Practice problems: Augmented Red-Black Trees

1. (CLRS 13.1-6) What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

2. (CLRS 13-2) The **join** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $key[x_1] \leq key[x] \leq key[x_2]$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

   (a) Given a red-black tree T, we store its black-height as the field $bh[T]$. Argue that this field can be maintained by RB-Insert and RB-Delete without requiring extra storage in the tree and without increasing the asymptotic running times. Show while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

   We wish to implement the operation RB-Join(T_1, x, T_2) which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

   (b) Assume without loss of generality that $bh[T_1] \geq bh[T_2]$. Describe an $O(\lg n)$ time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $bh[T_2]$.

   (c) Let T_y be the subtree rooted at y. Describe how T_y can be replaced by $T_y \cup \{x\} \cup T_2$ in $O(1)$ time without destroying the binary-search-tree property.

   (d) What color should we make x so that red-black properties $1, 2$, and 4 are maintained? Describe how property 3 can be enforced in $O(\lg n)$ time.

   (e) Argue that the running time of RB-Join is $O(\lg n)$

3. In this problem we consider a data structure for maintaining a multi-set M. We want to support the following operations:

   - $Init(M)$: create an empty data structure M.
   - $Insert(M, i)$: insert (one copy of) i in M.
   - $Remove(M, i)$: remove (one copy of) i from M.
   - $Frequency(M, i)$: return the number of copies of i in M.
   - $Select(M, k)$: return the k'th element in the sorted order of elements in M.

   If for example M consists of the elements

   $$< 0, 3, 3, 4, 4, 7, 8, 8, 8, 9, 11, 11, 11, 11, 13 >$$

then $Frequency(M, 4)$ will return 2 and $Select(M, 6)$ will return 7.

Let $|M|$ and $\|M\|$ denote the number of elements and the number of *different* elements in M, respectively.

a) Describe an implementation of the data structure such that $Init(M)$ takes $O(1)$ time and all other operations take $O(\log \|M\|)$ time.

b) Design an algorithm for sorting a list L in $O(|L| \log \|L\|)$ time using this data structure.

4. **(Duke final spring 2001)** We want to maintain a data structure $\mathcal{D}$ representing an infinite array of integers under the following operations:

   - INIT($\mathcal{D}$): Create a data structure for an infinite array with all entries being zero.
   - LOOKUP($\mathcal{D}$, x): Return the value of integer with index x.
   - UPDATE($\mathcal{D}$, x, k): Change the value of integer with index x to k.
   - MAX($\mathcal{D}$): Return the maximal index for which the corresponding integer is non-zero.
   - SUM($\mathcal{D}$): Return the sum of all integers in the array.

   Describe an implementation of $\mathcal{D}$ such that INIT, MAX, and SUM runs in $O(1)$ time and LOOKUP and UPDATE in $O(\log n)$ time, where n is the number of non-zero integers in the list.

5. **(Duke final spring 2002)** The *mean* M of a set of k integers $\{x_1, x_2, \ldots x_k\}$ is defined as

$$M = \frac{1}{k} \sum_{i=1}^{k} x_i.$$

   We want to maintain a data structure $\mathcal{D}$ on a set of integers under the normal INIT, INSERT, DELETE, and FIND operations, as well as a MEAN operation, defined as follows:

   - INIT($\mathcal{D}$): Create an empty structure $\mathcal{D}$.
   - INSERT($\mathcal{D}$,x): Insert x in $\mathcal{D}$.
   - DELETE($\mathcal{D}$,x): Delete x from $\mathcal{D}$.
   - FIND($\mathcal{D}$,x): Return pointer to x in $\mathcal{D}$.
   - MEAN($\mathcal{D}$,a, b): Return the mean of the set consisting of elements x in $\mathcal{D}$ with $a \leq x \leq b$.

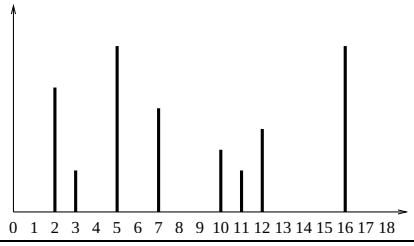
   (a) What does MEAN($\mathcal{D}$,7, 17) return if $\mathcal{D}$ contains integers

$$(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 27)?$$

   (b) Describe how to modify a standard red-black tree in order to implement $\mathcal{D}$ such that INIT is supported in $O(1)$ time and INSERT, DELETE, FIND, and MEAN are supported in $O(\log n)$ time.

6. **(Duke midterm spring 2001)** In this problem we consider a data structure $\mathcal{D}$ for maintaining *bar-diagrams*, where we have a *bar* of height $h \geq 0$ associated with every non-negative integer.

The following is an example of a bar-diagram:



We want to support the following operations:

- INIT($\mathcal{D}$): Create a bar-diagram where all bars have height 0.
- CHANGE($\mathcal{D}$,b,k): Add k to bar b (k can be negative but a bar cannot have negative height).
- HEIGHT($\mathcal{D}$,b): Return the height of bar b.
- TOTALSUM($\mathcal{D}$): Return the sum of the height of all bars.

(a) Describe an implementation of $\mathcal{D}$ such that INIT and TOTALSUM run in $O(1)$ time and CHANGE and HEIGHT run in $O(\log n)$ time. Here n is the number of non-zero bars in the diagram.

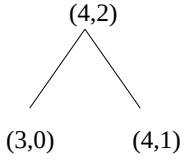
(b) Consider the problem of extending the data structure with operations PREFIXSUM and INTERVALSUM as follows:

- PREFIXSUM($\mathcal{D}$,b_2): Return the sum of the height of all bars b_2 with $b \leq b_2$.
- INTERVALSUM($\mathcal{D}$,b_1,b_2): Return the sum of the height of all bars b such that $b_1 < b \leq b_2$.

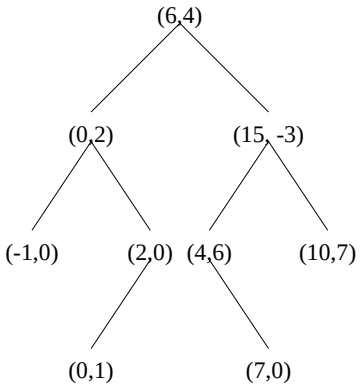
Note that INTERVALSUM can easily be implemented using two calls to PREFIXSUM. Describe an implementation of $\mathcal{D}$ such that INIT and TOTALSUM run in $O(1)$ time and all other operations—including PREFIXSUM and INTERVALSUM—run in $O(\log n)$ time.

7. Consider a binary search tree in which each node v contains a key as well as an additional value called *addend*. The addend of node v is implicitly added to all keys in the subtree rooted at v (including v). Let (*key, addend*) denote the contents of any node v.

For example, the following tree contains the elements $\{5, 6, 7\}$:



(a) Which elements does the following tree contain?



(b) Let h be the height of a tree as defined above. Describe how to perform the following operations in $O(h)$ time:

- FIND(x, T): return YES if element x is stored in tree T
- INSERT(x, T): inserts element x in tree T
- PUSH(x, k, T): add k to all elements $\geq x$

(c) Describe how it can be insured that $h = O(\log n)$ during the above operations. (*Hint:* show how to perform rotation)

8. Consider two *balanced* search trees T_1 and T_2 representing two sets of integers S_1 and S_2 with n_1 and n_2 elements, respectively. Consider the problem of determining if $S_1 \subseteq S_2$ using T_1 and T_2.

(a) Describe an $O(n_1 \log n_2)$ time algorithm for determining if $S_1 \subseteq S_2$ using $O(1)$ extra space.

(b) Describe an $O(n_1+n_2)$ time algorithm for determining if $S_1 \subseteq S_2$ using $O(n_1+n_2)$ extra space.

(c) Describe an $O(n_1+n_2)$ time algorithm for determining if $S_1 \subseteq S_2$ using $O(\log n_1+\log n_2)$ extra space.

9. In this problem we consider data structures for maintaining a matrix M. We want to support the following operations:

- $Init(M)$: create an empty matrix (zeros on all positions)
- $Lookup(M, i, j)$: return the value at index (i, j) $(i, j \geq 0)$
- $Update(M, i, j, e)$: change the value at index (i, j) to e $(i, j \geq 0)$
- $Transpose(M)$: transpose the matrix (that is, element at index (i, j) becomes element at index (j, i))
- $Add(M)$: return the sum of the elements in M

Assume first that we are working on a $n \times n$ matrix.

(a) Describe a data structure such that *Init* runs in $O(n^2)$ time, and all other operations run in $O(1)$ time. (*Hint:* Does anything really need to be done when performing a transpose?)

Assume now that the matrix is of arbitrary size.

(b) Describe a data structure such that *Init* runs in $O(1)$ time, *Lookup* and *Update* in $O(\lg k)$ time, and *Transpose* and *Add* in $O(1)$ time, where k is the number of non-zero entries in the matrix. (*Hint:* Maintain all non-zero elements in a row in a red-black tree)

10. **(Duke final spring 2000)** In this problem we consider divide-and-conquer algorithms for building a heap H on n elements given in an array A. Recall that a heap is an (almost) perfectly balanced binary tree where key(v) $\geq$ key(parent(v)) for all nodes v. We assume $n = 2^h - 1$ for some constant h, such that H is perfectly balanced (leaf level is "full").

First consider the following algorithm SLOWHEAP$(1, n)$ which constructs (a pointer to) H by finding the minimal element x in A, making x the root in H, and recursively constructing the two sub-heaps below x (each of size approximately $\frac{n-1}{2}$).

SLOWHEAP(i, j)

If $i = j$ then return pointer to heap consisting of node containing $A[i]$
Find $i \leq l \leq j$ such that $x = A[l]$ is the minimum element in $A[i \ldots j]$
Exchange $A[l]$ and $A[j]$
Ptr_{left} = SLOWHEAP$(i, \lfloor \frac{i+j-1}{2} \rfloor)$
Ptr_{right} = SLOWHEAP$(\lfloor \frac{i+j-1}{2} \rfloor + 1, j - 1)$
Return pointer to heap consisting of root r containing x with child pointers Ptr_{left} and Ptr_{right}

End

a) Define and solve a recurrence equation for the running time of SLOWHEAP.

Recall that given a tree H where the heap condition is satisfied except possibly at the root r (that is, $key[r] \geq key[\text{leftchild}(r)]$ and/or $key[r] \geq key[\text{rightchild}(r)]$ and $key[v] \geq key[\text{parent}(v)]$ for all nodes $v \neq r$), we can make H into a heap by performing a DOWN-HEAPIFY operation on the root r (DOWN-HEAPIFY on node v swaps element in v with element in one of the children of v and continues down the tree until a leaf is reached or heap order is reestablished).

Consider the following algorithm FASTHEAP$(1, n)$ which constructs (a pointer to) H by placing an arbitrary element x from A (the last one) in the root of H, recursively constructing the two sub-heaps below x, and finally performing a DOWN-HEAPIFY operation on x to make H a heap.

FastHeap(i, j)

Ptr_{left} = FastHeap$(i, \lfloor \frac{i+j-1}{2} \rfloor)$

Ptr_{right} = FastHeap$(\lfloor \frac{i+j-1}{2} \rfloor + 1, j-1)$

Let Ptr be pointer to tree consisting of root r containing $x = A[j]$ with child pointers Ptr_{left} and Ptr_{right}

Perform Down-Heapify on Ptr

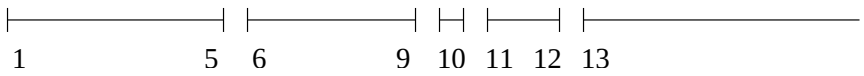
Return Ptr

End

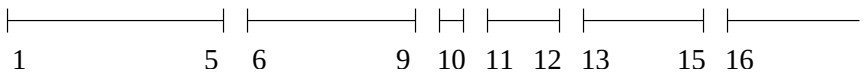
b) Define and solve a recurrence equation for the running time of Fastheap.

11. **(Duke final spring 2000)** In this problem we consider the so-called Interval-Union-Split-Find problem. In this problem, we want to maintain a set of disjoint intervals covering all the natural numbers (one interval is considered to be infinite) such that we can support the operations Union, Split, and Find. Union combines two consecutive intervals, Split splits an interval in two, and Find returns a unique representative for a given interval.

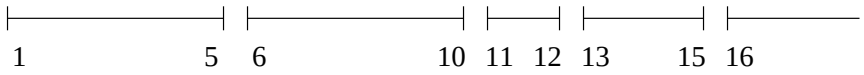
Example: Consider the following intervals:



Performing a Split on the last interval at 16 results in the following set of intervals:



if we then perform a Union on the intervals $[6, 9]$ and the interval after that ($[10, 10]$) we obtain the following set of intervals:



and if we then perform a Find on element 14 we get the unique representative for the interval $[13, 15]$

It is natural to chose the first element in an interval as the unique representative. (In the last figure of the above example the intervals would then be represented by the elements $(1, 6, 11, 13, 16)$ and the Find on element 14 would return 13). The Interval-Union-Split-Find can now be formalized as maintaining a data structure $\mathcal{D}$ under the following operations:

- Init($\mathcal{D}$): Create an interval-union-split-find structure containing the interval $[1, \infty]$.
- Find($\mathcal{D}$,x): Return the representative (the first element) in the interval containing x.

- UNION($\mathcal{D}$,x): Union the interval containing x with the interval after that. No UNION is performed if x is in the last interval.
- SPLIT($\mathcal{D}$,x): Split the interval $[a, \ldots, b]$ containing x into two intervals $[a, \ldots x-1]$ and $[x, \ldots, b]$. No SPLIT is performed if x is the representative of $[a, \ldots, b]$, that is, if $x = a$.

Describe an implementation of $\mathcal{D}$ such that INIT runs in $O(1)$ time and such that all other operations run in $O(\log n)$ time, where n is the number of intervals in the structure. (*Hint:* What happens to the list of n unique representatives when a UNION or SPLIT is performed?)