Assignment #3

Due date: Friday, 12 February 2016, 5:59 pm

- *Write your solutions in the dialect of C++ used in class.*
- *Store your solution set of functions in a file named* `a3p1q1.cc`, `a3p1q2.cc`, ... , `a3p2.cc`
- *You may define your own* `main` *function in these files, but we will use our own* `main` *to test your code.*

Part I

We're going to implement a doubly-linked list — let's call it a *Stew*, because it's a bit like both a Stack and a Queue — that allows new elements to be added / removed / peeked at both ends of the list. A Stew supports the following operations (details to follow): `initStew`, `isEmpty`, `addFront`, `leaveFront`, `peekFront`, `addBack`, `leaveBack`, `peekBack`, `print`, and `nuke`. We will provide implementations for `initStew` and `isEmpty`; they do the obvious things. The next three operations operate on the front of the list, while the last three operate on the back.

The following main program should produce the output indicated.

```
int main (int argc, char* argv[]) {
    Stew s1;
    initStew (s1);
    addFront (s1, "alpha");
    addFront (s1, "beta");
    addFront (s1, "gamma");
    addBack (s1, "delta");
    cout << "This prints \"gamma beta alpha delta\" across four lines\n";
    print (s1, 'f');
    cout << "This prints \"delta alpha beta gamma\" across four lines\n";
    print (s1, 'r');
    leaveFront (s1);
    leaveBack (s1);
    cout << "This prints \"beta alpha\" in one line\n";
    cout << peekFront (s1) << " " << peekBack (s1) << endl;
    cout << "This nuke has no output, but is good form to call when done\n";
    nuke (s1);
    cout << "This assertion should succeed\n";
    assert (isEmpty (s1));
    cout << "Illegal direction should cause error mesg\n";
    print (s1, 'k');
}
```

We're going to do this by using `Node`s that have links in both directions (forwards and backwards), plus we're going to keep two special pointers: one to the first element and one to the last element. We've already defined `initStew` and `isEmpty` for you in the skeleton code on the course web page. Your job will be to implement the other functions (whose signatures we have defined for you in the skeleton code). For each of the following questions, use these definitions (which you must type in yourself into your code):

```
struct Node {
    string val;
    Node* next;
    Node* prev;
};
```

```
struct Stew {
    Node* first;
    Node* last;
};
```

Note: You must implement the Stew as a doubly-linked list, building on the code we have provided above. You may *not* use a `vector` or any other C++ library data structure to do the work for you.

1. Define `addFront`, `leaveFront`, and `peekFront` similar to how they were done in class, but make sure you adjust all pointers appropriately and `delete` unneeded `struct` instances. The first line of `leaveFront` and `peekFront` should be:

   ```
   assert (!isEmpty(s));
   ```

2. Define `addBack`, `leaveBack`, and `peekBack` similar to how they were done in class, but make sure you adjust all pointers appropriately and `delete` unneeded `struct` instances. Note that `addBack` adds new elements to the *end* of the list. The first line of `leaveBack` and `peekBack` should be:

   ```
   assert (!isEmpty(s));
   ```

3. Define `print` which takes a `Stew` and a single `char` (the direction: `'f'` for forward and `'r'` for reverse), and prints the stew in the desired direction. If the direction passed in is not `'f'` or `'r'`, then print the following:

   ```
   cerr << "Error, illegal direction: " << direction << endl;
   ```

   If an illegal direction is detected, just print that message and carry on; you don't need to abort execution.

4. Define `nuke` which takes a `Stew`, deletes all of the internal `Nodes`, and sets the `first` and `last` pointers of the `Stew` instance to `nullptr`.

Part II

In considering the various ways of implementing data structures, we have tended to prefer linked list approaches over, say, statically allocated approaches. For example, if we were limited to only statically allocated storage, then one way to implement a stack would be by using an array to hold the elements, and keeping track of the top element by keeping an integer index to the top element, as depicted in Figure 1.

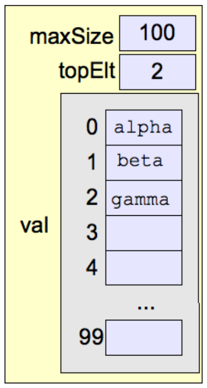

Figure 1: A statically allocated array stack.

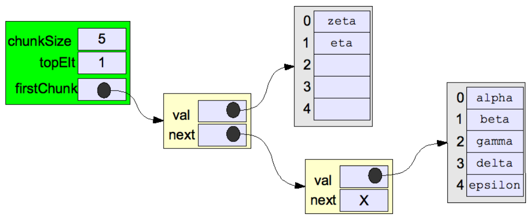

Figure 2: A "chunky" stack.

The obvious disadvantage of this approach is that there is an upper bound on how many elements the stack can hold at the same time (here, the bound is 100). Additionally, you must allocate the whole array at once; if you were using only a small percentage of the available elements most of the time, then you could be wasting a lot of space depending on how big the array was.[1] The advantages of this approach over a linked list are simplicity and speed: linked lists are easy to get wrong, and with an array you have direct access to all elements (that's not much of an advantage for a stack, but it would be if the ADT required immediate access to any given element).

[1]Yes, `vectors` don't have these problems, but they use dynamically allocated storage under the hood.

Compare this to using a linked list approach as done in class. Each element is stored in its own node, so there is a storage overhead of one pointer (typically, about four bytes) per element. Also, while creating and deleting new nodes are constant time operations in principle, if real-time performance is a big concern they can be relatively expensive operations compared to just accessing an array element.

We're going to investigate a hybrid approach inspired by B-trees (which you will learn about in later courses), which we're going to call a Chunky Stack. Basically, we're going to use a linked list approach, but instead of just one element each node will contain an array of `chunkSize` elements, for some constant `chunkSize`. This means that the stack will be unbounded, but that there will be an storage overhead of only one pointer per `chunkSize` elements (instead of one per element). Of course, this might mean some wasted space, but that will amount to at most `chunkSize-1` elements at any given moment. The second diagram shows an example of a Chunk Stack with seven elements and a `chunkSize` of 5. The order of insertion was: `alpha`, `beta`, `gamma`, `delta`, `epsilon`, `zeta`, `eta`.

Note that because we want to be flexible about the chunk size, you will have to use a dynamically allocated array as discussed in class. We also suggest you use two different kinds of `struct`s, one called `Stack` for the stack itself (the green box; you have access to a colour printer, right?) and one called `NodeChunk` for the various nodes (the pale yellow boxes) that store the elements (or more precisely, store pointers to the dynamic arrays that store the elements).

The procedures you are to implement are given below:

```
NodeChunk* createNewNodeChunk (int N) {...}
void initStack (int chunkSize, Stack& s) {...}
bool isEmpty (const Stack& s) {...}
void push (string val, Stack& s) {...}
void pop (Stack& s) {...}
string top (const Stack& s) {...}
void swap (Stack& s) {...}
int size (const Stack& s) {...}
```

Any given stack has a fixed `chunkSize` (a positive integer), which is set when you initialize it (via `initStack`); that is, all of the `NodeChunk`s will have the same number of elements for a given stack. However, you should note that two different stacks can have different `chunkSize`s:

```
Stack s1;
initStack(5,s1);
Stack s2;
initStack(100,s2);
```

Some notes on the functions:

- The procedure `createNewNodeChunk` creates a new `NodeChunk` and initializes it appropriately (look at the diagram to get an idea of what needs to be done).
- The procedures `push`, `pop`, and `top` do what you expect. However, `push` needs to check if the current first `NodeChunk` is full; if so, another `NodeChunk` needs to be allocated and linked in place. Similarly, `pop` needs to check if the element being removed is the last one in the current first `NodeChunk`; if so, that `NodeChunk` should be deleted (and so should its dynamic array), and the appropriate links should be adjusted. This means, for example, that an empty Chunky Stack would have a nullptr in `firstChunk`. You should probably `assert` at the beginning of `pop` and `top` that the stack isn't empty.
- The procedure `size` should return the current number of elements in the Stack (it would return 7 for the example in the diagram).
- The procedure `swap` should swap the top two elements. In the example shown in the diagram `zeta` and `eta` would change positions. In general, there is one tricky case you have to consider. Note that it's an error if `swap` is called when there are fewer than two elements (you should use `assert` to check this).

When you test your program on your own, try different values of `chunkSizes`. Two good examples are 1 (effectively giving you a linked list stack) and, say, 100 (effectively giving you an array stack). Try some other values too.