ЛЕКЦИЯ №5

КРАСНО-ЧЕРНЫЕ ДЕРЕВЬЯ

SPLAY-ДЕРЕВЬЯ

БАЛАНСИРОВКА ДЕРЕВА

Бинарные деревья работают лучше всего, когда они сбалансированы, когда длина пути от корня до любого из листьев находится в определенных пределах, связанных с числом вершин.

КЧ-деревья – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color, обозначающее цвет: красный или черный, и для которых выполнены приведенные на следующем слайде свойства.

```
struct RBNode
{
key_type key;
struct RBNode *left;
struct RBNode *right;
struct RBNode *parent;
char color; // цвет
};
```

Будем считать, что если left или right равны NULL, то это «указатели» на фиктивные листья. Таким образом, все узлы – внутренние (нелистовые).

Красно-черное дерево (*Red-Black-Tree, RB-Tree*) – это бинарное дерево со следующими свойствами:

1. каждый узел либо красный, либо черный;

2. каждый лист (фиктивный) – черный;

3. если узел красный, то оба его сына – черные;

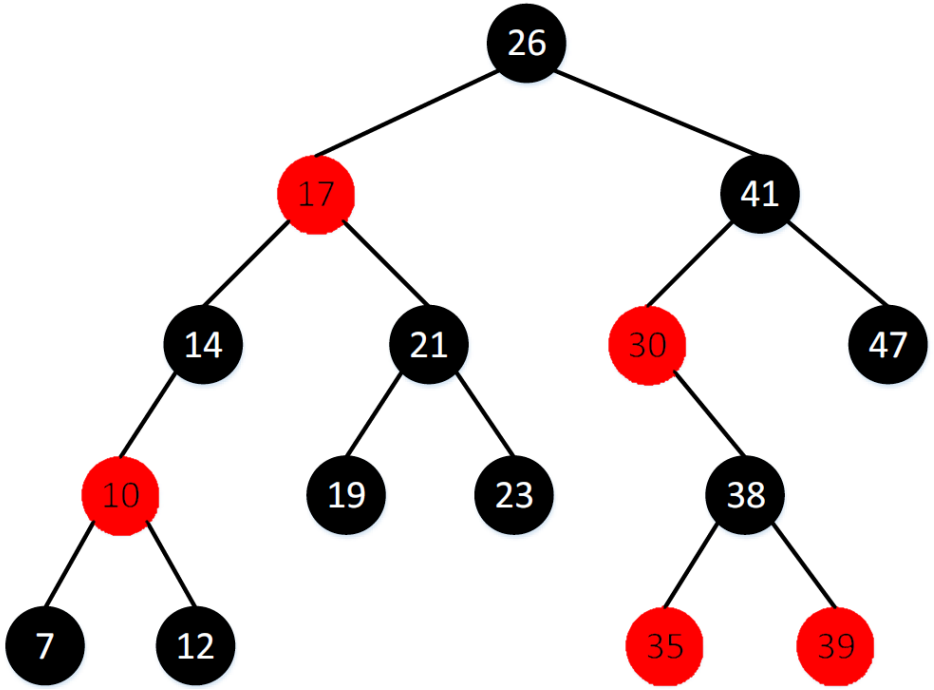
4. все пути, идущие от корня к любому фиктивному листу, содержат одинаковое количество черных узлов;

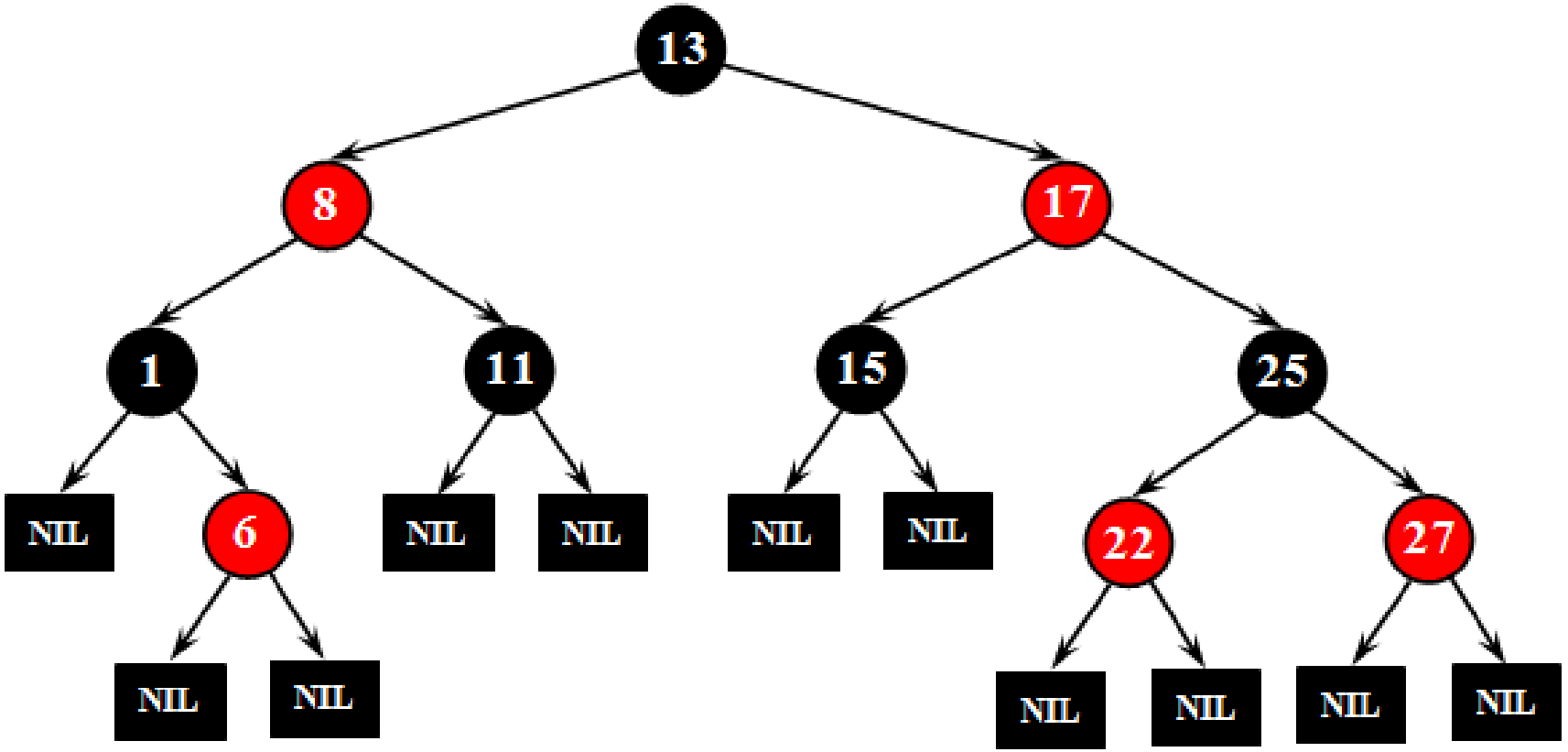
5. корень – черный.

Определение 4: ***Черной высотой узла называется количество черных узлов на пути от этого узла к узлу, у которого оба сына – фиктивные листья.***

Сам узел не включается в это число. Например, у дерева, приведенного на рисунке, черная высота корня равна 2.

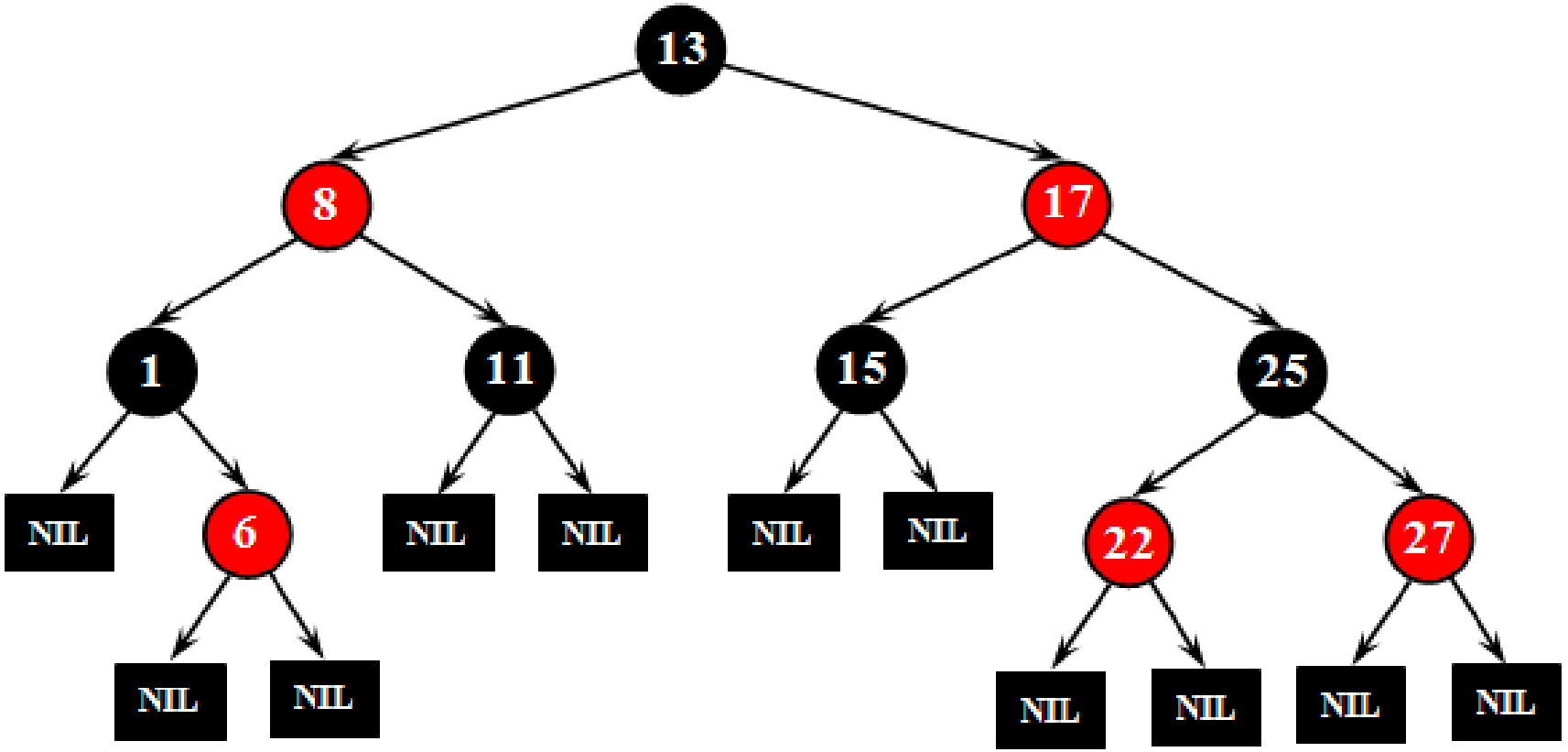
Определение 5: ***Черная высота дерева – черная высота его корня.***




13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL

Иными словами, в такой структуре баланс достигается за счет поддержания раскраски вершин в два цвета (красный и черный, как видно из названия), подчиняющейся следующим правилам:

- **Красная вершина не может быть сыном красной вершины**
- **Черная высота любого листа одинакова (черной высотой называют количество черных вершин на пути из корня)**
- **Корень дерева черный**


13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL

Количество черных вершин на ветви от корня до листа называется *черной высотой дерева*. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу.

Над красно-черными деревьями можно выполнять все те же основные операции, что и над бинарными деревьями.

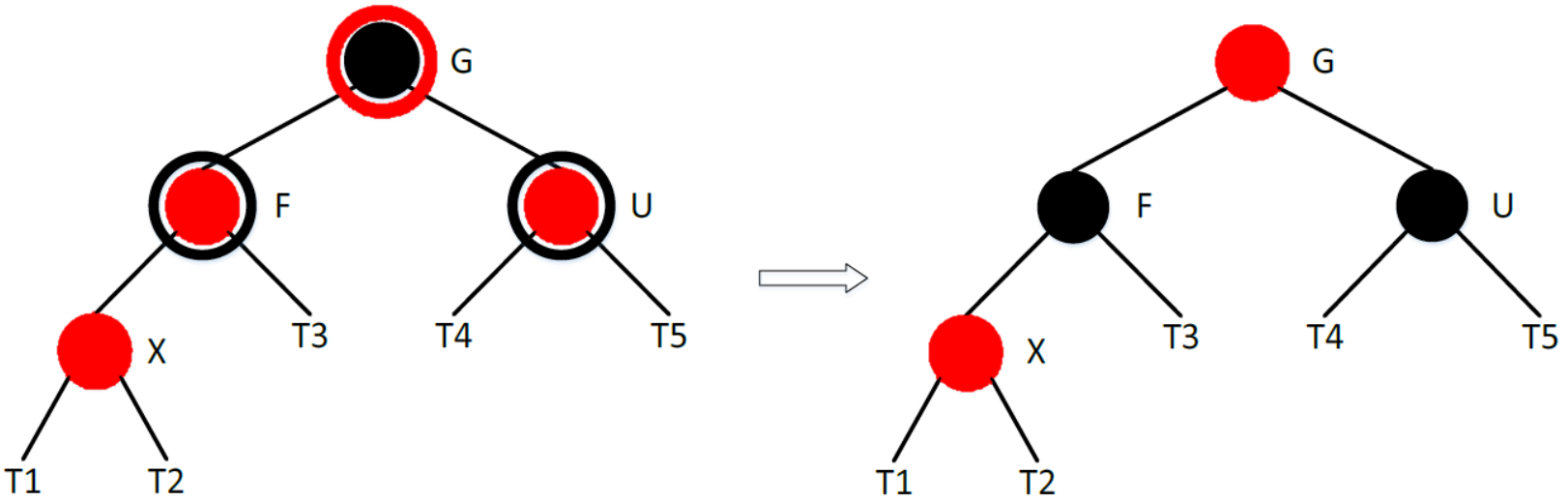
Приведем функции следующих операций над красно-черными деревьями: создание дерева, печать (просмотр) дерева, обход дерева, проверка пустоты дерева и удаление дерева.

```
        //создание красно-черного дерева
void Make_RBTree(RBTree** Node, int n){
  int Data;
  while (n > 0) {
    cout << "Введите значение ";
        cin >> Data;
    Insert_Node(Node, Data);
    n--;
  }
}
```

```
//добавление узла в красно-черное дерево
void Insert_Node(RBTree** Node,int Data) {
  RBTree **Curent, *Parent, *New_Node;
  Curent = Node;
  Parent = NIL;
  // Поиск местоположения
  while (*Curent != NIL) {
    Parent = (*Curent);
    Curent = Data < (*Curent)->Data ? &((*Curent)->Left) :
&((*Curent)->Right);
  }
```

Сначала узел добавляется в дерево с помощью стандартного алгоритма вставки узла в двоичное дерево поиска. Вновь добавленный узел красится в красный цвет. Если это первый узел в дереве, то он становится корнем и перекрашивается в черный цвет. Далее производится проверка, не нарушились ли свойства КЧ-дерева. Если добавленный узел не первый, то он красный, поэтому свойство 4 об одинаковом количестве черных узлов на любом пути от корня к листу, не нарушается. Если родитель нового узла черный, то свой- ство 3 о том, что если узел красный, то оба его сына черные, также не нарушается.

Будем считать, что узел X находится в левом поддереве своего деда (G), иначе все приведенные ниже изображения будут симметричны относительно вертикальной оси, проходящей через деда.



Рассматривается случай, когда узел X и его отец – красные, поэтому дед G узла X – черный, иначе красно-красное нарушение было бы еще до добавления узла X. Только дядя U узла X может иметь в данном случае как черный, так и красный цвет. При этом цепочка узлов X-F-G может образовывать как прямую линию, так и угол. П

```
        // Поиск местоположения
  while (*Curent != NIL) {
    Parent = (*Curent);
    Curent = Data < (*Curent)->Data ? &((*Curent)->Left) :
&((*Curent)->Right);
  }
```

```
// Создание нового узла
New_Node = new RBTree();
New_Node->Data = Data;
New_Node->Parent = Parent;
New_Node->Left = NIL;
New_Node->Right = NIL;
New_Node->color = RED;
```

```
        // Вставка элемента в дерево
  if(Parent != NIL){
    if (Data < Parent->Data) Parent->Left = New_Node;
    else Parent->Right = New_Node;
  }
  else (*Curent) = New_Node;
  Insert_Fixup(Node, New_Node);
}
```

```
// Поддержка баланса дерева после вставки нового элемента
void Insert_Fixup(RBTree** Node,RBTree* New_Node){
  RBTree* Current = New_Node;
```

```
        // Проверка свойств дерева
  while (Current != *(Node) && Current->Parent->color ==
RED){
    // если есть нарушение
    if (Current->Parent == Current->Parent->Parent->Left) {
      RBTree *ptr = Current->Parent->Parent->Right;
      if (ptr->color == RED) {
        Current->Parent->color = BLACK;
        ptr->color = BLACK;
        Current->Parent->Parent->color = RED;
        Current = Current->Parent->Parent;
      }
      else {
        if (Current == Current->Parent->Right) {
```

```
        // сделать Current левым потомком
      Current = Current->Parent;
      Rotate_Left(Node,Current);
    }
    // перекрасить и повернуть
    Current->Parent->color = BLACK;
    Current->Parent->Parent->color = RED;
    Rotate_Right(Node,Current->Parent->Parent);
  }
}
else {
  RBTree *ptr = Current->Parent->Parent->Left;
  if (ptr->color == RED) {
    Current->Parent->color = BLACK;
    ptr->color = BLACK;
    Current->Parent->Parent->color = RED;
    Current = Current->Parent->Parent;
  }
```

```
            else {
          if (Current == Current->Parent->Left) {
            Current = Current->Parent;
            Rotate_Right(Node,Current);
          }
          Current->Parent->color = BLACK;
          Current->Parent->Parent->color = RED;
          Rotate_Left(Node,Current->Parent->Parent);
        }
      }
    }
    (*Node)->color = BLACK;
}
```

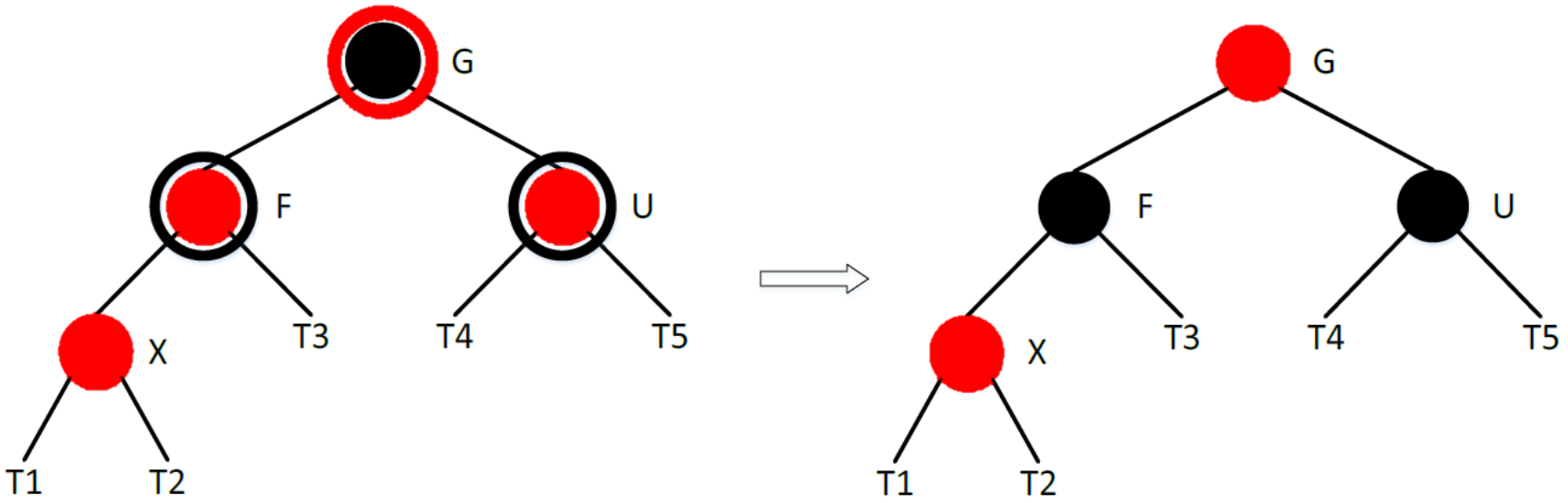
```
        //поворот узла Current влево
void Rotate_Left(RBTree** Node,RBTree *Current) {
  RBTree *ptr = Current->Right;
  Current->Right = ptr->Left;
  if (ptr->Left != NIL) ptr->Left->Parent = Current;
  if (ptr != NIL) ptr->Parent = Current->Parent;
  if (Current->Parent != NIL) {
    if (Current == Current->Parent->Left)
      Current->Parent->Left = ptr;
    else
      Current->Parent->Right = ptr;
    }
  else {
    (*Node) = ptr;
  }
  ptr->Left = Current;
  if (Current != NIL) Current->Parent = ptr;
}
```

```
        //поворот узла Current вправо
void Rotate_Right(RBTree** Node,RBTree *Current) {
  RBTree *ptr = Current->Left;
  Current->Left = ptr->Right;
  if (ptr->Right != NIL) ptr->Right->Parent = Current;
  if (ptr != NIL) ptr->Parent = Current->Parent;
  if (Current->Parent != NIL) {
    if (Current == Current->Parent->Right)
      Current->Parent->Right = ptr;
    else
      Current->Parent->Left = ptr;
  }
  else {
    (*Node) = ptr;
  }
  ptr->Right = Current;
  if (Current != NIL) Current->Parent = ptr;
}
```

1. Дядя U добавляемого узла X красный.

В этом случае достаточно выполнить только перекраску: отца и дядю (F и U) – в черный цвет, деда (G) – в красный цвет. Тогда свойство, что у красного узла оба сына черные, будет выполнено. Свойство, что все пути, идущие от корня к листу, содержат одинаковое количество черных узлов, также не будет нарушено, потому что в каждом из путей два узла просто поменялись цветами (G и F слева, G и U справа). Количество черных узлов в путях при этом не изменилось. Если G – корень, то он просто перекрашивается в черный цвет.

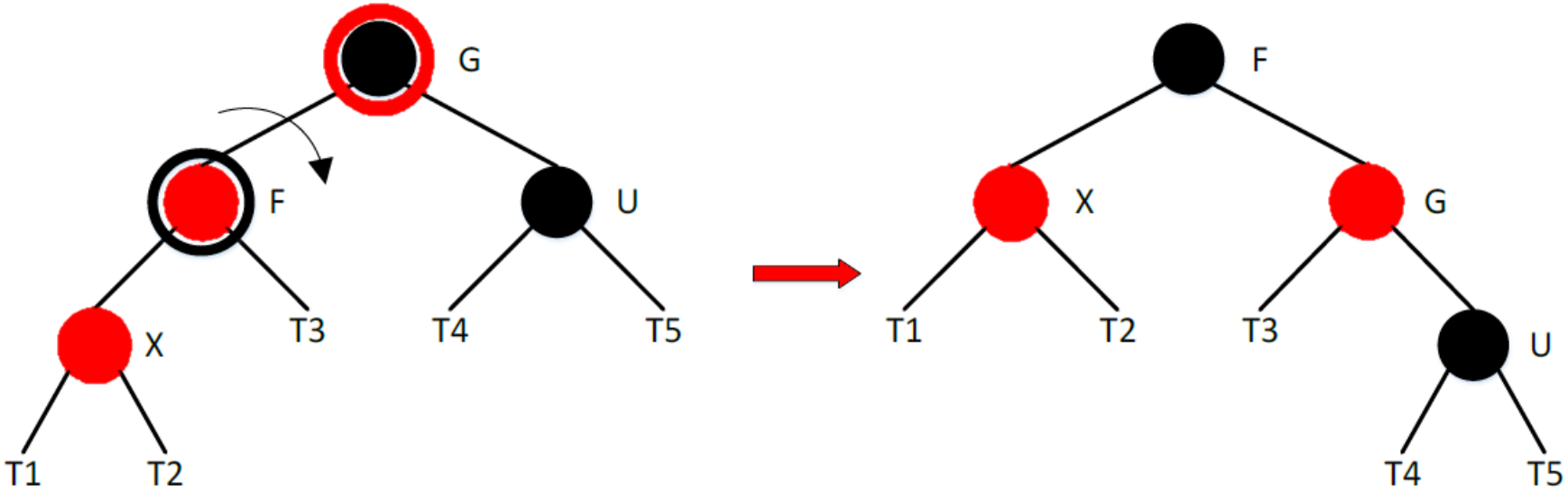
При этом все 5 свойств КЧ-деревьев оказываются выполненными. Если отец узла G тоже красный, то появится новое красно-красное нарушение, и понадобится дальнейшее восстановление свойств КЧ-дерева, только в роли узла X теперь будет вы- ступать узел G. Может иметь место один из 3-х вариантов.



2. Дядя U добавляемого узла X черный и при этом цепочка узлов X-F-G образует прямую линию.

Только перекраски отца F узла X в черный цвет, а деда G в красный будет недостаточно для восстановления свойств КЧ-дерева, потому что количество черных узлов в путях, проходящих через G направо, сократится на 1. Поэтому потребуется одинарный поворот деда (G) относительно отца, в данном случае направо. Тогда количество черных узлов в путях, идущих от F, который теперь стал корнем поддерева, налево и направо, вновь станет одинаковым и будет равно первоначальному количеству.

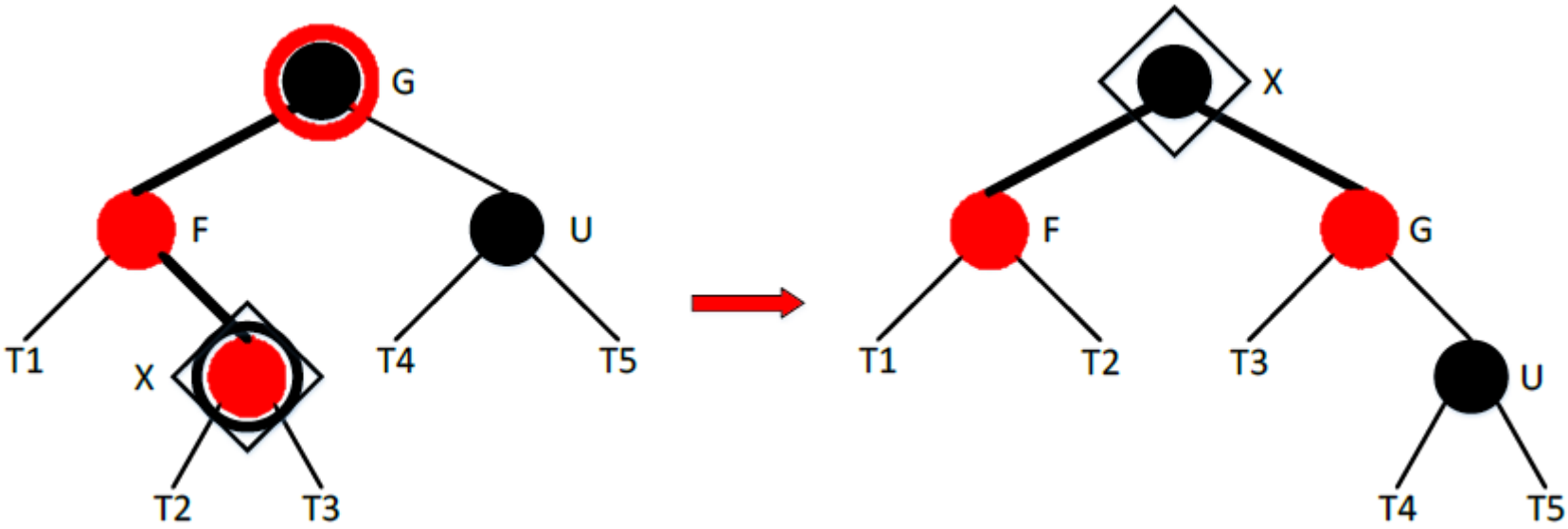
Действительно, количество черных узлов в поддеревьях Ti не изменилось. При этом слева от корня в комбинации X-F-G-U не было черных узлов, а справа был только один черный узел U. В результирующем дереве, изображенном на рисунке, в комбинации X-F-G-U слева от корня также отсутствуют черные узлы, а справа – также только один черный узел.



3. Дядя U добавляемого узла X черный и при этом цепочка узлов X-F-G образует угол.

Перекрасив узел X в черный цвет, а его деда G – в красный, получим новое красно-красное нарушение F-G между отцом и дедом X. Ситуацию разрешает двойной поворот комбинации X-F-G, знакомый нам из лекций про АВЛ-деревья, когда нижний узел (X) оказывается наверху комбинации. Из рисунка видно, что количество черных узлов, идущих от корня поддерева налево и направо, не изменилось относительно первоначальной конфигурации, но при этом красно-красное нарушение ликвидировалось.

Изменилось только количество красных узлов – слева уменьшилось на 1, а справа увеличилось на 1. А количество красных узлов в путях ни на что не влияет.



Таким образом, получается, что после вставки в КЧ-дерево для восстановления свойств дерева требуется не более 2 поворотов. Действительно, повороты требуются только в случаях 2 и 3, а в случае 1 достаточно только перекраски. При этом случаи 2 и 3 являются терминальными, а рекурсивно продолжиться наверх может только процедура в случае 1, а она не требует поворотов.

Можно заметить, что повороты при балансировке как АВЛ- деревьев, так и КЧ-деревьев, делают одно и то же – поднимают поддеревья, содержащие более глубокие узлы, и опускают поддеревья, содержащие менее глубокие узлы. Различие заключается лишь в определении высоты. Для АВЛ-деревьев это высота в обычном понимании, а для КЧ-деревьев это черная высота.

```
        //печать красно-черного дерева
void Print_RBTree(RBTree* Node, int l){
  int i;
  if (Node != NIL) {
    Print_RBTree(Node->Right, l+1);
    for (i=0; i< l; i++) cout << "    ";
    if (Node->color == RED)
      SetConsoleTextAttribute(hStd,FOREGROUND_RED);
    cprintf ("%4ld", Node->Data);
    SetConsoleTextAttribute(hStd,atr);
    Print_RBTree(Node->Left, l+1);
  }
  else cout << endl;
}
```

```
        //прямой обход красно-черного дерева
void PreOrder_RBTree(RBTree* Node){
  if (Node != NIL) {
    printf ("%3ld",Node->Data);
    PreOrder_RBTree(Node->Left);
    PreOrder_RBTree(Node->Right);
  }
}
```

```
        //обратный обход красно-черного дерева
void PostOrder_RBTree(RBTree* Node){
  if (Node != NIL) {
    PostOrder_RBTree(Node->Left);
    PostOrder_RBTree(Node->Right);
    printf ("%3ld",Node->Data);
  }
}
```

```
        //проверка пустоты красно-черного дерева
bool Empty_RBTree(RBTree* Node){
  return ( Node == NIL ? true : false );
}

        //освобождение памяти, выделенной под красно-черное дерево
void Delete_RBTree(RBTree* Node){
  if (Node != NIL) {
    Delete_RBTree(Node->Left);
    Delete_RBTree(Node->Right);
    delete(Node);
  }
}
```

Удаление узла из КЧ-дерева

Удаление узла из КЧ-дерева, так же, как и удаление узла из АВЛ-дерева производится в два этапа: собственно удаление с помощью алгоритма, и восстановление свойств дерева, если они были нарушены. Алгоритм удаления узла из ДДП зависит от того, сколько сыновей у удаляемого узла – 0, 1 или 2. Если у узла два сына, то удаляемым узлом становится его последователь – самый левый элемент правого поддерева. В этом случае у удаляемого узла уже будет максимум один сын (правый). Поэтому, в дальнейшем предполагается, что у удаляемого узла не более одного сына.

Если удаляемый узел является красным, то после его удаления свойства КЧ-дерева не будут нарушены.

Таким образом, восстановление свойств дерева может понадобиться только в том случае, если удаляемый узел – черный. Если при этом сын удаляемого узла красный, то после удаления достаточно будет перекрасить сына удаленного узла в черный цвет, чтобы восстановить количество черных узлов на этом пути. Остальные свойства КЧ-деревьев не будут нарушены. Подобная ситуация может возникнуть и в том случае, если удаляемый узел является корнем. В этом случае дерево может состоять только из двух узлов – корня, который всегда черный, и его красного сына. Любые другие конфигурации в этом случае не отвечают свойствам КЧ-дерева. После удаления корня его сын станет единственным узлом в дереве.

Рассчитаем максимальную глубина корректного красно-черного дерева с n вершинами.

Возьмем самый глубокий лист. Пусть он находится на глубине h. Из-за правила 1, как минимум половина вершин на пути из корня будет черными, то есть черная высота дерева будет не меньше h/2. Можно показать, что в таком дереве будет не менее 2^(h/2)-1 черных вершин (так как у каждой черной вершины с черной глубиной k, если она не лист, должно быть как минимум два потомка с черной глубиной k+1). Тогда

$$2^{(h/2)-1} \leq n \text{ или } h \leq 2*\log_2(n+1).$$

Все основные операции с красно-черным деревом можно реализовать за O(h) шагов, то есть O(log n) исходя из расчетов сделанных ранее. Классическая реализация основана на разборе большого количества случаев и довольно трудна для восприятия. Существуют более простые и понятные варианты, например в статье Криса Окасаки. К сожалению, в ней описана только операция вставки в дерево. Простота по сравнению с классической реализацией получается за счет ориентации на понятность, а не на оптимизацию количества элементарных модификаций дерева (вращений).

Для реализации этого вида сбаласированных деревьев, нужно в каждой вершине хранить дополнительно 1 бит информации (цвет). Иногда это вызывает большой overhead из-за выравнивания. В таких случаях предпочтительно использовать структуры без дополнительных требований к памяти.

Красно-черные деревья широко используются — реализация set/map в стандартных библиотеках, различные применения в ядре Linux (для организации очередей запросов, в ext3 etc.), вероятно во многих других системах для аналогичных нужд.

Для реализации этого вида сбаласированных деревьев, нужно в каждой вершине хранить дополнительно 1 бит информации (цвет). Иногда это вызывает большой overhead из-за выравнивания. В таких случаях предпочтительно использовать структуры без дополнительных требований к памяти.

Красно-черные деревья широко используются — реализация set/map в стандартных библиотеках, различные применения в ядре Linux (для организации очередей запросов, в ext3 etc.), вероятно во многих других системах для аналогичных нужд.

ОЦЕНКА СЛОЖНОСТИ ПОИСКА В КЧ - ДЕРЕВЕ

Лемма 1. Красно-черное дерево с *n* внутренними узлами (без фиктивных листьев) имеет высоту не более $2log_2(n+1)$.

Доказательство.

1. Обозначим через *bh(x)* черную высоту поддерева с корнем в узле *x*. Покажем вначале, что оно содержит не менее $2_{bh(x)} - 1$ внутренних узлов.

а. Индукция. Для листьев (не фиктивных) *bh(x)* = 0, то есть, $2_{bh(x)} - 1 = 2_0 - 1 = 0$.

b. Пусть теперь *x* – не лист и имеет черную высоту *bh(x) = k*. Тогда каждый сын *x* имеет черную высоту не менее *k* – 1. Действительно, если узел красный, его сыновья могут быть только черными, и в этом случае черная высота сына *x* будет на 1 меньше, чем черная высота *x*, т.е. будет равна *k* – 1, потому как при расчете черной высоты узла сам узел в это число не включается. Если узел черный, то его сыновья могут быть как черными, так и красными. Если сын черный, то аналогично предыдущему случаю его черная высота на 1 меньше, чем у *x* и равна *k* – 1. Если сын красный, то он имеет черную высоту такую же, как и *x*, т.е. *k*, т.к. количество чер-ных узлов от *x* до листа, не включая *x*, такое же, как и коли-чество черных узлов от его красного сына до листа.

c. По предположению индукции каждый сын имеет черную высоту $bh \geq k - 1$, а, следовательно, не менее $2_{k-1} - 1$ узлов. Поэтому поддерево с корнем x имеет не менее $2_{k-1} - 1 + 2_{k-1} - 1 + 1 = 2_k - 1$ узлов.

2. Теперь пусть высота дерева равна h.

a. По свойству красно-черных деревьев, что если узел красный, то оба его сына черные, черные узлы составляют не меньше половины всех узлов на пути от корня к листу. Поэтому, черная высота дерева bh не меньше $h/2$.

b. Тогда $n \geq 2_{h/2} - 1$, откуда

$$h \leq 2log_2(n + 1).$$

Лемма доказана.

Из Леммы 1 следует, что поиск по КЧ-дереву имеет сложность $O(log_2 n)$.

c. По предположению индукции каждый сын имеет черную высоту $bh \geq k - 1$, а, следовательно, не менее $2_{k-1} - 1$ узлов. Поэтому поддерево с корнем x имеет не менее $2_{k-1} - 1 + 2_{k-1} - 1 + 1 = 2_k - 1$ узлов.

2. Теперь пусть высота дерева равна h.

a. По свойству красно-черных деревьев, что если узел красный, то оба его сына черные, черные узлы составляют не меньше половины всех узлов на пути от корня к листу. Поэтому, черная высота дерева bh не меньше $h/2$.

b. Тогда $n \geq 2_{h/2} - 1$, откуда

$$h \leq 2log_2(n + 1).$$

Лемма доказана.

Из Леммы 1 следует, что поиск по КЧ-дереву имеет сложность $O(log_2 n)$.

САМОПЕРЕСТРАИВАЮЩИЕСЯ ДЕРЕВЬЯ (SPLAY TREES)

Самоперестраивающееся дерево – это двоичное дерево поиска, которое, в отличие от предыдущих двух видов деревьев не содержит дополнительных служебных полей в структуре данных (баланс, цвет и т.п.). Оно позволяет находить быстрее те данные, которые использовались недавно. Самоперестраивающееся дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году.

Эти деревья не являются перманентно сбалансированными и на отдельных запросах могут работать даже линейное время. Однако, после каждого запроса они меняют свою структуру, что позволяет очень эффективно обрабатывать часто повторяющиеся запросы. Более того, амортизационная стоимость обработки одного запроса у них $O(logn)$, что делает splay-деревья хорошей альтернативой для перманентно сбалансированных собратьев.

Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным с точки зрения использования памяти. После каждого обращения, даже поиска, splay-дерево меняет свою структуру.

Идея самоперестраивающихся деревьев основана на принципе пе-ремещения найденного узла в корень дерева. Эта операция назы-вается splay(*T, k*), где *k* – это ключ, а *T* – двоичное дерево поиска. После выполнения операции splay(*T, k*) двоичное дерево *T* пере-страивается, оставаясь при этом деревом поиска, так, что:

- **если узел с ключом *k* есть в дереве, то он становится корнем;**
- **если узла с ключом *k* нет в дереве, то корнем становится его предшественник или последователь.**

Таким образом, поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay. Эвристика move-to-front (перемещение найденного узла в корень) основана на предположении, что если тот же самый элемент потребуется в ближайшее время, он будет найден быстрее.

Таким образом, поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay. Эвристика move-to-front (перемещение найденного узла в корень) основана на предположении, что если тот же самый элемент потребуется в ближайшее время, он будет найден быстрее.