CS/240/Lab/6

In lab 4, you have implemented a function that deletes the Least Recently Used (LRU) node from the list. As we have shown during the lab sessions, we iterate on the linked list trying to find the word with smallest line, if we have a tie, we try to break the tie based on the word count and finally compare the words to get a final decision on which node should be removed.

Another alternative would be to sort the list and then remove the first node (head) as it will represent the smallest node. As the book and lectures have shown, sorting algorithms make comparisons and exchange items until the items are in order. Because we may want to use different rules to sort a list in different ways, sorting functions make an ideal place to use function pointers. No change needs to be made to the sorting algorithm to change the sorting rules. Instead we can just call the sorting algorithm with a pointer to a different comparison function.

As you are familiar now with function pointers, you will exercise this in writing a *bubble-sort* using function pointers. In this lab you will be required to write:

- comparator functions
- swap function
- a bubble sort implementation that works on a doubly linked list with the same node struct we had in lab 4

You should not worry about linked list implementation. You will be using the library we provide and use the functions described in `list.h` to handle the list manipulation.

A: Swappers

Before we get into sorting, let's start by writing a simple function that given a list and two nodes, it will swap the nodes positions in the inked list. Example: given a list with the following words

```
{"1", "5", "3", "4", "2", "6"}
```

and we want to swap `"2"` with `"5"`, then the list should become like :

```
{"1", "2", "3", "4", "5", "6"}
```

`void swap (struct lnode** head, struct lnode* n1, struct lnode* n2)`

Description: In the linked list with `*head` as head pointer, swap the nodes with `n1` with `n2` as node pointers. If `n1` and `n2` are pointing to the same node, then the list should not be affected. The swap should not access the `lnode` fields directly. Instead it has to use the interface provided in the header file `list.h`. The swap should not use any dynamic allocation to swap the two nodes. In other word calling `malloc`, `calloc`..etc is not allowed.

B: Sorting

In this part you will be implementing a bubble sort on a doubly linked list. For reference, you can find below a pseudo-code implementation for bubble sort on an array of elements. You do not have to implement the same pseudo-code in `C`. You can have different implementation as long as you follow the conditions mentioned in the `sort` description below.

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
       newn = 0
       for i = 1 to n-1 inclusive do
          if A[i-1] > A[i] then
             swap(A[i-1], A[i])
             newn = i
          end if
       end for
       n = newn
    until n = 0
end procedure
```

Given the above pseudo-code you need to make it work for `doubly-linked` lists. You can assume that the struct `lnode` has three main fields: *count, line, and word* that can be used for sorting the list. The `list.h` lists all the getters that you may need to access node values. You cannot access fields without calling these getters. The `sort` header is as follows:

```
void sort (struct lnode** head,
           void (*swapPtr) (struct lnode** head, struct lnode* n1, struct lnode* n2),
           int (*comparePtr) (void* v1, void* v2))
```

Description: The sort implementation for bubble sort. It takes a linked list to be sorted by passing `*head` as head pointer. The output of that function should be the same doubly linked list sorted in ascending order such that the header node will be the smallest value in the list.
To decide how comparison should be done between two given nodes, the third argument will be used. The latter is a function pointer with the given signature and return negative value if the data of `v1` is smaller than `v2`, 0 if they are equal and positive value if the data of `v1` is larger than the data in `v2`.
`sort` will also get the swap function as a parameter to swap two nodes if necessary. The content of `head` will be modified to be the smallest element in the list. Similar to `swap`, `sort` should not be doing any kind of memory allocation.

C: Comparators

You will implement three different comparators. Each one takes two `lnode` pointers and returns an integer value indicating the result of the comparison.

```
int wordCmp (struct lnode* n1, struct lnode* n2)
```

Description: Returns an integral value indicating the relationship between the strings in two nodes: A zero value indicates that both strings are equal. A value greater than zero indicates that the first character that does not match has a greater value in n1-¿word than in n2-¿word; And a value less than zero indicates the opposite. You can use `strcmp` to implement this function.

`int lineCmp (struct lnode* n1, struct lnode* n2)`

Description: Returns an integral value indicating the relationship between the lines values in two nodes: A zero value indicates that both values are equal. A value greater than zero indicates that value in n1-¿line is greater than in n2-¿line; And a value less than zero indicates the opposite.

`int countCmp (struct lnode* n1, struct lnode* n2)`

Description: Returns an integral value indicating the relationship between the count values in two nodes: A zero value indicates that both values are equal. A value greater than zero indicates that the value in n1-¿count is greater than in n2-¿count; And a value less than zero indicates the opposite.

List

The list implementation is the same one we had previously in lab4. For convenience we added three more functions listed here in details. The first function `evictNode` can be called to remove a list from a list without destroying its contents. An evicted node can be added back into the list using `insertNode`. Finally, We have added `nodeGetPrev` to return the previous node. The rest of the available function are listed in `list.h` for your reference and were described in details in lab4 handouts.

You do not need to implement any of these new functions. They can be used by linking against the list reference implementation provided in the piazza post.

`void evictNode (struct lnode** head, struct lnode* node)`

Description: Evict the specified node from the list, but does not free the memory used by it.

`void insertNode (struct lnode** head, struct lnode* prevNode, struct lnode* insertingNode)`

Description: Inserts the given node (`insertingNode`) *after* the node `prevNode` into the list with head pointer `head`. If the `prevNode` is `NULL`, then the node `insertingNode` is inserted at the front of the list.

`struct lnode *nodeGetPrev(struct lnode *node)`

Description: Simply returns the prev node in the list, or NULL if there are no previous nodes.

Requirements

Implement the requirement listed in Sections A, B and C in a source file called `sort.c`. Your code should be using the list reference library. You must not use any functions that is not listed in the `list.h` file to deal with the linked list.

Keep in mind that `sort.c` does not have a main function and does not deal with input. It has only the implementations for the functions listed in sections A, B and C. In order to test your code, you will need to create a separate `test.c` file that will create a doubly linked list using `pushNode` and `newNode`, then you call the `sort` function implemented in `sort.c` and give it one of the comparators `(wordCmp, lineCmp or countCmp)` and the `swap` function implemented in the same file. For convenience, you can use `printList` available in the reference library to see the list after calling the sort function. You should be able to verify that your list is sorted based on the comparator you use each time you call `sort`.

We are not providing commands to compile/run your code. It is similar to what you had earlier in lab4. You are required to come up with the commands by yourself as part of the assignment as we will hold answering questions about compilation/running commands. Your `sort.c` should compile successfully with the list library provided.

Finally, You may use any function in the C standard library except any call that may produce dynamic memory allocation. The list of **forbidden functions are listed** below:

- malloc
- realloc
- calloc
- free

Turning in

Go to the submission web page, under "currently open projects" will be listed "lab 6 sort" . Click "lab 6 sort" , use the file selector to choose your `sort.c` file, then click "submit".

Lab is due Wednesday, October 10th before midnight. No late labs accepted.

Grading criteria

- source file is named `sort.c`
- code compiles
- code does not override functions provided in `list` library
- sort is generic and takes the swap and the comparator functions as parameters.
- sort generates the correct list given the comparator and the swapper
- no dynamic allocation is used in your code
- nodes in a linked list are not destroyed after calling your sort function
- no main function in your submitted code
- swap works as described above
- all the comparator functions are implemented as described above.