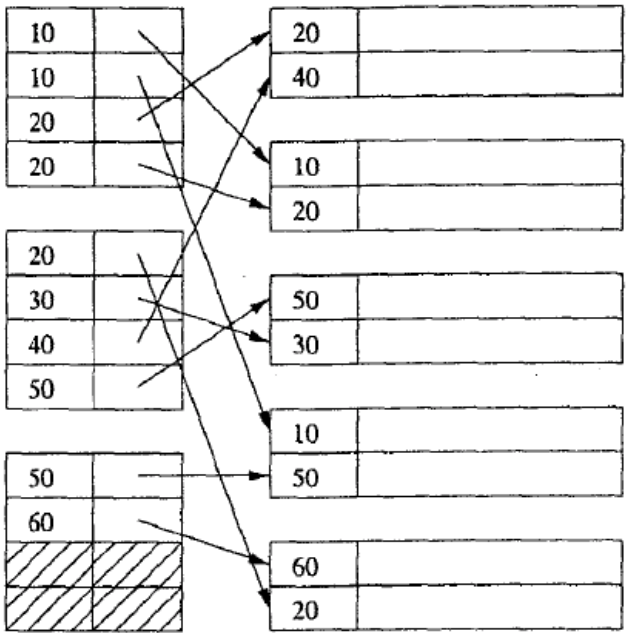
Secondary index

- So far we have considered indexes where the search-key of the index is also the sort-key of the data file.
- However, when may want indexes that are not based on the sort-order of the datafile.
- A **secondary index** does not determine the placement of tuples in the data file.
  - It does, however, tell us about the current location of tuples in the data file.
- Import consequence: a secondary index **must be dense**
  - We cannot predict the relative order of data-file tuples from the relative order of keys in the index.

Secondary index



- In our example, the search key is again an integer
- The data-file is not sorted according to the search key
- However, in the dense index the keys are sorted.
  - Note that keys in the same index block can point to many different data-file blocks
  - Also note that multiple instances of a search key can exist.

How many disk blocks must be accessed to obtain all tuples with the search key of 20?

Where they can be used

- If the data file is not in sorted order, even for the primary key, then a secondary index is necessary.
- One option for the data file is to make it a **clustered file**:
  - Here the tuples from two relations are intermixed.
  - However, they are intermixed in a clever way
- Clustered file example:
  - Given relations R and S...
  - ... where there is a many-to-one relationship from R tuples to S tuples ...
  - ... then we could store R tuples near the S tuple to which it is related.

Clustered file: example

```
Movie(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

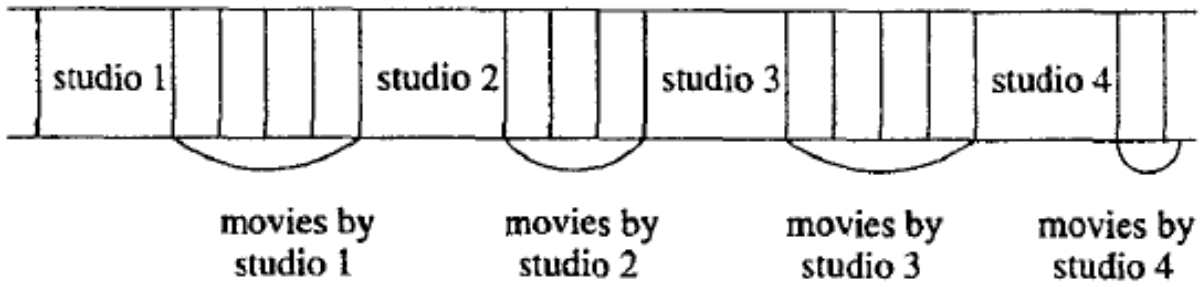
```
select title, year
from Movie, Studio
where presC# = <num> and Movie.studioName = Studio.name;
```

Given the unique ID of the president of a movie studio, find all movies made by that studio. (Recall we need both movie title and movie name to uniquely identify a movie.)

Let's also assume the query is the most common for the database.

Clustered file: example

- Assuming we have statistics showing us the query is the most common, then we can:
  - Avoid ordering Movie tuples by title and year...
  - ... and instead create a clustered file structure for both Studio and Movie...
  - ... where following each Studio tuple are all Movie tuples corresponding to movies own by the studio

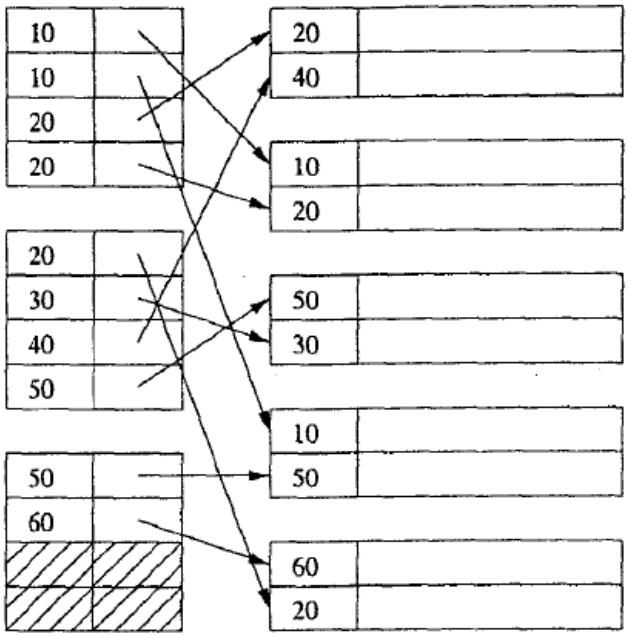


Clustered file: example

- The secondary index for our clustered file would use presC# as the search key
  - This index helps us quickly locate the tuple for the proper studio in the clustered file.
  - Movie tuples now no longer need their own index!
  - Movie tuples are packed together closely into the data file...
  - ... and therefore we achieve our goal of minimizing disk-block transfers.

```
select title, year
from Movie, Studio
where presC# = <num> and Movie.studioName = Studio.name;
```

Back to our secondary index example...



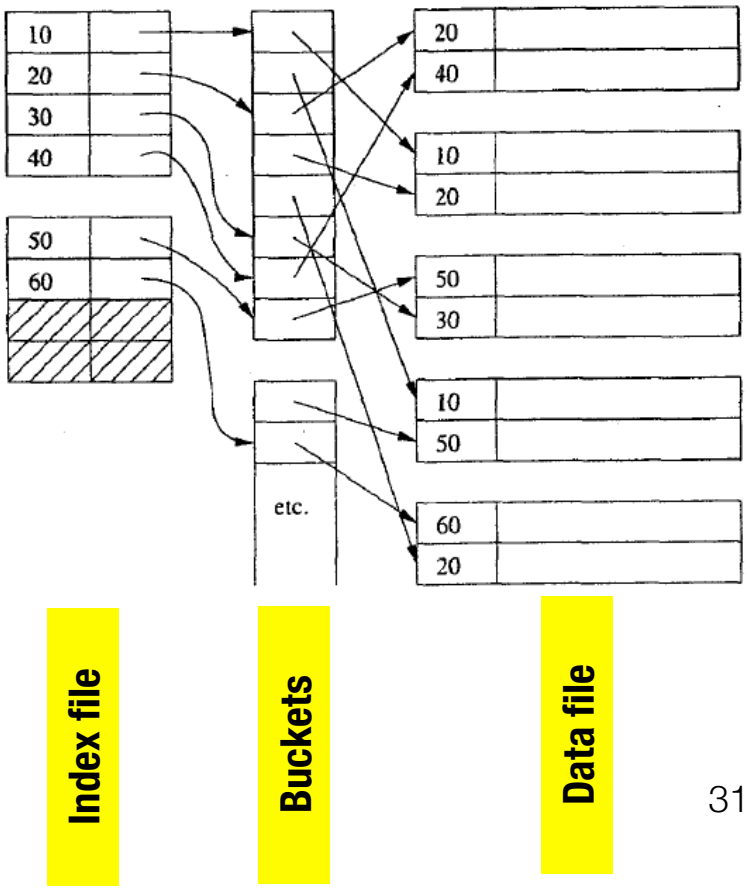
- There is a particular form of wastage here that could be significant...
- What is it?
- And how might we eliminate it?

If the search key is repeated n times in the data file, then it appears n times in the index. If the key and n are large, we could have wasted space.

We can solve this by inserting a level of indirection from keys to data-file.

Indirection in secondary indexes

- There is now one pair of links between keys and **buckets**
  - We save space as long as search-key values are larger than pointers
- The bucket contains pointers to the data file
  - We follow from the bucket position indicated by the key...
  - ... until we reach a bucket entry for some following key.



Indirection: another use

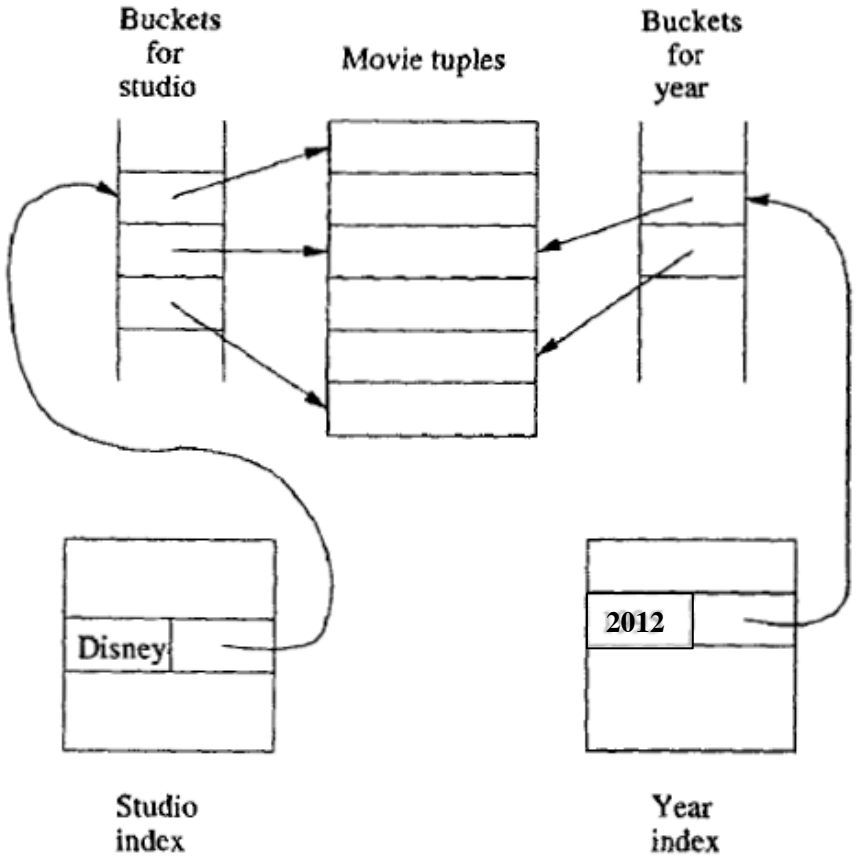
- This structure can help us even if the search keys are larger than pointers!
- Idea:
  - If an SQL statement has several conditions in its **where** clause...
  - ... and if each condition involves a tuple having a secondary index ...
  - ... then we can find bucket pointers satisfying the conditions by intersecting the set of pointers in memory...
  - ... and only then going to the data file!

Indirection use: another example

```
Movie(title, year, length,
genre,
    studioName, producerC#)
```

```
select title
from Movie
where studioName = 'Disney'
   and year = 2012;
```

Find all the Disney movies made in 2012.

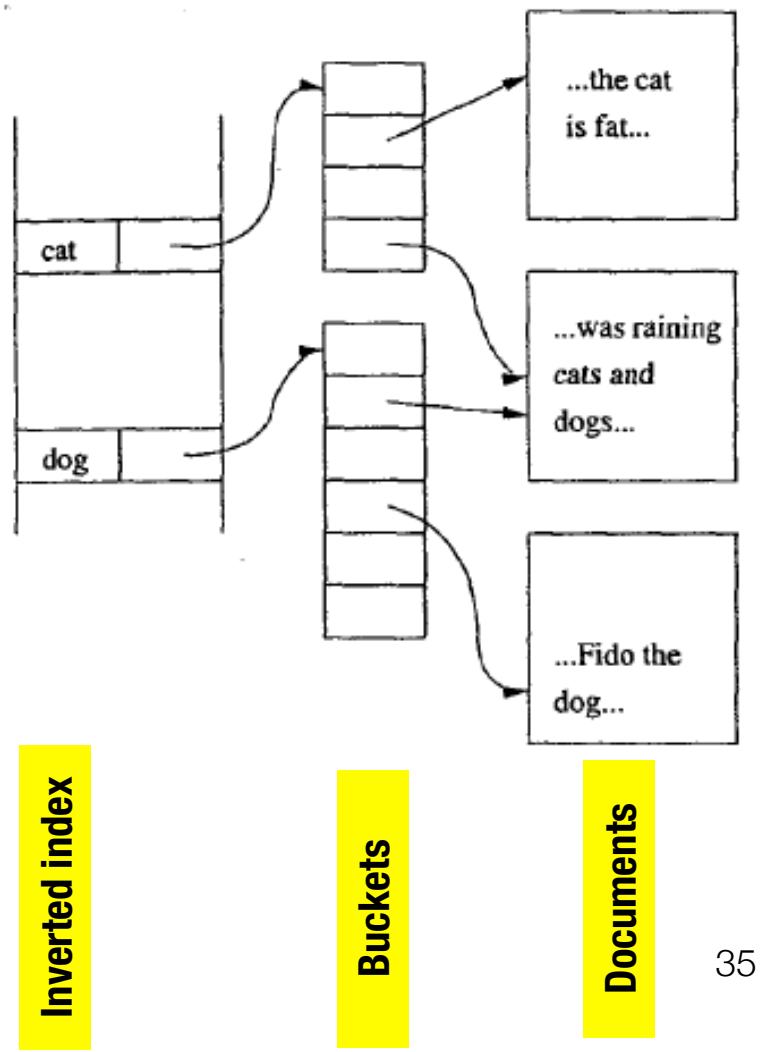


Inverted index

- Consider searching for a tuple given a search key
  - If the tuple is in the data file, it can be found by either examining the data file...
  - ... or by examining the index for the search key, and then retrieving only the block(s) from the data file containing tuples with the search key.
- In essence, the data file is a **document**.
- However, suppose we don't even know which document to search for the key?
- **Inverted index**:
  - Given a search key, such an index returns points to documents containing the search key.
  - Also called an **inverted file**.

Inverted index: example

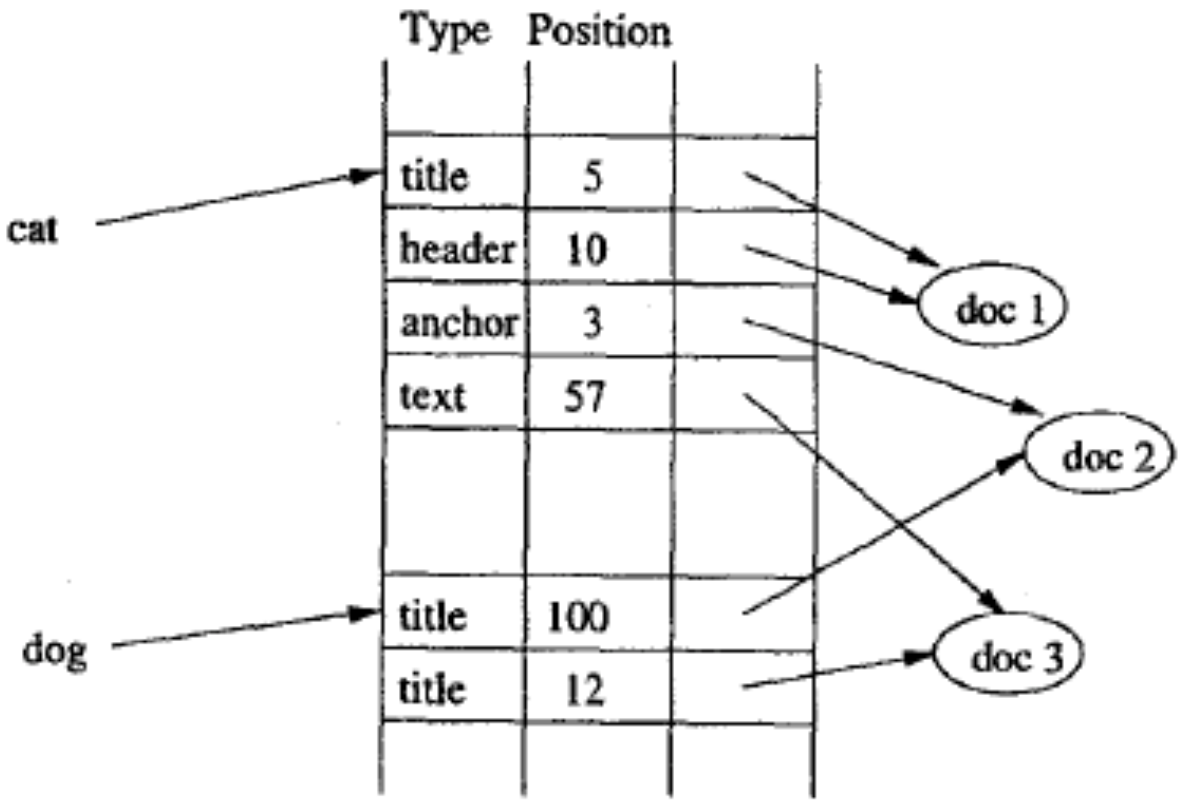
- Instead of a data file, we have a collection of documents
- Index consists of word-pointer pairs
  - Words therefore become the search key for the index
  - Pointers lead to a **bucket file**.
- Each bucket itself has a **list of pointers**
  - Here the pointers are to documents.
  - "cat" bucket has pointers to documents containing this word.
  - Note that to find documents with "cat" and "dog", we take the intersection of document pointers



Variants

- Bucket file pointers can be one of two types depending on the style of inverted index
- **record-level inverted index**
  - Pointers are from words to documents
  - The term record here means individual document (i.e., multiple documents == multiple records)
- **word-level inverted index**
  - Pointers are from words to documents and to the position of words in those documents.
  - Also called a **full inverted index**.
- For both variants there is a significant cost to addition of a new document!

Full inverted index (example)



B+ trees

- We can do better than one- or two- (or more) of index levels
- Commercial DBMSes use a family of structures called B-trees
- We will focus on the B+ tree
  - These store search key / value pairs (i.e., values are pointers to disk-file blocks)
  - Automatically maintains as many levels of indexes as needed.
  - Tree algorithms manage space in disk blocks store tree structures in order to keep the blocks half or completely full.

B+ trees

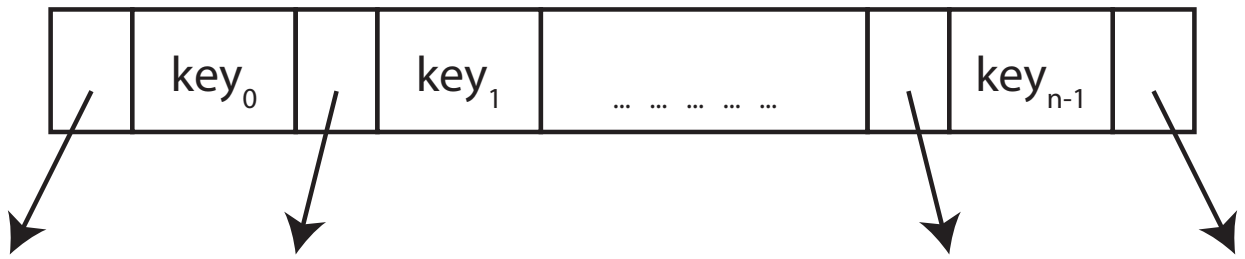
- A B+ tree:
  - Is a tree whose nodes are blocks/pages on a disk.
  - We can distinguish between **leaf nodes** and **interior nodes**
- **Leaf nodes**:
  - Store data entries in the form of (key, value) pairs.
  - All leaf nodes in a B+ tree are organized into a linked list of pages.
- **Interior nodes**:
  - Form the tree structure
  - Starts from a root node
  - Goal is to speed lookup to leaf node that contains desired key.

Leaf node

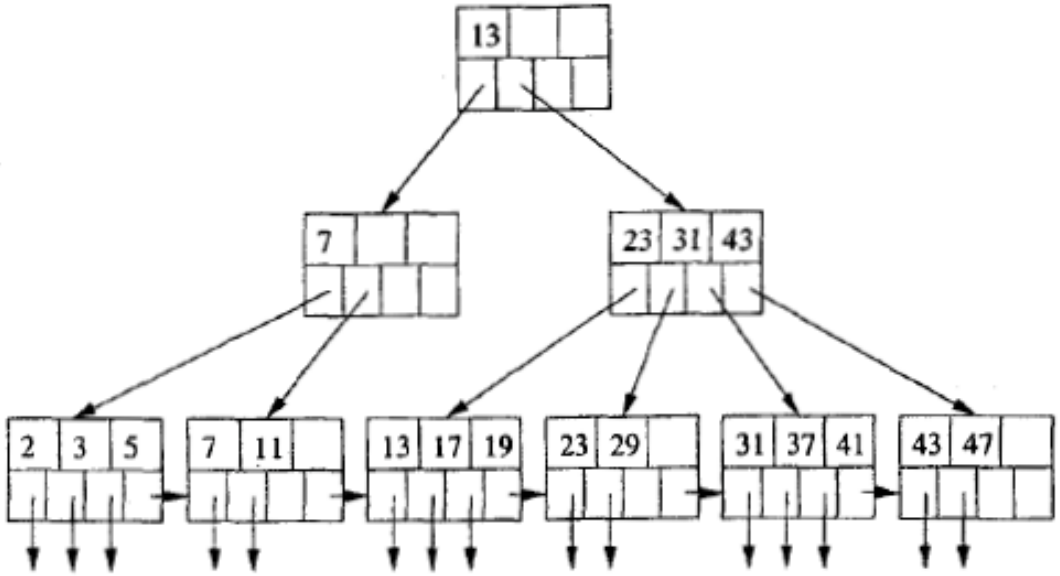
- May be visualized as shown in the figure below.
- Values are usually pointers
  - Information on block for key_i is stored in $value_i$.
  - $value_n$ reserved as a pointer to the next leaf node in the B-Tree

Interior node

- Alternating sequence of node pointers and keys
  - Node pointers are really page pointers (as each node is stored in a block/page).
  - Structure of interior nodes is suspiciously similar to that of an internal node...
  - Number of pointers is always one greater than the number of keys.

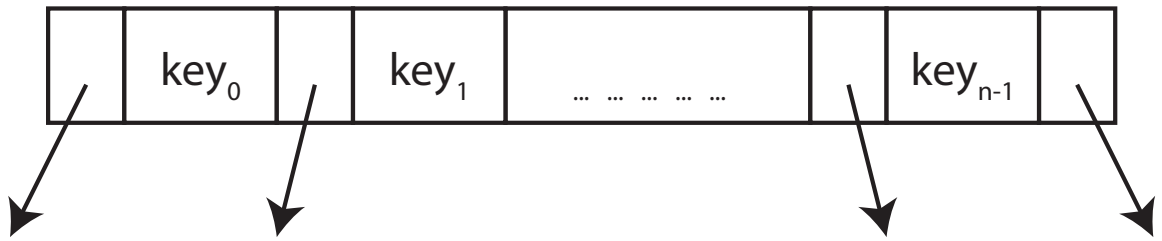


Textbook rendering of a B+ tree



Some constraints on B+ trees

- Nodes are sorted by their keys
  - Leaf nodes are sorted such that we can traverse through the (key, value) pairs in order of the search key
  - Interior nodes are sorted such that leaf nodes are divided yet accessible in sorted order.



To all keys that are < key_0

To all keys that are >= key_0 and < key_1

To all keys that are > key_{n-1}

Some constraints on B+ trees

- The tree is balanced
  - All paths from the root to the leaf nodes must be of equal length
- Nodes are sufficiently filled
  - Nodes may be partially filled with key-value pairs, but ideally they are completely filled.
  - A design parameter for a B+ tree is the **fill factor** of the tree which indicates what the minimal occupancy must be for non-root nodes.
  - Sometimes this can be relaxed and instead applied only to leaf nodes.

B+ tree operations

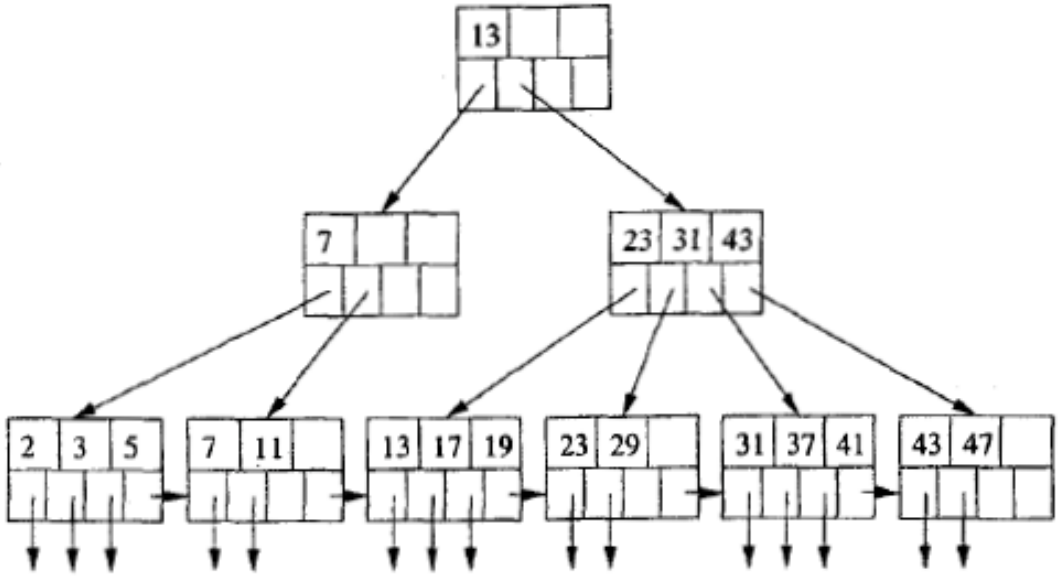
- Search
- Insertion
- Deletion
- We'll focus on just search and insertion
  - Test does describe deletion, but also explains that some implementations do not deal with fixing up leafs with too few key/value pairs.
- Also assumed:
  - Leaf nodes must be at least half full
  - Interior nodes need not meet this constraint

Search

- Goal of search: find leaf node with key **k**.
- Straight forward tree traversal
- Returns the leaf node that must contain **k** (if k is in the index)
- Called to search must therefore look through node p for key/ value pair corresponding to k.

```
LeafNode search(Node p, Key k) {
    if (p is LeafNode) {
        return p; }
    } else if (k < p.key[0]) {
        return search(p.pointer[0], k);
    } else if (k > p.key[n-1]) {
        return search(p.pointer[n], k);
    } else {
        for (i=0; i<n-2; i++) {
            if (p.keys[i] <= k && k < p.keys[i+1]) {
                return search(p.pointer[i+1], k);
            }
        }
    }
}
```

Search



Search for 31?

Search for 42?