COMP 2012 Final Exam - Spring 2016 - HKUST

Date: May 25, 2016 (Wednesday)
Time Allowed: 3 hours, 4:30–7:30pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **7** questions on **24** pages (including this cover page).
3. Write your answers in the space provided in black/blue ink. *NO pencil please, otherwise you are not allowed to appeal for any grading disagreements.*
4. All programming codes in your answers must be written in the ANSI C++ version as taught in the class.
5. For programming questions, you are **NOT** allowed to define additional helper functions or structures, nor global variables unless otherwise stated. You also **cannot** use any library functions not mentioned in the questions.

Student Name	Marking Scheme
Student ID	
Email Address	
Lecture & Lab Section	

For T.A.

Use Only

Problem	Topic	Score
1	Operator Prototype	/ 6
2	Namespace	/ 9
3	Static Data/Method	/ 9
4	Function Object	/ 14
5	Template	/ 20
6	Hashing	/ 18
7	BST	/ 24
	Total	/ 100

Problem 1 [6 points] Overloaded Operator Prototypes

Fill in the **SIX** (underlined) missing parts of the following `Vector` class definition with the parameter or returned types so that the class will work with the following main program

```
#include <iostream>
using namespace std;
#include "vector-operator.h"

int main( )
{
    Vector a(3.5, 6.1), c(2.1, 3.6), d;
    const Vector b(5.1, -3.2);

    cout << "a = " << a++ << endl;
    cout << "b = " << '(' << b[0] << ", " << b[1] << ')' << endl;

    ++c = b;
    d[0] = 2.1; d[1] = 3.6;
    (a = b) = d;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;

    return 0;
}
```

to produce the following output. You may assume that all the operator functions are implemented correctly.

```
a = (3.5, 6.1)
b = (5.1, -3.2)
a = (2.1, 3.6)
b = (5.1, -3.2)
c = (5.1, -3.2)
d = (2.1, 3.6)
```

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

class Vector
{
    friend ostream& operator<<(ostream& os, const Vector& a);

  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }

    double operator[ ](int) const;

    double& operator[ ](int);

    Vector& operator++( );

    Vector operator++(int);

    Vector& operator=(const Vector&);

  private:
    double x, y;
};
```

Grading scheme: 1 point for each answer

Problem 2 [9 points] Namespace

What is the output when the following program is run?

```
#include <iostream>

namespace SciencePark { class Person
  {
    private:
      int id;
    public:
      Person(int i): id(i) { print( ); }
      virtual int get( ) const { return id; }
      virtual void print( ) const { std::cout << "In SciencePark" << std::endl; }
      virtual bool compare(const Person& p) { return get( ) < p.get( ); }
  };
}

namespace Cyberport { class Person
  {
    private:
      int id;
    public:
      Person(int i): id(i) { print( ); }
      Person(SciencePark::Person& p) { id = p.get( ); }
      virtual int get( ) const { return id; }
      virtual void print( ) const { std::cout << "In Cyberport" << std::endl; }
      virtual bool compare(const Person& p) { return get( ) < p.get( ); }
  };
}

using Cyberport::Person;
class Student : public Person
{
  public:
    Student(int i) : Person(i) { }
    int get( ) const { return Person::get( ); }
    void print( ) const { std::cout << "In Student" << std::endl; }
    bool compare(const Person& p) { print( ); p.print( ); return get( ) < p.get( ); }
};

using namespace std;
```

```
int main( )
{
    SciencePark::Person person1(18);
    person1.print( );
    cout << "person1's ID: " << person1.get( ) << endl;

    Person* person2 = new Student(20);
    person2->print( );
    cout << "person2's ID: " << person2->get( ) << endl;

    bool result = person2->compare(person1);
    cout << "person2's ID < person1's ID: " << boolalpha << result << endl;
    delete person2;
    return 0;
}
```

Answer:

```
In SciencePark
In SciencePark
person1's ID: 18
In Cyberport
In Student
person2's ID: 20
In Student
In Cyberport
person2's ID < person1's ID: false
```

Grading scheme: 1 point for each line of output

Problem 3 [9 points] Static Data Members and Member Functions

The following program contains **5 ERRORS** (syntax errors, omissions, etc.). Study the program carefully, identify all the errors by writing down the line number where an error occurs, and provide a corrective action with an optional explanation wherever you find necessary in the table below so that the program produces the expected output. A corrective action may consist of removing some codes, adding line(s) of codes, or correcting some codes.

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 class WiFi
 6 {
 7   private:
 8     static int numWiFi;
 9     string ssid;
10     int numUser;
11
12   public:
13     WiFi(string ssid = "MyWifi")
14     {
15         this->ssid = ssid;
16         numUser = 0;
17         numWiFi++;
18         cout << "WiFi's constructor" << endl;
19         printNumWiFi( );
20     }
21
22     ~WiFi( )
23     {
24         numWiFi--;
25         cout << "WiFi's destructor" << endl;
26     }
27
28     void connect( )
29     {
30         numUser++;
31         cout << "Connected to SSID [" << ssid << "]; #users = " << numUser << endl;
32     }
33
34     static void printNumWiFi( ) const
35     {
36         cout << "#Allocated WiFi's = " << numWiFi << endl;
37         cout << "SSID: " << ssid << endl;
38     }
39 };
```

```
class Pantry
{
  private:
    static const int maxAccess = 2;
    int numAccess;

  public:
    Pantry( ): numAccess(0) { cout << "Pantry's constructor" << endl; }
    ~Pantry() { cout << "Pantry's destructor" << endl; }

    bool access( )
    {
        if (numAccess < maxAccess) { numAccess++; return true;  }
        else return false;
    }
};

class Employee
{
  private:
    static WiFi wifi;
    static Pantry pantry;

  public:
    Employee( ) { cout << "Employee's constructor" << endl; }
    ~Employee( ) { cout << "Employee's destructor" << endl; }

    void accessWiFiAndPantry( ) const
    {
        (Employee.wifi)::connect( );
        cout << ( (pantry.access( )) ?
                  "Access pantry!" : "No more access to pantry!");
        cout << endl;
    }
};

int WiFi::numWiFi = 0;

int main( )
{
    cout << "------ BEGIN MAIN ------" << endl;
    Employee* employee = new Employee[3];

    for(int i = 0; i < 3; i++)
        employee[i].accessWiFiAndPantry( );

    delete employee;
    cout << "------ END MAIN ------" << endl;
    return 0;
}
```

Expected output of the program:

```
Pantry's constructor
WiFi's constructor
#Allocated WiFi's = 1
------ BEGIN MAIN ------
Employee's constructor
Employee's constructor
Employee's constructor
Connected to SSID [MyWifi]; #users = 1
Access pantry!
Connected to SSID [MyWifi]; #users = 2
Access pantry!
Connected to SSID [MyWifi]; #users = 3
No more access to pantry!
Employee's destructor
Employee's destructor
Employee's destructor
------ END MAIN ------
WiFi's destructor
Pantry's destructor
```

Answers:

Error#	Line#	Corrected Code or Action	Explanation
1	34	static void printNumWiFi( )	no const static member function
2	37	remove: cout $\ll$ “SSID:” $\ll$ ssid $\ll$ endl	can't use non-static member
3	69	wifi.connect( )	wrong syntax
4	77	Add: Pantry Employee::pantry; WiFi Employee::wifi;	initialize static members globally: pantry and wifi
5	86	delete [ ] ptr;	wrong syntax for deleting an array

Grading scheme: 1.5 points for each corrective action (and explanation). Explanation is not required if the corrective action is clearly given; otherwise, there may be penalty.

Problem 4 [14 points] Inheritance, Function Objects, and Hashing

Double hashing in open addressing collision resolution involves two hash functions: a primary hash function $hash_1(\cdot)$ and a secondary hash function $hash_2(\cdot)$, and they have the following forms (where k is a key):

$$
\begin{aligned}
\text{hash}_1(k) &= k \mod m \\
\text{hash}_2(k) &= R - (k \mod R) \\
h_i(k) &= [\text{hash}_1(k) + i \times \text{hash}_2(k)] \mod m
\end{aligned}
$$

All hash functions are implemented as function objects as follows. An abstract base class `Hash` is first defined, from which 2 sub-classes, `Hash1` and `Hash2` are derived to implement the 2 hash functions respectively. Finally, the `Double_Hash` class is defined with 2 data members: `hash1` and `hash2` function objects to implement the final hash function $h_i(k)$. Complete the following program in the space provided under Part(a)–(e) "`ADD YOUR CODE HERE`". The whole program is assumed to be all written in a single source file. A sample run of the program is given below:

```
---------- PROGRAM BEGINS ----------
Input size of the hash table? 13
Input base of the 2nd hash function? 7
Enter the hash key: 345
7
12
4
----------- PROGRAM ENDS -----------
```

```
class Hash              /* Parent Hash function object class — an abstract base class */
{
  protected:
    // Base of the mod function in a hash function,
    // which sometimes may be the size of a hash table
    int base;
  public:
    Hash(int m) : base(m) { }
    virtual int operator( )(int) = 0;
    /* Part (a) [2 points] ADD YOUR CODE HERE
     * Add the operator! function to return the base of the Hash table
     */
    int operator!( ) { return base; }   // 2 points
};
```

```
/* Part (b) [4 points] ADD YOUR CODE HERE
 * Implement a "minimal" sub-class Hash1 which inherits from Hash publicly.
 * Hash1 should implement the hash function object: hash1(k) = k mod m,
 * where m also represents the size of a hash table.
 */
class Hash1: public Hash                                          // 1 point
{
  public:
    Hash1(int m) : Hash(m) { }   // 1 point
    int operator( )(int key) { return key % base; }   // 2 points
};

/* Part (c) [4 points] ADD YOUR CODE HERE
 * Implement a "minimal" sub-class Hash2 which inherits from Hash publicly.
 * Hash2 should implement the hash function object: hash2(k) = R - (k mod R).
 */
class Hash2: public Hash                                          // 1 point
{
  public:
    Hash2(int m) : Hash(m) { }   // 1 point
    int operator( )(int key) { return base - (key % base); }   // 2 points
};

/* Double_Hash should implement the hash function object:
 * hash(k) = [ hash1(k) + i * hash2(k) ] mod m.
 */
class Double_Hash
{
  private:
    Hash1 hash1;                                   // Primary hash function
    Hash2 hash2;                                   // Secondary hash function
  public:
    /* Part (d) [3 points] ADD YOUR CODE HERE
     * Add necessary member function(s) to complete Double_Hash's definition
     */
    Double_Hash(int m, int R) : hash1(m), hash2(R) { }   // 1 point
    int operator( )(int key, int i)                        // 2 points
    {
        return (hash1(key) + i*hash2(key)) % (!hash1);
    }
};
```

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

int main( )
{
    int m;                                                  // Size of hash table
    int R;                          // Base of the mod function in the 2nd hash function
    int k;                                                  // Hash key
    cout << "---------- PROGRAM BEGINS ----------" << endl;
    cout << "Input size of the hash table?  "; cin >> m;
    cout << "Input base of the 2nd hash function?  "; cin >> R;
    cout << "Enter the hash key:  "; cin >> k;
    Double_Hash dhash(m, R);              // Create the double-hashing function object

    for (int i = 0; i < 3; ++i)
    {
        /* Part (e) [1 point] ADD YOUR CODE HERE
         * Add code here to print out the first 3 double-hashed values for key k
         */
        cout << dhash(k, i) << endl;                        // 1 point
    }

    cout << "----------- PROGRAM ENDS -----------" << endl;
    return 0;
}
```

Grading scheme: Note that the question asks for a minimal design for the subclasses. -1 point for additional (useless) data members.

Problem 5 [20 points] Class Template and Operator Overloading

Write a template class called `ListSum` that stores a list of numeric items using the STL `vector` and their numeric sum. *Assumption: only numeric types will be used and the value 0 is well-defined for them.* Your solution **MUST** implement the following functions:

- a default constructor
- a conversion constructor converting an object of type T to an object of `ListSum`
- a destructor
- overloaded operator≪ function (for outputting)
- overloaded operator= function (for assignment)
- overloaded operator+= function (for adding an item)

Moreover, whenever you have to go through the vector of stored items, you **MUST** use an appropriate iterator to do so. *Hint:* you may use the following typedef statement where appropriate; it defines a `const iterator` for `vector` of type `T`.

```
typedef typename vector<T>::const_iterator Tconst_iterator;
```

For simplicity, put all your function definition **inside** the class template definition in a single file which will be called "list-sum.h". Your solution should produce the following output:

```
list1: Items: 0  Sum: 0
list2: Items: 2  Sum: 2

list1: Items: 0 8  Sum: 8
list2: Items: 2 -5  Sum: -3

list1: Items: 2 -5  Sum: -3
list2: Items: 2 -5  Sum: -3
```

with the following driver program "list-sum-main.cpp".

```
#include <iostream>                                    /* File: list-sum-main.cpp */
using namespace std;

#include "list-sum.h"

// Print all elements of list1 and list2 together with their sums
void print_lists(const ListSum<int>& list1, const ListSum<int>& list2)
{
    cout << "list1:   " << list1 << endl;
    cout << "list2:   " << list2 << endl;
    cout << endl;
}

int main( )
{
    ListSum<int> list1;
    // list1's vector now contains an item 0 and its sum is initialized to 0
    ListSum<int> list2(2);
    // list2's vector now contains an item 2 and its sum is initialized to 2
    print_lists(list1, list2);

    list1 += 8;          // adds 8 to list1; now list1 contains [0, 8] and its sum is 8
    list2 += -5;        // adds -5 to list2; now list2 contians [2, -5] and its sum is -3
    print_lists(list1, list2);

    list1 = list2;                                                    // deep copy
    print_lists(list1, list2);
    return 0;
}
```

A reference to the STL vector class is given in Appendix I.

Answer: /* ADD YOUR CODE HERE */

```
#include <vector>                                        // [1 points] File: list-sum.h

template <class T>
class ListSum
{
    typedef typename vector<T>::const_iterator Tconst_iterator;

    friend ostream& operator<<(ostream& os, const ListSum& list_sum)   // [5 points]
    {
        os << "Items:  ";
        for (Tconst_iterator p = list_sum.vec.begin( ); p != list_sum.vec.end( ); p++)
        {
            os << *p << "  ";
        }

        os << "\tSum:  " << list_sum.sum;
        return os ;
    }

  public:
    ListSum( )                    // [2 points] Default constructor: initialize sum to zero
    {
        sum = 0;
        vec.push_back(0);
    }

    ListSum(T initValue)                           // [2 points] Conversion constructor
    {
        sum = initValue;
        vec.push_back(initValue);
    }

    ~ListSum( ) { }   // [1 points]

    const ListSum& operator+=(T x)                                    // [3 points]
    {
        sum += x;
        vec.push_back(x);
        return *this;
    }
```

```
    const ListSum& operator=(const ListSum& list_sum)                    // [5 points]
    {
        sum = list_sum.sum;
        vec = list_sum.vec;
        return *this;
    }

  private:                                                               // [1 points]
    vector<T> vec;
    T sum;
};
```

Problem 6 [18 points] Hashing

(a) A hash table of size $m = 11$ is used to store values with the hash function $hash(k) = k \bmod m$. Assume that the keys 15, 22, and 34 are already put into the table. Further insert the keys 32, 44, 76, 33, 91, and 54 in the given order (from left to right) to the hash table using each of the following open addressing collision resolution schemes, and compute the average number of collisions encountered (which is equal to the total number of collisions divided by 9). Truncate your answer to 1 decimal place.

Grading scheme:

0.5 point for each correct insertion; 1 point for average #collisions.

(i) [4 points] Linear probing

Index	Initial	Insert 32	Insert 44	Insert 76	Insert 33	Insert 91	Insert 54
0	22	22	22	22	22	22	22
1	34	34	34	34	34	34	34
2			44	44	44	44	44
3				76	76	76	76
4	15	15	15	15	15	15	15
5					33	33	33
6						91	91
7							54
8							
9							
10		32	32	32	32	32	32

Average number of collisions encountered: 22 / 9 = 2.4

(ii) [4 points] Quadratic probing

Index	Initial	Insert 32	Insert 44	Insert 76	Insert 33	Insert 91	Insert 54
0	22	22	22	22	22	22	22
1	34	34	34	34	34	34	34
2							
3				76	76	76	76
4	15	15	15	15	15	15	15
5					33	33	33
6							
7						91	91
8							54
9			44	44	44	44	44
10		32	32	32	32	32	32

Average number of collisions encountered: 14 / 9 = 1.5

(iii) [4 points] Double hashing with $hash2(k) = (k \mod (m-1)) + 1$.

Index	Initial	Insert 32	Insert 44	Insert 76	Insert 33	Insert 91	Insert 54
0	22	22	22	22	22	22	22
1	34	34	34	34	34	34	34
2							
3						91	91
4	15	15	15	15	15	15	15
5			44	44	44	44	44
6				76	76	76	76
7							
8					33	33	33
9							54
10		32	32	32	32	32	32

Average number of collisions encountered: 6 / 9 = 0.6

(b) [2 points] Based on the average number of collisions computed for each scheme in part(a), which scheme performed best? Explain your answer.

Answer:
Based on the average number of collisions computed in part(a), double hashing performed best. Since the probing sequence depends on keys, which drastically reduces the number of collisions when doing insertion.

(c) [2 points] For the two hash functions, $hash(k)$ and $hash2(k)$, used in part(a)(iii), will it be good to use $hash2(k)$ as the primary hash function and $hash(k)$ as the secondary hash function? Explain your answer.

Answer:
No. Since hash(k) can return 0 and hash2(k) skips the entry 0.

(d) [2 points] Suggest how to implement the deletion of a key from a hash table when using open addressing, and state how it affects the insertion and searching operations. Give your answers in no more than 60 words.

Answer:
Each hash table entry will have a special marker to indicate if it was deleted. This method is referred to as "Lazy Deletion". In this method, a deletion is done by just marking the table entry "deleted", rather than physically erasing deleting the entry. Deleted locations are treated as empty during insertion and search.

Problem 7 [24 points] Binary Search Tree

Generally, binary search trees (BSTs) may be used to store objects of type T. Except for part (a) below, other parts of this question assume that a BST is implemented as shown in our lecture notes, which is repeated below. *Assumption: all stored objects are distinct.*

```
template <typename T> class BST                                    /* File: bst.h */
{
    /* Part (c): friend declaration for the template global operator<< function
     *             which takes an ostream& and a const BST&, returning an ostream&
     *             and print the values in the latter level by level.
     */
  private:
    struct bst_node                                    // 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

        bst_node(const T& x) : value(x), left( ), right( ) { };
        /* Part (b):
         * Add a new constructor for bst_node
         */
    };
    bst_node* root;

  public:
    BST( ) : root(NULL) { }    // Empty BST when its root is NULL

    /* Part (b): Re-implement the following shallow copy constructor to a
     *             deep copy constructor for the BST class
     */
    BST(const BST& bst) { root = bst.root; }

    ~BST( ) { delete root; }
    bool is_empty( ) const { return root == NULL; }
    // ... and other member functions that you don't need ...
};
```

(a) [2 points] Draw 2 different BSTs that may be used to store the following 5 integers: $1, 2, 3, 4$, and 5.

Answer:

```
1                                   4
 \                                /   \
  2                              3     5
   \                            / \
    3                          1   2
     \
      4
       \
        5       etc.
```

(b) [6 points] In the given BST definition, its copy constructor is only doing shallow copy which is usually undesirable. Rewrite the BST copy constructor so that it will perform deep copy. That is, your answer will construct a new BST object by cloning the input BST object. You will also need to add a new constructor for the `bst_node` struct.

i. Add the following bst_node constructor inside its struct definition at the indicated location in the "bst.h" file.

Answer: `/* ADD YOUR CODE HERE */` [2 point]

```
bst_node(const T& x, const BST& L, const BST& R)
   : value(x), left(L), right(R) { };
```

ii. Implement the deep copy constructor outside the BST class template definition.

Answer: `/* ADD YOUR CODE HERE */` [4 points]

```
template <typename T>
BST<T>::BST(const BST& x)                                    // 1 point
{
    if ( x.is_empty( ) )                                     // 1 point
        root = NULL;
    else                                                     // 2 points
        root = new bst_node(x.root->value, x.root->left, x.root->right);
}
```

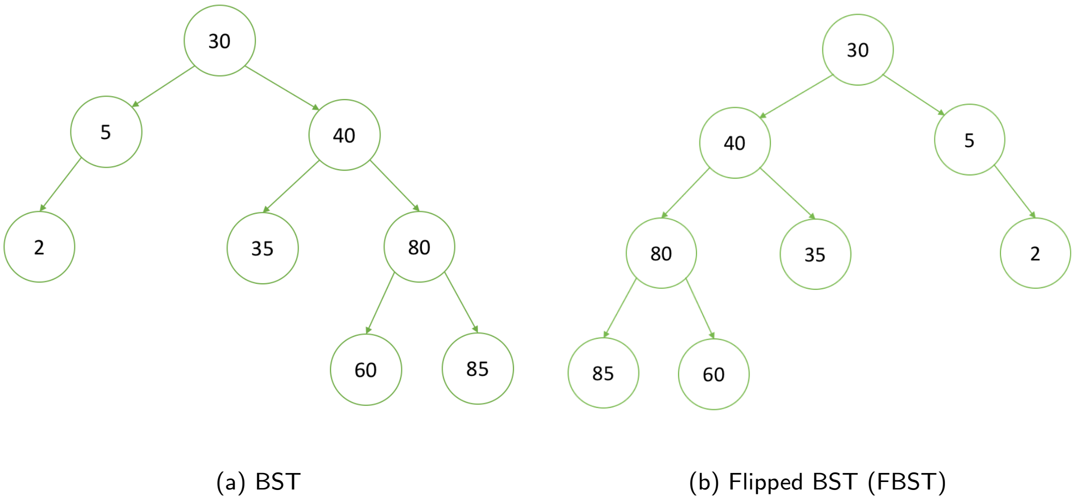

Figure 1: A BST and its flipped version (FBST)

Another solution:

```
bst_node(const bst_node* node) // 3 points
   : value(node->value), left(node->left), right(node->right) { };
```

```
template <typename T>
BST<T>::BST(const BST& x)                          // 1 point
{
    if ( x.is_empty( ) )
        root = NULL;                                // 1 point
    else
        root = new bst_node(x.root);                // 1 point
}
```

(c) [9 points] Implement the overloaded `operator`≪ as a friend non-member operator function to print out all values (separated by a single space) of a BST from top to bottom level by level — that is printing by level-order traversal of the tree. For example, the output for the BST in Figure 1(a) will be: `30 5 40 2 35 80 60 85`. To do that, you **MUST** use a STL queue to cache the un-visited nodes. A reference to the STL queue class is given in Appendix II.

i. Write down the declaration of the friend function `operator`≪ that you would put inside the BST class template definition in the place shown in the file "bst.h" above.
Answer: `/* ADD YOUR CODE HERE */` [1 point]

```
template <typename S>
friend ostream& operator<<(ostream& os, const BST<S>& bst);
```

ii. Write the friend function <u>definition</u> of `operator<<` <u>outside</u> the BST class template definition.

Answer: `/* ADD YOUR CODE HERE */` [8 points]

```
#include <queue>                                              // 0.5 point
template <typename T>
ostream& operator<<(ostream& os, const BST<T>& bst)          // 0.5 point
{
    if (bst.is_empty( ))                                      // 0.5 point for the test
        return os;

    queue<const BST<T>*> Q;                                   // 0.5 point
    Q.push(&bst);                                             // 1 point

    while (!Q.empty( ))                                       // 0.5 point for the test
    {

        const BST<T>* bstp = Q.front( );                      // 1 point
        Q.pop( );                                             // 1 point
        os << bstp->root->value << " ";                       // 0.5 point

        if (!bstp->root->left.is_empty( ))                    // 1 point
            Q.push(&bstp->root->left);

        if (!bstp->root->right.is_empty( ))                   // 1 point
            Q.push(&bstp->root->right);
    }

    return os;
    // -0.5 point if no return value (otherwise no point for this statement)
}
```

(d) [7 points] Now we want to convert a BST to a flipped BST (FBST). An FBST is similar to a BST but with the opposite property: the value stored in any children in the left subtree of a parent node is greater than the parent node's value, and any children in the right subtree has a value smaller than parent's. An example is given in Figure 1. Below is the class template definition of FBST. *For this part, it is assumed that FBST<T> has been declared as a friend class of BST<T>.*

```
template <typename T>                                              /* File: fbst.h */
class FBST
{
  private:
    struct fbst_node                                   // A node in a binary search tree
    {
        T value;
        FBST left;                                     // Left sub-tree or called left child
        FBST right;                                    // Right sub-tree or called right child
        fbst_node(const T& x) : value(x), left( ), right( ) { };
        /* Part (d): Add a new constructor for fbst_node in order to implement
         *           the conversion constructor of the FBST class
         */
    };
    fbst_node* root;

  public:
    FBST( ) : root(NULL) { }   // Empty FBST when its root is NULL
    ~FBST( ) { delete root; }
    bool is_empty( ) const { return root == NULL; }
    void print(int depth = 0) const;              // No need to implement this function
    FBST(const BST<T>& bst);                      // Part (d): Convert a BST to FBST
};
```

Implement the conversion constructor of FBST which converts a BST to an FBST <u>outside</u> its class definition by doing the following 2 steps.

i. Add the following `fbst_node` constructor <u>inside</u> its struct definition at the indicated location in the "fbst.h" file.

Answer: `/* ADD YOUR CODE HERE */`

```
fbst_node(const T& x, const FBST& L, const FBST& R) // 2 points
    : value(x), left(L), right(R) { };
```

ii. Implement the conversion constructor <u>outside</u> the FBST class template definition.

Answer: /* ADD YOUR CODE HERE */

```
template <typename T>
FBST<T>::FBST(const BST<T>& x)                                    // 1 point
{
    if ( x.is_empty( ) )                                          // 1 point
        root = NULL;
    else
    {
        FBST* left_fbst = new FBST(x.root→right);
        FBST* right_fbst = new FBST(x.root→left);
        root = new fbst_node(x.root→value, *left_fbst, *right_fbst);   // 3 points

        /* The following code doesn't work but is close because the 2nd and 3rd
           parameters are temporary FBST's which will be deleted after the call.
           Thus, their associated fbst_node* will become dangling pointers!
           => Give 2 points.

        root = new fbst_node(
                    x.root->value, FBST(x.root->left), FBST(x.root->right) );
        */
    }
}
```

Another solution:

```
    typedef typename BST<T>::bst_node bst_node_T;   // 3 points
    fbst_node(bst_node_T* node)
       : value(node->value), left(node->right), right(node->left) { };

template <typename T>
FBST<T>::FBST(const BST<T>& x)                                    // 1 point
{
    if ( x.is_empty( ) )                                          // 1 point
        root = NULL;
    else
        root = new fbst_node(x.root);                             // 2 points
}
```

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

Appendix I

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( )	Constructor
iterator begin( ) const_iterator begin( ) const	Returns an iterator pointing to the first element in the vector. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
iterator end( ) const_iterator end( ) const	Returns an iterator referring to the past-the-end element in the vector container. If the vector object is const-qualified, the function returns a const_iterator. Otherwise, it returns iterator.
void clear( )	Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.
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.

Appendix II

STL Container Adaptor: Queue

```
template <class T, class Container = deque<T> > class queue;
```

Defined in the standard header **queue**.

Description:

Queues are implemented as container adaptors, and are designed to operate in a FIFO context (first-in first-out), where elements are pushed into the "back" of the specific container and popped from its "front". The underlying container may be one of the standard container class template or some other specifically designed container class.

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

Member function	**Description**
queue( )	Constructor
bool empty( ) const	Returns true if the size is 0, false otherwise
size_type size( ) const	Returns the number of elements in the queue. (Note: size_type is an unsigned integral type)
T& front( ) const T& front( ) const	Returns a reference to the next element in the queue. The next element is the "oldest" element in the queue.
T& back( ) const T& back( ) const	Returns a reference to the last element in the queue. The last element is the "newest" element in the queue.
void push(const T& val)	Inserts a new element, val, at the end of the queue, after its current last element.
void pop( )	Removes the next element in the queue. The element removed is the "oldest" element in the queue.