Red Black Tree

jask

10/02/2024

红黑树

红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 `O ( log n )` 时间内完成查找、插入和删除，这里的 `n` 是树中元素的数目。

结构

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是 `NIL` 节点）。
4. 每个红色节点必须有两个黑色的子节点。（或者说从每个叶子到根的所有路径上不能有两个连续的红色节点。）（或者说不存在两个相邻的红色节点，相邻指两个节点是父子关系。）（或者说红色节点的父节点和子节点均是黑色的。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

特点

这些约束确保了红黑树的关键特性：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

操作

因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。恢复红黑树的性质需要少量 `O ( log n )` 的颜色变更（实际是非常快速的）和不超过三次树旋转（对于插入操作是两次）。虽然插入和删除很复杂，但操作时间仍可以保持为 `O ( log n )` 次。

插入

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（`color flips`）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意：

性质 `1` 和性质 `3` 总是保持着。

性质 `4` 只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。

性质 `5` 只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

删除

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质 3 和性质 4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质 5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质 5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质 5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该节点的两个儿子都是叶子节点，否则若其中一个儿子是黑色非叶子节点，另一个儿子是叶子节点，那么从该节点通过非叶子节点儿子的路径上的黑色节点数最小为 2，而从该节点到另一个叶子节点儿子的路径上的黑色节点数为 1，违反了性质 5)。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为 N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为 S。在下面的示意图中，我们还是使用 P 称呼 N 的父亲，SL 称呼 S 的左儿子，SR 称呼 S 的右儿子。

实现

左旋

左旋的目的是将一个节点 x 向下移动，使得它的右子节点 y 取代它成为新的父节点。具体来说，x 的右子树被提升，而 x 变为右子树的新左子节点。

假设节点 x 和它的右子节点 y 在左旋之前的关系如下：

```
x
  \
   y
  / \
 A   B
```

经过旋转：

```
   y
  / \
 x   B
/
```

A

代码：

```
void RedBlackTree::leftRotate(Node *x) {
  if (x == nullptr || x->right == nullptr)
    return;

  Node *y = x->right;//获取右子节点
  x->right = y->left;//重新链接 x 的右子节点
  if (y->left != nullptr)//如果 y 的左子节点存在，就更新他的父节点
    y->left->parent = x;
  //更新 y,x 的父节点
  y->parent = x->parent;
  if (x->parent == nullptr)//如果 x 是根节点，就更新根节点为 y
    root = y;
  else if (x == x->parent->left)
    x->parent->left = y;
  else
    x->parent->right = y;
  y->left = x;//将 x 设置为 Y 的左子节点
  x->parent = y;
}
```

右旋逻辑相同

插入修复

当在红黑树中插入一个新节点时，如果红黑树的性质被破坏，需要通过调整颜色和旋转来恢复平衡。

代码：

```
void RedBlackTree::fixInsert(Node *z) {
  while (z != root && z->parent->color == RED) {//终止条件: z 已经是根节点或者 z 的父节点是黑色
    if (z->parent == z->parent->parent->left) {//父节点是左子节点的情况
```

```
      Node *y = z->parent->parent->right;//如果 x 的叔叔节点存在且为红色，这时叔叔，父亲，祖父构成'red-red-red'的结构
      if (y != nullptr && y->color == RED) {//处理方法
        z->parent->color = BLACK;//父节点和叔叔节点改为黑色
        y->color = BLACK;
        z->parent->parent->color = RED;//祖父节点改为红色
        z = z->parent->parent;//将 z 上移到祖父节点继续检查
      } else {//叔叔节点不存在或者是黑色
        if (z == z->parent->right) {//如果 z 是父节点的右子节点
          z = z->parent;
          leftRotate(z);//左旋，使 z 转换为父节点的左子节点
        }
        z->parent->color = BLACK;//将 z 的父节点染为黑色
        z->parent->parent->color = RED;//将祖父节点染色为红色
        rightRotate(z->parent->parent);//对祖父节点右旋，调整平衡
      }
    } else {//父节点是右节点的情况
      Node *y = z->parent->parent->left;//对称处理
      if (y != nullptr && y->color == RED) {
        z->parent->color = BLACK;
        y->color = BLACK;
        z->parent->parent->color = RED;
        z = z->parent->parent;
      } else {
        if (z == z->parent->left) {
          z = z->parent;
          rightRotate(z);
        }
        z->parent->color = BLACK;
        z->parent->parent->color = RED;
        leftRotate(z->parent->parent);
      }
    }
  }
  root->color = BLACK;
}
```

节点移植

该函数主要用于删除操作中的辅助步骤，它将树中的一个节点 `u` 替换为另一个节点 `v`，并维护父节点与子节点之间的关系。具体作用是将 `u` 的位置让给 `v`，使得树的结构能够继续保持。

代码：

```
void RedBlackTree::transplant(Node *u, Node *v) {
  if (u->parent == nullptr) {//u 若为根节点，直接替换根节点为 b
    root = v;
  } else if (u == u->parent->left) {//判断 u 是左子节点还是右子节点
    u->parent->left = v;//如果 u 是父节点的左子，设置 v 为该位置
  } else {
    u->parent->right = v;
  }
  if (v != nullptr)
    v->parent = u->parent;//如果 v 不为空（即 v 不是 nullptr），需要更新 v 的父节点指向 u 的父节点，确保 v 正确接替了 u 的位置。
}
```

删除

代码：

```
void RedBlackTree::deleteNode(Node *z) {
  if (z == nullptr)
    return;
```

```
  Node *y = z;//记录要删除的节点
  Node *x = nullptr;
  Color y_original_color = y->color;
  if (z->left == nullptr) {//如果没有左子节点
    x = z->right;//用右节点替换他
    transplant(z, z->right);
  } else if (z->right == nullptr) {//对称
    x = z->left;
    transplant(z, z->left);
  } else {//有两个子节点的情况
    y = minimum(z->right);//找到 z 右子树最小节点作为替换节点
    y_original_color = y->color;
    x = y->right;//x 是 y 的右字节点，用来替代 y 的位置
    if (y->parent == z) {//y 是 z 的直接子节点，就直接设置 x 为 y 的子节点
      if (x != nullptr)
        x->parent = y; // Check if x is not nullptr before assigning parent
    } else {//y 不是 z 的子节点
      if (x != nullptr)//先用 y 的右子节点替换 y 自己，在用 y 替换 z
        x->parent = y->parent; // Check if x and y->parent are not nullptr
                               // before assigning parent
      transplant(y, y->right);
      if (y->right != nullptr)
        y->right->parent =
            y; // Check if y->right is not nullptr before assigning parent
      y->right = z->right;
      if (y->right != nullptr)
        y->right->parent =
            y; // Check if y->right is not nullptr before assigning parent
    }
    transplant(z, y);
    y->left = z->left;
    if (y->left != nullptr)
      y->left->parent =
          y; // Check if y->left is not nullptr before assigning parent
    y->color = z->color;
  }

  if (y_original_color == BLACK && x != nullptr) // Check if x is not nullptr
    fixDelete(x);//平衡修复

  delete z; // Free memory allocated for the deleted node
}
```

插入

代码：

```
void RedBlackTree::insert(int val) {
  Node *newNode = new Node(val);//创建一个新节点
  Node *y = nullptr;//记录父节点，寻找用来插入位置的父节点
  auto x = root;//遍历树
  while (x != nullptr) {
    y = x;
    if (newNode->data < x->data)
      x = x->left;
    else
      x = x->right;
  }
  newNode->parent = y;//插入新节点
  if (y == nullptr)//空的则为根节点
    root = newNode;
  else if (newNode->data < y->data)
```

```
    y->left = newNode;
  else
    y->right = newNode;
  fixInsert(newNode);//修复插入
}
```