Lecture 29-30

B-Trees

Lecture 29-30

Today:

- B-Trees

Data Structures on Disk: Databases

Properties of Disk:

- access time
- varies by position(0-30ms)
- size 1 TB <—> 1 PB
- grain size 2 KB <—> 16 KB
- persistent

Properties of RAM:

- access time
- random access
- size 4-32GB
- grain size 1–16 bytes
- volatile

Consequences:

- 4 KB grain size ⇒ store > 1 key per node
- running time largely depends on number of disk accesses
  - a case where coefficients matter

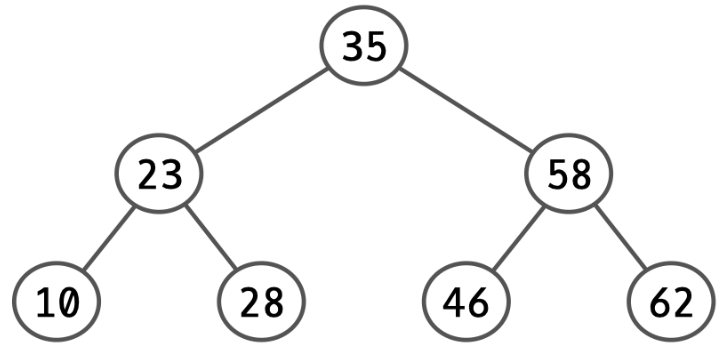
Balanced tree on disk?

Types of Balanced Trees (Review)

Perfect binary tree:

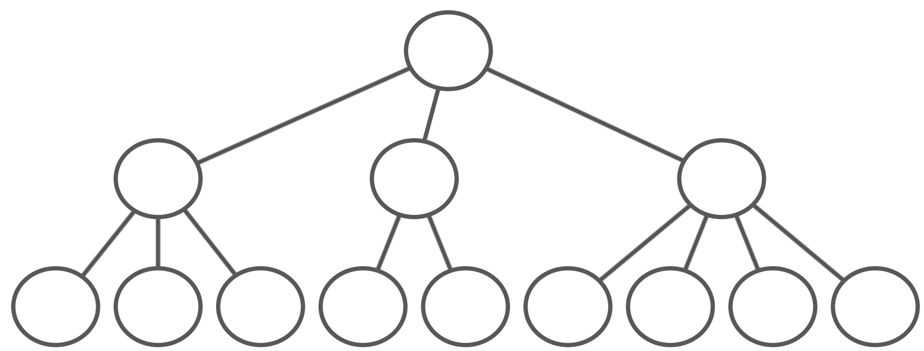
- a full binary tree with
- Q. How many nodes in a tree of height *h*?

  ⇒ N = 2^(h+1) - 1
- ideal balance, but works only for N = 0, 1, 3, 7, 15



All leaves at same level:

- augment perfect tree with Text
- branching factor between
-



B-Tree Properties

- all leaves at same depth
- given parameter
  - branching factor is
  - each node contains
  - for an internal node with *n* keys
  - children satisfy

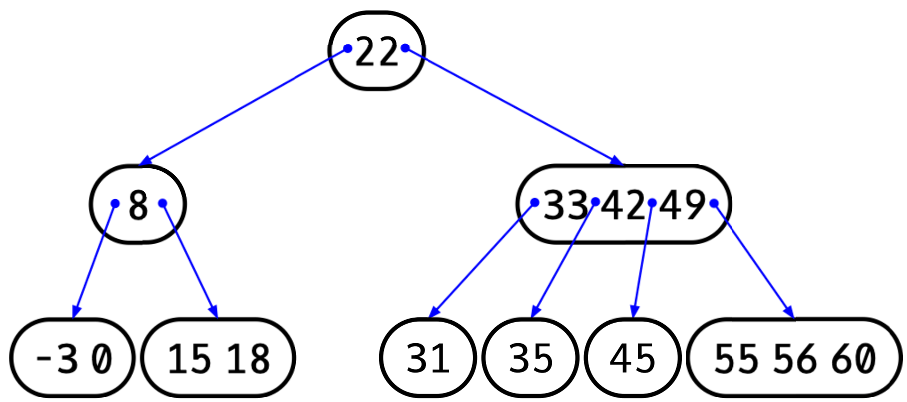
E.g., When $t = 2$, you get a

- branching factor is

Each node stored within one disk block

- higher $t \Rightarrow$

```
class BTreeNode {
   public:
       bool leaf;
       int numkeys;
       int keys[
       BTreeNode * c[
};
```



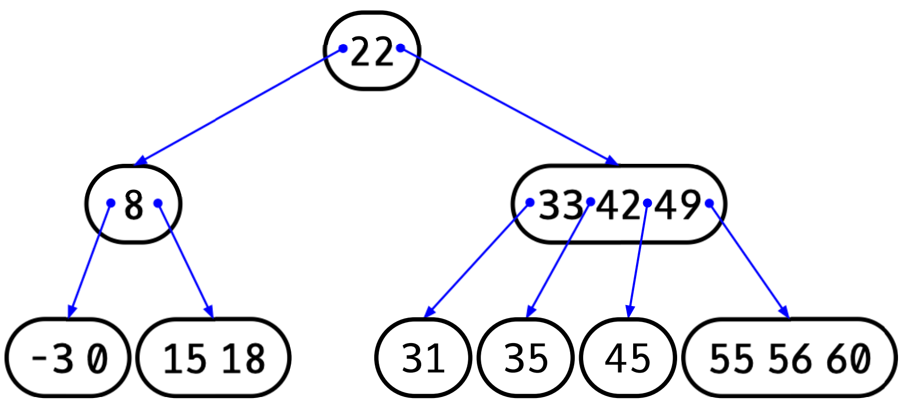
Search

Strategy: Traverse like for binary search tree except

- use   to pick subrange

E.g.,

- `.search(55)`
- `.search(8)`
- `.search(-2)`



B-Tree Insert

Strategy: Perform same steps as search to locate leaf.

- new key must be

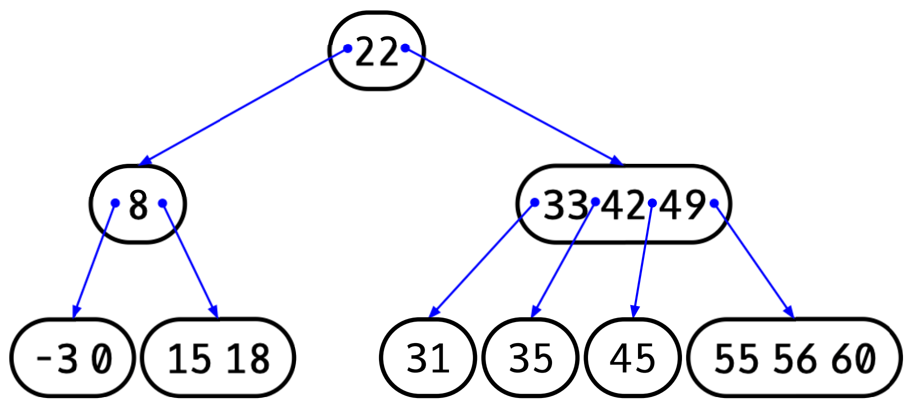
Two cases:

1. if leaf node has room, then
2. if leaf node is full, then
    -

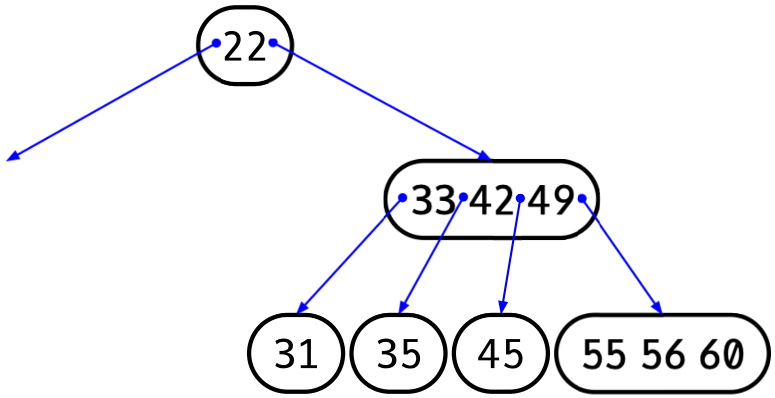
E.g.,

- `.insert(52)`

Split:



.insert(52)



B-Tree Growth and B-Tree Density

Q. How does tree grow in height?

- root is full -> split the root !

Q. How many keys are in a B-Tree of height *h*?

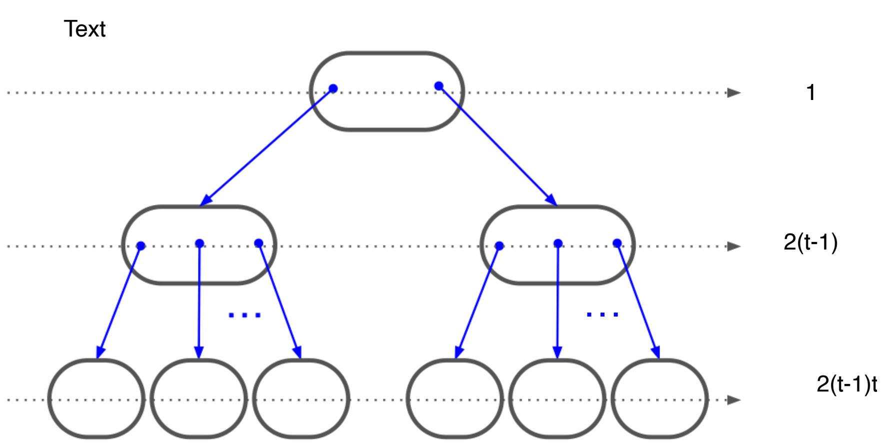
- since branching factor ≥ *t*, there are at least 2 * t^h -1 keys in a tree of height h
- *h* = o(logt(N))

In practice, B-Tree nodes are stored on disk

- size of each node 4 KB
- *t* 10^2 - 10^3

Running time largely ∝ # of disk access

- the higher the *t*, the fewer disk accesses



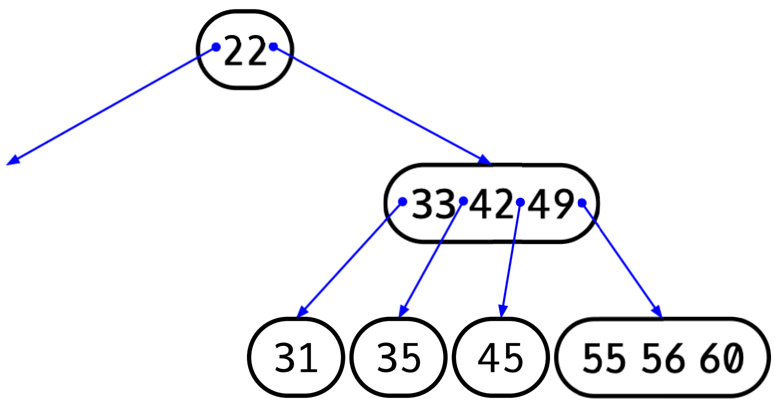
One-Pass Insert

All B-Tree operations should do at most one pass down the B-Tree

- `.insert()` did one pass down, but .split() may do a second pass up

New strategy: As you pass down the B-Tree, split any full node

- E.g., `.insert(52)`



Deletion

This time, worry about underfull nodes.

Two operations: <u>merge</u> and <u>transfer</u>

x ← root
while *x* not a leaf:
    if *key* in *x*:   key in internal node

delete k = seuccessor of key
replace key by k
return

find c, the child of x which might contain key
if c has t-1 keys:
transftер a key to c or merge c with sibling

if key in x, then delete it. (leaf node)

eventually reach a node with two nodes, if key is nuked it will not be an

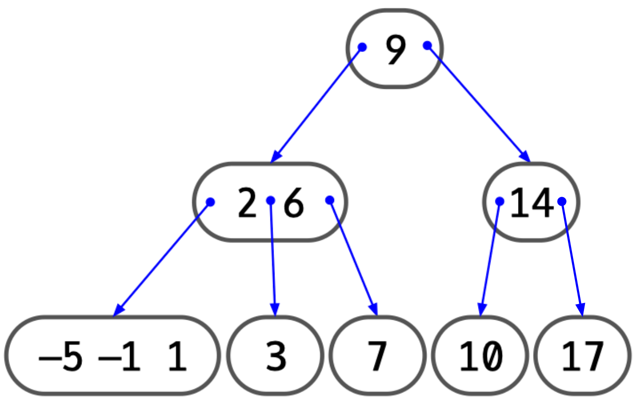
if *key* in *x*, then delete it

Merge:

Transfer:

Deletion Example

- `.delete(10)`
- OR `.delete(9)` which triggers `.delete(10)`



Text

Text
Text
Text

Extensions of B-Trees

B+-Trees

- keep all keys at the leaf level
- maximize branching factor on internal nodes
- can implement .range() and .successor() easily

B*-Trees

- keep all nodes at least 2/3 full
- fuller disk blocks ⇒ fuller disk blocks -> more effiecient disk usage