Chapter 7

B-Trees

In the previous chapter, we saw that binary sort trees can be used to implement the operations of the SortedMap and SortedSet ADT's, and that the run time for the operations is $\Theta(h)$, where h is the height of the tree. For trees that are built by inserting items in a random order, the expected height is $\Theta(\log(n))$, where n is the number of items in the tree. However, the height can be as much as n, unless some steps are taken to keep the tree balanced.

In this chapter, we look at *B-Trees*, one approach to building balanced trees. B-Trees are a generalization of binary sort trees in which each node can have a larger, variable number of children. Insertion and deletion on a B-Tree are implemented in a way that keeps the tree perfectly balanced in the sense that all the leaf nodes of the tree are on the same level of the tree. There are several possible variations in the exact definition of B-Trees. We look in detail at one possibility and discuss some of the variations at the end of the chapter. The B-Trees in this chapter are used to implement SortedSets or SortedMaps, in which no duplicate items or keys are allowed.

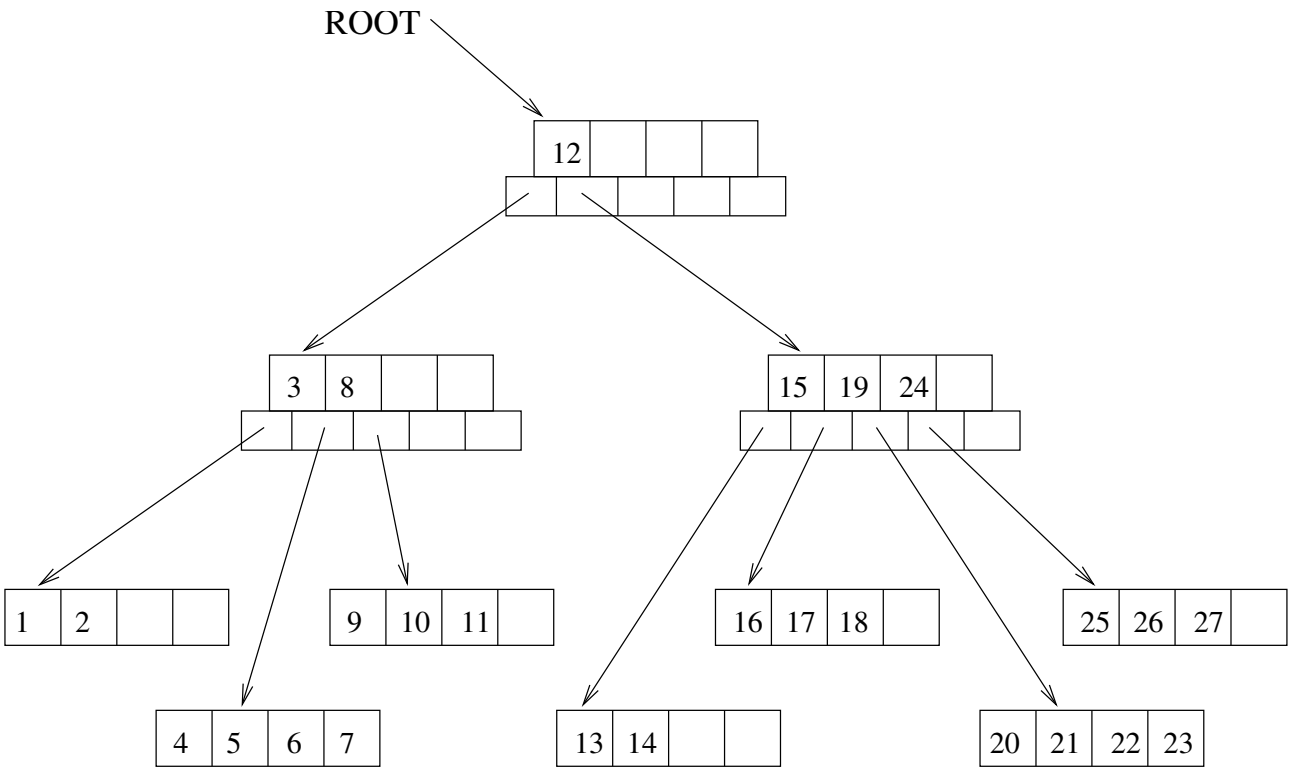
Definition 7.1. If d is an integer greater than or equal to 1, a **B-Tree of degree d** is a tree structure in which each node can contain up to $2d$ items. (When a B-Tree is used to implement a Map, the items are key/value pairs.) Every leaf node in the B-Tree is at the same level; that is, every path from the root to a leaf has the same length. This length is called the **height** of the tree. Every node, except for the root node, contains at least d items. A non-leaf node that contains c items has $c + 1$ children, so that a typical non-leaf node has between $d + 1$ and $2d + 1$ children. The items in the node and its subtrees are sorted in the following sense: The items in the node are sorted into increasing order. All items in the first subtree are less than the first item in the node. All items in the last subtree are greater than the last item in the node. And for other subtees, all the items in the i^{th} subtree are between item number $i - 1$ and item number i in the node.

For example, if the degree d is 2, then each node—except possibly for the root node—contains either 2, 3, or 4 items. And each non-leaf node—except possibly for the root—has either 3, 4, or 5 children. An exception is made for the root node because, for example, in a tree that contains only one item there is no way to avoid having a root that contains only one item.

Just as with a binary sort tree, the search, insert, and delete operations on a B-Tree have run time $\Theta(h)$, where h is the height of the tree. In a B-Tree of degree d, the height of the tree will be

$\leq \log_d(n)$, where n is the number of items in the tree, so these operations have run time $\Theta(\log(n))$. B-Trees often use a fairly large value of d, such as 100 or even 1000, so that the number of items that can be stored in the tree grows very rapidly with the height of the tree. That is, even for a very large number of items, the number of nodes visited during a search, insert, or delete operation will be quite small.

Here is a possible B-Tree structure with degree 2. Each node has space for 4 items and up to 5 children, but the child pointers are not shown in the leaf nodes. Many of the spaces in the nodes are empty. The numbers show the ordering of the items in the tree:



To be more definite, we will use B-Trees to implement sets. We assume that d is a constant that represents the degree of the B-Trees, that items in the set are of type *ItemType*, and that items can be compared using the usual comparison operators such as $<$ and $\leq$. We can then use the following type to represent nodes in a B-Tree:

```
struct BTreeNode {
   int itemCount;           // Number of items in this node.
   bool isLeaf;             // True if this is a leaf node.
   BTreeNode *parent;       // Pointer to parent node.
   ItemType item[2*d];      // Array containing the items.
   BTreeNode *child[2*d+1]; // Pointers to child nodes
};                          //    (not used in a leaf node).
```

7.1 B-Tree Operations

Using this structure, the *contains*(x) Set operation is easy to implement. The operation starts at the root and descends the tree until it either finds the item or reaches a leaf node. At a given node, it does a linear search[1] on the array of items in the node to find the first position where the item is less than or equal to x. If the item is in fact x, then the search is finished. If the item is not x, and the node is a leaf node, then the item is not in the tree and the search, again, is finished. Otherwise, the search descends into the appropriate subtree. Since the search follows a single path from the root down the tree, and it does a constant amount of work at each level of the tree, the run time for this operation is $\Theta(\log(n))$. (Note that we consider the degree, d, of the tree to be a constant.) The operation can be coded in C++ as follows:

```
bool contains(ItemType x) {
   if (!root)
      return false;
   BTreeNode *runner = root;
   while (true) {
      int pos = 0;
      while (pos < runner->itemCount && x > runner->item[pos])
         pos++;
      if (pos < runner->itemCount && x == runner->item[pos])
         return true;
      else if (runner->isLeaf)
         return false;
      else
         runner = runner->child[pos];
   }
}
```

Unfortunately, the insert and delete operations are much more complicated. Let's consider what happens when we insert a sequence of items into an initially empty tree. On the first insertion, a root node is created to contain the item. At this point, the root node is a leaf. Items can then be inserted into the root, keeping the items in increasing order, until the root contains $2d$ items. At this point, the root becomes full. When the next item is inserted, the root node must be *split* into two nodes. There are now $2d + 1$ items. A new node is created. The d smallest items are placed in the original node, and the d largest items are placed in the new node, leaving the middle item aside for the moment. This gives us two nodes that satisfy the condition that the number of items is at least d. However, we now need a new root node to point to the two nodes. A new root node is created, and the two nodes obtained by splitting the old root become children of the new root. The middle item, which we put aside previously, is placed in the new root node, as its only item. Note that items smaller than this item are in the first child of the root, while larger items are in the second child, so we have a properly formed B-Tree.

[1] Binary search could also be used.

Now, as we continue to insert items, each new item will be inserted into one of the two leaf nodes or the other. What happens when one of the leaf nodes becomes full and we want to insert an item into it? Again, a new node is created, and all the items except for the middle item are divided between the two nodes. The root already contains a pointer to the first of the two nodes. We just insert the middle item, along with a pointer to the new node, into the root node. The root now has two items and three children.

We can continue in this way, splitting leaf nodes and inserting new items and children into the root, until the root itself becomes full. At some point, we will want to do an insertion of a new item and child into a root that has become full, with $2d$ items and $2d + 1$ children. At that point, the root again splits in two, giving two nodes that have d items and $d + 1$ children, plus one extra item. Once again, we create a new root node to contain this item, and the two nodes obtained by splitting the old root become children of this new root. The height of the tree has increased from one to two.

It should not be hard to see how to continue this pattern. New items are always inserted into leaf nodes, after searching down the tree to find the item's correct position. When an item is inserted into a leaf node that is already full, a new leaf node is created. d items are left in the old node; d items are placed in the new node; and the middle item is inserted, along with a pointer to the new node, into the parent of the old node. This insertion might, in turn cause the parent node to split, resulting in an insertion into *its* parent. If this chain of insertions reaches the root, the root is split, a new root is created, and the height of the tree increases by one. Note that this splitting of the root is the only way that the height of the tree can increase.

The insertion operation starts at the root and follows a path to the bottom of the tree. It might then work its way back up all or part of the same path, performing insertions along the way. However, the amount of work that it does at each node along the path is still bounded by a constant, so that the run time of the insertion operation is $\Theta(h)$, where h is the height of the tree.

Deletion is even more complicated. If the item that we want to delete is in a leaf node, we can simply delete the item from the leaf node. However, this can cause the number of items in the node to drop below d. When this happens, assuming that the node is not the root, we no longer have a legal B-Tree, so we have to modify the structure of the tree. The idea is to look at a neighboring node. If that node has more than d items, we can rearrange the items in the two nodes so that both nodes contain at least d of them. However, if the neighboring node only has d items, we can combine the two nodes into a single node.

Let $n1$ be the node we are looking at, and let $n2$ be a node this is next to $n1$ in the tree. (If $n1$ is child number i of its parent, we can take $n2$ to be child number $i - 1$ of the parent, unless $n1$ is the first child of its parent. In that case, $n2$ can be taken to be the second child.) Let x be the corresponding item in the parent node, which divides the items in $n1$ from the items in $n2$. Consider all the item in $n1$, plus all the items in $n2$, plus x. If the total number of items is $2d + 1$ or more, replace x in the parent node with the middle item and divide the remaining items evenly between $n1$ and $n2$. This will give two nodes that each contain at least d items. On the other hand, if the total number of items is $2d$, we can delete $n2$ and place all $2d$ items (including x) in $n1$. Then, we can delete x and the pointer to $n2$ from the parent node. This might, in turn, cause the number of items in the parent node to drop below the legal minimum. In that case, we apply a

similar procedure to the parent node. That is, either we move some items from one of the parent's sibling nodes, or we combine the parent with its sibling. In the later case, we proceed up another level in the tree and do another deletion.

It is possible that this chain of deletions will reach the root. Consider the case where the root has only one item and, therefore, two children. If the two children of the root are combined into one node, we will delete the last item and its associated child pointer from the root, leaving a root node that contains no items at all and a single child pointer. In that case, we can delete the current root, and make the child of the current root the new root of the tree. Note that in this case, the height of the tree decreases by one. This is the only way in which the height of a B-Tree can decrease.

This discussion applies when the item that we are deleting is in a leaf node. Suppose that we want to delete an item that is in one of the non-leaf nodes of the tree? Let x be the item that is being deleted, let n be the node that contains x, and suppose that x is in position number i in node n. Find the successor of x in the tree. We can do this by following the $(i+1)^{th}$ child pointer in node n, and then following the first child pointer in each node until we get to a leaf node. The successor of x is the first item in that leaf node. Let y be the successor of x. We replace x in node n with y. We then delete the original copy of y from the leaf node that contains it using the procedure described above for deleting an item from a leaf node.

7.2 B-Tree Variations

B-Trees are most often used for data structures that can potentially grow very large and are stored on hard disk rather than in the computer's main memory. Data is read from and written to hard disks in chunks of a uniform size. Each chunk is called a "page." Accessing a hard disk is a very slow operation, when compared to accessing data stored in the computer's memory. In fact, the time spent processing the data in a page of hard disk memory tends to be almost insignificant compared to the time it takes to read that page from disk or write it back to disk. When analyzing algorithms that access the hard disk, we tend to worry most about the number of disk accesses.

For a B-Tree stored on disk, the size of a node is generally taken to be one page of disk memory. The degree, d, of the tree is chosen so that $2d$ items and their associated pointers can just fit in one page. This means that d will be rather large and therefore the height of the tree will be very small. If d is 1000, for example, a B-Tree of height three will hold more than a billion items. The number of disk accesses needed to perform a typical search, insert, or delete operation is equal to the height of the tree (assuming that the root node is always kept in memory), so the number of disk accesses needed to perform an operation even on a very large tree is quite small.

We have been using the same data structure for the leaf nodes as for the non-leaf nodes of our B-Trees. Since leaf nodes don't have any children, this wastes all the space that is reserved for the unused child pointers in the leaf nodes. Since the large majority of nodes are leaf nodes, this is often not acceptable. Therefore different structures can be used for leaf nodes and non-leaf nodes. Leaf nodes will then contain only items, with no child pointers. The maximum number of items in a leaf node does not have to be the same as the maximum number in a non-leaf node. Therefore, we associate two numbers, d and e, with the B-Tree. Non-leaf nodes, except for the root, contain between d and $2d$ items and their associated child pointers. Leaf nodes, except for the root, contain

between e and $2e$ items. The values of d and e are chosen so that the storage requirement for a leaf node will be about the same as the storage for a non-leaf node. For a B-Tree stored on a hard disk, this size would be the size of one page of disk memory.

When a B-Tree is used to implement a Map, the items in the tree are key/value pairs. As a variation on this, the values can be stored outside the tree, with the tree containing pointers to the values. That is, an item in the tree would consist of a key and a pointer to the value associated with that key. (For data stored on hard disk, a "pointer" really means the location of the data on disk.) When a value takes up significantly more space than a pointer, which is often the case, this allows us to pack many more items into each node, which in turn helps to minimize the height of the tree.

This idea is even more important because it makes it possible to implement *secondary keys* in a database. B-Trees are often used to organize the data in a database table for efficient searching. The data in the table is made up of records. One of the fields in the record is a *primary key*, which uniquely identifies the record. The data can be physically stored in a B-Tree in which the values are records and the keys are the primary keys of the records. This makes it very efficient to search for a record given its primary key. However, it is often useful to be able to search based on other information in the records. This other information is a secondary key. For example, in a table of student information in which the primary key is a student ID number, we might want to search for the student with a given name. The name would be a secondary key. We can build a second B-Tree in which the keys are names and the values are pointers to the actual records that are physically stored in another B-Tree.[2] By using multiple B-Trees, we make it possible to efficiently search the data based on any number of different criteria.

As a final variation on the theme of B-Trees, we can look at B^+-Trees.[3] B^+-Trees are used to implement Maps, not Sets. In a B^+-Tree, all key/value pairs are stored in leaf nodes. Non-leaf nodes contain only keys and child pointers, and the keys are copies of keys that are in the leaf nodes. The i^{th} key in a non-leaf node is a copy of the largest key in the i^{th} subtree of that node. Given a key, this makes it easy to determine which subtree of the node will contain the key/value pair for that key.

Some operations on B^+-Trees are simpler than the corresponding operations on B-Trees. However, the main advantage of a B^+-Tree is that storing pointers rather than values in the non-leaf nodes maximizes the number of items in a node and therefore minimizes the height of the tree.

[2] A B-Tree used in this way would have to allow duplicate keys, since a secondary key does not in general identify a unique record. There can be several students who have the same name. However, B-Trees can easily be extended to accommodate duplicate keys.

[3] There is no explanation for the name, unless it is supposed to imply that B^+-Trees are a little better than B-Trees.