CS 344
Spring 2024

Problem Set 2
Due: March 18, 2024, 11:59 p.m.

Name: (Replace me with your name) **NetID**: (Replace me with your NetID)

Honor Code: Students may discuss and work on homework problems in groups, which is encouraged. However, each student must write down their solutions independently to show they understand the solution well enough to reconstruct it by themselves. Students should clearly mention the names of the other students who offered discussions. We check all submissions for plagiarism. We take the honor code seriously and expect students to do the same.

Instruction for Submission: This homework has a total of 100 points, it will be rescaled to 10 points as the eventual score.

We encourage you to use LaTex to write your answer, because it's particularly suitable to type equations and it's frequently used for writing academic papers. We have provided the homework1.tex file for you, you can write your answer to each question in this .tex file directly after the corresponding question, and then compile the .tex file into a PDF file for submission. As a quick introduction to LaTex, please see this *Learn LaTeX in 30 minutes* [1]. Compiling a .tex file into a PDF file needs installing the LaTex software on your laptop. It's free and open source. But if you don't want to install the software, you can just use this website `https://www.overleaf.com/`, which is also free of charge. You can just create an empty project in this website and upload the homework1.zip, and then the website will compile the PDF for you. You can also directly edit your answers on the website and instantly compile your file.

You can also use Microsoft Word or other software if you don't want to use LaTex. If so, please clearly number your answers so that we know which answer corresponds to which question. You also need to save your answers as a PDF file for submission.

Finally, please submit your PDF file only. You should name your PDF file as "Firstname-Lastname-NetID.pdf''.

Make best use of picture in Latex: If you think some part of your answer is too difficult to type using Latex, you may write that part on paper and scan it as a picture, and then insert that part into Latex as a picture, and finally Latex will compile the picture into the final PDF output. This will make things easier. For instructions of how to insert pictures in Latex, you may refer to the Latex file of our homework 1, which includes several examples of inserting pictures in Latex.

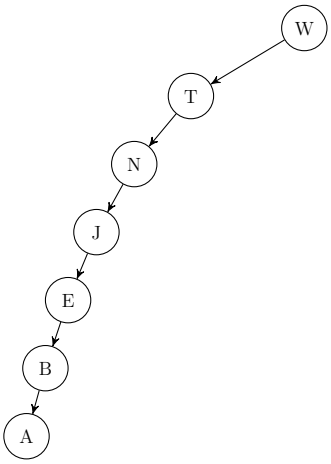
Discussion Group (People with whom you discussed ideas used in your answers if any):

I acknowledge and accept the Honor Code. Please type your initials below:
Signed: (Replace me with your initials)

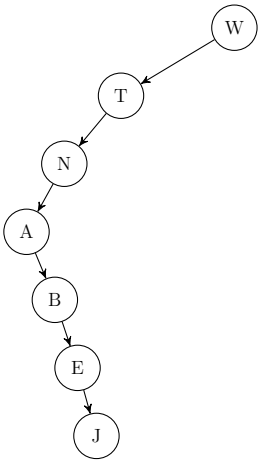
[1] `https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes`

1. **Warm up with BST.** (20 points) Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

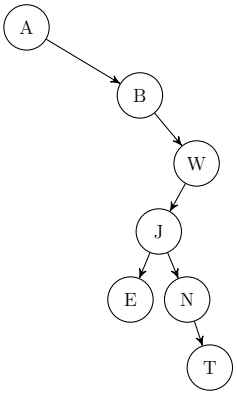
   (a) W, T, N, J, E, B, A



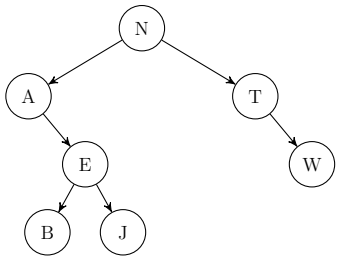
   (b) W, T, N, A, B, E, J



   (c) A, B, W, J, N, T, E



   (d) N, A, T, E, B, W, J

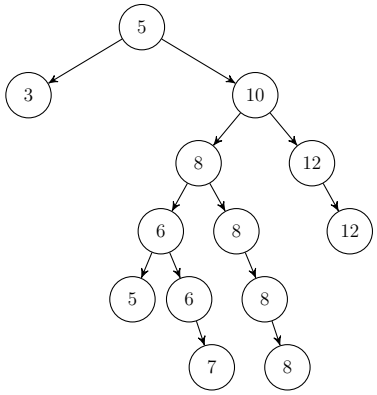


2. **Binary search tree with equal keys.** (40 points) In the class, we assumed that all the keys in a BST are different from each other. However, the BST can still store keys of the same value, in this case, we put a key that is less than a node to its left, and put a key that is greater than **or equal to** a node to its right. Here is the algorithm for inserting a new key z in to a binary search tree T:

Algorithm 1: Tree-Insert(T, z)

```
1  y = NIL;
2  x = T.root;
3  while x ≠ NIL
4      y=x;
5      if z.key<x.key
6          x = x.left;
7      else x = x.right
8  z.parent = y;
9  if y==NIL
10     T.root = z;    //tree T was empty
11 elseif z.key<y.key
12     y.left=z;
13 else y.right=z;
```

(a) (10 points) To better understand the algorithm, draw the tree generated by inserting the numbers 5, 3, 10, 8, 12, 12, 8, 8, 6, 6, 7, 5, 8 in this given order into an initially empty binary search tree using the above algorithm.



(b) (10 points) What is the asymptotic runtime of Tree-Insert when used to insert n items with identical keys into an initially empty binary search tree?

When inserting n identical keys the nodes will append one after one and eventually form a list. When the k-th node is inserted, it would be compared with each of the previously inserted keys, so there will be $k-1$ times of number comparisons. As a result, the total times of comparison is $1+2+\cdots+(n-1)=\frac{n(n-1)}{2}$, and the complexity is $O(n^2)$.

(c) We propose to improve TREE-INSERT by testing before line 5 to determine whether z.key==x.key and by testing before line 11 to determine whether z.key==y.key. If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic runtime of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x, and substitute y for x to arrive at the strategies for line 11.)

(c1) (10 points) Strategy 1: Keep a list of nodes with equal keys at x, and insert z into the list.

The tree will only contain the root node, and all additional equal keys will be placed into a list appending the root node. Each key will compare one time with the root node and then be inserted at the start of the list, which will take n number comparisons and n steps for inserting keys. As a result, the performance will thus be $O(2n)=O(n)$.

(c2) (10 points) Strategy 2: Randomly set x to either x.left or x.right. (Give the worst-case runtime and informally derive the expected runtime.)

Answer: The worst time run-time is all the keys happen to append one by one and eventually form a list, as a result, the runtime is the same as question (b) which is $O(n^2)$.

To calculate the average runtime, consider that when we have already inserted k keys into the tree, then the expected runtime of inserting the next key is equal to the expected depth of current tree of k keys. Because we are randomly inserting a key to the left or right with equal probability, so the expected depth of the tree with k keys will be the depth of a balanced tree of k keys, which is $O(\log k)$. As a result, the total runtime for inserting the first until the last key will be $O(\log 1+\log 2+\log 3+\cdots+\log(n-1))=O(\log(n-1)!)=O(\log\frac{n!}{n})=O(\log n!-\log n)=O(\log n!)=O(n\log n)$.

[**Note: for (b) (c1) and (c2), we are expecting the runtime represented as a function of** n].

3. **Hat-check problem.** (10 points) Use indicator random variables to solve the following problem, which is known as the hat-check problem. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

For each i. let X_i be the indicator variable for the event "customer i receives his hat back": $X_i = I customer i gets his own hat back.$

Then $X=X_1+\cdots+X_n$ is the number of customers receiving their hats back.

$E[X]=E[\sum_{i=1}^{n}X_i]$ is the expected number of customers receiving their hats back.

$E[X_i]$, for $i = 1 \cdots n$ is the expectation that customer i receives his hat back. Since the hat-check person operates randomly and uniformly, $E[X_i] = \frac{1}{n}$, for $i = 1 \cdots n$.

By linearity of expectation $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n} = 1$.

The expected number of people to receive their hats back is 1.

4. **Quicksort with equal element values.** (30 points) The analysis of the expected running time of randomized Quicksort in class (corresponding to Section 7.4.2 in textbook) assumes that all element values are distinct. In this problem, we examine what happens when they are not, i.e., there exist same-valued elements in the array to be sorted.

   The Quicksort algorithm relies on the following partition algorithm, which finds a pivot randomly, and then put all the numbers less than or equal to the pivot in the left, and put all the numbers greater than the pivot in the right, and then return the pivot location, as well as the left and right sublists for recursive calls.

Algorithm 2: Partition(A, p, r)

```
1 q=RANDOM(p,r);  //generate a random number in the range of [p,r]
2 L=empty list, R=empty list;
3 for each element a in A except A[q]:
4     if a≤A[q]:
5         append a to L;
6     else append a to R;
7 A=append(L,A[q],R);
8 return A, q=length(L);
```

In this algorithm, the list to be sorted is A, and we use p and r to denote the left-most and right-most indices of the currently processing subarray, respectively. For example, in the initial call of the Quicksort algorithm, we will let $p = 0$ and $r = n-1$, which correspond to the whole original array. The PARTITION(A, p, r) procedure returns an index q such that each element of $A[p : q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1 : r]$ is greater than $A[q]$.

(a) (10 points) Suppose there are n elements and all element values are equal. What would be randomized Quicksort's running time in this case?

At the first iteration the algorithm will pick up a value and compare it will all of the other (n-1) values, because all the values are the same, so all of the (n-1) values are put into the left list. As a result, for each of the following iterations, the algorithm will compare the selected value with all of the remaining values. The total times of comparison is $(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$. So the runtime is $O(n^2)$.

(b) (10 points) Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p : r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that:

   i. all elements of $A[q : t]$ are equal,

ii. each element of $A[p : q-1]$ is less than $A[q]$, and

iii. each element of $A[t+1 : r]$ is greater than $A[q]$

Like PARTITION, your PARTITON′ procedure should take $O(r-p)$ time.

Algorithm 3: Partition′(A, p, r)

```
1  q=RANDOM(p,r);  //generate a random number in the range of [p,r]
2  L=empty list; M=[A[q]]; R=empty list;
3  for each element a in A except A[q]:
4     if a < A[q]:
5        append a to L;
7     else if a == A[q]:
8        append a to M;
9     else: append a to R;
10 A=append(L,M,R);
11 return A, q=length(L), t=length(L)+length(M)-1;
```

(c) (10 points) Now you have a PARTITION′(A, p, r) algorithm, based on this algorithm, provide the pseudocode of a QUICKSORT′(A) algorithm which calls PARTITION′(A, p, r) as a subroutine, so that it recurses only on partitions of elements not known to be equal to each other. [Hint: it will looks very similar to our Quicksort pseudocode in the lecture slides, you only need to make minor modifications to recurse on the partitions produced by PARTITION′(A, p, r).]

Algorithm 4: Quicksort′(A)

```
1  if length(A)≤ 1:
2     return;
3  A, q, t = Partition′(A, 0, length(A)-1);
4  Quicksort′(A[0:q-1]);
5  Quicksort′(A[t+1:length(A)-1]);
```

we provide two implementations, other forms of pseudo codes also acceptable as long as it works correctly

Algorithm 5: Quicksort′(A, p, r)

```
1  if p ≥ r:
2     return;
3  A, q, t = Partition′(A, p, r);
4  Quicksort′(A,p,q-1);
5  Quicksort′(A,t+1,r);
```