Chapter 7
B-Tree Traversals

The B-tree, or, more specifically, the $B^+$-tree, is the most widely used physical database structure for primary and secondary indexes on database relations. Because of its balance conditions that must be maintained under all circumstances, the B-tree is a highly dynamic structure in which records are often moved from one page to another in structure modifications such as page splits caused by insertions and page merges caused by deletions. In a transaction-processing environment based on fine-grained concurrency control, this means that a data page can hold uncommitted updates by several transactions at the same time and an updated tuple can migrate from a page to another while the updating transaction is still active.

In this chapter we show how the B-tree is traversed when performing a read, insert, or delete action under the key-range locking protocol described in Sect. 6.4. The traversal algorithms to be presented augment the read, insert, and delete algorithms presented in Sect. 6.7. The latching protocol applied in the B-tree traversals is deadlock-free, ensuring that no deadlocks can occur between latches held by different process threads and that no deadlock can occur between a latch and a lock. Repeated work done for the traversals is minimized by heavy use of saved paths, which often make it possible to start a new traversal by the same thread at some lower-level page in the previously traversed path, rather than at the root page. Saved paths thus provide a simple means of shortening the access-path-length of database actions.

The B-tree structure we consider is the very basic one in which no sideways links are maintained and in which deletions are handled uniformly with insertions. The management of this basic B-tree structure seems to be the most challenging as compared to some B-tree variants such as those with sideways links maintained at the leaf level or to those in which the balance conditions are relaxed by allowing pages to become empty until they are detached from the B-tree and deallocated. One motivation of our approach is that for write-optimized B-trees (to be discussed in Chap. 15), sideways-linking is not feasible at all.
### 7.1 Sparse B-Tree Indexes

We assume that the physical database used to implement our logical database, that is, the relation $r(X,V)$, is a sparse B-tree defined as follows:

1. The sparse B-tree is a tree whose nodes are database pages and whose root-to-leaf paths are all of the same length; this length is called the **height** of the B-tree.
2. Each B-tree page $p$ stores records with keys $x$ satisfying
   \[ \text{low-key}(p) \leq x < \text{high-key}(p), \]
   where $\text{low-key}(p)$ and $\text{high-key}(p)$ are fixed keys, called the **low key** and **high key**, respectively, of page $p$. The low and high keys are not stored in the page; the low key may appear as the least key in the records stored in the page, but this need not be the case: the low key is just a lower bound on the keys stored in the page. We say that page $p$ covers the half-open key range $[\text{low-key}(p), \text{high-key}(p))$.
3. The **level** of a B-tree page is one for a leaf page and one plus the level of its child pages for a non-leaf page. For each level $l = 1, \ldots, h$, where $h$ is the height of the B-tree, the key ranges covered by the sequence of pages $p_1, \ldots, p_n$ at level $l$ form a disjoint partition of the entire key space $[-\infty, \infty)$, that is,
   \[
   \text{low-key}(p_1) = -\infty, \\
   \text{low-key}(p_i) < \text{high-key}(p_i) = \text{low-key}(p_{i+1}), \text{ for } i = 1, \ldots, n - 1, \\
   \text{and } \text{high-key}(p_n) = \infty.
   \]
4. The leaf pages of the B-tree are data pages and store the tuples of $r$.
5. The non-leaf pages $p$ of the B-tree are index pages and store **index records** of the form $(\text{low-key}(q), q)$, where $q$ is the page-id of a child page of page $p$. There is one index record for each child page of page $p$. The child pages of a B-tree page are ordered in ascending order by the low keys of the child pages.

We say that such a tree structure $b$ is a **sparse B-tree index on** relation $r$ and that $r$ is the **logical database indexed by** B-tree $b$, denoted $r = \text{db}(b)$.

#### Example 7.1

Figure 7.1 shows (a part of) a sparse B-tree index on relation $r(X,V)$ with tuples with keys 1, 3, 5, 6, 7, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 24, 25, 26, 27, etc. We assume (unrealistically) that only six tuples fit in a page. Pages $p_4$ to $p_9$ are leaf pages, $p_2$ and $p_3$ are index pages of height 2, and $p_1$ (of height 3) is the root page.

We call a B-tree as defined above a **consistent B-tree**. This basic definition states how the pages of the B-tree are organized as a tree structure and what key ranges are covered by each page. The following lemma states the most important properties of a consistent B-tree: