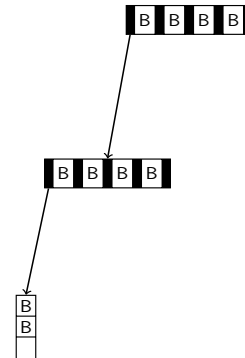# CSE 332

Data Structures and Parallelism

B-Trees
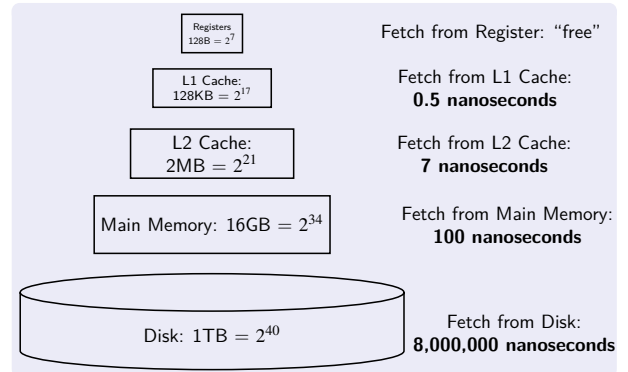

Outline

1 A New Model For Time Complexity

2 M-ary Search Trees

3 B-Trees

A New Model?   1

We've been assuming that **all memory accesses** are the same. In practice, this isn't true. The memory hierarchy looks something like this:

Registers $128B = 2^7$	Fetch from Register: "free"
L1 Cache: $128KB = 2^{17}$	Fetch from L1 Cache: **0.5 nanoseconds**
L2 Cache: $2MB = 2^{21}$	Fetch from L2 Cache: **7 nanoseconds**
Main Memory: $16GB = 2^{34}$	Fetch from Main Memory: **100 nanoseconds**
Disk: $1TB = 2^{40}$	Fetch from Disk: **8,000,000 nanoseconds**

The take-away is that **disk accesses** are very expensive.

A New Model?   2

Why do we care how the machine works?
Big-Oh is just an abstraction that says "all memory fetches are equal". . . but in practice, some memory fetches are more equal than others. (**The disk is prohibitively slow.**)

AVL Trees: Big-Oh vs. Practice
We've seen that AVL Trees are $\mathcal{O}(\lg n)$ which is great, but what if we account for disk accesses?
Consider an AVL Tree of height **40** where each node is b bytes.

- How many nodes in the tree? $\lg n = 40 \rightarrow n = 2^{40}$. So, we need about
$$b \text{ terabytes}$$
for the tree. This means **an overwhelming majority is on the disk**.
- How many disk accesses does a `find` take? It could take none (**3 nanoseconds**) or it could take 40 (**0.3 seconds**). That's a difference of:
$$100,000,000$$

If the data structure is mostly on disk, yes, we still need a data structure that is $\mathcal{O}(\lg n)$, but **it's not enough anymore**!

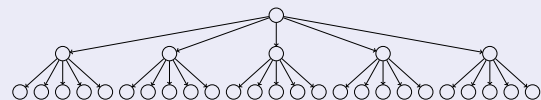## Okay. . . :(3

Problem
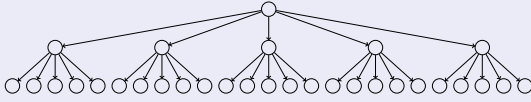A dictionary with so much data most of it is on disk

Goal
A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size

The Idea
Increase the branching factor of our tree
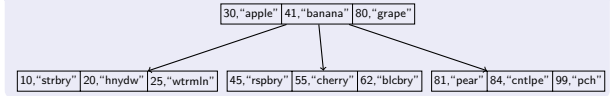
M-ary Search Tree

Like a binary tree, but with M branches instead of two.

M-ary Search Tree Properties

- Height (if balanced)? $\mathcal{O}(\log_M(n))$
- Ordering Property?
  - Binary Tree: smaller on the left, larger on the right
  - M-ary Tree: split the range into M equal sized groups
- Runtime of `find` (if balanced)? $\mathcal{O}(h \lg M) = \mathcal{O}(\log_M(n) \lg M)$
  - h possible nodes to visit: $\log_M(n)$
  - **Binary Search** on each node: $\lg M$

M-ary Search Tree Example?

30,"apple"	41,"banana"	80,"grape"

| 10,"strbry"	20,"hnydw"	25,"wtrmln"	45,"rspbry"	55,"cherry"	62,"blcbry"	81,"pear"	84,"cntlpe"	99,"pch"

Some Questions

- What should the order property be?
- How would re-balancing work? **We DON'T want to do more disk accesses!**

Some Thoughts

- We will have to load the **value**s (e.g., fruits) for all the internal nodes. This is very wasteful!
- Usually we are just "passing through" a node on the way to the value we are actually looking for.

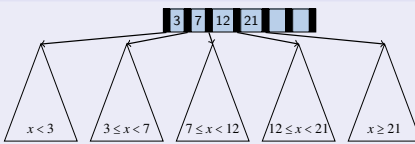### Two Types of Nodes

Internal Nodes
("sign posts")

K	K	K	K
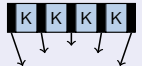
An internal node has $M-1$ **sorted** keys and M pointers to children

Leaf Nodes
("real data")

K,V
K,V
K,V

A leaf node has L **sorted** key/value pairs

B-Tree Order Property

3	7	12	21	

$x < 3$ $3 \le x < 7$ $7 \le x < 12$ $12 \le x < 21$ $x \ge 21$

Subtree between a and b contains all data x where $a \le x < b$

First, choose $M > 2$ and any L. (Here $M = 4, L = 5$.)

Very Few Nodes

If $n \le L$, the ROOT is a LEAF:

12				

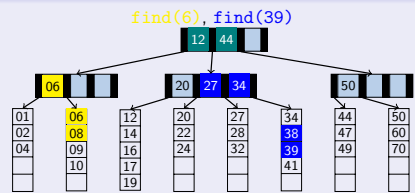Otherwise, the root must have between 2 and M children

B-Tree Example
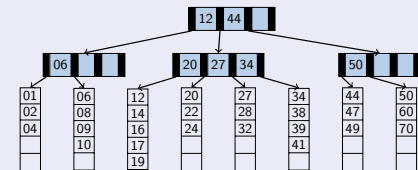
Internal Nodes must have between $\left\lceil \frac{M}{2} \right\rceil$ and M children (i.e., half full).

Leaf Nodes must have between $\left\lceil \frac{L}{2} \right\rceil$ and L children (i.e., half full).
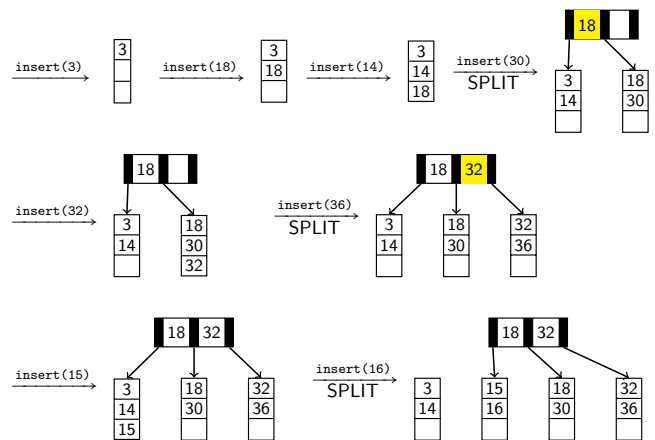
Find

`find(6)`, `find(39)`

Balanced Enough!

Let $M > 2$. Since all nodes are at least half full (ignoring the root), we have:
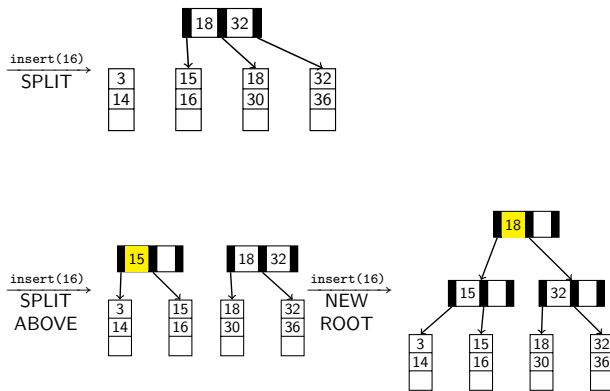
$$2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \text{ leaves, and each leaf has at least } \left\lceil \frac{L}{2} \right\rceil \text{ data items}$$

So, $n \ge 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \times \left\lceil \frac{L}{2} \right\rceil$. So, the height h is logarithmic in the number of data items n.
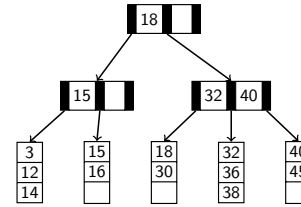
insert(16)
SPLIT

| 3 | 15 | 18 | 32 |
| 14 | 16 | 30 | 36 |

18 32

insert(16)
SPLIT
ABOVE

insert(16)
NEW
ROOT

15

18 32

| 3 | 15 | 18 | 32 |
| 14 | 16 | 30 | 36 |

18

15

32

| 3 | 15 | 18 | 32 |
| 14 | 16 | 30 | 36 |

insert(12), insert(40), insert(45), insert(38)

18

15

32 40

3	15	18	32	40
12	16	30	36	45
14			38	

Always fill the "signpost" with the <u>smallest value</u> to my right!

- Insert the data in the correct leaf **in sorted order**.

- If the leaf has $L+1$ items, overflow:
  - Split the leaf into two new nodes:
    - Original leaf with $\left\lceil \frac{L+1}{2} \right\rceil$ smaller items
    - New leaf with $\left\lceil \frac{L}{2} \right\rceil$ larger items
  - Attach the new child to the parent
  - Add the new key to the parent in sorted order

- Recursively continue overflowing if necessary. Noting that on the internal nodes we split using M instead of L.

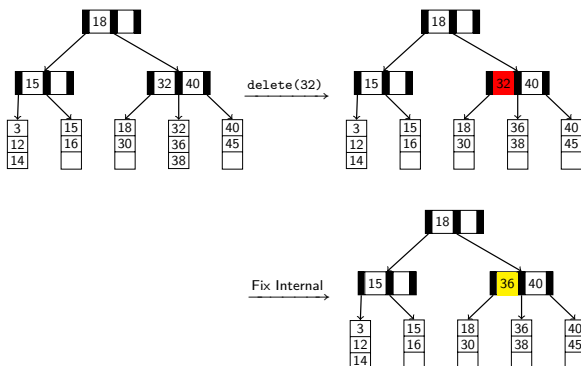- In the case where the **root** overflows, make a new root.

How Efficient is Insert?
- Find the correct leaf: $\mathcal{O}(\lg(M)\log_M(n))$
- Insert in the leaf: $\mathcal{O}(L)$
- Split leaf: $\mathcal{O}(L)$
- Split parents all the way up to the root: $\mathcal{O}(M\log_M(n))$
In total, this gives us $\mathcal{O}(L+M\log_M(n))$.

But It's Actually Pretty Good!
- Splits are very uncommon (think amortized analysis)
- Splitting the root almost never happens
- We're significantly more concerned about disk accesses than anything else: $\mathcal{O}(\log_M(n))$
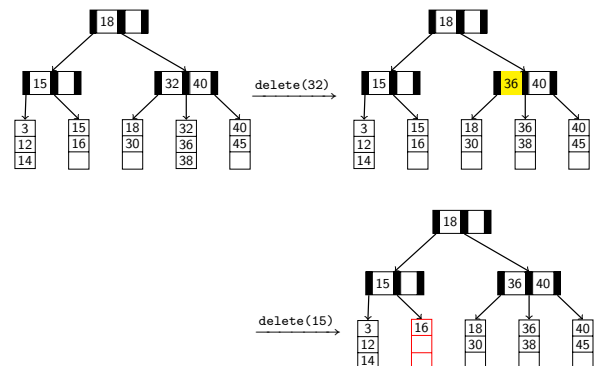
18

15

32 40

3	15	18	32	40
12	16	30	36	45
14			38	

delete(32)

18

15

32 40

3	15	18	36	40
12	16	30	38	45
14				

Fix Internal

18

15

36 40

3	15	18	36	40
12	16	30	38	45
14				

18

15

32 40

3	15	18	32	40
12	16	30	36	45
14			38	

delete(32)

18

15

36 40

3	15	18	36	40
12	16	30	38	45
14				

18

15

36 40

3	16	18	36	40
12		30	38	45
14				

delete(15)

This breaks our invariant. Leaves must have more than one node!
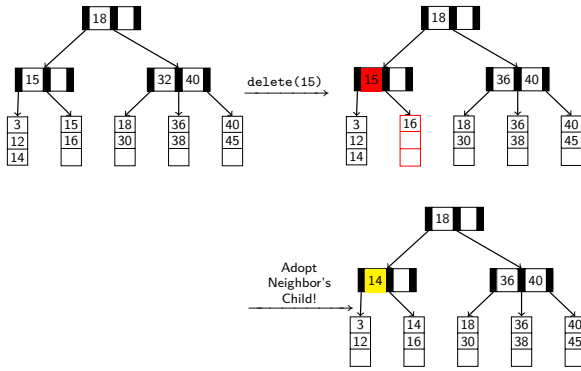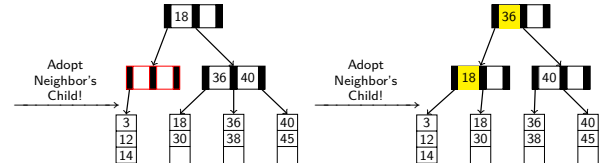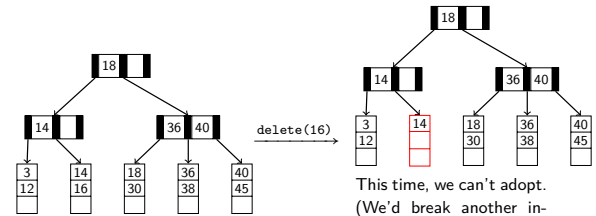
delete(15)

Adopt Neighbor's Child!

delete(16)

This time, we can't adopt. (We'd break another invariant.) The solution is to adopt **recursively**.

Adopt Neighbor's Child!

Adopt Neighbor's Child!

delete(14)

delete(18)

delete(18)
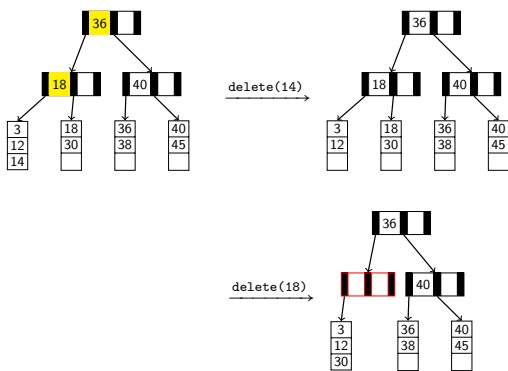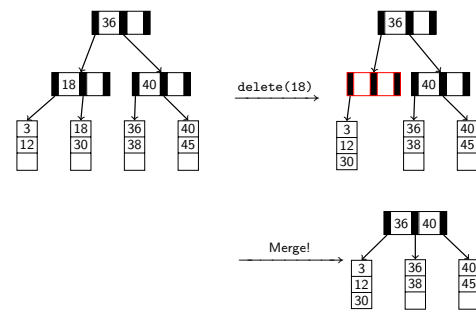
Merge!

- Remove the data from correct leaf.

- If the leaf has $\left\lceil \frac{L}{2} \right\rceil - 1$ items, underflow:

  - If a neighbor has more than $\left\lceil \frac{L}{2} \right\rceil$, adopt one!

  - Otherwise, **merge** with a neighbor (parent will now have one fewer node)

- Recursively continue underflowing if necessary. Noting that on the internal nodes we split using M instead of L.

- If we merge all the way up to the root and the root went from $2 \to 1$ children, then delete the root and make the child the root.

How Efficient is Delete?
- Find the correct leaf: $\mathcal{O}(\lg(M)\log_M(n))$
- Remove from the leaf: $\mathcal{O}(L)$
- Adopt/Merge with neighbor: $\mathcal{O}(L)$
- Merge parents all the way up to the root: $\mathcal{O}(M\log_M(n))$

In total, this gives us $\mathcal{O}(L + M\log_M(n))$.

But It's Actually Pretty Good!
- Merges are very uncommon (think amortized analysis)
- We're significantly more concerned about disk accesses than anything else: $\mathcal{O}(\log_M(n))$

Disk Friendlyness

What makes B-Trees so disk friendly?
- Many keys stored in one internal node: all brought into memory in one disk access
- Makes the binary search over $M-1$ keys totally worth it (insignificant compared to disk access times)
- Internal nodes contain only keys (it's a waste to load all the values)

We take advantage of the choice of M and L to ensure good behavior!

Choosing M and L

We want each of M and L to fit as best as possible in the **page size**.

Say we know the following:
- 1 page on disk is p bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

Then, we should choose the following:
- $p \geq M \times (\text{size of a pointer}) + (M-1) \times (\text{size of a key}) = Mt + (M-1)k.$
  So, $M = \left\lfloor \dfrac{p+k}{t+k} \right\rfloor.$
- $p \geq L \times v.$ So, $L = \left\lfloor \dfrac{p}{v} \right\rfloor.$

Wrap-Up

Balanced trees make good dictionaries because they guarantee logarithmic-time `find`, `insert`, and `delete`
- Essential and beautiful computer science

- But only if you can maintain balance within the time bound

- **AVL Trees** maintain balance by tracking height and allowing all children to differ in height by at most 1

- **B-Trees** maintain balance by keeping nodes at least half full and all leaves at same height

- Other great balanced trees (see text; worth knowing they exist)
    - Red-black trees: all leaves have depth within a factor of 2

    - Splay trees: self-adjusting; amortized guarantee; no extra space for height information