Erick Gonzalez Parada #178145

Andre Francois Duhamel Gutiérrez #177315

Alejandro González Díaz #178645

Activity #6 Simple Circular Lists

**Objective**

With this activity our goal is to domain every aspect of Circular lists. Most importantly to learn how to manage the nodes in a list and create efficient methods to manipulate them.

**Part 1)**

**Solution Design & Main steps to develop solution**

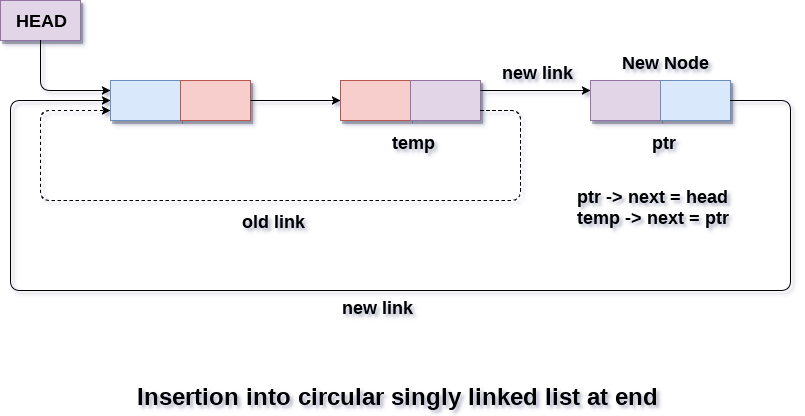
For the first part we decided to declare a typedef struct to have all the elements each node needs to have.

Texto

Descripción generada automáticamente

Img 1: struct node

Now to develop the methods it is useful to visualize what we need to implement and the parameters we need to pass to make it work. For the INSEND method, the functioning of the method is the next one:



And the parameters we need to pass are the next ones:

NODE \*INSEND(NODE \**head*, int *value*) {…}

For the INSORT method, the functioning of the method is the next one:

Diagrama

Descripción generada automáticamente

It is important to say that we as a team decided to make the INSORT method to sort the

values in an ascending way (1,3,8,10, 15,…). Although it is also possible to make it in and descending way. The parameters that we must pass to the function are:

NODE \*INSORT(NODE \**head*, int *value*) {…}

For the COPY methos, the functioning of the methos is very similar to a print list method. The only difference is that we need to do an assignation from a new head. It is important to say that we need to previously declare a new node and call the COPY function, which needs the following parameters:

NODE\* COPY(NODE\* *head*){…}

**Security aspects**

As for the Circular Lists it is possible to have the following security features:

Positive aspects

* Variable size arrays: As there is no fixed size of the data structure, we can add multiple values without major complication and with good efficiency.

Negative aspects

* Buffer overflow: the quantity of input data can exceed the allocated space. This can produce a memory overflow or the fact that there is no more available memory.
* NULL data: If there is no data in the list, the methods cannot work. The methods need to get data inserted to make the operations. Although this can be an exception this has been handled with a return statement that returns the newNode.

**Demonstration & analysis**

Firstly, lets test the INSEND method. The data that was inserted was:

Texto

Descripción generada automáticamente

There needs to be an assignation as the methods have a return value of NODE and to not loose the operations made by the methods. The output was:

Texto

Descripción generada automáticamente

If we try with the values:

Texto

Descripción generada automáticamente

We get the values:

Texto

Descripción generada automáticamente

As we can see the values are bein inserted at the end of the linked list. The code of the methos is:

*// Function to insert a new node at the end of the circular list*

NODE \*INSEND(NODE \**head*, int *value*) {

    NODE \*newNode = (NODE \*)malloc(sizeof(NODE));

    if (!newNode) {

        printf("Memory allocation failed.\n");

        return *head*;

    }

    newNode->val = *value*;

    newNode->next = newNode; *// Point to itself initially*

    if (*head* == NULL) {

*// If the list is empty, make the new node the head*

        return newNode;

    } else {

        NODE \*current = *head*;

        while (current->next != *head*) {

            current = current->next;

        }

*// Insert the new node at the end*

        current->next = newNode;

        newNode->next = *head*;

        return *head*;

    }

}

For the INSORT method the values are being sorted in an ascendant manner. The values inserted where the following:

Texto

Descripción generada automáticamente

The output was:

Texto

Descripción generada automáticamente

For the second test the values were:

Texto

Descripción generada automáticamente con confianza media

And the output was:

Texto

Descripción generada automáticamente

The method of INSORT was:

NODE \*INSORT(NODE\* *head*, int *value*) {

    NODE\* newNode = (NODE\*)malloc(sizeof(NODE));

    newNode->val = *value*;

    newNode->next = NULL;

*// Case 1: If the list is empty*

    if (*head* == NULL) {

        newNode->next = newNode;

        return newNode;

    }

*// Case 2: If the new node's value is less than the head's value*

    if (*value* < *head*->val) {

        NODE\* last = *head*;

        while (last->next != *head*) {

            last = last->next;

        }

        last->next = newNode;

        newNode->next = *head*;

        return newNode;

    }

*// Case 3: If the new node's value is greater than the head's value*

    NODE\* curr = *head*;

    while (curr->next != *head* && *value* > curr->next->val) {

        curr = curr->next;

    }

    newNode->next = curr->next;

    curr->next = newNode;

    return *head*;

}

For the copy method we just need to call the function and print it afterwards:

The values inserted where:

Texto

Descripción generada automáticamente

Before seeing the output, it is important to say that the COPY method calls the INSORT method to order the inserted values. The output was:

Texto

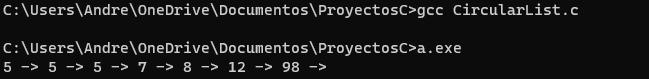
Descripción generada automáticamente

The second test had the values:

Texto

Descripción generada automáticamente

The output was:



The method used for COPY was:

NODE \*COPY(NODE \**head*) {

    if (*head* == NULL)

        return NULL;

    NODE \*temp = *head*;

    NODE \*new\_head = NULL;

    do {

        new\_head = INSORT(new\_head, temp->val);

        temp = temp->next;

    } while (temp != *head*);

    return new\_head;

}

**RESULTS**

As we can observe, the values were treated correctly depending on the function called. The exceptions and all cases where contemplated in an efficient and correct way as the tests show.

The entire C program is the following one:

#include <stdio.h>

#include <stdlib.h>

*// Define the NODE structure*

typedef struct node {

    int val;

    struct node \*next;

} NODE;

*// Function to insert a new node at the end of the circular list*

NODE \*INSEND(NODE \**head*, int *value*) {

    NODE \*newNode = (NODE \*)malloc(sizeof(NODE));

    if (!newNode) {

        printf("Memory allocation failed.\n");

        return *head*;

    }

    newNode->val = *value*;

    newNode->next = newNode; *// Point to itself initially*

    if (*head* == NULL) {

*// If the list is empty, make the new node the head*

        return newNode;

    } else {

        NODE \*current = *head*;

        while (current->next != *head*) {

            current = current->next;

        }

*// Insert the new node at the end*

        current->next = newNode;

        newNode->next = *head*;

        return *head*;

    }

}

*// Function to insert a new node in ascending order*

NODE \*INSORT(NODE\* *head*, int *value*) {

    NODE\* newNode = (NODE\*)malloc(sizeof(NODE));

    newNode->val = *value*;

    newNode->next = NULL;

*// Case 1: If the list is empty*

    if (*head* == NULL) {

        newNode->next = newNode;

        return newNode;

    }

*// Case 2: If the new node's value is less than the head's value*

    if (*value* < *head*->val) {

        NODE\* last = *head*;

        while (last->next != *head*) {

            last = last->next;

        }

        last->next = newNode;

        newNode->next = *head*;

        return newNode;

    }

*// Case 3: If the new node's value is greater than the head's value*

    NODE\* curr = *head*;

    while (curr->next != *head* && *value* > curr->next->val) {

        curr = curr->next;

    }

    newNode->next = curr->next;

    curr->next = newNode;

    return *head*;

}

NODE \*COPY(NODE \**head*) {

    if (*head* == NULL)

        return NULL;

    NODE \*temp = *head*;

    NODE \*new\_head = NULL;

    do {

        new\_head = INSORT(new\_head, temp->val);

        temp = temp->next;

    } while (temp != *head*);

    return new\_head;

}

*// Function to print the circular list*

void printCircularList(NODE \**head*) {

    if (*head* == NULL) {

        printf("Circular list is empty.\n");

        return;

    }

    NODE \*current = *head*;

    do {

        printf("*%d* -> ", current->val);

        current = current->next;

    } while (current != *head*);

    printf("\n");

}

int main() {

    NODE \*head = NULL;

*// Insert values in sorted order using the INSORT function*

    head = INSEND(head, 98);

    head = INSEND(head, 7);

    head = INSEND(head, 5);

    head = INSEND(head, 12);

    head = INSEND(head, 5);

    head = INSEND(head, 5);

    head = INSEND(head, 8);

    NODE \*newNode = COPY(head);

*// Print the circular list after sorting*

    printCircularList(newNode);

    return 0;

}

CONCLUSIONS

In conclusion the objective of the activity has been achieved, as the implementation of the methods was successful and helped us in the understanding of dynamic structures. Some final notes is that the insert at the end function can be used to make an undo/redo system, comparing it with the insert in order this one is just a great example of having spared data organized, lastly we have the copy list function that is also a great sample of coping spared data.