CS 161 Summer 2009
Homework #3 Sample Solutions

Regrade Policy: If you believe an error has been made in the grading of your homework, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your homework a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire homework, so your final grade may either increase or decrease.

Problem 1 [5 points]

What really is important for ”on-disk” sorting approaches is less about memory usage and more about how many times we're going to have to visit the disk.

In practice, Merge-Sort is the algorithm that is amenable to ”on-disk” sorting. Depending on the data set and implementation, Quicksort can also be feasible. The other algorithms have varying issues described below that tend to account for too many disk accesses.

As mentioned above, Merge-Sort is very good for this since once we've split the data down to small enough chunks to process in memory, we only need to process on those chunks until we need to do the final merger of the larger chunks. Even the merger of the larger chunks is fairly disk friendly since it's just a traversal of data one element at a time starting at the beginning of a chunk of memory. If you look at the unix/linux ”sort” function, which does on disk sorting, Merge-Sort is used.

Algorithms like Insertion-Sort and bubble sort that make multiple passes over all of the data tend to perform poorly because they're accessing all of the data over and over again and must continually read that data off disk and get very little ”reuse” of the data that is in memory.

Heapsort similarly has issues since in the process of inserting and removing elements from the disk we tend to jump around a lot in the data space, which can result in many disk reads just to access one data item.

Quicksort, like Merge-Sort, once the data is split enough times can process large chunks wholly in memory. The challenge with Quicksort is that the partition function for larger problems can access the disk many times since it is continually swapping elements.

If we are careful about what we keep in memory, though, we can reduce the number of times we need to visit the disk. Finally, because the partitions aren't 50/50 splits, we can end up with different sized chunks, but this isn't a major issue.

Problem 2 [5 points] CLRS 10.1-6 (pg. 204)

We use two stacks S_1 and S_2 to simulate a queue and its operations EnQueue and DeQueue. The idea is that we store all elements in either S_1 in its original order, or S_2 in its reverse order.

Algorithm 1 EnQueue(S_1, S_2, x)

```
while !Stack-Empty($S_2$) do
  Push($S_1$, Pop($S_2$))
end while
Push($S_1, x$)
```

Algorithm 2 DeQueue(S_1, S_2)

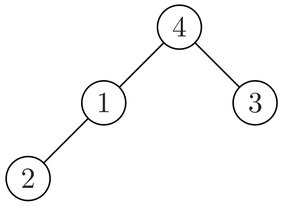
```
while !Stack-Empty($S_1$) do
  Push($S_2$, Pop($S_1$))
end while
Pop($S_2$)
```

Running time is $O(n)$ for both operations EnQueue and DeQueue.

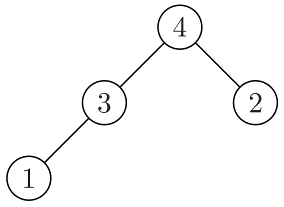
Alternative solutions: $O(1)$ implementation of EnQueue and $O(2n) = O(n)$ implementation of DeQueue. Vice versa.

Problem 3 [5 points] CLRS 6-1a (pg. 142)

They will produce different heaps. An example is for input $A = [1, 2, 3, 4]$.
Build-Max-Heap(A) procedure is illustrated as follows,



And for Build-Max-Heap'(A),



Problem 4 [10 points]

(a) (5 points) The pseudo-code is as follows. Please note that line 1 and line 8 are the changes we made, which update the *key* field.

Algorithm 3 BSTINSERT'(T, x)

```
$x.keys = 1$
**if** ROOT$(T)$ == null **then**
  ROOT$(T)$=$x$
**else**
  $y$ = ROOT$(T)$
  **while** $y \neq null$ **do**
    $prev = y$
    $prev.keys = prev.keys + 1$
    **if** $x < y$ **then**
      $y$ = LEFT$(y)$
    **else**
      $y$ = RIGHT$(y)$
    **end if**
  **end while**
  PARENT$(x) = prev$
  **if** $x < prev$ **then**
    LEFT$(prev) = x$
  **else**
    RIGHT$(prev) = x$
  **end if**
**end if**
```

The running time of BSTINSERT' is still $O(h)$, where h is the height of the input tree.

(b) (5 points) The pesudo-code is as in BSTKEYLESSTHAN(T, k).
Running time: Average-case $T(n) = T(n/2) + O(1)$. $T(n) = O(\log n)$, where n is the total number of nodes in the tree.
Worst-case: $T(n) = T(n-1) + O(1)$. $T(n) = O(n)$.

Algorithm 4 BSTKeyLessThan(T, k)

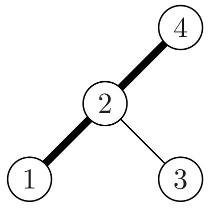
```
$count = 0$
$x = \text{Root}(T)$
**if** $x \neq null$ **then**
  **if** $x.value > k$ **then**
    $count = \text{BSTKeyLessThan}(Left(x), k)$
  **else if** $x.value == k$ AND $\text{Left}(x) \neq null$ **then**
    $count = \text{Left}(x).keys$
  **else**
    $count = \text{Left}(x).keys + 1 + \text{BSTKeyLessThan}(Right(x), k)$
  **end if**
**end if**
**return** $count$
```

Problem 5 [3 points]

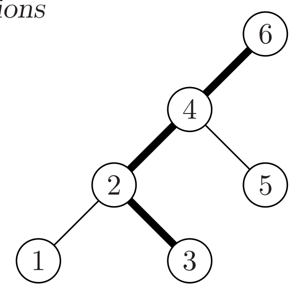
Points will be given based on creativity, knowledge tested and how well the questions fits the above criterion.

Problem 6 [5 points] CLRS 12.2-4 (pg. 260)

If we allow A or C to be empty. The smallest tree will have four nodes. An example is as follows. After searching for 1, $A = \{\}, B = \{1, 2, 4\}, C = \{3\}$. If we chose b=4, c=3, we got $b > c$.

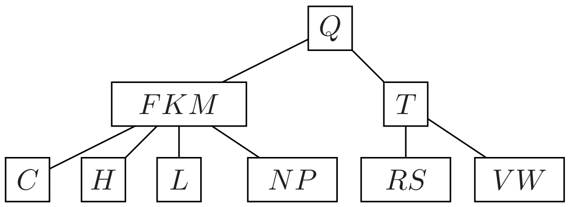


If we require A and C has at least one item each. The smallest tree we can get would have six nodes. An example is drawn below. Here $A = \{1\}, B = \{2, 3, 4, 6\}, C = \{5\}$. If we chose b=6, c=5, we got $b > c$.

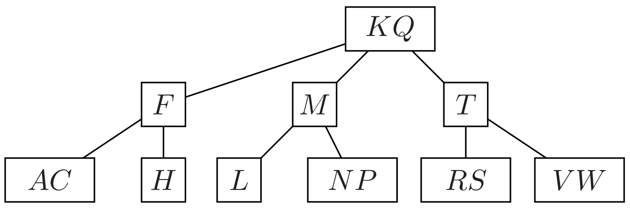


Problem 7 [3 points]

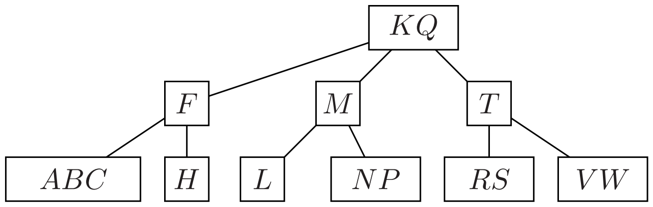
Inserting P:



Inserting A:



Inserting B:



Problem 8 [8 points] CLRS 18.2-6 (pg. 449)

Since there are at most $2t-1$ keys in a node, binary search at each node will cost $O(\log t)$. The depth of the B-tree is $O(\log_t n)$. Thus we have the total running time will be $O(\log t)O(\log_t n) = O(logn)$

Extra Credit [3 points]

Among all permutations of n number, the probability of the root is i is $1/n$, for all $i = 1, \ldots, n$. Thus, we have

$$\begin{aligned} & \text{Prob(root is 4, 5, or 6, left child of the root is 3, right child of the root is 7)} \\ = & \sum_{i=4}^{6} \text{Prob(root is } i) \times \text{Prob(left child of the root is 3|root is} i) \\ & \times \text{Prob(right child of the root is 7| root is } i) \\ = & \sum_{i=4}^{6} \frac{1}{n} \times \frac{1}{i-1} \times \frac{1}{10-i} \\ = & \frac{7}{450} = 1.56\% \end{aligned}$$

Notes: There are another interpretations of the problem that among all structurally unique binary tree of nodes 1 to 10, what is the probably that the left child of the root is 3 and the right child of the root is 7.
There are 2 structurally distinct trees for the left sub-tree of node 3.
There are 5 structurally distinct trees for the right sub-tree of node 5.
There are 5 possible ways to place 4,5,6 to the root and/or right sub-tree of node 3 and/or left sub-tree of node 5.
Thus the number of possible tree structure with 3 and 5 in designed position is $2\times5\times5 = 50$.
Give the total number of possible 10-node binary tree is $\frac{1}{n+1}C_n^{2n} = \frac{(2n)!}{n!(n+1)!} = 16796$
The probability of having ... is $\frac{50}{16796} = 0.298\%$.