COMP 2012 Final Exam - Fall 2016 - HKUST

Date: December 12, 2016 (Monday)

Time Allowed: 3 hours, 12:30 – 03:30pm

Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **7** questions on **34** pages (including this cover page, appendix and the four rough worksheets).
3. Write your answers in the space provided.
4. Allocate your time wisely. Show all your work to earn full credits.
5. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.
6. For programming questions, you are **NOT** allowed to define additional helper functions or structures, nor global variables unless otherwise stated. You also **cannot** use any library functions not mentioned in the questions.

Student Name	Model Answer & Marking Scheme
Student ID	
Email Address	
Lecture Section	L1 (QUAN, Long) / L2 (TSOI, Desmond)
Seat Number	

For T.A. Use Only

Problem	Topic	Score
1	Type of Inheritance	/ 6
2	Polymorphism & Dynamic Binding	/ 14
3	Binary Search Tree (BST), AVL Tree & Hashing	/ 18
4	Class Template & Operator Overloading	/ 9
5	Standard Template Library (STL)	/ 8
6	Abstract Base Class, Inheritance & Polymorphism	/ 27
7	Binary Search Tree (BST)	/ 18
	Total	/ 100

Problem 1 [6 points] Type of Inheritance

There are six parts in this question. For each part, decide whether the given program is syntactically INCORRECT – that is, it will produce compilation error(s). Circle "Yes" if it will give compilation error and "No" otherwise.

Assume the programs in the different parts are independent of each other, and "B.h" has been included in each part and is defined as follows:

1 point for each correct answer of "Yes / No" question → total 6 points

```
class B { /* B.h */
  protected:
    int a;
};
```

(a) [1 point] Yes / No

```
class C : public B { };
class D : private C { public: D() : a(5) {} };
int main() { D obj; return 0; }
```

(b) [1 point] Yes / No

```
class C : public B { };
class D : private C { public: void set(int a) { this->a = a; } };
int main() { D obj; return 0; }
```

(c) [1 point] Yes / No

```
class C : protected B { };
class D : private C { public: void set(int a) { this->a = a; } };
int main() { D obj; return 0; }
```

(d) [1 point] Yes / No

```
class C : private B { };
class D : public C { public: void set(int a) { this->a = a; } };
int main() { D obj; return 0; }
```

(e) [1 point] Yes / No

```
class D : public B { };
int main() { D obj; B& ref = obj; return 0; }
```

(f) [1 point] Yes / No

```
class D : protected B { };
int main() { D obj; B& ref = obj; return 0; }
```

Problem 2 [14 points] Polymorphism and Dynamic Binding

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

class Base {
    int base;
    static int count;
  public:
    Base(int b = 30) : base(b) {
      count++;
      cout << "Base's Ctor, count = " << count << endl;
    }
    Base(const Base& b) : base(b.base) {
      count++;
      cout << "Base's Copy Ctor, count = " << count << endl;
    }
    virtual ~Base() {
      count--;
      cout << "Base's Dtor, count = " << count << endl;
    }
    virtual void set(int b) { base = b; }
    virtual int get() const { return base; }
    void work() const { cout << "Work in Base!" << endl; }
    virtual void print() const { cout << "Value of base:  " << base << endl; }
};

class Derived : public Base {
    static Base objBase;
    int derived;
  public:
    Derived(int d = 40) : derived(d) { cout << "Derived's Ctor" << endl; }
    ~Derived() { cout << "Derived's Dtor" << endl; }
    void work() const {
      cout << "Work in Derived!" << endl;
      Base::work();
      (Derived::objBase).set(50);
      (Derived::objBase).print();
    }
    void set(int d) { derived = d; }
```

```
    int get() const { return derived; }
    void print() const {
      Base::print();
      cout << "Value of derived:   " << derived << endl;
    }
    static void printStaticObjValue() {
      cout << "*** Static object's value:   " << objBase.get() << " ***" << endl;
    }
};

Base Derived::objBase;
int Base::count = 0;

int main() {
  cout << "--- Begin of main ---" << endl;
  Derived::printStaticObjValue();
  cout << "------ Block 1 ------" << endl;
  Base obj1(9);
  Derived obj2(10);
  { Base obj3 = obj2; obj3.print(); }
  cout << "------ Block 2 ------" << endl;
  Base* objPtr = new Derived(18);
  Base& objRef = obj2;
  cout << "------ Block 3 ------" << endl;
  objPtr->print();
  objPtr->work();
  cout << "------ Block 4 ------" << endl;
  ((Derived*)(objPtr))->print();
  ((Derived*)(objPtr))->work();
  cout << "------ Block 5 ------" << endl;
  objRef.print();
  objRef.work();
  cout << "------ Block 6 ------" << endl;
  delete objPtr;
  cout << "------ Block 7 ------" << endl;
  Derived::printStaticObjValue();
  cout << "---- End of main ----" << endl;
  return 0;
}
```

Write the outputs of the above program. Part of the outputs is given to you already; so only fill out the remaining outputs.

Answer:

```
Base's Ctor, count = 1
--- Begin of main ---
*** Static object's value: 30 ***
------ Block 1 ------
Base's Ctor, count = 2
Base's Ctor, count = 3
Derived's Ctor
Base's Copy Ctor, count = 4
Value of base: 30
Base's Dtor, count = 3
------ Block 2 ------
Base's Ctor, count = 4
Derived's Ctor
------ Block 3 ------
Value of base: 30
Value of derived: 18
Work in Base!
------ Block 4 ------
Value of base: 30
Value of derived: 18
Work in Derived!
Work in Base!
Value of base: 50
------ Block 5 ------
Value of base: 30
Value of derived: 10
Work in Base!
------ Block 6 ------
Derived's Dtor
Base's Dtor, count = 3
------ Block 7 ------
*** Static object's value: 50 ***
---- End of main ----
Derived's Dtor
Base's Dtor, count = 2
Base's Dtor, count = 1
Base's Dtor, count = 0
```

0.5 point for each line of output → total 14 points

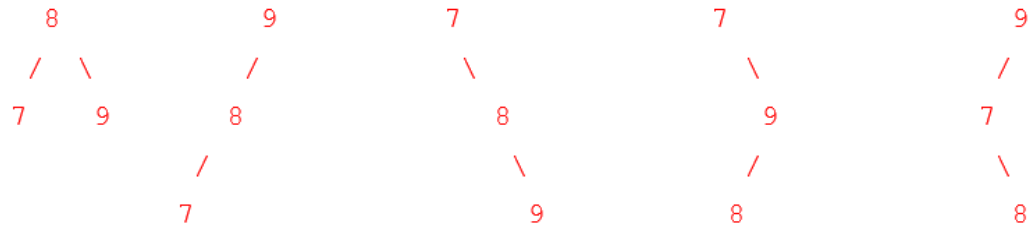
Problem 3 [18 points] Binary Search Tree (BST), AVL Tree and Hashing

(a) [6 points] Draw 5 different BSTs that can be constructed using the set of the following keys.

$$\{\ 7, 8, 9\ \}$$

Also, state whether there are only 5 different BSTs that can be constructed using the given set of keys.

Answer:



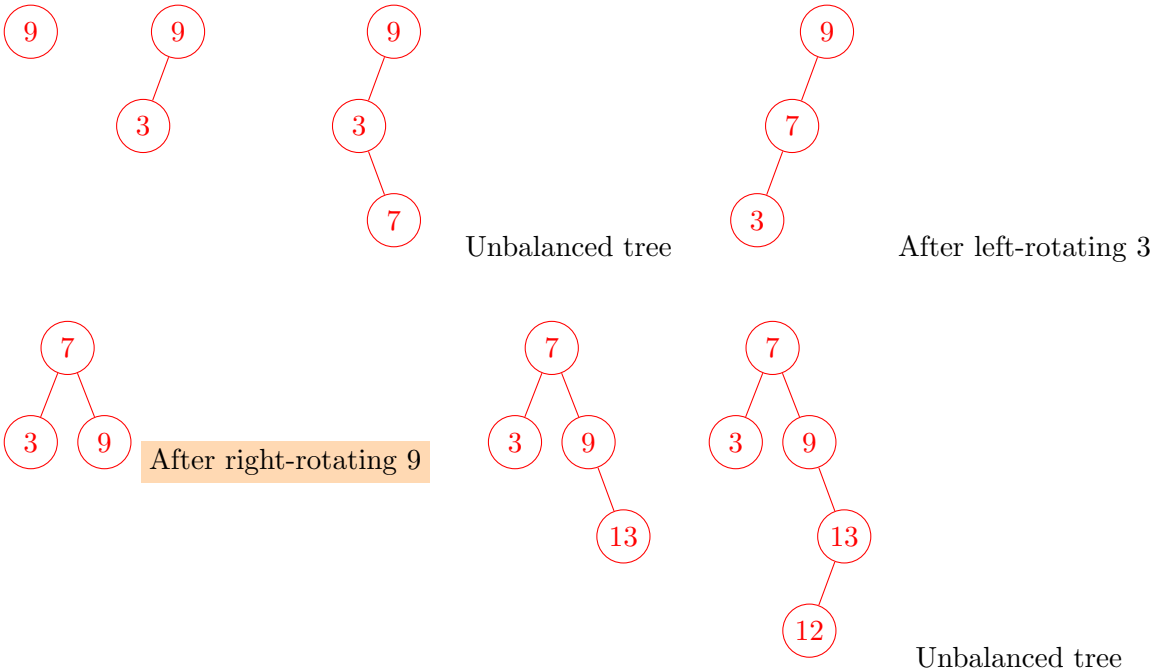
Yes, there are only 5 different BSTs can be constructed using the given set of keys.

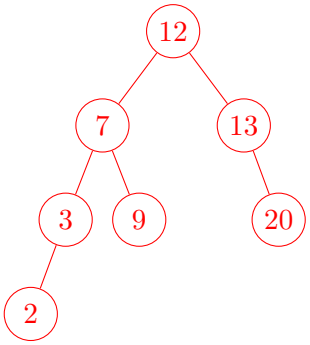
1 point for each BST → total 5 points
1 point for stating there are only 5 different BSTs

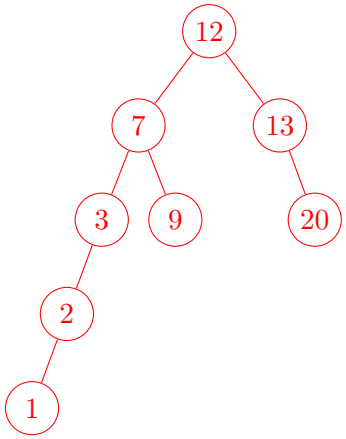
(b) [8 points] Draw the AVL tree resulting from the insertion of the following keys in given order (from left to right). Show the tree after each rotation, if any.

`9, 3, 7, 13, 12, 20, 2, 1`

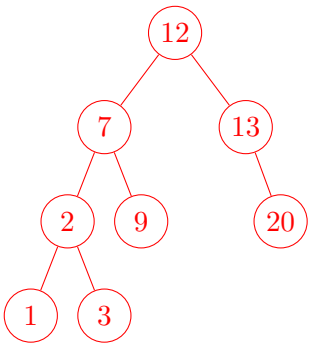
Answer:







Unbalanced tree



After right-rotating 3

3 points for showing the resulting tree marked with "After right-rotating 9"
3 points for showing the resulting tree marked with "After left-rotating 9"
1 point for showing the resulting tree marked with "After left-rotating 7"
1 point for showing the resulting tree marked with "After right-rotating 3"
→ total 8 points

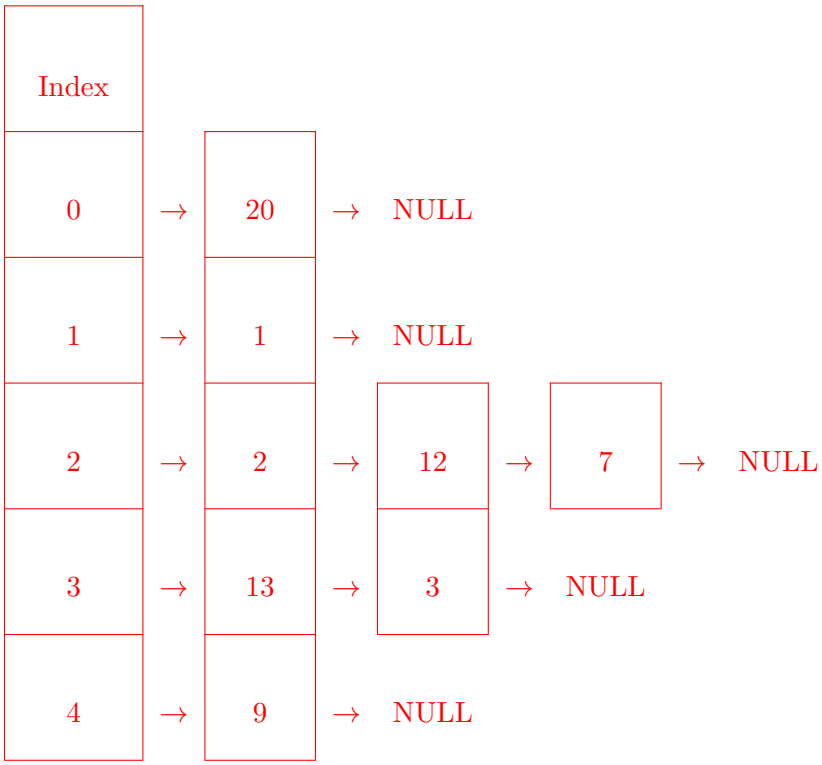
(c) [4 points] Now, a hash table of size $m = 5$ is used to store the same set of keys, that is

`9, 3, 7, 13, 12, 20, 2, 1`

with the hash function $hash(k) = k \bmod m$.

Insert the keys in the given order (from left to right) to the hash table using *separate chaining* collision resolution. Assume singly linked list is used for the chains and each key is inserted to the front. Draw the resulting hash table in the space given below.

Answer:



0.5 point for each correct insertion of number → total 4 points

Problem 4 [9 points] Class Template and Operator Overloading

Given the program "`Pair.cpp`" with incomplete definition and implementation of template class '`Pair`'. Complete the program by overloading the following member and non-member operator functions so that it produces the expected output given below when the program is compiled and executed.

- overload plus operator (i.e. `operator+`) as a member function (for adding two `Pair` objects)
- overload pre-increment and post-increment operator (i.e. `operator++`) as member functions (for adding 1 to each data member of a `Pair` object)
- overload insertion operator (i.e. `operator<<`) as non-member function (for outputting)

Write your answer in the space provided under Part (a)-(d) `"ADD YOUR CODE HERE"`.
Assumption: only numeric types will be used.

Expected output of the program:

```
Pair 1: (1,2.1)
Pair 2: (5,7.8)
Pair 3: (6,9.9)
Pair 3: (7,10.9)
Pair 3: (2,3.5)
Pair 3: (2,3.5)
Pair 3: (3,4.5)
```

```
#include <iostream>                                        /* Pair.cpp */
using namespace std;

template <typename T1, typename T2>
class Pair {
  private:
    T1 firstElement;
    T2 secondElement;
  public:
    Pair() {}
    Pair(const T1& e1, const T2& e2) { setElements(e1, e2); }
    T1 getFirstElement() const { return firstElement; }
    T2 getSecondElement() const { return secondElement; }
    void setElements(const T1& e1, const T2& e2) {
      firstElement = e1;
      secondElement = e2;
    }
```

```
    // Member function: plus operator, operator+
    // Part (a) - ADD YOUR CODE HERE

    Pair operator+(const Pair p) {                                    // 2 points
      return Pair(firstElement + p.firstElement, secondElement + p.secondElement);
    }

    // Member function: pre-increment operator, operator++
    // Part (b) - ADD YOUR CODE HERE

    Pair& operator++() {                                              // 2 points
      firstElement++;
      secondElement++;
      return *this;
    }

    // Member function: post-increment operator, operator++
    // Part (c) - ADD YOUR CODE HERE

    Pair operator++(int) {                                            // 2 points
      Pair temp = *this;
      firstElement++;
      secondElement++;
      return temp;
    }
};

// Non-member function: insertion operator, operator<<
// Part (d) - ADD YOUR CODE HERE

template <typename T1, typename T2>                                   // 3 points
ostream& operator<<(ostream& os, Pair<T1,T2> p) {
  os << "(" << p.getFirstElement() << "," << p.getSecondElement() << ")";
  return os;
}
```

```
int main() {
  Pair<int, double> pair1(1, 2.1), pair2(5, 7.8);
  cout ≪ "Pair 1:   " ≪ pair1 ≪ endl;
  cout ≪ "Pair 2:   " ≪ pair2 ≪ endl;
  Pair<int, double> pair3 = pair1 + pair2;
  cout ≪ "Pair 3:   " ≪ pair3 ≪ endl;
  ++pair3;
  cout ≪ "Pair 3:   " ≪ pair3 ≪ endl;
  ++pair3 = Pair<int, double>(2,3.5);
  cout ≪ "Pair 3:   " ≪ pair3 ≪ endl;
  cout ≪ "Pair 3:   " ≪ pair3++ ≪ endl;
  cout ≪ "Pair 3:   " ≪ pair3 ≪ endl;
  return 0;
}
```

For the sake of convenience, the expected output is replicated below.

```
Pair 1: (1,2.1)
Pair 2: (5,7.8)
Pair 3: (6,9.9)
Pair 3: (7,10.9)
Pair 3: (2,3.5)
Pair 3: (2,3.5)
Pair 3: (3,4.5)
```

Problem 5 [8 points] Standard Template Library (STL)

(a) [3 points] Given the following prototype of the STL function `for_each`

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last, UnaryFunction f);
```

implement the function which applies unary function f to each container element in the range [first, last) and returns the unary function after it has been applied to each element.

Answer:

```
template<class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last, UnaryFunction f)
{
  for(; first != last; ++first)                    // 1 point
    f(*first);                                     // 1 point
  return f;                                        // 1 point
}
```

(b) [5 points] for_each algorithm in STL accepts an user-defined function object or a function pointer. Implement a non-member function `sqr` and a function object class `Sqr`, both take an int reference and update the reference variable with its squared value. Write your code in the space provided under Part (b)(i) - (b)(ii) "`ADD YOUR CODE HERE`", so that the following program

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

typedef vector<int> vi;
typedef vector<int>::iterator vii;

// Non-member function: sqr
// Part (b)(i) - ADD YOUR CODE HERE

void sqr(int& i) {                                          // 1 point
  i = i*i;                                                  // 1 point
}

// Function object class: Sqr
// Part (b)(ii) - ADD YOUR CODE HERE

class Sqr {                                                 // 1 point
  public:
    void operator()(int& i) {                               // 1 point
      i = i*i;                                              // 1 point
    }
};
```

```
int main() {
  int arr[] = { 52, 61, 2, 6, 41, 312 };

  vi vec1(arr, arr+6);              // Create a vector container with all the 6 numbers
  cout ≪ "---- Using function pointer ----" ≪ endl;
  cout ≪ "vec1 (original values):\t";
  for(vii p = vec1.begin(); p != vec1.end(); p++)
    cout ≪ *p ≪ '\t';
  cout ≪ endl;

  for_each(vec1.begin(), vec1.end(), sqr);                    // Pass function pointer
  cout ≪ "vec1 (after sqr):\t";
  for(vii p = vec1.begin(); p != vec1.end(); p++)
    cout ≪ *p ≪ '\t';
  cout ≪ endl;

  cout ≪ "---- Using function object ----" ≪ endl;
  vi vec2(arr, arr+6);    // Create another vector container also with the 6 numbers
  cout ≪ "vec2 (original values):\t";
  for(vii p = vec2.begin(); p != vec2.end(); p++)
    cout ≪ *p ≪ '\t';
  cout ≪ endl;

  for_each(vec2.begin(), vec2.end(), Sqr());                  // Pass function object
  cout ≪ "vec2 (after sqr):\t";
  for(vii p = vec2.begin(); p != vec2.end(); p++)
    cout ≪ *p ≪ '\t';
  cout ≪ endl;
  return 0;
}
```

would produce the output below.

Output of the program:

```
---- Using function pointer ----
vec1 (original values): 52      61      2       6       41      312
vec1 (after sqr):       2704    3721    4       36      1681    97344
---- Using function object ----
vec2 (original values): 52      61      2       6       41      312
vec2 (after sqr):       2704    3721    4       36      1681    97344
```

Problem 6 [27 points] Abstract Base Class, Inheritance and Polymorphism

Given the definition of class '`Point`'.

```
#ifndef POINT_H                                          /* Point.h */
#define POINT_H

#include <cmath>
using namespace std;

class Point {
  private:
    double _x;
    double _y;
  public:
    Point(): _x(0), _y(0){};
    Point(double x, double y) : _x(x), _y(y){};
    Point& operator+=(const Point& pt){
        _x += pt._x;
        _y += pt._y;
        return *this;
    }
    void rotate(double angle) {
        double x = _x*cos(angle) - _y*sin(angle);
        double y = _x*sin(angle) + _y*cos(angle);
        _x = x;
        _y = y;
    }
    double get_x() const { return _x; }
    double get_y() const { return _y; }
    void set_x(double x) { _x = x; }
    void set_y(double y) { _y = y; }
    void print() const { cout << "(" << _x << ", " << _y << ")"; }
};

#endif
```

your task is to implement an abstract base class called '`Shape`', and two classes '`Circle`' and '`Polygon`' which are derived from '`Shape`' by public inheritance that will work with the testing program "shape-test.cpp" and produces the given output.

```
#include <iostream>                                          /* shape-test.cpp */
#include <vector>
#include "Point.h"
#include "Shape.h"
#include "Circle.h"
#include "Polygon.h"
using namespace std;

void printShape(Shape* s[], int size)
{
  for(int i=0; i<size; i++)
  {
    Point newCenter = s[i]->center();
    double area = s[i]->area();
    cout << "Shape " << i+1 << " - ";
    cout << ( ( typeid(*s[i]).name() == typeid(Circle).name() )
             ? "Circle:  " : "Polygon:  " );
    cout << endl;
    s[i]->print();
    cout << "Center:  ";
    newCenter.print();
    cout << endl;
    cout << "Area:  " << area << endl << endl;
  }
}
```

```
int main()                                                    /* shape-test.cpp (continued) */
{
  Shape* shapes[2];

  Point center(10,20);                                                    // For circle
  double radius = 5;
  shapes[0] = new Circle(center, radius);

  vector<Point> vertices;                                                 // For polygon
  vertices.push_back(Point(4,10));
  vertices.push_back(Point(9,7));
  vertices.push_back(Point(11,2));
  vertices.push_back(Point(2,2));
  shapes[1] = new Polygon(vertices);

  cout << "---- Before transformation ----" << endl;
  printShape(shapes, 2);

  Point moveAmt(2,2);                                     // Parameters for transformation
  double angle = 90;

  cout << "---- After translating (2,2) ----" << endl;
  for(int i=0; i<2; i++)
    shapes[i]->translate(moveAmt);
  printShape(shapes, 2);

  cout << "---- After rotating 90 ----" << endl;
  for(int i=0; i<2; i++)
    shapes[i]->rotate(angle);
  printShape(shapes, 2);

  for(int i=0; i<2; i++)
    delete shapes[i];
  return 0;
}
```

Output of the testing program:

```
---- Before transformation ----
Shape 1 - Circle:
Center: (10, 20)
Radius: 5
Center: (10, 20)
Area: 78.5398

Shape 2 - Polygon:
Vertices: (4, 10), (9, 7), (11, 2), (2, 2)
Center: (6.5, 5.25)
Area: 45.5

---- After translating (2,2) ----
Shape 1 - Circle:
Center: (12, 22)
Radius: 5
Center: (12, 22)
Area: 78.5398

Shape 2 - Polygon:
Vertices: (6, 12), (11, 9), (13, 4), (4, 4)
Center: (8.5, 7.25)
Area: 45.5

---- After rotating 90 ----
Shape 1 - Circle:
Center: (-25.0448, 0.87034)
Radius: 5
Center: (-25.0448, 0.87034)
Area: 78.5398

Shape 2 - Polygon:
Vertices: (-13.4164, -0.0129034), (-12.9748, 5.8013), (-9.40094, 9.82966), (-5.36828, 1.78369)
Center: (-10.2901, 4.35044)
Area: 45.5
```

(a) [6 points] Define the abstract base class `Shape` which will be saved in the "Shape.h". The class does not have any data members but contains 5 pure virtual functions: `center`, `area`, `translate`, `rotate` and `print`. Use const whenever possible. The following describes each function.

- `center`: Returns the center point of the shape, where the center is in type '`Point`'.
- `area`: Returns the area of shape. Assume that the returned value is in double type.
- `translate`: Takes a '`Point`' object which represents the amount of translation for the shape. It does not return anything.
- `rotate`: Takes a double type argument which represents the angle of rotation for the shape. It does not return anything.
- `print`: Prints all the data members of shape. It does not return anything.

Answer: /* File "Shape.h" */

```
#ifndef SHAPE_H
#define SHAPE_H

/* Shape.h */
class Shape {                                                    // 1 point
  public:
    virtual Point center() const = 0;                            // 1 point
    virtual void translate(Point c) = 0;                         // 1 point
    virtual void rotate(double angle) = 0;                       // 1 point
    virtual double area() const = 0;                             // 1 point
    virtual void print() const = 0;                              // 1 point
};

#endif
```

(b) [7 points] Define and implement the class `Circle` which will be saved in the "Circle.h" file. The `Circle` class is derived from `Shape` and has two data members: `center` (type: `Point`) and `radius` (type: double). Implement all the inherited member functions and a constructor to initialize all the data members.

Note: All the member functions should be implemented inside the class definition as inline member functions.
Hint:

- Area of circle is πr^2, where π is defined in C++ as M_PI. To use M_PI, the header file cmath should be included.
- The translation of circle is done by moving the center off the original position according to the amount of translation. Similarly, rotation of circle is done by rotating the center according to the specified angle.

Answer: /* File "Circle.h" */

```
#ifndef CIRCLE_H                                                    /* Circle.h */
#define CIRCLE_H

#include <iostream>
#include "Point.h"
#include "Shape.h"
using namespace std;

class Circle: public Shape {          // 1 point for defining Circle and data members
  public:
    Circle(const Point& c_, double r_) : c(c_), r(r_) { }                    // 1 point
    Point center() const { return c; }                                       // 1 point
    void translate(Point pt) { c += pt; }                                    // 1 point
    void rotate(double angle){ c.rotate(angle); }                            // 1 point
    double area() const { return r*r*M_PI; }                                 // 1 point
    void print() const {                                                     // 1 point
      cout << "Center:   ";
      c.print();
      cout << endl;
      cout << "Radius:   " << r << endl;
    }
  private:
    Point c;
    double r;
};
#endif
```

(c) [14 points] Define and implement the class `Polygon` which will be saved in the "Polygon.h". The `Polygon` class is derived from `Shape` and has a vector of vertex points (called `vertices` and in type vector<`Point`>). As in part (b), implement all the inherited member functions and a constructor to initialize the data member, all inside the class definition as inline member functions.

Hint:

- Assume you only have to deal with convex polygon, which the center and the area of the polygon can be computed as follows:

$$\text{Center} = \left(\frac{x_1 + x_2 + \ldots + x_n}{n}, \frac{y_1 + y_2 + \ldots + y_n}{n}\right)$$

$$\text{Area} = \frac{1}{2}\left|(x_1y_2 + x_2y_3 + x_3y_4 + \ldots + x_ny_1) - (y_1x_2 + y_2x_3 + y_3x_4 + \ldots + y_nx_1)\right|$$

  where $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are the vertices of polygon and $|\cdot|$ refers to absolute value function.
- The translation of polygon is done by moving each vertex off the original position according to the amount of translation. Similarly, rotation of polygon is done by rotating each vertex according to the specified angle.

Answer: /* File "Polygon.h" */

```
#ifndef POLYGON_H                                          /* Polygon.h */
#define POLYGON_H

#include <iostream>
#include <vector>
#include "Point.h"
#include "Shape.h"
using namespace std;

class Polygon: public Shape {     // 1 point for defining Polygon and data member
  public:
    Polygon(const vector<Point>& v_) : vertices(v_) { };                  // 1 point
    Point center() const {
      Point pt;
      for(int i=0; i < vertices.size(); i++)                              // 1 point
        pt += vertices[i];                                                // 1 point
      pt.set_x(pt.get_x()/vertices.size());                               // 0.5 point
      pt.set_y(pt.get_y()/vertices.size());                               // 0.5 point
      return pt;
    }
```

```
    void translate(Point pt) {                                    // 2 points
      for(int i=0; i < vertices.size(); i++)
        vertices[i] += pt;
    }
    void rotate(double angle){                                    // 2 points
      for(int i=0; i < vertices.size(); i++)
        vertices[i].rotate(angle);
    }
    double area() const {                                         // 3 points
      double s = 0;
      for(int i=0; i < vertices.size(); i++) {
        s += vertices[i].get_x() * vertices[(i+1)%vertices.size()].get_y();
        s -= vertices[i].get_y() * vertices[(i+1)%vertices.size()].get_x();
      }
      return fabs(s * 0.5);
    }
    void print() const {                                          // 2 points
      cout << "Vertices:  ";
      for(int i=0; i<vertices.size(); i++) {
        vertices[i].print();
        if(i != vertices.size() - 1)
          cout << ", ";
      }
      cout << endl;
    }

  private:
    vector<Point> vertices;
};

#endif
```

Problem 7 [18 points] Binary Search Tree (BST)

Complete the following program in the space provided under Part(a)-(d) "`ADD YOUR CODE HERE`". The whole program is assumed to be all written in a single source file. A sample run of the program is given below:

```
print:
        7
            6
    5
        4
3
    2
        1

findParentOf:
parent of 1 's value: 2
parent of 2 's value: 3
parent of 3 's value: -1
parent of 4 's value: 5
parent of 5 's value: 3
parent of 6 's value: 7
parent of 7 's value: 5
parent of 8 's value: -1
parent of 9 's value: -1
parent of 10 's value: -1

size:
7

findRankNItem:
rank 1 's value: 1
rank 2 's value: 2
rank 3 's value: 3
rank 4 's value: 4
rank 5 's value: 5
rank 6 's value: 6
rank 7 's value: 7
rank 8 's value: -1
rank 9 's value: -1
rank 10 's value: -1

findAllItemsLargerThanK:
5
6
7
```

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

template <typename T> struct bst_node              // A node in a binary search tree
{
  T value;                                                          // Node value.
  bst_node* left;                                                    // Left child.
  bst_node* right;                                                  // Right child.
  bst_node(const int& x) : value(x), left(NULL), right(NULL) { };
  ~bst_node() { delete left; delete right; }
};

template <typename T> class BST {
  private:
    bst_node<T>* root;
    void insert(bst_node<T>*& node, const T& x);
    void print(bst_node<T>* node, int depth = 0) const;
    bst_node<T>* findParentOf(bst_node<T>* node, const T& v) const;
    int size(bst_node<T>* node) const;
    bst_node<T>* findRankNItem(bst_node<T>* node, const int& n) const;
    void findAllItemsLargerThanK(bst_node<T>* node, const T& k,
                                 vector<bst_node<T>*>& result) const;
  public:
    BST() : root(NULL) {}   // Empty BST when its root is NULL
    ~BST() { delete root; }
    void insert(const T& x) { insert(root, x); }
    void print(int depth = 0) const { print(root, depth); }
    bst_node<T>* findParentOf(const T& v) const { return findParentOf(root, v); }
    int size() const { return size(root); }
    bst_node<T>* findRankNItem(const T& n) const {
      return findRankNItem(root, n);
    }
    void findAllItemsLargerThanK(const T& k,
                                 vector<bst_node<T>*>& result) const {
      findAllItemsLargerThanK(root, k, result);
    }
};
```

```
// Insert an item and preserve the BST property
template <typename T>
void BST<T>::insert(bst_node<T>*& node, const T& x)
{
  if(node == NULL)
    node = new bst_node<T>(x);
  else if(x < node->value)
    insert(node->left, x);
  else if(x > node->value)
    insert(node->right, x);
  else
    ;                                          // x == root->value; do nothing
}

// Print the BST
template <typename T>
void BST<T>::print(bst_node<T>* node, int depth) const
{
  if(node == NULL)                                                // Base case
    return;

  print(node->right, depth+1);                          // Recursion: right subtree

  for(int j = 0; j < depth; j++)                          // Print the node value
    cout << '\t';
  cout << node->value << endl;

  print(node->left, depth+1);                           // Recursion: left subtree
}
```

```
/* Part (a) [6 points] ADD YOUR CODE HERE
 * Implement the member function
 * bst_node<T>* findParentOf(bst_node<T>* node, const T& v) const
 * Return the pointer that points to the parent node of the node that has the value of v.
 * Return NULL if no node has the value of v.
 * Return NULL if root is NULL.
 * You CANNOT use stacks.
 * You CANNOT use recursion.
 */

template <typename T>
bst_node<T>* BST<T>::findParentOf(bst_node<T>* node, const T& v) const {
  if(node == NULL)                                                    // 0.5 point
    return NULL;
  bst_node<T>* parent = NULL;
  bst_node<T>* n = node;
  while(n != NULL) {                                                  // 1 point
    if(v == n->value)                                                 // 1 point
      return parent;                                                  // 1 point
    parent = n;
    if(v < n->value)                                                  // 1 point
      n = n->left;
    else if(v > n->value)                                             // 1 point
      n = n->right;
  }
  return NULL;                                                        // 0.5 point
}

/* Part (b) [3 points] ADD YOUR CODE HERE
 * Implement the member function int size(bst_node<T>* node) const
 * Return the size (i.e. number of nodes) of the whole BST.
 * You MUST use recursion.
 */

template <typename T>
int BST<T>::size(bst_node<T>* node) const {
  if(node == NULL) return 0;                                          // 1 point
  else return 1 + size(node->left) + size(node->right);               // 2 points
}
```

```
/* Part (c) [5 points] ADD YOUR CODE HERE
 * Implement the member function
 * bst_node<T>* findRankNItem(bst_node<T>* node, const T& n) const;
 * Return the pointer that points to the rank n node;
 * The node with the smallest value is of rank 1,
 * the node with the second smallest value is of rank 2, and so on.
 * You MUST use recursion.
 */

template <typename T>
bst_node<T>* BST<T>::findRankNItem(bst_node<T>* node, const T& n) const {
  if(node == NULL)                                                    // 1 point
    return NULL;
  int currentRank = size(node->left) + 1;                             // 1 point
  if(currentRank == n)                                                // 1 point
    return node;
  else if(currentRank > n)                                            // 1 point
    return findRankNItem(node->left, n);
  else                                                                // 1 point
    return findRankNItem(node->right, n - currentRank);
}

/* Part (d) [4 points] ADD YOUR CODE HERE
 * Implement the member function
 * void findAllItemsLargerThanK(bst_node<T>* node, const T& k,
 *                              vector<bst_node<T>*>& result)
 * Find all the nodes that have their values being larger than k
 * and add them to the result vector and make sure the nodes in the vector are sorted
 * ascendingly according to their values.
 * Assume the given vector is empty initially.
 * You MUST use recursion.
 */
template <typename T>
void BST<T>::findAllItemsLargerThanK(bst_node<T>* node, const T& k,
                                     vector<bst_node<T>*>& result) const {
  if(node == NULL) return;                                            // 1 point
  findAllItemsLargerThanK(node->left, k, result);                     // 1 point
  if(node->value > k) result.push_back(node);                         // 1 point
  findAllItemsLargerThanK(node->right, k, result);                    // 1 point
}
```

```
int main()
{
  BST<int> bst;
  bst.insert(3);
  bst.insert(5);
  bst.insert(7);
  bst.insert(2);
  bst.insert(4);
  bst.insert(6);
  bst.insert(1);

  cout ≪ "print:" ≪ endl;
  bst.print();

  cout ≪ endl ≪ "findParentOf:" ≪ endl;
  for(int i=1; i<=10; i++) {
    cout ≪ "parent of " ≪ i ≪ " 's value:  " ≪
      (bst.findParentOf(i)==NULL?-1:bst.findParentOf(i)→value) ≪ endl;
  }

  cout ≪ endl ≪ "size:" ≪ endl;
  cout ≪ bst.size() ≪ endl;

  cout ≪ endl ≪ "findRankNItem:" ≪ endl;
  for(int i=1; i<=10; i++) {
    cout ≪ "rank " ≪ i ≪ " 's value:  " ≪
      (bst.findRankNItem(i)==NULL?-1:bst.findRankNItem(i)→value) ≪ endl;
  }

  cout ≪ endl ≪ "findAllItemsLargerThanK:" ≪ endl;
  vector<bst_node<int>*> result;
  bst.findAllItemsLargerThanK(4, result);
  for(unsigned int i=0; i<result.size(); i++)
    cout ≪ result[i]→value ≪ endl;

  return 0;
}
```

Appendix

STL Sequence Container: Vector

```
template <class T, class Alloc = allocator<T> > class vector;
```

Defined in the standard header **vector**.

Description:

Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

The member functions in the vector<T> container class where T is the type of data stored in the vector are listed as follows:

Member function	**Description**
vector()	Constructor
iterator begin() const_iterator begin() const	Returns an iterator pointing to the first element in the vector. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
iterator end() const_iterator end() const	Returns an iterator referring to the past-the-end element in the vector container. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
void push_back(const T& val)	Adds a new element, val, at the end of the vector, after its current last element. The content of val is copied (or moved) to the new element.
size_type size() const	Returns the size of the vector

Useful Function in cmath

```
double fabs(double x);
```

- Returns the absolute value of x.

/* Rough work — You may detach this page */

/* Rough work — You may detach this page */

/* Rough work — You may detach this page */

/* Rough work — You may detach this page */