CS 106X Sample Final Exam #1
ANSWER KEY

1. Linked Lists (read)

```
front -> [10] -> [20] -> [41] -> [91] -> [71] -> [101] /
```

2. Linked Lists (write)

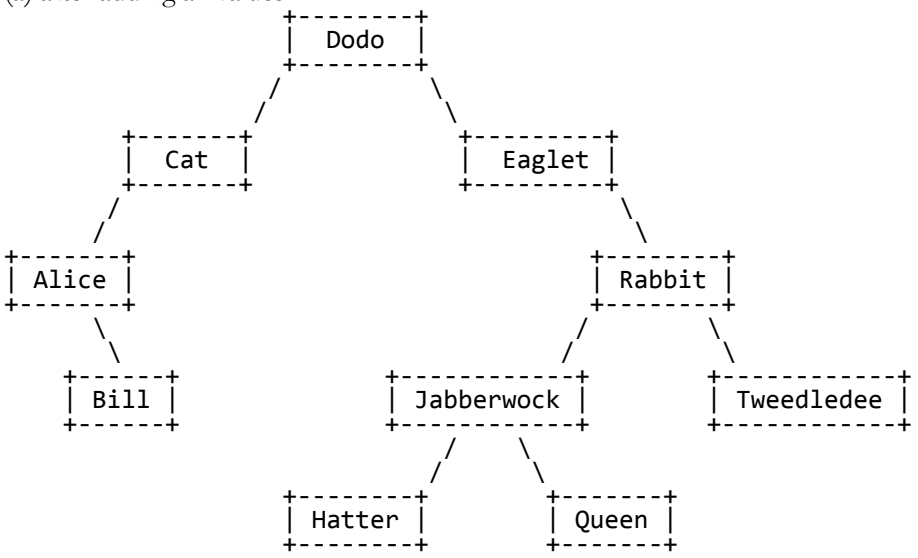
As with any programming problem, there are many correct solutions. Here are two:

```
// solution 1: build entirely new chain, then swap it in
void expand(ListNode*& front, int k) {
    if (k < 0) {
        throw k;
    } else if (front == nullptr) {
        return;
    } else if (k == 0) {
        while (front) {
            ListNode* trash = front;
            front = front->next;
            delete trash;
        }
    } else {
        // expand front node
        ListNode* front2 = new ListNode(front->data / k);
        ListNode* curr2 = front2;
        for (int i = 0; i < k - 1; i++) {
            curr2->next = new ListNode(front->data / k);
            curr2 = curr2->next;
        }

        // expand other nodes
        ListNode* curr = front->next;
        delete front;
        while (curr != nullptr) {
            for (int i = 0; i < k; i++) {
                curr2->next = new ListNode(curr->data / k);
                curr2 = curr2->next;
            }
            ListNode* trash = curr;
            curr = curr->next;
            delete trash;
        }

        front = front2;
    }
}
```

==========

3. Binary Search Trees (read)

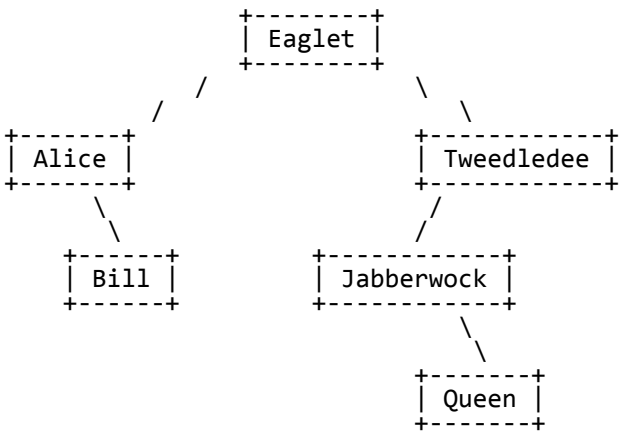
(a) after adding all values:



(b)

Pre: Dodo, Cat, Alice, Bill, Eaglet, Rabbit, Jabberwock, Hatter, Queen, Tweedledee
In: Alice, Bill, Cat, Dodo, Eaglet, Hatter, Jabberwock, Queen, Rabbit, Tweedledee
Post: Bill, Alice, Cat, Hatter, Queen, Jabberwock, Tweedledee, Rabbit, Eaglet, Dodo

(c) after removing Hatter, Cat, Rabbit, Dodo:



4. Binary Trees (write)

As with any programming problem, there are many correct solutions. Here are two:

```
// solution 1: counting up (passing current level)
void swapChildrenAtLevel(TreeNode*& node, int k) {
    if (k <= 0) {
        throw k;
    }
    swapChildrenAtLevelHelper(node, k, 1);
}

void swapChildrenAtLevelHelper(TreeNode* node, int k, int currentLevel) {
    if (node != nullptr) {
        if (currentLevel == k) {
            TreeNode* temp = node->left;
            node->left = node->right;
            node->right = temp;
        } else if (currentLevel < k) {
            swapChildrenAtLevelHelper(node->left,  k, currentLevel + 1);
            swapChildrenAtLevelHelper(node->right, k, currentLevel + 1);
        }
    }
}
```

==========

```
// solution 2: counting down
void swapChildrenAtLevel(TreeNode*& node, int k) {
    if (k <= 0) { throw k; }
    swapChildrenAtLevelHelper(node, k);
}

void swapChildrenAtLevelHelper(TreeNode* node, int k) {
    if (node != nullptr) {
        if (k == 1) {
            TreeNode* temp = node->left;
            node->left = node->right;
            node->right = temp;
        } else if (k > 0) {
            swapChildrenAtLevelHelper(node->left, k - 1);
            swapChildrenAtLevelHelper(node->right, k - 1);
        }
    }
}
```

5. Hashing (read)

```
map.put(8, 11);
map.put(26, 9);
map.put(16, 9);
map.put(26, 88);
map.put(196, 44);
map.put(38, 11);
```

```
   +---+
0  | / |
   +---+
1  | / |
   +---+
2  | / |
   +---+
3  | / |
   +---+
4  | / |
   +---+
5  | / |
   +---+
6  |   |--> 196:44 --> 16:9  --> 26:88
   +---+
7  | / |
   +---+
8  |   |-->  38:11 -->  8:11
   +---+
9  | / |
   +---+
```

(triggers rehash)

```
map.put(32, 77);
map.remove(26);
map.remove(9);
map.put(100, 44);
if (map.containsKey(196)) {
    map.remove(44);
}
map.put(56, 56);
map.put(-48, 34);
```

```
   +---+
0  |   |--> 100:44
   +---+
1  | / |
   +---+
2  | / |
   +---+
3  | / |
   +---+
4  | / |
   +---+
5  | / |
   +---+
6  | / |
   +---+
7  | / |
   +---+
8  |   |--> -48:34 --> 8:11
   +---+
9  | / |
   +---+
10 | / |
   +---+
11 | / |
   +---+
12 |   |--> 32:77
   +---+
13 | / |
   +---+
14 | / |
   +---+
15 | / |
   +---+
16 |   |--> 56:56 --> 16:9 --> 196:44
   +---+
17 | / |
   +---+
18 |   |--> 38:11
   +---+
19 | / |
   +---+

size        =  8                  (FINAL ANSWER)
capacity    =  20
load factor =  0.4
```

6. Graphs (read)

a) directed

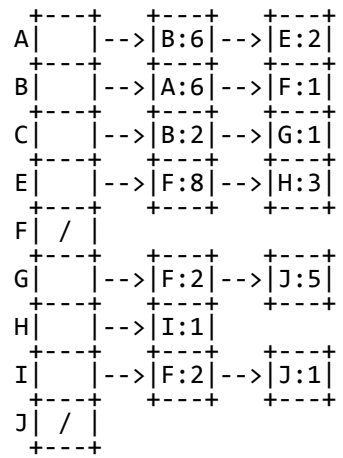
b) weighted

c) unconnected (example: can't get to C from any other vertex; can't get from F or J to any other vertex; etc.)

d) cyclic (example cycle: A -> B -> A)

e) A: in 1, out 2
B: in 2, out 2
C: in 0, out 2
E: in 1, out 2
F: in 4, out 0
G: in 1, out 2
H: in 1, out 1
I: in 1, out 2
J: in 2, out 0

f) adjacency list:



adjacency matrix:

	A	B	C	E	F	G	H	I	J
A	0	6	0	2	0	0	0	0	0
B	6	0	0	0	1	0	0	0	0
C	0	2	0	0	0	1	0	0	0
E	0	0	0	0	8	0	3	0	0
F	0	0	0	0	0	0	0	0	0
G	0	0	0	0	2	0	0	0	5
H	0	0	0	0	0	0	0	1	0
I	0	0	0	0	2	0	0	0	1
J	0	0	0	0	0	0	0	0	0

g) DFS: examines A, B, F, *(backtracks to A)*, E, H, I

returns path {A, E, H, I}

h) BFS: examines C, *(neighbors of C)* B, G, *(neighbors of B)* A, F, *(neighbors of G)* J,
(neighbors of A) E, *(neighbors of E)* H

returns path {C, B, A, E, H}

7. Graphs (write)

As with any programming problem, there are many correct solutions. Here are two:

```
// solution 1: two loops; each checks one way
bool containsBidiPath(BasicGraph& graph, Vector<Vertex*>& path) {
    // check that every vertex in the path is contained in the graph
    for (Vertex* v : path) {
        if (graph.getVertex(v->name) != v) {
            return false;
        }
    }
    // check that each adjacent pair of vertices are neighbors
    for (int i = 0; i < path.size() - 1; i++) {
        Vertex* first = path[i];
        Vertex* second = path[i + 1];
        if (!graph.isNeighbor(first, second)) {
            return false;
        }
    }
    for (int i = path.size() - 1; i > 0; i--) {
        Vertex* first = path[i];
        Vertex* second = path[i - 1];
        if (!graph.isNeighbor(first, second)) {
            return false;
        }
    }
    return true;
}

================================================================================

// solution 2: single loop that checks everything
bool containsBidiPath(BasicGraph& graph, Vector<Vertex*>& path) {
    for (int i = 0; i < path.size() - 1; i++) {
        if (graph.getNode(path[i]->name) != path[i]
                || graph.getNode(path[i + 1]->name) != path[i + 1]
                || !graph.isNeighbor(path[i], path[i + 1])
                || !graph.isNeighbor(path[i + 1], path[i])) {
            return false;
        }
    }
    return true;
}
```

8. **Inheritance and Polymorphism (read)**

```
var1->a();              // J A
var1->b();              // COMPILER ERROR
var1->c();              // J C / K C
var2->a();              // J A
var2->b();              // COMPILER ERROR
var2->c();              // C C / J C / K C
var3->a();              // K A / E C
var3->b();              // COMPILER ERROR
var4->a();              // J A
var5->a();              // K A / K C
((Jerry*) var1)->a();   // J A
((Jerry*) var1)->b();   // COMPILER ERROR
((Chris*) var2)->d();   // C D / C C / J C / K C
((Eddie*) var3)->b();   // K A / E C / E B
((Jerry*) var4)->a();   // J A
((Jerry*) var4)->b();   // COMPILER ERROR
((Jerry*) var5)->b();   // COMPILER ERROR
```

9. Inheritance and Object-oriented Programming (write)

```
// RiggedDice.h
class RiggedDice : public Dice {
public:
    RiggedDice(int count, int min);
    virtual int getMin() const;
    virtual void roll(int index);
    virtual int total() const;
    virtual string toString() const;

private:
    int min;
};

bool operator <(const RiggedDice& rig1, const RiggedDice& rig2);
```

```
// RiggedDice.cpp
RiggedDice::RiggedDice(int count, int min) : Dice(count) {
    if (min < 0 || min >= 6) {
        throw min;
    }
    this->min = min;
}

int RiggedDice::getMin() const {
    return min;
}

void RiggedDice::roll(int index) {
    Dice::roll(index);
    while (getValue(index) < min) {    // do/while is okay
        Dice::roll(index);
    }
}

int RiggedDice::total() const {
    return Dice::total() + 1;
}

string RiggedDice::toString() const {
    return string("rigged ") + Dice::toString() + " min " + integerToString(min);
}

bool operator <(const RiggedDice& rig1, const RiggedDice& rig2) {
    int total1 = rig1.total();
    int total2 = rig2.total();
    if (total1 != total2) {
        return total1 < total2;
    } else {
        return rig1.getMin() < rig2.getMin();
    }
}
```