Assignment 3 - Hady Ibrahim
SFWRENG 2CO3: Data Structures and Algorithms–Winter 2023

Deadline: March 5, 2023

Department of Computing and Software
McMaster University

Please read the *Course Outline* for the general policies related to assignments.

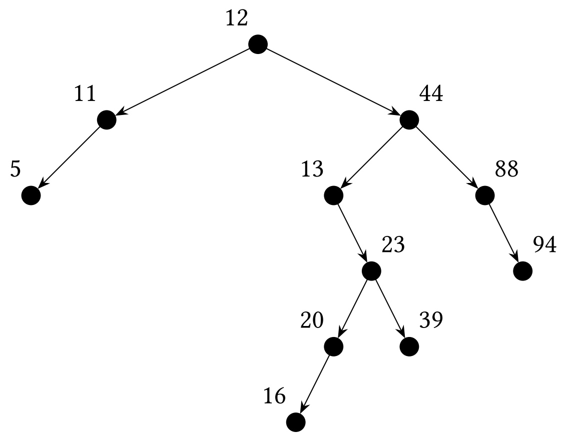
Plagiarism is a *serious academic offense* and will be handled accordingly.
All suspicions will be reported to the *Office of Academic Integrity*
(in accordance with the Academic Integrity Policy).

This assignment is an *individual* assignment: do not submit work of others. All parts of your submission *must* be your own work and be based on your own ideas and conclusions. Only *discuss or share* any parts of your submissions with your TA or instructor. You are *responsible for protecting* your work: you are strongly advised to password-protect and lock your electronic devices (e.g., laptop) and to not share your logins with partners or friends! If you *submit* work, then you are certifying that you have completed the work for this assignment by yourself. By submitting work, you agree to automated and manual plagiarism checking of all submitted work.

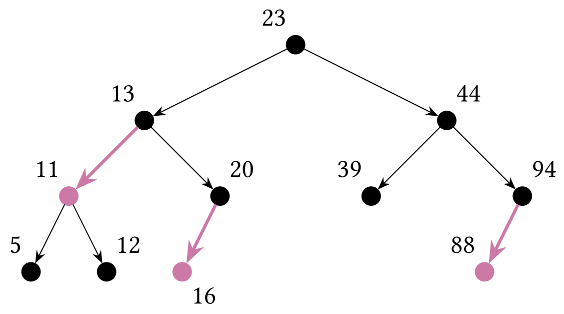
Late submission policy. Late submissions will receive a late penalty of 20% on the score per day late (with a five hour grace period on the first day, e.g., to deal with technical issues) and submissions five days (or more) past the due date are not accepted. In case of technical issues while submitting, contact the instructor *before* the deadline.

Problem 1. Consider the sequence of values $S = [12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5]$.

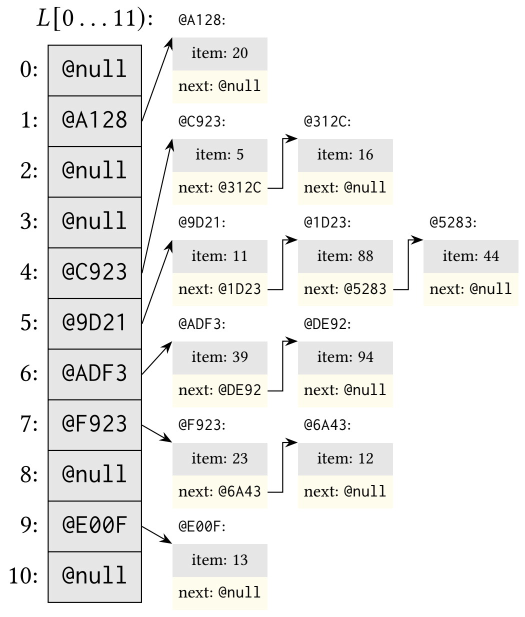
P1.1. Draw the binary search tree obtained by adding the values in S in sequence.



P1.2. Draw the red-black tree obtained by adding the values in S in sequence.



P1.3. Consider the hash-function $h(k) = (2k + 5) \bmod 11$ and a hash-table of 11 table entries that uses hashing with separate chaining. Draw the hash-table obtained by adding the values in S in sequence.



P1.4. Consider the hash-function $h(k) = (3k + 2) \bmod 11$ and a hash-table of 11 table entries that uses hashing with linear probing. Draw the hash-table obtained by adding the values in S in sequence.

$L[0 \dots 11)$:

0:	16
1:	5
2:	44
3:	88
4:	11
5:	12
6:	23
7:	20
8:	13
9:	94
10:	39

Problem 2. We say that a hash function $h : \mathcal{U} \to \mathbb{N}$ that maps values from a set $\mathcal{U}$ to integers in the range $[0 \dots M)$ is *n-perfect* if there exists at most n distinct values $u_1, \dots, u_j \in \mathcal{U}$ such that $h(u_1) = \dots = h(u_j)$.

P2.1. Consider the hash function $h(k) = (2k + 5) \bmod 11$. Is this hash function 2-perfect for the inputs $0, \dots, 21$? Explain why or why not.

Answer: This function is 2-perfect for the inputs 0,...,21 since after computation, there is only 2 values mapped to one spot in the hash table. In summary, since the function has 2k, as you loop through the values, the new value will most often be 2 more than the previous. This is until you hit 11 as it is mod 11, then it will start back at either 0 or 1 depending on if the previous value was 9 or 10. The following is the computation.

v	h(v)=(2v+5)mod 11	v	h(v)	v	h(v)	v	h(v)
0	5	6	6	12	7	18	8
1	7	7	8	13	9	19	10
2	9	8	10	14	0	20	1
3	0	9	1	15	2	21	3
4	2	10	3	16	4		
5	4	11	5	17	6		

Rearrange to see that there are only two values per mapping. We can see that if there was 1 more value, it would no longer be 2-perfect:

h(v)	v	h(v)	v	h(v)	v	h(v)	v
0	3, 14	3	10, 21	6	6, 17	9	13, 2
1	9, 20	4	5, 16	7	12, 1	10	8, 19
2	4, 15	5	11, 0	8	7, 18		

P2.2. Prove that a hash function $h : \mathcal{U} \to \mathbb{N}$ can only be n-perfect if $|\mathcal{U}| \leq n \cdot M$.

Answer:

Assume $h : \mathcal{U} \longrightarrow \mathbb{N}$ is n-perfect and let $u_1, \ldots, u_j$ be j distinct values in $\mathcal{U}$ such that $h(u_1) = \cdots = h(u_j)$. Then we will have that $j \leq n$.

$\mathcal{S}_i = \{x \in \mathcal{U} \mid h(x) = i\}$ –> In other words, $\mathcal{S}_i$ has elements in $\mathcal{U}$ that map to i

This means, $\mathcal{U} = \sum_{i=0}^{M-1} \mathcal{S}_i$ where $|\mathcal{U}| = M$

since h is n-perfect, we know $|\mathcal{S}_i| \leq n$

$\mathcal{U} = \sum_{i=0}^{M-1} \mathcal{S}_i$
$|\mathcal{U}| = |\sum_{i=0}^{M-1} \mathcal{S}_i|$
$= \sum_{i=0}^{M-1} |\mathcal{S}_i|$
$\leq \sum_{i=0}^{M-1} n$
$= nM$

In conclusion, we have proven that assuming $h : \mathcal{U} \longrightarrow \mathbb{N}$ is a n-perfect hash function, then $|\mathcal{U}| \leq nM$ holds.

As we provided an upper bound on the number of elements in $\mathcal{U}$ that can be hashed without collisions, we know that adding any more elements will result in more than n values being mapped to a single hash value, which violates n-perfect rules.

P2.3. Can a general purpose hash function be n-perfect for any M? Argue why or why not.

Answer:

No a general-purpose hash function cannot be n-perfect for any M because a general-purpose hash function must be able to take in any set of input and ensure each hash value contains $\leq n$ values from the input. This means it needs to handle inputs such as lists that have an infinite number of values (huge M), which is impossible to be n-perfect as you'd have to have an unrealistic number of spots in the hash-table to put the values in.

In other words, since the general purpose hash function depends on the value of n and M, it is not possible. If there are $n * M + 1$ inputs, then best case, each spot in the hash-table will have n values, but one spot will have $n + 1$ values, which violates the n-perfect rule.

Mathematically, the Probability a value V_r in the input set collides with a previously hashed value is:

Probability(V_r collides with V_1) $\cdot$ Probability(V_r collides with V_2) $\cdot \ldots$ Probability(V_r collides with V_{r-1})

1/M + 2/M + 3/M... n/M, where n is the n_{th} element in the input list.

FIX THISSSSSSSSSSSSSSSSSSSSSS

the probability to have a collision is $\sim 1 - \frac{n}{M}$

Problem 3. Consider non-empty binary search trees T_1 and T_2 such that all values in T_1 are smaller than the values in T_2. The SetUnion operation takes binary search trees T_1 and T_2 and returns a binary search trees holding all values originally in T_1 and T_2 (destroying T_1 and T_2 in the process).

P3.1. Assume the binary search trees storing T_1 and T_2 have the same height h. Show how to implement the SetUnion operation in $\sim h$ such that the resulting tree has a height of at-most $h + 1$.

1) Remove max value in T_1 and store it in variable m
h is worst-case time complexity, where the max value is at the h_{th} level in the tree. T_1 will still have a height of h or $h - 1$ if max value m was the only value in level h of the tree.

2) Add the max value m as the root of T_1
~ 1 time complexity, as it just takes a little pointer logic. Make m node root of T_1 and make m.left be the old root of T_1. This will lead the tree be max height $h + 1$ as we are only added 1 level (the new

root) to the tree. Also, we know all values in T_1 are smaller than m, so BST properties remain as we add T_1 as the left subtree of the new root, m. At this point, T_1 is of height $\leq h+1$ and T_2 is of height h

3) Take the root of T_2 and make it right subtree of new T_1
Again $\sim$ 1 time complexity, as it just requires to make m.right = T_2. We know the left subtree of m is max h height and the right subtree is h since we just added T_2 which is a tree of height h. We also know that all values in T_2 are greater than the values in the current T_1, including the root. This allows us to put the root of T_2 as the right child of m, ensuring we keep BST properties.

4) return T_1

To conclude, the resulting tree will be of height $h+1$ since we took 2 tree of height h and added 1 level to it (the root). We also know it keeps BST properties as we know all values in T_2 are larger than the maximum value in T_1, which allows us to put T_2 in the right subtree of the root of new T_1

P3.2. Assume that T_1 and T_2 are red-black trees with the same black height h. Show how to implement a SetUnion operation that returns a red-black tree in $\sim h$.

1) Remove min value in T_2 and store it in variable m

Removing the min value is max a $2h$ complexity function which is $\sim h$. This is because you might need to traverse the longest path in the tree and given the tree can be 1 red node then 1 black, this could be of height $2h$.

2a) Case where deletion of min child is easy to handle
- If the min child is red without children, just remove the min child and you're done - If the min child is red with children (would be only right children), you can set the min node.right to take its place.

These are both $\sim$1 operations.

2b) Case where deletion of min child could mess up the black height property: Traverse T_2 recursively and do any rotations necessary to maintain RB property.
Like 1), this is also $\sim h$ for the same reasons: you need to traverse the tree downwards where the max height is $2h$.

3) Calculate black height of T_2 and store it as h_{new}
This is $\sim h_n ew$ which is $\sim h$ as you must traverse the entire tree once. $h_{new} \leq 2h$ since for every 3 node (height of 2) (think of 2-3 trees), you can only turn it into a tree of height 3. This means you can only multiply the height of the tree by $\frac{3}{2}$, which we round to 2 for simplicity.

4) Make the min value of T_2 (m) the root of our new tree T_3
This is $\sim$1 operation.

5)

— SKIPPPPP ––

1) Remove min value in T_2 and store it in variable m

Removing the min value is max a $2h$ complexity function which is $\sim h$. This is because you might need to traverse the longest path in the tree and given the tree can be 1 red node then 1 black, this could be of height $2h$. Note: This may no longer be a RB Tree for the moment.

2) Calculate black height of T_2 and store it as h_{new}
This is $\sim h_n ew$ which is $\sim h$ as you must traverse the entire tree once. $h_{new} \leq 2h$ since for every 3 node (height of 2) (think of 2-3 trees), you can only turn it into a tree of height 3. This means you can only multiply the height of the tree by $\frac{3}{2}$, which we round to 2 for simplicity.

—- IGNORE 2 FOR NOW ––

3) Make the min value of T_2 (m) the root of our new tree T_3
This is $\sim$1 operation.

4) Add T_1 as the left subchild and T_2 as the right subchild of T_3
This is ~ 1 operation.

All values in T_1 are smaller than the root of T_3 since it is a value from T_2. All values from T_2 are larger than T_3 since the root was the min in T_3. We know the height of T_3 is $h + 1$ currently as all we have done is add 1 level from T_2.

5) Traverse T_3 recursively and do any rotations necessary to maintain RB property.
Like 1), this is also $\sim h$ for the same reasons: you need to traverse the tree downwards where the max height is $2(h + 1)$.

P3.3. Assume that T_1 and T_2 are red-black trees with black heights $h_1 > h_2$. Show how to implement a SetUnion operation that returns a red-black tree in $\sim h_1$.

1) Remove min value in T_2 and store it in variable m

Removing the min value is max a $2h_2$ complexity function which is $\sim h_2 \leq \sim h_1$. This is because you might need to traverse the longest path in the tree and given the tree can be 1 red node then 1 black, this could be of height $2h_2$. Note: This may no longer be a RB Tree for the moment.

2) Calculate black height of T_2 and store it as h_{new}
This is $\sim h_new$ which is $\sim h$ as you must traverse the entire tree once. $h_{new} \leq 2h$ since for every 3 node (height of 2) (think of 2-3 trees), you can only turn it into a tree of height 3. This means you can only multiply the height of the tree by $\frac{3}{2}$, which we round to 2 for simplicity.

3) Make the min value of T_2 (m) the root of our new tree T_3
This is ~ 1 operation.

4) Add T_1 as the left subchild and T_2 as the right subchild of T_3
This is ~ 1 operation.

All values in T_1 are smaller than the root of T_3 since it is a value from T_2. All values from T_2 are larger than T_3 since the root was the min in T_3. We know the height of T_3 is $h_1 + 1$ currently as all we have done is add 1 level from T_1 which is the larger tree.

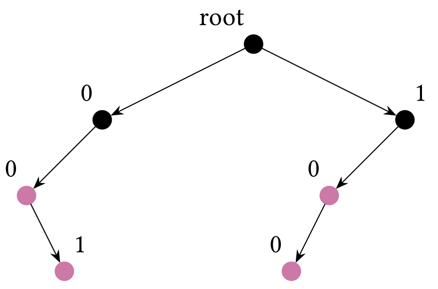
5) Traverse T_3 recursively and do any rotations necessary to maintain RB property.
This is $\sim h_1$ since you need to traverse the tree downwards where the max height is $2(h_1 + 1)$.

Problem 4. Consider *binary strings* (sequences of zeros and ones).

P4.1. Design a data structure BSSet that can be used to represent *sets of binary strings* such that any binary string W of length $|W| = N$ can be added or removed in $\sim N$ and such that one can check whether W is in the data structure in $\sim N$. Sketch why your data structure BSSet supports the stated operations in $\sim N$.

The data structure would be a binary tree where any left child is a 0 and any right child is a 1. Each bitstring is stored as a path down the tree. Each node has a flag (represented by red), which represents if the node is the terminal bit of a string.

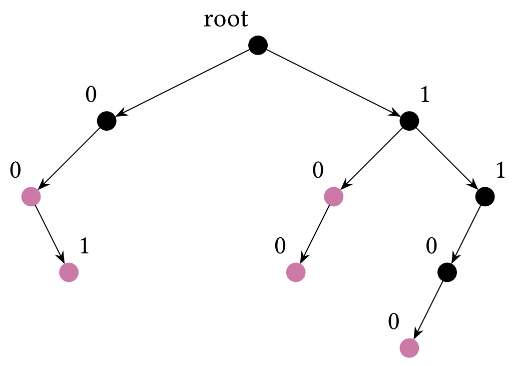
In the following example tree, this would represent the data structure holding the following bitstrings: "00", "001", "10", "100"



Adding:

To add a bitstring, you must traverse the path in relation to the new bitstring, and for any bit that isn't already in the data structure, add it. Once reaching the terminal bit of the bitstring, make the last node a red node. This traverses the tree only for the amount of bits in the bitstring, making it $\sim N$ complexity.

ex. Adding bitstring "1100", traverse tree 1->1->0->0, and for any non-existance node, create it. Make the last 0 a red node.



Removing:

To remove a bitstring, you traverse the path representing the bitstring to remove. Once reaching the last node, remove that node only if it has no children, else turn it from a red to black node. This takes $\sim N$ as you are traversing the tree downwards via length of the binary string. Then recursively retraverse the tree upwards and for every node that has no children, delete that node from the tree. This is also $\sim N$ complexity as you are traversing the same path as before, which makes the entire thing $\sim 2N$ which equals $\sim N$.

The previous method was the more neat way to keep the tree clean, however, it suffices to just traverse the tree and turn the terminal bit from a red node to a black one. In either case, you go down/up the tree max twice, making it $\sim N$ complexity.

Contains:

Traverse the tree representing the bitstring to check. If there is any element in the bitstring that does not exist in the path, return false. If all bitstrings are in the tree, once you reach the last node, check if the node is red. If it is red, return true, else return false.

This is a $\sim N$ complexity as you only need to traverse one path of the tree as long as the bitstring you are checking.

P4.2. Let W of length $|W| = N$ be a binary string and let S be a BSSET set. Provide an algorithm that prints all strings $V \in S$ that start with the prefix W (the first $|W|$ characters of S are equivalent to W). Your algorithm should have a worst-case complexity of $\sim N + k$ in which k is the number of characters printed to the output.

Assumption: The prefix exists in the tree. If needed, this can be checked before the algorithm in $\sim N$ as showed in q4.1, so the complexity won't change. This is done in the following algo.

```
Algorithm PrintPrefix(node, string):
1: if node = @null or !contains(string) then
2:     return
3: end if
4: if node = root then
5:     currNode = root
```

```
    prefix = string
    string = ""
    "//loop through prefix and traverse the tree while appending each value to string."
    for char ∈ prefix do
        if char = "0" then
            currNode = currNode.left
        else if char = "1" then
            currNode = currNode.right
        end if
        string.append(currNode.data)
    end for
    if currNode.color == "red" then
        print string "//this is to handle printing the prefix if it is in the tree."
    end if
    PrintPrefix(currNode.left, currNode.data + "") "//print all bitstrings in left subtree"
    PrintPrefix(currNode.right, currNode.data + "") "//print all bitstrings in right subtree"
else
    string.append(node.data) "//Add current node data to string"
    if node.color = "red" then
        "//if current node is a terminal node, print its string (the path traversed to get here)"
        print string
    end if
    PrintPrefix(currNode.left, currNode.data + "") "//print all bitstrings in left subtree"
    PrintPrefix(currNode.right, currNode.data + "") "//print all bitstrings in right subtree"
end if
```

You can call the method via PrintPrefix($root, prefix$). The complexity is $\sim N + k$ since the algorithm traverses the path of the given prefix which is $\sim N$. Then it traverses every path below it recursively. Anytime it gets to a red node, it prints out the path it took to get there, including the current node (note: it does not need to retraverse the graph backwards as it holds the path in a string). This second part is $\sim k$ as it checks only all characters after the prefix. As long as the data structure makes sure to only have paths that have a red node at the end (explained in q4.1 where deleting a string will remove all terminal black nodes), it will always be $\sim k$ since the longest path to traverse is as long as the largest binary string to print.

P4.3. Professor X claims to have developed a data structure BSSetX to which any binary string W of length $|W| = N$ can be added in $\sim N$. Furthermore, Professor X claims that BSSetX provides *ordered iteration*: one can iterate over all $M = |S|$ strings in a set S, implemented via BSSetX, in a lexicographical order in $\sim M + T$ in which T is the combined length of the M strings. Professor X claims that this method of sorting binary strings proves that the worst-case lower bound for sorting M binary strings is *not* $\sim M \log_2(M)$ comparisons. Argue why Professor X is wrong.

We note that strings S_1 and S_2 are *lexicographical ordered*, denoted by $S_1 \prec S_2$, if S_1 comes before S_2 in an alphabetical sort (e.g., as used in a dictionary). Next, we formalize $S_1 \prec S_2$ for binary strings: we have $S_1 \prec S_2$ if S_1 and S_2 are equivalent up to the $0 \leq i \leq \min(|S_1|, |S_2|)$-th character ($S_1[0] = S_2[0]$, $\dots$, $S_1[i-1] = S_2[i-1]$) and either $|S_1| = i < |S_2|$ or the $(i+1)$-th character of S_1 comes before the $(i+1)$-th character of S_2 (in which case $S_1[i] = 0$ and $S_2[i] = 1$). For example, $0 \prec 00$ and $00 \prec 01$, but not $100 \prec 10$.

a) Given a binary string of length N, adding takes $\sim N$ time. Now given there is a list of M binary strings all of length N, the complexity for adding all the binary strings to the data structure would be $\sim M \cdot N$, where $N, M \in \mathbb{N}$. Let's say $N = M$, in that case, we can already see our complexity would be $\sim M^2$ just to add the strings to the data structure, where $\sim M^2 \geq \sim M \log_2(M)$. Furthermore, T is the combined length of all binary strings to add, which would again be $M \cdot N$. Again we can see $\exists N, N > M \log_2(M)$ (ex. $N = M$), then this will be greater than the bound set by the Professor. Also the complexity of this algorithm relies on T which is the combined length of the strings. Professor X assumes that $T < log_2 M, \forall T, M \in \mathbb{N}$, which is not always the case, especially if you have large binary strings.

b) To prove a general-purpose sorting algorithm for M binary strings can beat the worst-case lower bound, there must be a use of comparisons to determine sorted order. The lower bound is based solely on # of comparisons. Professor X is suggesting to add the strings in an ordered fashion, then traverse the data structure in order. This contains no comparisons, thus cannot be a proof to contradict the worst-case lower bound for sorting.

Assignment Details

Write a report in which you solve each of the above problems. Your submission:

1. must be a PDF file;
2. must have clearly labeled solutions to each of the stated problems;
3. must be clearly presented;
4. must *not* be hand-written: prepare your report in LaTeX or in a word processor such as Microsoft Word (that can print or exported to PDF).

Submissions that do not follow the above requirements will get a grade of zero.

Grading

Each problem counts equally toward the final grade of this assignment.