Lesson 7: Data Structures
Introduction: dynamic array

- Conventional array in C has fix number of elements
- Dynamic array is array with variable number of elements: actually a pointer and a variable indicating \( n \) of elements
  
  ```
  int n = 5;
  int* arr = (int*)malloc(n*sizeof(int));
  ```

- Insert an element into dynamic array:
  
  ```
  pos = 2; val = 25;
  arr1 = (int*)malloc((n+1)*sizeof(int));
  memcpy(arr1, arr, pos*sizeof(int));
  memcpy(arr1+pos+1, arr+pos, (n-pos)*sizeof(int));
  arr1[pos] = val;
  free(arr);
  arr = arr1;
  ```

- Similarly for element removal
Overview

- Above example shows that dynamic arrays are slow in inserting/removing elements because of moving large memory areas, especially when there are many elements → necessity of a more flexible data structure

- Often used data structures:
  - Stack
  - Queue
  - Linked list
  - Dynamic array (vector)
  - Map, dictionary, hash table
  - Set
  - Tree
  - Graph
Linked list

- Group of nodes linked together by pointers and represent a sequence of elements. For presentation:
  - A pointer to the first node (or NULL for empty list)
  - Each node includes 2 members: data, and a pointer to the next node
  - The next pointer of last element is NULL
Declaration of linked list

- Example for `int` elements:
  ```c
  struct SELEM;
  typedef struct SELEM ELEM, *PELEM, *LLIST;
  struct SELEM {
      int data;
      PELEM next;
  };
  ```

- Linked list library:
  - llist.h
  - llist.c
Operation of linked list

- Necessary functions:
  - LLIST llInit();
  - LLIST llInsertHead(LLIST l, int data);
  - LLIST llInsertTail(LLIST l, int data);
  - LLIST llInsertAfter(LLIST l, PELEM a, int data);
  - LLIST llDeleteHead(LLIST l);
  - LLIST llDeleteTail(LLIST l);
  - LLIST llDeleteAfter(LLIST l, PELEM a);
  - LLIST llDeleteAll(LLIST l);
  - PELEM llSeek(LLIST l, int i);
  - void llForEach(LLIST l, LLCALLBACK func, void* user);
  - int llLength(LLIST l);
Initialization

LLIST llInit() {
    return NULL;
}

- Initialized to NULL → empty list
Insert element to head of a list

LLIST llInsertHead(LLIST l, int data) {
    PELEM e = (PELEM)malloc(sizeof(ELEM));
    e->data = data;
    e->next = l;
    return (LLIST)e;
}

Diagram:

- `list` node connected to `data` node.
- `data` node has a `next` field.
- A new node is inserted with `data` set to `data`, and `next` pointing to `l`.
- The new node is returned as the head of the list.
Insert element to tail of a list

```c
LLIST llInsertTail(LLIST l, int data) {
    PELEM p;

    PELEM e = (PELEM)malloc(sizeof(ELEM));
e->data = data;
e->next = NULL;

    if (l==NULL) return (LLIST)e;

    for (p=l; p->next; p = p->next) ;
p->next = e;
return l;
}
```
Insert element after another in a list

LLIST llInsertAfter(LLIST l, PELEM a, int data) {
    PELEM e;
    if (!a) return l;

    e = (PELEM)malloc(sizeof(ELEM));
    e->data = data;
    e->next = a->next;
    a->next = e;
    return l;
}
Delete first element

LLIST llDeleteHead(LLIST l) {
    PELEM p;
    if (!l) return NULL;

    p = l->next;
    free(l);
    return (LLIST)p;
}
Delete last element

LLIST llDeleteTail(LLIST l) {
    PELEM p;

    if (!l) return NULL;
    if (!l->next) {
        free(l);
        return NULL;
    }

    for (p=l; p->next->next; p = p->next) ;
    free(p->next);
    p->next = NULL;
    return l;
}
Delete element after another in a list

```
LLIST llDeleteAfter(LLIST l, PELEM a) {
    PELEM p;
    if (!a || !a->next) return l;

    p = a->next;
    a->next = p->next;
    free(p);
    return l;
}
```
Iterate through a list

- Find the $i^{th}$ element of a list

  ```c
  PELEM llSeek(LLIST l, int i) {
    for (; i>0 && l; i--)
    l = l->next;
    return (PELEM)l;
  }
  ```

- Do some operation to all elements of a list

  ```c
  typedef void (*LLCALLBACK)(int, void*);
  
  void llForEach(LLIST l, LLCALLBACK func, void* user) {
    for (; l; l=l->next)
    func(l->data, user);
  }
  ```
Other operations

- Count number of elements
  ```c
  int llLength(LLIST l) {
    int c;
    for (c=0; l; c++)
      l = l->next;
    return c;
  }
  ```

- Delete all elements (make list empty)
  ```c
  LLIST llDeleteAll(LLIST l) {
    PELEM p;
    for (; l; l=p) {
      p = l->next; free(l); }
    return NULL;
  }
  ```
Use example

```c
#include <stdio.h>
#include "llist.h"

void printList(LLIST l) {
    printf("%d elements: ( ",
            llLength(l));
    for (; l; l = l->next)
        printf("%d ", l->data);
    printf(") \n");
}

void listSum(int data, void* user) {
    int* pS = (int*)user;
    *pS += data;
}

int main() {
    LLIST l;
    PELEM p;
    int i, s;

    l = llInit();
```
```c
    for (i=0; i<5; i++) {
        l = llInsertTail(l, i);
        l = llInsertHead(l, -i);
    }

    printList(l);

    p = llSeek(l, 1);
    l = llDeleteAfter(l, p);
    l = llDeleteHead(l);
    l = llDeleteTail(l);
    printList(l);

    s = 0;
    llForEach(l, listSum, (void*) &s);
    printf("Sum of values: %d\n", s);

    l = llDeleteAll(l);
    printList(l);

    return 0;
}
```
Properties of linked lists

Advantages/drawbacks of linked lists compared to dynamic arrays:

+ Flexible: easy to insert/remove elements, sort or change the order of elements without moving them in memory
+ No need to allocate a large consecutive memory area to store all elements
  - Need additional memory to store next pointers
  - Sequential access: impossible to access to arbitrary $i^{th}$ element, but need to iterate from the beginning

Besides basic form (singly linked list), there are other forms: doubly linked list, ordered linked list, stack, queue, cycle linked list,… and can be implemented in many different ways depending on specific problems
Stack

- Can be considered as a specialized linked list, with only 2 operations:
  - push: insert element into head
  - pop: take value and remove first element

  → LIFO (last in, first out) list

- Applications:
  - Expression evaluation (inverse Polish notation)
  - Implementation of function call in programs
  - Recursion removal
  - Implementation of undo feature
Queue

- Can also be considered as a specialized linked list, with two operations:
  - enqueue: insert element to head
  - dequeue: take value and remove last element
  → FIFO (first in, first out) list
- Queue are usually implemented by using two pointers, one to the first element for insertion, the other to the last for removal
- Applications:
  - Data transfer and communication
  - Reading/writing buffer
  - Implementation of client/server services
Doubly linked list (DLL)

- In standard linked list, each node has one pointer to the next one → singly linked list, can be iterated only in one direction

- DLL: each node has two pointers, one to the next, the other to the previous node → possibly iterated in two directions
Problems

1. Declare a DLL and write functions for: add/remove first element
2. Write Pop(), Push() functions for stack and Enqueue(), Dequeue() for queue on the basis of the linked list library and give remarks on ineffective operations
3. Calculate average value of a linked list by two ways: iteration, using llForEach() function
4. Write llConvert(int* arr, int count) function to convert an array into linked list without using functions from the library
5. Write llReverse(LLIST l) function to reverse the order of a linked list without creating new nodes or allocating new memory
6. Same as Prob. 5, but do recursively
7. Given two linked lists l1 and l2, write llInsertListAfter(l1, l2, p) function to insert l2 into l1 after element p
8. Modify the library to store any type of data