.

COMP 2012H Final Exam - Fall 2020 - HKUST

Date:	December 10, 2020 (Thursday)
Time Allowed:	2 hours, 4:30pm–6:50pm (with breaks)
Instructions:	1. This is a closed-book, closed-notes examination.

2. There are 6 questions in 3 separate parts A, B and C.
3. Type your answers in the space provided on Canvas.
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, auto, nor any library functions not mentioned in the questions.

COMP 2012H Final Exam - Fall 2020 - HKUST: Part A

Time Allowed: 40 minutes

Instructions:

1. There are 4 short questions in this part.
2. This is a closed-book, closed-notes examination.
3. Type your answers in the space provided on Canvas.
4. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.

Problem	Topic	**Score**
1	Hashing	/ 12
2	AVL Tree	/ 8
3	Inheritance and Polymorphism	/ 10
4	Function Object and STL	/ 10
	Total	/ 40

Problem 1 [12 points] Hashing

Given the following integer keys to insert into a hash table of size 10 with hash function hash(k) = k mod 10.

69, 66, 80, 35, 18, 60, 89, 70, 12

Assume collision resolution strategy is double hashing and the second hash function is $hash_2$(k) = k / 10. ("/" here means integer division where the fractional part is discarded.)

a. Show the hash table after all the integer keys have been inserted.
(Put EMPTY to represent an empty cell.)

0	1	2	3	4	5	6	7	8	9
80	EMPTY	60	12	70	35	66	89	18	69

Marking scheme: Each correct insertion gives 1 point. If a student makes a mistake at one insertion but the next insertions are "consistent" with this mistake, then they should only lose one point for the original mistake.

b. Calculate the average number of probes (cells examined) needed for successful search over all the keys in the original key set, after all of the keys have been inserted.
I.e. (number of probes needed for finding 69 + number of probes needed for finding 66 + ... + number of probes needed for finding 12) / 9. **Correct your answer to 2 decimal places.**

The average number of probes needed is 4.67.

Marking scheme: 1 point for giving the correct answer.

Note: The following question is independent from Part a and Part b above.

The following is the hash table (size 10) after inserting SIX integer keys into an initially empty hash table using the hash function hash(k) = k mod 10.

0	1	2	3	4	5	6	7	8	9
		42	23	34	52	46	33		

Assume collision resolution strategy is linear probing. Which of the following **is / are** the possible insertion sequences resulting in the above hash table? Assume that the size of the hash table does not change throughout the insertions. Choose all that apply.

i. 46, 42, 34, 52, 23, 33
ii. 34, 42, 23, 52, 33, 46
iii. 46, 34, 42, 23, 52, 33
iv. 42, 46, 33, 23, 34, 52
v. 42, 23, 34, 52, 46, 33

Make sure to follow the formatting exactly, i.e. comma and space, as grading for this question will be done automatically (e.g. i, iii).

Possible insertion sequences resulting in the above hash table: iii, v

Marking scheme: Each correct answer gives 0.5 point. 1 point in total.

Problem 2 **[8 points]** AVL Tree

Insert the following keys:

13, 5, 20, 9, 8, 10, 11, 4, 12

in order (from left to right), into an initially empty AVL tree and delete 13 in the AVL tree that you got.

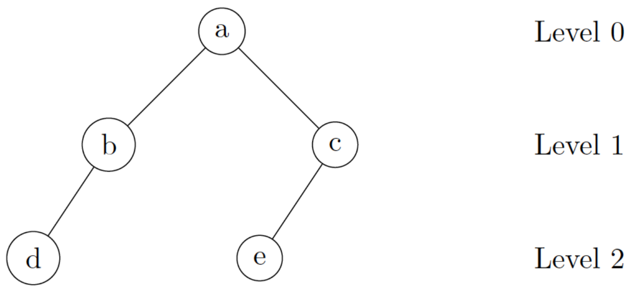
Remarks:

- For the case where the node to be removed has 2 children, replace the node with the **minimum node in its right sub-tree**.
- In case there are two possible ways to restore the resulting tree after deletion to AVL status, you should always pick the one that **requires less number of rotations**.

Write the **level order** traversal of the AVL tree that results from successive key insertions in the table below. You must use the algorithms taught in class for inserting, removing and re-balancing.

Note: Level order traversal prints the nodes of a tree level-by-level. That is, all nodes of level 0 are printed first, followed by nodes of level 1, and so on. All nodes of any level are printed from left to right. For example, the level order traversal of the following tree is:

a, b, c, d, e



To illustrate what you need to do, the insertion of the first three keys has been done for you. **Make sure to follow the formatting exactly, i.e. comma and space, as grading for this question will be done automatically**.

	Level order traversal of the AVL tree that results from successive key insertions / deletion
After the insertion of 13, 5 & 20	13, 5, 20
After further insertion of 9 & 8	13, 8, 20, 5, 9
After further insertion of 10 & 11	9, 8, 13, 5, 10, 20, 11
After further insertion of 4 & 12	9, 5, 13, 4, 8, 11, 20, 10, 12
After deletion of 13 in the AVL tree that you got from inserting all the 9 keys above	9, 5, 11, 4, 8, 10, 20, 12

Marking scheme:

For each line of output

- Give 2 points if the output sequence is exactly the same as the solution.
- Give 1 point if the output sequence is not the same, but it forms an AVL tree (use the given program to check whether the sequence forms an AVL tree).

Problem 3 [10 points] Inheritance and Polymorphism

The following program consists of 5 files: “`Weapon.h`”, “`Bow.h`”, “`Unit.h`”, “`Archer.h`”, “`main.cpp`”. The program runs with no errors after it is compiled with the following command:

```
g++ -std=c++11 -fno-elide-constructors main.cpp
```

Note: -fno-elide-constructors is a g++ compiler flag to **disable** the copy elision optimization (i.e. the compiler **won’t optimize** to remove any constructor calls).

Write down its output in the space provided.

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

class Weapon {
  public:
    virtual ~Weapon() { cout << "~Weapon()" << endl; }

  protected:
    Weapon(int damage) : damage(damage) {
      cout << "Weapon(int)" << endl;
    }
    Weapon(const Weapon& weapon) : damage(weapon.damage) {
      cout << "Weapon(const Weapon&)" << endl;
    }
    int damage;
};


#include <iostream> // File: Bow.h
#include "Weapon.h"
using namespace std;

class Bow : public Weapon {
  public:
    Bow(int damage, int range) : Weapon(damage), range(range) {
      cout << "Bow(int, int)" << endl;
    }
    Bow(const Bow& bow) : Weapon(bow), range(bow.range) {
      cout << "Bow(const Bow&)" << endl;
    }
    virtual ~Bow() { cout << "~Bow()" << endl; }

  protected:
    int range;
};
```

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

class Unit {
  public:
    virtual ~Unit() { cout << "~Unit()" << endl; }

  protected:
    Unit(char id) : id(id) { cout << "Unit(char)" << endl; }
    Unit(const Unit& unit) : id(unit.id) { cout << "Unit(const Unit&)" << endl; }
    char id;
};
```

```
#include <iostream> // File: Archer.h
#include "Unit.h"
#include "Bow.h"
using namespace std;

class Archer : public Unit {
  public:
    Archer(char id, const Bow& bow) : Unit(id), bow(bow) {
      cout << "Archer(char, const Bow&)" << endl;
    }
    Archer(const Archer& archer) : Unit(archer), bow(archer.bow) {
      cout << "Archer(const Archer&)" << endl;
    }
    virtual ~Archer() { cout << "~Archer()" << endl; }

  protected:
    Bow bow;
};
```

```
#include <iostream> // File: main.cpp
#include "Archer.h"
using namespace std;

int main() {
  cout << "*** archer1 construction ***" << endl;
  Archer archer1('A', Bow(5, 3));
  cout << endl;

  cout << "*** archer2 copy-construction ***" << endl;
  Unit* archer2 = new Archer(archer1);
  cout << endl;

  cout << "*** archer2 destruction ***" << endl;
  delete archer2;
  cout << endl;

  cout << "*** before main() returns ***" << endl;
  return 0;
}
```

```
*** archer1 construction ***
Weapon(int)
Bow(int, int)
Unit(char)
Weapon(const Weapon&)
Bow(const Bow&)
Archer(char, const Bow&)
~Bow()
~Weapon()

*** archer2 copy-construction ***
Unit(const Unit&)
Weapon(const Weapon&)
Bow(const Bow&)
Archer(const Archer&)

*** archer2 destruction ***
~Archer()
~Bow()
~Weapon()
~Unit()

*** before main() returns ***
~Archer()
~Bow()
~Weapon()
~Unit()
```

Marking scheme:

- No points for the blank lines and no penalty for missing them.
- No points for the partition lines with ***, but there is a penalty for missing them: -0.25 point for each missing line.
- Simply put, 0.5 point for each meaningful ouptut line, except (1) the lank lines and partition lines.
- The outputs have to be given in the order of the concepts.
- Partial credits are given based on identifying the concepts from top to bottom as much as we can.
- Penalty for extra lines: -0.25 each.

Problem 4 [10 points] Function Object and STL

'Largest Sum of Contiguous Subseqeunce (LSCS)' problem is to find a contiguous subsequence of an array with largest sum. For example:

```
-2 -3  4 -1 -2  1  5 -3
       |------v------|
            sum=7
```

The LSCS of the above array is 7.

The algorithm for finding LSCS of an array A is described below.

```
Initialize:
  max_so_far = 0
  max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + A[i]
  (b) if(max_so_far < max_ending_here)
        max_so_far = max_ending_here
  (c) if(max_ending_here < 0)
        max_ending_here = 0
return max_so_far
```

Now, suppose we want to implement this algorithm using '*for_each*' STL algorithm and function object using the following code.

```cpp
#include <iostream> /* File: test-lscs.cpp */
#include <algorithm>
#include <vector>

using namespace std;

// Part (a):
// Your completed function object class LSCS will be put here.

int main() {
  vector<int> arr { -2, -3, 4, -1, -2, 1, 5, -3 };
  // Part (b):
  // Call for_each with a function object of LSCS defined
  // in part (a) to find the largest sum contiguous
  // subsequence of arr.
  // Your code here
  cout << lscs.get_result() << endl; // Outputs 7
  return 0;
}
```

Reference: The prototype of '*for_each*' is given below.

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

Parameters:

- first, last: The range to apply the function to
- f: function object, to be applied to the result of dereferencing every iterator in the range [first, last). The signature of the function should be equivalent to the following:

  `void fun(const Type& a);`

  The signature does not need to have `const&`. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Return value: f

Reference: STL Sequence Container: vector.

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

Defined in the standard header **vector**

Member functions:

- `vector(initializer_list<value_type> il,`
  `const allocator_type& alloc = allocator_type())`
  A constructor that constructs a container with a copy of each of the elements in il, in the same order.
- `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.

(a) Define the function object class `LSCS` that will work with the given test program "test-lscs.cpp" to determine the largest sum of contiguous subsequence of a given array.

```
class LSCS {
  public:
    // Init of the member variables - 1.5 points
    LSCS(): max_so_far(0), max_ending_here(0) {};
    // (can also be int) function prototype - 1.5 points
    void operator() (const int& val) {
      max_ending_here += val;           // Algo step 1 - 0.5 point
      if (max_so_far < max_ending_here) // Algo step 2 - 0.5 point
        max_so_far = max_ending_here;
      if (max_ending_here < 0)          // Algo step 3 - 0.5 point
        max_ending_here = 0;
    };
    // Member function to get the result - 1 point
    int get_result() { return max_so_far; }
  private:
    // Should be defined as member variables - 1 point
    int max_so_far;
    int max_ending_here;
};
```

(b) Call *for_each* with a function object of LSCS class defined in part (a) to find the largest sum contiguous subsequence of arr.

```
LSCS lscs = for_each(arr.begin(), arr.end(), LSCS());
// Return value - 1.5 points
// First two parameters - 0.5 point for each (1 point in total)
// The third parameter - 1 point

// NOTE: You must use for_each, and you cannot run complete loops in each iteration
```

COMP 2012H Final Exam - Fall 2020 - HKUST: Part B

Time Allowed: 40 minutes

Instructions:
1. There are 1 long question in this part.
2. This is a closed-book, closed-notes examination.
3. Type your answers in the space provided on Canvas.
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, auto, nor any library functions not mentioned in the questions.

Problem	Topic	**Score**
5	Inheritance, Polymorphism and Dynamic Binding	/ 30

Problem 5 [30 points] Inheritance, Polymorphism and Dynamic Binding

This problem involves 4 classes and 2 enumeration types called ‘Lesson’, ‘F2F_Lesson’, ‘Online_Lesson’, ‘Course’, ‘F2F_Interaction’ and ‘Online_Interaction’. Below are the header files of the 4 classes and 2 enumeration types.

```
#ifndef INTERACTIONS_H /* File: Interactions.h */
#define INTERACTIONS_H

enum F2F_Interaction { EYE_CONTACT, ASK_QUESTION, RAISE_HAND };
enum Online_Interaction { YES_NO, POLL, CHAT };

#endif /* INTERACTIONS_H */
```

```
#ifndef LESSON_H /* File: Lesson.h */
#define LESSON_H

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

class Lesson {
  public:
    Lesson(string content) : content(content) {}
    virtual ~Lesson() {}
    virtual void print() const {
      cout << "[ Content: " << content << " ]" << endl;
    }
    virtual void run() const = 0;
  private:
    string content;
};

#endif /* LESSON_H */
```

```
#ifndef F2F_LESSON_H /* File: F2F_Lesson.h */
#define F2F_LESSON_H

#include "Lesson.h"

/***********************************************
   IMPORTANT: Assume all the member functions of
   F2F_Lesson class have been implemented.
 ***********************************************/

class F2F_Lesson : public Lesson {
  public:
    F2F_Lesson(string content);
    bool add_interaction(F2F_Interaction i);
    virtual void run() const;
  private:
    static const int MAX_NUM_INTERACTIONS = 10;
    F2F_Interaction interactions[MAX_NUM_INTERACTIONS];
    int num_interactions;
};

#endif /* FACE_TO_FACE_LESSON_H */
```

```
#ifndef ONLINE_LESSON_H /* File: Online_Lesson.h */
#define ONLINE_LESSON_H

#include "Lesson.h"

class Online_Lesson : public Lesson {
  public:
    // Constructor
    Online_Lesson(string content, int num_video_on);
    bool add_interaction(Online_Interaction i);
    virtual void run() const;

  private:
    static const int MAX_NUM_INTERACTIONS = 10;
    int num_video_on;
    Online_Interaction interactions[MAX_NUM_INTERACTIONS];
    int num_interactions;
};

#endif /* ONLINE_LESSON */
```

```
#ifndef COURSE_H /* File: Course.h */
#define COURSE_H

#include <typeinfo>
#include "F2F_Lesson.h"
#include "Online_Lesson.h"
using namespace std;

class Course {
  public:
    Course(string code);           // Conversion constructor
    Course(const Course& course); // Copy constructor
    ~Course();                     // Destructor
    bool addLesson(Lesson* lesson);
    // Copy assignment operator function
    Course& operator=(const Course& course);
    void run() const;

  private:
    string code;
    Lesson** lessons;
    int num_lessons;
    static const int MAX_LESSONS = 26;
};

#endif /* COURSE_H */
```

Below is the testing program "test_course.cpp".

```
#include "Course.h" /* File: test_course.cpp */
#include "Interactions.h"

int main() {
  Course comp2012h("COMP 2012H");

  F2F_Lesson* l1 = new F2F_Lesson("Pointers");
  l1->add_interaction(EYE_CONTACT);
  l1->add_interaction(ASK_QUESTION);
  comp2012h.addLesson(l1);

  Online_Lesson* l2 = new Online_Lesson("Linked List", 10);
  l2->add_interaction(YES_NO);
  l2->add_interaction(POLL);
  comp2012h.addLesson(l2);
  cout << "===== Run COMP 2012H =====" << endl;
  comp2012h.run();
  cout << "===== Run COMP 2012G =====" << endl;
  Course comp2012g = comp2012h;
  Online_Lesson* l3 = new Online_Lesson("Hashing", 5);
  l3->add_interaction(CHAT);
  l3->add_interaction(YES_NO);
  comp2012g.addLesson(l3);
  comp2012g.run();
  cout << "===== Run COMP 2012G again =====" << endl;
  comp2012g = comp2012h;
  comp2012g.run();
  return 0;
}
```

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

```
===== Run COMP 2012H =====
Course: COMP 2012H
[ Content: Pointers ]
----- Run F2F Lesson -----
Interactions: EYE_CONTACT, ASK_QUESTION
[ Content: Linked List ]
----- Run Online Lesson -----
Interactions: YES_NO, POLL
Number of videos turned on: 10

===== Run COMP 2012G =====
Course: COMP 2012H
[ Content: Pointers ]
----- Run F2F Lesson -----
Interactions: EYE_CONTACT, ASK_QUESTION
[ Content: Linked List ]
----- Run Online Lesson -----
Interactions: YES_NO, POLL
Number of videos turned on: 10

[ Content: Hashing ]
----- Run Online Lesson -----
Interactions: CHAT, YES_NO
Number of videos turned on: 5

===== Run COMP 2012G again =====
Course: COMP 2012H
[ Content: Pointers ]
----- Run F2F Lesson -----
Interactions: EYE_CONTACT, ASK_QUESTION
[ Content: Linked List ]
----- Run Online Lesson -----
Interactions: YES_NO, POLL
Number of videos turned on: 10
```

Based on the given information, complete the missing member functions of 'Online_Lesson' class, and 'Course' class in their respective .cpp files, namely "Online_Lesson.cpp", and "Course.cpp" in Question (a) to Question (h).

Note that your solution should work with the testing program "test_course.cpp" and produce the given outputs.

Reference: 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.

Reference: dynamic_cast Conversion

`dynamic_cast<new_type>(expression)`

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

(a) [1.5 points] Implement the constructor of Online_Lesson based on the given information.

Online_Lesson(string content, int num_video_on)

- Initialize the data members accordingly.

(Assume Online_Lesson.h has been included and your code would be in Online_Lesson.cpp.)

```
Online_Lesson::Online_Lesson(string content, int num_video_on)
  : Lesson(content), num_video_on(num_video_on), num_interactions(0) { }
// 0.5 point for correct initialization of each data member:
// content, num_video_on, num_interactions.
```

(b) [2 points] Implement the following member function of Online_Lesson based on the given information.

bool add_interaction(Online_Interaction i)

- Assume we start storing the interaction from position 0.
- Ensure that the interactions array doesn't overflow.
- Don't overwrite existing interactions.
- Return true if i is added to interactions array successfully, otherwise return false.

(Assume Online_Lesson.h has been included and your code would be in Online_Lesson.cpp.)

```
bool Online_Lesson::add_interaction(Online_Interaction i) {
  if(num_interactions < MAX_NUM_INTERACTIONS) { // 0.5 point
    interactions[num_interactions] = i;         // 0.5 point
    num_interactions += 1;                      // 0.5 point
    return true;                                // 0.25 point
  }
  return false;                                 // 0.25 point
}
```

(c) [4.5 points] Implement the following member function of Online_Lesson based on the given information.

virtual void run() const

- Print interactions and num_video_on according to the **given output**.

(Assume Online_Lesson.h has been included and your code would be in Online_Lesson.cpp.)

```
void Online_Lesson::run() const {
  print();                                                        // 0.5 point
  cout << "----- Run Online Lesson -----" << endl;                // 0.5 point
  cout << "Interactions: ";                                       // 0.5 point
  for (int i = 0; i < num_interactions; i += 1) {                 // 0.5 point
    switch (interactions[i]) {
      case YES_NO:
        cout << "YES_NO";                                         // 0.5 point
        break;
      case POLL:
        cout << "POLL";                                           // 0.5 point
        break;
      case CHAT:
        cout << "CHAT";                                           // 0.5 point
        break;
    }
    cout << ((i < num_intereactions - 1) ? ", " : "");            // 0.5 point
  }
  cout << endl;
  cout << "Number of videos turned on: " << num_video_on << endl; // 0.5 point
  cout << endl;
}
```

(d) [2 points] Implement the conversion constructor of Course based on the given information.

Course(string code)

- Initialize all the data members of Course accordingly.
- lessons is a dynamic array of Lesson pointers of size MAX_LESSONS.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
Course::Course(string code) : code(code) { // 0.5 point for code
  lessons = new Lesson*[MAX_LESSONS];      // 0.5 point for new Lessons*,
                                           // 0.5 point for array with correct size
  num_lessons = 0;                         // 0.5 point
}
```

(e) [2 points] Implement the copy constructor of Course based on the given information.

Course(const Course& course)

- It performs deep copy.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
Course::Course(const Course& course) {
  lessons = new Lesson*[MAX_LESSONS];    // 0.5 point for new Lesson*
                                         // 0.5 point for array with correct size
  num_lessons = 0;                       // 0.5 point
  *this = course;                        // 0.5 point (-0.25 point penalty of
                                         // forget to dereference this pointer)
  // Also acceptable to copy-paste the relevant code section from operator=.
}
```

(f) [2.5 points] Implement the destructor of Course based on the given information.

~Course()

- It deallocates all dynamically allocated memory such that the program has no memory leak.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
Course::~Course() {
  for(int i=0; i<num_lessons; ++i) // 0.5 point
    delete lessons[i];             // 1 point
  delete [] lessons;               // 1 point
                                   // (-0.5 penalty if forget [])
}
```

(g) [7.5 points] Implement the following member function of Course based on the given information.

bool addLesson(Lesson* lesson)

- Deep copy the Lesson type object pointed by the ”lesson” parameter.
- Assume we start storing the address of the cloned object from position 0.
- Don’t overwrite existing lessons.
- Ensure that the lessons array doesn’t overflow.
- Return true if ”lesson” is added successfully, otherwise return false.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
bool Course::addLesson(Lesson* lesson) {
  if(num_lessons < MAX_LESSONS) {                               // 0.5 point
    // 1 point (-0.5 point penalty for typeid the pointer)
    if(typeid(*lesson) == typeid(F2F_Lesson))
      lessons[num_lessons] =
        new F2F_Lesson(*dynamic_cast<F2F_Lesson*>(lesson));
        // 0.5 point each for correct position, new F2F_Lesson,
        // dynamic_cast, and proper copy.
        num_lessons += 1;                                       // 0.25 point
    else if(typeid(*lesson) == typeid(Online_Lesson))
      lessons[num_lessons] =
        new Online_Lesson(*dynamic_cast<Online_Lesson*>(lesson));
        // 0.5 point each for correct position,
        // new Online_Lesson, dynamic_cast, and proper copy.
        num_lessons += 1;                                       // 0.25 point
    return true;                                                // 0.25 point
  }
  return false;                                                 // 0.25 point
}
```

Alternate solution, as deep-copy is not necessary for the given main program. However, full points will only be given if operator= is implemented to deep-copy correctly, with the same marking scheme as above.

```
bool Course::addLesson(Lesson* lesson) {
  if(num_lessons < MAX_LESSONS) {    // 0.5 point
    lessons[num_lessons] = lesson;   // 0.5 point
    num_lessons += 1;                // 0.5 point
    return true;                     // 0.25 point
  }
  return false;                      // 0.25 point
}
// Remaining 5.5 points graded at the deep-copy of operator=.
```

(h) [6 points] Implement the following member function of Course based on the given information.

Course& operator=(const Course& course)

- It performs deep copy.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
Course& Course::operator=(const Course& course) {
  // 1 point for checking this pointer,
  // -0.5 point penalty if not checking course address.
  if(this != &course) {
    for(int i=0; i<num_lessons; ++i)        // 0.5 point
      delete lessons[i];                    // 0.5 point
    code = course.code;                     // 0.5 point
    num_lessons = 0;                        // 0.5 point
    for(int i=0; i<course.num_lessons; ++i) // 0.5 point
      addLesson(course.lessons[i]);         // 1 point
    // If alternate solution of addLesson is used,
    // them manual deep-copy is required and graded here
  }
  // 0.5 point for proper state of this->lessons.
  return *this; // 1 point (-0.5 point penalty if forget to dereference this pointer)
}
```

(i) [2 points] Implement the following member function of Course based on the given information.

void run() const

- It prints the course code and runs all the lessons.

(Assume Course.h has been included and your code would be in Course.cpp.)

```
void Course::run() const {
  cout << "Course: " << code << endl;  // 0.5 point
  for(int i=0; i<num_lessons; ++i)     // 0.5 point
    lessons[i]->run();  // 1 point (-0.5 point penalty for using . operator)
}
```

COMP 2012H Final Exam - Fall 2020 - HKUST: Part C

Time Allowed: 40 minutes

Instructions:

1. There are 1 long question in this part.
2. This is a closed-book, closed-notes examination.
3. Type your answers in the space provided on Canvas.
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, auto, nor any library functions not mentioned in the questions.

Problem	Topic	**Score**
6	Binary Search Tree (BST)	/ 30

Problem 6 [30 points] Binary Search Tree (BST)

This question is about an implementation of a Binary Search Tree (BST) using pointers, which supports finding the successor of a value in the tree. This implementation involves 2 class templates, namely '`BSTNode<T>`' and '`BST<T>`' in 3 files (BSTNode.h, BST.h and BST.tpp).

Given "BSTNode.h" and "BST.h", implement all the missing member functions of BST class template in "BST.tpp" according to the given details.

```
#ifndef BSTNODE_H
#define BSTNODE_H

template <typename T>
class BSTNode {
    template <typename S>
    friend class BST;
  public:
    BSTNode(const T& val) : val(val),
       left(nullptr), right(nullptr), parent(nullptr) {};
  private:
    T val;
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
};

#endif /* BSTNODE_H */

#ifndef BST_H
#define BST_H

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

template <typename T>
class BST {
  public:
    BST(): root(nullptr) {}
    ~BST() { deallocate(root); }
    bool search(const T& query) const {
      return search_helper(root, query) != nullptr;
    }
    void insert(const T& element) {
      if(search(element)) return;
      insert_helper(root, element);
    }
    bool remove(const T& element) {
      BSTNode<T>* find = search_helper(root, element);
      if(find == nullptr) return false;
      remove_helper(find);
      return true;
    }

    void list(std::ostream& os) const { list_helper(root, os); }

    // Assume query exist, return itself if query is the max
    T successor(const T& query) const;
    T min() const { return min_helper(root)->val; } // Ignore empty

  private:
    BSTNode<T>* search_helper(BSTNode<T>* p, const T& query) const;
    void insert_helper(BSTNode<T>* p, const T& element);
    void remove_helper(BSTNode<T>* p);
    void list_helper(BSTNode<T>* p, std::ostream& os) const;
    BSTNode<T>* min_helper(BSTNode<T>* p) const;

    void deallocate(BSTNode<T>* p) {
      if(p == nullptr) return;
      deallocate(p->left);
      deallocate(p->right);
      delete p;
    }

    BSTNode<T>* root;
};

#include "BST.tpp"
#endif /* BST_H */
```

(a) [4 points] Implement the private member function ‘search_helper’ which searches the ‘query’ in the tree ‘p’ and returns the address of the node if exist or ‘nullptr’ if not exist. Please also check the public ‘search’ function to see what corner cases you need to handle. You can assume that all the comparing operators are supported.

```
template <typename T>
BSTNode<T>* BST<T>::search_helper(BSTNode<T>* p, const T& query) const {
  if(p == nullptr || p->val == query)      // nullptr - 1 point
    return p;                              // Equal - 1 point
  else if(query < p->val)                  // Less - 1 point
    return search_helper(p->left, query);
  else if (query > p->val)                 // Greater - 1 point
    return search_helper(p->right, query);
}
```

(b) [2 points] Implement the private member function ‘min_helper’ that returns the address of the minimum element in the tree ‘p’.

```cpp
template <typename T>
BSTNode<T>* BST<T>::min_helper(BSTNode<T>* p) const {
  if(p == nullptr || p->left == nullptr)       // No left child - 1 point
    return p;
  return min_helper(p->left);                  // Has left child - 1 point
}
```

(c) [4 points] Implement the private member function ‘list_helper’ that prints the inorder traversal. Please also check the public ‘list’ function to see how list_helper is used. You can assume that ‘operator<<’ is defined for ‘T’.

```cpp
template <typename T>
void BST<T>::list_helper(BSTNode<T>* p, std::ostream& os) const {
  if(p == nullptr)                            // Empty - 1 point
    return;
  list_helper(p->left, os);                   // Left - 1 point
  os << p->val << std::endl;                  // Middle - 1 point
  list_helper(p->right, os);                  // Right - 1 point
}
```

(d) [5 points] Implement the private member function ‘insert_helper’ that creates a new node with content ‘element’ and inserts it to the tree ‘p’. Please also check the public ‘insert’ function to see how insert_helper is used.

```cpp
template <typename T>
void BST<T>::insert_helper(BSTNode<T>* p, const T& element) {
  if(p == nullptr)                            // Empty - 1 point
    root = new BSTNode<T>(element);
  else {
    BSTNode<T>*& next =
      element < p->val ? p->left : p->right;  // Find which side - 1 point
    if(next != nullptr)                       // Has subtree - 1 point
      insert_helper(next, element);
    else {
      next = new BSTNode<T>(element);         // Create node - 1 point
      next->parent = p;                       // Update parent - 1 point
    }
  }
}
// NOTE: Lab8 solution get 2, must have "<T>"
```

(e) [9 points] Implement the private member function ‘remove_helper’ that removes the node ‘p’ from the tree. Please also check the public ‘remove’ function to see how remove_helper is used. You can assume that the content type ‘T’ supports the assignment operation.

```
template <typename T>
void BST<T>::remove_helper(BSTNode<T>* p) {
  if(p == nullptr)
    return;
  else if (p->left != nullptr && p->right != nullptr) {
    BSTNode<T>* succ = min_helper(p->right);   // Find min in right subtree - 1 point
    p->val = succ->val;                        // Copy value - 1 point
    remove_helper(succ);                       // Recursive call - 1 point
  }
  else {
    BSTNode<T>*& pointer_to_p_in_tree = (
    p->parent != nullptr ?                     // No parent (p is root) case - 1 point
      (p == p->parent->left ?
            // Find the pointer to be modified (find which side) - 1 point
            p->parent->left : p->parent->right) : root
    );
    pointer_to_p_in_tree = p->left != nullptr ? p->left : p->right;
    if(pointer_to_p_in_tree != nullptr)        // Update parent - 1 point
      pointer_to_p_in_tree->parent = p->parent;
    delete p;                                  // Deallocate - 1 point
  }
}
```

(f) [6 points] Implement the public member function ‘successor’ that finds the successor of ‘query’ in the whole tree. The successor of an element ‘query’ is the smallest element that is larger than the ‘query’. Also the successor is the one right after ‘query’ in the inorder traversal. But in this function you cannot get the whole traversal and find the successor in the list. You can assume that the query exists in the tree. Return the query itself if it is the maximum element in the tree.

```
template <typename T>
T BST<T>::successor(const T& query) const {
  BSTNode<T>* query_p = search_helper(root, query); // Find the element - 1 point
  if(query_p->right != nullptr) {           // Distinguish two cases - 1 point
    return min_helper(query_p->right)->val; // Find min of right subtree - 1 point
  } else {
    BSTNode<T>* curr = query_p;
    while(curr->parent != nullptr) {        // Scan parents back to root - 1 point
      if(curr->parent->left == curr)
        return curr->parent->val;
      curr = curr->parent;
    }
    return query;                           // Handle "query is max" case - 1 point
  }
}
```

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