Problem 6 [28 points] Inheritance, Polymorphism & 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
    // - If the new hp is greater than 0, update the enemy's hp value
    //   with the new hp, otherwise set enemy's hp to 0.
    // [ 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
```

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
```

```cpp
#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.

```cpp
#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
```

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 */