B-Trees

**Aim:** To describe the B-tree data structure for implementing large dictionaries, and the algorithms to search, insert and delete keys from it.

We can generalise binary search trees to **multiway search trees**, which have more than one key in each internal node.

The keys are ordered within each node, and for every adjacent pair of keys there is a subtree containing keys between that pair. There is also a subtree containing keys smaller than the smallest key in the node, and one containing keys larger than the largest key in the node.

---

### B-trees

A **B-tree of order** $m$ is a multi-way search tree with the following properties:

- each node has at most $m$ children and $m-1$ keys.
- each internal node (except possibly the root) has at least $b = \text{ceiling}(m/2)$ children (and $b-1$ keys).
- all external nodes have the same depth.
Here is an order 5 B-tree \((m = 5, b = 3)\). Ignore the external nodes for the present; just consider the keys.

A B-tree node is called a **page**.

The properties ensure that each non-root node is at least half full, so no more than 50% of space is wasted.

---

**Worst-case structure**

In the worst case the root contains only one key, and has two children. All other pages have \(b = \text{ceiling}(m/2) - 1\) keys and \(b\) children.

<table>
<thead>
<tr>
<th>Height of B-tree</th>
<th>Number of keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1 + 2(b-1)</td>
</tr>
<tr>
<td>3</td>
<td>1 + 2(b-1) + 2(b-1)b</td>
</tr>
<tr>
<td>(h)</td>
<td>1 + 2(b-1) ((1 + b + b^2 + b^3 + \ldots + b^{h-2}))</td>
</tr>
</tbody>
</table>

The last summation reduces to \(2b^{h-1}-1\).

Hence, a B-tree with \(n\) keys has a height at most \(1 + \log_b((n+1)/2)\). This means that search time is \(O(\log n)\).

---

B-trees are often used as representations for very large dictionaries. There are two main ways to store the items associated with each key:

1. Combine the key and item into an entry type, store in the internal nodes. External nodes contain no data (like binary search trees).
2. Store keys only in internal nodes, push the items into external nodes.
For databases (essentially dictionaries on secondary storage) the order is quite large (256 or more) so that a page fills a disk block. The second arrangement above (items in external nodes) is preferred so that the order can be made as large as possible for the same page size.

The way the items are usually stored is illustrated in the example tree:

- External nodes are referenced by leaf pages.
- The number of external nodes is exactly one more than the number of keys (a property of trees). Consequently there are sufficient external nodes to store items for every key and have one external node left over.
- If a leaf page has \( k \) keys, the leftmost \( k \) external node references are associated with these \( k \) keys. The last, rightmost reference is associated with a key from an ancestor page.

---

**Searching a B-tree**

The B-tree search algorithm is a generalisation of the algorithm for binary trees:

Given a page \( p \) and a search key \( k \),

1. find \( i \) such that \( p\text{.keys}[i-1] < k \leq p\text{.keys}[i] \)
2. if \( (p\text{.keys}[i] != k) \)
   - if \( (p\text{.is_leaf}) \)
     - return unsuccessful search indication
   - else
     - reinitialise \( p \) from \( p\text{.subtree}[i] \)

How \( p \) is (re)-initialised will depend on whether the tree is in a file or not.

The search through the \( \text{keys} \) array would normally be a binary search since the array is in order. The goal can always be established if we imagine element \(-1\) to be minus infinity and element \( n\text{.keys} \) to be plus infinity. Importantly, if the key is not in the array, the value of \( i \) indicates where it should be inserted.
How do we locate the item associated with \( k \)? We simply continue the search, as if we hadn't found the key. We will continue tracing a path to a leaf node, still using a binary search. There we stop: \texttt{subtree}[i] from the leaf-node search points to the required item.

For example, in the example B-tree, if \( k = 22 \) we perform these steps:

1. Search for 22 in the root page \texttt{keys} array, set \( i = 0 \)
2. Follow \texttt{subtree}[0] to the leftmost of the two children
3. Search for 22 in the array [10,22,42], set \( i = 1 \)
4. Note key found; continue search with \texttt{subtree}[1]
5. Search for 22 in [12,18], set \( i = 2 \) (since 22 > 18)
6. Last page was a leaf, \texttt{subtree}[2] is the required item.

---

**B-tree insertion**

Insertions are always made into leaf nodes: the internal nodes at the bottom of tree. Insertion follows an unsuccessful search as described above. Two cases can occur, depending on whether the leaf page is full or not: Considering these in detail,

1. If there is room in the page, larger entries are moved one place to the right and the new entry inserted (\emph{e.g.} insert 15 in above tree). Items are shuffled the same way:

2. Otherwise, we have \( m \) entries in a node that can hold only \( m-1 \). \( \emph{e.g.} \) insert 6
   We split the page into three pieces:
   - A page containing keys less than the median;
   - The median and its associated item; and
   - A page containing keys greater than the median.
Again the items are shuffled around.

![Diagram of B-tree insertion and splitting](image)

The median is inserted in the parent of the split node, and the new pages become the children before and after it. (Note that the new nodes will have at least $b$ entries.)

If the parent node does not have enough room then it too will be split. For example, after inserting 37 into the tree resulting from the above insertion one leaf contains $[29 \ 36 \ 37 \ 40]$. Inserting 39 causes it to split: the median (37) then causes the parent $[5 \ 10 \ 22 \ 42]$ to split and its median (22) to move up.

Splits can continue up the tree until the root itself is split, creating a new root with a single entry. This is the only way that a B-tree can increase in height.

---

**B-tree deletion**

If the entry to be deleted is not in a leaf, just as with deletion from a binary tree, we can replace it with the largest element in its left subtree (which will be in a leaf) and then delete that element.

Thus, all deletions occur initially in leaves. The first step is to delete the key and item, moving the larger keys and items left one place. What happens next depends on the number of keys remaining in the page.
1. The page has at least $b-1$ keys: we are finished.

Otherwise the page is deficient: it has only $b-2$ keys. How we correct the deficiency depends on whether the page has a sibling with more than minimum size ($b-1$ keys). If so,

2. "borrow" a key from the sibling via the parent. Move the parent key down to the deficient node, replacing it from the largest (smallest) key in the left (right) sibling:

For example, if 12 were deleted from the example tree, then we can either

1. move the 10 down and move the 7 up from the left sibling; or
2. move the 22 down and move the 29 up from the right sibling.

It is also necessary to move the subtrees.

Finally, what if neither sibling is more than half full? Then,

3. the page and either sibling are concatenated or coalesce, absorbing the parent key. Concatenation is the opposite of splitting. It removes a key from the parent page, so one of (1), (2) or, recursively, (3) is needed to ensure the resulting page is not deficient.

Consider deleting 75 from the example tree. It can first be replaced by 67, its inorder predecessor (Fig. 11.20a). The deficient leaf node has only siblings with minimum size. It is concatenated with its right sibling (Fig. 11.20b). Removing the parent key (67) causes it to become deficient as well, requiring a "borrow" operation from the left (Fig. 11.20c):
In the extreme case, pages may concatenate at every level. If the last key in the root is deleted, its remaining child becomes the new root. This is the only way that the tree can decrease in height.

B-trees can be used as efficient main-memory data structures, but their most widespread application is with file indexing:

1. The large branching factor (= small height) mean that few disk accesses are needed to find a key and associated data.
2. Insertions and deletions can be performed efficiently (although with fairly complex algorithms).
3. For fixed data, the data file can be kept in sorted order, allowing both ordered sequential access and random access.

Strategies for storing items
The data associated with keys can be stored in several ways:

1. in each node along with the key
2. in external nodes (as shown in examples)
3. in a separate file with references (disk addresses) from the B-tree.

As with hash indexes, the page (node) size should be matched to the prevailing disk block size.

Example:

If the keys are 12 bytes long, and data and subtree references use 4-byte disk addresses, what order B-tree could be designed to match a disk block size of 1024 bytes? What is the height of a worst-case B-tree holding 100000 keys? How many disk accesses would be needed to return the data associated with any of these keys?

Answers:

Let \( m \) be the B-tree order. A page consists of \( m-1 \) keys, each of 12 bytes, \( m \) subtrees or data references (4 bytes each) a counter (up to 2 bytes), and 1 byte to record whether the page is a leaf node. Thus

\[
12(m-1) + 4m + 2 + 1 \leq 1024
\]

so

\[
16m - 9 \leq 1024
\]

or

\[ m = \text{trunc}(1033/16) = 64 \]

Caching

The minimum number of keys in a normal page is \( b-1 = 64/2-1 = 31 \).

If the root contains one key, then a 2-level tree contains at least \( 1 + 2*31 = 63 \) keys, a height-3 tree \( 63 + 2*32*31 = 2047 \) keys, and a tree of height \( 42047 + 2*32^2*31 = 65535 \) keys. We thus need a 5-level tree.

Since all operations go through the root, the root page is kept in main memory. Thus to access the data from a tree of height \( h \), we need \( h-1 \) disk accesses for the index and one more for the data itself. Our tree of height 5 thus requires 5 accesses in all.

B+ trees

Can we squeeze even more out of a B-tree index?

Take the data records in key order, and store several records per block, linking the blocks together symmetrically (forward and reverse references).
Now create a B-tree index, inserting only the largest key into the index. Since the keys in a block are clustered, we will be able to locate any of them by following the index.

This structure, with representative keys only stored in the index and data in linked data blocks, is called a B+-tree.

B+-trees permit insertion, deletion and searching in $O(\log n)$ time, although again considerable effort is needed to maintain both the index and the data file.

Since 1980 almost all database systems use some variation on the B-tree or B+-tree structure for data storage.