ECE368 Mid-Term Exam 2
Fall 2005

Monday, November 14
7:00-8:00pm
EE 270

READ THIS BEFORE YOU BEGIN

This is a *closed book* exam. Calculators are not needed nor are they allowed. Since time allotted for this exam is *exactly* 60 minutes, you should work as quickly and efficiently as possible. *Note the allocation of points and do 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. *An answer without justification would not receive full credit*. Be *concise*. It is fine to use the blank page opposite each question for your work.

This exam consists of 11 pages (including this cover sheet and a grading summary sheet at the end); 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.

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:`

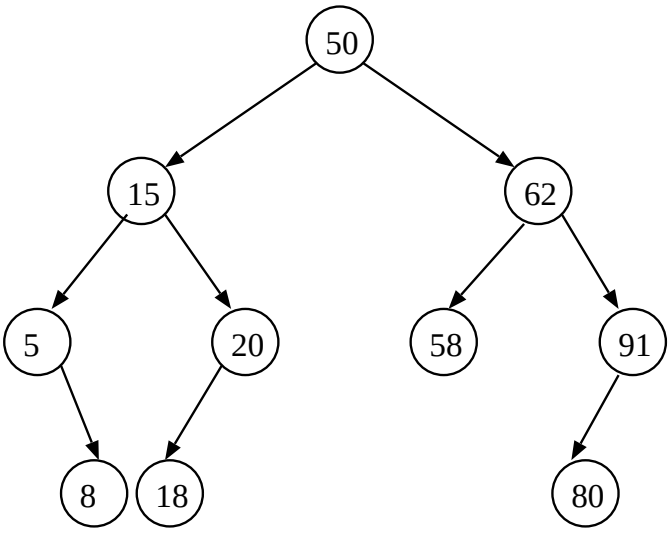
`Signature:`

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO ...

1. (20 points) Starting from an empty B-tree of order 4, perform the following operation: Insert 55, insert 66, insert 77, insert 88, insert 99, insert 11, insert 22, insert 33, insert 44, insert 40, insert 100, delete 40, and delete 11. Draw the resulting B-tree *after each operation.*

Clearly specify the operations (splitting a node, borrowing from a sibling, or concatenating two siblings) that you have used. When you split a node due to overflow, promote the median *after* the new key insertion to the parent node. (Consider a sorted array $A[lb..ub]$, the median is $A[\lfloor lb+ub/2 \rfloor]$.) When you delete a key from a non-leaf node, replace it with its immediate in-order predecessor.

2. (20 points) The following diagram shows a *height-balanced* binary search tree (AVL-tree). Performs the following operations: Insert 16, insert 19, delete 62. When you delete a non-leaf node with two child nodes, replace the deleted node with its immediate in-order predecessor. Draw the resulting height-balanced binary search tree *after each operation*. Clearly specify the rotation operations that you have used. Use the blank page opposite this page if necessary.



3. (20 points) Consider the following `Fast_Search` routine, where `a[0..`$n-1$`]` is sorted and `a[0]` $\leq$ `search_key` $\leq$ `a[`$n-1$`]`.

```
Fast_Search(a[0..n-1], search_key)
  bs_flag ← 0
  lb ← 0
  ub ← n-1
  while (lb ≤ ub) {
    if (bs_flag == 1) {
      mid ← (lb+ub)/2
      if (search_key == a[mid])
        return mid
      if (search_key < a[mid])
        ub ← mid - 1
      else
        lb ← mid + 1
      bs_flag ← 0
    } else {
      mid ← lb + (ub-lb)(search_key-a[lb])/(a[ub]-a[lb])
      if (search_key == a[mid])
        return mid
      if (search_key < a[mid]) {
        if (mid > (lb+ub)/2)
          bs_flag ← 1
        ub ← mid - 1
      } else {
        if (mid < (lb+ub)/2)
          bs_flag ← 1
        lb ← mid + 1
      }
    }
  }
  return -1;
```

3(a) (10 points) What is the worst case complexity of `Fast_Search(a[0..`$n-1$`], search_key)`? Justify your answer.

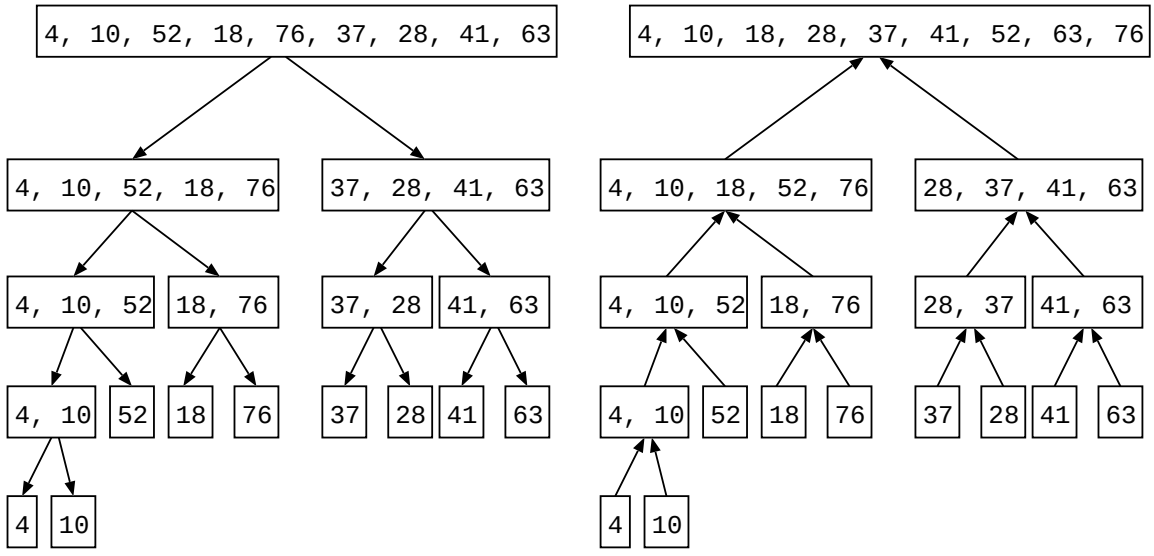
3(b) (10 points) What is the average case complexity of `Fast_Search(a[0..`$n-1$`], search_key)`? Justify your answer.

4. (20 points) The following routines perform a 2-way partitioning followed by 2-way-merging, that is, the array is divided into two subarrays, and `Mergesort` is recursively applied on the two subarrays.

```
2-Way-Mergesort(r[], lb, ub):
  if lb ≥ ub
    return
  mid ← lb+ub/2
  2-Way-Mergesort(r, lb, mid)
  2-Way-Mergesort(r, mid+1, ub)
  2-Way-Merge(r, lb, mid, ub)

2-Way-Merge(r[], lb, mid, ub):  // assume auxiliary space tmp[]
  memcpy(&tmp[lb], &r[lb], mid-lb+1)
  memcpy(&tmp[mid+1], &r[mid+1], ub-mid)
  i ← lb
  j ← mid+1
  for k ← lb to ub
    if (i > mid) r[k] ← tmp[j++]
    else if (j > ub) r[k] ← tmp[i++]
    else if (tmp[j] < tmp[i]) r[k] ← tmp[j++]
    else r[k] ← tmp[i++]
```

The following diagram illustrates the operation of the `2-Way-Mergesort` routine on an unsorted array $\langle 4, 10, 52, 18, 76, 37, 28, 41, 63 \rangle$.



We want to extend the `2-Way-Mergesort` routine to perform k-way `Mergesort`, that is, divide the array into k subarrays, and recursively apply k-way `Mergesort` on the subarrays.

4(a) (6 points) If $k = 3$, illustrate the application of 3-way `Mergesort` on the array $\langle 4, 10, 52, 18, 76, 37, 28, 41, 63 \rangle$.

4(b) (8 points) Write down an $O(kn)$ complexity algorithm to perform k`-Way-Merge` (not k`-Way-Mergesort`), i.e., to merge k sorted subarrays into one sorted array. Note that k can be an arbitrary number and n is the total number of elements in k subarrays.

4(c) (6 points) If you are told that the k`-Way-Merge` (not k`-Way-Mergesort`) can be performed in $O(n \log k)$, how could you achieve this? For this question, you do not have to write down the pseudo-code; a high level description of the solution is sufficient.

5. (20 points) Suppose you are given a "black-box" sub-routine `Median(A[], lb, ub)` that returns the *median* of an unsorted array `A[lb..ub]` of integers in *linear-time*. Define the median to be the element at position $\lfloor$`(lb+ub)/2`$\rfloor$ when the array is sorted. Based on the "black-box" sub-routine, a student wrote a sub-routine called `Function_A(A[], lb, ub, i)`, without adhering to the ECE368 Code Standard. In particular, the name given to the sub-routine is meaningless, and no meaningful comments were given.

```
Function_A(A[], lb, ub, i):
  if lb == ub {
    return A[lb]
  }
  median ← Median(A[], lb, ub)
  pivot_idx ← Partition(A[], lb, ub, median)
  k ← pivot_idx - lb + 1
  if i ≤ k {
    return Function_A(A[], lb, pivot_idx, i)
  } else {
    return Function_A(A[], pivot_idx+1, ub, i-k)
  }
```

Here, the student used the following partitioning algorithm, which is similar to the one given in the homework 4:

```
Partition(A[], lb, ub, pivot):
  down ← lb - 1
  up ← ub + 1
  while TRUE {
    do {
      up ← up - 1
    } while A[up] > pivot
    do {
      down ← down + 1
    } while A[down] < pivot
    if down < up {
      A[down] ↔ A[up]
    } else {
      return up
    }
  }
```

(Hint: For questions (a), (b), and (d), note that the overall flow of the algorithm is very similar to an implementation of binary search using recursion.)

a. (3 points) What does the routine `Function_A(A[], 0, n-1, i)` compute, assuming that `A[0..n-1]` is an unsorted array of integers? Essentially, what would be a more appropriate name for `Function_A`?

b. (3 points) What is the asymptotic order of the maximum stack size required by the sub-routine `Function_A(A[], 0, n-1, i)` implemented using recursion? For ease of analysis, you may assume that the unsorted array `A[0..n-1]` contains distinct integers.

c. (8 points) What is the time complexity of `Function_A(A[], 0, n-1, i)`? For ease of analysis, you may assume that the unsorted array `A[0..n-1]` contains distinct integers?

d. (6 points) Implement the subroutine `Function_A(A[], lb, ub, i)` with a while-loop instead of recursion.

Question	Max	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total	100	