目录

Lab1 括弧匹配实验.....1

1. 实验要求.....1
2. 实验思路.....1
3. 回答问题.....2

Lab3 bignumlab.....5

1. 实验要求.....5
2. 实验思路.....5
3. 回答问题.....6

Lab5 同义词实验.....13

1. 实验要求.....13
2. 回答问题.....13

Lab8 范围搜索实验.....25

1. 实验要求.....25
2. 回答问题.....25

Lab1 括弧匹配实验

1. 实验要求

给定一个由括号构成的串，若该串是合法匹配的，返回串中所有匹配的括号对中左右括号距离的最大值；否则返回 NONE。左右括号的距离定义为串中二者之间字符的数量，即 $max\{j-i+1 \mid (si, sj)$ 是串 s 中一对匹配的括号$\}$。要求分别使用枚举法和分治法求解。

2. 实验思路

2.1 分治法求解思路

1.首先对于一个定义一个(m,l,r,ld,rd),其中每个部分分别代表的意义为：

m:闭括号中最大括号长度

l:没有匹配的左括号数

r:没有匹配的右括号数

ld:最左边的左括号到串末尾的距离

rd:最右边右括号到串开始处的距离

2.设定空串：EMPTY（0,0,0,0,0）

左括号：OPAREN(0,1,0,1,0)

右括号：CPAREN(0,0,1,0,1)

3.如果不是空的或者左右括号，则分为左右两边(m1, l1, r1, ld1, rd1), (m2, l2, r2, ld2, rd2)，两边并行处理，合并规则为:

1).重新计算闭括号最大长度以及左右未匹配括号

如果左边没匹配左括号与右边没匹配右括号相等,则计算中间闭括号长度 m0 与 m1 和 m2 比较，得到最大值

如果左边没匹配左括号比右边没匹配右括号要少，合并之后的 l0 就是右边的左括号数，合并之后的 r0 是左右两边括号总数减去已匹配的左边左括号数

如果左边没匹配左括号不比右边没匹配右括号要少，合并之后 l0 就是两边括号数之和减去右边已经匹配的右括号，合并之后的 r0 就是左边的右括号数

2).重新计算左边最左到末尾以及右边最右到开始的距离

如果左边没匹配左括号数大于右边没匹配右括号数,则左边左括号没匹配完，新的 ld 就是 ld1 加上右边的总长度，否则就是右边的左括号数

如果右边没匹配右括号数大于左边没匹配左括号数,则右边右括号没

匹配完，新的的 rd 就是 rd2 加上左边的总长度，否则就是左边的右括号数
4. 对于输入的串 parens，返回其最长闭括号所含元素(max)，未匹配左右括号数，如果未匹配左右括号数均为 0 并且输入的串本身长度大于 0，则最长闭括号所含元素为 max，否则左右括号是没有匹配的，返回 NONE
（具体实践如代码所示）

3. 回答问题

3.1 关于枚举法求解

Task 5.2 (5%). What is the work and span of your brute-force solution? You should assume subseq has *O(1)* work and span, where m is the length of the resulting subsequence, and parenMatch has *O(n)* work and $O(log^2 n)$ span where *n* is the length of the sequence.

1. Work: brute force 算法遍历了所有的连续闭子串，如果我们假设 subseq 操作用 *O(1)* 的 work，而判断一个串是否匹配，即是 parenMatch 函数，用 O(n)的 work，那么判断我们所取出的一个子串是否匹配（即 parenMatch 函数）用 work 就是 O(n)。现在我们计算 MatchDist 函数的复杂度，它计算第 m 位开始的(第 m 位为左括号时)闭括号的长度，由于它要判断逐一 m 以后括号是否匹配并返回长度，并且每次判断是都调用 parenMatch 函数，故执行一次这个函数的 work 为$O(n^2)$。所以我们用 tabulate 从第位到第 n 位调用 MatchDist，记录从第 1 位到第 n 位所有连续闭子串长度的 work 为$O(n^3)$，然后我们计算 n 个长度中最大的长度，由于我们是两两都要比较，这一步用的 work 是$O(n^2)$，所以综上，这个算法的 work 是$O(n^3)$。
2. Span: 我们运用 subseq 操作用的 span 是 O(1)，而判断一个串是否匹配，即 parenMatch 函数，用的 span 是$O(log^2 n)$，这个就是我们计算每个闭子串所需要的 span，所以我们用 tabulate 记录所有连续闭子串长度所用的 span 为$O(log^2 n)$。（所有子串并行处理。）最后我们要计算 n 和长度中最大的元素，由于是所有 n 个值都两两比较，这一步的 span 为$O(n^2)$。所以综上，这个算法的 span 是$O(n^2)$。

3.2 关于分治法求解

Task 5.4 (20%). The specification in Task 5.3 stated that the work of your solution must follow a recurrence that was parametric in the work it takes to view a sequence as a tree. Naturally, this depends on the implementation of SEQUENCE.

1. Solve the work recurrence with the assumption that $Wshowt \in \Theta(lg\ n)$ where *n* is the length of the input sequence.

2. Solve the work recurrence with the assumption that $Wshowt \in \Theta(n)$ where n is the length of the input sequence.

3. In two or three sentences, describe a data structure to implement the sequence α seq that allows

showt to have *Θ(lg n)* work.

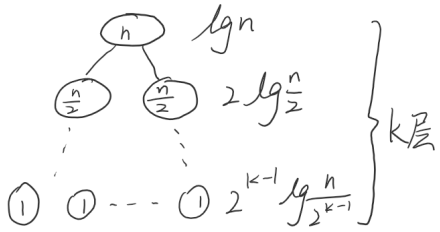
4. In two or three sentences, describe a data structure to implement the sequence α seq that allows showt to have *Θ(n)* work.

divide and conquer 的 work 可以表示成:

W(n)=2*W(n/2)+Wshowt(n)+O(1)

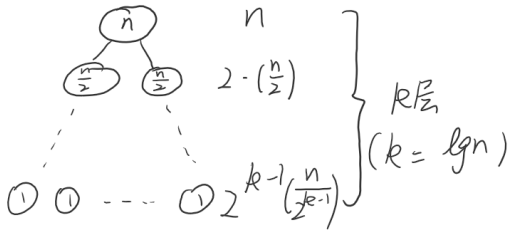
1.对于第一种情况，即 *Wshowt ∈Θ(lg n)*

W(n)=2*W(n/2)+ Θ(lg n)，我们令 n=2^k，所以我们有这颗树的第一层 work 为 lgn，第二层为2lg(n/2)，第k层有$2^{k-1}\lg\left(\frac{n}{2^{k-1}}\right)$。而我们可以看到第二层与第一层work的比为 2lg(n/2)/lgn = 2-2/lgn > 1（因为 n>2），所以这棵树是 leaf-dominated 的，即 W(n)=O($2^{k-1}\lg\left(\frac{n}{2^{k-1}}\right)$) $= O(n)$



2.对于第二种情况，即 *Wshowt ∈Θ(n)*

即 W(n)=2*W(n/2)+ *Θ(n)* ,所以我们有这颗树第一层 work 为 n，第二层 work 为 2*(n/2)=n，第 k 层为 n。这颗树是 balanced 的，一共有 logn 层，故 W(n)=(nlogn)



3.用一棵 balanced tree 来存这个串，这种情况下 showt 会产生一棵高为 logn 的树，复杂度也就为 O(logn)

4.用一个头指针在链表表头的链表去存这个串，所以 showt 每次都要遍历这个链表去执行操作，因为这个串的长度是 n，所以复杂度就是 O(n)

3.3 关于渐进复杂度分析

Task 6.1 (5%). Rearrange the list of functions below so that it is ordered with respect to O—that

is, for every index *i*, all of the functions with index less than i are in big-O of the function at index *i*. You can just state the ordering; you don't need to prove anything.

1. $f(n) = n^{log(n^2)}$
2. $f(n) = 2n^{1.5}$
3. $f(n) = (n^n)!$
4. $f(n) = 43^n$
5. $f(n) = lg(lg(lg(lg(n))))$
6. $f(n) = 36n^{52} + 15n^{18} + n^2$
7. $f(n) = n^{n!}$

复杂度由低到高为：

5 2 6 1 4 7 3

Task 6.2 (15%). Carefully prove each of the following statements, or provide a counterexample and prove that it is in fact a counterexample. You should refer to the definition of big-O. Remember that verbose proofs are not necessarily careful proofs.

1. *O* is a transitive relation on functions. That is to say, for any functions *f*, *g*, *h*, if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.
2. *O* is a symmetric relation on functions. That is to say, for any functions *f* and *g* , if $f \in O(g)$, then $g \in O(f)$.
3. *O* is an anti-symmetric relation on functions. That is to say, for any functions *f* and *g* , if $f \in O(g)$ and $g \in O(f)$, then $f=g$.

1. 传递性证明：因为 f∈0(g) 和 g∈0(h)，我们有一定存在n_1, n_2, c_1, c_2，使得对于 n>n_1，有 f(n)< $c_1 g(n)$，且对于 n>n_2，有$g(n) < c_2 h(n)$，所以得到，对于 n>max{n_1, n_2}，有 f(n)< $c_1 c_2 h(n)$，所以 f∈0(h)

2. 对称性反例：由 big-0 的定义有，存在 c 和n_0，使得对于所有的 n>n_0，有 f(n)<c*g(n) 成立。我们可以看出 $2n^2$=0(n^3)，在（c = 1， $n_0 = 2$）情况下成立，而我们找不出 c 和n_0，让所有的 n>n_0，有 f(n)<c*g(n)成立。故这和个性质时是不存在的。

3. 反对称性反例：我们可以设 f(n)=n+1，g(n)=n+2，由 big-0 的定义有,存在 c=1，对一切 n>0 都有 f(n)<g(n)，即 f∈*O(g)*。而我们也存在 c=2，对一切 n>1,有 g(n)<2*f(n),即 $g \in O(f)$。

而此时并没有 f=g，故这个性质不成立。

Lab3 bignumlab

1. 实验要求

实现 n 位二进制大整数的加法运算。输入 a, b 和输出 s 都是二进制位的串。要求算法的时间复杂度满足 work=O(n)， span=O(log n)。

2. 实验思路

2.1．加法求解思路

1.我们对于两个 bignum，我们对于每一位产生一个 pairs，pairs 内容包含产生的本位值和进位值。例如 pairs (ONE,ONE) = (ZERO,GEN)。

2.我们对于进位值进行 scan 操作确定到底有哪一位产生了进位。

3.将进位值与本位值进行合并，即若本位值碰到 GEN 即翻转，1 变为 0, 0 变为 1，其他情况保持不变。

4.最后去掉高位多余的 0

2.2．减法求解思路

1.我们把 y 的值翻转加一，再加上 x 的值。

2.去掉高于 x 长度的位数，得到减法所得的的数

2.3.乘法求解思路

1．首先通过补充 0，将两个 bignum 变得等长且为偶数长度

2．若是两个空串，则返回空串；若补充之后两个串的长度为 1，则判定若两个数都是 ONE，则为 ONE，否则为空串。

3.对于其他情况，我们将串 x 分成 p、q 相同长度的两份，将串 y 分成 r、s 两份，按照公式 $A \cdot B = pr \cdot 2n + (ps+rq) \cdot 2n/2 + qs$ 求得结果，其中 $ps+rq = (p+q)*(r+s) - pr - qs$ 求解。（其中我们并行计算$(p+q)*(r+s), p*r, q*s$ 这三个式子）

（具体实现参照下面代码）

3. 回答问题

3.1 提供加法计算的代码和注释

Task 4.1 (35%). Implement the addition function

++ : bignum * bignum -> bignum

in the functor MkBigNumAdd in MkBigNumAdd.sml. For full credit, on input with *m* and *n* bits, yoursolution must have $O(m+n)$ work and $O(\lg(m+n))$ span. Our solution has under 40 lines with comments.

```
 functor MkBigNumAdd(structure U : BIGNUM_UTIL) : BIGNUM_ADD =
struct
  structure Util = U
  open Util
  open Seq

  infix 6 ++
  exception NotYetImplemented
  datatype carry = GEN | PROP | STOP

  fun x ++ y =
    let
      val min = Int.min(length x, length y)
      val max = Int.max(length x, length y)

      (*定义一个 pairs，模拟二进制数每一位的加法，得到的结果也是一个数对，第一个是本位的结果，第二个判断是否进位*)
      fun pairs (ZERO,ZERO) = (ZERO,STOP)
        | pairs (ONE,ONE) = (ZERO,GEN)
        | pairs (ZERO,ONE) = (ONE,PROP)
        | pairs (ONE,ZERO) = (ONE,PROP)

      (*定义 first 和 second 函数，分别用于取 pairs 数对的第一个和第二个，用于后续分成两部分计算本位值和进位值*)
      fun first (i,_) = i
      fun second (_,i) = i

      (*定义 addbits 函数，遍历两个 bignum，使得加法的每一位都形成一个 pairs，较小的数高位用 ZERO 补齐*)
      fun addbits a =
        if(a < min) then pairs ((nth x a),(nth y a))
        else if (a<length x) then pairs ((nth x a),ZERO)
        else if (a<length y) then pairs (ZERO,(nth y a))
        else (ZERO,STOP)
```

```
        (*计算 x,y 经过 addbits 的结果*)
        val addresult0 = tabulate addbits max

        (*下面我们首先来计算进位，判断整个加法完成时候到底有哪些位置会产生进位
          首先定义一个 carrystep 判断进位关系的产生、传递和停止关系
          然后 carryseq 取用每一位单独的进位结果，即 addresult0 每一项的第二位进行 map
          然后执行 scan 操作，判断 carry 之后到底哪些位产生了真实进位
          最后很关键的一步，由于产生进位是从第二位开始的，是对下一位产生进位，所以进位关系是对于下一位而言，这里要在最低位补充一个 PROP 或者 STOP
          (代码中最低位补充的是 PROP)
          *)
        fun carrystep (a,PROP) = a
          | carrystep (_,GEN) = GEN
          | carrystep (_,STOP) = STOP
        val carryseq = map second addresult0
        val carryresult0 = scani carrystep PROP carryseq
        val carryresult = append(singleton(PROP), carryresult0)

        (*下面来计算每一位本位产生的值
          首先先取 addresult0 的每一项的第一个，即为本位值，同时为防止最高位向下一位进位，这里在给最高位留出一个位子，用 ZERO 补充
          *)
        val addresult1 = map first addresult0
        val addresult2 = append(addresult1, singleton(ZERO))

        (*下面将本位和进位加起来，
          定义一个反转函数，如果碰到进位就调用这个函数，ZERO 变为 ONE，ONE 变为 ZERO(因为这一位多加了一个 ONE)
          然后是主要的 addtwoparts 函数，对于本位的每一个数字，如果相应的碰到 GEN，就产生位翻转，否则不变
          最后对每一位使用这个函数，用 tabulate 并行实现
          *)
        fun reverse ZERO = ONE
          | reverse ONE = ZERO

        fun addtwoparts i =
          if (i >= length addresult2) then ZERO
          else if (nth carryresult i = GEN) then reverse (nth addresult2 i)
          else (nth addresult2 i)

        val result = tabulate addtwoparts (length addresult2)

    in
```

```
    (*最后判断最高位是否为 ONE，若不为 ONE，则为 ZERO，去掉多余的零*)
      if(nth result ((length result)-1) = ONE) then result
      else subseq result (0, length result - 1)
    end

  val add = op++
end
```

3.2 提供减法计算的代码和注释

Task 4.2 (15%). Implement the subtraction function

-- : bignum * bignum -> bignum

in the functor MkBigNumSubtract in MkBigNumSubtract.sml, where *x* -- *y* computes the number obtained by subtracting *y* from *x*. We will assume that $x \geqslant y$; that is, the resulting number will alwaysbe non-negative. You should also assume for this problem that ++ has been implemented correctly. Forfull credit, if *x* has *n* bits, your solution must have $O(n)$ work and $O(\lg n)$ span. Our solution has fewerthan 20 lines with comments.

```
  functor MkBigNumSubtract(structure BNA : BIGNUM_ADD) : BIGNUM_SUBTRACT =
struct
  structure Util = BNA.Util
  open Util
  open Seq

  exception NotYetImplemented
  infix 6 ++ --
  fun x ++ y = BNA.add (x, y)
  fun x -- y =
    let
      (*编写反转函数，将串 y 中的元素 0 变成 1,1 变成 0，其他位用 0 补齐*)
      fun reverse y i =
        if i < 0 orelse i >= length y orelse nth y i = ZERO
          then ONE
          else ZERO
      (*将 y 的值翻转加一得到负数的值，用 x 加上 y 的负数，然后舍弃最高位得到减法的
值*)
      val revadd = (singleton ONE) ++ (tabulate (reverse y) (length x)) ++ x
      val result0 = take (revadd , length x)
    in
      result0
    end

  val sub = op--
end
```

3.3 提供乘法计算的代码和注释

Task 4.3 (30%). Implement the function

** : bignum * bignum -> bignum

in MkBigNumMultiply.sml. For full credit, if the larger number has n bits, your solution must satisfy$W_{**}(n) = 3 \cdot W_{**}(n/2)+ O(n)$and have O ($\lg^2 n$) span. You should use the following function in the Primitives structure:

val par3 : (unit -> 'a) * (unit -> 'b) * (unit -> 'c) -> 'a * 'b * 'c

to indicate three-way parallelism in your implementation of **. You should assume for this problem that++ and -- have been implemented correctly, and meet their work and span requirements. Our solution has 40 lines with comments.

```
functor MkBigNumMultiply(structure BNA : BIGNUM_ADD
                         structure BNS : BIGNUM_SUBTRACT
                         sharing BNA.Util = BNS.Util) : BIGNUM_MULTIPLY =
struct
  structure Util = BNA.Util
  open Util
  open Seq
  open Primitives
  exception NotYetImplemented

  infix 6 ++ --
  infix 7 **

  fun x ++ y = BNA.add (x, y)
  fun x -- y = BNS.sub (x, y)
  fun x ** y =
    let
      (*我们的思路是先把 x,y 变得同样长且长度为偶数，方便后续 divide
        不够的位数我们用 0 补，定义 addzero 往高位补 n 个 0
        makeaqual 函数把两个 bignum 变得一样长，即若谁短，谁补差值个 0
        makeiteven 函数即在两个穿长度相同情况下，若长度为奇数，则高位加 0 补为偶数
        *)
      fun addzero (x,n) = append(x,(tabulate (fn i => ZERO) n))
      fun makeequal (x,y) =
        case Int.compare(length x, length y) of
            LESS =>((addzero(x,length y - length x)),y)
          | GREATER => (x,addzero(y,length x - length y))
          | _ => (x,y)
      fun makeiteven (x,y) =
```

```
            case(length x) mod 2 of
                  0 => (x,y)
                | 1 => (addzero(x,1),addzero(y,1))
          (*定义 pow2 意思为 x 乘以 2 的 n 次方，实际上就是往左移动 n 位，低位用 0 补齐*)
          fun pow2 (x,n) =
              case length x of
                    0 => empty()
                  | _ => append((tabulate(fn i => ZERO) n), x)
      in
          let
            (*先把两个串变得一样长，若长度是 0，就是空，若长度是 1 且两个都是 ONE，证明为 ONE，否则为 ZERO
              其余的，执行 divide 操作，先把串长 n 变成偶数，定义 p、q、r、s 分别为 x1 的高 n/2 位，低 n/2 位，x2 的高 n/2 位，低 n/2 位
              并行计算 p ** r，q ** s，(p ++ q) ** (r ++ s)
              计算 ps+rq=(p ++ q)(r ++ s)-pr-qs
              根据公式计 AB=pr*2^n+(ps+qr)*2^(n/2)+qs 算出两大数乘法*)
            val (x1,x2) = makeequal(x,y)
          in
            case length x1 of
                  0 => empty()
              | 1=> (case (nth x1 0, nth x2 0) of
                          (ONE, ONE) => singleton(ONE)
                        | _ => empty())
              | _ => let
                          val (x0,y0) = makeiteven(x1,x2)
                          val n = length x0
                          val p = drop(x0, n div 2)
                          val q = take(x0, n div 2)
                          val r = drop(y0, n div 2)
                          val s = take(y0, n div 2)

                          val (pr,qs,pqrs) = par3(fn() => p ** r, fn() => q ** s, fn() => ((p ++ q) ** (r ++ s)))
                          val psrq = pqrs -- pr --qs
                          val AB = pow2(pr, n) ++ pow2(psrq, n div 2) ++ qs
                        in
                          AB
                        end
          end
      end
    val mul = op**
end
```

3.4 迭代计算复杂度分析

Task 5.1 (15%). Determine the complexity of the following recurrences. Give tight Θ-bounds, and justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$. You may use any method (brick method, tree method, or substitution) to show that your bound is correct, except that you must use the substitution method for problem 3.

1. T (n) = 3T(n/2) + $\Theta(n)$
2. T (n) = 2T(n/4) +$\Theta(\sqrt{n})$
3. T (n) = 4T(n/4) +$\Theta(\sqrt{n})$ (Prove by substitution.

1．T (n) = $\Theta(n^{log_2 3})$，证明如下：

首先，明显这棵树的树高是$k = log_2 n$，每一层的值为$3^i \cdot \theta\left(\frac{n}{2^i}\right)$，即我们有，这棵树的总和为：

$$T(n) = \sum_{i=0}^{k}\left(3^i \cdot \theta\left(\frac{n}{2^i}\right)\right) = \theta\left(n\sum_{i=0}^{k}\left(\frac{3}{2}\right)^i\right) = \theta\left(n\frac{1-\left(\frac{3}{2}\right)^{k-1}}{1-\frac{3}{2}}\right)$$

$$= \theta\left(2n\left(\left(\frac{3}{2}\right)^{k-1}-1\right)\right) = \theta\left(2n\left(\left(\frac{3}{2}\right)^{log_2 n-1}-1\right)\right) = \theta\left(2n\left(\frac{3}{2}\right)^{\log_2 n}\right)$$

$$= \theta\left(3^{\log_2 n}\right) = \theta\left(3^{\log_2 3 \cdot \log_3 n}\right) = \theta\left(n^{\log_2 3}\right)$$

2. $T(n) = \theta\left(\sqrt{n} \cdot log_4 n\right)$, 证明如下：

首先，这棵数的树高是$K = log_4 n$，每一层的值为$2^i \cdot \theta\left(\sqrt{\frac{n}{4^i}}\right) = \theta\left(\sqrt{n}\right)$，即我们有，这和棵树的额总和为：

$$\mathrm{T(n)} = \sum_{i=0}^{k}\theta\left(\sqrt{n}\right) = \theta\left(\sqrt{n}\log_4 n\right)$$

3. $T(n) = \theta(n)$，证明如下：

要证明. $T(n) = \theta(n)$，即证明存在C_1, C_2，使得$C_1 n \le T(n) \le C_2 n$。令$n = 4^i$

对于 base case：i=0，有$C_1 \le T(1) = 1 \le C_2$，即$C_1 \le 1, C_2 \ge 1$即可

对于 induction：i=k 时首先若存在C_1, C_2，使得$c_1 4^k \le T(4^k) \le C_2 4^k$

i=k+1 时，有$T\left(4^i + 1\right) = 4T\left(4^i\right) + \theta\left(2^{i+1}\right)$

即$C_1 \cdot 4^{i+1} + C_1\theta\left(2^{i+1}\right) \le \bar{T}\left(4^{i+1}\right) \le C_2 \cdot 4^{i+1} + C_2\theta\left(2^{i+1}\right)$

即$C_1 \cdot 4^{i+1} \le C_1 \cdot 4^{i+1} + C_1\theta\left(2^{i+1}\right) \le \bar{T}\left(4^{i+1}\right) \le C_2 \cdot 4^{i+1} + C_2\theta\left(2^{i+1}\right) \le 2C_2 \cdot 4^{i+1}$

即$C_1 \cdot 4^{i+1} \le \bar{T}\left(4^{i+1}\right) \le 2C_2 \cdot 4^{i+1}$，存在$C_1' = C_1, C_2' = 2C_2$使得等式成立。

根据上述论述，有. $T(n) = \theta(n)$成立

Lab5 同义词实验

1. 实验要求

本次实验中，你将完成一个寻找无权图中最短路径的 ADT，最短路算法被广泛应用于很多地方，你将应用你的解决方案用于处理同义词问题。即给定任意两个单词，在给定的同义词库中找出它们之间的最短路，并且任意两个单词有边相连表示他们是同义词关系。

2. 回答问题

2.1.0 Graph Construction 定义 graph 的类型

Task 4.1 (2%). In MkAllShortestPaths.sml, define the type graph that would allow you to implement the following functions within the required cost bounds. **Leave a brief comment explaining why you chose the representation that you did.**

Graph 类型定义代码如下：

type graph = (((Set.set) table)*int*int)

定义的 graph 类型为：一个表示图的 table * 图的边的数量 * 图的节点的数量。

其中 table 的 key 为当前节点，value 为它的 outneighbors，即为一个子节点指向出度节点的 table。然后我们要把这个 table 用一个 set 存（一个 key 有多个对应 value），因为我们后续在计算节点数量时候需要用到 set 里面的 union 操作。

2.1.1 简述 makeGraph 函数的思路

Task 4.2 (8%). Implement the function
makeGraph : edge seq -> graph
which generates a graph based on an input sequence E of directed edges. The number of vertices in the the resulting graph is equal to the number of vertex labels in the edge sequence. For full credit, makeGraph must have O(|E|log|E|) work and O(log2|E|) span.

makegraph 求解思路如下：

1. 对于 sequence E(边的集合)进行 collect，得到我们想要的 key 指向 outneighbors 的一个 table，然后用 map 操作把 collect 之后得到的 seq 变成 set，方便后续 union 操作求得 vertices 的数量。
2. 将 table 中的 key 和 E 中的第二个元素(即每个 key 对应的 outneighbor)进行 union 操作，得到的集合即为图中所有的点。
3. 函数返回值即为我们构造好的 table_graph，边的数量 length E，以及节点的数量 Set.size vertices。

2.1.2 makeGraph 函数的代码实现及复杂度分析

代码实现如下：

```
fun makeGraph (E : edge seq) : graph =
  let
    val table_graph = Table.map (Set.fromSeq) (Table.collect E)
    val vertices = Set.union(domain table_graph, Set.fromSeq (map #2 E))
  in
    (table_graph, length E, Set.size vertices)
  end
```

复杂度分析如下：

第一句构造 table_graph,collect 复杂度为 O(|E|log|E|) work 和 O(log2 |E|) span，然后再用 map 把 seq 变成 set，复杂度为 O(V)的 work 和 O(1)的 span，不会超过 collect 的复杂度。

第二句把 key 和 value 进行 set.union, union 复杂度为 O(mlog(1+n/m)) work 和 O(log2 (1+n/m)),其中 n 为两个 set 中较大的，m 为两个 set 中较小的。显然 m 肯定远小于|E|（因为节点数量远少于边的数量），1+n/m 更是远小于|E|，这一句的复杂度远小于 collect 复杂度。

计算 length E 的 work 在 O(1),span 在 O(1);计算 Set.size vertices 的 work 在 O(|V|),span 为 O(1)

综上,这个函数主要复杂度在 collect 上,复杂度为 O(|E|log|E|) work 和 O(log2 |E|) span。

2.1.3 numEdges 和 numVertices 函数的实现

Task 4.3 (6%). Implement the functions
numEdges : graph -> int numVertices : graph -> int
which return the number of directed edges and the number of unique vertices in the graph, respectively.

思路如下：

我们在 makegarph 的时候已经计算了 numEdges 和 numVertices，这里直接调用即可，代码实现如下：

```
(* Task 2.2 *)
(*边的数量即为 graph 的第二个元素*)
 fun numEdges (G : graph) : int = #2 G
(*节点的数量即为 graph 的第三个元素*)
fun numVertices (G : graph) : int = #3 G
```

2.1.4 简述 outNeighbors 函数的思路

Task 4.4 (6%). Implement the function
outNeighbors : graph -> vertex -> vertex seq
which returns a sequence Vout containing all out neighbors of the input vertex. In other words, given a graph G =(V,E), outNeighbors G v containsall w s.t. (v,w)∈E. If the input vertex is not in the graph, outNeighbors returns an empty sequence. For full credit, outNeighbors must have O(|Vout|+log|V|) work and O(log|V|) span, where V is the set of vertices in the graph.

实现思路如下：
我们查找 v 的 value，如果 v 没有 value，证明它没有 outneighbors，返回空串；如果有 value，返回这些 value

2.1.5 outNeighbors 函数的代码实现及复杂度分析

代码实现如下：

```
fun outNeighbors (G : graph) (v : vertex) : vertex seq =
    case find (#1 G) v of
        NONE => Seq.empty()
        |SOME s => Set.toSeq s
```

复杂度分析：
Find 这一句的复杂度的 work 和 span 均为 O(log|V|)，Set.toSeq s 这一句 work 为 O(|Vout|)，span 为 0(1)。综上这个函数对的 work 为 O(|Vout|+log|V|)，span 为 O(log|V|)。

2.1.6 ASP 类型的确定及简短说明

Task4.5(2%). InMkAllShortestPaths.sml,definethetypeaspthatwouldallowyoutoimplement the following functions within the required cost bounds. Leave a brief comment explaining why you chose the representation that you did.

ASP 类型定义代码如下：

```
type asp = (vertex seq table)
```

我们定义的 asp 类型为一个由子节点指向父节点 seq 的 table

2.1.7 简述 makeASP 函数的思路

Task 4.6 (23%). Implement the function
makeASP : graph -> vertex -> asp
to generate an asp which contains information about all of the shortest paths from the input vertex v to all other reachable vertices. If v is not in the graph, the resulting asp will be empty. Given a graph G = (V,E), makeASP G v must have O(|E|log|V|) work and O(Dlog2|V|) span, where D is the longest shortest path (i.e., the shortest distance to the vertex that is the farthest from v).

解决思路如下：
1. 如果节点 v 没有 outneighbors，即返回空的 table。

2. 其他情况下，执行 BFS 操作，如果 F(frontier)为空，则返回 X(已访问过的节点)。
3. 如果 F 不为空，首先定义一个函数 mkparentpair，对于节点 s 的 outneighbors，定义由子节点反过来指向父节点的 pair。
4. 对于当前的 frontier 的节点，全部调用 mkparentpair 函数，得到每个节点由子节点指向父节点的串，将这个串 flatten 之后 collect，得到了每个子节点指向其所有父节点的 table，这些父节点用 seq 表示。
5. 将 F 加入 X 形成新的 X，将新指向的 frontier 减去已经访问过的 X 得到新的 F，将新的 X 和 F 输出，继续以其为参数调用 BFS,直到 F 的大小为 0

2.1.8 makeASP 函数的代码实现及复杂度分析

代码实现如下：

```
fun makeASP (G : graph) (v : vertex) : asp =
    (*构造一个由子节点指向父节点的 table，即我们所求的 asp，开始节点为 v*)
    case find (#1 G) v of
      (*如果 v 节点没有邻接的边，即返回空的 table*)
       NONE => Table.empty()
      |SOME _ =>
        let
          fun BFS (X,F) =
            if size F = 0 then X else
            let
              (*这个函数将 s 指向它的父节点(outNeighbors)*)
              fun mkparentpair s = Seq.map (fn v => (v,s)) (outNeighbors G s)
               (*这一步将所有 F 中的元素指向好的 pairs 进行 flatten 之后 collect 合并，最终变为 F 中的元素每个都指向其父元素的 table*)
val table_parent_pair = Table.collect (flatten (map mkparentpair (Set.toSeq (domain F))))
              (*这一步将 X 和 F 合并，形成新的 X，已访问过的节点*)
              val update_X = Table.merge (fn(a,b) => a) (X,F)
              (*这一步将新的将新的 Frontier，即从集合中去掉已经访问过的 X*)
              val update_F = Table.erase (table_parent_pair, domain update_X)
            in
              (*新的 X 和 F，递归进行 BFS 操作*)
              BFS(update_X, update_F)
            end
        in
          (*初始状态，已访问过节点集合 X 为空，Frontier 为 v*)
          BFS(Table.empty(), Table.singleton(v,Seq.empty()))
    end
```

复杂度分析如下：

回顾整个算法，首先看 v 节点在不在 graph 里面，如果不在，返回空的 table，其余情况调用 BFS,这一句 work 和 span 仅为 O(log|V|)。调用 BFS 即为,对当前 F(frontier)都执行当前节点指向其 parents

的操作然后再对 key（即子节点）进行 collect 构造出子节点指向父节点 seq 的 table，这样递归调用 BFS，这一句 work 为 O(||F||log|V|)，span 为 O(log2 |V|)，然后每次更新已经访问过的节点 X 和 F（frontier），更新 X 即把 F 并进原来的 X，work 不会超过 O（|F|），span 不会超过 O（1）；更新 F 即为寻找当前更新过的 X 新的 frontier，即为 collect 过后的 table 的元素，并去掉已访问过的元素，work 不会超过 O（log|V|），span 不会超过 O（1），结束条件是直到 F 里面元素个数为 0，遍历完整个 graph。又我们结束时遍历完了所有的边，边的总数为|E|，每一次调用 BFS 的 work 保持在 O(||F||log|V|)，故递归完成时我们的遍历完所有的边，work 为 O（|E|log|V|），又因为深度为 D，每一次 span 保持在 O(log2 |V|)，故总的 span 保持在 O(Dlog2 |V|)

综上，该算法的 work 为 O（|E|log|V|），span 为 O(Dlog2 |V|)。

2.1.9 简述 report 函数的思路

Task 4.7 (15%). Implement the function

report : asp -> vertex -> vertex seq seq

which, given an asp for a source vertex u, returns all shortest paths from u to the input vertex v as a sequence of paths (each path is a sequence of vertices). If no such path exists, report asp v evaluates to the empty sequence. For full credit, report must have O(|P||L|log|V|) work and span, where V is the set of vertices in the graph, P is the number of shortest paths from u to v, and L is the length of the shortest path from u to v.

代码实现思路：

1. 首先寻找 G 中有无 v，如果没有，返回空串
2. 如果有 v 但是 v 没有 frontier，返回 v
3. 其它情况下，从 v 开始调用 DFS，每一次找其 parent，遍历所有路径到各初始节点 u
4. 最后将串翻转，得到从 u 到 v 的路径

2.1.10 report 函数的实现及复杂度分析

代码实现如下：

```
    fun report (A : asp) (v : vertex) : vertex seq seq =
     let
      fun DFS (v : vertex) : vertex list seq =
          (*寻找节点 v 是否在 A 中，如果不在，依照题目要求返回空串，如果为孤立点，它的 path 为它自己*)
          case Table.find A v of
             NONE => Seq.empty ()
            |SOME parents => if (Seq.length parents) = 0 then Seq.singleton [v]
              else
              (*其他情况，从 v 开始，递归调用 DFS，倒过来沿着每条路径去找初始节点*)
               let
                  fun path_to_parent u = DFS u
                  fun add_v vpath = [v] @ vpath
```

```
                  val  parent_vpath  =  Seq.flatten  (Seq.map  path_to_parent
parents)
                  val vpath = Seq.map add_v parent_vpath
                in
                  vpath
                end
      in
      (*将串翻转，得到 u 到 v 的路径*)
        let
          fun reverse_seq_vpath vpath = Seq.rev (Seq.fromList vpath)
          val all_vpath = DFS v
          val afterRev_vpath = Seq.map (reverse_seq_vpath) (all_vpath)
        in
          afterRev_vpath
        end
      end
```

复杂度分析：

回顾整个算法，首先我们用 find 查找 v 是否在 A 里面，如果有返回其 parents(v 对应的 value)，如果没有返回空串，这一步复杂度为 O(log|V|)。然后我们递归调用 DFS 函数，并每次利用 map 操作对每个 parents 里面的元素指向下一个元素，将 map 后的结果 flatten，然后我们将构造好的路径加上 v（因为我们从 v 的 parents 开始构造的，没有算 v），由此构造出 v 到起始节点 u 的路径。由于我们是递归执行以上操作，所以这里的 work 和 span 就是每次遍历的节点的个数的和乘上 O(log|V|)(因为我们每次都要调用 find)。又因为一共有路径|P|条，路径深度为|L|，所以节点遍历个数和为|P|*|L|,每一次都要调用 find，所以构造从 v 到起点 u 的路径 work 和 span 为 O(|P||L|log|V|)。最后我们还要将串翻转，得到从起点 u 到 v 的路径，由于一共有路径|P|条，路径深度为|L|，所以 work 不会超过 O(|P|*|L|),我们可以并行执行此操作，所以 span 不会超过 O(|L|)。

综上，整个算法的 work 和 span 为 O(|P||L|log|V|)。

2.1.11 关于测试

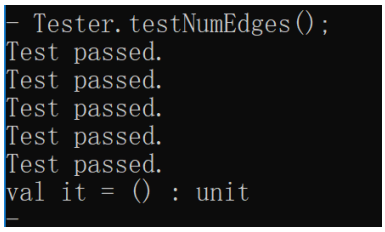
Task 4.8 (6%). Test your ALL_SHORTEST_PATHS implementation in the file Tests.sml by adding test cases to the appropriate lists (see the file for reference – there are existing test cases to guide you). The following functions are defined to you're your implementation of functions of ALL_SHORTEST_PATHS against your test cases in Tests.sml.

原本测试样例和自己添加的测试如下：

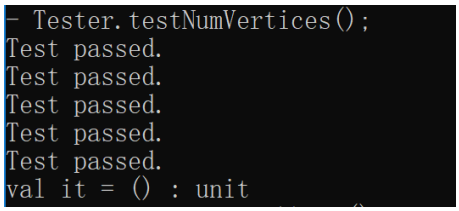
```
(*下面为原本的测试样例*)
  val edgeseq = [(1,2)]
  val edgeseq2 = [(1,2),(2,3),(3,4),(2,4),(1,5),(5,4),(5,6),(6,7)]
  val test3 = [(2,1),(4,2),(4,3),(4,5),(1,3),(3,5),(3,2),(1,4),(2,5),(5,1)]

(*下面为自己添加的测试数据*)
```

```
    val myedgeseq1 = [(1,4),(2,6),(3,4),(5,1),(4,2),(5,3),(1,2)]
    val myedgeseq2 = [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]

(*NumEdges 和 NumVertices 测试*)
val testsNum = [(*以下为本身提供的样例*)
                        edgeseq, edgeseq2,test3,

                        (*以下为自己添加的测试*)
                        myedgeseq1,myedgeseq2
                        ];
(*OutNeighbors 测试*)
val testsOutNeighbors = [(*以下为本身提供的样例*)
                    (edgeseq, 1), (edgeseq, 2),(test3,1),(edgeseq2,5),(edgeseq2,7),(test3,9),
                            (*以下为自己添加的测试*)
                            (edgeseq2,3),(test3,3),(test3,5),
                            (myedgeseq1,1),(myedgeseq1,3),(myedgeseq1,5),
                            (myedgeseq2,1),(myedgeseq2,3),(myedgeseq2,5)]

(*Report 测试*)
    val testsReport = [(*以下为本身提供的样例*)
                            ((edgeseq, 1), 2), ((edgeseq2, 1), 4), ((edgeseq2, 1), 7),((test3,4),2),
                            ((test3,1),3),((test3,6),2),((test3,1),6),

                            (*以下为自己添加的测试*)
                            ((edgeseq2,3),4),((test3,3),7),((test3,5),7),
                            ((myedgeseq1,2),6),((myedgeseq1,3),7),((myedgeseq1,4),8),
                            ((myedgeseq2,1),4),((myedgeseq2,4),1),((myedgeseq2,5),2)]
```

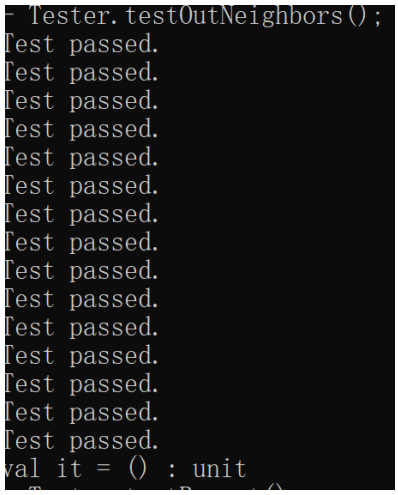
NumEdges 测试截图如下：



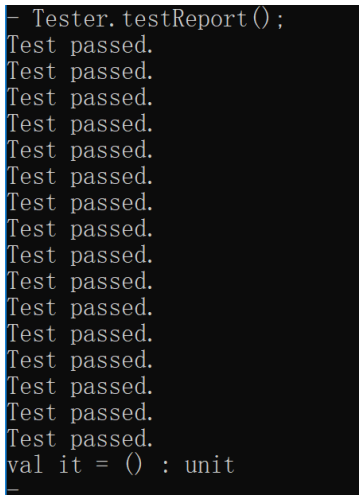
NumVertices 测试截图如下：



OutNeighbors 测试截图如下：



Report 测试截图如下：



2.1.12 定义 thesaurus 数据类型并简述理由

thesaurus 类型定义代码如下：

```
type thesaurus = graph
```

把 thesaurus 定义成 graph，对于近义词，即为 key 指向其近义词的一个 pair，这样形成一个 graph。

2.1.13 实现下列函数并简述算法核心

Task 5.1 (10%). Implement the function
make : (string * string seq) seq -> thesaurus which generates a thesaurus given an input sequence of pairs (w,S) such that each word w is paired with its sequence of synonyms S. You must define the type thesaurus yourself.

算法思路如下：

1. 定义一个 single_mkedges 函数，作出 word 指向每个 synonyms 的 pair。
2. 对于 S 中每一个 w 调用上述函数，构造所有的 pairs。

3. 用 makegraph 将其变成图。

实现代码如下：

```
fun make (S : (string * string seq) seq) : thesaurus =
    let
      (*这里对于一个 w，作出由 word 指向 synonyms 的 pair*)
      fun single_mkedges (w, syn) = Seq.map (fn s => (w, s)) syn
      (*对于 S 中的每一个 w，都作 single_mkedges 操作，构造所有的 pairs*)
      val mkedges = flatten (map single_mkedges S)
    in
      (*将 edges 数对变成图*)
      makeGraph mkedges
    end
```

Task 5.2 (6%). Implement the functions
numWords : thesaurus -> int
synonyms : thesaurus -> string -> string seq
where numWords counts the number of distinct words in the thesaurus while synonyms returns a sequence containing the synonyms of the input word in the thesaurus. synonyms returns an empty sequence if the input word is not in the thesaurus.

算法实现思路：

直接对已经变成 graph 类型的 T，调用 numVertices 以及 outNeighbors 即可。

算法实现如下：

```
    (* Task 3.2 *)
    (*word 的数量就是 graph 中节点的数量*)
    fun numWords (T : thesaurus) : int = numVertices T

    (*每个 w 的近义词就是它的 outneighbors*)
   fun synonyms (T : thesaurus) (w : string) : string seq = outNeighbors T w
```

Task 5.3 (10%). Implement the function
query : thesaurus -> string -> string -> string seq seq such that query th w1 w2 returns all shortest path from w1 to w2 as a sequence of strings with w1 first and w2 last. If no such path exists, query returns the empty sequence. For full credit, your function query must be staged.

算法实现思路：

要返回 w1 到 w2 的 all shortest path，只需要对其进行 report 操作，这里我们以 w1 为原点，构造一个 asp，report w1 到 w2 的 all shortest path

实现代码如下：

```
    fun query (T : thesaurus) (w1 : string) (w2 : string) : string seq seq =
     report (makeASP T w1) w2
```

2.1.14 关于测试

Task 5.4 (6%). Test your THESAURUS implementation in the file Tests.sml as before. The

following functions are defined to run your implementation of functions of ALL_SHORTEST_PATHS against your test cases in Tests.sml.

测试样例以及自己添加的样例如下：

```
  val testfile = "input/thesaurus.txt"
  val testfile2 = "input/simpletest.txt"

  (*NumWords 测试*)
  val testsNumWords =   [testfile, testfile2]

  (*Synonyms 测试*)
  val testsSynonyms =
    [(*以下为原始测试样例*)
      (testfile2, "HANDSOME"),
      (testfile2, "VINCENT"),
      (testfile2, "PRETTY"),
      (testfile2,"BADASS"),
      (testfile, "GOOD"),

      (*以下为自己添加的测试样例*)
      (testfile,"EDIT"),
      (testfile,"PRICE"),
      (testfile,"MAY"),
      (testfile2,"YOLO"),
      (testfile2,"BILL")]

  (*Query 测试*)
  val testsQuery =
    [ (*以下为原始测试样例*)
      (testfile2, ("HANDSOME", "YOLO")),(testfile2,("BADASS","STUPID")),(testfile2,("PRETTY","CHRIS")),
      (testfile, ("GOOD", "BAD")),(testfile, ("CLEAR", "VAGUE")),(testfile, ("LOGICAL", "ILLOGICAL")),
      (testfile,("HAPPY","SAD")),(testfile,("LIBERAL","CONSERVATION")),(testfile, ("EARTHLY", "POISON")),

      (*以下为自己添加的测试样例*)
      (testfile, ("EVENT","DOZE")),(testfile,("EQUIP","LAMENTATION")),

(testfile,("A","ONE")),(testfile,("ARC","ARCH")),(testfile,("HOME","WAX")),(testfile,("AMPLE","LESS"))]
```

NumWords 测试如下：

```
- Tester.testNumWords();
Test passed.
Test passed.
```

Synonyms 测试截图如下：

```
- Tester.testSynonyms();
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
```

Query 测试截图如下：

```
- Tester.testQuery();
HANDSOME -> VINCENT -> YOLO
GOOD -> EXCELLENT -> SUPERB -> IMPRESSIVE -> AWESOME -> TERRIBLE -> WORST -> BAD
GOOD -> WORTH -> MERIT -> EARN -> WIN -> CONQUER -> WORST -> BAD
CLEAR -> EVACUATE -> EMPTY -> EXHAUST -> WEAKEN -> FAINT -> INAUDIBLE -> INDISTINCT -> UNCLEAR -> VAGUE
LOGICAL -> VALID -> ACCURATE -> EXQUISITE -> DAINTY -> DELICATE -> UNHEALTHY -> UNSOUND -> IMPRACTICAL -> ILLOGICAL
HAPPY -> GAY -> LIVELY -> VOLATIBLE -> CHANGEABLE -> VARIABLE -> MOODY -> SAD
LIBERAL -> LAVISH -> SQUANDER -> WASTE -> EBB -> WITHDRAW -> RETREAT -> SHELTER -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> WASTE -> LEAK -> ESCAPE -> RETREAT -> SHELTER -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> DISSIPATE -> SCATTER -> SPREAD -> COVER -> SHELTER -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> DISSIPATE -> DIFFUSE -> SPREAD -> COVER -> SHELTER -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> WASTE -> LEAK -> OOZE -> FILTER -> SCREEN -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> DISSIPATE -> SCATTER -> SPREAD -> COVER -> GUARD -> PROTECTION -> CONSERVATION
LIBERAL -> LAVISH -> SQUANDER -> DISSIPATE -> DIFFUSE -> SPREAD -> COVER -> GUARD -> PROTECTION -> CONSERVATION
EARTHLY -> SENSUAL -> EROTIC -> OBSCENE -> FOUL -> POLLUTE -> TAINT -> POISON
EVENT -> EPISODE -> INTERLUDE -> PAUSE -> REST -> SLEEP -> DOZE
EVENT -> EPISODE -> INTERLUDE -> PAUSE -> REST -> REPOSE -> DOZE
EVENT -> EPISODE -> INTERLUDE -> INTERMISSION -> LULL -> REPOSE -> DOZE
EQUIP -> OUTFIT -> SUIT -> TRAIN -> INSTRUCT -> CHARGE -> COMPLAINT -> LAMENTATION
A -> ONE
ARC -> ARCH
HOME -> REFUGE -> SHELTER -> PROTECT -> COAT -> WAX
HOME -> REFUGE -> SHELTER -> COVER -> COAT -> WAX
AMPLE -> WIDE -> BROAD -> GENERAL -> COMMON -> SHODDY -> INFERIOR -> LESS

val it = () : unit
-
```

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：

```
- Tester.testFirst();
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
val it = () : unit
- Tester.testLast();
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
val it = () : unit
-
```

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)))
```

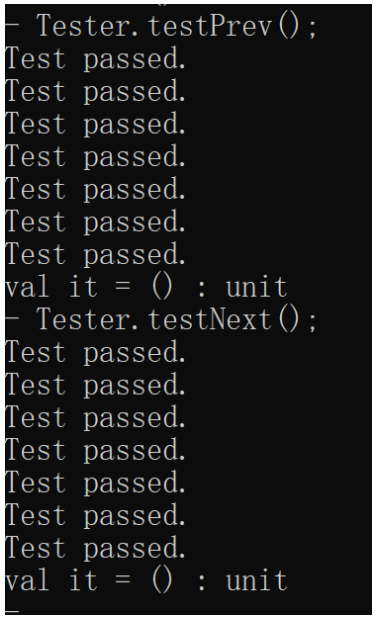
- 关于测试：

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：

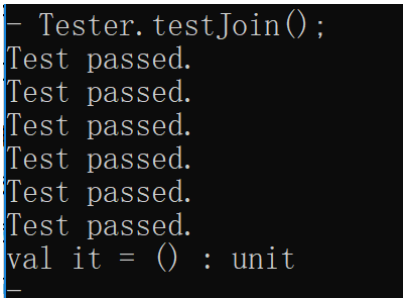
```
fun join (L : 'a table, R : 'a table) : 'a table =
    Tree.join (L, R)
```

- 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：

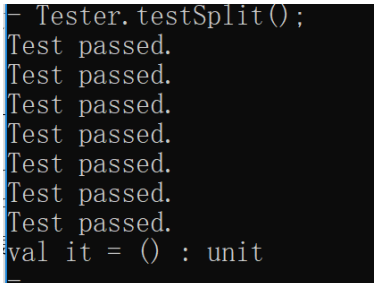
```
fun split (T : 'a table, k : key) : 'a table * 'a option * 'a table =
        Tree.splitAt(T,k)
```

- 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：

```
- Tester.testRange();
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
val it = () : unit
-
```

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(n^2)。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(n^2)。

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：

```
- Tester.testCount();
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
Test passed.
val it = () : unit
```