



Doubly Linked List

Joel Castellanos

Due: Monday, 2/23/2009 at midnight in WebCT

Problem Specification

Write a C program that reads a text data file, creates a doubly linked list from the data, and displays the list structure with the following requirements:

1. Your source code must consist of the single file: `linkedList.c`
2. You must compile on the `cs.unm.edu` Linux machines with `gcc` version 4.2.3 for `x86_64-linux-gnu`.
3. Your program must open and read the file given as a command line argument.
4. Each correct record of the data file will consist of a sequence of ASCII characters with codes [32,126] ending with the ASCII line feed character [10].
5. Your program must process two types of records:
 - a. `name:quote`
 - b. `:remove:quoteNum`

Where *name* is an author's name that will not contain the ':' character.

6. Your program must declare and maintain doubly linked list data structure sorted alphabetically by author, and for the same author, by quote:

```
#define MAX_CHARACTERS 16384
#define MAX_QUOTES 1024
char data[MAX_CHARACTERS]; //Storage for input data.

//dataEnd is the index into data[] of the first unused location.
int dataEnd = 0;

//dataQuote[i] is an index into data[] of the start of a quote.
int dataQuote [MAX_QUOTES];

//dataAuthor[i] is an index into data[] of the start of an author.
int dataAuthor [MAX_QUOTES];

//quoteNext[i] and quotePre[i] are the indexes into dataQuote[]
// of the alphabetically next and previous quote.
// Additionally, if i is an unused location, then quoteNext[i] is
// the index of the next unused location.
int quoteNext[MAX_QUOTES], quotePre[MAX_QUOTES];

//quoteStart is the index into dataQuote[] of the alphabetically
// first author/quote pair.
int quoteStart = -1;

//quoteFree is an index into dataQuote[], dataAuthor[],
// quoteNext[], and quotePre[] of the next to be used free spot.
int quoteFree = 0;
```

7. Each input data record of the form `name:quote`, must be placed, in the order that the characters and records are read, into the single dimensional, `char` array `data`.

8. Within the single dimensional array, the delimiting ':' character must be replaced with a '\0' character, and a '\0' character must follow the last character in *quote*.
9. When each record is read, the doubly linked list must maintained so that the above data structure remains sorted alphabetically by author, and for the same author, by quotation.
10. When a data file record begins with the character ':', the record should consist of the sequence of characters: " :remove:" followed by characters representing a decimal integer. You may use the `atoi()` library function from `#include <stdlib.h>` to convert the string to an integer. The integer after the key word " :remove:" specifies a previously added quotation/author pair that need to be removed from the linked list. For example, " :remove:15" specifies the removal of the 15th quotation/author pair by starting the count of 1 at the *alphabetically* first pair and progressing through the pairs in alphabetical order.
11. If any data errors are encountered within the data file, then an error message should be printed and your program should exit. The exact error messages printed must match those given in the test cases.
12. Finally, when all records of the input data file have been processed without error, your program must print a table showing the data structure of the doubly linked list.

There is a great description of linked lists and doubly linked lists in Wikipedia: http://en.wikipedia.org/wiki/Linked_list

Example 1: A Small, Already Sorted Data File

Data file: `LinkedList0.txt`

```
Fonzie:Cool
Homer:D'OH!
Pooh:Oh Bother
```

After reading this file, the values of the array `data[]` must be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
F	o	n	z	i	e	\0	C	o	o	l	\0	H	o	m	e	r	\0	D	'
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
O	H	!	\0	P	o	o	h	\0	O	h		B	o	t	h	e	r	\0	

The output for this amazing set of quotations must be:

index	Next index	Previous index	Author Data	Quote Data	Author	Quote
0	1	-1	0	7	[Fonzie]	[Cool]
1	2	0	12	18	[Homer]	[D'OH!]
2	-1	1	24	29	[Pooh]	[Oh Bo]

The full author name should be printed and the first 5 characters of the quotation. The data in this table was printed using the format string:

```
"%5d %5d %7d %8d %8d [%20s] [%c%c%c%c%c]\n"
```

For debugging, it is a good idea to print this table after every insert. Thus, do it in a function.

Example 2: A Small, Unsorted Data File

Data file: LinkedList1.txt

```
Homer:D'OH!
Pooh:Oh Bother
Fonzie:Cool
```

To understand the process of creating and maintaining the doubly linked list, the steps below walk through a simplified symbolic representation that shows only three parts for each record: the record index (**i**), the author's name (**data[dataAuthor[i]]**), and the index of the next sorted record (**quoteNext[i]**). In order to represent the entire structure, in this example we use **MAX_QUOTES = 4**. Figure 2-1 shows the initial state of the structure: There are no items in the list, therefore start points to an invalid index (-1). Since there are no items in the list, all 4 list items are free. Thus the free list starts at index 0 which points to 1, which points to 2, which points to 3 which points to -1 (end of list).

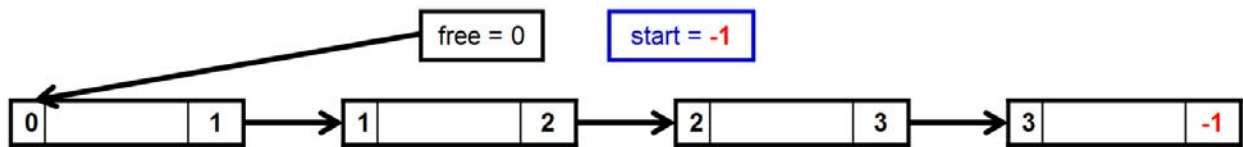


Figure 2-a

When the first data record is read, the links to that data are placed in the linked list location pointed to by **free**, which is index 0. Since the Homer quote is the first item in the list, it must be placed at the start of the list. The procedure for inserting an item at the start of the list is:

1. **int newIdx = free** //Every new record is always taken from the front of the linked list of free items.
2. **free = quoteNext[free]** //This removes the used index from the front of the free list.
3. **dataAuthor[newIdx] =** // The index into **data[]** of the first character of the author of the new record. In this case that is "Homer" with a starting index of 0.
4. **quoteNext[newIdx] = start** //This makes the new item point to whatever **start** had been pointing to. In this case, that is just -1 or end of list. Notice that this breaks the link in figure 2-a from index 0 to index 1. This is necessary because index 1 is still on the free list and index 0 is the end of the used list. Also notice that it was necessary to assign **free = quoteNext[free]** before assigning
5. **start = newIdx** // This makes the new record at the start of the list.

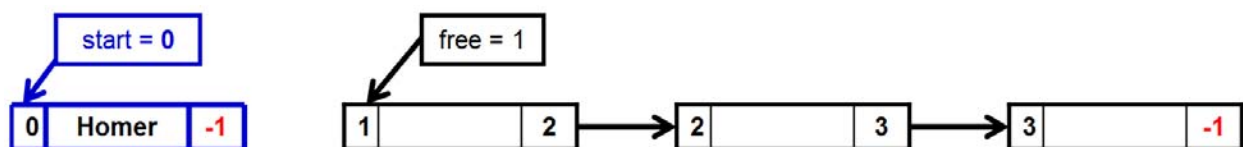


Figure 2-b

The next record is the famous quotation by Pooh. Regardless of where this second author falls alphabetically relative to the first record's author, each new record is always taken from the front of the linked list of free items.

1. `int newIdx = free`
2. `free = quoteNext[free]`
3. `dataAuthor[newIdx] = //` The index into `data[]` of the first character of the author of the new record. In this case that is "Pooh" with a starting index of 12.
4. Now is when the sorting happens. Regardless of the alphabetical order of the new record, the actual character data in the `data[]` array is not moved. The indexes of that data in the `dataAuthor[]` and `dataQuote[]` arrays also is not ever moved. What changes is the order of the linked list from start to `quoteNext[i] = -1`. The simplest way to find where `newItem` gets inserted in the linked list is to start at start, follow each `quoteNext[i]` until one of three cases occurs:
 - a. An author is found that is alphabetically after `data[dataAuthor[newIdx]]`,
 - b. An author is found that is the same as `data[dataAuthor[newIdx]]` (in which case you need to look at the alphabetical order of the quotation text.
 - c. The end of the linked list of used items is reached (`quoteNext[i] = -1`).

After the search is done, let the variable `beforeMeIdx` be the index of the record that comes alphabetically after the new record, and let `afterMeIdx` be the index of the record that points to `beforeMeIdx`. In this case, Pooh belongs at the end of the list. Therefore, `beforeMeIdx = -1`.

5. `quoteNext[newIdx] = beforeMeIdx`
6. `quoteNext[afterMeIdx] = newIdx`

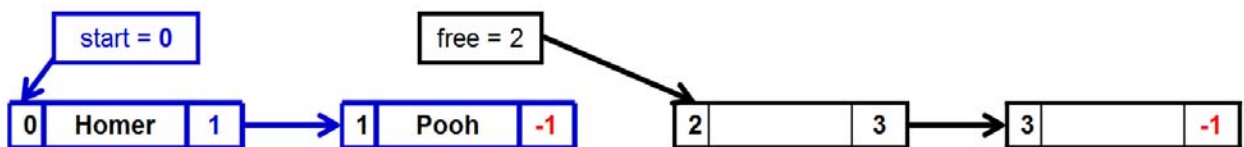


Figure 2-c

The next and final record is, as always, placed at the end of the data array and in the linked list structure with `newIdx = free`. This record's author is alphabetically before Homer and therefore needs to go at the start of the linked list: `quoteNext[newIdx]` needs to be set to the current start and start needs to be set to `newIdx`:

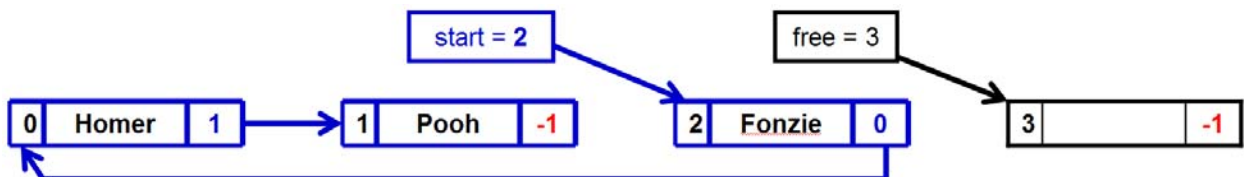


Figure 2-d

Note that the above analysis only shows the forward links and inly shows the string at `&data[dataAuthor[i]]` the string `&data[dataQuote[i]]`.

After reading this file, the values of the array `data[]` must be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H	o	m	e	r	\0	D	'	O	H	!	\0	P	o	o	h	\0	O	h	
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
B	o	t	h	e	r	\0	F	o	n	z	i	e	\0	C	o	o	l	\0	

If the doubly linked list data structure is printed after each record is read then the results would be:

Read “Homer :D'OH!” and insert at start of empty list.

	Next	Previous	Author	Quote		Author	Quote
index	index	index	Data	Data			
0	-1	-1	0	6	[Homer]	[D'OH!]

Read “Pooh :Oh Bother” and insert at end of list.

	Next	Previous	Author	Quote		Author	Quote
index	index	index	Data	Data			
0	1	-1	0	6	[Homer]	[D'OH!]
1	-1	0	12	17	[Pooh]	[Oh Bo]

Read “Fonzie :Cool” and insert at start of the list. This is the final state after all records have been read. This is the required output.

	Next	Previous	Author	Quote		Author	Quote
index	index	index	Data	Data			
2	0	-1	27	34	[Fonzie]	[Cool]
0	1	2	0	6	[Homer]	[D'OH!]
1	-1	0	12	17	[Pooh]	[Oh Bo]

Example 3: A Slightly Larger, Unsorted Data File with :remove:

Data file: `LinkedList2.txt`

```

Thomas Jefferson:I do not take a single newspaper, nor read one a
month, and I feel myself infinitely the happier for it.
John Adams:Great is the guilt of an unnecessary war.
George Washington:Few men have virtue to withstand the highest bidder.
Jimmy Carter:We will not learn how to live together in peace by
killing each other's children.
:remove:2
:remove:2
Thomas Jefferson:I cannot live without books.
Franklin Roosevelt:A nation that destroys it's soils destroys itself.
Forests are the lungs of our land, purifying the air and giving fresh
strength to our people.
Theodore Roosevelt:I am a part of everything that I have read.
    
```

After reading the first 4 records, the data structure is:

index	Next index	Previous index	Author Data	Quote Data	Author	Quote
2	3	-1	175	193	George Washington	[Few m]
3	1	2	246	259	Jimmy Carter	[We wi]
1	0	3	122	133	John Adams	[Great]
0	-1	1	0	17	Thomas Jefferson	[I do]

The next record read is “`:remove:2`”. This removes the alphabetically second quote, which is the Jimmy Carter quote with index = 3. The second “`:remove:2`” takes the John Adams quote with index = 1.

After the two removes, a second Thomas Jefferson quote is added. Since the last remove freed index 1, this new quote is added in index 1. When the new quote is inserted in the linked list, it must be inserted between the George Washington and the first Jefferson quote (since it comes alphabetically before the first Jefferson quote). The final record to be added reuses the index of the second to last deleted record. The third record to be added after the two removes, must go in index 4 because the two deleted records have already been reused. The final output is thus:

index	Next index	Previous index	Author Data	Quote Data	Author	Quote
3	2	-1	388	407	Franklin Roosevelt	[A nat]
2	4	3	175	193	George Washington	[Few m]
4	1	2	552	571	Theodore Roosevelt	[I am]
1	0	4	341	358	Thomas Jefferson	[I can]
0	-1	1	0	17	Thomas Jefferson	[I do]

Grading

+10x	Where x is the number of the 6 <code>LinkedList<n>.txt</code> data files passed (<code>LinkedList0.txt</code> does not count).
+2x	Where x is the number of error cases passes in <code>LinkedListErrorScript.txt</code>
+5	Comments are sufficient, accurate, and informative
+10	Algorithm is well organized (no spaghetti code)
+13	Adheres to the UNM CS-241 coding standards
-10x	Where x is the number of compiler warning messages.

Extra Credit: Clean up after yourself! (20 points)

Implement a lazy defragmentation algorithm. A *lazy* algorithm is one that does not execute until it is needed. This is advantageous in situations where the problem can often, but not always, be solved without applying the algorithm. For example, in the linked list of quotations, whenever a quote is deleted, space in `data[]` is never reclaimed. This is *sometimes* wasteful. In particular, it is wasteful in the case where a data file fails to be processed because it runs out of memory when the program space actually contained enough memory to have succeeded. Reclaiming deleted nodes of a linked list adds only a small, constant time to each deletion. Therefore, it is worth doing, even if not needed. However, reclaiming the space in `data[]` requires moving, on average, half of the values in the array. This is an expensive order- n process. Another situation where a lazy algorithm is advantageous is when the resources required to maintain a structure at every step is not much less than the resources required to clean up lots of steps all at once.

The extra credit implementation should always add new data to the end of the `data[]` array just as does the basic implementation; however, if it occurs that `data[]` runs to the end before the end of file is reached, then your program should make a single pass through the entire array removing all dead space. Then, if at least one byte was recovered, your program should continue where it left off.