

## Section Solution

---

### Solution 1: The Sparse String Vector

- a. Present implementations for the constructor and destructor.

```
SparseStringVector::SparseStringVector(int arrayLength, int groupSize)
{
    this->arrayLength = arrayLength;
    this->groupSize = groupSize;
    numGroups = arrayLength / groupSize;
    groups = new group[numGroups];

    for (int i = 0; i < numGroups; i++) {
        groups[i].bitmap = new bool[this->groupSize];
        for (int j = 0; j < this->groupSize; j++) {
            groups[i].bitmap[j] = false;
        }
    }
}

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

    delete[] groups;
}
```

- 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.

```
bool SparseStringVector::insert(int index, string str)
{
    int group = index / groupSize;
    int indexWithinBitmap = index % groupSize;
    int indexWithinVector = 0;

    for (int i = 0; i < indexWithinBitmap; i++) {
        if (groups[group].bitmap[i])
            indexWithinVector++;
    }

    bool previouslyInserted = groups[group].bitmap[indexWithinBitmap];
    if (previouslyInserted)
        groups[group].strings[indexWithinVector] = str;
    else
        groups[group].strings.insertAt(indexWithinVector, str);
    groups[group].bitmap[indexWithinBitmap] = true;
    return !previouslyInserted;
}
```

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

```
string SparseStringVector::serialization()
{
    string str = "";
    for (int i = 0; i < numGroups; i++) {
        for (int j = 0; j < groups[i].strings.size(); j++) {
            str += groups[i].strings.getAt(j);
        }
    }
    return str;
}
```

### Solution 2: Removing Duplicates

```
void RemoveDuplicates(node *list)
{
    for (node *cur = list; cur != NULL; cur = cur->next) {
        if (cur->next != NULL && cur->value == cur->next->value) { // match?
            node *duplicate = cur->next; // remember
            cur->next = cur->next->next; // circumvent
            delete duplicate; // dispose
        }
    }
}
```

### Solution 3: 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. 😊