Indexing is finding the whole information <u>quicker</u> using only part of it.

Part of information is called <u>Search Key</u>.
It points to the whole information.

~~Primary Key~~
~~Surrogate Key~~
~~Candidate Key~~
~~Foreign Key~~

Index

Index

How to find a webpage in WWW?

Search Key = 'COE848'
Whole Information = https://www.ryerson.ca/calendar/2019-2020/courses/computer-engineering/COE/848/

> 10^{10} seconds by no index, traverse all webpages
< 0.31 seconds by Google
~ 0 seconds by ?

SQL × INDEX


```
SELECT * FROM Director WHERE Id=1
SELECT * FROM Director WHERE LastName='Kubrick'
SELECT * FROM Director WHERE LastName='Kubrick' AND FirstName = 'Stanley'
```

SQL × INDEX

CREATE [UNIQUE] INDEX IndexName ON TableName (c1, c2, ...);

Could be any name, but by convention we follow this:
IX_ColumnName1_ColumnName2_...
UIX_ColumnName1_ColumnName2_...

UNIQUE INDEX does not allow duplicate in indexed columns.
A way to create a <u>candidate key</u> set of columns in a table.

SQL × INDEX


```
SELECT * FROM Director WHERE Id = 1
CREATE UNIQUE INDEX UIX_Id ON Director(Id)
```

By default, most DBMSs CREATE UNIQUE INDEX on primary key set of a table.

SQL × INDEX

What other columns of a table should to be indexed?

- Those columns of table that appears a lot in WHERE clause.
- The search key of the table to find a single or range of rows.

It's a tuning task:

- After the DB goes under heavy load DB designer need to increase retrieval speed.
- Recently is done automatically by DBMS

SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick'
CREATE INDEX IX_LastName ON Director(LastName)
```

SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick' AND FirstName = 'Stanley'
```

Which one?

A) CREATE INDEX IX_LastName_FirstName ON Director(LastName, FirstName)
B) CREATE INDEX IX_FirstName_LastName ON Director(FirstName, LastName)
C) CREATE INDEX IX_FirstName ON Director(FirstName)
D) CREATE INDEX IX_LastName ON Director(LastName)
E) All
F) A & B are the same

SQL × INDEX


```
ALTER TABLE TableName ADD [UNIQUE] INDEX IndexName ON  (c1, c2, ...);
ALTER TABLE TableName DROP INDEX IndexName;
```

DBMS × INDEX

As far as DB designer is concerned, knowing how to CREATE | ADD | DROP INDEX in SQL is more than enough.

However, knowing the implementation details inside DBMS helps DB designer with right decisions about indexing.

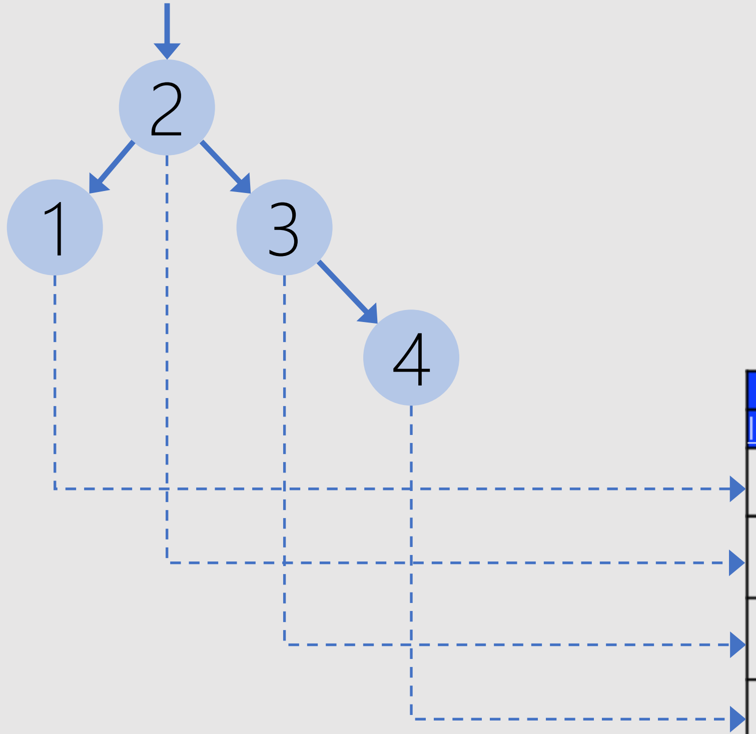
DBMS × INDEX × Binary Search Tree

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

SELECT * FROM Movie WHERE Id = 1

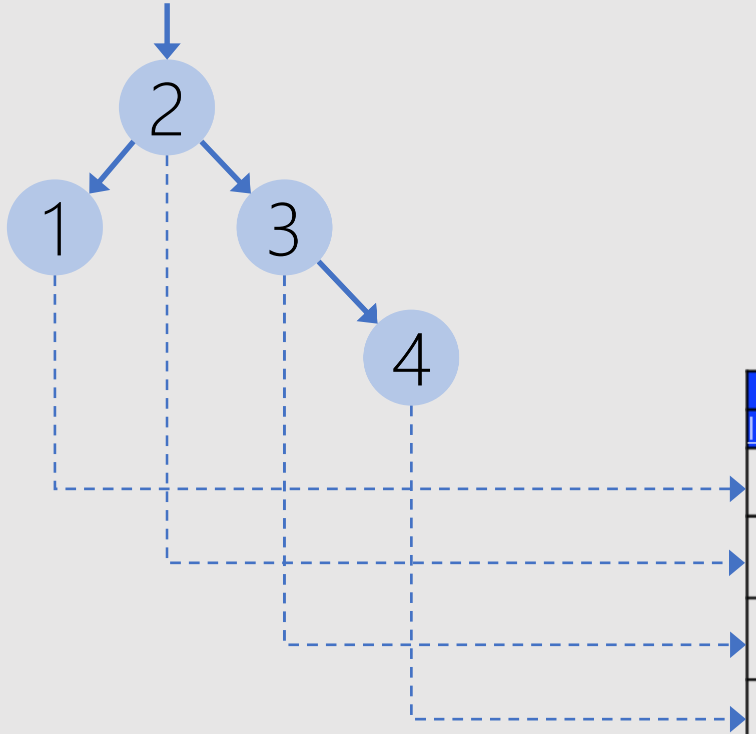
A. Sequential search, check all movies' Id with the given Id, i.e., 1
B. Binary search, after sorting elements in the list by Id
C. Creating a Binary Search Tree (BST).

DBMS × INDEX × Binary Search Tree



Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	2001: A Space Odyssey	English	1968	142
2	Rosemary's Baby	English	1968	NULL
3	The Birds	English	1963	119
4	Planet of the Apes	EN	1968	112

DBMS × INDEX × Binary Search Tree

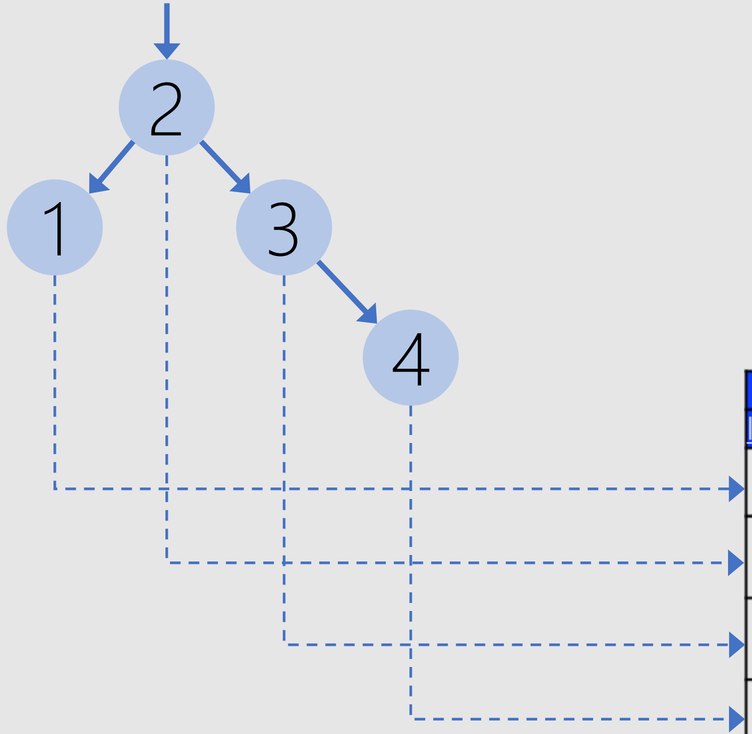


Average
Search O(log n)
Insert O(log n)
Delete O(log n)

Worst (When?)
Search O(n)
Insert O(n)
Delete O(n)

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	2001: A Space Odyssey	English	1968	142
2	Rosemary's Baby	English	1968	NULL
3	The Birds	English	1963	119
4	Planet of the Apes	EN	1968	112

DBMS × INDEX × Binary Search Tree



Average

Search O(log n)
Insert O(log n)
Delete O(log n)

Worst (When?)

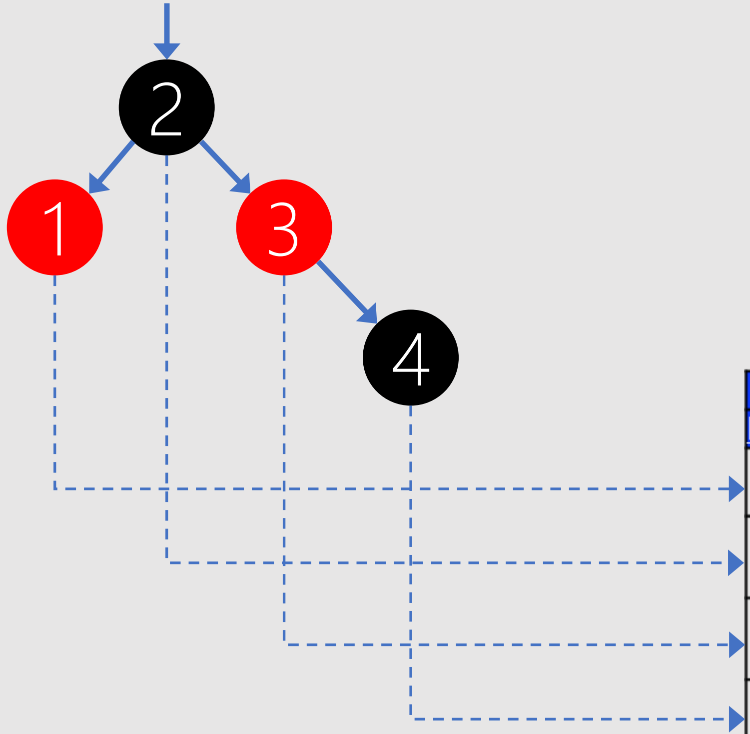
Search O(n)
Insert O(n)
Delete O(n)

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

Overhead:

Each DML on the table needs additional DML on indexes of the table by DBMS

DBMS × INDEX × Balanced Binary Tree 15



Average		Worst	
Search	O(log n)	Search	O(log n)
Insert	O(log n)	Insert	O(log n)
Delete	O(log n)	Delete	O(log n)

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

Overhead:

Each DML on the table needs additional DML on indexes of the table by DBMS

DBMS × INDEX × B-tree

Balanced Multi-way Tree

Bayer, R.; McCreight, E. (1972)
Organization and Maintenance of Large Ordered Indexes
Acta Informatica, 1 (3): 173–189

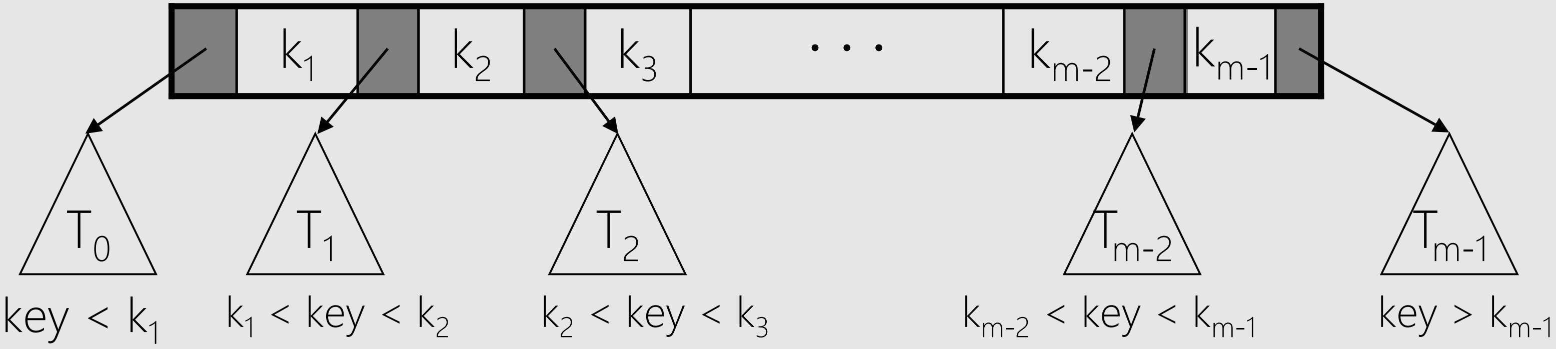
@Boeing

DBMS × INDEX × B-tree

B-tree of order m (branching factor) is a tree in which:

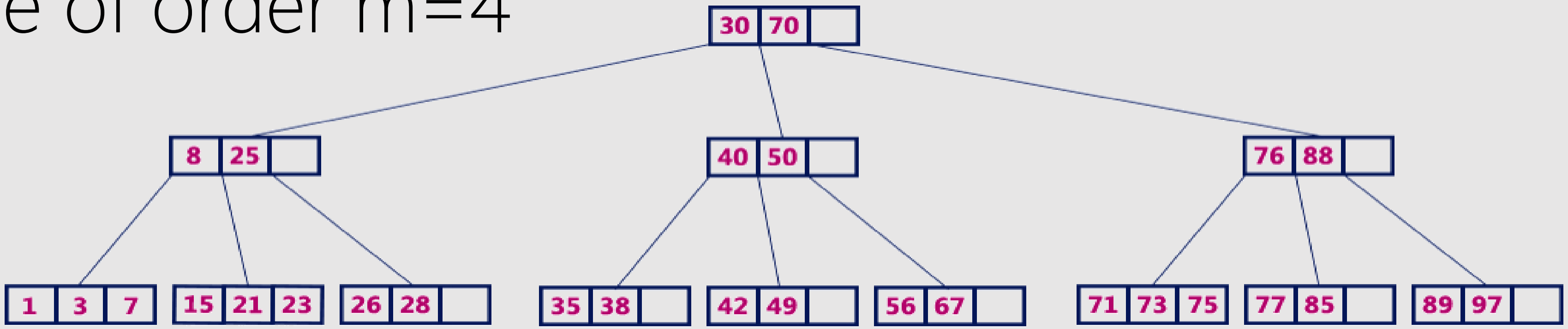
- $0 \leq$ #keys in root $\leq (m-1)$
- $0 \leq$ #subtrees in root $\leq m$
- $(½\, m) \leq$ #keys in other nodes $\leq (m-1)$
- $1+(½\, m) \leq$ #subtrees in other nodes $\leq m$

- The keys in each node are sorted.
- It is balanced!

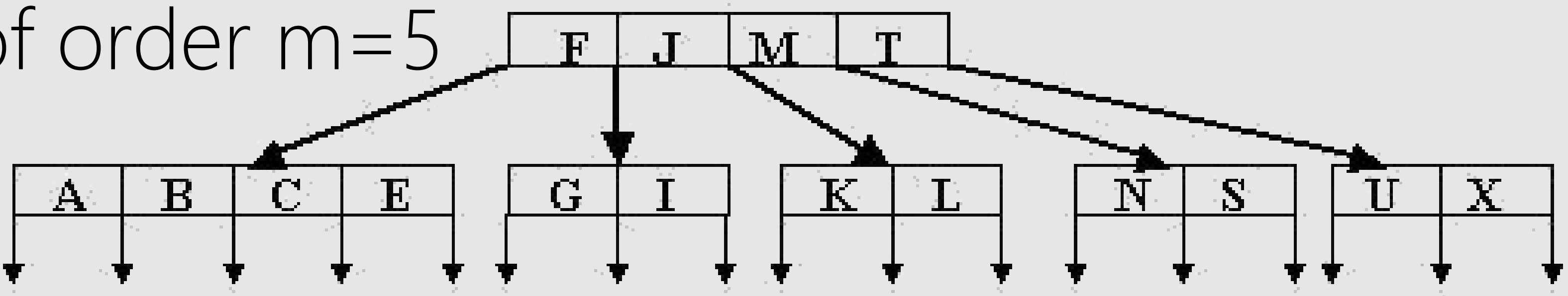


DBMS × INDEX × B-tree

B-tree of order m=4



B-tree of order m=5



DBMS × INDEX × B-tree

The height h of a B-tree of order m, with a total of n keys:

$$log_m^{(n+1)} \leq h \leq 1 + log_{\lceil m/2 \rceil}^{(n+1/2)}$$

If m=300 and n = 16,000,000 then $h \approx 4$.

i.e., the worst case finding a key in such B-tree requires ? accesses.

B-tree × DML

INSERT	Overflow, more than m-1 keys
DELETE	Underflow, less than m/2 keys

B-tree × INSERT

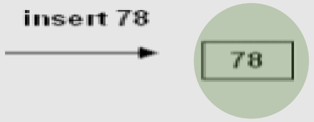
Empty B-tree of order m=3

Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order

B-tree × INSERT

Empty B-tree of order m=3

Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

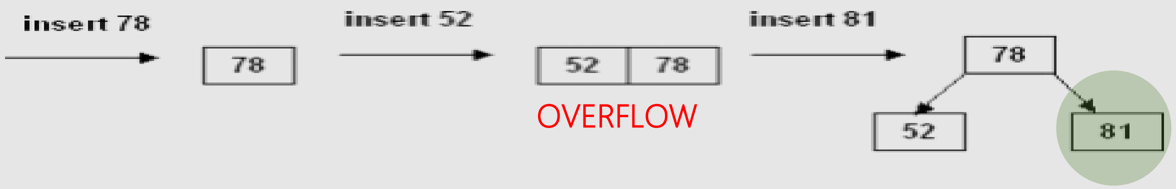
Empty B-tree of order m=3

Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order

B-tree × INSERT

Empty B-tree of order m=3

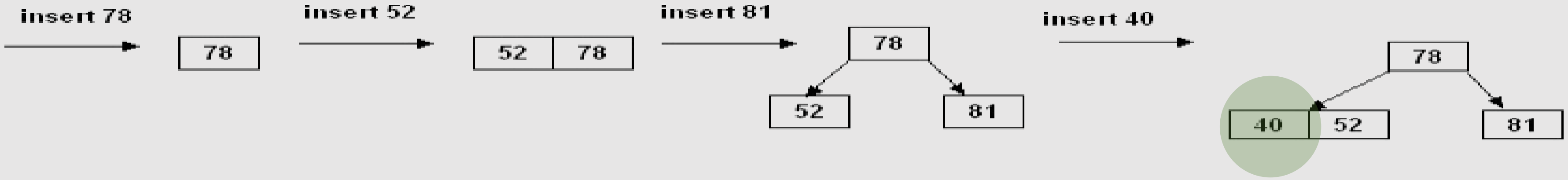
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

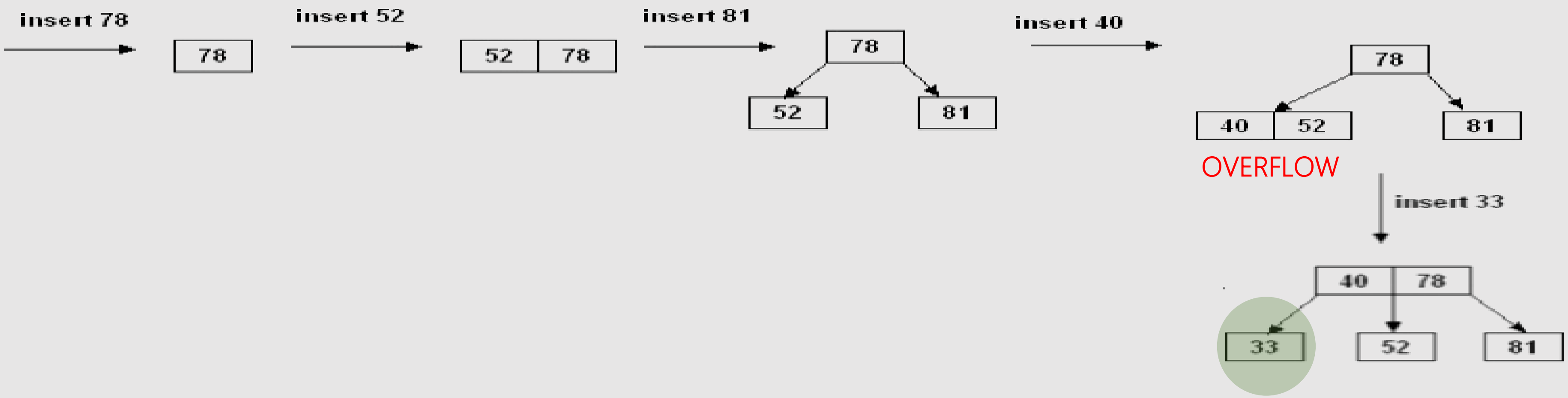
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

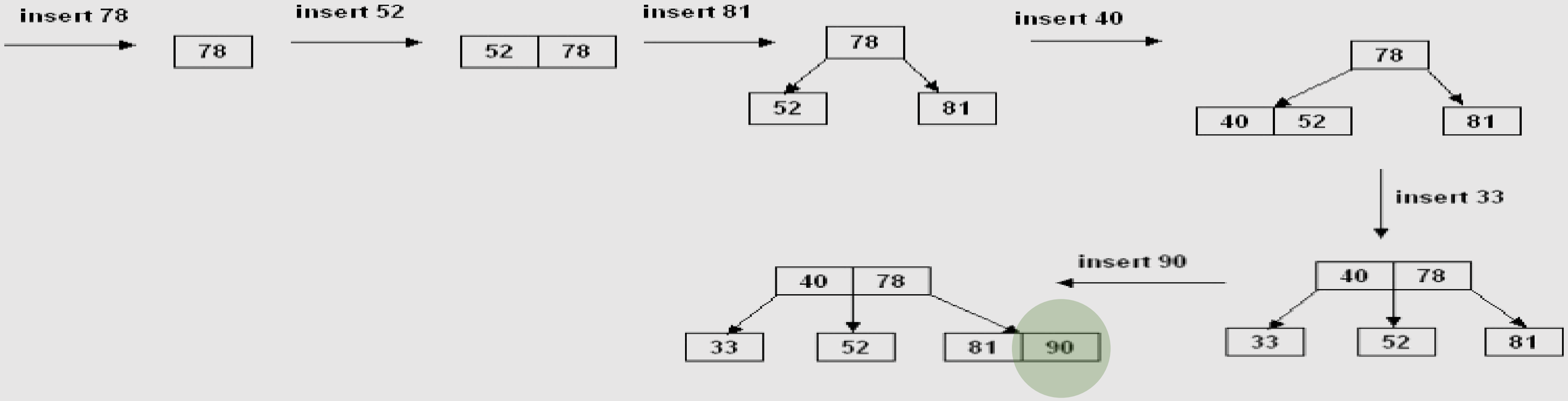
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

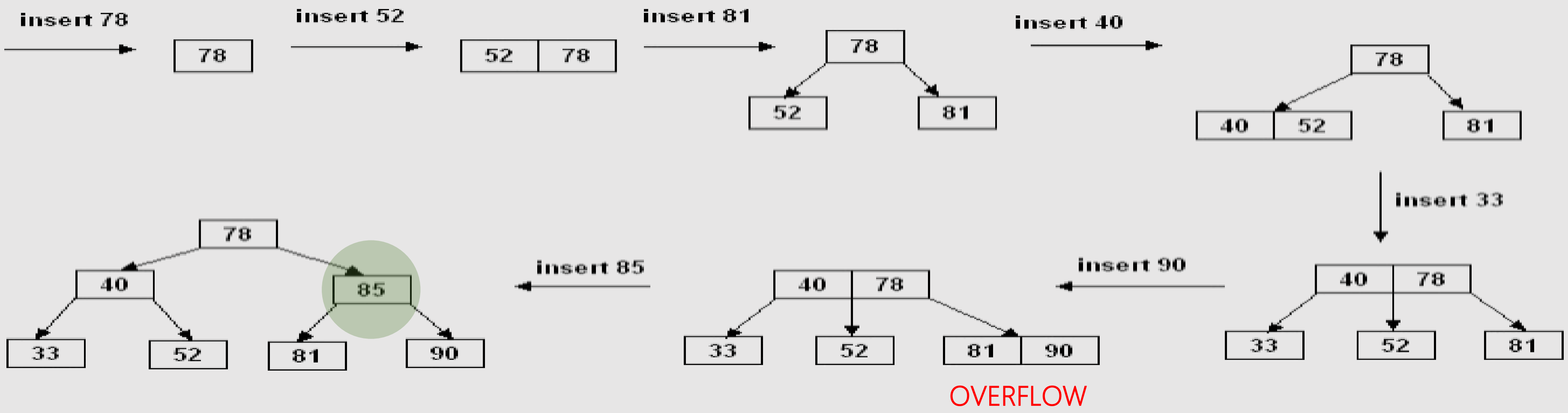
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

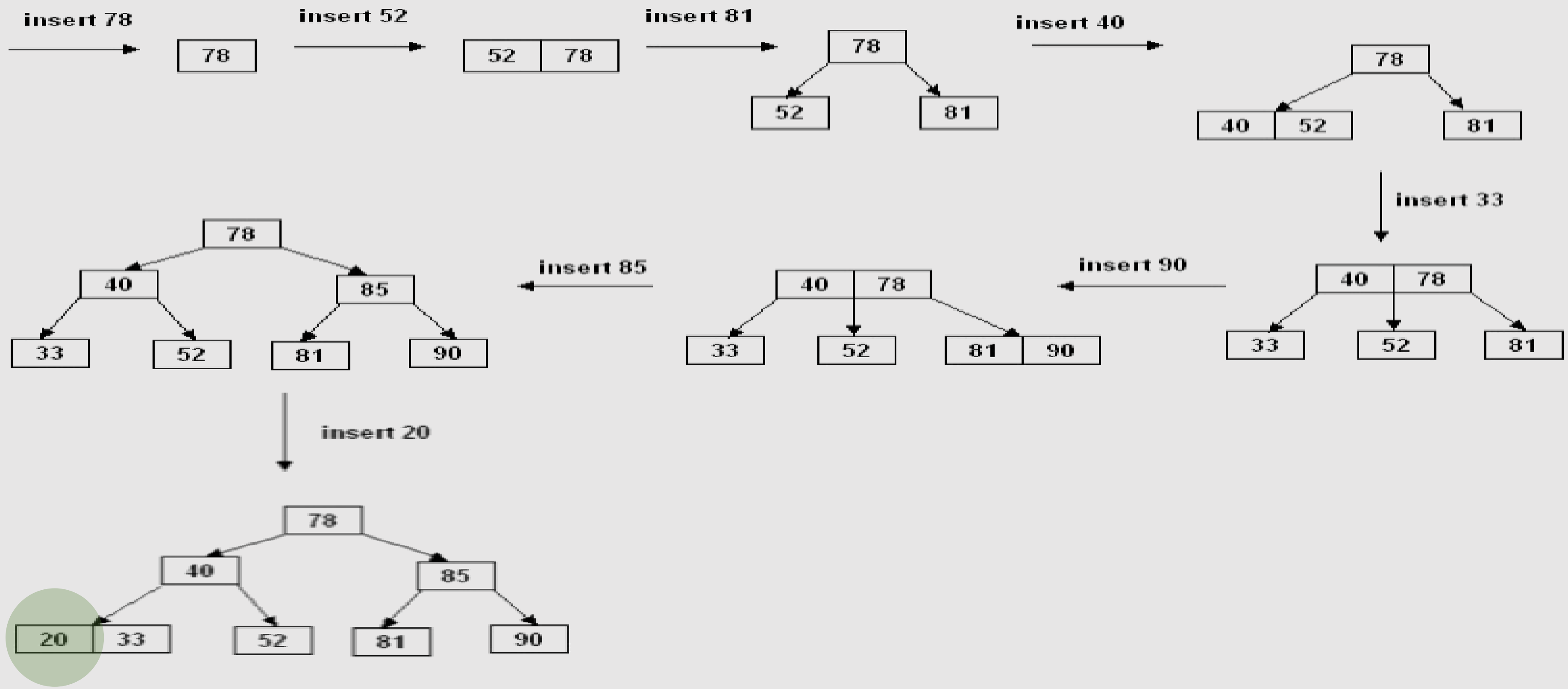
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

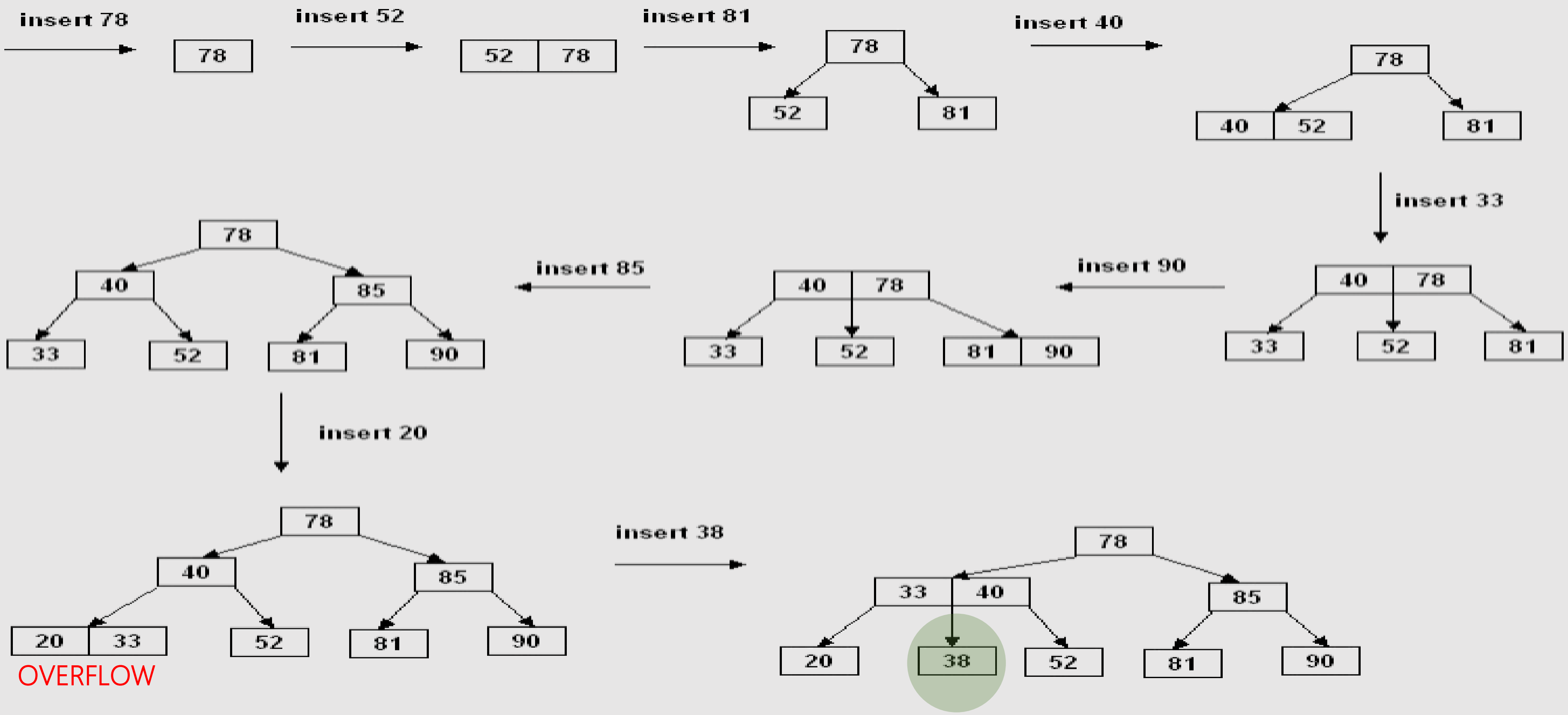
Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

Empty B-tree of order m=3

Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in order



B-tree × INSERT

INSERT(key):

Find the correct spot in a <u>leaf node</u> and do insert.

WHILE OVERFLOW

    Split it into two

    IF the node has parent

        Insert the middle key to the parent (propagate)

        Create right and left siblings

    ELSE

        Create a new root node

B-tree × DELETE

DELETE can happen at leaf or non-leaf!

IF the key is in a leaf node

  Delete the key

ELSE //non-leaf

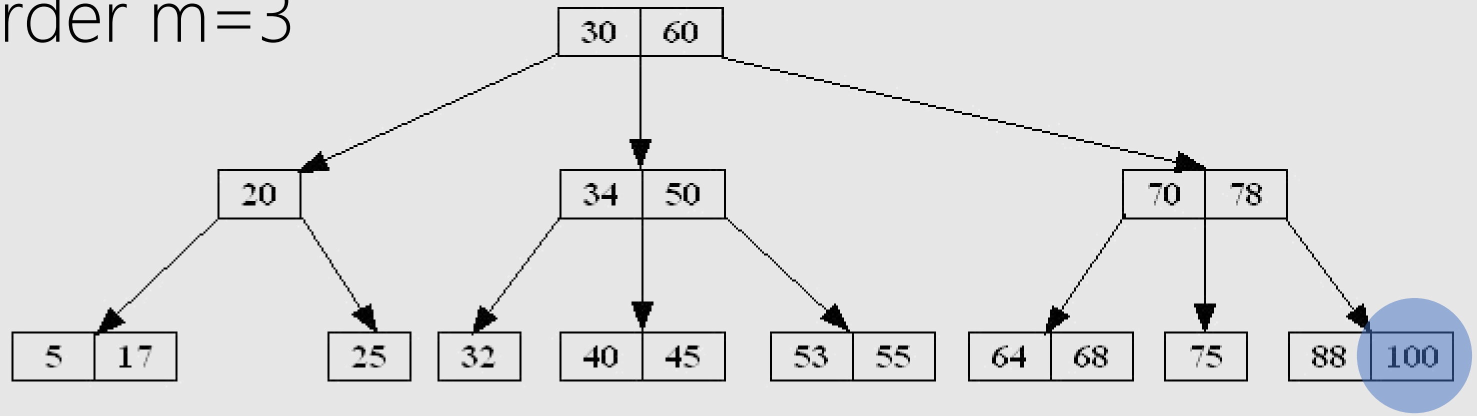
  Replace it with a key in a leaf

  Delete the key in the leaf

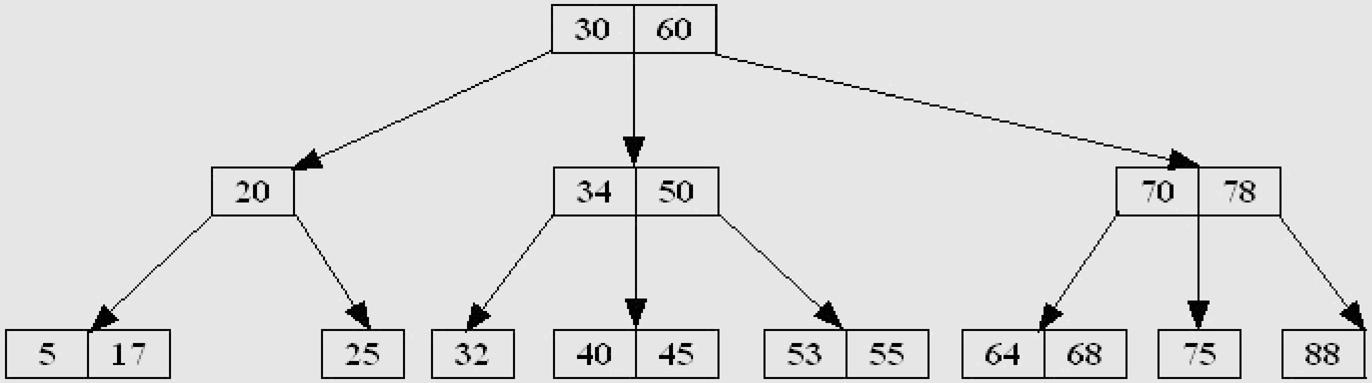
The actual delete always happen at leaf node.

B-tree × DELETE × Leaf

B-tree of order m=3

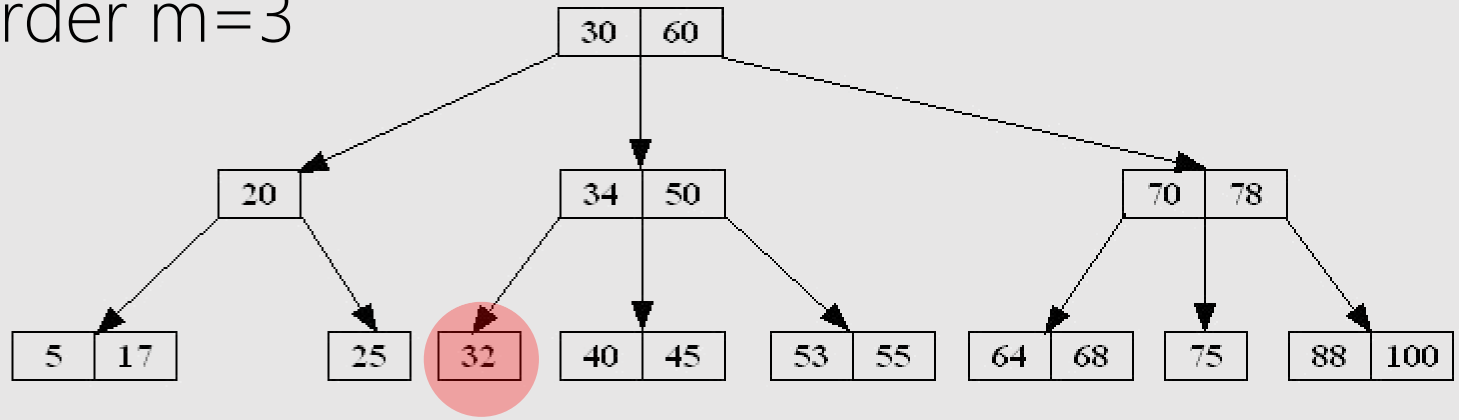


Simple delete key in leaf node. There is no underflow.



B-tree × DELETE × Leaf

B-tree of order m=3

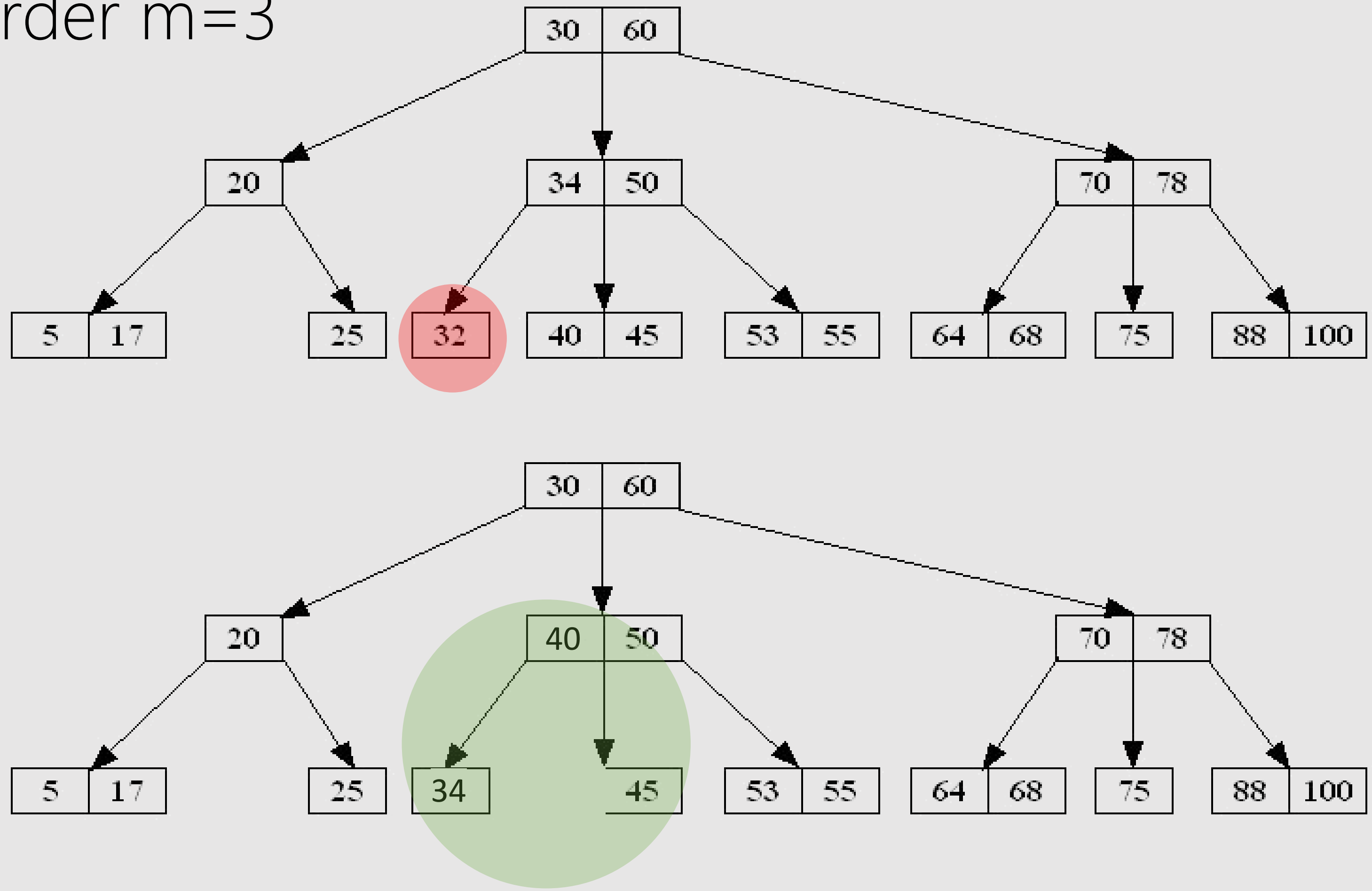


Delete the key, UNDERFLOW!

- Merge with right | left sibling & parent key and sort: [34, 40, 45]
- OVERFLOW:
  - Split by middle key: [34], [45]
  - Movie middle key to parent: [40]

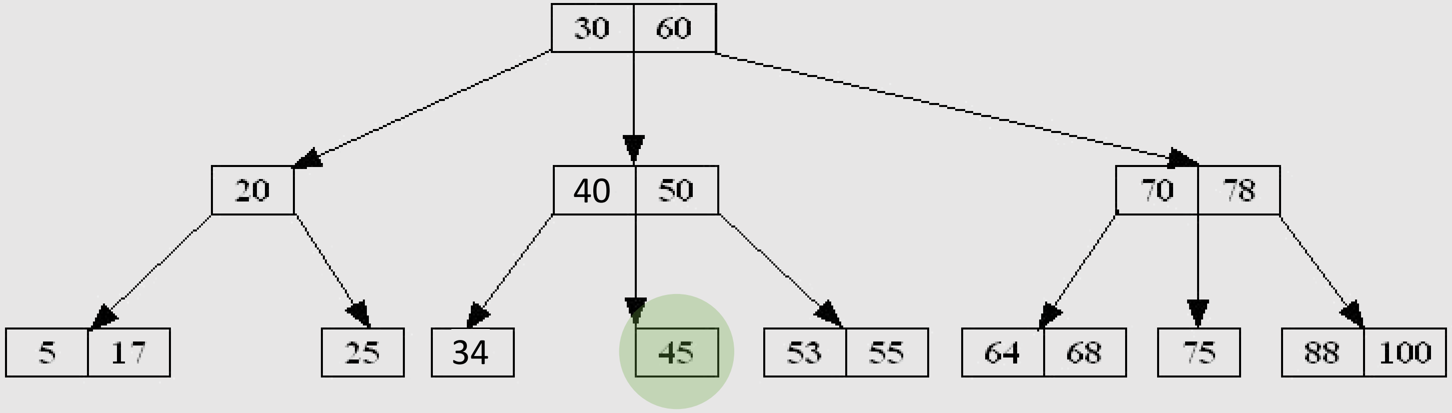
B-tree × DELETE × Leaf

B-tree of order m=3



B-tree × DELETE × Leaf

B-tree of order m=3

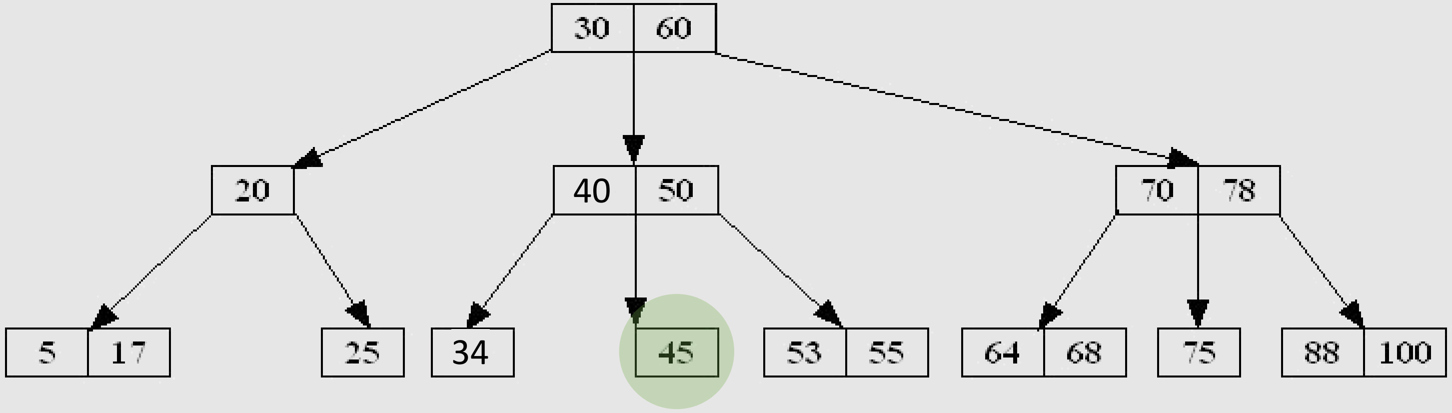


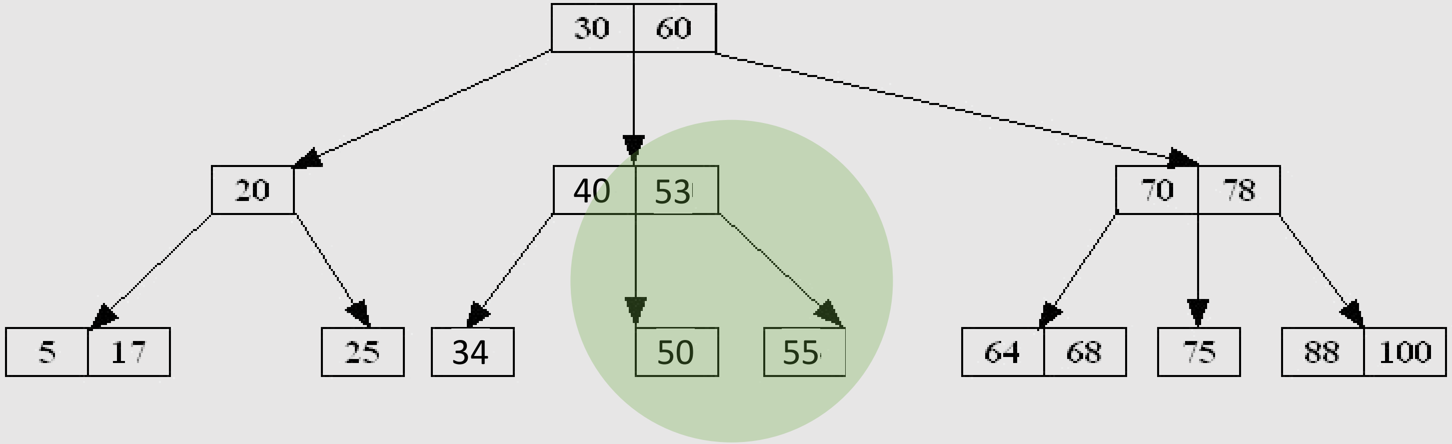
Delete the key, UNDERFLOW!

- Merge with right | ~~left~~ sibling & parent key and sort: [50, 53, 55]
- OVERFLOW:
  - Split by middle key: [50], [55]
  - Movie middle key to parent: [53]

B-tree × DELETE × Leaf

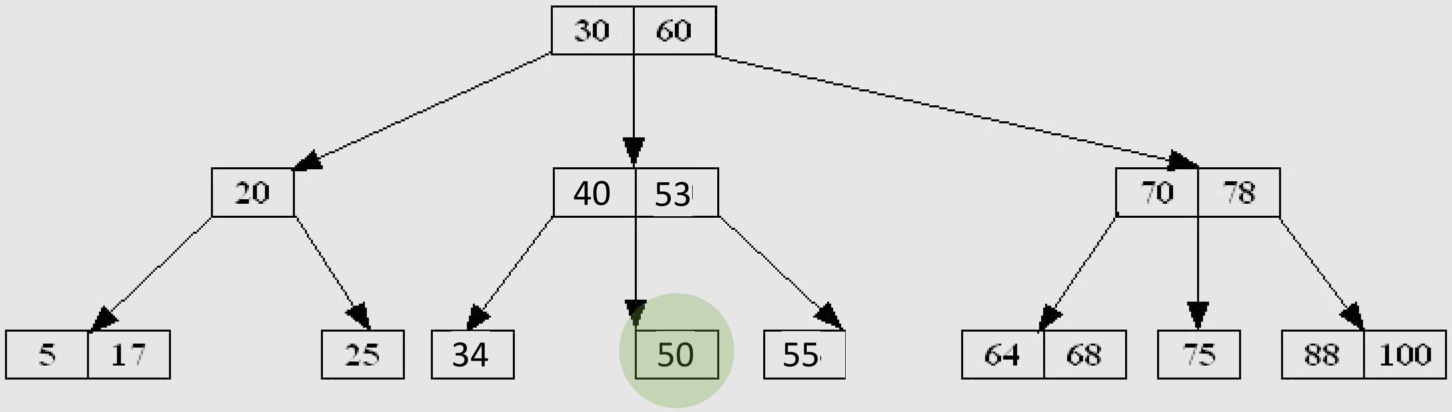
B-tree of order m=3





B-tree × DELETE × Leaf

B-tree of order m=3



Delete the key, UNDERFLOW!

Merge with right | left sibling & parent key and sort: [53, 55]

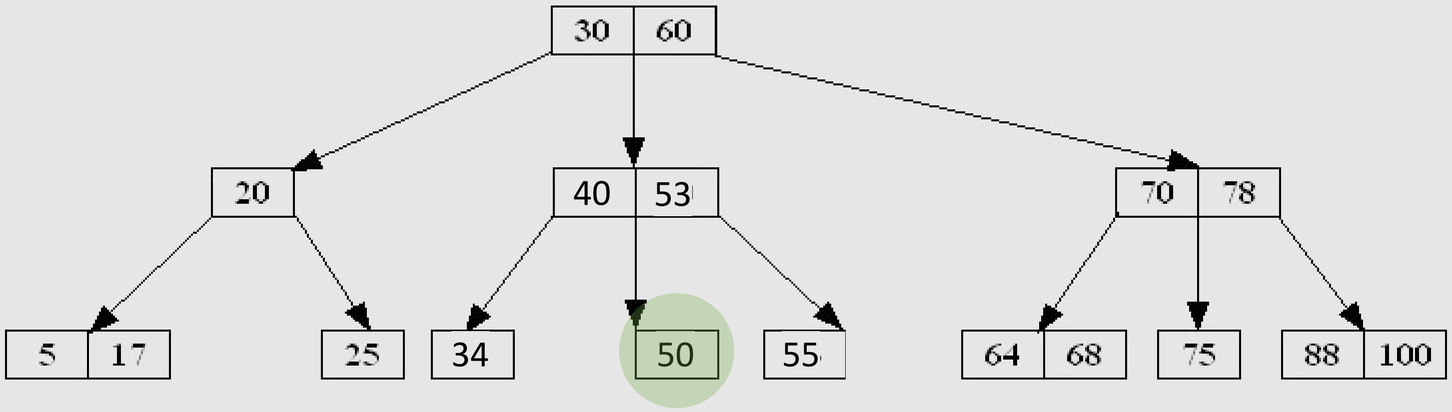
~~OVERFLOW:~~

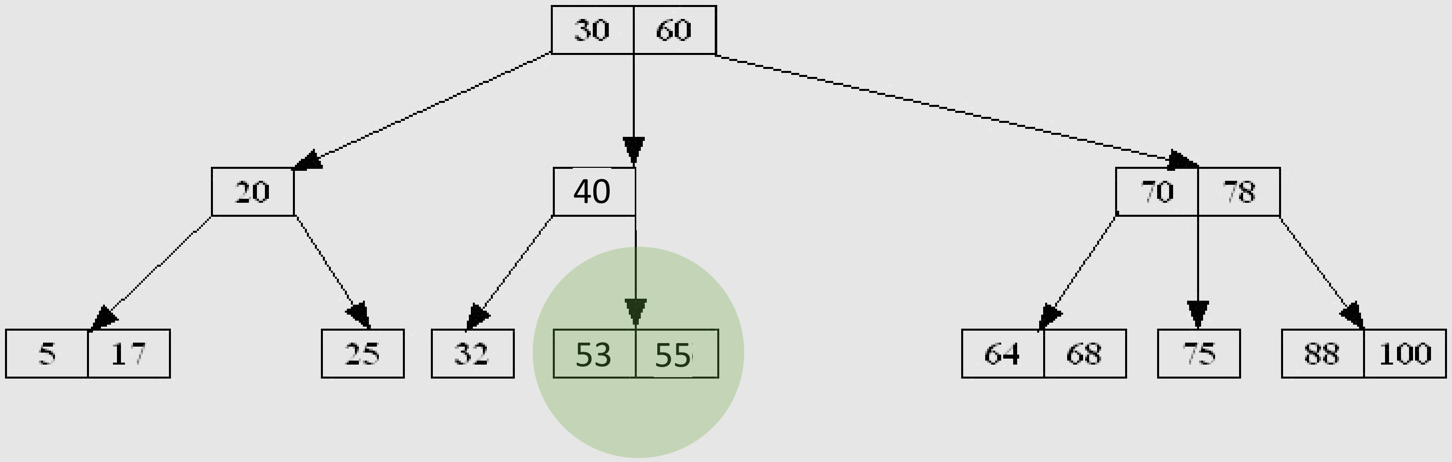
~~Split by middle key~~

~~Movie middle key to parent~~

B-tree × DELETE × Leaf

B-tree of order m=3





B-tree × DELETE × Leaf

DELETE(key):

Find the key

IF the key is in a leaf node

    Delete the key

    IF (NOT root) & UNDERFLOW

        Merge with a adjacent sibling with more keys (right | left) & parent key

        IF OVERFLOW

            Split based on middle key

            Replace middle key with the parent key

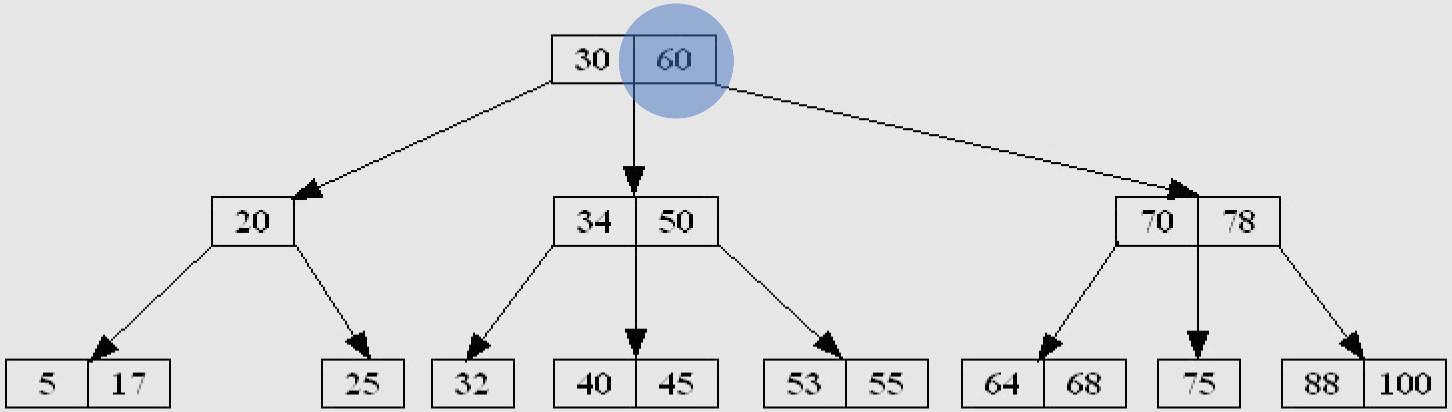
        ELSE //parent node lost a key and might UNDERFLOW

            IF UNDERFLOW in parent

                GO TO

B-tree × DELETE × Non-leaf

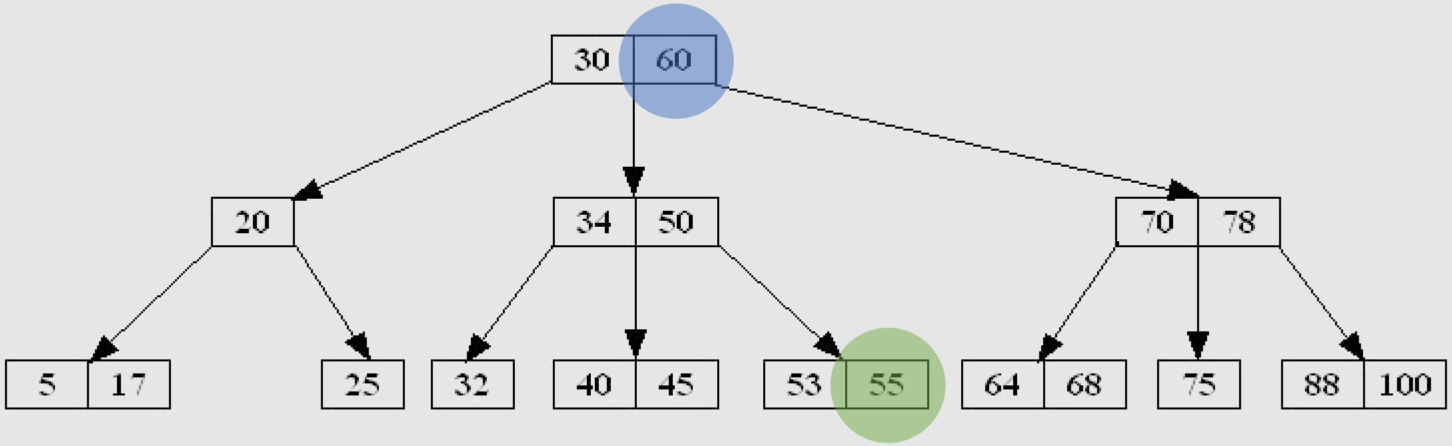
B-tree of order m=3



To replace the current key with a key in leaves, what would be the options?

B-tree × DELETE × Non-leaf

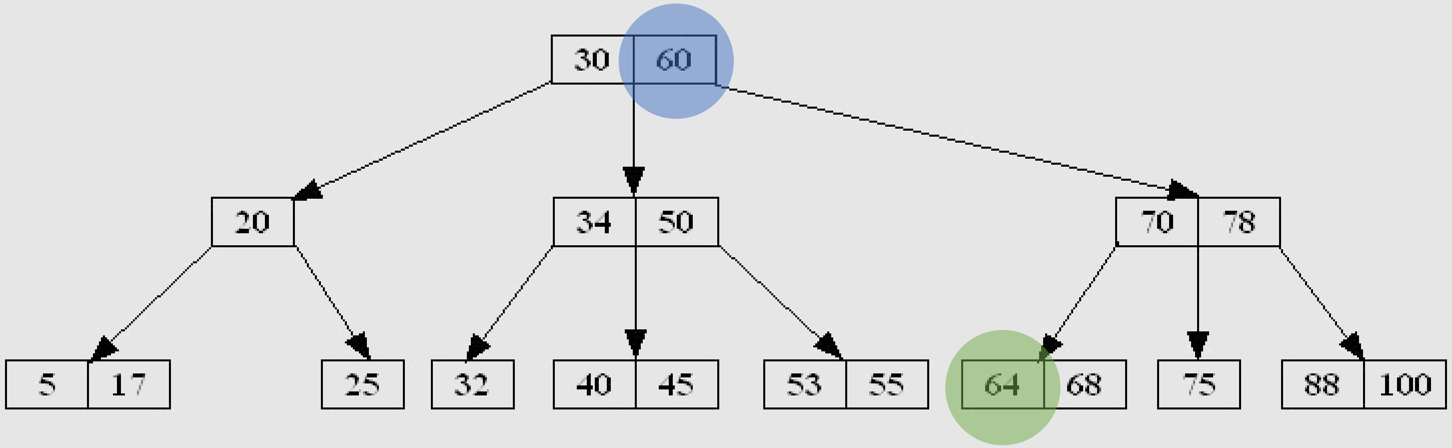
B-tree of order m=3



PREDECESSOR(key): Largest key less than the current key
Largest key in left subtree of the current key
Right-most key in left subtree of the current key

B-tree × DELETE × Non-leaf

B-tree of order m=3



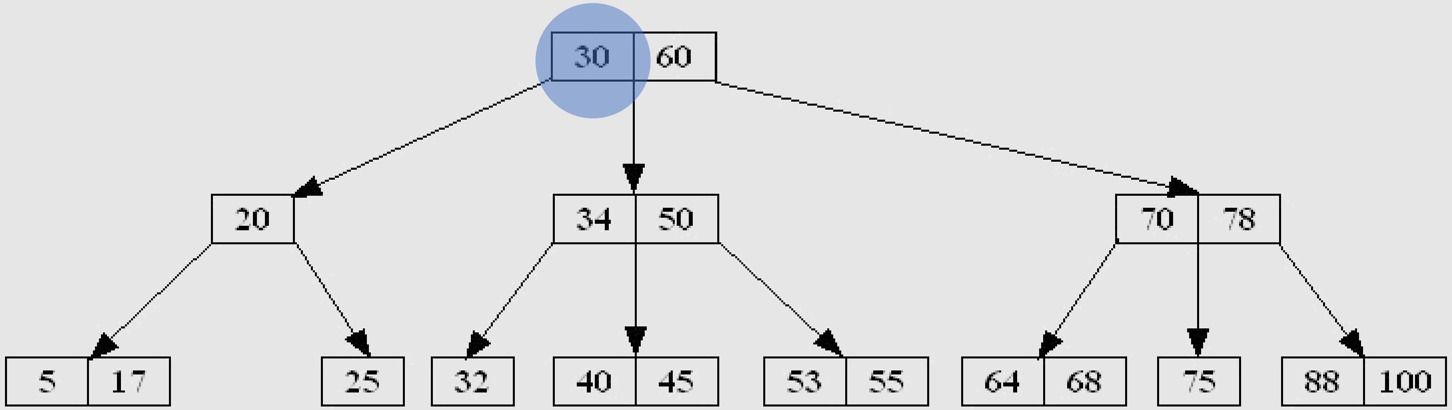
SUCCESSOR(key): Smallest greater than the current key
Smallest in right subtree of the current key
Left-most key in right subtree of the current key

B-tree × DELETE × Non-leaf

4

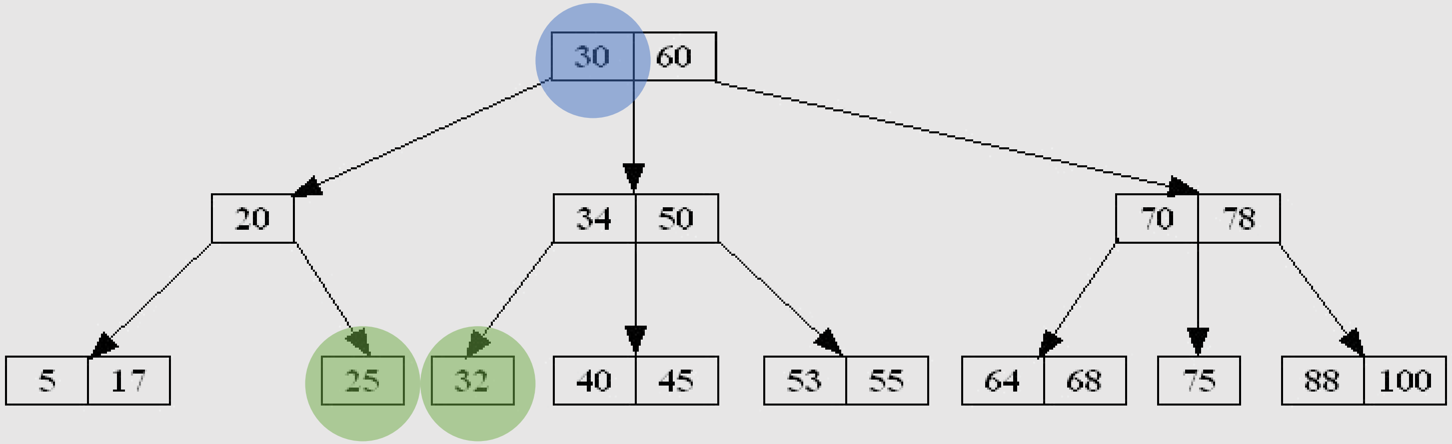
B-tree of order m=3

4



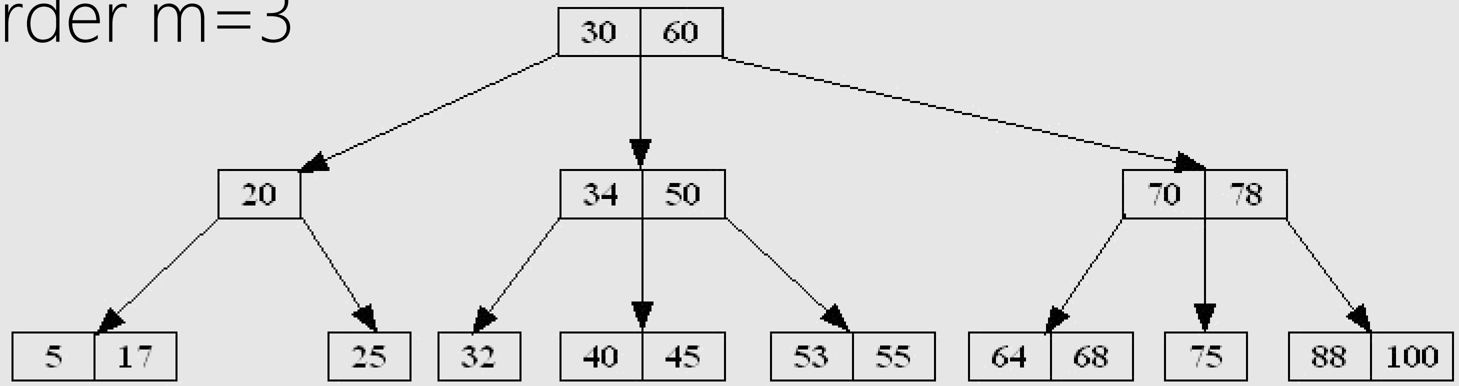
B-tree × DELETE × Non-leaf

B-tree of order m=3



B-tree × DELETE × Non-leaf

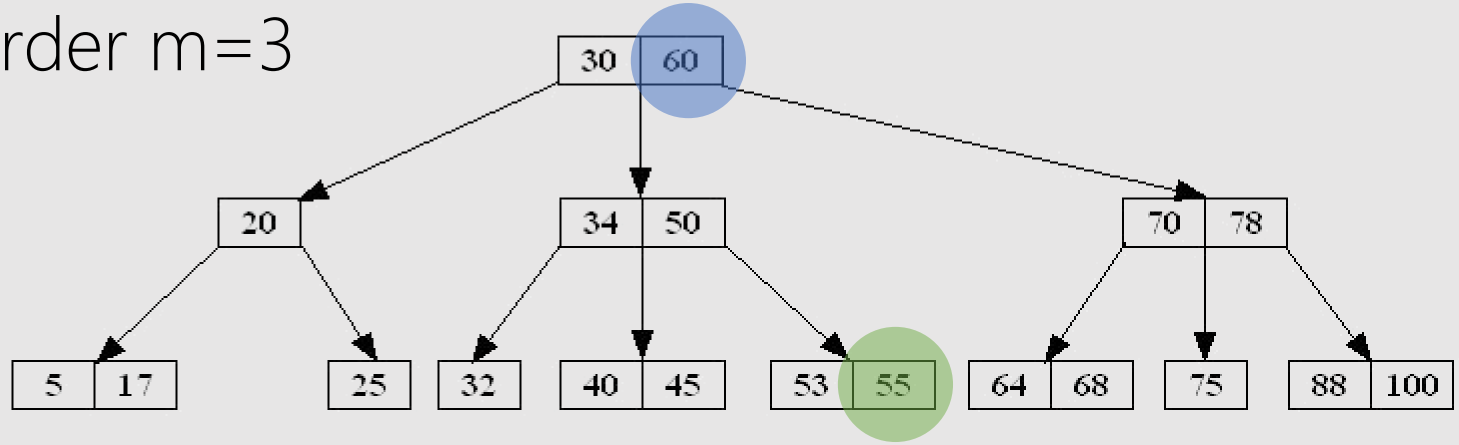
B-tree of order m=3



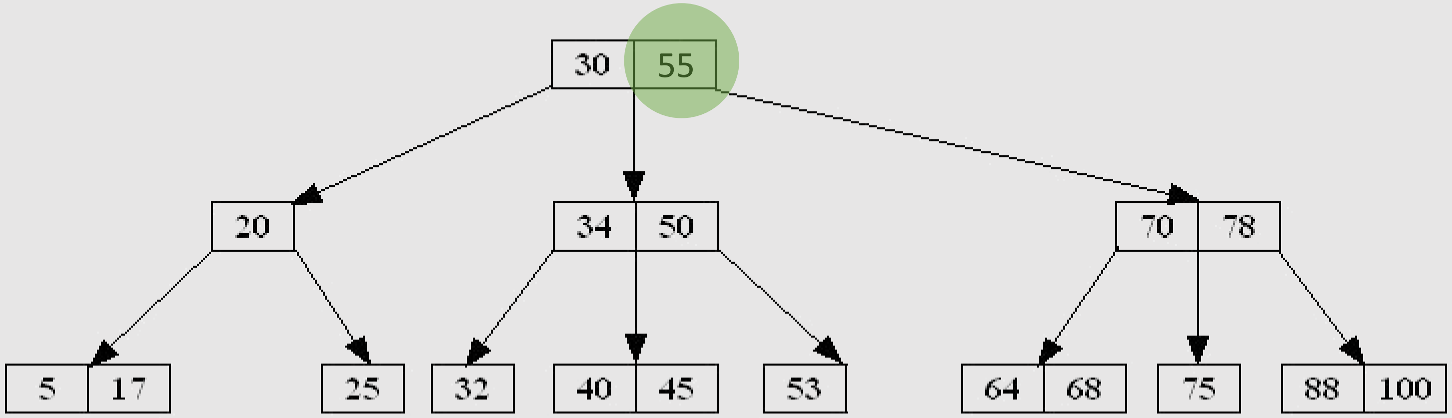
PREDECESSOR	Key	SUCCESSOR
17	20	25
25	30	32
32	34	40
45	50	53
55	60	64
68	70	75
75	78	88

B-tree × DELETE × Non-leaf

B-tree of order m=3

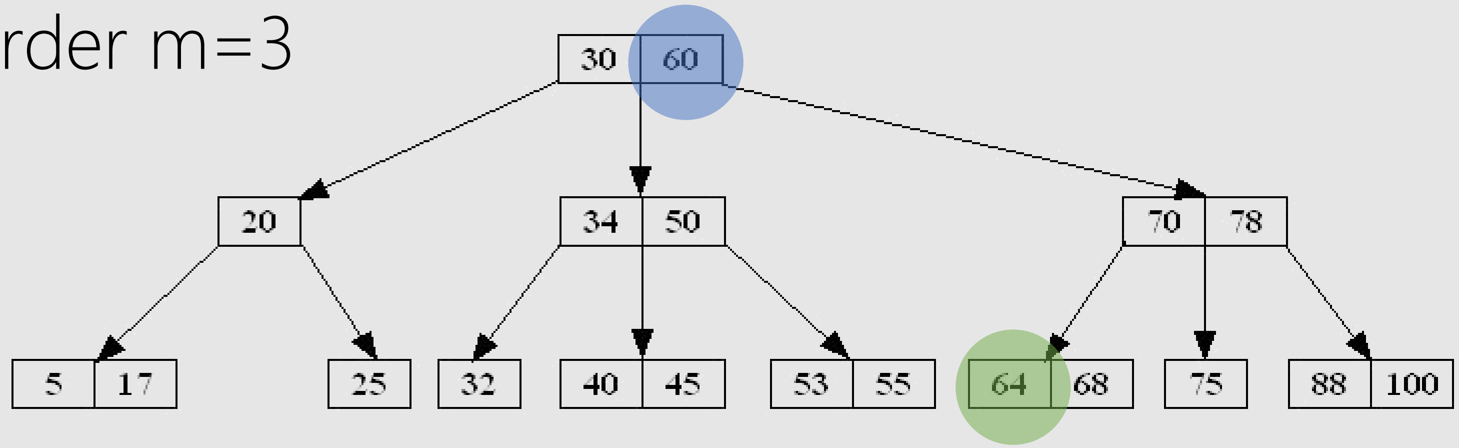


Replace key with PREDECESSOR in non-leaf node

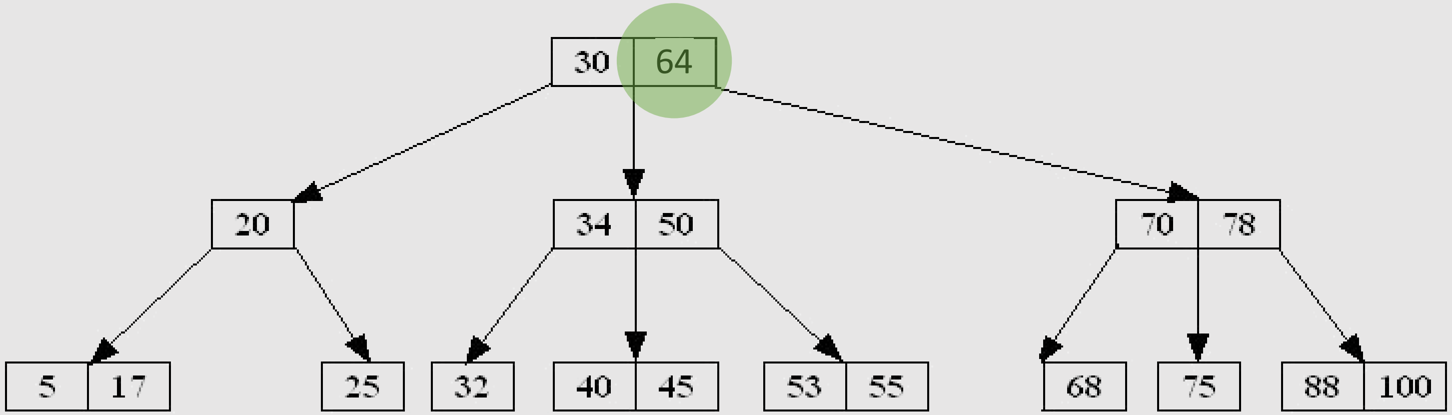


B-tree × DELETE × Non-leaf

B-tree of order m=3

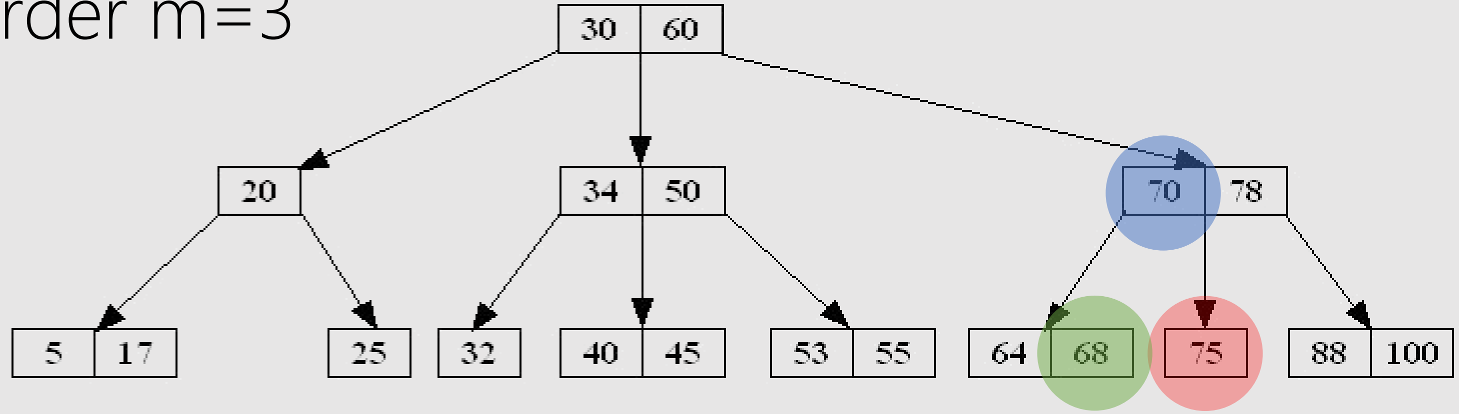


Or replace key with SUCCESSOR in non-leaf node



B-tree × DELETE × Non-leaf

B-tree of order m=3



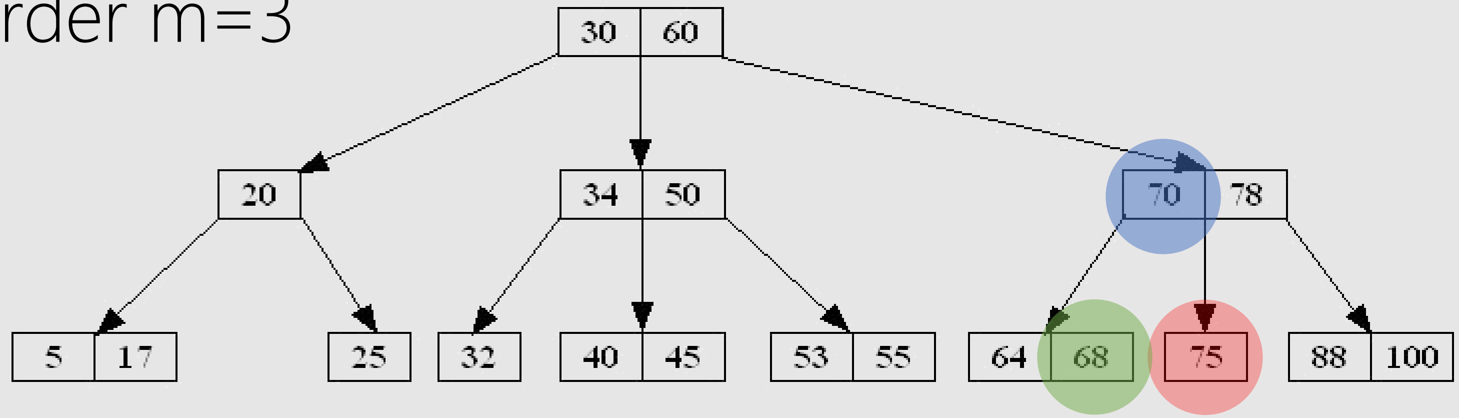
To choose between SUCCESSOR | PREDESSESOR:

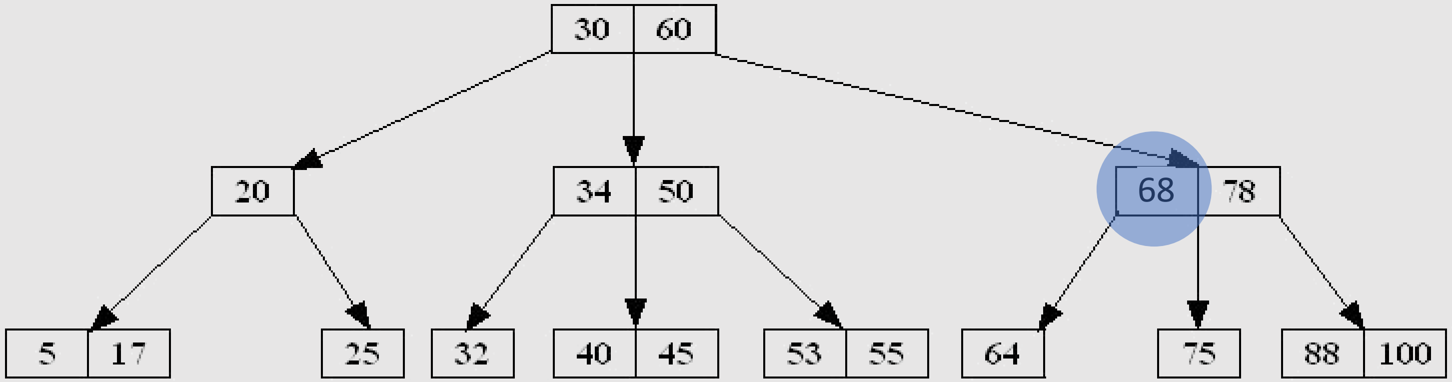
The one whose node is bigger.

B-tree × DELETE × Non-leaf

50

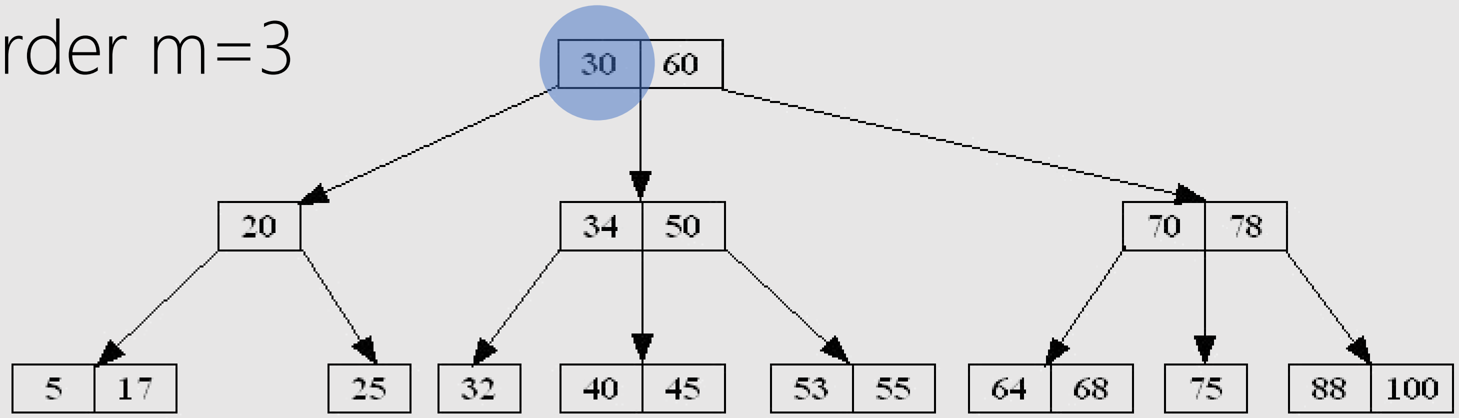
B-tree of order m=3





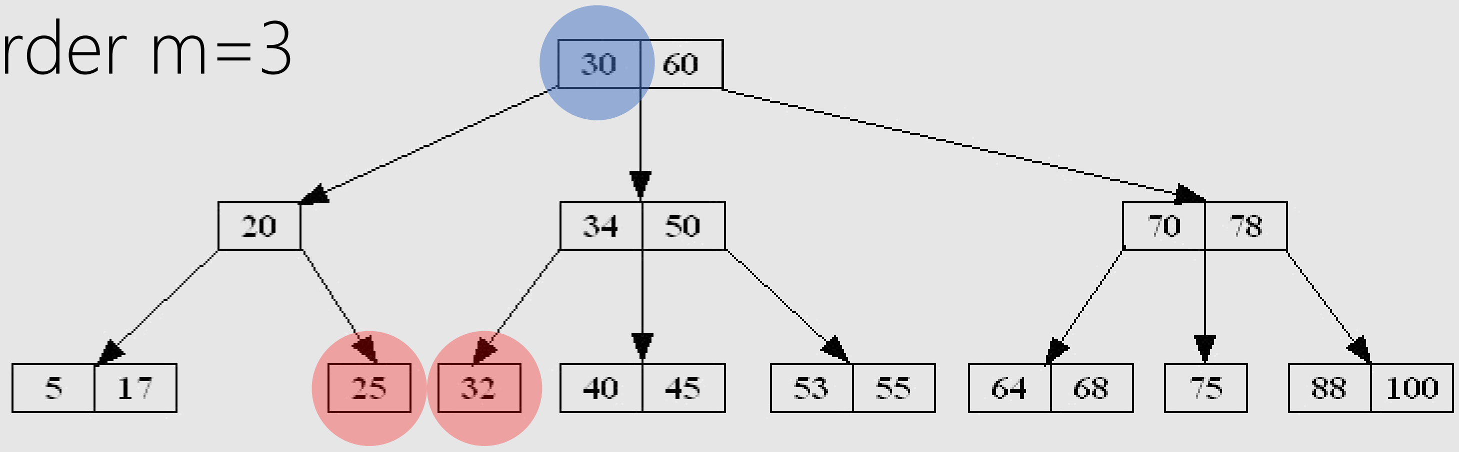
B-tree × DELETE × Non-leaf

B-tree of order m=3



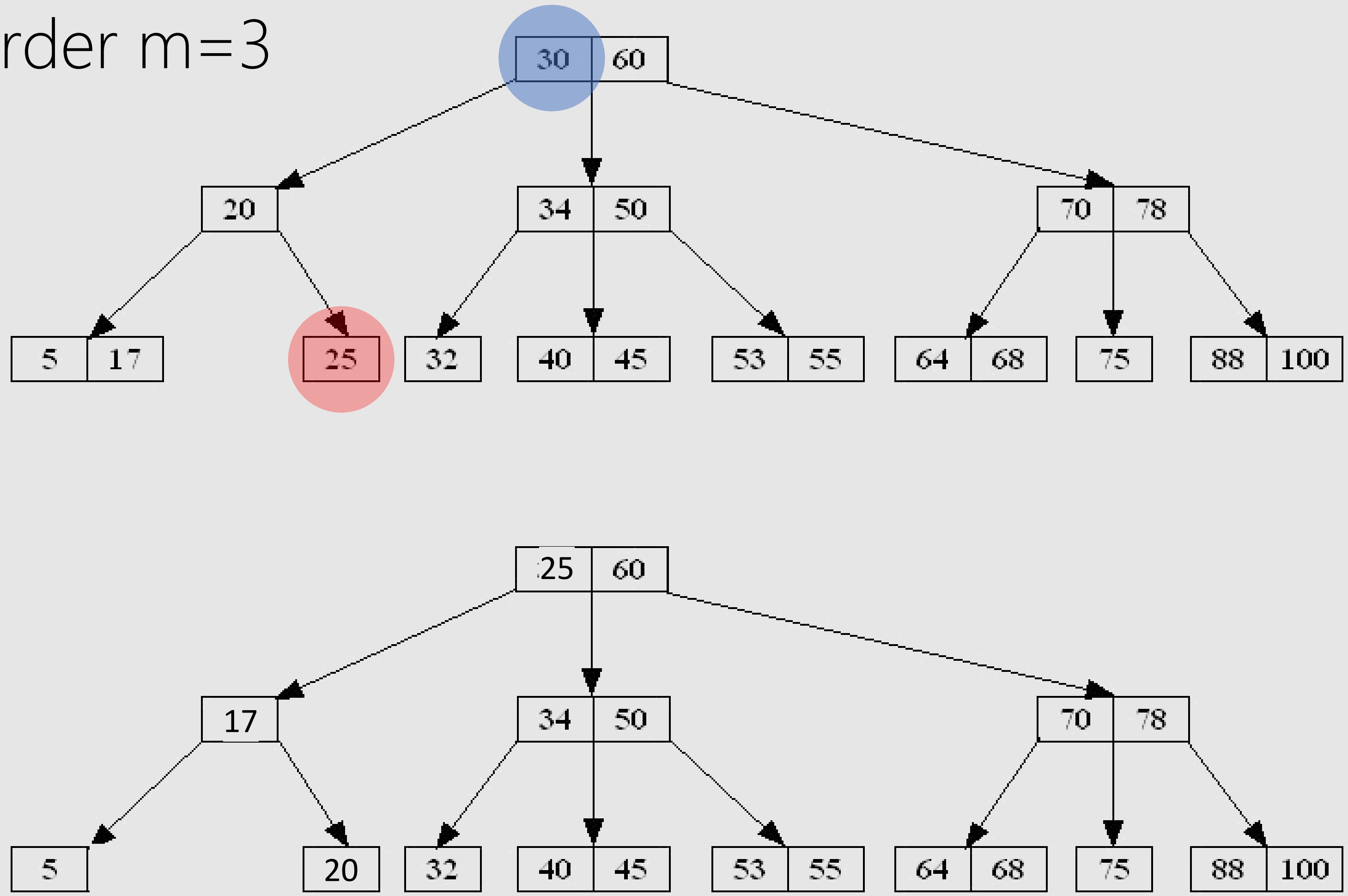
B-tree × DELETE × Non-leaf

B-tree of order m=3



B-tree × DELETE × Non-leaf

B-tree of order m=3



DBMS × INDEX × B-tree × DELETE


```
DELETE(key):
Find the key
IF the key is in a non-leaf node
    key' = PREDECESSOR(key)
    key'' = SUCCESSOR(key)
    IF |NODE(key')| >= |NODE(key'')|
        DELETE(key')
    ELSE
        DELETE(key'')
```

The actual delete always happen at leaf node.

DBMS × SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick' AND FirstName = 'Stanley'
```

Which one?

A) CREATE INDEX IX_LastName_FirstName ON Director(LastName, FirstName)
B) CREATE INDEX IX_FirstName_LastName ON Director(FirstName, LastName)
C) CREATE INDEX IX_FirstName ON Director(FirstName)
D) CREATE INDEX IX_LastName ON Director(LastName)
E) All
F) A & B are the same

DBMS × SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick' AND FirstName = 'Stanley'
```

Which one?

A) `CREATE INDEX IX_LastName_FirstName ON Director(LastName, FirstName)`

100% helps

A) `CREATE INDEX IX_FirstName_LastName ON Director(FirstName, LastName)`

100% helps

A) `CREATE INDEX IX_FirstName ON Director(FirstName)`

50% helps. Finding FirstName is fast, but after that sequential search to find LastName

A) `CREATE INDEX IX_LastName ON Director(LastName)`

50% helps. Finding LastName is fast, but after that sequential search to find FirstName

A) All

Having 4 indexes on a table is too much overhead

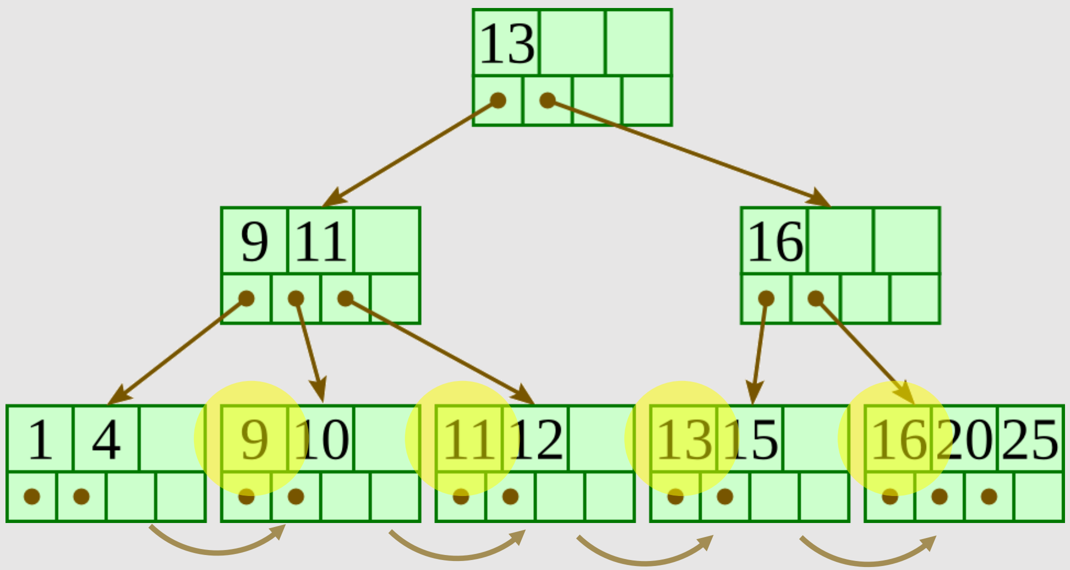
A) A & B are the same

The sorting in B-tree is different, but the effect (help) is the same.

DBMS × INDEX × B^{+}-Tree

B+-tree is an extension to B-tree in which

- Leaf nodes has a copy of all parents key.
- Leaf nodes are linked together in order to support range queries



DBMS × INDEX × B^+-Tree

B+-tree is an extension to B-tree in which

- Leaf nodes has a copy of all parents key.
- Leaf nodes are linked together in order to support range queries

```
SELECT * FROM Movie
WHERE ReleaseDate BETWEEN 1960 AND 2000
```

B^+-tree is the *de facto* standard for indexing in DBMS

DBMS × INDEX × B⁺-Tree


```
CREATE INDEX IX_DateOfBirth_PlaceOfBirth ON  Director(DateOfBirth, PlaceOfBirth)
```

Helps with which one?

A) `SELECT * FROM Director WHERE DateOfBirth = 1955 AND PlaceOfBirth = 'USA'`

B) `SELECT * FROM Director WHERE DateOfBirth = 1955`

C) `SELECT * FROM Director WHERE PlaceOfBirth = 'USA'`

D) `SELECT * FROM Director WHERE DateOfBirth BETWEEN 1955 AND 1980`

E) ALL

F) None

DBMS × INDEX × Best Practice

What columns should be indexed by default on any DB?

Why?

DBMS × INDEX × Best Practice

A. Unique Indexes on Primary Keys

SELECTs usually have primary keys in WHERE clause

Joins are usually based on primary keys equal to foreign keys

B. Unique Indexes on Candidate Keys

To provide uniqueness check

SELECTs usually have candidate keys in WHERE clause

C. Indexes on Foreign Keys

Joins are usually based on primary keys equal to foreign keys

DBMS × INDEX × Book

2nd Ed.

- CH08: Indexes in SQL
- CH14: Index Structures (B-tree, HashTable, Bitmap, ...)

1st Ed.

- CH06: The Database Language SQL (6.6.5 & 6.6.6)
- CH13: Index Structures (B-tree, HashTable)
- CH14: Multidimensional & Bitmap Indexes