

## Section Handout

---

### Discussion Problem 1: Farmers Market Memory Trace

Analyze the following program, and draw the state of memory at the two points indicated. Be sure to differentiate between stack and heap memory, mark un-initialized values, and identify where memory has been orphaned.

```
struct fruit {
    int apple;
    bool banana;
    int *cherry[4];
    fruit **date;
};

int main() {
    fruit salad;
    salad.apple = 7;
    salad.cherry[0] = &salad.apple;
    salad.cherry[1] = *(salad.cherry);
    salad.date = new fruit *[6];
    salad.date[0] = NULL;
    salad.date[1] = new fruit;
    salad.date[1]->cherry[2] = new int[3];
    bool *currant = &salad.banana;
    currant[0] = false;
```

← Draw the state of memory here, before the function call.

```
FarmersMarket(salad, *(salad.date[1]), &salad.date[1]);
return 0;
}

void FarmersMarket(fruit grape, fruit& lemon, fruit **lime) {
    if (grape.banana) {
        lime[1] = new fruit;
        lime[1]->date = &lime[2];
        lime[0] = lime[1];
        lemon.cherry[1][0] = 17;

        return;
    }

    grape.cherry[2] = &lemon.apple;
    grape.apple *= 2;
    lemon.apple = 20;
    lemon.banana = true;
    FarmersMarket(lemon, grape, &lime[1]);
    lemon.date = NULL;
}
```

← And draw the state of memory here, just before return.

## Lab Problem 1: The Sparse String Array

A **SparseStringArray** is an array-like data structure that provides superduper fast access to its elements, and near constant-time **set** and **get** operations. It layers array semantics over an ordered sequence of C++ strings, with the understanding that an overwhelming majority of the strings are **empty**. The **SparseStringArray** is different from our C++ **Vector** and other array-like data structures, because its size is set at construction time (as with a traditional array), and it's memory footprint is kept to an absolute minimum. In theory, each empty string requires just one bit of storage, which is less than 3% of the memory cost incurred by the allocation of a full empty **string**. The implementation is slower than our **Vector**, but it's a wise choice when memory is at a premium and an overwhelmingly large fraction of the strings being stored are empty.

Our **SparseStringArray** is backed by an array of **groups**, where each **group** is responsible for managing a contiguous subset of array indices. The programmer specifies not only the logical length of the **SparseStringArray**, but also the group size. If the logical length of the full **SparseStringArray** is, for instance, established as 50,000, and the group size is established to be 100, then group 0 manages indices 0 through 99, group 1 manages indices 100 through 199, group 2 manages indices 200 through 299, and so forth. All search, set, and get operations are passed on to the appropriate group. Here's a glimpse of our class's **private** sector:

```
private:
    group *groups; // dynamically allocated array of structs, defined below
    int numGroups; // number of groups
    int arrayLength; // logical length of the full SparseStringArray
    int groupSize; // number of strings managed by each group
```

Each group contains a bitmap, which is an array of **bools** whose length is equal to the group size, and a C++ **Vector<string>** to store *just* the non-empty strings. Search for a particular element amounts to search within a particular group at index **i**. If **bitmap[i]** is **false**, then the string at the **i**<sup>th</sup> position is understood to be the empty string. If instead **bitmap[i]** is **true**, then the group needs to find the corresponding string in the C++ **Vector<string>**.

```
struct group {
    bool *bitmap; // set to be of size 'groupSize'
    Vector<string> strings; // ordered Vector<string> on the non-empty strings
};
```

Each **true** in a group's bitmap corresponds to some **string** in the same group's **Vector<string>**. The **true** at the lowest index in the bitmap corresponds to the 0<sup>th</sup> entry in the **Vector**; the **true** at the second lowest index in the bitmap corresponds to the 1<sup>st</sup> entry in the **Vector**, and so forth; and the total number of **true**s should be equal to the logical length of the accompanying **Vector**. (In practice, the **bool** array would be compressed to use just one bit of memory for each Boolean value, but for our purposes we

won't implement that optimization, since it requires advanced C directives we haven't covered.)

Variations of this data structure are used in industry when insanely large arrays—with lengths in the billions or trillions—contribute to a larger system. It also has the neat feature that individual groups can be distributed across multiple processors or multiple machines.

Here's the .h file for the **SparseStringArray** class.

```
class SparseStringArray {
public:
    SparseStringArray(int arrayLength, int groupSize);
    ~SparseStringArray();

    void set(int index, string str);
    string get(int index);
    template <typename ClientData>
        void mapAll(void (mapfn)(int index, string str, ClientData& data),
                    ClientData& data);

private:
    struct group {
        bool *bitmap;           // set to be of size 'groupSize'
        Vector<string> strings; // ordered Vector<string> on the non-empty strings
    };

    group *groups; // dynamically allocated array of groups
    int numGroups; // number of groups
    int arrayLength; // logical length of the full SparseStringArray
    int groupSize; // number of strings managed by each group
};
```

Of course, the **SparseStringArray** presents the **illusion** that all strings, both empty and nonempty, are stored in a sequential, array-like manner. But **you** understand the smoke and mirrors of the implementation: the internal representation is such that only nonempty strings are really stored. Your job is to implement the constructor, destructor, and the three methods in a way that's consistent with the description outlined above.

Here's a test program that illustrates how a client can interact with a **SparseStringArray**.

```
void PrintElement(int ignored, string str, ostream& os) {
    if (str.empty()) return; // technically not needed, but faster than operator<<
    os << str;
}

int main() {

    SparseStringArray ssa(70000, 35);

    ssa.set(33001, "need");
    ssa.set(58291, "more");
    ssa.set(33000, "Eye");
    ssa.set(33000, "I");
    ssa.set(67899, "cowbell!");

    cout << "Serialization: ";
    ssa.mapAll(PrintElement, cout);
    cout << endl;

    return 0;
}
```

Here's the output of that test program:

```
Serialization: Ineedmorecowbell!
```

We've written a full set of unit tests to help you nail the implementation and all of its edge cases. You should only need to update the **sparse-string-array.cpp** file and nothing else.