# CSE250, Spring 2014 Assignment 6 Due Mon. Apr. 1, in class

**Hardcopy submission only**—no online submission.

**Reading:** For the rest of this week and next week, read Chapter 4 with a "bird's-eye view." By this I mean treat it as having three major topics: (i) vectors, (ii) linked lists, and (iii) iterators. These topics will be treated together rather than separately and sequentially. Also read Chapter 5, sections 5.1–5.3 only, and treat section 5.3 as an example of how a linked list could be an alternate implementation of a stack. Treat sections 6.1–6.3 the same way, and focus on section 6.4. In Figure 6.11 on page 378, the left-hand side is an array, but imagine instead that it is a linked list—of pointers to "chunks" of data that are vectors. This is the souped-up more-than-just-a-deque that the C++ Standard Template Library actually provides, and programming a partial work-alike of it (called `FlexArray` for "flexible array") is the project for upcoming weeks.

For code we are also transiting from the `Java2C++` directory to the `.../PROJECTS/BASE/` directory on `timberlake`. Recitations this week are introducing the templated singly-linked list class `SList.h` (all code in `.h` file for templates), which has been added to both locations. Your `FlexArray` can build upon its structure—you are welcome to grab this file and "morph" it, changing the header comment appropriately. The main difference in going to `FlexArray` will be in going to a two-tier system of indexing. The following exercises are for conceptual understanding of the abstract idea of arrays tiered into "chunks" and practice with the math that will go into your code. Again they are hardcopy-only, and owing to spring break, due on **Monday**.

(1) Suppose we make a chart of all the children at a combination day-care center and elementary school, with one row for each age. Each row "node" has an array `elements` of the children of that age who are present. We also consider all of the $N$ children present to be "globally numbered" $0 \ldots N-1$ beginning with the babies in age row 0. Suppose the first few rows are:

```
age 0: 7 babies, elements[0..6]
age 1: 9 toddlers, elements[0..8]
age 2: 6, elements[0..5]
age 3: 10, elements[0..9]
etc.
```

The problem at hand is, given the "global number" of a child, find both the age node and the index of the child in that node. For instance, the eighteenth child—whose global index is 17—is the second child of the age-2 node, and because we index from 0, we say the child's node is 2 and "local index" is 1. Global index 0 becomes node 0, local index 0. Global index 31 is the last child in node 3, local index 9.

*Sketch* code to find the node and the local index, given any global index. Pretend that the nodes are in a linked list whose `firstNode` has the `elements` array and a field `Node* next;` which is a pointer to the next node (in this case, the age 1 node). Assume that the elements array has no extra space, so that its `size()` method always returns the actual number of children of the corresponding age who are present. (I.e., it is like the `chains` array

on Assignment 4, and unlike the deque/queue arrays which were intiialized with extra space to make a faster implementation.) Also detect whether a given global index is out-of-range, i.e., $N$ or higher. (Do a C++ sketch; indeed we want you to get the logic right on-paper before you even start to code it. 18 pts.)

(2) Now we *tweak* the problem to find, not the global index of a child, but rather ine index of a place where an additional child could go. Here we don't know the actual age of thenew child, but are given only the global index that this child needs to have. To see the issue, suppose the given global index is 16. The child could go as the tenth toddler in the age-1 node, with local index 9, calling `push_back` on the `elements` array. Or the child could become the new first member of the age-2 node.

We prefer the *former*. Revise the code sketch to find the node and new local index of an inserted child. Note that global index $N$ is now considered in-range—it will cause the child to be inserted at the end of the last row. Explain how the bound in your while-loop (or for-loop) has to change as a result. (A buzzword for the difference from problem (3) is that now we are looking for a "fencepost" between sections of fence, or at the ends, rather than a section of fence itself. 12 pts.)

(3) Now suppose that whenever an age group exceeds a given `capacity` limit, it has to be split into two sub-nodes, each with half (give-or-take one child) of the children. Sketch a C++ routine `split` that would be called when after inserting a child, `elements->size()` becomes equal to (or exceeds—your choice) the given `capacity`. Sketch code to allocate a new `Node` after the current node, then move half of the elements from the current node into the new node. You may pretend that the `erase` and `insert` methods of the `vector` class take an integer index parameter rather than an "iterator" (syntax for iterators will be given on the project itself). Again, exact C++ syntax is not required in a "sketch," and we want you to do this on-paper first. You will receive graded feedback on your hardcopy before the coding part is due. (18 pts., for 48 total on this assignment)

**Project 1** will center on coding a templated class `FlexArray<T>` which includes these routines, and will provide (at least) these public constructors and methods:

```
FlexArray<T>(size_t nodeCap)
FlexArray<T>()
virtual ~FlexArray<T>()    (assignment and copy-ctor optional)
size_t size() const
bool empty() const
T& at(size_t i)
T& operator[](size_t i)
iterator begin() const                    //returns iterator to first item
iterator end() const                      //iterator *one-past* last item
iterator insert(size_t i, const T& item)  //returns iterator to new item
iterator erase(size_t i)                  //returns iterator to next item
```