Report of Debug_Lab2

522031910732 王熠笑

bug1

测试

进入文件后运行，发现有红体字报错

类别

头文件缺失

发现

对红体字报错进行阅读查阅

修复

在头文件处添加

- 发现进入文件后有红色报错
- 添加对应头文件解决bug

bug2

测试

对初始main函数进行运行，发现无法成功运行结束

类别

函数逻辑错误

发现

进行debug，发现tree.set(1, 42);运行时会报错（segmentation default）

进行逐步分析发现while循环有问题

修复

```
node_t *find_leaf(int key) {
  node_t *node = root;
  while (node->is_leaf) {
    node = node->get_child(key);
  }
  return node;
}
```

将while (node->is_leaf)改为while (!node->is_leaf)

bug3

测试

修改main函数为

```
tree_t tree(3);
for(int i=10;i>0;--i){
    tree.set(i,3*i);
}
```

类别

函数逻辑错误

发现

发现在第四次插入时会发生错误

进行逐步分析发现在

```
if (prev_) {
prev_->right = this;
}
```

处会产生segmentation default

进一步寻找原因，发现在split_leaf函数中

```
std::tuple<int, node_t *, node_t *> split_leaf() {
  node_t *left = new node_t(up, true, /*left*/this->left, this);
  int mid = key_list.size() / 2;

  left->key_list = std::vector<int>(key_list.begin(), key_list.begin() + mid);
  left->value_list =
      std::vector<int>(value_list.begin(), value_list.begin() + mid);

  key_list.erase(key_list.begin(), key_list.begin() + mid);
  value_list.erase(value_list.begin(), value_list.begin() + mid);

  return {key_list[0], left, this};
}
```

```
node_t *left = new node_t(up, true, left, this);
```

对prev的初始化有误。

修复

将left改为this->left;

即

```
node_t *left = new node_t(up, true, /*left*/this->left, this);
```

bug4

测试

进行下面代码测试

```
tree_t tree(3);
  for(int i=10;i>0;--i){
  tree.set(i,3*i);
  }
  tree.remove(1);
  tree.remove(2);
  tree.remove(3);
  tree.remove(4);
  tree.remove(5);
```

类别

delete错误

发现

在remove(5)处打断点并进行单步处理，发现在

```
void merge_node_with_right_internal(int my_position_in_parent, node_t *node,
                                    node_t *next)
```

函数中最后一步delete next;

会使得原本转移到node的元素也被销毁，导致内容错乱

修复

在delete next前对next的子节点进行消除处理，处理后的函数为：

```
  void merge_node_with_right_internal(int my_position_in_parent, node_t *node,
                                      node_t *next) {
    node->key_list.insert(node->key_list.end(),
                          node->up->key_list[my_position_in_parent]);
    node->up->key_list.erase(node->up->key_list.begin() +
                             my_position_in_parent);
    node->up->down.erase(node->up->down.begin() + my_position_in_parent + 1);
    node->key_list.insert(node->key_list.end(), next->key_list.begin(),
                          next->key_list.end());
    node->down.insert(node->down.end(), next->down.begin(), next->down.end());
```

```
      for (node_t *child : node->down) {
        child->up = node;
      }
      next->down.erase(next->down.begin(),next->down.end()) ;
      delete next;
    }
```

bug5

类似的，猜测merge_node_with_left_internal也会有类似的问题，所以对其进行类似的处理

测试

进行下面的代码测试

```
tree_t tree(3);
for(int i=10;i>0;--i){
    tree.set(i,3*i);
}
tree.remove(10);
tree.remove(9);
tree.remove(8);
tree.remove(7);
tree.remove(6);
```

类别

delete错误

发现

在remove(6)处打断点并进行单步处理，发现在

```
void merge_node_with_left_internal(int my_position_in_parent, node_t *node,
                                     node_t *prev)
```

函数中最后一步delete node;

会使得原本转移到next的元素也被销毁，导致内容错乱

修复

在delete node前对其子节点进行消除处理，处理后的函数为：

```
void merge_node_with_left_internal(int my_position_in_parent, node_t *node,
                                     node_t *prev) {
    prev->key_list.insert(prev->key_list.end(),
                          node->up->key_list[my_position_in_parent - 1]);
    node->up->key_list.erase(node->up->key_list.begin() +
                             my_position_in_parent - 1);
    node->up->down.erase(node->up->down.begin() + my_position_in_parent);
    prev->key_list.insert(prev->key_list.end(), node->key_list.begin(),
                          node->key_list.end());
    prev->down.insert(prev->down.end(), node->down.begin(), node->down.end());
    for (node_t *child : prev->down) {
      child->up = prev;
    }
    node->down.erase(node->down.begin(),node->down.end());
    delete node;
  }
```

bug6

类别

内存泄露

修复

merge_node_with_right_leaf 和 merge_node_with_left_leaf中在进行转移后没有对舍弃的node进行处理，故对其进行消除子节点并delete处理

```
void merge_node_with_right_leaf(node_t *node, node_t *next) {
  node->key_list.insert(node->key_list.end(), next->key_list.begin(),
                        next->key_list.end());
  node->value_list.insert(node->value_list.end(), next->value_list.begin(),
                          next->value_list.end());
  node->right = next->right;
  if (node->right)
    node->right->left = node;
  for (int i = 0; i < next->up->down.size(); i++) {
    if (node->up->down[i] == next) {
      node->up->key_list.erase(node->up->key_list.begin() + i - 1);
      node->up->down.erase(node->up->down.begin() + i);

      break;
    }
  }
  next->down.erase(next->down.begin(),next->down.end()) ;
  delete next;
}

void merge_node_with_left_leaf(node_t *node, node_t *prev) {
  prev->key_list.insert(prev->key_list.end(), node->key_list.begin(),
                        node->key_list.end());
  prev->value_list.insert(prev->value_list.end(), node->value_list.begin(),
                          node->value_list.end());

  prev->right = node->right;
  if (prev->right)
    prev->right->left = prev;

  for (int i = 0; i < node->up->down.size(); i++) {
    if (node->up->down[i] == node) {
      node->up->key_list.erase(node->up->key_list.begin() + i - 1);
      node->up->down.erase(node->up->down.begin() + i);
      break;
    }
  }
  node->down.erase(node->down.begin(),node->down.end());
  delete node;
}
```

bug7

类别

逻辑错误 use after delete

发现

在remove函数中，发现

```
if (node->up) {
  remove(key, node->up);
}
```

中的node可能会是空指针，导致node->up产生segmentation fault，回溯后发现是因为在merge_node_with_left_internal函数中已经将node进行销毁处理，在逐步debug后发现应该在merge_node_with_left_internal函数后将node赋值为prev；

修复

在merge_node_with_left_internal函数后将node赋值为prev；

修复后的代码如下：

```
void remove(int key, node_t *node = nullptr) {
  if (node == nullptr) {
    node = find_leaf(key);
  }
  if (node->is_leaf) {
    remove_from_leaf(key, node);
  } else {
    remove_from_internal(key, node);
  }

  if (node->key_list.size() < min_capacity) {
    if (node == root) {
      if (root->key_list.empty() && !root->down.empty()) {
        root = root->down[0];
        root->up->down.erase(root->up->down.begin(),root->up->down.end());
        delete root->up;
        root->up = nullptr;
        height -= 1;
      }
      return;
    }

    else if (node->is_leaf) {
      node_t *next = node->right;
      node_t *prev = node->left;

      if (next && next->up == node->up &&
          next->key_list.size() > min_capacity) {
        borrow_key_from_right_leaf(node, next);
      } else if (prev && prev->up == node->up &&
                 prev->key_list.size() > min_capacity) {
        borrow_key_from_left_leaf(node, prev);
      } else if (next && next->up == node->up &&
                 next->key_list.size() <= min_capacity) {
        merge_node_with_right_leaf(node, next);
      } else if (prev && prev->up == node->up &&
                 prev->key_list.size() <= min_capacity) {
        merge_node_with_left_leaf(node, prev);
      }
    } else {
      int my_position_in_parent = -1;

      for (int i = 0; i < node->up->down.size(); i++) {
        if (node->up->down[i] == node) {
          my_position_in_parent = i;
          break;
        }
      }

      node_t *next;
      node_t *prev;

      if (node->up->down.size() > my_position_in_parent + 1) {
        next = node->up->down[my_position_in_parent + 1];
      } else {
        next = nullptr;
      }

      if (my_position_in_parent) {
        prev = node->up->down[my_position_in_parent - 1];
      } else {
        prev = nullptr;
      }

      if (next && next->up == node->up &&
          next->key_list.size() > min_capacity) {
        borrow_key_from_right_internal(my_position_in_parent, node, next);
      }

      else if (prev && prev->up == node->up &&
               prev->key_list.size() > min_capacity) {
        borrow_key_from_left_internal(my_position_in_parent, node, prev);
      }

      else if (next && next->up == node->up &&
               next->key_list.size() <= min_capacity) {
        merge_node_with_right_internal(my_position_in_parent, node, next);
      }
      else if (prev && prev->up == node->up &&
               prev->key_list.size() <= min_capacity) {
        merge_node_with_left_internal(my_position_in_parent, node, prev);
        node = prev;
      }
    }
  }
  if (node->up) {
    remove(key, node->up);
  }
}
```

类似的，我们要对merge_node_with_left_leaf函数后进行修改，

修改后的remove函数为：

```
void remove(int key, node_t *node = nullptr) {
  if (node == nullptr) {
    node = find_leaf(key);
  }
  if (node->is_leaf) {
    remove_from_leaf(key, node);
  } else {
    remove_from_internal(key, node);
  }

  if (node->key_list.size() < min_capacity) {
    if (node == root) {
      if (root->key_list.empty() && !root->down.empty()) {
        root = root->down[0];
        root->up->down.erase(root->up->down.begin(),root->up->down.end());
        delete root->up;
        root->up = nullptr;
        height -= 1;
      }
      return;
    }

    else if (node->is_leaf) {
      node_t *next = node->right;
      node_t *prev = node->left;

      if (next && next->up == node->up &&
          next->key_list.size() > min_capacity) {
        borrow_key_from_right_leaf(node, next);
      } else if (prev && prev->up == node->up &&
                 prev->key_list.size() > min_capacity) {
        borrow_key_from_left_leaf(node, prev);
      } else if (next && next->up == node->up &&
                 next->key_list.size() <= min_capacity) {
        merge_node_with_right_leaf(node, next);
      } else if (prev && prev->up == node->up &&
                 prev->key_list.size() <= min_capacity) {
        merge_node_with_left_leaf(node, prev);
        node = prev;
      }
    } else {
      int my_position_in_parent = -1;

      for (int i = 0; i < node->up->down.size(); i++) {
        if (node->up->down[i] == node) {
          my_position_in_parent = i;
          break;
        }
      }

      node_t *next;
      node_t *prev;

      if (node->up->down.size() > my_position_in_parent + 1) {
        next = node->up->down[my_position_in_parent + 1];
      } else {
        next = nullptr;
      }

      if (my_position_in_parent) {
        prev = node->up->down[my_position_in_parent - 1];
      } else {
        prev = nullptr;
      }

      if (next && next->up == node->up &&
          next->key_list.size() > min_capacity) {
        borrow_key_from_right_internal(my_position_in_parent, node, next);
      }

      else if (prev && prev->up == node->up &&
               prev->key_list.size() > min_capacity) {
        borrow_key_from_left_internal(my_position_in_parent, node, prev);
      }

      else if (next && next->up == node->up &&
               next->key_list.size() <= min_capacity) {
        merge_node_with_right_internal(my_position_in_parent, node, next);
      }

      else if (prev && prev->up == node->up &&
               prev->key_list.size() <= min_capacity) {
        merge_node_with_left_internal(my_position_in_parent, node, prev);
        node = prev;
      }
    }
  }
  if (node->up) {
    remove(key, node->up);
  }
}
```

bug8

类别

内存泄露

发现

在上面的bug进行修复后，使用valgrind进行检测发现还是存在内存泄漏

修复

发现是remove函数中存在内存泄露

```
if (node->key_list.size() < min_capacity) {
  if (node == root) {
    if (root->key_list.empty() && !root->down.empty()) {
      root = root->down[0];
      root->up->down.erase(root->up->down.begin(),root->up->down.end());
      delete root->up;
      root->up = nullptr;
      height -= 1;
    }
    return;
  }
```

此处修改为

```
if (node->key_list.size() < min_capacity) {
  if (node == root) {
    if (root->key_list.empty() && !root->down.empty()) {
      root = root->down[0];
      root->up->down.erase(root->up->down.begin(),root->up->down.end());
      delete root->up;
      root->up = nullptr;
      height -= 1;
      node->down.pop_back();
      delete node;
    }
    return;
  }
```

从而内存泄漏解决。