Exercise 1: Binary Search Tree - Range Queries

(a) Implement the binary search tree (BST) data structure and the `insert` operation. You can use the template `BST.py`.  

(b) Implement the operation `getrange(x_{min}, x_{max})` efficiently on binary search trees which returns all keys `x` in the tree with `x_{min} \leq x < x_{max}` (cf. lecture notes week 6 slide 21).

(c) Use your implementation of BST and your `insert` function to insert all words from the file `inputs.txt` into a BST with respect to the lexicographic ordering on words over the alphabet `{a, . . . , z}`\(^1\). Use your data structure to output all words from the BST beginning with a certain prefix.\(^2\) Output all words with prefix “qw”. Copy the result into your `experiences.txt` file.

Sample Solution

Cf. `BST.py` for part (a) and (b). For part (c) it was sufficient to run `getrange('qw', 'qx')` on the BST filled with the words from `input.py`. The correct output is `['qwb', 'qwdjbcsm', 'qweli', 'qwconj', 'qwgykg', 'qwvkay', 'qwlybcn', 'qwmmwi', 'qwo', 'qwohudf', 'qwph', 'qwqrn', 'qwrmd', 'qwtq', 'qwxyjl', 'qwxrm', 'qwyiwh']`.

Exercise 2: Binary Search Tree - Operations

(a) Describe a function which returns the depth of a binary search tree and analyze the runtime.

(b) Describe a function that for a given binary search tree with `n` nodes and a given `k \leq n` returns a list with the `k` smallest keys from the tree. Analyze the runtime.

(c) Describe a function that takes a binary search tree `B` and a key `x` as input and generates the following output:

- If there is an element `v` in `B` with `v.key = x`, return `v`.
- Otherwise, return the pair `(u, w)` where `u` is the tree element with the next smaller key and `w` is the element with the next larger key. It should be `u = None` if `x` is smaller than any key in the tree and `w = None` if `x` is larger than any key in the tree.

For your description you can use pseudo code or a sufficiently detailed description in English. Analyze the runtime of your function.

---

\(^1\)Python supports the comparison of strings with respect to the lexicographic ordering, i.e., you can use "<", "<=".

\(^2\)If you enter `Python3` and `from BST import BST` into the command prompt you can use the class `BST` from the command line. We provided a method for inserting the content of `inputs.txt`. 

Sample Solution

(a) We can do a recursive traversal of the tree where we keep track of the current recursion depth. Then a call of \texttt{depth}(r) on the root \emph{r} of the BST returns its depth.

\begin{algorithm}
\textbf{Algorithm 1} \texttt{depth}(v,R) \\
\textbf{if} \ v = \text{None} \textbf{then} \\
\hspace{1em} \textbf{return} -1 \quad \texttt{▷ depth of a childless node must be 0, hence we define the depth of None as -1} \\
\textbf{else} \ \textbf{return} \ \max(\texttt{depth}(v.\text{left})+1, \texttt{depth}(v.\text{right})+1) \\
\end{algorithm}

The runtime corresponds to the runtime of the traversal of the whole tree which is \mathcal{O}(n) as we have just one recursive call for each node and each recursive call costs \mathcal{O}(1) (c.f., pre-, in-, post-order traversal algorithms given in the lecture).

As an alternative solution, we can run a BFS which takes \mathcal{O}(n). If \emph{v} is the node visited last by the BFS, do

\begin{algorithm}
\textbf{Algorithm 2} \texttt{traverse-up}(v) \\
\hspace{1em} d \leftarrow 0 \\
\hspace{2em} \textbf{while} \ v.\text{parent} \neq \text{None} \textbf{do} \\
\hspace{3em} d \leftarrow d + 1 \\
\hspace{3em} v \leftarrow v.\text{parent} \\
\hspace{2em} \textbf{return} \ d \\
\end{algorithm}

This takes \mathcal{O}(d) where \emph{d} is the depth of the tree. Since \emph{d} \leq \emph{n} the overall runtime is \mathcal{O}(n+d) = \mathcal{O}(n).

(b) Initialize an empty list \emph{K}. We roughly do the following. Make an in-order traversal of the tree and each time visiting a node, add it to \emph{K}. Stop if |\emph{K}| \geq \emph{k}. The following pseudocode formalizes this.

\begin{algorithm}
\textbf{Algorithm 3} \texttt{inorder\_variant}(node) \quad \texttt{▷ Assume list \emph{K} is given globally, initially empty} \\
\textbf{if} \ \texttt{node} \neq \text{None} \textbf{then} \\
\hspace{1em} \texttt{inorder\_variant}(\texttt{node.\text{left}}) \\
\hspace{1em} \textbf{if} \ |\emph{K}| \geq \emph{k} \textbf{then} \\
\hspace{2em} \textbf{return} \\
\hspace{2em} \emph{K}.\text{append}(\texttt{v.\text{key}}) \\
\hspace{1em} \texttt{inorder\_variant}(\texttt{node.\text{right}}) \\
\end{algorithm}

The runtime is \mathcal{O}(d + k) where \emph{d} is the depth of the tree. We prove this in the following.

Let \emph{K} be the set of \emph{k} nodes representing the \emph{k} smallest keys in the BST. Obviously, the in-order traversal must visit all nodes in \emph{K} once. In accordance with the lecture a call of \texttt{inorder\_variant}(\texttt{root}) adds all keys in ascending order to \emph{K}.

Let \emph{A} be the set of nodes in the BST on which are not in \emph{K} but in which a recursive call will be made. Since the recursion is aborted (with the \texttt{return} statement) after reporting \emph{k} nodes, the set \emph{A} contains exactly the nodes which are ancestors of a node in \emph{K}, but are not in \emph{K} themselves. Since the runtime of a single recursive call (neglecting subcalls) is (1) the total runtime is \mathcal{O}(|\emph{A}| + |\emph{K}|).

By definition we have |\emph{K}| = \emph{k}, so it remains to determine the size of \emph{A}. We claim that all nodes in \emph{A} are on a path from the root to a leaf, that is, |\emph{A}| \leq \emph{d}. This is the case if there do not exist two nodes in \emph{A} so that neither is an ancestor of the other.

For a contradiction, suppose that two such nodes \emph{u}, \emph{v} exist so that neither \emph{u} is ancestor of \emph{v} nor vice versa. Assume (without loss of generality) that key(\emph{u}) \leq key(\emph{v}). That means \emph{u} is in the left and \emph{v} is in the right subtree of some common ancestor \emph{a} of \emph{u} and \emph{v}.

By definition \emph{v} has a node \emph{w} \in \emph{K} in its subtree. Since \emph{v} is in the right subtree and \emph{u} is in the left subtree of \emph{a}, we have key(\emph{w}) \geq key(\emph{u}) and \emph{w} has a higher in-order-position. But then we would have \emph{u} \in \emph{K} as well, a contradiction to \emph{u} \in \emph{A}.
Algorithm 4: return-closest($x$)

$v \leftarrow \text{find}(x)$

\begin{align*}
\text{if } v \neq \text{None} \text{ then} & \\
& \quad \text{return } v \\
\text{else} & \\
& \quad \text{insert}(x) \\
& \quad (p, s) \leftarrow (\text{pred}(x), \text{succ}(x)) \\
& \quad \text{delete}(x) \\
& \quad \text{return } (p, s)
\end{align*}

All subprocedures that we call ($\text{find}$, $\text{insert}$, $\text{pred}$, $\text{succ}$) are known from the lecture and take $O(d)$ with $d$ being the depth of the tree. So the overall runtime is $O(d)$. 