COMP 2012 Final Exam - Spring 2017 - HKUST

Date: May 19th, 2017 (Friday)
Time Allowed: 3 hours, 8:30 – 11:30 am
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **8** questions on **34** pages (including this cover page, 2 appenedix pages, and 3 blank pages at the end).
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 structures, or use global variables nor any library functions not mentioned in the questions. But you may use the STL functions given in the Appendix.

Student Name	Solution & Marking Scheme
Student ID	
Email Address	
Seat Number	

For T.A. Use Only

Problem	Topic	Score
1	True or False	/ 10
2	Class Template	/ 3
3	STL & Function Object	/ 7.5
4	Static Member Data/Function	/ 7.5
5	STL & Operator Overloading	/ 15
6	BST and AVL Tree	/ 23
7	Hashing: Separate AVL Trees	/ 21
8	Hashing: Open Addressing	/ 13
Total		/ 100

Problem 1 [10 points] True or false

Indicate whether the following statements are *true* or *false* by circling **T** or **F**. You get 1.0 point for each correct answer, −0.5 for each wrong answer, and 0.0 if you do not answer.

T **F** (a) The "this" pointer is implicitly supplied to all static and non-static member functions of a class as the first function argument.

T **F** (b) Overloading a post-increment operator can be done by defining a member function that takes an int argument.

T **F** (c) Public data members of a base class can always be accessed by objects of its directly derived classes regardless of the type of inheritance being used — that is, regardless of whether public, protected or private inheritance is used.

T **F** (d) Constructor **cannot** be virtual.

T **F** (e) A friend function must be declared/defined as public inside a class, otherwise it cannot be called by other functions not belonged to the class.

T **F** (f) There is no way to instantiate the following function template.

```
template <typename T>
T* func() { return new T; }
```

T **F** (g) When the following C++ program is executed, the displayed output contains "Derived".

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base { public: void func() {} };
class Derived : public Base { public: void func() {} };
int main()
{
    Base* bp = new Derived;
    cout << typeid(*bp).name() << endl;
    return 0;
}
```

T **F** (h) Non-static member functions can access both static and non-static members of class.

T **F** (i) Suppose a hash table has more slots than the size of the universe of (unique) keys, then there exists a hash function such that there will be no collisions.

T **F** (j) If we retrieve the key values stored in a binary search tree (BST) using the postorder traversal method, the first key value retrieved will always be the smallest key value stored in the BST.

Problem 2 [3 points] Template of N-ary Tree

N-ary trees are trees in which each node may have up to a maximum of N subtrees. Let's call them `Ntree`s. Write down the **template** class definition of `Ntree` (in the spirit of the BST defined in the lecture notes and in Problem 6) in which each node should have N sub-Ntree objects (not their pointers) though they can be empty trees. Here are some requirements:

- each tree node should store a value of type **T**
- each tree node should have N subtree objects of type **Ntree**
- give a complete definition of the private struct `node` including one constructor of any kind
- you do not need to declare/define any member functions of `Ntree`
- your definition should support `Ntree` instantiations such as

```
Ntree<int, 5> x;
Ntree<string, 8> y;
```

Solution:

```
template <typename T, int N>                // 1 point
class Ntree
{
  private:
    struct node
    { // TO DO: Complete the definition of node
        T value;                            // 0.5 point
        Ntree child[N];                     // 1 point
        node(const T& x) : value(x) { }     // 0.5 point
                                            // Accept any correct constructor
    };

    node* root;

  public:
    /* Assume member functions of Ntree are given here */
};
```

Problem 3 [7.5 points] STL and Function Objects

(a) [3 points] Given the following prototype of the STL `replace_if` algorithm,

```
template <typename ForwardIt, typename UnaryPredicate, typename T>
void replace_if(ForwardIt first, ForwardIt last,
UnaryPredicate p, const T& new_value);
```

implement the algorithm so that it replaces those elements of a sequence container accessed by its iterator in the range of [`first, last`) to the `new_value` if the predicate `p` returns true for them.

Solution

```
template <typename ForwardIt, typename UnaryPredicate, typename T>
void replace_if(ForwardIt first, ForwardIt last, UnaryPredicate p, const T& new_value )
{
    while (first != last)       // 0.5 point
    {
        if (p(*first))          // 0.5 for dereference, 0.5 for passing it to function
            *first = new_value; // 0.5 for dereference, 0.5 for assigning new_value

        ++first;                // 0.5 point
    }
}
```

(b) [3 points] Write a function object class called **LessThan** which will be put in a file called "LessThan.h" that has the following:

- a private data member, **threshold**, of type `int`
- a constructor
- an overloaded function-call operator function

The constructor of the **LessThan** class should initialize its function objects with a threshold so that when a function object of the class is called with a value, it will compare the given value with its threshold and returns true if the value is less than its threshold, otherwise false.

Solution

```
class LessThan      // 0.5 point
{
  private:
    int threshold;  // 0.5 point

  public:
    LessThan(int t) : threshold(t) { }                          // 1 point
    bool operator()(int value) { return (value < threshold); } // 1 point
};
```

(c) [1.5 points] Complete the following program in the space provided after the "**TO DO**" comment lines so that it will run and gives the output below:

```
Updated marks:
50
0
20
0
```

Solution:

```
#include <iostream>     /* File: test-replace-if.cpp */
#include <algorithm>    // This includes the definition of replace_if
#include <vector>
#include "LessThan.h"
using namespace std;

int main()
{
    vector<int> marks;
    marks.push_back(50); marks.push_back(-10);
    marks.push_back(20); marks.push_back(-1);

    /*
     * TO DO: Using replace_if from part (a) together with some function object
     * of type LessThan defined in part (b), replace all negative values stored
     * in the STL vector marks to 0.
     *
     *****************  CODE BEGINS  *****************/

    // 0.5 point for marks.begin()
    // 0.5 point for marks.end()
    // 0.5 point for LessThan(0)

    replace_if(marks.begin(), marks.end(), LessThan(0), 0);

    /****************   CODE ENDS   *****************/

    cout << "Updated marks:" << endl;
    for(int i = 0; i < marks.size(); i++)
        cout << marks[i] << endl;

    return 0;
}
```

Problem 4 [7.5 points] Static Member Data/Function

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

class Snacks
{
  protected:
    static int num;
    string name;

  public:
    Snacks() { ++num; }
    Snacks(string name) : name(name) { ++num; }
    ~Snacks() { --num; }
    string get_name() const { return name; }
    void total() { cout << "Total snacks created is : " << num << "\n\n"; }
};

class Biscuit: public Snacks
{
  private:
    static int num;

  public:
    Biscuit() : Snacks("Oreo") { ++num; show(num); }
    ~Biscuit() { --num; show(num); }
    void show(int pieces)
        { cout << "There are " << pieces << " biscuits now!" << endl; }
};

class Candy: public Snacks
{
  private:
    static int num;
    string color;

  public:
    Candy() : Snacks("Coffee Candy"), color("Brown") { ++num; show(num); }
    ~Candy() { --num; show(num); }
    void show(int pieces)
        { cout << "There are " << pieces << " candies now!" << endl; }
};

```

```
44  class Child
45  {
46    private:
47      string name;
48      Snacks** s;
49
50    public:
51      Child(string name) : name(name)
52      {
53          cout << "Hello I am " << name << endl;
54          s = new Snacks*[4];
55          for (int i = 0; i < 2; i++)
56          {
57              s[i] = new Biscuit;
58              s[i+2]= new Candy;
59          }
60          cout << endl;
61      }
62
63      ~Child()
64      {
65          for (int i = 0; i < 4; i++)
66              delete s[i];
67          delete [] s;
68      }
69  };
70
71  int main()
72  {
73      Snacks::total();
74      Child C1("Tommy");
75      Child C2("John");
76      Snacks::total();
77      return 0;
78  }
```

The program on the previous page contains 5 errors (syntax errors, logical errors, missing statements, etc). Study the program carefully, identify all the errors by writing down the line numbers at which they occur, and give the reasons. For the missing statements, sometimes there are more than one place to add them; choose any line number where they should be added to make the program correct. After fixing the 5 errors, the program is expected to produce the following output.

```
Total snacks created is : 0

Hello I am Tommy
There are 1 biscuits now!
There are 1 candies now!
There are 2 biscuits now!
There are 2 candies now!

Hello I am John
There are 3 biscuits now!
There are 3 candies now!
There are 4 biscuits now!
There are 4 candies now!

Total snacks created is : 8

There are 3 biscuits now!
There are 2 biscuits now!
There are 3 candies now!
There are 2 candies now!
There are 1 biscuits now!
There are 0 biscuits now!
There are 1 candies now!
There are 0 candies now!
```

Solution

Error#	Line#	Explanation
1	14	This should be a virtual destructor. virtual ~Snacks(){−−num;}
2	16	total() must be a static function, otherwise can't call it before creating an object in the main() function . The line should start with "static"
3	anywhere in the open space after Snacks class	Line missing, we need to init the static data member outside the class. We need to add the line: int Snacks::num=0;
4	anywhere in the open space after Biscuit class	Line missing, we need to init the static data member outside the class. We need to add the line: int Biscuit::num=0;
5	anywhere in the open space after Candy class	Line missing, we need to init the static data member outside the class. We need to add the line: int Candy::num=0;

Marking scheme: 0.5 points for each correct line number, 1 point for each correct explanation.
For Error 2: Alternative explanation accepted, "we should not call Snacks::total() in this way, it is not a static function".
For Errors 3/4/5: (1) 0.5 for correct line number, 0.75 for explanation, 0.25 for the correct statement. (2) If the line number is 8/22/34 (where the static members are defined), 0.5 each if the answer correctly points out where to add the missing line in the explanation. (3) If these 3 errors are written as a whole, the explanation must explicitly point out every static data member of three classes. Otherwise it is treated as 1 error.

Problem 5 [15 points] STL and Operator Overloading

```
#include <iostream>     /* File "set.h" */
#include <vector>
using namespace std;

template <typename T>
class Set           // Note: Its elements are ALWAYS arranged in ascending order
{                   //       of their values BEFORE and AFTER any operation
  private:
    vector<T> set; // A vector container that stores set elements

  public:
    Set() { }       // Default constructor

    // Constructor that initializes this "set" with a given array "arr"
    // Note: After initialization, the elements are arranged in ascending order
    Set(T* arr, int size) : set(arr, arr + size) // Using STL's set constructor
        { sort(set.begin(), set.end()); } // STL sort() puts elements into ascending order

    // Compare this "set" and another set "s". Return true if they contain the
    // same set of elements, otherwise return false
    bool operator==(const Set& s) const;

    // Perform union of this "set" and another set "s" and return the resulting set
    // Definition of union operation: Union of two sets is the set that contains
    // all the elements of two sets with no duplicates
    // Note: After union, the elements in the resulting set should be arranged
    //       in ascending order
    Set operator+(const Set& s) const;

    // Perform union of this "set" and an "item", and return the resulting set
    // Note: After union, the elements in the resulting set should be arranged
    //       in ascending order
    Set operator+(const T& item) const;

    // Update this "set" with union of this "set" and another set "s"
    Set& operator+=(const Set& s);

    // Update this "set" with union of this "set" and an "item"
    Set& operator+=(const T& item);

    // Check whether "item" is in this set.
    // Return true if it is, otherwise return false
    bool contains(const T& item) const;

    // Overload the insertion operator<< for the Set class
    template <typename S>
        friend ostream& operator<<(/* complete the prototype as well */);
};

#include "set.cpp"
```

Implement the 7 undefined member functions of the above template class 'Set' in "set.cpp" so that they will work with the testing program below to produce the following output.

```
#include <string>       /* File: "test-set.cpp" */
#include "set.h"
int main()
{
    string arr1[] = { "Desmond", "Alex", "Brian" };
    Set<string> set1(arr1, sizeof(arr1) / sizeof(string));
    string arr2[] = { "James", "Desmond", "Raymond" };
    Set<string> set2(arr2, sizeof(arr2) / sizeof(string));

    cout << "set1: " << set1 << endl;
    cout << "set2: " << set2 << endl;
    cout << ( (set1 == set2) ? "set1 = set2" : "set1 != set2" ) << endl << endl;

    Set<string> set3 = set1 + set2;
    cout << "After set3 = set1 + set2" << endl;
    cout << "set3: " << set3 << endl << endl;

    set3 += "Cecia";
    cout << "After adding \"Cecia\" to set3" << endl;
    cout << "set3: " << set3 << endl << endl;

    string arr4[] = { "Albert", "Gary" };
    Set<string> set4(arr4, sizeof(arr4) / sizeof(string));
    cout << "set4: " << set4 << endl << endl;

    set3 += set4;
    cout << "After set3 += set4" << endl;
    cout << "set3: " << set3 << endl;
    return 0;
}
```

Output of the program:

```
set1: [ Alex, Brian, Desmond ]
set2: [ Desmond, James, Raymond ]
set1 != set2

After set3 = set1 + set2
set3: [ Alex, Brian, Desmond, James, Raymond ]

After adding "Cecia" to set3
set3: [ Alex, Brian, Cecia, Desmond, James, Raymond ]

set4: [ Albert, Gary ]

After set3 += set4
set3: [ Albert, Alex, Brian, Cecia, Desmond, Gary, James, Raymond ]
```

Hint: You may use previously defined overloaded operator functions to define other ones.

Solution:

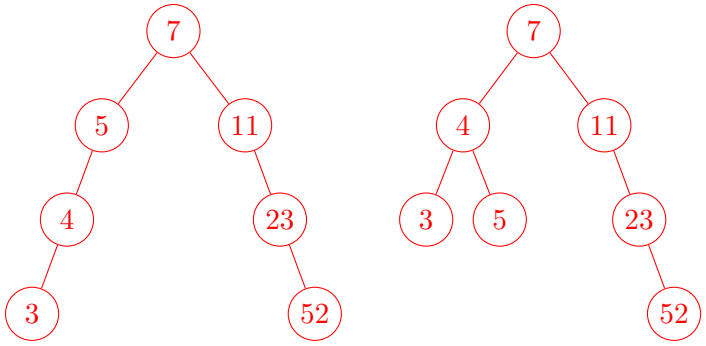
```
// Operator function: bool operator==(const Set& s) const;
template <typename T>
bool Set<T>::operator==(const Set& s) const
{
    return (set == s.set);                          // 1 point
}


// Operator function: Set operator+(const Set& s) const;
template <typename T>
Set<T> Set<T>::operator+(const Set& s) const
{
    Set<T> newSet = *this;                          // 1 point
    for (int i = 0; i < s.set.size(); i++)          // 0.5 point
    {
        if (!newSet.contains(s.set[i]))             // 1 point
            newSet.set.push_back(s.set[i]);         // 1 point
    }

    sort(newSet.set.begin(), newSet.set.end());     // 1 point
    return newSet;                                  // 0.5 point
}


// Operator function: Set operator+(const T& item) const;
template <typename T>
Set<T> Set<T>::operator+(const T& item) const
{
    T arr[] = {item};                               // 0.5 point
    return (*this + Set<T>(arr,1));                 // 1 point
}


// Operator function: Set& operator+=(const Set& s) const;
template <typename T>
Set<T>& Set<T>::operator+=(const Set& s)
{
    return (*this = *this + s);                     // 1.5 points
}
```

```
// Operator function: Set& operator+=(const T& item) const;
template <typename T>
Set<T>& Set<T>::operator+=(const T& item)
{
    return (*this = *this + item);                  // 1.5 points
}


// Member function: bool contains(const T& item) const;
template <typename T>
bool Set<T>::contains(const T& item) const
{
    return ( find(set.begin(), set.end(), item) != set.end() );    // 1.5 points
}


// Operator function: ostream& operator<<(ostream& os, const Set<T>& s)
template <typename T>
ostream& operator<<(ostream& os, const Set<T>& s)
{
    os << "[ ";
    for(int i = 0; i < s.set.size(); i++)                          // 0.5 point
        os << s.set[i] << ( (i != s.set.size() - 1) ? ", " : "");  // 2 points
    os << " ]";

    return os;                                                     // 0.5 point
}
```

Problem 6 [23 points] BST and AVL Trees

(a) [2 points] Given 7 values: 3, 4, 5, 7, 11, 23 and 52, draw *2 different* binary search trees (BSTs) that store the 7 values and have a height of 3.

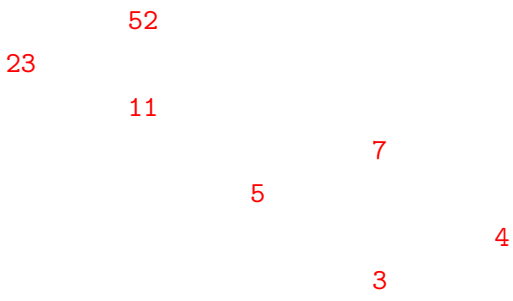
Answer:



Marking scheme: 1 point for giving each tree (2 points in total)

(b) [3 points] Suppose the postorder traversal of a BST that holds the above 7 values will visit the nodes in the following order: 4, 3, 7, 5, 11, 52, 23. Draw the original BST.

Answer:



(c) [18 points] Below is the template class definition of `BST` (which is similar to the one on our lecture notes) in the file "bst.h".

```
/* File: bst.h */
template <typename T> class BST
{
  protected:
    struct node
    {
        T value;
        BST left;
        BST right;
        node(const T& x) : value(x), left(), right() { }
    };

    node* root;

  public:
    BST() : root(NULL) { }
    ~BST() { delete root; }

    /* ASSUME THAT THE FOLLOWING 3 FUNCTIONS ARE GIVEN */
    bool empty() const { return root == NULL; }
    void insert(const T& x); // Insert value x to the tree
    void print(int depth = 0) const; // Print a rotated BST

    /***** TO DO: IMPLEMENT ONLY THE FOLLOWING 4 FUNCTIONS *****/

    // To find the height of a BST
    // Note that the height of an empty tree is -1, and that of a leaf is 0.
    int height() const;

    // To print the values of leaf nodes from left to right in ascending order
    void print_leaves() const;

    // To check if the BST is also an AVL tree
    bool is_avl() const;

    // To perform an inorder BST traversal and print ONLY those values that
    // satisfy the boolean function which is passed as the only argument to this
    // member function. You have to decide the exact prototype of the function.
    void inorder_print_if( /* complete the prototype */ ) const;
};
```

The following is a test program of the class.

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

bool is_even(const int& x) { return x%2 == 0; }

int main()
{
    BST<int> x;
    x.insert(10); x.insert(6); x.insert(3); x.insert(4);
    x.insert(15); x.insert(18); x.insert(19); x.insert(13);
    x.insert(8); x.insert(2); x.insert(1); x.insert(19);
    x.print();

    cout << "\n\nHeight = " << x.height();

    cout << "\n\nPrint leaves:  "; x.print_leaves();

    cout << "\n\nPrint even node values:  "; x.inorder_print_if(is_even);

    cout << "\n\nIs the BST an AVL?  " << boolalpha << x.is_avl() << endl;
    return 0;
}
```

which gives the following output:

```
                        19
                18
        15
                13
10
                8
        6
                        4
                3
                        2
                                1
Height = 4

Print leaves:  1 4 8 13 19

Print even node values:  2 4 6 8 10 18

Is the BST an AVL?  false
```

Implement the following 4 member functions:

i. `int height() const;`

ii. `void print_leaves() const;`

iii. `bool is_avl() const;`

iv. `void inorder_print_if(/* complete the prototype */) const;`

in a *separate* file called "`bst.cpp`" so that they will work with the test program to produce the above required outputs. Note:

- You have to complete the prototype of the 4th member function `inorder_print_if`.
- You may assume that the data type T always supports the `operator<<` function so that its value can be output using it.
- For the 2 *print* functions, separate output values by one space.

Solution

```
/* 3 points */
template <typename T>
int BST<T>::height() const // 0.5 point
{
    if (empty())           // 0.5 point
        return -1;

    return 1 + max(root->left.height(), root->right.height()); // 2 points
}


// 5 points
template <typename T>
void BST<T>::print_leaves() const // 0.5 point
{
    if (empty())                  // Base case:  0.5 point
       return;

    if (root->left.empty() && root->right.empty())  // 2 points
        cout << root->value << ' ';
    else
    {
        root->left.print_leaves();  // Recursion: left subtree -- 1 point
        root->right.print_leaves(); // Recursion: right subtree -- 1 point
    }
}
```

```
/* 5 points */
template <typename T>
bool BST<T>::is_avl() const // 0.5 point
{
    if (empty())            // 0.5 point
        return true;

    int bfactor = root->right.height() - root->left.height(); // 1 point

    if (bfactor > 1 || bfactor < -1) // 1 points
        return false;
    else
        return root->left.is_avl() && root->right.is_avl(); // 2 points
}


/* 5 points */
template <typename T>
void BST<T>::inorder_print_if(bool (*f)(const T&)) const // 1.5 point
{
    if (empty())        // Base case // 0.5 point
       return;

    root->left.inorder_print_if(f);  // Recursion: left subtree -- 1 point

    if (f(root->value))
        cout << root->value << ' ';  // 1 point

    root->right.inorder_print_if(f); // Recursion: right subtree -- 1 point
}
```

Problem 7 [21 points] Hashing: Separate AVL Trees

A hash table of size $m = 5$ is used to store keys with the hash function

$$hash(k) = k \mod m.$$

Fifteen keys of the following integral values:

18, 10, 7, 61, 11, 64, 40, 12, 21, 31, 34, 17, 121, 41, 14

are to be inserted into the table in the given order (from left to right) using *separate chaining-like* collision resolution. That is, instead of using a linked list, an AVL tree is used for linking up all the keys that are hashed to the same slot. Each entry of the hash table will store a pointer to the root of the AVL tree.

For each key insertion, in the space provided in the following table,

i. **Draw the AVL tree** that results from each key insertion.

ii. If rotation(s) is/are required, indicate the type of insertion that causes violation of the AVL tree property by putting a **tick** ✓ in one of the given checkboxes, namely LL, LR, RL, and RR, where

- LL stands for inserting the key into the left sub-tree of the left child of a node
- LR stands for inserting the key into the right sub-tree of the left child of a node
- RL stands for inserting the key into the left sub-tree of the right child of a node
- RR stands for inserting the key into the right sub-tree of the right child of a node

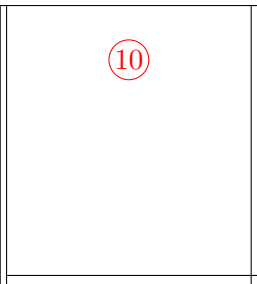
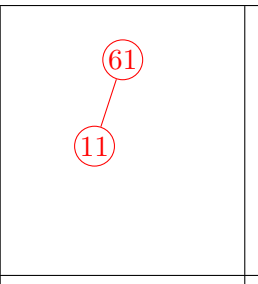
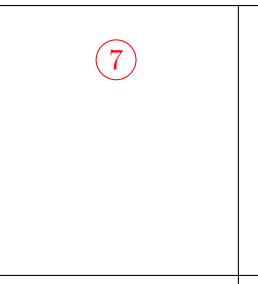
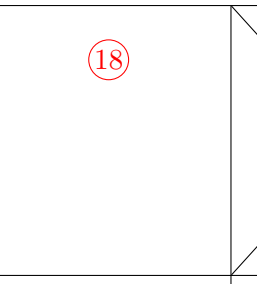
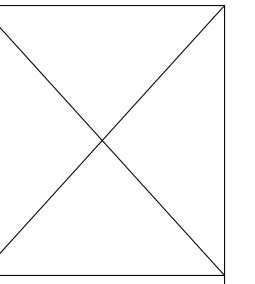
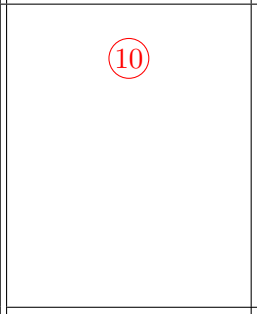
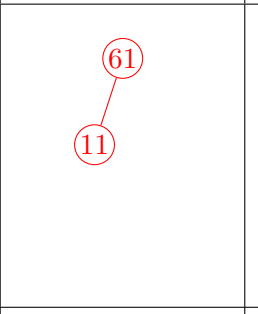
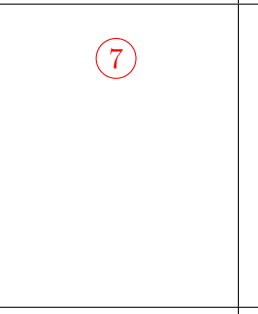
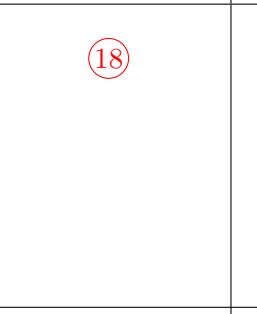
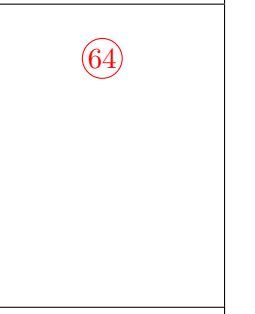
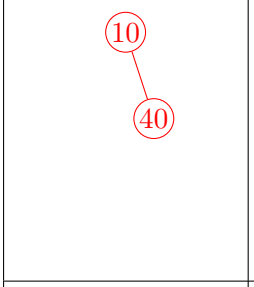
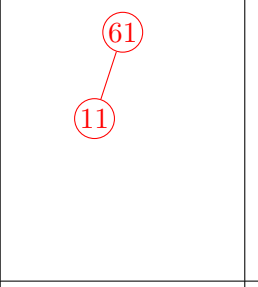
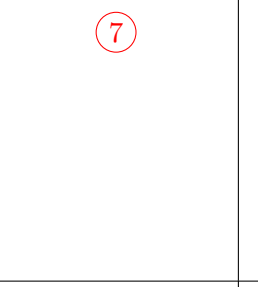
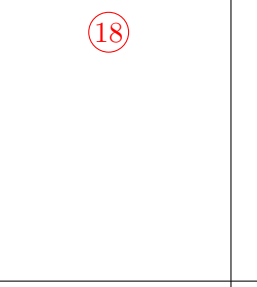
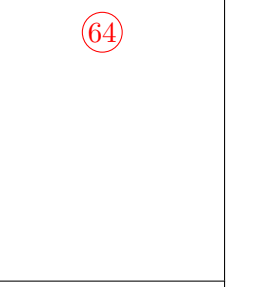
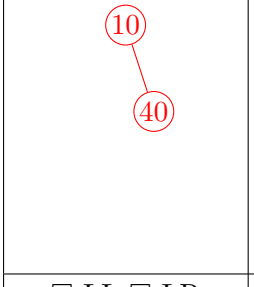
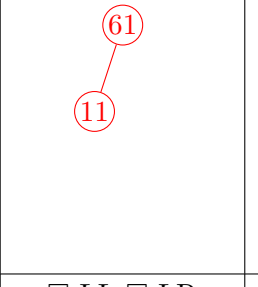
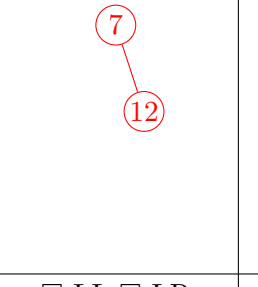
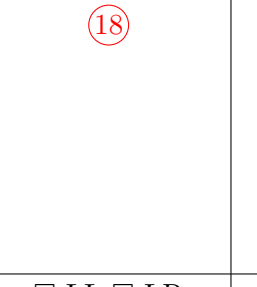
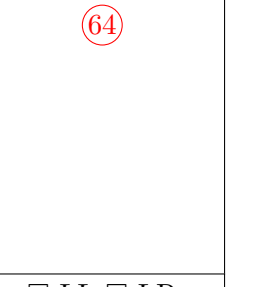
iii. **Cross out** the table entries where there will be no inserted keys. The first two keys, 18 and 10, have been inserted for you as an example.

Marking scheme: 1 point each for simple insertion; 3 points for the 4 with rotations

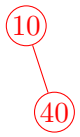
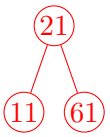
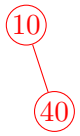
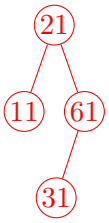
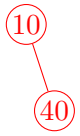
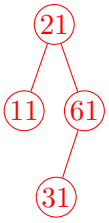
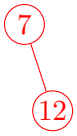
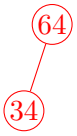
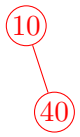
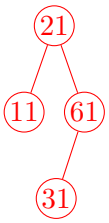
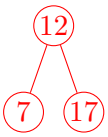
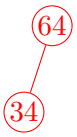
Solution:

Table Index	0	1	2	3	4
Insert 18				(18)	
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 10	(10)			(18)	
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 7	(10)		(7)	(18)	
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 61	(10)	(61)	(7)	(18)	
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR

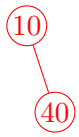
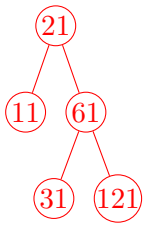
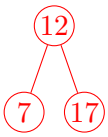
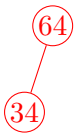
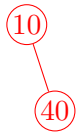
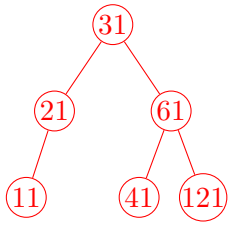
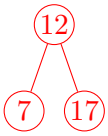
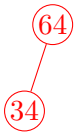
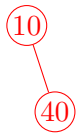
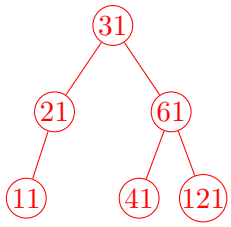
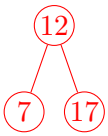
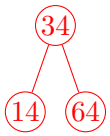
Continued:

	0	1	2	3	4
Insert 11					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 64					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 40					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 12					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR

Continued:

	0	1	2	3	4
Insert 21					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☑ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 31					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 34					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 17					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☑ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR

Continued:

	0	1	2	3	4
Insert 121					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 41					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☑ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR
Insert 14					
	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☐ LL ☐ LR ☐ RL ☐ RR	☑ LL ☐ LR ☐ RL ☐ RR

Problem 8 [13 points] Hashing: Open Addressing

Fill in the following hash tables for each of the stated open addressing hashing methods when the following eight integers: 2, 3, 7, 33, 24, 29, 30, and 22 are inserted sequentially to the tables. The following hash function is to be used for this question:

$$hash(k) = k \mod 11$$

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, 2, has been done for you in the table below.

Solution:

(a) [6 points] Linear probing: $h_i(k) = (hash(k) + i) \mod 11$

Table index	insert 2	insert 3	insert 7	insert 33	insert 24	insert 29	insert 30	insert 22
0				33	33	33	33	33
1								22
2	2	2	2	2	2	2	2	2
3		3	3	3	3	3	3	3
4					24	24	24	24
5								
6								
7			7	7	7	7	7	7
8						29	29	29
9							30	30
10								

Marking scheme:

- 0.5 points for inserting 2, 3, 7, 33 each (2 is in fact done, so that 0.5 is free!).
- 1 point for inserting 24, 29, 30, 22 as these insertions encounter collisions.

(b) [7 points] Double hashing: Use the $hash_2$ function below as the second hash function:

$$hash_2(k) = 7 - (k \mod 7).$$

Table index	insert 2	insert 3	insert 7	insert 33	insert 24	insert 29	insert 30	insert 22
0				33	33	33	33	33
1							30	30
2	2	2	2	2	2	2	2	2
3		3	3	3	3	3	3	3
4								
5								
6					24	24	24	24
7			7	7	7	7	7	7
8						29	29	29
9								22
10								

Marking scheme:

- 0.5 points for inserting 2, 3, 7, 33 each.
- 1 point for inserting 24, 29, 30, as they encounter one to three collisions.
- 2 points for inserting 22, as it encounters seven collisions.

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

Appendix

(1) 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 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.

(2) STL vector has another constructor that can be used to construct a vector object from an array as shown below:

```
int array[] = {1,2,3};
vector<int> object(array, array + sizeof(array)/sizeof(int));
```

The constructor creates a vector with the range [first,last) using a specified allocator and is defined as follows:

```
template <typename InputIterator>
vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

(3) STL's **find** function which returns the first element in the range [*first*, *last*) that is equal to *value*.

```
template <typename InputIt, typename T>
InputIt find(InputIt first, InputIt last, const T& value);
```

(4) STL's **sort** function sorts the elements in a container in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. Elements are compared using `operator<()`.

```
template <typename RandomIt>
void sort(RandomIt first, RandomIt last);
```