Tutorial 12

Today we’ll be working with trees:

- Orderings: PREORDER, INORDER, POSTORDER.
- Tree of int to array.
- Review of Data Storage Runtimes
PREORDER

Execute code on self before going to children
INORDER

Execute code on left, then self, then right
POSTORDER

Execute code on children before self
BST to Array (treesort)

The **treesort** algorithm sorts items by building a BST and then traversing it INORDER.

For this problem, the items in the BST will be **ints**, and we will generate a **new** array containing all of the items in order.

We will attempt this in two ways:

- call `bst_select(k, t)` \(n\) times [slow - \(O(nh)\) or \(O(n \log n)\) if balanced]

- traverse the tree in order [fast - \(O(n)\)]
Find Substrings

Given the partial implementation on seashell, complete the function `substring_find()`. The function will take in a string and a substring, and will return a linked list structure which contains pointers to all instances of the substring (if any) within the string.

Examples:

**Banana / can** Will produce no pointers

**Banana / an** Will produce pointers to index 1 and index 3
ADT Pro/Con

When creating an ADT, your method of storage can vary depending on a number of different circumstances.

Besides ease of implementation, one of the major factors that determines which framework you should use for an ADT is runtime. The runtimes on the following slide should be used to inform your decision of which you should use.
<table>
<thead>
<tr>
<th>Functions</th>
<th>Linked List</th>
<th>Dynamic Array</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(i)(auto-shifts)</td>
<td>O(n)</td>
<td>O(h)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(i)</td>
<td>O(n)</td>
<td>O(h)</td>
</tr>
<tr>
<td>Lookup</td>
<td>O(i)</td>
<td>O(i) (with address)</td>
<td>O(h)</td>
</tr>
<tr>
<td>Destroy</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>