Multiway Search Trees

Shiv Shankar Dayal

Theory

A *multiway search tree* of order n is a general tree in which each node has n or fewer subtrees and contains one fewer key than it has subtrees. For example, in a binary tree a node has 2 subtrees and 1 key; similarly in multiway search trees a node may has 3 keys and 4 subtrees.

In addition, if $s_0, s_1, \ldots, s_{n-1}$ are the n subtrees of a node containing keys $k_0, k_1, \ldots, k_{n-2}$ in ascending order, all keys in subtree s_0 are less than or equal to k_0. The subtree s_i is called the *left subtree* of key k_i and its root is called *left son* of key k_i. Similarly, s_i is called the right subtree and its root the *right son* of s_i. When a node contains maximum number of keys and has maximum number of subtrees then the node is called a *full node*.

In a top-down multiway search tree any non-full node is a leaf. A *semileaf* is a node which has at least one empty subtree.

The search function can be written as below:

```
int search(NODEPTR root, int key) {
  if(root == NULL) {
    return -1;
  }

  int i = nodesearch(root, key);

  if(i < numtrees(root) - 1 && key == k(root, i)) {
    return p;
  }

  return search(son(root, i));
}
```

Here, `numtrees` equals the no. of subtrees of node. `k(root, i)` gives the $i$the key of the node. `nodesearch(root, key)` returnes the smallest integer n such that `key <= k(root, n)` or `numtrees(root) - 1` if `key` is is greater than all the nodes in node `root`.

Theory contd.

The previous search operation can be written in an iterative manner as given below:

```
int search(NODEPTR root, int key) {
  node = root;

  while(node != NULL) {
    i = nodesearch(node, key)
    if(i < numtrees(node) - 1 && key == k(node, i)) {
      return i;
    }
    node = son(node, i)
  }

  return -1;
}
```

Multiway Trees vs General/m-ary Trees

The m-ary tree can have any no. of sons of a node while a multiway trees have a limit on no. of sons. However, when the nodes are not full we waste considerable space. The reason for using multiway trees is because storing the nodes on external storage is quite cheap and is relatively expensive. In the days of rotating hard disks the read operation was relatively expensive. However, once the head is in position reading a large amount of data is relatively fast. This means that the total time for reading a storage block(a node) is only minimally affected by size of the node. Nowadays SSDs are quite fast so storing nodes on disk is even faster. Trees of order 200 or more are not uncommon.

The second factor to consider in implementing multiway search trees is storage of the data records themselves. As in any storage system, the records may be stored with the keys or remotely from the keys. The first technique requires keeping entire records within the tree nodes, whereas the second requires keeping a pointer to the associated record with each key in a node. (Another technique involves duplicating keys and keeping records only at the leaves.)

Given below is nonrecursive algorithm for searching a multiway tree stored on disk:

```
int search(NODEPTR root, int key) {
  node = root;

  while (node != NULL) {
    directread(node, block); // block is internal storage
    int i = nodesearch(block, key)

    if(i < block.numtrees - 1 && key == block.son(i)) {
      return p;
    }
    node = block.son(i);
  }

  return -1;
}
```

Traversal in multiway tree

```
int traverse(NODEPTR root, int nt) {
  if(root != NULL) {
    NODEPTR blcok;
    directread(tree, &block);
    nt = block.numtrees();

    for(int i = 0; i < nt - 1; i++) {
      traverse(block.son(i));
    }
    traverse(block.son(nt));
  }
}
```

Another common operatbn. closely related to traversal, is direct sequential access. This refers to acessing the next key fllowing a key whose location in the tree is known. Let us assume that we have located a key k by searching the tree and that it is located at position $k(n_l, i_)$. Ordinarily , the successor of $k1$ can be found by executing the following routine `next(n1, i1)` (`nullkey` is a special value indicating that a proper key cannot he found.)

Direct Sequential Access

```
NODEPTR next(n1, i1) {
  node = son(n1, i1 + 1);
  q = NULL;

  while(node != NULL) {
    q = p;
    p = son(p, 0);
  }

  if(q != NULL)
    return k(q, 0);
  if(i1 < numtrees(n1) - 2)
    return k(n1, i1 + 1);
  return nullkey;
}
```

This algorithm relies on the fact that the successor of $k1$ is the first key in the subtree that follows $k1$ in `node(n1)` or if that subtree is empty(`son(n1, i1 + 1)` equals null)and if $k1$ is not the last key in its node (`i1 <numtrees(n1) - 2`), the successor is next key in `node(n1)`.

However, if $k1$ is the last key in its node and if the subtree following it is empty, its successor can only be found by backing up the tree. Usually, it happens by stack unwinding in a recursive function but practically the path is stored in memory from root and used to go back.