STAT 534 Lecture 4
April, 2019
Radix trees

mmp@stat.washington.edu

Reading CLRS Exercise 12–2 page 269.

1 Representing sets of strings with radix trees

Radix trees are used to store sets of strings. Here we will discuss mostly binary strings, but they can be extended easily to store sets of strings over any alphabet.

Below is an example of a radix tree that stores strings over the alphabet $\{0, 1\}$ (binary strings). Each left-going edge is labeled with a 0 and each right-going edge with a 1. Thus, the path to any node in the tree is uniquely described as a string of 0's and 1's. The path to the root is the empty string "–". For example, the path to the leftmost node in the tree is the string "0"; the path to the lowest node in the tree is the string "1011".

Note that there can be at most 2 strings of length 1, at most 4 strings of length 2, and at most 2^h strings of length h. All the strings of a given length are at the same depth in the tree.

A **prefix** of the string $a = (a_1 a_2 \dots a_n)$ is any "beginning part" of a, i.e any string $a' = (a'_1 a'_2 \dots a'_k)$ with $k < n$ such that $a_i = a'_i$ for $i = 1, \dots k$. In a radix tree, if node b is an ancestor of node a, then the string represented by b is a prefix of the string (represented by) a. For example, "10" is a prefix and an ancestor of "100".

In order to be able to store an arbitrary set of strings in a radix tree, we adopt the following convention: nodes that correspond to a stored string are shown in white; nodes that do not correspond to a stored string, but are merely on

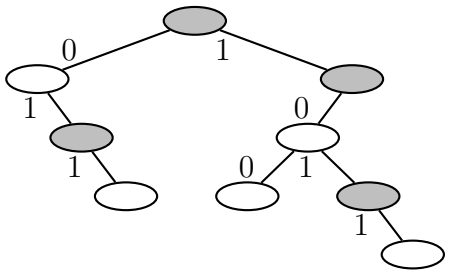


Figure 1: A radix tree storing the numbers `0, 011, 10, 100, 1011`.

the path to one, are shown in grey. For example, the tree in figure 1 stores the set $\{0, 011, 10, 100, 1011\}$. The grey node representing 101 does not store a string: it is present merely to make the path to the white node 1011 possible. Note that all the leaves of a radix tree are white nodes; there is no reason to have grey nodes that are not interior nodes. In an implementation, the "white" and "grey" colors are replaced by a local variable attached to each node, indicating whether the node is "occupied" or not.

For every set of strings, there is a unique radix tree structure that represents it. This is different from binary search trees, where every set of keys can be stored in many equivalent structures.

Radix trees are efficient representations for "dictionaries". They can save significant space. For exmple, the tree in figure 1 stores 13 symbols with only 9 nodes. There is an overhead for having a node, represented by the additional space needed for 3 pointers and the "occupied" variable. But even so there can be considerable space savings for large sets of strings. Figure 2 shows a small dictionary over english words storing 19 symbols with 13 nodes.

The worst case for a radix tree representation is the case when no two strings share any suffix. Then each string is a leaf. The total storage in this case is $\mathcal{O}($ the total number of symbols in the strings $)$.

In addition to being memory efficient, as we shall see in the next section, radix tree are as efficient as binary search trees for the operations of insertion, deletion and search.

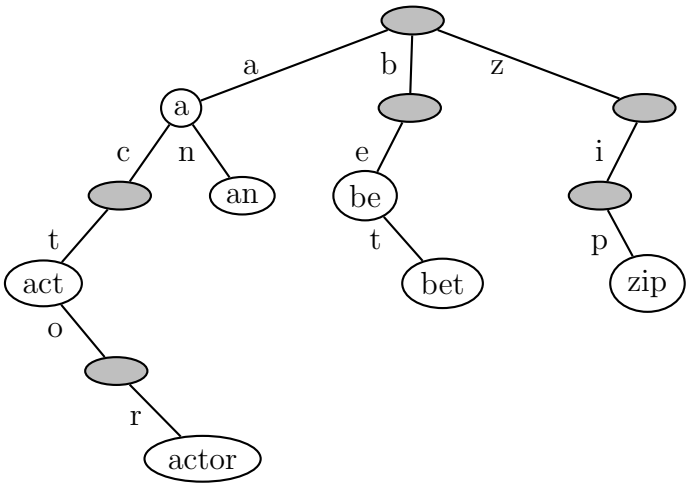
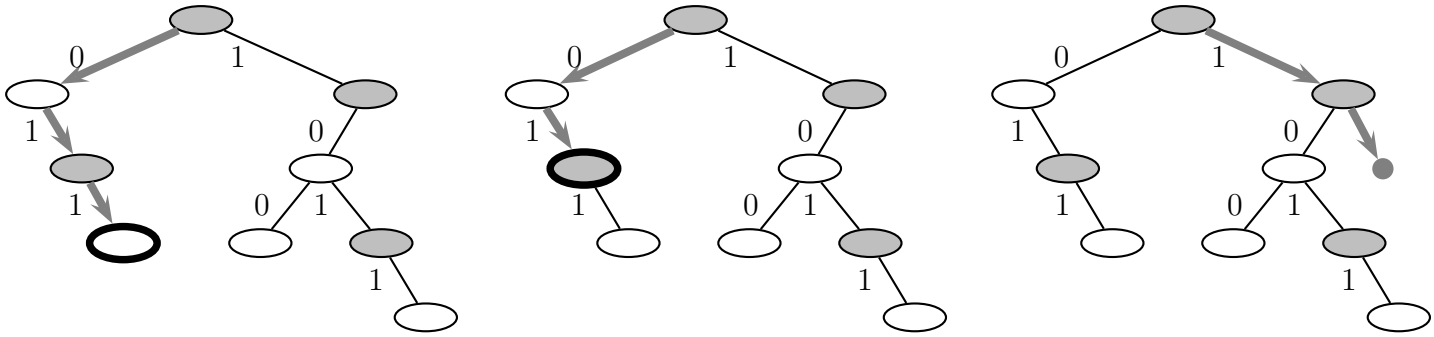


Figure 2: A radix tree storing the set of strings { `a, act, actor, an, be, bell, bet, zip`}. Note that the **branching factor** of the tree, i.e the max number of children of a node is equal to the size of the alphabet. In the figure, the null pointers are not shown.

1.1 Search, insertion and deletion in a radix tree

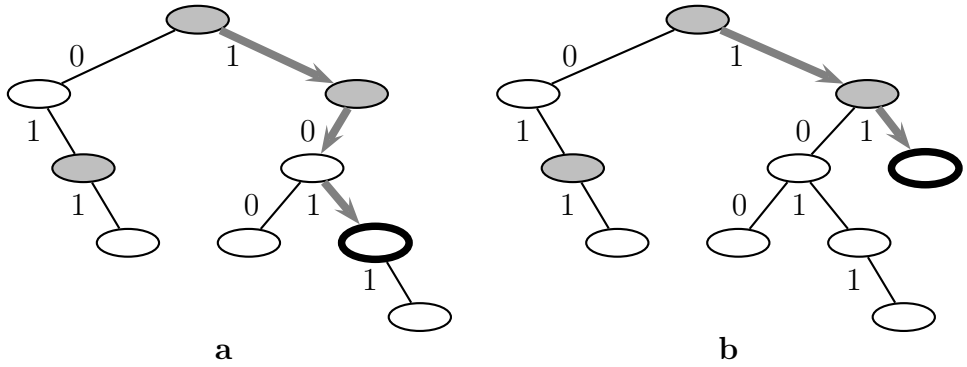
To search for a string $a = (a_1 a_2 \dots a_n)$ in a radix tree, we start from the root and follow the path described by $a_1 a_2 \dots$ down the tree. The search ends either when we find the node corresponding to the string of if we traverse a prefix of the string and then encounter a null pointer. In the latter case, the string cannot be in the tree. If the node corresponding to a exists and is white, then the string is found; otherwise, the string is not in the set. Figure 3 depicts some examples. Search cannot take longer than the shortest of the length of the string and the depth of the tree. Unlike many other search scenarios, we are often able to say that a string is not in the tree after having examined only part of the string.

Insertion in a radix tree is illustrated in figure 4. Given a string $a = (a_1 a_2 \dots a_n)$ not in a radix tree, we start from the root and follow the path described by $a_1 a_2 \dots$ down the tree as in the search procedure. The search ends either when we find the node corresponding to the string or when we encounter a null pointer. In the former case, the found node is colored white,



a. `011` found **b.** `01` not found **c.** `111` not found

Figure 3: Searching in a radix tree.



a **b**

Figure 4: Inserting `101` (a), then `11` (b) in the radix tree from figure 1. In (a), the node already exists and it is colored in white; in (b), a new leaf has to be created.

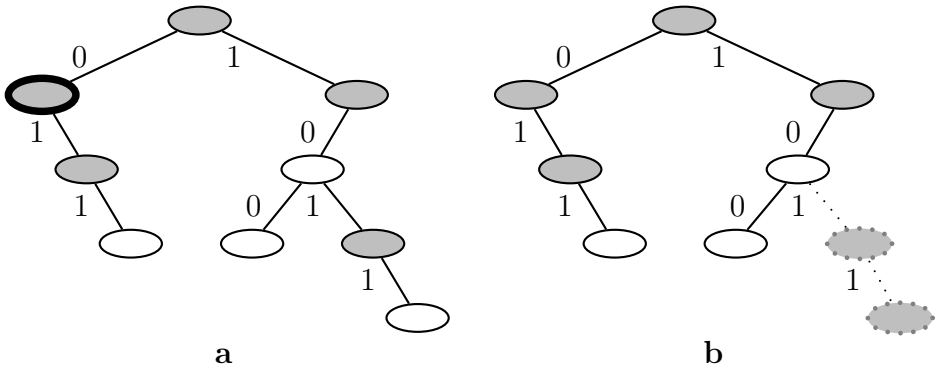


Figure 5: Deleting `0` (a) and `1011` (b) from the radix tree in figure 1. In (a), the deleted string is at an interior node; the node is colored gray. In (b) the deleted string is at a leaf; the leaf is deleted, then recursively all the other resulting gray leaves.

indicating that the string a is now in the set. This situation occurs when a is a prefix for another string already in the tree. In the latter case, assume that the last node encountered is the prefix $(a_1 a_2 \ldots a_k)$. We continue from this node by inserting the nodes corresponding to $a_{k+1},\ a_{k+2}\ \ldots a_n$ and coloring them grey, with the exception of the last one, which is colored white. Insertion thus takes time proportional to the length of the string.

For **deletion** there are two cases: (1) The deleted string is an internal node. In this case all we need to do is to color the node grey. (2) The deleted string is a leaf. If a leaf is deleted there is no reason to keep the corresponding tree node in existence, so it can be pruned from the tree. Then we can continue pruning the tree upward from the deleted leaf, removing all grey leves that appear. Pruning continues until we encounter either a white node or a grey node that has a white descendant. Figure 5,b shows an example.

2 Lexicographic sorting and preorder traversal of radix trees

The **lexicographic order** is a usual way of ordering strings of different lengths. It is the ordering used in dictionaries, hence the term "lexicographic". Any two strings $a = (a_1 \dots a_n),\ \neq\ b = (b_1 \dots b_m)$ can be compared. We have $a < b$ iff $a_i = b_i$ for $i = 1, \dots k-1$, with $k \geq 1$, and $a_k < b_k$ or $n = k-1,\ m > k-1$. Otherwise, $a > b$. A consequence of this rule is that the empty string is smaller than any other string.

In words, to compare strings a, b, we eliminate their longest common prefix, then compare the first characters of the remaining suffixes. For example, the following is a lexicographic ordering of the strings stored in the tree shown in figure 1.

```
0
011
10
100
1011
```

The lexicographic ordering of the strings stored in the tree shown in figure 2 is

```
a
act
actor
an
be
bet
zip
```

Now we show how to obtain the lexicographic ordering of the strings stored in a radix tree. First, we need to notice that radix trees have a property (or an **invariant**) similar to the binary search tree property.

Let x be any node in a radix tree. Then,

$$x\ <\ y_L\ < z_R\ \text{ for all } y_L \in \text{leftsubtree}[x],\ z_R \in \text{rightsubtree}[x]$$

where the "$<$" sign represent the lexicographic ordering.

It follows now that outputting all the strings in a radix tree in sorted order can be done by first outputting the root, then the left subtree (in sorted order), then the right subtree. This traversal of a tree is called **preorder** traversal.

PreorderPrint(x)

if $x \neq$ Null
    print x
    PreorderPrint($left[x]$)
    PreorderPrint($right[x]$)

To print the nodes of a radix tree T in lexicographic order, one needs to call PreorderPrint($root[T]$).

3 Extensions

Radix trees and their operations can be extended in many ways. Here are a few possibilities (the details of these are left as an exercise).

- **Radix trees with counts, or storing multiple identical strings.** In each node of a radix tree, a binary variable v=grey/white indicates whether the node is occupied by a string or not. By making v an integer variable, one can store a count (0 or larger) for each string in the tree. Such a data structure can be used, for example, to count the number of times each word appears in a text. Constructing the radix tree structure is linear in the number of characters in the text.
- **Radix trees with satellite data.** The strings in a radix tree can be used as keys and additional data can be stored at each white node, just like in a binary search tree. In this case, however, there can be no duplicate keys, because all records with equal keys will be stored at the same node.

- **Union and intersection of radix trees.** Denote by T_1 and T_2 two sets of strings represented as radix trees. Let $T_\cup$ and $T_\cap$ denote the set union and set intersection of T_1, T_2 represented as radix trees. It turns out that $T_\cup$, $T_\cap$ can be computed efficiently by exploiting the tree structure. The complexity of the algorithms is proportional to the size of the intersection $T_\cap$. Similar algorithms exist for computing the set differences $T_1 \setminus T_2,\ T_2 \setminus T_1$.

- **Exploiting common suffixes.** Radix trees save space by exploiting the fact that several strings may share the same prefix. For sets of words in the English words it is also the case that many words share the same suffix. For example, the final `-s` for noun plurals, suffixes like `-ment, -ing, -ion, -ed` are very common in English. One can construct structures where tree "branches" that share a suffix "meet", thus avoiding to represent the common suffix more than once. Such structures can also bring huge savings in practice. Note that the representation of a dictionary with such a structure is not unique.