B-Trees
Binomial Heaps
Fibonacci Heaps

18 B-Trees

• B-trees are similar to RBTs, but they are better at minimizing disk I/O operations
• Many database systems use B-trees, or variants of them, to store information
• B-tree nodes may have many children, from a few to thousands
• The branching factor of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used
- B-trees are similar to RBTs in that every \( n \)-node B-tree has height \( O(\lg n) \)
- The exact height of a B-tree can be considerably less than that of a RBT, because its branching factor – the base of the logarithm that expresses its height – can be much larger
- Therefore, we can also use B-trees to implement many dynamic-set operations in time \( O(\lg n) \)
- B-trees generalize BSTs in a natural manner
  - If an internal B-tree node \( x \) contains \( x \cdot n \) keys, then \( x \) has \( x \cdot n + 1 \) children
18.1 Definition of B-trees

A B-tree $T$ is a rooted tree (whose root is $T.\text{root}$) having the following properties:

1. Every node $x$ has the following attributes:
   a) $x.n$, the number of keys currently stored in $x$,
   b) the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$,
   c) $x.leaf$, a boolean value that is TRUE if $x$ is a leaf and FALSE if $x$ is an internal node

2. Each internal node $x$ also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their $c_i$ attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$

4. All leaves have the same depth, which is the tree’s height $h$
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer \( t \geq 2 \) called the **minimum degree** of the B-tree:

a) Every node (except the root) must have at least \( t - 1 \) keys. Every internal node (except the root) thus has at least \( t \) children. If the tree is nonempty, the root must have at least one key.

b) Every node may contain at most \( 2t - 1 \) keys. Therefore, an internal node may have at most \( 2t \) children. We say that a node is **full** if it contains exactly \( 2t - 1 \) keys.

### The height of a B-tree

- The simplest B-tree occurs when \( t = 2 \)
- Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree
- In practice, however, much larger values of \( t \) yield B-trees with smaller height

**Theorem 18.1**  
\[ h \leq \log_t \frac{n + 1}{2} \]  

 *_If \( n \geq 1 \), then for any \( n \)-key B-tree \( T \) of height \( h \) and minimum degree \( t \geq 2 \),*
Proof The root of a B-tree $T$ contains at least one key, and all other nodes contain at least $t - 1$ keys. Thus, $T$, whose height is $h$, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth $h$ it has at least $2t^{h-1}$ nodes. Thus, the number $n$ of keys satisfies the inequality
\[ n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right) = 2t^h - 1 \]
We get $t^h \leq (n + 1)/2$. Taking base-$t$ logarithms of both sides proves the theorem.
18.2 Basic operations on B-trees

- Searching a B-tree, at each internal node \( x \), we make an \((x.n + 1)\)-way branching decision
- B-TREE-SEARCH is a simple generalization of the TREE-SEARCH procedure defined for BSTs
- B-TREE-SEARCH inputs are a pointer to the root node \( x \) and key \( k \) to be searched in that subtree
- The top-level call is B-TREE-SEARCH\((T\. root, k)\)
- If \( k \) is in the tree, it returns the ordered pair \((y, i)\) of a node \( y \) and an index \( i \) s.t. \( y.key_i = k \)

B-TREE-SEARCH\((x, k)\)

1. \( i = 1 \)
2. while \( i \leq x.n \) and \( k > x.key_i \)
3. \( i = i + 1 \)
4. if \( i \leq x.n \) and \( k == x.key_i \)
5. return \((x, i)\)
6. elseif \( x.leaf \)
7. return NIL
8. else DISK-READ\((x.c_i)\)
9. return B-TREE-SEARCH\((x.c_i, k)\)
As in the TREE-SEARCH procedure for BSTs, the nodes encountered during the recursion form a simple path downward from the root of the tree.

B-TREE-SEARCH accesses $O(h) = O(\log_t n)$ disk pages, where $h$ is the height of the B-tree and $n$ is the number of keys.

Since $x \cdot n < 2t$, the while loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

- ALLOCATE-NODE allocates one disk page to be used as a new node in $O(1)$ time.
- It requires no DISK-READ, since there is as yet no useful information stored on the disk.

B-TREE-CREATE($T$)
1. $x = \text{ALLOCATE-NODE}()$
2. $x.\text{leaf} = \text{TRUE}$
3. $x.\text{n} = 0$
4. $\text{DISK-WRITE}(x)$
5. $T.\text{root} = x$
Inserting a key into a B-tree

- We insert the new key into an existing leaf node.
- We need an operation that splits a full node \( y \) (having \( 2t - 1 \) keys) around its median key \( y.\text{key}_t \) into two nodes having only \( t - 1 \) keys.
- Median key moves up into \( y \)’s parent to identify the dividing point between the two new trees.
- But if \( y \)’s parent is also full, we must split it before we can insert the new key; we could end up splitting full nodes all the way up the tree.

As with a BST, we can insert a key into a B-tree in a single pass down from the root to a leaf.
- We do not wait to find out whether we will actually need to split a full node in order to do the insertion.
- As we travel down the tree, we split each full node we come to along the way (including the leaf itself).
- Thus whenever we want to split a full node \( y \), we are assured that its parent is not full.
Splitting a node in a B-tree

- **B-TREE-SPLIT-CHILD** takes as input a *nonfull* internal node \( x \) (in main memory) and an index \( i \) such that \( x.c_i \) (in main memory) is full.
- The procedure then splits this child in two and adjusts \( x \) so that it has an additional child.
- To split a full root, we will first make the root a child of a new empty root node, so that we can use **B-TREE-SPLIT-CHILD**.
- The tree thus grows in height by one; splitting is the only means by which the tree grows.
B-TREE-SPLIT-CHILD\((x,t)\)
1. \(z = \text{ALLOCATE-NODE()}\)
2. \(y = x.c_t\)
3. \(z.leaf = y.leaf\)
4. \(z.n = t - 1\)
5. for \(j = 1\) to \(t - 1\)
6. \(z.key_j = y.key_{j+t}\)
7. if not \(y.leaf\)
8. for \(j = 1\) to \(t\)
9. \(z.c_j = y.c_{j+t}\)
10. \(y.n = t - 1\)
11. for \(j = x.n + 1\)
12. \(x.c_{j+1} = x.c_j\)
13. \(x.c_{i+1} = z\)
14. for \(j = x.n\)
15. \(x.key_{j+1} = x.key_j\)
16. \(x.key_t = y.key_t\)
17. \(x.n = x.n + 1\)
18. \(\text{DISK-WRITE}(y)\)
19. \(\text{DISK-WRITE}(z)\)
20. \(\text{DISK-WRITE}(x)\)

B-TREE-INSERT\((T,k)\)
1. \(r = T.root\)
2. if \(r.n == 2t - 1\)
3. \(s = \text{ALLOCATE-NODE()}\)
4. \(T.root = s\)
5. \(s.leaf = \text{FALSE}\)
6. \(s.n = 0\)
7. \(s.c_1 = r\)
8. B-TREE-SPLIT-CHILD\((s,1)\)
9. B-TREE-INSERT-NONFULL\((s,k)\)
10. else B-TREE-INSERT-NONFULL\((r,k)\)
B-TREE-INSERT-NONFULL\( (x, k) \)
1. \( i = x.n \)
2. if \( x.leaf \)
3. while \( i \geq 1 \) and \( k < x.key_i \)
4. \( x.key_{i+1} = x.key_i \)
5. \( i = i - 1 \)
6. \( x.key_{i+1} = k \)
7. \( x.n = x.n + 1 \)
8. DISK-WRITE\( (x) \)
9. else while \( i \geq 1 \) and \( k < x.key_i \)
10. \( i = i - 1 \)
11. \( i = i + 1 \)
12. DISK-READ\( (x.c_i) \)
13. if \( x.c_i.n == 2t - 1 \)
14. B-TREE-SPLIT-CHILD\( (x, i) \)
15. if \( k > x.key_i \)
16. \( i = i + 1 \)
17. B-TREE-INSERT-
NONFULL\( (x.c_i, k) \)
- For a B-tree of height $h$, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL.
- The total CPU time used is $O(th) = O(t \log_t n)$.
- B-TREE-INSERT-NONFULL is tail-recursive, and can alternatively be implemented as a `while` loop.
  - the number of pages that need to be in main memory at any time is $O(1)$.