Full Name: _____

Andrew ID: _____ **Section:** _____

15–210: Parallel and Sequential Data Structures and Algorithms

Practice Final (Solutions)

May 2016

- **Verify:** There are 22 pages in this examination, comprising 8 questions worth a total of 152 points. The last 2 pages are an appendix with costs of sequence, set and table operations.
- **Time:** You have 180 minutes to complete this examination.
- **Goes without saying:** Please answer all questions in the space provided with the question. Clearly indicate your answers.
- **Beware:** You may refer to your *two¡* double-sided $8\frac{1}{2} \times 11$in sheet of paper with notes, but to no other person or source, during the examination.
- **Primitives:** In your algorithms you can use any of the primitives that we have covered in the lecture. A reasonably comprehensive list is provided at the end.
- **Code:** When writing your algorithms, you can use ML syntax but you don't have to. You can use the pseudocode notation used in the notes or in class. For example you can use the syntax that you have learned in class. In fact, in the questions, we use the pseudo rather than the ML notation.

Sections		
A	9:30am - 10:20am	Edward/Angie
B	10:30am - 11:20am	Jake/Narain
C	12:30pm - 1:20pm	Sonya/Anisha
D	12:30pm - 1:20pm	Nick/Yongshan
E	1:30pm - 2:20pm	William/Bryan
F	1:30pm - 2:20pm	Sam S./Yutong
G	3:30pm - 4:20pm	Howard/Yongshan

Full Name: _____ **Andrew ID:** _____

Question	Points	Score
Binary Answers	30	
Costs	12	
Short Answers	26	
Slightly Longer Answers	20	
Neighborhoods	20	
Median ADT	12	
Geometric Coverage	12	
Swap with Compare-and-Swap	20	
Total:	152	

Question 1: Binary Answers (30 points)

(a) (2 points) **TRUE** or **FALSE**: The expressions (`Seq.reduce f I A`) and (`Seq.iterate f I A`) always return the same result as long as `f` is commutative.

Solution: FALSE

(b) (2 points) **TRUE** or **FALSE**: The expressions (`Seq.reduce f I A`) and (`Seq.reduce f I (Seq.reverse A)`) always return the same result if `f` is associative and commutative.

Solution: TRUE

(c) (2 points) **TRUE** or **FALSE**: If a randomized algorithm has expected $O(n)$ work, then there exists some constant c such that the work performed is guaranteed to be at most cn.

Solution: FALSE

(d) (2 points) **TRUE** or **FALSE**: Solving recurrences with induction can be used to show both upper and lower bounds?

Solution: TRUE

(e) (2 points) **TRUE** or **FALSE**: Let p be an odd prime. In open address hashing with a table of size p and given a hash function $h(k)$, quadratic probing uses $h(k, i) = (h(k) + i^2)$ mod p as the ith probe position for key k. If there is an empty spot in the table quadratic hashing will always find it.

Solution: FALSE

(f) (2 points) **TRUE** or **FALSE**: Bottom-Up Dynamic Programming can be parallel, whereas the Top-Down version as described in class (ie, purely functional) is always sequential.

Solution: TRUE

(g) (2 points) **TRUE** or **FALSE**: The height of any treap is $O(\log n)$.

Solution: FALSE

(h) (2 points) **TRUE** or **FALSE**: It is possible to write insert for treaps that uses the split operation but not the join operation.

Solution: TRUE

(i) (2 points) **TRUE** or **FALSE**: Dijkstra's algorithm always terminates even if the input graph contains negative edge weights.

Solution: TRUE

(j) (2 points) **TRUE** or **FALSE**: A $\Theta(n^2)$-work algorithm always takes longer to run than a $\Theta(n \log n)$-work algorithm.

Solution: FALSE

(k) (2 points) **TRUE** or **FALSE**: We can improve the work efficiency of a parallel algorithm by using granularity control.

Solution: TRUE

(l) (2 points) **TRUE** or **FALSE**: We can measure the work efficiency of a parallel algorithm by measuring the running time (work) of the algorithm on a single core, divided by the running time (work) of the sequential elision of the algorithm.

Solution: FALSE

(m) (2 points) **TRUE** or **FALSE**: Some atomic read-modify-write operations such as compare-and-swap suffer from the ABA problem.

Solution: TRUE

(n) (2 points) **TRUE** or **FALSE**: Race conditions are just when two concurrent threads write to the same location.

Solution: FALSE

(o) (2 points) **TRUE** or **FALSE**: In a greedy scheduler a processor cannot sit idle if there is work to do.

Solution: TRUE

Question 2: Costs (12 points)

(a) (6 points) Give tight assymptotic bounds (Θ) for the following recurrence using the tree method. Show your work.

$$W(n) = 2W(n/2) + n \log n$$

Solution: At i^{th} level there are 2^i subproblems each of which cost is $\frac{n}{2^i} \log \frac{n}{2^i}$ for total cost of $n(\log n - i)$.

$$\begin{aligned} W(n) &= \sum_{i=0}^{\log n - 1} n(\log n - i) \\ &= n \sum_{j=1}^{\log n} j \\ &= n \log n(\log n + 1)/2 \\ W(n) &\in \Theta(n \log^2 n) \end{aligned}$$

(b) (6 points) Check the appropriate column for each row in the following table:

	root dominated	leaf dominated	balanced
$W(n) = 2W(n/2) + n^{1.5}$			
$W(n) = \sqrt{n}W(\sqrt{n}) + \sqrt{n}$			
$W(n) = 8W(n/2) + n^2$			

Solution:

	root dominated	leaf dominated	balanced
$W(n) = 2W(n/2) + n^{1.5}$	X		
$W(n) = \sqrt{n}W(\sqrt{n}) + \sqrt{n}$		X	
$W(n) = 8W(n/2) + n^2$		X	

Question 3: Short Answers (26 points)

Answer each of the following questions in the spaces provided.

(a) (3 points) What simple formula defines the parallelism of an algorithm (in terms of work and span)?

Solution: $P(n) = \frac{W(n)}{S(n)}$

(b) (3 points) Name two algorithms we covered in this course that use the greedy method.

Solution: Dijkstra's, Prim's, Kruskal's ...

(c) (3 points) Given a sequence of key-value pairs A, what does the following code do?

```
Table.map Seq.length (Table.collect A)
```

Solution: Makes a histogram of A mapping each key to how many times it appears (the values are ignored).

(d) (5 points) Consider an undirected graph G with unique positive weights. Suppose it has a minimum spanning tree T. If we square all the edge weights and compute the MST again, do we still get the same tree structure? Explain briefly.

Solution: Yes we get the same tree. The minimum spanning tree only depends on the ordering among the edges. This is because the only thing we do with edges is compare them.

(e) (3 points) What asymptotically efficient parallel algorithm/technique can one use to count the number of trees in a forest (tree and forest have their graph-theoretical meaning)? *(Hint: the ancient saying of "can't see forest from the trees" may or may not be of help.)* Give the work and span for your proposed algorithm.

Solution: Run tree contraction over the entire forrest to contract each tree into a single vertex. (You can use either star contract or rake and compress.) Count the number of vertices at the end.

$$W(n) = O(n) \quad S(n) = O(\log^2 n)$$

expected case.

(f) (3 points) What are the two ordering invariants of a Treap? (Describe them briefly.)

Solution: Heap property: Each node has a higher priority than all of its descendants.
BST property: Each node's key is greater than the keys in its left subtree and less than the keys in its right subtree.

(g) (6 points) Is it the case that in a leftist heap the left subtree of a node is always larger than the right subtree. If so, argue why (briefly). If not, give an example.

Solution: False, as shown by the following example of a shape of a leftist heap:

Question 4: Slightly Longer Answers (20 points)

(a) (6 points) Certain locations on a straight pathway recently built for robotics research have to be covered with a special surface, so CMU hires a contractor who can build arbitrary length segments to cover these locations (a location is covered if there is a segment covering it). The segment between a and b (inclusive) costs $(b-a)^2 + k$, where k is a non-negative constant. Let $k \geq 0$ and $X = \langle x_0, \ldots, x_{n-1} \rangle$, $x_i \in \mathbb{R}_+$, be a sequence of locations that have to be covered. Give an $O(n^2)$-work dynamic programming solution to find the cheapest cost of covering these points (all given locations must be covered). Be sure to specify a recursive solution, identify sharing, and describe the **work** and **span** in terms of the DAG.

Solution:

function CCC(X) =
let
  % *The cheapest cover cost for* $X[0, \ldots, i]$
  function $f(i)$ =
    if $(i < 0)$ **then** 0
    else $\min_{0 \leq j \leq i} \left(f(j-1) + k + (x_i - x_j)^2 \right)$
in $f(|X| - 1)$ **end**

Sharing:
There are at most $|X| = n$ distinct subcalls to f since i ranges from 0 to $n-1$.

DAG and costs:
Each node in the DAG does work $O(n)$ and has span $O(\log n)$. The DAG has depth and size $O(n)$. Therefore the total work is $O(n^2)$ and the total span is $O(n \log n)$.

(b) (7 points) Here is a slightly modified version of the algorithm given in class for finding the optimal binary search tree (OBST):

function OBST (A) =
let
  function OBST' (S, d) =
    if $|S| = 0$ **then** 0
    else $min_{i \in \langle 1, \ldots, |S| \rangle}$(OBST'$(S_{1,i-1}, d+1) + d \times p(S_i) +$ OBST'$(S_{i+1,|S|}, d+1)$)
in
  OBST'$(A, 1)$
end

Recall that $S_{i,j}$ is the subsequence $\langle S_i, S_{i+1}, \ldots, S_j \rangle$ of S. For $|A| = n$, place an asymptotic upper bound on the number of distinct arguments OBST′ will have (a tighter bound will get more credit).

Solution: There are $\binom{n+1}{2} = n(n+1)/2$ possible subsequences of S and d is between 1 and n, so the number of distinct arguments is upper-bounded by $O(n^3)$.

(c) (7 points) Given n line segments in 2 dimensions, the 3-intersection problem is to determine if any three of them intersect at the same point. Explain how to do this in $O(n^2)$

work and $O(\log^2 n)$ span. You can assume the lines are given with integer endpoints (i.e. you can do exact arithmetic and not worry about roundoff errors).

Solution: First, we compute all possible intersection points between pairs of line segments. There can be at most $O(n^2)$ points. Then, insert these points into a hash table, checking if any collision seen is a result of 3 lines intersecting at the same point. This meets the time bound since hashing $O(n^2)$ points takes $O(n^2)$ work and $O(\log^2 n^2) \subseteq O(\log^2 n)$ span.

Question 5: Neighborhoods (20 points)

Suppose that you are given a weighted, directed graph G representing the road network in a city. Your mission is to develop a "walking paths" algorithm that may not always return the shortest paths but will return a path between two points of interest that is enjoyable to walk. To this end, suppose that the graph G is labeled with its neighborhood. For example, a vertex representing the Gates building may have an "oakland" label.

In G, a *walking path* from a source in a neighborhood to another vertex in the same neighborhood is defined as the shortest path that never leaves that neighborhood—all the vertices on the shortest path are in the neighborhood.

Throughout assume that G contain no negative edges. Use n for the number of vertices in the graph and m for the number of edges.

(a) (5 points) Describe how to modify Dijkstra's algorithm so that it calculates in H walking paths from a source to all the other vertices in the same neighborhood.

Solution: Dijkstra as usual but we will insert into the priority queue only the vertices that have the neighborhood as the source. This way, we never step out of the neighborhood.

(b) (5 points) What is the work and span of your algorithm? Give a tight bound. Define any extra variables that you may use, if any.

$Work =$ _____

$Span =$ _____

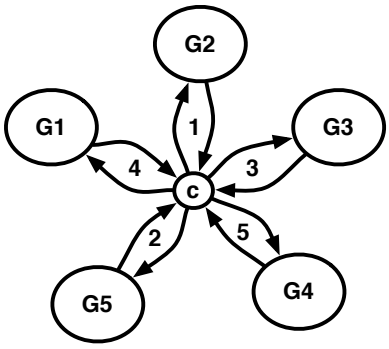
Solution: We only need to consider the vertices in the same neighborhood of the source and the edges outgoing from these vertices, including the boundary vertices that cross out to other neighborhoods.

Let n' denote the number of vertices in the neighborhood and m' denote the total number of out-edges originating from them.

$Work =$ $\underline{O(m' \log n')}$

$Span =$ $\underline{O(m' \log n')}$

(c) For this part, assume that you live in a city that is planned to be walkable. Specifically, the city consists of a single center vertex c with k outgoing edges/streets each of which connects with one of k neighborhoods with $n_1 \dots n_k$ vertices and $m_1 \dots m_k$ edges respectively. Furthermore, you can walk on a street in either direction, i.e., each edge has a corresponding reverse edge with the same weight. The graph below illustrates an example with $k = 5$, where $G_1 \dots G_k$ represents the neighborhoods.



Give a parallel algorithm for the SSSP (single-source shortest paths) problem that given a source s finds the shortest paths to all vertices in the graph. Your algorithm should take advantage of the special topology of your city.

You are not allowed to use Bellman-Ford because it will likely perform too much work and it still has a relatively large span.

i. (5 points) Describe your algorithm. Let G_s denote the neighborhood for the source s.

Solution: Add the center to each neighborhood.
In parallel do: run the modified Dijkstra's algorithm from the previous part on each neighborhood with c as the source. This establishes the shortest paths between the center and any two vertices. Store this information in a new graph that has only a single edge from the center to each vertex in the neighborhood. Also run another local instance of the algorithm with source s for the neighborhood of s.
Now calculate shortest paths as the distance to the center and the distance from the center to each destination vertex for all neighborhoods except G_s. For G_s return the shortest paths from the local run.

ii. (5 points) What is the work and span of your algorithm

$Work =$ _____

$Span =$ _____

Solution:

$$Work = \underline{O(\log n \cdot \sum_{i=1...k} m_i)}$$

$$Span = \underline{O(\log n \cdot \max_{i=1...k} m_i)}$$

A more precise answer use $O(\log n_i)$.

Question 6: Median ADT (12 points)

The *median* of a set C, denoted by `median`(C), is the value of the $\lceil n/2 \rceil$-th smallest element (counting from 1). For example,

$$\texttt{median}(\{1,3,5,7\}) = 3$$
$$\texttt{median}(\{4,2,9\}) = 4$$

In this problem, you will implement an abstract data type `medianT` that maintains a collection of integers (possibly with duplicates) and supports the following operations:

`insert`(C, v)	:	`medianT` $\times$ `int` $\rightarrow$ `medianT`	add the integer v to C.
`median`(C)	:	`medianT` $\rightarrow$ `int`	return the median value of C.
`fromSeq`(S)	:	`int Seq.t` $\rightarrow$ `medianT`	create a `medianT` from S.

Throughout this problem, let n denote the size of the collection at the time, i.e., $n = |C|$.

(a) (5 points) Describe how you would implement the `medianT` ADT using (balanced) binary search trees so that `insert` and `median` take $O(\log n)$ work and span.

> **Solution:** As described in the augmented tree lecture, we keep a balanced BST where each node is augmented with the size of the subtree, so that the $(n/2)$-th element can be found in $O(\log n)$; inserting an element also takes $O(\log n)$ because we simply need to "update" the size information on a revelant path, which has length $O(\log n)$.

(b) (7 points) Using some other data structure, describe how to improve the work to $O(\log n)$, $O(1)$ and $O(|S|)$ for the three operations respectively. The `fromSeq S` function needs to run in $O(\log^2 |S|)$ expected span and the work can be expected case. *(Hint: think about maintaining the median, the elements less than the median, and the elements greater than the median separately.)*

> **Solution:** Keep a max heap of values smaller than the median, the current median, and a min heap of values bigger than the median. When inserting, put the new value in the correct heap, rebalancing as necessary. The function `fromSeq` can be easily supported since using quick select to the initial median only requires $O(n)$ expected work and $O(\log^2 n)$ span. The build heaps can be done in the same work and span using a meldable heap such as leftist heaps.

Question 7: Geometric Coverage (12 points)

For points $p_1, p_2 \in \mathbb{R}^2$, we say that $p_1 = (x_1, y_1)$ *covers* $p_2 = (x_2, y_2)$ if $x_1 \geq x_2$ and $y_1 \geq y_2$. Given a set $S \subseteq \mathbb{R}^2$, the *geometric cover number* of a point $q \in \mathbb{R}^2$ is the number of points in S that q covers. Notice that by definition, every point covers itself, so its cover number must be at least 1.

In this problem, we'll compute the geometric cover number for every point in a given sequence. More precisely:

> **Input:** a sequence $S = \langle s_1, \ldots, s_n \rangle$, where each $s_i \in \mathbb{R}^2$ is a 2-d point.
>
> **Output:** a sequence of pairs each consististing of a point and its cover number. Each point must appear exactly once, but the points can be in any order.

Assume that we use the `ArraySequence` implementation for sequences.

(a) (4 points) Develop a brute-force solution `gcnBasic` (in pseudocode or Standard ML). Despite being a brute-force solution, your solution should not do more work than $O(n^2)$.

Solution:

fun GCN S =
  let fun covers$((x_1, y_1), (x_2, y_2))$ = $(x_1 \geq x_2) \wedge (y_1 \geq y_2)$
  in
    $\langle\, (p, |\, \langle\, p' \in P \mid \texttt{covers}(p, p')\, \rangle\, |) : p \in P\, \rangle$
  end

(b) (4 points) In words, outline an algorithm `gcnImproved` that has $O(n \log n)$ work. You may assume an implementation of `OrderedTable` in which `split`, `join`, and `insert` have $O(\log n)$ cost (i.e., work and span), and `size` and `empty` have $O(1)$ cost.

Solution: We'll keep an ordered table T of points ordered by their x values. Initially, T is empty. To compute the cover number for every point, we'll first sort these points by their y values. Then, for each of these points, we insert them one by one into T—and the cover number of this point can be found by splitting T using its x value and taking the size of the left side. This assumes we can calculate size in $O(\log n)$ work, which is easy with an augmented tree implementation of ordered tables.

(c) (4 points) Show that the work bound cannot be further improved by giving a lower bound for the problem.

Solution: We'll reduce comparison-based sorting to GCN, which means that GCN cannot be solved in less than $\Omega(n \log n)$ work. The reduction is as follows: for a given input sequence $s = \langle s_1, \ldots, s_n \rangle$, we create a sequence of points

$$P = \langle (s_1, s_1), (s_2, s_2), \ldots, (s_n, s_n) \rangle$$

(using map in $O(n)$ work and $O(1)$ span). Running GCN on this P gives the "rank" of each element, which we can then use as indicies to inject and get a sorted sequence.

Question 8: Swap with Compare-and-Swap (20 points)

(a) (10 points) Write a function `swap` that takes two memory locations la and lb and atomically swaps their values using compare-and-swap. Recall that compare-and-swap takes a memory location ℓ, an old value v, and a new value w and atomically replaces the contents of ℓ with w if the contents of ℓ is equal to v.

```
long lock = 0;

function swap-with-cas (la: long, lb: long) =
let
  function take_lock () =
    while (true) do
      if compare-and-swap (lock, 0, 1) then
        break;

  function release_lock () =
    while (true) do
      if compare-and-swap (lock, 1, 0) then
        break;

in
  take_lock ();
  long x <- load lx
  long y <- load ly
  store y into lx
  store x into ly
  release_lock ();
end
```

(b) (10 points) Does your algorithm suffer from the ABA problem? If so, explain how it does, and whether the problem affects the correctness of your algorithm. If so, then can you describe briefly a way to fix the problem (no pseudo-code needed)?

Solution: Yes it does, because the contents of `lock` can change between the load and the compare-and-swap to 1 and then back to 0. This however does not effect correctness because the atomicity is still guaranteed, because when the `lock` is 0, there is no other thread is the criticas section.

But if we still want to fix the ABA problem, then we can do so by making sure that each update to the location `lock` increments some version number. We would then insist on having the same version number during compare and swap. This would reduce the chances of the ABA problem but would not absolutely prevent it.

Appendix: Library Functions

```
signature SEQUENCE =
sig
  type 'a t
  type 'a seq = 'a t
  type 'a ord = 'a * 'a -> order
  datatype 'a listview = NIL | CONS of 'a * 'a seq
  datatype 'a treeview = EMPTY | ONE of 'a | PAIR of 'a seq * 'a seq

  exception Range
  exception Size

  val nth : 'a seq -> int -> 'a
  val length : 'a seq -> int
  val toList : 'a seq -> 'a list
  val toString : ('a -> string) -> 'a seq -> string
  val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool

  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val tabulate : (int -> 'a) -> int -> 'a seq
  val fromList : 'a list -> 'a seq

  val rev : 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
  val flatten : 'a seq seq -> 'a seq

  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val zip : 'a seq * 'b seq -> ('a * 'b) seq
  val zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq

  val enum : 'a seq -> (int * 'a) seq
  val filterIdx : (int * 'a -> bool) -> 'a seq -> 'a seq
  val mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq
  val update : 'a seq * (int * 'a) -> 'a seq
  val inject : 'a seq * (int * 'a) seq -> 'a seq

  val subseq : 'a seq -> int * int -> 'a seq
  val take : 'a seq -> int -> 'a seq
  val drop : 'a seq -> int -> 'a seq
  val splitHead : 'a seq -> 'a listview
  val splitMid : 'a seq -> 'a treeview



  val iterate : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
  val iteratePrefixes : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b seq * 'b
  val iteratePrefixesIncl : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val scan : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq * 'a
  val scanIncl : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq

  val sort : 'a ord -> 'a seq -> 'a seq
  val merge : 'a ord -> 'a seq * 'a seq -> 'a seq
  val collect : 'a ord -> ('a * 'b) seq -> ('a * 'b seq) seq
```

```
  val collate : 'a ord -> 'a seq ord
  val argmax : 'a ord -> 'a seq -> int

  val $ : 'a -> 'a seq
  val % : 'a list -> 'a seq
end
```

ArraySequence	Work	Span

`empty ()`
`singleton a`
`length s`
`nth s i`
`subseq s (i, len)`
	$O(1)$	$O(1)$

`tabulate f n`
if $f(i)$ has W_i work and S_i span
`map f s`
if $f(s[i])$ has W_i work and S_i span, and $\|s\| = n$
`zipWith f (s, t)`
if $f(s[i], t[i])$ has W_i work and S_i span, and $\min(\|s\|, \|t\|) = n$
	$O\left(\sum_{i=0}^{n-1} W_i\right)$	$O\left(\max_{i=0}^{n-1} S_i\right)$

`reduce f b s`
if `f` does constant work and $\|s\| = n$
`scan f b s`
if `f` does constant work and $\|s\| = n$
`filter p s`
if `p` does constant work and $\|s\| = n$
	$O(n)$	$O(\lg n)$

`flatten s`	$O\left(\sum_{i=0}^{n-1} \left(1 + \|s[i]\|\right)\right)$	$O(\lg \|s\|)$
`sort cmp s` if `cmp` does constant work and $\|s\| = n$	$O(n \lg n)$	$O(\lg^2 n)$
`merge cmp (s, t)` if `cmp` does constant work, $\|s\| = n$, and $\|t\| = m$	$O(m + n)$	$O(\lg(m + n))$
`append (s,t)` if $\|s\| = n$, and $\|t\| = m$	$O(m + n)$	$O(1)$

```
signature TABLE =
sig
  structure Key : EQKEY
  structure Seq : SEQUENCE

  type 'a t
  type 'a table = 'a t

  structure Set : SET where Key = Key and Seq = Seq

  val size : 'a table -> int
  val domain : 'a table -> Set.t
  val range : 'a table -> 'a Seq.t
  val toString : ('a -> string) -> 'a table -> string
  val toSeq : 'a table -> (Key.t * 'a) Seq.t

  val find : 'a table -> Key.t -> 'a option
  val insert : 'a table * (Key.t * 'a) -> 'a table
  val insertWith : ('a * 'a -> 'a) -> 'a table * (Key.t * 'a) -> 'a table
  val delete : 'a table * Key.t -> 'a table

  val empty : unit -> 'a table
  val singleton : Key.t * 'a -> 'a table
  val tabulate : (Key.t -> 'a) -> Set.t -> 'a table
  val collect : (Key.t * 'a) Seq.t -> 'a Seq.t table
  val fromSeq : (Key.t * 'a) Seq.t -> 'a table

  val map : ('a -> 'b) -> 'a table -> 'b table
  val mapKey : (Key.t * 'a -> 'b) -> 'a table -> 'b table
  val filter : ('a -> bool) -> 'a table -> 'a table
  val filterKey : (Key.t * 'a -> bool) -> 'a table -> 'a table

  val reduce : ('a * 'a -> 'a) -> 'a -> 'a table -> 'a
  val iterate : ('b * 'a -> 'b) -> 'b -> 'a table -> 'b
  val iteratePrefixes : ('b * 'a -> 'b) -> 'b -> 'a table -> ('b table * 'b)

  val union : ('a * 'a -> 'a) -> ('a table * 'a table) -> 'a table
  val intersection : ('a * 'b -> 'c) -> ('a table * 'b table) -> 'c table
  val difference : 'a table * 'b table -> 'a table

  val restrict : 'a table * Set.t -> 'a table
  val subtract : 'a table * Set.t -> 'a table

  val $ : (Key.t * 'a) -> 'a table
end
```

```
signature SET =
sig
  structure Key : EQKEY
  structure Seq : SEQUENCE

  type t
  type set = t

  val size : set -> int
  val toString : set -> string
  val toSeq : set -> Key.t Seq.t

  val empty : unit -> set
  val singleton : Key.t -> set
  val fromSeq : Key.t Seq.t -> set

  val find : set -> Key.t -> bool
  val insert : set * Key.t -> set
  val delete : set * Key.t -> set

  val filter : (Key.t -> bool) -> set -> set

  val reduceKey : (Key.t * Key.t -> Key.t) -> Key.t -> set -> Key.t
  val iterateKey : ('a * Key.t -> 'a) -> 'a -> set -> 'a

  val union : set * set -> set
  val intersection : set * set -> set
  val difference : set * set -> set

  val $ : Key.t -> set
end
```

MkTreapTable	Work	Span
`size` T	$O(1)$	$O(1)$
`filter` f T `map` f T	$\sum_{(k \mapsto v) \in T} W(f(v))$	$\lg \lvert T \rvert + \max_{(k \mapsto v) \in T} S(f(v))$
`tabulate` f X	$\sum_{k \in X} W(f(k))$	$\max_{k \in X} S(f(k))$
`reduce` f b T if f does constant work	$O(\lvert T \rvert)$	$O(\lg \lvert T \rvert)$

`insertWith` f `(`T`,(`k`,`v`))`
if f does constant work
`find` T k
`delete (`T`,`k`)`
	$O(\lg \lvert T \rvert)$	$O(\lg \lvert T \rvert)$

`domain` T `range` T `toSeq` T	$O(\lvert T \rvert)$	$O(\lg \lvert T \rvert)$
`collect` S `fromSeq` S	$O(\lvert S \rvert \lg \lvert S \rvert)$	$O(\lg^2 \lvert S \rvert)$

For each argument pair (A, B) below, let $n = \max(|A|, |B|)$ and $m = \min(|A|, |B|)$.

MkTreapTable	Work	Span

`union` f `(`X`,`Y`)`
`intersection` f `(`X`,`Y`)`
`difference (`X`,`Y`)`
`restrict (`T`,`X`)`
`subtract (`T`,`X`)`
	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$