Radix Searching

The most simple radix search method is digital tree searching - the binary search tree with the branch in the tree according to the bits of keys: at the first level the leading bit is used, at the second level the second leading bit, and so on until an external node is encountered. The code of the algorithm is similar to binary search tree. The data structures for the program, i.e. how to initialize the memory, are the same as those that for elementary binary search trees. The constant $maxb$ is the number of bits in the keys to be sorted:

```pascal
function digitalsearch(u: integer; x: link): link;
var b: integer;
begin
  z.key:=u; b:=maxb;
  repeat
    if bits(v, b, 1)=0 then x:=x.l else x:=x.r;
    b:=b-1;
  until v=x.key;
digitalsearch:=x
end;
```

Equal keys are anathema in radix sorting, and the same is true in radix searching. Thus, all the keys to appear in the data structure are distinct: if necessary, a linked list could be maintained for each key value of the records whose keys have that value. It is assumed that the $i$th letter of the alphabet is represented by the five-bit binary representation of $i$:

The insert procedure for digital search trees also derives directly from the corresponding procedure for binary search trees:

```pascal
function digitalinsert(v: integer; x: link): link;
var p: link; b: integer;
begin
  b:=maxb;
  repeat
    p:=x;
    if bits(v, b, 1)=0 then x:=x.l else x:=x.r;
    b:=b-1;
  until x=z;
  new(x); x.key:=u; x.l:=z; x.r:=z;
  if bits(u, b+l, l)=0 then p.l:=x else p.r:=x;
digitalinsert:=x
end;
```
Figure below shows what happens when a new key \( Z = 11010 \) is added to the tree:

The worst case for trees built with digital searching is much better than for binary search trees, if the number of keys is large and the keys are not long. The length of the longest path in a digital search tree is the length of the longest match in the leading bits between any two keys in the tree, and this is likely to be relatively small for many applications (for example, if the keys are comprised of random bits).

**Property:** A search or insertion in a digital search tree requires about \( \log_2 N \) comparisons on the average and \( b \) comparisons in the worst case in a tree built from \( N \) random \( b \)-bit keys.

It is obvious that no path will ever be any longer than the number of bits in the keys: for example, a digital search tree built from eight-character keys with, say, six bits per character will have no path longer than 48, even if there are hundreds of thousands of keys.

**Radix Search Tries**

It is quite often the case that search keys are very long, consisting of many characters. In such a situation, the cost of comparing a search key for equality with a key from the data structure can be a dominant cost which cannot be neglected. Digital tree searching uses such a comparison at each tree node; and it is possible in most cases to get by with only one comparison per search.

The idea is to not store keys in tree nodes at all, but rather to put all the keys in external nodes of the tree. Thus, we have two types of nodes:
- internal nodes, which just contain links to other nodes,
- external nodes, which contain keys and no links.

To search for a key in such a structure, we just branch according to its bits, as above, but we don't compare it to anything until we get to an external node. Each key in the tree is stored in an external node on the path described by the leading bit pattern of the key and each search key winds up at one external node, so one full key comparison completes the search:

For example, to reach E in the figure, we go left, left, right from the root, since the first three bits of E are 001; but none of the keys in the trie begin with the bits 101, because an external node is
encountered if one goes right, left, right. Before thinking about insertion, the reader should ponder the rather surprising property that the trie structure is independent of the order in which the keys are inserted: there is a unique trie for any given set of distinct keys.

As usual, after an unsuccessful search, we can insert the key sought by replacing the external node which terminated the search, provided it doesn't contain a key. If the external node which terminates the search does contain a key, then it must be replaced by an internal node which will have the key sought and the key which terminated the search in external nodes below it. Unfortunately, if these keys agree in more bit positions, it is necessary to add some external nodes which correspond to no keys in the tree (or put another way, some internal nodes with an empty external node as a child):

Implementing this method in Pascal is actually relatively complicated because of the necessity to maintain two types of nodes, both of which could be pointed to by links in internal nodes. This is an example of an algorithm for which a low-level implementation might be simpler than a high-level implementation. The left subtree of a binary radix search trie has all the keys which have 0 for the leading bit; the right subtree has all the keys which have 1 for the leading bit. This leads to an immediate correspondence with radix sorting: binary trie searching partitions the file in exactly the same way as radix exchange sorting.

Property: A search or insertion in a radix search trie requires about $\log_2 N$ bit comparisons for an average search and $b$ bit comparisons in the worst case in a tree built from $N$ random $b$-bit keys.

An annoying feature of radix tries, and one which distinguishes them from the other types of search trees we've seen, is the "one-way" branching required for keys with a large number of bits in common. For example, keys which differ only in the last bit require a path whose length is equal to the key length, no matter how many keys there are in the tree. The number of internal nodes can be somewhat larger than the number of keys.

Property: A radix search trie built from $N$ random $b$-bit keys has about $N/\log_2 N \approx 1.44N$ nodes on the average.
The height of tries is still limited by the number of bits in the keys, but we would like to consider the possibility of processing records with very long keys (say 1000 bits or more) which perhaps have some uniformity, as might arise in encoded character data. One way to shorten the paths in the trees is to use many more than two links per node (though this exacerbates the "space" problem of using too many nodes); another way is to "collapse" paths containing one-way branches into single links.

Multiway Radix Searching

For radix sorting, we could get a significant improvement in speed by considering more than one bit at a time. The same is true for radix searching: by examining \( m \) bits at a time, we can speed up the search by a factor of \( 2^m \). However, the problem is that considering \( m \) bits at a time corresponds to using tree nodes with \( M = 2^m \) links, which can lead to a considerable amount of wasted space for unused links:

Note that there is some wasted space in this tree because of the large number of unused external links. As \( M \) gets larger, this effect gets worse: it turns out that the number of links used is about \( MN/\ln M \) for random keys. On the other hand, this is a very efficient searching method: the running time is about \( \log_m N \). A reasonable compromise can be struck between the time efficiency of multiway tries and the space efficiency of other methods by using a "hybrid" method with a large value of \( M \) at the top (say the first two levels) and a small value of \( M \) (or some elementary method) at the bottom. Again, efficient implementations of such methods can be quite complicated, however, because of multiple node types.

Patricia

The radix trie searching method as outlined above has two annoying flaws: the "one-way branching" leads to the creation of extra nodes in the tree, and there are two different types of nodes in the tree, which complicates the code somewhat (especially the insertion code). D. R. Morrison discovered a way to avoid both of these problems in a method which he named Patricia ("Practical Algorithm To Retrieve Information Coded In Alphanumeric"). In the present context, Patricia allows searching for \( N \) arbitrarily long keys in a tree with just \( N \) nodes, but requires only one full key comparison per search.

One-way branching is avoided by a simple device: each node contains the index of the bit to be tested to decide which path to take out of that node. External nodes are avoided by replacing links to external nodes with links that point upwards in the tree, back to our normal type of tree node with a key and two links. But in Patricia, the keys in the nodes are not used on the way down the tree to control the search; they are merely stored there for reference when the bottom of the tree is reached:
To search in this tree, we start at the root and proceed down the tree, using the bit index in each node to tell us which bit to examine in the search key—we go right if that bit is 1, left if it is 0. The keys in the nodes are not examined at all on the way down the tree. Eventually, an upwards link is encountered: each upward link points to the unique key in the tree that has the bits that would cause a search to take that link. For example, S is the only key in the tree that matches the bit pattern 10*11. Thus if the key at the node pointed to by the first upward link encountered is equal to the search key, then the search is successful; otherwise it is unsuccessful. For tries, all searches terminate at external nodes, whereupon one full key comparison is done to determine whether or not the search was successful; for Patricia all searches terminate at upwards links, whereupon one full key comparison is done to determine whether or not the search was successful. Furthermore, it's easy to test whether a link points up, because the bit indices in the nodes (by definition) decrease as we travel down the tree. This leads to the following search code for Patricia, which is as simple as the code for radix tree or trie searching:

```plaintext
type link= | node;
    node= record key, info. b: integer; l, r: link end;
    var head.z: link;
function patriciasearch (v: integer; x: link): link;
    var p: link;
    begin
        repeat
            p:=x;
            if bits (v, x|.b, 1)=0 then x:=x|.l else x:=x|.r;
        until p|.b <= x|.b;
        patriciasearch:=x
    end;
```

This function returns a link to the unique node which could contain the record with key v. The calling routine then can test whether the search was successful or not. Thus to search for Z=11010 in the above tree we go right and then up at the right link of X. The key there is not Z, so the search is unsuccessful.

Figure below shows the result of inserting Z=11010 into the Patricia tree:
By the defining property of the tree, X is the only key in the tree for which a search would terminate at that node. If Z is inserted, there would be two such nodes, so the upward link that was followed into the node containing X must be made to point to a new node containing Z, with a bit index corresponding to the leftmost point where X and Z differ, and with two upward links: one pointing to X and the other pointing to Z. This corresponds precisely to replacing the external node containing X with a new internal node with X and Z as children in radix trie insertion, with one-way branching eliminated by including the bit index.

Inserting $T=10100$ illustrates a more complicated case. The search for T ends at $P=10000$, indicating that P is the only key in the tree with the pattern $10^*0^*$. Now, T and P differ at bit 2, a position that was skipped during the search. The requirement that the bit indices decrease as we go down the tree dictates that T be inserted between X and P, with an upward self-pointer corresponding to its own bit 2. Note carefully that the fact that bit 2 was skipped before the insertion of T implies that P and R have the same bit-2 value:

Patricia is the quintessential radix searching method: it manages to identify the bits which distinguish the search keys and build them into a data structure (with no surplus nodes) that quickly leads from any search key to the only key in the data structure that could be equal. Clearly, the same technique as used in Patricia can be used in binary radix trie searching to eliminate one-way branching, but this only exacerbates the multiple-node-type problem.

Unlike standard binary tree search, the radix methods are insensitive to the order in which keys are inserted; they depend only upon the structure of the keys themselves. For Patricia the placement of the upwards links depend on the order of insertion, but the tree structure depends only on the bits in the keys, as in the other methods. Thus, even Patricia would have trouble with a set of keys like 001, 0001, 00001, 000001, etc., but for normal key sets, the tree should be relatively well-balanced so the number of bit inspections, even for very long keys, will be roughly proportional to $\lg N$ when there are N nodes in the tree.

**Property:** A Patricia trie built from $N$ random b-bit keys has $N$ nodes and requires $\lg N$ bit comparisons for an average search.

The most useful feature of radix trie searching is that it can be done efficiently with keys of varying length. In all of the other searching methods we have seen the length of the key is "built into" the searching procedure in some way, so that the running time is dependent on the length as well as the number of the keys. The specific savings available depends on the method of bit access used. For example, suppose we have a computer which can efficiently access 8-bit "bytes" of data, and we have to search among hundreds of 1000-bit keys. Then Patricia would require accessing only about 9 or 10 bytes of the search key for the search, plus one 125-byte equality comparison, while hashing would require accessing all 125 bytes of the search key to compute the hash function plus a few equality comparisons, and comparison-based methods require several long comparisons. This effect makes Patricia (or radix trie searching with one-way branching removed) the search method of choice when very long keys are involved.