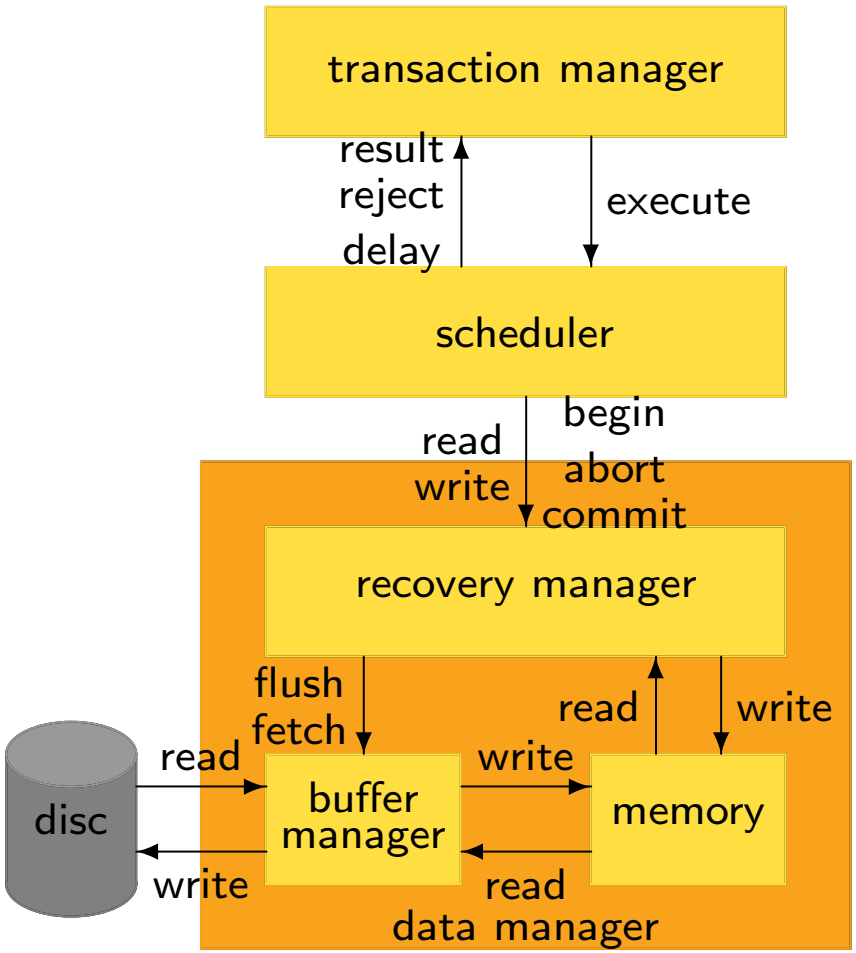
Storage & Indexing

P.J. McBrien

Imperial College London

DBMS Architecture



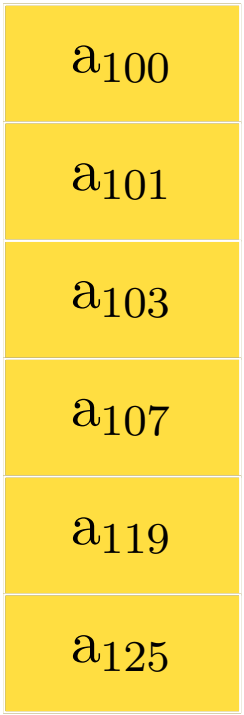
Storing Tables/Relations in Memory and on Disc

account				
no	type	cname	rate	sortcode
100	'current'	'McBrien, P.'	NULL	67
101	'deposit'	'McBrien, P.'	5.25	67
103	'current'	'Boyd, M.'	NULL	34
107	'current'	'Poulovassilis, A.'	NULL	56
119	'deposit'	'Poulovassilis, A.'	5.50	56
125	'current'	'Bailey, J.'	NULL	56

How is the data of a table held in memory?

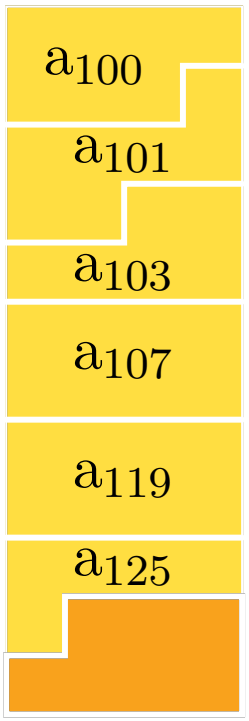
- call a row/tuple a **record** in this context

Records: Fixed Size



- Stored all fields (and hence records) in a fixed size
- Allows simple offset calculation to find nth record
- Common is simple database systems

Records: Variable Size



- Stored all fields with just enough bytes for data of a record
- Can only find nth record by sequential scan through records
- Common in complex database systems

Records: Sorted or Unsorted

a_{100}
a_{101}
a_{103}
a_{107}
a_{119}
a_{125}

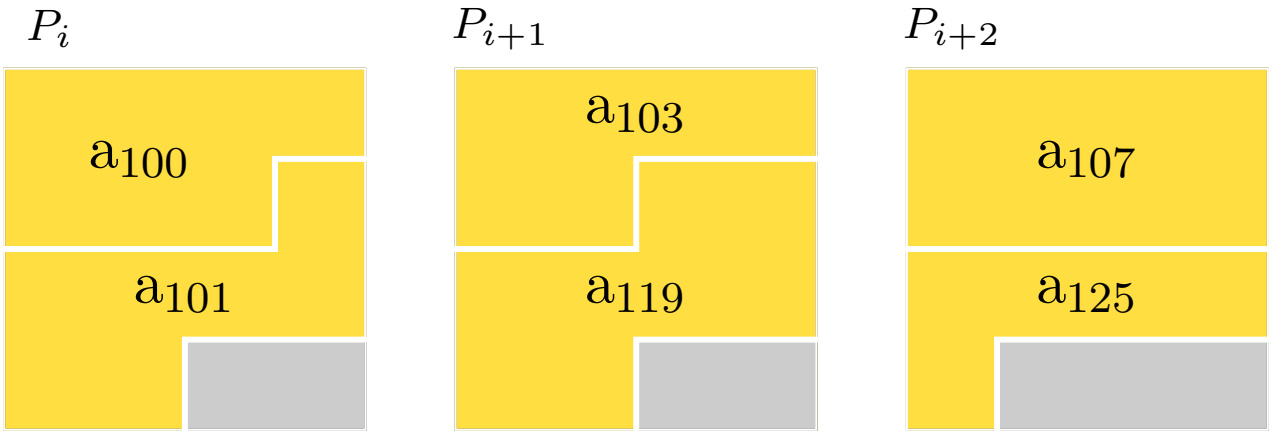
a_{100}
a_{101}
a_{103}
a_{119}
a_{107}
a_{125}

Pages

Databases normally deal with data in fixed sized blocks called ***pages***

Unspanned

- reading one record only requires reading one page
- efficient when size of record is much smaller than size of page

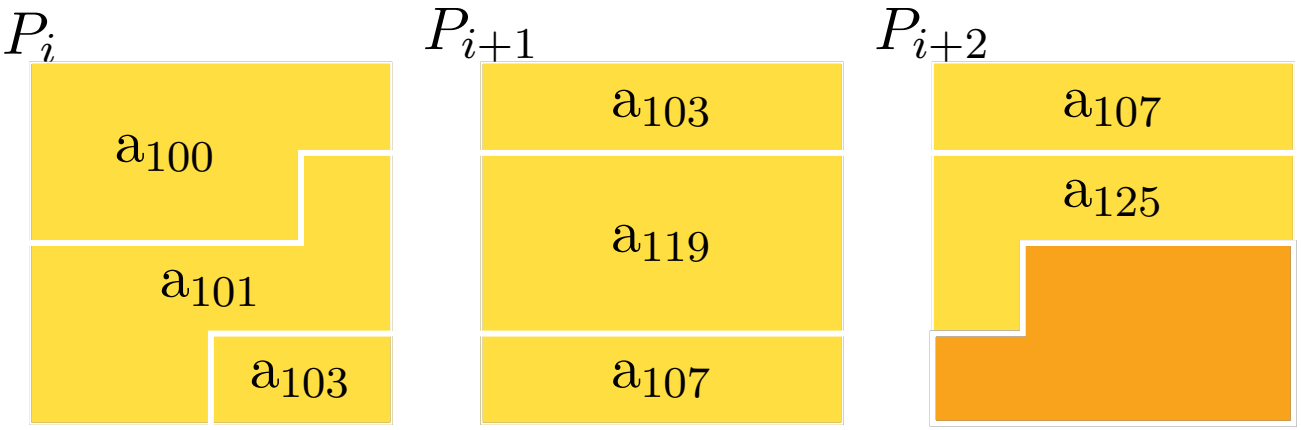


Pages

Databases normally deal with data in fixed sized blocks called ***pages***

Spanned

- supports record sizes greater than one page
- more space efficient



Quiz 1: Records in a Set of Unspanned Pages

In a particular database, each record occupies 620 bytes, and records are stored in unspanned pages of 2KB.

how many pages are required to stored 100 records?

A	B	C	D
30	31	33	34

Record: Access Requirements

sequential scan

Sometimes we need to read all records, *e.g.*

```
SELECT AVG(rate)
FROM   account
```

random access

Sometimes we just want one record, *e.g.*

```
SELECT rate
FROM   account
WHERE  no=101
```

provide an **index**

range queries

Sometimes we want a range of records

```
SELECT no,rate
FROM   account
WHERE  no BETWEEN 100 AND 120
```

Quiz 2: Spanned and Unspanned Storage

Compared with Spanned Storage, is Unspanned Storage:

A

More Compact, Faster Random Access

B

More Compact, Slower Random Access

C

Less Compact, Faster Random Access

D

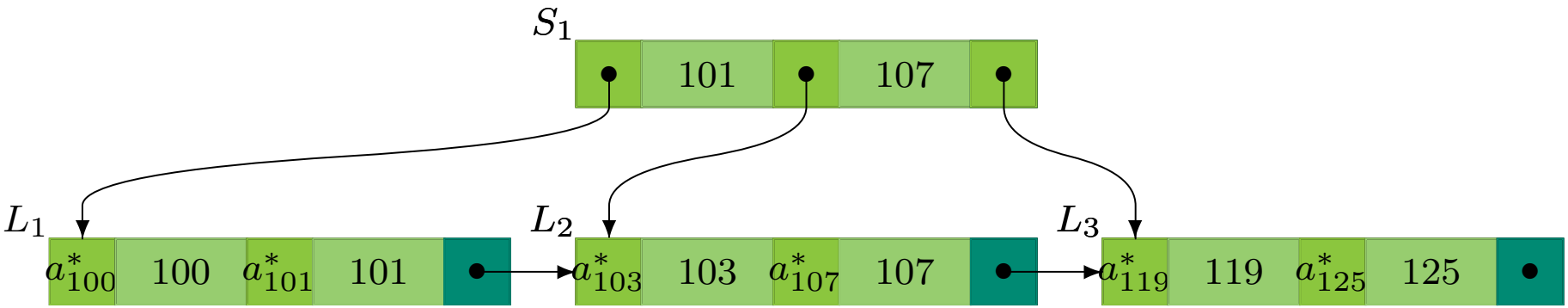
Less Compact, Slower Random Access

B$^+$-tree structure

- A **leaf node** L will contain
  - a list of m pairs $\langle R_i^*, V_i \rangle$, where
    - V_i is the value of the index attribute(s) in R_i
    - R_i^* is a pointer to the location of R_i
  - a next pointer L^* to the next leaf node in order of the index values
- A **search node** S will contain
  - a list of pairs $\langle N_i^*, V_i \rangle$ $(1 \leq i < n)$ where
    - N^* is pointer to a search node or leaf nodes, and
    - all nodes reachable via that pointer (apart from following the next pointer) have index attribute(s) less than or equal to V_i
  - a pointer N_n^* to a search node or leaf where all nodes reachable via that pointers have values exceeding V_{n-1}
- search and leaf nodes must be at least 50% full of pointers, root node can have just two pointers

B$^+$-tree for account(no)

	account				
	no	type	cname	rate	sortcode
a_{100}	100	'current'	'McBrien, P.'	NULL	67
a_{101}	101	'deposit'	'McBrien, P.'	5.25	67
a_{103}	103	'current'	'Boyd, M.'	NULL	34
a_{107}	107	'current'	'Poulovassilis, A.'	NULL	56
a_{119}	119	'deposit'	'Poulovassilis, A.'	5.50	56
a_{125}	125	'current'	'Bailey, A.'	NULL	56



- nodes made to fit on one disc page
- odd number of pointers

Quiz 3: Capacity of a B$^+$-tree

In particular B$^+$-tree, each search node contains three pointers to child nodes, and each leaf node contains two pointers to records.

how many records can be indexed by a B$^+$-tree of depth 3?

A	B	C	D
6	12	18	54

Size of B$^+$-tree Pages and Fan-Out

H size of page header
P size of page
V size of key value
R size of pointer reference to B$^+$-tree node or record page

B$^+$-tree Fan Out

$$n = \lfloor (P - H - R)/(V + R) \rfloor + 1$$

Fan-Out

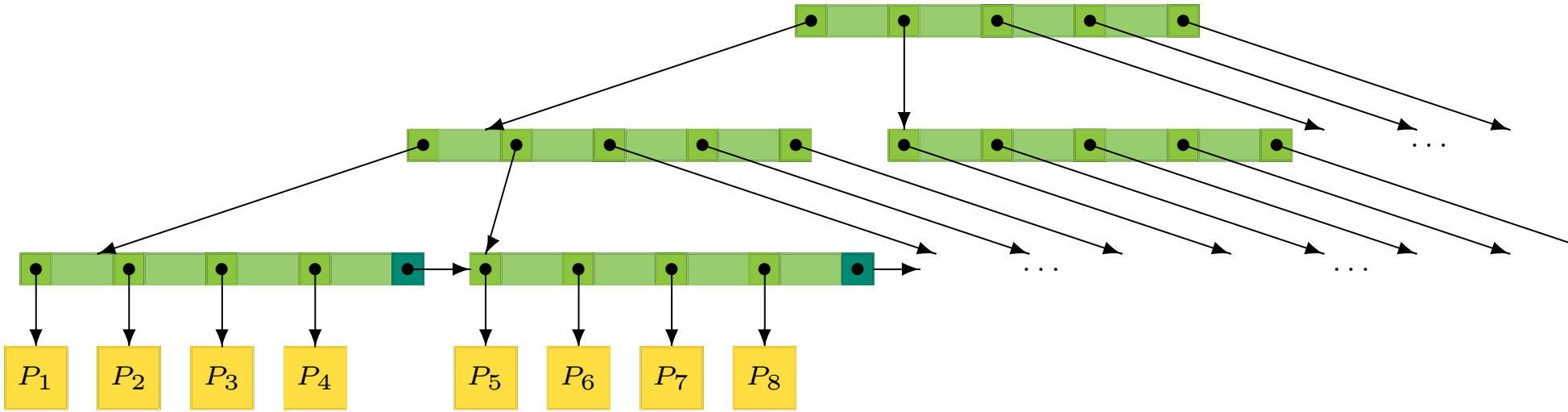
What is the fan-out n of a B$^+$-tree index of depth 3, where 64 bit pointers are used, and the index value is a 32 bit integer, and each page is 2KB in size?

Size of page P=2048, assume page header $H = 1$ byte
Size of value V=4
Size of pointer references R=8
fan out $n = \lfloor (P - H - R)/(V + R) \rfloor + 1 = \lfloor (2048 - 1 - 8)/12 \rfloor + 1 = 170$

Maximum Capacity of a B$^+$-tree

Definition (Maximum Capacity of B$^+$-tree)

If there are n pointers in each search node, then there are $n-1$ record pointers in each leaf node.
For a B$^+$-tree of depth d, maximum capacity $=n^{d-1} \times (n-1)$



Maximum Capacity of a B$^+$-tree

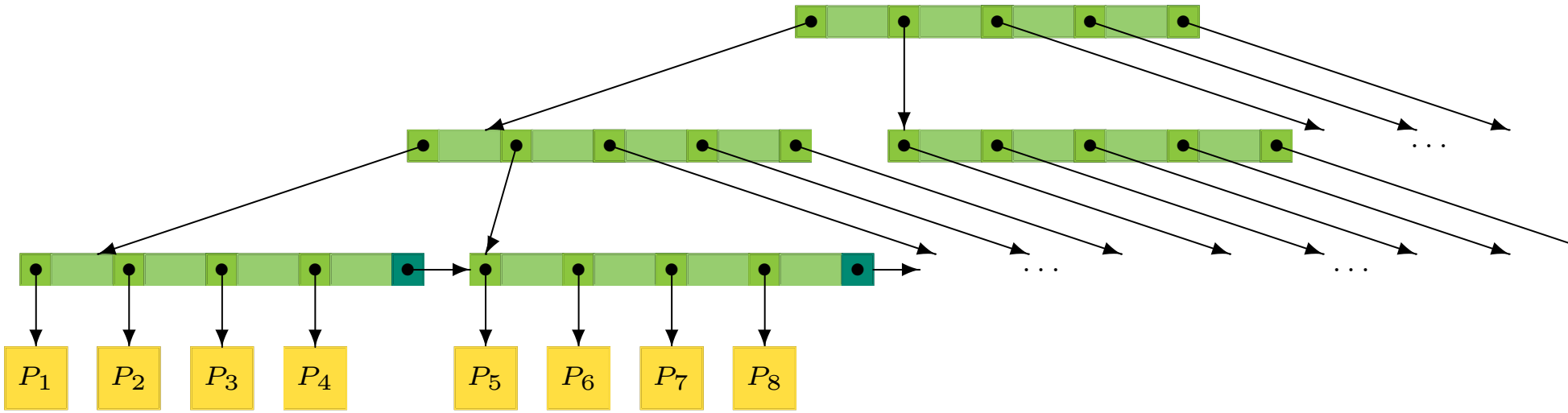
Definition (Maximum Capacity of B$^+$-tree)

If there are n pointers in each search node, then there are $n-1$ record pointers in each leaf node.
For a B$^+$-tree of depth d, maximum capacity $=n^{d-1} \times (n-1)$

Calculation of the Maximum Capacity of a B$^+$-tree

What is the maximum capacity of a B^+-tree index of depth 3, where 64 bit pointers are used, and the index value is a 32 bit integer, and each page is 2KB in size?
n references from each search node=170
maximum capacity=$170^2 * 169$=4,884,100

Quiz 4: Maximum Capacity of a B$^+$-tree



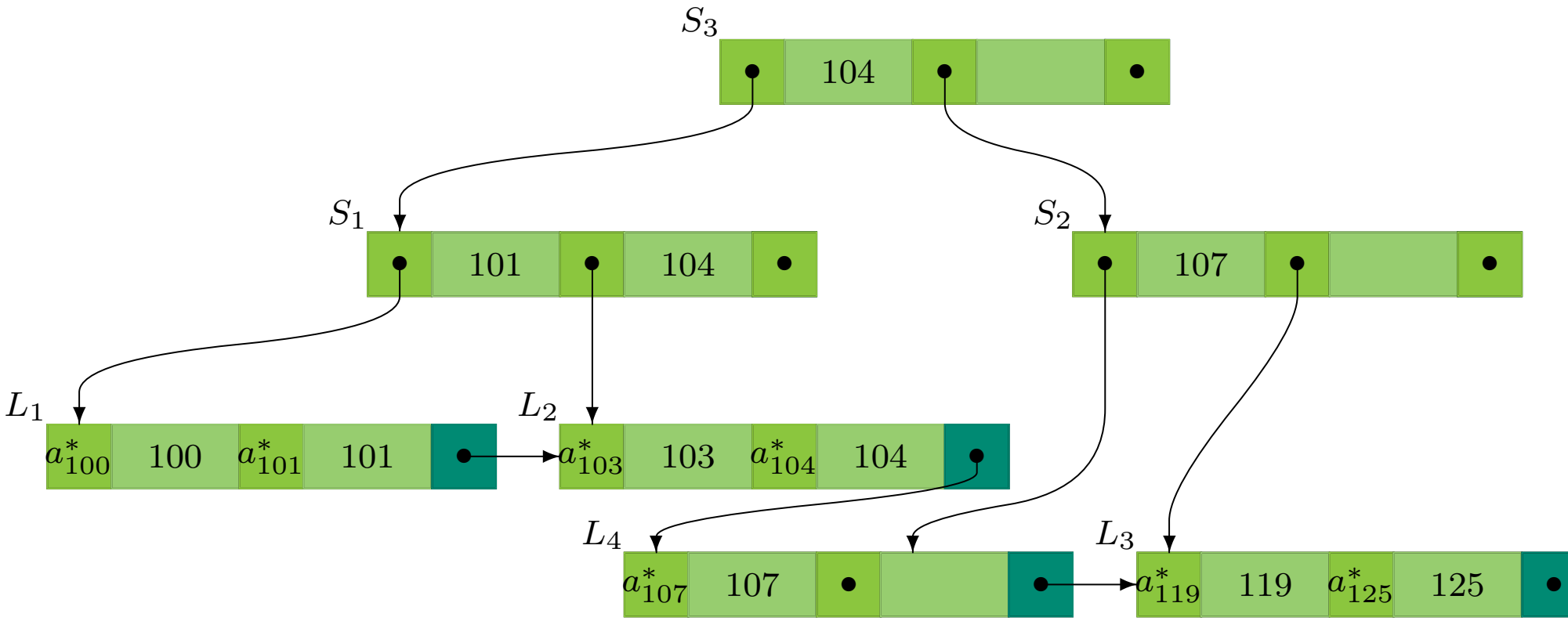
If a B$^+$-tree has a fan-out of five on search nodes, and depth three, what is its maximum capacity?

A	B	C	D
25	100	125	500

Worksheet: Table and Index File Sizes

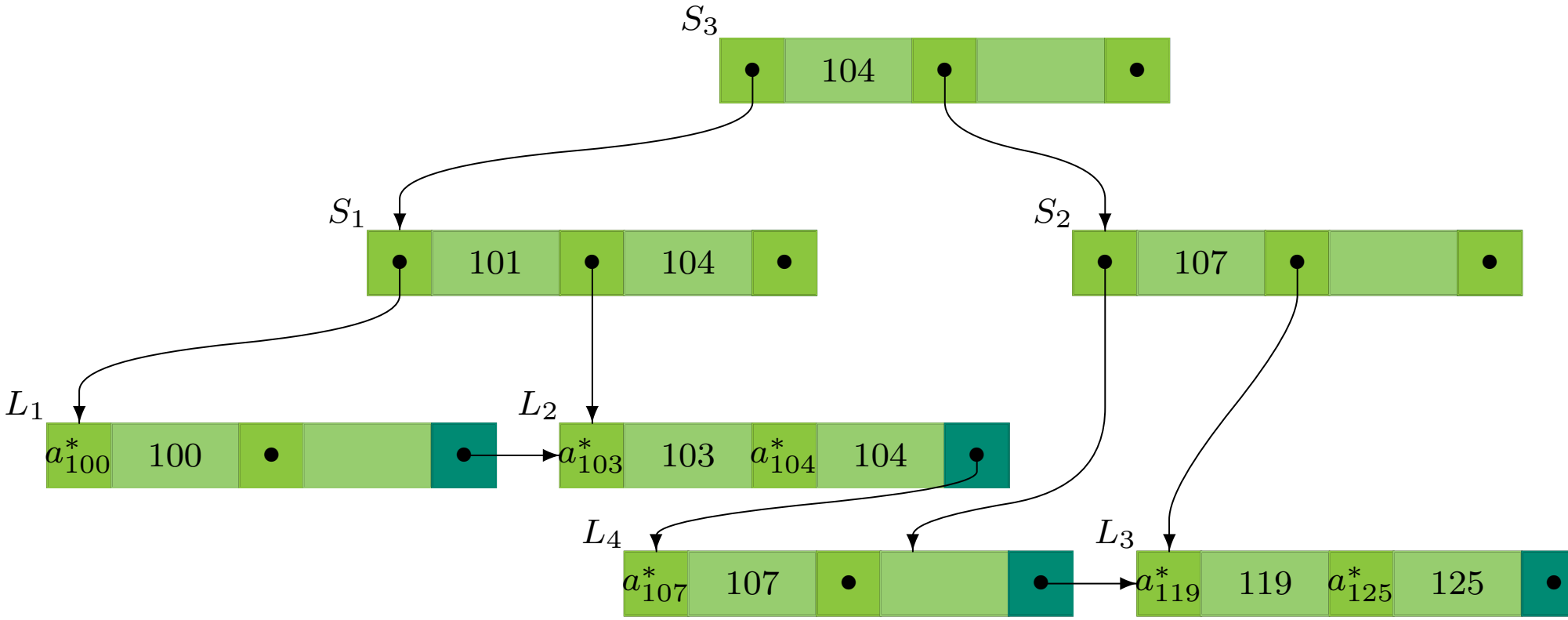
column	bytes
no	4
type	10
cname	20
rate	4
sortcode	6

B$^+$-tree for account(no) after $ins[a_{104}]$



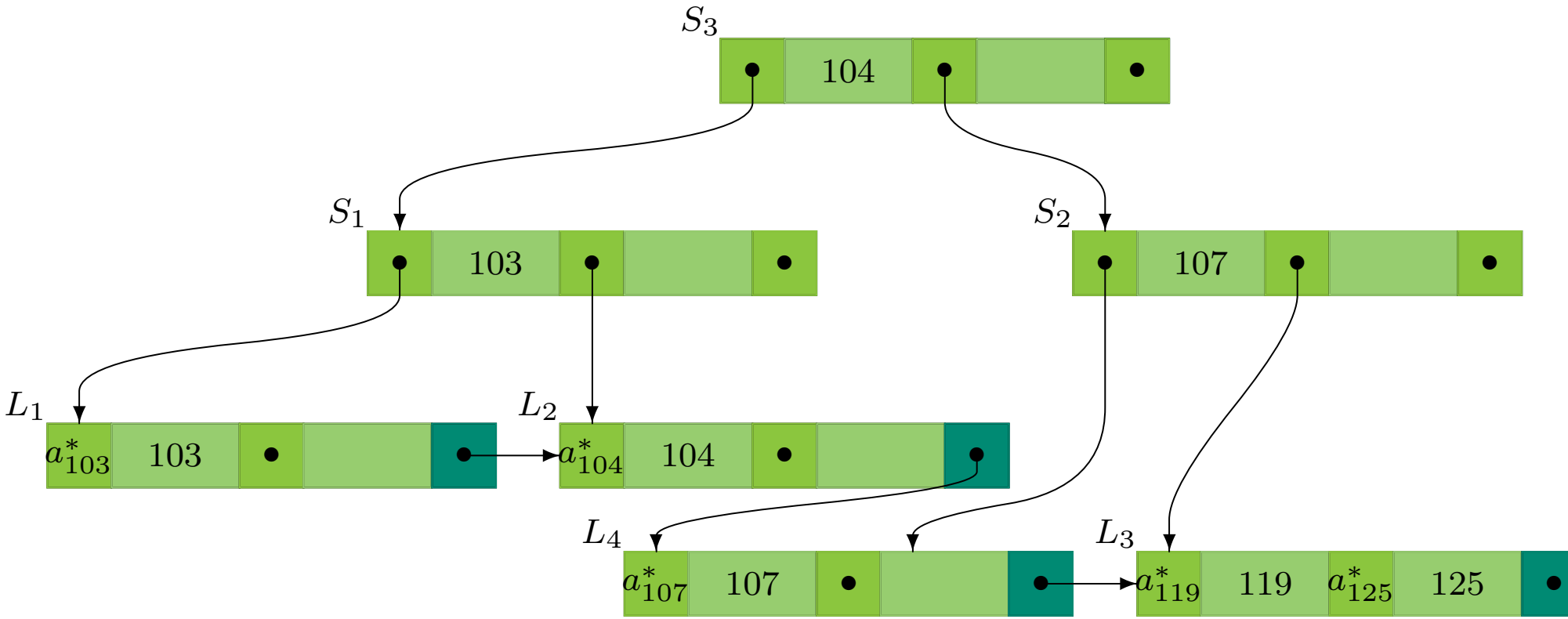
- if search node full, split and update (create) parent

B$^+$-tree for account(no) after $del[a_{101}]$



- if leaf still 50% full, just remove data from leaf
- note that index value in search node may not now appear in leaf

B$^+$-tree for account(no) after $del[a_{100}]$



- if leaf not full enough, redistribute from siblings

Quiz 5: Minimum Capacity of a B$^+$-tree

A B$^+$-tree must has at least 2 pointers in the root node, and 50% of the child pointers used in search and leaf nodes.
In particular B$^+$-tree, each search node contains three pointers to child nodes, and each leaf node contains two pointers to records.

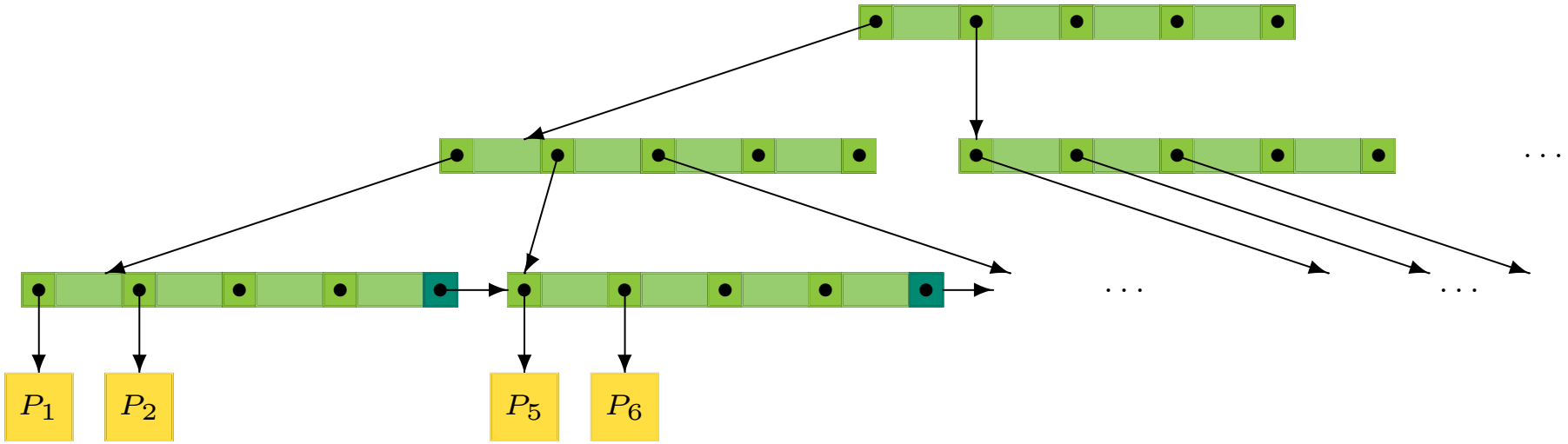
what is the minimum number of records referenced by such a tree of depth 3?

A	B	C	D
4	6	12	18

Minimum Capacity of a B$^+$-tree Before Shrink Occurs

Definition (Minimum Capacity of B$^+$-tree)

For a B$^+$-tree of depth d ($d > 1$):
minimum capacity before B$^+$-tree must shrink $= 2.\lceil \frac{n}{2} \rceil^{d-2}.\lceil \frac{n-1}{2} \rceil$



Minimum Capacity of a B$^+$-tree Before Shrink Occurs

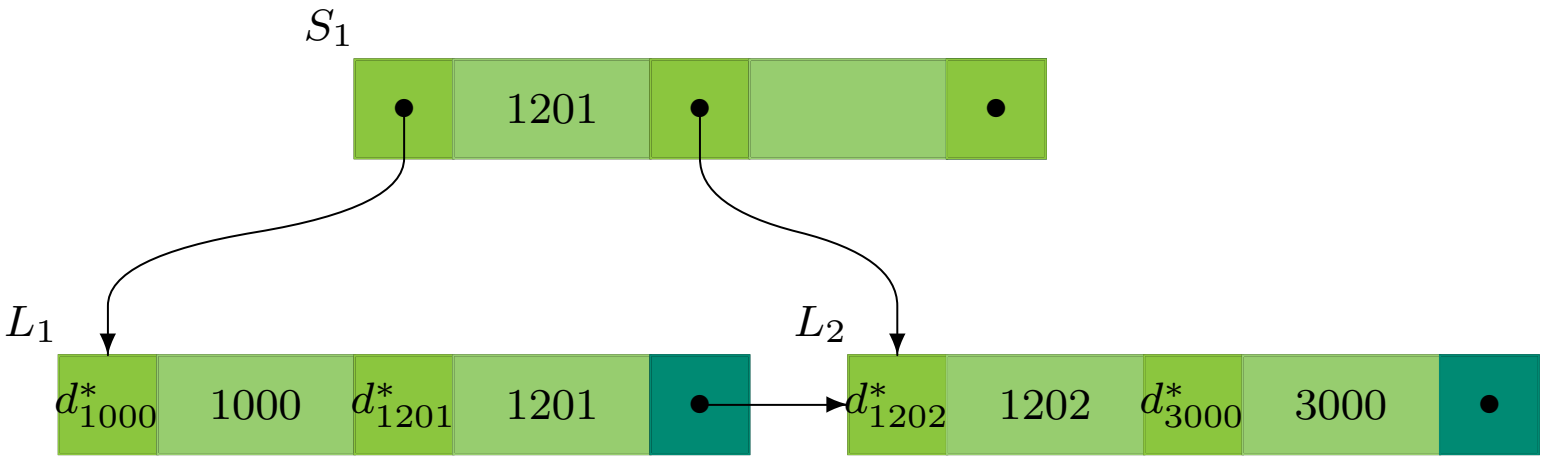
Definition (Minimum Capacity of B$^+$-tree)

For a B$^+$-tree of depth d ($d > 1$):
minimum capacity before B$^+$-tree must shrink $= 2.\lceil \frac{n}{2} \rceil^{d-2}.\lceil \frac{n-1}{2} \rceil$

Calculation of the Minimum Capacity of a B$^+$-tree

What is the minimum capacity of a B$^+$-tree index of depth 3, where 64 bit pointers are used, and the index value is a 32 bit integer, and each page is 2KB in size?
As before $n = 170$
minimum capacity$= 2.\lceil \frac{170}{2} \rceil.\lceil \frac{169}{2} \rceil = 14450$

Worksheet: B$^+$-tree Updates



directory		
telephone	name	charge
1000	Adams	10.00
1201	Jones	120.25
1202	Black	344.00
3000	Khan	30.00

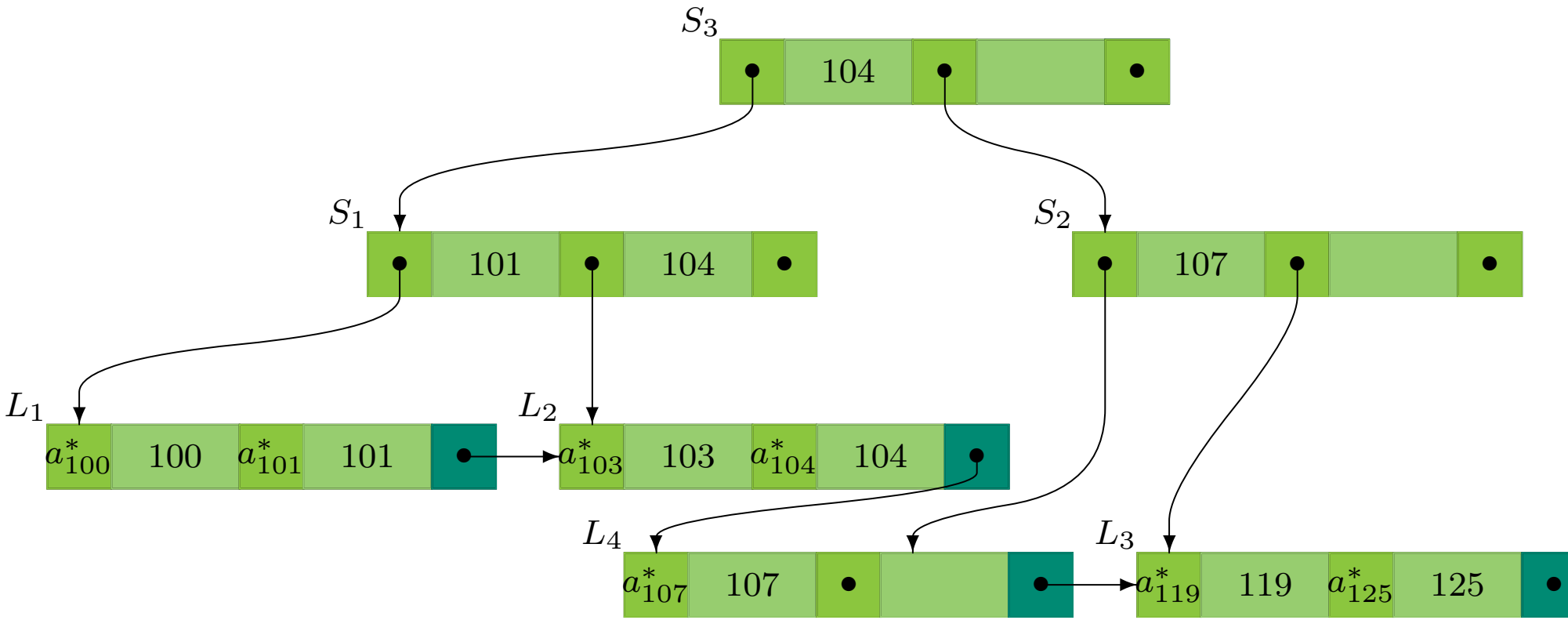
Reindexing

- Over time, deletes and inserts may leave the B$^+$-tree leaves with gaps
- A reindex process simply builds a new B$^+$-tree from the old, reading the values in the leaf nodes by index order

Rules for B$^+$-tree locking

- for find operations
  - obtain read lock on each node as read
  - release lock on parent once lock obtained on child
- for insert/delete operations
  - obtain read lock on node as correct leaf is found
  - obtain write locks on all nodes that are changed

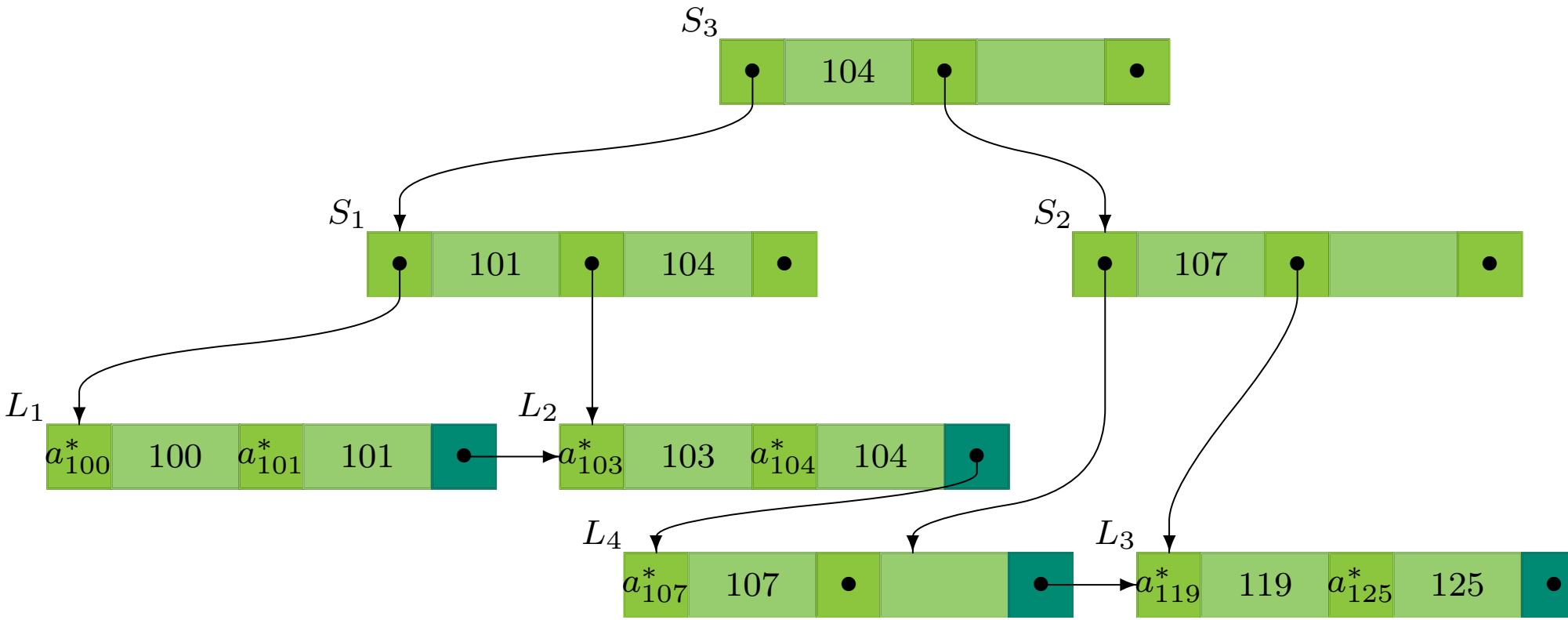
Examples of B$^+$-tree locking



SELECT * FROM account WHERE no=104

$rl[S_3], rl[S_1], ru[S_3], rl[L_2], ru[S_1], rl[a_{104}], ru[L_2]$

Examples of B$^+$-tree locking

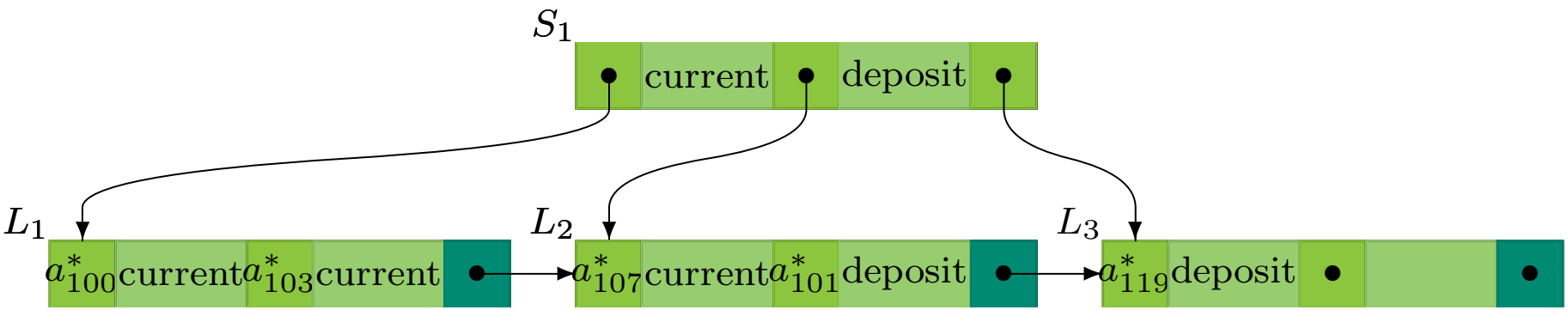


DELETE FROM account WHERE no=101

$rl[S_3], rl[S_1], wl[L_1], wl[a_{101}], wu[L_1], ru[S_2], ru[S_3]$

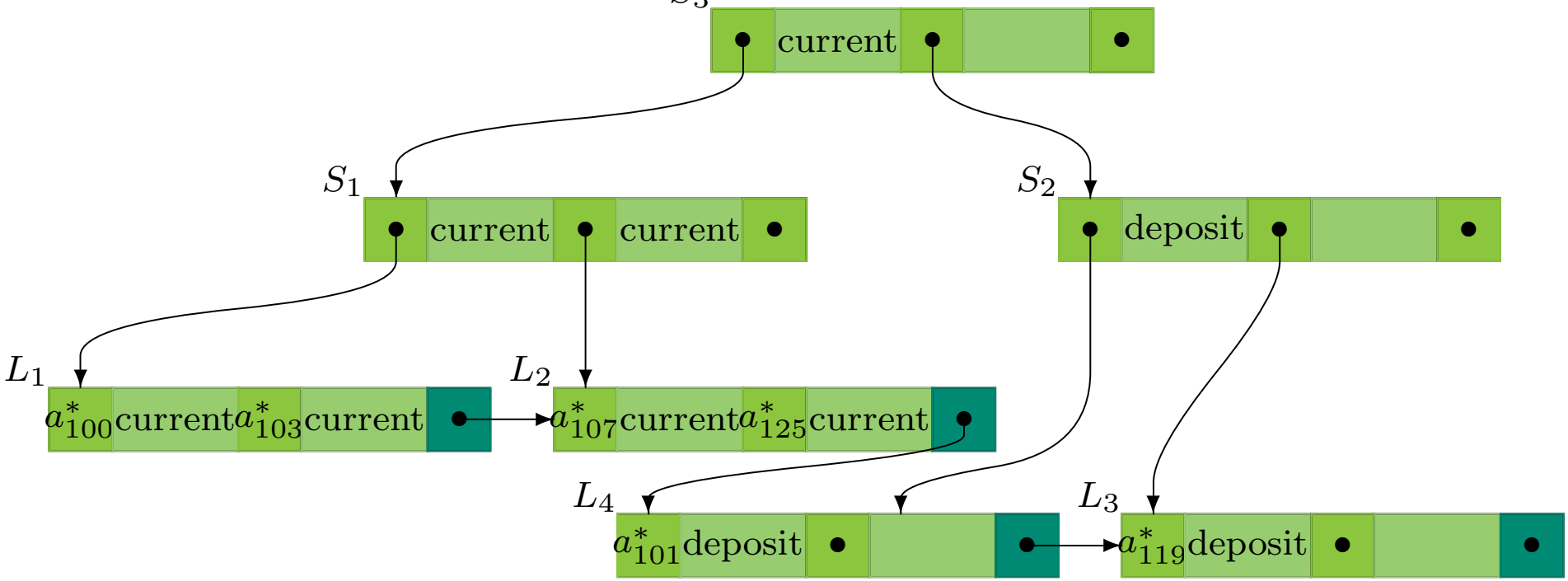
B$^+$-tree for account(type)

	account				
	no	type	cname	rate	sortcode
a_{100}	100	'current'	'McBrien, P.'	NULL	67
a_{101}	101	'deposit'	'McBrien, P.'	5.25	67
a_{103}	103	'current'	'Boyd, M.'	NULL	34
a_{107}	107	'current'	'Poulovassilis, A.'	NULL	56
a_{119}	119	'deposit'	'Poulovassilis, A.'	5.50	56
a_{125}	125	'current'	'Bailey, A.'	NULL	56



B$^+$-tree for account(type)

	account				
	no	type	cname	rate	sortcode
a_{100}	100	'current'	'McBrien, P.'	NULL	67
a_{101}	101	'deposit'	'McBrien, P.'	5.25	67
a_{103}	103	'current'	'Boyd, M.'	NULL	34
a_{107}	107	'current'	'Poulovassilis, A.'	NULL	56
a_{119}	119	'deposit'	'Poulovassilis, A.'	5.50	56
a_{125}	125	'current'	'Bailey, A.'	NULL	56



Bit-Map Index

- Sometimes the cardinality of an attribute A is small, meaning that the domain of values $\{V_1, \ldots, V_n\}$ of A has a small value of n
  e.g. account(type)
- If there is some method to identify the jth row of R, then can form a bitmap for each $V_i \in \{V_1, \ldots, V_n\}$ where bit j is set if and only if jth row has $R.A = V_i$

Example Bit-Map Indexes

row	account				
	no	type	cname	rate	sortcode
1	100	'current'	'McBrien, P.'	NULL	67
2	101	'deposit'	'McBrien, P.'	5.25	67
3	103	'current'	'Boyd, M.'	NULL	34
4	107	'current'	'Poulovassilis, A.'	NULL	56
5	119	'deposit'	'Poulovassilis, A.'	5.50	56
6	125	'current'	'Bailey, A.'	NULL	56

	bit					
index	1	2	3	4	5	6
account(type)='current'	1	0	1	1	0	1
account(type)='deposit'	0	1	0	0	1	0
account(sortcode)=34	0	0	1	0	0	0
account(sortcode)=56	0	0	0	1	1	1
account(sortcode)=67	1	1	0	0	0	0

Quiz 6: Efficiency of Bit-Map Indexes

account				
no	type	cname	rate	sortcode
100	'current'	'McBrien, P.'	NULL	67
101	'deposit'	'McBrien, P.'	5.25	67
103	'current'	'Boyd, M.'	NULL	34
107	'current'	'Poulovassilis, A.'	NULL	56
119	'deposit'	'Poulovassilis, A.'	5.50	56
125	'current'	'Bailey, J.'	NULL	56

Which bit-map index will occupy more space, and hence also be less likely to be efficient in query processing

A
no

B
type

C
cname

D
sortcode

Should an Index always be used?

- If you expect to need only a very few tuples from a relation, use an index to fetch those specific tuples
- If you expect to need most of the tuples of a relation, use a sequential scan, since
  - you will need to load almost all the pages of the table anyway
  - accessing via the index introduces an overhead (extra disc reads)
  - since you access the disc is index order and not physical order, may introduce some thrashing of disc heads
- If somewhere between the two, sometimes extra 'tricks' can be used
  - Postgres will read the entire index and process it as a bit-map index, and thus fetch the rows of the relation in order

Quiz 7: Usefulness of indexes

account				
no	type	cname	rate	sortcode
100	'current'	'McBrien, P.'	NULL	67
101	'deposit'	'McBrien, P.'	5.25	67
103	'current'	'Boyd, M.'	NULL	34
107	'current'	'Poulovassilis, A.'	NULL	56
119	'deposit'	'Poulovassilis, A.'	5.50	56
125	'current'	'Bailey, J.'	NULL	56

Which query benefits least from the provision of an index on no?

A

```
SELECT *
FROM   account
WHERE  no=101
```

B

```
SELECT cname
FROM   account
WHERE  no=101
```

C

```
SELECT no
FROM   account
WHERE  NOT no=101
```

D

```
SELECT cname
FROM   account
WHERE  NOT no=101
```