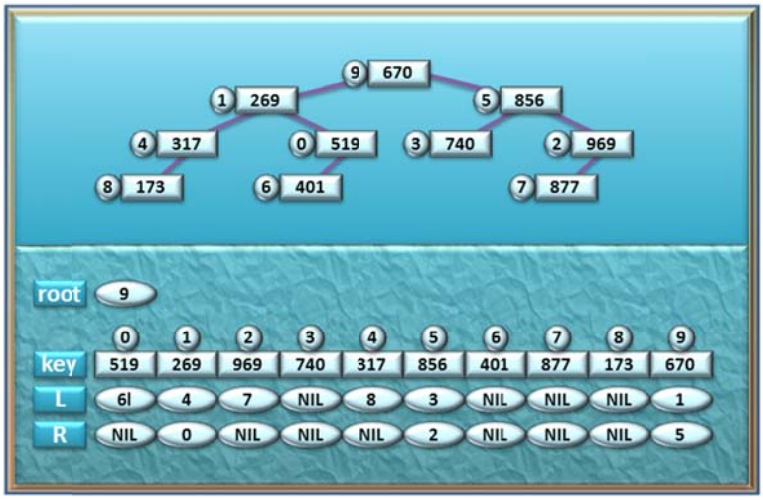
ALG-BVS a AVL stromy - komentář a řešení

Úloha nevžaduje žádnou zvláštní manipulaci se stromy nebo jejich uzly, kromě jediné neustále opakované operace Insert, proto bude vhodné volit reprezentaci pokud možno úsporně.

Nejprve ukáže me řešení pro redukovanou úlohu, kdybychom měli měřit pouze obyčejný BVS. Přechod k AVL pak bude jednoduchým (byť uvážlivým) rozšířením.



Obr. 1. Příklad jednoduché reprezentace BVS.

Dopředu víme, kolik bude uzlů, a proto si je očíslujeme 0,1, ..., M+N–1. Hodnoty klíčů můžeme uložit do jednorozměrného pole **key** indexovaného právě čísly uzlů. Protože samotné uzly jako takové (jako objekty) vůbec nebudeme potřebovat, může celý uzel představovat jediná položka pole **key**. Reference na pravého a levého potomka uzlu můžeme pak opět uložit do dalších dvou 1D polí **L** a **R**, které budou indexovány stejně a budou obsahovat indexy levého a pravého potomka každého uzlu. S těmito datovými strukturami jsme již vybaveni pro celou úlohu.

Vkládání nového uzlu do stromu provedeme funkcí Insert, jež je rekurzivní, kód je kratší a riziko přeplnění zásobníku nevelké. Přesun všech uzlů ze stromu B do A zařídí kratičká rovněž rekurzivní funkce, jiným způsobem ji ani nemá cenu psát. S další funkcí pro vybudování stromů opakovaným voláním funkce Insert a funkcí pro načtení a inicializaci dat a datových struktur jsme prakticky hotovi, uvedený kód snad nepotřebuje další zdlouhavá vysvětlení. Jako obvykle je nejdelší funkce načítající data.

```
static int M, N;
static int [] key, L, R;              // node keys Left and Right children
static int NIL = -1;                  // null pointer
static int cost;                      // to be calculated
static int rootA, rootB;              // tree roots

public static void main(String[] args) throws IOException {
  readAndInitAll();
  makeBSTtrees();
  transferTree(rootB);
  System.out.printf("%d\n", cost);
}

static void makeBSTtrees(){
  L[rootA] = R[rootA] = L[rootB] = R[rootB] = NIL;
  cost = 2;                           // cost of the roots
  for(int iNode = 1; iNode < M; iNode++)
     Insert(rootA, iNode);
  for(int iNode = M+1; iNode < M+N; iNode++)
     Insert(rootB, iNode);
}

static void transferTree(int iNode) {
  if (iNode == NIL) return ;
  transferTree(L[iNode]);
  transferTree(R[iNode]);
  Insert(rootA, iNode);
}
```

```
static int Insert(int iParent, int iNode){
  cost++;
  int [] nextBranch =  (key[iNode] < key[iParent] ) ? L : R;
  if (nextBranch[iParent] != NIL)
    return Insert(nextBranch[iParent], iNode) ;
  nextBranch[iParent] = iNode;
  L[iNode] = R[iNode] = NIL;
  cost++;
  return iNode;
  }

static void readAndInitAll() throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  M = Integer.valueOf(st.nextToken());
  N = Integer.valueOf(st.nextToken());
  // initialize all structures
  rootA = 0;
  rootB = M;
  L = new int [M+N];
  R = new int [M+N];
  for(int i = 0; i < M+N; i++)
     L[i] = R[i]  = NIL;
  key = new int [M+N];
  key[0] = 519217;
  for(int i = 1; i < key.length; i++)
    key[i] = (int) (( 754043 *(long) key[i-1] + 500009) % 1000003);
}
```

Při přechodu k AVL stromu zachováme strukturu předchozího programu, pouze operaci Insert je nutno vhodně rozšířit o postup zpět z vloženého uzlu ke kořeni a případnou rotaci. U rotace je nutno dát pozor, může proběhnout i v kořeni a pak se změní kořen celého stromu.
Rotace v AVL stromu jsou teoreticky čtyři: L, R, LR a RL. Každá ze dvojitých rotací RL a LR se skládá ze dvou jednoduchých rotací, takže dvojité rotace nemusíme explicitně programovat. Rotace R je ale strukturně zcela totožná s rotací L, až na to, že je jejím zrcadlovým obrazem. Kdybychom si nakreslili strom na skleněné dveře, tak by nám stačilo umět provádět jen např. levou rotaci, protože při potřebě pravé rotace bychom jen obešli dveře z druhé strany a z tohoto nového pohledu bychom udělali levou rotaci. V rotacích nehrají roli klíče uzlů, tedy ani pořadí klíčů ve stromu, a proto tento náš malý trik je bude strom upravovat korektně. V implementaci podobného efektu dosáhneme ještě snáze. Stačí soustavně prohodit ukazatele (reference) na levé a pravé potomky uzlů a spolu s nimi ukazatele (reference) na výšky levých a pravých podstromů registrované v AVL.
Dejme tomu, že naprogramujeme pouze rotaci L. Abychom ji mohli snadno používat, budeme předpokládat, že postupujeme stromem nahoru stále zprava doleva. Tak zajistíme, že i do rozváženého uzlu příjdeme vždy zprava zdola, a tím bude i dáno, že jediné rotace, které nás mohou čekat, budou vždy jednoduchá L rotace nebo dvojitá RL rotace. I neustálý postup zprava doleva nahoru je snadné zařídit, prostě vždy při přechodu z uzlu do jeho rodiče zjistíme, zda je uzel levým nebo pravým potomkem tohoto rodiče. Pokud je pravým potomkem, postoupíme rovnou vzhůru (zprava doleva), a pokud je levým potomkem, obejdeme pomyslné dveře, resp. prohodíme reference na pravé a levé potomky a opět postoupíme zprava doleva nahoru.

Funkce pro rotaci a funkce pro postup vzhůru stromem ovšem musí pracovat s kopiemi (jsou to formální pole Lptr, Rptr, LHptr, RHptr v implementaci) původních ukazatelů (referencí) na skutečná data, protože původní ukazatele (reference) nesmí být ovlivněny, pomocí nich je nutno přistupovat k dané struktuře stromu i v jiných okamžicích než při postupu vzhůru nebo při rotacích.

```
static int M, N;
static int [] key;              // each node contains a single key
static int [] L, R, P;          // Left & Right children, Parent
static int [] LH, RH;           // Left and Right heights

static int [] Lptr, Rptr;       // pointers to L, R in flipped tree
static int [] LHptr, RHptr;     // pointers to LH, RH in flipped tree
static int [] nextBranch;       // auxiliary  reference, array not allocated
static int [] tmp;              // auxiliary  reference, array not allocated
static int NIL = -1;
static int rootA;
static int rootB;
static int currRoot;            // determines the current tree
static int cost;                // to be calculated

public static void main(String[] args) throws IOException {
   readAndInitAll();
  // perform actions for simple BST tree
  rootA = 0;  rootB = M;
  makeTreesBSTorAVL(false);
  transferSubTreeBSTorAVL(rootB, false);    // false == not AVL
  System.out.printf("%d", cost);            // BST cost

  // perform actions for AVL tree
  cost = 0;
  makeTreesBSTorAVL(true);
  transferSubTreeBSTorAVL(rootB, true);     // true == is AVL
  System.out.printf(" %d\n", cost);         //  AVL cost
}

static void makeTreesBSTorAVL(boolean isAVL ){
  L[rootA] = R[rootA] = L[rootB] = R[rootB];            // init
  if (isAVL)
    P[rootA] = LH[rootA] = RH[rootA] = P[rootB] = LH[rootB] = RH[rootB] = NIL;
  cost = 2;             // cost of the roots, which are not inserted
  if (isAVL) cost += 2;
  // insert all other nodes one by one
  for(int iNode = 1; iNode < M; iNode++)
     InsertBSTorAVL(rootA, iNode, isAVL);
  for(int iNode = M+1; iNode < M+N; iNode++)
     InsertBSTorAVL(rootB, iNode, isAVL);
}

static void transferSubTreeBSTorAVL(int iNode, boolean isAVL) { // standard postOrder
  if (iNode == NIL) return ;
  transferSubTreeBSTorAVL(L[iNode], isAVL);
  transferSubTreeBSTorAVL(R[iNode], isAVL);
  InsertBSTorAVL(rootA, iNode, isAVL);
}

static void InsertBSTorAVL(int iRoot, int iNode, boolean isAVL) {
  iNode = InsertBST(iRoot, iNode);
  if (!isAVL ) return;        // all done for usual BST

  currRoot = iRoot;           // AVL rotation might change the root, so remember it
  LH[iNode] = RH[iNode] = -1;
```

```
  // go up the tree
  Lptr = L; Rptr = R; LHptr = LH; RHptr = RH;  // set ptrs which can flip the tree
  int iParent = P[iNode];
  cost += 1;                                   // cost of the leaf on the way up
  if (iNode == L[iParent])
    swapLRptrs();                     // make sure iNode is right child of parent
  goUpLeftInAVL(iNode, iParent);      // to ascend up and to the left
}

static int InsertBST(int iRoot, int iNode){
  nextBranch =  (key[iNode] < key[iRoot] ) ? L : R;
  cost++;               // count nodes during descent
  if (nextBranch[iRoot] != NIL)
    return InsertBST(nextBranch[iRoot], iNode) ;
  nextBranch[iRoot] = iNode;
  L[iNode] = R[iNode] = NIL;
  P[iNode] = iRoot;    // unnecessary in bare BST :-)
  cost++;              // count the leaf too
  return iNode;
  }

static void goUpLeftInAVL(int iNode, int iParent) { // recursively node by node
  cost++;                                    // count nodes during ascent
  RHptr[iParent] = 1+ Math.max(LHptr[iNode], RHptr[iNode]); // recalc balance
  if (LHptr[iParent]+1 >=  RHptr[iParent] ) {
    // no rotation required, go up
     iNode = iParent;
     iParent = P[iParent];
     if (iParent == NIL)
        return;                         // root reached
     if (iNode == Lptr[iParent])
        swapLRptrs();                   // make sure iNode is right child of parent
     goUpLeftInAVL(iNode, iParent);     // to ascend up and to the left
  }
 else  // else must rotate and stop
  if (LHptr[iNode] < RHptr[iNode])
      Lrot(iParent, iNode);
    else {
      swapLRptrs(); Lrot(iNode, Rptr[iNode]);     // flip tree to get R rotation
      swapLRptrs(); Lrot(iParent, Rptr[iParent]); // flip it back to get L rotation
    }                                             // and stop ascent
}


static void swapLRptrs(){           // flips the tree effectively: left <-> right
  tmp = Lptr; Lptr = Rptr; Rptr = tmp;
  tmp = LHptr; LHptr = RHptr; RHptr = tmp;
}


static void Lrot(int iNode, int iLchild) {  // right rotation never used
   int origParent = P[iNode];
   //move right subtree of the Child to the oposite side
   Rptr[iNode] = Lptr[iLchild];
   if (Lptr[iLchild] != NIL)
     P[Lptr[iLchild]] = iNode;
   // iChild becomes subtree root
   Lptr[iLchild] = iNode;
   P[iNode] = iLchild;
```

```
  // link the new subtree root with the parent
   if (origParent == NIL) {
     if (currRoot == rootA)
       rootA = currRoot = iLchild;
     else
       rootB = currRoot = iLchild;
   }
   else {
     if  (Lptr[origParent] == iNode)
       Lptr[origParent] = iLchild;
     else
       Rptr[origParent] = iLchild;
   }
   P[iLchild] = origParent;
   // recalc depths;
   RHptr[iNode] = LHptr[iLchild];
   LHptr[iLchild] = 1+Math.max(RHptr[iNode], LHptr[iNode]);
 }

static void readAndInitAll() throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  M = Integer.valueOf(st.nextToken());
  N = Integer.valueOf(st.nextToken());

  // initialize all structures
  L = new int [M+N];
  R = new int [M+N];
  P = new int [M+N];
  LH = new int [M+N];
  RH = new int [M+N];
  for(int i = 0; i < M+N; i++)
     L[i] = R[i] = P[i] = LH[i] = RH[i] = NIL;
  key = new int [M+N];
  key[0] = 519217;
  for(int i = 1; i < key.length; i++)
    key[i] = (int) (( 754043 *(long) key[i-1] + 500009) % 1000003);
}
```