2-3-4 tree

- Nodes store 1, 2, or 3 keys and have 2, 3, or 4 children, respectively.
- All leaves have the same depth.
2-3-4 tree

• 2-node
  • 1 key, 2 kids
  • Same a binary tree

• 3-node
  • 2 keys and 3 kids

• 4-node
  • 3 keys and 4 kids
Insert to 2-3-4 tree

• Insert the **new key** at the **lowest internal node** reached in the search
  - 2-node becomes 3-node

• 3-node becomes 4-node

• What about a 4-node?
Insert to 2-3-4 tree

- In our way down the tree, whenever we reach a **4-node**, we break it up into **two 2-nodes**, and move the middle element up into the parent node.
2-3-4 tree and Red-black tree

- 2-3-4 tree and red-black tree are closely related
Matrix Sum

Given an M by N matrix X, sum its elements

- M rows, N columns

**Sum R**

given X, M, N
sum = 0
for i=0 to M-1{
   for j=0 to N-1 {
      sum += X[i][j]
   }
}

**Sum C**

given X, M, N
sum = 0
for j=0 to N-1{
   for i=0 to M-1 {
      sum += X[i][j]
   }
}

- What is the difference?
- What is the complexity of each?
- Should the execution speed be different?
The memory pyramid
Edited Excerpt of Jeff Dean’s talk on data centers.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Time</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>-</td>
<td>Your brain</td>
</tr>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td>Your desk</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>Neighbor’s Desk</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>This Room</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>Salt Lake City</td>
</tr>
</tbody>
</table>

Does Big-O analysis capture these effects?
Problem

• Secondary storage devices is 1 million times (at least) slower than the primary storage and even more so than L1 and L2 cache!
  • Reading each node that stores on the disk might take 1/10 second
  • Even for a red-black tree it will take $O(\log_2 n)$ reads from the disk
  • For 10 million record, the height of a red-black tree is about 25
    • That is 25 disk accesses

• Solution?
Problem

• Data (record) size might not be uniform (some can be much bigger)
  • Store index (key or address)

• Properties
  • Disk seek time is very slow, but reading data (once it is found on disk) is much faster
  • So, we should read a much as possible once a record is found
  • Usually a block of data will be read from the disk anyway, even if you just ask for a byte
  • This implies that, each node should store entire disk block (page size)
  • Each node should also store index of the record instead of the data itself
  • So, we are talking about something like 512-ary tree

• Finally, how do we balance, insert, delete data from such a tree?
B-tree

• “B” means several things: balanced, broad, Boeing, Bayer, etc.

• Motivated by
  • Limited resources in early years
  • Data that cannot fit into the main memory at once
  • Even today, we have gigabytes of memories, we still face similar problem
    • Better data acquisition techniques (many more sensors)
    • More data capturing devices (your cellphone)
    • More people are connected
    • Mobile devices also have limited resources
    • Resources can be shared by many people (cloud computing/storage)
B-tree

- Data items are all in the leaves
- Root is a leaf or has 2 to $M$ children
- Non-leaf node has $M - 1$ indices (keys)
  - Key $i$ is the smallest key in $(i + 1)$—th subtree
  - Must have $\left\lceil \frac{M}{2} \right\rceil$ to $M$ children
- All leaves are at the same level and have $\left\lceil \frac{L}{2} \right\rceil$ to $L$ data
- Note $M$ and $L$ are user inputs
B-tree

• Insert($x$)
  • Find the leaf that $x$ correspond to
  • Adjust the tree so it does not violate the B-tree properties
    • Split a node into two nodes
    • Promote the index (key) to a parent node

• Delete($x$)
  • Find the leaf that $x$ correspond to
  • Adjust the tree so it does not violate the B-tree properties
A B-tree of order 5
Insert 57
• After inserting 57
• Insert 55
• After inserting 55
• Insert 40
• After inserting 40
• Delete 99
• After deleting 99
• ADD($x, bt$)
  • find the right leaf in $bt$
  • if space in leaf
    • add $x$ to leaf else
  • if parent has room
    • new leaf
    • split data
    • add $x$ to leaf
  • else
    • recurse up
    • split internal
    • new leaves
    • split data
    • back down to add $x$

• REMOVE($x, bt$)
  • find leaf with $x$
  • remove $x$
  • if leaf < $1/2$ full
    • merge with neighbor leaf
    • steal leaves if needed
    • recurse up to adjust
B-tree Take-home

• Multi-way tree
• If order-M nodes are all \( \frac{1}{2} \) full \( O(\log_M N) \) height
• Hybrid of array/tree
• Good for data that doesn’t fit in memory
  • Large database
  • Filesystems (e.g., BFS, NTFS uses B+ trees for directory)
  • Sensitive to memory
• Simple idea, complex implementation (many cases)
• Many variants on the idea (B-tree, B+ tree, B* tree, etc.)