Lab8 - 范围搜索实验

1. 实验要求

本次实验你将基于BST扩展order table 的ADT接口，完成一些基本函数，你可以从一般的库函数出发扩展此库。此外，你将完成一个范围搜索实验，即给定一个二维点集，以及一个矩形（用左上和右下坐标表示）范围，找出在此范围内点的个数，你需要自定义数据结构以满足复杂度需求。

2. 回答问题

2.1 完成函数first，last简述思路

- *Task 4.1 (6%)*. Implement the functions
  - `fun first (T : 'a table) : (key * 'a) option`
  - `fun last (T : 'a table) : (key * 'a) option`
  - Given an ordered table T, `first T` should evaluate to $SOME\ (k, v)$ iff $(k, v) \in T$ and k is the minimum key in T. Analagously, `last T` should evaluate to $SOME\ (k, v)$ iff $(k, v) \in T$ and k is the maximum key in T. Otherwise, they evaluate to $NONE$.

2.1.1 算法思路

1. 判断树是否为空，若为空则直接返回$NONE$
2. 对`first`而言：判断左子树是否为空
    1. 若是则表明已经没有比当前节点的优先级小的节点了，返回当前节点的键
    2. 若否则表明原树中优先级最小的节点与其左子树中优先级最小的节点相同，于是更新输入树为其左子树
3. 对`last`同理，将左/右，大/小互换即可

2.1.2 代码实现

```
fun first (T : 'a table) : (key * 'a) option =
        case Tree.expose T
        of NONE => NONE
        | SOME {key=k, value=v, left=l, right=r} =>
            if Tree.size l = 0 then SOME (k, v) else first l

fun last (T : 'a table) : (key * 'a) option =
        case Tree.expose T
        of NONE => NONE
        | SOME {key=k, value=v, left=l, right=r} =>
            if Tree.size r = 0 then SOME (k, v) else last r
```

2.1.3 关于测试

1. 测试任意多元素集（乱序排列，首/尾项不在对应首/尾位置）
2. 测试空集是否能返回正确结果

2.2 完成函数previous和next并简述思路

- *Task 4.2 (8%)*. Implement the functions
  - `fun previous (T : 'a table) (k : key) : (key * 'a) option`
  - `fun next (T : 'a table) (k : key) : (key * 'a) option`
  - Given an ordered table T and a key k, `previous T k` should evaluate to $SOME(k', v)$ if $(k_0, v) \in T$ and k_0 is the greatest key in T strictly less than k. Otherwise, it evaluates to $NONE$. Similarly, `next T k` should evaluate to $SOME\ (k', v)$ iff k_0 is the least key in T strictly greater than k.

2.2.1 算法思路

1. previous是要求绝对小于所给key的优先级的元素中最大的那一个，恰能用上面完成的last函数结合库函数中的splitAt实现
2. next同理
3. 自己尝试实现了一个previous，但是只能放在库函数里使用，因为提交部分中，NODE操作没有被载入

2.2.2 代码实现

```
fun next (T : 'a table) (k : key) : (key * 'a) option = first (#3 (Tree.splitAt (T, k)))

fun previous (T : 'a table) (k : key) : (key * 'a) option = last (#1 (Tree.splitAt (T,
k)))

(* 下面是自己尝试的一个更节省的实现，但是因为待完成的函子内部不能使用NODE操作，没法在要提交的代码里使用
*)
fun previous (T : 'a table) (k : key) : (key * 'a) option =
  let
    fun chk (EMPTY, _, _) = NONE
      | chk (NODE {data=d, left=l, right=r, ...}, SOME p, stats) =
        case Key.compare (k, #key d)
          of EQUAL => if l = EMPTY then NONE else SOME (#key (#data l), #value (#data
l))
           | LESS => if stats = GREATER then SOME p else chk (l, SOME (#key d, #value
d), LESS)
           | GREATER => if stats = LESS then SOME (#key d, #value d) else chk (r, SOME
(#key d, #value d), GREATER)
  in
    chk (T, NONE, EQUAL)
  end
```

2.2.3 关于测试

1. 测试k为任意多元素集的非首尾元素时能否给出正确结果（乱序排列，首/尾项不在对应首/尾位置）
2. 测试k为任意多元素集的首/尾元素时能否给出正确结果$NONE$
3. 测试空集是否能返回正确结果$NONE$

2.3 完成下列函数，做必要说明

- *Task 4.3 (2%)*. Implement the function
  - `fun join (L : 'a table, R : 'a table) : 'a table`
  - Given ordered tables L and R, where all the keys in L are strictly less than those in R, `join (L, R)` should evaluate to an ordered table containing all the keys from both L and R.
- *Task 4.4 (2%)*. Implement the function
  - `fun split (L : 'a table, k : key) : 'a table * 'a option * 'a table`
  - Given an ordered table T and a key k, split should evaluate to a triple consisting of
    1. an ordered table containing every $(k',\ v) \in T$ such that $k' < k$
    2. $SOME\ v$ if $(k,\ v) \in T$ and $NONE$ otherwise
    3. an ordered table containing every $(k',\ v) \in T$ such that $k' > k$.

2.3.1 算法思路

1. 库函数中的join与splitAt很显然能满足该题需求
2. 由于上述原因，不再重复实现一遍库函数内容，大致原理与上述previous的自主实现方式相同，需通过额外变量来保存两侧的树结构（库中使用了一个函数变量）

2.3.2 代码实现

```
fun join (L : 'a table, R : 'a table) : 'a table = Tree.join(L, R)

fun join ((EMPTY, t) | (t, EMPTY)) = t
  | join (nL as NODE {data=dL, left=lL, right=rL, ...},
          nR as NODE {data=dR, left=lR, right=rR, ...}) =
    if #pri dL < #pri dR
    then mk (dL, lL, join (rL, nR))
    else mk (dR, join (nL, lR), rR)

fun split (T : 'a table, k : key) : 'a table * 'a option * 'a table = Tree.splitAt (T,
k)

(* MkTreap.sml *)
fun splitAt (t, k) =
    let
      fun spl EMPTY f = f (EMPTY, NONE, EMPTY)
        | spl (NODE {data=d, left=l, right=r, ...}) f =
          case HashKey.compare (k, #key d)
            of EQUAL => f (l, SOME (#value d), r)
             | LESS => spl l (fn (L, M, R) => f (L, M, mk (d, R, r)))
             | GREATER => spl r (fn (L, M, R) => f (mk (d, l, L), M, R))
    in
      spl t (fn x => x)
    end
```

2.3.3 关于测试

1. 测试从树中有的节点切割，返回左右子树均不包含切割节点
2. 测试从树中没有的节点切割，返回正确的左右子树
3. 测试从边界以外切割，空子树能否正确返回

2.4 完成函数getRange并详述思路

- *Task 4.5 (7%)*. Implement the function
  - `fun getRange (T : 'a table) (low : key, high : key) : 'a table`
  - Given an ordered table T and keys l and h, `getRange T (l, h)` should evaluate to an ordered table containing every $(k, v) \in T$ such that $l \leq k \leq h$.

2.4.1 算法思路

1. split函数可以进行切割，因此进行两次切割后可以切出一个满足指定边界的开区间
2. 由于要求进行闭区间搜索，需在切割后判断切割点是否在原树中，若是则需在结果中加入该点

2.4.2 代码实现

```
fun getRange (T : 'a table) (low : key, high : key) : 'a table =
    case (Table.find T low, Table.find T high)
      of (NONE, NONE) => #1 (split (#3 (split (T, low)), high))
       | (SOME lv, NONE) => #1 (split (join (Tree.singleton(low, lv), #3 (split (T,
low))), high))
       | (NONE, SOME hv) => join (#1 (split (#3 (split (T, low)), high)), Tree.singleton
(high, hv))
       | (SOME lv, SOME hv) => join (#1 (split (join (Tree.singleton (low, lv), #3
(split (T, low))), high)), Tree.singleton(high, hv))
```

2.4.3 关于测试

1. 测试切割点在原树中存在的情况
2. 测试切割点在树中不存在但指定区间不包含树的所有节点的情况
3. 测试指定区间包含了树的所有节点的情况
4. 测试空树
5. 测试单元素树

2.5 完成函数makeCountTable并回答相关问题

- *Task 5.1 (25%)*. In the `MkRangeQuery` functor, define the `countTable` type and implement the function
    - `fun makeCountTable: point seq -> countTable`
    - The type point is defined to be `OrdTable.Key.t * OrdTable.Key.t` where `OrdTable` is an ordered table structure provided to you. You should choose the type of countTable such that you can implement `count (range queries)` in $\mathcal{O}(\log n)$ work and span. For full credit, your `makeCountTable` must run within $\mathcal{O}(n \log n)$expected work.
- *Task 5.2 (10%)*. Briefly describe how you would parallelize your code so that it runs in $\mathcal{O}(\log^2 n)$ span. Does the work remain the same? You don't need to formally prove the bounds, just briefly justify them.
- *Task 5.3 (5%)*. What is the expected space complexity of your `countTable` in terms of n the number of input points? That is, how many nodes in the underlying binary search tree(s) does your countTable use in expectation? Explain in a few short sentences.

2.5.1 算法思路

1. countTable做成二级Table，一级保存X值，一级保存Y值，便可以方便地依次排除两个方向上不在区域内的点
2. 根据提示，通过牺牲空间，每个横坐标的键保存大于等于它的所有横坐标上的点，可以通过简单的减法计算实现搜索时的时间加速
3. 先对X轴进行排序，使得每个X坐标上的点在比其大的X坐标处复制一份，从而制出满足要求的，以每个X坐标为键的有序Table，再对每个X坐标键对应的点集以Y坐标为键制成有序Table
4. 本题代码主要长度来源于格式修改操作，以保证最终能得到没有多余级别的二级Table，核心代码较短，为seqwithkeys变量的赋值操作，进行空间上的复制从而节省搜索的耗时

2.5.2 代码实现

```
  type countTable = point seq table table

  fun makeCountTable (S : point seq) : countTable =
    case Seq.length S
      of 0 => empty()
       | _ =>
          let
            (* W: nlgn, S: lg^2n *)
            val sorted = Seq.sort (fn ((x1, _), (x2, _)) => Key.compare (x1, x2)) S
            (* S: 1, W: n *)
            val withkeys = Seq.map (fn (x,y) => (x, (x, y))) S
            val keyTable = (OrdTable.collect withkeys)
            val keys = Set.toSeq (OrdTable.domain keyTable)
            (* S:  lg^2n, W: nlgn *)
            val seqwithkeys = Seq.sort (fn ((x1, _),(x2,_)) => Key.compare (x1, x2))
(Seq.tabulate (fn i => let val (x, s) = (Seq.nth keys i, valOf(OrdTable.find keyTable
(Seq.nth keys i))) in (x, s) end) (OrdTable.size keyTable))
            fun generateOne (m) =
              let
                val (x, s) = Seq.nth seqwithkeys m
              in
                Seq.tabulate (fn i => (#1 (Seq.nth seqwithkeys i), s)) (m+1)
```

```
              end
            val xtableraw = Seq.flatten (Seq.tabulate (fn n => generateOne n)
(OrdTable.size keyTable))
            val xtable = OrdTable.collect xtableraw
            val redundent = OrdTable.map (fn t => Seq.map (fn (x,y) => (y, (x,y)))
(Seq.flatten t)) xtable
          in
            OrdTable.map (fn t => OrdTable.collect t) redundent
          end
```

2.5.3 关于测试

- 本题测试在2.6.3中完成

2.5.4 Task 5.2

- 待并行部分为Work与Span相差较大且Work与Span本身较大的部分，已于注释中标出
  - `sorted`：对X坐标排序以进行后续复制操作，并行后$Span \in \mathcal{O}(\log^2 n)$
  - `seqwithkeys`：主要步骤，调用span较低的`tabulate`、`find`、`size`等库函数进行各点依X坐标大小的复制，主要span来源于sort的快速排序，并行后$Span \in \mathcal{O}(\log^2 n)$
  - `withkeys`：该变量的赋值用到map操作，Work较高，但$Span \in \mathcal{O}(1)$
  - 其余用到Tabulate之类的Work高但$Span \in \mathcal{O}(1)$的操作
- 综上，总Span为$\mathcal{O}(\log^2 n)$，显然这些并行操作只是将同样的工作分给多个处理器进行，不会改变总工作量，即work不变

2.5.5 Task 5.3 (空间复杂度计算)

1. `sorted+withkeys+keyTable`: $\mathcal{O}(n)$
2. `seqwithkeys`:
   1. 复制每个X坐标上的所有点到比该坐标大的X坐标上，最坏情况假设点的分布是稀疏的，|x| = |y| = n

$$S_1 \in \mathcal{O}(\sum_{i=1}^{n}(n-i)) = \mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2)$$

   2. 对1中所得的串的每个元素依据其#1进行排序，快排空间复杂度最坏为$\mathcal{O}(m)$，其中$m = S_1$
   3. 故$S \in \mathcal{O}(n^2)$
3. `xtableraw+xtable+redundent`: $\mathcal{O}(seqwithkeys)$
4. 综上，空间复杂度为$\mathcal{O}(n^2)$

2.5.4 时间复杂度计算（检查代码留的题）

1. Span：由2.5.2可知，$Span \in \mathcal{O}(\log^2 n)$
2. Work：
   - `sorted`：快排，$Work \in \mathcal{O}(n \log n)$
   - `seqwithkeys`：各点依X坐标大小进行复制操作，先进行$Work \in \mathcal{O}(n)$的tabulate，后进行sort，$Work \in \mathcal{O}(n \log n)$
   - 综上，总Work为$\mathcal{O}(n \log n)$

2.5.5 flatten的复杂度计算（检查代码留的题）

1. 记有$|a|$个串, 每个串$|x_a|$个元素
2. Work：由于flatten表示将所有的串合并为一个，因此其Work相当于append作用于X个串，总工作量为$\mathcal{W}(flatten) = |a| * \mathcal{W}(append)$, 在ArraySeq与ListSeq下为$\mathcal{O}(1 + |a| + \sum_{x \in a} |x_a|)$, 在TreeSeq下, 以$\mathcal{O}(\log (\sum_{x \in a} |x_a|))$为单次最坏复杂度，则Work为$\mathcal{O}(1 + |a| * \log (\sum_{x \in a} |x_a|))$
3. Span：以ArraySeq为例，计算每个串将被写入到的新串中的位置需要用到`scan`，而写入是完全并行的，复杂度为$\mathcal{O}(1)$,因此总复杂度与`scan`相同，为$\mathcal{O}(1 + \log |a|)$

2.6 完成函数count并做相关分析

- *Task 5.4 (25%)*. Implement the function
  - count: countTable -> point * point -> int
  - As described earlier, $count\ T\ ((x_1,\ y_1),\ (x_2,\ y_2))$ will report the umber of points within the rectangle with the top-left corner (x1, y1) and bottom-right corner $(x_2,\ y_2)$. Your function should return the number of points within and on the boundary of the rectangle. You may find the `OrdTable.size` function useful here. Your implementation should have $\mathcal{O}(\log n)$ work and span.

2.6.1 算法思路

1. 由于上述countTable的结构为二级Table，第一级Table的某一个键表示一个X坐标，其所对应的值表示大于等于该X坐标的所有点以Y坐标为键组成的Table
2. 因此，根据题目提示知，要找区域内的点的数目，只需要先求出较小X坐标右侧（包含该坐标）所有Y值满足要求的点（用getRange求Y值满足条件的点），再求出较大X坐标右侧（不包含该坐标）的所有Y值满足要求的点，二者相减即为所求
3. 函数中用到的map与reduce并不是针对所有点进行的,而是针对range进行的，有着与OrdTable.range同样的work
4. 由于仅需进行`getRange`、`find`、`reduce`、`range`等操作，work与span都为$\mathcal{O}(\log n)$

2.6.2 代码实现

```
(* Right of (leftbound) - Right of (rightbound) *)
fun count (T : countTable)
                  ((xLeft, yHi) : point, (xRght, yLo) : point) : int  =
  case OrdTable.size T
    of 0 => 0
     | _ => (Seq.reduce (fn (a,b) => a+b) 0 (Seq.map (fn x => Seq.length x)
(OrdTable.range (getRange (getOpt(OrdTable.find T xLeft, (#2 (getOpt(next T xLeft,
(xLeft, OrdTable.empty()))))))) (yLo, yHi))))) - (Seq.reduce (fn (a,b) => a+b) 0 (Seq.map
(fn x => Seq.length x) (OrdTable.range (getRange (#2 (getOpt(next T xRght, (xRght,
OrdTable.empty()))))) (yLo, yHi)))))
```

2.6.3 关于测试

1. 测试点全在区域内/外的情况
2. 分别测试X/Y边界上有点，区域内有/无点的四种情况
3. 测试边角有点，区域内无点情况
4. 测试根本没有点的情况、只有一个处于上述不同位置的点的情况