COMP207 Lab Exercises
Tutorial 5 (Week 7)

The exercises below provide the opportunity to practice the concepts and methods discussed during previous week's lectures (lectures 13–15). If you haven't done so, it is worthwhile to spend some time on making yourself familiar with these concepts and methods. Don't worry if you cannot solve all the exercises during the lab session, but try to tackle at least one or two of them. If at some point you do not know how to proceed, you could review the relevant material from the lecture notes and return to the exercise later.

You could start with exercises where you anticipate you might need advise from the demonstrators or from other students in the lab. So if you're fine with query plans, you can skip to B+ trees, and if you think you've mastered B+ trees and know how to implement these, you could use the time to have a look at the textbooks or other resources on the web to study B+ trees or other variants of indexes such as hash indexes further. The references for the lectures on the Vital course page provide some entry points.

Solutions to most of the questions will be be provided after the last lab session of this week, so you could compare against these later. One of the few exceptions is the programming exercise at the end of this sheet, for which no solutions are provided.

Query Plans

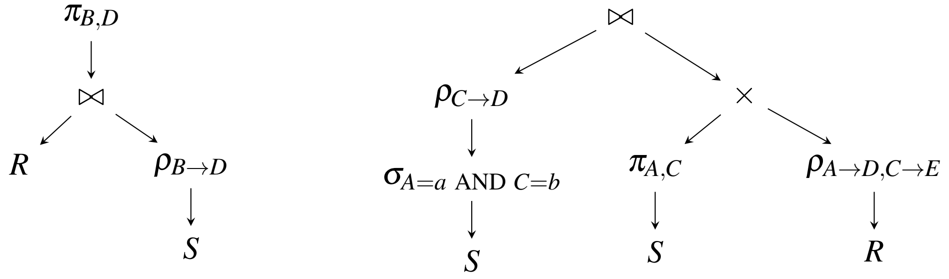
As you know from lectures 12 and 13, a DBMS takes SQL queries as input and translates these into *query plans* that are not yet optimised. Such plans can be represented as relational algebra expressions or as trees.

Exercise 1. Represent the following relational algebra expressions as query plans (in the form of a tree):

(a) $\pi_{B,D}(R \bowtie \rho_{B\to D}(S))$

(b) $\rho_{C\to D}(\sigma_{A=a \text{ AND } C=b}(S)) \bowtie (\pi_{A,C}(S) \times \rho_{A\to D,C\to E}(R))$

Solution: The query plans for (a) and (b) are (from left to right):



Let us now assume a database with the following two relations:

R

A	B	C
a	a	b
b	d	a

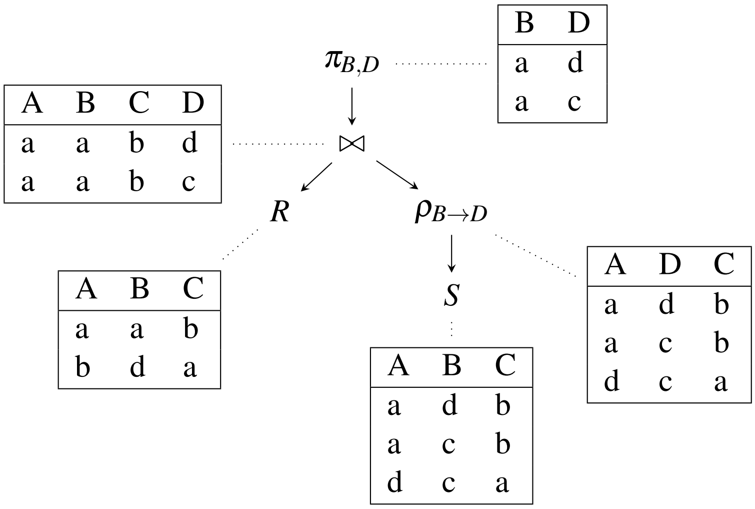
S

A	B	C
a	d	b
a	c	b
d	c	a

Exercise 2. Execute your query plans from Exercise 1 step by step on this database, as done in the examples and exercises in lecture 13.

Solution:

(a) Execution of the query plan from Exercise 1 (a):



B	D
a	d
a	c

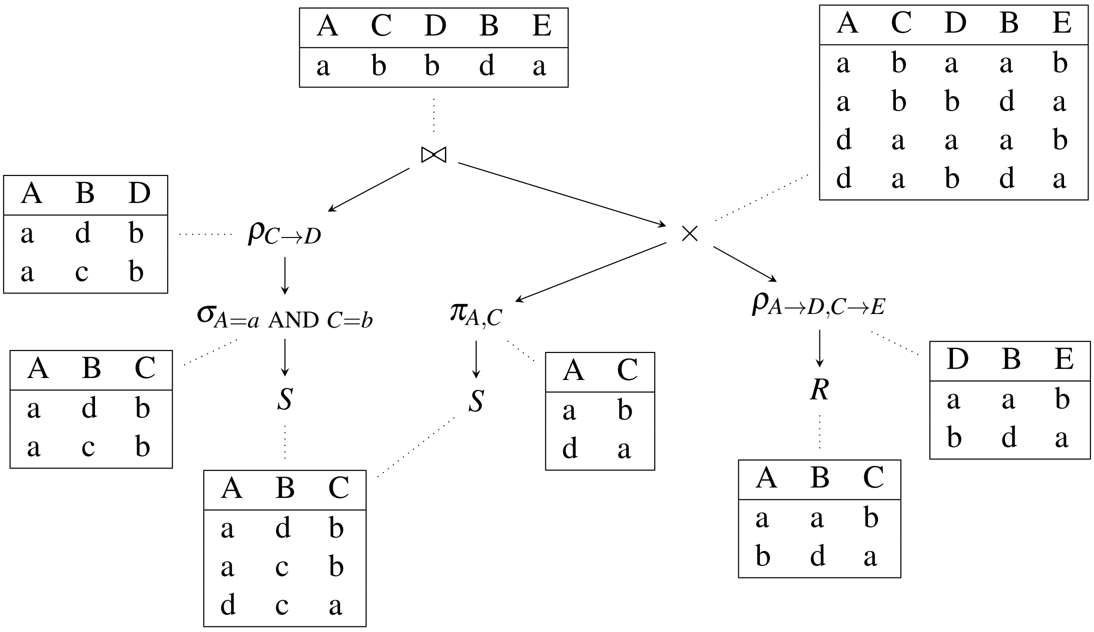
A	B	C	D
a	a	b	d
a	a	b	c

A	B	C
a	a	b
b	d	a

A	B	C
a	d	b
a	c	b
d	c	a

A	D	C
a	d	b
a	c	b
d	c	a

(b) Execution of the query plan from Exercise 1 (b):



A	C	D	B	E
a	b	b	d	a

A	C	D	B	E
a	b	a	a	b
a	b	b	d	a
d	a	a	a	b
d	a	b	d	a

A	B	D
a	d	b
a	c	b

A	B	C
a	d	b
a	c	b

A	B	C
a	d	b
a	c	b
d	c	a

A	C
a	b
d	a

D	B	E
a	a	b
b	d	a

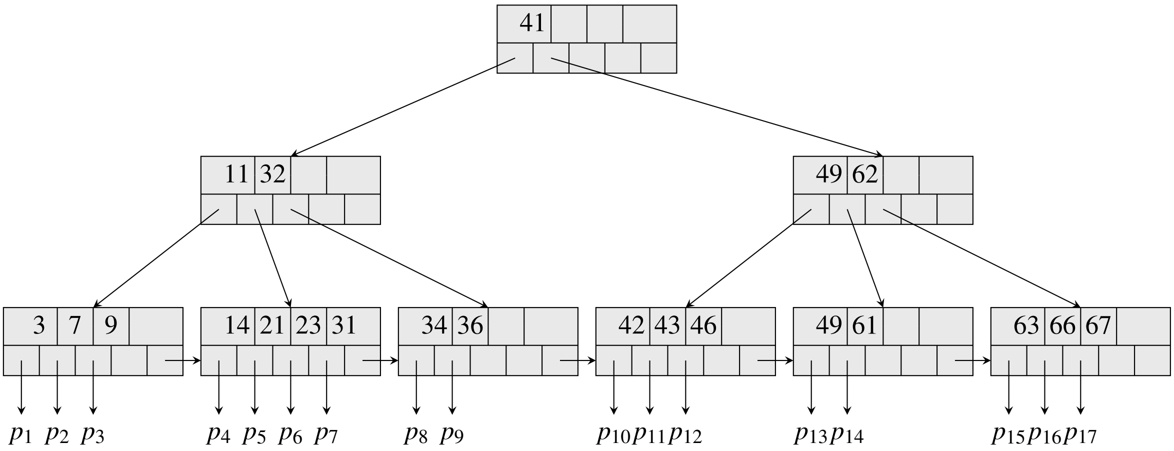
A	B	C
a	a	b
b	d	a

B+ Trees

In lecture 14 and 15 we saw that indexes can be realised using B+ trees. The following provides an opportunity for you to gain some more experience with B+ trees.

Looking Up Values

Consider the following B+ tree:



Exercise 3. Use the method from lecture 15 to look up the pointers associated with the following values in the B+ tree. Try to follow the method step by step.

(a) 61

(b) 62

(c) All values greater than or equal to 44

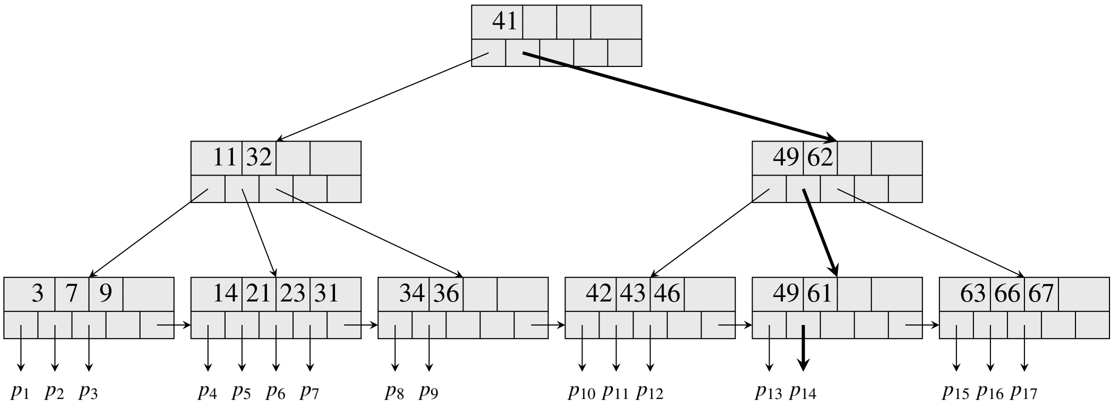
(d) All values in the range from 47 to 64 (i.e., all values x with $47 \leq x \leq 64$)

Solution:

(a) Procedure for looking up 61 in the B+ tree:

- Start at the root.
- Since $41 \leq 61$ and 41 is the last value at the root, proceed to the second child.
- Since $49 \leq 61 < 62$, proceed to the second child.
- Return the pointer associated with 61: p_{14}

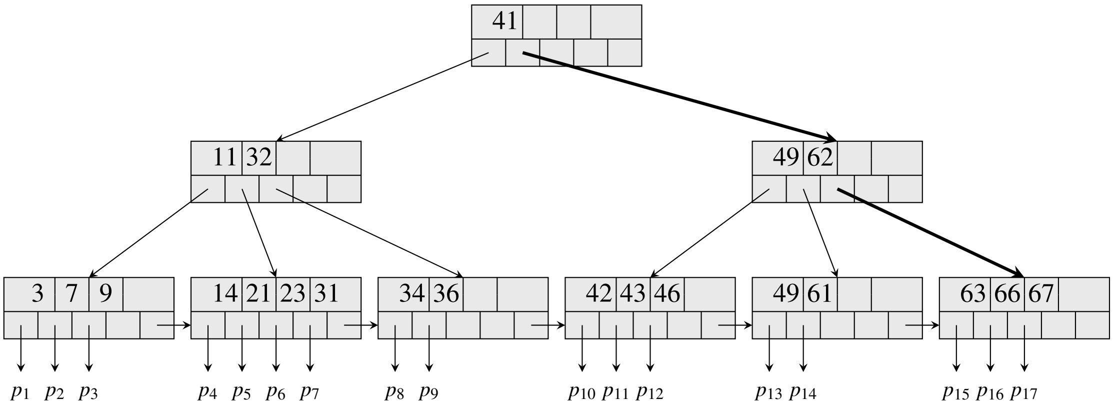
Here is a visual representation, with bold edges indicating the path taken through the tree to arrive at the pointer p_{14} associated with 61:



(b) Procedure for looking up 62 in the B+ tree:

- Start at the root.
- Since $41 \leq 62$ and 41 is the last value at the root, proceed to the second child of the root.
- Since $62 \leq 62$ and 62 is the last value at the current node, proceed to the third child of that node.
- The value 62 does not occur at the leaf, so return “not found”.

Again, here is a visual representation, with bold edges indicating the path taken through the tree to arrive at the leaf for 62:

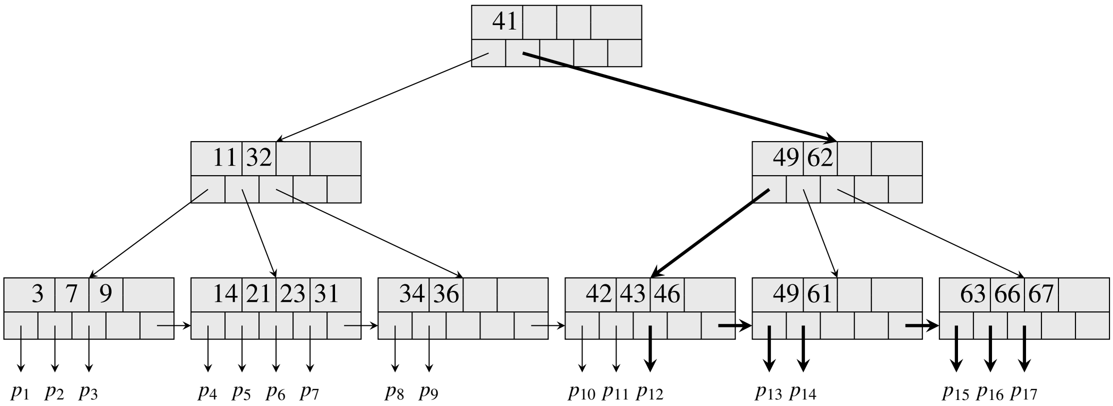


(c) Procedure for looking up all values greater than or equal to 44:

- Start at the root.
- Since $41 \leq 44$ and 41 is the last value at the root, proceed to the second child.
- Since $44 < 49$ and 49 is the first value at the current node, proceed to the first child.
- Return the pointers to all values of the leaf that are greater than or equal to 44. Thus, return p_{12}.

- Proceed to the leaf to the right and return the pointers to all values of that leaf. Thus, return p_{13} and p_{14}.
- Once again, proceed to the leaf to the right and return the pointers to all values of that leaf. Thus, return p_{15}, p_{16}, and p_{17}.

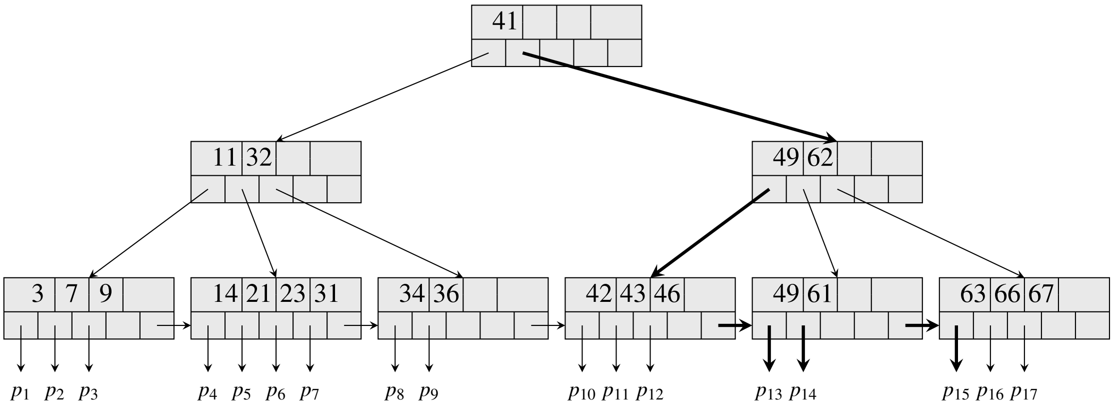
Here is a visual representation, with bold edges indicating all the edges followed to return the pointers associated with all the values that are greater than or equal to 44:



(d) Procedure for looking up all values in the range from 47 to 64:

- First find the leaf for 47. Start at the root.
- Since $41 < 47$ and 41 is the last value at the root, proceed to the second child.
- Since $47 \leq 49$ and 49 is the first value at the current node, proceed to the first child.
- Return the pointers to all values of the leaf in the range between 47 and 64. There are no such pointers, so proceed to the next leaf on the right.
- Return the pointers to all values of the current leaf that are less than or equal to 64. Thus, return p_{13} and p_{14}. Then, proceed to the next leaf on the right.
- Return the pointers to all values of the current leaf that are less than or equal to 64. Thus, return p_{15}.

Visual representation:

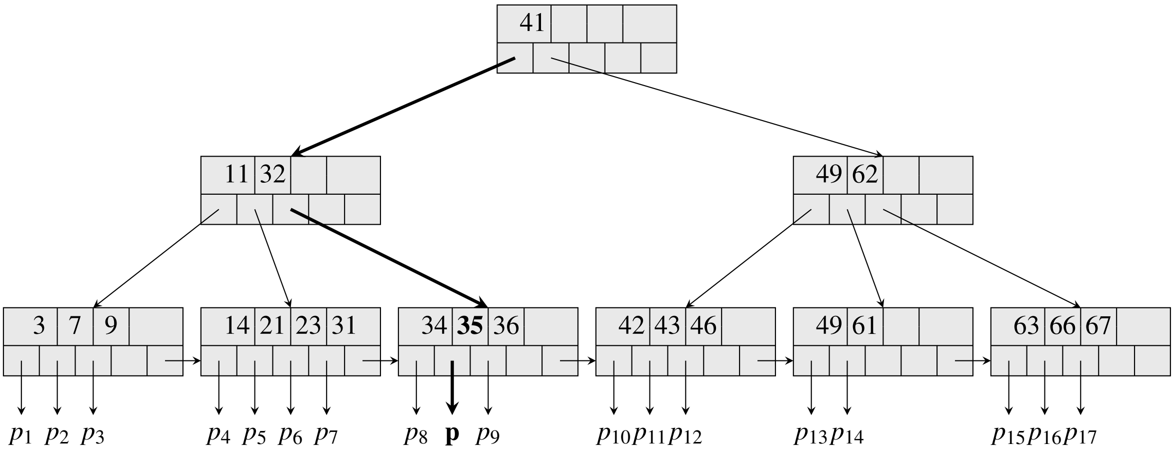


Insertion of Value/Pointer Pairs

We now turn to insertion of value/pointer pairs into B+ trees. As mentioned in lecture 15, to associate a pointer p with a value v in a B+ tree, we first find the leaf node that should contain the value v. If this leaf is not *full* (i.e., it contains at least one free slot), then we insert v and p into the leaf node so that the resulting sequence of values is sorted.

Example 1. Let us insert the value 35 with pointer p into the example B+ tree above. We first find the leaf that should contain 35: Start at the root node. Since $35 < 41$, we proceed to the left child of the root. Since $32 \leq 35$, we then proceed to the right child of that child, so we arrive at the third leaf from the left (the one containing the values 34 and 36).

We now have to insert 35 between 34 and 36, so that the values remain sorted increasingly. The new B+ tree looks as follows (the bold edges mark the path to 35 and the pointer inserted):



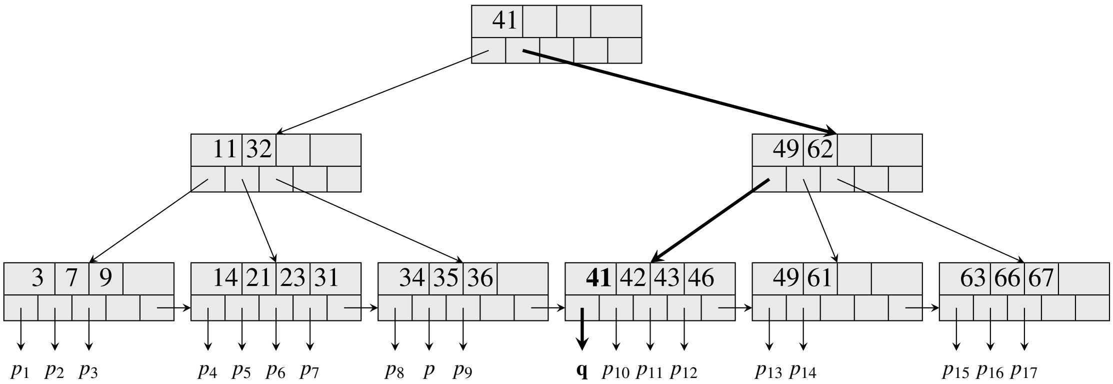
Exercise 4. Insert the following value/pointer pairs into the above B+ tree:

(a) Value: 41; pointer: q

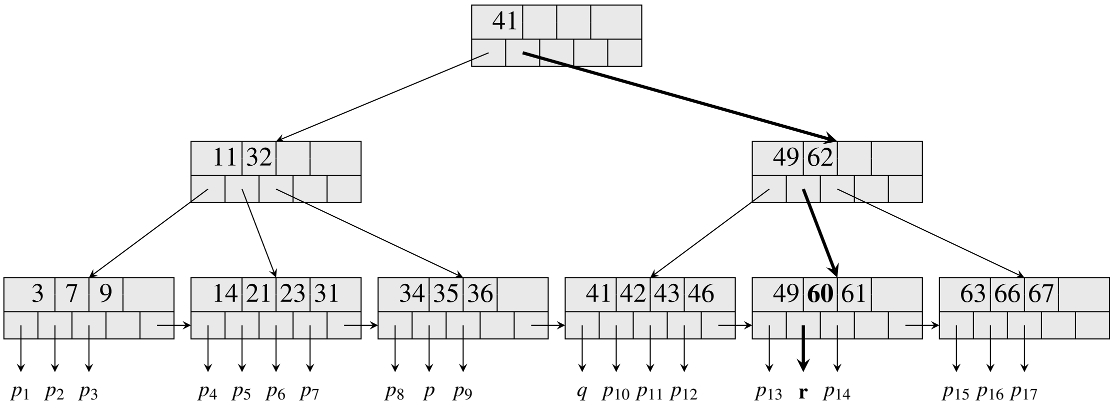
(b) Value: 60; pointer: r

Solution:

(a) We first navigate to the leaf associated with 41, which is the fourth leaf from left (the one containing 42, 43, and 46). At this leaf, we push all values and their pointers to the right and insert 41 and q into the first slot. This results in the following B+ tree (again, we mark the path to the leaf for 41 and to the pointer inserted):



(b) We navigate to the fifth leaf node from left (the one containing 49 and 61), and insert 60 and r between 49 and 61. The resulting B+ tree is:



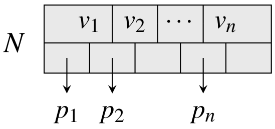
The interesting case

Now suppose that the leaf in which we want to insert the value v and its associated pointer p is full. In lecture 15, we have seen that in this case we have to split the leaf, and this might lead to splitting other nodes higher up in the tree. We now take a closer look at how to split a node (see also the references for lecture 15 on the Vital course page for detailed information).

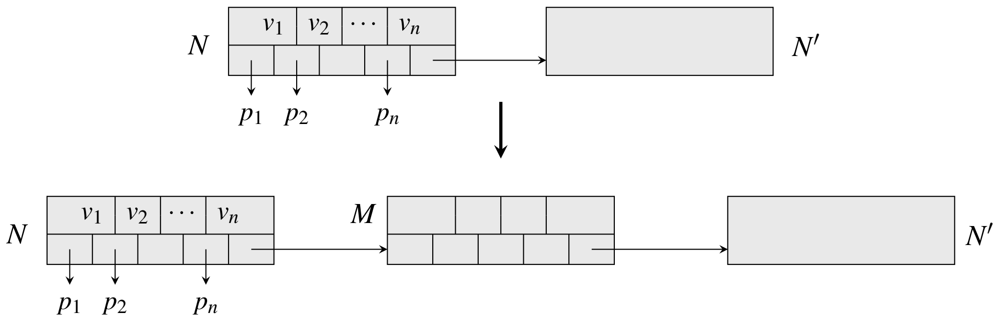
Let N be a node in the B+ tree, not necessarily a leaf, in which we want to insert a new value v and an associated pointer p. If N is a leaf, then p is a pointer to a tuple in a relation R containing the value v (or to a list containing tuples with this value), and if N is not a leaf, then p is a pointer to a different B+ tree node, which we want to add as a child of N. In any case, we need to split N in order to make room for the new value/pointer pair. We do this as follows:

If N is a leaf:

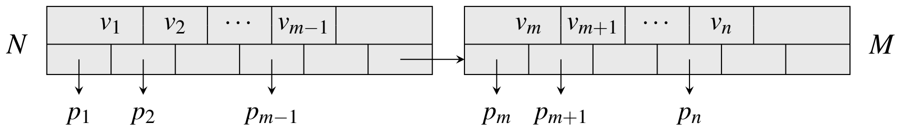
- Let $v_1, v_2, \dots, v_n$ be the values that are currently stored in N, in the order in which they occur in the node, and let p_i be the pointer associated with v_i:



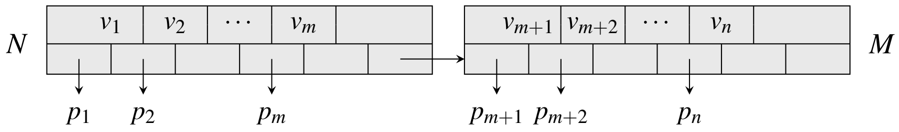
- Create a new node M and make it the new right neighbour of N. If N' was the right neighbour of N before inserting M, make N' the right neighbour of M:



- Move the "upper half" of the values and their associated pointers from N to M. What counts as "upper half" depends on where in the sequence $v_1, \ldots, v_n$ we have to insert the value v. Let $m := \lfloor \frac{n+1}{2} \rfloor$.
  - If $v < v_m$, then we keep the first $m-1$ values and pointers in N, and move the remaining values $v_m, \ldots, v_n$ and pointers $p_m, \ldots, p_n$ to M:

    Afterwards, we insert v and p into N.
  - If $v > v_m$, then we keep the first m values and pointers in N, and move the remaining values $v_{m+1}, \ldots, v_n$ and pointers $p_{m+1}, \ldots, p_n$ to M:

    Afterwards, we insert v and p into M.

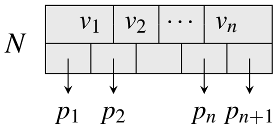
  Note that both N and M contain at least $\lfloor \frac{n+1}{2} \rfloor$ pointers (not counting the pointers leading to the right neighbours), and therefore satisfy the requirements for B+ tree leaves.

- If N is a child of a node P, make M a child of P. To this end, let w be the minimum of all the values that occur in any leaf that is reachable from M. Then, insert the value w together with a pointer to M into P.[1] If N is not a child of any node, then add a new node P first, and then make N and M children of P (P only contains the value w and the two pointers to N and M).

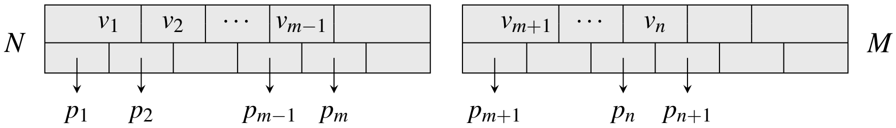
[1] If P is not full, then we insert w and the pointer to M in a similar way as we would do when we insert a value/pointer pair into a leaf. If P is full, then first we split P using the procedure just described.

If N is not a leaf:

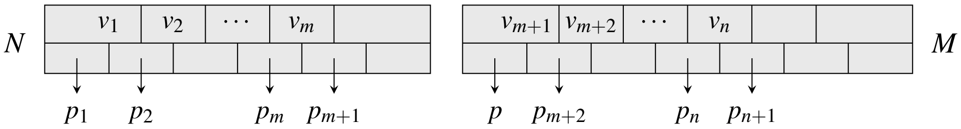
- Let $v_1, v_2, \ldots, v_n$ be the values that are currently stored in N, in the order in which they occur in the node, and let $p_1, \ldots, p_{n+1}$ be the pointers to the child nodes:



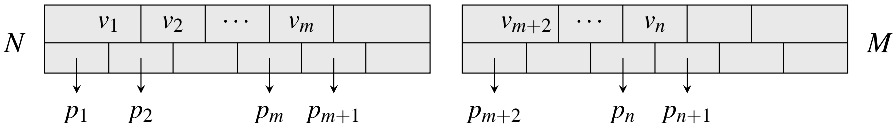
- Create a new node M, and move the "upper half" of the values and pointers from N to M. The procedure slightly differs from that for leaves. Let $m := \lfloor \frac{n+1}{2} \rfloor$.
  - If $v < v_m$, then we keep the first $m-1$ values and the first m pointers in N and move $v_{m+1}, \ldots, v_n$ and $p_{m+1}, \ldots, p_{n+1}$ to M as follows:[2]



    Afterwards, we insert v and p into N.
  - If $v_m < v < v_{m+1}$, then we keep the first m values and the first $m+1$ pointers in N, move $v_{m+1}, \ldots, v_n$ and $p_{m+2}, \ldots, p_{n+1}$ to M, and insert p into M as follows:[3]



  - If $v_{m+1} < v$, then we keep the first m values and the first $m+1$ pointers in N and move $v_{m+2}, \ldots, v_n$ and $p_{m+2}, \ldots, p_{n+1}$ to M as follows:[4]



    Afterwards, we insert v and p into M.

  It is easy to verify that both N and M contain at least $\lceil \frac{n+1}{2} \rceil$ pointers, and therefore satisfy the requirements for B+ tree nodes.

- The final step is to connect M to the parent of N, or to create a new root that has both N and M as children. This is done as in the case of leafs.

Exercise 5 (no solution, but see lecture notes)**.** Using the above procedure, try to understand the process of inserting the value 42 (say, with a pointer p) into the B+ tree from lecture 15.

[2] Note that we drop v_m.

[3] Note that we do not insert v.

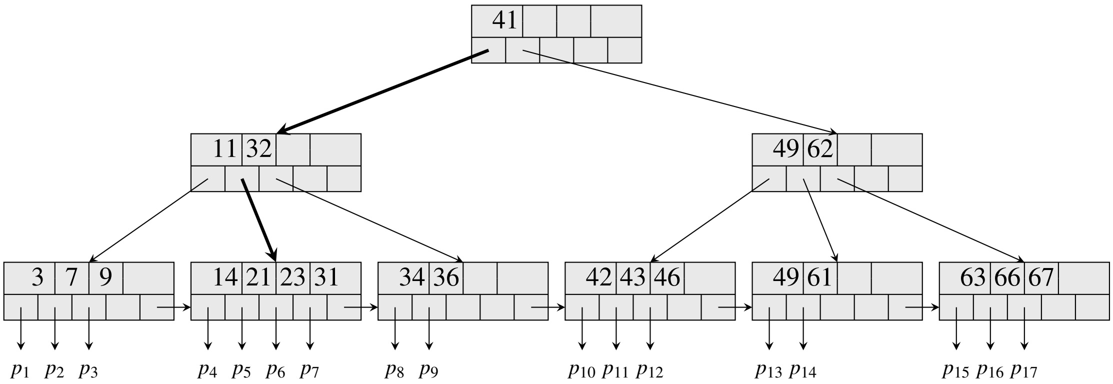
[4] Note that we drop v_{m+1}.

Exercise 6. Insert the following value/pointer pairs into the (unmodified) B+ tree from the beginning of this section:

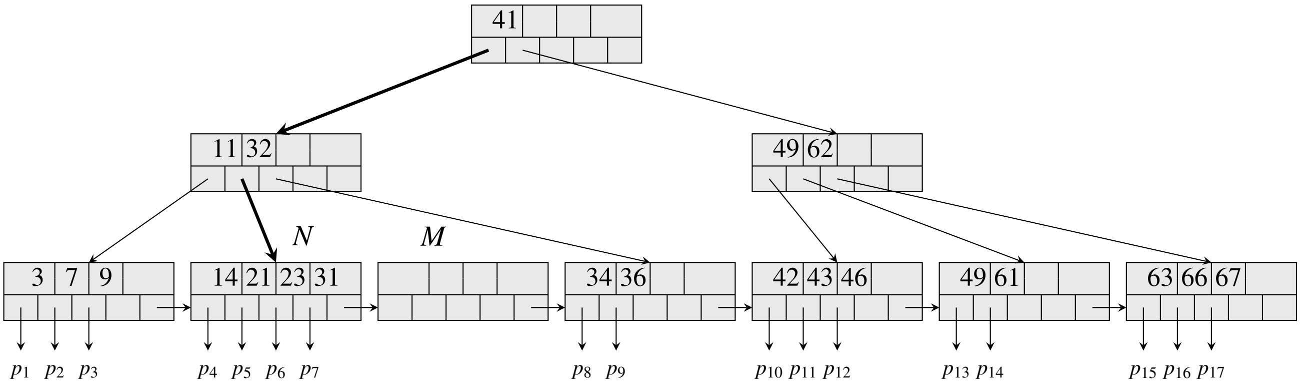
(a) Value 22; pointer p

(b) Value 17; pointer p (no solution provided)

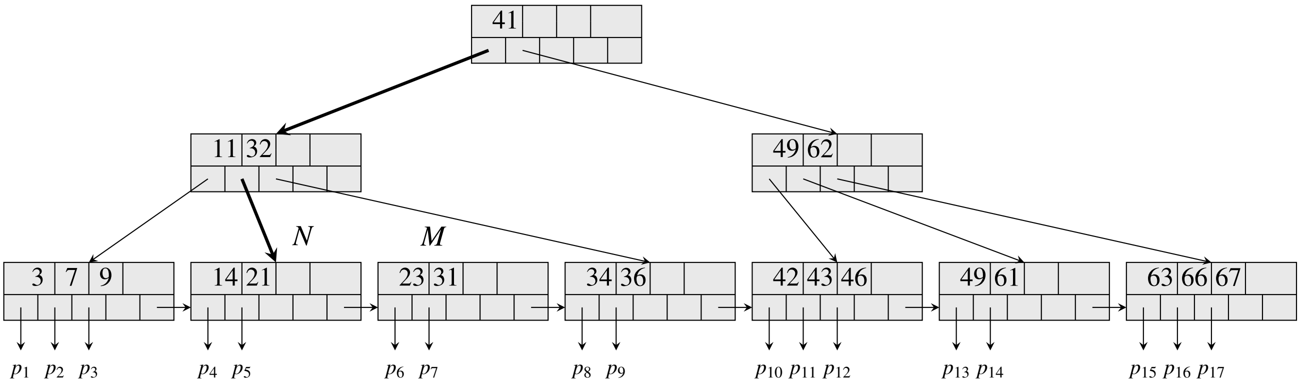
Solution for (a): Here is what happens when we insert 22 with pointer p into the B+ tree. We first navigate to the appropriate leaf:



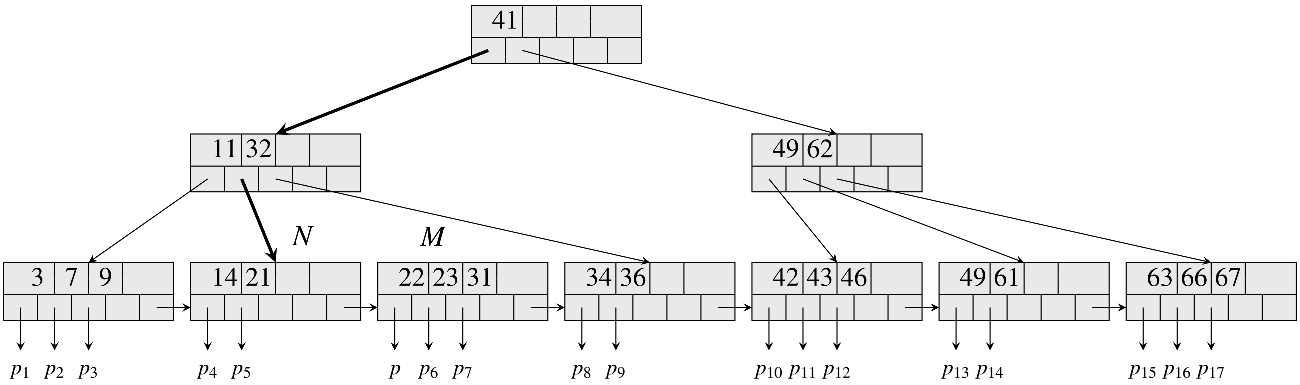
Let us denote this leaf by N. Since N is full (i.e., it does not have any room to store 22), we have to split it. We first create a new leaf M and insert it between N and its right neighbour:



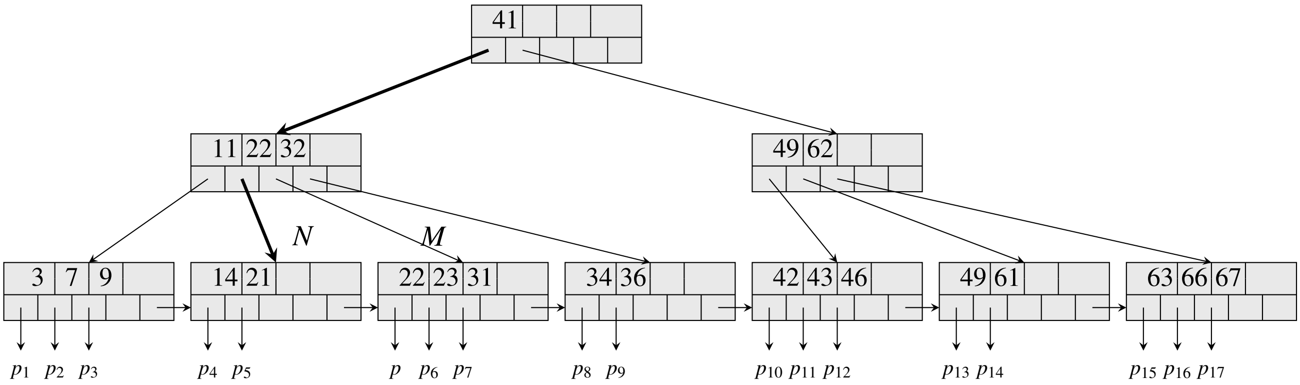
The next step is to move a sufficient number of values and pointers from N to M. In our case, we have $n = 4$, so in order to decide how many values and pointers we have to move we compare 22 with the value 21 at position $m = \lfloor \frac{n+1}{2} \rfloor = 2$. Since $22 > 21$, we keep 14, 21 and their pointers in N and only move 23, 31 and their pointers to M:



We now insert 22 and p into M:[5]



It remains to add M as a child to the parent P of N. Note that M is the only leaf that can be reached from M, so the minimum value w stored in any leaf reachable from M is 22. In order to add M as a child to P we therefore insert the value 22 with a pointer to M into P. Since P is not full, we insert 22 and the pointer between 11 and 32. The final B+ tree is:



[5] This is arbitrary. We could have inserted 22 and p into N without destroying the property of B+ tree nodes, but here we follow the procedure given above.

Implementing B+ Trees

Whenever you can, you should try to implement data structures and algorithms yourself. This is a very good way of getting a deep understanding of these data structures and algorithms, even of the finer details. In order for this to be effective, you should try out tutorials on the web only if you do not know how to start, but then try to look up only as many details as you need in order to progress.

Exercise 7 (no solutions provided)**.** Implement the B+ tree data structure in a programming language of your choice. Then try to experiment with your implementation by building a B+ tree for a large collection of values, by looking up values, by performing range searches, by inserting values, etc.