.

# COMP 2012H Final Exam - Fall 2018 - HKUST

| | |
|---|---|
| Date: | December 13, 2018 (Thursday) |
| Time Allowed: | 3 hours, 8:30am–11:30am |

Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **7** questions on **41** pages (including this cover page, appendix and 4 blank pages at the end) printed on **2** sets of papers:
   - Paper I: Description of problem 1 - 4 **AND** space for ALL your answers.
   - Paper II: Description of problem 5 - 7.
3. Write your answers in the space provided in paper I.
4. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.
5. For programming questions, unless otherwise stated, you are **NOT** allowed to define additional structures, classes, helper functions and use global variables, nor any library functions not mentioned in the questions.

| | |
|---|---|
| Student Name | SOLUTION & MARKING SCHEME |
| Student ID | |
| Email Address | |
| Seat Number | |

For T.A. Use Only

| Problem | Topic | Score |
|---|---|---|
| 1 | True or False Questions | / 10 |
| 2 | Function Objects and STL | / 8 |
| 3 | Order of Construction and Destruction | / 10.5 |
| 4 | AVL Tree | / 8.5 |
| 5 | Inheritance, Polymorphism and Dynamic Binding | / 26 |
| 6 | Binary Search Tree (BST) | / 17 |
| 7 | Hashing | / 20 |
| Total | | / 100 |

**Problem 1 [10 points] True or False Questions**

Indicate whether the following statements are *true* or *false* by circling **T** or **F**.

**T** **F** (a) Functions can be overloaded on the basis of const-ness of parameters only if the const parameter is a pointer or a reference.

**T** **F** (b) Initialization of non-static data members in class will proceed according to the order of the member initialization list.

**T** **F** (c) `this` pointer is an implicit parameter to ALL member functions.

**T** **F** (d) C++ DOES NOT prohibit abstract base class from providing a definition for the pure virtual function.

**T** **F** (e) The type of inheritance (public, protected or private) does NOT change the accessibility of members inherited from the base class in the member functions of the direct derived class.

**T** **F** (f) There is NO compilation error in the following program AND it outputs `A's func`.

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

class A {
  public:
    virtual void func(int a = 20, int b = 10) { cout << "A's func" << endl; }
};

class B : public A {
  public:
    void func(int a, int b = 10) { cout << "B's func" << endl; }
};

int main() {
  A* p = new B;
  p->func();
  return 0;
}
```

**T** **F** (g) `typeid` only gives correct type information of the specified expression if it refers to a class that declares or inherits at least one virtual function.

**T** **F** (h) There is NO compilation error in the following program.

```
template <typename T>
class A {
  public:
    T funcWithSyntaxError() {
      int a = 10;
      int* p = a; // This line has syntax error
      return a;
    }
};

int main() { A<int> obj; }
```

**T** **F** (i) A unique binary search tree can be constructed from a given preorder traversal sequence.

**T** **F** (j) An AVL tree is balanced, therefore the median of all elements in the tree is always at the root or one of its children.

**Problem 2 [8 points] Function Objects and STL**

(a) [3 points] Define a function object class `Line` (in the file "`func-obj-line.h`") that will work with the given test program "`test-line.cpp`" to determine the y value of a given x value using line equation, y = mx + c, where m is the slope, and c is the y-intercept.

For example, running the test program will give the following output:

```
For line: y = 1.0 * x + 1.0, x = 5.0, y = 6
For line: y = 1.0 * x + 2.0, x = 6.0, y = 8
```

```
#include <iostream>     /* File: test-line.cpp */
using namespace std;

#include "func-obj-line.h" // Define function objects class Line in this file

int main()
{
  Line line1;
  double y1 = line1(5.0); // y1 = 1.0 * 5.0 + 1.0 = 6.0
  cout << "For line: y = 1.0 * x + 1.0, x = 5.0, y = " << y1 << endl;

  Line line2(1.0, 2.0);
  double y2 = line2(6.0); // y2 = 1.0 * 6.0 + 2.0 = 8.0
  cout << "For line: y = 1.0 * x + 2.0, x = 6.0, y = " << y2 << endl;
  return 0;
}
```

The constructor of `Line` will initialize such function objects with the slope and y-intercept, so that when the function object is called with an x value, it will compute the y value with its "memorized" slope and y-intercept.

Note that the `Line` class should only have

- two private data members,
- one constructor, and
- one overloaded operator function.

**Answer:**

```
File: func-obj-line.h
// Complete the class definition of Line here

class Line
{
  private:
    double m;                         // 0.5 point
    double c;                         // 0.5 point

  public:
    Line(double m = 1.0, double c = 1.0) : m(m), c(c) { }  // 0.5 point

    double operator()(double x) {  // 1 point
      return m * x + c;            // 0.5 point
    }
};
```

(b) [3 points] Given the following prototype of the STL `transform` algorithm.

```
template <typename InputIterator, typename OutputIterator, typename UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation Op);
```

implement the algorithm so that it applies operation `Op` to each of the elements in the range `[first, last)` and store the value returned by each operation in the range that begins at `result`. It returns an iterator pointing to the element that follows the last element written in the `result` sequence.

**Answer:**

```
template <typename InputIterator, typename OutputIterator, typename UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation Op) {
  /* ADD YOUR CODE HERE */
  while (first != last) {      // 0.5 point
    *result = op(*first);      // 1 point
    ++result;                  // 0.5 point
    ++first;                   // 0.5 point
  }
  return result;               // 0.5 point
}
```

(c) [2 points] Complete the following program in the space provided after "`TODO:`" comment lines so that it will run and gives the output below:

```
For line: y = 5x - 9, x = 1.8, y = 0
For line: y = 5x - 9, x = 2.4, y = 3
For line: y = 5x - 9, x = 6.7, y = 24.5
For line: y = 5x - 9, x = -23.1, y = -124.5
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include "func-obj-line.h"
using namespace std;

int main()
{
  vector<double> x;
  x.push_back(1.8);
  x.push_back(2.4);
  x.push_back(6.7);
  x.push_back(-23.1);

  vector<double> y(x.size());

  /*
   *  TODO: Using transform from part (b) together with a function object of
   *        Line defined in part (a), find the y values corresponding to the
   *        x values in container x and store the results in container y.
   *        Assume the line equation we use for this question is y = 5x - 9.
   */

  // ------------------------------ CODE HERE ---------------------------------

  transform(x.begin(), x.end(), y.begin(), Line(5, -9));

  // 0.5 point for each part, i.e. x.begin(), x.end(), y.begin(), and Line(5,-9)
  // 2 points in total.

  // ----------------------------------------------------------------------------

  for(int i=0; i<x.size(); ++i)
    cout << "For line: y = 5x - 9, x = " << x[i] << ", y = " << y[i] << endl;
  return 0;
}
```

**Problem 3 [10.5 points] Order of Construction and Destruction**

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

class A {
  public:
    A() { cout << "A" << endl; }
    A(int a) { cout << "Conv A" << endl; }
    A(const A& a) { cout << "Copy A" << endl; }
    virtual ~A() { cout << "~A" << endl; }
};

class B : public A {
  public:
    B() { cout << "B" << endl; }
    B(const B& b) : A(b) { cout << "Copy B" << endl; }
    ~B() { cout << "~B" << endl; }
};

class C {
    static A a;
    static B b;
  public:
    C() { cout << "C" << endl; }
    C(int c) { cout << "Conv C" << endl; }
    C(const C& c) { cout << "Copy C" << endl; }
    virtual ~C() { cout << "~C" << endl; }
};

A obj(20);
A C::a(obj);

class D : public C {
  private:
    B b;
    A** a = new A*[2] { new A(b), new B(b) };
  public:
    D() { cout << "D" << endl; }
    D(int d) { cout << "Conv D" << endl; }
    D(const D& d) { cout << "Copy D" << endl; }
    ~D() {
      cout << "~D" << endl;
      for(int i=1; i>=0; i--)
        delete a[i];
      delete [] a;
    }
};
```

```
void process(const A aObj, const C cObj) { cout << "Processed" << endl; }

int main() {
  cout << "--- Block 1 ---" << endl;
  C cObj(D(10));
  cout << "--- Block 2 ---" << endl;
  process(10, 20);
  cout << "--- Block 3 ---" << endl;
  D dObj;
  cout << "--- Block 4 ---" << endl;
}
```

Write down the output of the above program when it is run. Some lines of outputs are already given. Assume the compiler DOES NOT do any optimization.

**Answer:**

```
Conv A
Copy A
--- Block 1 ---
C
A
B
Copy A
Copy A
Copy B
Conv D
Copy C
~D
~B
~A
~A
~B
~A
~C
--- Block 2 ---
Conv C
Copy C
Conv A
Copy A
Processed
~A
~A
```

```
~C
~C
--- Block 3 ---
C
A
B
Copy A
Copy A
Copy B
D
--- Block 4 ---
~D
~B
~A
~A
~B
~A
~C
~C
~A
~A
```

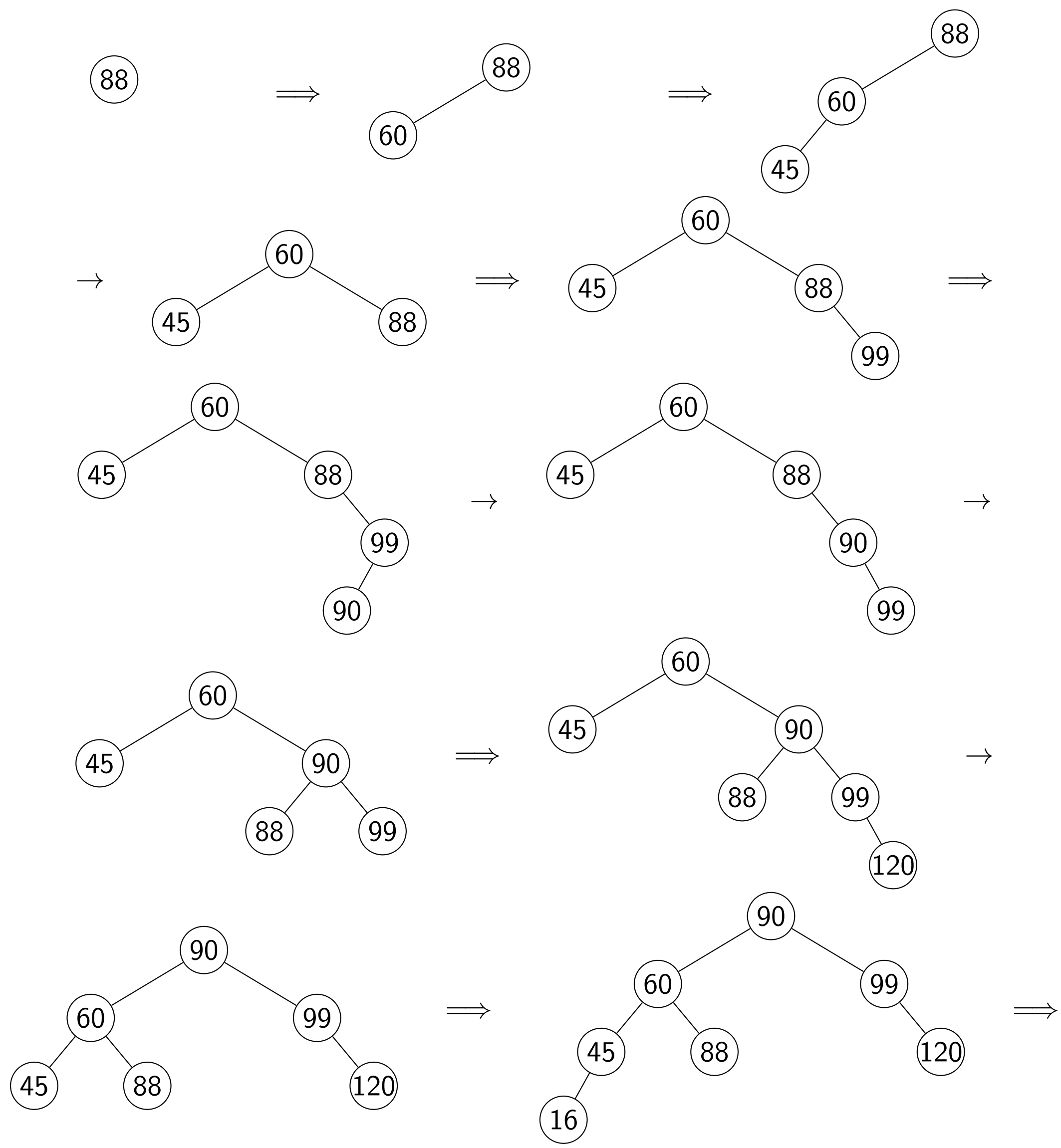
Marking scheme:

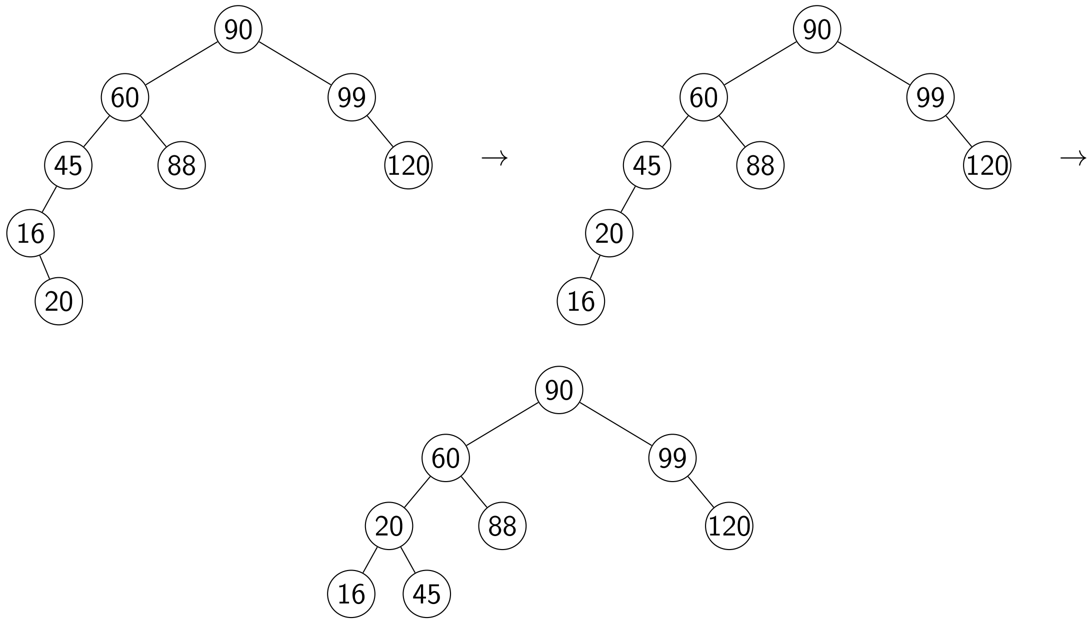
- 0.25 point for each statement before — Block 1 — (0.5 point in total)
- 0.25 point for each statement after — Block 1 — (3.75 points in total)
- 0.25 point for each statement (except Processed) after — Block 2 — (2 points in total)
- 0.25 point for each statement after — Block 3 — (1.75 points in total)
- 0.25 point for each statement after — Block 4 — (2.5 points in total)

**Problem 4 [8.5 points] AVL Tree**

(a) [7 points] Insert the following sequences of keys: 88, 60, 45, 99, 90, 120, 16, and 20 to an initially empty AVL tree. Draw all the intermediate trees (including the tree after insertion and each rotation, if any) and the final tree in the space provided below. You must use the algorithms discussed in class for inserting and re-balancing.

**Answer:**



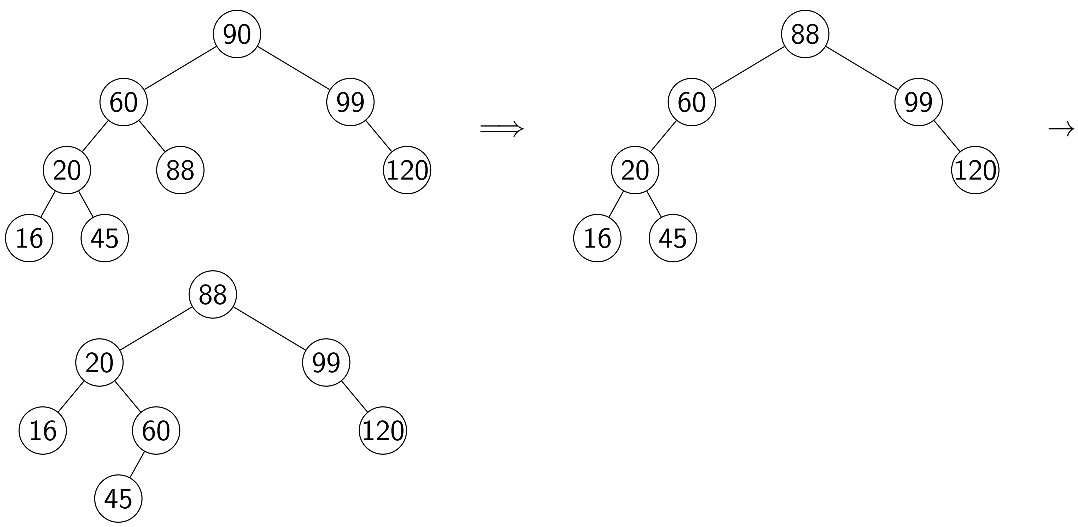



Marking scheme: 0.5 point for giving each tree. 14 trees, 7 points in total.

(b) [1.5 points] Delete the key 90 from the final AVL tree obtained from part (a). Show all the intermediate trees (including the tree after deletion and each rotation, if any) and the final tree in the space provided.

Note: If the node to be removed has 2 children, replace the node with an appropriate value from the node's left sub-tree.

**Answer:**




Marking scheme: 0.5 point for giving each tree. 3 trees, 1.5 points in total.

**Problem 5 [26 points] Inheritance, Polymorphism and Dynamic Binding**

This problem involves 4 classes called '`Fruit`', '`Banana`' (derived from '`Fruit`' using public inheritance), '`Pineapple`' (also derived from '`Fruit`' using public inheritance) and '`FruitSalad`'. Below are the header files of the 4 classes.

```
#ifndef FRUIT_H     /* File: Fruit.h */
#define FRUIT_H

#include <iostream>
#include <vector>    // For vector container
#include <typeinfo>  // For typeid
using namespace std;

enum Color { ORANGE, YELLOW, GREEN };
enum Size { SMALL, MEDIUM, BIG };

class Fruit {
  private:
    Color color;      // Color of the fruit
    double weight;    // Weight of the fruit
    double calories; // Calories of the fruit
    Size size;        // Size of the fruit
    bool hasCut;      // Has cut or not

  public:
    // A constructor that takes color, weight, calories, size and hasCut as
    // parameters to initialize all the data members using member initialization list.
    Fruit(Color color, double weight, double calories, Size size, bool hasCut)
        : color(color), weight(weight), calories(calories),
          size(size), hasCut(hasCut) { }

    // A pure virtual function.
    // It will be inherited and overridden by the Banana and Pineapple class.
    virtual void cut() = 0;

    // A virtual function that prints all the data members.
    virtual void print() const {
      cout << "Color: " << color << ", Weight: " << weight
           << ", Calories: " << calories << endl;
      cout << "Size: ";
      switch(size) {
        case SMALL: cout << "SMALL" << endl; break;
        case MEDIUM: cout << "MEDIUM" << endl; break;
        case BIG: cout << "BIG" << endl; break;
      }
      cout << "Cut? " << ((hasCut) ? "Yes" : "No") << endl;
    }
```

```
    // Accessor functions of all the data members.
    Color getColor() const { return color; }
    double getWeight() const { return weight; }
    double getCalories() const { return calories; }
    Size getSize() const { return size; }
    bool getHasCut() const { return hasCut; }
};

#endif /* FRUIT_H */
```

```
#ifndef BANANA_H     /* File: Banana.h */
#define BANANA_H

#include "Fruit.h"

class Banana : public Fruit {  // Derived from Fruit using public inheritance
  private:
    double amountFiber; // Amount of fiber in the banana

  public:
    // TODO (a)(i): Constructor, implement it in Banana.cpp
    // It takes color, weight, calories, size, hasCut and
    // amount of fiber as parameters to initialize all the data members
    // (including those inherited from Fruit).
    Banana(Color color, double weight, double calories,
           Size size, bool hasCut, double amountFiber);

    // TODO (a)(ii): Override cut virutal member function in Banana.cpp
    // - It prints a string describing the cutting method of banana.
    //   Refer to the sample output for this.
    // - Set inherited member hasCut to true.
    void cut();

    // TODO (a)(iii): Override print virtual member function in Banana.cpp
    // - It prints "=== Banana ===" and all the data members on screen.
    //   Refer to the sample output for this.
    void print() const;
};

#endif /* BANANA_H */
```

```cpp
#ifndef PINEAPPLE_H     /* File: Pineapple.h */
#define PINEAPPLE_H

#include "Fruit.h"

class Pineapple : public Fruit {  // Derived from Fruit using public inheritance
  private:
    double percentageJuice; // Percentage of juice in the pineapple

  public:
    // TODO (b)(i): Constructor, implement it in Pineapple.cpp
    // It takes color, weight, calories, size, hasCut and
    // percentage of juice as parameters to initialize all the data members
    // (including those inherited from Fruit).
    Pineapple(Color color, double weight, double calories,
              Size size, bool hasCut, double percentageJuice);

    // TODO (b)(ii): Override cut virtual member function
    // - It prints a string describing the cutting method of pineapple.
    //   Refer to the sample output for this.
    // - Set inherited member hasCut to true.
    void cut();

    // TODO (b)(iii): Override print virtual member function
    // - It prints "=== Pineapple ===" and all the data members on screen.
    //   Refer to the sample output for this.
    void print() const;
};

#endif /* PINEAPPLE_H */
```

```cpp
#ifndef FRUITSALAD_H     /* File: FruitSalad.h */
#define FRUITSALAD_H

#include "Fruit.h"
#include "Banana.h"
#include "Pineapple.h"

class FruitSalad {
    // Make operator<< as a friend of FruitSalad class
    friend ostream& operator<<(ostream& os, const FruitSalad& fs);

  private:
    vector<Fruit*> fruits; // A vector container that stores addresses of Fruits
    int numOfFruits;       // Number of fruit pointers in the vector container
```

```
  public:
    // TODO (c)(i): Implement default constructor in FruitSalad.cpp
    // - It initalizes numOfFruits to 0.
    FruitSalad();

    // TODO (c)(ii): Implement copy constructor in FruitSalad.cpp
    // - It performs deep copy.
    FruitSalad(const FruitSalad& fs);

    // TODO (c)(iii): Implement destructor in FruitSalad.cpp
    // It deallocates all dynamically allocated memory to avoid
    // any memory leak.
    ~FruitSalad();

    // TODO (c)(vi): Overload operator= for FruitSalad class in FruitSalad.cpp
    // - It initializes numOfFruits using the data member of the given object.
    // - It performs deep copy of vector container, i.e.
    //   * Each object pointed by the fruits container of fs should be cloned
    //     dynamically and the address of the cloned object will be stored in
    //     fruits container of the current object.
    //     - Note:
    //       Two different types of objects will be pointed by the vector container.
    //       Create Banana object when the object to be cloned is in Banana type.
    //       Create Pineapple object when the object to be cloned is in Pineapple type.
    //       [ You may find dynamic_cast<new type>(<expression>),
    //         typeid(<type>) / typeid(<expression>) useful for this. ]
    // # You should also add other statement(s) in the function to make sure
    //   that the program compiles and has no memory leak.
    FruitSalad& operator=(const FruitSalad& fs);

    // TODO (c)(v): Overload operator+= for FruitSalad class in FruitSalad.cpp
    // - It dynamically creates an object with the same content as f.
    //   [ You may find dynamic_cast<new type>(<expression>),
    //     typeid(<type>) / typeid(<expression>) useful for this. ]
    // - Add the address of the created dynamic object to the back of the vector
    //   container fruits.
    // - Increase the data member numOfFruits by 1.
    // # You should also add other statement(s) in the function to make sure
    //   that the program compiles.
    FruitSalad& operator+=(const Fruit& f);
};

#endif /* FRUITSALAD_H */
```

Below is the testing program "`test-fruit.cpp`".

```
#include "Fruit.h"     /* File: test-fruit.cpp */
#include "Banana.h"
#include "Pineapple.h"
#include "FruitSalad.h"
using namespace std;

// TODO d(i): Define operator<< for Fruit class here.
// It polymorphically invokes print function.

// TODO d(ii): Define operator<< for FruitSalad class here.
// It prints all the fruits in the vector container of fruitsalad.

int main() {
  cout << "Construct a Banana object: " << endl;
  Banana b1(YELLOW, 1.5, 32.5, SMALL, false, 0.7);
  cout << b1;
  cout << "How to cut the Banana? ";
  b1.cut();
  cout << endl;

  cout << "Construct a Pineapple object: " << endl;
  Pineapple p1(YELLOW, 7.9, 22.1, BIG, false, 0.5);
  cout << p1;
  cout << "How to cut the Pineapple? ";
  p1.cut();
  cout << endl;

  FruitSalad* fsp1 = new FruitSalad;
  cout << "Add a banana to fruit salad!" << endl;
  (*fsp1) += b1;
  cout << "Add a pineapple to fruit salad!" << endl;
  (*fsp1) += p1;
  cout << endl;

  cout << "Fruit salad consists of the following: " << endl;
  cout << *fsp1 << endl;

  cout << "Prepare another fruit salad same as the first one!!!" << endl << endl;
  FruitSalad fs2(*fsp1);

  cout << "Eat the first fruit salad" << endl << endl;
  delete fsp1;

  cout << "The second fruit salad consists of the following: " << endl;
  cout << fs2 << endl;

  return 0;
}
```

A sample run of the test program is given as follows:

Output of the testing program

```
Construct a Banana object:
=== Banana ===
Color: 1, Weight: 1.5, Calories: 32.5
Size: SMALL
Cut? No
Amount of fiber: 0.7
How to cut the Banana? Slicing in a grid

Construct a Pineapple object:
=== Pineapple ===
Color: 1, Weight: 7.9, Calories: 22.1
Size: BIG
Cut? No
Percentage of juice: 0.5
How to cut the Pineapple? Cutting into rings and then into pieces

Add a banana to fruit salad!
Add a pineapple to fruit salad!

Fruit salad consists of the following:
=== Banana ===
Color: 1, Weight: 1.5, Calories: 32.5
Size: SMALL
Cut? Yes
Amount of fiber: 0.7
=== Pineapple ===
Color: 1, Weight: 7.9, Calories: 22.1
Size: BIG
Cut? Yes
Percentage of juice: 0.5

Prepare another fruit salad same as the first one!!!

Eat the first fruit salad

The second fruit salad consists of the following:
=== Banana ===
Color: 1, Weight: 1.5, Calories: 32.5
Size: SMALL
Cut? Yes
Amount of fiber: 0.7
=== Pineapple ===
Color: 1, Weight: 7.9, Calories: 22.1
Size: BIG
Cut? Yes
Percentage of juice: 0.5
```

Based on the given information, complete the implementation of 'Banana' class, 'Pineapple' class and 'FruitSalad' class in their respective .cpp files, namely "Banana.cpp", 'Pineapple.cpp', 'FruitSalad.cpp' respectively, and implement the missing operator functions in "test-fruit.cpp".

(a) [5 points] Complete the 'Banana' class by implementing all the following member functions.

(i)
```
Banana(Color color, double weight, double calories,
       Size size, bool hasCut, double amountFiber);
```

**Answer:**

```
Banana::Banana(Color color, double weight, double calories,
               Size size, bool hasCut, double amountFiber)
       : Fruit(color, weight, calories, size, hasCut),       // 0.5 point
         amountFiber(amountFiber) { }                        // 0.5 point
```

(ii)
```
void cut();
```

**Answer:**

```
void Banana::cut() {
  cout << "Slicing in a grid" << endl;                       // 0.5 point
  *this = Banana(getColor(), getWeight(), getCalories(),     // 2 points
                 getSize(), true, amountFiber);
}
```

(iii)
```
void print() const;
```

**Answer:**

```
void Banana::print() const {
  cout << "=== Banana ===" << endl;                          // 0.5 point
  Fruit::print();                                            // 0.5 point
  cout << "Amount of fiber: " << amountFiber << endl;        // 0.5 point
}
```

(b) [5 points] Complete the 'Pineapple' class by implementing all the following member functions.

(i) `Pineapple(Color color, double weight, double calories, Size size, bool hasCut, double percentageJuice);`

**Answer:**

```
Pineapple::Pineapple(Color color, double weight, double calories,
                     Size size, bool hasCut, double percentageJuice)
       : Fruit(color, weight, calories, size, hasCut),        // 0.5 point
         percentageJuice(percentageJuice) { }                 // 0.5 point
```

(ii) `void cut();`

**Answer:**

```
void Pineapple::cut() {
  cout << "Cutting into rings and then into pieces" << endl;  // 0.5 point
  *this = Pineapple(getColor(), getWeight(), getCalories(),   // 2 points
                    getSize(), true, percentageJuice);
}
```

(iii) `void print() const;`

**Answer:**

```
void Pineapple::print() const {
  cout << "=== Pineapple ===" << endl;                         // 0.5 point
  Fruit::print();                                             // 0.5 point
  cout << "Percentage of juice: " << percentageJuice << endl; // 0.5 point
}
```

(c) [14 points]Complete the '`FruitSalad`' class by implementing all the following member functions.

(i) `FruitSalad();`

**Answer:**

```
FruitSalad::FruitSalad() : numOfFruits(0) { }                     // 0.5 point
```

(ii) `FruitSalad(const FruitSalad& fs);`

**Answer:**

```
FruitSalad::FruitSalad(const FruitSalad& fs) {
  numOfFruits = 0; // This is important!                          // 0.5 point
  *this = fs;                                                     // 0.5 point
}
```

(iii) `~FruitSalad();`

**Answer:**

```
FruitSalad::~FruitSalad() {
  for(int i=0; i<numOfFruits; ++i)                                // 0.5 point
    delete fruits[i];                                             // 0.5 point
}
```

(iv) `FruitSalad& operator=(const FruitSalad& fs);`

**Answer:**

```
FruitSalad& FruitSalad::operator=(const FruitSalad& fs) {
  if(this != &fs) {                                          // 0.5 point
    for(int i=0; i<numOfFruits; ++i)                         // 0.5 point
      delete fruits[i];                                      // 0.5 point
    fruits.clear();                                          // 0.5 point

    for(int i=0; i<fs.numOfFruits; ++i) {                    // 0.5 point
      if(typeid(*fs.fruits[i]) == typeid(Banana))            // 0.5 point
        fruits.push_back(new Banana(                         // 1.5 points
                         *dynamic_cast<const Banana*>(fs.fruits[i])));
      else
        fruits.push_back(new Pineapple(                      // 1.5 points
                         *dynamic_cast<const Pineapple*>(fs.fruits[i])));
    }
    numOfFruits = fs.numOfFruits;                            // 0.5 point
  }
  return *this;                                              // 0.5 point
}
```

(v) `FruitSalad& operator+=(const Fruit& f);`

**Answer:**

```
FruitSalad& FruitSalad::operator+=(const Fruit& f) {
  if(typeid(f) == typeid(Banana))                            // 0.5 point
    fruits.push_back(new Banana(
                     dynamic_cast<const Banana&>(f)));       // 1.5 points
  else
    fruits.push_back(new Pineapple(                          // 1.5 points
                     dynamic_cast<const Pineapple&>(f)));
  ++numOfFruits;                                             // 0.5 point
  return *this;                                              // 0.5 point
}
```

(d) [2 points] Complete the "`test-fruit.cpp`" by overloading all the following operator functions.

(i) `ostream& operator<<(ostream& os, const Fruit& f);`

**Answer:**

```
ostream& operator<<(ostream& os, const Fruit& f) {
  f.print();                                              // 0.5 point
  return os;                                              // 0.5 point
}
```

(ii) `ostream& operator<<(ostream& os, const FruitSalad& fs);`

**Answer:**

```
ostream& operator<<(ostream& os, const FruitSalad& fs) {
  for(int i=0; i<fs.numOfFruits; ++i)                     // 0.5 point
    os << *(fs.fruits[i]);                                // 0.5 point
  return os;
}
```

**Problem 6 [17 points] Binary Search Tree (BST)**

Given "`bst.h`", implement all the missing member functions of BST class template in "`bst-more.tpp`" according to the details given under Part (a)-(e) so that the class template will work with the testing program "`test-bst.cpp`" and produce the given output.

```
template <typename T>     /* File: bst.h */
class BST {
  private:
    struct BSTnode {   // A node in a binary search tree
      T value;
      BST left;        // Left sub-tree or called left child
      BST right;       // Right sub-tree or called right child
      BSTnode(const T& x) : value(x), left(), right() { }
                            // Assume a copy constructor for T
      BSTnode(const BSTnode& node) // Copy constructor
          : value(node.value), left(node.left), right(node.right) { }
      ~BSTnode() { }
    };
    BSTnode* root = nullptr;

  public:
    BST() = default;              // Empty BST
    ~BST() { delete root; }       // Actually recursive
    // Shallow BST copy using move constructor
    BST(BST&& bst) { root = bst.root; bst.root = nullptr; }

    BST(const BST& bst) {         // Deep copy using copy constructor
        if (bst.is_empty())
            return;
        root = new BSTnode(*bst.root);  // Recursive
    }

    bool is_empty() const { return root == nullptr; }
    void print(int depth = 0) const;
    const T& find_min() const;    // Find the minimum value

    void insert(const T&);        // Insert an item with a policy

    // TODO (a): Implement find_lca in bst-more.cpp
    const T& find_lca(const T& x, const T& y) const;
    // TODO (b): Implement find_cell in bst-more.cpp
    const T& find_ceil(const T& x, const T& minVal) const;
    // TODO (c): Implement sum_left_boundary_nodes in bst-more.cpp
    T find_sum_left_boundary_nodes() const;
    // TODO (d): Implement sum_right_boundary_nodes in bst-more.cpp
    T find_sum_right_boundary_nodes() const;
    // TODO (e): Implement find_sum_boundary_nodes in bst-more.cpp
    T find_sum_boundary_nodes() const;
};
```

```cpp
#include <iostream>     /* File: test-bst.cpp */
#include <climits>
using namespace std;
#include "bst.h"
#include "bst-print.tpp"
#include "bst-find-min.tpp"
#include "bst-insert.tpp"
#include "bst-more.tpp"

void print(int ceilVal, int minValMinusOne) {
  cout << ( (ceilVal == minValMinusOne) ? "Not found" : to_string(ceilVal) ) << endl;
}

int main() {
  BST<int> bst;
  bst.insert(25);  bst.insert(20);  bst.insert(36);  bst.insert(10);
  bst.insert(22);  bst.insert(30);  bst.insert(40);  bst.insert(5);
  bst.insert(12);  bst.insert(28);  bst.insert(38);  bst.insert(48);
  bst.insert(1);   bst.insert(8);   bst.insert(15);  bst.insert(45);

  cout << "Insert 25, 20, 36, 10, 22, 30, 40, 5, 12, 28, 38, 48, 1, 8, 15, 45, 50";
  cout << endl << endl;
  cout << "[ BST tree ]" << endl;
  bst.print();

  cout << endl << endl;
  cout << "Lowest Common Ancester of 5 and 22: " << bst.find_lca(5, 22) << endl;
  cout << "Lowest Common Ancester of 10 and 12: " << bst.find_lca(10, 12) << endl;
  cout << "Lowest Common Ancester of 5 and 45: " << bst.find_lca(5, 45) << endl;
  cout << "Lowest Common Ancester of 22 and 22: " << bst.find_lca(22, 22) << endl;
  cout << endl;

  int minValMinusOne = bst.find_min() - 1;
  int ceilVal = bst.find_ceil(29, minValMinusOne);
  cout << "Ceil of 29: ";
  print(ceilVal, minValMinusOne);
  ceilVal = bst.find_ceil(36, minValMinusOne);
  cout << "Ceil of 36: ";
  print(ceilVal, minValMinusOne);
  ceilVal = bst.find_ceil(50, minValMinusOne);
  cout << "Ceil of 50: ";
  print(ceilVal, minValMinusOne);
  ceilVal = bst.find_ceil(80, minValMinusOne);
  cout << "Ceil of 80: ";
  print(ceilVal, minValMinusOne);

  cout << endl;
  int sumL = bst.find_sum_left_boundary_nodes();
  int sumR = bst.find_sum_right_boundary_nodes();
  int sum = bst.find_sum_boundary_nodes();
  cout << "Sum of left boundary nodes: " << sumL << endl;
  cout << "Sum of right boundary nodes: " << sumR << endl;
  cout << "Sum of boundary nodes: " << sum << endl;
}
```

Output of the testing program

```
Insert 25, 20, 36, 10, 22, 30, 40, 5, 12, 28, 38, 48, 1, 8, 15, 45

[ BST tree ]
                        48
                                45
                40
                        38
        36
                30
                        28
25
                22
        20
                                15
                        12
                10
                                8
                        5
                                1


Lowest Common Ancester of 5 and 22: 20
Lowest Common Ancester of 10 and 12: 10
Lowest Common Ancester of 5 and 45: 25
Lowest Common Ancester of 22 and 22: 22

Ceil of 29: 30
Ceil of 36: 36
Ceil of 50: Not found
Ceil of 80: Not found

Sum of left boundary nodes: 61
Sum of right boundary nodes: 194
Sum of boundary nodes: 230
```

(a) [3 points] Implement

```
const T& find_lca(const T& x, const T& y) const;
```

to find the Lowest Common Ancestor of x and y. The lowest common ancestor of x and y is the shared ancestor of x and y that is located farthest from root.
Note: Recursion must be used for this question.

**Answer:** /* File: bst-more.tpp */

```
template <typename T>
const T& BST<T>::find_lca(const T& x, const T& y) const {       // 0.5 point
  if(root->value > x && root->value > y)                          // 0.5 point
    return root->left.find_lca(x, y);                             // 0.5 point
  if(root->value < x && root->value < y)                          // 0.5 point
    return root->right.find_lca(x, y);                            // 0.5 point
  return root->value;                                             // 0.5 point
}
```

(b) [5 points] Implement

```
const T& find_ceil(const T& x, const T& minVal) const;
```

to find the ceiling of x. The ceiling of x is defined as the value equal to next greater key in the BST, if exists. If such a value does not exist, return `minVal`. If `x` is in the BST, then ceil is equal to `x`.
Note: Recursion must be used for this question.

**Answer:** /* File: bst-more.tpp */

```
template <typename T>
const T& BST<T>::find_ceil(const T& x, const T& minVal) const {  // 0.5 point
  if(!root)                                                       // 0.5 point
    return minVal;                                                // 0.5 point
  if(root->value == x)                                            // 0.5 point
    return root->value;                                           // 0.5 point
  if(root->value < x)                                             // 0.5 point
    return root->right.find_ceil(x, minVal);                      // 0.5 point
  T value = root->left.find_ceil(x, minVal);                      // 0.5 point
  return (value >= x) ? value : root->value;                      // 1 point
}
```

(c) [3 points] Implement

```
T find_sum_left_boundary_nodes() const;
```

to find the sum of all nodes on the left boundary in a BST.

Note: Recursion must be used for this question.

**Answer:** `/* File: bst-more.tpp */`

```
template <typename T>
T BST<T>::find_sum_left_boundary_nodes() const {                        // 0.5 point
  if(root->left.is_empty() && root->right.is_empty())                   // 0.5 point
    return root->value;                                                  // 0.5 point
  else if(!root->left.is_empty())                                        // 0.5 point
    return root->value + root->left.find_sum_left_boundary_nodes();     // 0.5 point
  else
    return root->value + root->right.find_sum_left_boundary_nodes();    // 0.5 point
}
```

(d) [3 points] Implement

```
T find_sum_right_boundary_nodes() const;
```

to find the sum of all nodes on the right boundary in a BST.

Note: Recursion must be used for this question.

**Answer:** `/* File: bst-more.tpp */`

```
template <typename T>
T BST<T>::find_sum_right_boundary_nodes() const {                        // 0.5 point
  if(root->left.is_empty() && root->right.is_empty())                    // 0.5 point
    return root->value;                                                   // 0.5 point
  else if(!root->right.is_empty())                                        // 0.5 point
    return root->value + root->right.find_sum_right_boundary_nodes();    // 0.5 point
  else
    return root->value + root->left.find_sum_right_boundary_nodes();     // 0.5 point
}
```

(e) [3 points] Implement

```
T find_sum_boundary_nodes() const;
```

to find the sum of all nodes on the boundary in a BST.

Note: Recursion must be used for this question.

**Answer:** `/* File: bst-more.tpp */`

```
template <typename T>
T BST<T>::find_sum_boundary_nodes() const {                              // 0.5 point
  T l_sum, r_sum;
  if(!root->left.is_empty())                                             // 0.5 point
    l_sum = root->left.find_sum_left_boundary_nodes();                   // 0.5 point
  if(!root->right.is_empty())                                            // 0.5 point
    r_sum = root->right.find_sum_right_boundary_nodes();                 // 0.5 point
  return root->value + l_sum + r_sum;                                    // 0.5 point
}
```

**Problem 7 [20 points] Hashing**

This question is about an implementation of a hash table using open addressing, which involves 1 class and 1 structure, namely '`Hash`' and '`HashCell`' as shown below.

```
#include <iostream>     /* File: Hash.h */
using namespace std;

enum Status { DELETED = -1, EMPTY, ACTIVE };

// isPrime checks whether n is prime
bool isPrime(int n);

// nextPrime returns the smallest prime number that is greater than n
int nextPrime(int n);

template <typename T, typename KeyType>
class Hash {
  // Make operator<< as a friend of Hash
  friend ostream& operator<<(ostream& os, const Hash& hash) {
    for(int i=0; i<hash.capacity; ++i) {
      if(hash.table[i].flag == ACTIVE)  // Only print those marked with ACTIVE
        os << "Index = " << i << ": " << *(hash.table[i].data);
    }
    return os;
  }

  private:
    struct HashCell {  // HashCell refers to a cell in the hash table
      T* data;         // A pointer to data in type T
      Status flag;     // Status of the cell which could be DELETED, EMPTY or ACTIVE
      HashCell() : data(nullptr), flag(EMPTY) { } // Default constructor
      ~HashCell() { delete data; } // Destructor which de-allocates dynamic data
    };

    int capacity;                     // Capacity of hash table
    int(*hashFunction)(int,int);      // Hash function
    int(*offsetFunction)(int,int);    // Offset function for probing
    HashCell* table;                  // A pointer to an array representing a hash table
    int size;                         // Number of occupied cells in the hash table

    // Accessor function of size
    int getSize() const { return size; }

    // Function to check if the hash table is empty
    bool isEmpty() const { return size == 0; }

    // Function to check if the hash table is half-full
    bool isHalfFull() const { return size > (capacity / 2); }

  public:
    // TODO: Constructor
```

```
    // - Construct a hash table with the given capacity, hash function and
    //   offset function.
    // - A dynamic array should be constructed according to the parameter
    //   capacity and pointed by the "table" data member.
    // - In addition, the data members capacity, hashFunction and offsetFunction
    //   should also be initialized with the given parameters. As the hash table is
    //   empty initially, the data member "size" should be set to 0.
    Hash(int capacity, int(*hashFunction)(int,int), int(*offsetFunction)(int,int));

    // TODO: Destructor
    // - De-allocate all the dynamically-allocated memory to avoid any memory leak.
    ~Hash();

    // TODO:
    // - The function searchs for the key in the hash table according to
    //   index = ( hasFunction(key, capacity) + offsetFunction(key, i) ) % capacity,
    //   where i is the number of collisions. You need to figure out how to confirm
    //   the key does not exist in the table.
    // - It also finds the index of an EMPTY or DELETED cell that can be used for
    //   storing the search key, i.e. the parameter "key", and assign the found index
    //   to the reference parameter "next".
    // - It returns the index of the cell where the key is found. If the key is not
    //   found, returns -1.
    int search(const KeyType& key, int& next) const;

    // TODO:
    // - The function first checks if the hash table is half-full. If so,
    //   i.   Output the current capacity and size
    //   ii.  Perform rehashing
    //   iii. Output "Rehashed ..." and show the new capacity of the hash table
    // - Then it inserts the data to the hash table, sets the status of
    //   the cell that carries the data to ACTIVE, and increase the data member
    //   "size" by 1.
    void insert(T* data);

    // TODO:
    // - The function removes the data specified by key from the hash table.
    // - If the key is found, perform lazy deletion, i.e. set status of the cell
    //   with the removed data to DELETED, and decrease the data member "size" by 1.
    void remove(const KeyType& key);

    // TODO:
    // - The function performs rehashing of data.
    // - It dynamically allocates a new array of size
    //   = the nearest prime larger than (2 x the original capacity).
    //   Hint: You may use the given function, nextPrime, for this.
    // - Rehash all the data in the original hash table to the new hash table.
    // - Make the data member "table" points at the new array and update all the
    //   corresponding data members.
    // - Make sure there is no memory leak after performing all the operations above.
    void rehashing();
};
```

```
bool isPrime(int n){
  bool prime = true;
  for(int i = 2; i <= n / 2; ++i) {
    if(n % i == 0) {
      prime = false;
      break;
    }
  }
  return prime;
}

int nextPrime(int n) {
  while(!isPrime(++n));
  return n;
}

#include "Hash.tpp"
```

Note that '`HashCell`' is a private structure defined inside the "`Hash`" class to prevent the access of other classes and global functions.

Your task is to implement all the missing functions of '`Hash`' class template in "`Hash.tpp`". Make sure your implementations will work with the testing program "`test-hash.cpp`" and produce the given output.

```
#include <cmath>     /* File: test-hash.cpp */
#include "Hash.h"

class Pair {
  private:
    int key;
    int value;
  public:
    Pair(int key, int value) : value(value), key(key) { }
    int getKey() const { return key; }
    friend ostream& operator<<(ostream& os, const Pair& p) {
      return os << "Key: " << p.key << ", value: " << p.value << endl;
    }
};

int hash_function(int key, int capacity) {
  return key % capacity;
}

int offset_function(int key, int i) {
  if(i == 0)
    return 0;
  return i * i * i;
}
```

```cpp
int main() {
  const int capacity = 3;
  Hash<Pair, int> hash(capacity, hash_function, offset_function);

  cout << "Inserting Pair(16, 20)" << endl;
  hash.insert(new Pair(16, 20));
  cout << "Inserting Pair(30, 50)" << endl;
  hash.insert(new Pair(30, 50));
  cout << "Inserting Pair(21, 67)" << endl;
  hash.insert(new Pair(21, 67));
  cout << "Inserting Pair(56, 7)" << endl;
  hash.insert(new Pair(56, 7));
  cout << "Inserting Pair(5, 12)" << endl;
  hash.insert(new Pair(5, 12));

  cout << "Output data..." << endl;
  cout << hash << endl;

  hash.remove(21);
  cout << "After removing key 21" << endl << endl;
  cout << "Output data..." << endl;
  cout << hash << endl;

  hash.insert(new Pair(44, 32));
  cout << "After inserting key 44" << endl << endl;
  cout << "Output data..." << endl;
  cout << hash << endl;
}
```

Output of the testing program

```
Inserting Pair(16, 20)
Inserting Pair(30, 50)
Inserting Pair(21, 67)
Capacity: 3, Size: 2
Rehashed ..., new capacity: 7
Inserting Pair(56, 7)
Inserting Pair(5, 12)
Capacity: 7, Size: 4
Rehashed ..., new capacity: 17
Output data...
Index = 4: Key: 21, value: 67
Index = 5: Key: 56, value: 7
Index = 6: Key: 5, value: 12
Index = 13: Key: 30, value: 50
Index = 16: Key: 16, value: 20
```

```
After removing key 21

Output data...
Index = 5: Key: 56, value: 7
Index = 6: Key: 5, value: 12
Index = 13: Key: 30, value: 50
Index = 16: Key: 16, value: 20

After inserting key 44

Output data...
Index = 5: Key: 56, value: 7
Index = 6: Key: 5, value: 12
Index = 10: Key: 44, value: 32
Index = 13: Key: 30, value: 50
Index = 16: Key: 16, value: 20
```

Implement the following 6 missing member functions of the class template '`Hash`' in a separate file called "Hash.tpp".

- `Hash(int capacity, int(*hashFunction)(int,int), int(*offsetFunction)(int,int));`
- `~Hash();`
- `int search(const KeyType& key, int& next) const;`
- `void insert(T* data);`
- `void remove(const KeyType& key);`
- `void rehashing();`

(a) [3 points] Implement

`Hash(int capacity, int(*hashFunction)(int,int), int(*offsetFunction)(int,int));`

below.

**Answer:** `// File: Hash.tpp`

```
template <typename T, typename KeyType>
Hash<T, KeyType>::Hash(int capacity,
                       int(*hashFunction)(int,int), int(*offsetFunction)(int,int))
        : capacity(capacity),                                    // 0.5 point
          hashFunction(hashFunction), offsetFunction(offsetFunction), // 1 point
          table(new HashCell[capacity]), size(0) { }               // 1.5 points
```

(b) [0.5 point] Implement

```
~Hash();
```

below.

**Answer:** // File: Hash.tpp

```
template <typename T, typename KeyType>
Hash<T, KeyType>::~Hash() {
  delete [] table;                                                 // 0.5 point
}
```

(c) [6 points] Implement

```
int search(const KeyType& key, int& next) const;
```

below.

**Answer:** // File: Hash.tpp

```
template <typename T, typename KeyType>
int Hash<T, KeyType>::search(const KeyType& key, int& next) const {
  bool foundNext = false;
  int hashValue = hashFunction(key, capacity);                     // 0.5 point
  int i = 0;
  int count = 0;
  while(true) {
    int index = (hashValue + offsetFunction(key, i)) % capacity;   // 0.5 point
    if(table[index].flag == EMPTY || table[index].flag == DELETED) {  // 1 point
      if(!foundNext) {
        next = index;                                              // 0.5 point
        foundNext = true;
      }
      if(table[index].flag == EMPTY)                               // 0.5 point
        return -1;                                                 // 0.5 point
    }
    else // flag is ACTIVE
      if(table[index].data->getKey() == key)                       // 1 point
        return index;                                              // 1 point
    ++i;                                                           // 0.5 point
  }
}
```

(d) [4 points] Implement

```
void insert(T* data);
```

below.

**Answer:**
```
// File: Hash.tpp
template <typename T, typename KeyType>
void Hash<T, KeyType>::insert(T* data) {
  if(isHalfFull()) {                                                    // 0.5 point
    cout << "Capacity: " << capacity << ", Size: " << size << endl;     // 0.5 point
    rehashing();                                                        // 0.5 point
    cout << "Rehashed ..., new capacity: " << capacity << endl;         // 0.5 point
  }
  int index;
  if(search(data->getKey(), index) == -1) {                            // 0.5 point
    table[index].data = data;                                           // 0.5 point
    table[index].flag = ACTIVE;                                         // 0.5 point
    ++size;                                                             // 0.5 point
  }
}
```

(e) [2 points] Implement

```
void remove(const KeyType& key);
```

below.

**Answer:**
```
// File: Hash.tpp
template <typename T, typename KeyType>
void Hash<T, KeyType>::remove(const KeyType& key) {
  int next;
  int index = search(key, next);                                        // 0.5 point
  if(index != -1) {                                                     // 0.5 point
    table[index].flag = DELETED;                                        // 0.5 point
    --size;                                                             // 0.5 point
  }
}
```

(f) [4.5 points] Implement

```
void rehashing();
```

below.

**Answer:** `// File: Hash.tpp`

```
template <typename T, typename KeyType>
void Hash<T, KeyType>::rehashing() {
  int originalCapacity = capacity;
  HashCell* originalTable = table;                          // 0.5 point
  capacity = nextPrime(originalCapacity * 2);               // 0.5 point
  table = new HashCell[capacity];                           // 0.5 point
  size = 0;                                                 // 0.5 point

  for(int i=0; i<originalCapacity; ++i) {                   // 0.5 point
    if(originalTable[i].flag == ACTIVE) {                   // 0.5 point
      insert(originalTable[i].data);                        // 0.5 point
      originalTable[i].data = nullptr;                      // 0.5 point
    }
  }
  delete [] originalTable;                                  // 0.5 point
}
```

-------------------- END OF PAPER --------------------

# Appendix

## dynamic_cast Conversion

`dynamic_cast<new_type>(expression)`

Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.

## typeid Operator

`typeid(type) / typeid(expression)`

Defined in the standard header **typeinfo**.

Used to determine the type of an object at runtime. It returns an `type_info` object that represents the type of the expression.

## 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.

Some of the member functions of the vector<T> container class where T is the type of data stored in the vector are listed below.

| **Member function** | **Description** |
|---|---|
| vector( ) | Default constructor (another constructor later) |
| iterator begin( )<br>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( )<br>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 clear( ) | Removes all elements from the vector (which are destroyed), leaving the container with a size of 0. |
| size_type size( ) | Returns the number of elements in the vector. |
| 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. |

/* Rough work */

/* Rough work */

/* Rough work */

/* Rough work */