Please follow **the most efficient** strategy to insert/delete records.

1 Hash Tables

Consider linear-probe hashing. Describe with an example why not using tombstones will incur in problems. In which operation?

Solution. *Consider 3 values v1, v2, v3, v4, with hash(v1)=3, hash(v2)=4, hash(v3)=4, hash(v4)=5. If we do not use tombstones we can consider the following sequence of operations:*

1. *PUT v1*
2. *PUT v2*
3. *PUT v3*
4. *DELETE v2*
5. *GET v3*

The last get will fail to find the value.

2 Indexes

1. Indexes speed up query processing but it is usually a bad idea to create indexes on every attribute and every combination of attributes that might potentially be useful for an arbitrary query. Explain why?

   Solution.

   - *Every index requires additional CPU time and disk I/O overhead during inserts and deletions.*
   - *Indexes on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).*
   - *Each extra index requires additional storage space.*
   - *For queries that involve conditions on several search keys, efficiency might not be bad even if indexes are available for only some of them.*

2. Is it possible in general to have two clustering indexes on the same relation for different search keys? Explain your answer.

Solution.

In general, it is not possible to have two clustering indexes on the same relation for different keys because the tuples in a relation would have to be stored in different order to have similar values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

3 B$^+$-tree

1. Construct a B$^+$-tree for the following set of key values:

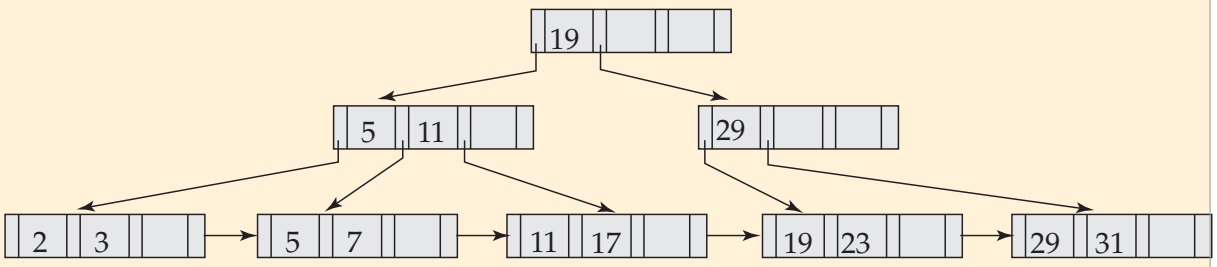
$$(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)$$

Assume that the tree is initially empty and values are added one-by-one in ascending order. Construct B$^+$-trees for the cases where the number of pointers that will fit in one node is as follows:

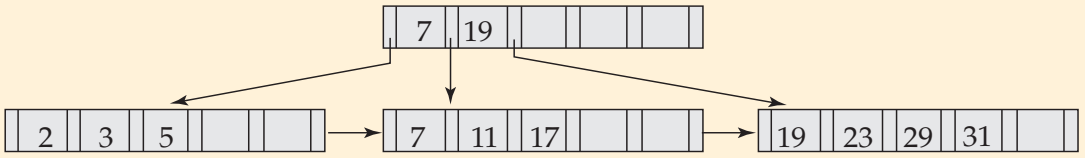
(a) 4
(b) 6
(c) 8

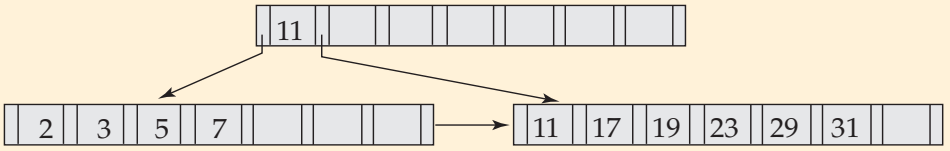
Solution. *The following were generated by inserting values into the B+- tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil \frac{n}{2} \rceil$ values/pointers.*

b.



c.

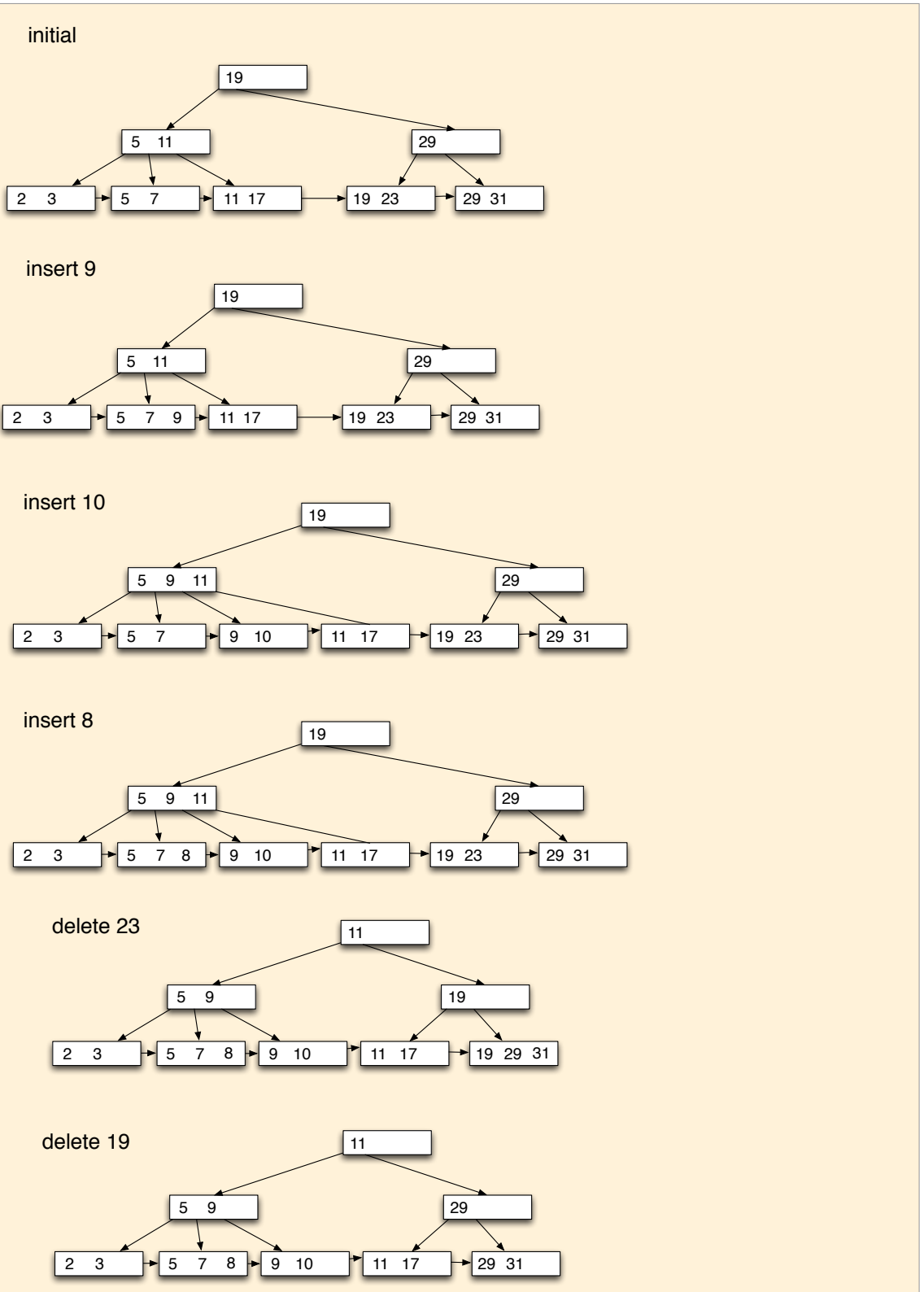




2. For each B$^+$-tree that you have constructed, show the form of the tree after each of the following series of operations (each executed on the result of the previous):

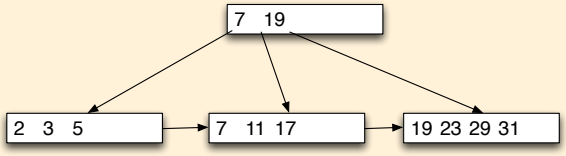
   (a) Insert 9
   (b) Insert 10
   (c) Insert 8
   (d) Delete 23
   (e) Delete 19

Solution.
For $n = 4$

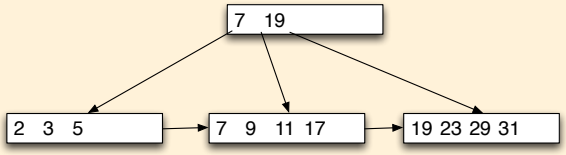

initial
19
5 11
29
2 3
5 7
11 17
19 23
29 31
insert 9
19
5 11
29
2 3
5 7 9
11 17
19 23
29 31
insert 10
19
5 9 11
29
2 3
5 7
9 10
11 17
19 23
29 31
insert 8
19
5 9 11
29
2 3
5 7 8
9 10
11 17
19 23
29 31
delete 23
11
5 9
19
2 3
5 7 8
9 10
11 17
19 29 31
delete 19
11
5 9
29
2 3
5 7 8
9 10
11 17
29 31

For $n = 6$

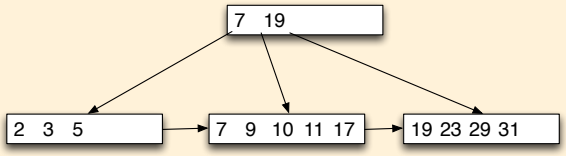
initial



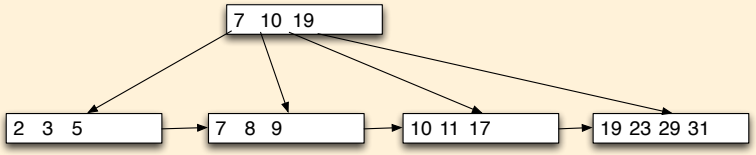
insert 9



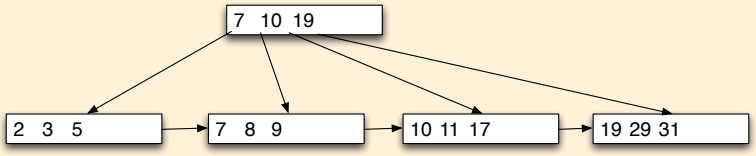
insert 10



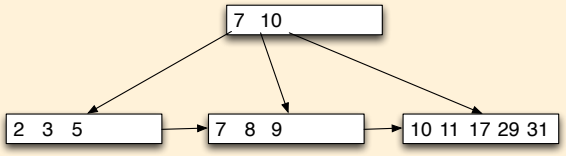
insert 8



delete 23

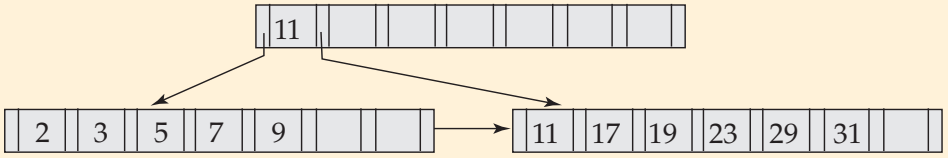


delete 19

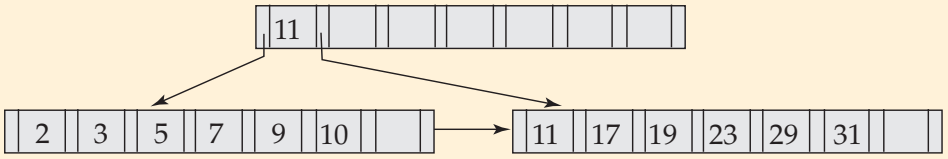


For $n = 8$

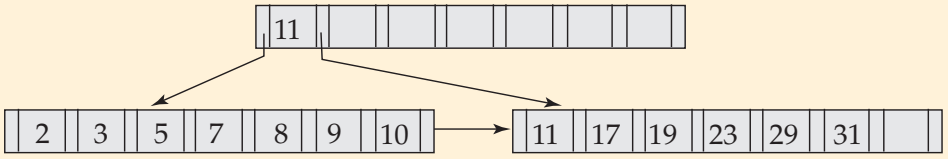
Insert 9:



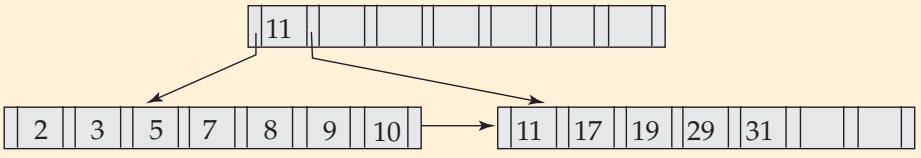
Insert 10:



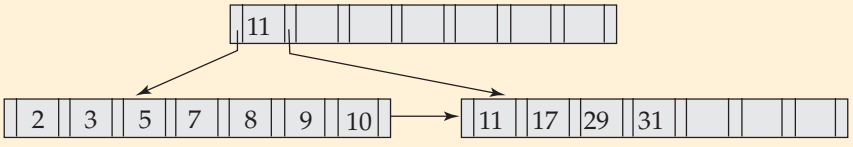
Insert 8:



Delete 23:



Delete 19:



4 Useful index

Consider the following relational schema, which obviously does not describe the standard situation at Aalborg University.

We assume that tutors are responsible for one or multiple study groups, students individually (not per group) hand in solutions for exercise sheets and receive individual grades in terms of the number of achieved points per sheet. Some of the tutors are more experienced (senior) than others.

student: {[sid: int, firstname: string, lastname: string, semester: int, birthdate: date]}

tutor: {[tid: int, firstname: string, lastname: string, issenior: boolean]}

studygroup: {[gid: int, tid → tutor, weekday: string, room: string, starttime: time]}

exercisesheet: {[eid: int, maxpoints: int]}

handsin: {[sid → student, eid → exercisesheet, achievedpoints: int]}

member: {[sid → student, gid → studygroup]}

For the following queries, discuss which index(es) would be useful?

Please describe all the characteristics of the index (sparse or dense? primary or secondary? hashing? multi-level?) as well as the attribute(s) used as search key in the index(es). Include any assumptions that are relevant to justify your choice.

1.

```
SELECT s.sid, s.lastname
FROM student s
    JOIN member n ON n.sid = s.sid
    JOIN studygroup g ON g.gid = n.gid
    JOIN tutor t ON t.tid = g.tid
WHERE t.issenior = false
    AND s.semester < 4;
```

Solution.

- *Indexes on s.sid, (n.sid, n.gid), g.gid, t.tid would be useful*
- *These indexes are indexes that use the primary keys as search keys*
- *B+-trees indexes would be a good choice. These indexes are multi-level indexes with sparse indexes on all levels except by the leaf level that is a dense index*

2.


```sql
SELECT w.eid, s.sid, s.lastname
FROM student s
    JOIN handsin w ON w.sid = s.sid
    JOIN exercisesheet b ON b.eid = w.eid
WHERE w.achievedpoints = b.maxpoints;
```

Solution.

- *Indexes on s.sid, (w.sid, w.eid), b.eid would be useful*
- *These indexes are indexes that use the primary keys as search keys*
- *B+-trees indexes would be a good choice. These indexes are multi-level indexes with sparse indexes on all levels except by the leaf level that is a dense index*

3.

```sql
SELECT e.eid, a
FROM exercisesheet e LEFT OUTER JOIN (
    SELECT eid, AVG(achievedpoints) AS a
    FROM handsin
    GROUP BY eid
  ) AS averages
  ON e.eid = averages.eid;
```

Solution.

- *Indexes on handsin.eid, exercisesheet.eid would be useful*
- *These indexes are indexes that use the primary keys as search keys*
- *B+-trees indexes would be a good choice. These indexes are multi-level indexes with sparse indexes on all levels except by the leaf level that is a dense index*
- *The index on e.eid provides enough information from the table exercisesheet and therefore there is no need to access the records*
- *The index on handsin.eid could allow for having a good implementation of the LEFT OUTER JOIN, but most of the records from the table handsin need to be accessed (handsin.eid references e.eid)*