Discussion 3
Part 1
Basic knowledge about C++
1. C++ Inheritance

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
class derived-class: access-specifier base-class

```cpp
#include <iostream>

using namespace std;

// Base class
class Shape {
    public:
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
};

// Derived class
class Rectangle: public Shape {
    public:
        int getArea()
        {
            return (width * height);
        }
};

int main(void)
{
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << Rect/Area() << endl;
    return 0;
}
```
A derived class inherits all base class methods with the following exceptions:

Constructors, destructors and copy constructors of the base class.

Overloaded operators of the base class.

The friend functions of the base class.

<table>
<thead>
<tr>
<th>Access</th>
<th>public</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Derived classes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Outside classes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Type of Inheritance:

- When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

- We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

  - Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class’s private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

  - Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

  - Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.
Let’s read the code from PA2

using namespace std;

template <typename Data>
class RST : public BST<Data> {
};
2. Virtual function

- A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references.
class Animal
{
    public:
    void eat() { std::cout << "I'm eating generic food."; }
}

class Cat : public Animal
{
    public:
    void eat() { std::cout << "I'm eating a rat."; }
}

Animal *animal = new Animal;
Cat *cat = new Cat;

animal->eat();  // outputs: "I'm eating generic food."
cat->eat();     // outputs: "I'm eating a rat."
//this can go at the top of the main.cpp file
void func(Animal *xyz) { xyz->eat(); }

Now our main function is:

Animal *animal = new Animal;
Cat *cat = new Cat;

func(animal); // outputs: "I'm eating generic food."
func(cat);    // outputs: "I'm eating generic food."

The solution is to make `eat()` a virtual function:

```cpp
class Animal
{
public:
virtual void eat() { std::cout << "I'm eating generic food."; }
}
```

Main:

func(animal); // outputs: "I'm eating generic food."
func(cat);    // outputs: "I'm eating a rat."
Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.
Part2 PA2
- benchtree.cpp
- BST.hpp
- BSTIterator.hpp
- BSTNode.hpp
- countint.cpp
- countint.hpp
- Makefile
- **RST.hpp (This is the file you’ll need to modify)**
- test_RST.cpp
How can you start

- Start from insert()
- Inside insert() you need to:
  - Step1: create a new node
  - Step2: find the parent of the node
  - Step3: left/right rotate until we remain the priority property
  - Step4: return
void rotateLeft(BSTNode<Data>* par, BSTNode<Data>* child ){
    // Relocate *par, *par’s parent’s child is *child
    BSTNode<Data>* leftTree = child->left; // Get the left subtree of *child
    child->left = par; // *par is the left child of *child
    par->parent = child; // *child is the parent of *par
    par->right = leftTree; // And reattach the left subtree as the right child of *par
    if (leftTree != 0) {
        // *par is the parent of *leftTree
    }
}

Pseudorandom numbers in C++

- The `rand()` and `srand(int)` functions are defined in the C standard library, for use in C and C++ programs.

- The `rand()` function returns pseudorandom numbers in the range `[0,RAND_MAX]`
  - `#include <cstdlib>` to define the constant `RAND_MAX`

- `srand(x)` sets the seed for `rand()`’s generator.

- If `rand()` is called in your program before any call to `srand(int)`, it first calls `srand(1)`.

- Call `srand(int)` at any time to reset the seed.
  - call `srand(1)` to restart the default sequence.
  - `#include <ctime>`, and call `srand(time(0))` to get a seed that depends on the current system clock time, but only to 1 second resolution.

- **Put `srand()` at the top of your main function() and we can have the same random number every round.**