

Section Handout

Problem 1: The Sparse String Vector

A **SparseStringVector** is a **Vector<string>**-like data structure that provides super fast access to its elements, and near constant-time insertion and deletion. It layers array semantics over an ordered collection of C++ strings, with the understanding that **most** of the strings are **empty**. The **SparseStringVector** is different from our C++ **Vector** and other array-like data structures, because it's stingy in its use of memory. 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 **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 **SparseStringVector** 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 **SparseStringVector**, but also the group size. If the logical length of the full **SparseStringVector** 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, insert, and delete 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 SparseStringVector
    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 **ith** 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 0th entry in the **Vector**; the **true** at the second lowest index in the bitmap corresponds to the

1st 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.)

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

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

    bool insert(int index, string str);
    string serialization();

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 SparseStringVector
    int groupSize; // number of strings managed by each group
};
```

Of course, the **SparseStringVector** gives the illusion that all strings, both empty and nonempty, are stored in an array-like manner. But **you** know as the implementer the internal representation is such that only nonempty strings are really stored. Your job is to implement the constructor, destructor, and the two methods in a way that's consistent with the description outlined on the previous page.

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

```
int main()
{
    SparseStringVector ssv(70000, 35);

    ssv.insert(33001, "need");
    ssv.insert(58291, "more");
    ssv.insert(33000, "Eye");
    ssv.insert(33000, "I");
    ssv.insert(67899, "cowbell!");
    cout << "Serialization: " << ssv.serialization() << endl;
    return 0;
}
```

Here's the output of that test program:

```
Serialization: Ineedmorecowbell!
```

- a. Present implementations for the constructor and destructor. The constructor is designed to initialize a **SparseStringVector** to represent an array of the specified logical length, where every single element is the empty string. The destructor disposes of all resources that have been tapped during a **SparseStringVector**'s lifetime.

```
SparseStringVector::SparseStringVector(int arrayLength, int groupSize);
SparseStringVector::~~SparseStringVector();
```

- b. Now implement the more involved **insert** function, which ensures that the string received as **str** stored in the proper **Vector<string>** within the proper group and that the proper **bool** in the **bitmap** is set. Note that you need to handle two different cases: the one where a string is being inserted as a particular index for the very first time, and the other where a previously inserted string is being replaced by something else.

You may assume the string being inserted is nonempty. You should return **true** if a string is being inserted at the specified position for the very first time, **false** otherwise.

```
bool SparseStringVector::insert(int index, string str);
```

- c. Finally, implement the **serialize** method, which returns the ordered concatenation of every single string held by the **SparseStringVector**.

```
string SparseStringVector::serialization();
```

Problem 2: Removing Duplicates

Write a function **RemoveDuplicates** that given a linked list will remove and free the second of all neighboring duplicates found in the list. If the incoming list is (5 5 22 37 89 89 15 15 22) the function will destructively modify the list to contain (5 22 37 89 15 22). Don't worry about duplicate sequences longer than 2 or duplicates that aren't right next to each other in the list.

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

void RemoveDuplicates(node *list);
```

Problem 3: Briaded Lists

Write a function called **Braid** that takes the leading address of a singly linked list, and weaves the reverse of that list into the original.

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

Here are some examples:

list	list after call Braid(list)
1 → 4 → 2	1 → 2 → 4 → 4 → 2 → 1
3	3 → 3
1 → 3 → 6 → 10 → 15	1 → 15 → 3 → 10 → 6 → 6 → 10 → 3 → 15 → 1

You have this page and the next page to present your solution.

```
void Braid(node *list);
```