Computer Science III (22C:30, 22C:115)
Exam 2, 11/1/02

The exam is worth 150 points (15% of your total grade).

Problem 1. [40 points]
The problem concerns the adjacency list representation of the `graph` class that you have to implement for Project 2.

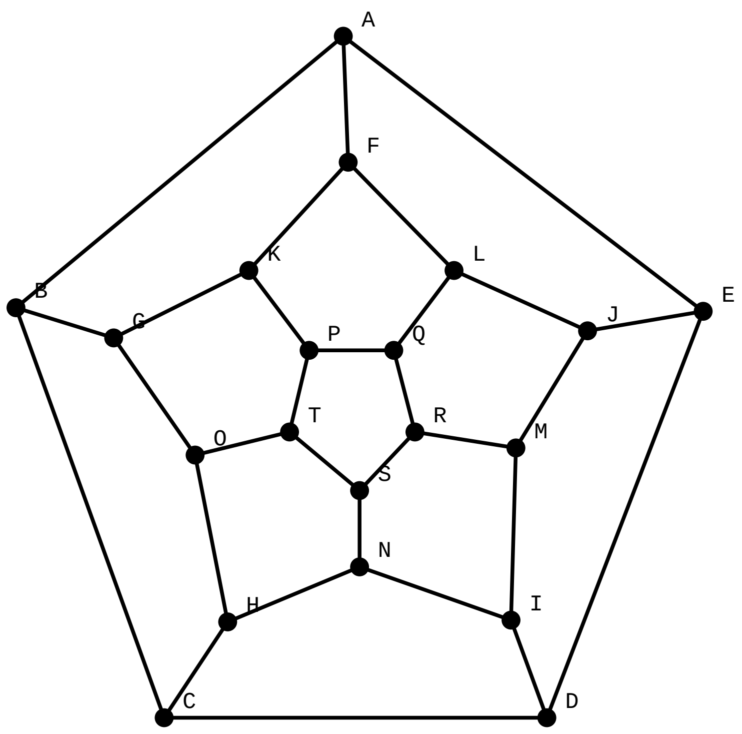
(a) [**15 points**] Write down the data members of the `graph` class for this implementation. Remember that the `graph` class is a template class. It will help to use the definition of the `node` class from your textbook (pages 306–313).

(b) [**25 points**] Assuming the data members from Part (a) of this problem, write a member function of the `graph` class called `NumberOfEdges`, that returns the number of edges in the graph. This function is just 4-5 lines of code if you use the `list_length` function defined in the linked list toolkit (page 309 of your textbook).

Problem 2. [40 points]
Consider the implementation of the `bfs` function discussed in class (a printout of this function is attached to your exam). Now make the following two changes to this function:

1. Replace the definition "`queue<int> Q`" by "`stack<int> Q`".
2. Replace the line "`int current = Q.front();`" by "`int current = Q.top();`"

Show how this modified `bfs` will work for the graph given below.



In particular, assume that node "A" is the source for the search. Show distance-values that are assigned by the function to each of the nodes. These are the values that are computed by `bfs` and stored in the `apvector distances`. Assume that the function `get_neighbors` returns neighbors of a node in alphabetical order.

To answer this question, just mark the distance-values next to the nodes. If you are confused about which label matches up with which node in the above picture, the rule is that each node's label occurs just to its northeast.

Problem 3. [**40 points**]
Here is a recursive function that takes pointers `head1` and `head2` that point to singly linked lists and returns a pointer to a linked list that is somehow obtained by "merging" the lists pointed to by `head1` and `head2`.

```
node<int> * merge(node<int> * head1, node<int> * head2){
        if(head1 == NULL)
                return head2;
        else if (head2 == NULL)
                return head1;
        else{
                if(head1->data() < head2->data()){
                        node<int> * temp = merge(head1->link(), head2);
                        head1->set_link(temp);
                        return head1;
                }
                else{
                        node<int> * temp = merge(head1, head2->link());
                        head2->set_link(temp);
                        return head2;
                }
        }
}
```

(a) [**20 points**] Suppose that we send to the above function two pointers, `head1` and `head2`, that are pointing to lists 3, 7, 11, 30 and 1, 3, 7, 9 respectively. The function returns a node pointer; write down the list of integers that this is pointing to.

(b) [**20 points**] In two sentences explain what the function does.

Problem 4. [30 points]
The graphs we have been talking about in class are *undirected* graphs, because the edges in these graphs have no associated direction. As a result, if there is an edge between a node with index i and a node with index j, then in the adjacency matrix representation, the edge between these two nodes is represented by a `true` value in slot $[i][j]$ as well as in slot $[j][i]$ in the matrix. Suppose we decide that this is somewhat redundant and we will adopt the convention that the only slots in the matrix that we will use are slots $[i][j]$ where $i < j$. This is the same as saying that we will only use the triangle of the matrix that is above and to the right of the diagonal that can be drawn from the top-left corner to the bottom-right corner.

If we adopt this convention, then we will have to change each of the functions in the `graph` class that we implemented for Project 1. Here is the implementation of the `get_neighbors` function from Version 3 of my Project 1 solution, but with a few lines removed. The lines that were removed are the lines of code that would have to be modified as a result of adopting the above convention. Fill in these lines to complete the function.

```
apvector<node> graph::get_neighbors(const node & x){
        // Set up a vector for the list of neighbors
        apvector<node> neighbors;

        // Get the index of the node
        int i = getIndex(x);

        // Check if node exists in the graph
        if(i < 0){
                cout << "Node " << x.getName() << " is not in the graph" << endl;
                return neighbors;
        }

        // Fill in your lines here









        return neighbors;
}
```