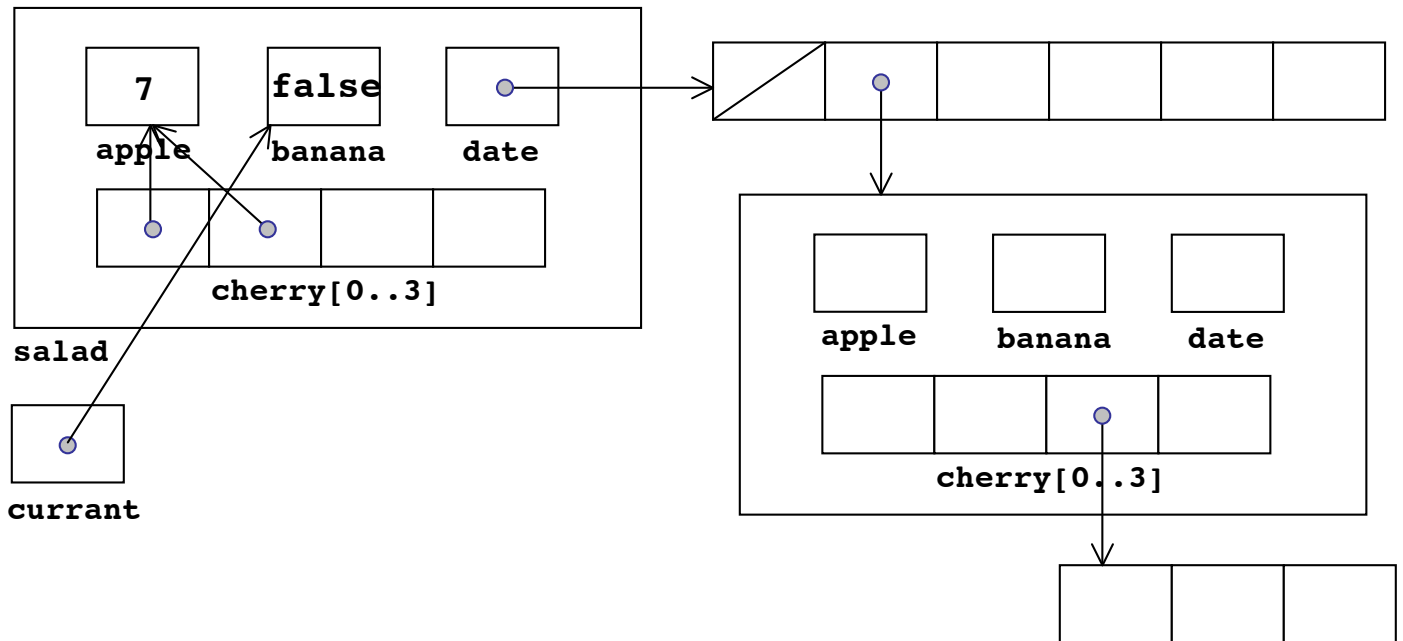


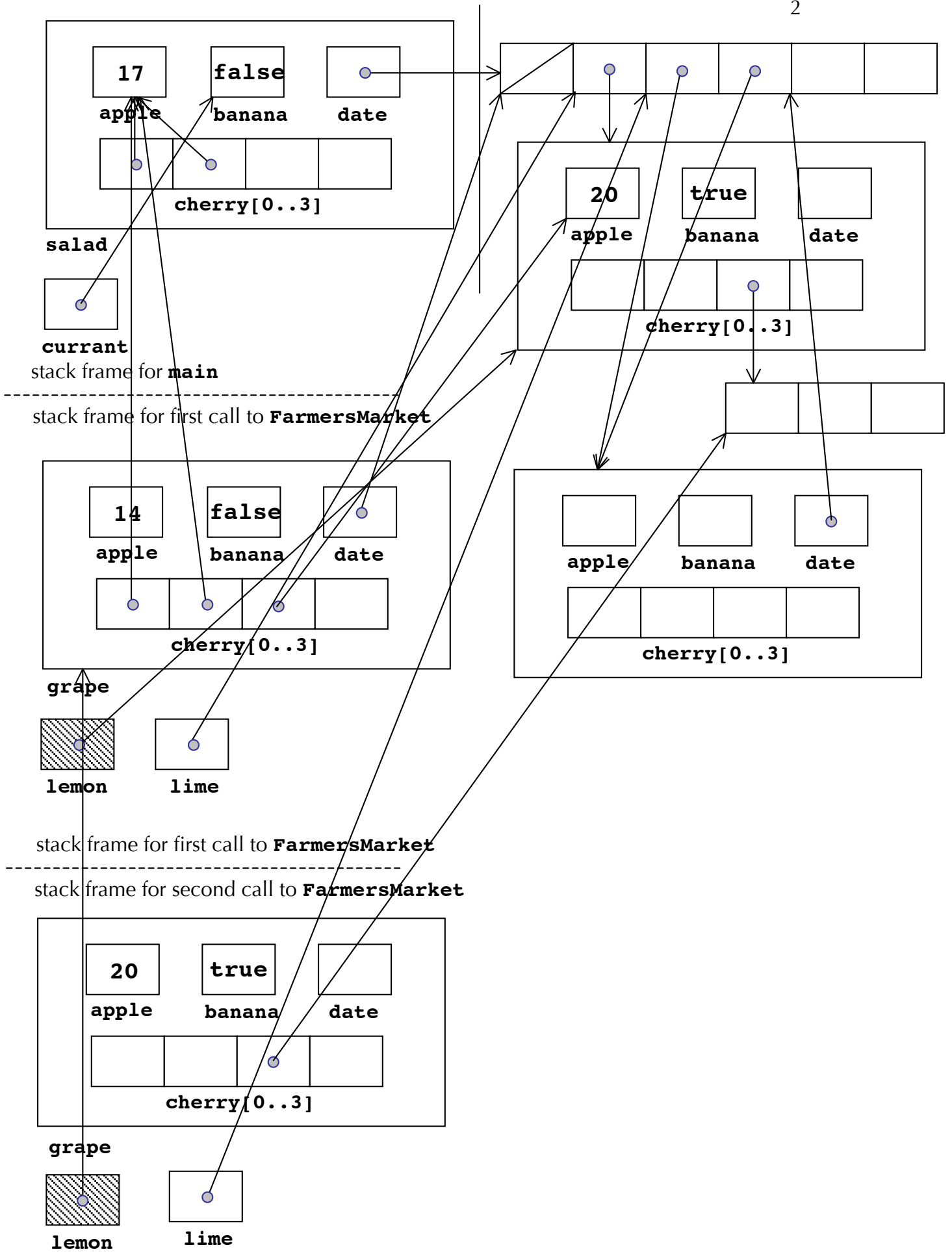
## Section Solution

### Discussion Problem Solution 1: Farmers Market Memory Trace



The drawing above depicts the state of memory—stack on the left, heap on the right—at the moment the problem asks for a snapshot of memory (right before the call to **FarmersMarket**). All dynamically allocated memory is drawn from the heap, and all local variables (in this case, **salad** and **currant**) are allocated on the stack.

The next page includes an elaborate drawing detailing the state of memory just after a base case is detected but just before the recursive call returns. [Note that no memory is orphaned at all, since everything is accessible from some active local variable.]



## Lab Problem 1 Solution: The Sparse String Array

This problem is difficult for many reasons:

- For many of you, this is the first time you've dealt with internal representations and external representations that are so very different from one another. The takeaway point is that the external representation defines behavior, and the internal representation is dictated by the need to clearly and efficiently implement.
- This is almost certainly the first time you've used exposed memory addresses and dynamic memory allocation for the purposes of building a dynamic data structure. Mastering pointers and memory is one of the most difficult C++-specific topics that students need to learn during the course of CS106B and X. If you're to truly master the material, you need to drive yourself to implement lots of data structures and force yourself to understand exactly what each line of its internal implementation is doing for you.

Here's the core of my own solution:

```

SparseStringArray::SparseStringArray(int arrayLength, int groupSize) {
    this->arrayLength = arrayLength;
    this->groupSize = groupSize;
    numGroups = arrayLength / groupSize;
    groups = new group[numGroups];
    for (int group = 0; group < numGroups; group++) {
        groups[group].bitmap = new bool[groupSize];
        for (int i = 0; i < groupSize; i++) {
            groups[group].bitmap[i] = false;
        }
    }
}

void SparseStringArray::set(int index, string str) {
    int group = index / groupSize;
    int indexWithinBitmap = index % groupSize;
    int indexWithinElements = 0;
    for (int i = 0; i < indexWithinBitmap; i++) {
        if (groups[group].bitmap[i]) {
            indexWithinElements++;
        }
    }

    if (groups[group].bitmap[indexWithinBitmap]) {
        groups[group].elements[indexWithinElements] = str;
    } else {
        groups[group].elements.insertAt(indexWithinElements, str);
        groups[group].bitmap[indexWithinBitmap] = true;
    }
}

string SparseStringArray::get(int index) {
    int group = index / groupSize;
    int indexWithinBitmap = index % groupSize;
    if (!groups[group].bitmap[indexWithinBitmap]) {
        return "";
    }
    int indexWithinElements = 0;
    for (int i = 0; i < indexWithinBitmap; i++) {

```

```

        if (groups[group].bitmap[i]) {
            indexWithinElements++;
        }
    }

    return groups[group].elements[indexWithinElements];
}

template <typename ClientData>
void SparseStringArray::mapAll(void (mapfn)(int, string, ClientData&),
                               ClientData& data) {
    for (int group = 0; group < numGroups; group++) {
        int indexWithinElements = 0;
        for (int indexWithinBitmap = 0;
             indexWithinBitmap < groupSize; indexWithinBitmap++) {
            int index = group * groupSize + indexWithinBitmap;
            string str =
                groups[group].bitmap[indexWithinBitmap] ?
                groups[group].elements[indexWithinElements++] : "";
            mapfn(index, str, data);
        }
    }
}

SparseStringArray::~SparseStringArray() {
    for (int i = 0; i < this->numGroups; i++) {
        delete[] groups[i].bitmap;
    }

    delete[] groups;
}

```