Dictionaries

A dictionary is an important abstract data type (ADT). It represents the following object and operations:

**ADT: DICTIONARY**

**objects**: Sets of elements \( x \) such that each \( x \) has a value \( key(x) \) such that \( key(x) \) comes from a **totally ordered** universe

Note: Totally ordered just means that for any two keys \( a \) and \( b \), either \( a > b \), \( a < b \), or \( a = b \).

**operations**:

- \( \text{ISEMPTY(Set } S\text{)} \): check whether set \( S \) is empty or not
- \( \text{SEARCH(Set } S\text{, Key } k\text{)} \): return some \( x \) in \( S \) such that \( key(x) = k \) or null if no such \( x \) exists
- \( \text{INSERT(Set } S\text{, Element } x\text{)} \): insert \( x \) into \( S \)
- \( \text{DELETE(Set } S\text{, Element } x\text{)} \): remove \( x \) from \( S \)

There are many possible data structures that could implement a dictionary. We list some of them with their **worst case running times** for `SEARCH`, `INSERT`, `DELETE`.

<table>
<thead>
<tr>
<th>DATA STRUCTURE</th>
<th>SEARCH</th>
<th>INSERT</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted singly linked list</td>
<td>( n )</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>unsorted doubly linked list</td>
<td>( n )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sorted array</td>
<td>( \log n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>hash table</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>binary search tree</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>balanced search tree</td>
<td>( \log n )</td>
<td>( \log n )</td>
<td>( \log n )</td>
</tr>
</tbody>
</table>

**5 Binary Search Trees**

**Definition.** For a node \( x \) in a tree, \( height(x) \) is equal to the length of the longest path from \( x \) to a leaf.
Definition. For a node $x$ in a tree, $\text{depth}(x)$ is equal to the length of the path from $x$ to the root.

A binary tree is a binary search tree (BST) if it satisfies the BST Property

**BST Property.** For every node $x$, if node $y$ is in the left subtree of $x$, then $\text{key}(x) \geq \text{key}(y)$. If node $y$ is in the right subtree of $x$, then $\text{key}(x) \leq \text{key}(y)$.

We will see why this property is useful for searching for a particular key. However, we will need to ensure that **INSERT** and **DELETE** maintain the BST Property. We will now consider a binary search tree as a data structure for the **DICTIONARY** ADT. We will begin by implementing **SEARCH** as follows:

**Search (BST root $R$, key $k$):**
- if $R = \text{null}$ then return null
- else if ($k = \text{key}(R)$) then return $R$
- else if ($k < \text{key}(R)$) then return Search (leftChild($R$), $k$)
- else if ($k > \text{key}(R)$) then return Search (rightChild($R$), $k$)

In the worst case, we’ll start at the root of the tree and follow the longest path in the tree and then find that there is no node with key $k$. Since the length of the longest path in the tree is the definition of the *height* of the tree, this takes time $\Theta(\text{height of tree})$. For a tree with $n$ nodes, the height can be $n$ (if there are no right children, for instance)! So the worst-case running time (that is, for the worst tree and the worst $k$) is $\Theta(n)$.

Our implementation of **INSERT** follows:

**Insert (BST root $R$, node $x$):**
- if $R = \text{null}$ then $R := x$
- else if ($\text{key}(x) < \text{key}(R)$) then Insert (leftChild($R$), $x$)
- else if ($\text{key}(x) > \text{key}(R)$) then Insert (rightChild($R$), $x$)
- else if ($\text{key}(x) = \text{key}(R)$) then /* depends on application */

$x$ will always be added as a leaf. Again we might have to follow the longest path from the root to a leaf and then insert $x$, so in the worst case, **Insert** takes time $\Theta(n)$.

The **Delete** operation is more complicated, so we describe it at a higher level.

**Definition.** $\text{succ}(x)$ is the node $y$ such that $\text{key}(y)$ is the lowest key that is higher than $\text{key}(x)$

This definition of $\text{succ}(x)$ captures the intuitive notion of the *successor* of $x$.

Notice that if $x$ has a right child, then $\text{succ}(x)$ is the left-most node in the right subtree of $x$. In other words, starting from $x$’s right child, go left until there are no left children to follow. In this section, we will only call $\text{succ}(x)$ when $x$ has a right child.
Now, \texttt{Delete ( BST root R, node x )} has three cases:

1. If \( x \) has no children, simply remove it by setting \( x = \text{null} \).

2. If \( x \) has one child \( y \) and \( z \) is the parent of \( x \), then we remove \( x \) and make \( y \) the appropriate child of \( z \) (i.e. the left child if \( x \) was the left child of \( z \) and the right child if \( x \) was the right child of \( z \)).

3. If \( x \) has two children, then let \( A \) and \( B \) be the left and right subtrees of \( x \), respectively. First we find \( \text{succ}(x) \). Then we set \( x \) to be \( \text{succ}(x) \) and \texttt{Delete succ}(x) \ (using either case 1 or case 2). By the definition of \( \text{succ}(x) \), we know that everything in \( A \) has key less than or equal to \( \text{key}(\text{succ}(x)) \). Let \( B' \) be \( B \) with \( \text{succ}(x) \) removed. Everything in \( B' \) must have key greater than or equal to \( \text{key}(\text{succ}(x)) \). Therefore, the \textbf{BST Property} is still maintained.

\textbf{Exercise}. \emph{Why is it guaranteed that deleting \text{succ}(x) \ always falls into case 1 or case 2 (i.e. case 3 never occurs when deleting \text{succ}(X))?}

Again, if \( x \) is the root and \( \text{succ}(x) \) is the leaf at the end of the longest path in the tree, then searching for \( \text{succ}(x) \) will take \( \Theta(\text{height of tree}) \) in the worst case. Since everything else we do takes constant time, the worst-case running time is \( \Theta(n) \) (Since in the worst case, the height of the tree is \( \Theta(n) \)).

Notice that the running times for these operations all depend on the height of the tree. If we had some guarantee that the tree’s height was smaller (in terms of the number of nodes it contains), then we would be able to support faster operations.

6 \textbf{Red-Black Trees}

A \emph{red-black tree} is a BST that also satisfies the following three properties:

1. Every node \( x \) is either red or black \((\text{color}(x) = \text{red or color}(x) = \text{black})\).

2. Both children of a red node are black.
3. For every node $x$, any path from $x$ to a descendant leaf contains the same number of black nodes.

**Definition.** For a node $x$ in a Red-Black tree, the black height or $BH(x)$ is the number of black nodes on a path between $x$ and a descendant leaf (not including $x$).

**Definition.** The black height of a tree $T$ ($BH(T)$) is the black height of its root.

Notice that this definition is well defined since the number of black nodes between $x$ and a leaf is always the same because of Property 3.

To make things work out easier, we’ll consider every node with a key value to be an internal node and the null values at the bottom of the tree will be the leaves and will be colored black.

**Example:**

![Tree Diagram](image)

6.1 Red-Black Trees Are Short

These three extra properties guarantee that the tree is approximately balanced and therefore, the height is bounded. More precisely:

**Theorem.** Any red-black tree with $n$ internal nodes has height at most $2 + 2 \log(n + 1)$.

To prove this theorem, we first prove the following lemma:

**Lemma.** For any node $x$ in a red-black tree, the number of nodes in the subtree rooted at $x$ is at least $2^{BH(x)} - 1$.

**Proof.** By induction on the height of $x$ (i.e. the length of the longest path from $x$ to a descendant leaf).
Base Case: The height of $x$ is 0.

Since the height of $x$ is 0, $x$ has no children. Certainly $BH(x)$ is 0 if its regular height is 0, so $x$ must have at least $2^0 - 1 = 0$ children. This is trivially true.

Inductive Step: Assume that the lemma is true for height of $x$ less than $n$. Prove that the lemma is true for height of $x$ equal to $n$.

We know that for any $x$ of height $n$, its children’s heights are less than $n$ and therefore the lemma is true for $x$’s children. Note that we are considering null to be a valid child. Each child $y$ must have black height at least $BH(x) - 1$ because if $y$ is black, it will have black height $BH(x) - 1$ and if $y$ is red, it will have black height $BH(x)$. Therefore, there must be at least $2^{BH(x) - 1} - 1$ internal nodes in the subtrees rooted at each child. Then, including $x$, we have at least $2(2^{BH(x) - 1} - 1) + 1 = 2^{BH(x)} - 1$ nodes in the subtree rooted at $x$.

Now we can easily prove the theorem:

Proof of Theorem. Let $h$ be the height of the tree. Property 2 says that on any path from the root to leaf, at least half of the nodes are black. So the black height of the root must be at least $\lceil h/2 \rceil - 1$ (since the root could be red). If $n$ is the number of internal nodes, then we know from the lemma that

\[
\begin{align*}
n & \geq 2^{\lceil h/2 \rceil - 1} - 1 \\
n + 1 & \geq 2^{\lceil h/2 \rceil - 1} \\
\log(n + 1) & \geq (\lceil h/2 \rceil - 1) \log 2 \\
1 + \log(n + 1) & \geq \lceil h/2 \rceil \\
1 + \log(n + 1) & \geq h/2 \\
2 + 2\log(n + 1) & \geq h
\end{align*}
\]