COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #6 - Due: November 29, 2018 – 11:59PM

Name: **Date:**

- Number of questions: 10
- Points per question: 0.3
- Total: 3 points

---

1. For the bag class defined in Appendix 1, complete the following function: (you can use the `tree_copy` function)

```
1.  template < class Item >
2.  void bag <Item>::operator = (const bag<Item>& source)
3.  // Header file used: bintree.h
4.  {
5.      if (this == &source || root_ptr == source.root_ptr)
6.          return;
7.
8.      tree_clear (root_ptr);
9.      root_ptr = tree_copy (source.root_ptr);
10. }
```

- In addition, since the above function uses `tree_copy`, please complete the following implementation as well.

```
1.  template < class Item >
2.  binary_tree_node <Item>* tree_copy (const binary_tree_node <Item>* root_ptr)
3.  // Library facilities used: cstdlib
4.  {
5.      binary_tree_node <Item>* l_ptr;
6.      binary_tree_node <Item>* r_ptr;
7.
8.      if (root_ptr == NULL)
9.          return NULL;
10.     else
11.     {
12.         l_ptr = tree_copy (root_ptr -> left());
13.         r_ptr = tree_copy (root_ptr -> right());
14.         return new binary_tree_node <Item> (root_ptr -> data(), l_ptr, r_ptr);
15.     }
16. }
```

2. For the bag class defined in Appendix 1, complete the following function:

```
template < class Item >
void bag<Item>::insert(const Item& entry)
// Postcondition: A new copy of entry has been inserted into the bag.
// Header file used: bintree.h
{
        binary_tree_node <Item>* cursor = root_ptr;
        bool done = false;

        if (root_ptr == NULL) {
            root_ptr = new binary_tree_node < Item > (entry);
            return;
        }

        do {

            if ( entry <= cursor -> data() ) {
            // Go left
                if (cursor -> left() == NULL) {
                    cursor -> set_left (new binary_tree_node <Item> (entry));
                    done = true;
                }
                else cursor = cursor - > left();
            }

            else {
            // Go right
                if (cursor -> right() == NULL) {
                    cursor -> set_right (new binary_tree_node <Item> (entry));
                    done = true;
                }
                else cursor = cursor -> right();
            }

        } while (!done);
    }
```

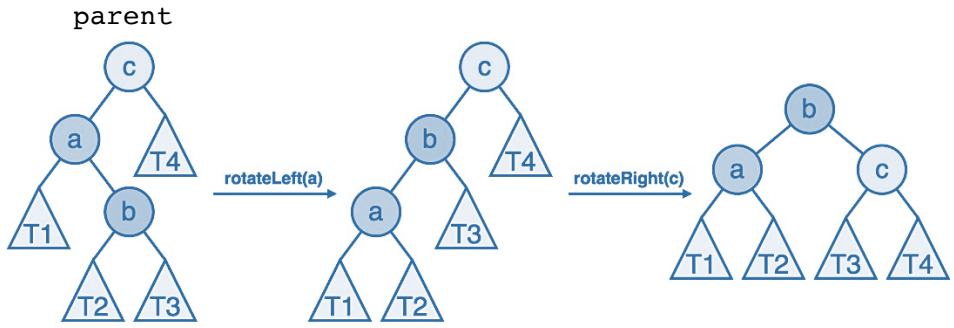
3. Suppose MINIMUM is 1000 for a B-tree. The tree has a root and one level of 1000 nodes (children) below that. How many *entries* are in the tree? (Please provide a range and explain your answer)

Answer:

The root has 999 entries, and each of the 1000 nodes at the next level has between 1000 and 2000 entries.

The total number of entries in the tree is between `999+1000x1000 = 1,000,999` and `999+1000x2000 = 2,000,999` entries.

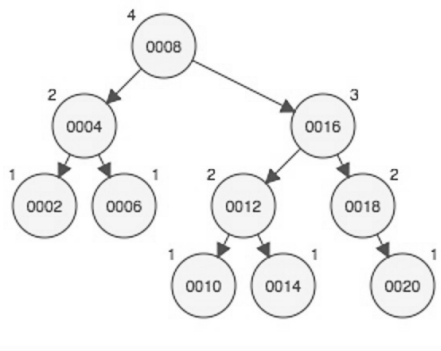
4. Write a code to perform *left-right* rotation on the following AVL tree. The figure shows the steps. (Note: Please implement the function in two steps: (1) left rotation, (2) right rotation.)



Answer:

```
1.  template < class Item >
2.  binary_tree_node <Item>* left_right_rotation (binary_tree_node <Item>*& parent)
3.  {
4.      binary_tree_node <Item>* temp;
5.
6.      temp = parent -> left();
7.      parent -> set_left(temp -> right());
8.      temp -> set_right(temp -> right() -> left());
9.      parent -> left() -> set_left(temp);
10.
11.     temp = parent -> left();
12.     parent -> set_left(temp -> right());
13.     temp -> set_right(parent);
14.
15.     return temp;
16. }
```

5. Add the following numbers to an AVL tree. Draw the final tree.
   `2, 4, 6, 8, 10, 12, 20, 18, 16, 14`



6. The following functions are available:

```
template < class Item >
int height (const binary_tree_node <Item>* temp)
{
    int h = 0;
    if (temp != NULL) {
        int l_height = height(temp -> left());
        int r_height = height(temp -> right());
        int max_height = std::max (l_height, r_height);
        h = max_height + 1;
    }
    return h;
}
```

```
template < class Item >
int diff (const binary_tree_node <Item>* temp)
{
    int l_height = height(temp -> left());
    int r_height = height(temp -> right());
    int b_factor = l_height - r_height;

    return b_factor;
}
```

Also assume the following functions are available:

- `binary_tree_node<Item>* left_rotation (binary_tree_node<Item>*& parent)`
- `binary_tree_node<Item>* right_rotation (binary_tree_node<Item>*& parent)`
- `binary_tree_node<Item>* left_right_rotation (binary_tree_node<Item>*& parent)`
- `binary_tree_node<Item>* right_left_rotation (binary_tree_node<Item>*& parent)`

Complete the following function, which balances a tree rooted at `temp`.

Answer:

```
template < class Item >
binary_tree_node<Item>* balance(binary_tree_node <Item>*& temp)
{
    int bal_factor = diff(temp);

    if (bal_factor > 1) {
        if (diff(temp -> left()) > 0)
            temp = right_rotation(temp);
        else
            temp = left_right_rotation(temp);
    }
    else if (bal_factor < -1) {
        if (diff(temp -> right()) > 0)
            temp = right_left_rotation(temp);
        else
            temp = left_rotation(temp);
    }

    return temp;
}
```

Add the following numbers to a B-Tree with MIN = 2. Draw the final tree.

```
2, 4, 6, 8, 10, 12, 20, 18, 16, 14
```

7. Complete the following function. Use a *recursive* implementation.

```
1.  template < class Item >
2.  void bst_remove_max (binary_tree_node <Item>*& root_ptr, Item & removed)
3.  // Precondition: root_ptr is a root pointer of a non-empty binary search tree.
4.  // Postcondition: The largest item in the binary search tree has been removed,
5.  // and root_ptr now points to the root of the new (smaller) binary search tree.
6.  // The reference parameter, removed, has been set to a copy of the removed item.
7.      {
8.          binary_tree_node<Item>* oldroot_ptr;
9.          assert(root_ptr != NULL);
10.
11.         if (root_ptr -> right() != NULL)
12.             bst_remove_max(root_ptr - > right(), removed);
13.
14.         else {
15.             removed = root_ptr -> data();
16.             oldroot_ptr = root_ptr;
17.             root_ptr = root_ptr -> left();
18.
19.             delete oldroot_ptr;
20.         }
21.     }
```

8. Please implement the following function (*recursively*).

```
template < class Item >
void flip(binary_tree_node < Item > * root_ptr)
// Precondition: root_ptr is the root pointer of a non-empty binary tree.
// Postcondition: The tree is now the mirror image of its original value.
```

Example:

```
//            1                                   1
//           / \                                 / \
//          2   3                               3   2
//         / \                                     / \
//        4   5                                   5   4
```

Answer:

```
template < class Item >
void flip (binary_tree_node <Item>* root_ptr)
{
    binary_tree_node <Item>* temp;

    assert(root_ptr != NULL);

    temp = root_ptr -> left();
    root_ptr -> set_left(root_ptr -> right());
    root_ptr -> set_right(temp);

    if (root_ptr -> left() != null)
        flip(root_ptr -> left());

    if (root_ptr -> right() != null)
        flip(root_ptr -> right());
}
```

9. What are the outputs of the following codes?

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class Base1 {
5.      public:
6.          ~Base1()  {
7.          cout << " Base1's destructor" << endl;  }
8.  };
9.  class Base2 {
10.     public:
11.         ~Base2()  {
12.         cout << " Base2's destructor" << endl;  }
13. };
14. class Derived: public Base1, public Base2 {
15.     public:
16.         ~Derived()  {
17.         cout << " Derived's destructor" << endl; }
18. };
19.
20. int main() {
21.     Derived d;
22.     return 0;
23. }
```

Output: Destructors are always called in reverse order of constructors.

- `Derived's destructor`
- `Base2's destructor`
- `Base1's destructor`

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class Base {
5.      private:
6.          int i, j;
7.      public:
8.          Base (int _i = 0, int _j = 0): i(_i), j(_j) {}
9.  };
10.
11. class Derived: public Base {
12.     public:
13.         void show() { cout << " i = " << i << "  j = " << j;   }
14. };
15.
16. int main(void) {
17.     Derived d;
18.     d.show();
19.     return 0;
20. }
```

Answer:
Compiler Error: i and j are private in `Base`. We cannot access these variables inside `Derived`.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class P {
5.      public:
6.          void print()  {
7.          cout << " Inside P";
8.      }
9.  };
10.
11. class Q: public P {
12.     public:
13.         void print() {
14.         cout << " Inside Q";
15.     }
16. };
17.
18. class R: public Q {};
19.
20. int main(void) {
21.     R r;
22.     r.print();
23.     return 0;
24. }
```

Output:

```
Inside Q
```

The print function is not present in class R. So it is looked up in the inheritance hierarchy. `print()` is present in both classes P and Q, which of them should be called? If there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class Base {};
5.
6.  class Derived: public Base {};
7.
8.  int main() {
9.      Base * bp = new Derived;
10.     Derived * dp = new Base;
11. }
```

Answer:

Compiler Error in line "`Derived *dp = new Base;`"
A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.

Object-Oriented Programming and Advanced Data Structures

Appendix 1: Bag class with binary search tree.

```
template < class Item >
class bag {

public:
    // TYPEDEFS
    typedef std::size_t size_type;
    typedef Item value_type;

    // CONSTRUCTORS and DESTRUCTOR
    bag() {  root_ptr = NULL;  }
    bag(const bag& source);
    ~bag();

    // MODIFICATION functions
    size_type erase(const Item& target);
    bool erase_one(const Item& target);
    void insert(const Item& entry);
    void operator += (const bag& addend);
    void operator = (const bag& source);

    // CONSTANT functions
    size_type size() const;
    size_type count(const Item& target) const;
    void debug() const {  print(root_ptr, 0);  }

private:
    binary_tree_node<Item>* root_ptr; // Root pointer of binary search tree
    void insert_all (binary_tree_node<Item>* addroot_ptr);
};
```