B^+-trees

Kerttu Pollari-Malmi

This text is based partly on the course text book by Cormen and partly on the old lecture slides written by Matti Luukkainen and Matti Nykänen.

1 Introduction

At first, read the beginning of the chapter 18 from the text book. It explains why B-trees are used when the search structure is on disk.

In this course, we consider B^+-trees instead of classical B-trees introduced in the text book. The difference is that in B^+-trees only leaf nodes contain the actual key values. The non-leaf nodes of the B^+-trees contain *router values*. Routers are entities of the same type as the key values, but they are not the keys stored in the search structure. They are only used to guide the search in the tree. In classical B-trees, the key values are stored in both leaf and non-leaf nodes of the tree.

2 The structure of a B^+-tree

A B^+-tree T consists of nodes. One of the nodes is a special node $T.root$.

If a node x is a non-leaf node, it has the following fields:

- $x.n$ the number of router values currently stored in node x
- The router values stored in node x in increasing order $x.router_1 < x.router_2 < \ldots < x.router_{x.n}$
- $x.leaf$, a boolean field whose value is FALSE meaning that x is a non-leaf node
- $x.n+1$ pointers $x.c_1, x.c_2, x.c_3, \ldots, x.c_{x.n+1}$ to the children of x.

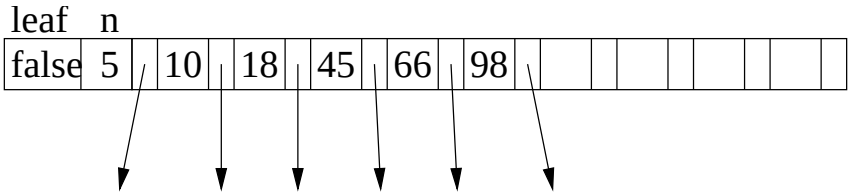
If the node x is a leaf node, it has the following fields

- $x.n$ number of key values currently stored in x.

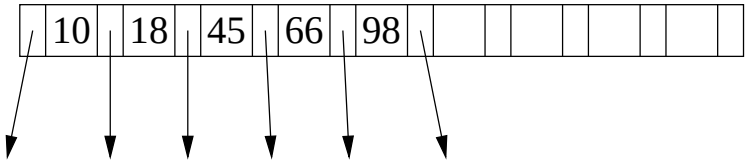
- The key values stored in node x in increasing order
  $x.key_1 < x.key_2 < \ldots < x.key_{x.n}$
- $x.leaf$, a boolean field whose value is TRUE meaning that x is a leaf node.

If we consider a non-leaf node x and pointers in it $x.c_1, x.c_2, x.c_3, \ldots, x.c_{x.n+1}$
the following condition is true:
$k_1 \leq x.router_1 < k_2 \leq x.router_2 < k_3 \ldots < k_{x.n} \leq x.router_{x.n} < k_{x.n+1}$
where k_i is any key or router value in the subtree pointed by the pointer $x.c_i$.

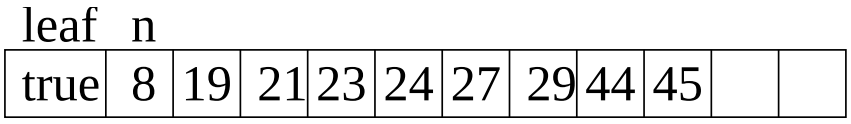
An example of the non-leaf node containing 5 router values:



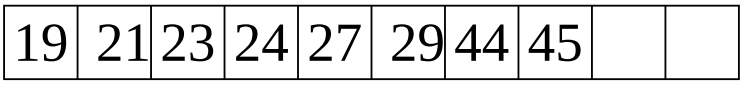
Usually, we do not draw the values of the fields $x.n$ and $x.leaf$:



The leaf nodes contain only the fields n and $leaf$ and key values (and maybe some data connected to the key values, but for the case of simplicity we do not draw that data)

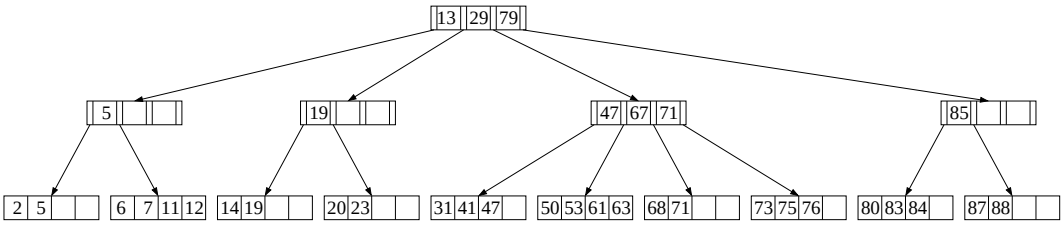


Usually, we draw the leaf node without the fields n and $leaf$:



Usually, each non-leaf node has dozens or hundreds of children.

An example of a B^+-tree:



3 Properties of the B^+-tree nodes

The B^+-tree has to satisfy the following balance conditions:

- Every path from the root node to a leaf node has an equal length, i.e. every leaf node has the same depth which is the height of the tree.
- Every node of the B^+-tree except the root node is at least half-filled.

The latter condition can be formulated more exactly: denote by $s(x)$ the number of children of node x if x is a non-leaf node and the number of keys stored in x if x is a leaf node. Let $t \geq 2$ be a fixed integer constant. For each node x of the B^+-tree, the following is true:

- $1 \leq s(x) \leq 2t$, if x is the only node in the tree.
- $2 \leq s(x) \leq 2t$, if x is the root node and the tree contains also other nodes in addition to the root node.
- $t \leq s(x) \leq 2t$, otherwise.

Because the B^+-tree satisfies the given balance conditions, we can prove that the hight h of the B^+-tree

$$h \leq \log_t n$$

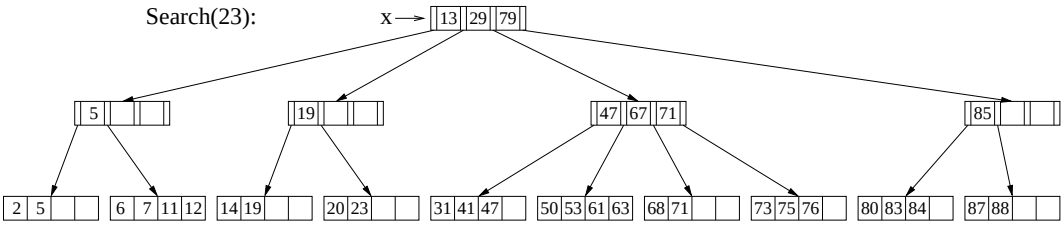
where n is the number of the keys stored in the tree.

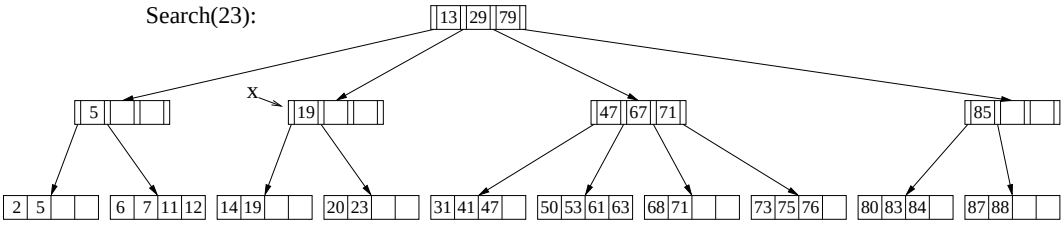
4 Search in B^+-tree

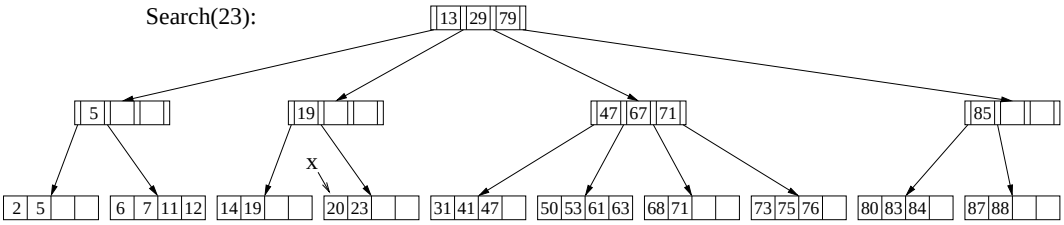
The search operation is started in the root node and it proceeds in every node x as follows:

- If x is a non-leaf node, we seek for the first router value $x.router_i$ which is greater than or equal to the key k searched for. After that the search continues in the node pointed by $x.c_i$
- If all router values in node x are smaller than the key k searched for, we continue in the node pointed by the last pointer in x.
- If x is a leaf node, we inspect whether x is stored in this node.

An example of a search operation:







4.1 Code of the search operation

```
BTreeSearch(T,k)
1   x = T.root
2   while not x.leaf
3       i = 1
4       while i ≤ x.n and k > x.router_i
5           i = i+1
6       x = x.c_i
7       DiskRead(x)
8   i = 1                                   // ollaan lehtisolmussa
9   while i ≤ x.n and k > x.key_i
10      i = i+1
11  if i ≤ x.n and k = x.key_i
12      return ( x, i )
13  else return NIL
```

4.2 Time complexity of the search operation

During the search operation, h nodes are read from the disk to the main memory where h is the height of the B$^+$-tree. As previously stated, the height of the B$^+$-tree is $h = O(\log n)$ where n is the number of the keys stored in the tree. In addition of the disk reads, the algorithm performs a linear search in every node read from the disk. The time complexity of each linear search is $O(t)$. Thus, the total time complexity of the B$^+$-tree search operation is $O(t \log_t n)$.

If the linear search inside each node is changed to a binary search, the total time complexity of the B^+-tree search operation becomes $O(\log_2 t \log_t n)$. However, in practise the disk read operations dominate the time demand of the operation.

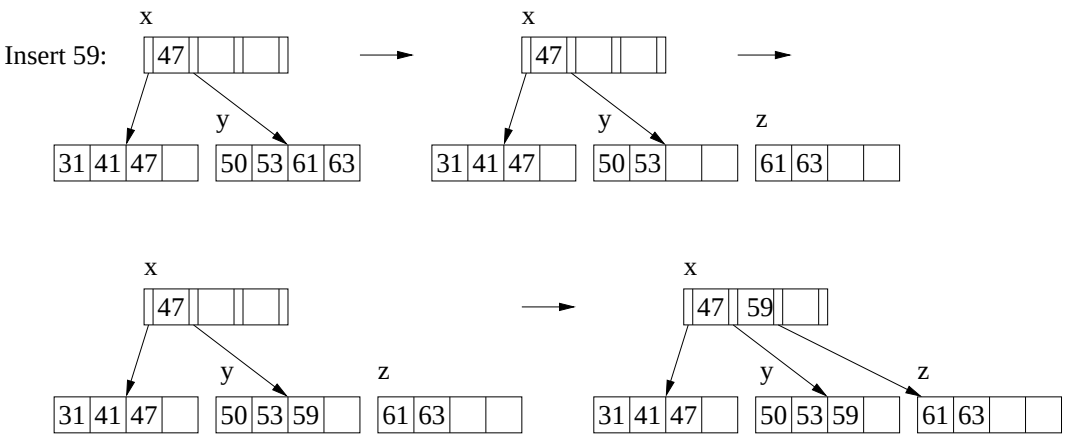
5 B^+-tree insertion

The insertion of the key k to a B^+-tree is started by searching for the leaf node y which should contain k. This is performed in the same way as when performing the B^+-tree-search operation. If there is enough space in y to insert the key k, k is inserted and no other actions are needed.

If the node y is full before the insertion, it must be split as follows:

- We allocate memory for a new node z.
- The first t keys of node y are left in y
- The last t keys of node y are transferred to the new node z.
- The key k is inserted to either node y or z according to the value of k.
- A pointer to the new node z and a new router value are inserted to node x which is the parent of both y and z. A good choice for the new router value is the last key value in y.
- If there is not enough space for a new pointer and router value in x, x must be split. In the worst case, all nodes in the path from the leaf to the root must be split.

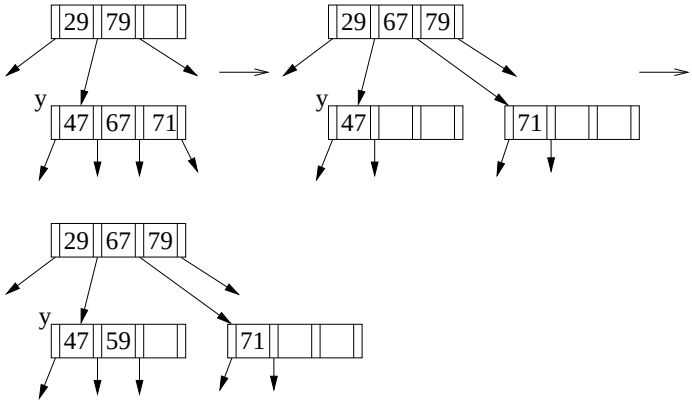
An example of splitting a leaf node:



In non-leaf nodes, the number of router values is one smaller than the number of pointers. Thus, if we split a non-leaf node, we have an “extra” router value. This router value is

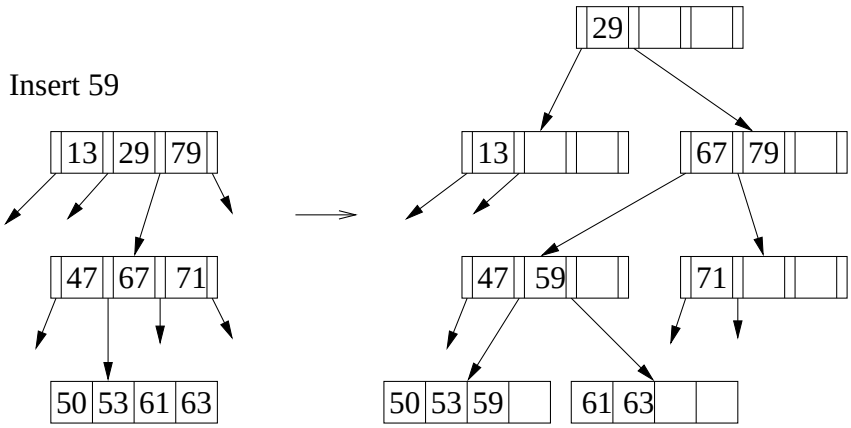
added to the parent of the split node between the pointers to the original node and to its new sibling.

An example of the situation where a new pointer and a new router value 59 are inserted into node *y* is presented below.



If the parent of the split non-leaf node is full, we must split the parent to be able to add a new pointer and router value to the parent. In the worst case, all nodes in the path from the split node to the root node have to be split.

An example of the situation where inserting the value 59 into the leaf node leads to splitting of three nodes. (Only part of the B^+-tree is presented in the picture below):



The time complexity of the insertion algorithm is $O(t \cdot log_t n)$, where the cofficient t is due to the operations performed for each node in the main memory. In practice, the time demand is dominated by the number of disk reads and writes needed, which is $O(log_t n)$.

We have presented the algorithm which performs the insertion to a leaf node at first and then goes back to the root node and does node splits when necessary. It is also possible to do the insertion in a single pass down the tree: if we encounter a full node, we split it on the

way from the root down to the leaf. This means that we maybe do some unnecessary split operations, but on the other hand, we guarantee that it is always possible to insert a key into a leaf node without going back to the root after insertion.

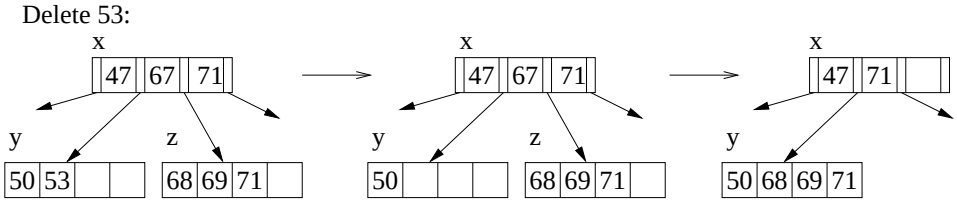
6 B^+-tree deletion

We start the deletion by searching for the leaf node containing the key to be deleted. Notice that we do not delete the same value from the non-leaf nodes although a non-leaf node would contain this value as a router value.

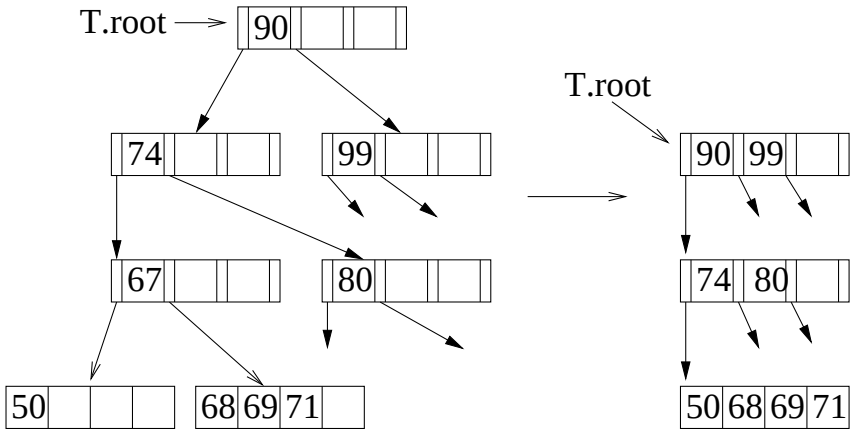
Because every leaf node contains more than one key values, we do not delete the node itself. We just remove the key value from the node.

If a leaf node contains at least t keys after the deletion, we are ready and no other actions are needed. If the number of keys in the leaf node is $t-1$ after deletion, rebalancing operation are needed.

When rebalancing node y which contains $t-1$ keys, we use the sibling z of y. The action performed depends on the number of nodes in z. If the node z contains at most $t+1$ keys, the keys in z are transferred to node y. After that, the pointer to node z and one router value are removed from the parent of y and z. This action is called *fusing*.

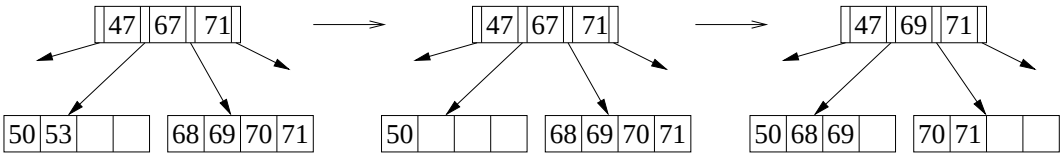


If the parent x has less than t children after fusing, we must continue the rebalancing process in x, which is a non-leaf node. If fusing is applied to two non-leaf nodes, the router value between the pointers to these nodes in the parent is removed from the parent and added to the fused node. In the worst case, all nodes in the path from the fused leaf to the root must be fused. The figure below shows the situation where fusing is started after one key has been removed from the leaf in the left. (The figure contains only part of the B^+-tree.)



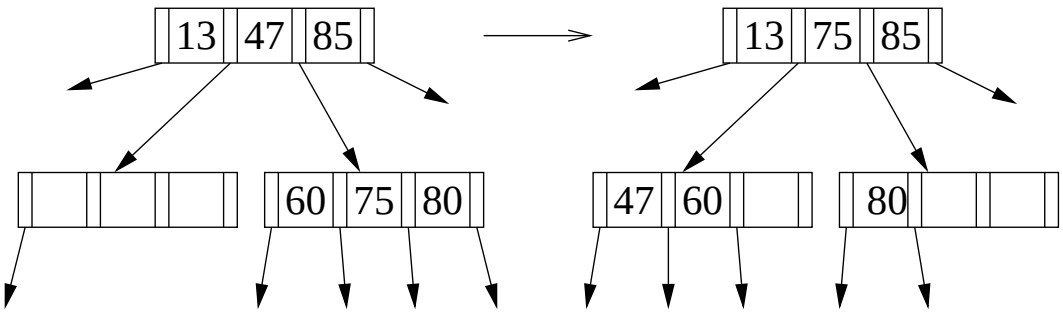
If the node y contains $t-1$ keys after the deletion and its sibling z contains at least $t+2$ keys, fusing cannot be applied. In this case, we use *sharing:* Some of the keys in z are transferred to the node y so that the number of keys in both nodes is about the same:

Delete 53:



Because the parent of y and z does not loose any children in this process, we do not need any rebalancing in the parent. However, we must update the router value between the pointers to the nodes y and z.

It is also possible that we must apply sharing to non-leaf nodes, if fusing nodes lower in the tree has led to the situation where a non-leaf node contains $t-1$ children and its sibling contains $t+2$ children. Notice how the router values are changed between the nodes to which sharing is applied and their parent in the example below:



The time complexity of the delete operation is the same as the time complexity of the insertion:

- The total time complexity is $O(t \cdot \log_t n)$
- The number of disk operations needed is $O(\log_t n)$. This is the dominating factor of the time demand in practice.

It is also possible to perform the delete operation and rebalancing in a single pass down the tree: Each time we encounter a node having exactly t children (non-leaf node) or keys (leaf node) on the way from the root to the leaf in the search phase, we apply sharing or fusing to this node and its sibling. When we reach the leaf, we can always remove at least one key from it without any further rebalancing operations.