ECE264 Spring 2016
Section 2 (830-920AM MWF)
Final Exam, 1030AM-1230PM, May 7

Please remove the first sheet and return only the first sheet.

In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

Signature:

You must sign here. Otherwise you will receive a **1-point** *penalty.*

Read the questions carefully.
Some questions have conditions and restrictions.

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may **not** borrow books from other students.

This exam tests four learning objectives:

- Structure (Question 1)
- Structure, Dynamic Structure (Questions 2 and 3)
- Recursion (Question 3)
- File (Question 4)

You must obtain 50% or more points in the corresponding question to pass the learning objective.

Contents

Learning Objective

File	Pass	Fail
Recursion	Pass	Fail
Structure	Pass	Fail
Dynamic Structure	Pass	Fail

Q1		Q2		Q3		Q4	
1.		1.		1.		1.	
2.		2.		2.		2.	
3.		3.		3.		3.	
4.		4.		4.		4.	
5.		5.		5.		5.	
6.		6.		6.		-	
7.		-					
8.		-					

The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	`
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	'	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

1 Structure (4 points)

ASCII uses 8 bits to store each character. This is inefficient if only decimal digits (0, 1, 2, ..., 9) are needed. For storing one decimal digit, only 4 bits are necessary. This question asks you to implement a structure called `DecPack`. This structure has three attributes:

- `size`: the maximum number of decimal digits to be stored in a `DecPack` object
- `used`: the actual number of decimal digits already stored in a `DecPack` object
- `data`: an array of `unsigned short`. Each element has two bytes and stores **four** decimal digits. The four digits start from the lowest 4 bits to the highest 4 bits. In other words, if only decimal digit is stored, only the lower 4 bits are used. The other bits are zero.

Please select the answers for

`<--- FILL CODE --->`

```c
#ifndef DECPACK_H
#define DECPACK_H
typedef struct
{
  int size; // how many digits can be stored
  int used; // how many digits are actually stored
  unsigned short * data; // store the digits
  // each element has 2 bytes and can store four digits
} DecPack;

// create a DecPack object with the given size.  initialize all
// elements to zero.  set used to zero

DecPack * DecPack_create(int sz);

// insert a decimal value to a DecPack object, pointed by dp
// val must be between 0 and 9
//
// used increments by one if there is still space
//
// The function returns
//     0 if the value has been inserted successfully
//    -1 otherwise
//
// Please remember that each unsigned short can store 4
// decimal digits

int DecPack_insert(DecPack * dp, unsigned char val);
```

```c

// read an element specified by ind
//
// if dp is NULL, return -1
// if ind is invalid (negative or greater than used, return -1)
// otherwise, return the decimal value (must be between 0 and 9)

int DecPack_read(DecPack * dp, int ind);

// print the elements
// format: (index, value)
// The indexes should start from zero to used - 1
// The values should be between '0' and '9'

void DecPack_print(DecPack * dp);

// release the memory

void DecPack_destroy(DecPack * dp);

#endif
```

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

// create a DecPack object with the given size

DecPack * DecPack_create(int sz)
{
  // allocate memory for DecPack
  // <--- FILL CODE --->
  // Q1
  A. DecPack * dp = malloc(sizeof(DecPack));
  B. DecPack * dp = malloc(sizeof(int));
  C. DecPack * dp = malloc(sizeof(unsigned short));
  D. DecPack * dp = malloc(sizeof(unsigned char));
  E. DecPack * dp = malloc(sizeof(char));
  F. DecPack * dp = malloc(sizeof(double));
  G. DecPack * dp = malloc(sizeof(DecPack *));
  H. None of the above

```

```
  // check whether allocation fails
  if (dp == NULL) { return NULL; }

  // initialize size to sz and used to 0
  dp -> size = sz;
  dp -> used = 0;

  // allocate memory for data, should be sz/4 because each
  // element can store four digits

  int numelem = sz / 4;
  // if sz is not a multiple of 4, increment numelem by one
  if ((sz % 4) != 0) { numelem ++; }

  // <--- FILL CODE --->
  // Q2
  A. dp -> data = malloc(sizeof(DecPack));
  B. dp -> data = malloc(sizeof(unsigned short) * numelem);
  C. dp -> data = malloc(sizeof(unsigned short));
  D. dp -> data = malloc(sizeof(int) * numelem);
  E. dp -> data = malloc(sizeof(unsigned char) * numelem);
  F. dp -> data = malloc(sizeof(char) * numelem);
  G. dp -> data = malloc(sizeof(double));
  H. None of the above

  // check whether allocation fails
  if (dp -> data == NULL)
    {
      free (dp);
      return NULL;
    }

  // initialize all elements to zero
  int ind;
  for (ind = 0; ind < numelem; ind ++)
    { dp -> data[ind] = 0; }

  // return the allocate memory
  return dp;
}

int DecPack_insert(DecPack * dp, unsigned char val)
```

```
{
  // if the object is empty, do nothing
  if (dp == NULL) { return -1; }

  // if val < 0 or val > 9, ignore and do nothing
  // <--- FILL CODE --->
  // Q3
  A. if ((val < ’0’) || (val > ’9’)) { return -1; }
  B. if ((val < 30) || (val > 3A)) { return -1; }
  C. if ((val < 0) || (val > 9)) { return -1; }
  D. if ((val < 30) && (val > 39)) { return -1; }
  E. if ((val < 0X48) || (val > 0X57)) { return -1; }
  F. if ((val < 0X0) || (val > 0X48)) { return -1; }
  G. if ((val < 0) && (val > 9)) { return -1; }
  H. None of the above

  // If the allocated memory is full, the value cannot
  // be inserted
  if ((dp -> used) == (dp -> size))
    { return -1; }

  // find the index of the element
  int ind = (dp -> used) / 4;
  unsigned short toinsert = val;
  // shift it to the right location
  // <--- FILL CODE --->
  // Q4
  A. toinsert >>= ((dp -> used) % 4) * 4;
  B. toinsert <<  ((dp -> used) % 4) * 4;
  C. toinsert <<= ((dp -> used) % 8) * 8;
  D. toinsert <<= ((dp -> used) % 4) * 4;
  E. toinsert >>= ((dp -> used) % 8) * 8;
  F. toinsert <<= ((dp -> used) % 4);
  G. toinsert <<= ((dp -> used) / 4);
  H. None of the above;

  // <--- FILL CODE --->
  // Q5
  A. dp -> data[ind] = toinsert;
  B. dp -> data[ind] &= toinsert;
  C. dp -> data[ind] >>= toinsert;
  D. dp -> data[ind] != toinsert;
```

```
  E. dp -> data[ind] |= toinsert;
  F. dp -> data[ind] = (toinsert << 4);
  G. dp -> data[ind] = (toinsert << ind);
  H. None of the above;

  (dp -> used) ++;
  return 0;
}

int DecPack_read(DecPack * dp, int ind)
{
  if (dp == NULL) { return -1; }
  if (ind >= (dp -> used)) { return -1; }
  if (ind < 0) { return -1; }

  // <--- FILL CODE --->
  // Q6
  A. unsigned short val = dp -> data[ind / 2];
  B. unsigned char  val = dp -> data[ind / 4];
  C. short          val = dp -> data[ind / 4];
  D. char           val = dp -> data[ind / 4];
  E. unsigned short val = dp -> data[ind];
  F. unsigned short val = dp -> data[ind / 4];
  G. unsigned char  val = dp -> data[ind / 2];
  H. None of the above;

  // <--- FILL CODE --->
  // Q7
  A. val <<= (ind % 2) * 2;
  B. val >>= (ind % 3) * 3;
  C. val >>= (ind % 8) * 2;
  D. val >>= (ind % 8) * 4;
  E. val >>  (ind % 2) * 2;
  F. val <<= (ind % 4) * 4;
  G. val >>= (ind % 4) * 4;
  H. None of the above;

  // <--- FILL CODE --->
  // Q8
  A. val &= 0XF;
  B. val != 0XF;
  C. val &= 0X10;
```

```
  D. val |= 0XF;
  E. val ^= 0XF;
  F. val &= 0XFF;
  G. val &= 0XFF00;
  H. None of the above;

  return val;
}

void DecPack_print(DecPack * dp)
{
  // if the object is empty, do nothing
  if (dp == NULL) { return; }
  int iter;
  int used = dp -> used;
  printf("DecPack size = %d, used = %d\n", dp -> size, used);
  // go through every value stored in the data attribute
  for (iter = 0; iter < used; iter ++)
    {
      printf("(%d, %d)\n", iter, DecPack_read(dp, iter));
    }
}

// destroy the whole DecPack object, release all memory
void DecPack_destroy(DecPack * dp)
{
  // if the object is empty, do nothing
  if (dp == NULL) { return; }
  // release the memory for the data
  free (dp -> data);
  // release the memory for the object
  free (dp);
}
```

2 Dynamic Structure and Structure– Linked List (3 points)

For each question, select the correct answer.
Write a function that can take a linked list and sort the nodes' values in the ascending order. This function **must not** allocate additional memory.

```
// --- list.h ---

#ifndef LIST_H
#define LIST_H
typedef struct listnode
{
  struct listnode * next;
  int value;
} Node;

/* sort the nodes' values in the ascending order */
Node * List_sort(Node * head);
#endif
```

```
// --- question.c ---

#include "list.h"
#include <stdlib.h>
#include <stdio.h>

// swap the two values stored in * a and * b
static void swap(int * a, int * b)
{
  // <--- FILL CODE --->
  // Q1
  A.
  int t = a;
  * a = * b;
  * b = t;

  B.
  int t = * a;
  * a = * b;
  * b = t;

  C.
```

```
  int t = * a;
  a = b;
  * b = t;

  D.
  int * t = a;
  a = b;
  b = t;

  E.
  int * t = * a;
  * a = * b;
  b = t;

  F.
  int t = * a;
  * a = * b;
  b = & t;

  G.
  int t = * b;
  a = & t;
  * b = * a;

  H. None of the above
}

// Algorithm: selection sort
// 1. p points to a node, starting from the first node
// 2. save p's value in minval
// 3. q iterates through all nodes after p
// 4.    if q's value is smaller than minval
// 5.       save q's value in minval
// 6.       save q in r
// 7. swap p's value and r's value
// 8. move p to the next node
Node * List_sort(Node * head)
{
  Node * p = head;
  while (p != NULL)
    {
      // <--- FILL CODE --->
```

```
    // Q2
    A. int minval = p;
    B. Node * minval = p -> value;
    C. int minval = p -> value;
    D. Node * minval = p -> next;
    E. unsigned char minval = p -> value;
    F. int minval = (p -> value) & 0XFF;
    G. unsigned char minval = (p -> value) | 0X0F;
    H. Non of the above

    Node * q = p -> next;
    Node * r = p;
    // r should be p, not q,
    while (q != NULL)
      {
        if (minval > (q -> value))
          {
            // <--- FILL CODE --->
            // Q3
            A. minval = p -> value;
            B. minval = q;
            C. minval = (q -> value) & 0XFF;
            D. minval = q -> value;
            E. minval = (p -> value) & 0X0F;
            F. minval = (q -> value) << 1;
            G. minval = (q -> value) >> 1;
            H. None of the above

            // <--- FILL CODE --->
            // Q4
            A. r = p;
            B. r = p -> value;
            C. r = q -> next;
            D. r -> value = q -> value;
            E. r = q;
            F. r = q -> value;
            G. r = (p -> value) & 0X0F;
            H. None of the above
          }
        q = q -> next;
      }
    // <--- FILL CODE --->
```

```
107         // Q5
108         A. swap(& (p -> value), & (q -> value));
109         B. swap(p, q);
110         C. swap(& p, & q);
111         D. swap(p -> value, r -> value);
112         E. swap(p -> value, q -> value);
113         F. swap(& (p -> value), & (r -> value));
114         G. swap(p, r);
115         H. None of the above
116
117         p = p -> next; // move to the next node
118       }
119     return head;
120   }
```

Q6
The program has the following statement

```
Node * r = p;
```

at line 76. What may happen if this statement is changed to

```
Node * r = q;
```

A. The function may enter an infinite loop.

B. The function becomes recursive (i.e., the function calls itself).

C. The function may sort the nodes' values in the descending order.

D. The function **always** has Segmentation Fault.

E. The function **sometimes but not always** has Segmentation Fault.

F. The order of the nodes' values may be unchanged, i.e., **sometimes** this function returns the original list.

G. This function can sort the list only when all values are distinct. If the same value appears more than once, this function is unable to sort the values.

H. None of the above.

3 Recursion and Dynamic Structure– Binary Tree (6 points)

Write a function that calculate the numbers of nodes at specific distances to the root.

```
// tree.h
#ifndef TREE_H
#define TREE_H
#include <stdio.h>
typedef struct treenode
{
  struct treenode * left;
  struct treenode * right;
  int value;
} TreeNode;

// find the numbers of nodes at specific distances to the root
//
// The root has distance 0
// The child (or children) of the root has (or have) distance 1
// If a node has distance i from the root, this node's child
//    (or children) has distance i + 1
//
// This function has three arguments
// root is the root of a tree
// distances is the address of an array allocated in
//     this function
// * maxdistance is the number of elements of this array
// * distances[i] stores the number of nodes at distance i
//
// If the tree does not exist (i.e., root is NULL)
//     * distances is NULL and * maxdistance is 0
// Otherwise, * distances should have at least one element
// If the tree exists, * distances[0] is 1 because there is exactly
//    one node (the root) at distance zero
// If * maxdistance is not zero, none of * distances' elements can
//    be zero (i.e., * maxdistance must not be larger than
//    necessary)
//
// This function allocates memory. The caller of this
// function is responsible releasing the memory

void Tree_distances(TreeNode * root,
```

```
                    int * * distances,
                    int * maxdistance);
#endif
```

```
// treedistance.c
#include "tree.h"
#include <stdlib.h>
#include <string.h>

// find the distance of the tree
// if tn is NULL, the distance is 0
// otherwise, calculate the distance of the left
// and the right subtrees
//
// the distance is 1 + max (left distance, right distance)
static int Tree_maxdistance(TreeNode * tn);
static int Tree_maxdistance(TreeNode * tn)
{
  if (tn == NULL) { return 0; }
  int leftdistance = Tree_maxdistance(tn -> left);
  int rightdistance = Tree_maxdistance(tn -> right);
  int distance = leftdistance;

  // <--- FILL CODE --->
  // Q1 (what is the condition for if?)
  A. if (distance < rightdistance)
  B. if (distance > rightdistance)
  C. if (distance == rightdistance)
  D. if (distance != rightdistance)
  E. if ((distance & rightdistance) != 0)
  F. if ((distance | rightdistance) != 0)
  G. if ((distance > 0) && (rightdistance < 0))
  H. None of the above
    {
      distance = rightdistance;
    }

  // <--- FILL CODE --->
  // Q2
  A. return (1 + distance);
  B. return (distance - 1);
  C. return distance;
  D. return (2 * distance);
```

```
  E. return (2 * distance + 1);
  F. return (distance + 2);
  G. return (2 * distance + 2);
  H. None of the above
}

// a recursive function
static void Tree_distances_helper(TreeNode * tn,
                                  int * counts,
                                  int curdistance)
{
  if (tn == NULL) { return; }
  // count the node at the current distance
  // <--- FILL CODE --->
  // Q3
  A. counts[curdistance] = 1;
  B. counts[curdistance] --;
  C. counts[curdistance] += curdistance;
  D. counts[curdistance] -= curdistance;
  E. counts[curdistance] ++;
  F. counts[curdistance] *= 2;
  G. counts[curdistance] += 2;
  H. None of the above

  // the children
  Tree_distances_helper(tn -> left, counts, curdistance + 1);
  Tree_distances_helper(tn -> right, counts, curdistance + 1);
}

void Tree_distances(TreeNode * root,
                    int * * distances,
                    int * maxdistance)
{
  int hi = Tree_maxdistance(root);
  * maxdistance = 0;
  * distances = NULL;
  if (hi == 0)
    {
      return;
    }
  // root must not be NULL since it has already been tested
  // <--- FILL CODE --->
```

```
  // Q4
  A. int * counts = malloc(sizeof(int) * (hi + 1));
  B. int * counts = malloc(sizeof(int) * hi);
  C. int * counts = malloc(sizeof(int) * (hi - 1));
  D. int * counts = malloc(sizeof(int) * (hi * 2));
  E. int * counts = malloc(sizeof(int) * (hi * 2 + 1));
  F. int * counts = malloc(sizeof(int) * (hi >> 2));
  G. int * counts = malloc(sizeof(int) * (hi << 3));
  H. None of the above

  if (counts == NULL) // malloc fail
    {
      return;
    }
  * maxdistance = hi;
  // <--- FILL CODE --->
  // Q5
  A. * distances =   counts;
  B. distances   =   counts;
  C. distances   = * counts;
  D. distances   = & counts;
  E. * distances = * counts;
  F. & distances =   counts;
  G. & distances = * counts;
  H. None of the above

  // initialize all elements to zero
  memset(counts, 0, sizeof(int) * hi);

  // <--- FILL CODE --->
  // Q6
  A. Tree_distances_helper(root,   counts, 0);
  B. Tree_distances_helper(root,   counts, 1);
  C. Tree_distances_helper(root, & counts, 0);
  D. Tree_distances_helper(root, * counts, 0);
  E. Tree_distances_helper(root, & counts, 1);
  F. Tree_distances_helper(root, * counts, 1);
  G. Tree_distances_helper(root, & maxdistance, 0);
  H. None of the above
}
```

4 File (2.5 points)

The following about `fseek` and `ftell` is for your reference.

```
SYNOPSIS
       #include <stdio.h>
       int fseek(FILE *stream, long offset, int whence);
       long ftell(FILE *stream);
DESCRIPTION
       The fseek() function sets the file position indicator for the
       stream pointed to by stream.  The new position, measured in
       bytes, is obtained by adding offset bytes to the position
       specified by whence.  If whence is set to SEEK_SET, SEEK_CUR,
       or SEEK_END, the offset is relative to the start of the file,
       the current position indicator, or end-of-file, respectively.
       A successful call to the fseek() function clears the
       end-of-file indicator for the stream and undoes any effects of
       the ungetc(3) function on the same stream.

       The ftell() function obtains the current value of the file
       position indicator for the stream pointed to by stream.
```

The following about `fread` and `fwrite` is for your reference.

```
NAME
       fread, fwrite - binary stream input/output

SYNOPSIS
       #include <stdio.h>
       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
       size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                     FILE *stream);
DESCRIPTION
       The  function  fread()  reads  nmemb  elements of data, each size bytes
       long, from the stream pointed to by stream, storing them at  the  loca
       tion given by ptr.

       The  function  fwrite()  writes nmemb elements of data, each size bytes
       long, to the stream pointed to by stream, obtaining them from the loca
       tion given by ptr.

       For nonlocking counterparts, see unlocked_stdio(3).
```

```
RETURN VALUE
       On  success,  fread()  and  fwrite() return the number of items read or
       written.  This number equals the number of bytes transferred only  when
       size  is 1.  If an error occurs, or the end of the file is reached, the
       return value is a short item count (or zero).

       fread() does not distinguish between end-of-file and error, and callers
       must use feof(3) and ferror(3) to determine which occurred.
```

Please write down the output of the program. Assume all file function calls are successful and the program returns EXIT_SUCCESS.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  if (argc < 2) { return EXIT_FAILURE; }
  int size = 5;
  // assume malloc succeeds
  int * arr = malloc(sizeof(int) * size);
  // initialize every elements
  int ind;
  for (ind = 0; ind < size; ind ++) { arr[ind] = ind; }
  FILE * foutptr = NULL;
  foutptr = fopen(argv[1], "w");
  // assume fopen succeeds
  fwrite(arr, sizeof(int), size, foutptr);

  // back to the beginning of the file
  fseek(foutptr, 0, SEEK_SET);
  fwrite(& arr[2], sizeof(int), size - 2, foutptr);
  long loc1 = ftell(foutptr);
  printf("1. %ld\n", loc1);
  fclose(foutptr);

  // open the same file for read now
  FILE * finptr = NULL;
  finptr  = fopen(argv[1], "r");
  fread(arr, sizeof(int), size, finptr);
  printf("2. %d\n", arr[0]);
  printf("3. %d\n", arr[4]);
  loc1 = ftell(finptr);
  printf("4. %ld\n", loc1);

```

```
  fseek(finptr, 0, SEEK_SET);
  int val;
  loc1 = ftell(finptr);
  fread(&val, sizeof(int), 1, finptr);
  long loc2 = ftell(finptr);
  printf("5. %ld\n", loc2 - loc1);
  fclose(finptr);
  free(arr);
  return EXIT_SUCCESS;
}
/* for your reference
   sizeof(char)   = 1
   sizeof(int)    = 4
   sizeof(int *)  = 8
   sizeof(double) = 8
*/
```