CS106X
Winter 2017

Chris Gregg

Section Handout #9

This week has continued practice with graph structures, including A* and Dijkstra's algorithms, inheritance and polymorphism, and review for the final exam.

If you would like additional practice before the exam, we recommend you look through past section handouts for problems you didn't complete in section! Many of our section problems are modeled after past exam problems.

For the problems on this handout, assume the following structures have been declared:

Vertex
```
struct Vertex {
  string name;
  Set<Edge *> edges;
  double cost;
  bool visited;
  Vertex *previous;
};
```

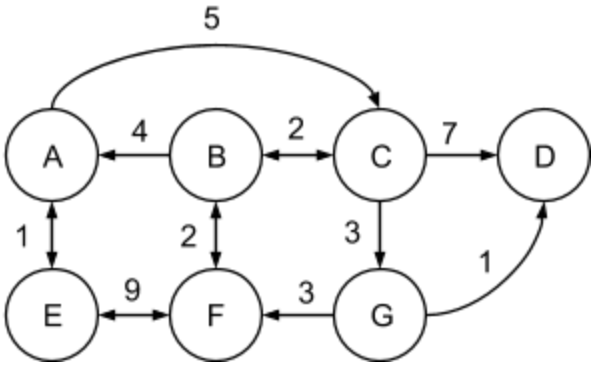
Edge
```
struct Edge {
  Vertex *start;
  Vertex *finish;
  double cost;
  bool visited;
}
```

TreeNode
```
struct TreeNode {
  int data;
  TreeNode *left;
  TreeNode *right;
}
```

ListNode
```
struct ListNode {
  int data;
  ListNode *next;
}
```

1. Dijkstra and A*

Trace through Dijkstra's algorithm on the following graph to find the shortest paths from node A to each other node in the graph. Then use A* to find the shortest path from A to G, where the heuristic is defined such that the distance between two nodes is the distance between those two letters in the alphabet. For example, the distance between B and D according to this heuristic is 2.



2. Good Burger

Consider the following set of class definitions.

Thanks to Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106 instructors for contributing content on this handout.

```
class Lettuce {
public:
  virtual void m1() { cout << "L 1" << endl; m2(); }
  virtual void m2() { cout << "L 2" << endl; }
};

class Bacon : public Lettuce {
  virtual void m1() { Lettuce::m1(); cout << "B 1" << endl; }
  virtual void m3() { cout << "B 3" << endl; }
};

class Hamburger : public Bacon {
  virtual void m2() { cout << "H 2" << endl; Bacon::m2(); }
  virtual void m4() { cout << "H 4" << endl; }
}

class Mayo : public Hamburger {
  virtual void m3() { cout << "M 3" << endl; m1(); }
  virtual void m4() { cout << "M 4" << endl; }
```

Now assume that the following variables are defined:

```
Lettuce *var1 = new Bacon();
Bacon *var2 = new Mayo();
Lettuce *var3 = new Hamburger();
Bacon *var4 = new Hamburger();
Lettuce *var5 = new Lettuce();
```

Figure out the output produced by each of the following statements. Be sure to note which statements wouldn't compile and which would crash at runtime or cause unpredictable behavior.

```
/* a) */                    /* b) */                    /* c) */
var1->m1();                 var1->m2();                 var1->m3();

/* d) */                    /* e) */                    /* f) */
var2->m1();                 var2->m2();                 var2->m3();

/* g) */                    /* h) */                    /* i) */
var2->m4();                 var3->m1();                 var3->m2();

/* j) */                    /* k) */                    /* l) */
var4->m2();                 var4->m3();                 var4->m4();

/* m) */                    /* n) */                    /* o) */
((Bacon *) var1)->m1();     ((Bacon *) var1)->m3();     ((Mayo *) var5)->m3();3

/* p) */                    /* q) */                    /* r) */
((Lettuce *) var4)->m3();   ((Hamburger *) var2)->m4(); ((Mayo *) var2)->m4();
```

3. It's Time To Meet the Muppets

Consider the following set of class definitions.

```
class Kermit {
public:
  virtual void animal() = 0;
  void beaker() { muppet("Kermit::beaker"); animal(); }
  virtual void fozzie() { muppet("Kermit::fozzie"); rowlf(); }
  virtual void misspiggy() = 0;
  void rowlf() { muppet("Kermit::rowlf"); misspiggy(); }
  void muppet(string s) { cout << s << endl; }
}
class Statler : public Kermit {
public:
  void beaker() { muppet("Statler::beaker"); rowlf(); }
  virtual void misspiggy() { muppet("Statler::misspiggy"); rowlf(); }
  void rowlf() { muppet("Statler::rowlf"); animal(); }
};

class Waldorf : public Statler {
public:
  virtual void animal() { muppet("Waldorf::animal"); rowlf(); }
  void rowlf() { muppet("Waldorf::rowlf"); }
};

class Gonzo : public Kermit {
public:
  virtual void animal() { muppet("Gonzo::animal"); rowlf(); }
  virtual void misspiggy() { muppet("Gonzo::misspiggy"); beaker(); }
  void rowlf() { muppet("Gonzo::rowlf"); }
};
```

Now consider the following function:

```
void muppetShow(Kermit *kermit) {
  kermit->fozzie();
}
```

What type of object can `kermit` legitimately address during execution? For each object type, list the output that would be produced by calling `muppetShow` against that type.

4. Append (Linked Lists)
Write a function that accepts two references to pointers to the front of linked lists of integers. After the call is done, all the elements from the second list should be appended to the end of the first one in the same order, and the second list should be empty (null). Note that either linked list could be initially empty.

5. Transferring Evens (Linked Lists)
Write a function that accepts a reference to a pointer to the front of a linked list of integers. It should remove the even-index elements from its linked list and return a pointer to the front of a new list with those elements in the same order.

6. Hacking and Cracking (Recursive Backtracking)
Write a function that takes in the maximum length a site allows for a user's password to break into an account by using recursive backtracking to attempt all possible passwords up to that length (inclusive). Assume you have access to the function `bool login(string password)` that returns true if a password is correct. You can also assume that the passwords are entirely alphabetic and

case-sensitive. You should return the correct password you find, or the empty string, if there isn't one. You should return the empty string if the maximum length passed is 0, or throw an integer exception if the length is negative.

7. ReCuReNCe (Recursive Backtracking)
Write a function that takes in a Lexicon that contains the symbol for all elements in the Periodic Table (e.g. Fe, Li, C) and a word and recursively finds whether or not you can spell that word using only symbols from the periodic table. For example, the word "began" can be spelled using "BeGaN" (beryllium, gallium, and nitrogen). If passed the empty string, you should return true.

8. Limit Leaves (Trees)
Write a function that accepts a reference to the root of a binary tree and an integer *n* and removes nodes from the tree to guarantee that all leaf nodes store values greater than *n*. Do not use any additional data structures.

9. Child Swap (Trees)
Write a function that accepts the root of a binary tree and an integer *k* and and swaps the left and right children of all nodes at level *k* in the tree. For this problem, the root is defined to be at level 1. If *k* is invalid, you should throw an integer exception. Don't traverse any more of the tree than you have to, and don't use any auxiliary data structures. You should not be constructing new nodes, and you shouldn't be modifying the data fields in nodes – swap children entirely by modifying pointers.