CS 137

Programming Principles

Chapter 13

Linked Lists

Victoria Sakhnini

Table of Contents

Introduction

A linked list is a typical data structure where the data is held in individual pieces chained together, making it easy to insert new elements.

A linked list node consists of:

- A piece of data (we will use an integer for now).
- A pointer to the next linked list element, or `NULL` if it is the last element.

Basic Syntax

```
struct ll {
    struct llnode *head;
};

struct llnode {
    int item;
    struct llnode *next;
};
```

📌 Each node stores a value and a pointer to the next node. The last node's next pointer is NULL. The ll struct holds a pointer to the head (first) node of the list.

Example: Polynomial

We will model a polynomial as a linked list where each node stores a degree and a coefficient. The list is ordered so that the largest degree is at the front. For example, $p = 8x^3 + 3x + 3$ would be stored as three nodes in descending degree order.

Interface — poly.h

```
#ifndef POLY_H
#define POLY_H

// Polynomial stored in decreasing degree order.
struct polynode;
struct poly;

// Pre:  None
// Post: Creates a null polynomial
struct poly *polyCreate(void);

// Pre:  *p is a valid polynomial (even null)
// Post: Destroys the polynomial
void polyDelete(struct poly *p);
```

```c
// Pre:  poly *p is valid
// Post: Returns p(x)
double polyEval(struct poly *p, double x);

// Pre:  poly *p is valid; deg is non-negative
// Post: Sets the coefficient at degree deg to coeff
// Assume: deg not already in poly
struct poly *polySetCoeff(struct poly *p, int deg, double coeff);

// Pre:  poly *p is valid
// Post: Returns the largest non-zero degree in poly
int polyDegree(struct poly *p);

// Pre:  poly *p is valid
// Post: Returns a deep copy of the polynomial
struct poly *polyCopy(struct poly *p);

#endif
```

Implementation — poly.c

```c
#include "poly.h"
#include <math.h>
#include <stdlib.h>
#include <assert.h>

// Polynomial stored in decreasing degree order.
// Each node stores a degree, coefficient, and pointer to the next term.

typedef struct polynode {
    int    deg;
    double coeff;
    struct polynode *next;
} polynode;

typedef struct poly {
    struct polynode *head;
} poly;

// Empty poly: head points to NULL
poly *polyCreate(void) {
    poly *p  = malloc(sizeof(poly));
    p->head  = NULL;
    return p;
}

/*
 * polyDelete:
 *   curnode  -> node to be freed
 *   nextnode -> next node to be freed
 * We save nextnode before freeing curnode so we don't lose
 * access to the rest of the list.
 */
void polyDelete(poly *p) {
    polynode *nextnode = p->head;
    polynode *curnode  = NULL;
    while (nextnode) {
        curnode  = nextnode;
        nextnode = nextnode->next;
        free(curnode);
    }
    free(p);
}

double polyEval(poly *p, double x) {
    double f = 0.0;
    polynode *node = p->head;
    // Iterate over nodes until NULL, accumulate each term
    for (; node; node = node->next)
        f += pow(x, node->deg) * node->coeff;
    return f;
}

poly *polySetCoeff(poly *p, int deg, double coeff) {
    if (!coeff) return p;  // zero coefficient: nothing to add
    polynode *node = p->head;
    if (!node || deg > node->deg) {
        // New highest-degree term: insert at front
        polynode *r = malloc(sizeof(polynode));
        r->coeff = coeff;
        r->deg   = deg;
        r->next  = node;
        p->head  = r;
        return p;
    }
    // Find the correct position to maintain decreasing degree order
    polynode *cur = p->head;
    for (; cur->next && cur->next->deg > deg; cur = cur->next) ;
    if (cur->next && cur->next->deg == deg) {
        cur->next->coeff = coeff;  // update existing coefficient
    } else {
        polynode *r = malloc(sizeof(polynode));
        r->coeff   = coeff;
        r->deg     = deg;
        r->next    = cur->next;
        cur->next  = r;
    }
    return p;
}

int polyDegree(poly *p) {
    assert(p);
    return p->head->deg;
}

poly *polyCopy(poly *p) {
    poly *q = polyCreate();
    polynode *node = p->head;
    while (node) {
        q    = polySetCoeff(q, node->deg, node->coeff);
        node = node->next;
    }
    return q;
}
```

Testing — main.c

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

int main(void) {
    // Create 3x^4 + 2.5x^2 - 10
    struct poly *p = polyCreate();
    p = polySetCoeff(p, 4,  3.0);
    p = polySetCoeff(p, 2,  2.5);
    p = polySetCoeff(p, 0, -10.0);

    printf("p(0)=%.2f\n",  polyEval(p, 0));  // p(0)=-10.00
    printf("p(1)=%.2f\n",  polyEval(p, 1));  // p(1)=-4.50
    printf("degree=%d\n",  polyDegree(p));   // degree=4

    // Create a deep copy
    struct poly *p2 = polyCopy(p);

    // Free the original (important: avoids memory leak)
    polyDelete(p);

    // The copy still works
    printf("p2(0)=%.2f\n", polyEval(p2, 0));
    printf("p2(1)=%.2f\n", polyEval(p2, 1));
    printf("degree=%d\n",  polyDegree(p2));

    polyDelete(p2);
    return 0;
}
```

Generalized Lists

We just built a linked list of integers and one for polynomial terms. What if we want a linked list of floats? Or a linked list of linked lists? Do we have to create an entirely new structure with its own functions for each type?

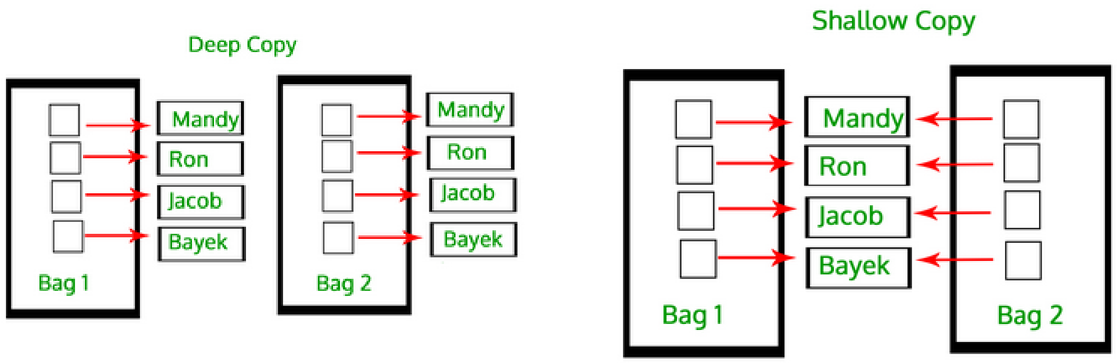
Ideally, no. The answer in C is to use `void *`:

A void * can point at anything. This is how generalizations are often done in C — by using parameters of type void *. The caller provides function pointers for type-specific operations (comparison, copy, free), making the list implementation completely type-agnostic.

Deep Copy

Copying lists is common. There are two kinds of copy:

- **Shallow copy** — creates a new variable that shares the same references as the original. Modifying data through one copy affects the other.
- **Deep copy** — creates a completely independent copy. Each node (and its data) is newly allocated; no references are shared.



To make a deep copy of a linked list, we must create a new node for each node in the original. If the data are themselves pointers, we must deep-copy the data too (rather than just copying the pointers). This requires the caller to supply a `cpy` function. It is most natural to implement this recursively via a helper that deep-copies a single node and its `next` chain.

Interface — glist.h

```
#ifndef GLIST_H
#define GLIST_H

struct llnode;
struct gl;  // general list

// Post: Creates an empty list
struct gl *gl_Create(void);

// Pre:  elem is a pointer to heap-allocated data that this list now owns.
// Post: Adds a new node storing elem at the front of the list.
void gl_addToFront(struct gl *lst, void *elem);

// Pre:  lst is already sorted with respect to cmp.
// Post: Adds a node storing elem in sorted order.
//   cmp(a, b) returns:  <0 if a before b,  0 if equal,  >0 if a after b
void gl_addInOrder(struct gl *lst, void *elem,
                   int (*cmp)(void *, void *));

// Pre:  *lst is a valid list
// Post: Destroys the list; calls kill() on each data element
void gl_delete(struct gl *lst, void (*kill)(void *));

// Pre:  lst is valid
// Post: Returns the length of lst
int gl_length(struct gl *lst);

// Pre:  lst is valid; n <= length of lst
// Post: Returns a pointer to the nth element's data (1-indexed)
void *gl_nthElem(struct gl *lst, int n);

// Post: Returns a deep copy of lst; cpy() is provided by the caller
struct gl *gl_copy(struct gl *lst, void *(*cpy)(void *));

#endif
```

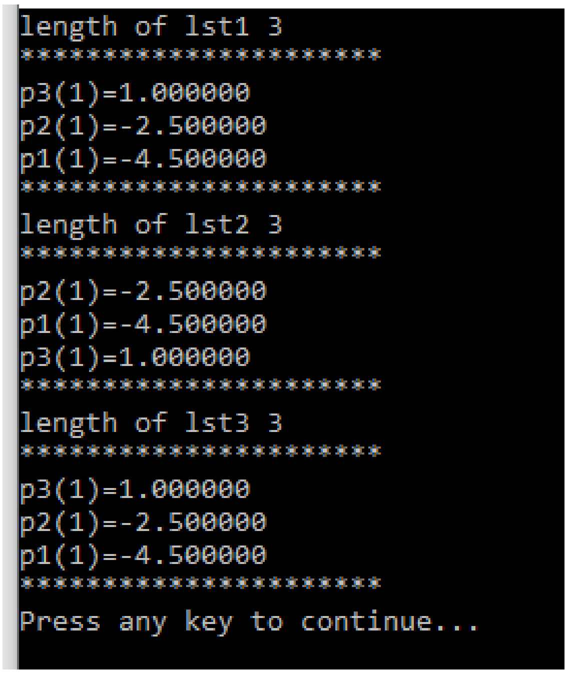
Implementation — glist.c

```
#include "glist.h"
#include <stdlib.h>
#include <assert.h>

typedef struct llnode {
    void           *data;
    struct llnode  *next;
} llnode;

typedef struct gl {
    llnode *head;
} gl;

gl *gl_Create(void) {
    gl *ret   = malloc(sizeof(gl));
    ret->head = NULL;
    return ret;
}

void gl_addToFront(gl *lst, void *elem) {
    llnode *newNode = malloc(sizeof(llnode));
    newNode->data   = elem;
    newNode->next   = lst->head;
    lst->head       = newNode;
}

// cmp(a,b): <0 if a before b, 0 if equal, >0 if a after b
void gl_addInOrder(gl *lst, void *elem, int (*cmp)(void *, void *)) {
    llnode *prev = NULL;
    llnode *cur  = lst->head;
    // Advance until we find where elem belongs
    for (; cur && cmp(elem, cur->data) > 0; prev = cur, cur = cur->next) ;
    if (!prev)
        gl_addToFront(lst, elem);
    else {
        llnode *n = malloc(sizeof(llnode));
        n->data    = elem;
        n->next    = cur;
        prev->next = n;
    }
}

void gl_delete(gl *lst, void (*kill)(void *)) {
    llnode *nextnode = lst->head;
    llnode *curnode  = NULL;
    while (nextnode) {
        curnode  = nextnode;
        nextnode = nextnode->next;
        kill(curnode->data);
        free(curnode);
    }
    free(lst);
}

int gl_length(gl *lst) {
    int len = 0;
    llnode *node = lst->head;
    for (; node; node = node->next) len++;
    return len;
}

void *gl_nthElem(struct gl *lst, int n) {
    assert(n <= gl_length(lst));
    int num;
    llnode *node = lst->head;
    for (num = 1; num < n; num++) node = node->next;
    return node->data;
}

// Recursive helper: deep-copies one node and its entire next chain
llnode *llnode_copy(llnode *n, void *(*cpy)(void *)) {
    if (!n) return NULL;
    llnode *ret = malloc(sizeof(llnode));
    ret->next   = llnode_copy(n->next, cpy);
    ret->data   = cpy(n->data);
    return ret;
}
```

```
gl *gl_copy(gl *lst, void *(*cpy)(void *)) {
    gl *ret   = malloc(sizeof(gl));
    ret->head = llnode_copy(lst->head, cpy);
    return ret;
}
```

Testing — main with Polynomials

```
#include "glist.h"
#include "poly.h"
#include <stdio.h>

int cmp_deg(void *a, void *b) {
    struct poly *pa = a;
    struct poly *pb = b;
    return polyDegree(pa) - polyDegree(pb);
}

void *cpy_data(void *a) {
    struct poly *pa = a;
    return polyCopy(pa);
}

void freeData(void *d) {
    struct poly *p = d;
    polyDelete(p);
}

int main(void) {
    struct gl *lst1 = gl_Create();

    // create 3x^4 + 2.5x^2 - 10
    struct poly *p = polyCreate();
    p = polySetCoeff(p, 4,   3.0);
    p = polySetCoeff(p, 2,   2.5);
    p = polySetCoeff(p, 0, -10.0);

    // create -5x^2 + 2.5x
    struct poly *p2 = polyCreate();
    p2 = polySetCoeff(p2, 2, -5.0);
    p2 = polySetCoeff(p2, 1,  2.5);

    // create x^8
    struct poly *p3 = polyCreate();
    p3 = polySetCoeff(p3, 8, 1.0);

    // lst1: p3 (head), p2, p1 (tail)  — added front-to-front
    gl_addToFront(lst1, p);
    gl_addToFront(lst1, p2);
    gl_addToFront(lst1, p3);

    int len = gl_length(lst1);
    printf("length of lst1 %d\n", len);
    printf("**********************\n");
    struct poly *tmp;
    tmp = gl_nthElem(lst1, 1);  printf("p3(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst1, 2);  printf("p2(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst1, 3);  printf("p1(1)=%f\n", polyEval(tmp, 1));
    printf("**********************\n");

    // lst2: sorted by degree (ascending) using gl_addInOrder
    // order: p2 (deg 2), p1 (deg 4), p3 (deg 8)
    struct gl *lst2 = gl_Create();
    gl_addInOrder(lst2, polyCopy(p),  cmp_deg);
    gl_addInOrder(lst2, polyCopy(p2), cmp_deg);
    gl_addInOrder(lst2, polyCopy(p3), cmp_deg);

    len = gl_length(lst2);
    printf("length of lst2 %d\n", len);
    printf("**********************\n");

    tmp = gl_nthElem(lst2, 1);  printf("p2(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst2, 2);  printf("p1(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst2, 3);  printf("p3(1)=%f\n", polyEval(tmp, 1));
    printf("**********************\n");

    // lst3 is a deep copy of lst1
    struct gl *lst3 = gl_copy(lst1, cpy_data);

    gl_delete(lst1, freeData);
    gl_delete(lst2, freeData);

    len = gl_length(lst3);
    printf("length of lst3 %d\n", len);
    printf("**********************\n");

    tmp = gl_nthElem(lst3, 1);  printf("p3(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst3, 2);  printf("p2(1)=%f\n", polyEval(tmp, 1));
    tmp = gl_nthElem(lst3, 3);  printf("p1(1)=%f\n", polyEval(tmp, 1));
    printf("**********************\n");

    gl_delete(lst3, freeData);
    return 0;
}
```



```
length of lst1 3
**********************
p3(1)=1.000000
p2(1)=-2.500000
p1(1)=-4.500000
**********************
length of lst2 3
**********************
p2(1)=-2.500000
p1(1)=-4.500000
p3(1)=1.000000
**********************
length of lst3 3
**********************
p3(1)=1.000000
p2(1)=-2.500000
p1(1)=-4.500000
**********************
Press any key to continue...
```

Extra Practice Problems

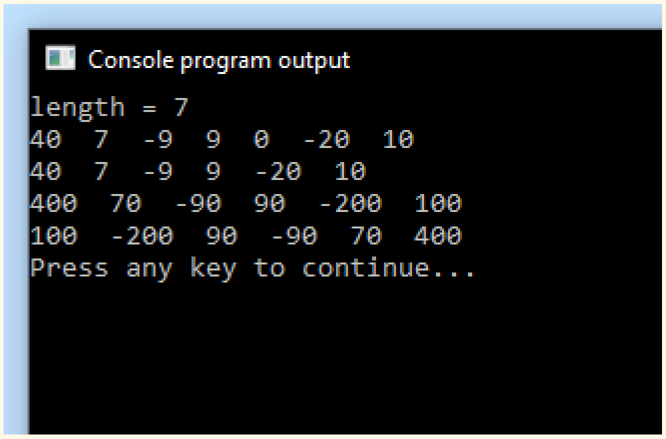
1.

Implement the following interface for a linked list of integers.

```
#ifndef INTLIST_H
#define INTLIST_H

struct lnode;
struct nlst;

// Post: Creates an empty list
struct nlst *listCreate(void);

// Post: Destroys the list
void listDestroy(struct nlst *lst);

// Post: Adds elem to the front of lst
void listAddToFront(struct nlst *lst, int elem);

// Pre:  lst has at least n elements
// Post: Removes the nth element
struct nlst *listRemoveElem(struct nlst *lst, int n);

// Post: Returns the number of elements in lst
int listLength(struct nlst *lst);

// Post: Returns a reversed copy of lst
struct nlst *listReverseCopy(struct nlst *lst);

// Post: Prints the list from start to end
void listPrint(struct nlst *lst);

// Post: Multiplies each element by factor (in place)
void listScaleFactor(struct nlst *lst, int factor);

#endif
```

Sample test program:

```
#include "intlist.h"
#include <stdio.h>

int main(void) {
    struct nlst *lst1 = listCreate();
    listAddToFront(lst1, 10);
    listAddToFront(lst1, -20);
    listAddToFront(lst1, 0);
    listAddToFront(lst1, 9);
    listAddToFront(lst1, -9);
    listAddToFront(lst1, 7);
    listAddToFront(lst1, 40);
    printf("length = %d\n", listLength(lst1));
    listPrint(lst1);
    listRemoveElem(lst1, 5);
    listPrint(lst1);
    listScaleFactor(lst1, 10);
    listPrint(lst1);
    struct nlst *lst2 = listReverseCopy(lst1);
    listDestroy(lst1);
    listPrint(lst2);
    listDestroy(lst2);
    return 0;
}
```

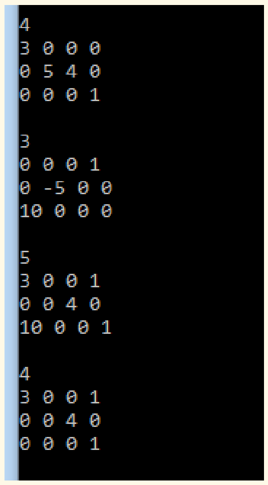

Console program output

```
length = 7
40  7  -9  9  0  -20  10
40  7  -9  9  -20  10
400  70  -90  90  -200  100
100  -200  90  -90  70  400
Press any key to continue...
```

2.

A sparse matrix contains very few non-zero elements. Representing it as a 2D array wastes memory. Complete the following program that represents a sparse matrix using a linked list of non-zero cells. Check for memory leaks using valgrind.

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

// Given — do not change
typedef struct cell {
    int i, j;
    int value;
    struct cell *next;
} cell;

typedef struct sparsematrix {
    cell *head;
    int rows, cols;
} sparsematrix;

// Given — do not change
cell *createCell(int i, int j, int value) {
    cell *c  = malloc(sizeof(cell));
    c->i     = i; c->j = j; c->value = value; c->next = NULL;
    return c;
}

sparsematrix *createMatrix(int rows, int cols) {
    sparsematrix *m = malloc(sizeof(sparsematrix));
    m->head = NULL; m->rows = rows; m->cols = cols;
    return m;
}

void printm(const sparsematrix *m) {
    int i, j, val;
    for (i = 0; i < getNumberOfRows(m); i++) {
        for (j = 0; j < getNumberOfCols(m); j++) {
            val = getValue(m, i, j);
            printf("%d ", val);
        }
        printf("\n");
    }
    printf("\n");
}

int MatrixLength(sparsematrix *sm) {
    int len = 0;
    cell *p = sm->head;
    for (; p; p = p->next) len++;
    return len;
}

// --- TO COMPLETE ---
void DestroyMatrix(sparsematrix *sm)   { }
void removeCell(sparsematrix *sm, cell *c) { }
int  getNumberOfRows(const sparsematrix *sm) { }
int  getNumberOfCols(const sparsematrix *sm) { }
cell *find(const sparsematrix *sm, int i, int j) { }
int  getValue(const sparsematrix *sm, int i, int j) { }
void setValue(sparsematrix *sm, int i, int j, int val) { }
sparsematrix *add(const sparsematrix *a, const sparsematrix *b) { }

int main(void) {
    sparsematrix *a = createMatrix(3, 4);
    setValue(a, 0, 0, 3); setValue(a, 1, 1, 5);
    setValue(a, 1, 2, 4); setValue(a, 2, 3, 1);
    printf("%d\n", MatrixLength(a)); printm(a);

    sparsematrix *b = createMatrix(3, 4);
    setValue(b, 0, 3, 1); setValue(b, 1, 1, -5); setValue(b, 2, 0, 10);
    printf("%d\n", MatrixLength(b)); printm(b);

    sparsematrix *c = add(a, b);
    printf("%d\n", MatrixLength(c)); printm(c);
    setValue(c, 2, 0, 0);
    printf("%d\n", MatrixLength(c)); printm(c);

    DestroyMatrix(a); DestroyMatrix(b); DestroyMatrix(c);
    return 0;
}
```



```
4
3 0 0 0
0 5 4 0
0 0 0 1

3
0 0 0 1
0 -5 0 0
10 0 0 0

5
3 0 0 1
0 0 4 0
10 0 0 1

4
3 0 0 1
0 0 4 0
0 0 0 1
```

3.

Using linked lists, create a program to manage a music playlist.

Each song node contains: title (string), artist (string), duration (int, seconds), next pointer.

Implement in playlist.c:

CreatePlaylist	— allocate and return an empty playlist
AddSong	— dynamically allocate a new song and add it to the playlist
RemoveSong	— remove a song by title
FindSong	— find and return a pointer to a song by title
PrintPlaylist	— print all song details
FreePlaylist	— free all allocated memory

```c
#ifndef PLAYLIST_H
#define PLAYLIST_H

typedef struct Song {
    char        *title;
    char        *artist;
    int          duration;    // seconds
    struct Song *next;
} Song;

typedef struct Playlist {
    Song *head;
} Playlist;

Playlist *CreatePlaylist(void);
void AddSong(Playlist *playlist, const char *title,
             const char *artist, int duration);
void RemoveSong(Playlist *playlist, const char *title);
Song *FindSong(Playlist *playlist, const char *title);
void PrintPlaylist(Playlist *playlist);
void FreePlaylist(Playlist *playlist);

#endif
```

```c
// Example usage:
```

```
int main(void) {
    Playlist *myPlaylist = CreatePlaylist();
    AddSong(myPlaylist, "Bohemian Rhapsody", "Queen",       355);
    AddSong(myPlaylist, "Imagine",           "John Lennon", 187);
    PrintPlaylist(myPlaylist);
    RemoveSong(myPlaylist, "Imagine");
    PrintPlaylist(myPlaylist);
    FreePlaylist(myPlaylist);
    return 0;
}
```

4.

A linked list is "balanced" if, for every node, the sum of all values to its left equals the sum of all values to its right (excluding the node itself), and the left and right sub-sequences also satisfy this property.

Write:
  int isBalanced(struct ListNode *head);

Returns 1 if balanced, 0 otherwise.

struct ListNode { int value; struct ListNode *next; };

5.

Given a linked list sorted in ascending order, write:
  void delete_duplicates(struct ll *list)
that modifies the list in place to remove ALL nodes whose value appears more than once, leaving only values that appear exactly once. The result must remain sorted.

Also implement: create_list, print_list, destroy_list.

```
Example 1:  Input: 1 2 3 3 4 4 5   =>  Output: 1 2 5
Example 2:  Input: 1 1 1 2 3       =>  Output: 2 3
```

6.

Using the standard ll / llnode definition, implement:
  void reverse_list_iter(struct ll *list)   — iterative reversal
  void reverse_list_recur(struct ll *list) — recursive reversal

Both must update list->head to point to the new first node.

```
Example:
  Input:    1 2 3 4 5
  After iterative reverse:   5 4 3 2 1
  After recursive re-reverse: 1 2 3 4 5
```

7.

Given a singly linked list of gift nodes, each with a gift ID (int) and category (char: 'T'=Toys, 'B'=Books, 'E'=Electronics), sort the list so that all Toys come first, then Books, then Electronics.

The relative order of gifts within each category must be preserved (stable sort).

Implement this in holiday_gift.c.

8.

Design and implement a course management system using linked lists.

A Student contains: username, student ID, section number, 10 assignment grades, 3 midterm grades, final grade.
Final grade = 40% assignments + 20% per midterm.

An Instructor contains: name, section number (teaches one section).

A Course contains: a list of students, a list of instructors.

Implement (you decide the function signatures and assumptions):
a) creating a course
b) destroying a course
c) adding a student (given ID, username, section)
d) adding an instructor (given name, section)
e) removing a student (given ID)
f) adding an assignment grade (given username, assignment number, grade)
g) adding a midterm grade (given username, midterm number, grade)
h) calculating and updating a student's final grade (given username)
i) printing all students' usernames and final grades sorted by username
j) printing usernames of students who failed a given section
k) any additional functions you find useful

Testing is critical — test all cases and check for memory leaks.

9.

Using the same ll / llnode definition as in this chapter, write functions for:
1) creating a list of integers
2) adding an integer to the start / end
3) removing an integer from the start / end
4) removing all duplicates
5) sorting the list
6) printing the list
7) checking if the list is sorted
8) reversing the list
9) destroying the list
10) searching for an integer in the list

10.

As in problem 9, but make each node contain two pointers:
one to the next node, and one to the previous node (doubly linked list).

11.

Implement the Sequence ADT — an ADT similar to an array that can grow, shrink, and support insertion/removal at any index.

Example: sequence [42,10,27,31,99], insert 28 at index 2 => [42,10,28,27,31,99].

a) Implement using a singly linked list. [seqlist.c]

b) Implement using a dynamic array with the doubling strategy.
Grow when full (double the size); shrink when less than 25% filled. [seqarray.c]

c) Implement using a dynamic array with LAZY DELETION.
- remove_at(pos): marks item as DELETED in O(1) — does not remove immediately.
- insert_at: removes all DELETED items, then inserts the new item.
- item_at / delete_at: pos is the sequence index with DELETED items excluded.
These operations run in O(d) where d is the number of pending deletions.
- Use INT_MIN to mark DELETED (valid ints are in range [INT_MIN+1, INT_MAX]).

Example of lazy deletion on [42, 10, 27, 31, 99]:

```
remove_at(1)       =>  [42, DELETED, 27, 31, 99]
remove_at(1)       =>  [42, DELETED, DELETED, 31, 99]
length returns 3, item_at(2) returns 99
insert_at(1, 22)    =>  [42, 22, 31, 99]
```

Testing is critical — test all cases and check for memory leaks.