6
B-Trees

6.1 Higher-Order Search Trees

A simple way to generalize the binary search trees seen in previous chapters is to define m-ary search trees, for $m \geq 2$, in which any node may have up to m children and store up to $m - 1$ numbers. Binary search trees are the special case $m = 2$. Just as the only element x in a node v_x of a binary search tree partitions the elements stored in the subtrees of v_x into those that are smaller than x and stored in the left subtree, and those that are larger than x and stored in the right subtree, there are $k - 1$ elements $x_1, x_2, \ldots, x_{k-1}$ stored in a node $v_{x_1,x_2,\ldots,x_{k-1}}$ of an m-ary search tree, with $k \leq m$, and they partition the elements in the subtrees of this node into k disjoint sets of numbers, each corresponding to one of the subtrees.

The partition is done generalizing the ordering induced by binary search trees:

- The first (leftmost) subtree contains elements that are smaller than x_1.
- The ith subtree, for $2 \leq i < k$ contains elements that are larger than x_{i-1} but smaller than x_i.
- The kth (rightmost) subtree contains elements that are larger than x_{k-1}.

Figure 6.1 displays an example search tree for $m = 4$. The number of elements stored in each node is thus between 1 and 3. For instance, the root stores two elements, and partitions the rest of the elements into three subtrees, one for elements smaller than 23, one for elements between 23 and 56, and one for elements larger than 56. Arrows not leading to other nodes represent NIL-pointers.

One of the applications of these higher-order search trees is to files so large that they do not fit any more into the available memory. They have therefore to be split into *pages*, most of which are stored on some secondary storage device. The assumption is then that an access to such a page has to be done by a read

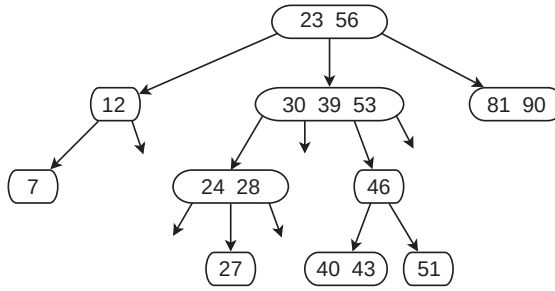


Figure 6.1. Example of a 4-ary search tree.

operation, which is much more time consuming than any processing within a given page, once it is loaded into the main memory. The idea is to choose the order m of the search tree so that a node, whose size depends on the given data, will roughly correspond to a page, whose size depends on the hardware used. Typical page sizes may be 4–8 KB, and typical values of m may be 200–400. To explain the algorithms and give manageable examples, we shall use small values of m, like $m = 4$, but one shall bear in mind that these structures are especially efficient for much larger values, like $m = 300$.

When coming to evaluate the processing time, one can thus restrict the attention to the number of nodes that have to be accessed; the work done within a node, be it by binary or by sequential search, will be deemed as negligible relative to the expensive read operations.

6.1.1 Searching in m-ary Search Trees

As in the binary case, the search for a number y always starts at the root $v_{x_1,x_2,\ldots,x_{k-1}}$ of the tree, only that several comparisons are needed to check whether y is stored there. If not, one continues recursively with the first pointer if $y < x_1$, with the ith pointer if $x_{i-1} < y < x_i$, for $2 \leq i < k$, or with the last pointer if $y > x_{k-1}$. Reaching a NIL-pointer means that an element with value y is not in the tree. As for $m = 2$, the number of required steps is at most the depth of the tree.

Referring to Figure 6.1 and searching for $y = 29$, we see that y is not in the root, but that $23 < y < 56$, so we follow the second pointer; y is not in $v_{30,39,53}$ and $y < 30$, so the first pointer is chosen, leading to $v_{24,28}$; y is larger than 28, but the last pointer is NIL – we conclude that 29 is not in the tree.

To continue the similar treatment given here to higher-order search trees, we should now turn to how to insert or delete elements. We shall, however, skip over these details, because m-ary search trees without further constraints

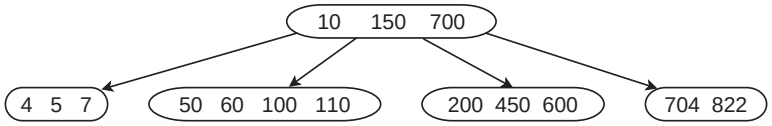


Figure 6.2. Example of a B-tree of order $m = 5$.

suffer from the same deficiencies as did their binary counterparts: in the worst case, the depth of the tree, on which the complexities of all required operations depend, may be $\Omega(n)$. We saw AVL trees in the previous chapter as a possible solution to this problem in the binary case. The alternative suggested in the next section presents a completely different approach.

6.2 Definition of B-Trees

As earlier, the solution will be some additional constraints, strong enough to imply a logarithmic depth of the tree, yet weak enough to be maintained efficiently when the tree is updated by insertions and deletions.

Definition 6.1. A *B-tree* of order m, with $m > 2$, is an m-ary search tree with the following additional constraints:

(i) Every node, except the root, stores at least $\lceil \frac{m}{2} \rceil - 1$ elements, and the root stores at least one element.
(ii) NIL-pointers emanate only from nodes on the lowest level.

The second condition implies that all the leaves of a B-tree are on the same level; this shows that the search tree in Figure 6.1 is not a B-tree. The first condition, together with the definition of m-ary search trees, bounds the number of children of an internal node, which is not the root, in a B-tree to be between $\lceil \frac{m}{2} \rceil$ and m, and accordingly, the number of stored elements to be between $\lceil \frac{m}{2} \rceil - 1$ and $m - 1$.

Because any newly allocated node would anyway reserve space for at least $m - 1$ elements and m pointers, imposing the lower bound of the first condition helps to avoid a waste of space. Requiring that any node should at least be about half full implies that the space utilized by the data structure is exploited at least to 50%.

Figure 6.2 shows a B-tree of order $m = 5$, so that any node may have between 3 and 5 children and contain between 2 and 4 elements. In this example, all the leaves are on level 1.

To evaluate the depth of a B-tree or order m, consider the lower bound on the number of elements stored at each level. At level 0, the root may contain a single element. At level 1, there are at least two nodes, each with roughly $\frac{m}{2}$ elements. At level 2, there are at least $2\frac{m}{2}$ nodes, with about $2\left(\frac{m}{2}\right)^2$ elements. For the general case, at level t, there are at least $2\left(\frac{m}{2}\right)^{t-1}$ nodes, with about $2\left(\frac{m}{2}\right)^t$ elements. If one considers a B-tree of order m storing N elements, N must be larger than the number of elements stored in the leaves, thus

$$N > 2\left(\frac{m}{2}\right)^t,$$

from which one can derive that

$$t < \log_{m/2}\frac{N}{2} = \frac{\log_2 N - 1}{\log_2 m - 1}.$$

To give a numerical example, suppose a huge file of half a billion items is to be handled. The depth of a completely balanced (full) binary tree would be 29 and that of an AVL tree at most 42, while a B-tree of order $m = 256$ would reach at most level $t = 4$. We conclude that typical B-trees are extremely flat structures with very little depth, even for very large files.

What has to be discussed next is how to perform the basic operations on B-trees. Search has already been discussed for general m-ary search trees, which leaves insertions and deletions. They are dealt with in the following sections.

6.3 Insertion into B-Trees

To insert a new value y into a B-tree, one first has to verify that y is not yet in the tree, which can be done by searching for y. An unsuccessful search is detected when a NIL-pointer is encountered, which in our case means that a leaf $v_{x_1,x_2,\ldots,x_{k-1}}$ on the lowest level has been reached. If $k < m$, that is, the node has not yet reached its full capacity, the number y is just inserted at the proper place in the sequence $x_1, x_2, \ldots, x_{k-1}$.

For example, to insert the value 9 into the tree of Figure 6.2, one reaches the leaf $v_{4,5,7}$; it contains only three values but could accommodate four, so the value 9 is added to form the sequence 4, 5, 7, 9, and no further action needs to be taken.

Suppose we want now to insert the value 55. The corresponding leaf is $v_{50,60,100,110}$, but it is already filled to capacity. This is the case $k = m$, for which the content of the node cannot be further extended. The solution in this case is applied in several steps. In a first step, the new element is inserted into its proper

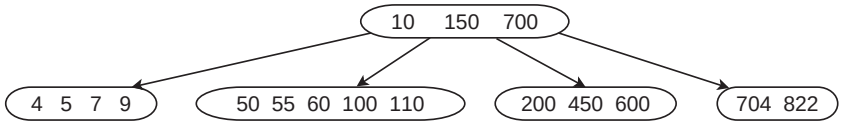


Figure 6.3. After inserting 9 and 55.

position in the node, disregarding the overflow it causes. Figure 6.3 shows the tree at this stage.

To rectify the situation, the violating node with m elements will be split into two nodes, each containing about half of the elements, that is, the list $\mathsf{S} = r_1, \ldots, r_m$ of elements stored in this node is split into

$$\mathsf{SL} = r_1, \ldots, r_{\lceil m/2 \rceil - 1} \qquad \text{and} \qquad \mathsf{SR} = r_{\lceil m/2 \rceil + 1}, \ldots, r_m,$$

the node v_{S} is split into v_{SL} and v_{SR}, and the middle element $r_{\lceil m/2 \rceil}$, which has not been included in either SL or SR, will be inserted, recursively, into their parent node. To understand this operation, notice that when a node v is split into two, this is equivalent to inserting a new node w as an immediate brother of v, that is, as an additional child of $p(v)$, the parent node of v. But if $p(v)$ has an additional child, it needs an additional pointer to it, and thus, by the rules of search trees, an additional value to separate the corresponding subtrees. The most convenient choice for this value is one of the middle of the list of values originally stored in v.

To continue our running example, $\lceil \frac{m}{2} \rceil = 3$, the node $v_{50,55,60,100,110}$ is split into $v_{50,55}$ and $v_{100,110}$, and the middle element, 60, is inserted into the parent node, which is the root. There is enough space in the root for an additional element, so the insertion of 55 is thereby completed. The resulting tree is given in Figure 6.4.

The following is an example in which the insertion of a new element triggers a chain reaction. Let us insert the value 3, so the first step is to store it in $v_{4,5,7,9}$, and this leads to an overflow, as shown in Figure 6.5.

The second step of the insertion of 3 is therefore splitting the node $v_{3,4,5,7,9}$ into $v_{3,4}$ and $v_{7,9}$, and inserting the middle element 5 into the parent node, which is the root. This is shown in Figure 6.6.

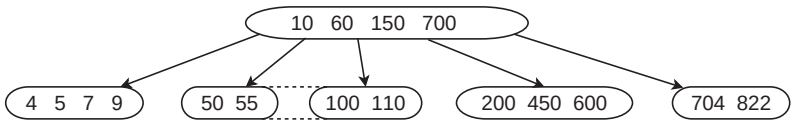


Figure 6.4. Example of splitting a node.

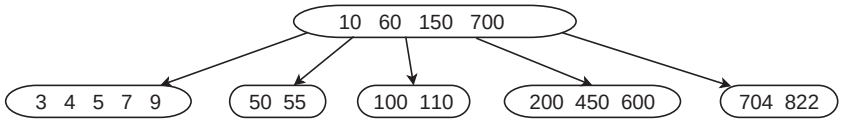


Figure 6.5. After inserting 3.

The problem is that the insertion of 5 into the root causes again an overflow, and the node $v_{5,10,60,150,700}$ has also to be split. Since this is not a leaf, the splitting process has also to take the corresponding pointers into account. In the general case, a node containing the m values $r_1, \ldots, r_m$ and the corresponding $m+1$ pointers $p_0, p_1, \ldots, p_m$, will be split into

$$\begin{array}{ccc} & r_{\lceil m/2 \rceil} & \\ r_1, \ldots, r_{\lceil m/2 \rceil - 1} & & r_{\lceil m/2 \rceil + 1}, \ldots, r_m \\ p_0, p_1, \ldots, p_{\lceil m/2 \rceil - 1} & & p_{\lceil m/2 \rceil}, \ldots, p_{m-1}, p_m \end{array} ,$$

where the top line shows the middle element inserted into the parent node, and the left and right parts show the contents of the new nodes, namely the values and the corresponding pointers. Note that while the set of m values has been distributed over three nodes, the whole set of $m+1$ pointers is accounted for in the two new nodes.

The insertion of $r_{\lceil m/2 \rceil}$ may in turn cause an overflow, and this chain reaction may continue for several steps. It may terminate either by reaching a node that can still absorb the element to be inserted without exceeding its limits, or when getting to the root, as in our example.

When the root has to be split, there exists no parent node into which $r_{\lceil m/2 \rceil}$ could be inserted, so a new node is created for this purpose, and this will be the new root. The only element stored in it will be $r_{\lceil m/2 \rceil}$, and its two pointers will lead to the two halves created from the root that has been split, as shown in Figure 6.7. This special case was the reason for formulating a different condition for the root than for the other nodes in the definition of a B-tree.

As a result of splitting the root, the B-tree has thus increased its depth by 1, and this is the only possibility for a B-tree to grow in height. Contrary to binary search trees, for which each inserted element creates a new leaf, the growth process of B-trees proceeds "backwards," and therefore, no special care

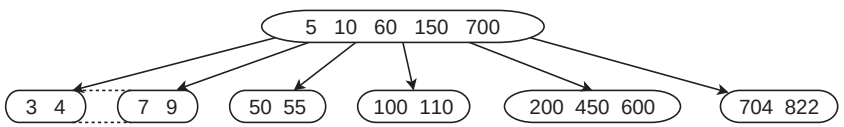


Figure 6.6. After splitting a node.

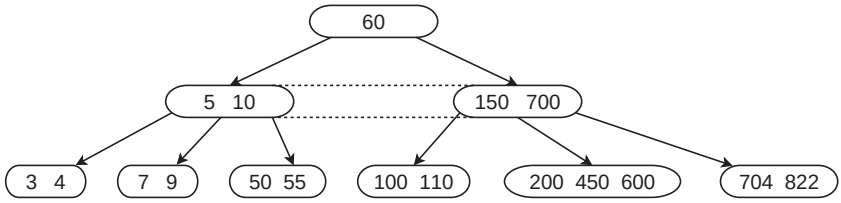


Figure 6.7. After a second split.

is needed during insertions to maintain the condition that all the leaves have to be on the same level.

Returning to our example, the new root contains only the value 60, and its two pointers are to the nodes $v_{5,10}$ and $v_{150,700}$. This ends the insertion process for 3, which involved two node splits and changed the shape and depth of the tree.

In the worst case, the number of node splits for a single insertion may thus be as the depth of the tree. Even though the depth is generally small, this might still seem a heavy price to pay. Fortunately, the picture is not as dark as it seems when considering the worst case. This is a good example for an application in which a more interesting evaluation can be given using *amortized analysis*: rather than concentrating on the pessimistically worst choice for an insertion operation, let us consider the total cost of inserting many elements into a B-tree.

Suppose N elements are inserted into an initially empty B-tree of order m. The total number of node splits caused by the N insertions is exactly equal to the number of nodes in the tree, since each split adds a single node. If we assume that roughly each node contains about $\frac{3}{4}m$ elements, then there are about $\frac{4N}{3m}$ nodes, and the amortized number of node splits per inserted element is $\frac{4}{3m}$. For example, if $m = 400$, the overwhelming majority of insertions will not result in any split at all; only about one of 300 insertions will cause a single split and a double split like in our second example will occur only for one of 90,000 insertions.

The readers might want to check their understanding by inserting, in order, the elements 25, 43, and 32 into the tree of the running example, given in Figure 6.7. This affects the left subtree, which should look like the one in Figure 6.8.

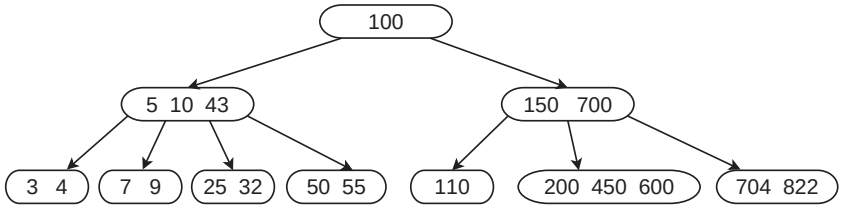


Figure 6.8. Replacing 60 by 100 and deleting 100 from its leaf.

6.4 Deletions from B-Trees

As in the previous chapters, we may restrict our discussion to the deletion of an element stored in one of the leaves. If one of the other elements should be deleted, it will be replaced by its *successor* so that the overall structure of the tree is maintained, and the successor will be erased. This has already been discussed for general binary trees and for AVL trees in particular. In the special case of a B-tree, the successor of an element x stored in an internal node is always in a leaf, and can be found as follows.

(i) Locate the node v that contains the value x, and follow the pointer immediately to its right.
(ii) While the current node is not a leaf, follow its leftmost pointer.
(iii) The successor of x is the leftmost element in the leaf that has been reached.

For example, in Figure 6.7, the successor of 700 is 704 and the successor of 60 is 100.

The simplest case of deleting an element x from a leaf is when it stores more than the minimum required $\lceil m/2 \rceil - 1$ numbers. In that case, x can simply be erased. This would be the case, for example, if we wanted to erase any of the elements of the leaf $v_{200,450,600}$ in the tree of Figure 6.7.

Suppose then that the element to be deleted is the one stored in the root, 60. It is replaced by its successor 100, which itself has now to be erased from its leaf $v_{100,110}$. This leads to an *underflow*, because the node now contains less than the minimum number of elements, which is 2. Figure 6.8 shows the tree of our running example at this stage.

If the remedy to an overflowing node was its splitting into two parts, one might have expected that the answer to an underflow is a merge. But it takes two to tango! While the possibility to split a node depends only on the node itself, a merge will only be possible if there is an appropriate partner node. The only possible candidates are therefore the immediate neighbors.

In our example, the underflowing leaf v_{110} has only one neighbor, $v_{200,450,600}$, but merging the two is not immediately possible. Recall that when a node was split for an insertion, a pointer, and therefore also a new value, had to be added to the parent node. Similarly, the fusion of two nodes reduces the number of children of their parent node, so the number of elements stored in it needs also to be decreased. We would thus like the merged node to contain the values 110, 150, 200, 450, and 600, but this is more than permitted.

The solution in this case is a step called *balancing*. In its general form, suppose a node gets below the limit, storing the elements $r_1, \ldots, r_{\lceil m/2 \rceil - 2}$, but its neighbor, say, the right one, stores the elements $s_1, \ldots, s_k$, with $k \geq \lceil m/2 \rceil$, that

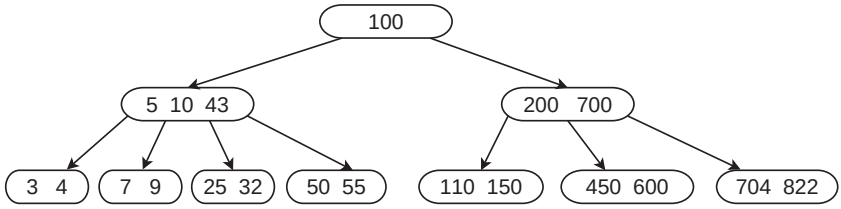


Figure 6.9. Example of a balancing step.

is, the neighbor has a surplus, above the minimum number of elements. Suppose that the separating value in their parent node is t, so that $r_{\lceil m/2\rceil-2} < t < s_1$. All these elements will then be rearranged into two sequences of roughly equal length. Specifically, the total number of elements is $\ell = (\lceil m/2\rceil - 2) + 1 + k$, and after renaming them $u_1, \ldots, u_\ell$, the left node will hold $u_1, \ldots, u_{\lceil \ell/2\rceil-1}$, the right node will hold $u_{\lceil \ell/2\rceil+1}, \ldots, u_\ell$, and the separating element in the parent node will be $u_{\lceil \ell/2\rceil}$. Figure 6.9 shows how the nodes v_{110} and $v_{200,450,600}$, with separating element 150, have been rebalanced into $v_{110,150}$ and $v_{450,600}$, with separating element 200.

If we wish now to delete the element 600, then no balancing is possible, since both neighbors of $v_{450,600}$ are at their lowest permitted levels. In this case, when one of the nodes is at its lowest limit and the other is even one below it after the deletion of one of its elements, the two nodes may be merged, absorbing also the separating element.

Merging v_{450} with, say, its right neighbor gives $v_{450,700,704,822}$, as can be seen in Figure 6.10. The value 700 has to be erased, recursively, from the parent node, which again results in an underflow. Here again, balancing can be applied. Note that since this is not performed on the lowest level, one also has to take care of the pointers in a way that is symmetrical to the treatment for insertions.

As can be seen in Figure 6.11, the nodes $v_{5,10,43}$ and v_{200}, with separating element 100 are rebalanced into $v_{5,10}$ and $v_{100,200}$, with separating element 43; the leaf $v_{50,55}$ has not been touched, but it belongs now to another subtree.

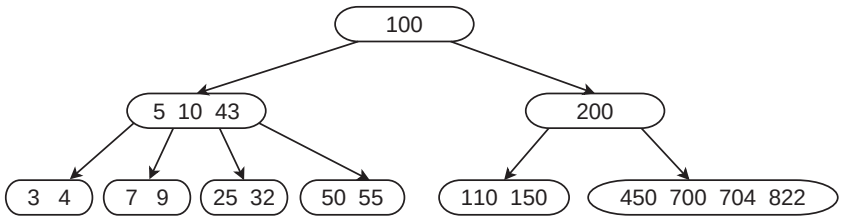


Figure 6.10. First step for the deletion of 600.

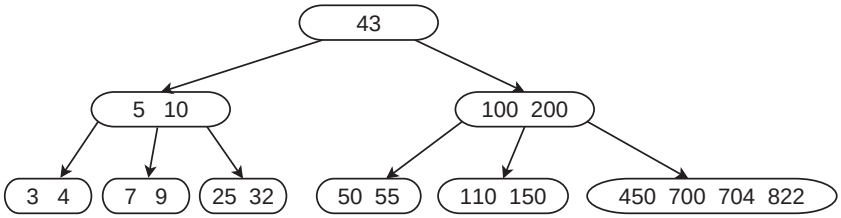


Figure 6.11. Balancing of internal nodes.

This concludes the deletion of 600, which involved a merging followed by a balancing step.

As a last example, let us delete the value 55. No balancing is possible, hence there will be a merging step at the leaf level, resulting in the tree of Figure 6.12.

As in Figure 6.10, the node v_{200} is below minimal capacity, but unlike the previous example, no balancing is possible. The two nodes $v_{5,10}$ and v_{200} should thus be fusioned, and the united node should also include the separating value 43, which has to be deleted, recursively, from the parent node. We are, however, in the special case for which 43 is the only element in its node, which is only possible if this node is the root. The old root will thus be erased and the newly merged node will be the new root, as shown in Figure 6.13.

This example shows the only possibility for a B-tree to shrink in height: when a chain of node mergers, triggered by the deletion of a single value, extends all the way up to the root, and the only value stored in the root has to be erased.

6.5 Variants

The basic definitions of B-trees have meanwhile been extended in many ways and we shall mention only a few of these variants. A first improvement may be obtained at almost no price, as in the following example.

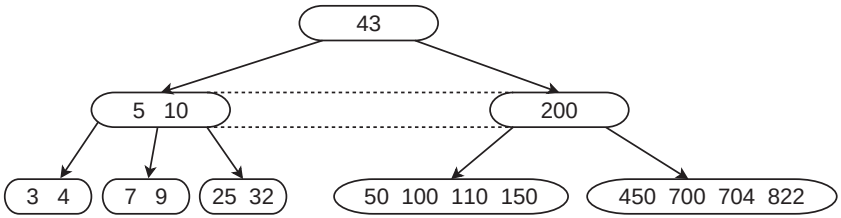


Figure 6.12. First step of the deletion of 55.

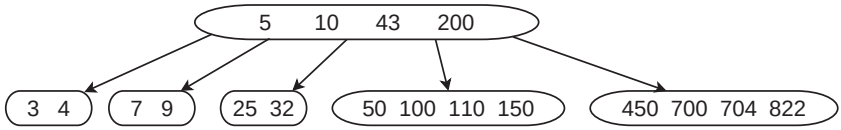


Figure 6.13. Erasing the root.

Suppose we wish to reinsert the element 55 into the tree of Figure 6.13. This would cause an overflow in the leaf $v_{50,55,100,110,150}$, which would be split into $v_{50,55}$ and $v_{110,150}$, with the middle element 100 being inserted into the parent node; this provokes a second split, this time of the root, and we get in fact back to the tree of Figure 6.11. The increase in height could however be avoided, by applying a *balancing* step with $v_{25,32}$, the left neighbor of the overflowing leaf $v_{50,55,100,110,150}$. The resulting tree is in Figure 6.14.

There is an essential difference for the use of balancing between insertions and deletions. For deletions, the balancing step prior to a merge is *mandatory*: if the neighboring node, with which the underflowing node should be merged, is not at its lowest permitted capacity, no merging is possible, so there is no other choice than balancing. On the other hand, for insertions, an overflowing node can be split without taking the status of its neighbors into consideration. A balancing step is therefore only *optional*. Since the balancing procedure needs to be written for deletion, and the same procedure can be used for insertion as well, it might be advantageous to try balancing also for insertion, as it might help reducing the height of the tree.

6.5.1 B$^+$-Trees

A useful feature for some applications is the ability to scan the entire file sequentially in order in an efficient way. For larger values of m, like those used in practice, almost the whole file resides in the leaf nodes, and less than, say, 1%, in the other levels. Yet the values stored in these internal nodes disrupt the scanning order, so it could be preferable to have *all* the values in the leaves.

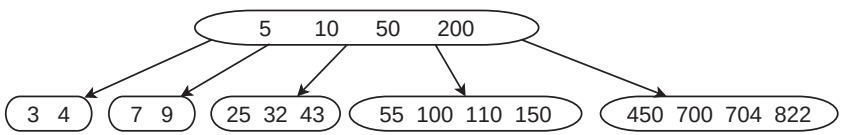


Figure 6.14. Using balancing for insertion.

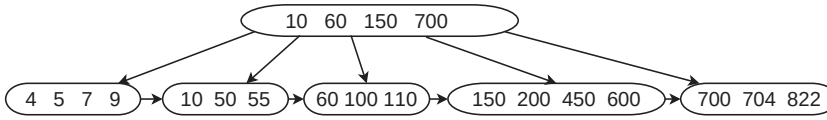


Figure 6.15. A B^+-tree.

This can be achieved with some small adaptations to the original B-trees. The resulting structure is known as a B^+-tree. The difference is that in a B^+-tree, all the data is stored in the leaves, hence they can be connected by means of pointers to form a linked list; the values in the internal nodes of the tree serve only as a guide to enable a faster access to the actual data.

Insertion

First, the leaf v into which the new element should be inserted is located. If it can accommodate another element, we are done. If the leaf overflows, it has to be split (unless balancing is applied, if possible), just that instead of moving the middle element r up to the parent node $p(v)$, it is *a copy* of r that will be inserted into $p(v)$. All the elements remain therefore at the leaf level, and the elements in the nodes above them are all copies of a part of these values.

A B^+-tree with the same elements as the B-tree of Figure 6.4 may look like the one in Figure 6.15. Pointers have been added to show how the leaves could form here a linked list.

When the insertion of the copy of the middle element into the parent node leads there too to an overflow, the insertion algorithm continues with further node splits, but like the algorithm for B-trees. That is, only on the lowest (leaf) level do we keep all of the elements and insert a copy of one of them into the parent node; when an internal node is split, we proceed as before, moving the middle element itself, and not just a copy of it. The reason for this asymmetric behavior is that one would need an additional pointer emanating from one of the newly created nodes. For the lowest level, there is no such problem, since all the pointers are NIL.

For example, insert the element 172 into the tree of Figure 6.15. The node $v_{150,\ldots,600}$ is split into $v_{150,172}$ and $v_{200,450,600}$ (including the element 200), and a copy of 200 is inserted into the root, yielding in a first stage $v_{10,60,150,200,700}$. Since this is again an overflow, the node is split, but in the regular B-tree way. The result can be seen in Figure 6.16. If the element 150 had been kept also in one of the nodes on level 1, as in, say, $v_{150,200,700}$, this would require four outgoing pointers from this node, and thus four corresponding leaves. But the

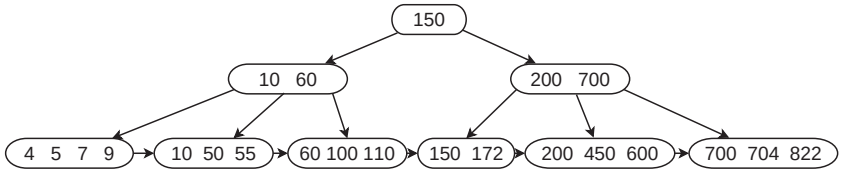


Figure 6.16. Insertion of 172.

total number of leaves in the tree is only six, not seven. The awkward distinction, applying a different split algorithm on the lowest (leaf) level than for the levels above, is therefore necessary.

Deletion

Since all the elements are in the leaves, one has only to deal with deletions at the leaf level. At first sight, it may seem that one has to check whether there is a copy of the element to be deleted in the upper part of the tree, so that the copy may also be removed. But in fact, such a check is not needed, for even if the value x is not any more in one of the leaves, it may still work perfectly well as a separator.

If the node from which an element has been removed underflows, the balancing procedure must be applied, taking care of replacing also the separating element. For example, if we remove the element 150 from the tree in Figure 6.16, the root containing a copy of 150 will not be touched, the nodes v_{172} and $v_{200,450,600}$ will be re-balanced into $v_{172,200}$ and $v_{450,600}$, and their parent node $v_{200,700}$ will be changed into $v_{450,700}$. If no balancing is possible, two nodes will be merged, and their separating element will simply be discarded. If deleting the separating element causes another underflow, we continue like for B-trees, similarly to what we have seen for the insertion.

Search

The only difference between the searches in B-trees or B^+-trees is that for the latter, the search has always to continue until the leaf level is reached. The reason is that even if the value is found in one of the internal nodes, this does not imply that the corresponding element is still in the tree; it might have been deleted.

Summarizing, the processing of B^+-trees is only slightly more involved, the time complexity is about the same and the space is only marginally increased. In many situations, this will seem as a reasonable tradeoff for getting the improved sequential scan.

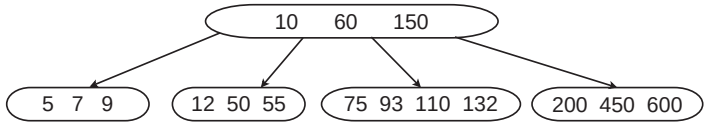


Figure 6.17. Example of a B*-tree with $m = 6$.

6.5.2 B*-Trees

Another extension of the basic B-tree concentrates on reducing the wasted space within the nodes, which can reach about 50%. This is improved in the variant called B*-trees to about $\frac{1}{3}$. The exact definition is as follows:

Definition 6.2. A *B*-tree* of order m, with $m > 2$ and m being a multiple of 3, is an m-ary search tree with the following additional constraints:

(i) Every node, except the root, stores at least $\frac{2m}{3} - 1$ elements.
(ii) The root stores at least one and at most $\frac{4m}{3} - 2$ elements.
(iii) NIL-pointers emanate only from nodes on the lowest level.

The restriction to orders that are divisible by 3 is for convenience only, as it simplifies the presentation. For the large values of m used in practice, it has no importance. Figure 6.17 is an example of a B*-tree, with $m = 6$. All the nodes store at least three elements.

Since more data than for B-trees is stored in each node, the number of nodes will tend to be smaller, and hence also the depth of the tree may be reduced. The search algorithm for B*-trees, which is the same as for B-tree, may be expected to work faster. The question is, of course, how to maintain the stricter bounds on the number of elements per node during updates.

Insertion

Unlike B-trees, for which a balancing step in case of an overflow was optional, an overfull node v in a B*-tree *requires* a balancing attempt, to assure that one of the neighbors w of v has reached full capacity. The reason is that the splitting process is applied to two neighboring nodes simultaneously, of which one should be with a maximum number of elements $m - 1$, and the other even beyond, storing, temporarily, m elements. If one adds the element separating v and w, this gives $2m$ elements, that should be redistributed into three nodes.

It will be easier to work through an example and infer from it the general case. Assume $m = 99$, so that any node has between 66 and 99 outgoing pointers and thus stores between 65 and 98 values. Suppose a new value is added to

a node v, causing an overflow, that the right neighbor w of v is full, and that the separating value between v and w is a, $r_{99} < a < r_{100}$. The situation can be described schematically like earlier as

$$\begin{array}{cc} & a \\ r_1, \ldots, r_{99} & r_{100}, \ldots, r_{197} \\ p_0, p_1, \ldots, p_{99} & q_{99}, q_{100}, \ldots, q_{197} \end{array} .$$

Splitting the two nodes into three yields

$$\begin{array}{ccc} & r_{66} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r_{132} \\ r_1, \ldots, r_{65} & r_{67}, \ldots, r_{99}, a, r_{100}, \ldots, r_{131} & r_{133}, \ldots, r_{197} \\ p_0, p_1, \ldots, p_{65} & p_{66}, p_{67}, \ldots, p_{99}, q_{99}, q_{100}, \ldots, q_{131} & q_{132}, \ldots, q_{197}. \end{array}$$

Note that the middle node contains one more element than the minimal allowed.

The insertion of r_{66} and r_{132} instead of a into the parent node may again cause an overflow, which is handled recursively, like for B-trees.

If v does not have any neighbor, it must be the root, which has to be split on its own. But if the upper bound on the number of elements in the root had been $m - 1$ like for the other nodes, the result of the split would be two nodes that contain only about $m/2$ elements, violating the lower bound. This is circumvented by defining, only for the root, which anyway already gets a special treatment, an upper bound of about $\frac{4}{3}m$, by $\frac{1}{3}$ larger than for the other nodes. More precisely, on our example, the root may hold up to 130 values. Suppose then that it overflows and denote the values by $r_1, \ldots, r_{131}$ and the pointers by $p_0, \ldots, p_{131}$. This gives, after the split,

$$\begin{array}{cc} & r_{66} \\ r_1, \ldots, r_{65} & r_{67}, \ldots, r_{131} \\ p_0, p_1, \ldots, p_{65} & p_{66}, p_{67}, \ldots, p_{131}. \end{array}$$

Deletion

As for B^+-trees, we deal only with the deletion of an element in a leaf node v. If this induces an underflow, balancing should be attempted, but this time with *both* left and right neighbors of v, if they exist at all. If no balancing is possible, this is the situation where v has one element short of the minimum, and its two neighbors are exactly at minimum capacity, as in

$$\begin{array}{ccc} & a & \qquad\qquad b \\ r_1, \ldots, r_{65} & s_1, \ldots, s_{64} & t_1, \ldots, t_{65} \\ p_0, p_1, \ldots, p_{65} & q_0, q_1, \ldots, q_{64} & y_0, y_1, \ldots, y_{65}. \end{array}$$

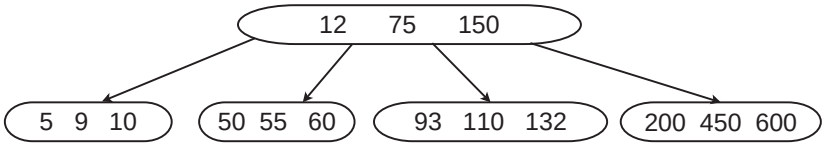


Figure 6.18. Example of extended balancing.

The three nodes will then be merged into two as shown:

$$\begin{array}{cc} & s_{33} \\ r_1, \ldots, r_{65}, a, s_1, \ldots, s_{32} & s_{34}, \ldots, s_{64}, b, t_1, \ldots, t_{65} \\ p_0, p_1, \ldots, p_{65}, q_0, q_1, \ldots, q_{32} & q_{33}, q_{34}, \ldots, q_{64}, y_0, y_1, \ldots, y_{65}. \end{array}$$

Note that the right node contains one less element than the maximum allowed.

There is a slight complication in this case, relating to nodes that are leftmost or rightmost children of their parent nodes: they do not have two immediate neighbors, like, e.g., the node $v_{5,7,9}$ in Figure 6.17. But for the merging to work properly, three nodes have to be handled together. The solution for this particular case is to allow *extended balancing*, not just with the immediate neighbor, but even with the neighbor's neighbor. Returning to Figure 6.17, if the element 7 is deleted, extended balancing would involve the three leftmost leaves and transform the tree into the one shown in Figure 6.18.

If even extended balancing cannot be performed, the three nodes that will be merged into two will be v, its neighbor and the neighbor's neighbor.

And what if there is no further neighbor to the neighbor? This can only happen if there is only a single value in the parent node of v; this parent must therefore be the root. The starting situation is thus

$$\begin{array}{cc} & a \\ r_1, \ldots, r_{65} & s_1, \ldots, s_{64} \\ p_0, p_1, \ldots, p_{65} & q_0, q_1, \ldots, q_{64}. \end{array}$$

and the nodes will be merged to form a new root with maximum load

$$\begin{array}{c} r_1, \ldots, r_{65}, a, s_1, \ldots, s_{64} \\ p_0, p_1, \ldots, p_{65}, q_0, q_1, \ldots, q_{64}. \end{array}$$

6.5.3 2-3-Trees

A special case of a B-tree that deserves being mentioned is for $m = 3$. Any node then contains one or two elements, and may have two or three children if it is not a leaf. Actually, 2-3-trees have been invented earlier than the more

general B-trees. The application area is also different, since B-trees are often used with large values of m, as mentioned previously. Therefore 2-3-trees can be seen as an alternative to AVL-trees for controlling the depth of a search tree with n elements, which will be bounded by $O(\log n)$.

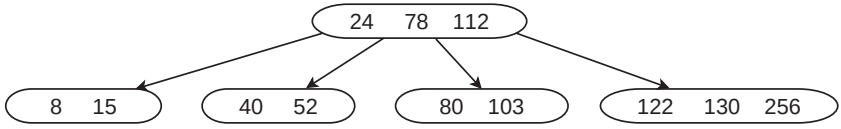
Exercises

6.1 Given is a B-tree of order 5 with four levels.

(a) A new element x is inserted, and immediately afterward, x is deleted. Is the B-tree after these two operations necessarily identical to that before the operations?
(b) What if the order of the operations is inverted: first, an element x is deleted, and then it is re-inserted?

6.2 Consider two B-trees A and B, both of order m. Denote the number of their elements by a and b, respectively, and assume $a \geq b$. Assume also that all the elements in A are smaller than any element in B. Show how to merge A and B in time $O(h)$ into a single B-tree of order m, where h is the height of A.

6.3 Given is the B-tree of order 5 in the following figure. Draw the tree after having removed the element 40 in the following cases:



(a) merging with the left neighbor;
(b) merging with the right neighbor;
(c) merging three elements into two;
(d) using extended balancing;
(e) assuming the order of the tree is 4.

6.4 Show how the B-tree of Exercise 6.3 changes if you insert, in order, the elements 104, 105, 106, 107, 108, using balancing, if necessary. Then insert 109, splitting the two rightmost leaves into three. Finally, insert 140, 150, and 158.

6.5 To increase the space exploitation of the nodes even further, one may define a B-tree variant in which every node, except the root, stores between $\frac{3m}{4} - 1$ and $m - 1$ elements.

(a) How should the number of elements in the root be defined?
(b) Show how to adapt the insertion and deletion procedures to comply with the new constraints.