

## Section Solution

---

### Discussion Problem 1 Solution: Braided Lists

```
struct node {
    int value;
    node *next;
};
```

The recursive approach is easier to code up, but requires more a much more clever algorithm.

```
void braid(node *list) {
    Queue<int> numbers;
    braidRec(list, numbers);
}

void braidRec(node *list, Queue<int>& numbers) {
    if (list == NULL) return;
    numbers.enqueue(list->value);
    braidRec(list->next, numbers);
    node *newNode = new node;
    newNode->value = numbers.dequeue();
    newNode->next = list->next;
    list->next = newNode;
}
```

A fully iterative approach is best handled in two passes:

```
void build(node *list) {
    node *reverse = NULL;
    for (node *curr = list; curr != NULL; curr = curr->next) {
        node *newNode = new node;
        newNode->value = curr->value;
        newNode->next = reverse;
        reverse = newNode;
    }

    // reverse now addresses a memory-independent version of the original list,
    // where all of the nodes are in reverse order.

    node *rest = reverse; // rest addresses part that has yet to be braided in
    for (node *curr = list; curr != NULL; curr = curr->next->next) {
        node *next = rest->next;
        rest->next = curr->next;
        curr->next = rest;
        rest = next;
    }
}
```

**Note:** for some reason, industry interviews tend to include a question that asks you to reverse a linked list. Commit the first half of the iterative solution to memory. ☺

## Discussion Problem 2 Solution: Skip Lists

This is conceptually dense, but an optimal implementation is fairly short. There are several ways to implement this, and I take the **node \*\*-list** approach of tracking the address of the relevant links vector at any one moment, and decide whether I can advance to the node with just as many forward links, or whether I need to descend down through the current links vector and be less aggressive about skipping forward.

The will most certainly be the basis for your skip list implementation when the time comes for you to code these up for the lab part of section.

```
bool skipListContains(Vector<skipListNode *>& heads, int key) {
    Vector<skipListNode *> *currs = &heads;
    int level = currs->size() - 1;

    while (level >= 0) {
        skipListNode *curr = (*currs)[level];
        if (curr != NULL && curr->value == key) return true;
        if (curr == NULL || curr->value > key) {
            level--;
        } else {
            currs = &(curr->links);
        }
    }

    return false;
}
```

## Lab Problem 1 Solution: Finding Words In Character Trees

As is always the case, there are many ways to get this to work. I'm just presenting the solution I came up with when I came up with this problem a few years ago.

```
bool wordExists(node *tree, string str) {
    if (str.empty()) return true; // the empty tree contains the empty string
    if (tree == NULL) return false;

    if (suffixExists(tree, str))
        return true;

    return wordExists(tree->left, str) || wordExists(tree->right, str);
}

bool suffixExists(node *tree, string suffix) {
    if (suffix.empty()) return true;
    if (tree == NULL) return false;
    if (suffix[0] != tree->ch) return false;

    return suffixExists(tree->left, suffix.substr(1)) ||
           suffixExists(tree->right, suffix.substr(1));
}
```

## Lab Problem 2 Solution: Implementing the SkipSet

I'll post my own solution this weekend, but just so you see my implementation in print form, here you go:

```
class SkipSet {
public:
    SkipSet();
    ~SkipSet();

    int size() { return numElems; }
    bool isEmpty() { return size() == 0; }

    bool contains(int value);
    void add(int value);

private:
    struct node {
        int value;
        Vector<node *> links;
    };

    int numElems;
    Vector<node *> spine;
    int generateRandomLevel();
    void deleteAll();
};
```

As complicated as this data structure is, its implementation is so compact it's almost rude.

```
SkipSet::SkipSet() { numElems = 0; }
SkipSet::~SkipSet() { deleteAll(); }

bool SkipSet::contains(int value) {
    Vector<node *> *currs = &spine;
    int level = currs->size() - 1;

    while (level >= 0) {
        node *curr = (*currs)[level];
        if (curr != NULL && curr->value == value) return true;
        if (curr == NULL || curr->value > value) {
            level--;
        } else {
            currs = &(curr->links);
        }
    }

    return false;
}
```

```

void SkipSet::add(int value) {
    if (contains(value)) return; // traditional set
    int numLevelsRequired = generateRandomLevel();
    node *newNode = new node;
    newNode->value = value;
    while (newNode->links.size() < numLevelsRequired) {
        newNode->links.add(NULL);
    }
    while (spine.size() < numLevelsRequired) {
        spine.add(NULL);
    }

    Vector<node *> *currs = &spine;
    int level = numLevelsRequired - 1;
    while (level >= 0) {
        while ((*currs)[level] != NULL && (*currs)[level]->value < value) {
            currs = &((*currs)[level]->links);
        }
        newNode->links[level] = (*currs)[level];
        (*currs)[level] = newNode;
        level--;
    }

    numElems++;
}

int SkipSet::generateRandomLevel() {
    int numLevels = 1;
    while (RandomChance(0.25)) {
        numLevels++;
    }
    return numLevels;
}

void SkipSet::deleteAll() {
    if (isEmpty()) return;
    node *curr = spine[0];
    while (curr != NULL) {
        node *next = curr->links[0];
        delete curr;
        curr = next;
    }
}

```