its parent, resulting in two trees, respectively, the subtree T_v rooted at v and the tree $T = B_r \setminus V(T_v)$. Clearly, T_v is a binomial tree, whereas T may not be.

   Now, suppose that, after cutting out T_v, our objective is to execute a sequence of t cuts such that at the end of the process, every tree in our collection of trees is a binomial tree. Give the best bound you can on how small t can be in the worst case and justify your answer.

   For questions 5-10, assume that we are dealing with the strict variant of binomial heaps.

5. Suppose that a binomial heap H has n elements. Show that
   (a) none of the binomial trees present in H has a height greater than $\lfloor \log n \rfloor$.
   (b) H contains at most $\lfloor \log n \rfloor + 1$ binomial trees.
   (c) H contains a tree of rank i if and only if the i^{th} least significant bit in the binary representation of n is 1.
   (d) the minimum element in H is present in the root of one of its binomial trees.

6. Write algorithms to implement each of the following operations and analyze their worst case complexities, where H, H_1, H_2 are binomial heaps.
   1. $merge(H_1, H_2)$
   2. $insert(x, H)$
   3. $findMin(H)$
   4. $deleteMin(H)$

7. Suppose that we do not need the $findMin$ operation in our algorithm. Do we still need to maintain the min pointer?

8. Show that the $decreaseKey$ operation can be implemented such that it takes $\Theta(\log n)$ time in the worst case. Implement $delete$ by using $decreaseKey$ and $deleteMin$ as subtasks.

   Hint: Try to adapt the $heapify$ operation of *binary heaps*.

9. Show that the $insert$ operation has an amortized cost of $\mathcal{O}(1)$.

10. Take an arbitrary list of 12 elements and perform *Heap Sort* on it using a binomial heap. Also, draw the structure of the heap after each $deleteMin$ operation.

    For questions 11-17, assume that we are dealing with the lazy variant of binomial heaps.

11. Suppose that a binomial heap H has n elements. Show that none of the binomial trees present in H has a rank greater than $\lfloor \log n \rfloor$.

12. Write an algorithm to implement $merge$ to have a worst case complexity of $\Theta(1)$. Note that the with this version of $merge$, we can implement $insert$ to take only $\Theta(1)$ time even in the worst case, which is *not* possible in the strict variant.

13. Consider the algorithms for the $decreaseKey$, $delete$ and $deleteMin$ operations in questions 6 and 8. Adapt these to work even in the lazy variant of binomial heaps (hopefully, minor modifications, if any, should suffice). Compute the worst case complexities for each of these operations.

    (Assuming that $delete$ is not used to delete the minimum element, their respective worst case complexities should ideally be $\Theta(\log n)$, $\Theta(\log n)$ and $\Theta(n)$.)

14. Give complete details of the crediting scheme used to show that the amortized cost of $deleteMin$ is $\Theta(\log n)$. Moreover, show for each operation that the actual cost of the operation is upper bounded by the sum of its amortized complexity and the stored credits that get used.

15. Read the chapter on binomial heaps in Kozen[1].

    An alternate proof to show that $deleteMin$ has an amortized cost of $\Theta(\log n)$ is given in the section *Lazy Melds*. Unfortunately, while the argument is overall correct, they make an assumption that violates the credit invariant.

    (a) Specify what this erroneous assumption is.

    (b) Make changes to the crediting scheme so that their argument works even without making the said assumption.

16. Recall that, during the $deleteMin$ operation, after linking a root r_1 into another root r_2, we also moved the new tree (rooted at r_2) to the next index in the underlying array of pointers. Explain clearly, and concisely, why

    (a) this needs to be done.

    (b) we need to pay for this using the credit that was stored in r_1.

17. Repeat question 10 for the lazy variant.

18. Think about where the following facts were crucial in the analysis of the various aspects of binomial trees.

    (a) The two definitions of binomial trees.

    (b) Height of a binomial tree is equal to its rank.

    (c) Degree of the root of a binomial tree is equal to its rank.

19. Suppose that you are given a vertex-weighed graph on n vertices, labeled from 1 to n, using its adjacency list representation. Suppose also that you need to implement an algorithm that executes operations that either delete a vertex with smallest weight or decreases a vertex's weight. To this end, you decide to store the vertices in a binomial heap (lazy variant), with the vertex weights used as keys.

    (a) What pointers do you need to associate with the various data objects to implement this algorithm in C? Ensure that the actual running time of your implementation is $k \log n$, where k is the total number of $deleteMin$ and $decreaseKey$ operations executed.

    (b) Give an implementation for $decreaseKey$, ensuring that your pointers are consistently maintained.

References

[1] Dexter C. Kozen. *Design and Analysis of Algorithms*. Texts and Monographs in Computer Science. Springer, 1992.