Section Solution

Problem 1 Solution: Dictionaries and Ternary Search Trees

a.  
```cpp
Dictionary::node *Dictionary::createNode(char ch) const {
    node n = {ch, NULL, NULL, NULL, NULL};
    return new node(n);
}
```

```cpp
void Dictionary::add(const string& word, const string& definition) {
    node **currp = &root;
    size_t len = word.size();
    size_t pos = 0;
    while (true) {
        if (*currp == NULL) *currp = createNode(word[pos]);
        node *curr = *currp;
        if (word[pos] == curr->letter) {
            pos++;
            if (pos == len) break;
            currp = &curr->equal;
        } else if (word[pos] > curr->letter) {
            currp = &curr->greater;
        } else {
            currp = &curr->less;
        }
    }
    if ((*currp)->definitions == NULL)
        (*currp)->definitions = new Vector<string>;
    (*currp)->definitions->add(definition);
}
```

b.  
```cpp
void Dictionary::deleteNode(node *curr) {
    if (curr == NULL) return;
    delete curr->definitions; // delete NULL is a no-op
    deleteNode(curr->less);
    deleteNode(curr->equal);
    deleteNode(curr->greater);
    delete curr;
}
```

```cpp
Dictionary::~Dictionary() {
    deleteNode(root);
}
```
Problem 2 Solution: Regular Expressions

```java
static void matchAllWords(const node *root, const string& regex,
    Set<string>& matches, const string& workingPrefix) {
    if (root == NULL) return;
    if (regex.empty()) {
        if (root->isWord) matches.add(workingPrefix);
        return; // return regardless of whether the working prefix was a word
    }
    if (regex.size() == 1 || !ispunct(regex[1])) {
        matchAllWords(root->suffixes.get(regex[0]), regex.substr(1),
            matches, workingPrefix + regex[0]);
    } else if (regex[1] == '?') {
        matchAllWords(root, regex.substr(2), matches, workingPrefix);
        matchAllWords(root->suffixes.get(regex[0]), regex.substr(2),
            matches, workingPrefix + regex[0]);
    } else if (regex[1] == '*') {
        matchAllWords(root, regex.substr(2), matches, workingPrefix);
        matchAllWords(root->suffixes.get(regex[0]), regex,
            matches, workingPrefix + regex[0]);
    } else {
        // assume regex[1] == '+'
        string equivregex = regex;
        equivregex[1] = '*'; // reframe y+ as yy*
        matchAllWords(root->suffixes.get(equivregex[0]), equivregex,
            matches, workingPrefix + equivregex[0]);
    }
}
```

```java
static void matchAllWords(const node *trie, const string& regex,
    Set<string>& matches) {
    matchAllWords(trie, regex, matches, "");
}
```

Problem 3 Solution: People You May Know

a. We use a brute-force double for loop to gain access to all of your friends’ friends. We maintain a `peopleYouAlreadyKnow` set (yourself, all of your friends) so that we don’t accidentally include a friend in the return value.

The solution here makes the reasonable assumption that each user is uniquely identified by the address of his or her user record.

```java
static Set<user *> getFriendsOfFriends(user *loggedinuser) {
    Set<user *> peopleYouMayKnow;
    Set<user *> peopleYouAlreadyKnow =loggedinuser->friends;
    peopleYouAlreadyKnow += loggedinuser;
    for (user *fr: loggedinuser->friends) {
        for (user *friendOfFriend: fr->friends) {
            if (!peopleYouAlreadyKnow.contains(friendOfFriend)) {
                peopleYouMayKnow += friendOfFriend;
            }
        }
    }
    return peopleYouMayKnow;
}
```
Note that there’s no real need to check to see if a friend of a friend has already been added to the `peopleYouMayKnow` set. There’s no harm in adding the same item multiple times, as the set discards all duplicates.

b. Let’s go with breadth-first search, as with this right here:

```cpp
static Set<user *> getKthDegreeFriends(user *loggedinuser, int k) {
  Set<user *> result;
  HashMap<user *, int> distances;
  Queue<user *> q;

  // initialize BFS
  user *curr =loggedinuser;
  q.enqueue(curr);
  distances[curr] = 0;

  while (!q.isEmpty()) {
    curr = q.dequeue();
    if (distances[curr] > k) break;
    if (distances[curr] == k) result += curr;

    // friend is a C++ keyword, so go with buddy
    for (user *buddy: curr->friends) {
      if (!distances.containsKey(buddy)) {
        distances[buddy] = distances[curr] + 1;
        q.enqueue(buddy);
      }
    }
  }

  return result;
}
```

**Problem 4 Solution: Detecting Cycles**

This is a variation on the depth-first traversal example presented in the textbook. The trick is to maintain a list of `nodes *s` being explored, and if we ever trip over the same vertex twice during a depth-first exploration, then we have a cycle and need to say so.

One neat feature of this particular solution is that it properly handles those graphs that aren’t fully connected, but instead come as two or more disconnected components.

```cpp
static bool isReachable(node *n, Set<node *> & activelyBeingVisited, Set<node *> & previouslyVisited) {
  if (activelyBeingVisited.contains(n)) return true;
  if (previouslyVisited.contains(n)) return false;
  activelyBeingVisited += n;
  for (arc *arc: n->arcs) {
    if (isReachable(arc->to, activelyBeingVisited, previouslyVisited))
      return true;
  }
  activelyBeingVisited -= n;
  previouslyVisited += n;
  return false;
}
```
static bool containsCycle(graph& graph) {
    Set<node *> previouslyVisisted;
    Set<node *> toBeVisited = graph.nodes;
    while (!toBeVisited.isEmpty()) {
        node *front = toBeVisited.first();
        Set<node *> activelyBeingVisited;
        if (isReachable(front, activelyBeingVisited, previouslyVisisted))
            return true;
        toBeVisited -= previouslyVisisted;
    }
    return false;
}