Javier Mantilla

G00329649

G00329649@gmit.ie

Advanced Procedural Programming Laboratory

Lab 8

Write a program to create and maintain a singly linked list of students. Each student is composed of the following structure and each student must have a random number:

• Student Name – String

• Student Age – Int

• Student ID – String

• Average Grade – float

• Email – String

• Mobile – String

The source code should include functions to implement the following operations:

• Add a student to the end of the list. (Unique Student ID must be verified)

• Add a student to the top of the list. (Unique Student ID must be verified)

• Display number of students in the the list.

• Delete a student from the list.

• Display all of the students in the list.

• Output all of the students in the list to file.

• Search the list for a student’s specified ID. The function should output the location of that element within the list.

• Delete a student from the start of the list

• Insert a new node at a defined location (The code should validate that this position exists and Unique Student ID must be verified)

• Delete a node at a defined location (The code should validate that this position exists)

DESIGN

Inputs:

1. head\_ptr (struct node\*)
2. option (int)
3. position (int)

Outputs:

1. head\_ptr (struct node\*)

CODE

/\* Javier Mantilla : Lab7 \*/

/\* Linked list enhanced \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

// Structures

struct node {

int value;

struct node \*next;

};

typedef struct node NODE; // I saw this on Deitel & Deitel's book

// Constants

#define INPUT\_MESSAGE "Enter data for this node: "

// Functions prototypes

// Use of 'const' keyword, because no way we allow modifications to the headPtr in these functions

void addNode(const \*headPtr);

int main(void);

void addFirst(const NODE \*\*headPtr);

void displayList(const NODE \*headPtr);

void deleteNode(NODE \*headPtr);

int size(const NODE \*headPtr);

void searchValue(const NODE \*headPtr);

void deleteAtStart(NODE \*\*headPtr);

void deleteAtPosition(NODE \*headPtr, int position);

void addNodeAtPosition(NODE \*headPtr, int position);

// Auxiliary functions prototypes

void printTitle(char \*title);

int captureInteger(char \*message);

int findIndex(int needle, NODE \*hayStack);

int main(void) {

// Variables

// Input variables

NODE \*head\_ptr;

int option;

int position;

// Output variables

// '\*head\_ptr' as declared above

// Initializing variables

head\_ptr = NULL;

do {

printTitle("Integer linked list");

puts("Please enter 1 to add a node at the end (normal add)");

puts("Please enter 2 to add a node at the start");

puts("Please enter 3 to display the linked list");

puts("Please enter 4 to delete a node at the end");

puts("Please enter 5 to display the size of linked list");

puts("Please enter 6 to search a value in the linked list");

puts("Please enter 7 to delete a node at the start the linked list");

puts("Please enter 8 to delete a node at a specified location");

puts("Please enter 9 to insert a node at a specified location");

printf("Please enter -1 to exit: ");

scanf("%d%\*c", &option);

switch (option) {

case 1:

printTitle("Adding a node");

if (head\_ptr != NULL)

addNode(head\_ptr);

else

addFirst(&head\_ptr);

break;

case 2:

printTitle("Adding at the beginnig of the linked list");

addFirst(&head\_ptr);

break;

case 3:

printTitle("Display linked list");

displayList(head\_ptr);

break;

case 4:

printTitle("Delete the last node in the linked list");

if (size(head\_ptr) < 2)

puts("Cannot perform deletion. Too few nodes. Consider adding more nodes.");

else

deleteNode(head\_ptr);

break;

case 5:

printTitle("Size of the linked list");

printf("The size of the list is %d nodes\n", size(head\_ptr));

break;

case 6:

printTitle("Search integer value in linked list");

searchValue(head\_ptr);

break;

case 7:

printTitle("Delete the first node in the linked list");

if (head\_ptr == NULL)

printf("The list is already empty\n");

else

deleteAtStart(&head\_ptr);

break;

case 8:

printTitle("Delete the last node at a specified location");

if (head\_ptr == NULL)

printf("The list is already empty\n");

else {

position = captureInteger("Please enter the position you wish to delete the node: ");

if (position <= 0)

puts("This position does not exist");

else if (position == 1)

deleteAtStart(&head\_ptr);

else if (position <= size(head\_ptr))

deleteAtPosition(head\_ptr, position);

else

puts("The value that you entered is greater than the length of the list");

}

break;

case 9:

printTitle("Insert a node at a specified location");

position = captureInteger("Please enter the position you wish to insert the node: ");

if (position <= 0)

puts("This position does not exist");

else if (position == 1)

addFirst(&head\_ptr);

else if (position <= size(head\_ptr))

addNodeAtPosition(head\_ptr, position);

else

puts("The value that you entered is greater than the length of the list");

break;

case -1:

break; // Do nothing just exit

default:

puts("Invalid option please try again");

// Clearing the input buffer so the next scanf will work as should do

while ((option = getchar()) != '\n' && option != EOF); // As seen in StackOverflow

}

} while (option != -1);

puts("\nThank you for using this integer linked list. Have a nice day.");

getchar();

return EXIT\_SUCCESS;

} // main

// Functions definitions

void addNode(const NODE \*headPtr) {

// Variables

// Input variables

NODE \*newNode;

// Auxiliary variables

NODE \*temp;

// Initializing variables

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

newNode->next = NULL;

temp = headPtr;

// Process

while (temp->next != NULL)

temp = temp->next;

newNode->value = captureInteger(INPUT\_MESSAGE);

temp->next = newNode;

} // addNode

void addFirst(const NODE \*\*headPtr) {

// Local variables

// Local input variables

NODE \*newNode;

// Initializing local variables

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

newNode->next = \*headPtr;

// Process

newNode->value = captureInteger(INPUT\_MESSAGE);

\*headPtr = newNode;

} // addFirst

void displayList(const NODE \*headPtr) {

// Local variables

// Local output variables

NODE \*temp;

// Local auxiliary variables

int i;

// Initializing local variables

temp = headPtr;

i = 0;

// Process

while (temp != NULL) {

printf("Node %d value: %d\n", ++i, temp->value);

temp = temp->next;

}

} // displayList

void deleteNode(NODE \*headPtr) {

// Local variables

// Local auxiliary variables

NODE \*temp;

NODE \*oldTemp;

// Initializing local variables

oldTemp = headPtr;

temp = headPtr->next;

// Process

while (temp->next != NULL) {

oldTemp = temp;

temp = temp->next;

}

printf("Deleted node value: %d\n", temp->value);

oldTemp->next = NULL;

free(temp);

} // deleteNode

int size(const NODE \*headPtr) {

// Local variables

// Local output variables

int i;

// Local auxiliary variables

NODE \*temp;

// Initializing local variables

temp = headPtr;

i = 0;

// Process

while (temp != NULL) {

++i;

temp = temp->next;

}

return i;

} // size

void searchValue(const NODE \*hayStack) {

// Local variables

// Local input variables

int needle;

// Local output variables

// 'needle' as defined above

int found;

// Process

needle = captureInteger("Please enter integer value to search: ");

// List traversing is actually done in the following function 'findIndex'

found = findIndex(needle, hayStack);

if (found)

printf("Value %d found in the linked list at position %d\n", needle, found);

else

puts("Value not found");

} // searchValue

void deleteAtStart(NODE \*\*headPtr) {

// Local auxiliary variables

NODE \*temp;

// Initializing local variables

temp = \*headPtr;

// Process

\*headPtr = temp->next;

printf("Deleted node value: %d\n", temp->value);

free(temp);

} // deleteAtStart

void deleteAtPosition(NODE \* headPtr, int position) {

// Local auxiliary variables

NODE \*temp;

NODE \*oldTemp;

int i;

// Initializing local variables

temp = headPtr->next;

oldTemp = headPtr;

// Process

// 'i' starts at '1' because the pointers 'temp' and 'oldTemp' were set to the first node of the list in the initialization above

for (i = 1; i < position - 1; i++) {

oldTemp = temp;

temp = temp->next;

}

oldTemp->next = temp->next;

printf("Deleted node value: %d\n", temp->value);

free(temp);

} //deleteAtPosition

void addNodeAtPosition(NODE \* headPtr, int position) {

// Local auxiliary variables

NODE \*temp;

NODE \*newNode;

int i;

// Initializing local variables

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

temp = headPtr;

// Process

for (i = 1; i < position - 1; i++)

temp = temp->next;

// printf("%d\n", temp->value);

newNode->value = captureInteger(INPUT\_MESSAGE);

newNode->next = temp->next;

temp->next = newNode;

} // addNodeAtPosition

// Auxiliary functions definitions

void printTitle(char \*title) {

// Local auxiliary variables

int titleLength;

int i;

// Initializing local variables

titleLength = strlen(title);

printf("\n%s\n", title);

for (i = 0; i < titleLength; i++)

printf("-");

puts("");

} // printTitle

int captureInteger(char \*message) {

// Local variables

// Local input variables

int integer;

// Local output variables

// 'integer' as declared above

printf(message);

scanf("%d", &integer);

return integer;

} // captureInteger

int findIndex(int needle, NODE \*hayStack) {

// Local variables

// Local output variables

int i;

// Local auxiliary variables

NODE \*temp;

// Initializing local variables

i = 0;

temp = hayStack;

// Process

while (temp != NULL) {

i++;

if (temp->value == needle)

return i;

temp = temp->next;

}

return 0;

} // findIndex

SCREENSHOTS

Figure 1 Deleting from the start with just one node in the list

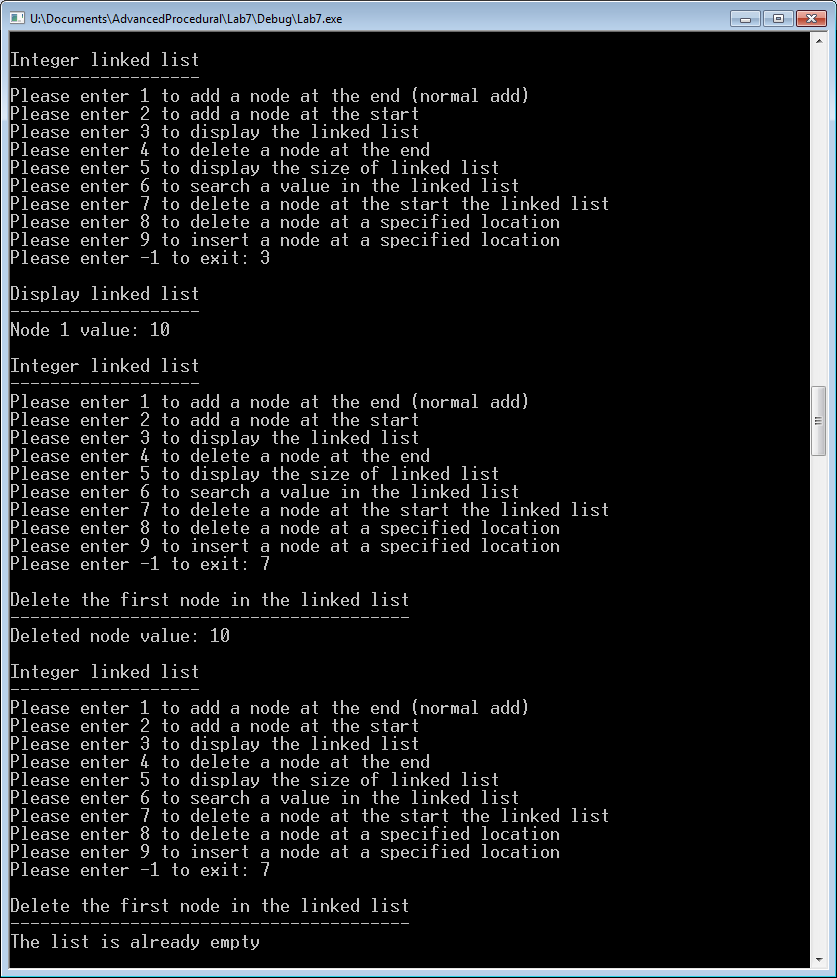


Figure 2 Trying to delete at an invalid location in the list

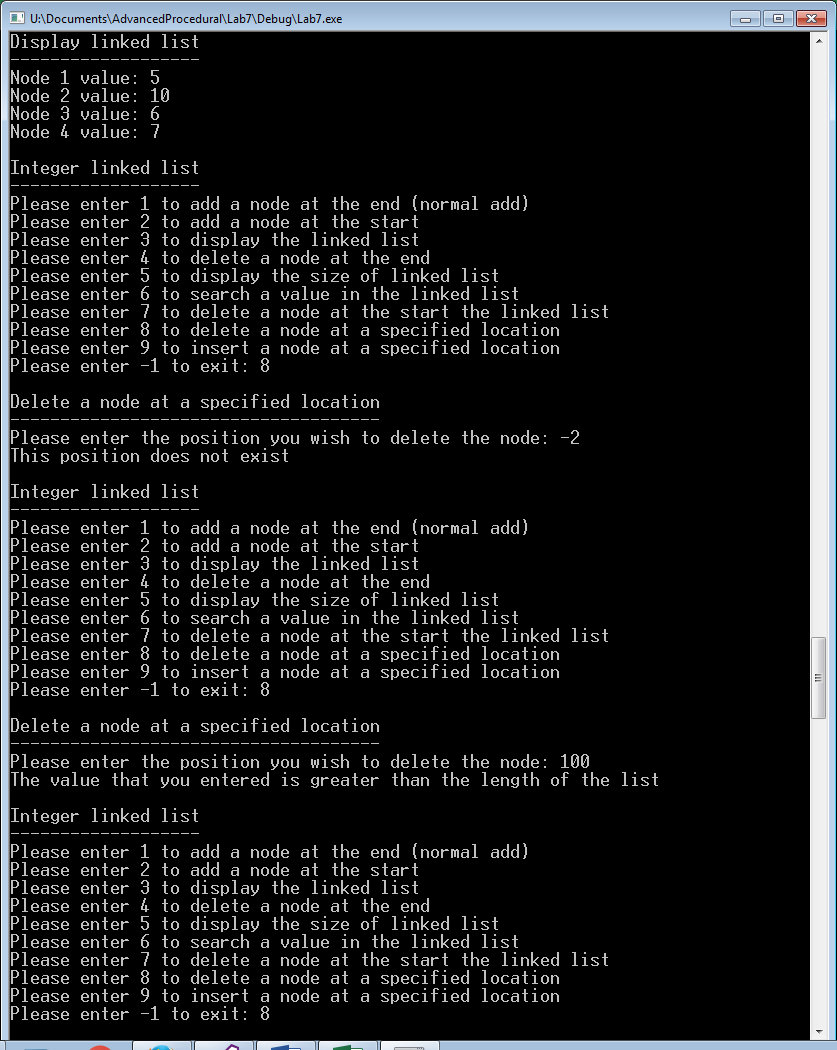


Figure 3 Deleting nodes at various locations in the list

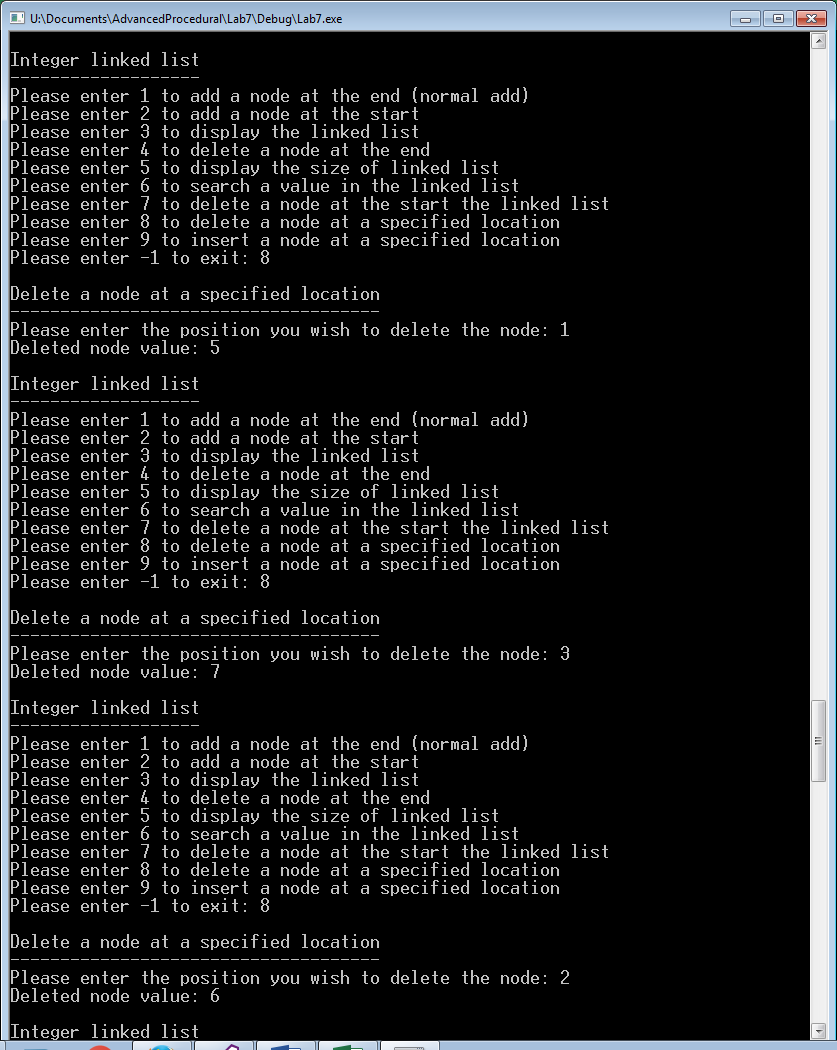


Figure 4 Trying to insert nodes into invalid locations in the list

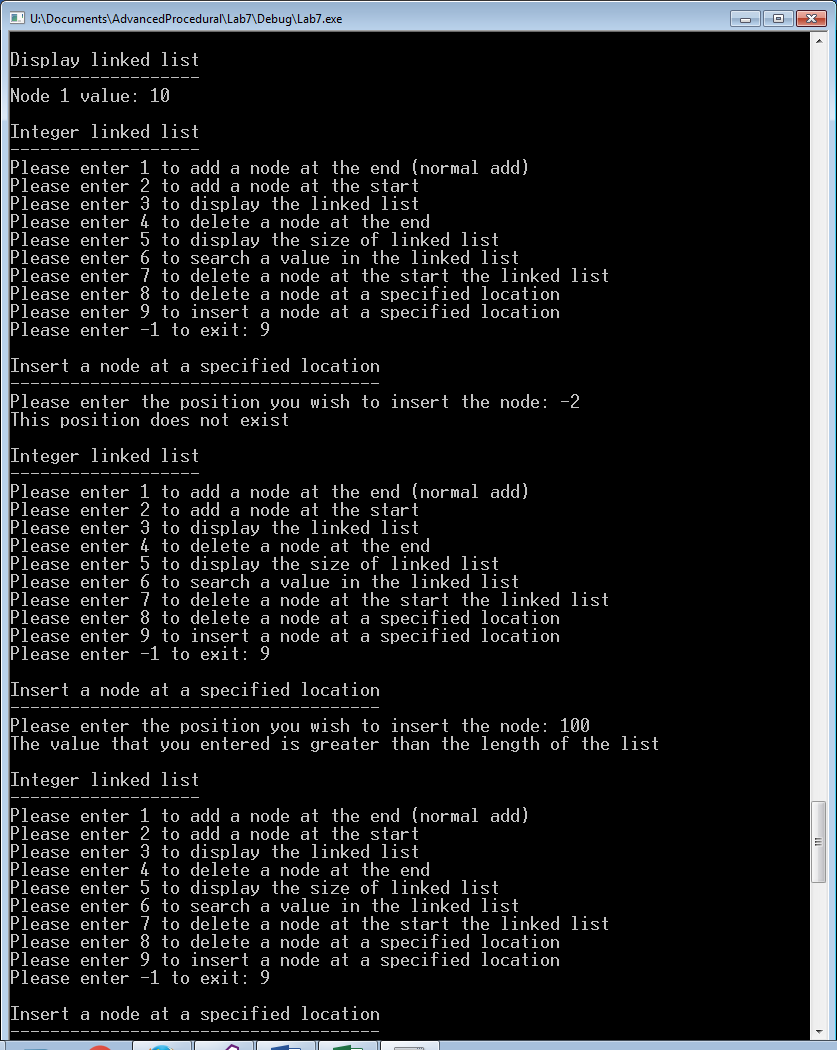


Figure 5 Inserting nodes at various locations in the list

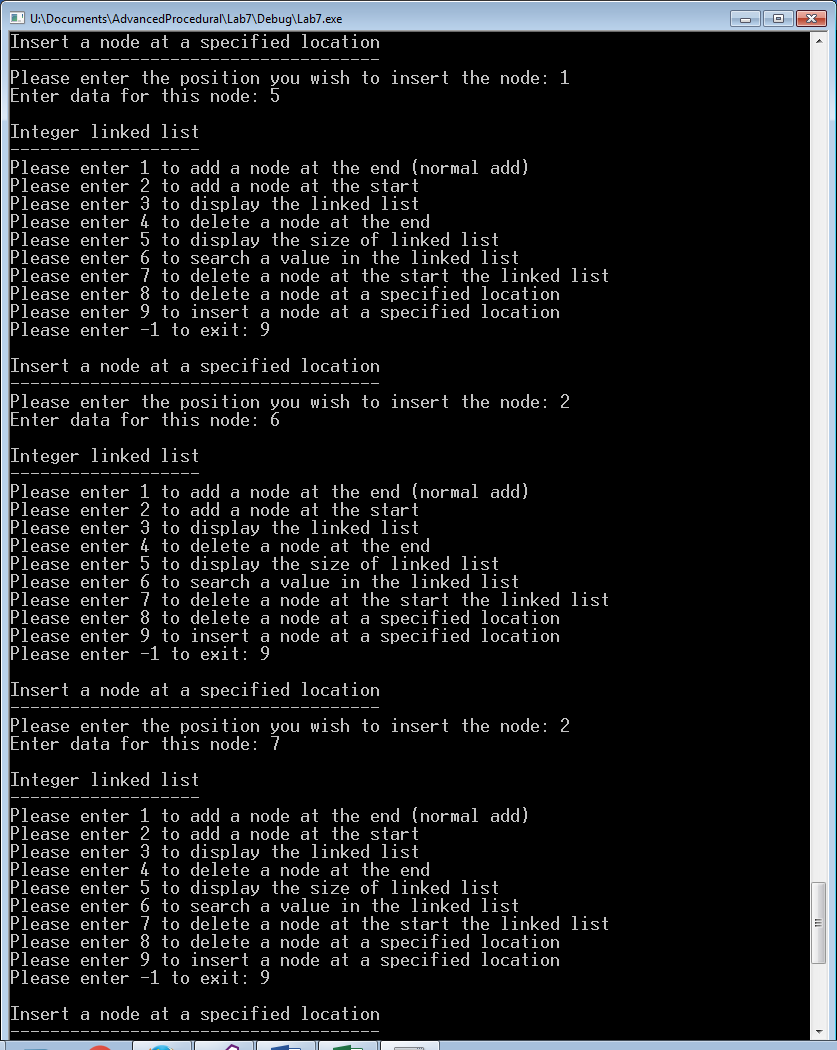


Figure 6 Inserting nodes at various locations in the list

