COEN 79

Object-Oriented Programming and Advanced Data Structures

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

Name: Jordan Murtiff **Date:** 11-28-18

- 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)

```
template < class Item >
void bag <Item>::operator = (const bag<Item>& source)
// Header file used: bintree.h
{
 if(this == &source)
 {
     return;
 }
 tree_clear(root_ptr);
 root_ptr = tree_copy(source.root_ptr);
 }
```

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

```
template < class Item >
binary_tree_node <Item>* tree_copy (const binary_tree_node <Item>* root_ptr)
// Library facilities used: cstdlib
{
    binary_tree_node <Item>* l_ptr;
    binary_tree_node <Item>* r_ptr;
 if(root_ptr == NULL)
 {
     return NULL;
 }
 else
 {
     l_ptr = tree_copy(root_ptr->left());
     r_ptr = tree_copy(root_ptr->right());
     return new binary_tree_node<Item>(root_ptr->data(), l_ptr, r_ptr);
 }
 }
```

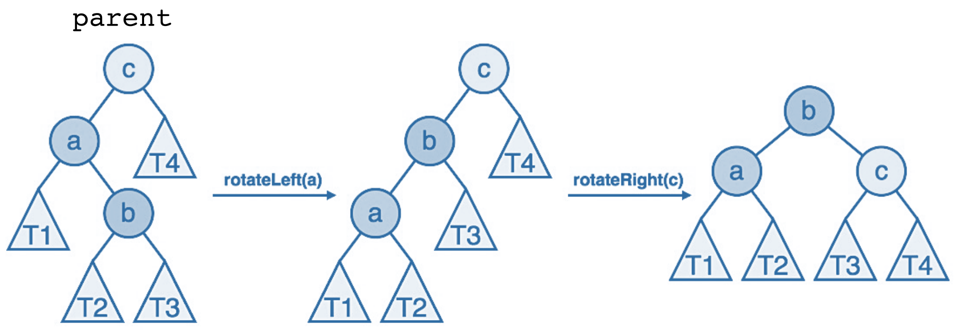
2. For the bag class defined in Appendix 1, complete the following function:

```
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(cursor->data() >= entry)
             {
                 if(cursor->left != NULL)
                 {
                     cursor = cursor->left();
                 }
                 else
                 {
                     done = true;
                     cursor->set_left(new binary_tree_node<Item>(entry));
                 }
             }
             else
             {
                 if(cursor->right() != NULL)
                 {
                     cursor = cursor->right();
                 }
                 else
                 {
                     done = true;
                     cursor->set_right(new binary_tree_node<Item>(entry));
                 }
             }
        } 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)

If the minimum is 1000, the maximum must be 2000 for a B-tree. The root has 999 entries, so that means that there can be 1000 children under the root node (following rule #4 that states that the number of subtrees below a non-leaf node is always one more than the number of entries in the node). This means that each of the 1000 nodes at the next level has between 1000 (minimum) and 2000 entries (maximum). If each node has between 1000 and 2000 entires, then the total number of entries must be between 1,000,999 and 2,000,999 entries (counting both the 999 entries at the root with either 1000 * 1000 total entries or 2000*1000 total 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:

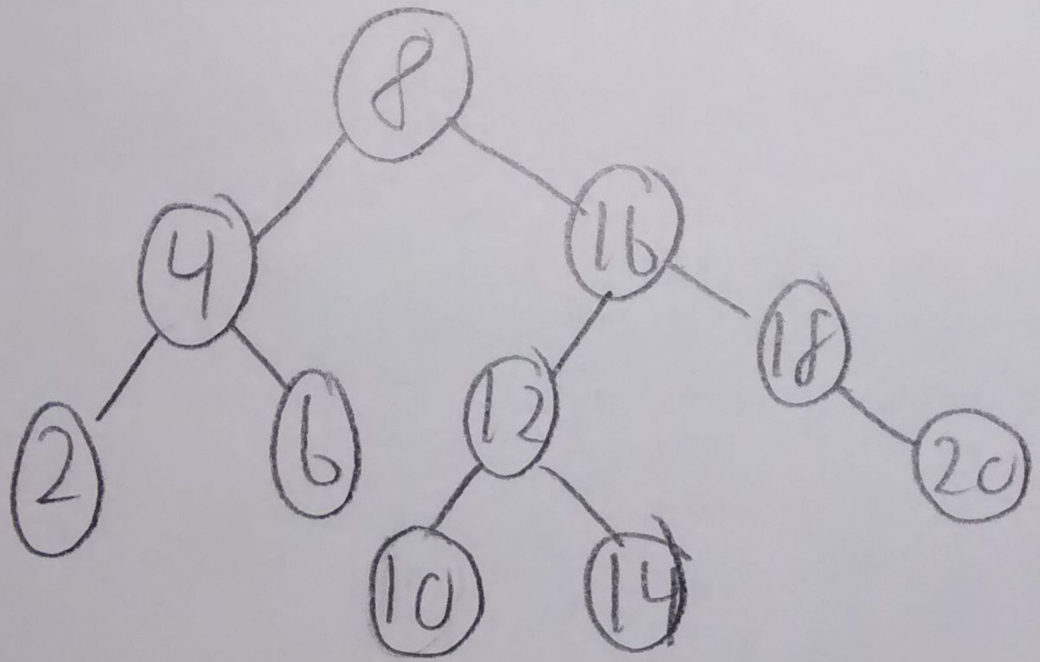
```
template < class Item >
binary_tree_node <Item>* left_right_rotation (binary_tree_node <Item>*& parent)
{
    binary_tree_node <Item>* temp;
    temp = parent->left()->right();
    parent->right()->set_right(temp->left());
    temp->rightset_left(parent);
    temp = parent->left();
    parent->set_left (temp->right());
    temp->set_right(parent);
    return temp;
```

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

```
          8
        /   \
       4     16
      / \   /   \
     2  6  12  18
           / \   \
         10 14  20
```

Also see next page for picture of answer.

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`.

```
template < class Item >
binary_tree_node<Item>* balance(binary_tree_node <Item>*& temp)
{
    int bal_factor = diff(temp);
     if(bal_factor == -2)
     {
         if(diff(temp->right() == -1)
         {
             left_rotate(temp);
         }
         else if(diff(temp->right() == 1)
         {
             right_left_rotation(temp);
         }
     }
     else if (bal_factor == 2)
     {
         if(diff(temp->left() == 1)
         {
             right_rotate(temp);
         }
         else if(diff(temp->left() == -1)
         {
             left_right_rotation(temp);
         }
     }
 }
```

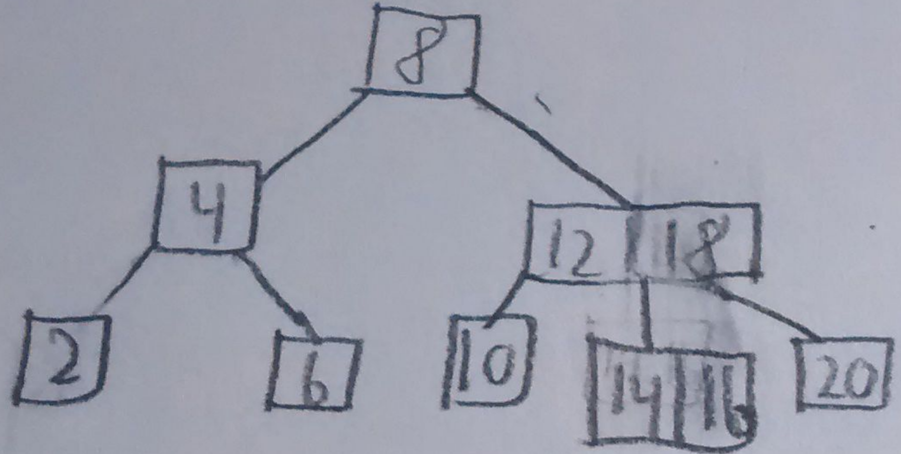
7. Add the following numbers to a B-Tree with MIN = 2. Draw the final tree.
   `2, 4, 6, 8, 10, 12, 20, 18, 16, 14`

See Next page for Answer to this question.

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

```
template < class Item >
void bst_remove_max (binary_tree_node <Item>*& root_ptr, Item & removed)
// Precondition: root_ptr is a root pointer of a non-empty binary search tree.
// Postcondition: The largest item in the binary search tree has been removed,
// and root_ptr now points to the root of the new (smaller) binary search tree.
// The reference parameter, removed, has been set to a copy of the removed item.
    {
        binary_tree_node<Item> *oldroot_ptr;
        assert(root_ptr != NULL);
         if(root_ptr->right() != NULL)
        {
            bst_remove_max(root_ptr->right(), removed);
        }
        else
        {
            oldroot_ptr = root_ptr;
            removed = oldroot_ptr->data();
            if(root_ptr->left() != NULL)
            {
                root_ptr = root_ptr -> left();
            }
            else
            {
                root_ptr = NULL;
            }
                delete oldroot_ptr;
        }
    }
```

7. 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`



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

9. 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
```

```
template < class Item >
void flip (binary_tree_node <Item>* root_ptr)
{
  if(root_ptr == NULL)
  {
      return root_ptr node;
  }
  if(root_ptr->left() == NULL && root_ptr->right() == NULL)
  {
      return root_ptr;
  }
  binary_tree_node <Item>* flippedRoot = flip(root_ptr->left());
  root_ptr->left()->left() = root_ptr->right();
  root_ptr->left()->right() = root_ptr;
  root_ptr->left() = root->right() = NULL;
  }
```

10. What are the outputs of the following codes?

```
#include < iostream >
using namespace std;

class Base1 {
    public:
        ~Base1()  {
        cout << " Base1's destructor" << endl;  }
};
class Base2 {
    public:
        ~Base2()  {
        cout << " Base2's destructor" << endl;  }
};
class Derived: public Base1, public Base2 {
    public:
        ~Derived()  {
        cout << " Derived's destructor" << endl; }
};

int main() {
    Derived d;
    return 0;
}
```

In this case, the destructors will be called in opposite order as the constructor. This means the Dervied class destructor will be called first, and then Base2 destuctor will be called, and finally Base1 destructor will be called.

Therefore the output is:
Derived's destructor
Base2's destructor
Base1's destructor

```
#include < iostream >
using namespace std;

class Base {
    private:
        int i, j;
    public:
        Base (int _i = 0, int _j = 0): i(_i), j(_j) {}
};

class Derived: public Base {
    public:
        void show() { cout << " i = " << i << "  j = " << j;   }
};

int main(void) {
    Derived d;
    d.show();
    return 0;
}
```

The code will not compile. Since the Dervied object "d" contains the private member variables of a "Base" object, we cannot directly access these private member variables. Even though Derived inherits from the Base class, the private member variables do not become public and therefore cannot be directly accessed through the "show" function. But if we were to use a "getter" function in the Base class, then we would be able to directly access the "i" and "j" private member variables.

```
#include < iostream >
using namespace std;

class P {
    public:
        void print()  {
        cout << " Inside P";
    }
};

class Q: public P {
    public:
        void print() {
        cout << " Inside Q";
    }
};

class R: public Q {};

int main(void) {
    R r;
    r.print();
    return 0;
}
```

The code will print "Inside Q" because in this case when the "R" object calls the print function, the compiler first looks in the R class, doesn't find the implementation, and goes one step up the hierarchy and looks at the Q class. The Q class does have an implementation for the print function and so the compiler calls the Q class print function and prints out Inside Q. The compiler doesn't need to look at the P class because once it has found an already suitable implementation, it just calls that function.

```
#include < iostream >
using namespace std;

class Base {};

class Derived: public Base {};

int main() {
    Base * bp = new Derived;
    Derived * dp = new Base;
}
```

The code will not compile. In this case we can create a Base pointer that points to a Derived object (since a Derived object has all the private member variables of a Base object), but we cannot have a Derived pointer point to a Base object (since a Derived object may have more member variables than a Base object has). We are esentially creating a pointer that has more power (or more member variables) and so when we point to a Base object, we cannot access the member variables that would normally be in a Derived object.

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);
};
```