Indexing

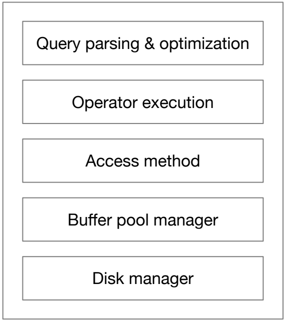
April 21, 2023

Announcements

- Assignment (IV) has been released. DDL: May 4, 2023.
- Sample solution to Assignment (III) has been posted on Canvas.

DBMS: Access method

Purpose: Support DBMS's execution engine to read/write data from pages more efficiently.


Query parsing & optimization

Operator execution

Access method

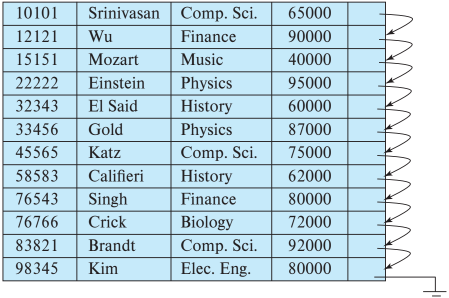
Buffer pool manager

Disk manager

Figure: DBMS architecture

Indexing basics

Example



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- Table instructor uses sequential file organization based on search key ID.
  - Records are ordered according to the attribute ID.
- Total number of pages of table instructor: 1,000 pages.
- Estimate the number of I/O's (#pages to read from disk) for query
  `SELECT * FROM instructor WHERE ID = '22222';`

Index data structure

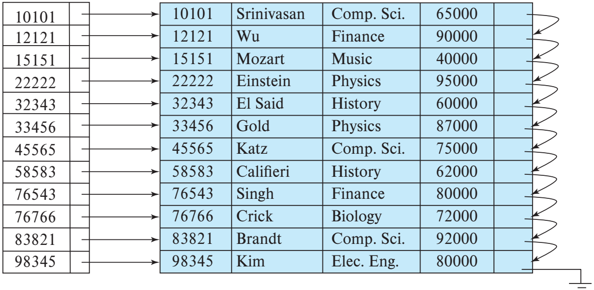
- Search key: an attribute or a set of attributes used to look up records in a file.
- An index file consists of records (called index entries) of the form

search key	pointer

- An index files is usually much smaller than the original file.
- We will only consider ordered indexes in this lecture.
    - Ordered indexes: search keys are organized in sorted order.
    - Hash indexes: search keys are distributed uniformly across buckets via a has function.

Dense indexes

- One index entry for each search key value.



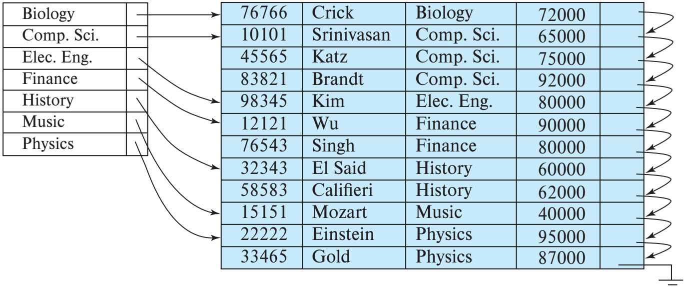
10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Figure: Dense index on attribute ID of table instructor

Dense indexes

- One index entry for each search key value.
- One index entry may point to multiple records.



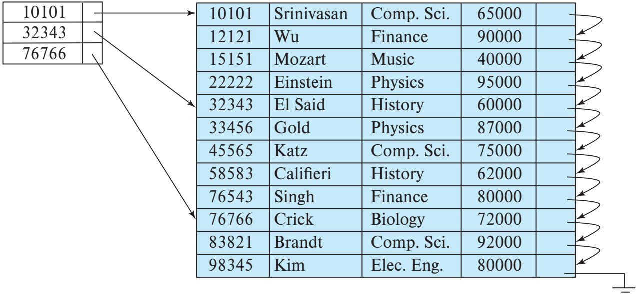
Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Figure: Dense index on attribute dept_name of table instructor

Spare indexes

- Index entries for only some search key values.
  – Typically one index entry for each block.
- Applicable only when records are ordered by the search key. Why?

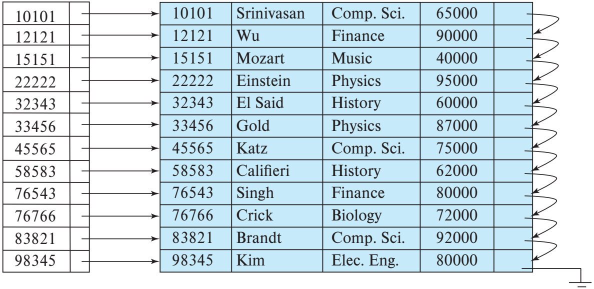


10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure: Sparse index on attribute ID of table instructor

Clustering indexes

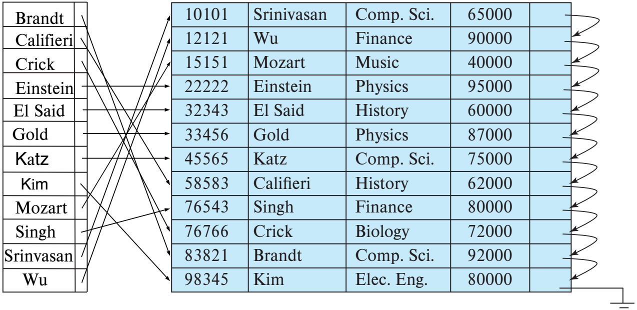


10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- Recall that index entries are sorted on the search key in an ordered index.
- Clustering index: search key order also defines the sequential order of data records.
- A clustering index is also known as a primary index.

Non-clustering index



Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinvasan
Wu

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- Non-clustering index: search key order differs from the sequential order of data records.
- A non-clustering index is also know as a secondary index.
- Secondary index is always dense. Why?

B^+-tree

B$^+$-tree

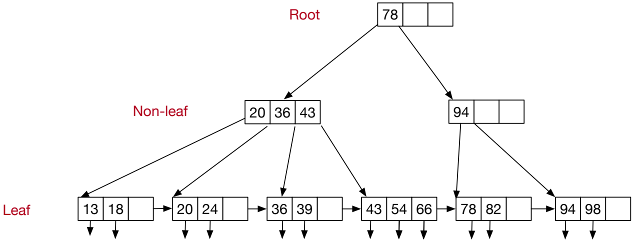
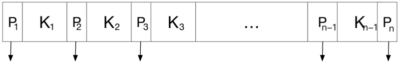


Figure: A sample B$^+$-tree with max_fanout$= 4$

A B$^+$-tree in a self-balancing search tree with following properties.

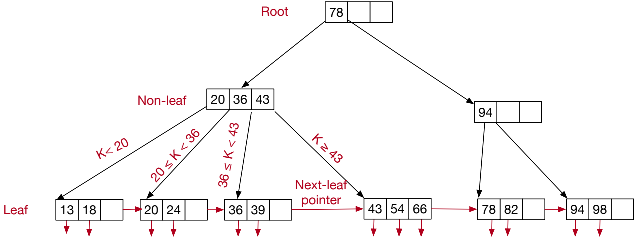
- Perfectly balanced: search, insertions, and deletions in logarithmic time.
- Optimized for disk-based DBMS: one node per block; large fan-out.

B^+-tree node



- Each B^+-tree node contains at most $n - 1$ search keys and n pointers.
  - n is referred to as the max_fanout parameter.
- Search keys are arranged in sorted order: $K_1 < K_2 < \cdots < K_m < \ldots$
- Every active pointer P_i points to a node in the next level.
- In practice, n can be hundreds, i.e., large fanout.

B$^+$-tree node



- P_i points the sub-tree of search keys K with $K_{i-1} \leqslant K < K_i$.
- Leaf nodes are chained up by the last pointer P_n, i.e., next-leaf pointer.
- Other active pointers P_i in leaf nodes point to the data page corresponding to key K_i.
- Index entries to data pages are stored in leaf nodes only.

B^+-tree invariant

	Min #(Active pointers)	Min #(Keys)
Root	2	1
Internal node	$\lceil n/2 \rceil$	$\lceil n/2 \rceil - 1$
Leaf node	$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor$

Table: Half-full constraint for B^+-trees with max fanout 4

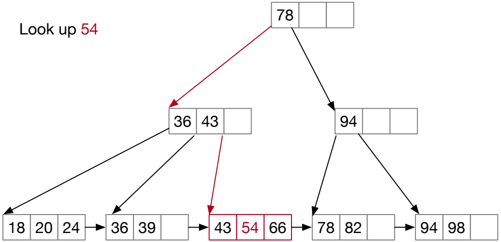
- Balance invariant: all leaves are at the same level.
- Occupancy invariant: all nodes (except root) are at least half-full.

Claim. The height of a B^+-tree with N search keys is at most $\lceil \log_{\lceil n/2 \rceil} N \rceil$.

B^+-tree in practice

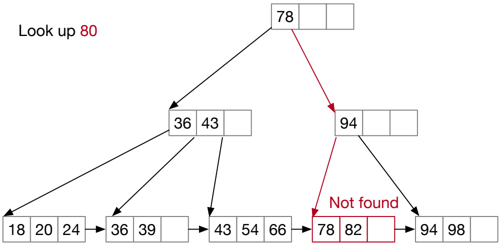
- $N = 1,000,000$.
- Page size: 4k bytes, index entry size 40 bytes.
- n = 100.
- $\lceil \log_{\lceil n/2 \rceil} N \rceil = 4$. That is, at most 4 I/O's for every lookup.
- If we cache the root node in buffer pool, then at most 3 I/O's are needed.

Query (1)



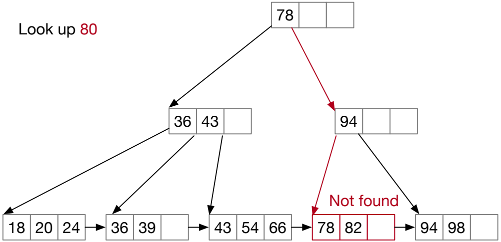
- `SELECT * FROM R WHERE K=54;`

Query (1)


Look up 80

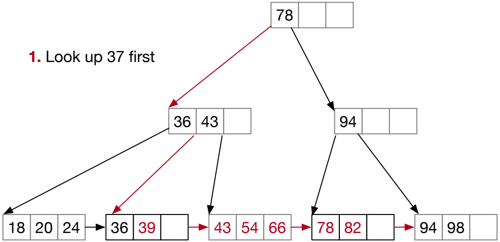
- `SELECT * FROM R WHERE K=54;`
- `SELECT * FROM R WHERE K=80;`

Query (1)



- `SELECT * FROM R WHERE K=54;`
- `SELECT * FROM R WHERE K=80;`
- This type of query is known as point query.

Query (2)



1. Look up 37 first

2. Follow the next leaf pointer until hit the upper bound

- `SELECT * FROM R WHERE k >= 37 AND K <= 90;`
- This type of query is known as range query.

Insertion (1)

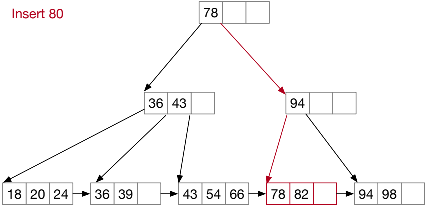


Figure: Insert key 82 ($n = 4$)

- Locate the leaf node for the key to be inserted.
- Insert the key directly when the target node has enough space.

Insertion (1)

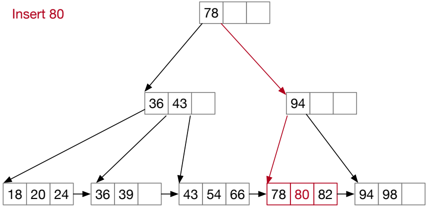


Figure: Insert key 82 ($n = 4$)

- Locate the leaf node for the key to be inserted.
- Insert the key directly when the target node has enough space.

Insertion (2)

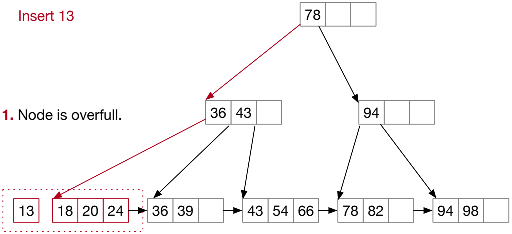


Figure: Insert key 13 ($n = 4$)

- Split the target node if the insertion make it overfull.

Insertion (2)

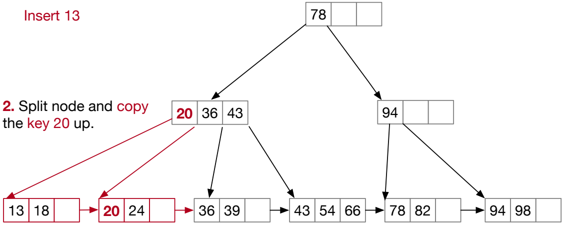


Figure: Insert key 13 ($n = 4$)

- Split the target node if the insertion make it overfull.
- Need to copy the middle key up and adjust the pointers accordingly.

Insertion (3)

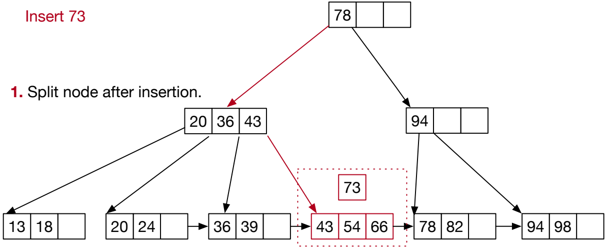


Figure: Insert key 73 ($n = 4$)

- Node splitting can be propagated up recursively.

Insertion (3)

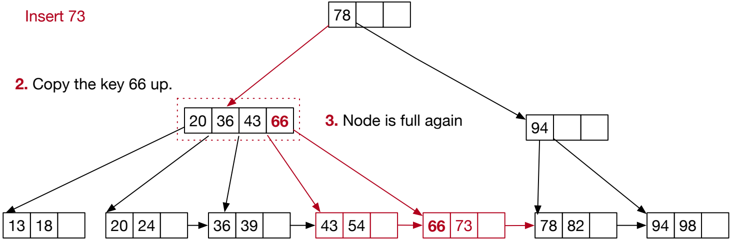


Figure: Insert key 73 ($n = 4$)

- Node splitting can be propagated up recursively.

Insertion (3)

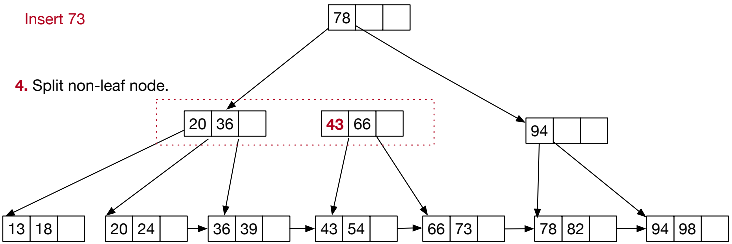


Figure: Insert key 73 ($n = 4$)

- Node splitting can be propagated up recursively.
- When splitting a non-leaf node, we push up the middle key instead of copying it up.

Insertion (3)

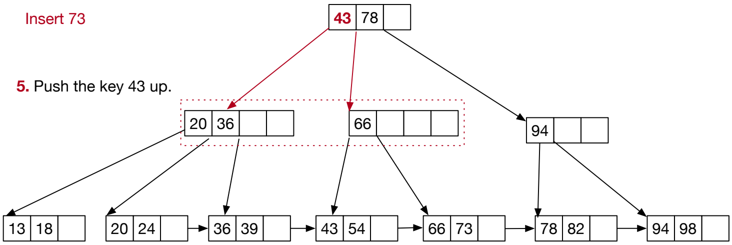


Figure: Insert key 73 ($n = 4$)

- Node splitting can be propagated up recursively.
- When splitting a non-leaf node, we push up the middle key instead of copying it up.
- In the worst case, we have to split the root and create a new root linking to the split nodes.
  – In that case, the tree height increases by one.

Insertion recap

1. Find the correct leaf L for the given key to be inserted.
2. Add a new entry into L in sorted order.
    - If L has enough space, the operation is done.
    - If L becomes overfull, then
        (a) Split L into two nodes L and L'.
        (b) Redistribute entries evenly and copy up the middle key.
        (c) Adjust the pointers accordingly, including
        (i) next-leaf pointers, and (ii) a pointer from parent of L to L'.
3. To split a non-leaf node, redistribute entries evenly and push up the middle key.
4. Process the nodes recursively until all nodes are half-full.

Deletion (1)

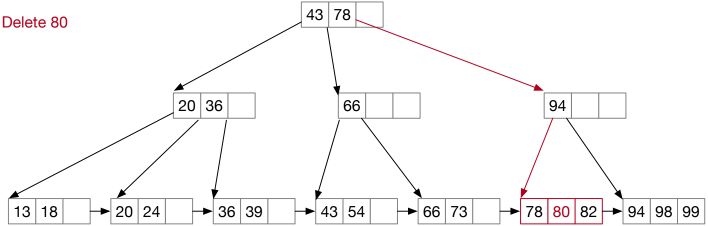


Figure: Delete key 80 ($n = 4$)

Deletion (1)

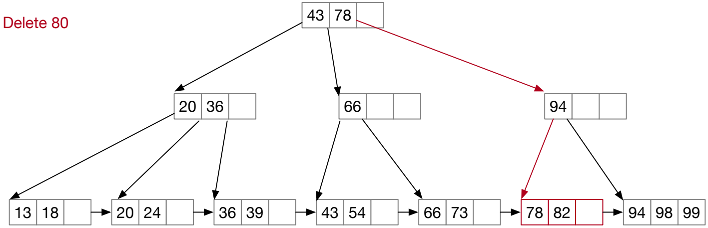


Figure: Delete key 80 ($n = 4$)

Deletion (2)

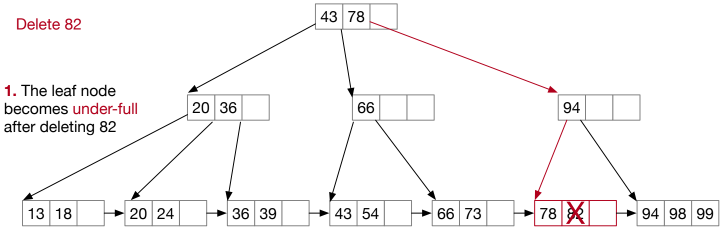


Figure: Delete key 82 ($n = 4$)

- If the target node becomes underfull after deletion, then try to borrow one from siblings.

Deletion (2)

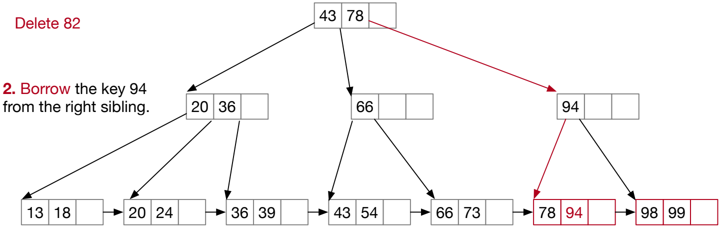


Figure: Delete key 82 ($n = 4$)

- If the target node becomes underfull after deletion, then try to borrow one from siblings.

Deletion (2)

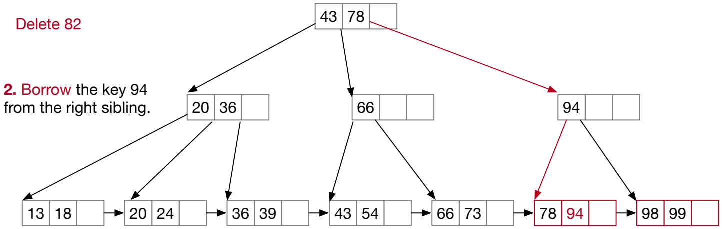


Figure: Delete key 82 ($n = 4$)

- If the target node becomes underfull after deletion, then try to borrow one from siblings.
- Remember to fix the key in the affected parent node.
  - By replacing the affected key with the middle key of the two updated children.

Deletion (2)

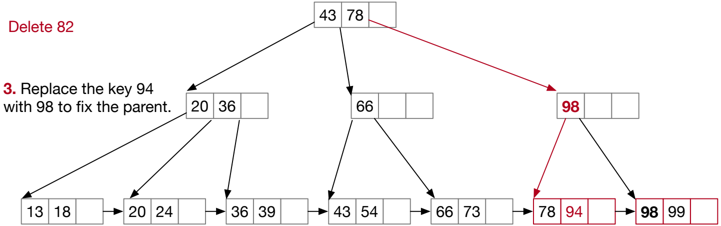


Figure: Delete key 82 ($n = 4$)

- If the target node becomes underfull after deletion, then try to borrow one from siblings.
- Remember to fix the key in the affected parent node.
  - By replacing the affected key with the middle key of the two updated children.

Deletion (3)

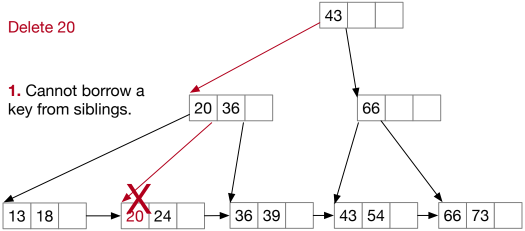


Figure: Delete key 20 ($n = 4$)

- If borrow is not possible, then merge the affected node with one sibling.

Deletion (3)

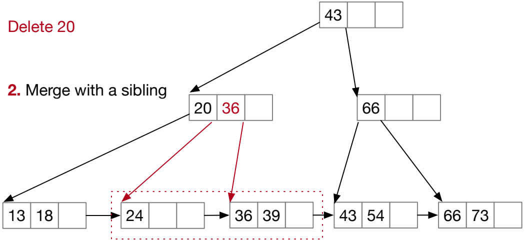


Figure: Delete key 20 ($n = 4$)

- If borrow is not possible, then merge the affected node with one sibling.
- When merging leaf nodes, remove the key associated with the merged nodes from parent.

Deletion (3)

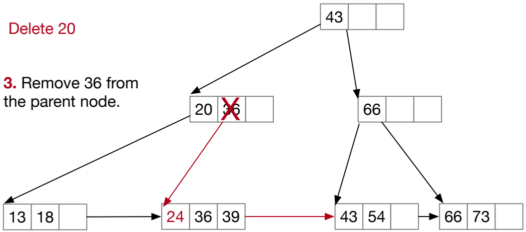


Figure: Delete key 20 ($n = 4$)

- If borrow is not possible, then merge the affected node with one sibling.
- When merging leaf nodes, remove the key associated with the merged nodes from parent.

Deletion (3)

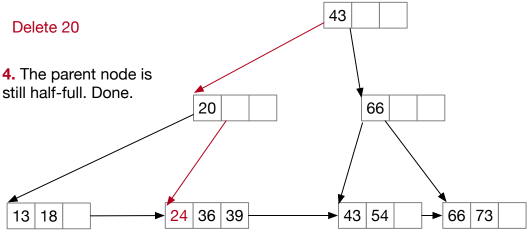
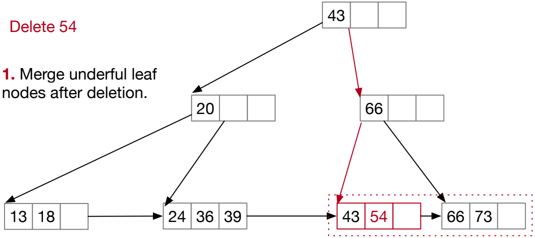


Figure: Delete key 20 ($n = 4$)

- If borrow is not possible, then merge the affected node with one sibling.
- When merging leaf nodes, remove the key associated with the merged nodes from parent.

Deletion (4)



Delete 54

1. Merge underful leaf nodes after deletion.

Figure: Delete key 54 ($n = 4$)

- Deletion can be propagated up all the way to root.

Deletion (4)

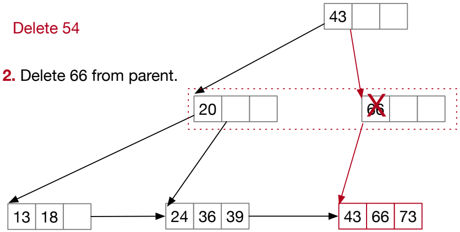
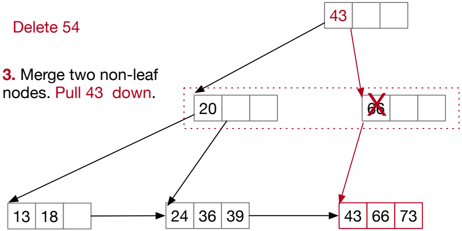


Figure: Delete key 54 ($n = 4$)

- Deletion can be propagated up all the way to root.

Deletion (4)



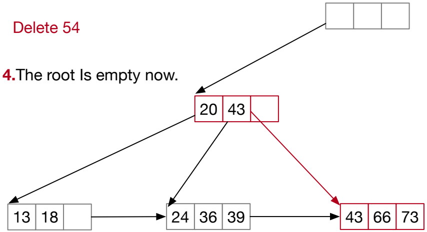
Delete 54

3. Merge two non-leaf nodes. Pull 43 down.

Figure: Delete key 54 ($n = 4$)

- Deletion can be propagated up all the way to root.
- When merging two non-leaf nodes, we need to pull a key down from parent.

Deletion (4)

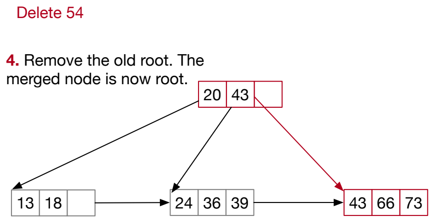

Delete 54

4.The root Is empty now.

Figure: Delete key 54 ($n = 4$)

- Deletion can be propagated up all the way to root.
- When merging two non-leaf nodes, we need to pull a key down from parent.
- When root becomes empty, remove it and make its child as the new root.

Deletion (4)


Delete 54

4. Remove the old root. The merged node is now root.

Figure: Delete key 54 ($n = 4$)

- Deletion can be propagated up all the way to root.
- When merging two non-leaf nodes, we need to pull a key down from parent.
- When root becomes empty, remove it and make its child as the new root.

Deletion Recap

1. Find the correct leaf L.
2. Remove the entry from L for the given key.
    - If L is still half-full, the operation is done.
    - If L becomes under-full, then
        (a) First try to redistribute by borrowing one from siblings.
        (b) If redistribution fails, then merge L and a sibling.
3. When merging two leaf nodes, remove from the parent the key associated with the two leaf nodes to be merged.
4. When merging two non-leaf nodes, pull down the associated key instead.
5. Process the nodes recursively until all nodes are half-full.

Performance analysis

	I/O Cost
Query	$\log_{\lceil n/2 \rceil} N$
Insertion	$\log_{\lceil n/2 \rceil} N$
Deletion	$\log_{\lceil n/2 \rceil} N$

B^+-tree vs. B-tree

- B^+-trees store data entries in leaf nodes only.
  – Look up any key require the same number of I/O's.
- B-trees also store data entries in non-leaf nodes.
  – These recorded can be accessed with fewer I/O's.

Problems with B-tree in disk-based DBMS:

1. Storing more data in non-leaf nodes decreases fanout and increases the tree height.
2. Records in leaves requires more I/O's to access and the majority records are in leaves.
3. Range query is more complicated in B-trees.