EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Splay Trees

Another binary tree approach: Splay trees are self-adjusting binary trees.

- “Lazy” data structure: Don’t insist on balance all the time, but recover balance when needed
- If elements of a BST are not to be accessed uniformly, then “balance” may not be

Begin with an arbitrary BST.

After looking up an element, repeatedly rotate that element with its parent until it becomes the root.

Recently-accessed elements will be up near the root of the tree, lowering access time.

Unused elements stay low in the tree.

Splay Tree Operations

- Insert: Standard BST insert, then splay
- Find: find as in BST, then splay to root using splay operation
- Delete: find as in BST, then splay to root using splay operation.
  - Subsequently, remove root, divide into 2 subtrees, do another splay, merge into single tree

Key: Splay operation

Double rotations to avoid problems above!

Splay Operation

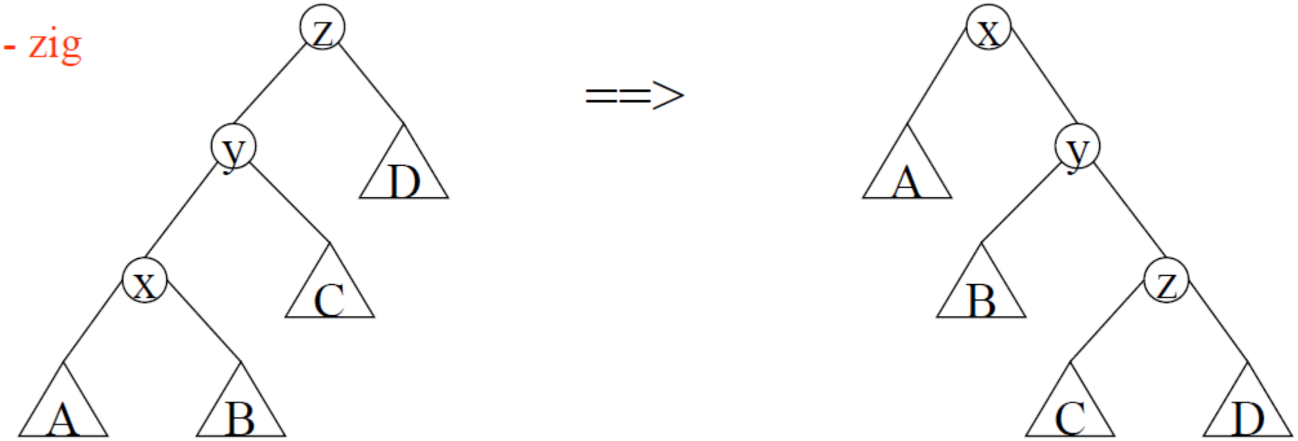
Splay node X:

- If parent is root, do rotation towards parent and end
- If parent is not root, and grandparent is aligned with parent (zig-zig):
  - Rotate parent towards grandparent
  - Then rotate X towards parent
- If parent is not aligned with grandparent (zig-zag):
  - Rotate X towards parent
  - Then rotate X towards grandparent

Splay Operation - 1

- Splay(x): Parent is not root, and grandparent is aligned with parent (zig-zig):
  - Rotate parent towards grandparent
  - Then rotate X towards parent

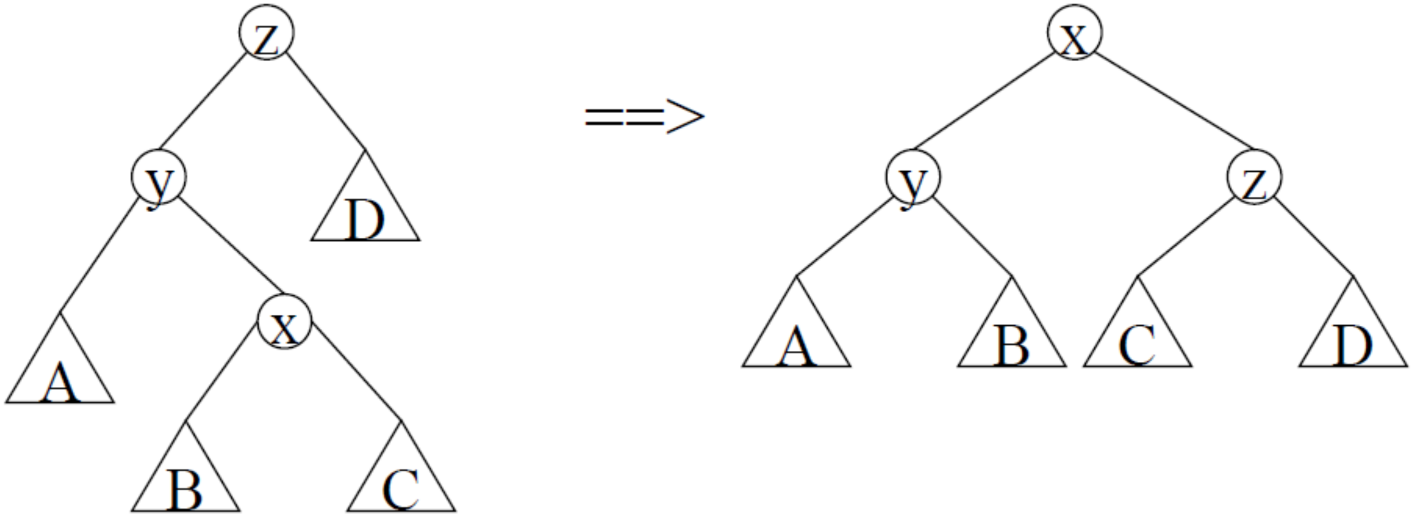
(1) zig - zig



Splay Operation - 2

- Splay(x): Parent is not root, and grandparent is not aligned with parent (zig-zag):
  - Rotate X towards parent
  - Then rotate X towards grandparent

(2) zig - zag



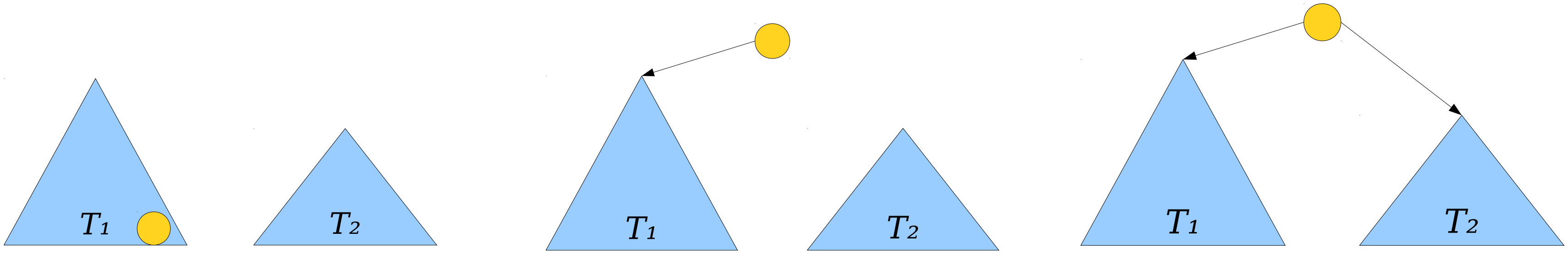
Splay Tree Operations

Operations:

- **Access**(x,T): Search for x as in a BST. If x is in T, return pointer to x and splay x; if not in T, splay the last non-null node reached looking for x.
- **Join**(T_1, T_2): two splay trees where all of $T_1 \leq$ all of T_2:
  - Find largest element in T_1 (right most non-null node!) and splay it to top of T_1. Since it is largest, new root has no right child. Attach root of T_2 to right child of new root of T_1.
- **Split**(x,T): first, **Access** (x,T). If the new root greater than x, break the left child link and form two trees. Otherwise, break the right child and form two trees.
- **Insert**(x,T): Insert x as in Binary Search Tree T. Splay x to root.
- **Delete**(x,T): **Access**(x,T). Let T_1, T_2 denote the right and left subtrees of the new root x. Delete x, and **Join**(T_1, T_2).

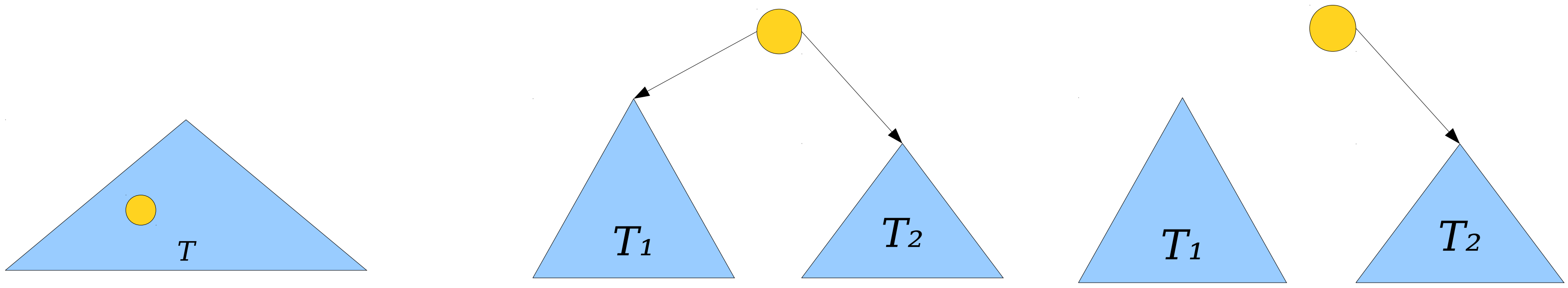
Splay Tree Join

- **Join**(T_1, T_2): two splay trees where all of $T_1 \leq$ all of T_2:
  - Find largest element in $T_1$and splay it to top of T_1. Since it is largest, new root has no right child. Attach root of T_2 to right child of new root of T_1



Splay Tree Split

- **Split**(x,T): first, **Access** (x,T). If the new root greater than x, break the left child link and form two trees. Otherwise, break the right child and form two trees.



Splay Tree Delete

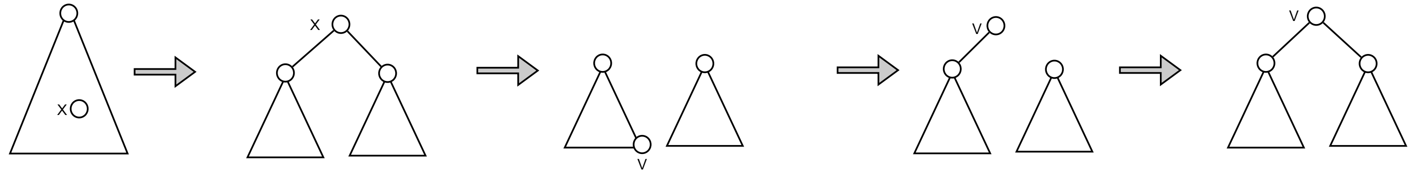
- **Delete**(x,T): **Access**(x,T). Let T_1, T_2 denote the right and left subtrees of the new root x. Delete x, and **Join**(T_1, T_2).

Find x and splay it

Delete x

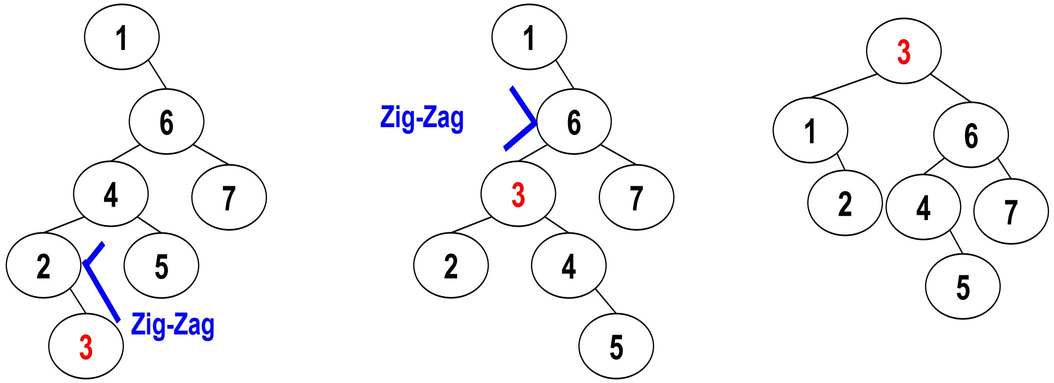
Find the predecessor v of x and splay it

Make v the parent of the root of the right subtree



Delete 3: Access

Delete and Join:



Complexity of Splay Trees

Amortized complexity is O(log(n))

- Will show (after we learn about amortized complexity

Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.

Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.

Splay trees are widely used basic data structure, because they're the fastest type of balanced search tree for many applications

- Splay trees are used in Windows NT (in the virtual memory, networking, and file system code),
- the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers,
- Unix malloc, Linux loadable kernel modules, and in much other software

Amortized Analysis

Amortized analysis is a different way of bounding the runtime of a sequence of operations.

- Each operation op_i really takes time t(op_i).
- Idea: Assign to each operation op_i a new cost a(op_i), called the amortized cost, such that

$$\sum_{i=1}^{m} t(op_i) \leq \sum_{i=1}^{m} a(op_i)$$

- If the values of a(op_i) are chosen wisely, the second sum can be much easier to evaluate than the first

Methods for Amortized Analysis

In the **aggregate method**, we directly evaluate $T = \sum_{i=1}^{m} t(op_i)$, and set a($op_i$) = T/m

- Assigns each operation the average of all the operation costs

In the **accounting method**, we place “credits” on the data structure for some operations redeemable for units of work in other operations

In the **potential method,** we assign a potential function to the data structure and factoring in changes to that potential for each operation to the overall runtime.

The Accounting Method

- In the accounting method, operations can place credits on the data structure or spend credits that have already been placed.
  - Placing a credit somewhere takes time O(1).
  - Credits may be removed from the data structure to pay for O(1) units of work.
  - Note: the credits don't actually show up in the data structure. It's just an accounting trick.
- The amortized cost of an operation is

$$\mathrm{a}(op_i) = \mathrm{t}(op_i) + \mathrm{O}(1) \cdot (added_i - removed_i\ )$$

- If we never spend credits we don’t have,

$$\sum_{i=1}^{m} a(op_i) = \sum_{i=1}^{m} t(op_i) + O(1) \cdot (added_i - removed_i)$$

The Accounting Method - 2

- If we never spend credits we don’t have,

$$\sum_{i=1}^{m} a(op_i) = \sum_{i=1}^{m} t(op_i) + O(1) \cdot (added_i - removed_i)$$

$$\sum_{i=1}^{m} a(op_i) = \sum_{i=1}^{m} t(op_i) + O(1) \cdot netCredits \geq \sum_{i=1}^{m} t(op_i)$$

(Upper bound)

The Potential Method

- In the potential method, we define a potential function Φ that maps a data structure to a non-negative real value.
  - Each operation on the data structure might change this potential.
- If we denote by Φ_k the potential of the data structure just before operation k, then we can define a(op_k) as $a(op_i) = t(op_i) + O(1) \cdot (\Phi_{i+1} - \Phi_i)$
- Intuitively:
  - Operations that increase the potential have amortized cost greater than their true cost.
  - Operations that decrease the potential have amortized cost less than their true cost.
- Assuming that $\Phi_m - \Phi_1 \geq 0$, the sum of the amortized costs upper-bounds the sum of the real costs.

$$\sum_{i=1}^{m} a(op_i) = \sum_{i=1}^{m} [t(op_i) + O(1) \cdot (\Phi_{i+1} - \Phi_1)] = \sum_{i=1}^{m} t(op_i) + O(1) \cdot (\Phi_{m+1} - \Phi_1)$$

Example: FIFO Queue with 2 Stacks

- 2 Stacks:
  - enqueue(x): push x onto Stack 1
  - dequeue(}: If Stack 2 is empty, pop entire contents of Stack 1 into Stack 2. Pop from Stack 2

Note: cost of dequeue() can be O(n) some times, following n consecutive enqueues!

Clearly, this is not a good bound on a sequence of operations.

Aggregate Method: each element is pushed at most twice and popped at most twice.

Total cost of operations on n elements is $\leq 4n \in O(n)$, so amortized cost per element is O(1).

Accounting Method: Start with 0, and two empty stacks. Pay 3 for each enqueue: one today for the push onto Stack 1, and credit 2 for future ops. Pay 1 for each pop from Stack 2. Pay 1 for each pop from Stack 1 from credit, and 1 for push onto Stack 2 from credit. Note that you always have a positive balance. You've paid the cost of transferring all the elements from Stack 1 to Stack 2 already! Amortized cost of operation when stack 2 is empty and stack 1 has n elements is 1 + 2n - 2n (from credit).

Example: FIFO Queue with 2 Stacks

- 2 Stacks:
  - enqueue(x): push x onto Stack 1
  - dequeue(}: If Stack 2 is empty, pop entire contents of Stack 1 into Stack 2. Pop from Stack 2

Potential Method:

$\Phi(D) =$ 2*height of stack 1

- enqueue(x): amortized cost O(1): one unit of work, plus increases potential by 2.
- dequeue(): if Stack 2 is not empty: O(1), which is one unit of work, no change in potential
- dequeue(): if Stack 2 is empty, and stack 1 has height k, does 2k + 1 work, but decreases potential by 2k, so amortized cost is 1

All operations are amortized cost O(1), and final potential is non-negative, the cost of m operations is O(m).

Amortized Analysis of Splay Trees

- Use Potential Method
  - Each node in tree T has weight 1

    $Size(x, T) \equiv s(x)$ = sum of weights of nodes in subtree rooted at x (includes x)

    $Rank(x, T) \equiv r(x) = \log_2(Size(x, T))$

    Potential $\Phi(T) = \sum_{x \in T} Rank(x, T)$
- Observations
  - $\Phi(T) \leq n \log_2(n)$
  - In the most unbalanced tree (a line), $\Phi(T) = \log_2(1) + \log_2(2) + \ldots \log_2(n) \in O(n \log(n))$
  - In a perfect binary tree with n nodes,

$$\Phi(T) = \frac{n+1}{2}\log_2(1) + \frac{n+1}{4}\log_2(3) + \frac{n+1}{8}\log_2(7) + \ldots + \log_2(n) \leq (n+1)\sum_{i=1}^{\infty}\frac{i}{2^i} \in O(n)$$

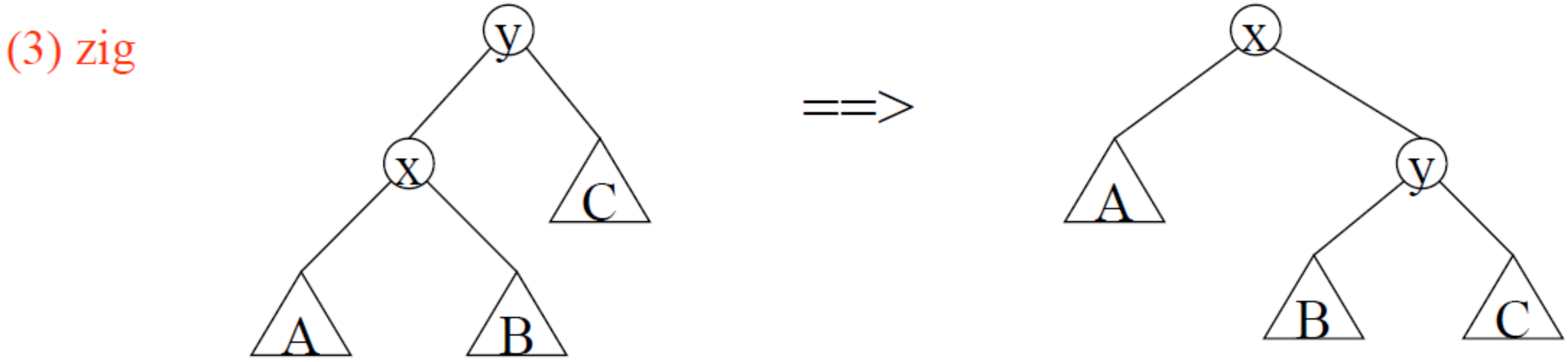
Amortized Analysis of Splay Trees - 2

- How rotations change potential

$s'(y) = s(y) – s(A) - 1$

$s'(x) = s(x) + s(C) + 1$

Cost = 1 (rotation) + $r'(x) + r'(y) – r(x) – r(y)$. Since y decreases in rank, this is less than $1 + r'(x) – r(x) \leq 1 + 3(r'(x) – r(x))$



Amortized Analysis of Splay Trees - 3

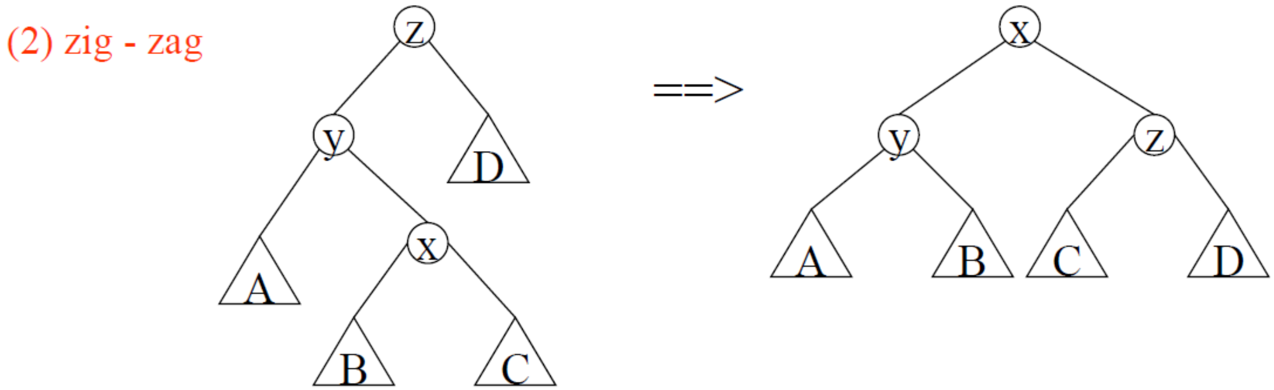
How rotations change potential

s’(x) = s(x) + 2 + s(A) + s(D); s’(y) = s(y) – 1 – s(C); s’(z) = s(z) – 2 – s(A) – s(B)

Cost = 2 + r’(x) + r’(y) + r’(z) - r(x) - r(y) - r(z) = 2 + r’(y) + r’(z) - r(x) - r(y)

Observation: $\log_2(s'(x)/2) = r'(x) - 1 > \log_2(\frac{s'(y)+s'(z)}{2}) \geq \frac{1}{2}(r'(y)+r'(z))$ (concavity of log())

Thus, 2r’(x) - r’(y) - r’(z) > 2, and Cost < 2r’(x) - r’(y) - r’(z) - r’(y) - r’(z) - r(x) - r(y) < 3(r’(x) - r(x))



Amortized Analysis of Splay Trees - 4

How rotations change potential

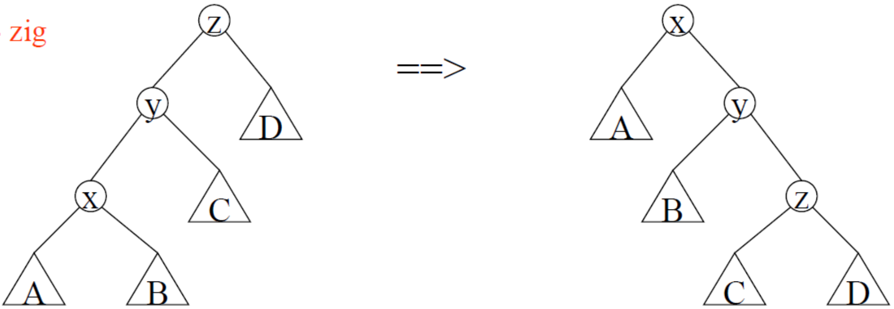
$s'(x) = s(x) + 2 + s(C) + s(D) = s'(z) + s(x) + 1 = s(z);$

$\text{Cost} = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) = 2 + r'(y) + r'(z) - r(x) - r(y)$

Observation: $\log_2(s'(x)/2) = r'(x) - 1 > \log_2(\frac{s'(y) + s(x)}{2}) \geq \frac{1}{2}(r'(y) + r(x))$ (concavity of log())

Thus, $2r'(x) - r'(y) - r(x) > 2$, and $\text{Cost} < 2r'(x) - r'(y) - r(x) + r'(y) + r'(z) - r(x) - r(y) < 3(r'(x) - r(x))$

(1) zig - zig



amortized time(zig-zig) =

Amortized Analysis of Splay Trees - 5

Amortized cost of splay operation:

Splay is a sequence of double rotations, potentially finishing with a single rotation

Amortized cost of each double rotation is < 3(r’(x) - r(x))

Amortized cost of single rotation < 3(r’(x) - r(x)) + 1

x moves up to root after splay —> amortized cost of splay is **3(r’(root) - r(x)) + 1**

r’(root) = log(n) —> amortized cost of splay is O(log(n))

Since every operation in tree is dominated by cost of 1 or 2 splays, this shows amortized cost of splay trees is O(log(n)) for all operations

Small Issue with Splay Tree Implementation

- Often, you have to splay back up the tree
  - Hard to do unless you keep track of the parents, grandparents on the way to finding node to splay
  - Two cycles: down the tree to find access node, up the tree to splay it to root

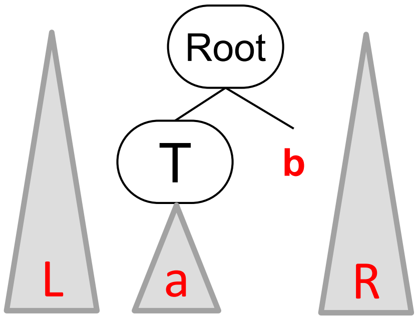
- Better idea: **Top-Down** Splay trees
  - Do splay as you descend towards node
  - Simpler, more efficient implementation
  - Original algorithm suggested by Sleator-Tarjan

Top-Down Splay Trees

- Often, you have to splay back up the tree
  - Hard to do unless you keep track of the parents, grandparents on the way to finding node to splay
  - Two cycles: down the tree to find access node, up the tree to splay it to root

- Better idea: **Top-Down** Splay trees
  - Do splay as you descend towards node
  - Simpler, more efficient implementation
  - Original algorithm suggested by Sleator-Tarjan

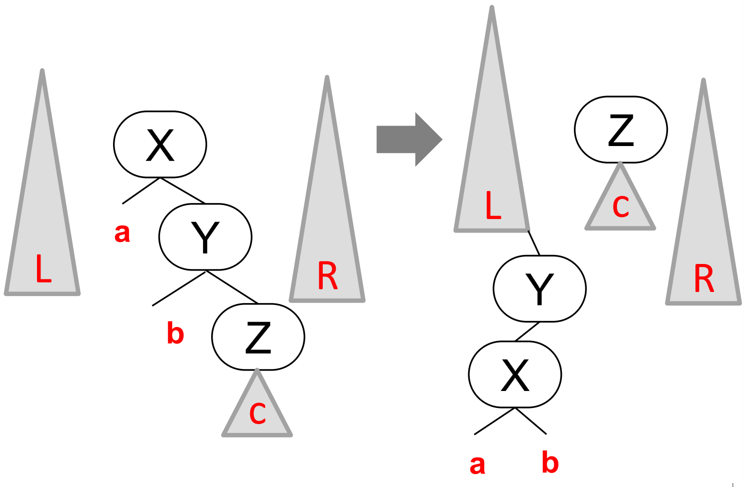
Top-Down Splay Trees: Access



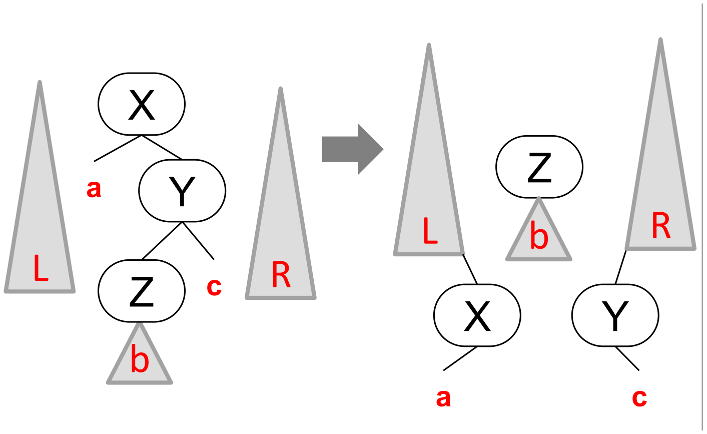
- **Access(T)** Start with empty left, right trees L, R
    - Splay tree is middle tree
    - L has null right pointer to attach items (from its largest element); R has null left pointer to attach items (from its smallest element)
    - All items in L are less than or equal to items in middle tree, all items in R are greater than items in middle tree
- Search for T in middle tree: if T is one level down, do single rotation (zig or zag); if 2 or more levels down, descend towards T with double rotation (zig-zig or zig-zag)
- Once T is found, and is root of middle tree, reattach L and R.

Top-Down Splay Trees: Access - 2

- **Top-Down splaying: zag-zag**
  - Y becomes largest element in L, has no left child
  - New search subtree rooted in Z

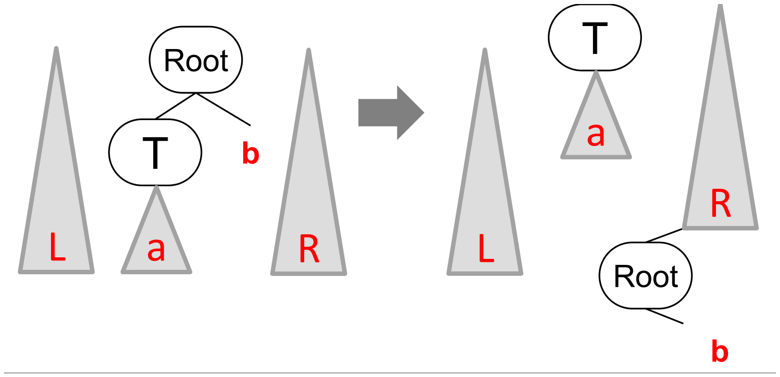


- **Zig-zag:**
  - X becomes largest element in L
  - Y becomes smallest element in R
  - New search subtree rooted in T



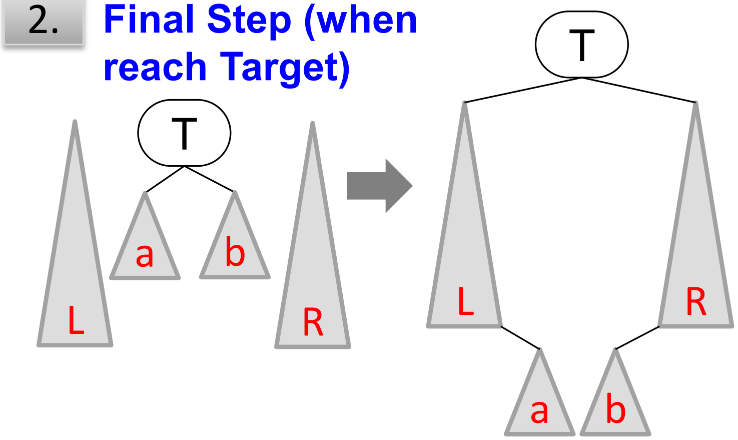
Top-Down Splay Trees: Access - 3

- **Top-Down splaying: zig**
  - Search element is one level below root
  - Root becomes smallest element of L in this case
  - Access element T is now root of center tree

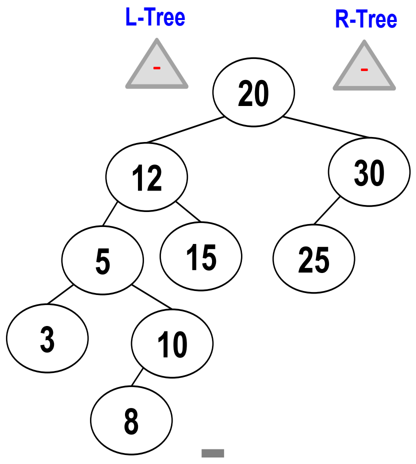


- **Assemble the splay tree:**
  - Right subtree of T attached to L (its elements are larger than L)
  - Left subtree of T attached to R (its elements are smaller than R
  - L, R become left, right subtrees of T

2. **Final Step (when reach Target)**



Example: Insert(11)

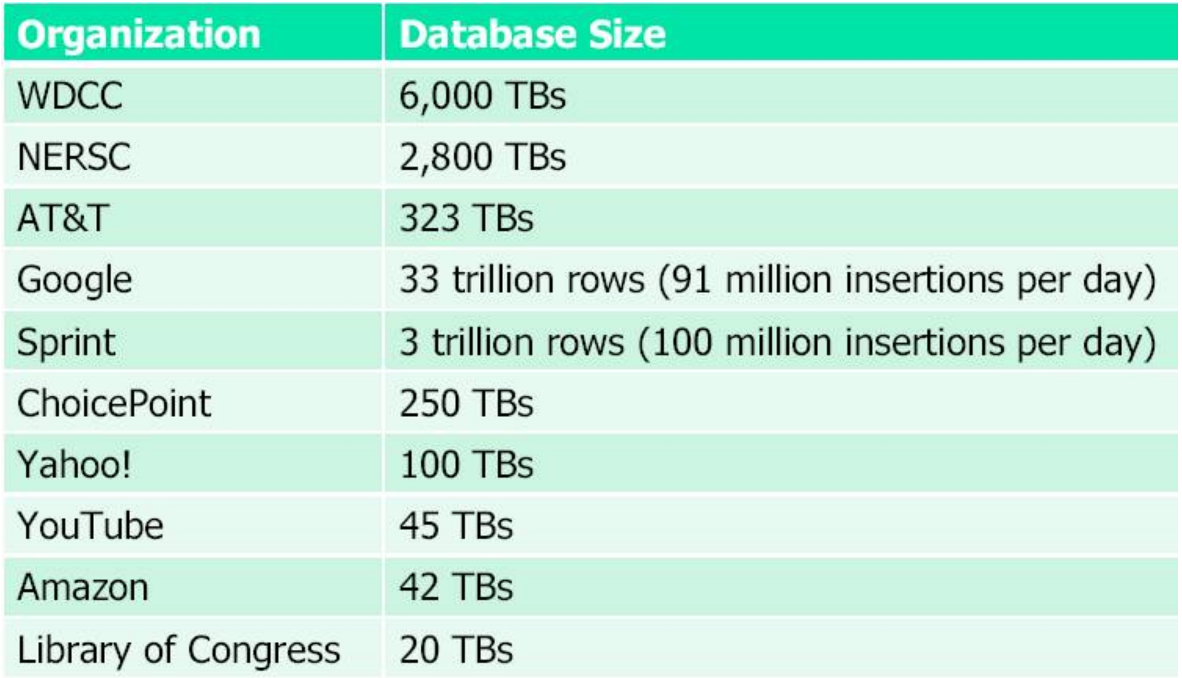


Splay Trees: Summary

- Lightweight, easily implemented balanced search tree
  - Balance is not strictly enforced, so avoid needless overhead
  - Amortized complexity is O(log(n))
  - Provides fast access to recently-accessed keys
  - Fastest balanced search tree for many applications

- Splay trees are widely used
  - Windows NT (in the virtual memory, networking, and file system code)
  - gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers,
  - Unix malloc, Linux loadable kernel modules, and in much other software

New Problem: Large Databases



Organization	Database Size
WDCC	6,000 TBs
NERSC	2,800 TBs
AT&T	323 TBs
Google	33 trillion rows (91 million insertions per day)
Sprint	3 trillion rows (100 million insertions per day)
ChoicePoint	250 TBs
Yahoo!	100 TBs
YouTube	45 TBs
Amazon	42 TBs
Library of Congress	20 TBs

Source: www.businessintelligencelowdown.com, 2007.

- How are these stored for efficient search? Oracle, SQL? Not in RAM…

Large Databases

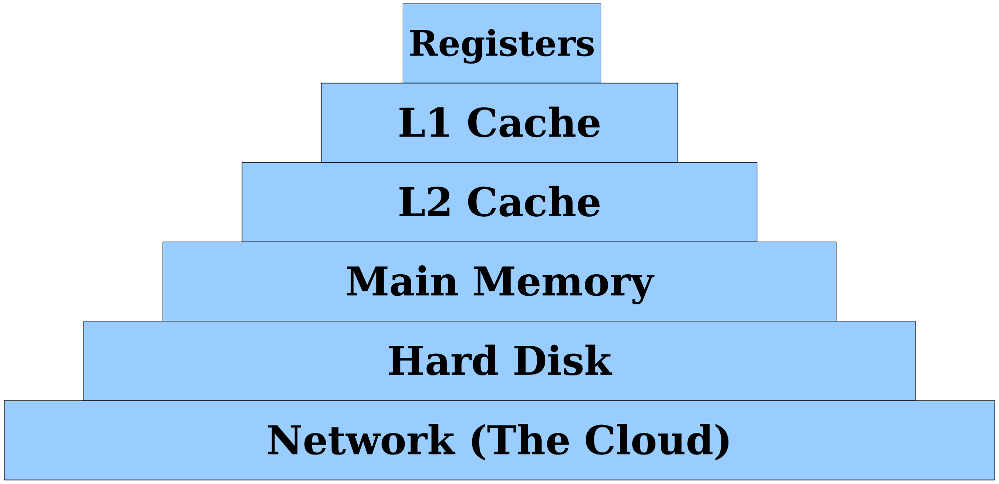
- Use a BST?
  - Google – 33 trillion items, indexed by IP
  - Access time
    - Height =log $(33x10^{12})$ = 44.9
    - Assume 120 disk accesses per second: Search takes 0.37 seconds

- What does Oracle, SQL do?
  - Use better search trees to reduce disk access

- How about other solutions (e.g. hash tables, like Python dictionaries?)
  - Many data base queries imply sorting (hard in hash tables)
  - Huge data bases need to manage disk I/O (not fit in memory!)

Want to Exploit Memory Hierarchy

- The lower you go in the hierarchy, the less you want to go back
  - Limit accesses to very small numbers



Registers	256B - 8KB	0.25 – 1ns
L1 Cache	16KB – 64KB	1ns – 5ns
L2 Cache	1MB - 4MB	5ns – 25ns
Main Memory	4GB – 256GB	25ns – 100ns
Hard Disk	1TB+	3 – 10ms
Network (The Cloud)	*Lots*	10 – 2000ms

Idea: Multiway Search Trees

- Idea: allow a node in a tree to have many children
- Less disk access = less tree height = more branching
- As branching increases, the depth decreases
- An M-ary tree allows M-way branching
- Each internal node has at most M children
    - A complete M-ary tree has height that is roughly $\log_M(N)$ instead of $\log_2(N)$
    - If M = 20, then $\log_{20}(2^{20}) < 5$
- We can speedup the search significantly

Example

- Standard disk page size 8192 bytes
- Assume keys use 32 bytes, pointers use 4 bytes
- Keys uniquely identify data elements
- $32*(M-1)+4*M = 8192$
- $M = 228$ nodes in a page
- $\mathrm{Log}_{228}\ 33\mathrm{x}10^{12} = 5.7$ (disk accesses)
- Each search takes 0.047 seconds

B-Tree (actually, B+-Tree)

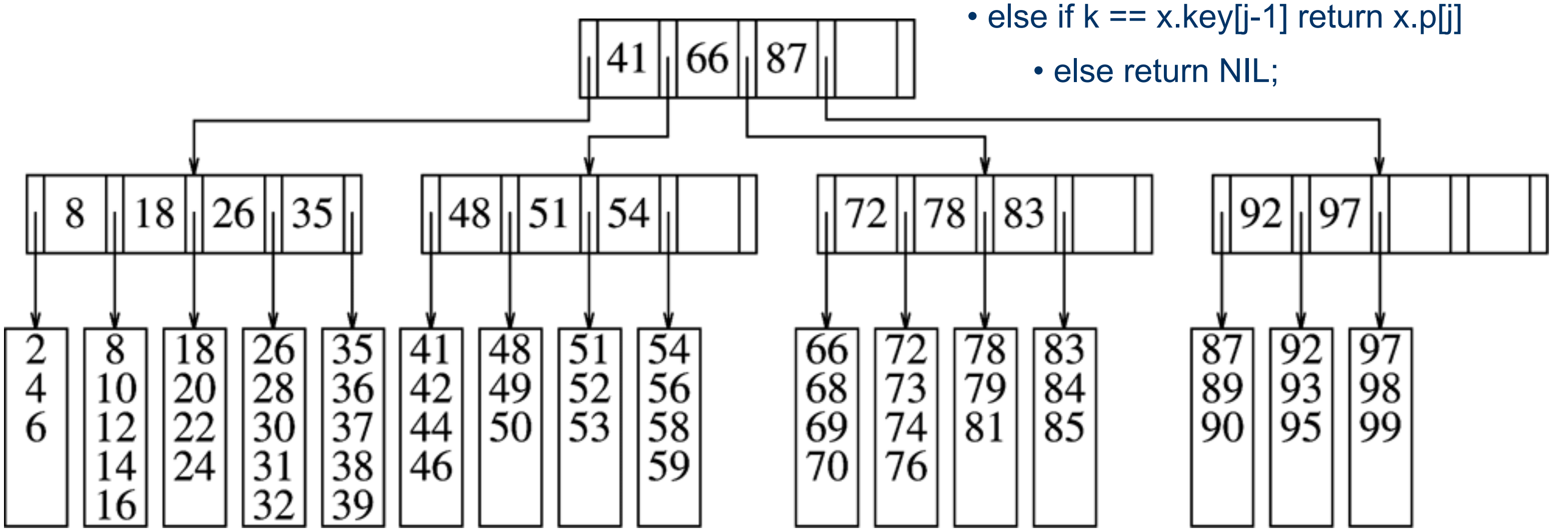
- A B^+-tree of order M is an M-ary tree with the following properties
  - Data items are stored at the leaves
  - Non-leaf nodes store up to M-1 keys; keys at node are sorted
  - Non-leaf nodes have between $\left\lceil \frac{M}{2} \right\rceil$ and M links to children, except for root
  - The root is either a leaf or has from two to M children, 1 and M-1 keys
  - All leaves at the same depth, and contain the data items for the tree
  - Leaves have between $\left\lceil \frac{L}{2} \right\rceil$ and L data items
- Requiring nodes to be half full avoids degenerating into binary tree

B-Tree vs B$^+$-Tree

- B Trees originally proposed in 1970 (Bayer and McCreight)
    - Data items are stored at interior nodes and at the leaves
    - Problem: data items can be much larger than simple navigation nodes, making this impractical!
    - Makes interior nodes larger, less keys possible, increased height
- B$^+$ trees store all data at leaves
    - Leaf nodes store less data items (L vs M-1) because data items are larger
    - Shorter trees, better fit to hierarchical memory
    - To make things fast, first couple of levels of B$^+$ trees kept in main memory
    - Choosing L:
        - Assuming a data element requires 256 bytes; leaf node 8192 bytes implies L=32
        - Each leaf node has between 16 and 32 data elements

B^+ Tree Search

- B^+ tree of order 5
  - Nodes have 2-4 keys and 3-5 children, leaves have 3-5 data elements
- Example: Search for 53, search for 41
  - Searching node can be done with binary search
  - Some ambiguity, resolved by convention: pointer to the right of 41 includes keys $41 \leq k < 66$



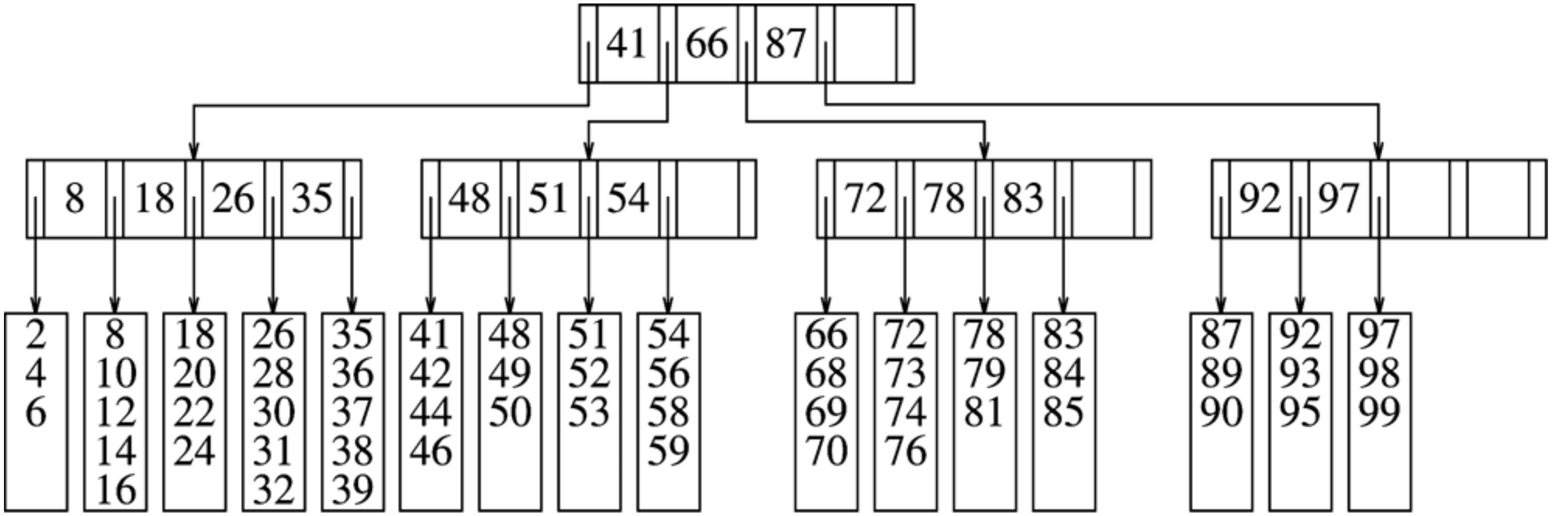
- Node: x.n = number of keys. x.key[j] = j-th key
  - x.c[j] = pointer to j-th child node; x.leaf = Boolean,
  - x.p[j] = pointer to the record corresponding to x.key[j] in leaf

Search(node, k) for k starting at node x:

- j= 0; while j< x.n and k $\geq$ x.key[j]: j++;
- if not x.leaf: Search(x.c[j], k)
- else if k == x.key[j-1] return x.p[j]
  - else return NIL;

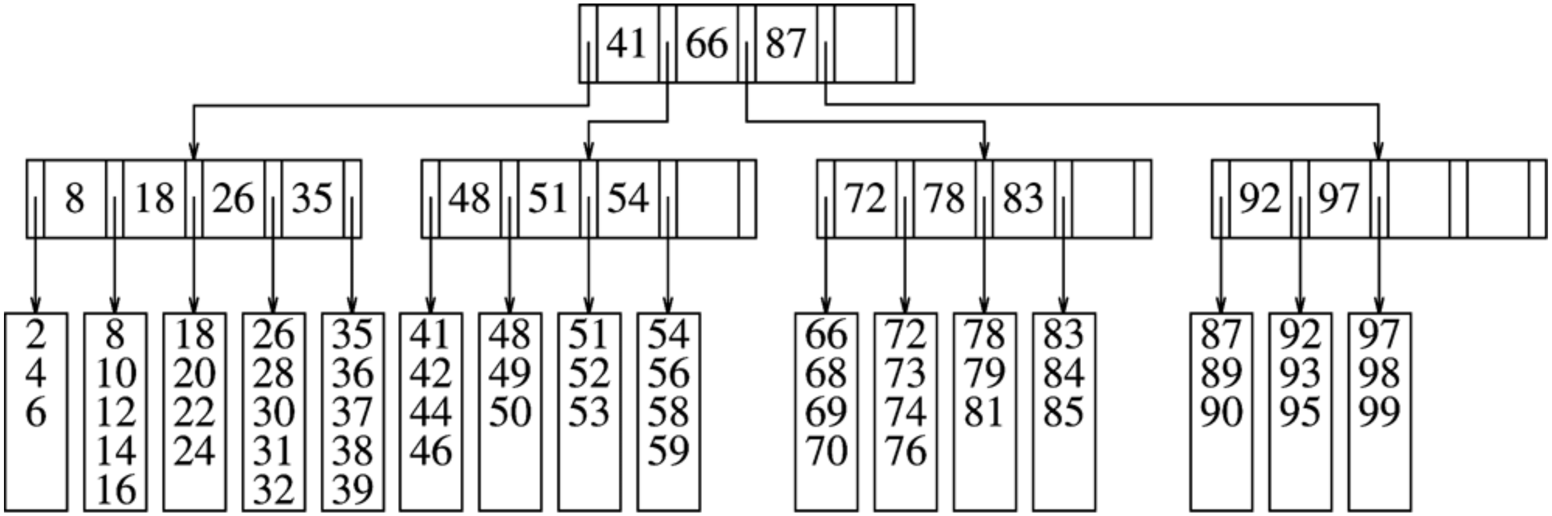
B^+ Tree Insert

- Insert k
  - Navigate to leaf where key should be
  - **Case 1**: if there is room in that leaf (less than L keys): just add it there, sort key with other keys
  - e.g. Insert 55



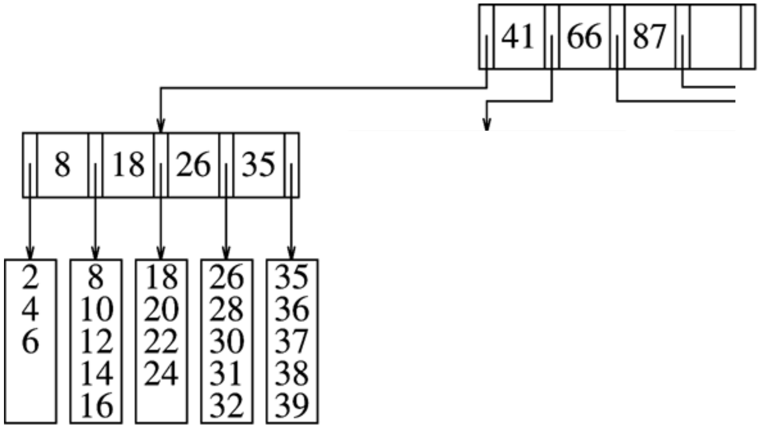
B$^+$ Tree Insert - 2

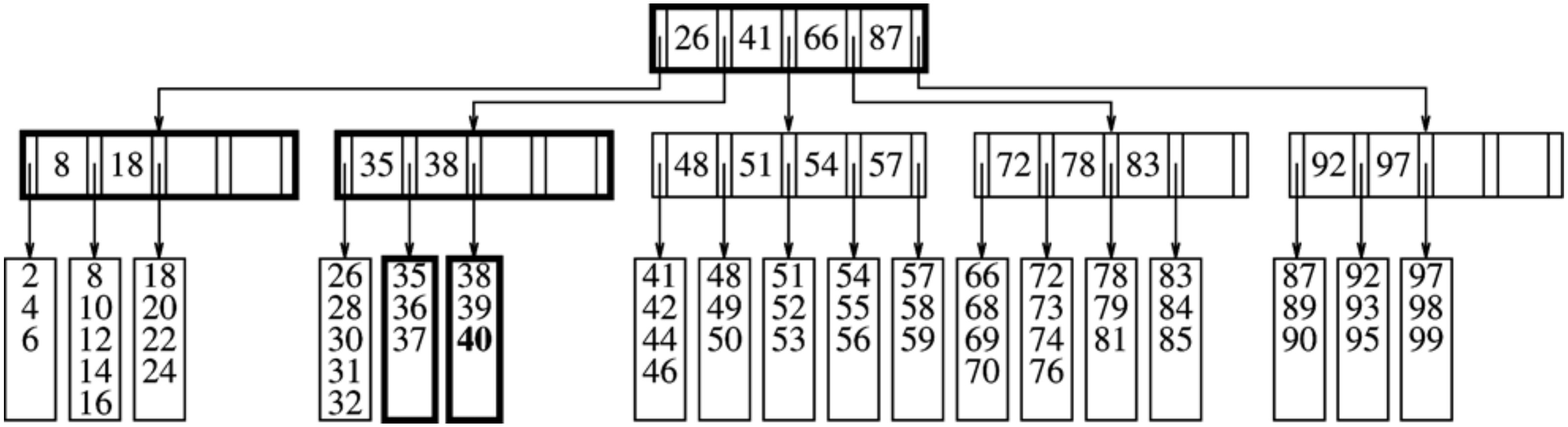
- Insert k
  - Navigate to leaf where key should be
  - **Case 2**: Leaf has L keys already (e.g. insert 40)
    - Need to split the leaf; split the leaf, promote middle key to parent node
    - May need to split parent if no room there…



B^+ Tree Insert: Splitting

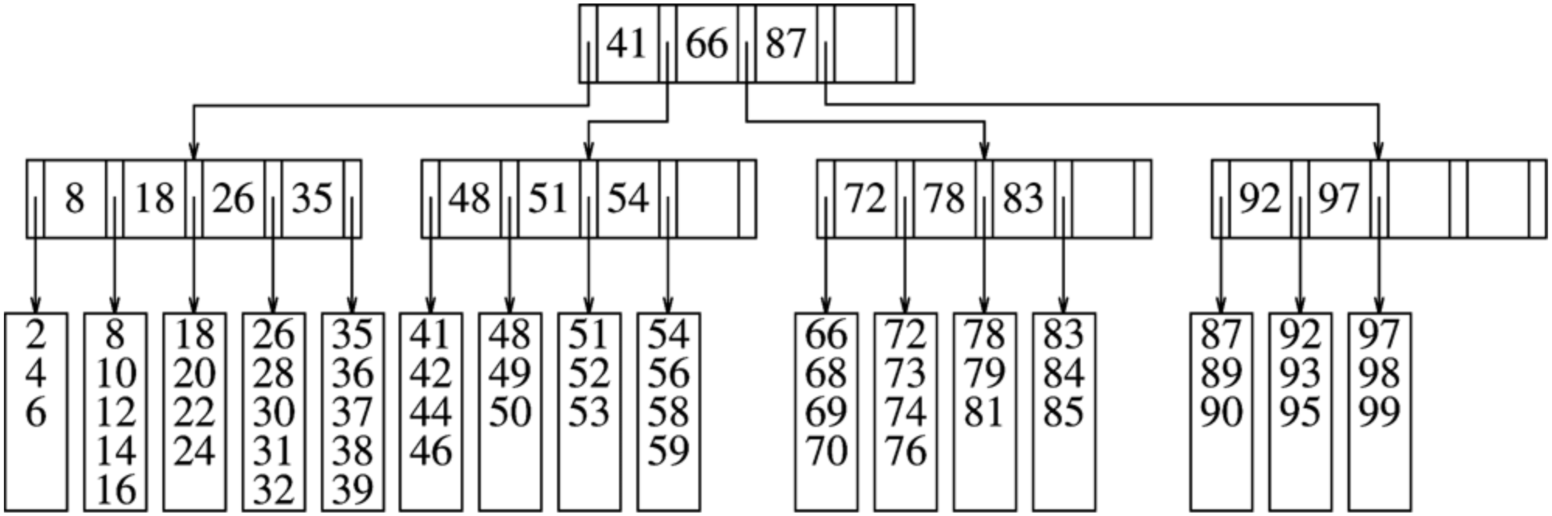
- Insert 40: split leaf, leave keys in tree. Move rightmost key in right split up to parent for navigation, add navigation link in parent
- Splitting interior node: remove move middle key of saturated node and add to parent, break node into two nodes, add navigation links in parent




26 41 66 87
8 18
35 38
48 51 54 57
72 78 83
92 97
2 4 6
8 10 12 14 16
18 20 22 24
26 28 30 31 32
35 36 37
38 39 **40**
41 42 44 46
48 49 50
51 52 53
54 55 56
57 58 59
66 68 69 70
72 73 74 76
78 79 81
83 84 85
87 89 90
92 93 95
97 98 99

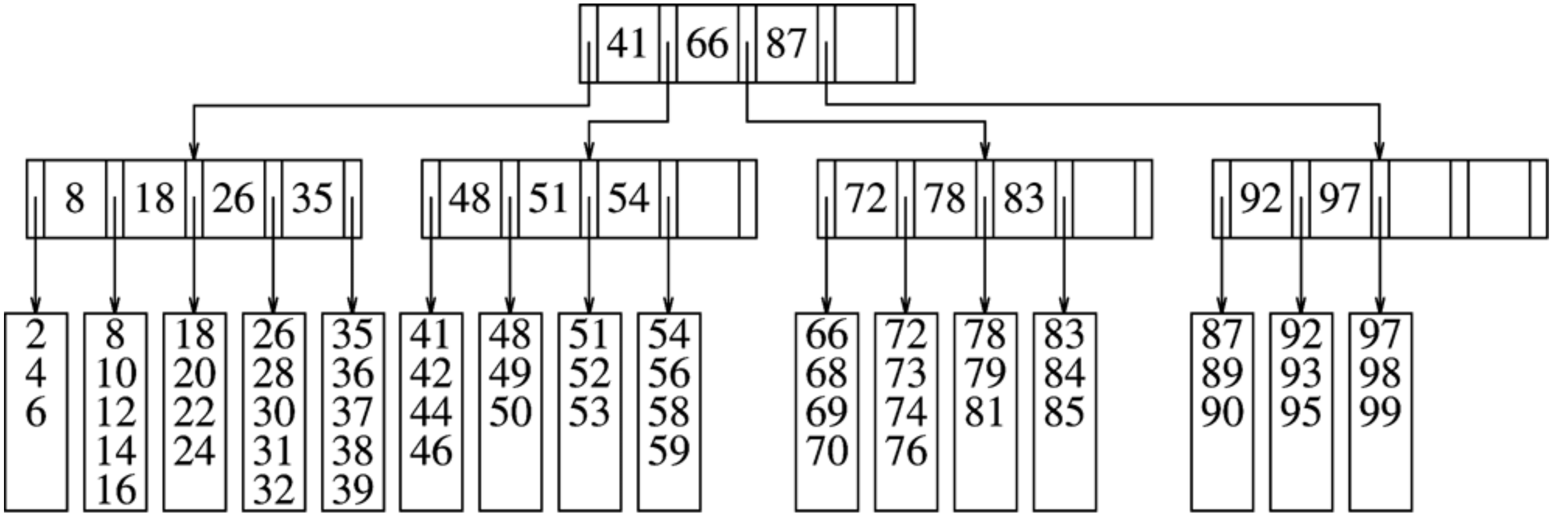
B$^+$ Tree Delete

- Delete k
  - Navigate to leaf where key should be
  - **Case 1**: leaf has more than minimum number: Just delete it!
  - e.g. Delete 31



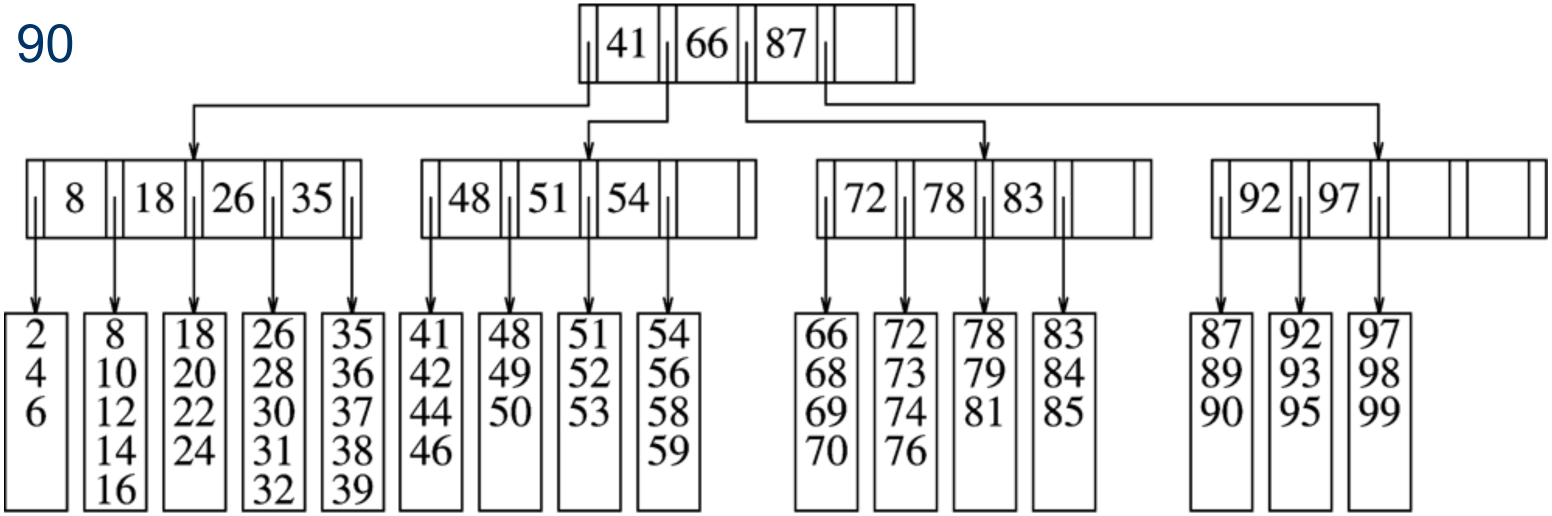
B+ Tree Delete

- Delete k
    - Navigate to leaf where key should be
    - **Case 2**: leaf has minimum number: Delete key. If one of immediate neighbor siblings has more than minimum number, transfer key to leaf, adjust value of key in parent
    - e.g. Delete 53



B^+ Tree Delete

- Delete k
  - Navigate to leaf where key should be
  - **Case 3**: leaf has minimum number: Delete key. If immediate neighbor siblings have only minimum number, merge with one of neighbors, remove key in parent (May cause underflow in parent, so continue delete)
  - e.g. Delete 90


41
66
87
8
18
26
35
48
51
54
72
78
83
92
97
2 4 6
8 10 12 14 16
18 20 22 24
26 28 30 31 32
35 36 37 38 39
41 42 44 46
48 49 50
51 52 53
54 56 58 59
66 68 69 70
72 73 74 76
78 79 81
83 84 85
87 89 90
92 93 95
97 98 99