Алгоритмы работы с АВЛ-деревьями

Каждая внутренняя вершина АВЛ-дерева представлена узлом, состоящим из полей *left, right, key* и *bal*, где *left* и *right* содержат указатели на левого и правого сыновей соответственно, поле *key* содержит имя (ключ), хранящееся в узле, поле *bal* – баланс узла (вершины). Баланс некоторой вершины дерева определяется как разность высот левого и правого поддеревьев, т.е. $h_L - h_R$, где h_L и h_R – высоты левого и правого поддеревьев соответственно. Поскольку в АВЛ-деревьях высоты поддеревьев каждой вершины отличаются не более чем на единицу, баланс вершины может иметь только три значения: 1, 0 или –1 в зависимости от того, что высота ее левого поддерева больше, равна или меньше высоты правого поддерева. Доступ к дереву обеспечивается с помощью внешнего указателя *root* на его корень. Очевидно, что для пустого дерева *root* = Λ. Если указатель (*left* или *right*) имеет значение Λ, это означает, что у узла нет соответствующего сына, т. е. он указывает на пустое поддерево.

Все ссылки на рисунки, поясняющие приведенные алгоритмы, даны на учебное пособие: *Павлов, Л.А*. Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2008. – 252 с.

1. Вращения для балансировки.

Для удобства указатели на узлы будем обозначать теми же, но строчными, буквами, что и ключи (имена), хранящиеся в узле. Например, если узел содержит ключ *A*, то указатель на этот узел обозначается *a*. Для обращения к полю узла будем использовать составное имя, т.е. имя указателя, точка и имя поля (как это реализовано во многих языках программирования). Например, запись *a.left* означает обращение к полю *left* узла *a* (т.е. узла, на который ссылается указатель *a*).

Процедура правого вращения *RightRot* (см. рис. 4.7). Входные параметры: *root* – указатель на корень дерева, *b* – указатель на узел *B*, вокруг которого выполняется вращение, и *f* – указатель на отца узла *B* (если узел *B* является корнем дерева, т.е. *root = b*, то у *B* нет отца и *f* = **nil**). Процедура левого вращения *LeftRot* симметрична процедуре *RightRot*.

```
procedure RightRot(root,b,f);
begin
  a:=b.left;
  b.left:=a.right  // переподчинение поддерева β узлу B
  a.right:=b;  // переподчинение узла B узлу A
  if b <> root then  // узел B не являлся корнем дерева, у него был отец f
    if a.key < f.key then
      f.left:=a  // узел A – левый сын узла f
    else f.right:=a  // узел A – правый сын узла f
  else root:=a;  // узел B являлся корнем дерева, теперь корень – узел A
end
```

Двойное вращение (см. рис. 4.8) можно реализовать поочередным применением операций вращения (на рис. 4.8 сначала правое вращение вокруг вершины *C*, затем левое вращение вокруг вершины *A*). С точки зрения числа выполняемых операций лучше реализовать собственные процедуры двойного вращения.

Процедура двойного вращения *RightLeftRot*. Входные параметры: *root* – указатель на корень дерева, *a* – указатель на узел *A*, и *f* – указатель на отца узла *A* (если узел *A* является корнем дерева, т.е. *root = a*, то у *A* нет отца и *f* = Λ). Другая процедура двойного вращения *LeftRightRot* симметрична процедуре *RightLeftRot*.

```
procedure RightLeftRot(root,a,f)
begin
  c:=a.right;
  b:=c.left;
  a.right:=b.left;  // переподчинение поддерева β узлу A
  c.left:=b.right;  // переподчинение поддерева γ узлу C
  b.left:=a;  // переподчинение узла A узлу B
  b.right:=c;  // переподчинение узла C узлу B
  if a <> root then  // узел A не являлся корнем дерева, у него был отец f
    if b.key < f.key then
      f.left:=b  // узел B – левый сын узла f
    else f.right:=b  // узел B – правый сын узла f
  else root:=b  // узел A являлся корнем дерева, теперь корень – узел B
end
```

2. Поиск в дереве бинарного поиска

Входные параметры: *T* – указатель на корень дерева, *z* –ключ поиска. Возвращает указатель на найденный узел или **nil** при безуспешном поиске. Функция использует глобальный стек *S* для сохранения пути поиска (т.е. в стеке сохраняются указатели пройденных в процессе поиска узлов). Стек нужен только при выполнении операций включения и исключения узлов для реализации прохода в обратном направлении. Поэтому для «чистого» поиска его можно убрать. При выполнении экспериментальных оценок в функцию следует добавить счетчик для подсчета числа интересующих операций.

```
function SearchBST(T,z):TPointerTree;
begin
  StNull(S); // сделать стек S пустым
  p:=T; b:=false;
  while (p <> nil)and not b do
  begin
    Push(S,p); // запоминаем путь в стеке
    if z < p.key then p:=p.left
    else
      if z > p.key then p:=p.right
      else b:=true; // узел найден
  end;
  if b then // успешный поиск
    SearchBST:=p;
  else // безуспешный поиск
    SearchBST:=nil;
end;
```

3. Включение новой вершины

Входные параметры: *T* – указатель на корень дерева, *z* – добавляемый ключ. Путь, пройденный в процессе поиска, хранится в стеке *S*. Операции со стеком: *StNull* (*S*) – сделать стек пустым; *Push* (*S*, *p*) – добавить в стек *S* элемент *p*, *Pop* (*S*) исключает элемент из вершины стека и возвращает его значение; *Top* (*S*) возвращает значение элемента из вершины стека без его исключения; *Empty* (*S*) – проверка пустоты стека (**true**, если стек пуст). Оператор Break реализует досрочный выход из текущего цикла. Переменная *Grow* с возможными значениями *grLeft* и *grRight* указывает, какое поддерево увеличило свою высоту

(можно сделать типа Boolean, **false** – рост высоты левого поддерева, **true** – правого поддерева, или наоборот). Процедура вызывается только при безуспешном поиске, т.е. должен быть оператор типа **if** *SearchBST* (*T*, *z*) = **nil then** *AddNodeAVL* (*T*, *z*)

```
procedure AddNodeAVL(T, z)
begin
  new(p); p.key:=z;
  p.left:=nil; p.right:=nil; p.bal:=0;
  if T <> nil then // добавление не в пустое дерево
  begin
    f:=Top(S); // f – отец узла p, находится в вершине стека
    if z < f.key then f.left:=p else f.right:=p;
    // движение вверх, проверка баланса и, при необходимости, балансировка
    while not Empty(S) do
    begin
      q:=p; // q нужен для определения, какое п/д выросло
      p:=Pop(S);
      if p.left = q then Grow:=grLeft else Grow:=grRight;
      if Grow = grLeft then // рост левого п/д
      begin
        case p.bal of // проверка баланса текущего узла
          0:p.bal:=1;
         -1:begin
              p.bal:=0;
              Break; // прекращаем продвижение вверх
            end;
          1:begin // левое п/д выше, требуется балансировка
              if p = T then // p – корень дерева, отца нет
                f:=nil
              else f:=Top(S); // f – отец узла p, находится в вершине стека
              if q.bal = -1 then
              // два последних шага в разных направлениях
              begin // двойное LR-вращение
                r:=q.right;
                LeftRightRot(T,f,p);
                if r.bal = 1 then
                begin
                  p.bal:=-1;
                  q.bal:=0;
                end else
                begin
                  p.bal:=0;
                  if r.bal = 0 then q.bal:=0 else q.bal:=1;
                end;
                r.bal:=0;
              end else
              begin
                RightRot(T,f,p); // правое вращение
                p.bal:=0; q.bal:=0;
              end;
              Break; // прекращаем продвижение вверх
            end;
```

```
        end;
      end else // рост правого п/д
      begin
        case p.bal of // проверка баланса текущего узла
          0:p.bal:=-1;
          1:begin
              p.bal:=0;
              Break;
            end;// прекращаем продвижение вверх
         -1:begin // правое п/д выше, требуется балансировка
              if p = T then // p – корень дерева, отца нет
                f:=nil
              else f:=Top(S); // f – отец узла p
              if q.bal = 1 then
              // два последних шага в разных направлениях
              begin // двойное RL-вращение
                r:=q.left;
                RightLeftRot(T,f,p);
                if r.bal = 1 then
                begin
                  p.bal:=0;
                  q.bal:=-1;
                end else
                begin
                  if r.bal = 0 then p.bal:=0 else p.bal:=1;
                  q.bal:=0;
                end;
                r.bal:=0;
              end else
              begin
                LeftRot(T,f,p); // левое вращение
                p.bal:=0; q.bal:=0;
              end;
              Break; // прекращаем продвижение вверх
            end;
        end;
      end;
    end;
  end else T:=p; // добавление в пустое дерево
  // очистка стека
  while not Empty(S) do Pop(S);
end;
```

4. Исключение вершины

Входные параметры: *T* – указатель на корень дерева, *p* – указатель на исключаемый узел. Путь, пройденный в процессе поиска, хранится в стеке *S*. Операции со стеком: *StNull* (*S*) – сделать стек пустым; *Push* (*S*, *p*) – добавить в стек *S* элемент *p*, *Pop* (*S*) исключает элемент из вершины стека и возвращает его значение; *Top* (*S*) возвращает значение элемента из вершины стека без его исключения; *Empty* (*S*) – проверка пустоты стека (**true**, если стек пуст). Оператор Break реализует досрочный выход из текущего цикла. Переменная *Reduce* с возможными значениями *reLeft* и *reRight* указывает, какое поддерево умень-

шило свою высоту (можно сделать типа Boolean, **false** – уменьшение высоты левого поддерева, **true** – правого поддерева, или наоборот). Процедура вызывается только при успешном поиске, т.е. должна быть последовательность операторов вида

```
p:= SearchBST(T,z);
if p <> nil then DelNodeAVL(T,p)
```

```
procedure DelNodeAVL(T,p);
begin
  if (p.left <> nil)and(p.right <> nil) then // 2 сына
  begin
    q:=p; // сохраняем ссылку на узел с удаляемым ключом
    // поиск предшественника
    p:=p.left;
    Push(S,p); // продолжение сохранения пути
    while p.right <> nil do
    begin
      p:=p.right;
      Push(S,p); // продолжение сохранения пути
    end;
    // p – предшественник
    q.key:=p.key; // заменяем исключаемый ключ его предшественником
  end;
  // удаление узла p
  if p.left <> nil then q:=p.left else q:=p.right;
  if p <> T then // p – не корень дерева
  begin
    Pop(S); // удаление из стека узла с исключаемым ключом
    f:=Top(S); // f – отец узла p
    if p.key <= f.key then
    // здесь p.key <= f.key, т.к. после q.key:=p.key может быть p.key = f.key
    begin
     f.left:=q;
     Reduce:=reLeft; // укоротилось левое п/д
    end else
    begin
      f.right:=q;
      Reduce:=reRight; // укоротилось правое п/д
    end;
  end else T:=q; // p – корень дерева, отца нет
  dispose(p);
  // движение вверх, проверка баланса и, при необходимости, балансировка
  while not Empty(S) do
  begin
    p:=Pop(S);
    if Reduce = reLeft then // укоротилось левое п/д
    begin
      case p.bal of
        0:begin
            p.bal:=-1;
            Break; // прекращаем продвижение вверх
          end;
        1:p.bal:=0;
```

```
    -1:begin // требуется балансировка
         if p = T then // p – корень дерева, отца нет
           f:=nil
         else f:=Top(S); // f – отец узла p
         q:=p.right;
         case q.bal of
           0:begin // левое вращение и прекращение движения вверх
               LeftRot(T,f,p);
               p.bal:=-1;
               q.bal:=1;
               Break;
             end;
          -1:begin // левое вращение и движение вверх
               LeftRot(T,f,p);
               p.bal:=0;
               q.bal:=0;
               p:=q; // нужно для определения, какое п/д укоротилось
             end;
           1:begin // двойное RL-вращение и движение вверх
               r:=q.left;
               RightLeftRot(T,f,p);
               case r.bal of
                 0:begin p.bal:=0; q.bal:=0; end;
                 1:begin p.bal:=0; q.bal:=-1; end;
                -1:begin p.bal:=1; q.bal:=0; end;
               end;
               r.bal:=0;
               p:=r; // нужно для определения, какое п/д укоротилось
             end;
         end;
       end;
  end;
end else // укоротилось правое п/д
begin
  case p.bal of
    0:begin
        p.bal:=1;
        Break; // прекращаем продвижение вверх
      end;
    -1:p.bal:=0;
     1:begin // требуется балансировка
         if p = T then // p – корень дерева, отца нет
           f:=nil
         else f:=Top(S); // f – отец узла p
         q:=p.left;
         case q.bal of
           0:begin // правое вращение и прекращение движения вверх
               RightRot(T,f,p);
               p.bal:=1;
               q.bal:=-1;
               Break;
             end;
```

```
                1:begin  // правое вращение и движение вверх
                    RightRot(T,f,p);
                    p.bal:=0;
                    q.bal:=0;
                    p:=q;  // нужно для определения, какое п/д укоротилось
                  end;
               -1:begin  // двойное LR-вращение и движение вверх
                    r:=q.right;
                    LeftRightRot(T,f,p);
                    case r.bal of
                      0:begin p.bal:=0; q.bal:=0; end;
                      1:begin p.bal:=-1; q.bal:=0; end;
                     -1:begin p.bal:=0; q.bal:=1; end;
                    end;
                    r.bal:=0;
                    p:=r;  // нужно для определения, какое п/д укоротилось
                  end;
            end;
          end;
      end;
    end;
    // определение укоротившегося поддерева
    if S <> nil then //в стеке еще есть элементы
      if Top(S).left = p then
        Reduce:=reLeft
      else Reduce:=reRight;
  end;
  // очистка стека
  while not Empty(S) do Pop(S);
end;
```

$$\underline{\textbf{for}}\ i := -(h_1 - 1)\ \underline{\textbf{to}}\ 0\ \underline{\textbf{do}}\ x_i := -\infty \ \langle \text{установка имен-сторожей} \rangle$$

$$\underline{\textbf{for}}\ l := 1\ \underline{\textbf{to}}\ t\ \underline{\textbf{do}} \begin{cases} \langle h_l \text{-сортировка} \rangle \\ k := h_l \\ \underline{\textbf{for}}\ j := k+1\ \underline{\textbf{to}}\ n\ \underline{\textbf{do}} \begin{cases} i := j - k \\ X := x_j \\ \underline{\textbf{while}}\ X < x_i\ \underline{\textbf{do}} \begin{cases} x_{i+k} := x_i \\ i := i - k \end{cases} \\ x_{i+k} := X \end{cases} \end{cases}$$