CS193D
Autumn 2000

Handout 06
October 5 , 2000

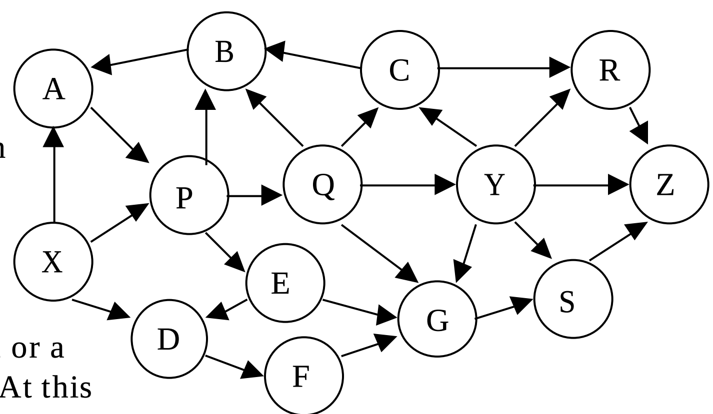
Section Handout 2

This handout was written by Derek Poppink and Hiroshi Ishii.

Depth-First Search (DFS)

Before we talk about DFS, we should define a graph, which is the data structure that DFS works with. A *graph* consists of a set of nodes N, and a set of edges E where each edge connects two nodes. For example, cities in California would be the nodes and the highways that connect them would be the edges. Extending the analogy of roads, sometimes streets are only one-way, i.e., you can go from point A to point B, but cannot go from B to A. So, a *directed* graph is one in which the edges only go one-way. (This definition is just an intuitive one. I'll spare y'all the mathematical definition 'coz this ain't an algorithm class.) So, for HW1, the nodes are the `Airport` objects, and the edges are the `Flight` objects. And obviously, the graph is directed, as `Flights` only go one-way.

The DFS is just a way of exploring the nodes in a graph by following edges from a root node. Upon visiting a node P, DFS will follow an outgoing edge from P and visit the next node Q. We call Q a descendent of P. The same happens at Q, until it reaches a node Z that has no outgoing edges, or a node that has been visited already. At this point, DFS backtracks up one node to Y. If Y has another outgoing edge, DFS will follow that one.



For example, for the graph above, the sequence of nodes that are visited for a DFS starting at node X might be X → P → Q → Y → Z back to Y → S back to Y → R back to Y → G back to Y → C → B → A back to B back to C back to Y back to Q back to P → E → D → F. There are other possible depth-first traversals starting from node X because the outgoing edges from a node can be followed in any order.

DFS can be implemented very nicely as a recursive procedure. The following is a simple DFS algorithm in pseudocode.

```
DFS(Node P, NodeSet visited)
{
   if (P NotIn visited) {
      AddNode(P, visited);
      foreach edge of OutgoingEdges(P) {
         Node Q = EndNode(edge);
         DFS(Q, visited);
      }
   }
}
```

You can adapt the DFS algorithm above for your trip search problem in HW#1. Start a DFS at the origin airport, and every time you reach the destination airport, print out the sequence of flights that lead to the destination airport. To do this, you need to keep track of the current partial path during the traversal. You will need to pass this into the DFS procedure on each recursive call.

You will also need to reject flights that are too close together or too far apart. For this, you will need to look at the arrival time of the incoming flight at an airport, and decide which outgoing flights are within the required time interval.

And finally, you must reject paths that have more than 4 stops. This should be straightforward given that you keep track of the partial path during the traversal.

Const and References

You `const` and references early & often. In particular, use `const` whenever you think something is constant, and use references instead of pointers whenever you can. It's important to use `const` from the beginning, and not try to first code without `consts` and then add them later. Adding `consts` afterwards will be very difficult because every `const` you add will result in five compilation errors for the places you need them also.

To illustrate these ideas, here is the perennial students-in-a-class example. A class is composed of a list of students, each of which contains a name and a grade:

```
struct StudentRecord
{
  char * name;
  int id;
  int grade;
};

struct ClassRecord
{
  StudentRecord **students;  // stores pointers to StudentRecords
  int numStudents;           // number of students in the class
  int numAllocated;          // number of pointers in the array students
};
```

For an operation such as printing, comparison, and search which does not change the contents of the class record, you pass the class record as a constant reference:

```
void print(const StudentRecord & student)
{
  // print out name and grade
}

bool match(const StudentRecord& s1, const StudentRecord& s2)
{
  return strcmp(s1.name, s2.name) == 0 && s1.id == s2.id;
}

void print(const ClassRecord& roster)
{
   for(int i=0; i<roster.numStudents; ++i)
      print(*(roster.students[i]));
}
```

Note the function overloading between the two print functions. For `findStudentByName`, the search key name is passed also as const, since the function will not modify it.

```
StudentRecord* findStudentByName(const ClassRecord& roster,
                                 const char* name)
{
   for (int i = 0; i < roster.numStudents; i++)
      if (strcmp(roster.students[i]->name, name) == 0)
         return roster.students[i];

   return NULL;
}
```

In the function above we are returning a pointer to a `StudentRecord` instead of a reference to it. This is because in case the record isn’t found, we need to return a value indicating failure. On other other hand, if we are positive that the specified record exists, we can just return a reference to `StudentRecord`.

```
StudentRecord& getStudent(const ClassRecord& roster, int i)
{
   return *roster.students[i];
}
```

In `findStudent`, both the `roster` and `student` are taken as `const` parameters:

```
int findStudent(const ClassRecord& roster, const StudentRecord& student)
{
   for(int i=0; i<roster.numStudents; ++i)
      if(match(*(roster.students[i]), student))
         return i;

   return -1;
}
```

On the other hand, if you need to modify the database, you should pass it without `const`:

```
void removeAllStudents(ClassRecord& roster)
{
   for (int i=0; i<roster.numStudents; ++i)
      delete roster.students[i];

   roster.numStudents = 0;
   delete [] roster.students;
   roster.students = NULL;
   roster.numAllocated = 0;
}
```

In the following function, you need to pass the student as non-`const` even though the function does not modify it directly. This is because, on the line marked with an asterisk, the address of the student record is assigned to an element in the student array, which is of type non-`const` pointer to student. Note that applying the address-of (`&`) operator to a reference variable results in the address of the object that the reference refers to, not the address of the reference itself. In fact, in C++, there is no way to calculate the address of a reference. Correspondingly, there is no way to declare a variable of type "pointer to reference".

```
void addStudent(ClassRecord & roster, StudentRecord & student)
{
   if (roster.numStudents == roster.numAllocated) {
      // grow students array
   }

   roster.students[roster.numStudents] = &student;
   roster.numStudents++;
}
```

Note that the `const`ness of a structure refers only to it's direct members. The objects pointed to (or referred to) by members of the structure are not automatically `const`. In the following function, `roster` is a reference to a `const` class record. Therefore, the fields `numStudents`, `numAllocated`, and the value of the pointer `students` cannot be modified. However, student pointers in the `students` array can be modified at will. Also, since the pointers in the array are non-`const` pointers, the contents of the student objects contained in the array can be modified as well. However, doing so will violate our intuitive understanding of the `const` as not changing the class database. Therefore, the following is legal in C++, although it's in very bad style.

```
void inflateGrades(const ClassRecord & roster, int inflation)
{
   // inflate grades by 10 points each
   for(int i = 0; i < roster.numStudents; i++)
      roster.students[i]->grade =
            min(roster.students[i]->grade + inflation, 100);

   // get "rid" of students with bad grades
   for(int i = 0; i < roster.numStudents; i++) {
      if(roster.students[i]->grade < 70) {
         delete roster.students[i];
         roster.students[i] = NULL;
      }
   }
}
```

Note that everything I've said above about `const` and references applies equally as well to `const` and pointers. You should use references instead of pointers to pass parameters whenever possible, though.

You can also have multiple references to the same object. In the following function, `removeStudentByName` takes a non-`const` `ClassRecord`. It then passes that on to `findStudentByName` as a `const` `ClassRecord`. When this is done, both `roster` in the `removeStudentByName` function and `roster` in the `findStudentByName` function are the referring to the same `ClassRecord`. It's not the case that the `roster` in `findStudentByName` is a reference to the `roster` reference in the `removeStudentByName` function. There's no such thing as a reference to a reference.

```
void removeStudentByName(ClassRecord & roster, const char * name)
{
   StudentRecord * student = findStudentByName(roster, name);
   if (student == NULL) return;

   int i = findStudent(roster, *student);
   assert(i != -1);
   assert(student == roster.students[i]);
   delete student;
   // fill empty student spot with last student element
   roster.numStudents--;
   roster.students[i] = roster.students[roster.numStudents];
}
```

Note that since `removeStudentByName` takes roster as non-`const`, it's ok to pass this on to `findStudentByName`, which takes it as `const`. The reverse of this situation will be illegal. That is, if `removeStudentByName` took `roster` as `const`, and `findStudentByName` took roster as non-`const`, you will not be able to pass the `const` roster in `removeStudentByName` to the non-`const` roster in `findStudentByName`.

Using the `SymbolTable` class

The `SymbolTable` class is very useful in storing objects with a key. The key is sort of like the index to an array, but the key does not have to be a number in a predescribed range.

In our case, the key is a string, either the `Airport's` 3-letter code or its long name. The `SymbolTable` also does not limit how many elements it can store. Because the `SymbolTable` is very generic, it doesn't store pointers of any particular type, but instead untyped pointers, or (`void *`). It is up to the programmer to convert the generic pointer that come back from the `SymbolTable` to be `Airport *`, and vice-versa. We use casting to do this, as in the two following examples:

a. To store an `Airport` in a `SymbolTable`, you can simply insert without a cast, since `void*` is all accepting..

```
SymbolTable symtab;
Airport *airport = new Airpoirt("SFO", "San Francisco, CA");
symtab.enter(airport->code(), airport);
```

b. To look up, you need to supply the key. The return value of `lookup(key)` is a generic pointer, so you'll have to cast it back to an `Airport` pointer:

```
char *key = "LAX";
Airport *airport;
airport = (Airport *) symtab.lookup(key);
```

Using the `FlightList` class

The `FlightList` class is useful for storing an unbounded collection of flights. To be precise, it stores pointers to `Flight` objects. The `FlightList` class can contain as many pointers to `Flight` objects as possible. It will automatically allocated more memory as necessary. You may find the `FlightList` very useful to store outgoing flights from an `Airport` object, and to keep track of visited airports and connecting flights during the DFS in the form of a `Trip` object.

The `FlightList` actually behaves like a stack. It allows two mutators, `addFlight`, which adds a flight to the end of the list, and `removeLastFlight`, which removes the last flight of the list. So, the flight that was added last will be the first to get removed. The difference between the `FlightList` and a stack that in the `FlightList` you can access any element in it. The accessor `count` returns the number of stored `Flight` objects, and the accessor `flightAt(i)` returns the i^{th} flight pointer in the list.