TREES

Data relationships can also be represented by *trees.* Trees support quicker, random access (in a somewhat similar manner to a binary search on a table) to a record location on a file. The simplest form is a **BINARY TREE** (see Figure 1), which has a root node, intermediate nodes, and leaf nodes.

In general, a binary tree has only two offspring. That is every node (except the leaf nodes) have two offspring, and each node with offspring contains pointers to each of the offspring.

If we were to form an ordered, binary tree of data, we would require of the binary tree structure that the key of the offspring to the left of the parent node be *less* than the key of the parent node, and that the offspring to the right has a key *greater* than that of the parent node.

An example will be presented in class.

Implicit in this representation is that there are pointers (links) from each parent to each offspring.

How do we search? The basic thought is similar to the action of a binary search. If, for example, we are searching for the key of P, we first go to the root node, compare, then go to the right, since P > L. Then we compare at that node which has a key of T, and then go left. There we are!

We can generalize to multiway trees, in which we have multiple offspring, as shown in Figure 2. Figure 3 illustrates an ordered multiway tree and an implementation table.

Then, Bayer and McCreight at Boeing Corporation continued this approach with trees in an effort to devise a structure to serve as an index for relatively large random-access files that changed dynamically (i.e., caused by adds and deletes). (NOTE: The term "index" refers to a directory-type structure combined with tree concepts to direct the search to the desired record.) They came up with what is called a B-tree (because of Boeing).

One of the important elements of a B-tree is that all leaf nodes are at the same level. This can be seen in Figure 4. (Figure 5 shows an implementation of this tree.) The design and structure of the B-tree allows for node splits to accommodate insertions, and for node modification and node combination to accommodate deletions (Figures 7 and 8). Note that the nodes contain multiple keys, and each key has a pointer (address) to the data associated with that key.

B-trees provide fast random access, and they supported inserts and deletes by the dynamic modifications of the nodes.

But what shall be done if we wish to process sequentially (this is something we still want to be able to do)? The random access was great, but sequential access was laborious.

For example, if we try to access our ordered, binary tree sequentially, we will be traversing back and forth, up and down. And, we would need an additional pointer from each child to its parents.

Class example.

This is a very tortuous task.

So along came the B+ tree, as shown in Figures 9 and 10.

The essential aspects of a B+ tree are

- The leaf nodes contain ***all*** the keys, and each leaf node has multiple keys
- Each key has a pointer to the associated data record
- The upper indexes (i.e., nodes) or only there to direct the search to the proper leaf node (which can be considered a 'sub-list')
- One the proper leaf node has been identified, it is searched sequential, since the order of the keys in each leaf node is ordered.
- The leaf nodes are linked to one another. This will support ***sequential processing*** of the data.

Now we have a nice compromise: The B+ tree supports very decent random access processing, and it also supports sequential processing. And, as with the B-tree, the structure of the tree (i.e., the nodes and the links) can be modified to accommodate inserts and deletes.