COMP 2012 Final Exam - Fall 2018 - HKUST

Date: December 11, 2018 (Tuesday)
Time Allowed: 3 hours, 8:30am–11:30am
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **7** questions on **32** 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 - 5 **AND** space for **ALL** your answers.
   - Paper II: Description of problem 6 - 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	SOLUTIONS & MARKING SCHEME
Student ID	
Email Address	
Venue	LG1 Table Tennis Room / Tsang Shiu Tim Art Hall * (* Delete as appropriate)
Seat Number	

For T.A. Use Only

Problem	Topic	Score
1	True or False Questions	/ 10
2	Standard Template Library (STL)	/ 6
3	Order of Construction and Destruction	/ 11
4	AVL Tree	/ 7
5	Hashing	/ 12
6	Inheritance, Polymorphism & Dynamic Binding	/ 28
7	Binary Search Tree (BST)	/ 26
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) Subscript operator, `operator[]`, can be overloaded to accept multiple arguments.

T **F** (b) The following is an invalid template declaration, that is it CANNOT be instantiated into a function.

```
template <int x>
int function() {
  return x;
}
```

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

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

template <typename T>
void f(T x, T y) {
  cout << "Template" << endl;
}

void f(int x, int y) {
  cout << "Non-template" << endl;
}

int main() {
  f(1, 2);
  f('a', 'b');
  f(1, 'b');
}
```

T **F** (d) A `const_iterator` is an iterator that cannot be used to modify the element to which it refers to.

T **F** (e) A privately inherited class can be inherited further.

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

```
class Base { };

class Derived : private Base { };

int main() {
  Derived d;
  Base& bRef = d;
}
```

T **F** (g) Dynamic binding is done for the function call at line 16 of the following program.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5    public:
6      virtual void f() { cout << "A::f()" << endl; }
7  };
8
9  class B : public A {
10   public:
11     virtual void f() { cout << "B::f()" << endl; }
12 };
13
14 int main() {
15   A* aPtr = new B;
16   aPtr->A::f();
17 }
```

T **F** (h) A static data member may be initialized while defining it.

T **F** (i) Declaring a friend function in private section of a class rather than public alters its meaning.

T **F** (j) It is sufficient to construct the original Binary Search Tree from the preorder traversal sequence.

Problem 2 [6 points] Standard Template Library (STL)

Define the `print` function which prints the integers contained in a STL vector container. The order of the integers printed should be the same as they are stored in the container, and they should be separated by a single comma. For example, with the following program:

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

// Your complete print function definition will be put here

int main()
{
    vector<int> v;
    v.push_back(3);
    v.push_back(2);
    v.push_back(4);
    v.push_back(5);
    print(v);

    return 0;
}
```

The `print` function should print

```
3,2,4,5
```

on the screen.

Note: You MUST make use of iterators in this question, and you CANNOT use `size()` and `operator[]` of the `vector` class.

Answer:

```
// Implement print function here


void print(vector<int>& v) // pass-by-value is also OK       // 1 point
{
    vector<int>::iterator it = v.begin(); // const_iterator is also OK  // --
    cout << *it;                                                 // 1 point      |-- 3 points
    for(it++; it != v.end(); it++)                                          // --
    {
        cout << "," << *it;                                     // 1 point
    }
}
```

Alternative solution 1

```
void print(vector<int>& v) // pass-by-value is also OK
{
    for(vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it;
        it++;
        if(it != v.end())
             cout << ",";
         it--;
    }
}
```

Alternative solution 2

```
void print(vector<int>& v) // pass-by-value is also OK
{
    for(vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it;
        if(it+1 != v.end())
            cout << ",";
    }
}
```

Problem 3 [11 points] Order of Construction and Destruction

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

class Weapon { };

class Sword : public Weapon {
  public:
    Sword(int n = 1) { cout << n << " x S" << endl; }
    ~Sword() { cout << "~S" << endl; }
};

class Hero {
    Weapon* w;
  public:
    Hero() { cout << "H" << endl; w = new Weapon; }
    ~Hero() { cout << "~H" << endl; delete w; }
    virtual const Hero& operator=(const Hero& h) { cout << "H=" << endl; }
};

class SpiderMan : public Hero {
    Weapon* w;
  public:
    SpiderMan() { cout << "SM" << endl; w = new Weapon; }
    SpiderMan(const SpiderMan& s) { cout << "Copy SM" << endl; w = new Weapon; }
    virtual ~SpiderMan() { cout << "~SM" << endl; delete w; }
    const SpiderMan& operator=(const SpiderMan& s) { cout << "SM=" << endl; }
};

class DeadPool : public SpiderMan {
    Sword sword;
  public:
    DeadPool() : sword(2) { cout << "DP" << endl; }
    DeadPool(const DeadPool& d) : SpiderMan(d) { cout << "Copy DP" << endl; }
    ~DeadPool() { cout << "~DP" << endl; }
    const DeadPool& operator=(const DeadPool& d) { cout << "DP=" << endl; }
};

int main() {
  cout << "--- Block 1 ---" << endl;
  Hero* hero = new DeadPool;
  cout << "--- Block 2 ---" << endl;
  SpiderMan* spiderman = new DeadPool;
  cout << "--- Block 3 ---" << endl;
  DeadPool deadpool(*dynamic_cast<DeadPool*>(spiderman));
  cout << "--- Block 4 ---" << endl;
  *hero = *spiderman;
  cout << "--- Block 5 ---" << endl;
  delete hero;
  cout << "--- Block 6 ---" << endl;
  delete spiderman;
}
```

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:

```
--- Block 1 ---
H
SM
2 x S
DP
--- Block 2 ---
H
SM
2 x S
DP
--- Block 3 ---
H
Copy SM
1 x S
Copy DP
--- Block 4 ---
H=
--- Block 5 ---
~H
--- Block 6 ---
~DP
~S
~SM
~H
~DP
~S
~SM
~H
```

Marking scheme:

- 0.5 point for giving each line of correct output. (There are 22 lines, 11 points in total)

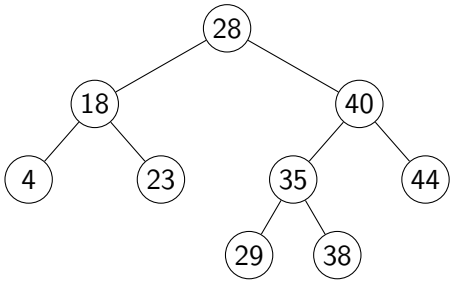
Problem 4 [7 points] AVL Tree

(a) [1 point] What is the maximum height of any AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.

Answer:

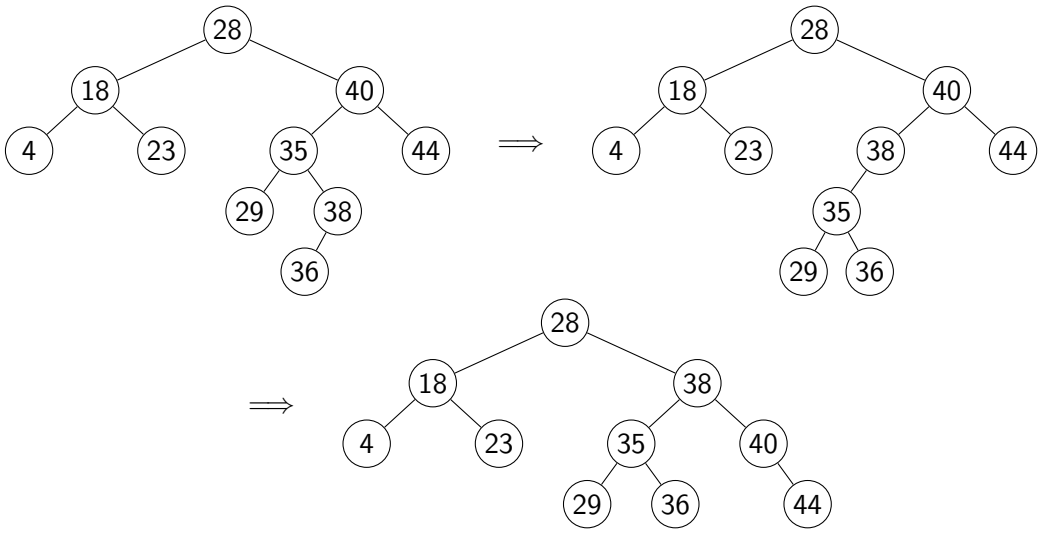
The maximum height of any AVL tree with 7 nodes is 3.

(b) [6 points] Consider the following AVL tree.



Insert the key 36 into the tree and re-balance if needed. Draw all the intermediate trees (including the tree after insertion and each rotation, if any) and final tree. You must use the algorithms discussed in class for inserting and re-balancing.

Answer:



Marking scheme:

- 2 points for giving each correct tree. (3 trees, 6 points in total.)

Problem 5 [12 points] Hashing

Fill in the following hash tables to show their contents when different open addressing methods are used to insert the following 6 strings (in the given order) into the tables of size 7: "best", "stapler", "learn", "bets", "plaster", "renal", and compute the total number of probes for hashing the six strings for each method.

Assume the mapping of English alphabets to numeric codes is as follows:

Alphabet	a	b	c	d	e	f	g	h	i	j	k	l	m
Code	1	2	3	4	5	6	7	8	9	10	11	12	13
Alphabet	n	o	p	q	r	s	t	u	v	w	x	y	z
Code	14	15	16	17	18	19	20	21	22	23	24	25	26

and the following hash function is to be used for this question:

$$hash(s) = (\text{code of } s[0] + \text{code of } s[1] + \ldots + \text{code of } s[L-1]) \bmod 7$$

where s is the input string (key) to be hashed and L is the key length.

For example, the hash value of string key "best" is computed as follows.
'b' = 2, 'e' = 5, 's' = 19, 't' = 20
hash value = $(2 + 5 + 19 + 20) \bmod 7 = 46 \bmod 7 = 4$

Note that, once a key value gets into the hash table, it stays in the table. To illustrate that, the insertion of the first key "best" has been done for you in the tables below.

(a) [4 points] Linear probing: $h_i(s) = (hash(s) + i) \bmod 7$

Table Index	Insert best	Insert stapler	Insert learn	Insert bets	Insert plaster	Insert renal
0		stapler	stapler	stapler	stapler	stapler
1			learn	learn	learn	learn
2					plaster	plaster
3						renal
4	best	best	best	best	best	best
5				bets	bets	bets
6						

The total number of probes for hashing the six strings (including hashing "best"):
$1 + 1 + 1 + 2 + 3 + 3 = 11$

(b) [4 points] Quadratic probing: $h_i(s) = (hash(s) + i^2) \bmod 7$

Table Index	Insert best	Insert stapler	Insert learn	Insert bets	Insert plaster	Insert renal
0		stapler	stapler	stapler	stapler	stapler
1			learn	learn	learn	learn
2					plaster	plaster
3						renal
4	best	best	best	best	best	best
5				bets	bets	bets
6						

The total number of probes for hashing the six strings (including hashing "best"):
$1 + 1 + 1 + 2 + 4 + 4 = 13$

(c) [4 points] Double hashing: Use the $hash'(k)$ function below as the second hash function:

$$hash'(s) = 5 - (\text{code of } s[0] + \text{code of } s[1] + \ldots + \text{code of } s[L-1]) \bmod 5$$

Table Index	Insert best	Insert stapler	Insert learn	Insert bets	Insert plaster	Insert renal
0		stapler	stapler	stapler	stapler	stapler
1			learn	learn	learn	learn
2					plaster	plaster
3						
4	best	best	best	best	best	best
5				bets	bets	bets
6						renal

The total number of probes for hashing the six strings (including hashing "best"):
$1 + 1 + 1 + 3 + 5 + 2 = 13$

Marking scheme:

- 0.25 point for giving each correct cell value. (15 cells per table, 3 tables, 45 cells in total. 11.25 points)
- 0.25 point for giving each correct number of probes. (1 value per table, 3 tables, 3 values in total. 0.75 point)

Problem 6 [28 points] Inheritance, Polymorphism and Dynamic Binding

This problem involves 3 classes called '`PocketMonster`', '`ElectricPocketMonster`' (derived from '`PocketMonster`' using public inheritance) and '`Team`'. Below are the header files of the 3 classes.

```
#ifndef POCKETMONSTER_H     /* File: PocketMonster.h */
#define POCKETMONSTER_H

#include <iostream>
using namespace std;

class PocketMonster
{
  private:
    string name;          // Name of the pocket monster
    int hp;               // Health point
    int attack;           // Attack stat
    int defense;          // Defense stat
    int speed;            // Speed
    int exp;              // Experience value
    string beforeEvolve; // Species before evolution
    string afterEvolve;  // Species after evolution

  public:
    // Construct a pocket monster with the given name, health point, attack stat,
    // defense stat, speed, experience value, and species before & after evolution
    PocketMonster(string name = "", int hp = 0, int attack = 0,
                  int defense = 0, int speed = 0, int exp = 0,
                  string beforeEvolve = "", string afterEvolve = "")
            : name(name), hp(hp), attack(attack), defense(defense), speed(speed),
              exp(exp), beforeEvolve(beforeEvolve), afterEvolve(afterEvolve) {}

    // Print all the data of pocket monster
    virtual void print() const {
      cout << "Name: " << name << endl;
      cout << "HP: " << hp << ", ATT: " << attack << ", DEF: " << defense
           << ", SPD: " << speed << ", EXP: " << exp << endl;
      cout << beforeEvolve << " -> " << name << " -> " << afterEvolve << endl;
    }

    // Accessor functions of all the data members
    string getName() const { return name; }
    double getHP() const { return hp; }
    int getAttack() const { return attack; }
    int getDefense() const { return defense; }
    int getSpeed() const { return speed; }
    int getExp() const { return exp; }
    string regress() const { return beforeEvolve; }
    string evolve() const { return afterEvolve; }
```

```
    /* ----- Function to implement ----- */

    // Battle with the enemy by updating the enemy's hp and increase the
    // experience value of the pocket monster according to the following:
    // [ Update the enemy's hp ]
    // - Compute the amount of damage
    //   = attack value of the pocket monster - defense stat of the enemy
    // - If the computed damage is greater than 0, compute the enemy's new hp
    //   = current hp of the enemy - computed damage
    //   Otherwise, enemy's new hp = current hp of the enemy
    //   If the new hp is greater than 0, update the enemy's hp value
    //   with the new hp, otherwise enemy's hp remains unchanged.
    // [ Increase the experience value of the pocket monster ]
    // - If the amount of damage is greater than 0, increase the experience value
    //   of the pocket monster by the amount = computed damage + 5,
    //   otherwise only increase the experience value by 5
    virtual void battle(PocketMonster& enemy);
};

#endif /* POCKETMONSTER_H */

#ifndef ELECTRICPOCKETMONSTER_H     /* File: ElectricPocketMonster.h */
#define ELECTRICPOCKETMONSTER_H

#include <iostream>
#include <typeinfo>
#include <cmath>
#include "PocketMonster.h"

class ElectricPocketMonster : public PocketMonster {
  private:
    int electricPower;  // Electric power stat of the electric pocket monster

  public:
    // Accessor function of electric power
    int getEP() const { return electricPower; }

    /* ----- Functions to implement ----- */

    // Construct an electric pocket monster with the given name, health point,
    // attack stat, defense stat, speed, experience value, species before &
    // after evolution and electric power stat
    ElectricPocketMonster(string name = "", int hp = 0, int attack = 0,
                          int defense = 0, int speed = 0, int exp = 0,
                          string bEvolve = "", string aEvolve = "",
                          int electricPower = 0);

    // Battle with the enemy by updating the enemy's hp and increase the experience
    // value of the electric pocket monster according to the following:
```

```
    // [ Update the enemy's hp ]
    // - The rules for updating the enemy's hp are SAME AS the battle function in
    //   PocketMonster class EXCEPT on the computation of damage
    // - The amount of damage is calculated according to the rules below:
    //   * If the enemy is of ElectricPocketMonster type, the damage is
    //     = attack value of the electric pocket monster - defense stat of the enemy
    //   * If the enemy is of PocketMonster type, electric pocket monster takes
    //     advantage of its electric power and the damage is
    //     = attack value of the electric pocket monster -
    //       defense stat of the enemy + floor(0.5 * electricPower)
    //     Hint: You may find the function: double floor(double x) in cmath library
    //           useful for this computation.
    // [ Increase the experience value of the ElectricPocketMonster ]
    // - SAME AS the battle function in PocketMonster
    void battle(PocketMonster& enemy);

    // Print all the data of electric pocket monster according to the sample output
    void print() const;
};

#endif /* ELECTRICPOCKETMOSTER_H */

#ifndef TEAM_H     /* File: Team.h */
#define TEAM_H

#include "ElectricPocketMonster.h"

class Team {
  private:
    // A pointer which points to an array of pointers in PocketMonster type
    PocketMonster** members;
    // The size of the pointer array
    int numMembers;

  public:
    // Default constructor
    // Construct a team by initializing members to NULL and numMembers to 0
    Team() : members(NULL), numMembers(0) {}

    /* ----- Functions to implement ----- */

    // Copy constructor which performs deep copy
    // Note:
    // - Two different types of objects will be pointed by the array of pointers
    // - Dynamically create PocketMonster object when the object to be copied
    //   is in PocketMonster type
    // - Dynamically create ElectricPocketMonster object when the object to be
    //   copied is in ElectricPocketMonster type
```

```
    // Hint:
    // - Use typeid(<expression>) to identify the type of the object
    Team(const Team& t);

    // Destructor which deallocates all the dynamically-allocated memory
    ~Team();

    // Add the address of pocket monster / electric pocket monster to the pointer array
    // As the size of the original pointer array is fixed once it is created, you need
    // to do the following in order to store the pointer to a new pocket monster /
    // electric pocket monster
    // - Allocate a new array of size = original size of the array + 1
    // - Copy all the pointers in the original array to the new array
    // - Insert the pointer to the new pocket monster / electric pocket monster to
    //   the end of the new array
    // - Make "members" point at the new array
    // - Make sure there is NO memory leak problem after performing all the
    //   above operations
    void addMember(PocketMonster* pm);

    // Print team information on screen according to the sample output
    void print() const;
};

#endif /* TEAM_H */
```

Below is the testing program "test-pocket-monster.cpp"

```
#include "ElectricPocketMonster.h"     /* File: test-pocket-monster.cpp */
#include "Team.h"

void train_and_print(PocketMonster* pm, PocketMonster& enemy) {
  cout << "=== Start ===" << endl;
  cout << "[ Pocket Monster ]" << endl;
  pm->print();
  cout << "[ Enemy ]" << endl;
  enemy.print();
  cout << "=== Battle ===" << endl;
  if(pm->getSpeed() > enemy.getSpeed()) {
    pm->battle(enemy);
    enemy.battle(*pm);
  }
  else {
    enemy.battle(*pm);
    pm->battle(enemy);
  }
  cout << "[ Pocket Monster ]" << endl;
  pm->print();
  cout << "[ Enemy ]" << endl;
  enemy.print();
}
```

```
int main() {
  cout << ">>> Create a Pocket Monster \"Charmeleon\" " <<
          "and an Electric Pocket Monster \"Pikachu\" <<<" << endl;

  // The following dynamic objects will be de-allocated by the destructor
  // of the Team class.
  PocketMonster* pocketMonster[2] = {
    new PocketMonster("Charmeleon", 80, 25, 30, 8, 50, "Charmander", "Charizard"),
    new ElectricPocketMonster("Pikachu", 100, 30, 25, 10, 60, "Pichu", "Raichu", 40)
  };

  cout << ">>> Create Team A by adding Charmeleon and Pikachu as members <<<" << endl;
  Team teamA;
  for(int i=0; i<sizeof(pocketMonster)/sizeof(PocketMonster*); ++i)
    teamA.addMember(pocketMonster[i]);

  cout << ">>> Create Team B as a clone of Team A using copy constructor <<<" << endl;
  Team teamB(teamA);
  cout << endl;

  PocketMonster enemy("Metapod", 40, 5, 10, 2, 50, "Caterpie", "Butterfree");
  for(int i=0; i<sizeof(pocketMonster)/sizeof(PocketMonster*); ++i) {
    cout << "Train <<< "
         << ( (i==0) ? "PocketMonster" : "ElectricPocketMonster" )
         << " >>>" << endl;
    train_and_print(pocketMonster[i], enemy);
    cout << endl;
  }

  cout << "[ Print Team A ]" << endl;
  teamA.print();
  cout << endl;

  cout << "[ Print Team B ]" << endl;
  teamB.print();

  return 0;
}
```

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

```
>>> Create a Pocket Monster "Charmeleon" and an Electric Pocket Monster "Pikachu" <<<
>>> Create Team A by adding Charmeleon and Pikachu as members <<<
>>> Create Team B as a clone of Team A using copy constructor <<<

Train <<< PocketMonster >>>
=== Start ===
[ Pocket Monster ]
Name: Charmeleon
HP: 80, ATT: 25, DEF: 30, SPD: 8, EXP: 50
Charmander -> Charmeleon -> Charizard
```

```
[ Enemy ]
Name: Metapod
HP: 40, ATT: 5, DEF: 10, SPD: 2, EXP: 50
Caterpie -> Metapod -> Butterfree
=== Battle ===
[ Pocket Monster ]
Name: Charmeleon
HP: 80, ATT: 25, DEF: 30, SPD: 8, EXP: 70
Charmander -> Charmeleon -> Charizard
[ Enemy ]
Name: Metapod
HP: 25, ATT: 5, DEF: 10, SPD: 2, EXP: 55
Caterpie -> Metapod -> Butterfree

Train <<< ElectricPocketMonster >>>
=== Start ===
[ Pocket Monster ]
Name: Pikachu
HP: 100, ATT: 30, DEF: 25, SPD: 10, EXP: 60
Pichu -> Pikachu -> Raichu
Electric Power: 40
[ Enemy ]
Name: Metapod
HP: 25, ATT: 5, DEF: 10, SPD: 2, EXP: 55
Caterpie -> Metapod -> Butterfree
=== Battle ===
[ Pocket Monster ]
Name: Pikachu
HP: 100, ATT: 30, DEF: 25, SPD: 10, EXP: 105
Pichu -> Pikachu -> Raichu
Electric Power: 40
[ Enemy ]
Name: Metapod
HP: 0, ATT: 5, DEF: 10, SPD: 2, EXP: 60
Caterpie -> Metapod -> Butterfree

[ Print Team A ]
Name: Charmeleon
HP: 80, ATT: 25, DEF: 30, SPD: 8, EXP: 70
Charmander -> Charmeleon -> Charizard
Name: Pikachu
HP: 100, ATT: 30, DEF: 25, SPD: 10, EXP: 105
Pichu -> Pikachu -> Raichu
Electric Power: 40

[ Print Team B ]
Name: Charmeleon
HP: 80, ATT: 25, DEF: 30, SPD: 8, EXP: 50
Charmander -> Charmeleon -> Charizard
Name: Pikachu
HP: 100, ATT: 30, DEF: 25, SPD: 10, EXP: 60
Pichu -> Pikachu -> Raichu
Electric Power: 40
```

Based on the given information, complete the implementation of '`PocketMonster`' class, '`ElectricPocketMonster`' class and '`Team`' class in their respective .cpp files, namely, "`PocketMonster.cpp`", "`ElectricPocketMonster.cpp`" and "`Team.cpp`" respectively.

(a) [6 points] Implement the member function

```
virtual void battle(PocketMonster& enemy);
```

of the class '`PocketMonster`' in a separate file called "`PocketMonster.cpp`". Include all the required header file(s) to make sure the program compiles.

Answer: `/* File: PocketMonster.cpp */`

```
#include "PocketMonster.h"     /* File: PocketMonster.cpp */

void PocketMonster::battle(PocketMonster& enemy) {
  int newEnemyHP = enemy.getHP();
  int damage = attack - enemy.getDefense();                    // 0.5 point
  if( damage > 0 )                                             // 0.5 point
    newEnemyHP = enemy.getHP() - damage;                       // 0.5 point
  newEnemyHP = (newEnemyHP > 0) ? newEnemyHP : 0;              // 1 point
  enemy = PocketMonster(enemy.name, newEnemyHP, enemy.attack,
                        enemy.defense, enemy.speed, enemy.exp,
                        enemy.beforeEvolve, enemy.afterEvolve); // 2 points
  exp += ((damage > 0) ? damage : 0) + 5;                      // 1.5 points
}
```

(b) [11 points] Implement the following 3 member functions

- `ElectricPocketMonster(string name, int hp, int attack, int defense, int speed, int exp, string bEvolve, string aEvolve, int electricPower);`
- `void battle(PocketMonster& enemy);`
- `void print() const;`

of the class '`ElectricPocketMonster`' in a separate file called "`ElectricPocketMonster.cpp`". Include all the required header file(s) to make sure the program compiles.

Answer: `/* File: ElectricPocketMonster.cpp */`

```
#include "ElectricPocketMonster.h"     /* File: ElectricPocketMonster.cpp */

ElectricPocketMonster::ElectricPocketMonster(string name, int hp, int attack,
                                             int defense, int speed, int exp,
                                             string bEvolve, string aEvolve,
                                             int electricPower)
    : PocketMonster(name, hp, attack, defense, speed, exp, bEvolve, aEvolve),
      electricPower(electricPower) {}                              // 2 points

void ElectricPocketMonster::battle(PocketMonster& enemy) {
  int newEnemyHP = enemy.getHP();
  int damage = getAttack() - enemy.getDefense();                   // 0.5 point
  if(typeid(enemy) == typeid(PocketMonster))                       // 0.5 point
    damage += floor(0.5 * electricPower);                          // 0.5 point
  if( damage > 0 )                                                 // 0.5 point
    newEnemyHP = enemy.getHP() - damage;                           // 0.5 point
  newEnemyHP = (newEnemyHP > 0) ? newEnemyHP : 0;                  // 1 point
  if(typeid(enemy) == typeid(PocketMonster))                       // 1.5 points
    enemy = PocketMonster(enemy.getName(), newEnemyHP,
                          enemy.getAttack(), enemy.getDefense(),
                          enemy.getSpeed(), enemy.getExp(),
                          enemy.regress(), enemy.evolve());
  else {
    ElectricPocketMonster& e = dynamic_cast<ElectricPocketMonster&>(enemy);
    enemy = ElectricPocketMonster(e.getName(), newEnemyHP,
                                  e.getAttack(), e.getDefense(),
                                  e.getSpeed(), e.getExp(),
                                  e.regress(), e.evolve(),
                                  e.electricPower);
  }
  int newExp = getExp() + ((damage > 0) ? damage : 0) + 5;         // 1.5 points
  *this = ElectricPocketMonster(getName(), getHP(), getAttack(),
                                getDefense(), getSpeed(), newExp,
                                regress(), evolve(), electricPower);   // 1.5 points
}

void ElectricPocketMonster::print() const {
  PocketMonster::print();                                          // 0.5 point
  cout << "Electric Power: " << getEP() << endl;                   // 0.5 point
}
```

(c) [11 points] Implement the following 4 member functions

- `Team(const Team& t);`
- `~Team();`
- `void addMember(PocketMonster* pm);`
- `void print() const;`

of the class 'Team' in a separate file called "Team.cpp". Include all the required header file(s) to make sure the program compiles.

Answer: `/* File: Team.cpp */`

```
#include "Team.h"     /* File: Team.cpp */

Team::Team(const Team& t) {
  members = new PocketMonster*[t.numMembers];                    // 0.5 point
  numMembers = t.numMembers;                                     // 0.5 point
  for(int i=0; i<numMembers; ++i) {                              // 0.5 point
    if(typeid(*t.members[i]) == typeid(PocketMonster))           // 1 point
      members[i] = new PocketMonster(*(t.members[i]));           // 1 point
    else
      members[i] = new ElectricPocketMonster(
                    *dynamic_cast<ElectricPocketMonster*>(t.members[i])
                   );                                            // 1 point
  }
}

Team::~Team() {
  for(int i=0; i<numMembers; ++i)                                // 0.5 point
    delete members[i];                                           // 0.5 point
  delete [] members;                                             // 0.5 point
}

void Team::addMember(PocketMonster* pm) {
  PocketMonster** temp = new PocketMonster*[numMembers + 1];     // 1 point
  for(int i=0; i<numMembers; ++i)                                // 0.5 point
    temp[i] = members[i];                                        // 0.5 point
  temp[numMembers] = pm;                                         // 0.5 point
  delete [] members;                                             // 0.5 point
  members = temp;                                                // 0.5 point
  numMembers++;                                                  // 0.5 point
}

void Team::print() const {
  for(int i=0; i<numMembers; ++i)                                // 0.5 point
    members[i]->print();                                         // 0.5 point
}
```

Problem 7 [26 points] Binary Search Tree (BST)

Given "`BtreeNode.h`" and "`BinarySearchTree.h`" which are similar to what you have worked on in Lab 9, complete all the missing functions in "`Solution.tpp`". Be aware that some of the functions have different requirements than those in the lab.

```
#ifndef BTREE_NODE_H     /* File: BtreeNode.h */
#define BTREE_NODE_H

template <typename T>
class BtreeNode {
  public:
    BtreeNode(const T& x, BtreeNode* L = 0, BtreeNode* R = 0) :
              data(x), left(L), right(R) {}

    ~BtreeNode() {
      delete left;
      delete right;
    }

    const T& get_data() const { return data; }

    BtreeNode* get_left() const { return left; }

    BtreeNode* get_right() const { return right; }

  private:
    T data;
    BtreeNode* left;
    BtreeNode* right;
};

#endif // BTREE_NODE_H
```

```
#ifndef BINARYSEARCHTREE_H     /* File: BinarySearchTree.h */
#define BINARYSEARCHTREE_H

#include <iostream>
#include <iomanip>     // For setw function
using namespace std;

template <typename T>
class BinarySearchTree {
  private:
    class BinaryNode;

  public:
    // Constructor
    BinarySearchTree() : root(NULL) {}

    // Deep-copy constructor
    BinarySearchTree(const BinarySearchTree &src) : root(src.clone(src.root)) {}

    // Destructor
    ~BinarySearchTree() { makeEmpty(); }

    // Return true if the tree is empty
    // Return false otherwise
    bool isEmpty() const { return !root; }

    // TO DO: Print "The maximum key is X" where X is the maximum key in the tree,
    // if the tree is not empty or print "Empty tree!" otherwise
    void printMax() const;

    // Print the tree
    // the output is rotated -90 degrees just like the BST lab
    void printTree() const { printTree(root, 0); }

    // Make the tree empty
    // Deallocate all the allocated dynamic memory for the tree
    // used by the destructor
    void makeEmpty() { makeEmpty(root); }

    // Insert a key to the BST
    // do nothing and return false if the key already exists in the tree
    // insert the key and return true otherwise
    bool insert(const T& x) { return insert(x, root); }

    // Remove a key from the BST
    // do nothing and return false if the key does not exist in the tree
    // do nothing and return false if the key is not at a leaf node
    // remove the key and return true otherwise
    bool remove(const T& x) { return remove(x, root); }
```

```
    // Return true if the tree is full, return false otherwise
    // a full binary tree is a tree in which every node other than the leaves
    // has two children; this function also simply returns true if the tree is empty
    bool isFull() const { return isFull(root); }

  private:
    struct BinaryNode {
      T x;
      BinaryNode* left;
      BinaryNode* right;

      BinaryNode() : left(NULL), right(NULL) {}

      BinaryNode(const T& x, BinaryNode* lt = NULL, BinaryNode* rt = NULL)
                 : x(x), left(lt), right(rt) {}

    };

    BinaryNode* root; // The root node; will be NULL if the tree is empty

    // See above
    void printTree(BinaryNode* t, int depth) const {
      if (t == NULL) return;
      const int offset = 6;
      printTree(t->right, depth + 1);
      cout << setw(depth * offset) << t->x << endl;
      printTree(t->left, depth + 1);
    }

    // TO DO: See above
    bool insert(const T& x, BinaryNode*& t);

    // TO DO: See above
    bool remove(const T& x, BinaryNode*& t);

    // TO DO: See above
    void makeEmpty(BinaryNode* t);

    // TO DO: Return a deep copy of the BST with root t
    // the return value is a pointer that points to the root of the copied tree
    // used by the copy constructor
    BinaryNode* clone(BinaryNode* t) const;

    // TO DO: See above
    bool isFull(BinaryNode* t) const;
};

#include "Solution.tpp"

#endif // BINARYSEARCHTREE_H
```

To test the program, the following "main.cpp" has been used.

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

int main() {
  cout << "Case 1:" << endl;
  BinarySearchTree<int>* bst = new BinarySearchTree<int>();
  cout << "Full? " << boolalpha << bst->isFull() << endl;
  bst->printMax();
  cout << endl;
  cout << "insert 3? " << bst->insert(3) << endl;
  cout << "insert 2? " << bst->insert(2) << endl;
  cout << "insert 5? " << bst->insert(5) << endl;
  cout << "insert 6? " << bst->insert(6) << endl;
  cout << "insert 4? " << bst->insert(4) << endl;
  cout << "insert 3? " << bst->insert(3) << endl;
  cout << "insert 4? " << bst->insert(4) << endl;
  cout << endl;
  bst->printTree();
  cout << endl;
  cout << "Full? " << boolalpha << bst->isFull() << endl;
  bst->printMax();
  cout << endl;

  cout << "Case 2:" << endl;
  cout << "insert 7? " << bst->insert(7) << endl;
  cout << endl;
  bst->printTree();
  cout<<endl;
  cout << "Full? " << boolalpha << bst->isFull() << endl;
  bst->printMax();
  cout << endl;

  cout << "Case 3:" << endl;
  BinarySearchTree<int>* bst2 = new BinarySearchTree<int>(*bst);
  cout << "remove 6?" << bst->remove(6) << endl;
  cout << "remove 7?" << bst->remove(7) << endl;
  cout << "remove 8?" << bst->remove(8) << endl;
  cout << endl;
  cout << "bst:" << endl;
  bst->printTree();
  cout << endl;
  cout << "bst2:" << endl;
  bst2->printTree();

  delete bst;
  delete bst2;
}
```

The following output is expected.

```
Case 1:
Full? true
Empty tree!

insert 3? true
insert 2? true
insert 5? true
insert 6? true
insert 4? true
insert 3? false
insert 4? false

           6
     5
           4
3
     2

Full? true
The maximum key is 6

Case 2:
insert 7? true

                 7
           6
     5
           4
3
     2

Full? false
The maximum key is 7

Case 3:
remove 6?false
remove 7?true
remove 8?false

bst:
           6
     5
           4
3
     2

bst2:
                 7
           6
     5
           4
3
     2
```

Based on the given information, complete "`Solution.tpp`" in the space below. Write your functions in the corresponding places.

(a) [5 points] Implement

```
bool insert(const T& x, BinaryNode*& t);
```

below.

Answer:

```
template <typename T>
bool BinarySearchTree<T>::insert(const T& x, BinaryNode*& t) {
  if (t == NULL)
  {
    t = new BinaryNode(x);
    return true;
  }
  else if (x < t->x)
    return insert(x, t->left);
  else if (x > t->x)
    return insert(x, t->right);
  else
    return false;
}
```

(b) [6 points] Implement

```
bool remove(const T &x, BinaryNode*& t);
```

below.

Answer:

```
template <typename T>
bool BinarySearchTree<T>::remove(const T &x, BinaryNode*& t) {
  if (t == NULL)
    return false;
  else if (x < t->x)
    return remove(x, t->left);
  else if (x > t->x)
    return remove(x, t->right);
  else
  {
    if(t->left||t->right)
      return false;

    delete t;
    t = NULL;
    return true;
  }
}
```

(c) [4 points] Implement

```
void printMax() const;
```

below.

Answer:

```
template <typename T>
void BinarySearchTree<T>::printMax() const {
  const BinaryNode *node = root;

  if (isEmpty()) {
    cout << "Empty tree!" << endl;
    return;
  }

  while (node->right != NULL)
    node = node->right;

  cout << "The maximum key is " << node->x << endl;
}
```

(d) [4 points] Implement

```
bool isFull(BinaryNode* t) const;
```

below.

Answer:

```
template <typename T>
bool BinarySearchTree<T>::isFull(BinaryNode* t) const {
  if(t == NULL)
    return true;

  if(!t->left && !t->right)
    return true;

  if(t->left && t->right)
    return isFull(t->left) && isFull(t->right);

  return false;
}
```

(e) [3 points] Implement

```
BinaryNode* clone(BinaryNode* t) const;
```

below.

Answer:

```
template <typename T>
typename BinarySearchTree<T>::BinaryNode*
  BinarySearchTree<T>::clone(BinaryNode* t) const {
    return !t ? NULL :
           new BinaryNode(t->x, clone(t->left), clone(t->right));
}
```

(f) [4 points] Implement

```
void makeEmpty(BinaryNode* t);
```

below.

Answer:

```
template <typename T>
void BinarySearchTree<T>::makeEmpty(BinaryNode* t) {
  if (t == NULL)
    return;
  makeEmpty(t->left);
  makeEmpty(t->right);
  delete t;
  t = NULL;
}
```

Marking scheme:

- -1 point for each logical error / mistake
- -0.5 point once if any function header is incorrect
- -0.5 point once for each syntax error

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

Appendix

Math Function

`double floor(double x);`
Defined in the standard header **cmath**.
Rounds x downward, returning the largest integral value that is not greater than x.

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( ) 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.

/* Rough work */

/* Rough work */

/* Rough work */

/* Rough work */