Problem 1. Consider the following table:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search(S,key)</th>
<th>Insert(S,key)</th>
<th>Delete(S,obj)</th>
<th>Min(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted Doubly Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Doubly Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table with the worst-case running times for the best algorithms for implementing the given operations for the dynamic data structure representations listed in the leftmost column. If you made any assumptions while filling up the table above, please state them.

Note: Make sure you have no empty spaces in the middle of your array. key is the value of the item, obj is a pointer to the entry to be removed.


The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn’t matter what is the output for these values.

Here’s a simple algorithm to solve this problem.

1: for $i = 1$ to $n$ do
2:     for $j = i + 1$ to $n$ do
3:         Add up array entries $A[i]$ through $A[j]$
4:     Store the result in $B[i, j]$
5: end for
6: end for

(a) For some function $f$ that you should choose, give a bound of the form $\Theta(f(n))$ on the running time of this algorithm on an input of size $n$ (i.e., a bound on the number of operations performed by the algorithm).
(b) Although the algorithm above is a natural way to solve the problem - it iterates through the relevant entries of the array $B$, filling in a value for each entry - it contains some highly unnecessary sources of inefficiency. Design a different algorithm with an asymptotically better running time, i.e., a faster algorithm.

**Problem 3.** A Binary tree is a tree (acyclic graph with a root node), such that each parent node has at most two children. The layout of the tree depends completely on how it was built and the elements in it. The only rule is on the maximum number of children that each node can have.

Given this definition of a Binary Tree, we define a search algorithm as shown below. The input to this algorithm is the root node of the tree, and the element to be found. The output of this algorithm is a boolean indicating if the element has been found in the tree or not.

1: function SearchA(root, element) returns boolean
2: if root.value = element then
3: return true
4: end if
5: found ← false
6: if root.left ≠ null then
7: found ← SearchA(root.left, element)
8: end if
9: if found = false and root.right ≠ null then
10: found ← SearchA(root.right, element)
11: end if
12: return found
13: end function

A Binary Search tree is a binary tree such that, for each parent the values of the children on the left have to be less than or equal to that of the parent, and the values of the children on the right have to be more than that of the parent.

Given the definition of a Binary Search tree, we present the search algorithm below.

1: function SearchB(root, element) returns boolean
2: if root.value = element then
3: return true
4: end if
5: if element < root.value then
6: if root.left ≠ null then
7: return SearchB(root.left, element)
8: end if
9: else if root.right ≠ null then
10: return SearchB(root.right, element)
11: end if
12: return false
13: end function

(a) What is the worst-case running time ($T(n)$) of SearchA? And SearchB? Show your
work.

(b) What is the big-$O$ of SearchA? And SearchB? Prove your answer.

Proof Problem

\LaTeX source for template: https://www.overleaf.com/read/fvndzjwnwbp

Consider a version of the stable matching problem where there are $n$ men and $n$ women as before. Assume each man ranks the women (and vice versa), but now we allow ties in the ranking. In other words, we could have a man that is indifferent two women $w_1$ and $w_2$, but prefers either of them over some other woman $w_3$ (and vice versa). We say a woman $w$ prefers man $m_1$ to $m_2$ if $m_1$ is ranked higher on the $w$’s preference list and $m_1$ and $m_2$ are not tied.

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) **Strong Instability.** A strong instability in a perfect matching, $S$, consists of a man $m$ and a woman $w$, such that each of $m$ and $w$ prefers the other to their partner in $S$.

**Does there always exist a perfect matching with no strong instability?** Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) **Weak Instability.** A weak instability in a perfect matching, $S$, consists of a man $m$ and a woman $w$, such that their partners in $S$ are $w'$ and $m'$, respectively, and one of the following holds:

- $m$ prefers $w$ to $w'$, and $w$ either prefers $m$ to $m'$ or is indifferent between these two choices; or
- $w$ prefers $m$ to $m'$, and $m$ either prefers $w$ to $w'$ or is indifferent between these two choices.

In other words, the pairing between $m$ and $w$ is either preferred by both, or preferred by one while the other is indifferent.

**Does there always exist a perfect matching with no weak instability?** Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.