ECE368 Exam 1
Fall 2013

Wednesday, Oct 2, 2013
6:30-7:30pm
EE 129

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) The following is a Shell sort routine for sorting an array `r[0..n-1]` of `n` integers, `n` ≥ 1. Lines 00 through 04 generate the sequence in ascending order and the integers in the sequence are stored in the variable `gap`. Lines 05 through 17 discard the last `gap`, take the remainder of the sequence in reverse order, and for each instance of `gap`, perform sorting on each of the `gap` subarrays.

```
Shell_Sort(r[0..n-1])
00.     /* Generate Sequence */
01.     gap ← 1;
02.     while (gap < n) {
03.         gap ← 2 × gap + 1;
04.     }                           /* end of while (gap < n) */
05.     /* Use the sequence in reverse order for Shell sort */
06.     while (gap > 1) {
07.         gap ← (gap - 1)/2;
08.         for j ← gap to (n - 1) { /* insertion sort of gap subarrays */
09.             temp_r ← r[j];
10.             i ← j;
11.             while (i >= gap and r[i-gap] > temp_r) {
12.                 r[i] ← r[i-gap];
13.                 i ← i-gap;
14.             }                   /* end of while (i >= gap and ...)  */
15.             r[i] ← temp_r;
16.         }                       /* end of for j ← gap to (n - 1) */
17.     }                           /* end of while (gap > 1) */
```

(a) (4 points) What are the sequences of `gap`'s generated in lines 00 through 04 for the following values of `n`'s? Write down each sequence in the order in which the `gap`'s are generated.

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

(b) (6 points) For lines 01, 02, and 06, write down the expression for the number of times each of the instructions is executed in terms of `n`. For this question, you may have to use the ceiling or the floor function. Given a real number, x, the ceiling of x, denoted as $\lceil x \rceil$ is the smallest integer that is not less than x. The floor of x, denoted as $\lfloor x \rfloor$ is the largest integer that is not greater than x.

Line 01:

Line 02:

Line 06:

(c) (6 points) Consider the array `r = [21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`. The first `gap` used for Shell sorting in Lines 05-17 is 15. Show the array `r` right after 15-sorting.

(d) (4 points) Now, the routine begins to perform 3-sorting for `gap = 3` (after sorting for any `gap`'s between 15 and 3 in the sequence). At this moment, we have `r = [6, 5, 4, 3, 2, 1, 0, 7, 13, 12, 11, 10, 9, 8, 14, 21, 20, 19, 18, 17, 16, 15]`. Show the array *r* right after the iteration of `j = 14` in the `for`-loop of line 08 is completed while in the middle of 3-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 in `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 doubly linked lists through type-casting. The structure used for doubly linked lists 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 = NULL;
```

The following table shows the contents of the array before the execution of the two lines. The contents of `dummy_header` and `unused_list` before the execution of the two lines are also shown.

index	info	next	prev
9	109	-1	4
8	208	5	0
7	107	1	4
6	106	0	4
5	305	4	8
4	0	6	5
3	103	2	4
2	102	7	4
1	101	9	4
0	200	8	6

`dummy_header`: ?
`unused_list`: 3

Show in the preceding table how the contents of the array would change and also how the contents of `dummy_header` and `unused_list` would change after the execution of the two lines. It may be helpful to draw the linked list corresponding to `unused_list`.

(c) (10 points) 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	4
8	208	5	0
7	107	1	4
6	106	0	4
5	305	4	8
4	0	6	5
3	103	2	4
2	102	7	4
1	101	9	4
0	200	8	6

`dummy_header`: 4
`unused_list`: 3

(i) (6 points) The `dummy_header` points to a circular doubly list that implements a *priority queue*, where the `info` field of a node in the priority queue determines the priority. Draw the circular doubly linked list, clearly indicating the `dummy_header`. For each node, clearly indicate the fields `info`, `next`, and `prev`.

(ii) (4 points) Suppose we now execute the following statement in the `main` function:

```
lnode_enqueue(dummy_header, 204); /* enqueue 204 into priority queue */
```

Update the preceding table and the pointers `dummy_header` and `unused_list` to reflect the effects of the statement. The function `lnode_enqueue` should maintain the nodes in the *priority queue* in the right order such that the dequeue operation can be performed in $O(1)$ time complexity.

3. (20 points) Consider a binary tree implemented with the following structure:

```
typedef struct _tnode {
   int value;
   struct _tnode *left;
   struct _tnode *right;
} tnode
```

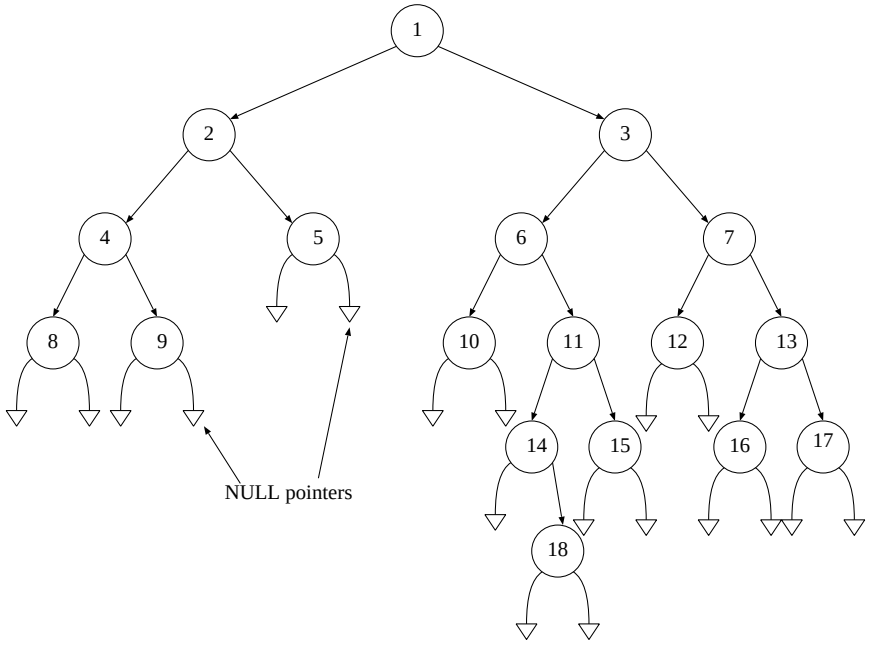
(a) (8 points) Suppose you are now given the following routine to construct a binary search tree from a sorted array.

```
tnode *tnode_construct_bst(int *array, int lb, int ub)
{
   if (lb > ub) {
      return NULL;
   }
   int mid = (lb+ub)/2;
   tnode *curr = malloc(sizeof(tnode));
   curr->value = array[mid];
   curr->left = tnode_construct_bst(array, lb, mid-1);
   curr->right = tnode_construct_bst(array, mid+1, ub);
   return curr;
}
```

If you pass in a sorted array `array = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]`, `lb = 0`, and `ub = 17` into the routine `tnode_construct_bst`, draw the computation tree corresponding to the construction of the binary search tree and the resulting binary search tree. In the computation tree, write down for each node, the corresponding `lb` and `ub`.

(b) (6 points) Now, we pass the root node '1' of the following binary tree to the following recursive function:

```
void tnode_destroy_tree(tnode *node)
{
   if (node == NULL) return;
   tnode_destroy_tree(node->left);
   tnode_destroy_tree(node->right);
   free(node);
}
```

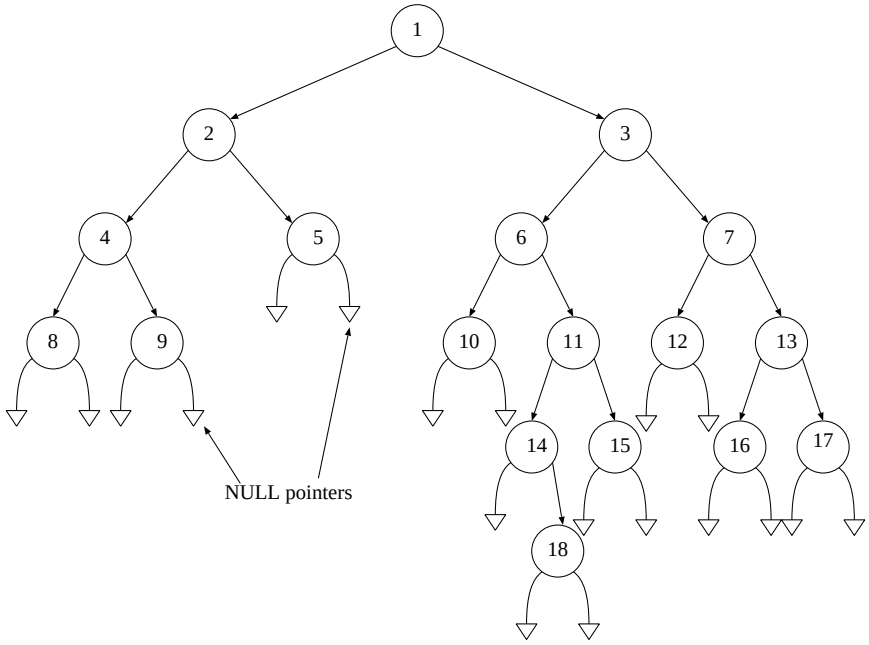


Identify the nodes in the binary tree that should be in the call stack when the call stack contains the most number of stack frames corresponding to the function `tnode_destroy_tree`. There may be multiple combinations of nodes that result in the most number of stack frames. For the combination given by you, cross out all the nodes that have been freed at that moment. If no nodes have been freed, state that clearly.

(c) (6 points) We relocate the `free(node)` statement and remove the tail recursion in `tnode_destroy_tree` to obtain the following function:

```
void tnode_iter_destroy_tree(tnode *node)
{
   while (node != NULL) {
      tnode_iter_destroy_tree(node->left);
      tnode *right_child = node->right;
      free(node);
      node = right_child;
   }
}
```

Now, we pass the root node '1' of the following binary tree to the preceding recursive function.



Identify the nodes in the binary tree that should be in the call stack when the call stack contains the most number of stack frames corresponding to the function `tnode_iter_destroy_tree`. There may be multiple combinations of nodes that result in the most number of stack frames. For the combination given by you, cross out all the nodes that have been freed at that moment. If no nodes have been freed, state that clearly.