Due Monday 02/27/2006 11:59 pm

1. Sorted Singly-Linked List

A linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential datatype because it contains a pointer or link to another data of the same type.

The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node. A separate pointer, usually called head points to the first node in the linked list. The head points to NULL in case the list is empty.

Read Chapter 7.5 Pg. 340 – 347 of the textbook, to understand how linked lists are implemented.

You have to implement functions to do sorted insert and remove from a list whose nodes are defined as

```c
struct list {
    char name[20];    // The data that is being stored in the list
    int key;          // The sorting key ,the integer value based on which the list is stored in a sorted order
    struct list * link;  // Pointer to the next node in the list
};
```

*a. struct list * SortedInsert( struct list * head, int keyValue, char * data);*

You will have to implement a function

```c
struct list * SortedInsert( struct list * head, int keyValue, char * data);
```

The function should be passed the head of the list, the key value, and the data (any string). The function should create a new node (using malloc()) in the linked list, and fill in the name, key and link values, such that all the nodes are sorted in increasing order of key values. You should return the head pointer after completing the insert, this is required to handle cases where the head value changes during the insert process, i.e. when the node to be inserted, has to be placed at the beginning of the list.

Initially your head would point to NULL.

```c
struct list * head = NULL;
```

The link list would look like this

```c
head => NULL ( => is not a part of C , it's just used here to illustrate the linked list )
```
If `SortedInsert()` is invoked with ........
    head = SortedInsert(head, 3, “Andre”);
Assuming the new node is located at memory address 2322( some address returned by malloc()), after the SortedInsert(), the linked list should look like this

    head=>[“Andre”,3,NULL]
    2322
This is the first and only node in the list, so head points to it and the node's link points to NULL.
The pointer returned would hold the address 2322.

Now, if `SortedInsert()` is invoked with ........
    head = SortedInsert(head, 1,”Adam”);
Assuming the new node is located at memory address 2500, after the SortedInsert(), the linked list should look like this

    head=>[“Adam”,1,2322]=>[“Andre”,3,NULL]
    2500   2322
Node at 2500 is put before node at 2322 because it has a smaller key value.
The pointer returned would hold the address 2500.

Now, if `SortedInsert()` is invoked with ........
    head = SortedInsert(head, 2,”Alfred”);
Assuming the new node is located at memory address 6500, after the SortedInsert(), the linked list should look like this

    head=>[“Adam”,1,6500]=>[“Alfred”,2,2322]=>[“Andre”,3,NULL]
    2500   6500   2322
Node at 6500 is inserted in the middle of the other 2 nodes because, its key value is lesser than key value of node at 2322, and greater than that of node at 2500.
The pointer returned would hold the address 2500.

b. `struct list * Remove (struct list * head , int keyValue)`

You will have to implement a function `Remove()`, with parameters as shown above.
This function takes in the head value and the key value, and deletes( using free() ) the first node in the list whose key value matches the passed key value. You should return the head pointer after completing the `Remove()`. `Remove()` should do nothing if the passed head value is NULL, or if none of the nodes have a key value matching the passed key value. `Remove()` should return NULL, if the node deleted was the last in the list.

For the list given below

    head=>[“Adam”,1,6500]=>[“Alfred”,2,2322]=>[“Andre”,3,NULL]
    2500   6500   2322
head = Remove(head,2);
should make the list look like this

head=>[“Adam”,1,2322]=>[“Andre”,3,NULL]
    2500     2322
Node at 2500 was pointing to node at 6500, now it points to the node at 2322.
The pointer returned would hold the address 2500

Now if I call

head = Remove(head,1);

the linked list should look like this

head=>[“Andre”,3,NULL]
    2322
head was pointing to node at 2500, now it points to the node at 2322.
The pointer returned would hold the address 2322

c. void PrintList( struct list * head);

This function should take in the head value and print out the nodes in the linked list. For the following linked list

    head=>[“Adam”,1,6500]=>[“Alfred”,2,2322]=>[“Andre”,3,NULL]
    2500     6500     2322

on calling PrintList(head); the output should be

[Adam,1,6500]=>[Alfred,2,2322]=>[Andre,3,0]

if the linked list is empty , i.e. head=>NULL , then print

Linked list is empty.

Starting with a head pointing to NULL, run your program to SortedInsert/Remove the following

SortedInsert with Name= “Aconcagua” , key=20
SortedInsert with Name= “Everest” , key=10
SortedInsert with Name= “McKinley” , key=30
Remove with key=30
Remove with key =10
SortedInsert with Name= “Kilimanjaro” , key=40
SortedInsert with Name= “Elbrus” , key=50
Remove with key=50
SortedInsert with Name= “McKinley”, key=30
SortedInsert with Name= “Everest”, key=10

Use PrintList() to print the list after each SortedInsert() / Remove().

Your final PrintList() should print

[Everest,10,1000]=>[Aconcagua,20,2000]=>[McKinley,30,3000]=>[Kilimanjaro,40,0]

*Note*: The addresses shown here are some random values, your program should print the actual address stored in each node’s link

then, call....

Remove with key=50
Remove with key=40
Remove with key=10
Remove with key=20
Remove with key=30

call PrintList() after each Remove() shown above, the last PrintList() should print

Linked list is empty.

3. Complexity Calculation

Indicate the execution time of SortedInsert() and Remove(), using the $O()$ notation, in your .txt file. Refer to Lecture 16 for explanation of $O()$.

4. Bonus problem

Write a function

struct list * ReverseList(struct list * head);

which takes in the head pointer of a sorted linked list, and then returns a head pointer, to a linked list with the nodes sorted in the descending order of keys.

If the head pointer of the following sorted linked list

head =>[Everest,10,1000] =>[Aconcagua,20,2000] =>[McKinley,30,3000] =>[Kilimanjaro,40,0]

is passed to ReverseList(), by

head = ReverseList(head);
then the reversed list should look like

head => [Kilimanjaro, 40, 2000] => [McKinley, 30, 1000] => [Aconcagua, 20, 5000] => [Everest, 10, 0]

3000 2000 1000 5000

Note: Only the link pointer of the nodes and the head pointer have changed, the nodes still are in the same memory location.

Use appropriate SortedInsert() calls to generate the following linked list

head => [Everest, 10, 1000] => [Aconcagua, 20, 2000] => [McKinley, 30, 3000] => [Kilimanjaro, 40, 0]

5000 1000 2000 3000

and then reverse it. Print the reversed list using PrintList().

Indicate the execution time of ReverseList(), using the $O()$ notation, in your .txt file.

4. Turn in

Submission for these files will be similar to earlier assignments. You need to create a directory called hw6/. Put all the files listed above in that directory and run the /ecelib/bin/turnin command to submit your homework. You should turn in the following files:

linkedlist.c
linkedlist.txt
linkedlist.script

In case you do the bonus problem....

reverselist.c
reverselist.txt
reverselist.script