CPSC 221: Programming Project 1

First Milestone (`proj1A`) Due: Friday September 28, 2012 at 9pm via `handin`.
Last Milestone (`proj1B`) Due: Monday October 10, 2012 at 9pm via `handin`.

Out: September 12, 2012 Last Updated: September 12, 2012

Indexing a book

The goal of this assignment is to create a program `index` that will produce the index for a book. The input to `index` is a text file containing the book. The output of `index` is a text file with the words of the book in sorted order and for each word the number of occurrences of the word (in parentheses) and a list of the pages on which that word appears.

I should be able to type "`index book.txt index.txt`" at a Unix prompt to produce the index `index.txt` for the book `book.txt` using your program.

For the sample `ch1.txt` (Chapter 1 of Alice's Adventures in Wonderland), the first 16 lines of the output index are:

```
a (52) 1-6
about (8) 2-5
across (2) 1
actually (1) 1
adventures (1) 1
advice (1) 5
advise (1) 5
afraid (1) 3
after (5) 1-2,5
afterwards (1) 1
again (4) 1-3
against (1) 5
air (2) 2-3
alas (2) 3,5
alice (27) 1-6
alice's (2) 1,3
```

Overview

The basic steps of the program are:

1. Read the input file (book) one line at a time.
2. Remove punctuation from the line.

   In order that your "words" are the same as my "words", use `getline(fin,line)` to get a line of text (`fin` is a `ifstream`) and the following code to remove punctuation and convert characters to lowercase:

```
#include <string>       // provides string class
#include <cctype>       // provides isalpha() and tolower()

// Remove all characters except letters (A-Z,a-z) from line,
// except keep '-' or '\'' if they are between letters.
void lowercaseWords(string & line) {
  for( string::iterator it = line.begin(); it != line.end(); ++it ) {
    if( !isalpha(*it) ) {
      if( (*it != '-' && *it != '\'') ||
          it == line.begin() || it+1 == line.end() ||
          !isalpha(*(it-1)) || !isalpha(*(it+1)) ) {
        *it = ' ';
      }
    } else {
      *it = tolower(*it);
    }
  }
}
```

3. For each word in the line:

   (a) Look up (`find`) the entry for that word in a data structure that you implement called the *index structure*.

   (b) If it is not found, `insert` a new entry.

   (c) Update the entry to reflect the new occurrence of the word.

   Note that you can extract words from the line using a `stringstream` as follows:

   (Remember `#include <sstream>`.)

   ```
   string word;
   istringstream iss(line, istringstream::in);
   while( iss >> word ) {
   ...
   }
   ```

4. After all the lines are processed, print out the entries in the index structure in word-order. Please **make your index output look exactly like the sample shown above**.

   Each entry should contain a word, an occurrence count, and a vector of the distinct page numbers on which the word appears. Assume that each page contains 40 lines and that page numbering starts at 1. Note that the length of the vector may differ from the occurrence count because a word may appear more than once on a page. The word `alice` appears 27 times on pages 1, 2, 3, 4, 5, and 6. Rather than print `1,2,3,4,5,6` in the index, you should print runs of consecutive pages as `1-6`. **But add this feature only for Milestone #2 and only after everything else works.**

Comment your code so that someone else (as well as you) can understand it.

Milestone #1 Array as Index Structure

The big question is how to implement the index structure. It must support `find`, `insert`, and `printInOrder` operations.

The first milestone of the assignment is to create the program `index` using an **ordered array** (ordered by word) of index entries to implement the index structure. You may use `std::vector` to implement this ordered array. Finding an entry in an ordered array can be done using binary or linear search - you may implement either. Inserting a new entry (after you find where it goes) is easy, but you should allow your array to increase in size if necessary (`std::vector` does this automatically).

The goal of this milestone is to get the file reading and word processing parts of the assignment done, using a very simple data structure to handle the indexing.

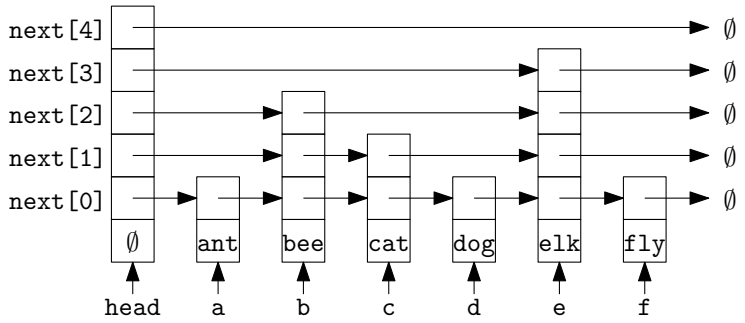
Milestone #1 is due September 28.

See the Deliverables and `handin` sections at the end of this assignment for instructions on how to submit your program. The name of this part of the assignment is `proj1A`.

Milestone #2 Skip List as Index Structure

The second part of the assignment is to implement the index structure using a **skip list**. A skip list is like an ordered singly-linked list, but each node in a skip list contains an array of next pointers, `next[]`. The number of next pointers that a node contains is called the *height* of the node. Different nodes can have different heights but every node has height at least one. The `next[0]` pointers form a regular, ordered singly-linked list of all the nodes in the list. So printing out the nodes in order is easy; just follow the `next[0]` pointers.

The interesting (and powerful) part about skip lists is what the other next pointers permit. For a node with height h, its `next[`i`]` pointer (for all i from 0 to $h-1$) points to the first node that follows it (in the ordering) that has height at least $i+1$. These "higher-up" next pointers allow a program to traverse the skip list more quickly than a regular linked list because they skip over many nodes in one step. Here's a figure to help make it clear:



This skip list stores the words `ant`, `bee`, `cat`, `dog`, `elk`, and `fly`. The first (leftmost) node in the skip list is a dummy node (`head` points to it and its data is ∅ in the picture). The dummy node always has the maximum height of any node in the skip list. As you can see, its `next[3]` pointer is `e` (the address of the `elk` node) because the `elk` node is the first node that has height ≥ 4. (Its height is 4.) Similarly, `bee`'s `next[2]`, `cat`'s `next[1]`, and `dog`'s `next[0]` pointers are all equal to `e`. (The figure shows these pointers pointing to different parts of the `elk` node, but they all point to the same object, i.e., `head->next[3]` = `b->next[2]` = `c->next[1]` = `d->next[0]` = `e`.)

Find

Finding a word `word` in this skip list is pretty easy:

1. Let `h` be the height minus one of the `head` node and let `node` = `head`.
2. While `node->next[h]` is not `NULL` and points to something smaller than `word`, set `node = node->next[h]`.
3. If `node->next[h]` points to `word`, great! You've found it. Return it.
4. Otherwise, decrease `h` by one and if `h` ≥ 0, go back to step 2.

   (Note: You probably want to use a for-loop that decrements `h` rather than a `goto`.)
5. `word` doesn't exist in the skip list.

Try finding `elk` in the figure using this algorithm. You get to it after examining only two next pointers.

Note that your skip list for this assignment will hold index entries not simply words.

Node Height

Where does the height of a node come from? It is chosen when the node is inserted into the skip list. Ideally, finding a word in a skip list should work like binary search: compare with the middle of the list, then recursively search in the first or second half. This implies that the middle node should have the maximum height, and the middle nodes of the first and second halves should have the maximum height of the remaining nodes, etc. Unfortunately, we don't know if a node will be the middle node when we insert it. So instead we choose the height randomly, as follows:

```
for( i=1; i<MAX_HEIGHT; ++i ) {
  if( randBit() == 1 ) break;
}
height = i;
```

On average, half the nodes will have height 1, a quarter of the nodes will have height 2, and, in general, a $1/2^k$ fraction will have height k. In my code, I've limited the height to `MAX_HEIGHT` (which equals 16) to make the code simpler. You may do that as well. It means that you can statically allocate an array `next[]` of size `MAX_HEIGHT` when you create a node, rather than dynamically allocating it, and you can set the height of the dummy node to be `MAX_HEIGHT`.

The code for `randBit()` is:

```
#include <ctime>                // for time()
#include <cstdlib>              // for rand(), srand(), and RAND_MAX

int randBit(void) {             // return a "random" bit
  static int bitsUpperBd=0;
  static int bits;              // store bits returned by rand()
  if( bitsUpperBd == 0 ) {      // refresh store when empty
    bitsUpperBd = RAND_MAX;
```

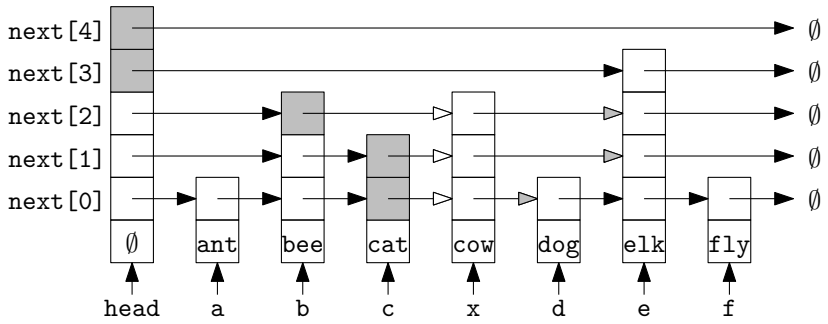
```
    bits = rand();
  }
  int b = bits & 1;
  bits >>= 1;
  bitsUpperBd >>= 1;
  return b;
}
```

You should call `srand(time(0))` when you create your skip list to initialize (seed) the random number generator.

Insert

Inserting a word `word` uses almost the same code as `find`. The only difference is that you need to update some of the next pointers in the skip list to point to the new node for `word`.

Which pointers need to be updated? Here's the figure above after we insert `cow`, where the height of the new `cow` node is 3.



The pointers that were updated are shown as ⟶▷. The next pointers for the new node (shown as ⟶▶) are simply copies of the old values of the corresponding updated next pointers.

The gray boxes ▒ are the next pointers visited during `find(cow)` (on the skip list before `cow` is inserted) that cause the while-loop in step 3 (of the find code) to terminate. One way to write the `insert` code is to make the `find` code record (in a global array of size `MAX_HEIGHT`) the node pointers that cause these terminations for each value of `h`. So add this step to the `find` code:

2.5 `update[h] = node`

After executing `find(cow)` (on the skip list before `cow` is inserted), the `update` array is `[c, c, b, head, head]`. Now to insert the `cow` node (pointed to by `x`), we just:

```
for( int h = 0; h < x->height; ++h ) {
  x->next[h] = update[h]->next[h]; //initialize x's next pointers
  update[h]->next[h] = x;          //update next pointers that change
}
```

Milestone #2 is due October 10.

See the Deliverables and `handin` sections at the end of this assignment for instructions on how to submit your program. The name of this part of the assignment is `proj1B`.

Deliverables (to be submitted online, not on paper)

For each milestone (`proj1A` and `proj1B`), you should submit, using `handin`:

- Your source code (.cpp or .C and .h files).
- A `Makefile` so that typing `make` in your handin directory on an undergrad Unix server will produce an executable called `index`.

  If you produce your `Makefile` on a Windows machine, you might have to remove the extra carriage-returns at the end of each line of the `Makefile`. You can use
  `tr -d '\r' < Makefile.old > Makefile.new` to do it.
- A `README` file containing:
  1. Your name or, if a team submission, both your names.
  2. Approximately how long the project took you to complete.
  3. Acknowledgment of any assistance you received from anyone but your team members, the 221 staff, or the 221 textbooks, but please cite code quoted or adapted directly from the texts (per the course's Collaboration policy).
  4. A list of the files in your submission with a brief description of each file.
  5. Any special instructions for the marker.
- DO NOT HAND IN: .o files, executables, core dumps, irrelevant stuff.

How to `handin` this assignment

1. Create a directory called `~/cs221/proj1A` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj1A`).
2. Move or copy all of the files that you wish to hand in, to the `proj1A` directory that you created in Step 1.
3. Before the deadline, hand in your directory electronically, as follows: `handin cs221 proj1A`

   Note that you will receive a set of confirmation messages. If you don't get any kind of an acknowledgment, then something went wrong. Please re-read the instructions and try again.
4. You can overwrite an earlier submission by including the `-o` flag, and re-submitting, as follows: `handin -o cs221 proj1A`

   You can hand in your files electronically as many times as you want, up to the deadline.
5. Additional instructions about `handin`, if you need them, are listed in the man pages (type: `man handin`). At any time, you can see what files you have already handed in (and their sizes) by typing the command: `handin -c cs221 proj1A`

   If your files have zero bytes, then something went wrong and you should run the original `handin` command again.