COMP 152H: OOP and Data Structures (Honors Track)
Spring Semester 2010
Final Exam
Instructor: Chi Keung Tang
Monday, May 24, 2010
8:30 – 11:30am
Room 2463

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of **nine** (9) problems. Follow the instructions carefully. Please write legibly in the space provided. Keep the exam booklet stapled.

K E Y

Problem	Points	your score
1 UNKNOWN	10	
2 STATIC CLASS	10	
3 FUNCTORS AND FUNCTION POINTERS	10	
4 COMPLEXITY	10	
5 SORTING	10	
6 HEAP	10	
7 AVL AND BINARY TREES	20	
8 GRAPH	8	
9 DIFFICULT PROBLEM	12	
Total	100	

1 Abstract Data Type

This problem concerns with the design and implementation of a container class `Array`, where you should apply what you learned about templates and operator overloading.
Note that your implementation should be free of any dangling pointers/references or memory leaks in order to avoid mark deduction.
Code up `array.h` so that the following main program can compile. The sample input and output are shown.

```
//------------------------main.cpp---------------------------------
#include <iostream>
#include "array.h"
using namespace std;

int main()
{
  int i, size;
  cout << "Enter size : "; cin >> size;
  Array<char> a(size);
  // ASCII code for 'A' is 65.
  for (i=0;i<6;i++) { a[i] = i+65; cout << (char)(a[i]) << " " ; }
  cout << endl;

  Array<char> b(a);
  for (i=0;i<size;i++) { cout << (char)(b[i]) << " "; }
  cout << endl;
  cout << ((a.getsize() == b.getsize()) ? "true" : "false") << endl;
  cout << a[100] << endl;  // out of bound
  b[10000] = 'j';          // out of bound
  for (i=0;i<size;i++) {  // the array b is unchanged
   cout << (char)(b[i]) << " ";
  }
  cout << endl;

  Array<char> c(10), d(10);
  d = c = b;  cout << "cascading assignment..." << endl;
  for (i=0;i<d.getsize();i++) d[i] += 1;
  for (i=0;i<d.getsize();i++) cout << d[i] << " "; cout << endl;
  for (i=0;i<c.getsize();i++) c[i] += 2 ;
  for (i=0;i<c.getsize();i++) cout << c[i] << " "; cout << endl;
  for (i=0;i<b.getsize();i++) cout << b[i] << " ";
  cout << "<--- b remains unchanged..." << endl;

  return 0;
}
```

Sample input and output:

```
Enter size : 4
A B C D array index 4 out of bound - value UN trustworthy
array index 4 out of bound - value UN trustworthy
E array index 5 out of bound - value UN trustworthy
array index 5 out of bound - value UN trustworthy
F
A B C D
true
array index 100 out of bound - value UN trustworthy
```

```
F
array index 10000 out of bound - value UN trustworthy
A B C D
cascading assignment...
B C D E
C D E F
A B C D <--- b remains unchanged...
```

Write your answer on this and the next page. **Key:**

```
#ifndef ARRAY_H
#define ARRAY_H

template <typename T>
class Array {
public:
   Array (int = 10);
   Array (const Array &);
   T& operator[](int i);
   const T& operator[](int i) const;
   Array& operator= (const Array &);
   int getsize() const { return _size; };
   ~Array();
private:
   T* _data;
   int _size;
};

template<typename T>
Array<T>::Array (int size) : _size(size) {
  _data = new T[size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
}

template<typename T>
Array<T>::Array (const Array<T> &a) : _size (a._size) {
  _data = new T[a._size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
  memcpy(_data,a._data,_size*sizeof(T));
}

template<typename T>
Array<T>::~Array () {
  if (_data) {
    delete [] _data;
    _data = NULL;
    _size = 0;
  }
}
```

This is a blank page.

```
template<typename T>
T& Array<T>::operator[](int i) {
  static T tmp = 'Z';
  if ((i < 0) || (i >= _size)) {
    cerr << "array index " << i << " out of bound - value UN trustworthy" << endl;
    return tmp; // return a dummy reference
  } else
    return _data[i];
}

template<typename T>
const T& Array<T>::operator[](int i) const {
  static T tmp = 'Z';
  if ((i < 0) || (i >= _size)) {
    cerr << "array index " << i << " out of bound - value UN trustworthy" << endl;
    return tmp; // return a dummy value
  } else
    return _data[i];
}

template<typename T>
Array<T>& Array<T>::operator= (const Array &a) {
  _size = a._size;
  if (_data) delete [] _data;    // free the old memory if it exists
  _data = new T[a._size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
  memcpy(_data,a._data,_size*sizeof(T));
  return *this;
}

#endif
```

2 Static Class

Define a small class called `Doubleton` that cannot be instantiated more than twice. That is, during run-time, a program that uses the `Doubleton` class can have either zero or one or two instances of `Doubleton`. If you try to create a third instance, you would get NULL.

The content of the class is irrelevant for the problem. The key is that there could be no way to create more than on instance of the class.

Requirements (Hints): The solution involves three things 1) Make the default constructor private; 2) declare a friend function `NewDoubleton` that is used to create the instance of the class, or return NULL if you are attempting to create a third instance; 3) declare something `static` to keep track of the created instances itself. The sample usage and output is as shown below:

```
int main()
{
  Doubleton* array[10];
  for (int i = 0; i < 10; i++) {
     array[i] = NewDoubleton();
     if (!array[i]) cout << i << " " ;
  }
  return 0;
}
The output is: 2 3 4 5 6 7 8 9
```

Key:

```
class Doubleton {
private:
  static Doubleton* instance[2];
  Doubleton() {}       // must have constructor private so no (default)
                       // public constructor function to construct instance
public:
  friend Doubleton* NewDoubleton();
};


Doubleton* Doubleton::instance[2];

Doubleton* NewDoubleton() {
  for (int i=0;i<2;i++) {
    if (Doubleton::instance[i] == NULL) {
      Doubleton::instance[i] = new Doubleton();
      return Doubleton::instance[i];
    }
  }
  return NULL;
}
```

3 Functors and Function Pointers

Functors are functions with a state. In C++ you can realize them as a class with one or more private members to store the state and with an overloaded operator () to execute the function. Functors can encapsulate C and C++ function pointers employing the concepts templates and polymorphism. You can build up a list of pointers to member functions of arbitrary classes and call them all through the same interface without bothering about their class or the need of a pointer to an instance. All the functions just have got to have the same return-type and calling parameters.
In the following, we define an abstract base class `TFunctor` which provides a virtual function named `Call` or a virtually overloaded `operator ()` with which you will be able to call the member function. From the base class you derive a template class `TSpecificFunctor` which is initialized with a pointer to an object and a pointer to a member function in its constructor. The derived class overrides the function `Call` and/or the `operator ()` of the base class: In the overridden version you call the member function using the stored pointers to the object and to the member function.
Your task here is to complete the four subtasks specified in the `main` function. The output of running the program is

```
TClassA::Display called!
TClassB::Display called!

Hit Enter to terminate!
```

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

//-----------------------------------------------------------------------------------------

   // How to Implement Functors
   // abstract base class
   class TFunctor
   {
   public:

      // two possible functions to call member function. virtual cause derived
      // classes will use a pointer to an object and a pointer to a member function
      // to make the function call
      virtual void operator()(const char* string)=0;  // call using operator
      virtual void Call(const char* string)=0;        // call using function
   };


   // derived template class
   template <class TClass> class TSpecificFunctor : public TFunctor
   {
   private:
      void (TClass::*fpt)(const char*);   // pointer to member function
      TClass* pt2Object;                  // pointer to object

   public:

      // constructor - takes pointer to an object and pointer to a member and stores
      // them in two private variables
      TSpecificFunctor(TClass* _pt2Object, void(TClass::*_fpt)(const char*))
         { pt2Object = _pt2Object;  fpt=_fpt; };

      // override operator "()"
      virtual void operator()(const char* string)
```

```
    { (*pt2Object.*fpt)(string);};              // execute member function

   // override function "Call"
   virtual void Call(const char* string)
     { (*pt2Object.*fpt)(string);};             // execute member function
};


//-----------------------------------------------------------------------------------------
// Example of How to Use Functors

// dummy class A
class TClassA{
public:

   TClassA(){};
   void Display(const char* text) { cout << text << endl; };

   /* more of TClassA */
};

// dummy class B
class TClassB{
public:

   TClassB(){};
   void Display(const char* text) { cout << text << endl; };

   /* more of TClassB */
};

// main program
int main(int /*argc*/, char* /*argv[]*/)
{
   // Subtask 1. instantiate objects of TClassA and TClassB
   TClassA objA;
   TClassB objB;


   // Subtask 2. instantiate TSpecificFunctor objects ...
   //    a ) functor which encapsulates pointer to object and to member of TClassA
   TSpecificFunctor<TClassA> specFuncA(&objA, &TClassA::Display);

   //    b) functor which encapsulates pointer to object and to member of TClassB
   TSpecificFunctor<TClassB> specFuncB(&objB, &TClassB::Display);


   // Subtask 3. make array with pointers to TFunctor, the base class, and initialize it
   TFunctor* vTable[] = { &specFuncA, &specFuncB };


   // Subtask 4. use array to call member functions without the need of an object
   vTable[0]->Call("TClassA::Display called!");        // via function "Call"
   (*vTable[1])   ("TClassB::Display called!");        // via operator "()"


   // hit enter to terminate
   cout << endl << "Hit Enter to terminate!" << endl;
   cin.get();
   return 0;
}
```

4 Comlexity

Suppose Module A requies M units of time to be executed, where M is a constant. find the time units $T(n)$ and complexity $C(n)$ of each algorithm, where n is the size of the input data and b is a positive integer greater than 1. Observe in the following algorithms we use N for n and B for b.

(a) **Algorithm A**:

```
1. Repeat for I = 1 to N:
2.   Repeat for J = 1 to N:
3.     Repeat for K = 1 to N:
4.       Module A.
       [End of Step 3 loop.]
     [End of Step 2 loop.]
   [End of Step 1 loop.]
5. Exit.
```

(b) **Algorithm B**:

```
1. Set J = 1.
2. Repeat Steps 3 and 4 while J <= N:
3.   Module A.
4.   Set J = B x J.
   [End of Step 2 loop.]
5. Exit.
```

Key:

(a) Here

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} M = Mn^3$$

The number of times M occurs is the sum equal to the number of triplets (i, j, k), where i, j, k are integers from 1 to n inclusive. There are n^3 such triplets. Hence

$$C(n) = O(n^3)$$

(b) Observe that the values of the loop index J are the powers of b:

$$1, b, b^2, b^3, \cdots$$

Therefore, Module A will be repeated exactly K times, where K is teh first exponent such that

$$\begin{aligned} b^K &> n \\ K &= \lfloor \log_b n \rfloor + 1 \\ T(n) &= MK \\ C(n) &= O(\log_b n) \end{aligned}$$

5 Sorting

The following procedure implements quicksort:

```
Quicksort(A,p,r)
1. if p < r
2.    then q <- Partition(A,p,r)
3.         Quicksort(A,p,q)
4.         Quicksort(A,q+1,r)
```

To sort an entire array A, the initial call is `Quicksort(A,1,length[A])`. The following procedure implements `Partition`:

```
Partition(A,p,r)
 1.  x <- A[p]
 2.  i <- p-1
 3.  j <- r+1
 4.  while true
 5.      do repeat j <- j - 1
 6.           until A[j] <= x
 7.         repeat i <- i + 1
 8.           until A[i] >= x
 9.         if i < j
10.            then exchange A[i] <-> A[j]
11.            else return j
```

Suppose the input array, with the initial values of i and j just off the left and right ends of the array, is

```
   5, 3, 2, 6, 4, 1, 3, 7
 i                         j
```

Consider the first call to `Partition` in line 2 of `Quicksort` before the subsequent recursive calls. Show the array and the positions of i and j at line 9 of the third and last iteration of the **while** loop of `Partition`. What is the return value of `Partition`?

Key:
The return value is 5. The array and the positions of i and j are shown below:

```
  3, 3, 2, 1, 4, 6, 5, 7
      return  j  i
```

6 Heap

Show all the steps to build a heap from the following list of numbers by inserting the eight numbers one after the other into an initially emtpy heap:

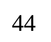
$$44, 30, 50, 22, 60, 55, 77, 55$$

The root of the final heap is 77. Draw the intermediate heap after each insertion. The first two steps are done for you.

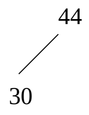
```
          44                        44
                                   /
                                  /
                                30

    (a) ITEM = 44            (b) ITEM = 30
```

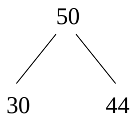
key:



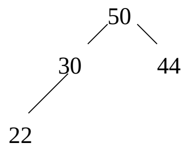
(a) ITEM=44



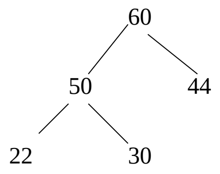
(b) ITEM=30



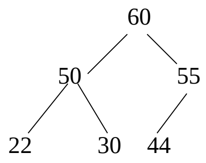
(c) ITEM=50



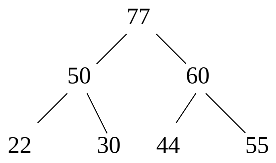
(d) ITEM=22.



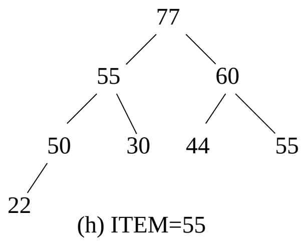
(e) ITEM=60.



(f) ITEM=55



(g) ITEM=77



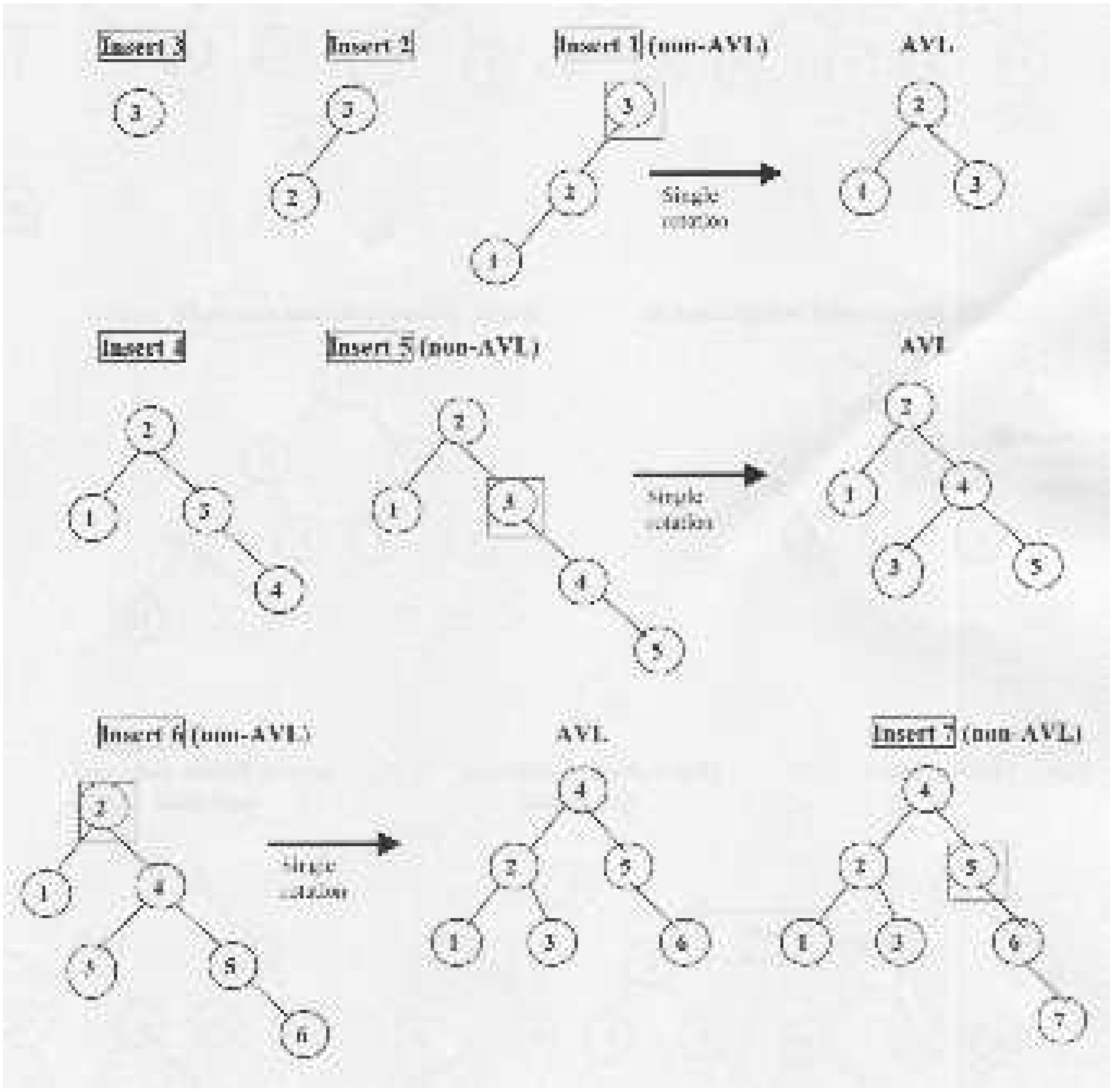
(h) ITEM=55

7 AVL and Binary Trees

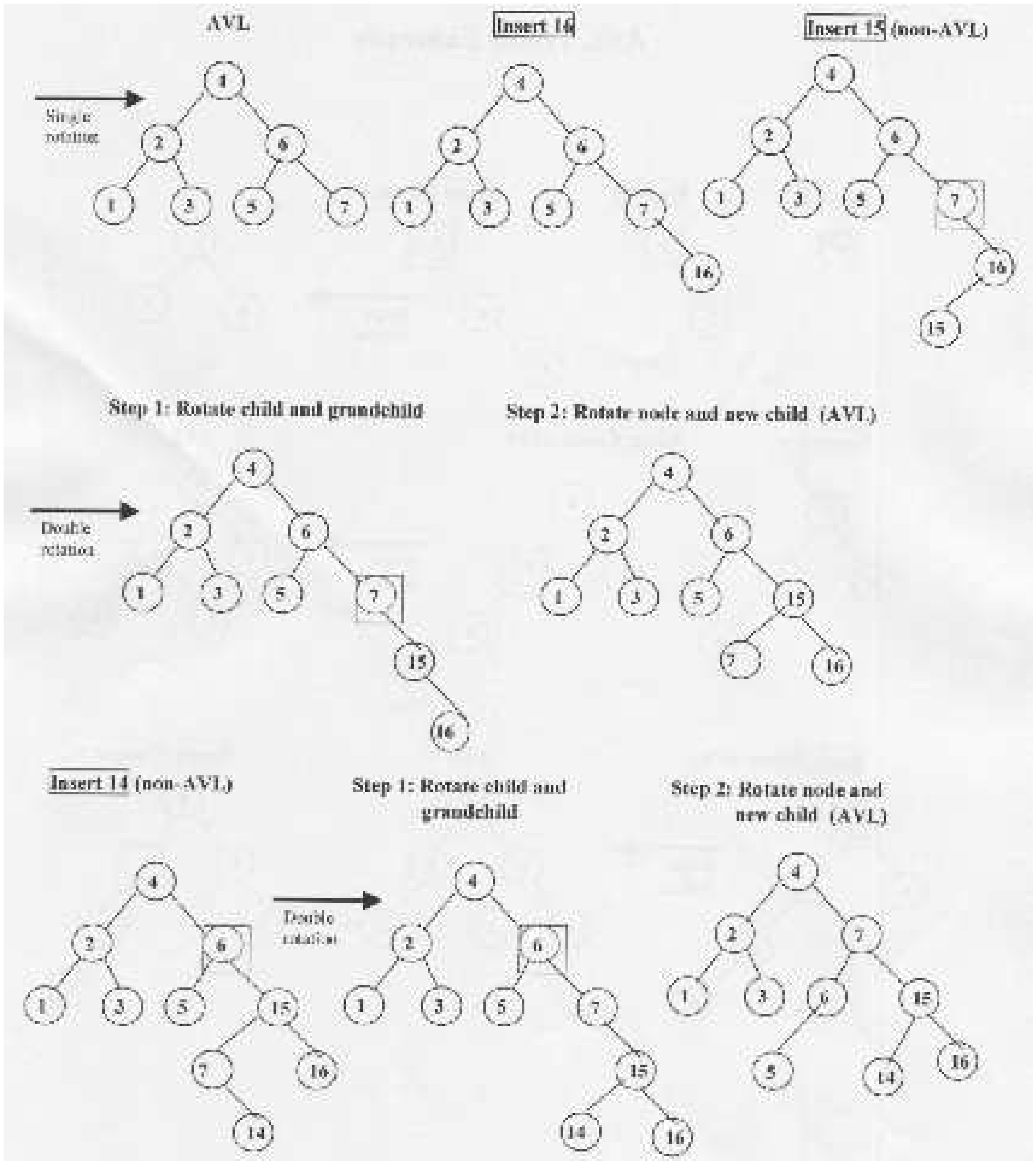
AVL Tree. Create an AVL tree by inserting the following list numbers. Show the tree after each insertion. Perform tree rotation to main the AVL property.

$$3, 2, 1, 4, 5, 6, 7, 16, 15, 14$$

Key:



This is an emtpy page.


AVL
Insert 16
Insert 15 (non-AVL)
Single rotation
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
Insert 14 (non-AVL)
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation

Binary Tree. A binary Tree T has 9 nodes. The inorder and preorder traversal of T yield the following sequences of nodes:

```
 Inorder:  E A C K F H D B G
Preorder:  F A E K C D H G B
```

(a) What is the root of T?

(b) What are nodes that consists of the left subtree T_1 of T? What is the left child of r?

(c) What are nodes that consists of the right subtree T_2 of T? What is the right child of r?

(d) If you know the answers to the above, you are now ready to draw the tree T. Draw T.

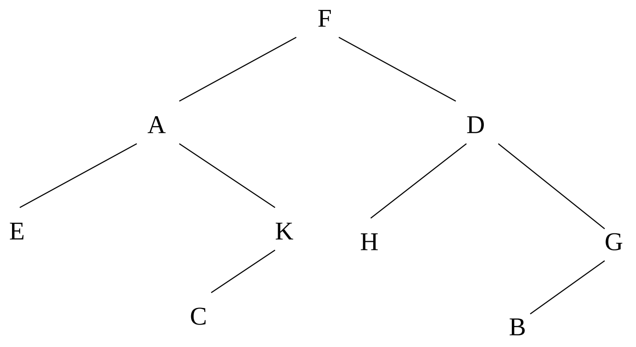
Key:

(a) The root of T is obtained by choosing the first node in its preorder. Thus **F** is the root of T.

(b) T_1 consists of the nodes $\mathbf{E}, \mathbf{A}, \mathbf{C}$ and **K** The left child of **F** is obtained by choosing the first node in the preorder of T_1 (which appears in the preorder of T. Thus **A** is the left child of **F**.

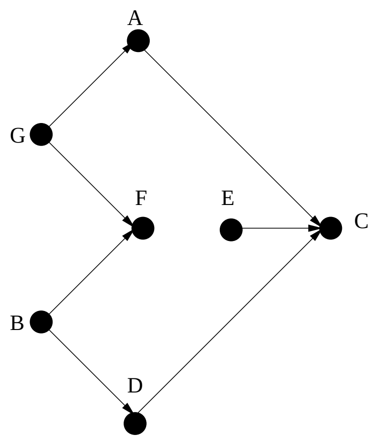
(c) Similarly, the right subtree T_2 of **F** consists of the nodes $\mathbf{H}, \mathbf{D}, \mathbf{B}$ and **G**, that is, **D** is the right child of **F**.

(d)



8 Directed Graph

Assume the following directed graph and its adjacency lists representation.



Adjacency Lists

A: C
B: D, F
C:
D: C
E: C
F:
G: A,F

Find the topological sort using the algorithm taught in class.

Key:
One possible solution is: B, E, G, D, A, F, C

9 Difficult Problem

Prove the following proposition: *Let A be the adjacency matrix of a graph G. Then $a_k(i,j)$, the ij entry in the matrix $A^K = \underbrace{A \cdot A \cdots A}_{K}$, gives the number of paths of length K from node v_i to node v_j.*

This is a challenging problem so we will guide you step-by-step to prove the proposition. The proof is by induction.

(a) (basis case) Prove the proposition for $K = 1$, that is, path length of 1 from v_i to v_j.

(b) (inductive case)

- (i) Suppose $K > 1$. Express A^K in terms of A^{K-1} and A.
- (ii) Assume G has m nodes. Express each entry $a_k(i,j)$ in terms of $a_{k-1}(\cdot)$ and $a_1(\cdot)$.
- (iii) Based on (i) and (ii), prove the inductive case.

(a) (basis case) Note first that a path of length 1 from v_1 to v_j is precisely an edge $(v_i, v_j$. By definition of the adjaceny matrix A, $a_1(i,j = a_{ij}$ gives the number of edges from v_i to v_j. Hence the proposition is true for $K = 1$.

(b) (inductive case) Suppose $K > 1$. Assume G has m nodes.

- (i) $A^K = A^{K-1}A$.
- (ii) $a_K(i,j) = \sum_{s=1}^{m} a_{K-1}(i,s)a_1(s,j)$.
- (iii) By induction, $a_{K-1}(i,s)$ gives the number of paths of length $K-1$ from v_i to v_s, and $a_1(s,j)$ gives the number of paths of length 1 from v_s to v_j. Thus, $a_{K-1}(i,s)a_1(s,j)$ gives the number of paths of length K from v_i to v_j where v_s is the next-to-last node. Thus all the paths of length K from v_i to v_j can be obtained by summing up the $a_{K-1}(i,s)a_1(s,j)$ for all s. That is, $a_K(i,j)$ is the number of paths of length K from v_i to v_j. Thus the proposition is proved.