B-Trees

Thomas A. Anastasio

5 April 2000

1 Introduction

B-Trees find their principal use as a data structure suitable for dictionary operations when the data are too large to be stored in main memory. When the data are in secondary storage, such as disk, the relatively slow disk access time becomes a major bottleneck for structures such as binary search trees. The B-Tree reduces the number of disk accesses needed, and therefore can significantly improve performance.

Before proceeding with our discussion of B-Trees, let's step back and consider an alternative approach to implementing binary search trees. The approach we used in preceding chapters assumed that each node of the tree stored data. That's fine, but there is another way to organize the tree - store the data only at the leaves. In this approach, the internal nodes serve only to guide the search to the leaves. We will restrict our discussion of such trees to those in which all leaves are at the same level.

Figure 1 shows an example of this different type of binary search tree. It stores the integers $\{1, 4, 7, 9, 10, 14, 16, 19\}$. Note that the data are stored in the leaves and the leaves are all at the same level. Each internal node stores a key that guides the search. Data values smaller than the key in an internal node are to be found in the node's left subtree. Larger (or equal) values are to be found in the right subtree. For example, the root node stores the key 10, indicating that data values less than 10 are in its left subtree, values greater than or equal to 10 are in its right subtree.

We can make a few observations about this type of binary search tree.

1. Each interior node stores exactly one key and has exactly two subtrees (although some of the subtrees may be empty).

2. All search paths are of the same length, equal to that of an unsuccessful search in an "ordinary" binary search tree (in which each node stores a datum). The path length for n leaf nodes will be $\lceil \lg n \rceil$. In Figure 1, there are 8 leaf nodes and the height of the tree is $\lceil \lg 8 \rceil = 3$. A corollary of this is that the tree can hold up to 2^h leaves.

3. It is possible to store multiple data elements in a leaf. For example, in the tree of Figure 1, the leftmost leaf could store the values $\{1, 2, 3\}$ with

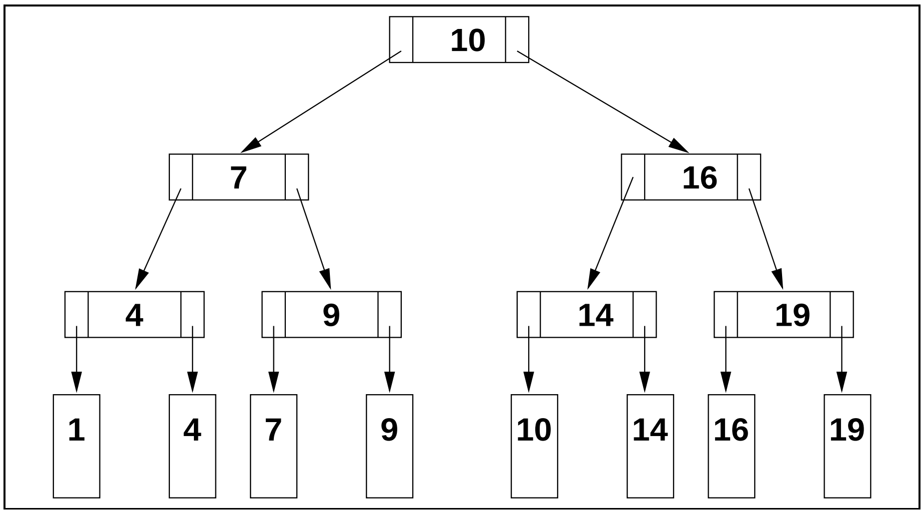

Figure 1: A BST with Data Stored in the Leaves

no other change to the tree. We will make use of this feature, but will restrict the number of elements that a leaf can store. If we did not have such a restriction, we would take the chance of having a degenerate tree in which there is one leaf that stores all the elements (*i.e.*, a list).

4. In an ordinary binary search tree, every node (interior as well as exterior) has the same structure – it stores a datum and two subtrees. In Figure 1, the interior nodes do not have the same structure as the exterior nodes.

1.1 M-way Trees

We can generalize Figure 1 so that each interior node has more than 2 subtrees. In general, these trees are called *M-way* trees, where M is the number of subtrees possible at each interior node. The tree of Figure 1 would be called a *2-way* tree, or an M-way tree of order 2. Sometimes M-way trees are called *k-ary* trees, where k has the same meaning as M in an M-way tree. We'll use the "M-way" terminology.

As M increases, the height of the tree decreases. The height of an $M - way$ tree with n leaves is $\lceil \log_M n \rceil$. Figure 2 shows an M-way tree of order 3 that stores the same data as the tree of Figure 1.

A perfect M-way tree of height h has M^h leaves. In this example, $M = 3$ and $h = 2$, so the tree can support 9 leaves, although it has only 8 leaves. The height of the tree is $\lceil log_3 8 \rceil = \lceil 1.89 \rceil = 2$, less than that of the M-way tree of order 2. Note that each interior node can hold 2 keys and can have up to 3 subtrees. In general, an interior M-way tree node can hold $M - 1$ keys and have up to M subtrees.

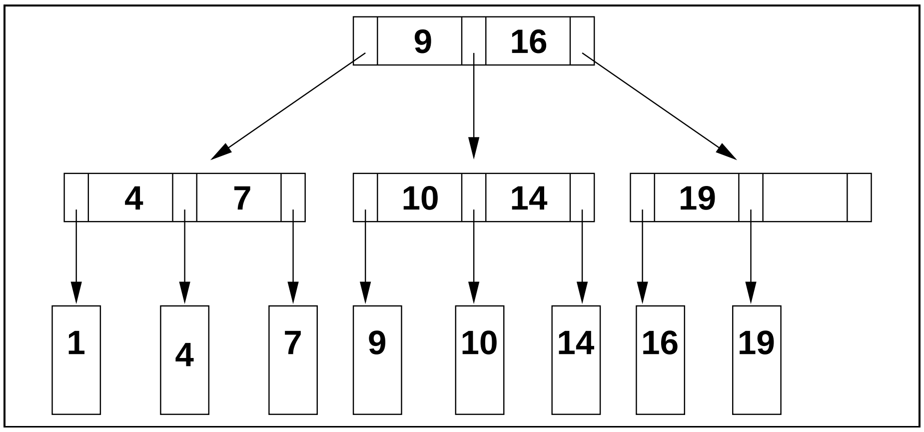

Figure 2: An M-way Tree of Order 3

One way to look at the reduced path length with increasing M is that the number of nodes to be visited in searching for a leaf is smaller for large M. We'll see that when data are stored on disk, each node visited requires one disk access. By reducing the number of nodes visited, we reduce the number of disk accesses.

1.2 M-way Nodes

Just for specificity, let's define an `MwayNode` class. These nodes can be used as interior or exterior nodes. It is certainly possible to design a more space-efficient family of nodes suitable for Mway trees, but the intent here is to show a plausible and simple implementation.

```
template <class KeyType, class DataType>
class MwayNode
{
private:
  bool _isLeaf;  // true if this is an exterior node
  int  _m;       // the order of this node
  int  _nkeys;   // number of keys actually held by this node
  KeyType * _keys; // array of key values (size = _m)
  MwayNode * _subtrees; // array of subtrees of this node
  int _l;  // max number of data elements storable in a leaf
  List<DataType> _data; // data storage if this is leaf
public:
  Constructors
  Destructors
  Accessors
  Mutators
};
```

If an `MwayNode` instance is an interior node, `_data` will be an empty list. If it is an exterior node (a leaf), _data stores one or more data items.

Note that this is a template class with two template parameters. The KeyType parameter is the type of the key by which the search is guided. This will often be int, but not necessarily. The DataType parameter is the type of the data to be stored in the leaves. To construct an MwayNode for which the data elements are char * and the keys are int, using the default constructor, we would invoke MwayNode<int, char *> foo;

1.3 Search in an M-way Tree

Not surprisingly, search in an M-way tree is a generalization of search in a binary search tree with two significant additional elements:

1. Search always proceeds to a leaf node.
2. When at the leaf, more than one datum might have to be examined.

Here's a possible version of Search. The search proceeds from MwayNode v, searching for element elm. If found, it is returned along with success being true. If not found, some arbitrary element is returned along with success being false.

```
template <class KeyType, class DataType>
DataType &
MwayTree<KeyType,DataType>::Search(MwayNode<KeyType,DataType> * v,
                                   DataType & elm, bool & success)
{
  if (v == NULL)
    {
      success = false;
      return elm;  // something handy
    }
  if (v->isLeaf() == true) // linear search of leaf for elm
    {
      ListItr<DataType> iter;
      iter = v->getData().first();
      while (!iter.isPastEnd()) // search for elm
        {
          DataType & test_elm = iter.retrieve();
          iter.advance();
          if (test_elm == elm)
            {
              success = true;
              return test_elm;
            }
        }
       success = false;
       return elm;  // an arbitrary value
    }

  for (int i = 0; i < v->GetNKeys(); i++)
    if (elm < v->getKeys()[i])
      {
        return Search(v->getSubtrees()[i], elm, success);
      }

  // there are M subtrees and (M - 1) keys
  return Search(v->getSubtrees()[v->getNKeys()], elm, success);
}
```

2 External Storage

Is it worthwhile to reduce the height of search trees by letting M increase? Although the number of nodes visited decreases, the amount of computation at each node increases. Where's the payoff?

For example, consider storing 10^7 items in a balanced binary search tree and in an M-way tree of order 10. The height of the binary search tree will be 24 ($\lg(10^7) = 23.3$) and the height of the M-way tree will be about 7 ($\lceil \log(10^7) \rceil =$

7). However, in the binary search tree, just one comparison need be done at each interior node. In the M-way tree, 9 comparisons must be done worst case.

It is generally not worth the extra computation to have a high-order tree unless somehow it takes much longer time to descend the tree than to compute the search loop at each node. When the nodes are stored externally (*e.g.*, on disk), this is precisely the situation that prevails. Disk accesses are very slow compared to computations in main memory. By increasing M, we widen the tree and require more computation, but we shorten the height - fewer nodes get visited, and therefore fewer disk accesses are needed. This leads to the following observation:

It can make sense to use M-way trees (in particular, B-Trees) when the tree nodes are stored externally in slow memory.

2.1 Disk Storage

Data is stored on disk in *blocks*, a fixed number of bytes. Each block may hold many *records*. A record corresponds to the data stored at an interior or exterior node, one node per record.

The time required to access a block is relatively long. The head must be moved to the appropriate *cylinder* and the block must then rotate to be placed under the head. Once the disk is positioned at the block, the entire block is accessed (read or write). The smallest amount of data that can be accessed on disk is a block.

By sizing the M-way tree to match the block and record sizes, we can reduce the number of disk accesses. Compared to disk access time, the time for the extra computation is insignificant.

3 B-Trees

A *B-Tree of order M* is an M-way tree with the following constraints:

1. The root is either a leaf or has between 2 and M subtrees.
2. All interior nodes (except possibly root) have between $\lceil \frac{M}{2} \rceil$ and M subtrees (*i.e.* each interior node is at least "half-full").
3. All leaves are at the same level. A leaf may store between $\lceil \frac{L}{2} \rceil$ and L data elements, where L is a fixed constant, $L \geq 1$.

Note: There are numerous varieties of B-Tree. The one described here is sometimes called a B^+ tree and is very commonly used.

Figure 3 shows a B-Tree with $M = 4$ and $L = 3$. The root node can have between 2 and $M = 4$ subtrees. Each of the other interior nodes can have between $\lceil \frac{M}{2} \rceil = \lceil \frac{4}{2} \rceil = 2$ and $M = 4$ subtrees and up to $M - 1 = 3$ keys. Each exterior node can hold between $\lceil \frac{L}{2} \rceil = \lceil \frac{3}{2} \rceil = 2$ and $L = 3$ data elements.

Each node, interior as well as exterior, is stored on disk. As the tree is traversed, the node is read into internal storage (RAM). In general, at any

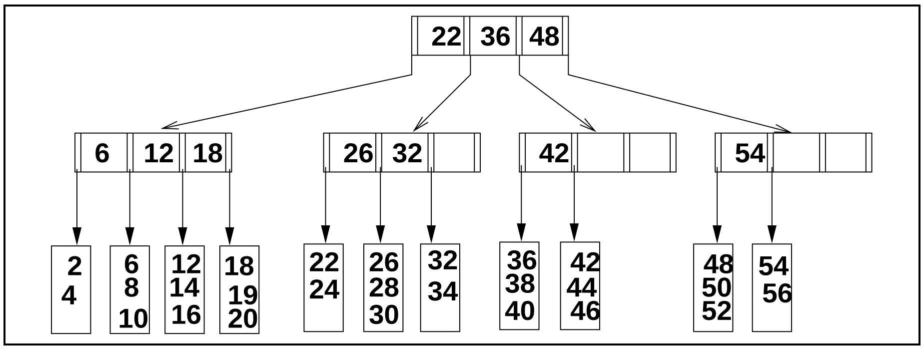

Figure 3: A B-Tree with M=4, L=3

given time during a B-Tree operation, there is only one node in internal storage. We want to design our B-tree such that each node, interior as well as exterior, fits within a disk block. This requires us to determine the values of M and L in terms of the block size, data size, key size, and pointer size.

For example, suppose the B-Tree stores student records. Let's say the key for a student record is 8 bytes long. Suppose each student record contains the student's name, address and other data totaling 1024 bytes, including the student ID.

Now, suppose a disk block hold 4096 bytes. Therefore, a disk block can store $\lfloor \frac{4096}{1024} \rfloor = 4$ student records. This leads us to choose $L = 4$ for the leaves. Each leaf will hold no more than 4 student records.

How "wide" should the B-Tree be (*i.e.*, what is the value of M)? Each interior node stores $M - 1$ keys and M pointers (where a pointer is actually a block number on disk, not a pointer into RAM). Assume a pointer requires 4 bytes. M is therefore chosen to satisfy:

$$\begin{aligned} 4M + 8(M-1) &\leq 4096 \\ 12M &\leq 4104 \\ M &\leq 342 \end{aligned}$$

So a good value for M might be 300. For N students, the height of the B-Tree will be $\lceil \log_{300}(N) \rceil$. For example for $N = 10^5$ student records (UMBC is about 1/10 of this size), the height of the B-Tree with $M = 300$ would be no more than 3 (because $\log_{300} 10^5 = 2.5$. Thus, any student record can be found in 3 disk accesses.

4 Insertion into a B-Tree

Insertion of a new item X starts with a search to find the leaf into which X belongs. If that leaf has room (it contains fewer than L items), X is added to the leaf and the leaf is written back to disk. However, if the leaf already contains

L items, it must be split into two leaves. Since we are dealing with $L+1$ items, we are guaranteed that each new leaf will contain at least $\lceil \frac{L}{2} \rceil$ items. The key values in the parent node must be updated to match the new leaf structure. Two disk writes are needed to write the new leaves. One disk read to access the parent and one disk write to write the updated parent are also required.

It is, however, possible that the parent has no room for a new leaf (it already has M subtrees). Then, it is also split in the same way. This may propagate all the way to the root. If so, a new root is created, the original root is split, and each of these "semi-roots" becomes a subtree of the new root. This shows why the B-Tree constraints allow the root to have as few as 2 subtrees.

Here's an example of insertions into the B-Tree of Figure 3 on page 7:

Insert 33 . Traversing the tree from the root, we find that 33 is less than 36 and greater than 22, leading us to the second subtree. Since 33 is greater than 32, we are led to the third leaf (the one containing 32 and 34). Since there is room for an additional data item in this leaf, it is inserted (in the correct order which means reorganizing the leaf). The resulting tree is shown in Figure 4.

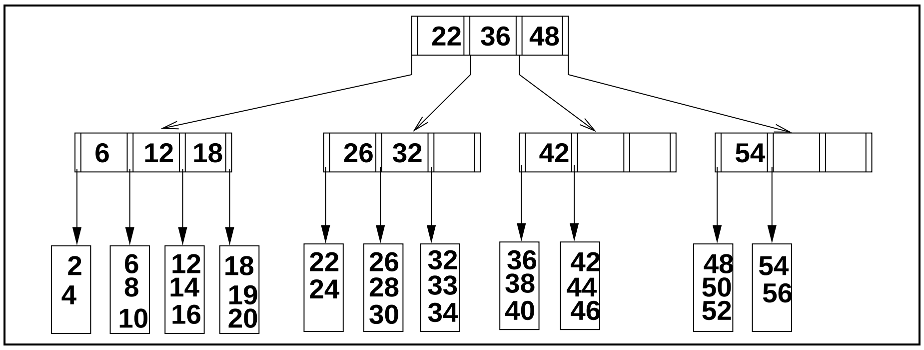

Figure 4: B-Tree After Insertion of 33

Insert 35 This item also belongs in the third leaf of the second subtree. However, it will not fit since the leaf already stores 3 items. We split the leaf into two and update the parent to get the tree of Figure 5.

Insert 21 This item belongs in the fourth leaf of the first subtree of the root (the leaf containing 18, 19, and 20). Since the leaf is full, we split it. However, its parent is also full (has 4 subtrees already), so it must also be split. That would give the root node 5 subtrees, which is not allowed, so the root must also be split. The resulting tree is shown in Figure 6.

5 Deletion From a B-Tree

Deletion involves many of the same ideas as insertion, but instead of splitting nodes, we combine them. The deletion process begins by traversing the tree to

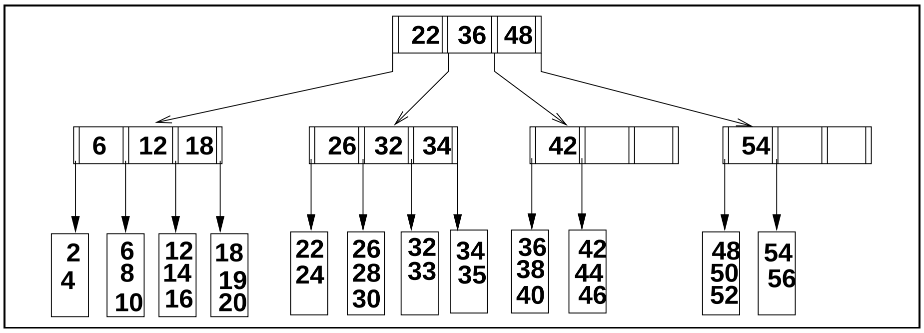

Figure 5: B-Tree After Insertion of 35

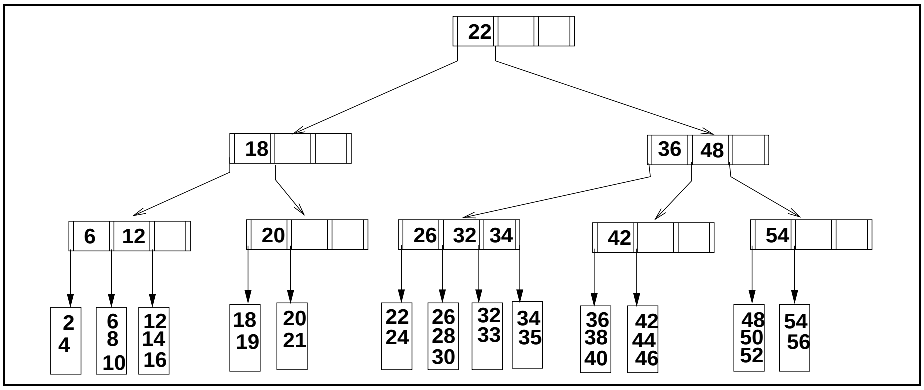

Figure 6: B-Tree After Insertion of 21

find the leaf from which the item is to be deleted. If, after the deletion, the leaf still has at least $\lceil \frac{L}{2} \rceil$ items, no further action is required - the leaf is written back to disk.

However, if the number of items in the leaf falls below $\lceil \frac{L}{2} \rceil$, we take an item from a neighboring leaf if the number of items in the neighbor would not fall below the minimum allowed. If the neighbor cannot give up an item, then we combine the leaf with its neighbor. This will work because both the leaf and its neighbor have fewer than $\lceil \frac{L}{2} \rceil$ items, so the combined number will not exceed L.

This combining process could result in the parent node having too few subtrees. In that case, the same process of borrowing or combining is used. This could take us all the way to the root. In that case, we delete the root and make its subtree be the new root.