## 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)∈T and k is the minimum key in T. Analagously, last T should evaluate to SOME (k,v) iff (k,v)∈T and k is the maximum key in T. Otherwise, they evaluate to NONE.

|  |
| --- |
| * 算法思路： * first求解思路：   1.如果树为NONE，则返回NONE  2.若不为NONE，则沿着左子树一直找到叶子结点，递归完成这个函数，返回叶子结点的值  last求解思路：  1.如果树为NONE，则返回NONE  3.若不为NONE，则沿着右子树一直找到叶子结点，递归完成，返回last的值 |
| * 代码实现： * 1.first的实现：   fun first (T : 'a table) : (key \* 'a) option =  case (Tree.expose T) of  NONE => NONE  | SOME {key, value, left, right} =>  case (Tree.expose left) of  NONE => SOME(key,value)  | \_ => first (left)  2. last的实现：  fun last (T : 'a table) : (key \* 'a) option =  case (Tree.expose T) of  NONE => NONE  | SOME {key, value, left, right} =>  case Tree.expose right of  NONE => SOME(key,value)  | \_ => last (right) |
| * 关于测试：   测试样例如下：  val ordSet1 = % [5, 7, 2, 8, 9, 1]  val testsFirst = [  (\*以下为原始样例\*)  ordSet1,  % [],  (\*以下为自己添加的测试\*)  % [1,3,5,7,9],  % [8,7,6,5,4,3,2,1],  % [100],  % [0]  ]  val testsLast = [  (\*以下为原始样例\*)  ordSet1,  % [],  (\*以下为自己添加的测试\*)  % [1,3,5,7,9],  % [6,5,4,3,2,1],  % [1],  % [100000]  ]  测试截图如下： |

2.2 完成函数previous和last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 (k0,v)∈ T and k0 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 k0 is the least key in T strictly greater than k.

|  |
| --- |
| * 算法思路： * 详见库中Mktreap.sml，splitAt(T,K)，将T分成三个部分，key比k小的一个table，一个pair(k,v)，和一个key比k大的table   1. Previous的实现：  previous取splitAt后的第一个元素，即key比k小的table，取这个table的last即为k之前key最大的元素  2. next的实现：  next取splitAt后的第二个元素，即key比k大的table，取这个table的first，即为k之后key最小的元素 |
| * 代码实现：   1.previous的代码：  fun previous (T : 'a table) (k : key) : (key \* 'a) option =  last (#1 (Tree.splitAt (T, k)))  2.next的代码：  fun next (T : 'a table) (k : key) : (key \* 'a) option =  first (#3 (Tree.splitAt (T, k))) |
| * 关于测试：   测试样例如下：  val ordSet1 = % [5, 7, 2, 8, 9, 1]  val testsPrev = [  (\*以下为原始样例\*)  (ordSet1, 8),  (ordSet1, 1),  (% [], 8),  (\*以下为自己添加的测试\*)  (% [1,3,5,7,9], 1),  (% [6,5,4,3,2,1], 5),  (% [1], 2),  (% [100000], 4)  ]  val testsNext = [  (\*以下为原始样例\*)  (ordSet1, 8),  (ordSet1, 9),  (% [], 8),  (\*以下为自己添加的测试\*)  (% [1,3,5,7,9], 6),  (% [6,5,4,3,2,1], 3),  (% [1], 1),  (%[1,8,2,6,5], 3)  ]  测试截图如下： |
|  |

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.

|  |
| --- |
| * 算法思路：   详见库中Mktreap.sml，要把两个table进行join，我们直接调用库中的join函数即可 |
| * 代码实现：   fun join (L : 'a table, R : 'a table) : 'a table =  Tree.join (L, R) |
| * 关于测试：   Join测试样例：  val ordSet1 = % [5, 7, 2, 8, 9, 1]  val testsJoin = [  (\*以下为原始样例\*)  (ordSet1, % [100]),  (ordSet1, % [3]),  (% [], % [100]),  (\*以下为自己添加的测试\*)  (% [], % [1,2,3,4,5]),  (% [1,2,3,4,5], %[]),  (%[], %[])  ]  实验截图如下： |

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)∈T such that k' < k,

2. SOME v if (k,v)∈T and NONE otherwise, and

3. an ordered table containing every (k',v)∈T such that k'> k.

|  |
| --- |
| * 算法思路：   详见库中Mktreap.sml，要将table进行split，获得key比k小的一个table，一个pair(k,v)，和一个key比k大的table，直接调用函数splitAt(T,k)即可 |
| * 代码实现：   fun split (T : 'a table, k : key) : 'a table \* 'a option \* 'a table =  Tree.splitAt(T,k) |
| * 关于测试：   Split测试样例：  val ordSet1 = % [5, 7, 2, 8, 9, 1]  val testsSplit = [  (\*以下为原始样例\*)  (ordSet1, 7),  (ordSet1, 100),  (% [], 7),  (\*以下为自己添加的测试\*)  (% [], 1),  (% [], 0),  (% [100,99,98,97,96], 0),  (% [1,2,3,4,5,6,7], 4)  ]  测试截图如下： |

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)∈T such that l≤k≤h.

|  |
| --- |
| * 算法思路：   我们要做的是获取key的值在low和high之间的一个table(含low和high)，步骤如下：  1.首先我们从lowerbound(low)进行split，取右子树，我们得到了key大于等于low的树，并且注意，如果low是这棵树的key，我们要把low这个节点加入  2.对于我们已经取出的key大于等于low的树，对其upperbound进行界定，在high进行split，取左子树，同样，如果high是这棵树的key，则加入high这个节点  3.返回截取好的树 |
| * 代码实现：   fun getRange (T : 'a table) (low : key, high : key) : 'a table =  let  val lower\_bound = case split(T, low) of  (L, NONE, R) => R  |(L, SOME v, R ) => join(Tree.singleton(low,v),R)  val upper\_bound = case split(lower\_bound,high) of  (L, NONE, R) => L  |(L, SOME v, R) => join(L,Tree.singleton(high,v))  in  upper\_bound  end |
| * 关于测试：   getRange测试样例：  val ordSet1 = % [5, 7, 2, 8, 9, 1]  val testsRange = [  (\*以下为原始样例\*)  (ordSet1, (5,8)),  (ordSet1, (10,12)),  (% [], (5,8)),  (\*以下为自己添加的测试\*)  (% [1,2,5,3,7,4,9],(2,7)),  (% [], (1,2)),  (% [], (3,10)),  (% [1,2,3,4,5,6,7,8,9,10], (1,10)),  (% [1,2,3,4,5,6,7,8,9,10], (5,6))  ]  测试结果如下： |

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 toyou. You should choose the type of countTable such that you can implement count (range queries) in O(logn) work and span. For full credit, your makeCountTable must run within O(nlogn) expected work.

|  |
| --- |
| * 算法思路：   我们构造一个table，table的key是x的值，table的value是一个table和另一个数的数对，我们用这个table来存储比当前x小于或等于的x所对应的y值，而这另一个数就是当前的y值，实现如下：  1.我们按照x对整个table进行排序。  2.定义一个函数table\_insert,参数值为((count\_table, y\_table),(x,y))，其中count\_table是我们要求的table，y\_table用来存储比当前x小的x所对应的y值，(x,y)为当前要加入的table的数对  3.用iterate调用该函数，构造该table，将(x,y)一个个加入，知道iterate完成 |
| * 代码实现：   fun makeCountTable (S : point seq) : countTable =  if Seq.length S = 0 then empty()  else  let  fun compare\_x ((x1,\_),(x2,\_)) = compareKey(x1,x2)  val sorted\_x = Seq.sort (compare\_x) (S)  fun table\_insert ((count\_table, y\_table),(x,y)) =  let  val new\_y\_table = insert (fn (v,v') => v) ((y,y)) (y\_table)  val new\_count\_table = insert (fn (v,v') => v)((x,(new\_y\_table, y))) (count\_table)  in  (new\_count\_table, new\_y\_table)  end  val (final\_table, \_) = Seq.iter(table\_insert)((empty(),empty()))(sorted\_x)  in  final\_table  end |
| * 复杂度分析：   首先分析sorted\_x这一句，这一句按照x对整个points sequence进行排序，即这一句work为O(nlogn), span为O(log2 n)。  然后是构造final\_table这一句，我们用iterate将当x前坐标对应的y坐标插入到y\_table中，构造当前x坐标对应的y\_table。每次insert的work和span都是O(logn)，而我们用iterate操作构造n个x坐标对应的y\_table，故总的work和span都是O(nlogn)。  综上，这个算法work和span都是O(nlogn)。 |

Task 5.2 (10%). Brieﬂy describe how you would parallelize your code so that it runs in O(log2 n) span. Does the work remain the same? You don’t need to formally prove the bounds, just brieﬂy justify them.

|  |
| --- |
| 我们可以用scan来代替iter实现这个算法的并行实现。运用的函数仍然是table\_insert，id仍然是(empty(),empty())。这样这个函数实现的work不变，还是O(nlogn),，span降为O(log2 n)。 |

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.

|  |
| --- |
| 节点的数量大概为O()。在makeCountTable函数中，我们每回往y\_table里面加入当前坐标x对应的y值，即每一个x节点对应相应的y\_table都为当前x坐标以及之前的x坐标对应的所有y的集合。所以我们可以看到第一个x对应为1，第二个为2，一直到n，这所有节点的总和就为O()。 |

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

Asdescribedearlier, count T ((x\_1,y\_1), (x\_2, y\_2))willreportthenumberofpointswithinthe rectangle with the top-left corner (x1, y1) and bottom-right corner (x2, y2). 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 O(logn) work and span.

|  |
| --- |
| * 算法思路：   我们用小于xRght且y范围在yLo和yHi之间的点减去小于xLeft且y范围在yLo和yHi之间的点，然后加上x值为xLefty范围在yLo和yHi之间的点，得到范围内的点  1.如果table的大小为0，则返回0  2.大小不为0情况下，首先看小于xLeft且y范围在yLo和yHi之间的点，由于我们构造的countTable的value值为一个key为所有在当前x之前的点的y值和当前y值组成的数对，所以我们先寻找xLeft是不是在该countTable中，如果在，返回所有在该点及其之前的对应的y值(那个table)，如果不在，则找xLeft前一最大的x所对应的ytable  3.对于截取出来的那个y值table进行getrange，命名为leftrange  4.寻找x=xLeft这一条线上有没有区域内的点。如果xLeft上面有对应的y值且这个y值在yLo和yHi之间，则我们可以判断这个点是被重复减去的，命名为repeat\_left  5.再看小于等于xRght且y范围在yLo和yHi之间的点，若xRght在该countTable中返回所有在该点及其之前的对应的y值(那个table)，如果不在，则找xLeft前一最大的x所对应的ytable  6.对于这回截取出来的那个y值table进行getrange,命名为rightrange  7.将rightrange的大小减去leftrange的大小加上repeat\_left(因为repeat\_left本应该属于区域但被减去)，得到区域内点的数量 |
| * 代码实现：   fun count (T : countTable)  ((xLeft, yHi) : point, (xRght, yLo) : point) : int =  if size T = 0 then 0  else  let  val SOME(\_,(left\_y,y1)) = case (find(T)(xLeft)) of  NONE => previous(T)(xLeft)  | SOME(v) => SOME(xLeft,v)  val leftrange = getRange(left\_y)(yLo,yHi)  fun whether\_cross y1 = case (compareKey(y1,yHi),compareKey(y1,yLo)) of  (GREATER,\_) => 0  | (\_,LESS) => 0  | (\_,\_) => 1  val repeat\_left = case (find(T)(xLeft)) of  NONE => 0  | \_ => whether\_cross y1  val SOME(\_,(right\_y, y2)) = case (find(T)(xRght)) of  NONE => previous(T)(xRght)  |SOME y\_pair => SOME(xRght,y\_pair)  val rightrange = getRange(right\_y)(yLo,yHi)  in  OrdTable.size(rightrange) - OrdTable.size(leftrange) + repeat\_left  end |
| * 复杂度分析： * 首先我们有对于ordered table，操作find，getRange，previous的work和span都是O(logn)。  1. 确定左边边界，即left\_y, 运用了一次find操作，如果NONE，再用一次previous操作，故整体的work和span保持在O(logn) 2. 对左边边界getRange（即leftrange），work和span为O(logn) 3. 判断左边界上是否有压线的点(repeat\_left)，运用了一次find操作work和span为O(logn)，compare操作work和span均为O(1) 4. 同理，确定右边界和确定左边界一样，work和span保持在O(logn) 5. 同理，对右边边界getRange（即leftrange），work和span为O(logn) 6. 最后计算点的数量时候，运用了两个size，work和span不会超过O(logn)，加法和减法work和span为O(1) 7. 综上，总体的work和span在为(logn) |
| * 关于测试：   Count测试样例：    (\*以下为原始测试点\*)  val points1 = % [(0,0),(1,2),(3,3),(4,4),(5,1)]  val points2 : point seq = % []  val points3 = % [(10000,10000),(0,0)]  val points4 = tabulate (fn i => (i,i)) 1000  (\*以下为自己添加的测试点\*)  val mypoint1 = % [(11,0),(10,1),(9,2),(8,3),(7,4),(6,5),(5,6),(4,7),(3,8),(2,9),(1,10),(0,11)]  val mypoint2 = % [(0,9),(1,1),(2,8),(3,2),(4,7),(5,3),(6,6),(7,4),(8,5)]  val testsCount = [  (\*以下为原始样例\*)  (points1, ((1,3),(5,1))),  (points1, ((2,4),(4,2))),  (points1, ((100,101),(101,100))),  (points2, ((0,10),(10,0))),  (points3, ((0,10000),(10000,0))),  (points4, ((0,500),(1000,0))),  (\*以下为自己添加的测试测试\*)  (mypoint1, ((1,9),(6,1))),  (mypoint1, ((0,10),(10,0))),  (mypoint1, ((3,5),(5,3))),  (mypoint2, ((0,5),(5,0))),  (mypoint2, ((1,4),(5,0)))  ]  测试结果如下： |