Estructura

Para resolver este ejercicio hay que tener en cuenta dos cosas:

- a lo sumo puede haber n puntajes diferentes y n posiciónes diferentes, una para cada equipo
- por cómo definimos las posiciones, cuando se registra un partido a lo sumo van a haber cambios entre la posición actual del ganador y la nueva posición, es decir, una cantidad constante de cambios

Teniendo eso en cuenta podemos usar la siguiente estructura

```
Equipo es int
Puntaje es int
Posición es int

Modulo TorneoDeFútbol implementa Torneo {
    puntajePorEquipo:   DiccionarioAVL<Equipo, Puntaje>
    posiciónPorPuntaje: DiccionarioAVL<Puntaje, struct<posición: Posición, cantidad: Int>>
}
```

Donde para obtener el puntaje de un equipo lo buscamos en el `puntajePorEquipo`y para obtener su posición buscamos primero el puntaje y con eso obtenemos la posición en en `posiciónPorPuntaje`.

Invariante de representación y función de abstracción

Para el invariante necesitamos:

- Todos los puntajes tienen que ser ≥ 0 (las posiciones son positivas por la condición que aparece más abajo)
- Para todo puntaje en las claves de `posiciónPorPuntaje`tiene que haber al menos un equipo en las claves de `puntajePorEquipo` tenga como significado ese puntaje
- Para todo puntaje en las claves de `posiciónPorPuntaje`su significado `<posicion, cantidad>` cumple que
  - La `cantidad` es la cantidad de equipos en `puntajePorEquipo`que tienen ese puntaje
  - La `posición` es la que corresponde al puntaje en relación a los demás puntajes

```
InvRep(t: TorneoDeFútbol) {
  forall p: Puntaje :: p in t.posiciónPorPuntaje.data ==>L (
      p >= 0
      &&
      (exists e: Equipo :: e in t.puntajePorEquipo.data &&L t.puntajePorEquipo.data[e] = p)
      &&
      (t.posiciónPorPuntaje.data[p].cantidad =
        sum e: Equipo ::
          e in t.puntajePorEquipo.data ::
          if t.puntajePorEquipo.data[e] = p then 1 else 0)
      &&
      (t.posiciónPorPuntaje.data[p].posicion = 1 +
        sum p': Puntaje ::
          p' in t.posicionPorPuntaje.data ::
          if p' > p then t.posicionPorPuntaje.data[p'].cantidad else 0)
  )
}
```

Para la función de abstracción necesitamos:

- Todos los equipos del observador son claves de `puntajePorEquipo`
- El puntaje de cade equipo en `puntajePorEquipo`es la suma de los partidos ganados en el observador

```
FuncAbs(t': TorneoDeFútbol): Torneo {
  t: Torneo |
    forall e: Equipo :: e in t.equipos <==>
    e in t'.puntajePorEquipo.data &&
      t'.puntajePorEquipo.data[e] =
        sum 0 <= i < |t.partidos| :: if t.partidos[i].ganador = e then 1 else 0
}
```

Algoritmos

```
proc nuevoTorneo(out t: Torneo)
  complejidad: O(1)
{
  t.puntajePorEquipo := nuevo DiccionarioAVL<Equipo, Puntaje>()                // O(1)
  t.posiciónPorPuntaje := nuevo DiccionarioAVL<Puntaje, Tupla<Posición, Int>>  // O(1)
}

proc puntos(in t: Torneo, in e: Equipo, out p: Puntaje)
  requiere e in t.puntajePorEquipo
  complejidad: O(log n)
{
  p := t.puntajePorEquipo.obtener(e)                              // O(log n) (buscar en dicc AVL)
}

proc posición(in t: Torneo, in e: Equipo, out p: Posición)
  requiere e in t.puntajePorEquipo
  complejidad: O(log n)
{
  Puntaje puntaje := t.puntajePorEquipo.obtener(e)                // O(log n) (buscar en dicc AVL)
  p := t.posiciónPorPuntaje.obtener(puntaje).posición             // O(log n) (ídem)
}

proc registrarPartido(inout t: Torneo, in ganador: Equipo, in perdedor: Equipo)
  complejidad: O(log n)
{
  if (ganador in t.puntajePorEquipo)                              // O(log n) (buscar en dicc AVL)
  then
      // Ganó un equipo existente => Acomodar los puntajes y posiciónes
      acomodarPosiciónes(t, ganador)                              // O(log n) (ver más abajo)
  else
      // Ganó un equipo nuevo => se agrega con un punto y la
          posición que corresponda a los equipos de 1 punto
      t.puntajePorEquipo.definir(ganador, 1)                      // O(log n) (definir en dicc AVL)
      agregarEquipoConUnPunto(t)                                  // O(log n) (ver más abajo)
  fi

  // Si no estaba el perdedor hay que agregarlo también, último y con 0 puntos
  if (perdedor not in t.puntajePorEquipo)
      t.puntajePorEquipo.definir(perdedor, 0)                     // O(log n) (definir en un dicc AVL)
      agregarEquipoConCeroPuntos(t)                               // O(log n) (ver más abajo)
  fi
}
```

Auxiliares

```
proc acomodarPosiciones(inout t: Torneo, in ganador: Equipo)
  requiere ganador in t.puntajePorEquipo
{
  Puntaje pje_anterior := t.puntajePorEquipo.obtener(ganador)    // O(log n)
  Tupla<Posición, int> pos_anterior
      := t.posiciónPorPuntaje.obtener(pje_anterior)              // O(log n)

  t.puntajePorEquipo.definir(ganador,                            // O(log n)
      t.puntajePorEquipo.obtener(ganador) + 1)

  // Opción 1: el equipo estaba solo y había alguno a un punto de diferencia
  // Opción 2: el equipo estaba solo y hay más de un punto de diferencia con el siguiente
  // Opción 3: el equipo estaba con alguno y había alguno a un punto de diferencia
  // Opción 4: el equipo estaba con alguno y hay más de un punto de diferencia con el siguiente

  if (pos_anterior.cantidad = 1)                                 // O(1)
  then
      if (pje_anterior.posición + 1 in t.posiciónPorPuntaje)     // O(log n)
      then
          var pos_nueva :=
            t.posiciónPorPuntaje.obtener(pje_anterior + 1)       // O(log n)
          pos_nueva.cantidad := pos_nueva.cantidad + 1           // Asumo aliasing para modificar
                                                                 //  los structs sin reasignar

          t.posiciónPorPuntaje.borrar(pje_anterior)              // O(log n)
      else
          t.posiciónPorPuntaje.definir(
            pje_anterior + 1, (pos_anterior.posición, 1)         // O(log n)
          )
          t.posiciónPorPuntaje.borrar(pje_anterior)              // O(log n)
      fi
  else
      if (pos_anterior.posición + 1 in t.posiciónPorPuntaje)     // O(log n)
      then
          var pos_nueva :=
            t.posiciónPorPuntaje.obtener(pje_anterior + 1)       // O(log n)
          pos_nueva.cantidad := pos_nueva.cantidad + 1

          var pos_anterior :=
            t.posiciónPorPuntaje.obtener(pje_anterior)           // O(log n)
          pos_anterior.posición := pos_anterior.posición - 1
          pos_anterior.cantidad := pos_anterior.cantidad - 1
      else
          t.posiciónPorPuntaje.definir(
            pje_anterior + 1,(pos_anterior.posición, 1)          // O(log n)
          )

          var pos_anterior :=
            t.posiciónPorPuntaje.obtener(pje_anterior)           // O(log n)
          pos_anterior.posición := pos_anterior.posición - 1
          pos_anterior.cantidad := pos_anterior.cantidad - 1
      fi
  fi
}
```

```
proc agregarEquipoConUnPunto(inout t: Torneo)
{
  if (1 in t.posiciónPorPuntaje)                                  // O(log n)
  then
      var pos_1 := t.posiciónPorPuntaje.obtener(1)                // O(log n)
      pos_1.cantidad := pos_1.cantidad + 1
  else
      if (0 in t.posiciónPorPuntaje)
      then
          // Hay alguno con 0 puntos, así que el que agrego va con esa
          //  posición y el o los que estaba(n) con 0 bajan una
          var pos_0 := t.posiciónPorPuntaje.obtener(0)
          t.posiciónPorPuntaje.definir(1,
              (pos_0.posicion, 1)                                 // O(log n)
          )
          pos_0.posicion := pos_0.posicion + 1
      else
          // Si no hay ninguno con 0 punto es el último
          t.posiciónPorPuntaje.definir(1,
            (t.puntajePorEquipos.tamaño, 1)                       // O(log n)
          )
      fi
  fi
}

proc agregarEquipoConCeroPuntos(inout t: Torneo)
{
  if (0 in t.posiciónPorPuntaje)                                  // O(log n)
  then
      var pos_0 := t.posiciónPorPuntaje.obtener(0)                // O(log n)
      pos_0.cantidad := pos_0.cantidad + 1
  else
      // Si no hay ninguno con 0 punto es el último
  t.posiciónPorPuntaje.definir(0,
        (t.puntajePorEquipos.tamaño, 1)                          // O(log n)
      )
  fi
}
```