Ceng351
Programming Assignment 2

Recitation

Burçak Otlu

B+ tree concepts

- BucketFactor : number of records in a bucket
- With ln2 fullness ≈ A bucket can hold at most ln 2 * BucketFactor records.
- Order of B+ tree defines the number of keys in an internal node.
- A B+ tree of order v can have at most 2v and at least v keys at its internal nodes (except root).
- *Fanout* of a node: the number of pointers out of the node.

B+ Tree Properties

- *Balanced*
- Every node *except root* must be at least ½ full.
- An internal node with k keys has k+1 pointers.
- All leaves are at the same distance from the root.
- Every key from the table appears in a leaf, in left-to-right sorted order.

B+ Tree in practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

B+ Tree in summary

- Searching:
  - $\log_d(n)$ – Where *d* is the order, and *n* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

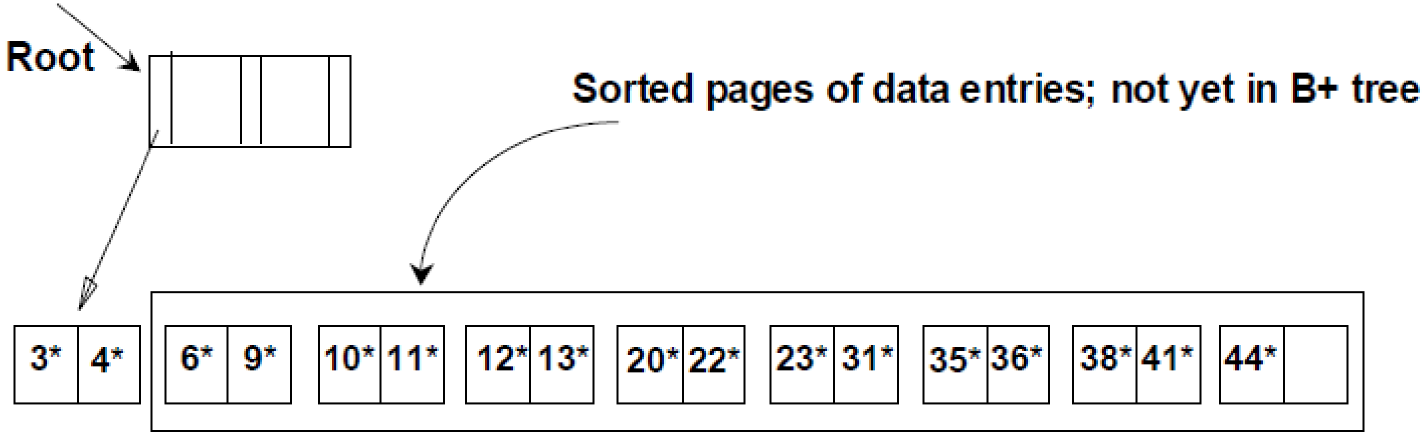
Bulk Loading of a B+ tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.

- Bulk Loading for creating a B+ tree index on existing records is much more efficient

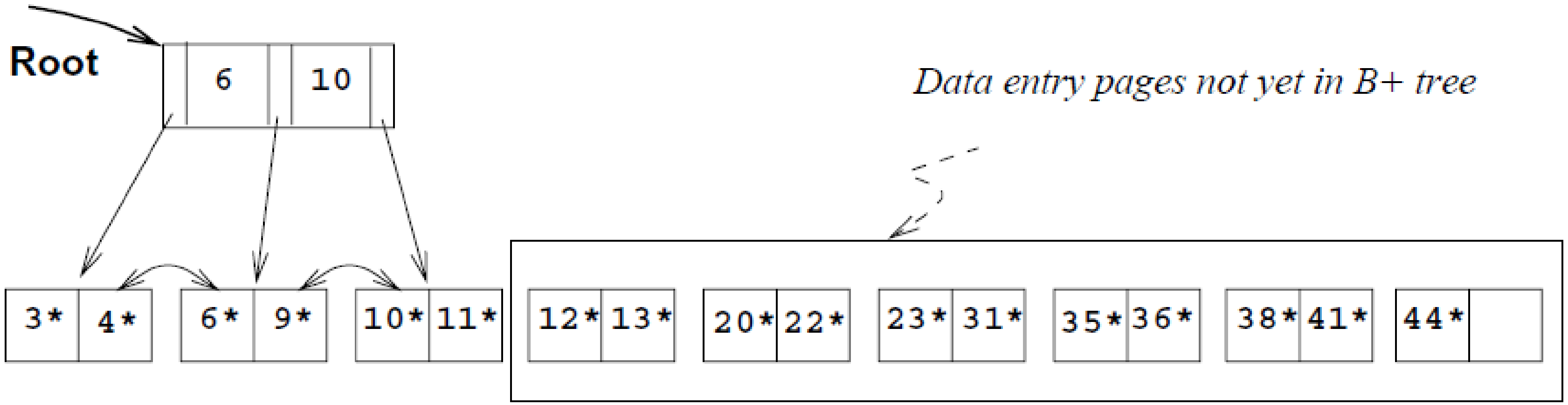
Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



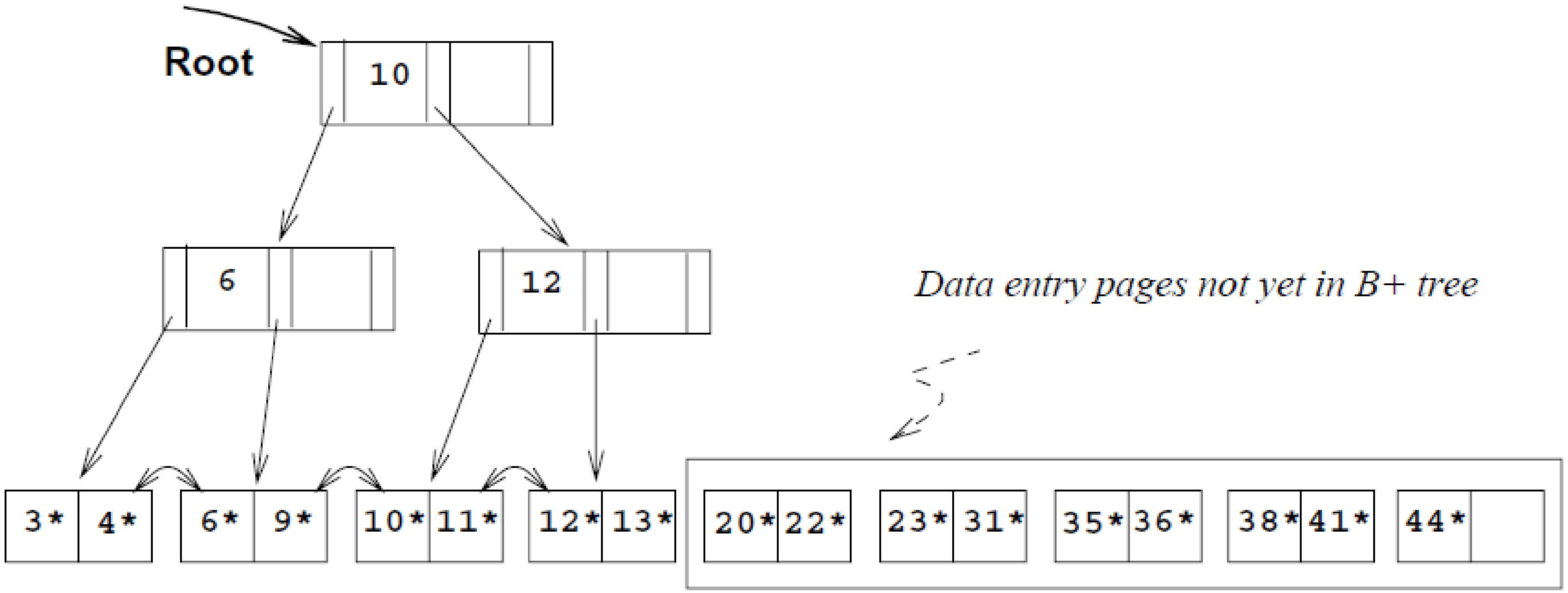
Bulk Loading (Contd.)

- Add *<low key value on page, pointer to page>* to the root page



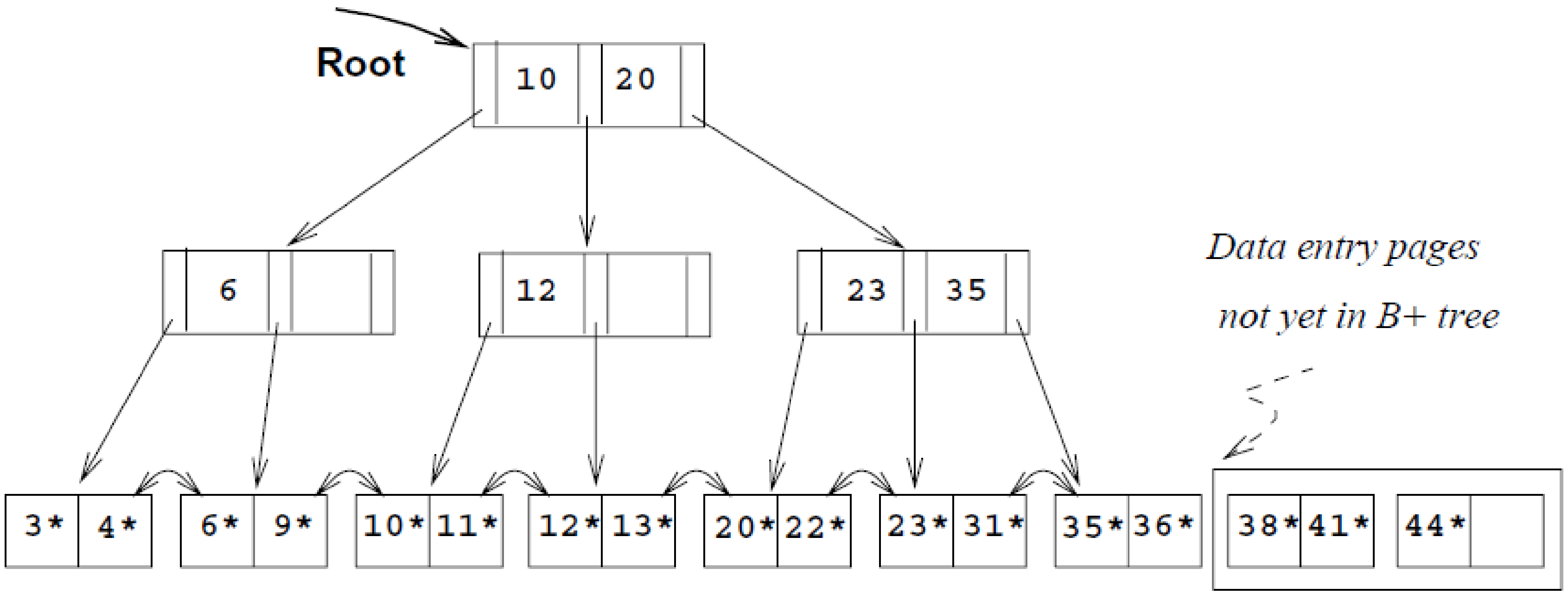
Bulk Loading (Contd.)

- Split the root and create a new root page.



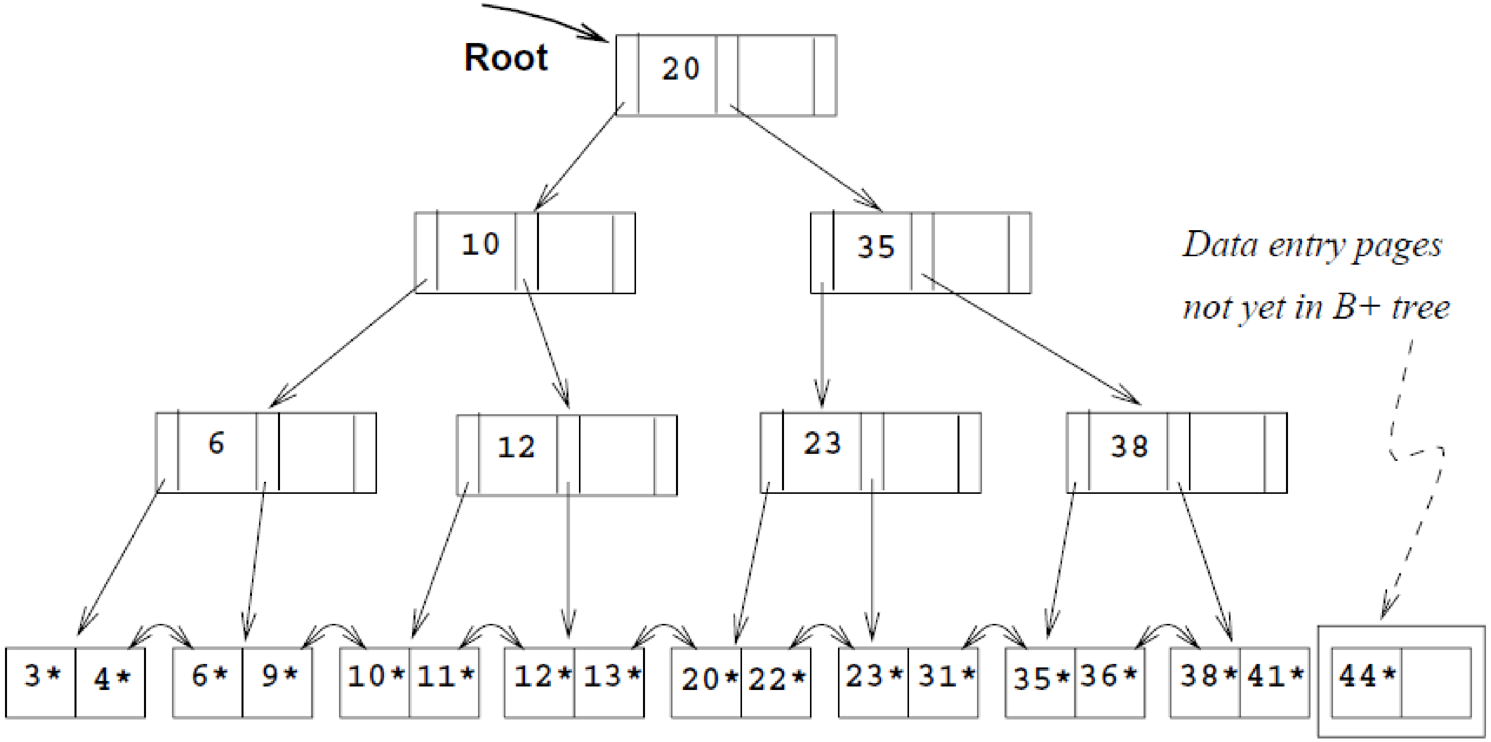
Bulk Loading (Contd.)

- Index entries for leaf pages always entered into rightmost index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)



Bulk Loading (Contd.)

- Much faster than repeated inserts.



Programming Assignment2 (PA2)

- There is an input.txt file (Please note that input.txt is sorted w.r.t. Akey6)

```
#34 6
#string(8) akey1
#float akey2
#char akey3
#int akey4
#char akey5
#string(6) akey6
rdmdquiz|8910.485|e|26936|l|aahdyc
nnxgjzzm|6779.772|z|49459|e|aaraqv
nqbflmxo|9650.604|u|61032|p|abjyvj
infhbnjg|7292.245|t|54354|x|abkczm
txplmpkg|2500.297|s|80453|y|abqdhw
qqoqwzez|5805.430|t|51432|t|abyvlo
dsrbrhdf|8633.386|d|22583|s|acgbfd
……
```

PA2

- PA2 will be implemented in C or C++.
- In order to run PA2, you have to provide three arguments.

1. bucketFactor
2. treeOrder
3. primaryKey (sparse Index)

PA2

- You will read input.txt
- You will first read the header
- While reading the header,

1. You will get recordSize in bytes and number of fields in each record.
2. You will get the each field type and field name
3. In case of an string field you will get its size in bytes between parenthesis.

   e.g. #string(6) akey6

PA2

- You will read input.txt
- You will first read the header
- While reading the header,

1. You will get recordSize in bytes and number of fields in each record.
2. You will get the each field type and field name
3. In case of an string field you will get its size in bytes between parenthesis.

   e.g. #string(6) akey6

PA2

- You will continue reading input.txt until end of file is reached
- You will read (bucketFactor*0.7) records and create a new bucket
- You will insert buckets into a B+ Tree, construct B+Tree.
- Meanwhile you will write buckets into binary output.dat file.
- Your buckets will not keep records. Records will be written to binary file.
- Bucket will keep records offsets.
- During search, if you find a hit, you will read from the binary file by the offset’s you have stored earlier.
- Please notice that delimiter character between filed is ‘|’ (vertical bar)
- e.g.: rdmdquiz|8910.485|e|26936|l|aahdyc

PA2

- After input.txt is consumed
- You will execute the user entered commands from the terminal until «quit» is entered.

1. print btree
2. search btree equality <key> <field list>
3. search btree range <lower bound key> <upper bound key> <field list>
4. quit

Constructing B+ Tree

- You will have index nodes.
- Index nodes at parent-of-leaf level will have pointers to buckets.
- Index nodes at non-parent-of-leaf level (higher than parent-of-leaf) will have pointers to index nodes.
- You can think of an index node having pointers to buckets or index nodes depending on whether it is parent-of-leaf or not.

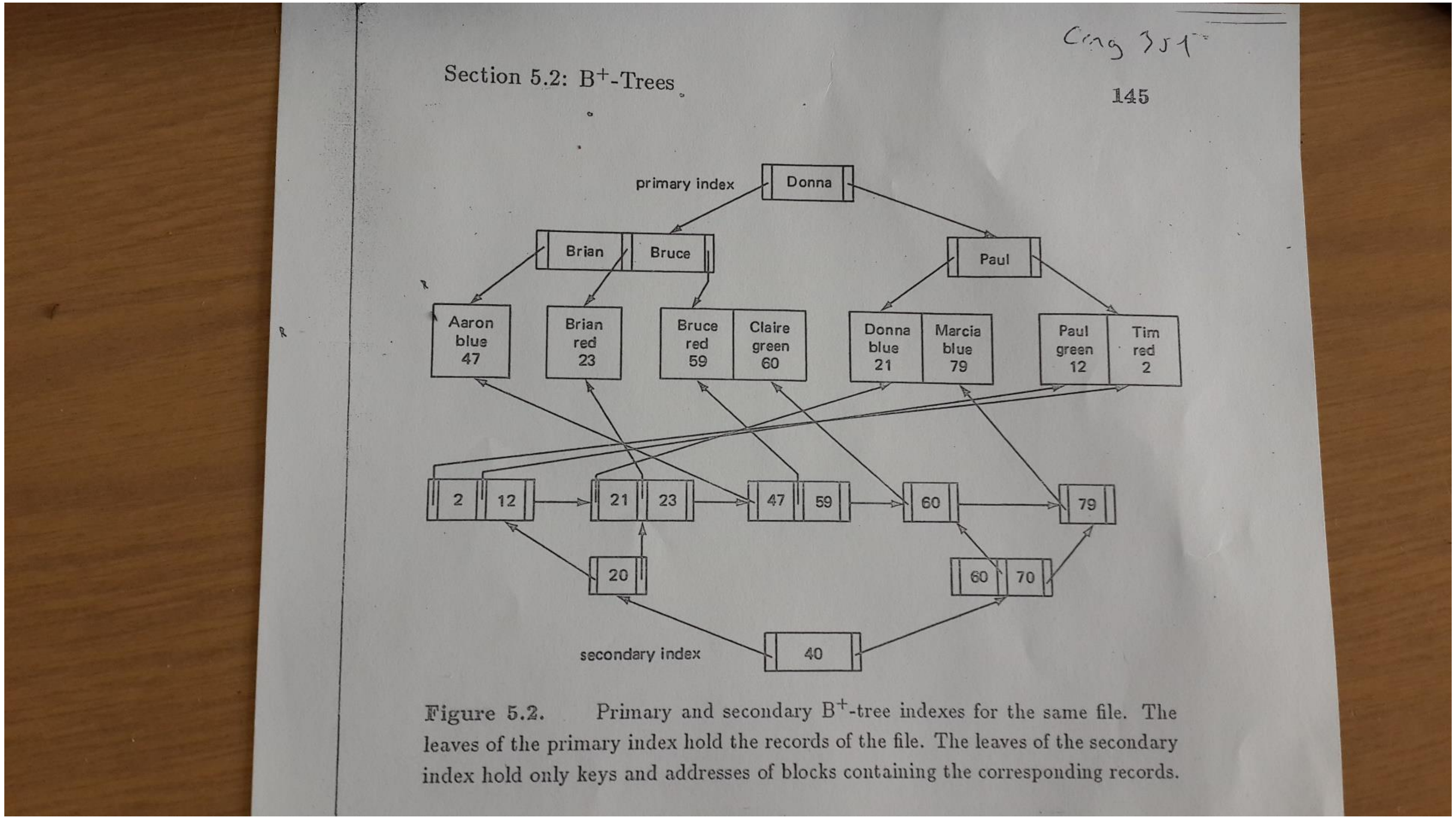
Constructing B+ Tree

- When inserting an bucket, you must first find the right most parent-of-leaf level index node (Let’s call this existing node).
- If it is not full, you can insert bucket and its key (key of first record in a bucket, there for it is a sparse index.).
- If it is full, you have to split, you need to create a new index node and distribute the keys and pointers evenly between the existing node and new node.
- Middle key has to be pushed up (parent of existing node).
- If existing node has no parent, create a new index node, make it the new root. Add middle key and pointers of existing node and new node.
- If existing node has a parent and it is not full, add middle key and pointer of new node.
- If existing node has a parent and it is full, you have to split parent (it may continue up to the root).

Constructing B+ Tree

- Don’t forget that in each index node (except root) , number of pointers must be one more than number of keys.

B+ Tree in Salzberg's book, pp.145



Ceng 351

Section 5.2: B^{+}-Trees

145

Figure 5.2. Primary and secondary B^{+}-tree indexes for the same file. The leaves of the primary index hold the records of the file. The leaves of the secondary index hold only keys and addresses of blocks containing the corresponding records.

Thank you for listening.

- Any questions?