05OGDLP - Algorithms & Programming
Laboratory 6

Learning objectives

- Sorting
- Linked lists
- Dynamic memory allocation

1 Comparison between sorting algorithms

Create a program that:

- Reads two positive integers, `M` and `N`.
- Generates `N` random integers and stores them in a dynamically allocated array.
- Sorts the array in ascending order using one of the following algorithms:
  - Insertion sort
  - Selection sort
  - Bubble sort
- Repeats the random generation and sorting `M` times.
- Compares the performance of the three algorithms for `M`=10, 100, 1000, and `N`=1000, 5000, 10000, measuring their execution times.

Random number generation

To generate (pseudo-)random integer values, use the `rand()` function included in the `<stdlib.h>` library. It returns an integer between `0` and `RAND_MAX` (at least 32767). If a random value less than `RAND_MAX` is needed, use the modulo function (`%` operator in C), as shown in the following example:

```
#include <stdlib.h>
#define MAX 1000
...
int val;
val = rand() % MAX;
...
```

Measuring excution times

To measure execution times, use the `clock()` function from the `<time.h>` library. It returns the CPU time elapsed since the start of the program, which can be converted in seconds by dividing the returned value by the `CLOCKS_PER_SEC` constant. An example of using this function is as follows:

```
#include <time.h>
...
clock_t tic = clock();

// here, do your time-consuming job

clock_t toc = clock();
double time_spent = (double)(toc - tic) / CLOCKS_PER_SEC;
```

Sorting algorithms pseudocode

Below is the pseudocode of the proposed sorting algorithms. In all cases, `N` represents the number of data to sort and `V[0...N-1]` is the array where the data is stored. The array is sorted in ascending order.

Insertion sort

```
for i=1 to N-1
    key = V[i];
    // insert V[i] in the ordered sequence V[0...i-1]
    j = i-1;
    while (j >= 0 and V[j] > key)
        V[j + 1] = V[j];
        j = j - 1;
    V[j + 1] = key;
```

Selection sort

```
for i=0 to N-2
    // find smallest value in the V[i...N-1] sequence
    min = i;
    for j=i+1 to N-1
        if V[j] < V[min]
            min = j;
    // swap V[min] and V[i] contents
    swap(V, min, i);
```

Bubble sort

```
for i=N-1 downto 1
    // swap all adjacent non-ordered values in the 0...i interval
    for j=0 to i-1
        if V[j] > V[j+1]
            swap(V, j, j+1);
```

2 Student Records Management System

A university's student affairs department requires an efficient way to manage student records. Each student's record includes personal and academic details, such as student ID, name, major, year of study, and GPA.

The Student Records Management System will feature a user-friendly, menu-driven interface that allows users to interact with the program and perform various operations. The interface will present a list of options, each corresponding to a specific task related to managing student records.

- Add new student record. The user is prompted to input the details of a new student (name, major, year, and GPA). The student's data, including the assignet id, are shown to the user as feedback.
- Delete a student record. The user is asked to enter the student ID of the record they wish to delete
- Search for a student record. The user can search for a student by entering the student's id. The student's data is printed as feedback, or an error message if not found.
- Display all student records
- Exit

Data structure definition

The core data structure for the Student Records Management System is a struct representing each student record. This structure contains personal and academic details about the students and a pointer to the next record in the list, forming a linked list.

```
struct student_s {
    int studentID;          // Unique identifier for each student
    char name[100];         // Full name of the student
    char major[50];         // Major field of study
    int year;               // Current year of study
    float gpa;              // Current Grade Point Average
    struct student_s * next; // Pointer to the next student record
};
```

This structure is the backbone of the list operations, allowing for the storage and management of student data in a linked list format.

Use `typedef` to create the `student_t` data type as an alias to a `struct student_s` pointer:

```
typedef struct student_s * student_t;
```

Such an alias makes your code more concise and easy to read and write.

Function prototypes

1. `student_t student_new()`
   Purpose: Allocates memory for a new `student_t` using `malloc`, checks if the memory allocation was successful. If not, returns NULL. Generates a progressive ID, and returns the newly allocated `student_t`.
   Return type: The newly created `student_t` (pointer to) structure. If the memory allocation fails, the function should return NULL.

2. `void student_free(student_t s)`
   Purpose: Removes a student record from the linked list based on the provided student id. This function is called when a student record is removed from the list (e.g., after using the `student_delete` function) or when the entire list is being destroyed and all its elements need to be deallocated.
   Parameters

   - `student_t s`: the `student_t` that needs to be deallocated. This is the record whose memory was previously allocated using `student_new` and now needs to be freed.

3. `void student_add(student_t * head, char * name, char * major, int year, float gpa)`
   Purpose: Adds a new student record to the existing linked list of student records. It uses `student_new` to create a new record and then inserts this record into the list.
   Parameters

   - `student_t * head`: reference to the start (or head) of the linked list. `head` points to the `student_t` that is at the beginning of the list. Notice it *hides* a first pointer, as it aliases a `struct student_s *`. The second pointer level (`* head`) allows the function to modify the actual head pointer of the list. This is necessary for operations that might change where the head points to (like adding or removing the first element of the list).

     In C, arguments are passed to functions by value, meaning the function receives a copy of each argument. If we were to pass a single-level pointer (`student_t head`), the function would receive a copy of this pointer, and any changes made to the pointer itself (like pointing it to a new (`student_t`)) would not affect the original pointer outside the function. By passing a double pointer (`student_t * head`), the

function receives a copy of the pointer to the pointer, allowing it to modify the original head pointer that exists outside the function.
    - `char * name`: name of the student
    - `char * major`: major field of study for the student
    - `int year`: The current year of study for the student.
    - `float gpa`: The current GPA of the student.

4. `void student_delete(student_t * head, int id)`
   Purpose: Removes a student record from the linked list based on the provided student id. It uses `student_free` to release the memory held by the removed student.

5. `student_t student_search(student_t head, int id)`
   Purpose: Finds and returns a pointer to a student record in the list based on the student id.
   Parameters
    - `student_t head`: Using a single pointer indicates that the function performs a read-only operation on the list. It traverses the list starting from the head but does not alter the list's links or the head pointer itself.

6. `void students_display(student_t head)`
   Purpose: Traverses the entire list and displays the details of each student record.

Testing

To ensure the robustness and efficiency of your Student Records Management System, especially in terms of memory management, it's highly recommended to test your program for memory leaks using Valgrind. Valgrind is a powerful tool that can help you detect memory leaks and other memory-related problems in your C programs.

If you haven't already installed Valgrind, you can usually do so through your system's package manager. For example, on Ubuntu, you can use `sudo apt-get install valgrind`.

- Compile your program with the `-g` flag to include debugging information. This makes the output of Valgrind more useful. For example: `gcc -g yourprogram.c -o yourprogram`.
- Use Valgrind to run your program by executing `valgrind --leak-check=full ./yourprogram`. This command will run your program, and Valgrind will check for memory leaks.

Valgrind will report any memory leaks it detects. Carefully analyze the output to find where in your code you might have forgotten to free memory. The output includes the number of bytes leaked and where the allocation happened, which can be traced back to your source code.

Run Valgrind multiple times during development, especially after making changes that affect memory allocation, to continually ensure that your program is free of memory leaks.