电子科技大学
格拉斯哥学院
Glasgow College, UESTC

UESTC(HN) 1005
Introductory Programming
Lab Manual

S1 2025–26

Contents

5 Lab Session V **1**
5.1 Linked List 1
5.1.1 Practical patterns 2
5.2 File Handling in C 6
5.2.1 Opening & closing a File 6
5.3 Exercise 5A 7
5.4 Exercise 5B 10

Lab Session V

5.1 Linked List

A linked list is a fundamental data structure in computer science, valued for its dynamic nature and pointer-based implementation. Although not always the most efficient option, linked lists are commonly used in scenarios that require flexible memory management.

In this section, we will familiarise you with this data structure by implementing a simple linked list in C, starting with the definition of the node structure.

Code 5.1: `node.c`

```c
struct Node {
    int data;
    struct Node* next;
};
```

In the `Node` structure, `data` represents the value stored in the node (in this case, an integer) and `next` is a pointer to the next node in the list.

Next, we implement functions to perform basic operations on a linked list, such as creating a new node, inserting a node at the beginning, and printing the list. In other words, let us define an *interface* for interacting with linked lists.

Code 5.2: `linkedList.c`

```c
/* Using the Node struct defined in Code 5.1 */
#include <stdio.h>
#include <stdlib.h>

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

```c
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    printf("Linked List: ");
    printList(head);

    return 0;
}
```

5.1.1 Practical patterns

Here, we present a curated set of additional design patterns useful for real-world applications that use a linked list data structure. Each function provides a different way to interact with linked lists. Read through each definition and see how the C code achieves the specified task.

Before you start, note that many linked-list tasks hinge on two choices: **in-place updates** vs **creating a new list**, and how you manage memory. In-place saves allocation but requires careful `free()` when removing nodes. Creating a new list avoids side effects but requires allocating new nodes. The patterns in this section show both styles.

Moreover, you may encounter some common slips when working with linked lists as follows, so double-check for these when facing an error.

- Forgetting to `free` nodes when removing consecutive repetitions.
- Reusing nodes from an existing list when building a new result.
- Not handling empty lists—every traversal should tolerate `NULL`.
- Advancing only one side after an equal match during intersection (duplicates/loops).

Building without edge cases (dummy head + tail)

Code 5.3: `ll_dummy_tail.c`

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

static Node* make_node(int v){
    Node *p = (Node*)malloc(sizeof *p);
    p->data = v; p->next = NULL;
    return p;
}

/* Construct a list from an array (order preserved) */
Node* build_from_array(const int *a, int n){
    Node dummy = {0, NULL};
    Node *tail = &dummy;
    for (int i = 0; i < n; ++i) {
        tail->next = make_node(a[i]);
        tail = tail->next;
    }
    return dummy.next;
}
```

Combining two ordered sequences (linear scan)

Code 5.4: `ll_combine.c`

```c
static Node* make_node(int v){
    Node *p = (Node*)malloc(sizeof *p);
    if(!p){ perror("malloc"); exit(1); }
    p->data = v; p->next = NULL;
    return p;
}

Node* combine_sorted_new(const Node *a, const Node *b){
    Node dummy = {0, NULL}, *tail = &dummy;
    while (a && b){
        const Node *pick = (a->data <= b->data) ? a : b;
        tail->next = make_node(pick->data);
        tail = tail->next;
        if (pick == a) a = a->next; else b = b->next;
    }
    const Node *rest = a ? a : b;
    while (rest){
        tail->next = make_node(rest->data);
        tail = tail->next;
        rest = rest->next;
    }
    return dummy.next;
}
```

Eliminating consecutive repeats (in place)

Code 5.5: `ll_compress.c`

```c
void compress_runs_in_place(Node *head){
    for (Node *cur = head; cur && cur->next; ){
        if (cur->data == cur->next->data){
            Node *dup = cur->next; // unlink duplicate
            cur->next = dup->next;
            free(dup);
        }
        else { cur = cur->next; }
    }
}
```

Extracting common elements from two ordered sequences

Code 5.6: `ll_common.c`

```c
static Node* make_node(int v){
    Node *p = (Node*)malloc(sizeof *p);
    if(!p){ perror("malloc"); exit(1); }
    p->data = v; p->next = NULL;
    return p;
}

Node* common_elements_sorted_new(const Node *a, const Node *b){
    Node dummy = {0, NULL}, *tail = &dummy;
    while (a && b){
        if (a->data < b->data){
            int v = a->data;
            do { a = a->next; } while (a && a->data == v);
        } else if (b->data < a->data){
            int v = b->data;
            do { b = b->next; } while (b && b->data == v);
        } else {
            int v = a->data; // match
            // Skip runs in both lists
            do { a = a->next; } while (a && a->data == v);
            do { b = b->next; } while (b && b->data == v);
            // Append once
            tail->next = make_node(v);
            tail = tail->next;
        }
    }
    return dummy.next;
}
```

5.2 File Handling in C

File handling in C allows you to read from and write to files. The standard library provides functions for opening, reading, writing, and closing files. We detail some of the basic steps involved below.

5.2.1 Opening & closing a File

In C, you can open a file using the `fopen` function, which requires two arguments: the name of the file and the mode in which the file should be opened. The available modes are the following.

- `r`: Open a file for reading. The file must exist.
- `w`: Open a file for writing. If the file exists, it is truncated; if it does not exist, a new file is created.
- `a`: Open a file for appending. If the file does not exist, it is created.
- `rb`, `wb`, `ab`: Binary modes for reading, writing, and appending, respectively. Binary file access modes are mainly a Windows system issue. For example, on a Windows system, a newline character `\n` will translate to the output `\n\r\` when using write mode (`w`), but will instead output without translation as `\n` when in binary write mode (`wb`). The historical reasons, MS-DOS based heritage, and typewriter era baggage resulting in this Windows system nuance are well beyond the scope of Introductory Programming.

After you have finished working on/with the file, you can (and should) call `fclose()` to close the file.

Code 5.7: `file.c`

```c
#include <stdio.h>

int main(){
    FILE *file;

    file = fopen("./example.txt", "w");

    //Write to the file if the file exists
    if (file != NULL) {
        fprintf(file, "Hello, World!\n");
    }

    fclose(file);

    return 0;
}
```

5.3 Exercise 5A

5A - Merging Two Lists

Problem Statement

You are given two ascending integer sequences (non-decreasing; duplicates allowed). Build two linked lists (A and B) using the node type defined in Code 5.1, and implement the following functionality:

- **Merge**: merge A and B into one ascending list (duplicates are allowed), then remove **adjacent** duplicates in place on the merged list.
- **Intersection**: set intersection of A and B, printed as a strictly increasing list (Build a new list for this; do not mutate A or B.).
- Helper functions to **free** and **print** a linked list.

Note that your code must handle the following situations: empty lists, lists where all entries duplicates, disjoint lists, negative values, and lists with highly unbalanced sizes.

Input format

For each set of inputs, you will receive 4 lines, with every two lines representing the length and actual data of one sequence:

```
n
a_1 a_2 ... a_n
m
b_1 b_2 ... b_m
```

Note that **both sequences are ascending** and all their values are `int` ($0 \leq n, m \leq 100000$).

Output

Print two lines (and print a blank line after the label if the list is empty):

```
MERGE: <values of Merged list, space-separated>
INTERSECTION: <values of Intersected list, space-separated>
```

Sample Input 1

```
5
1 2 2 5 10
4
2 3 5 5
```

Sample Output 1

```
MERGE: 1 2 3 5 10
INTERSECTION: 2 5
```

Sample Input 2

```
0

3
-2 -2 7
```

Sample Output 2

```
MERGE: -2 7
INTERSECTION:
```

Explanation

For the first example, values 2 and 5 are duplicates, so when merging two lists, you should delete the extra 2's and 5's and reorder the resultant list in ascending order. When intersecting the two, you should only keep 2 and 5 because they are the values that exist in both lists, and you only keep one occurrence following the no-duplication requirement.

In the second example, one list is empty (length 0), while the other contains negative values. When merging, you don't have to worry about the empty list. For intersection, you should output an empty list because an empty list is the result of intersecting any list (including an empty list) with an empty list.

Base Code

```c
#include <stdio.h>
#include <stdlib.h>

/* -------- Definitions & helper functions (do NOT modify) -------- */
typedef struct Node {
    int data;
    struct Node *next;
} Node;

static Node* make_node(int v){
    Node *p = (Node*)malloc(sizeof *p);
    p->data = v;
    p->next = NULL;
    return p;
}
```

```c
/* Append using a tail pointer. Use this in main() to build lists. */
static void push_back_tail(Node **head, Node **tail, int v){
    Node *n = make_node(v);
    if(!*head){ *head = *tail = n; }
    else { (*tail)->next = n; *tail = n; }
}

/* --------------------------- TODOs --------------------------- */
void print_list_prefixed(const char *label, const Node *h){
    //TODO: Print a linked list for output
}

void free_list(Node *h){
    //TODO: Free the entire list.
}

Node* merge_sorted_lists(const Node* a, const Node* b){
    //TODO: Merge two ascending lists into a NEW list (with duplicates)
}

void unique_in_place(Node* head){
    //TODO: Remove adjacent duplicates IN PLACE on the given list
}

Node* intersect_sorted_lists(const Node* a, const Node* b){
    //TODO: Intersect into a new strictly increasing list; NO mutation
}

int main(){
    int n, m;
    Node *A = NULL, *At = NULL; // head & tail of A
    Node *B = NULL, *Bt = NULL; // head & tail of B

    //TODO: build up the lists

    /* -------- Compute and Output (do NOT modify) -------- */
    Node *M = merge_sorted_lists(A, B); // duplicates-allowing merge
    unique_in_place(M); // remove adjacent duplicates IN PLACE
    Node *I = intersect_sorted_lists(A, B); // strictly increasing

    print_list_prefixed("MERGE", M);
    print_list_prefixed("INTERSECTION", I);

    //TODO: implement free_list() and call it for A, B, M, and I

    return 0;
}
```

5.4 Exercise 5B

5B - Pixel Frequency

Time limit per test: 1 seconds

Problem Statement

In digital imaging, an image is composed of a grid of small units called pixels (short for "picture elements"). Each pixel represents the smallest possible piece of the image and contains information about its colour or brightness. Pixels are typically represented using colour models like RGB, where each colour component (red, green, or blue) is assigned a value, often in the range of 0 to 255. These values can be represented in hexadecimal format, where two hexadecimal digits (ranging from `00` to `FF`) represent the colour value of a pixel.

In this exercise, you are given a text file containing an image. Each pixel in the image is represented by a two-digit hexadecimal number (from `00` to `FF`). Your task is to read the pixel data from the file, count the frequency of each pixel value, and output the pixel values with the highest frequency into another file.

Constraints:

- $1 \leq w, h \leq 256$ (width and height of the image).
- The pixel values range from `00` to `FF`.

Input

The input of your program is the name of a file. This file consists of h lines each contain w space-separated two-digit hexadecimal numbers, representing the pixel values of the image. Each pixel value is a 2-character hexadecimal number in the range `00` to `FF`.

Output

Output the pixel value with the highest frequency and its frequency to another file named `output.txt`. In the case of a tie, i.e., two or more pixel values appear with the same highest frequency in the image, only output the pixel value with the lowest value. If you are given n files, the content of `output.txt` should be n lines, i.e., run your program n times, once for each input file, and append the output to `output.txt`.

Sample input 1

```
image0.txt
```

The content of `image0.txt`:

```
77C0AA6A307134EAFBF9
86E6CDB3887DC9B8E711
12157D28AA6AEE018E73
5C0533066F6377A44D72
```

```
9DD3596B86E1E84F99CF
61ACE5DED48F48C390D7
36ECDC69F24CCC69F019
DB8DEC34F87216E0C2AF
B0235B950130244AF3B4
2129A0FD9292495EFB39
```

Sample Output — "output.txt"

```
01 2
```

Although both the pixel value `01` and `AA` appear 2 times, `01` is smaller than `AA`. After your program runs with the input of the filename `image0.txt`, the result should be stored in the file `output.txt`. When your program runs on the second test case, the results should be appended to the file instead of replacing the existing file.

Sample input 2

```
image1.txt
```

The content of `image1.txt`:

```
53AABF9E7A025A432C371B784DFF5C6ED836E4AC
06E37DCAE55833E1DC66D62F1195CE8B9828CEC4
5FE93CACE9981AC1CEFE6DD5E1EB9FC643D2A71F
397E4F4A131DD5AB45A36FA48DAB5176446B3712
6AA4E74B8F8712D259B9F2923741DC4B5EB1F6A3
556647E211985855048F686E334FB9C3D6CB9530
8587C2BCC89F072650FEC9A564118775A9DFCBAD
6E331BA282D56559A0FA8925824BE24AEAE9713B
E73AE04B4B68C1F5478CA2B6BFBE584193BD9A33
B72359396F3B845924F5940C2F75577BDD187024
A412DA63D032A563EF3F97A763F0E0D22B642B4F
59C05B8935B30412CB7436708611D3574378BA33
B851DA1B41BAED6C1F18BC78D817010DCA051F96
7956060067D957AA5211DD0A63B725A472121191
2ACD0903E40B10AF1030458A864B8AED24E19776
F27580552CA5FA9EB70B2FE2D839E5BC44F56B54
25B0DEABFB6898204930963CA51791D1BC8B7074
969F566ED83B2B1C30967156474F0142B89A6201
CAF93D6F10CF40CC5AB040F150965F28D18A4502
21B658680559AABDF30DBFBD06FC2C16CB6DE226
```

Sample Output — "output.txt"

```
01 2
11 6
```

Note that the second program output has been appended to `output.txt`.

Test Environment

Assume that your program is named `5B.c`, you can put `5B.c`, `image0.txt` and `image1.txt` **in the same directory**. And then run your program `5B.c` twice. For the first time, input `image0.txt`, and, for the second time, input `image1.txt`. Finally, you can check whether your `output.txt` is the same as above.

When you submit your program, you will find that there are 11 IO Tests in the logic test. Please note that IO Tests 1–10 are not graded, i.e, success or failure on these tests will not impact your score. Only the "IO Test - Grade" checks if you have generated a file containing the correct output (i.e., this is essentially a one-shot test, so check your program carefully).

Base Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char filename[20];
    scanf("%s", filename);

    //TODO
    return 0;
}
```