CS 137

Linked Lists

Fall 2025

Victoria Sakhnini

Table of Contents

Introduction

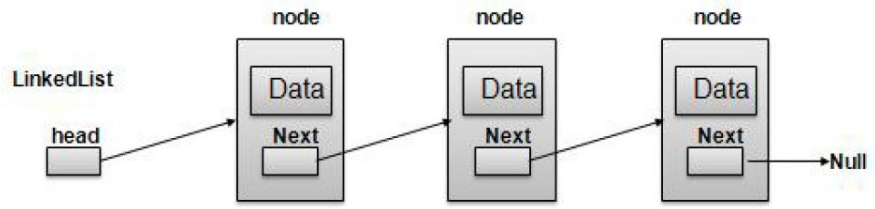
A linked list is a typical data structure where the data is in individual pieces chained together, and it is easy to insert new elements.

A linked list consists of:

1) A piece of data (I'll use an integer for now)
2) A pointer to the next Linked List element, or `NULL` if it is the last element in the list.

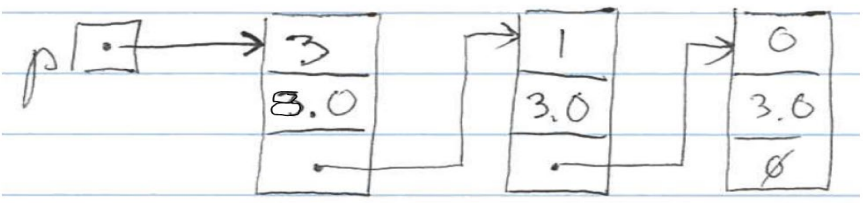
Syntax:

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



source: https://www.tutorialspoint.com/data_structures_algorithms/linked_lists_algorithm.htm

The following is a picture for $p=8x^3+3x+3$



Example: Polynomial

Interface:

```
1. #ifndef POLY_H
2. #define POLY_H
3.
4.
5. // Order polynomial, so the largest degree
6. // is at the beginning
7.
8. struct polynode;
9. struct poly;
10.
11.  //Pre: None
12.  //Post: Creates a null polynomial
13.  struct poly *polyCreate(void);
14.
15.  //Pre: *p is a valid polynomial (even null)
16.  //Post: Destroys the polynomial
17.  void polyDelete(struct poly *p);
18.
19.  //Pre: poly *p is valid
20.  //Post: Returns p(x)
21.  double polyEval(struct poly *p, double x);
22.
23.  //Pre: poly *p is valid; deg is non-negative
24.  //Post: Sets the coefficient at degree to be coeff
25.  //Assume: deg not in poly
26.  struct poly *polySetCoeff(struct poly *p, int deg, double coeff);
27.
28.  //Pre: poly *p is valid
29.  //Post: returns largest nonzero entry in poly
30.  int polyDegree(struct poly *p);
31.
32.  //Pre: poly *p is valid
33.  //Post: returns a polynomial copy of it.
34.  struct poly *polyCopy(struct poly *p);
35.
36.  #endif
```

Implementation:

```
1. #include "poly.h"
2. #include <math.h>
3. #include <stdlib.h>
4. #include <assert.h>
5.
6. // Order polynomial so the largest degree
7. // is at the beginning. Need degree,
8. // coefficient, and pointer to the next term.
9.
10.  typedef struct polynode {
11.          int deg;
12.          double coeff;
13.          struct polynode *next;
14.  } polynode;
15.
16.  typedef struct poly {
17.          struct polynode *head;
18.  } poly;
19.
20.  //empty poly has the head pointing to nothing (Null)
21.  poly *polyCreate(void){
22.          poly *p = malloc(sizeof(poly));
23.          p->head = NULL;
24.          return p;
25.  }
26.
27.  /*
28.  The basic idea of polyDelete:
29.  curnode points to the node to be deleted/freed
30.  nextnode points to the next node to be deleted/freed.
31.  after you free all the data/nodes then, you free the pointer to poly
32.  We make sure we don't break the chain to avoid losing
33.  access to nodes and
34.  not being able to free them.
35.  */
36.
37.  void polyDelete(poly *p){
38.          polynode *nextnode = p->head;
39.          polynode *curnode = NULL;
40.          while (nextnode)
41.          {
42.                  curnode = nextnode;
43.                  nextnode = nextnode->next;
44.                  free(curnode);
45.          }
46.          free(p);
47.  }
48.
49.
```

```
50.  double polyEval(poly *p, double x){
51.          double f = 0.0;
52.          polynode *node = p->head;
53.          // iterate over the nodes (items) until the node is NULL and
54.          // evaluate each appropriately
55.          for (; node; node = node->next)
56.                  f += pow(x, node->deg) * (node->coeff);
57.          return f;
58.  }
59.
60.  poly *polySetCoeff(poly *p, int deg, double coeff){
61.          if (!coeff)
62.                  return p;
63.          polynode *node = p->head;
64.          if (!node || deg > node->deg){
65.             // add to front, allocate space to a new node
66.                  polynode *r = malloc(sizeof(polynode));
67.                  r->coeff = coeff;
68.                  r->deg = deg;
69.                  r->next = node;
70.                  p->head = r;
71.                  return p;
72.          }
73.          //Find the right place to add the node to keep the order of the degrees
74.          polynode *cur = p->head;
75.          for (; cur->next && cur->next->deg > deg; cur = cur->next) ;
76.          if (cur->next && cur->next->deg == deg){
77.                  cur->next->coeff = coeff;
78.          }
79.          else{
80.                  polynode *r = malloc(sizeof(polynode));
81.                  r->coeff = coeff;
82.                  r->deg = deg;
83.                  r->next = cur->next;
84.                  cur->next = r;
85.          }
86.          return p;
87.  }
88.
89.  int polyDegree(poly *p){
90.          assert(p);
91.          return p->head->deg;
92.  }
93.
94.  poly *polyCopy(poly *p){
95.          poly *q = polyCreate();
96.          polynode *node = p->head;
97.          while (node){
98.                  q = polySetCoeff(q, node->deg, node->coeff);
99.                  node = node->next;
100.         }
101.         return q;
102. }
```

Testing:

```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);

          // Print results for first poly
          printf("p(0)=%.2f\n", polyEval(p, 0));  //output: p(0)=-10.00
          printf("p(1)=%.2f\n", polyEval(p, 1));  //output: p(1)=-4.50
          printf("degree=%d\n", polyDegree(p));   //output: degree=4

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

          //Delete the first poly
          // Important to free all allocated memory to avoid memory leaks.
          polyDelete(p);

          // Print results for the copy poly. Same output as above.
          printf("p2(0)=%.2f\n", polyEval(p2, 0));
          printf("p2(1)=%.2f\n", polyEval(p2, 1));
          printf("degree=%d\n", polyDegree(p2));

          // Delete the second poly
          // Important to free all allocated memory to avoid memory leak.
          polyDelete(p2);
          return 0;
 }
```

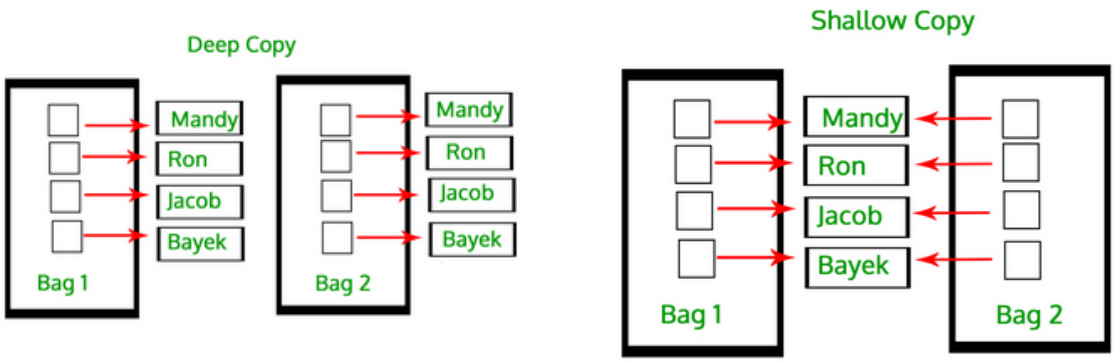
Generalized Lists

We discussed 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 a whole new structure with its own functions? Ideally, not; how can we create a linked list type that stores anything? (Pay attention to using `void *` to generalize the list)

Observation: a `void *` can point at anything. This is how generalizations are often done in `C`, with parameters of type `void *`.

Deep Copy

Copying lists is a common thing to do. One can make a deep or shallow copy when creating copies of arrays or objects. In a Shallow copy, we create a new variable that shares the reference of the original object. In a Deep copy, we create a real copy that doesn't share any references with the original list.
Example:



To make a deep copy, we must create a new node for each node in the list we'd like to copy. If the data are also pointers, we might want to deep copy those (rather than just copying pointers, so the nodes share data). This means that to do so with our generalized list, we need to take in a function from the user to copy the data. It's most straightforward to work with linked lists recursively, so we'll write a helper function that deep copies an individual node (and therefore also copies its next).

The code for a generalized list and deep copy.

```
1. Interface:
2. #ifndef GLIST_H
3. #define GLIST_H
4.
5. struct llnode;
6. struct gl;  //general list
7.
8. //Post: Creates an empty list
9. struct gl *gl_Create(void);
10.
11.  //Pre: elem is a pointer to a heap
12.  // allocated data that this list now owns.
13.  // Adds a new node storing elem at the front
14.  // of our list.
15.  void gl_addToFront(struct gl *lst, void *elem);
16.
17.  // Pre: lst is already sorted wrt to cmp.
18.  // Adds a node storing elem to lst in sorted order
19.  // wrt to cmp.
20.  // cmp(a,b) returns:
21.  // <0 if a should be before b
22.  // ==0 if a == b
23.  // >0 if a should be after b
24.  void gl_addInOrder(struct gl *lst, void *elem,
25.                                        int (*cmp )(void *, void *));
26.
27.  //Pre: *lst is a valid lst
28.  //Post: Destroys the list
29.  void gl_delete(struct gl *lst, void (*kill)(void *));
30.
31.  //Pre: lst is valid
32.  //Post: returns the length of lst
33.  int gl_length(struct gl *lst);
34.
35.  //Pre: lst is valid
36.  //Post: returns pointer to nth data
37.  // Assume n is not bigger than the length of the list
38.  void *gl_nthElem(struct gl *lst, int n);
39.
40.  // cpy provided by the user
41.  // deep copy
42.  struct gl *gl_copy (struct gl *lst, void *(*cpy)(void *));
43.
44.  #endif
```

Implementation:

```
1. #include "glist.h"
2. #include <stdlib.h>
3. #include <assert.h>
4.
5. typedef struct llnode {
6.          void *data;
7.          struct llnode *next;
8. } llnode;
9.
10.  typedef struct gl {
11.         llnode *head;
12.  } gl;
13.
14.  gl *gl_Create(void){
15.         gl *ret = malloc(sizeof(gl));
16.         ret->head = NULL;
17.         return ret;
18.  }
19.
20.  void gl_addToFront(gl *lst, void *elem){
21.         llnode *newNode = malloc(sizeof(llnode));
22.         newNode->data = elem;
23.         newNode->next = lst->head;
24.         lst->head = newNode;
25.  }
26.
27.  // cmp(a,b) returns:
28.  // <0 if a should be before b
29.  // ==0 if a == b
30.  // >0 if a should be after b
31.  void gl_addInOrder(gl *lst, void *elem, int (*cmp) (void *, void *)){
32.         llnode *prev = NULL;
33.         llnode *cur = lst->head;
34.
35.         // After this loop (one line), we want to insert elem
36.         // right after prev, right before cur.
37.         for (; cur && cmp(elem, cur->data) > 0; prev = cur, cur = cur->next) ;
38.
39.         // Check if we must update head
40.         if (!prev)
41.                 gl_addToFront(lst, elem);
42.         else{
43.                 llnode *n = malloc(sizeof(llnode));
44.                 n->data = elem;
45.                 n->next = cur;
46.                 prev->next = n;
47.         }
48.  }
49.
```

```
50.  void gl_delete(gl *lst, void (*kill)(void *)){
51.          llnode *nextnode = lst->head;
52.          llnode *curnode = NULL;
53.          while (nextnode){
54.                  curnode = nextnode;
55.                  nextnode = nextnode->next;
56.                  kill(curnode->data);
57.                  free(curnode);
58.          }
59.          free(lst);
60.  }
61.
62.  int gl_length(gl *lst){
63.          int len = 0;
64.          llnode *node = lst->head;
65.          // iterate over the nodes (items) until the node is NULL
66.          // and count
67.          for (; node; node = node->next)
68.                  len++;
69.          return len;
70.  }
71.
72.  void *gl_nthElem(struct gl *lst, int n){
73.          assert(n <= gl_length(lst));
74.          int num;
75.          llnode *node = lst->head;
76.          for (num = 1; num < n; num++)
77.                  node = node->next;
78.          return node->data;
79.  }
80.
81.  llnode *llnode_copy(llnode *n, void *(*cpy) (void *)){
82.          if (!n)
83.                  return NULL;
84.          llnode *ret = malloc(sizeof(llnode));
85.          ret->next = llnode_copy(n->next, cpy);
86.          ret->data = cpy(n->data);
87.          return ret;
88.  }
89.
90.  gl *gl_copy(gl *lst, void *(*cpy) (void *)){
91.          gl *ret = malloc(sizeof(gl));
92.          ret->head = llnode_copy(lst->head, cpy);
93.          return ret;
94.  }
```

Testing:

```
1. #include "glist.h"
2. #include "poly.h"
3. #include <stdio.h>
4.
5. int cmp_deg(void *a, void *b){
6.   struct poly *pa = a;
7.   struct poly *pb = b;
8.   int ia = polyDegree(pa);
9.   int ib = polyDegree(pb);
10.  return ia - ib;
11.}
12.
13.void *cpy_data(void *a){
14.  struct poly *pa = a;
15.  struct poly *q = polyCopy(pa);
16.  return q;
17.}
18.
19.void freeData(void *d){
20.  struct poly *p = d;
21.  polyDelete(p);
22.}
23.
24.int main(void){
25.  struct gl *lst1 = gl_Create();
26.
27.  // create 3x^4 + 2.5x^2 - 10
28.  struct poly *p = polyCreate();
29.  p = polySetCoeff(p, 4, 3.0);
30.  p = polySetCoeff(p, 2, 2.5);
31.  p = polySetCoeff(p, 0, -10.0);
32.
33.  // create -5x^2 + 2.5x
34.  struct poly *p2 = polyCreate();
35.  p2 = polySetCoeff(p2, 2, -5.0);
36.  p2 = polySetCoeff(p2, 1, 2.5);
37.
38.  // create x^8
39.  struct poly *p3 = polyCreate();
40.  p3 = polySetCoeff(p3, 8, 1.0);
41.
42.  // create a list of three polys
43.  // list1 will include p3 (first node), p2,
44.  // p1 (last node) in this specific order
45.
46.  gl_addToFront(lst1, p);
47.  gl_addToFront(lst1, p2);
48.  gl_addToFront(lst1, p3);
49.  int len = gl_length(lst1);
```

```
50.  printf("length of lst1 %d\n", len);
51.  printf("**********************\n");
52.
53.  struct poly *tmp;
54.  tmp = gl_nthElem(lst1, 1);
55.  printf("p3(1)=%f\n", polyEval(tmp, 1));
56.  tmp = gl_nthElem(lst1, 2);
57.  printf("p2(1)=%f\n", polyEval(tmp, 1));
58.  tmp = gl_nthElem(lst1, 3);
59.  printf("p1(1)=%f\n", polyEval(tmp, 1));
60.  printf("**********************\n");
61.
62.  struct gl *lst2 = gl_Create();
63.  // list2 will include p2 (first node), p1,
64.  // p3 (last node) in this specific order
65.  gl_addInOrder(lst2, polyCopy(p), cmp_deg);
66.  gl_addInOrder(lst2, polyCopy(p2), cmp_deg);
67.  gl_addInOrder(lst2, polyCopy(p3), cmp_deg);
68.
69.  len = gl_length(lst2);
70.  printf("length of lst2 %d\n", len);
71.  printf("**********************\n");
72.
73.  tmp = gl_nthElem(lst2, 1);
74.  printf("p2(1)=%f\n", polyEval(tmp, 1));
75.  tmp = gl_nthElem(lst2, 2);
76.  printf("p1(1)=%f\n", polyEval(tmp, 1));
77.  tmp = gl_nthElem(lst2, 3);
78.  printf("p3(1)=%f\n", polyEval(tmp, 1));
79.  printf("**********************\n");
80.
81.  // lst3 is a clean copy of lst1
```

```
82.  struct gl *lst3 = gl_copy(lst1, cpy_data);
83.
84.  gl_delete(lst1, freeData);
85.  gl_delete(lst2, freeData);
86.
87.  len = gl_length(lst3);
88.  printf("length of lst3 %d\n", len);
89.  printf("**********************\n");
90.
91.  tmp = gl_nthElem(lst3, 1);
92.  printf("p3(1)=%f\n", polyEval(tmp, 1));
93.  tmp = gl_nthElem(lst3, 2);
94.  printf("p2(1)=%f\n", polyEval(tmp, 1));
95.  tmp = gl_nthElem(lst3, 3);
96. printf("p1(1)=%f\n", polyEval(tmp, 1));
97. printf("**********************\n");
98.
99. gl_delete(lst3, freeData);
100.
101. return 0;
102. }
103.
```

```
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

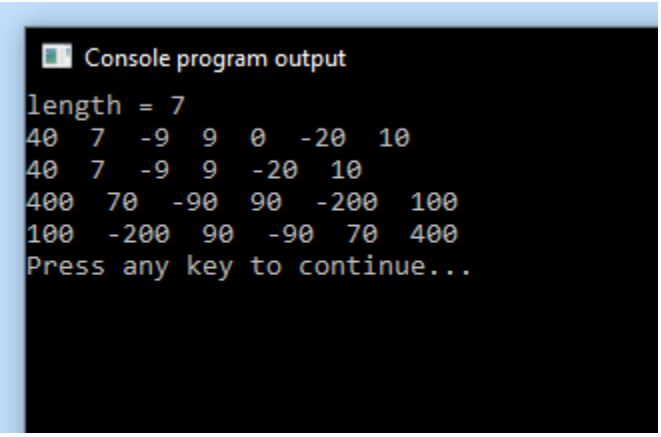
1) Write an implementation to the following Interface:

```
1. #ifndef INTLIST_H
2. #define INTLIST_H
3.
4. struct lnode;
5. struct nlst;
6.
7. //Pre: None
8. //Post: Creates a null intlst
9. struct nlst *listCreate(void);
10.
11.  //Pre: lst is a valid list ( even null )
12.  //Post: Destroys the list
13.  void listDestroy(struct nlst *lst);
14.
15.
16.  // lst is a valid list
17.  void listAddToFront(struct nlst *lst, int elem);
18.
19.  //Pre: lst is a valid list with a length of at least n
20.  //Post: nth item removed
21.  struct nlst *listRemoveElem(struct nlst *lst, int n);
22.
23.  //Pre: lst is valid
24.  //Post: returns the number of elements in the list
25.  int listLength(struct nlst *lst);
26.
27.  //Pre: lst is valid
28.  //Post: returns a reversed copy of it.
29.  struct nlst *listReverseCopy(struct nlst *lst);
30.
31.
32.  //Pre: lst is valid
33.  //prints the items in the list from start to end.
34.  void listPrint(struct nlst *lst);
35.
36.
37.  //Pre: lst is valid
38.  //mutates a list after multiplying each element by an int factor.
39.  void listScaleFactor(struct nlst *lst, int factor);
40.
41.
42.  #endif
```

Here is a sample program and expected results:

```c
#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;
  }
```



2) A sparse matrix is a matrix that contains very few nonzero elements. When a sparse matrix is represented by a 2-dimensional array, we waste a lot of space to represent that matrix.
Complete the following program that represents a Spare matrix in a more efficient way in terms of space/memory:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. // Given. Do not change
5.
6. typedef struct cell {
7.   int i, j;
8.   int value;
9.   struct cell *next;
10.  } cell;
11.
12.  typedef struct sparsematrix {
13.          cell *head;
14.          int rows, cols;
15.  } sparsematrix;
16.
17.  //create a single cell
18.  cell *createCell(int i, int j, int value){
19.          cell *c = malloc(sizeof(cell));
20.          c->i = i;
21.          c->j = j;
22.          c->value = value;
23.          c->next = NULL;
24.          return c;
25.  }
26.
27.  //create a matrix
28.  sparsematrix *createMatrix(int rows, int cols){
29.          sparsematrix *m = malloc(sizeof(sparsematrix));
30.          m->head = NULL;
31.          m->rows = rows;
32.          m->cols = cols;
33.          return m;
34.  }
35.
36.  //print a matrix
37.  void printm(const sparsematrix *m){
38.          int i, j, val;
39.          for (i = 0; i < getNumberOfRows(m); i++){
40.                  for (j = 0; j < getNumberOfCols(m); j++){
41.                          val = getValue(m, i, j);
42.                          printf("%d ", val);
43.                  }
```

```
                printf("\n");
        }
        printf("\n");
}

int MatrixLength(sparsematrix *sm){
        int len = 0;
        cell *p = sm->head;
        // iterate over the nodes (items) until node is NULL
        // and count
        for (; p; p = p->next)
                len++;
        return len;
}


// To Complete
void DestroyMatrix(sparsematrix *sm){

}

//remove the specified cell from matrix
void removeCell(sparsematrix *sm, cell *c){

}

// get the number of rows in a matrix
int getNumberOfRows(const sparsematrix *sm){

}

// get the number of cols in a matrix
int getNumberOfCols(const sparsematrix *sm){

}

// returns a pointer to cel[i,j] otherwise NULL
cell *find(const sparsematrix *sm, int i, int j){

}

// returns the value in cell[i,j]
int getValue(const sparsematrix *sm, int i, int j){

}

//set a value in [i,j]. Be careful of different cases.
// review the test cases
void setValue(sparsematrix *sm, int i, int j, int val){

}

```

```
96.  // add matrix a and matrix b of the same size.
97.  sparsematrix *add(const sparsematrix *a, const sparsematrix *b){
98.
99.  }
100.
101. // For testing. Make sure to check for memory leaks using valgrind.
102.
103. int main(void){
104.         sparsematrix *a = createMatrix(3, 4);
105.         setValue(a, 0, 0, 3);
106.         setValue(a, 1, 1, 5);
107.         setValue(a, 1, 2, 4);
108.         setValue(a, 2, 3, 1);
109.         printf("%d\n", MatrixLength(a));
110.         printm(a);
111.
112.         sparsematrix *b = createMatrix(3, 4);
113.         setValue(b, 0, 3, 1);
114.         setValue(b, 1, 1, -5);
115.         setValue(b, 2, 0, 10);
116.         printf("%d\n", MatrixLength(b));
117.         printm(b);
118.         sparsematrix *c = add(a, b);
119.         printf("%d\n", MatrixLength(c));
120.         printm(c);
121.         setValue(c, 2, 0, 0);
122.         printf("%d\n", MatrixLength(c));
123.         printm(c);
124.         DestroyMatrix(a);
125.         DestroyMatrix(b);
126.         DestroyMatrix(c);
127.         return 0;
128. }
129.
```

The expected output:

```
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 in C, you will create a program to manage a music playlist. This will help you understand the concepts of pointers, dynamic memory allocation, and the manipulation of linked data structures.

Task:

Develop a program that allows users to create and manage a playlist. Each song in the playlist will be represented as a node in a linked list, containing details such as song title, artist, and duration.

Files to Create:

1. **playlist.h**: Contains the structure definitions and function prototypes.
2. **playlist.c**: Contains the implementations of the functions.

Data Structures

- **Song**: A structure that contains information about a song (title, artist, duration).
- **Playlist**: A linked list of Song structures.

Functions to Implement

1. **AddSong**: Dynamically allocate a new song and add it to the playlist.
2. **RemoveSong**: Remove a song from the playlist based on title or artist.
3. **FindSong**: Find and return a pointer to a song in the playlist given a title or artist.
4. **PrintPlaylist**: Print the details of all songs in the playlist.
5. **FreePlaylist**: Free all memory allocated for the playlist to avoid memory leaks.

Example Usage:

```
 1. int main(void) {
 2.     Playlist *myPlaylist = CreatePlaylist();
 3.     AddSong(myPlaylist, "Bohemian Rhapsody", "Queen", 355);
 4.     AddSong(myPlaylist, "Imagine", "John Lennon", 187);
 5.     PrintPlaylist(myPlaylist);
 6.     RemoveSong(myPlaylist, "Imagine");
 7.     PrintPlaylist(myPlaylist);
 8.     FreePlaylist(myPlaylist);
 9.     return 0;
10. }
11.
```

Here is the .h file for you to look at:

```h
#ifndef PLAYLIST_H
#define PLAYLIST_H
// Definition of the Song structure
typedef struct Song {
    char *title;
    char *artist;
    int duration; // duration of the song in seconds
    struct Song *next; // pointer to the next song in the playlist
} Song;
// Definition of the Playlist structure
typedef struct Playlist {
    Song *head; // pointer to the first song in the playlist
} Playlist;
// Functions
Playlist *CreatePlaylist();
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 // PLAYLIST_H

```

4) Consider a mystical realm inhabited by magical linked lists. In this enchanted land, each node of a linked list has an associated value and a pointer to another node. However, these linked lists are not ordinary – they possess a peculiar property.

The property is as follows: A linked list is said to be "balanced" if, for any given node, the sum of the values of all nodes on its left is equal to the sum of the values of all nodes on its right (excluding the node itself). Furthermore, the left and right subtrees must also exhibit this balance property.

You are tasked with writing a C function that checks whether a given linked list is balanced. The following structure represents the linked list:

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

Write a function with the signature:

```
int isBalanced(struct ListNode* head);
```

The function takes the head of a linked list as an argument and returns 1 if the linked list is balanced, and 0 otherwise.

5) You are given a linked list data structure defined as follows:

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

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

The linked list is sorted in ascending order. Write a function `void delete_duplicates(struct ll *list)` that modifies the list in place to remove all nodes with duplicate values, leaving only distinct values in the list. The result should also remain sorted.

Additionally, implement helper functions to support this task:
1. `struct ll *create_list()` – Creates a linked list from input values.
2. `void print_list(const struct ll *list)` – Prints the linked list.
3. `void destroy_list(struct ll *list)` – Frees all memory allocated for the linked list.

Example Input and Output

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) You are provided with the following definitions for a singly linked list and its nodes:

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

Your task is to implement two functions to reverse the linked list: one using an **iterative approach** and another using a **recursive approach**. The goal is to reverse the struct ll linked list such that the order of the elements is completely reversed.

Function Specifications

- **Iterative Version** Implement the function `void reverse_list_iter(struct ll *list)` which:
  - Takes a pointer to a linked list (list) as input.
  - Reverses the linked list iteratively.
  - Updates the head of the list to point to the new first node.
- **Recursive Version** Implement the function `void reverse_list_recur(struct ll *list)` which:
  - Takes a pointer to a linked list (list) as input.
  - Reverses the linked list recursively.
  - Updates the head of the list to point to the new first node.

Input and Output

- **Input:**
  - A series of integers (terminated by EOF) to create the linked list.
- **Output:**
  - Print the original list.
  - Reverse the list iteratively and print the reversed list.
  - Reverse the list recursively and print the reversed list again.

Example Execution

Input:

1 2 3 4 5

Output:

Original list: 1 2 3 4 5
Reversed list: 5 4 3 2 1
Reverse Again: 1 2 3 4 5

7) The local orphanage is hosting its annual holiday gift exchange for their children. They accept gifts that belong to one of three categories: "Toys", "Books", and "Electronics" and assign each gift a unique ID. However, due to a mix-up, the gifts are stored in a linked list and not sorted by category.

They need your help to sort these gifts so that all the gifts in each category are grouped. The linked list must be sorted as follows: All "Toys" come first, then all "Books", and then all "Electronics".

Create a C program named `holiday_gift.c` that has a function to sort the linked list by these categories. Once sorted, the list should maintain the original order of the gifts within each category.

Input:

- A singly linked list where each node contains:
    - A gift ID (int)
    - A category (char) can be 'T' for Toys, 'B' for Books, 'E' for Electronics.

Changed:

- The list should be sorted so that all Toys appear first, then Books, and then Electronics. The order of gifts within each category should remain unchanged.

8) Given the following structure description:

A course is a structure that contains:
a list of students
a list of instructors

A student is a structure that contains:
a student username
a student id
section number
an array of 10 integers for 10 assignments grade
an array of 3 integers for 3 midterms
an integer for the final grade (40% assignments, 20% each midterm)

An instructor is a structure that contains:
name (string), section number.
you may assume that each instructor teaches one section

This is an open question, so you decide on the implementation, the header of each function, the assumptions, etc.
Write functions that support the following:
a) creating a course
b) destroying a course
c) adding a student to the course (given student id, and student username, and section number)
d) adding an instructor to the course(given instructor name, section number)
e) removing a student from the course (given student id)
f) adding assignment grade to a certain student (given student username, assignment number, and the grade)
g) adding a midterm grade to a particular student (given student username, midterm number, and the grade)
h) calculating and updating the final grade of a specific student (given student username), assuming that all grades for assignments and midterms are valid.
i) printing students' usernames and their final grades sorted by username for all sections
j) print students' usernames who failed the course in a given section
k) you can add any function you like for practicing
Testing is critical; test all different cases and test for memory leaks.

9) Write functions to support the following while using the same list definition as in M13:

1) creating a list of integers
2) adding an integer to the start/end of the list
3)removing an integer from the start/end of the list
4) removing all duplicates from the list
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 if an integer is in the list

10) As Q9 but, make each node have two pointers, one to the next node and one to the previous node.

11) In this question, you will complete the Sequence ADT implementation using a dynamic array and a linked list. The Sequence ADT is an ADT that gives a client many features similar to an array. Some differences include:

it can be empty

its size can grow and shrink

items can be inserted at any position

for example, if your sequence is [42,10,27,31,99] and you insert 28 at index 2, your new sequence is [42,10,28,27,31,99].

Similarly, items can be removed from any position.

Tasks:

a) implement the Sequence ADT using a singly linked list.

b) implement the Sequence ADT using a dynamic array in this part. As a reminder, you should follow good programming practices and have an implementation that is as efficient as possible. Your implementation should also be space efficient; i.e. you must use the doubling strategy to grow the array (when it is full, double the size) and should shrink the array if it becomes less than 25% filled. [seqarray.c]

c) In this part, you will re-implement the Sequence ADT using a dynamic array with a lazy deletion strategy instead of the regular deletion. As in the previous part, your implementation should be time- and space-efficient.

In lazy deletion, instead of removing the item immediately from the array, when the remove function is called in O(n) time, lazy deletion marks the item as DELETED in O(1) time. The actual removal of the item from the array is performed when the next insert operation is called. However, the remove function may be called many times before the next insert operation.

HINT: You should create a dynamic array or linked list field in the sequence structure to store the index locations of the items pending deletion.

Note: The Interface for this part is the same as the parts above; i.e. the client is not exposed to any changes in the implementation.

For operations with a sequence index parameter pos, such as delete_at and item_at, pos refers to the sequence index where all DELETED items have been removed (even though these operations will not remove the items). The implementation of these functions has a runtime of O(d), where d is the number of items pending deletion, to calculate the adjusted index for use in the internal array data structure.

The insert operation will remove all DELETED items and insert the new item.

For example, given an initial sequence of [42, 10, 27, 31, 99],

remove_at(1) results in the sequence [42, DELETED, 27, 31, 99],

calling remove_at(1) again results in [42, DELETED, DELETED, 31, 99],

length returns 3 and item_at(2) returns 99,

finally, insert_at(1, 22) results in the sequence [42, 22, 31, 99].

Note: For simplicity, we will use INT_MIN to mark an item as DELETED so valid ints stored in the sequence will be in the range [INT_MIN+1, INT_MAX].

Testing is critical; test all different cases.