ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

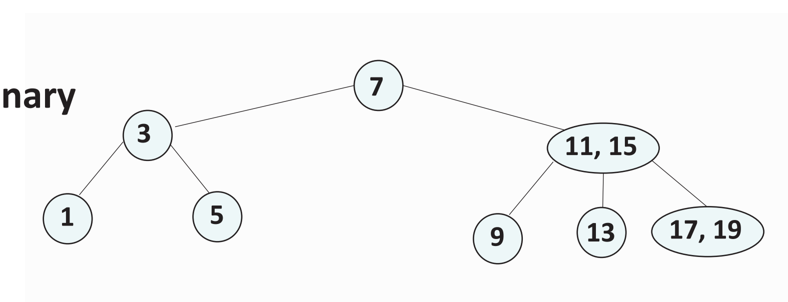
Module 15

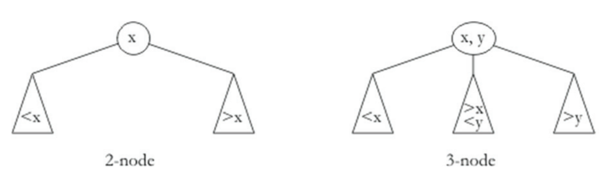
2-3 Trees, 2-3-4 Trees, B-Trees

Dayu Wang

- **2-3 Trees**

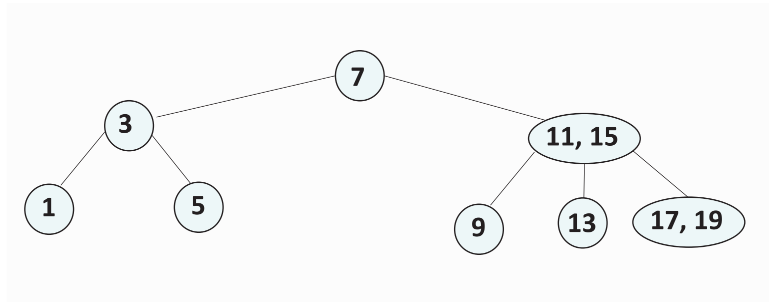
- These are **not binary** search trees.
- Because the nodes are not necessarily **binary**
  - They maintain all **leaves** at **same depth**
  - But number of children can vary
  - 2-3 tree: 2 or 3 children





- **Search in a 2-3 Tree**

```
if r is NULL, return NULL (not in tree)
if r is a 2-node
    if target equals data1, return data1
    else if target < data1, search left subtree
    else search right subtree
else // r is a 3-node
    if target < data1, search left subtree
    else if target = data1, return data1
    else if target < data2 , search middle subtree
    else if target = data2, return data 2
    else search right subtree
```



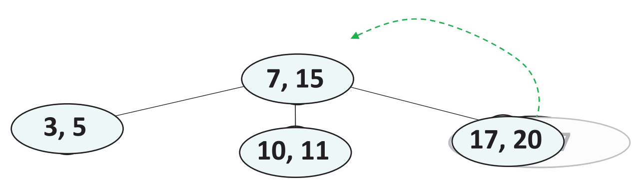
- **Inserting a key into a 2-3 tree**

Inserting into a 2-Node Leaf

- **(Inserting 15)** We insert it directly, creating a new 3-node.

Inserting into a 2-Item Leaf with a 2-Node Parent

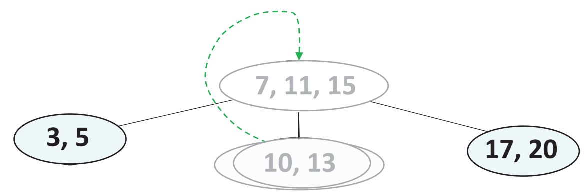
- **(Inserting 17)** The number will **virtually** be inserted into the 3 node at the bottom right
- Let’s insert: 5, 10, and 20.



- **Inserting a key into a 2-3 tree**

Inserting into a 3-Node Leaf with a 3-Node Parent

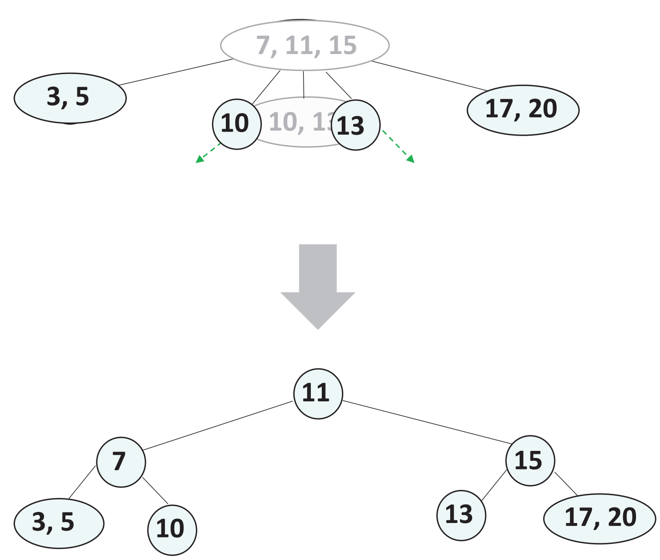
- **(Inserting 13)** If we insert an item and all leaf nodes are full, one of the leaf nodes will need to be split.
- This would result in two new 2-nodes with values 10 and 13, and 11 would propagate up to virtually inserted in the 3-node at the root.
- Because the root is full, it would split into two new root node.



- **Inserting a key into a 2-3 tree**

Inserting into a 3-Node Leaf with a 3-Node Parent

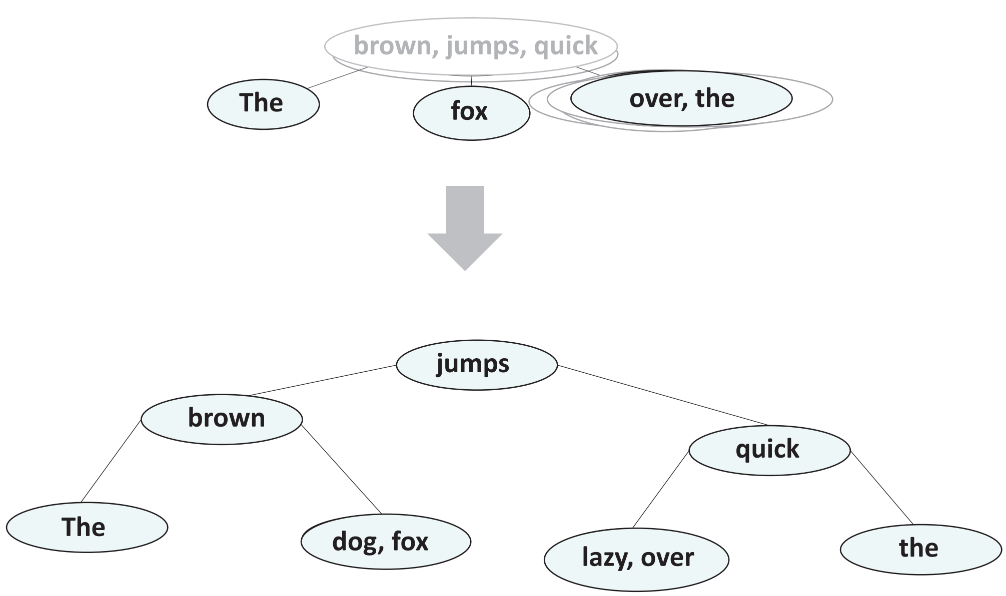
- If we insert an item and all leaf nodes are full, one of the leaf nodes will need to be split.
- This would result in two new 2-nodes with values 10 and 13, and 11 would propagate up to virtually inserted in the 3-node at the root.
- Because the root is full, it would split into two new root node.

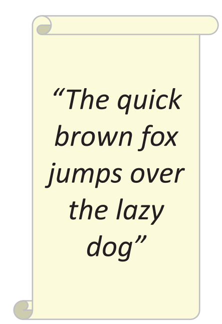


- **Inserting a key into a 2-3 tree**

```
if r is NULL, return new 2-node with item as data
if item matches r->data1 or r->data2, return false
if r is a leaf
    if r is a 2-node, expand to 3-node and return it
    split into two 2-nodes and pass them back up
else
    recursively insert into appropriate child tree
    if new parent passed back up
        if will be tree root, create and use new 2-node
        else recursively insert parent in r
return true
```

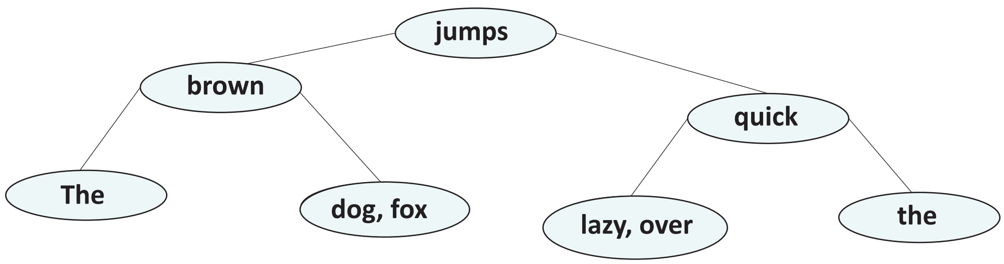

- **Building a 2-3 tree by inserting keys one-by-one**





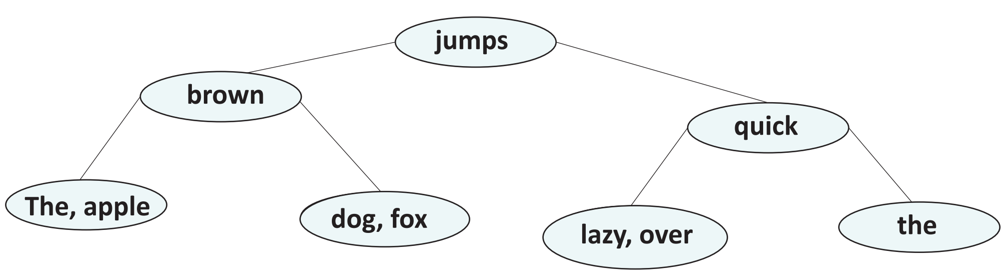
• **Exercise**

- Show how the 2-3 tree below as you insert “apple”, “cat”, and “hat” in that order.



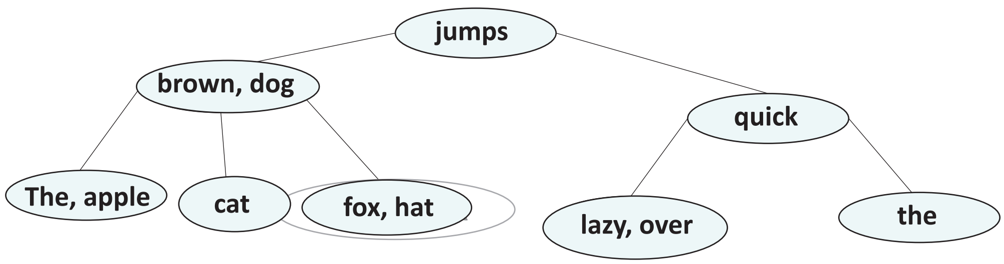
• **Exercise**

- Adding “apple”



• **Exercise**

- Adding “cat” and hat

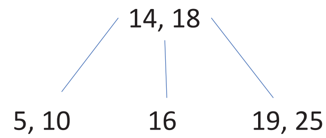


• **Exercise**

- Build a 2-3 tree from these numbers: 5, 18, 14, 16, 19, 25, 10

- **Exercise**

- Build a 2-3 tree from these numbers: 5, 18, 14, 16, 19, 25, 10

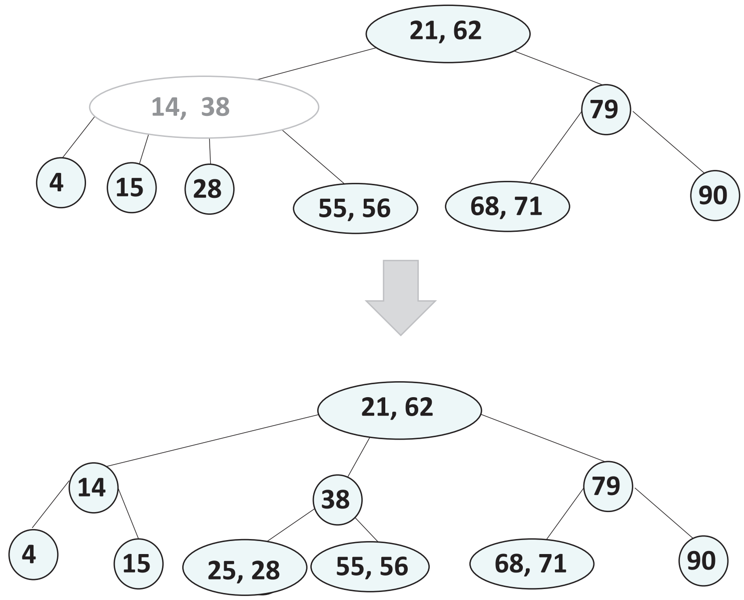


- **Performance of 2-3 trees**

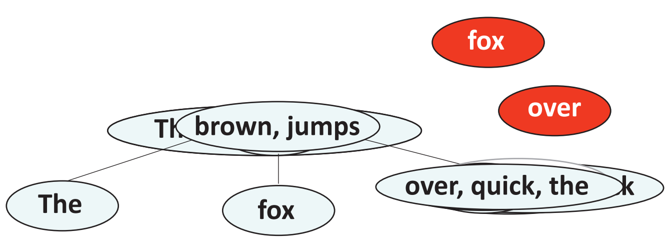
- If height is h, number of nodes in range 2^h-1 to 3^h-1
- height in terms of # nodes n in range $\log_2$ n to $\log_3$ n
- This is O(log n), since log base affects by constant factor
- So all operations are O(log n)

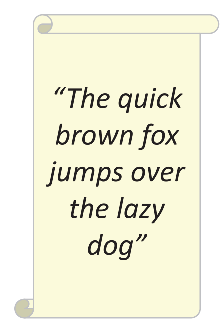
- **Inserting keys into 2-3-4 trees**

  - **Inserting 25:** You search for where the item belongs according to the binary search tree algorithm. If a 4 node is encountered (a node with 3 values), you split it.
  - **21** will propagate up to the parent.

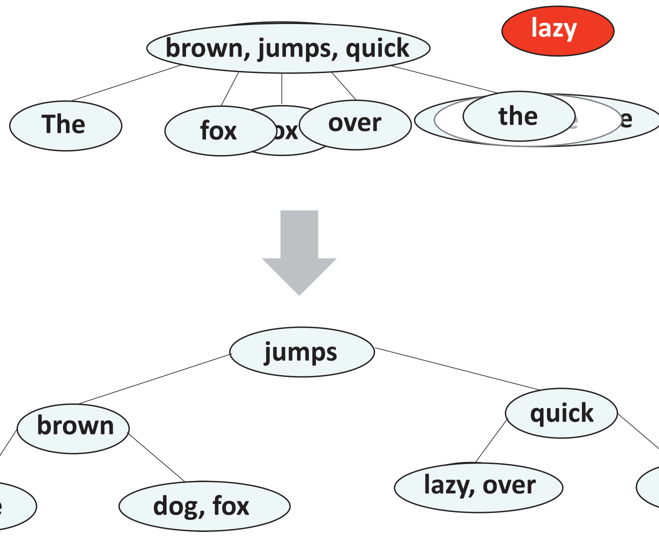


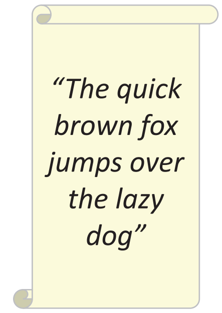
- **Building 2-3-4 trees by inserting keys one-by-one**





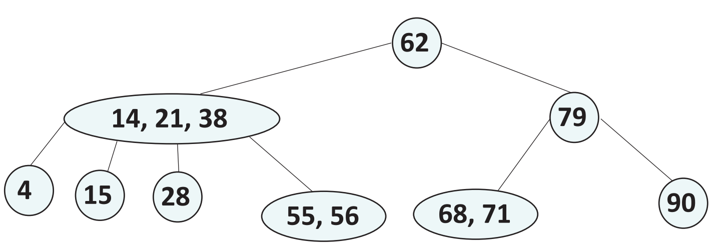
- **Building 2-3-4 trees by inserting keys one-by-one**



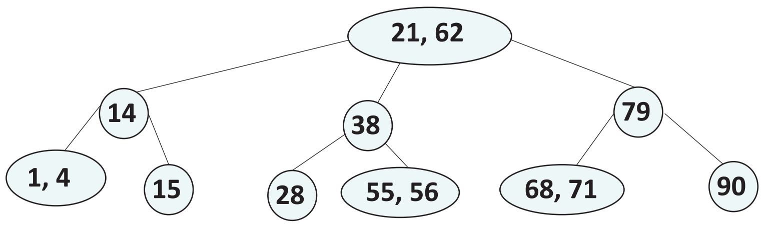


- **Exercise**

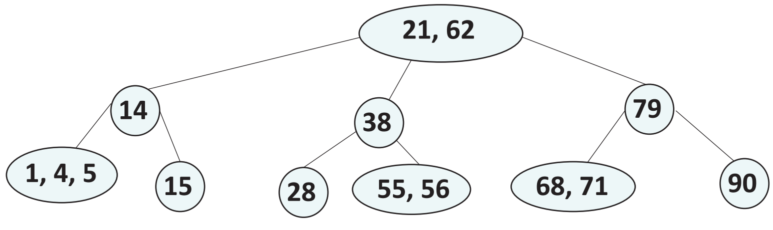
  - Show the following tree after inserting each of the following values one at a time: 1, 5, 9, and 13



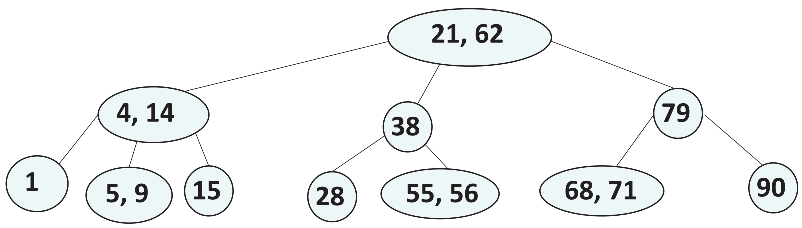
- **Exercise**



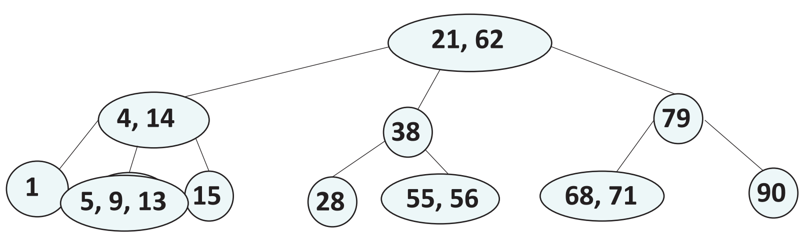
- **Exercise**



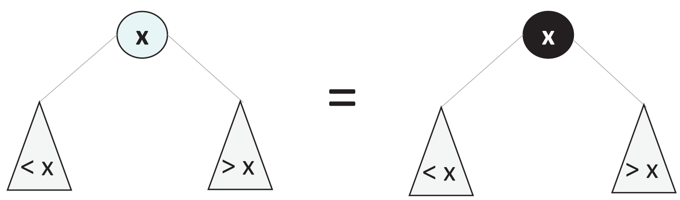
- **Exercise**

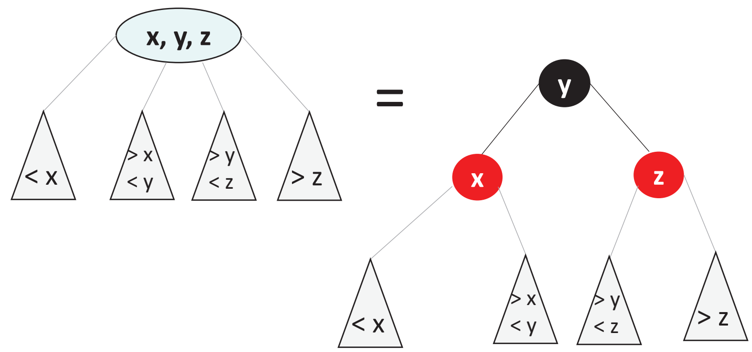


- **Exercise**



- **2-3-4 Trees *versus* Red-Black Trees**

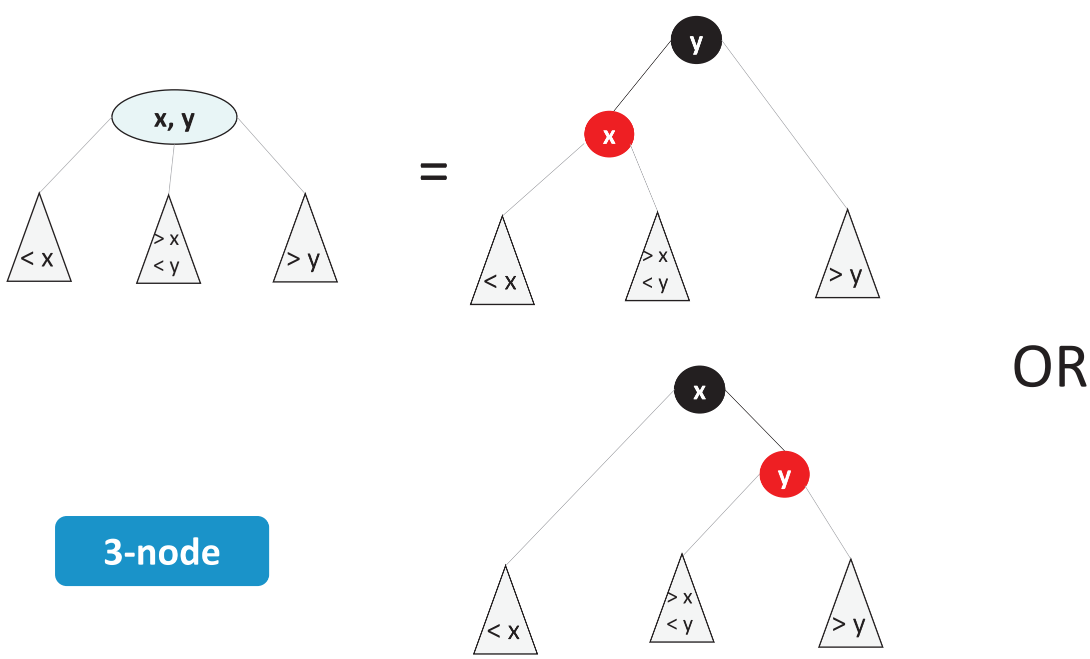




2-node

4-node

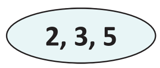
- **2-3-4 Trees *versus* Red-Black Trees**

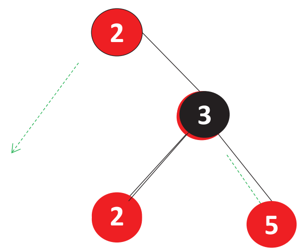


3-node

- **2-3-4 Trees *versus* Red-Black Trees**

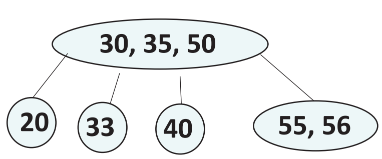
  - We want to insert **3** into a 2-3-4 tree.

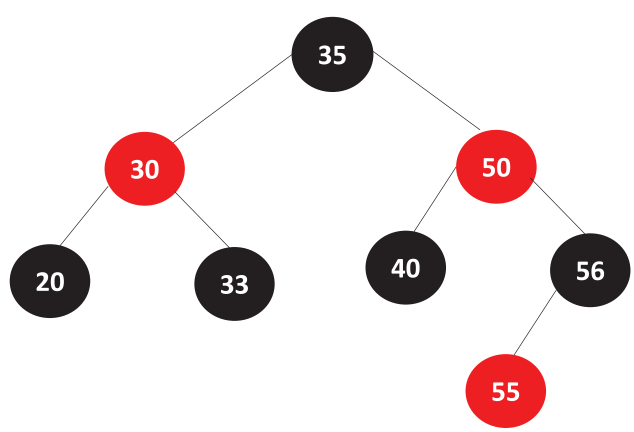




- **Exercise**

  - Convert the following 2-3-4 tree into a red-black tree





• **External Study Resource**

- 2-3 Trees (Insertion and Deletion)

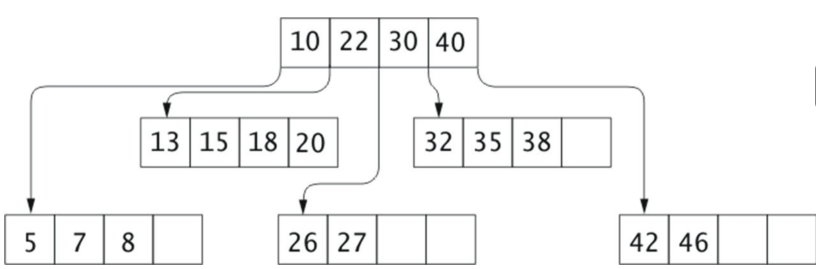
  http://slady.net/java/bt/view.php

- 2-3-4 Trees (Insertion and Deletion)

  http://www.cs.unm.edu/~rlpm/499/ttft.html

• **B-Trees**

- A **B-tree** extends idea behind 2-3 and 2-3-4 trees:
- Allows a maximum of data items in each node
- **Order** of a B-tree is maximum number of children for a node
- B-trees developed for indexes to databases on disk



number of children: **"5"**

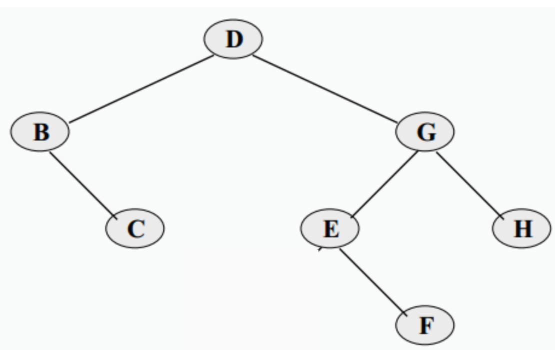
• **Motivation for B-Trees**

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

Reference: http://cecs.wright.edu/~tkprasad/teaching.html

• **Storing tree data on a disk**

Storing a tree on a disk



	Data	lChild	rChild
0	D	24	72
24	B	-1	48
48	C	-1	-1
72	G	96	168
96	E	-1	144
144	F	-1	-1
168	H	-1	-1

Reference: http://courses.cs.vt.edu/cs2604/fall05/wmcquain/Notes/C12.B-Trees.pdf

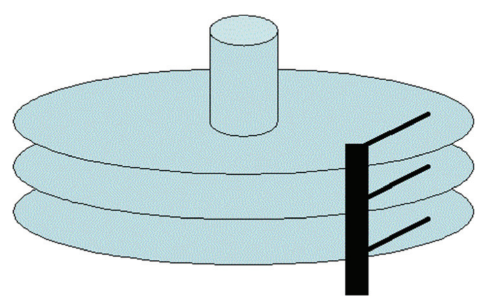
- **Motivation for B-Trees**

  - Assume that we use an AVL tree to store about 20 million records
  - We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2$ 20,000,000 is about 24. Assuming each node is stored on a sector. The height of the tree is about 24. To find an item, we might need to navigate across 24 nodes to reach the target. so this takes about 24*(1/60) which is about 0.4 seconds.
  - We know we can't improve on the log *n* lower bound on search for a binary tree
  - But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Reference: http://cecs.wright.edu/~tkprasad/teaching.html

- **Disk Storage**

  - Disk storage is broken into blocks, and the time to access a block is significant compared to the time required to access data in internal memory.
  - The nodes of a B-tree are sized to fit in a block, so each disk access to the index retrieves exactly one B-tree node.



http://www.dba-oracle.com/t_history_disk.htm

- **Analysis of B-Trees**

- The maximum number of items in a B-tree of order m and height h:

  root $m - 1$
  level 1 $m(m - 1)$
  level 2 $m^2(m - 1)$
  . . .
  level h $m^h(m - 1)$

- So, the total number of items is

  $(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$

  $[(m^{h+1} - 1)/ (m - 1)]\ (m - 1) = \boldsymbol{m^{h+1} - 1}$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reference: http://cecs.wright.edu/~tkprasad/teaching.html

- **Analysis of B-Trees**

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred:
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads.

Reference: http://cecs.wright.edu/~tkprasad/teaching.html