B-Trees

B-Trees are useful in the following cases:

The number of objects is too large to fit in memory.

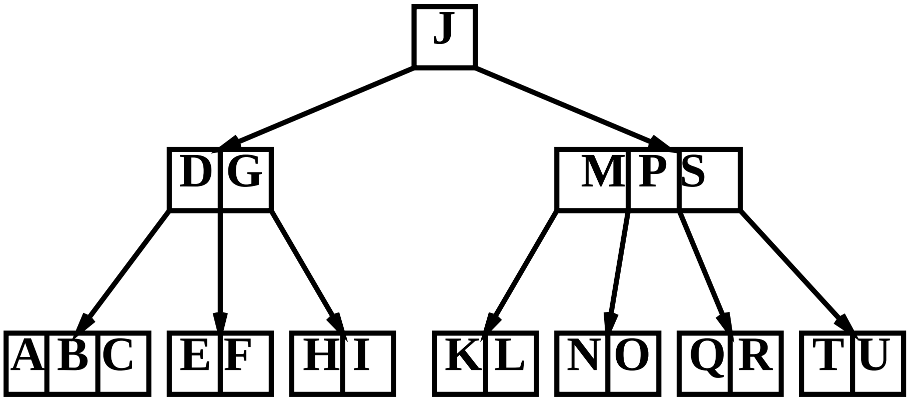
Need external storage.

Disk accesses are slow, thus need to minimize the number of disk accesses.

Red-Black trees are not good in these situations, only retrieves one key at a time from memory.

B-Trees

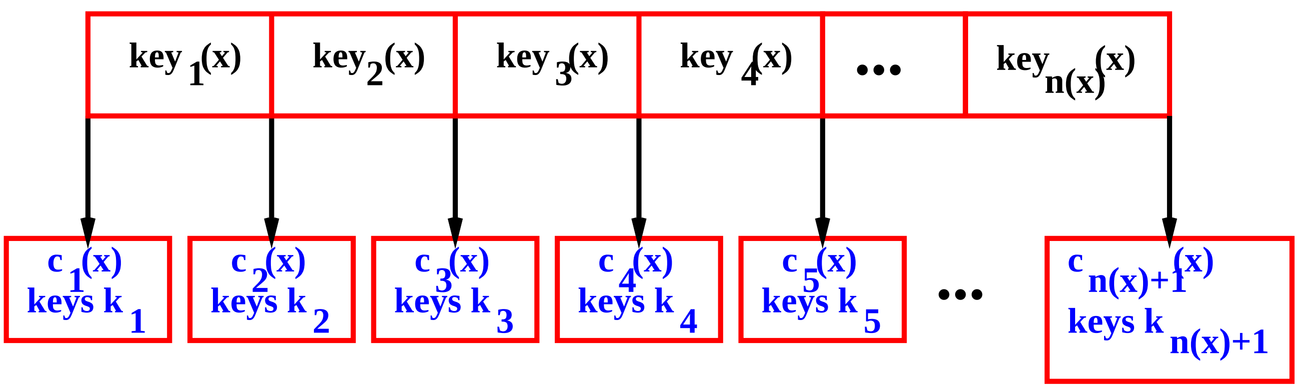
- B-Trees are balanced, like RB trees.
- They have a large number of children (large branching factor), unlike RB trees.
- The branching factor is determined by the size of disk transfers (page size).
- Each object (node) referenced requires a DiskRead.
- Each object modified requires a DiskWrite.
- The root of the tree is kept in memory at all times.
- Insert, Delete, Search = O(h), where h is the height of the tree. O(lgn), though much less in reality ($log_{BF}n$).



Properties of B-Trees

1. Node x

$$\text{n(x)} = \#\text{keys stored here}$$
$$\text{leaf(x)} = \text{true if leaf node}$$



$$k_1 \le key_1(x) \le k_2 \le key_2(x) \le \ldots \le key_{n(x)}(x) \le k_{n(x)+1}$$

Properties of B-Trees

2. Every leaf has the same depth equal to the height of the tree.

3. The number of keys is bounded in terms of the minimum degree t $\geq$ 2.

n(x) $\geq$ t-1 (except root $\geq$ 1)

#children(x) $\geq$ t (except root $\geq$ 0), leaves = 0

n(x) $\leq$ 2t - 1

#children $\leq$ 2t (except leaves which = 0)

If n(x) = 2t - 1 then n is a ____________.

For example, if t = 3:

- Root: n(x) = ________, #children = ________________
- Internal node: n(x) = ________, #children = ________________
- Leaf: n(x) = ________, #children = _________

What is h in terms of n and t?

Theorem 19.1

Given n $\geq$ 1, t $\geq$ 2, B-Tree of height h and minimum degree t, and number of keys n,

$$h \leq \ log_t \frac{n+1}{2}$$

Proof:

n $\geq$ minimum #nodes in tree of height h and minimum degree t

The minimum #nodes means root has one key (two children) and other nodes have t-1 (minimum) keys.

$$= 1 \text{ key at root} +$$
$$2(t-1) \text{ at depth } 1 +$$
$$2t(t-1) \text{ at depth } 2 +$$
$$2t^2(t-1) \text{ at depth } 3 + ...$$
$$= 1 + (t-1)\, \Sigma_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\, \Sigma_{i=0}^{h-1} t^i$$
$$= 1 + 2(t-1)(\frac{t^h - 1}{t-1})$$
$$= 1 + 2(t^h - 1)$$
$$= 2t^h - 1$$

$$n \geq 2t^h - 1$$
$$2t^h \leq n+1$$
$$t^h \leq \frac{n+1}{2}$$
$$log_t t^h \leq log_t \frac{n+1}{2}$$
$$h \leq log_t \frac{n+1}{2}$$

Operations

- Root always in memory
  - Never read
  - Write only when modified
- Nodes passed to operations must have been Read
- All operations go from root down in one pass, O(h)

Search

This is a generalization of binary tree search.

```
Search(x, k)
    if k in node x
    then return x and i such that key_i(x) = k
    else if x is a leaf
        then return NIL
        else find i such that key_{i-1}(x) < k < key_i(x)
            DiskRead(child_i(x))
            return Search(child_i(x), k)
```

Click mouse to advance to next frame.

Search

- Node size should be ___________ disk page size.
- Disk Accesses = $\Theta(\log_t n)$, where n is #keys in B-tree
- Run Time = O(th) = O(t $\log_t n$) = O(lg n), if t is constant

Example

Disk page size = 2048 bytes
4 bytes per key, 4 bytes per pointer, 4 bytes extra
Full node has (2t - 1) keys and 2t child pointers: 16t bytes per node
16t = 2048, t = 128

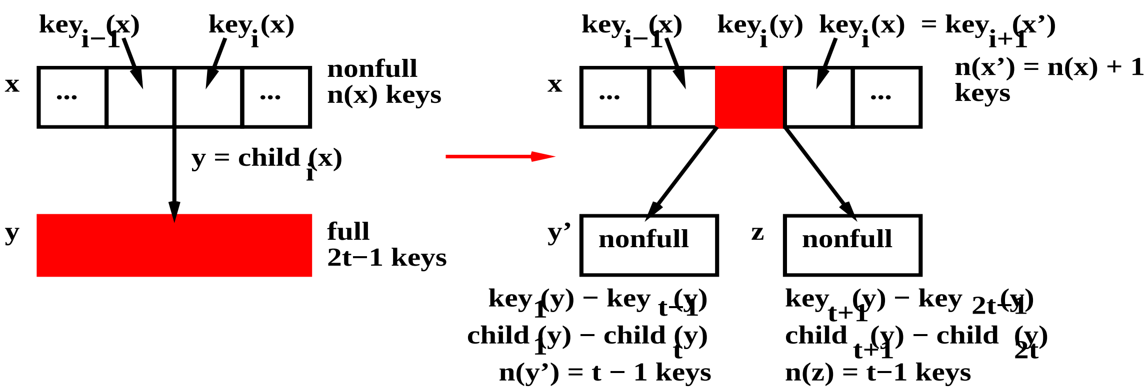
Insert

- If node x is a non-full (< 2t-1 keys) leaf, then insert new key k in node x

- If node x is non-full but not a leaf, then recurse to appropriate child of x
- If node x is full (2t-1 keys), then "split" the node into x_1 and x_2, and recurse to appropriate node x_1 or x_2.

In this example t = 2.
Click mouse to advance to next frame.

Splitting: B-Tree-Split-Child(x, i, y)



Note: If y is root(T), then allocate node x and link to y before calling split.

Splitting: B-Tree-Split-Child(x, i, y)

B-Tree-Split-Child(x, i, y)   ; x is parent, y is child in ith subtree
    Allocate(z)   ; n(z)=t-1, leaf(z) = leaf(y)
    Copy y's second half keys and children to z
    n(y) = t-1
    Shift x's keys and children one to the right from i

$\text{child}_{i+1}(x) = z$
$\text{key}_i(x) = \text{key}_t(y)$
$n(x) = n(x) + 1$
Write(x)
DiskWrite(y)
DiskWrite(z)

Running time is $\Theta(t)$ with 3 disk writes

Insert: B-Tree-Insert(T, k)

- Start at root(T) moving down the tree looking for the proper leaf to put k
- Split all full nodes along the way

```
B-Tree-Insert(T, k)
    r = root(T)
    if n(r) = 2t-1                    ; full
    then allocate empty node s pointing to r
        B-Tree-Split-Child(s, 1, r)
        B-Tree-Insert-Nonfull(s, k)
    else B-Tree-Insert-Nonfull(r, k)

B-Tree-Insert-Nonfull(x, k)
    if leaf(x)
    then shift keys of x higher than k one to the right
        put k in appropriate spot
        n(x) = n(x) + 1
        DiskWrite(x)
    else find smallest i such that k < key_i(x)
```

```
DiskRead(child_i(x))
if n(child_i(x)) = 2t - 1 ; full
then B-Tree-Split-Child(x, i, child_i(x))
     if k > key_i(x)
     then i = i + 1      ; adjust due to new node entry from child
B-Tree-Insert-Nonfull(child_i(x), k)
```

Disk Accesses: O(h)

Run Time: $O(th) = O(t \log_t n) = O(\lg n)$, if t constant

Example

Click mouse to advance to next frame.

Note how B-Trees grow from the top, not from the bottom like BSTs or RBTs.

Deletion: B-Tree-Delete(x, k)

- Search down tree for node containing k
- When B-Tree-Delete is called recursively, the number of keys in x must be at least the minimum degree t (the root can have $< t$ keys)
- If x is a leaf, just remove key k and still have at least t-1 keys in x
- If there are not $\geq$ t keys in x, then borrow keys from other nodes.

Deletion

There are three general cases:

[Case 1:] If key k in node x and x is a leaf, then remove k from x.

Click mouse to advance to next frame.

[Case 2:] If k is in x and x is an internal node.

One of three subcases:

Case 2a

If child y ____________ k in x has $\geq$ t keys:

- Find predecessor k' of k in subtree rooted at y
- Recursively delete k' (first two steps can be performed in one pass down the tree)
- Replace k by k' in x

Click mouse to advance to next frame.

Case 2b

If child z ____________ k in x has $\geq$ t keys:

- Find successor k' of k in subtree y
- Recursively delete k'
- Replace k by k' in x

Click mouse to advance to next frame.

Case 2c

If both y and z have t-1 keys:

- Merge k and all of z into y
- Free z
- Recursively delete k from y

Note: x loses both k and pointer to z, y now contains 2t-1 keys.
Click mouse to advance to next frame.

Case 3

if k not in internal node x
then determine subtree child_i(x) containing k
    if child_i(x) has $\geq$ t keys
    then B-Tree-Delete(child_i(x), k)
    else execute Case 3a or 3b until can descend to node having $\geq$ t keys

---

Case 3a

If child_i(x) has t-1 keys but has a left or right sibling with $\geq$ t keys, then borrow one from sibling
        move key from x to child_i(x)
    move key from sibling to x
    move child from sibling to child_i(x)
  Click mouse to advance to next frame.

Case 3b

If $\text{child}_i(\text{x})$ and its left and right siblings have t-1 keys
then merge $\text{child}_i(\text{x})$ with one sibling using median key from x.

Click mouse to advance to next frame.

Analysis

Delete

$$\text{Disk Accesses: O(h), where h} = \text{O}(\log_t n)$$
$$\text{Run Time: O(th)}$$

B-Tree Operations

$$\text{Disk Accesses: O(h)} = \text{O}(\log_t n) = \text{O(lg n)}$$
$$\text{Run Time: O(th)} = \text{O(t} \log_t n) = \text{O(lg n)}$$

Applications