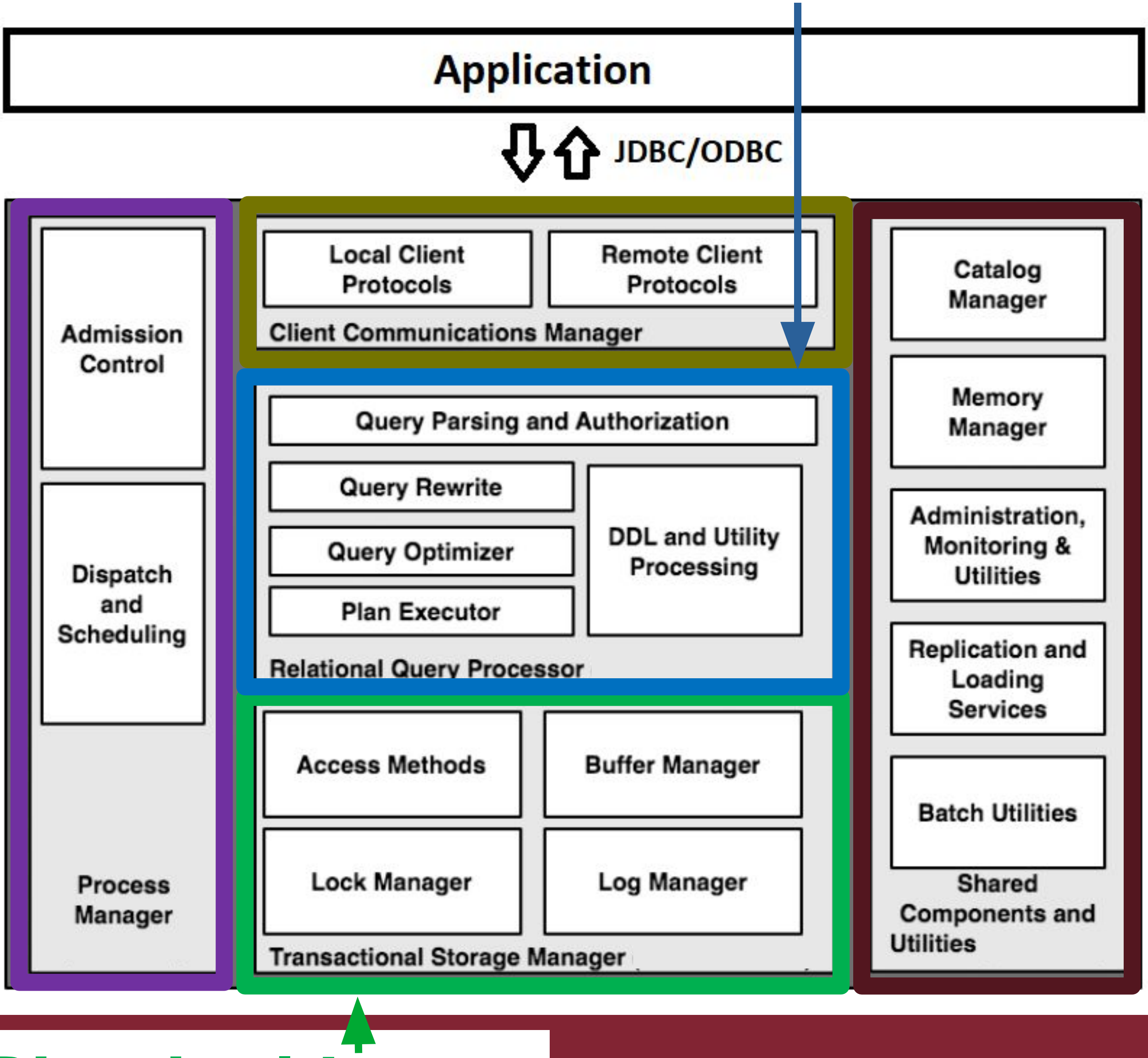
Physical Organization

SAPIENZA
UNIVERSITÀ DI ROMA

The content of the slides is authored by Prof. Giuseppe Pirrò, Sapienza University of Rome
Part of the material is taken from the book: Lemahieu, W., vanden Broucke, S., & Baesens, B. (2018). Principles of database management: The practical guide to storing, managing and analyzing big and small data. Cambridge University Press.

Architecture of a DBMS

Logical Layer

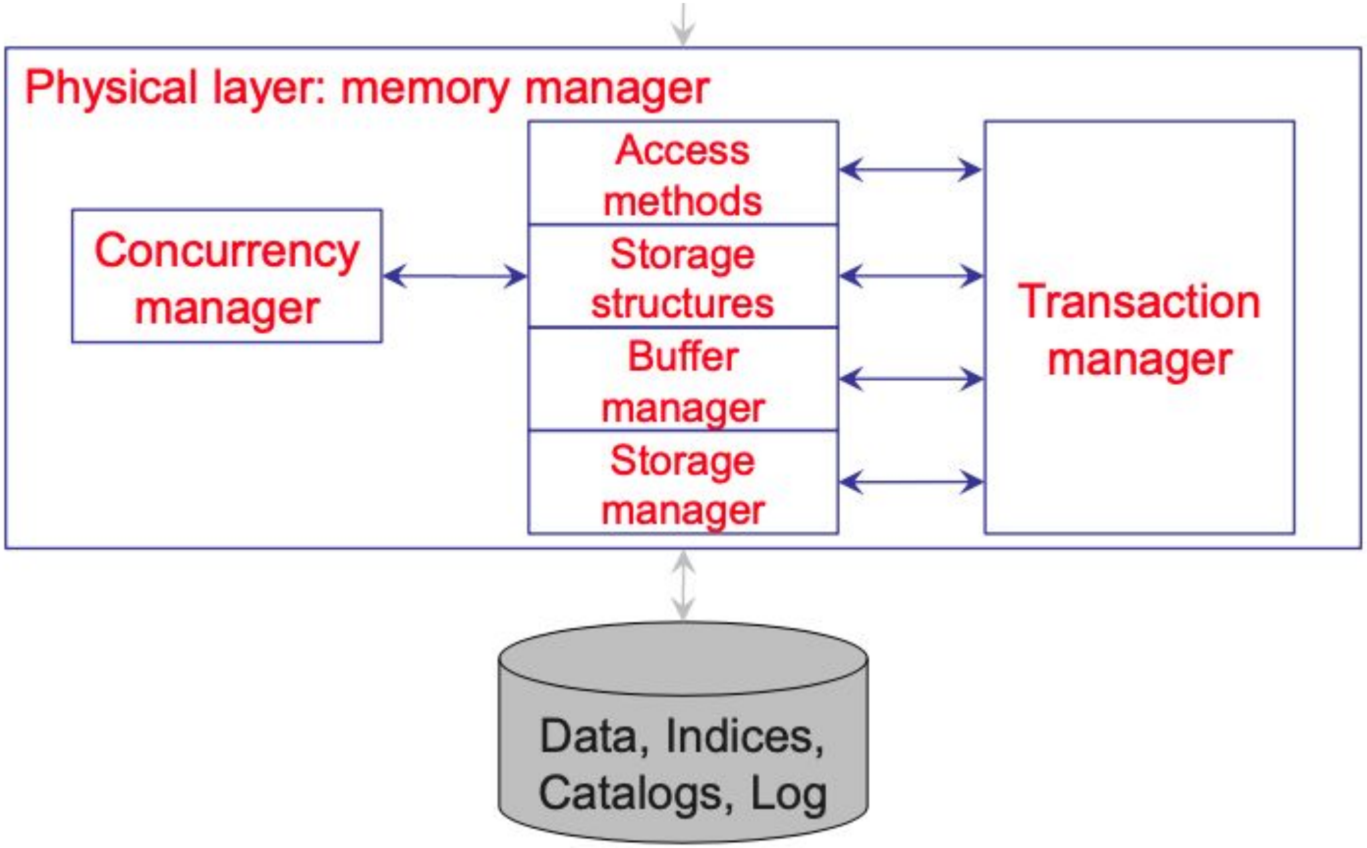


Physical Layer

Architecture of a DBMS

- **Physical Layer: Memory management**
  - Storage manager
  - Buffer
  - Access Methods
  - ….
- **Logical Layer: Query processor**
  - Query **parser**
    - Rewriting
  - Query **optimizer**
    - Query Plan generator
  - Query plan **executor**
  - …

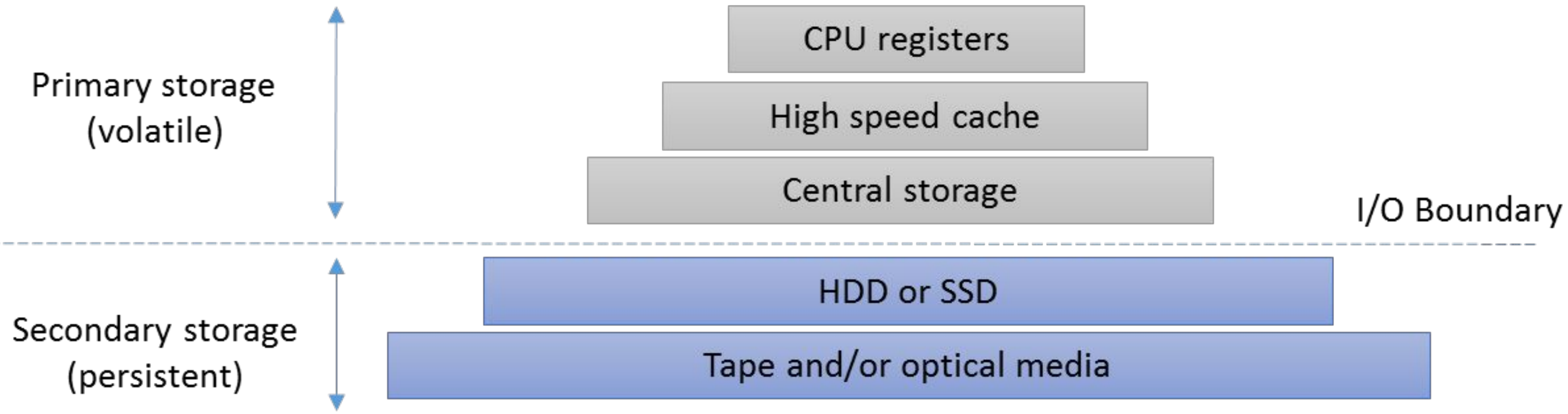
Database: Physical Layer



Database: Physical Layer

- **Storage Hardware and Physical Database Design**
  - The Storage Hierarchy
  - Internals of Hard Disk Drives
  - From Logical Concepts to Physical Constructs
- **Record Organization**
  - Pointers
  - Lists
- **File Organization**
  - Heap File Organization
  - Sequential File Organization
  - Random File Organization (Hashing)
  - Indexed Sequential File Organization
  - List Data Organization
  - Secondary Indexes and Inverted Files
  - B-trees and B^+-trees

The Storage Hierarchy



- Computer memory hierarchy
  - **Top**: high speed memory, expensive and limited in capacity
  - **Bottom**: slower memory, relatively cheap and larger in size

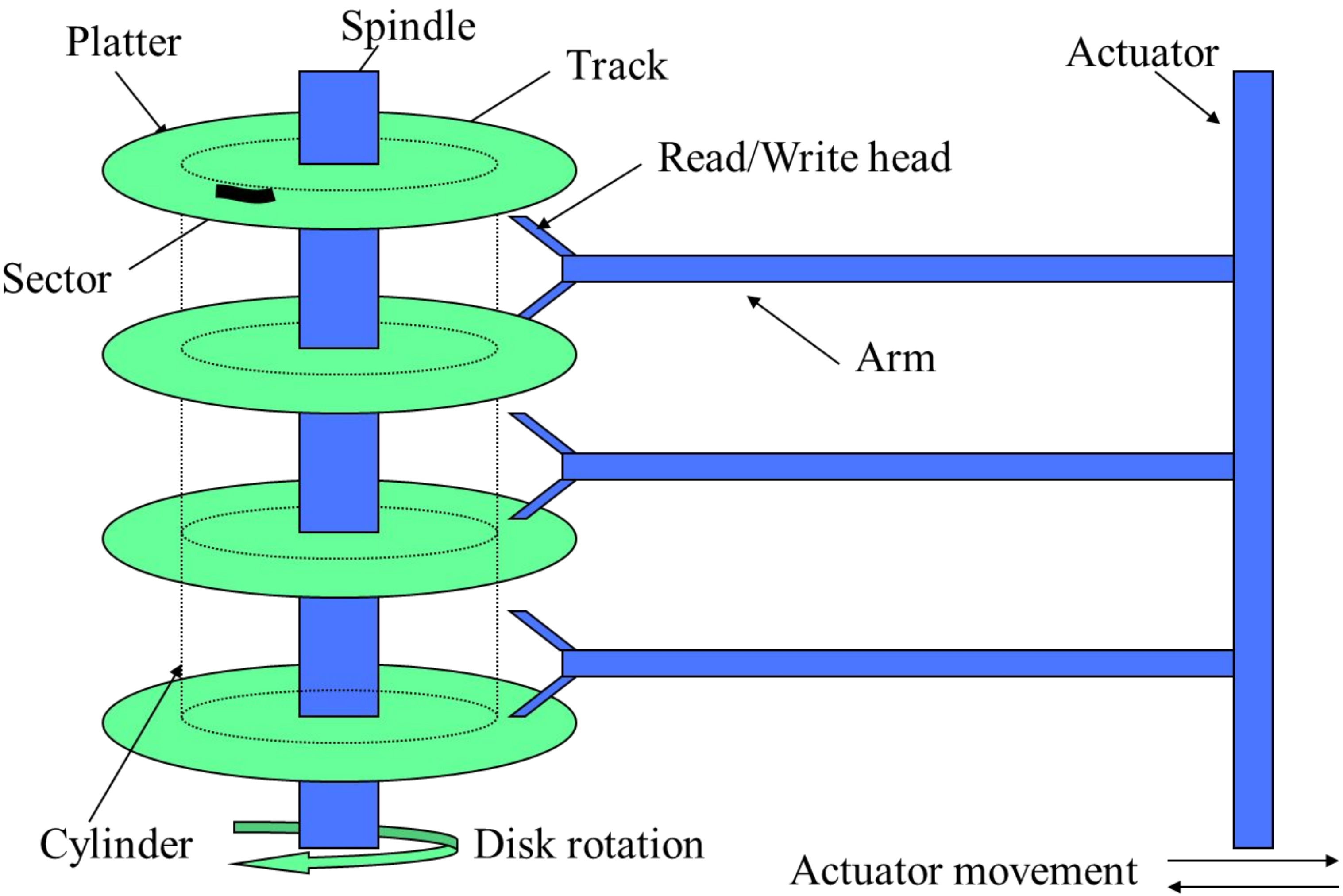
The Storage Hierarchy

- **Primary Storage** (a.k.a. volatile memory)
  - Central Processing Unit (**CPU**): executes mathematical and logical processor operations
  - **cache** memory operates at **nearly same speed** as CPU
  - **central** storage (a.k.a. internal memory, main memory):
    - consists of memory chips (also called Random Access Memory, or **RAM**) of which the performance is expressed in nanoseconds
  - **contains** database buffer and runtime **code** of the applications and **DBMS**
- **Secondary Storage**
  - **persistent** storage media
  - Hard disk drive (**HDD**) and solid-state drives (**SSD**) based on flash memory
  - **contains physical database files**

Internals of Hard Disk Drives

- Hard Disk Drive (HDD) stores data on circular platters, which are covered with magnetic particles
- A HDD also contains a hard disk controller
- HDDs are directly accessible storage devices (DASDs)
- **Platters** are secured on a **spindle**, which rotates at a **constant speed**
- Read/write **heads** can be positioned on arms, which are fixed to an actuator

Internals of Hard Disk Drives



Internals of Hard Disk Drives

- Reading from a block, or writing to a block implies
  - positioning the actuator (**seek time**)
    - **Seek time**: time to position the head on the correct track
  - wait until the desired sector has rotated under the read/write head (rotational delay, latency)
    - **Rotation delay**: **delay** waiting for the **rotation** of the **disk** to bring the required **disk** sector under the read-write head
- **Transfer time** depends on block size, density of magnetic particles and rotation speed of disks
- **Response time = service time + queueing time**
- **Service time = seek time + rotational delay + transfer time**

Internals of Hard Disk Drives

- Physical **file organization** can be **optimized** to **minimize** expected seek **time** and rotational delay
  - This can have impact on the DBMS performance
- We distinguish between:
  - $\mathbf{T_{rba}}$ refers to expected time to retrieve/write disk block independently of previous read/write:
    - $\mathbf{T_{rba}}$ = Seek + ROT/2 + BS/TR
  - $\mathbf{T_{sba}}$ refers to expected time to sequentially retrieve disk block with R/W head already in correct position:
    - T_{sba} = ROT/2 + BS/TR
  - **Note**:
  - Block size (BS)
  - Rotation time (ROT)
  - Transfer rate (TR)
  - RBA: Random block access
  - SBA: Sequential block access

Internals of Hard Disk Drives

• Average seek time	• 8.9 ms
• Spindle speed	• 7200 rpm
• Transfer rate (TR)	• 150 MBps
• Block size (BS)	• 4096 bytes

- ROT(ms)=(60*1000)/7200=8.33ms
- ROT/2=4.167ms

- $\mathbf{T_{rba}}$ = 8.9 ms + 4.167 ms + 0.026 ms = 13.093 ms
- $\mathbf{T_{sba}}$ = 4.167 ms + 0.026 ms = 4.193 ms

From Logical Concepts to Physical Constructs

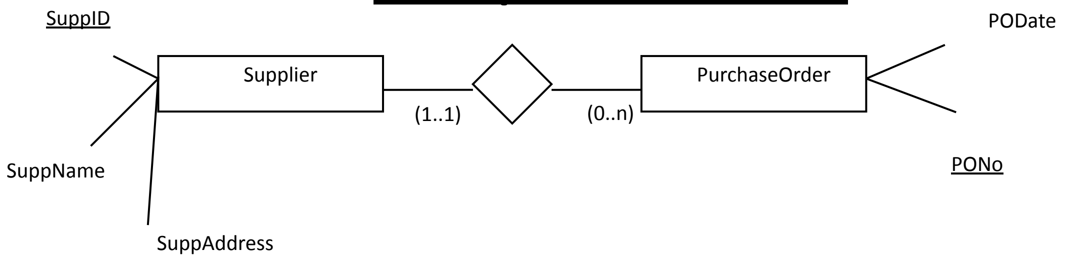
- **Desideratum:**
  - Organize physical files onto tracks and cylinders to minimize as much as possible seek time and (partially) rotation delay
  - Minimize the number of (random) block accesses
- **Physical database design**: translate logical data model into internal data model (a.k.a. physical data model)
- Trade-off between efficient update/retrieval and efficient use of storage space
- Focus on **physical organization of structured, relational data**

From Logical Concepts to Physical Constructs

Logical data model (general terminology)	Logical data model (**relational** setting)	Internal data model
Attribute type and attribute	Column name and (cell) value	Data item or field (bits or characters representing a specific value)
(Entity) record	Row or tuple	Stored record (collection of data items; represents all attributes of an entity)
(Entity) record type	Table or relation	Physical file or data set (similar entities, e.g., students)
Set of (entity) record types	Set of tables or relations	Physical database (collection of stored files)
Logical data structures	Foreign keys	Physical storage structures (logical interrelations like foreign keys)

From Logical Concepts to Physical Constructs

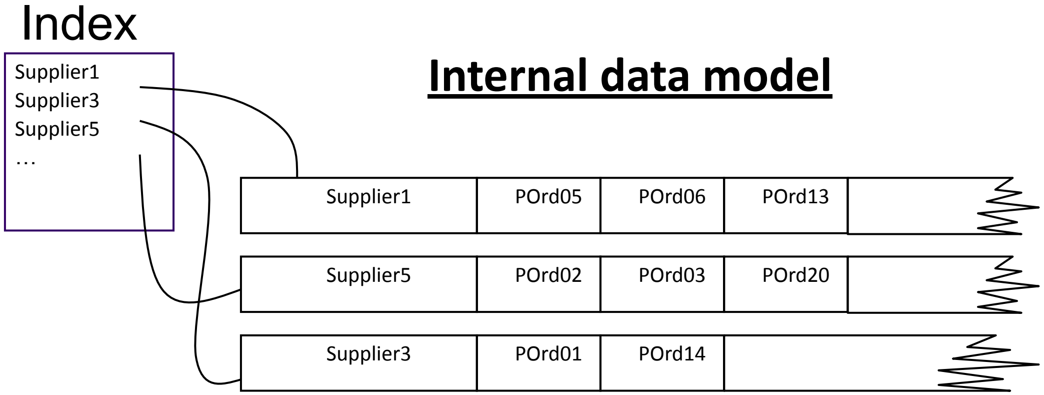
Conceptual data model



Logical data model

Supplier (SuppID, SuppName, SuppAddress)
PurchaseOrder (PONo, PODate, *SuppID*)

Internal data model


Index

Supplier1
Supplier3
Supplier5
…

Supplier1	POrd05	POrd06	POrd13
Supplier5	POrd02	POrd03	POrd20
Supplier3	POrd01	POrd14	

- This is just one possible organization
  - Assumes purchase orders are accessed via the supplier (hence that Index)

The Physical database

- At the **physical level**:
  - a **database** consists of a set of files
  - each **file** can be seen as a **collection of pages**, of fixed size (e.g., 4 KB)
  - each page stores several **records** (corresponding to logical tuples)
  - a record consists of **several fields**, with fixed and/or variable size, representing the tuple’s attributes

Primary file organization methods

- **Linear search**: each record in file is retrieved and assessed against search key
- **Hashing and indexing**: specify relationship between record's search key and physical location
    - Improve performance
    - Direct access to the location containing the records corresponding to the search key
- **Primary file organization methods**
    - Heap file
    - Sequential file
    - Random file
    - Indexed sequential file
    - List data
-

Heap File Organization

- Basic **primary file organization method**
  - New records inserted at end of file
  - No relationship between record’s attributes and physical location
  - **Only option** for record retrieval is linear search
  - For a file with NBLK blocks, it takes on average NBLK/2 to find record according to unique search key
  - Searching records according to non-unique search key requires scanning entire file

Sequential File Organization

- Records stored in **ascending/descending order** of search key
  - Efficient to retrieve records in the order determined by search key
    - Only require sequential block accesses
  - Records can still be retrieved by means of linear search
    - a **more effective stopping criterion** can be used, i.e., once first higher/lower key value than required one is found
-

Sequential File Organization

- **Binary search** technique can be used
- For unique search key K, with values K_i, algorithm to retrieve record with key value K_μ
  - Selection criterion: record with search key value K_μ
  - Set l = 1; h = number of blocks in file (suppose records are in ascending order of search key K)
  - Repeat until h ≥ l
    - i = (l + h) / 2, rounded to nearest integer
    - Retrieve block i and examine key values K_j of records in block i
  - if any $K_j = K_\mu$ □ record is found!
  - else if $K_\mu >$ all K_j □ continue with l = i+1
  - else if $K_\mu <$ all K_j □ continue with h = i-1
  - else record is not in file

Sequential File Organization

- Expected number of block accesses to retrieve record according to primary key by means of
  - **linear search**: NBLK/2 **sba**
  - **binary search**: $\log_2$(NBLK) **rba**
- Note:
  - **sba**:sequential block access
  - **rba**:random block access

Example: sequential file

Number of records (NR)	30000
Block size (BS)	2048 bytes
Records size (RS)	100 bytes

- **BF**=$\lfloor BS/RS \rfloor$ =$\lfloor 2048/100 \rfloor$=20
- **NBLK**=30000/20=1500
- If single record is retrieved according to primary key using linear search, expected number of required block accesses is 1500/2 = 750 sba
- If binary search is used, expected number of block accesses is $\log_2(1500) \approx 11$ rba
- Even if rba require more time than sba, their number is much lower than sba

Sequential File Organization

- **Updating** sequential file is more **cumbersome than** updating **heap** file
  - often done in batch
- Sequential files often combined with one or more indexes (see later, indexed sequential file organization)

Random File Organization (Hashing)

- Random file organization (hashing)
  - assumes direct relationship between value of search key and physical location
  - a record can be retrieved with one or at most few block accesses
- Hashing algorithm defines **key-to-address transformation**
  - the record physical address is calculated from its key
- Most effective when using primary key or other candidate key as search key

Random File Organization (Hashing)

- Hashing **cannot guarantee** that all keys are mapped to **different** hash **values**, hence bucket addresses
- **Collision occurs** when several records are assigned to same bucket (also called synonyms)
- If there are more synonyms than slots for a bucket, bucket is in overflow
  - additional block accesses needed to retrieve overflow records
- **Hashing algorithm** should **distribute keys** as **evenly** as possible over the respective bucket addresses

Random File Organization (Hashing)

- Popular hashing technique is division:
  - **address(key_i) = key_i mod M**
  - M is often a prime number (close to, but a bit larger than, the number of available addresses)
  - The remainder of the modulo division is the record address

Random File Organization (Hashing)

	Series 1			Series 2	
Key value	Division by 20	Division by 23	Key value	Division by 20	Division by 23
3000	00	10	3000	00	10
3001	01	11	3025	05	12
3002	02	12	3050	10	14
3003	03	13	3075	15	16
3004	04	14	3100	00	18
3005	05	15	3125	05	20
3006	06	16	3150	10	22
3007	07	17	3175	15	01
3008	08	18	3200	00	03
3009	09	19	3225	05	05
3010	10	20	3250	10	07
3011	11	21	3275	15	09
3012	12	22	3300	00	11
3013	13	00	3325	05	13
3014	14	01	3350	10	15
3015	15	02	3375	15	17
3016	16	03	3400	00	19
3017	17	04	3425	05	21
3018	18	05	3450	10	00
3019	19	06	3475	15	02

- first series: M=20 and M=23 result in a nearly uniform distribution of remainders
- second series: M=20 leads to a worse distribution (some buckets are empty while other have many records)

Random File Organization (Hashing)

- Efficiency of hashing algorithm measured by expected number of rba and sba
- Retrieving non-overflow record:
  - 1 rba to first block of bucket denoted by hashing algorithm, possibly followed by 1 or more sba
- Additional block accesses needed for **overflow records** depending on percentage of overflow records and overflow handling technique

Efficiency of Random File Organization

- Percentage of overflow records depends on hashing algorithm and key set
- Aim to achieve uniform distribution, spreading the set of records evenly over the set of available buckets
- Required number of buckets **NB**:
  - **NB = ⌈NR / (BS×LF)⌉**
    - with **NR** number of records, **BS** bucket size and **LF** loading factor
- **Trade-off**: larger bucket size implies smaller chance of overflow, but more additional overhead to retrieve non-overflow records

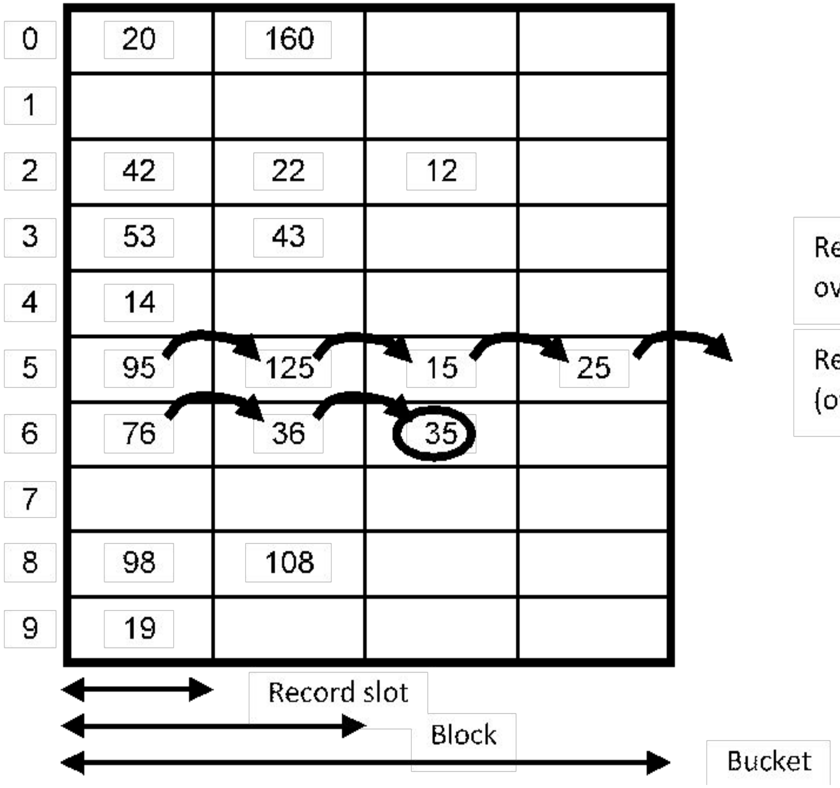
Efficiency of Random File Organization

1. Loading factor (**LF**) represents average number of records in bucket divided by bucket size
   - indicates how ‘full’ every bucket is on average
   - embodies trade-off between efficient use of storage capacity and retrieval performance
   - often set between 0.7 and 0.9
2. Different overflow handling techniques
   - overflow records stored either in primary area or in separate overflow area

Random File Organization (Hashing)

- **Open addressing**
  - overflow records stored in next free slot after full bucket where a record would normally have been stored

Hash = key mod 10
LF = 0,45
Bucket size = 4
BF = 2
Open addressing



0	20	160		
1				
2	42	22	12	
3	53	43		
4	14			
5	95	125	15	25
6	76	36	35	
7				
8	98	108		
9	19			

Required block accesses for retrieving non-overflow records: 1 rba or (1 rba + 1 sba)

Required block accesses for retrieving (overflow) record with ID 35: (1 rba + 3 sba)

Indexed Sequential File Organization

- **Random file** organization is efficient to retrieve **individual records** by search key value
- **Sequential file** organization is efficient if **many records** are to be retrieved in certain order
- **Indexed sequential file**
  - reconciles both concerns
  - combines **sequential** file organization with one or more **indexes**

Indexed Sequential File Organization

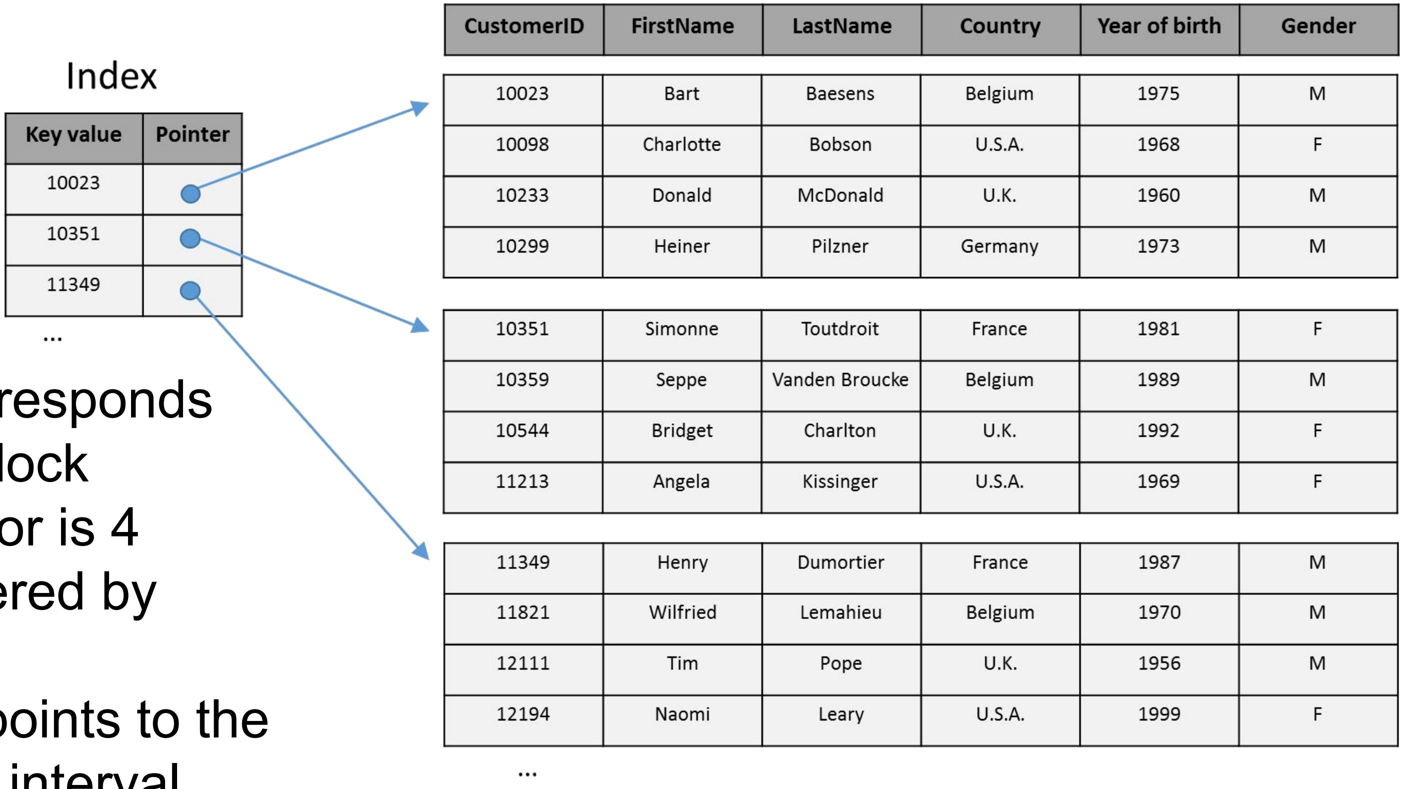
- File is divided into intervals or partitions
- Each interval is represented by **index entry** containing
  - search key value of first record in interval
  - pointer to physical position of first record in interval
- Pointer can be:
  - block pointer (referring to physical block address)
  - record pointer (consisting of combination of block address and record id or offset within block)
- The index is sequential file, ordered according to search key
  - Entry: <search key value, block pointer or record pointer>
- Search key can be atomic (e.g., a CustomerID) or composite (e.g. Year of Birth and Gender)
-

Indexed Sequential File Organization

- **Dense index** has index entry for every possible value of search key
- **Sparse index** has index entry for only some of search key values
  - Each entry refers to a group of records
- **Dense indexes are generally faster**, but require **more storage** space and are more complex to maintain than sparse indexes
- **Note**: index file occupies fewer disk blocks than data file and can be searched much quicker

Primary Index

- Data file is **ordered on unique key**
- **The** index is defined over this unique search key



Index

Key value	Pointer
10023	●
10351	●
11349	●

…

CustomerID	FirstName	LastName	Country	Year of birth	Gender
10023	Bart	Baesens	Belgium	1975	M
10098	Charlotte	Bobson	U.S.A.	1968	F
10233	Donald	McDonald	U.K.	1960	M
10299	Heiner	Pilzner	Germany	1973	M
10351	Simonne	Toutdroit	France	1981	F
10359	Seppe	Vanden Broucke	Belgium	1989	M
10544	Bridget	Charlton	U.K.	1992	F
11213	Angela	Kissinger	U.S.A.	1969	F
11349	Henry	Dumortier	France	1987	M
11821	Wilfried	Lemahieu	Belgium	1970	M
12111	Tim	Pope	U.K.	1956	M
12194	Naomi	Leary	U.S.A.	1999	F

…

- Each interval corresponds to a single disk block
  - Blocking factor is 4
- Records are ordered by primary key
- The key pointer points to the first record in the interval

It is a sparse index

Primary Index

• Linear search	• NBLK sba
• Binary search	• $\log_2$(NBLK) rba
• Index based search	• $\log_2$(NBLKI) + 1 rba, with NBLKI << NBLK

Note: **NBLKI** represents number of **blocks** in **index**!

Primary Index

Number of records (NR)	30000
Block size (BS)	2048 bytes
Records size (RS)	100 bytes
Index entry	15 bytes

- Blocking factor of index = $\lfloor 2048/15 \rfloor = 136$
- **NBLKI** = $\lceil 1500/136 \rceil$ = 12 blocks
- Binary search on index requires $\log_2(12) + 1 \approx 5$ rba (compare to 750 sba and 11 rba!)

Tree-Structured Indexes

Trees

- A **tree** consists of nodes and edges with following properties:
  - **one** *root* node
  - **every node**, except for root, has exactly **one *parent*** *node*
  - every node has 0, 1 or more *children* or *child nodes*
  - nodes with **same parent** node are called ***siblings***
  - all children, children-of-children, etc. of a node are the node's *descendants*

Trees

- A **tree** consists of nodes and edges with following properties:
  - a node without children is a *leaf node*
  - tree structure consisting of non-root node and all its descendants is a *subtree*
  - nodes are distributed in *levels*, representing distance from root
  - tree where all leaf nodes are at the same level is called *balanced,* otherwise *unbalanced*

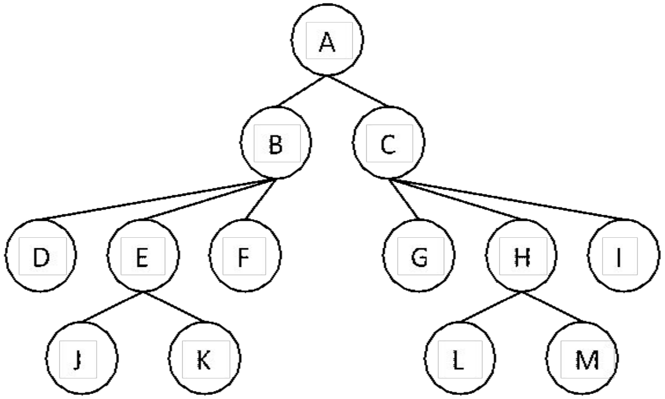
Trees

- **Tree data structures** can be used to provide
  - **physical representation** of logical hierarchy or tree structure (e.g. hierarchy of employees)
  - **purely physical index structure** to speed up search and retrieval of records by navigating interconnected nodes of tree (search tree)
    - **B-trees** and **B$^+$-trees**

Trees

- **Trees** can be implemented by means of **physical contiguity**
  - nodes stored in ‘top-down-left-right’ sequence:
    - first root, then root’s first child, then child’s first child etc.
  - if node has no more children, its next sibling (from left to right) is stored
  - if node has no more siblings, its parent’s next sibling is stored
  - each nodes’ level needs to be included explicitly

Trees



A	0	B	1	D	2	E	2	J	3	K	3	F	2

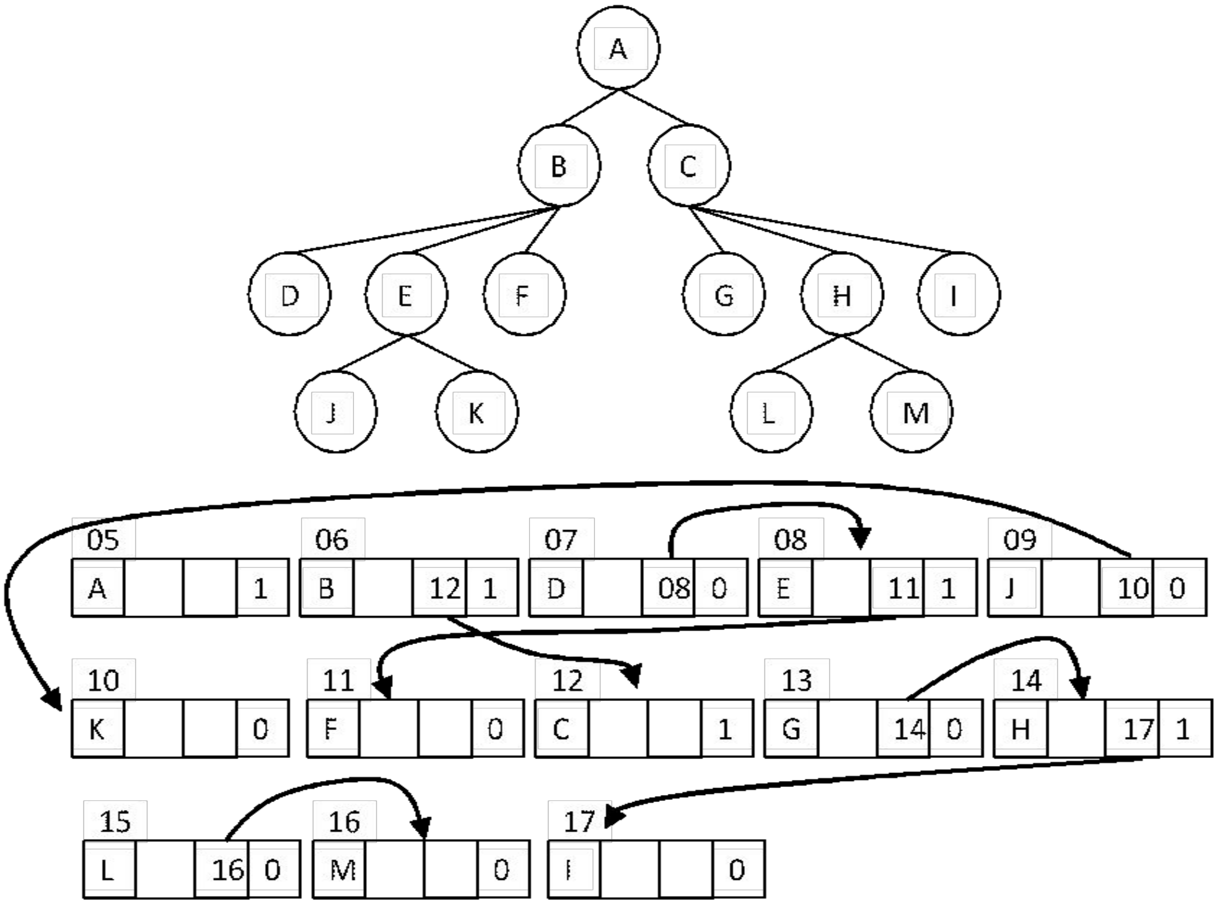
C	1	G	2	H	2	L	3	M	3	I	2

- Letters represent record keys; numbers denote level in tree for physical records
- Can only navigate in **sequential** way

Lists and Trees

- **Trees** can be **implemented** by means of **Linked Lists**
  - physical contiguity complemented with pointers
  - each node has pointer to its next sibling, if it exists
  - both parent-child and sibling-sibling navigation supported, respectively by accessing physically subsequent record and following pointer
  - single bit often added indicating whether node is a leaf node (bit = 0) or not (bit = 1)

Lists and Trees



- Many variations possible!

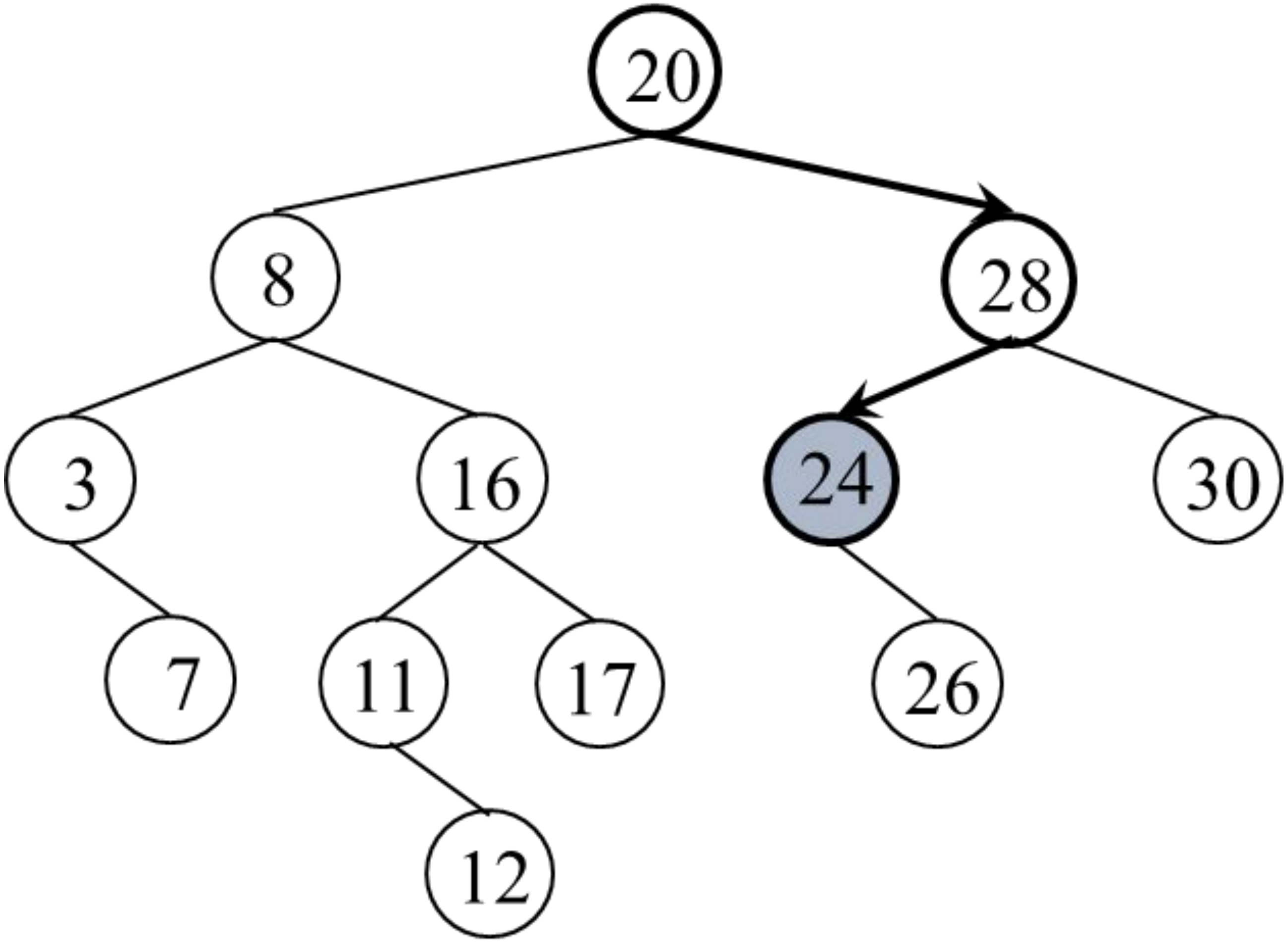
Binary Search Trees

- Binary search tree is a physical tree structure, where each node has at most 2 children
- Each tree node contains a search key value and **maximum 2 pointers to children**
- Both children are root nodes of subtrees, with one subtree only containing key values that are lower than the key value in the original node, and the other subtree only containing key values that are higher

Binary Search Trees

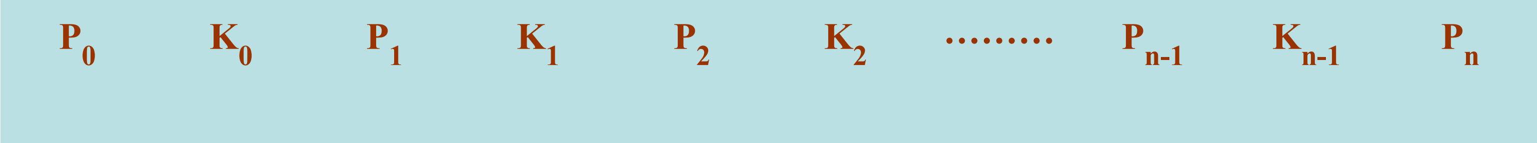
- Search efficiency improved by ‘skipping’ half of the search key values with every step (~ binary search)
- Suppose search key K is used with values K_i
  - to find node with search key value K_μ, key value K_i in root node is compared to K_μ
  - if $K_i = K_\mu$, search key is found
  - if $K_i > K_\mu$, pointer to root of the ‘left’ subtree is followed
  - if $K_i < K_\mu$, pointer to root of the ‘right’ subtree is followed
- Apply recursively

Binary Search Trees



Multiway search trees

- M-way (m-ary) search tree is generalization of binary search tree. Binary search tree is a 2-way search tree.



P_0 K_0 P_1 K_1 P_2 K_2 ……… P_{n-1} K_{n-1} P_n

- In the structure, P_0, P_1, P_2,…, P_n are pointers to the node's sub-trees and K_0, K_1, K_2,…, K_{n-1} are the key values of the node. **All the key values are stored in ascending order**. That is, $K_i < K_{i+1}$ for $0 \leq i \leq n-2$
  - Node has n key values, and n+1 pointers, n+1 <= m
  - All key values in subtree of P_i are less than K_i
  - All key values in subtree of P_{i+1} are bigger than K_i

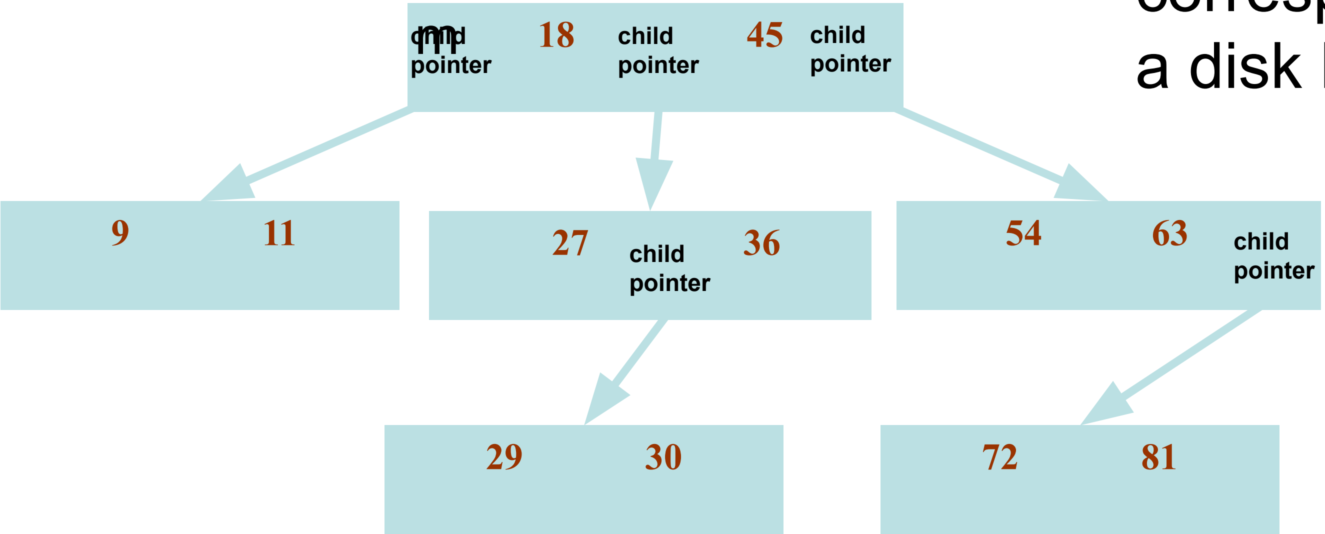
Multiway search trees

- A node can have anywhere from 1 to (m-1) values, and the number of sub-trees may vary from 0 (for a leaf node) to 1 + *i*, where *i* is the number of key values in the node.
- **m** is a *fixed upper limit* that **defines how many key values can be stored in the node**.
- Out degree of an m-way search tree is at most m.

Multiway search trees: example

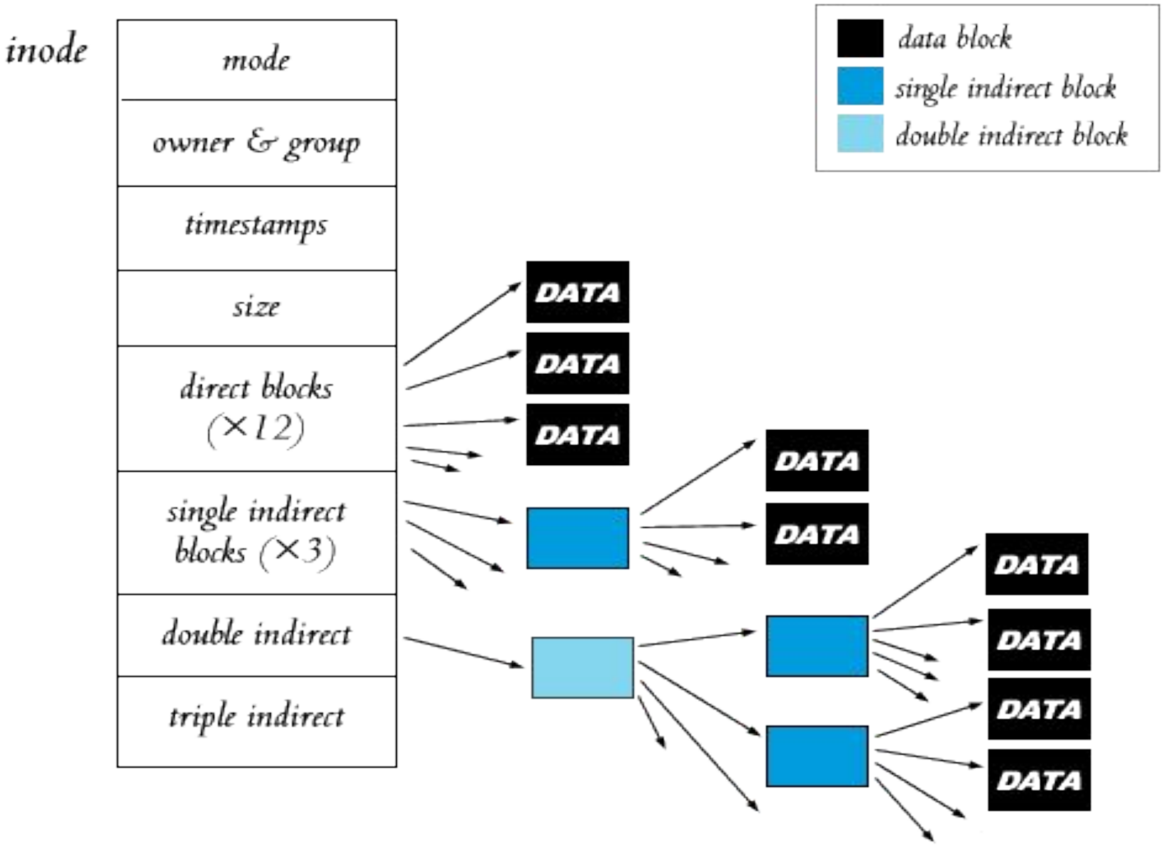
$k_1<k_2<k_3....<k_m$

Each node corresponds to a disk block



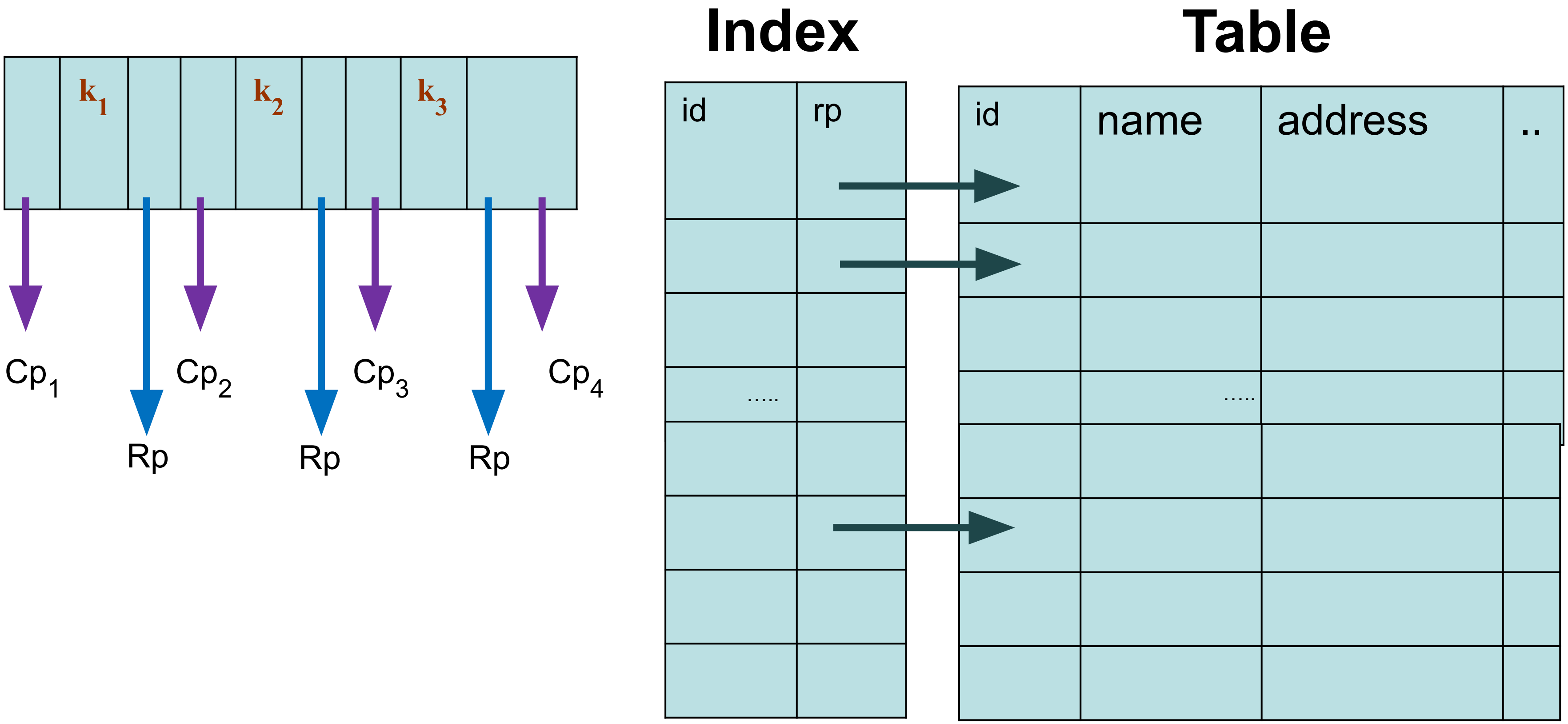
An m-way search tree has m-1 keys

Multiway search trees: example



Unix file system

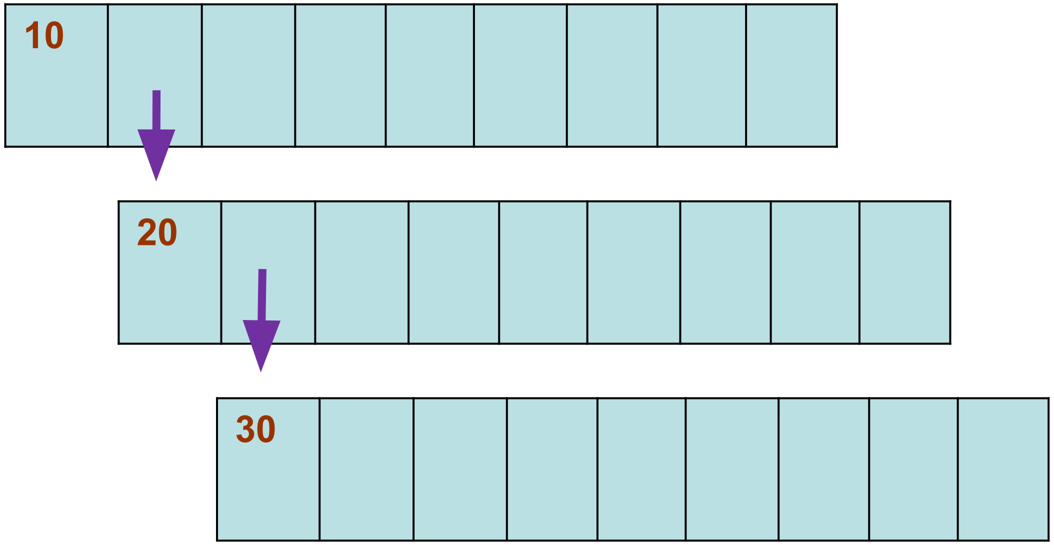
Multiway search trees to create indexes



- Besides child pointers (Cp_i) we consider record pointers (Rp) after every key

Problems with multiway search trees

- Consider a 10-way search tree (9 keys)
- **Keys to be inserted**: 10,20,30



- Wrong way of inserting!
- We should fill up the first node
- There is no control in the way keys are inserted
- With m keys we can have a tree of height m
  - Cost similar to linear search

B-tree

- A B-tree of order m is a **self-balanced** m-way search tree with following properties

1. Every node has at most **m** children
2. Every node in the B-tree except the root node and leaf nodes have at least (minimum) *m*/2 children
   - You must fill a node at least a half
   - With degree 10, we must have 5 children
   - After 5 children we can create a new node
   - This controls the height (remember m-way search trees?)

B-tree

- A B-tree of order m is a **self-balanced** m-way search tree with following properties

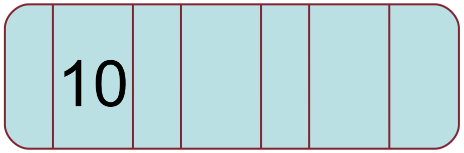
3. The **root node** has **at least two children** if it is not a terminal (leaf) node
4. All **leaf nodes** are at the **same level**
5. The **creation** process is **bottom-up**

B-tree

- A B-tree is a tree-structured index
  - each node corresponds to a disk block and nodes **are kept between half full and full** to cater for a certain dynamism
- Every node contains:
  - a set of search key values
  - a set of tree pointers that refer to child nodes
  - a set of data pointers that refer to data records, or blocks with data records, that correspond to search key values
- Data **records** are stored **separately** and are no part of B-tree

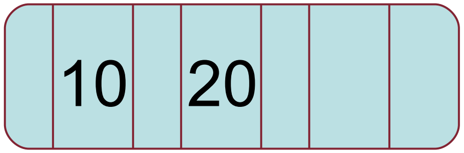
B-Tree Example with m = 4

Keys: 10, 20, 40, 50



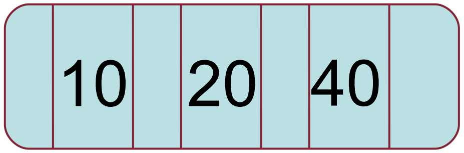
B-Tree Example with m = 4

Keys: 10, 20, 40, 50



B-Tree Example with m = 4

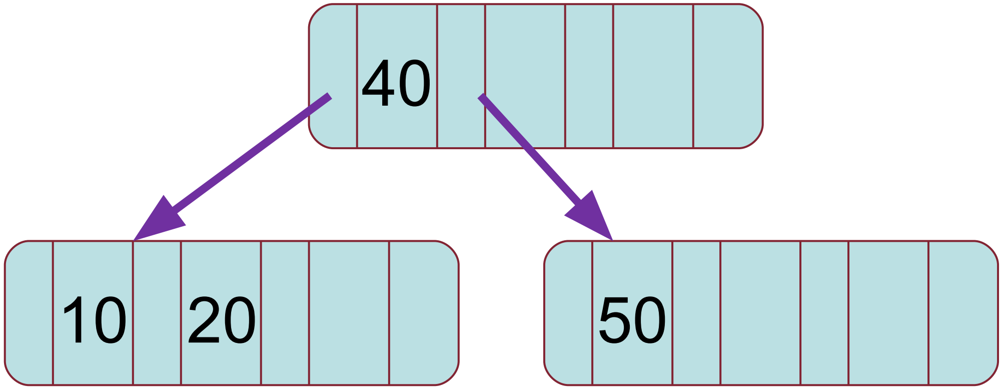
Keys: 10, 20, 40, 50



- There is no space to insert 50 (only 3 keys can be inserted)

B-Tree Example with m = 4

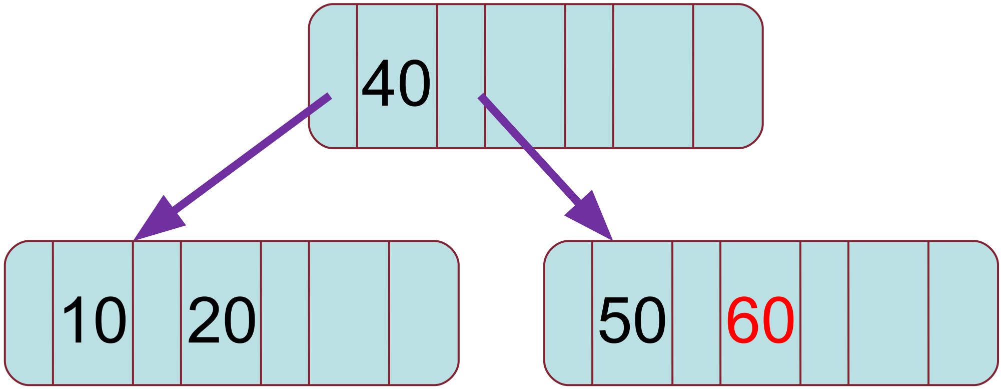
Keys: 10, 20, 40, 50



- Split the node
  - Keep 2 keys on the left
  - Send 40 to the root
- The tree is growing from the leaves

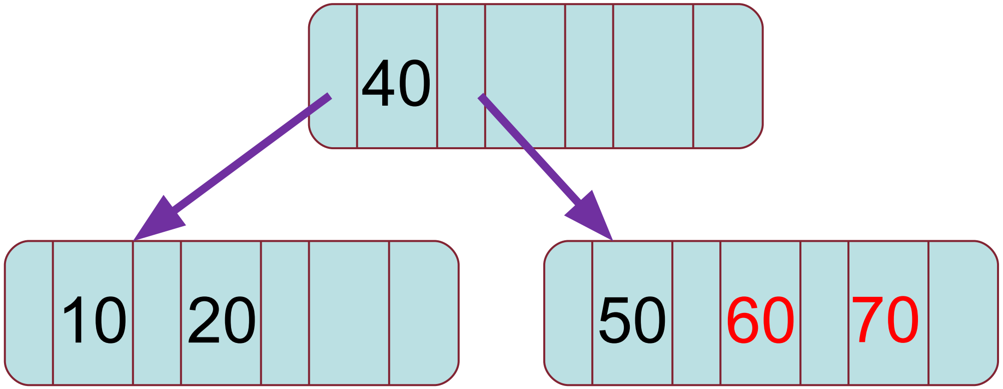
B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60



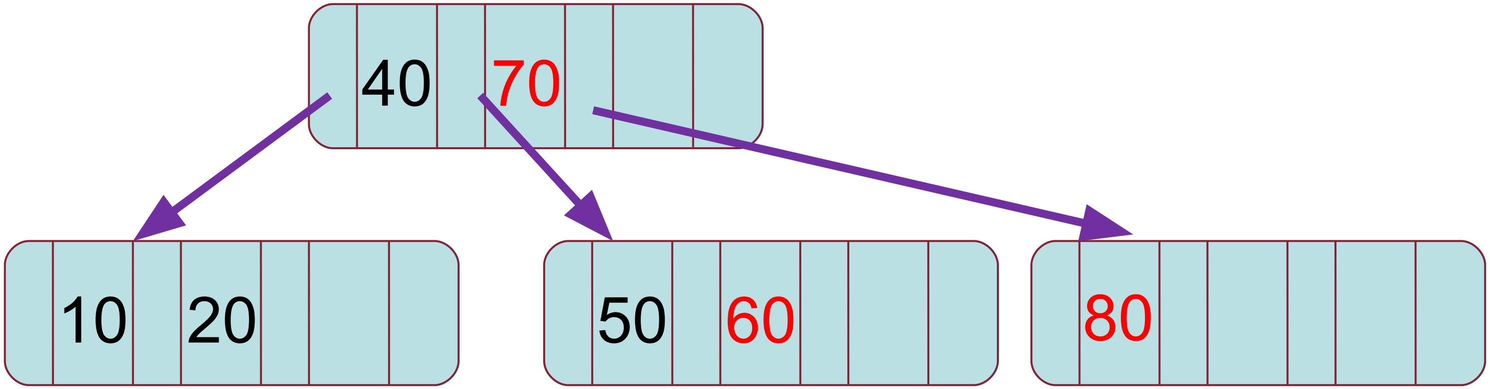
B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70



B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70, 80

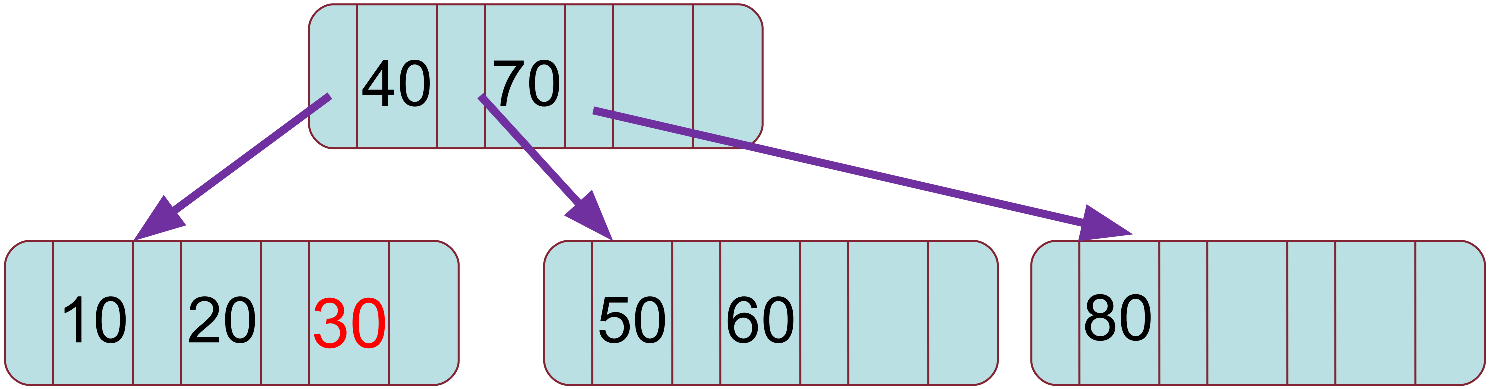


50, 60, 70, 80

- Split the node
- Move 70 to the root

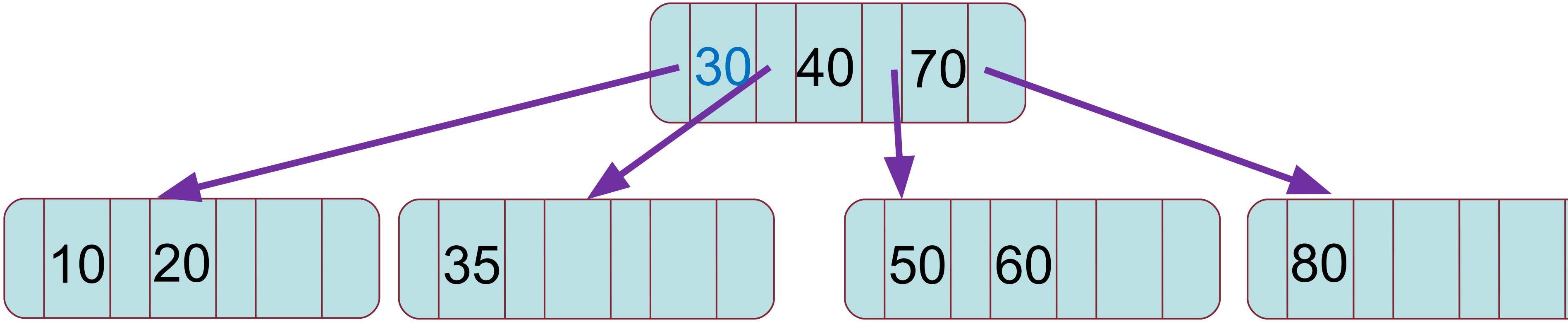
B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70, 80, 30



B-Tree Example with m = 4

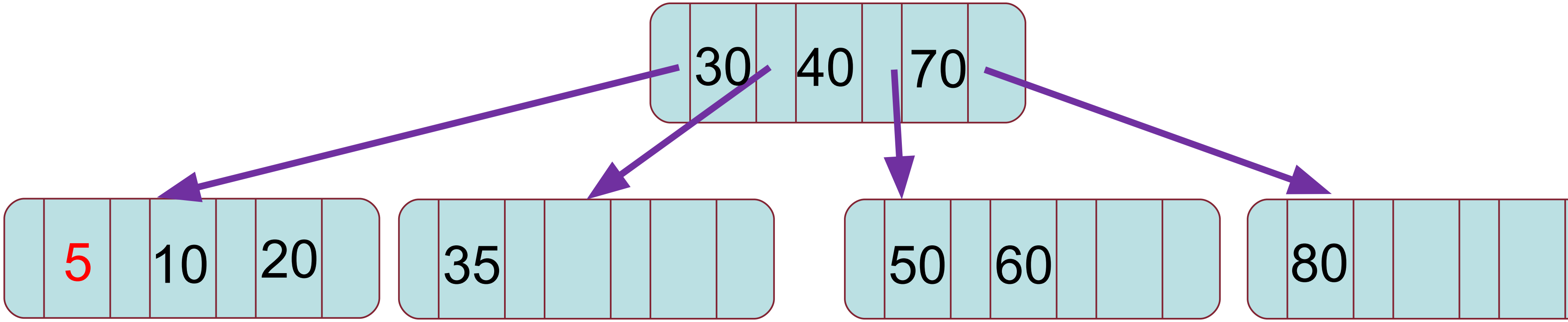
Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35



10, 20,30,35

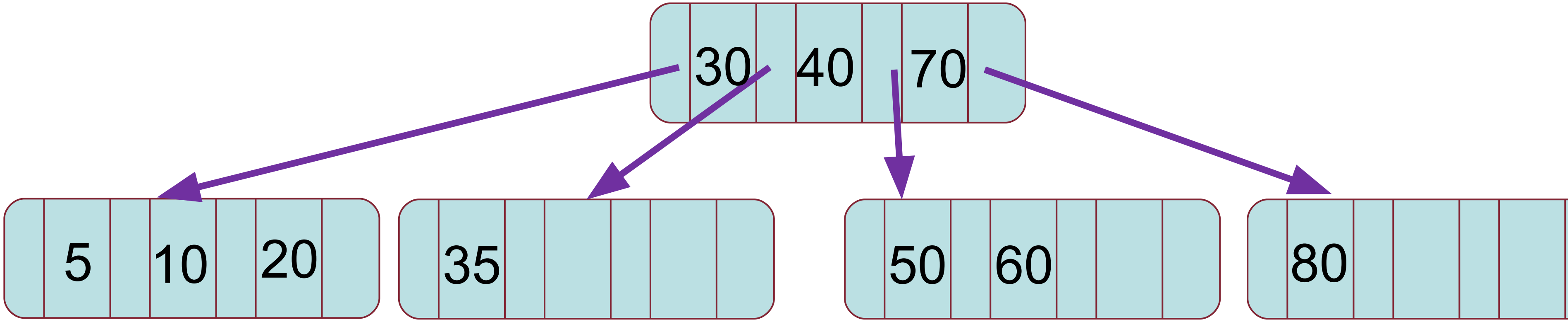
B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5



B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5,15

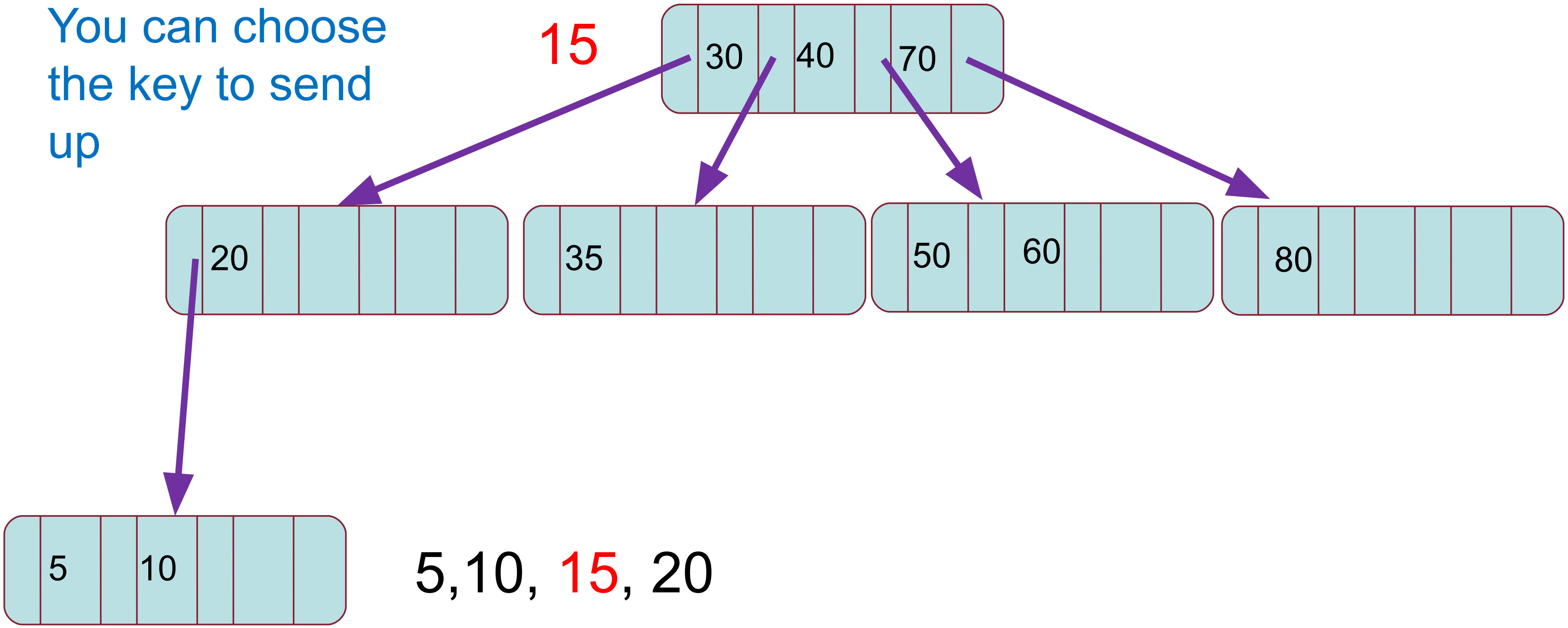


5,10, 15, 20

B-Tree Example with m = 4

Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5,15

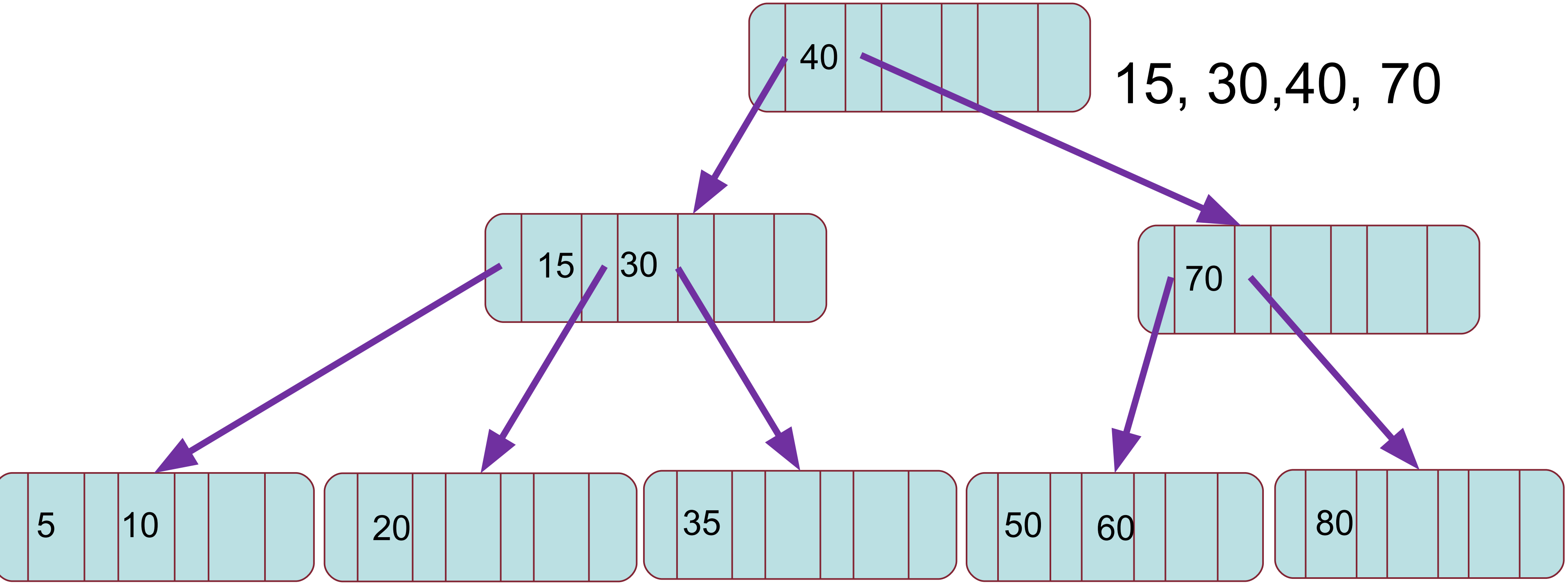
You can choose the key to send up



5,10, 15, 20

B-Tree Example with m = 4

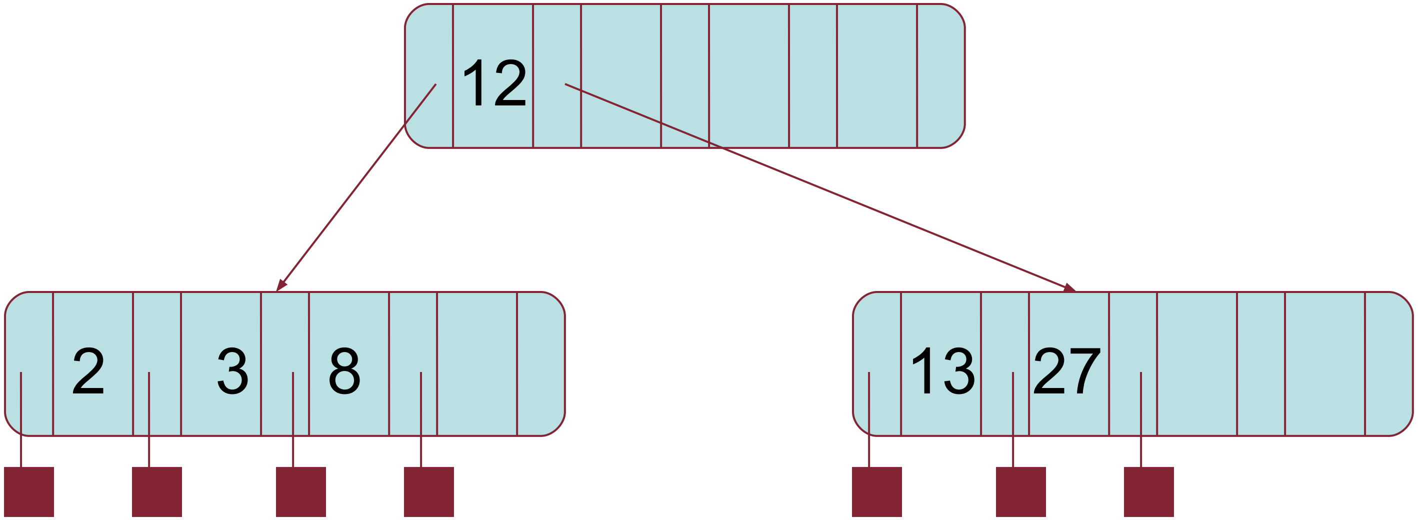
Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5,15



15, 30,40, 70

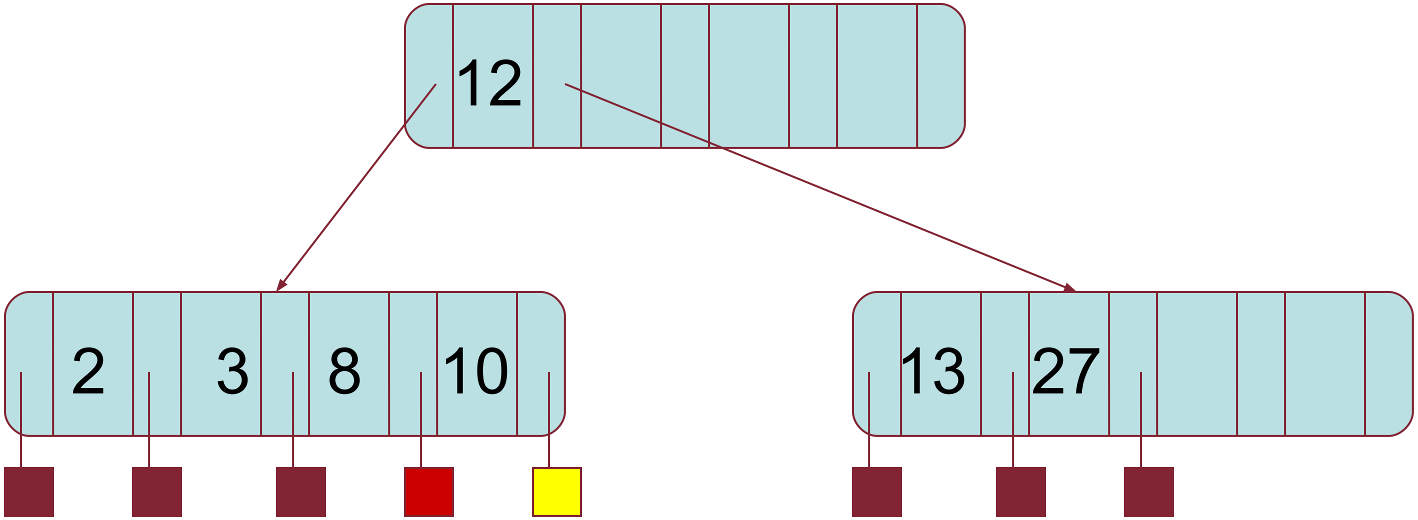
Idea similar to multilevel indexes

B-Tree Example with m=5



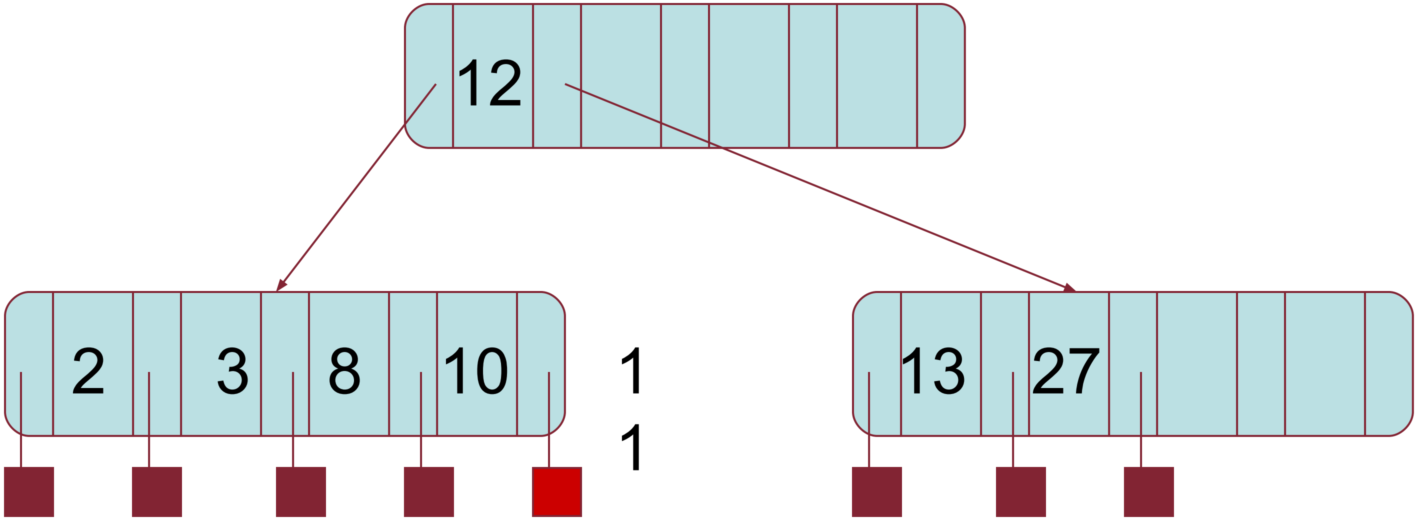
- The root has between 2 and *m* children.
- Each non-root internal node has between ⌈*m*/2⌉ and *m* children.
- All external nodes are at the same level. (External nodes are actually represented by null pointers in implementations.)

Insert 10



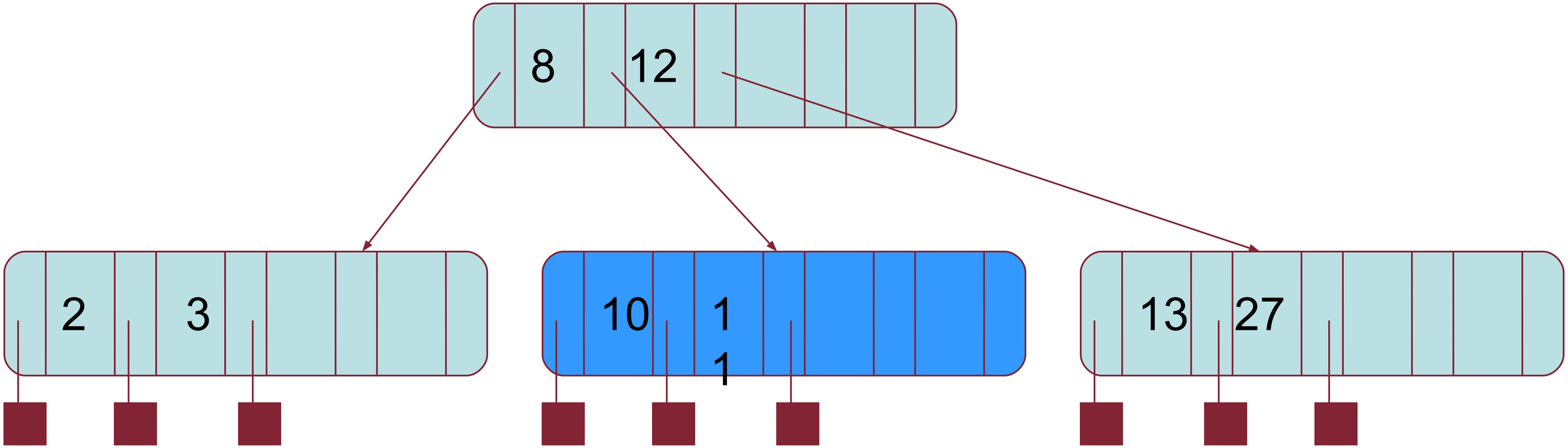
- We find the location for 10 by following a path from the root using the stored key values to guide the search.
- The search falls out the tree at the 4th child of the 1st child of the root.
- The 1st child of the root has room for the new element, so we store it there.

Insert 11



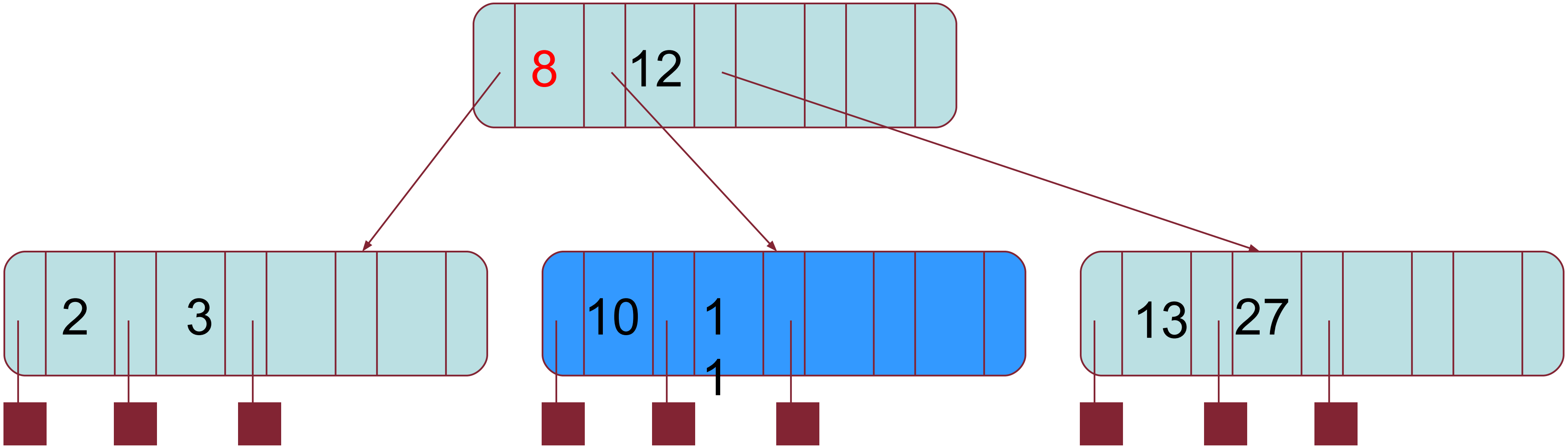
- We fall out of the tree at the child to the right of key 10.
- But there is no more room in the left child of the root to hold 11.
- Therefore, we must *split* this node...

Insert 11 (Continued)



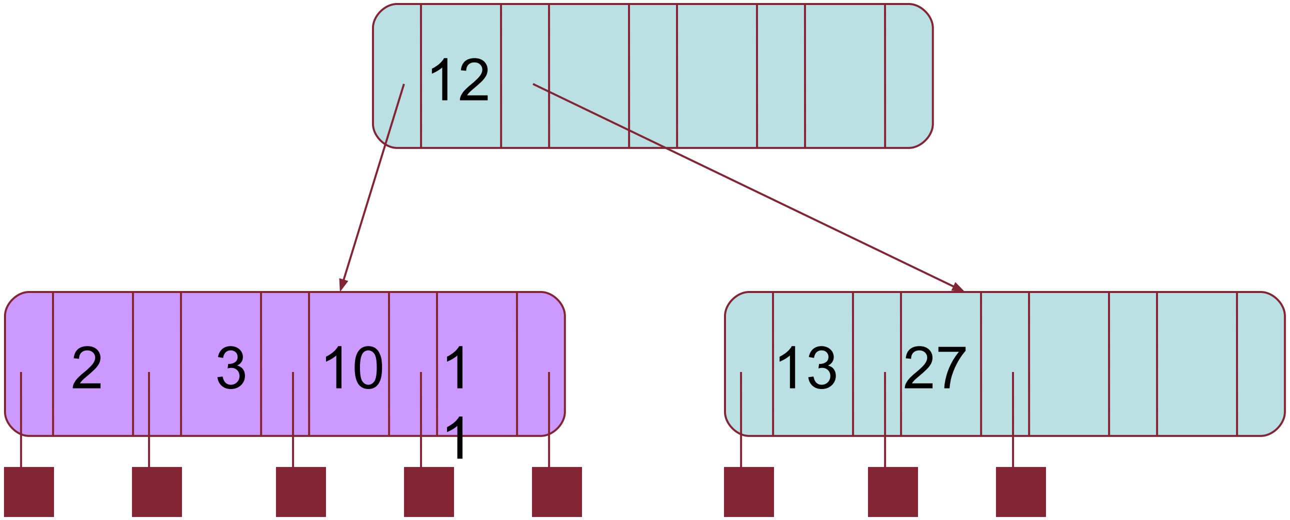
- The m + 1 children are divided evenly between the old and new nodes.
- The parent gets one new child. (If the parent become overfull, then it, too, will have to be split).

Remove 8



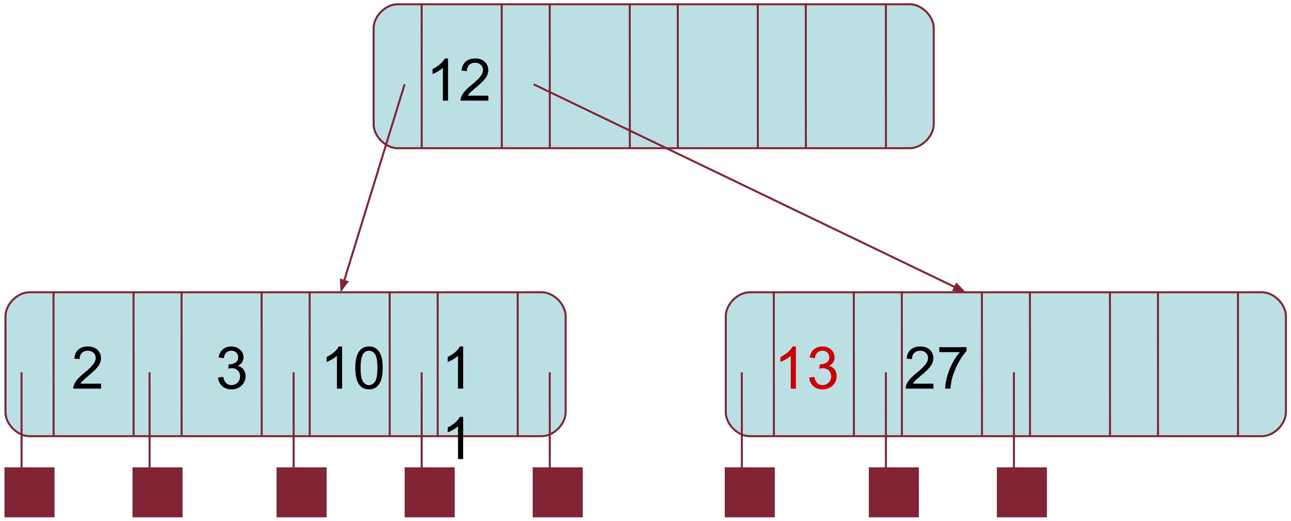
- Removing 8 might force us to move another key up from one of the children. It could either be the 3 from the 1st child or the 10 from the second child.
- However, neither child has more than the minimum number of children (3), so the two nodes will have to be merged. Nothing moves up.

Remove 8 (Continued)



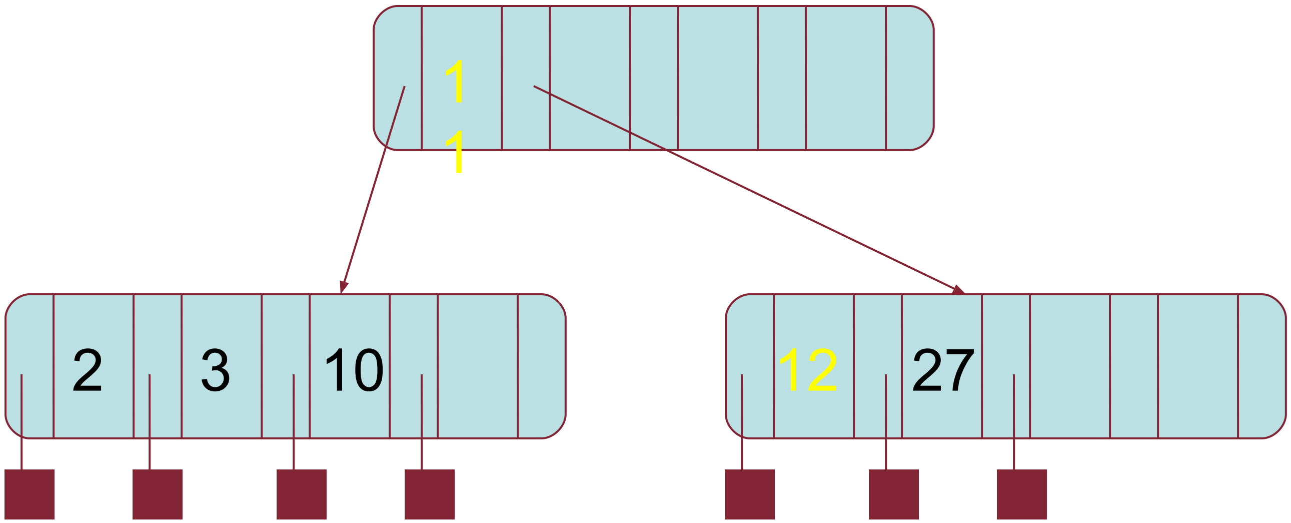
- The root contains one fewer key and has one fewer child.

Remove 13



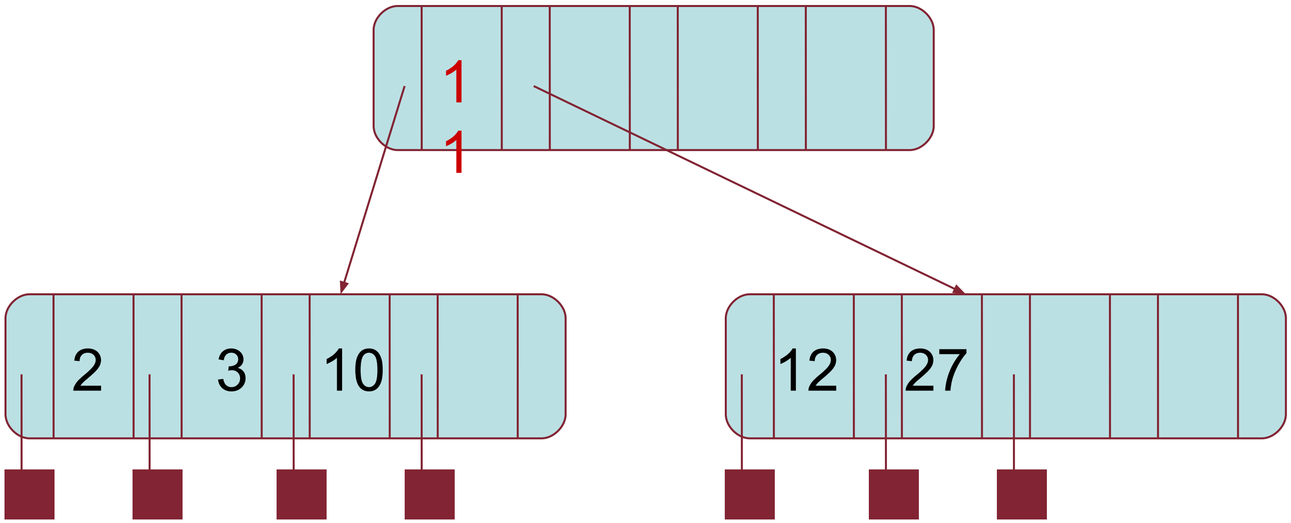
- Removing 13 would cause the node containing it to become underfull.
- To fix this, we try to reassign one key from a sibling that has spares.

Remove 13 (Cont)



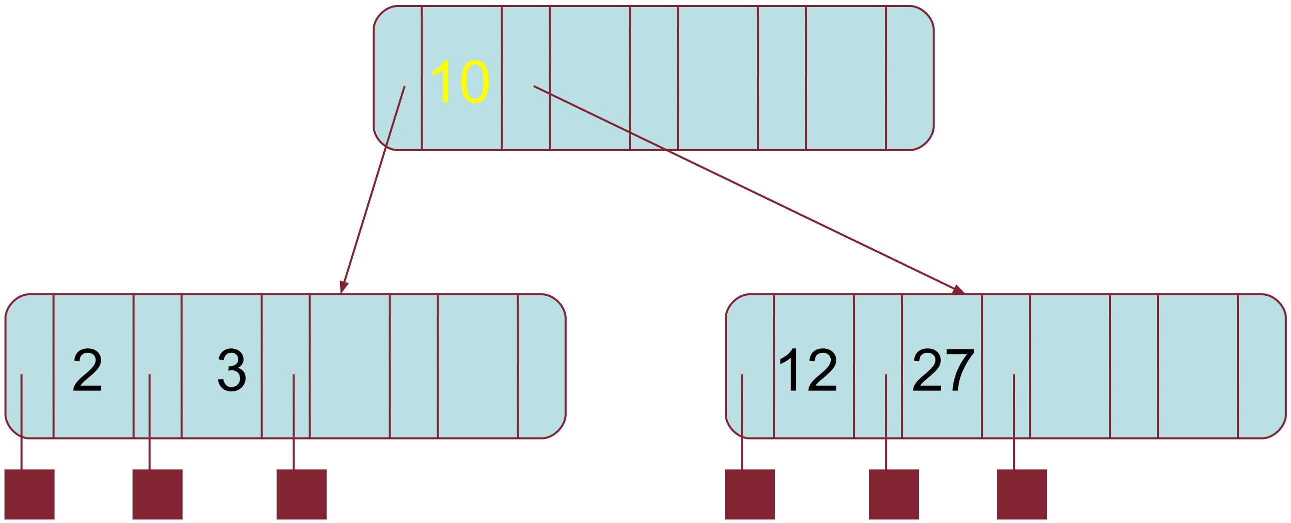
- The 13 is replaced by the parent’s key 12.
- The parent’s key 12 is replaced by the spare key 11 from the left sibling.
- The sibling has one fewer element.

Remove 11

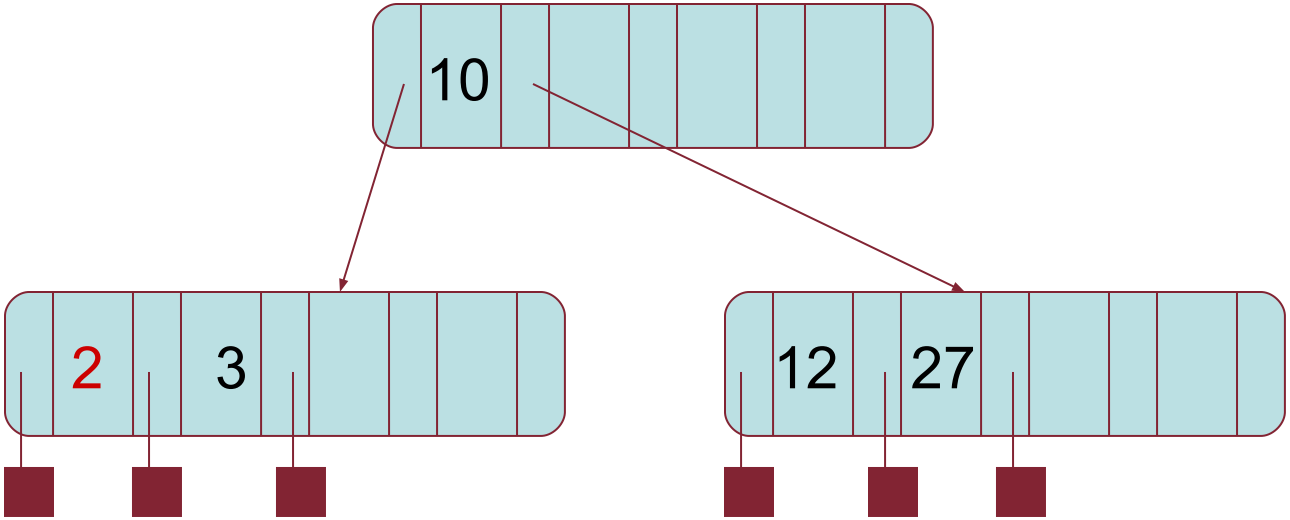


- 11 is in a non-leaf, so replace it by the value immediately preceding: 10.
- 10 is at leaf, and this node has spares, so just delete it there.

Remove 11 (Cont)

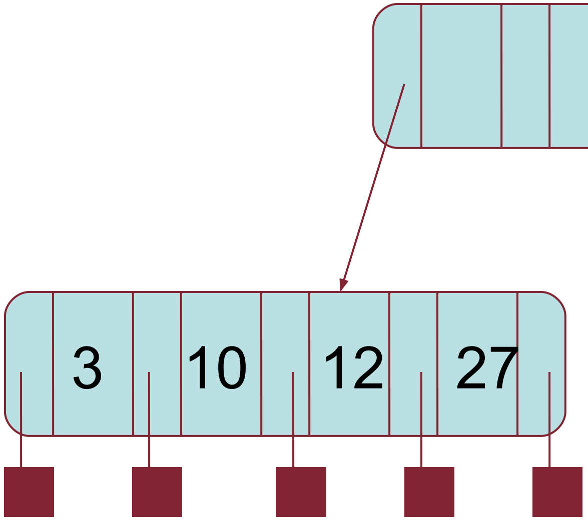


Remove 2

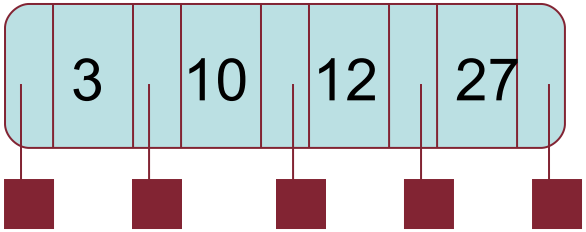


- Although 2 is at leaf level, removing it leads to an underfull node.
- The node has no left sibling. It does have a right sibling, but that node is at its minimum occupancy already.
- Therefore, the node must be merged with its right sibling.

Remove 2 (Cont)

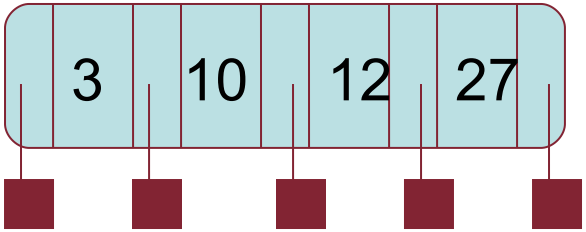


Remove 2 (Cont)



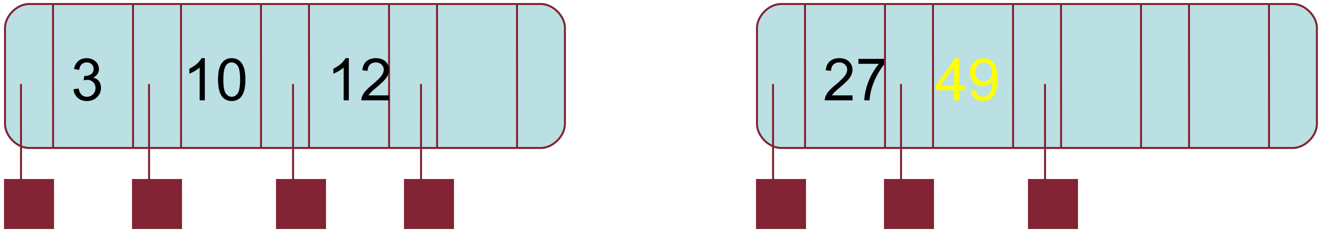
The new B-tree has only one node, the root.

Insert 49



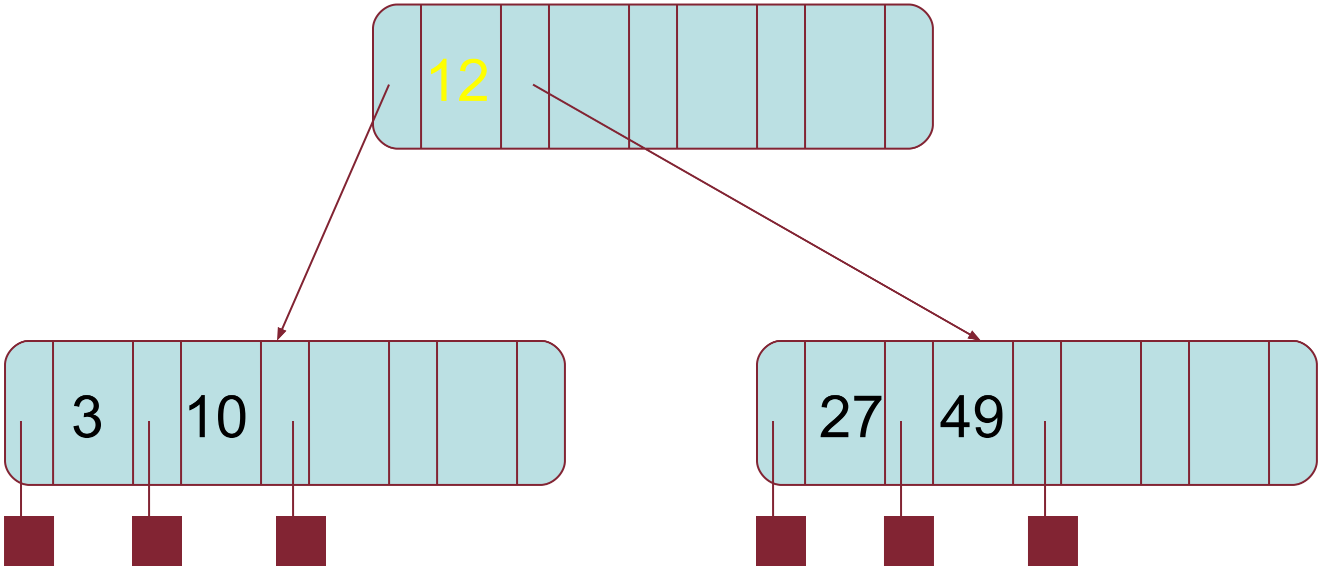
Let’s put an element into this B-tree.

Insert 49 (Cont)



- Adding this key make the node overfull, so it must be split into two.
- But this node was the root.
- We must construct a new root and make these its children.

Insert 49 (Cont)



- The middle key (12) is moved up into the root.
- The result is a B-tree with one more level.

B-tree search

- B-tree is searched recursively starting from the root
  - if desired key value X is found in a node (say $K_i = X$), then corresponding data record(s) can be accessed by following Pd_i
  - if desired value is not found in node, the subtree pointer P_i to be followed is the one corresponding to the smallest value of i for which $X < K_{i+1}$
    If $X >$ all K_i then the tree pointer P_{i+1} is followed
- **Note**: fan-out and search efficiency is much higher than with binary search!

B-tree search

- Every node is replaced by one of its children
- In the worst case we reach a leaf node
- **Cost ≤ tree height – 1 + 1**
  - – 1 because the root node is in RAM
  - + 1 for accessing the data file
    - Not needed if the index contains the data
- It follows that **we need to know how to compute the height of a B-tree with order *d***

Min and max height of a B-tree

- let:
  - N be the number of nodes in the tree
  - m be the max number of children a node can have
  - d be the min number of children a node can have (d=m/2)
- a B-tree of height h has between with all its nodes completely filled has N = $m^{h+1}–1$ nodes
- best case height (i.e., minimum height):
  - $h_{min} = [\log_m(N+1) - 1]$
- worst case height (i.e., maximum height):
  - $h_{max} = \lfloor \log_d((N+1)/2) \rfloor$

B-tree of height h: max. number of nodes

- Every node contains a number of entries that can vary in the range [d, 2d] (2*d* is the order of the tree)
- The number of children nodes of each node equals m+1 (thus it can vary in the range [d+1, 2d+1])
- The maximum number of nodes is reached when all nodes are full, that is they contain *2d* entries
  - Every internal node thus has 2d+1 child nodes

$$b_{\max} = \sum_{l=0}^{h-1} (2d+1)^l = \frac{(2d+1)^h - 1}{2d}$$

- The maximum number of nodes is:

$$N_{\max} = 2d \cdot b_{\max} = (2d+1)^h - 1$$

B-tree of height h: min. number of nodes

- The minimum number of nodes is reached when all nodes (except the root) are half full, that is they contain *d* entries
  - the root only contains a single entry
- Every internal node thus has d+1 child nodes

$$b_{\min} = 1 + 2\sum_{l=0}^{h-2}(d+1)^l = 1 + 2\frac{(d+1)^{h-1} - 1}{d}$$

- The minimum number of nodes is:

$$N_{\min} = 1 + d(b_{\min} - 1) = 2(d+1)^{h-1} - 1$$

Height of a B-tree

- We are now able to compute N as

$$N_{\min} \leq N \leq N_{\max}$$

$$2(d+1)^{h-1} - 1 \leq N \leq (2d+1)^{h} - 1$$

- **Thus, we have that:**

$$\lceil \log_{2d+1}(N+1) \rceil \leq h \leq \left\lfloor \log_{d+1}\left(\frac{N+1}{2}\right) \right\rfloor + 1$$

Height of a B-tree: example

- Let's make the following assumptions:
  - **Key**: 8 bytes
  - **RP** (record pointer): 4 bytes
  - **BP** (block pointer): 2 bytes
  - **block size**: 4096 bytes
- We have that:
  - (Key+RP)2d+BP(2d+1)=(8+4)2d+2(2d+1)=4096
  - **d** = |_(4096-1)/(24+4)_| = 146
- If **N=10⁹**, searching for a key k costs (at most):
  - $|\_\log_{147}((10^9+1)/2)\_|$ = **5 I/O operations**
  - Binary search would require 22

B-tree: recap

- **Capacity of node equals the size of a disk bock**
- All nodes, except for the root, are **filled** for **at least 50%**
  - impact on additions and removals
- Complex to make exact predictions about required number of block accesses when searching B-tree
- B-trees can also be used as **primary file organization technique**
  - **instead** of data **pointers**, **nodes contain actual data fields** of records that correspond to search key values

B-Tree performance

Let h = height of the B-tree.

get(k): at most h disk accesses. $O(h)$

put(k): at most $3h + 1$ disk accesses. $O(h)$

remove(k): at most $3h$ disk accesses. $O(h)$

$h < \log_d(n + 1)/2 + 1$ where $d = \lceil m/2 \rceil$ (Sahni, p.641).

An important point is that the constant factors are relatively low.

m should be chosen to match the maximum node size to the block size on the disk.

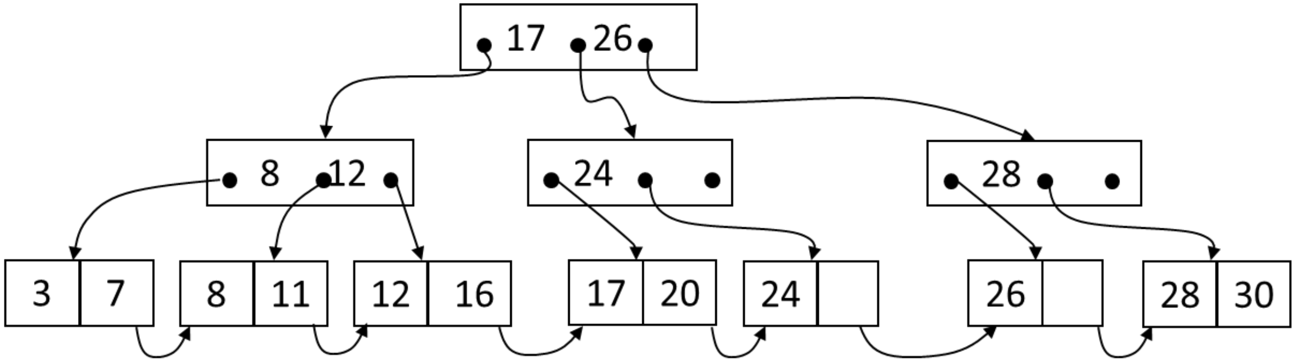
Example: $m = 128$, $d = 64$, $n \approx 64^3 = 262144$, $h = 4$.

B^+-tree

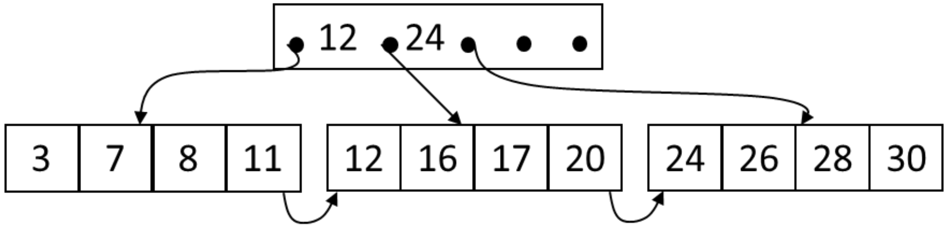
- In a B^+-tree
  - only leaf nodes contain data pointers
  - all **key values** that exist in non-leaf nodes are **repeated** in **leaf nodes**, such that every key value occurs in a leaf node, along with corresponding data pointer
  - higher-level nodes only contain subset of key values present in leaf nodes
  - every leaf node of a B^+-tree also has one tree pointer, pointing to its next sibling

B$^+$-tree

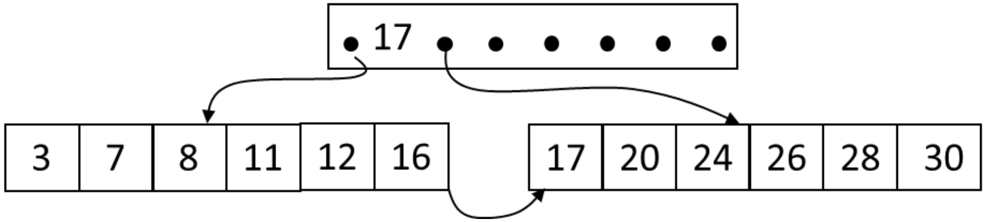
Order 1 (height = 3):



Order 2 (height = 2):



Order 3 (height = 2):



B^+-tree

- Searching and updating B^+-tree is similar as B-tree
- B^+-trees are often more efficient, because non-leaf nodes do not contain data pointers
  - height of B^+-tree is often smaller, resulting in less block accesses to search
- Variations on fill factor which is 50% for standard B-trees and B^+-trees
  - E.g., a B-tree with fill factor of 2/3 is called a B*-tree
-

Index specification in SQL

- CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table [USING method] ({ column | (expression) } [COLLATE collation] [opclass] [ASC | DESC] [NULLS { FIRST | LAST }] [, ...]) [WITH (storage_parameter = value [, ...])] [TABLESPACE tablespace] [WHERE predicate]