The Dictionary ADT & Binary Search Trees

Florestan Brunck

February 28, 2020

This introduction to dictionaries builds from a transcription of a lecture given by Luc Devroye in the winter semester of 2019 for the undergraduate class on Data Structures and Algorithms at McGill University (COMP 251). This lecture introduces the dictionary ADT and its implementation through binary search trees.

1 *Introduction*

The purpose of a **dictionary** is to store a collection of items in a way that makes them easily and quickly accessible by a search. To that effect, each object is associated a **key**, typically a value or elements of a set which has been endowed with a total order[1].

[1] We can for example think of an English dictionary, in which the items are the words and the key of a particular word is the word itself, seen as a number in binary (recall that each letter can be attributed a value ranging from 0 to 256 in the ASCII code.

The default operations of the dictionary ADT are the SEARCH operation, together with the INSERT and DELETE operation.

Additionally, some optional operations that are often expected to be implemented in a dictionary are the sometimes called *browsing* operations. Namely the MAX/MIN operation, which return the item whose key is maximal/minimal, and the SUCCESSOR/PREDECESSOR operations, which take as input an item and return the item whose key is the next/previous value in the total order.

2 *Implementations*

Dictionaries can be implemented with a variety of data structures, each with their own advantages and drawbacks. The particular choice of data structure must tailor to the problem at hand and make use of the nature and structure of the objects being stored.

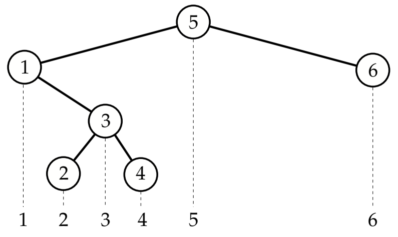

Figure 1: An implementation of a dictionary storing the numbers $\{1, 2, 3, 4, 5\}$ in a binary tree.

The main data structures are:

Binary Search Trees (BST)

Balanced Trees (e.g. red-black trees)

Hash Tables

B-Trees

Tries & Suffix Trees

Because of their simplicity, binary search trees are a fundamental data structure to which we will dedicate the rest of this lecture. Some of the others data structures also have dedicated lectures in this course.

3 *Binary Search Trees*

Definition 1. A **binary search tree (BST)** is a binary tree in which each node stores exactly one key, with the added requirement that each node has the *search tree property*. For each node, this property enforces that all the nodes situated in its left (resp. right) subtree have lower (resp. higher) key values (see Fig. 8).

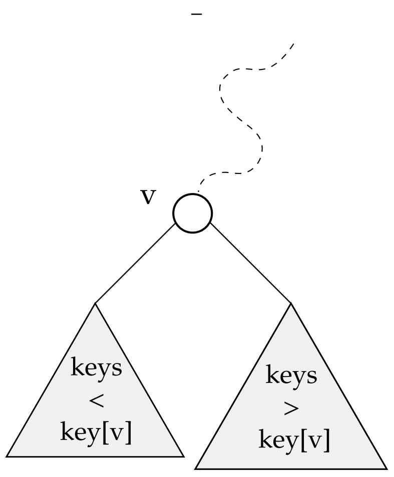

Figure 2: The Search Tree Property.

4 *Binary Search Trees on the Entropy Scale*

The problem at hand when building a dictionary is to store a list of ordered items in a way that makes them easily accessible via a search. We can therefore ask ourselves what makes BSTs good candidates for dictionaries. A useful tool in that regard is what one may call the *entropy scale*. On that scale we measure the amount of effort (i.e. the number of comparisons or use of elementary operations) needed to create the data structure from an unordered list. We can think about this amount of effort as *entropy* in the sense that it measures the amount of order in a given data structure. An unordered list realising the maximal amount of disorder with no given structure while an ordered list stands as the maximal amount of order, with the most structure. Indeed for an ordered list no effort (just printing the elements one by one) is required to output the ordered list (n comparisons).

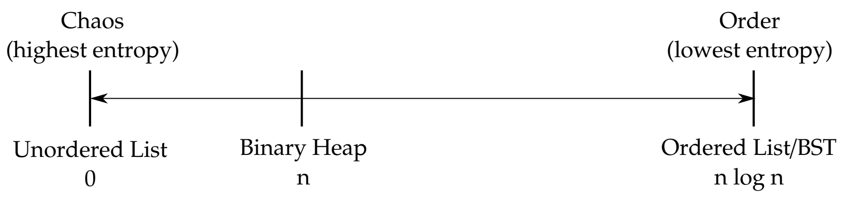

Figure 3: The entropy scale.

We can position the various potential data structures on that scale depending on how much work it takes to construct them from an unordered list. Where do binary search trees stand on that list? Well, notice first that to output an ordered list from a binary search tree, all that is required is to traverse the tree in order. Indeed in this traversal we first visit the left subtree, then the node itself and then the right subtree, the search tree property then guarantees that the output is the ordered list. Recall then from the lecture on trees that traversing a tree takes order $O(n)$. So a binary search tree really has the same order/entropy as an ordered list!

5 *Basic Operations*

In this section we provide an overview of the elementary operations that come with the dictionary ADT and describe their explicit implementations for binary search trees.

The first operation is the SEARCH operation, which allows us to retrieve the item associated to a given key.

```
SEARCH(k, x)
// Returns the node in a tree with root $t$ that has the key $k$
**if** $t = \text{nil}$ or $\text{key}[t] = k$
    return $t$
**elseif** $k < \text{key}[x]$
    SEARCH(left[$x$], $k$)
**elseif** $k > \text{key}[x]$
    SEARCH(right[$x$], $k$)
```

Exercise 2. *Re-write the* SEARCH *algorithm non-recursively.*

The second most elementary operation is the INSERT operation, which allows us to create and add a new item with a given key and place it at the right position in a given binary search tree.

```
INSERT(k, t)
// Inserts the node with the key $k$ in the BST rooted at $t$
// We start by positioning ourselves at the right node in the BST
while $x \neq \text{nil}$
    $y \longleftarrow x$
    if $k < \text{key}[x]$
        $x \longleftarrow \text{left}[x]$
    else
        $x \longleftarrow \text{right}[x]$
// We then create a new cell $x$ with the given $k$
// at that position in the tree
$\text{key}[x] \longleftarrow k$
$\text{parent}[x] \longleftarrow y$
if $k < \text{key}[y]$
    $\text{left}[y] \longleftarrow x$
else
    $\text{right}[y] \longleftarrow x$
$\text{left}[x] \longleftarrow \text{nil}$
$\text{right}[x] \longleftarrow \text{nil}$
```

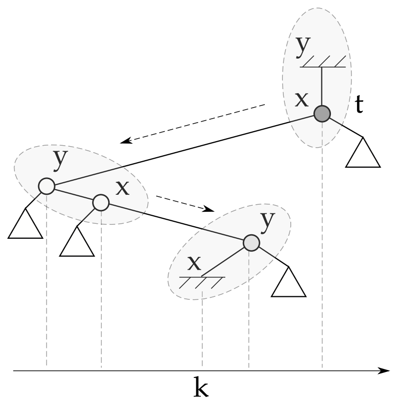

Figure 4: The INSERT algorithm uses a travelling pair of pointers to descend the binary search tree from the root to the correct position.

Conversely, we also need to be able to delete an item with a given key and update the tree so that it still retains the search tree property.

```
DELETE($x$)
// Delete the node $x$
$y \longleftarrow$ parent[$x$]]
// Case 0: the node $x$ is the root
if $y =$ nil
    root $\longleftarrow$ nil
// Case 1: $x$ has no children
if $y \neq$ nil and left[$y$] $= x$
    left[$y$] $\longleftarrow$ nil
if $y \neq$ nil and right[$y$] $= x$
    right[$y$] $\longleftarrow$ nil
// Case 2: $x$ only has a left child
// Either $x$ is left child of $y$
if $y \neq$ nil and left[$y$] $= x$
    left[$y$] $\longleftarrow$ left[$x$]
    parent[left[$x$]] $\longleftarrow y$
// Or $x$ is right child of $y$
if $y \neq$ nil and right[$y$] $= x$
    right[$y$] $\longleftarrow$ left[$x$]
    parent[left[$x$]] $\longleftarrow y$
// Case 3: $x$ only has a right child
// This case is symmetric to case 2
// Case 4: the node $x$ has two children
else
// Here $L$ denotes the left subtree
// based at $x$
    Let $l \longleftarrow$ maximum($L$)
    key[$x$] $\longleftarrow$ key[$l$]
    DELETE($l$)
    // Note that equivalently we could
    // replace $l$ with $r \longleftarrow$ minimum($R$)
    // with $R$ denoting the right subtree
    // based at $x$
```

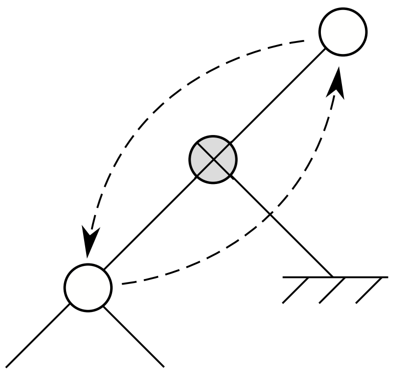
Figure 5: Case 2 of the DELETE algorithm.

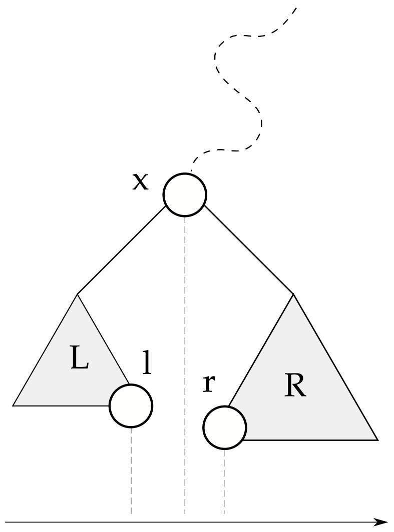

Figure 6: Case 4 of the DELETE algorithm.

```
SUCCESSOR($x$)
// Returns the node with the next key in the total order.
if right[$x$] $\neq$ nil
    return MINIMUM(right[$x$])
else
    $y \longleftarrow$ parent[$x$]
while $y \neq$ nil & $x =$ right[$y$]
    $x \longleftarrow y$
    $y \longleftarrow$ parent[$x$]
return $y$
```

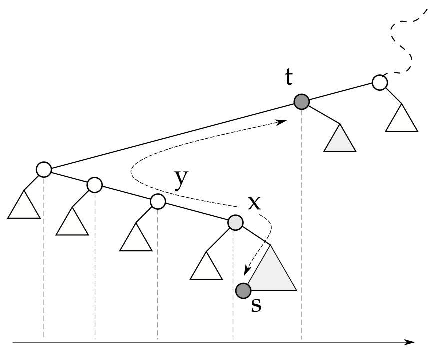

Figure 7: An example of an ordered rooted tree (or plane tree).

Since we go down a tree level at each call, the running time is at most the height of the binary search tree.

6 *Browsing in BSTs*

The BROWSING operation can be simply implemented by repeated application of the SUCCESSOR operation. Namely, browsing k-steps can be realised by k successive calls of the SUCCESSOR operation started at a given node.

What is the complexity of this operation? Well, the SUCCESSOR operation takes $O(h)$ so a naive first answer would be to multiply this by the number of steps to obtain a running time of $O(h \cdot k)$. However we can show that the running time is $O(h + k)$.

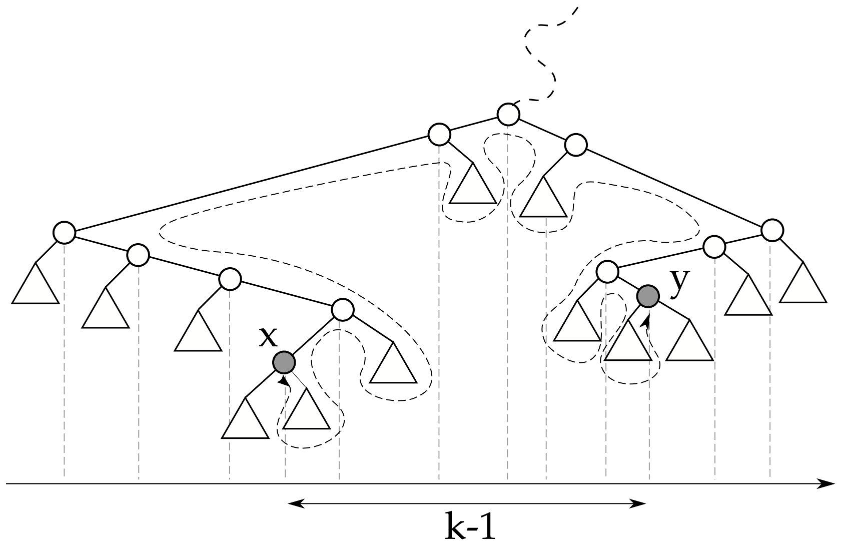

Figure 8: The BROWSING operation, obtained by iterating the SUCCESSOR operation.

To see this notice that the repeated application of the SUCCESSOR operation amounts in the path indicated in dashed lines which consists namely of two components:

1) Going down and back up each subtree between the start and end

node (all the subtrees visited in between in a preorder traversal, think "along the coastline" as in our lectures on trees). In each subtree an amount of time equal to twice the amount of nodes in these subtree (going down and back up). But there are $k-1$ nodes in between the start and end node by construction, which means that the total time spent is $2 \cdot (k-1)$

2) Going up the tree from the start node to the least common ancestor of the start and end node, and then climbing back down to the end node. This clearly takes at most time $2 \cdot h$ where h is the height of the tree.

The overall cost of the BROWSING operation is thus $O(h+k)$.

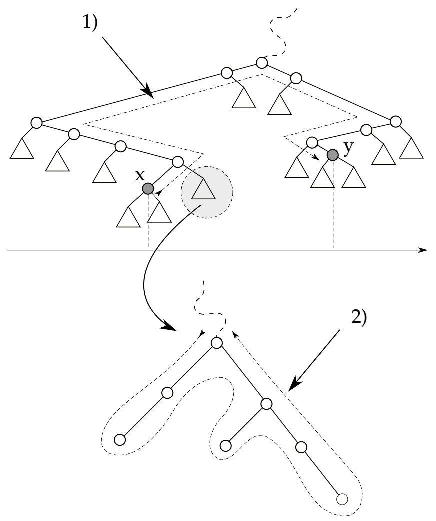

Figure 9: The complexity of the BROWSING operation can be broken down in two components.