Queen Mary University of London

ADMIN USE ONLY

Question	Mark
Q1	
Q2	
Q3	
Q4	
FINAL	

EECS ANSWER SHEET

Summer 2020 Examination Period

ENTER YOUR Student number:

190698348

Module code: ECS769P

Module name: Advanced Object-Orientated Programming

Name and type of calculator used (if applicable):

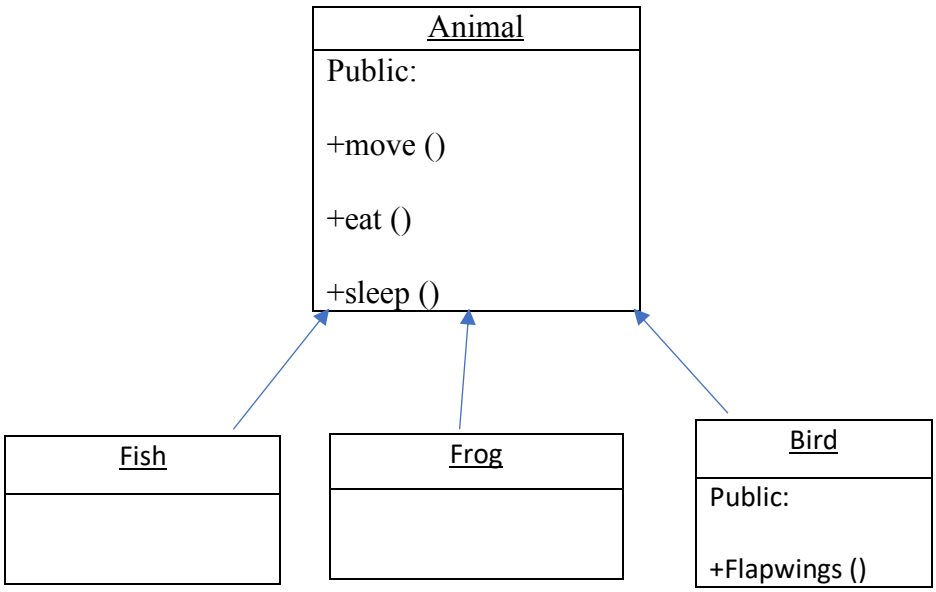
CASIO Scientific calculator

Final Alternative Exam (FAE) Instructions

- Ensure you have all the resources you require to complete and upload the final completed assessment.
- Follow the instructions from the question paper. If you answer more questions than specified in the rubric, only the first answers will be marked, up to the number of answers required. Ensure that you delete anything that you do not wish to be marked before submission.
- Use this Answer Sheet for entering your answers.
- Start on a new page for each question. Ensure you make clear which Question number you are on. Please save your work frequently, so that you do not risk losing it.
- You will have 48 hours from the start of the schedule time to submit, do not leave submissions too close to the deadline. No late submission will be accepted, no exceptions.
- **Do not collaborate.** Your submission must be your own work, and you must ensure that you do not break any of the rules in the Academic Misconduct Policy. Please be aware that submissions will be subject to review, including but not limited to plagiarism detection software.
- When finished save the file as pdf before uploading, only pdf will be accepted, any other file format will not be accepted.
- Any problems during the submission period relating to access to the paper or submission, you may contact the email given in the submission guidelines.
- Ensure you have read the FAQs and guidelines before asking questions.
- **Add your answer to the questions starting from the next page.**

Question 1:

a)



b)

Polymorphic programming allows you to make the base class an abstract class. It gives the user the choice to create virtual functions (functions which have no implementation and have to create within the child classes).
Polymorphic programming (polymorphism) will be an effective solution for this problem as you can define all the virtual functions for creating the different aircrafts and objects in the base class (Flight simulator) and leave the implementation for the graphical outputs to the child classes. Hence, when you create the base class pointer in your main function you can simultaneously call all the overridden virtual functions from the child classes.

c)

Header Files

```
Class DegreeCalculation
{
    Private:
     int Student_ID;
    string student_name;
    Public:
    Virtual void calculate () =0;
};

Class PostGraDegreeCalculation: Public DegreeCalculation
{
   Private:
   Bool isFullTime;
   Public:
    Virtual void calculate () override;
};
```

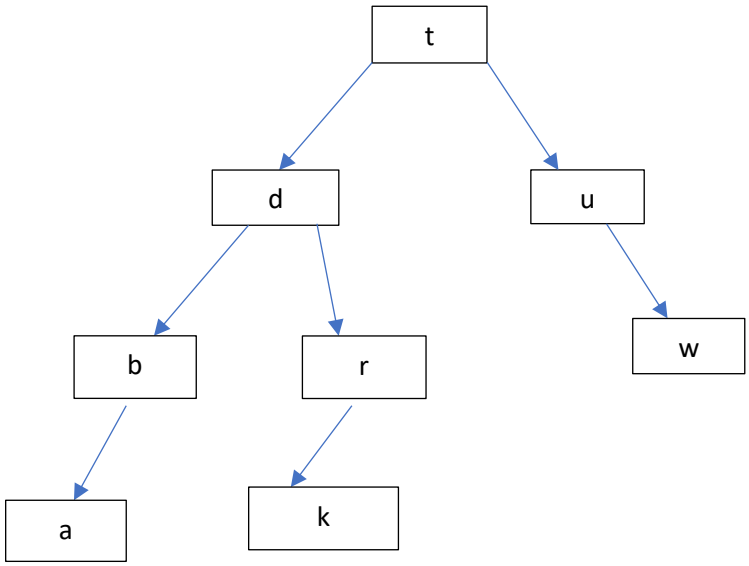
Question 2:

a)

```
Template<typename T>
Bool IsEqualTo(T a, T b)
{
    If(a==b)
    {
        return true;
    }
        else
        {
            return false
        }
}
```

b)

i)



ii) The output for in-order traversal is the following:
a,b,d,k,r,t,u,w.

iii) The output for preorder traversal is the following:
t,d,b,a,r,k,u,w.

iv) The output for postorder traversal is the following:
a,b,k,r,d,w,u,t.

v) Here is the code for theTestchar.cpp

```
#include <iostream>
#include "Tree.cpp"

void Test_char ()
{
    std::cout<<"start"<<std::endl;
    Tree<char> char_Tree;
    {
       std::cout<<"Enter 8 character values: \n";
    }
    char char_val;
    //insert 10 integers into the tree
    for (int i=0;i<8;++i)
    {

       std::cin>>char_val
       char_Tree.insertNode(char_val);
    }

    std::cout<<"\nPreorder traversal \n";
    char_Tree.preOrderTraversal();

    std::cout<<"\nInorder traversal \n";
    char_Tree.inOrderTraversal();

    std::cout<<"\nPostorder traversal \n";
    char_Tree.postOrderTraversal();

    std::cout<<std::endl;
}
```

Question 3:

a) Here is the code that stores the contact information:

```
#include <iostream>
#include <map>
#include <string>

int main ()
{
   std::multimap<std::string,int> phone_numbers;
   phone_numbers.insert({"joe",1234});
   phone_numbers.insert({"flo",2345});
   phone_numbers.insert({"joseph",1234});
   phone_numbers.insert({"mo",3456});
}
```

b) The eight associative containers are the following:

- set
- multiset
- unordered set
- unordered multiset
- Map
- Multimap
- unordered map
- unordered multimap

Set- A set would not be a sensible choice for storing the data in part(a) as a set does not allow duplicate values. The data set above has duplicate phone numbers.

Multiset- A multiset would be a sensible choice for storing the data in part(a) as multiple elements can have equivalent values. The data set above has equivalent phone numbers therefore both will be stored. Multiset also orders the data as it inserts it within the container.

Unordered set- An unordered set would not be a sensible choice for storing the data in part(a) as it has stores unique therefore no duplicate values are allowed. The data set above has duplicate phone numbers which will be lost. Also, unordered set doesn't order the data.

Unordered multiset- An unordered multiset would be a sensible choice for storing the data in part(a) as it does not support unique values therefore you can store multiple equivalent values. This supports the data set above as duplicate values will not be lost. However, an unordered multiset doesn't order the values.

Map – A map will be a sensible choice for storing the data in part(a) as it allows duplicate values to be entered. However, if one were to use duplicate keys a map will not be a good choice as duplicate keys aren't allowed. A map also sorts the keys in – order.

Multimap- A multimap will be a sensible choice for storing the data in part(a) as it allows both duplicate values and duplicate keys. A multimap also sorts the keys in -order.

Unordered-map- An un ordered map will be a sensible choice for storing the data in part(a) as it allows duplicate values. However, if one were to use duplicate keys an unordered map would not be a good idea as duplicate keys are not allowed. An unordered map also doesn't sort the keys.

Unordered- multimap- An unordered multimap will be a sensible choice for storing the data in part(a) as it allows both duplicate values and keys. However, a unordered multimap doesn't sort the keys.

c) Here is the code for removing duplicate phone numbers and writing the names and numbers to the standard output:

```
#include <iostream>
#include <map>
#include <string>
#include <set>

void Lower_case (std::string &s)
{
   std::string::iterator it=s.begin();
   {
       *it=tolower(*it);
   }
   for(;it!=s.end();it++)
   {
      if(*it=='!')
      {
         break;
      }
   else if(*it==' ')
      {
         *it++;
         *it=tolower(*it);
         *it--;

      }
   }
}

int main ()
{
   std::string s="joe";
   std::string s1="flo";
   std::string s2="joseph";
   std::string s3="mo";
   Lower_case(s);
   Lower_case(s1);
   Lower_case(s2);
   Lower_case(s3);
```

```
    std::set<int> values;
    std::map<std::string,int> phone_numbers;
    phone_numbers.insert({s,1234});
    phone_numbers.insert({s1,2345});
    phone_numbers.insert({s2,1234});
    phone_numbers.insert({s3,3456});

    for (std::map<std::string,int>::iterator
it=phone_numbers.begin();it!=phone_numbers.end();it++)
    {
        if(values.insert(it->second).second)
        std::cout<<it->first<<" "<<it->second<<std::endl;
    }

}
```

(d)

First Objection- STL containers have a lot of complexity so debugging them is quite difficult.
Counter argument – Ask peers for help and use stack overflow to help solve your issue with the STL. The STL is very well documented.

Second Objection – STL can lead to slower compile times, especially if you have an old compiler.
Counter argument-Use the latest compiler and get a good PC with a good CPU and GPU.

Third Objection – STL have very bad exception handling which makes it difficult to understand what is happening in the code.
Counter argument – Create your own exception handling code.

Question 4:

a)

```
#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

int main ()

{

   vector<int> seq= {1,2,3,4,5,6,7};

   for (vector<int>::iterator it=seq.begin();it!=seq.end();it++)

{

      *it=pow(*it,2);

      cout<<*it<<' '

   }

}
```

(b)

```
int deduce(int balance,vector<int> items)
{
  sort(items.begin(), items.end());
  int total_item_amount=std::accumulate(items.begin(),items.end(),balance,minus<int>)
  return new_balance;
}
```

(c)

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

class my_accumulate
{
private:
   vector<int> item;
   int thresholds;
public:
   my_accumulate(vector<int> items,int threshold): item(items),thresholds(threshold){};
   int operator() (vector<int> &item,int threshold)
   {
      int total=0;
      int count=0;
      for (vector<int>::iterator it=item.begin();it!=item.end();it++)
      {
         if(*it<threshold)
         {
            total=total+*it;
         }
         else if(*it>threshold)
         {
            cout<<"Error you have "<<count++<<" items above the given threshold"<<endl;
         }
      }
      return total;
   }

};


int deduce(int balance,vector<int> items)
{
   int threshold =500;
   my_accumulate acc(items,threshold);
   int new_balance=balance-acc(items,threshold);
   return new_balance;
}
```

d)

A set is suitable for storing the profile of current players because a set only allows unique keys. This will be useful for our current situation the player base will be unique for each individual player.

Furthermore, the insertion of a set is constant time therefore when players log into the game it will take less time. Also, lookup is in logarithmic time which is better than linear time.

In conclusion, for our problem a set is a good data structure because it allows insertion at constant time while simultaneously allowing lookup at logarithmic time. The combination of these two features make std::set a perfect choice.