Chapter 2
The Information Retrieval Process

Abstract What does an information retrieval system look like from a bird's eye perspective? How can a set of documents be processed by a system to make sense out of their content and find answers to user queries? In this chapter, we will start answering these questions by providing an overview of the information retrieval process. As the search for text is the most widespread information retrieval application, we devote particular emphasis to textual retrieval. The fundamental phases of document processing are illustrated along with the principles and data structures supporting indexing.

2.1 A Bird's Eye View

If we consider the information retrieval (IR) process from a perspective of 10,000 feet, we might illustrate it as in Fig. 2.1.

Here, the user issues a query q from the front-end application (accessible via, e.g., a Web browser); q is processed by a *query interaction* module that transforms it into a "machine-readable" query q' to be fed into the core of the system, a *search and query analysis* module. This is the part of the IR system having access to the *content management* module directly linked with the back-end information source (e.g., a database). Once a set of results r is made ready by the search module, it is returned to the user via the result interaction module; optionally, the result is modified (into r') or updated until the user is completely satisfied.

The most widespread applications of IR are the ones dealing with textual data. As textual IR deals with document sources and questions, both expressed in natural language, a number of textual operations take place "on top" of the classic retrieval steps. Figure 2.2 sketches the processing of textual queries typically performed by an IR engine:

1. The user need is specified via the user interface, in the form of a textual *query* q_U (typically made of keywords).
2. The query q_U is parsed and transformed by a set of textual operations; the same operations have been previously applied to the contents indexed by the IR system (see Sect. 2.2); this step yields a refined query q'_U.
3. Query operations further transform the preprocessed query into a system-level representation, q_S.

S. Ceri et al., *Web Information Retrieval*, Data-Centric Systems and Applications,
DOI 10.1007/978-3-642-39314-3_2,

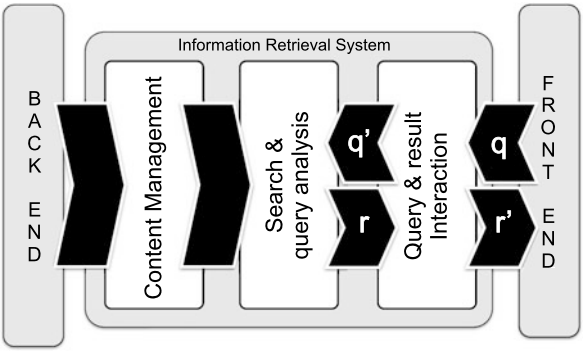

Fig. 2.1 A high-level view of the IR process

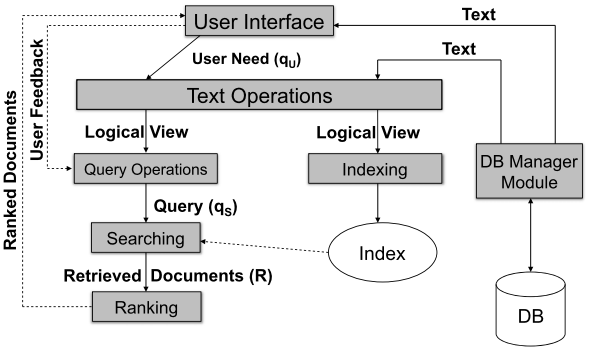

Fig. 2.2 Architecture of a textual IR system. Textual operations translate the user's need into a logical query and create a logical view of documents

4. The query q_S is executed on top of a document source D (e.g., a text database) to retrieve a set of relevant documents, R. Fast query processing is made possible by the index structure previously built from the documents in the document source.
5. The set of retrieved documents R is then ordered: documents are ranked according to the estimated relevance with respect to the user's need.
6. The user then examines the set of ranked documents for useful information; he might pinpoint a subset of the documents as definitely of interest and thus provide feedback to the system.

Textual IR exploits a sequence of text operations that translate the user's need and the original content of textual documents into a logical representation more amenable to indexing and querying. Such a "logical", machine-readable representation of documents is discussed in the following section.

2.1.1 Logical View of Documents

It is evident that on-the-fly scanning of the documents in a collection each time a query is issued is an impractical, often impossible solution. Very early in the history

of IR it was found that avoiding linear scanning requires *indexing* the documents in advance.

The index is a logical view where documents in a collection are represented through a set of *index terms* or *keywords*, i.e., any word that appears in the document text. The assumption behind indexing is that the semantics of both the documents and the user's need can be properly expressed through sets of index terms; of course, this may be seen as a considerable oversimplification of the problem. Keywords are either extracted directly from the text of the document or specified by a human subject (e.g., tags and comments). Some retrieval systems represent a document by the complete set of words appearing in it (logical full-text representation); however, with very large collections, the set of representative terms has to be reduced by means of *text operations*. Section 2.2 illustrates how such operations work.

2.1.2 Indexing Process

The indexing process consists of three basic steps: defining the data source, transforming document content to generate a logical view, and building an index of the text on the logical view.

In particular, data source definition is usually done by a database manager module (see Fig. 2.2), which specifies the documents, the operations to be performed on them, the content structure, and what elements of a document can be retrieved (e.g., the full text, the title, the authors). Subsequently, the text operations transform the original documents and generate their logical view; an index of the text is finally built on the logical view to allow for fast searching over large volumes of data. Different index structures might be used, but the most popular one is the inverted file, illustrated in Sect. 2.3.

2.2 A Closer Look at Text

When we consider natural language text, it is easy to notice that not all words are equally effective for the representation of a document's semantics. Usually, noun words (or word groups containing nouns, also called noun phrase groups) are the most representative components of a document in terms of content. This is the implicit mental process we perform when distilling the "important" query concepts into some representative nouns in our search engine queries. Based on this observation, the IR system also preprocesses the text of the documents to determine the most "important" terms to be used as index terms; a subset of the words is therefore selected to represent the content of a document.

When selecting candidate keywords, indexing must fulfill two different and potentially opposite goals: one is *exhaustiveness*, i.e., assigning a sufficiently large number of terms to a document, and the other is *specificity*, i.e., the exclusion of

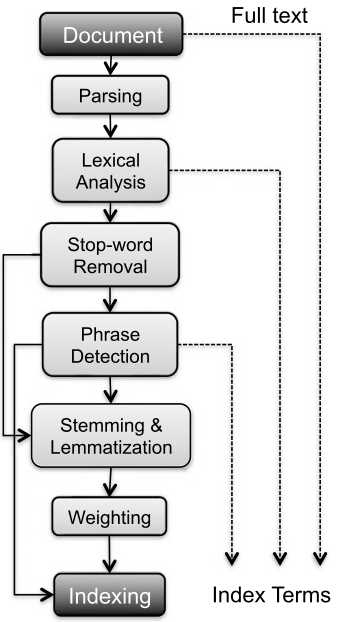

Fig. 2.3 Text processing phases in an IR system

generic terms that carry little semantics and inflate the index. Generic terms, for example, conjunctions and prepositions, are characterized by a low discriminative power, as their frequency across any document in the collection tends to be high. In other words, generic terms have high *term frequency*, defined as the number of occurrences of the term in a document. In contrast, specific terms have higher discriminative power, due to their rare occurrences across collection documents: they have low *document frequency*, defined as the number of documents in a collection in which a term occurs.

2.2.1 Textual Operations

Figure 2.3 sketches the textual preprocessing phase typically performed by an IR engine, taking as input a document and yielding its index terms.

1. *Document Parsing.* Documents come in all sorts of languages, character sets, and formats; often, the same document may contain multiple languages or formats, e.g., a French email with Portuguese PDF attachments. *Document parsing* deals with the recognition and "breaking down" of the document structure into individual components. In this preprocessing phase, unit documents are created; e.g., emails with attachments are split into one document representing the email and as many documents as there are attachments.
2. *Lexical Analysis.* After parsing, *lexical analysis* tokenizes a document, seen as an input stream, into words. Issues related to lexical analysis include the correct

identification of accents, abbreviations, dates, and cases. The difficulty of this operation depends much on the language at hand: for example, the English language has neither diacritics nor cases, French has diacritics but no cases, German has both diacritics and cases. The recognition of abbreviations and, in particular, of time expressions would deserve a separate chapter due to its complexity and the extensive literature in the field; the interested reader may refer to [18, 227, 239] for current approaches.

3. *Stop-Word Removal.* A subsequent step optionally applied to the results of lexical analysis is *stop-word removal*, i.e., the removal of high-frequency words. For example, given the sentence *"search engines are the most visible information retrieval applications"* and a classic stop words set such as the one adopted by the Snowball stemmer,[1] the effect of stop-word removal would be: *"search engine most visible information retrieval applications"*.

   However, as this process may decrease recall (prepositions are important to disambiguate queries), most search engines do not remove them [241]. The subsequent phases take the full-text structure derived from the initial phases of parsing and lexical analysis and process it in order to identify relevant keywords to serve as index terms.

4. *Phrase Detection.* This step captures text meaning beyond what is possible with pure bag-of-word approaches, thanks to the identification of noun groups and other phrases. Phrase detection may be approached in several ways, including rules (e.g., retaining terms that are not separated by punctuation marks), morphological analysis (part-of-speech tagging—see Chap. 5), syntactic analysis, and combinations thereof. For example, scanning our example sentence *"search engines are the most visible information retrieval applications"* for noun phrases would probably result in identifying *"search engines"* and *"information retrieval"*.

   A common approach to phrase detection relies on the use of thesauri, i.e., classification schemes containing words and phrases recurrent in the expression of ideas in written text. Thesauri usually contain synonyms and antonyms (see, e.g., *Roget's Thesaurus* [297]) and may be composed following different approaches. Human-made thesauri are generally hierarchical, containing related terms, usage examples, and special cases; other formats are the associative one, where graphs are derived from document collections in which edges represent semantic associations, and the clustered format, such as the one underlying WordNet's synonym sets or *synsets* [254].

   An alternative to the consultation of thesauri for phrase detection is the use of machine learning techniques. For instance, the Key Extraction Algorithm (KEA) [353] identifies candidate keyphrases using lexical methods, calculates feature values for each candidate, and uses a supervised machine learning algorithm to predict which candidates are good phrases based on a corpus of previously annotated documents.

[1] http://snowball.tartarus.org/algorithms/english/stop.txt.

5. *Stemming and Lemmatization.* Following phrase extraction, *stemming* and *lemmatization* aim at stripping down word suffixes in order to normalize the word. In particular, stemming is a heuristic process that "chops off" the ends of words in the hope of achieving the goal correctly most of the time; a classic rule-based algorithm for this was devised by Porter [280]. According to the Porter stemmer, our example sentence *"Search engines are the most visible information retrieval applications"* would result in: *"Search engin are the most visibl inform retriev applic"*.

   Lemmatization is a process that typically uses dictionaries and morphological analysis of words in order to return the base or dictionary form of a word, thereby collapsing its inflectional forms (see, e.g., [278]). For example, our sentence would result in *"Search engine are the most visible information retrieval application"* when lemmatized according to a WordNet-based lemmatizer.[2]
6. *Weighting.* The final phase of text preprocessing deals with *term weighting*. As previously mentioned, words in a text have different descriptive power; hence, index terms can be weighted differently to account for their significance within a document and/or a document collection. Such a weighting can be binary, e.g., assigning 0 for term absence and 1 for presence. Chapter 3 illustrates different IR models exploiting different weighting schemes to index terms.

2.2.2 Empirical Laws About Text

Some interesting properties of language and its usage were studied well before current IR research and may be useful in understanding the indexing process.

1. *Zipf's Law.* Formulated in the 1940s, Zipf's law [373] states that, given a corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. This can be empirically validated by plotting the frequency of words in large textual corpora, as done for instance in a well-known experiment with the Brown Corpus.[3] Formally, if the words in a document collection are ordered according to a ranking function $r(w)$ in decreasing order of frequency $f(w)$, the following holds:

$$r(w) \times f(w) = c$$

   where c is a language-dependent constant. For instance, in English collections c can be approximated to 10.
2. *Luhn's Analysis.* Information from Zipf's law can be combined with the findings of Luhn, roughly ten years later: "It is here proposed that the frequency of word

[2] See http://text-processing.com/demo/stem/.

[3] The Brown Corpus was compiled in the 1960s by Henry Kucera and W. Nelson Francis at Brown University, Providence, RI, as a general corpus containing 500 samples of English-language text, involving roughly one million words, compiled from works published in the United States in 1961.

occurrence in an article furnishes a useful measurement of word significance. It is further proposed that the relative position within a sentence of words having given values of significance furnish a useful measurement for determining the significance of sentences. The significance factor of a sentence will therefore be based on a combination of these two measurements." [233].

Formally, let $f(w)$ be the frequency of occurrence of various word types in a given position of text and $r(w)$ their rank order, that is, the order of their frequency of occurrence; a plot relating $f(w)$ and $r(w)$ yields a hyperbolic curve, demonstrating Zipf's assertion that the product of the frequency of use of words and the rank order is approximately constant.

Luhn used this law as a null hypothesis to specify two cut-offs, an upper and a lower, to exclude nonsignificant words. Indeed, words above the upper cut-off can be considered as too common, while those below the lower cut-off are too rare to be significant for understanding document content. Consequently, Luhn assumed that the *resolving power* of significant words, by which he meant the ability of words to discriminate content, reached a peak at a rank order position halfway between the two cut-offs and from the peak fell off in either direction, reducing to almost zero at the cut-off points.

3. *Heap's Law.* The above findings relate the frequency and relevance of words in a corpus. However, an interesting question regards how *vocabulary* grows with respect to the *size* of a document collection. Heap's law [159] has an answer for this, stating that the vocabulary size V can be computed as

$$V = KN^{\beta}$$

where N is the size (in words) of the document collection, K is a constant (typically between 10 and 100), and $0 < \beta < 1$ is a constant, typically between 0.4 and 0.6.

This finding is very important for the scalability of the indexing process: it states that vocabulary size (and therefore index size) exhibits a less-than-linear growth with respect to the growth of the document collection. In a representation such as the vector space model (see Sect. 3.3), this means that the dimension of the vector space needed to represent very large data collections is not necessarily much higher than that required for a small collection.

2.3 Data Structures for Indexing

Let us now return to the indexing process of translating a document into a set of relevant terms or keywords. The first step requires defining the text data source. This is usually done by the database manager (see Fig. 2.2), who specifies the documents, the operations to be performed on them, and the content model (i.e., the content structure and what elements can be retrieved). Then, a series of content operations transform each original document into its logical representation; an index of the text is built on such a logical representation to allow for fast searching over large

Table 2.1 An inverted index: each word in the dictionary (i.e., posting) points to a list of documents containing the word (posting list)

Dictionary entry	Posting list for entry
⋮	…
princess	1 3 8 22 41 55 67 68 78 120
⋮	…
witch	1 2 8 30 …
⋮	…
dragon	2 3 4 122 …
⋮	…

volumes of data. The rationale for indexing is that the cost (in terms of time and storage space) spent on building the index is progressively repaid by querying the retrieval system many times.

The first question to address when preparing indexing is therefore what storage structure to use in order to maximize retrieval efficiency. A naive solution would just adopt a *term-document incidence matrix*, i.e., a matrix where rows correspond to terms and columns correspond to documents in a collection C, such that each cell c_{ij} is equal to 1 if term t_i occurs in document d_j, and 0 otherwise. However, in the case of large document collections, this criterion would result in a very sparse matrix, as the probability of each word to occur in a collection document decreases with the number of documents. An improvement over this situation is the *inverted index*, described in Sect. 2.3.1.

2.3.1 Inverted Indexes

The principle behind the inverted index is very simple. First, a *dictionary* of terms (also called a vocabulary or lexicon), V, is created to represent all the unique occurrences of terms in the document collection C. Optionally, the frequency of appearance of each term $t_i \in V$ in C is also stored. Then, for each term $t_i \in V$, called the *posting*, a list L_i, the *posting list* or *inverted list*, is created containing a reference to each document $d_j \in C$ where t_i occurs (see Table 2.1). In addition, L_i may contain the frequency and position of t_i within d_j. The set of postings together with their posting lists is called the *inverted index* or *inverted file* or *postings file*.

Let us assume we intend to create an index for a corpus of fairy tales, sentences from which are reported in Table 2.2 along with their documents.

First, a mapping is created from each word to its document (Fig. 2.4(a)); the subdivision in sentences is no longer considered. Subsequently, words are sorted (alphabetically, in the case of Fig. 2.4(b)); then, multiple occurrences are merged and their total frequency is computed—document wise (Fig. 2.4(c)). Finally, a dictionary is created together with posting lists (Fig. 2.5); the result is the inverted index of Fig. 2.1.

Word	Document ID
once	1
upon	1
a	1
time	1
there	1
lived	1
a	1
beautiful	1
princess	1
the	1
witch	1
cast	1
a	1
terrible	1
spell	1
on	1
the	1
princess	1
the	2
witch	2
hunted	2
the	2
dragon	2
down	2
tThe	2
dragon	2
fought	2
back	2
but	2
the	2
witch	2
was	2
stronger	2

(a) map

Word	Document ID
a	1
a	1
a	1
back	2
beautiful	1
but	2
cast	1
down	2
dragon	2
dragon	2
fought	2
hunted	2
lived	1
on	1
once	1
princess	1
princess	1
spell	1
stronger	2
terrible	1
the	1
the	1
the	2
the	2
the	2
the	2
time	1
there	1
upon	1
was	2
witch	1
witch	2
witch	2

(b) sort

Word	Document ID	Frequency
a	1	3
back	2	1
beautiful	1	1
but	2	1
cast	1	1
down	2	1
dragon	2	2
fought	2	1
hunted	2	1
lived	1	1
on	1	1
once	1	1
princess	1	2
spell	1	1
stronger	2	1
terrible	1	1
the	1	2
the	2	4
time	1	1
there	1	1
upon	1	1
was	2	1
witch	1	1
witch	2	2

(c) merge

Fig. 2.4 Index creation. (**a**) A mapping is created from each sentence word to its document, (**b**) words are sorted, (**c**) multiple word entries are merged and frequency information is added

Inverted indexes are unrivaled in terms of retrieval efficiency: indeed, as the same term generally occurs in a number of documents, they reduce the storage requirements. In order to further support efficiency, linked lists are generally preferred to arrays to represent posting lists, despite the space overhead of pointers, due to their dynamic space allocation and the ease of term insertion.

2.3.2 Dictionary Compression

The Heap law (Sect. 2.2.2(3)) tells us that the growth of a dictionary with respect to vocabulary size is $O(n^{\beta})$, with $0.4 < \beta < 0.6$; this means that the size of the vocabulary represented in a 1 Gb document set would roughly fit in about 5 Mb,

Table 2.2 Example documents from a fairy tale corpus

Document ID	sentence ID	text
1	1	Once upon a time there lived a beautiful princess
	⋮	
1	19	The witch cast a terrible spell on the princess
2	34	The witch hunted the dragon down
	⋮	
2	39	The dragon fought back but the witch was stronger

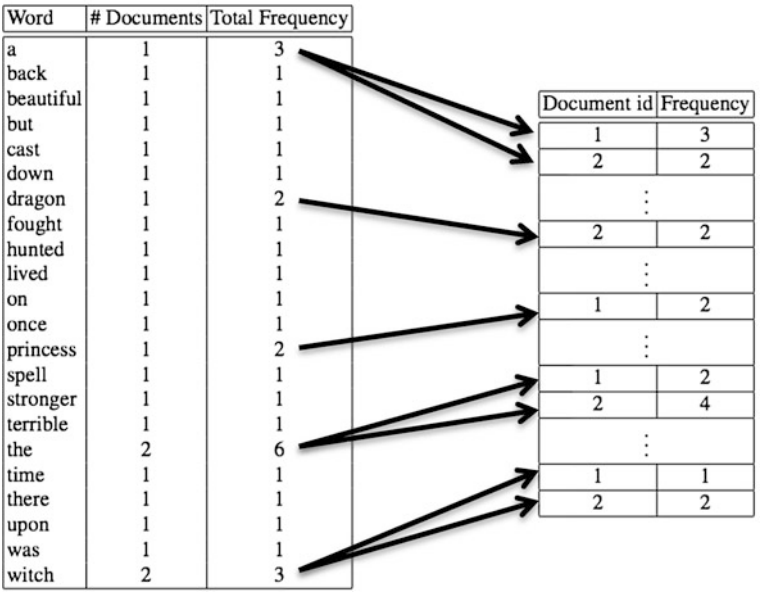

Fig. 2.5 Index creation: a dictionary is created together with posting lists

i.e., a reasonably sized file. In other words, the size of a dictionary representing a document collection is generally sufficiently small to be stored in memory. In contrast, posting lists are generally kept on disk as they are directly proportional to the number of documents; i.e., they are $O(n)$.

However, in the case of very large data collections, dictionaries need to be compressed in order to fit into memory. Besides, while the advantages of a linear index (i.e., one where the vocabulary is a sequential list of words) include low *access time* (e.g., $O(\log(n))$ in the case of binary search) and low space occupation, their construction is an elaborate process that occurs at each insertion of a new document.

To counterbalance such issues, efficient dictionary storage techniques have been devised, including string storage and block storage.

- In *string storage*, the index may be represented either as an array of fixed-width entries or as long strings of characters coupled with pointers for locating terms in such strings. This way, dictionary size can be reduced to as far as one-half of the space required for the array representation.
- In *block storage*, string terms are grouped into blocks of fixed size k and a pointer is kept to the first term of each block; the length of the term is stored as an additional byte. This solution eliminates $k - 1$ term pointers but requires k additional bytes for storing the length of each term; the choice of a block size is a trade-off between better compression and slower performance.

2.3.3 B and B+ Trees

Given the data structures described above, the process of searching in an inverted index structure consists of four main steps:

1. First, the dictionary file is accessed to identify query terms;
2. then, the posting files are retrieved for each query term;
3. then, results are filtered: if the query is composed of several terms (possibly connected by logical operators), partial result lists must be fused together;
4. finally, the result list is returned.

As searching arrays is not the most efficient strategy, a clever alternative consists in the representation of *indexes as search trees*. Two alternative approaches employ *B-trees* and their variant *B+ trees*, both of which are generalizations of binary search trees to the case of nodes with more than two children. In B-trees (see Fig. 2.6), internal (non-leaf) nodes contain a number of keys, generally ranging from d to $2d$, where d is the tree depth. The number of branches starting from a node is 1 plus the number of keys in the node. Each key value K_i is associated with two pointers (see Fig. 2.7): one points directly to the block (subtree) that contains the entry corresponding to K_i (denoted $t(K_i)$), while the second one points to a subtree with keys greater than K_i and less than K_{i+1}.

Searching for a key K in a B-tree is analogous to the search procedure in a binary search tree. The only difference is that, at each step, the possible choices are not two but coincide with the number of children of each node. The recursive procedure starts at the B-tree root node. If K is found, the search stops. Otherwise, if K is smaller than the leftmost key in the node, the search proceeds following the node's leftmost pointer (p_0 in Fig. 2.7); if K is greater than the rightmost key in the node, the search proceeds following the rightmost pointer (p_F in Fig. 2.7); if K is comprised between two keys of the node, the search proceeds within the corresponding node (pointed to by p_i in Fig. 2.7).

The maintenance of B-trees requires two operations: insertion and deletion. When the insertion of a new key value cannot be done locally to a node because it is full (i.e., it has reached the maximum number of keys supported by the B-tree structure), the median key of the node is identified, two child nodes are created,

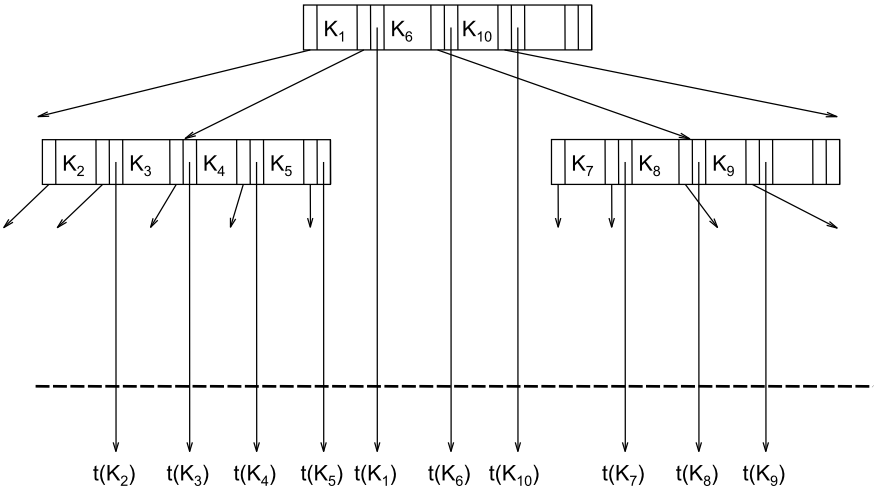

Fig. 2.6 A B-tree. The first key K_1 in the top node has a pointer to $t(K_1)$ and a pointer to a subtree containing all keys between K_1 and the following key in the top node, K_6: these are K_2, K_3, K_4, and K_5

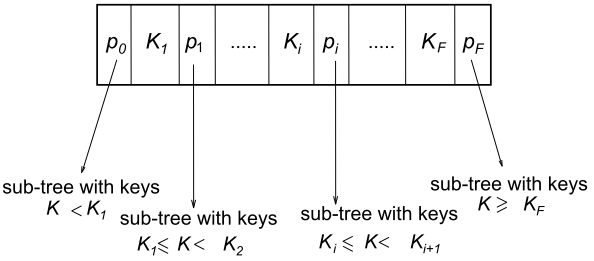

Fig. 2.7 A B-tree node. Each key value K_i has two pointers: the first one points directly to the block that contains the entry corresponding to K_i, while the second points to a subtree with keys greater than K_i and less than $K_i + 1$

each containing the same number of keys, and the median key remains in the current node, as illustrated in Fig. 2.8 (insertion of K_3).

When a key is deleted, two "nearby" nodes have entries that could be condensed into a single node in order to maintain a high node filling rate and minimal paths from the root to the leaves: this is the merge procedure illustrated in Fig. 2.8 (deletion of K_2). As it causes a decrease of pointers in the upper node, one merge may recursively cause another merge.

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

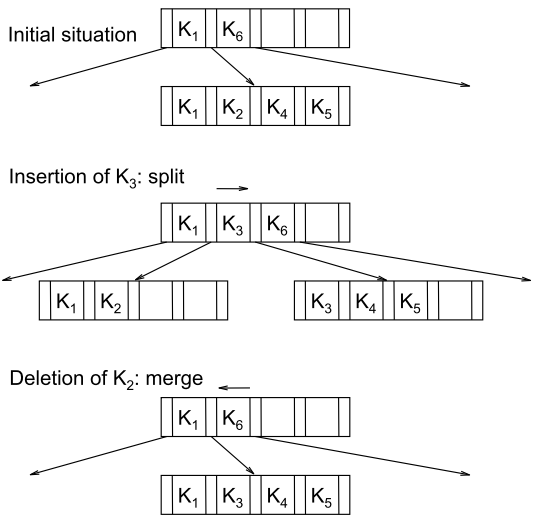

Fig. 2.8 Insertion and deletion in a B-tree

2.3.4 Evaluation of B and B+ Trees

B-trees are widely used in *relational database management systems* (RDBMSs) because of their short access time: indeed, the maximum number of accesses for a B-tree of order d is $O(\log_d n)$, where n is the depth of the B-tree. Moreover, B-trees are effective for updates and insertion of new terms, and they occupy little space.

However, a drawback of B-trees is their poor performance in sequential search. This issue can be managed by the B+ tree variant, where leaf nodes are linked forming a chain following the order imposed by a key. Another disadvantage of B-trees is that they may become unbalanced after too many insertions; this can be amended by adopting rebalancing procedures.

Alternative structures to B-trees and B+ trees include suffix-tree arrays, where document text is managed as a string, and each position in the text until the end is a suffix (each suffix is uniquely indexed). The latter are typically used in genetic databases or in applications involving complex search (e.g., search by phrases). However, they are expensive to construct, and their index size is inevitably larger than the document base size (generally by about 120–240 %).

2.4 Exercises

2.1 Apply the Porter stemmer[4] to the following quote from J.M. Barrie's *Peter Pan*:

> When a new baby laughs for the first time a new fairy is born, and as there are always new babies there are always new fairies.

[4] http://tartarus.org/~martin/PorterStemmer/.

Table 2.3 Collection of documents about information retrieval

Document	content
D_1	information retrieval students work hard
D_2	hard-working information retrieval students take many classes
D_3	the information retrieval workbook is well written
D_4	the probabilistic model is an information retrieval paradigm
D_5	the Boolean information retrieval model was the first to appear

How would a representation of the above sentence in terms of a bag of stems differ from a bag-of-words representation? What advantages and disadvantages would the former representation offer?

2.2 Draw the term-incidence matrix corresponding to the document collection in Table 2.3.

2.3 Recreate the inverted index procedure outlined in Sect. 2.3.1 using the document collection in Table 2.3.

2.4 Summarize the practical consequences of Zip's law, Luhn's analysis, and Heap's law.

2.5 Apply the six textual transformations outlined in Sect. 2.2.1 to the text in document D_2 from Table 2.3. Use a binary scheme and the five-document collection above as a reference for weighting.

http://www.springer.com/978-3-642-39313-6

Web Information Retrieval

Ceri, S.; Bozzon, A.; Brambilla, M.; Della Valle, E.;
Fraternali, P.; Quarteroni, S.

2013, XIV, 284 p., Hardcover
ISBN: 978-3-642-39313-6