Week 10 – SOFT7019 lab session

This week we will utilise an online C IDE called online gdb, please access it at <https://www.onlinegdb.com/>

In the top right corner, you will have the option to select the programming language, please select C.



If you have a problem with this IDE, I would recommend installing CodeBlocks on your local machine. Last week we saw long lag times and errors ub the onlinegdb.com website during some of the lab sessions. I would recommend this guide to follow if installing CodeBlocks:

<https://www.youtube.com/watch?v=GWJqsmitR2I>

# Linked lists exercise

Implement the data structure shown in this figure:

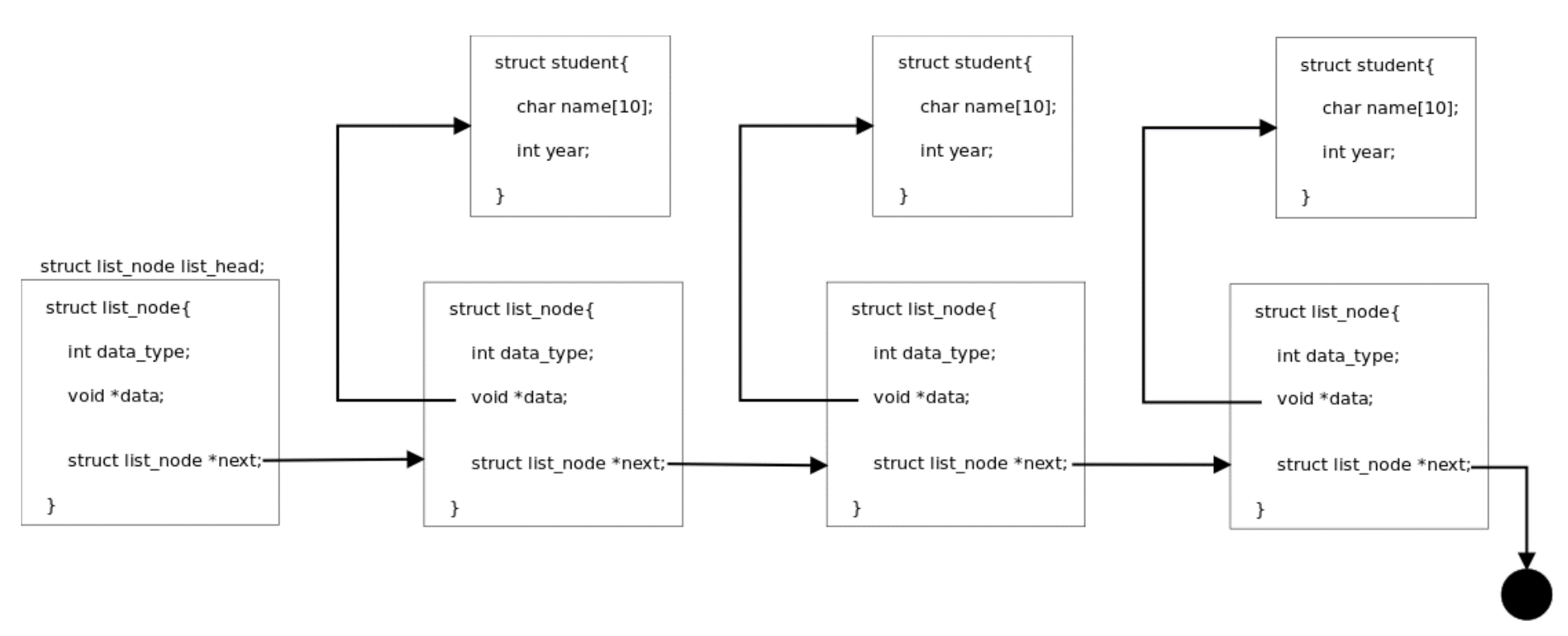


Figure 1 Linked list

1. Representing students

Define a structure to hold a student’s name (max 10 chars) and the year.

Implement a function to read student details. Use scanf to read the student’s name and year. There are two options to implement the function:

1. The function allocates memory for the student structure internally before reading the data and returning the structure. For example:

struct student \*new\_student();

1. The function takes the student structure as a parameter (pointer). For example:

void new\_student(struct student \*s);

Remember to use the -> operator to access the fields of the structure.

1. A linked list

Define a structure to represent a linked list node:

* the node should have a pointer towards the data that it contains, that is to a struct student variable
* if you want to keep the list node generic, able to hold any data for future use cases, the type of that pointer should be **void**; in that case, define an additional field in the node structure to indicate the type of data stored, you can use an **enum** to represent the different types
* the essential information in a linked list node is the next pointer. For convenience, define a function to create a new list node:

struct list\_node \*new\_node(void \*data);

or

struct list\_node \*new\_node(void \*data, int data\_type);

* allocate memory for a new struct list\_node inside the function
* set the data field in the struct to the data pointer provided as parameter to the function
* similarly, for the data\_type field, that indicates the type of the data
* the function will return the newly allocated structure.

1. Connect the linked list node manually

* Using the new\_student function defined above to create 3 students.
* Create a linked list node, this will be the **head** of the list that doesn’t contain information.
* For each student created, create a linked list node (you can use the new\_node function defined above), with the data pointer of the linked list node pointing to the student.
* Connect the list nodes manually through the **next** fields. The last node’s next field should point to NULL.

1. Loop through the list

Using the **next** field of the nodes write a for loop that starts with the first node and prints the values of all three nodes.

In an array you use an index variable that starts at 0 and goes all the way to the length of the list, accessing each array[index] element.

In the linked list we use a cursor that:

* starts at list\_head->next
* stops when it is equal to NULL
* you then access cursor->data; if the data field is a void pointer you will have to cast it to the right type (e.g. (struct student \*)cursor->data).

Example without loop:

struct list\_node \*cursor = &list\_head.next;

if (cursor == NULL) // stop and exit

printf("%d\n", (struct student\*)cursor->data);

cursor = cursor->next;

if (cursor == NULL) // stop and exit

printf("%d\n", (struct student\*)cursor->data);

cursor = cursor->next;

...

1. Putting it all together

**Linked list library:** define a set of functions to work with linked lists

1. Function to create list nodes. This was already developed above
2. Function to append a node to the end of the list.

* function prototype: void append(struct list\_node \*list\_head, struct list\_node \*node);
* function will start from the list head, go through the list until it finds the last node (for which ->next is NULL), then bind the new node.

1. Function to add a value after a list node (insert a value in the list).

* function prototype: void add\_after(struct list\_node \*n, struct list\_node \*node);
* function will insert the new node after n by updating the connections
* use the reference figure at the top to see how the connections must be set.

1. Function to find an element in the list

* function prototype: struct list\_node \*find\_node(struct list\_node \*list\_head, void \*data);
* the data parameter is the address of the contents of a node, not the address of the node
* the function must find the node that contains that address
* note this function is not very useful as it doesn’t allow looking for values stored in a linked list; that would require a function that looks inside the data address and whatever structure it might have, which can be done using function pointers. Without function pointers you will have to write custom functions for each application of the linked list.

1. Function to remove a node from the list.

* function prototype: void remove(struct list\_node \*list\_head, struct list\_node \*n);
* in order to remove a node from the list you have to reconnect its predecessor’s next field to its successor (use the figure at the top to understand the connections)
* as this is a singly linked list, you need to find the predecessor of the node to remove first; start at the list\_head, go through the nodes of the list using a cursor (as done above) until cursor->next == n ; the cursor will then be the predecessor of n
* after fixing the connections, free the memory of the node
* if the node to remove is the last in the list, it doesn’t have a successor; make sure you deal with this case as well.

You can test the linked list library in a simple program that reads a list of ints or chars, adding, and removing values:

char input = 0;

while (input != ’q’){

printf("1. Print list\n");

printf("2. Add value\n");

printf("3. Insert value\n");

printf("4. Remove value\n");

input = getchar();

switch (input){

case ’1’:

print\_list(&list\_head);

break;

case ’2’:

new\_value = malloc(...);

printf("New value:");

scanf("%d", new\_value);

// create new node to hold new value

append\_value(&list\_head, new\_node);

break;

case ’3’:

new\_value = malloc(...);

printf("New value:");

scanf("%d", new\_value);

// create new node to hold new value

printf("Insert after:");

scanf("%d", &after\_val);

after\_node = find\_node(&list\_head, after\_val);

// implement for finding ints in the list

insert\_value(after\_node, new\_node);

break;

case ’4’:

printf("Remove value:");

scanf("%d", &val\_to\_remove);

node\_to\_rem = find\_node(&list\_head,

val\_to\_remove);

// implement for finding ints in the list

remove\_node(&list\_had, node\_to\_rem);

break;

}

getchar(); // to get rid of the ’\n’

}