ECE368 Exam 1
Spring 2013

Wednesday, February 20, 2013
6:30-7:30pm
EE 170

READ THIS BEFORE YOU BEGIN

This is an *open-book, open-notes* exam. Electronic devices are not allowed. The time allotted for this exam is *exactly* 60 minutes. *It is in your best interest to not spend too much time on any problem*.

Always show as much of your work as practical–partial credit is largely a function of the *clarity and quality* of the work shown. Be *concise*. It is fine to use the blank page opposite each question (or at the back of each question) for your work. Do draw an arrow to indicate that if you do so.

This exam consists of 9 pages; please check to make sure that *all* of these pages are present *before* you begin. Credit will not be awarded for pages that are missing – it is *your responsibility* to make sure that you have a complete copy of the exam.

Learning objective 1 can be satisfied if you score 10 points or more in question 2 or question 3. Learning objective 2 can be met if you obtain 10 points or more in question 1 or question 3. Learning objective 3 can be achieved by scoring 10 points or more in question 1.

IMPORTANT: Write your login at the TOP of EACH page. Also, be sure to *read and sign* the *Academic Honesty Statement* that follows:

"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 exercise and will be subject to possible disciplinary action."

`Printed Name:`

`login:`

`Signature:`

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. Shell Sort (20 points) Consider the following procedure that generates a sequence of integers used by Shell sort to sort n integers, $n \geq 1$. The integers generated are stored in the variable gap.

```
Generate_Sequence(n)
01.      k ← 1;
02.      p ← 0;
03.      while (k < n) {
04.          k ← k × 3;
05.          p ← p + 1;
06.      }                                  /* end of while (k < n) */
07.      k ← k / 3;                         /* integer division */
08.      p ← p - 1;
09.      while (p >= 0) {
10.          gap ← k;                       /* first gap */
11.          seq_count ← p;
12.          do {
13.              gap ← (gap / 3) × 2;       /* more gaps:  integer operations */
14.              seq_count ← seq_count - 1;
15.          } while (seq_count >= 0);      /* end of do */
16.          k ← k / 3;                     /* integer division */
17.          p ← p - 1;
18.      }                                  /* end of while (p >= 0) */
```

(a) (4 points) What are the sequences generated when the following values of n are passed into the function Generate_Sequence? Write down each sequence in the order in which gap's are generated. If nothing is generated, write down "None."

- n = 1:
- n = 3:
- n = 10:
- n = 30:

(b) (9 points) For instructions 03, 09, and 15, write down the expression for the number of times each of the instructions is executed in terms of n. (*Hint*: Find out how p and k are related to n at the end of the first while-loop in line 6. Express p in terms of n and use that to answer this question.)

Line 03:

Line 09:

Line 15:

The sequence generation routine is now integrated in a Shell sort routine for sorting an array `r[0..n-1]` of `n` integers as follows, where new instructions in lines 13 through 21 are inserted between lines 12 and 13 of the `Generate_Sequence` routine.

```
Shell_Sort(r[0..n-1])
01.      k ← 1;
02.      p ← 0;
03.      while (k < n) {
04.          k ← k × 3;
05.          p ← p + 1;
06.      }                                  /* end of while (k < n) */
07.      k ← k / 3;                         /* integer division */
08.      p ← p - 1;
09.      while (p >= 0) {
10.          gap ← k;                        /* first gap */
11.          seq_count ← p;
12.          do {
13.              for j ← gap to (n - 1) { /* insertion sort of gap subarrays */
14.                  temp_r ← r[j];
15.                  i ← j;
16.                  while (i >= gap and r[i-gap] > temp_r) {
17.                      r[i] ← r[i-gap];
18.                      i ← i-gap;
19.                  }                      /* end of while (i >= gap and ...  */
20.                  r[i] ← temp_r;
21.              }                          /* end of for j ← gap to (n - 1) */
22.              gap ← (gap / 3) × 2;    /* more gaps:  integer operations */
23.              seq_count ← seq_count - 1;
24.          } while (seq_count >= 0);    /* end of do */
25.          k ← k / 3;                  /* integer division */
26.          p ← p - 1;
27.      }                                  /* end of while (p >= 0) */
```

(c) (7 points) Consider the array `r = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`. The first gap is 9. Show the array `r` right after 9-sorting.

2. (20 points) Consider an array-based implementation of singly and doubly linked list(s). Assume that each node in the linked list has the following structure:

```
typedef struct _generic_node {
   int info;
   struct _generic_node *next;
   struct _generic_node *prev;
} generic_node;
```

An array of `generic_node` was allocated and initialized as follows:

```
generic_node *array = (generic_node *)malloc(sizeof(generic_node) * 10);
generic_node *unused_list = &(array[0]);
int i;
for (i = 0; i < 9; i++) {
   array[i].next = &(array[i+1]);
}
array[i].next = NULL;
```

(a) (6 points) Show the contents of `array` in the following table after the initialization. If the value of a field (`info`, `next`, or `prev`) is indeterminate, use '-' for that entry in the table.

For an address field (`next` or `prev`) that is known, use an appropriate index picked from `-1`, `0`, ..., `9`. If the address field is assigned `NULL`, use '`-1`' for the entry. If the address field is assigned `&(array[i])`, use '`i`' for the entry. For example, `unused_list` is 'set' to '0'.

index	info	next	prev
9			
8			
7			
6			
5			
4			
3			
2			
1			
0			

`unused_list`: 0

Two routines are written to manage the nodes `unused_list`. Here, we assume that `unused_list` is a global variable.

```
void *mymalloc() {
   generic_node *node = unused_list;
   unused_list = unused_list->next;
```

```
   return (void *)node;
}

void myfree(void *node) {
   generic_node *gnode = (generic_node *)node;
   gnode->next = unused_list;
   unused_list = gnode;
}
```

(b) (4 points) The `generic_node` will be used to implement a priority queue using a circular doubly linked list through typecasting. The structure used for the priority queue is defined as follows:

```
typedef struct _lnode {
   int info;
   struct _lnode *next;
   struct _lnode *prev;
} lnode;
```

In the `main` function, the following two lines are executed:

```
lnode *dummy_header = (lnode *)mymalloc();
dummy_header->next = dummy_header->prev = dummy_header;
```

Show how the contents of the array in **(a)** would change in the following table. Also, show the contents of `dummy_header` and `unused_list`.

index	info	next	prev
9			
8			
7			
6			
5			
4			
3			
2			
1			
0			

`dummy_header`:

`unused_list`:

(c) (10 points) The following function implements the dequeue operation from a circular doubly linked list. Note that it dequeues the node next to `dummy_header`.

```
lnode *lnode_dequeue(lnode *dummy_header)
{
   lnode *curr = dummy_header->next;  /* curr is the node next to dummy_header */
   if (curr == dummy_header) {        /* empty priority queue */
      return NULL;
   }
   (curr->next)->prev = curr->prev;   /* remove curr from         */
   (curr->prev)->next = curr->next;   /* non-empty priority queue */
   return curr;
}
```

After running the program for a while, we have the following contents in `array`, `dummy_header`, and `unused_list`.

index	info	next	prev
9	109	-1	5
8	108	0	4
7	107	9	5
6	206	5	0
5	0	4	6
4	104	8	5
3	103	7	5
2	102	3	5
1	101	2	5
0	200	6	8

`dummy_header`: 5
`unused_list`: 1

(i) (4 points) Draw the doubly linked list, clearly indicating the `dummy_header`. For each node, clearly indicate the fields `info`, `next`, and `prev`.

(ii) (6 points) Suppose we execute the following statements in the `main` function:

```
lnode *node = lnode_dequeue(dummy_header);
myfree(node);
node = lnode_dequeue(dummy_header);
myfree(node);
```

Update the preceding table and the pointers `dummy_header` and `unused_list` to reflect the effects of those statements.

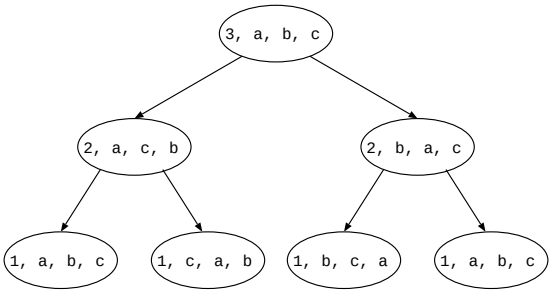
3. (20 points) If you are familiar with how to draw a computation tree for recursive function calls or a computation 'tree' after the elimination of tail recursion, you may skip this part and proceed to the questions **(a)**, **(b)**, and **(c)** directly.

Consider the following recursive function for solving the Tower of Hanoi problem:

```
void ToH(int n, char src, char intm, char dest)
{
   if (n == 1) {
      printf("move %d from %c to %c\n", n, src, dest);
      return;
   }
   ToH(n - 1, src, dest, intm);
   printf("move %d from %c to %c\n", n, src, dest);
   ToH(n - 1, intm, src, dest);
}
```

The following figure shows the computation tree when the following statement is executed:

```
ToH(3, 'a', 'b', 'c');
```

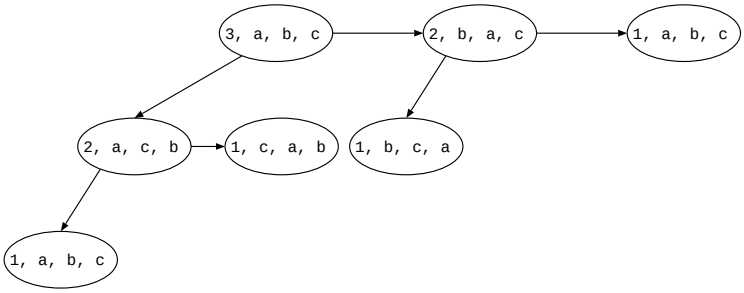


In this computation tree, each node is labeled with the arguments passed to the function.

The tail recursion in the function `ToH` can be eliminated as follows:

```
void iter_ToH(int n, char src, char intm, char dest)
{
   while (n != 1) {
      iter_ToH(n - 1, src, dest, intm);
      printf("move %d from %c to %c\n", n, src, dest);
      char tmp = intm;
      intm = src;
      src = tmp;
      n = n - 1;
   }
   printf("move %d from %c to %c\n", n, src, dest);
}
```

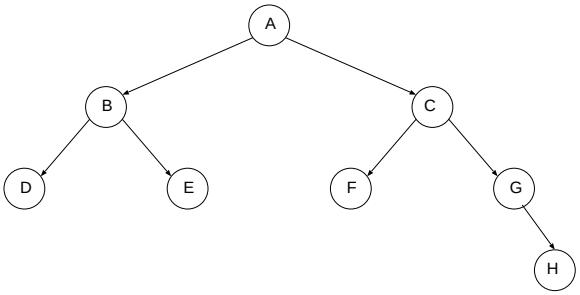
With the tail recursion removed, we have a computation ‘tree’ that shows the iterations as horizontal branches.



(a) (7 points) Consider a binary tree implemented with the following structure:

```
typedef struct _tnode {
   char info;
   struct _tnode *left;
   struct _tnode *right;
} tnode
```

Suppose you have a binary tree as follows, where the letter in the node is the `info` stored in the node:



Now, we pass the root node ‘A’ to the following recursive `preorder` function:

```
void preorder(tnode *node)
{
   if (node == NULL) return;
   printf("%c\n", node->label);
   preorder(node->left);
   preorder(node->right);
}
```

Draw the computation tree corresponding to the recursive calls of `preorder` (in the space provided next page). Each node of the computation tree should be labeled with the `info` field if a non-NULL `tnode` is passed to the function. If a NULL pointer is passed to the function, the node in the computation tree should be labeled with `NULL`.

Indicate on the computation tree a chain of recursive calls that result in the most number of stack frames for the function `preorder` on the call stack.

(b) (6 points) Modify the `preorder` function to eliminate the tail recursion.

```
void iter_preorder(tnode *node)
{




}
```

(c) (7 points) Draw the computation 'tree' corresponding to the recursive function `iter_preorder` when it is called with the same binary tree shown in **(a)**.

Indicate on the computation 'tree' a chain of recursive function calls that result in the most number of stack frames for the function `iter_preorder` on the call stack.