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

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 $(33 \times 10^{12}) = 44.9$
    - 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

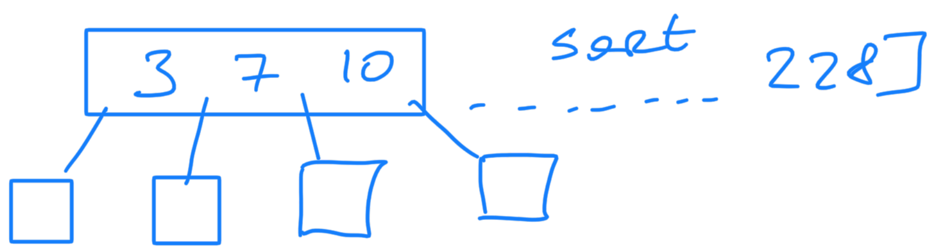
- Idea: allow a node in a tree to have many children
- Less disk access = less tree height = more branching
- As branching increases, the depth decreases
- An M-ary tree allows M-way branching
- Each internal node has at most M children
  - A complete M-ary tree has height that is roughly $\log_M(N)$ 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
- $\text{Log}_{228}\ 33\text{x}10^{12} = 5.7$ (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

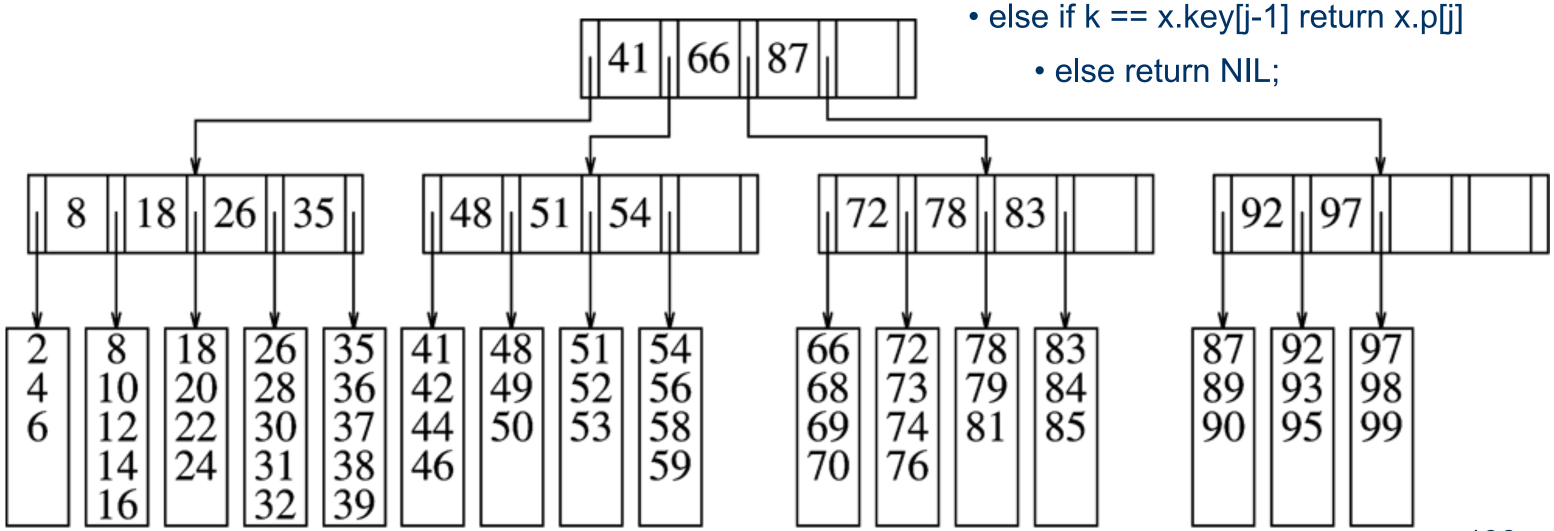
B+ Tree Search

- 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

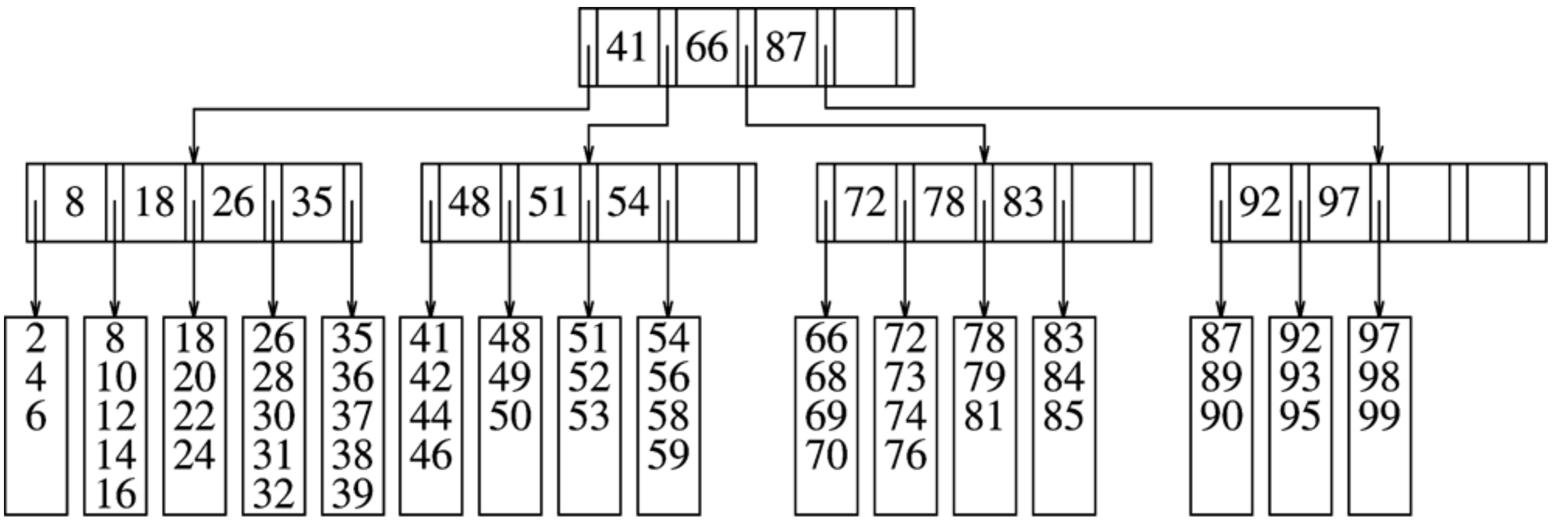
Search(node, k) for k starting at node x:

- 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;



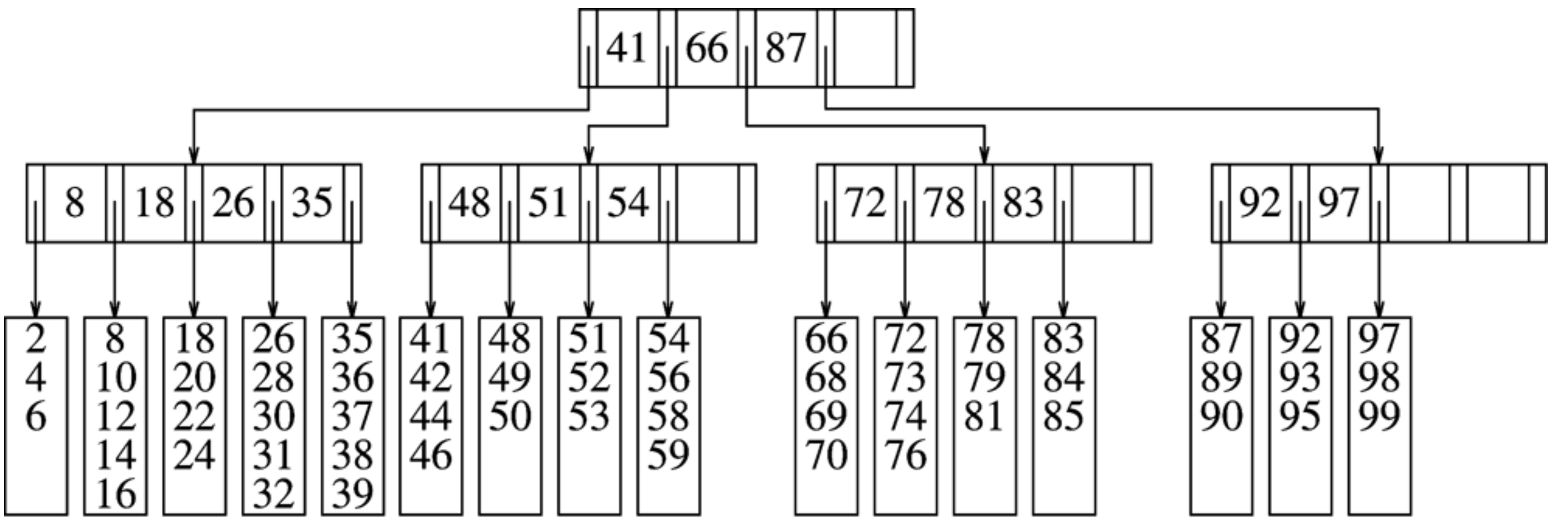
B+ Tree Insert

- 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



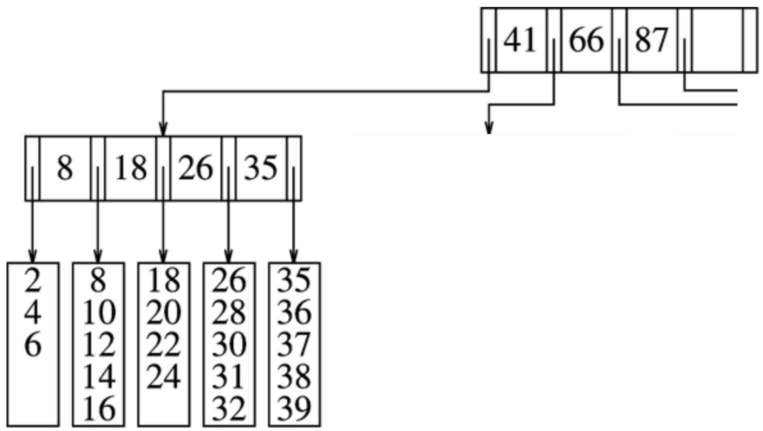
B+ Tree Insert - 2

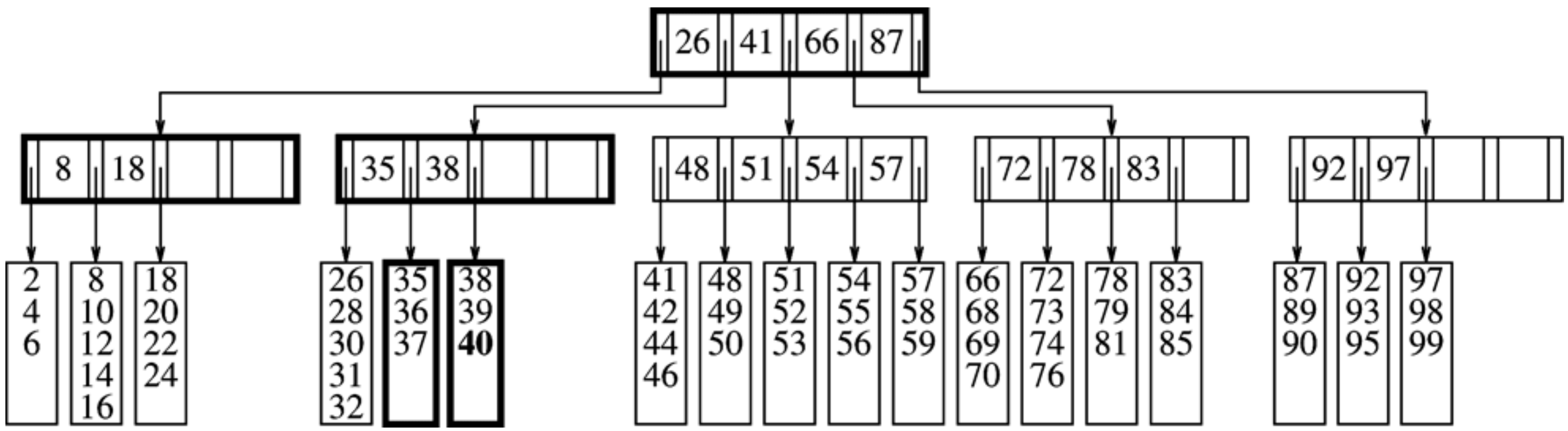
- Insert k
  - Navigate to leaf where key should be
  - **Case 2**: Leaf has L keys already (e.g. insert 40)
    - Need to split the leaf; split the leaf, promote middle key to parent node
    - May need to split parent if no room there…



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

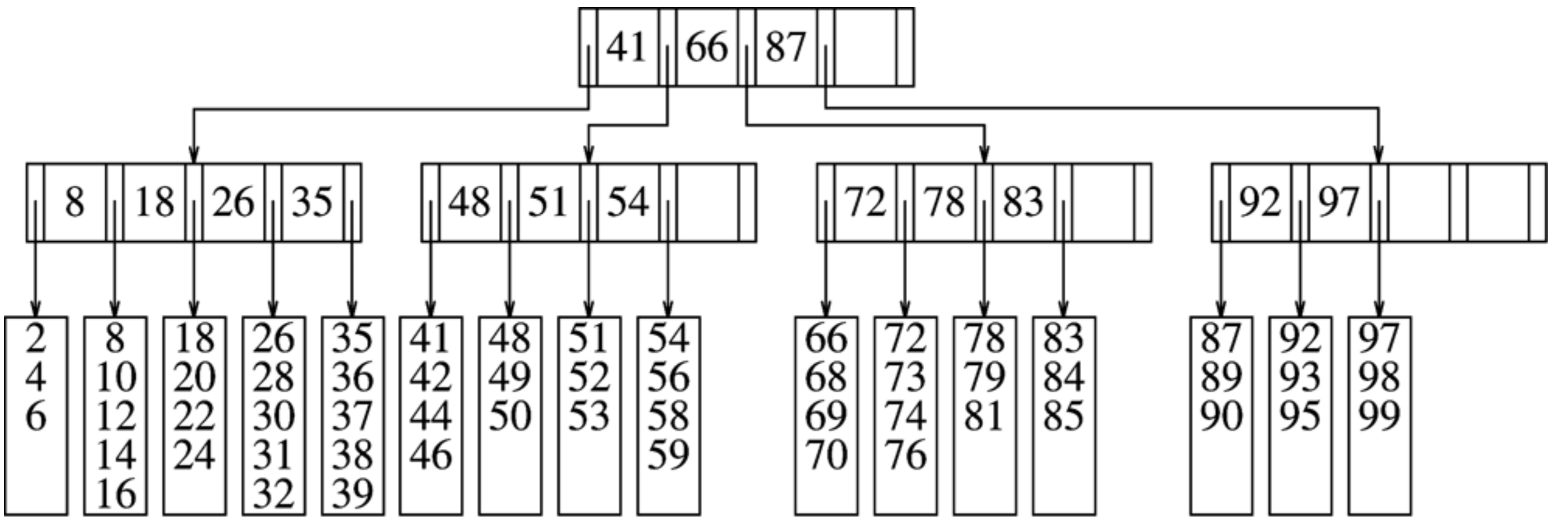




26 41 66 87
8 18
35 38
48 51 54 57
72 78 83
92 97
2 4 6
8 10 12 14 16
18 20 22 24
26 28 30 31 32
35 36 37
38 39 **40**
41 42 44 46
48 49 50
51 52 53
54 55 56
57 58 59
66 68 69 70
72 73 74 76
78 79 81
83 84 85
87 89 90
92 93 95
97 98 99

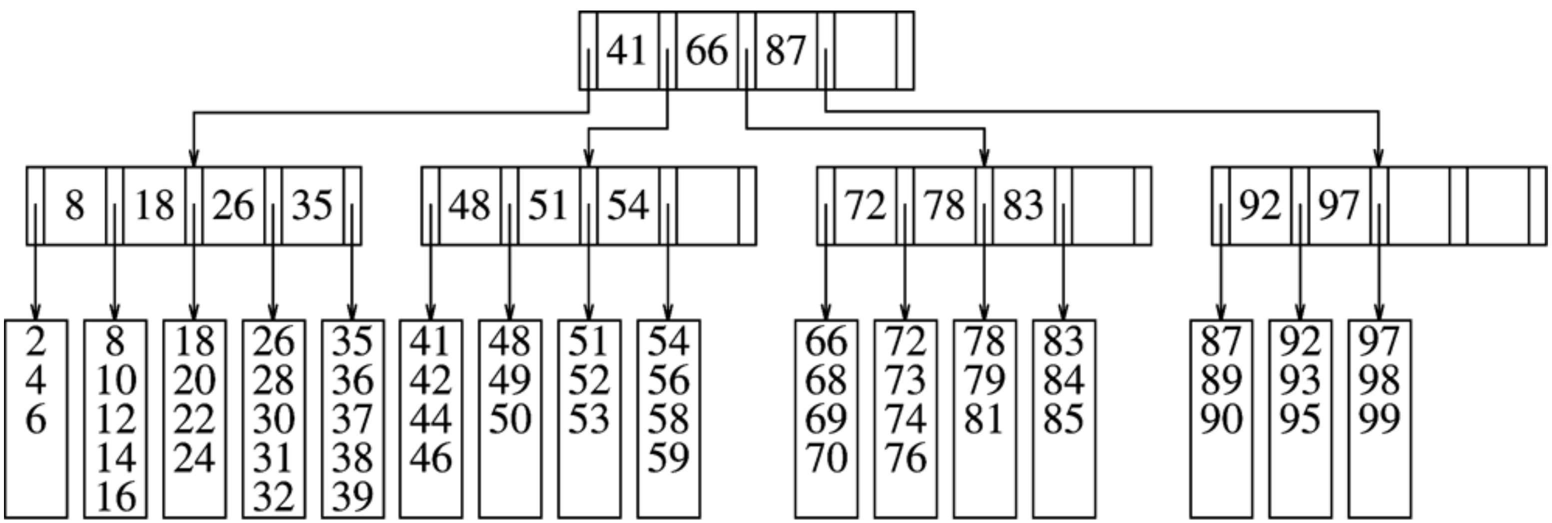
B+ Tree Delete

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



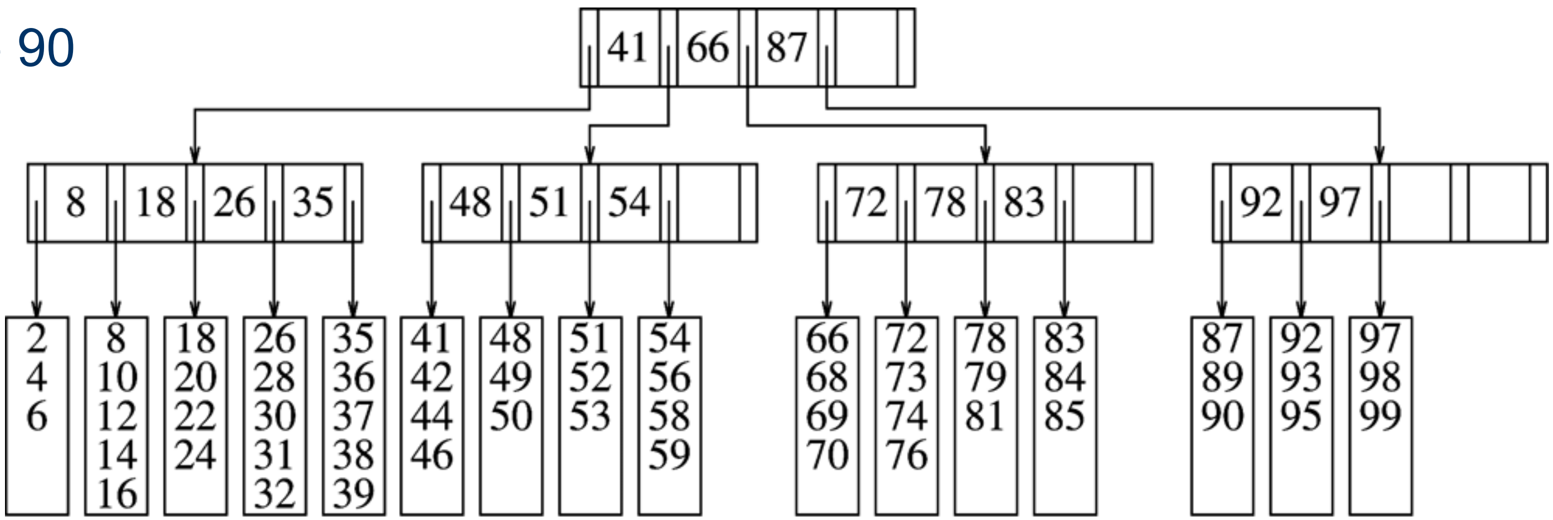
B⁺ Tree Delete

- 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



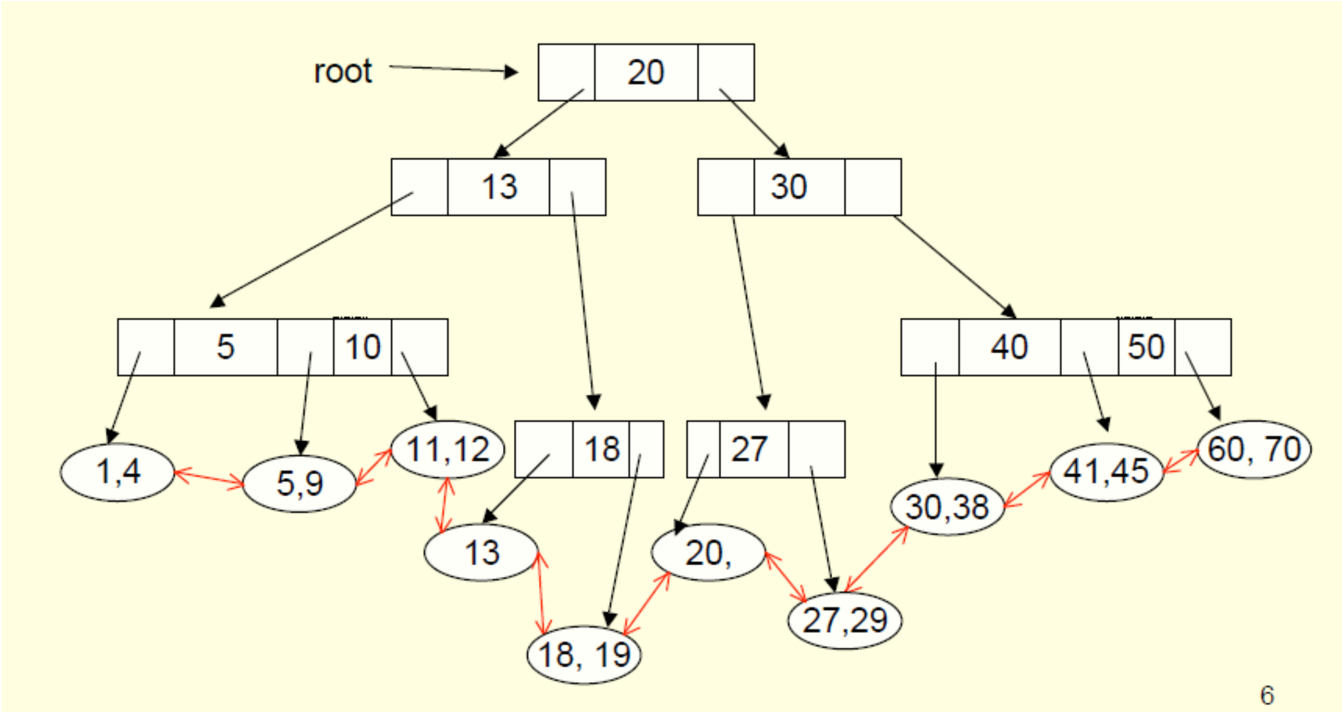
B^+ Tree Delete

- 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



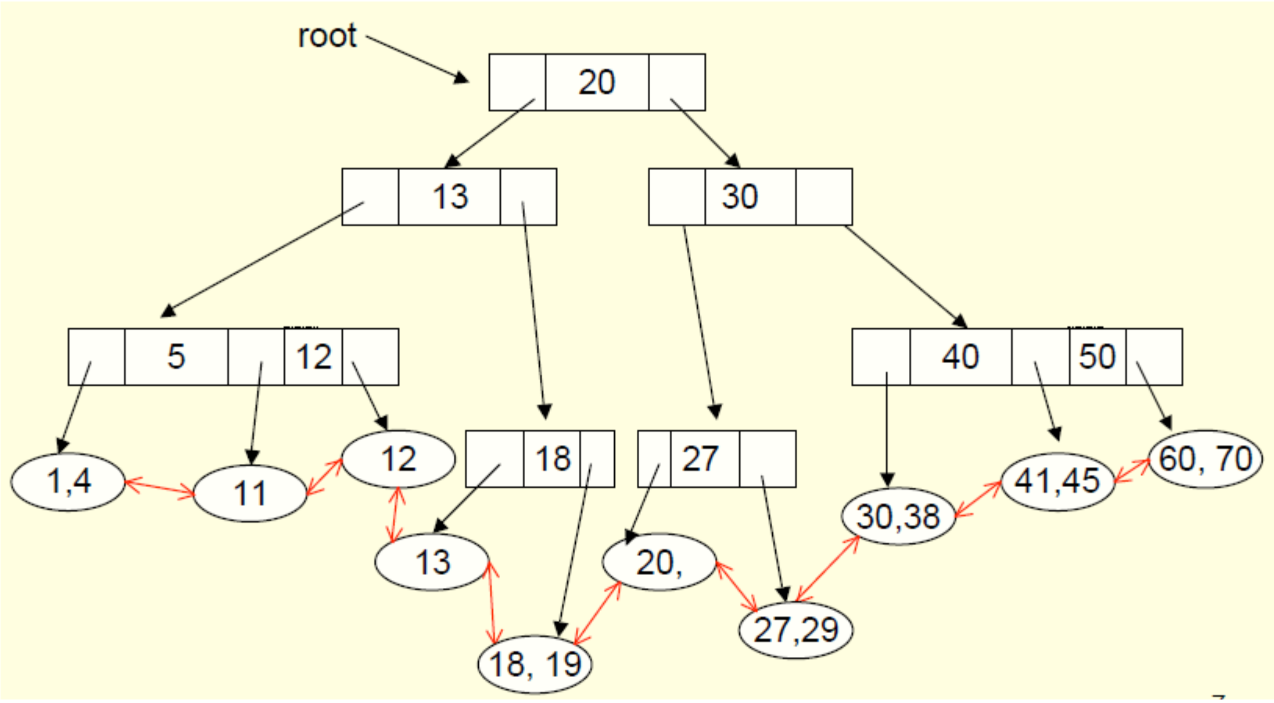
B⁺ Tree Delete

- Delete 5
  - No problem, leaf has extra key



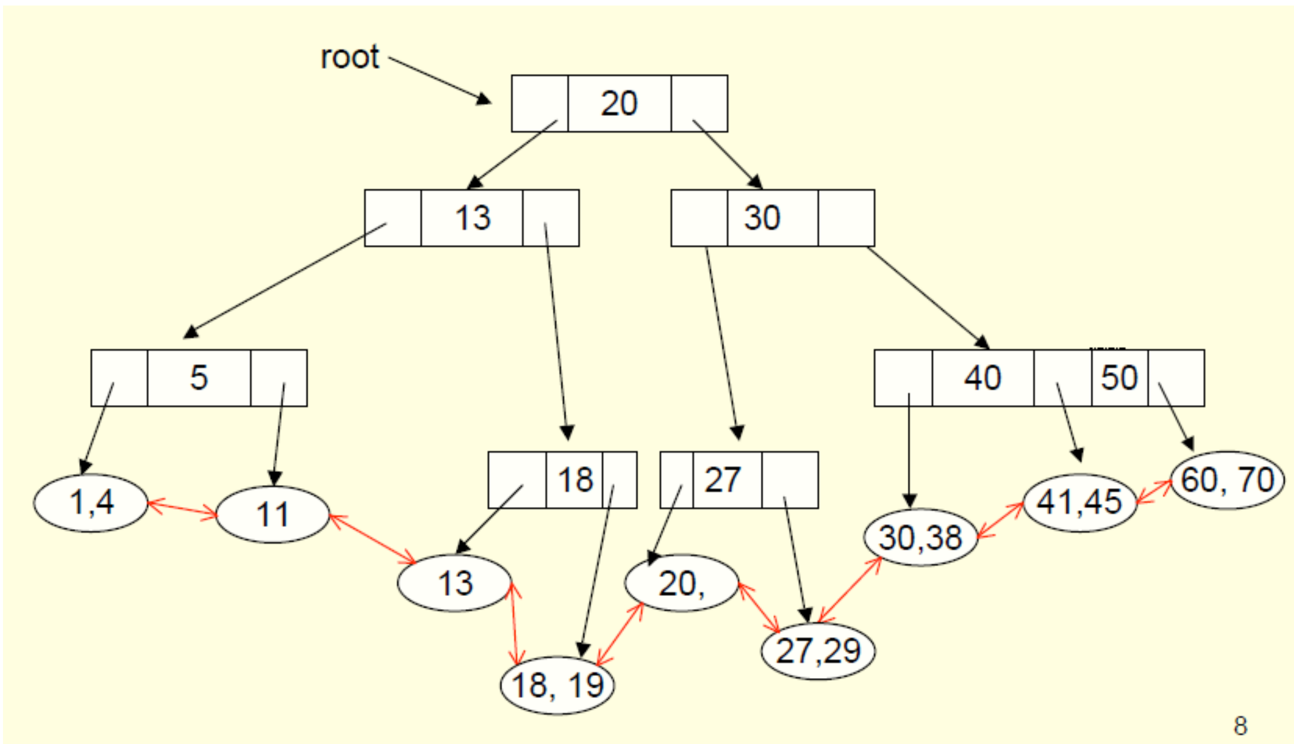
B^+ Tree Delete

- Delete 5
  - No problem, leaf has extra key
- Delete 9: Now have underflow, must borrow key



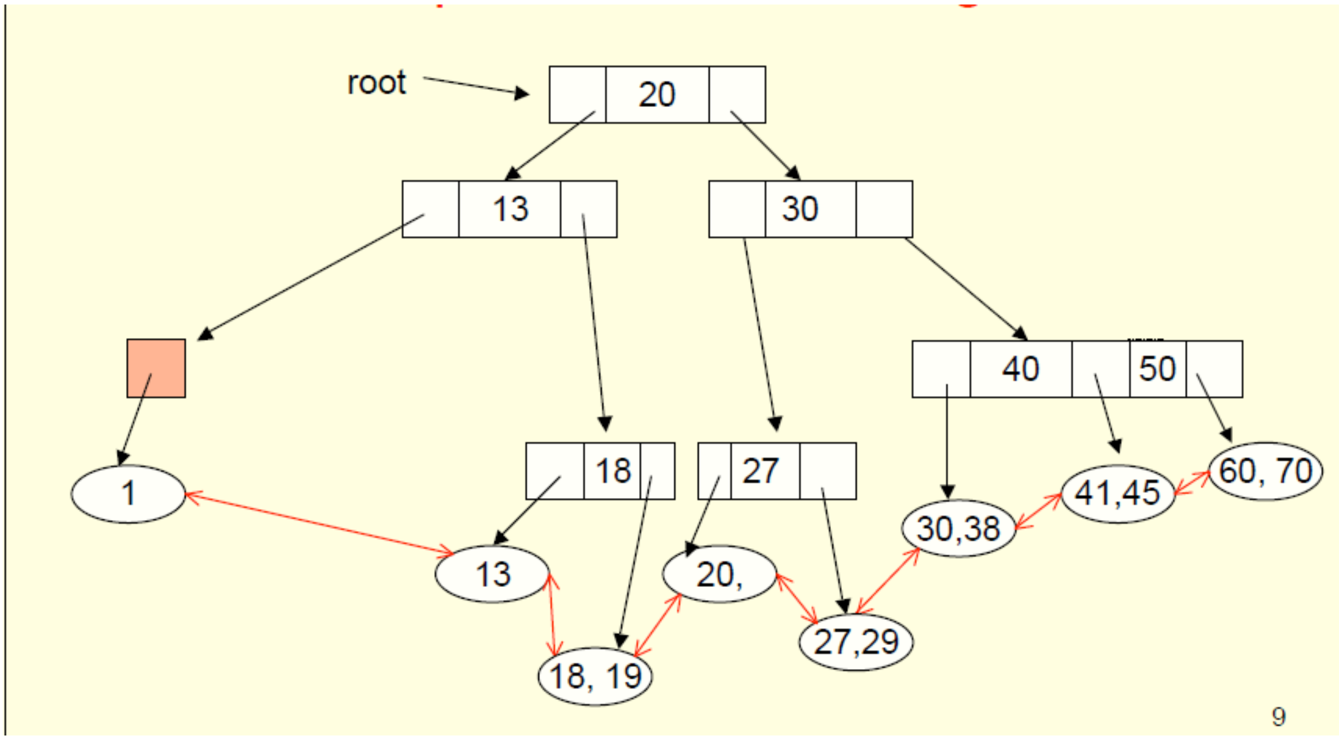
B+ Tree Delete

- Delete 12
  - Underflow, must delete node

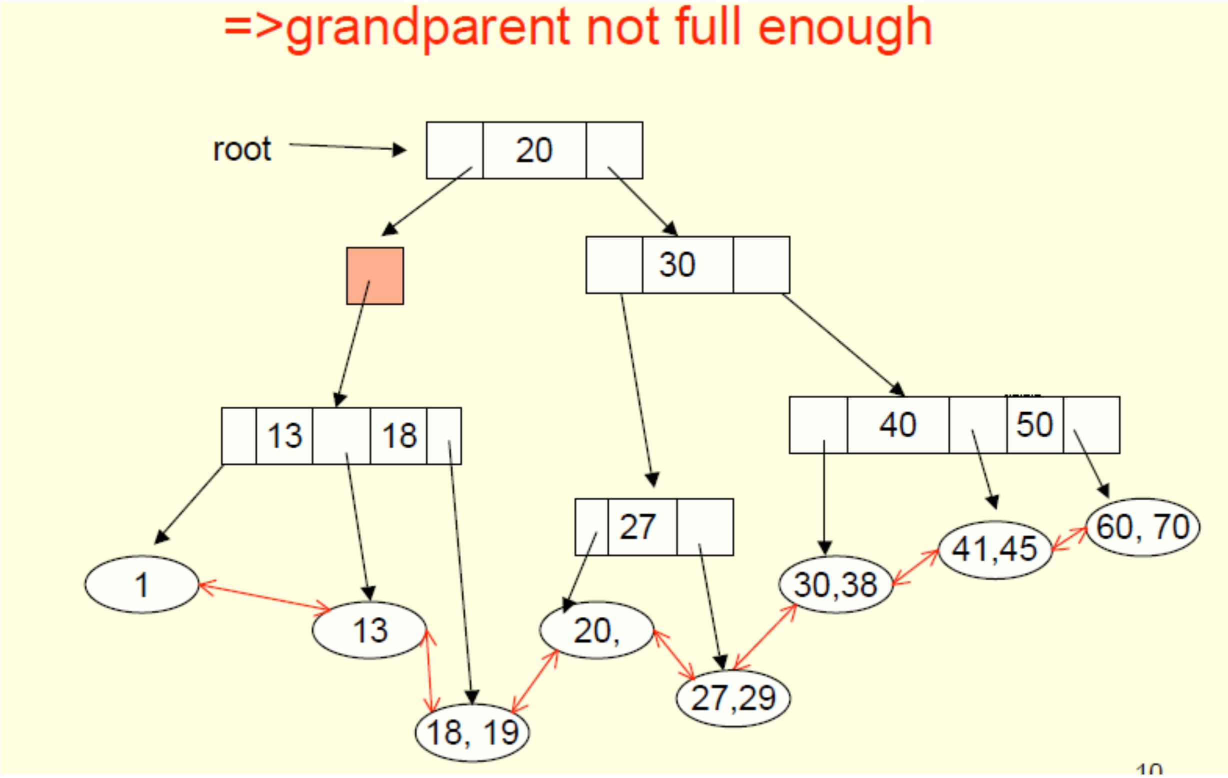


B+ Tree Delete

- Delete 4
  - No problem, extra key
- Delete 11
  - Big problem!
  - Underflow parent
  - Parent must merge

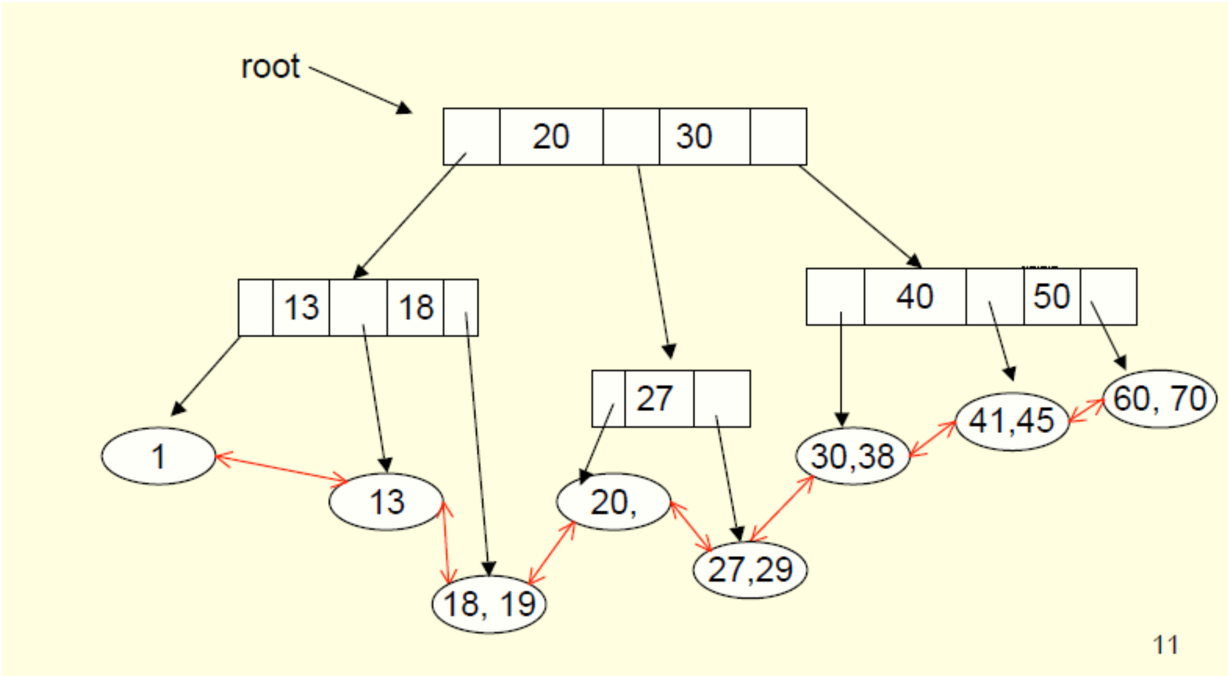


B^+ Tree Delete

- Merge parent
  - Grandparent now has underflow
- 

B+ Tree Delete

- Merge grandparent
  - Root has underflow, delete root
-



B^+ Tree Summary

- Insert, delete, find are O(log(n))
  - Navigate to leaf where key should be
- Often, use pointers between neighbor leaves to obtain sorted records
- Maintain sorted keys inside node for easier navigation
- Variations:
  - B* tree very similar to B+ tree
    - instead of splitting a page in 1/2 when it overflows, it gives some records to its neighbor siblings.
    - If neighbor is full, then 2 nodes split into 3. This makes nodes at least 2/3 full.
    - Similarly, in deletions, one starts to shift records when number is less than 2/3 of maximum

Priority Queues

- Queues are a standard mechanism for ordering tasks on a first-come, first-served basis
  - However, some tasks may be more important or timely than others (higher priority)

- Priority queues
  - Store tasks using a partial ordering based on priority
  - Ensure highest priority task at head of queue

- Heaps are the underlying data structure of priority queues

Priority Queue

- Main operations
  - **Insert** (i.e., enqueue)
  - **deleteMin** (i.e., dequeue)
    - Finds the minimum element in the queue, deletes it from the queue, and returns it
  - **getMin**: Find the highest priority job
  - **deleteAny**: Delete a job from the queue
  - **Create**: Create a priority queue from a list of jobs
  - **Merge**: merge two priority queues into one
  - **decreasePriority**: decrease priority of existing job in queue

- Desired Performance
  - Goal is for operations to be fast: O(1) to O(log(n))
  - Will be able to achieve O(logN) time insert/deleteMin

Simple Implementations

- Unordered list
  - O(1) insert
  - O(n) deleteMin
  - O(n) for several others
- Ordered list
  - O(N) insert
  - O(1) deleteMin
  - Still O(n) for others

- Balanced BST (red-black tree, splay tree)
  - O(log(n)) insert and deleteMin
  - O(log(n)) decrease priority

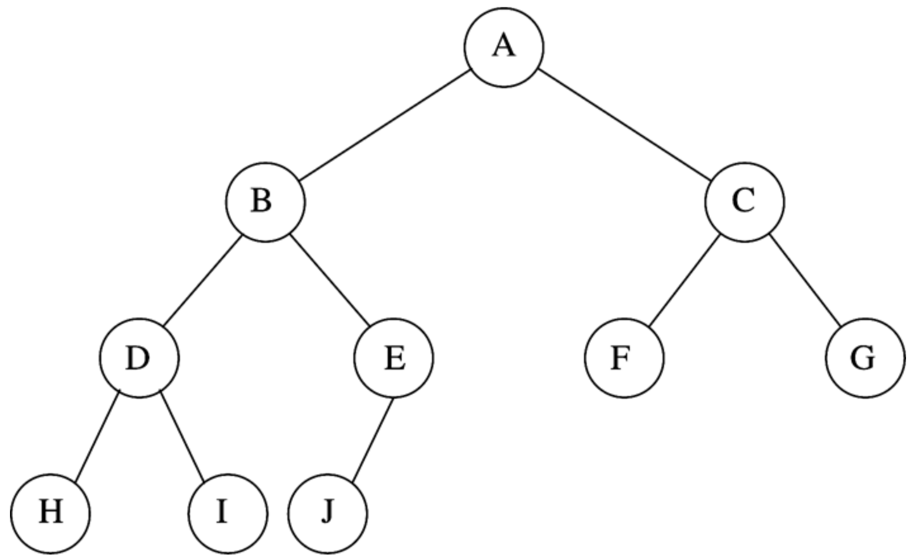
- Observation: We don't need to keep the priority queue completely ordered

Binary Heap

- A binary heap is a binary tree with two properties
  - Structure property
    - A binary heap is a complete binary tree
    - Each level is completely filled
    - Bottom level may be partially filled from left to right
  - Heap-order property
    - Parent node must have key less than or equal to the keys of its children

- Complete binary tree —> easy implementation as array
  - Height of a complete binary tree with N elements is **Floor**[logN]

Binary Heap Example



- Map nodes into array positions, starting from 0:
  - Root node: element 0
  - Children of node in position k: are in positions 2k+1, 2k+2
  - Parent of node in position k: is in position (k-1)/2

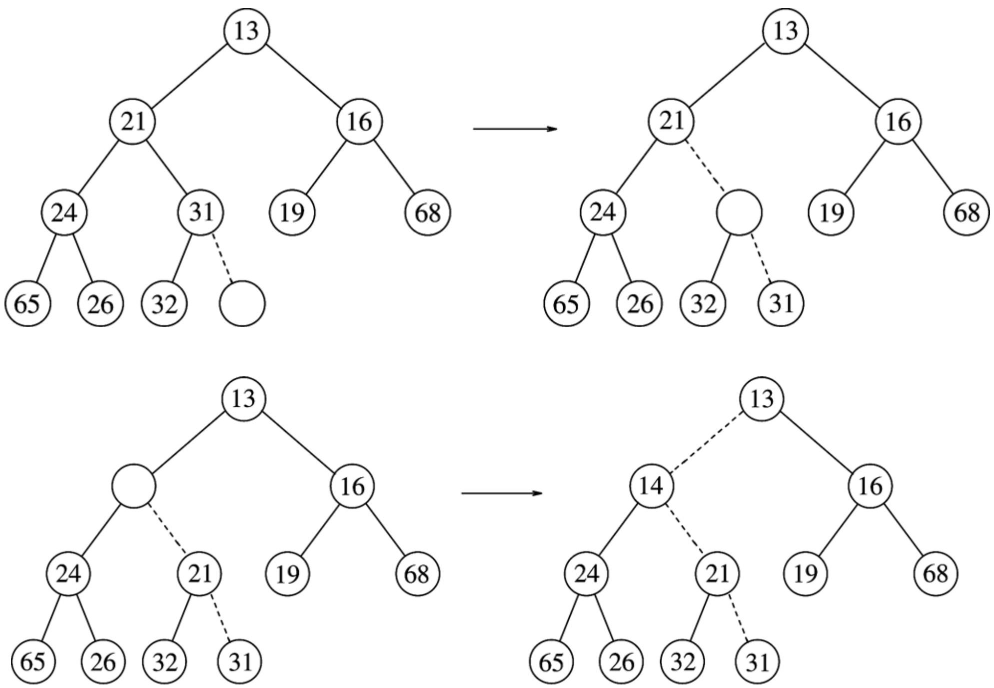
Heap Insert

- Insert new element into the heap at the next available slot ("hole")
  - According to maintaining a complete binary tree

- Then, "percolate" the element up the heap while heap-order property not satisfied
  - Upheap operation

Heap Insert

Insert 14:



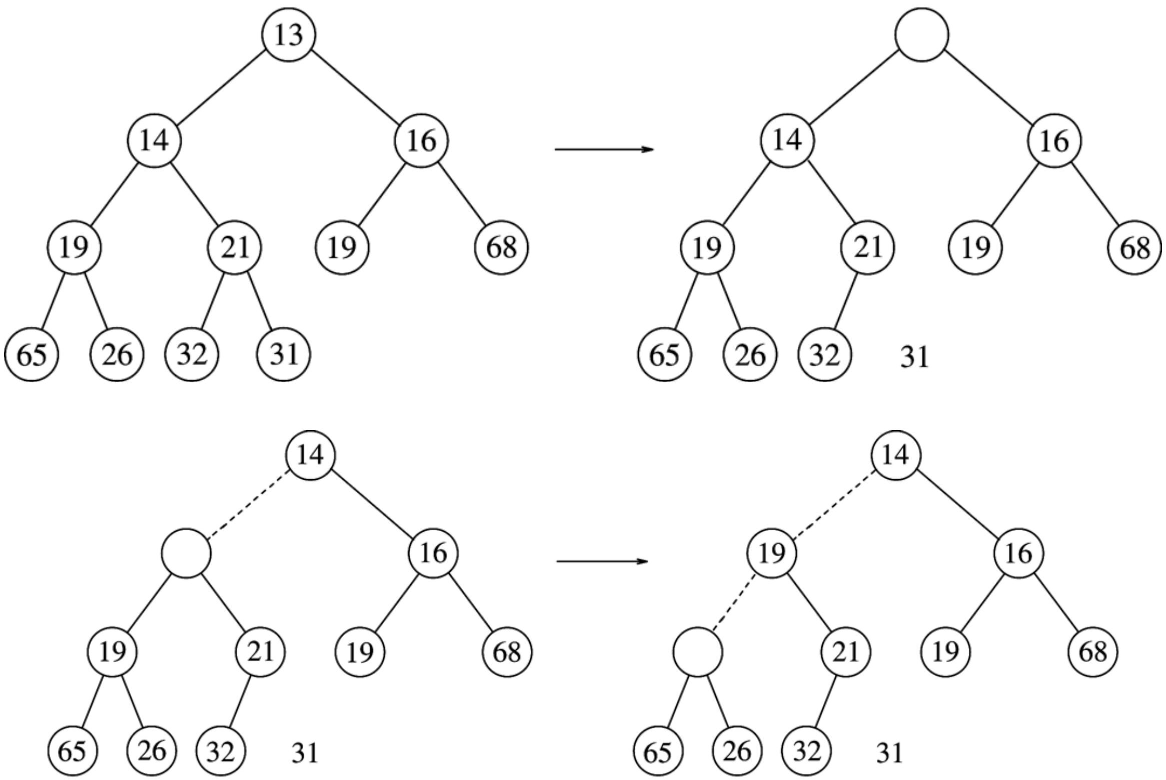
Creating the hole and building the hole up

```
Upheap(k)
    If (k=0) return;
    If V[k] < V[(k-1)/2]
        temp = V[(k-1)/2]
        V[(k-1)/2] = V[k]
        V[k] = temp;
        k = (k-1)/2
        Upheap(k)
    else return;
```

Heap DeleteMin

- Minimum element is always at the root
  - O(1) to find
  - On deleteMin, heap decreases by 1 in size
- Operations
  - Remove element in array[0]
  - Move last element into array[0], shrink array size by 1
  - Percolate down while heap-order property not satisfied
    - Downheap(k)

Heap DeleteMin




```
downheap(k,n): n = # of elements i
   V = V[k]
   j = (2*k+1)  // left child
   while (j <= n){
      if j < n-1 and V[j] > V[j+1]:
         j = j+1
      if  V[k] > V[j]:
         V[k] = V[j]
         k = j;
         j = 2*j + 1 // leftchild
      else exit
   V[j] = V;
```

Building a Heap from Array

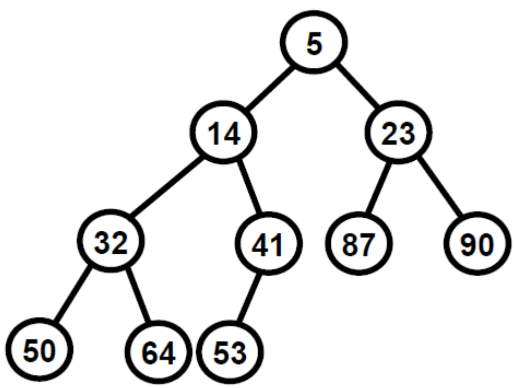
- Construct heap from initial array of n items
  - Solution 1: Perform n inserts
    - O(n) average case, but O(n log (n)) worst-case (reverse sorted)
  - Solution 2
    - Perform downheap for all nodes from position n/2 - 1 to 0.
    - Buildheap(n): for (i = n /2 -1; i > = 0; i--) downheap(i,n)
    - O(N) worst case!

Complexity: 1/2 elements don't move, 1/4 move 1, 1/8 move 2, ...

$$T(n) = \frac{n}{4}(1 + 2 * \frac{1}{2} + 3 * \frac{1}{4} + \ldots) = \frac{n}{4}\sum_{k=1}^{\log(n)} \frac{k}{2^{k-1}} \leq n \Rightarrow \in \Theta(n)$$

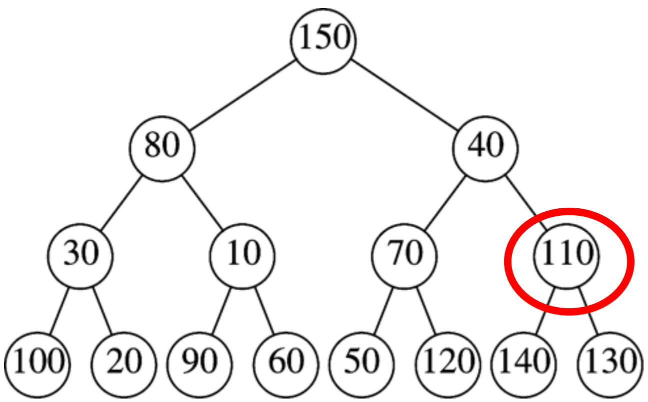
Examples

Insert 43, then 18



Examples

Building a Heap from an array



Complexity of Heap Operations

- Height of heap: O(log(n))
- insert: O(log(n))
  - 2.607 comparisons on average, i.e., O(1)
- deleteMin: O(log(n))
- decreaseKey: Find key, change value, upheap: O(log(n))
- increaseKey: Find key, change value, downheap: O(log(n))
- remove: Find key, swap last element into key position, downheap: O(log(n))
- buildHeap: O(n)
- findMinimum: O(1)

- Merge two heaps: O(n) —> link two arrays and buildHeap

Wishes: Merge of O(1), decrease key of O(1) —> useful in future algorithms…

Fast Sorting Algorithm: HeapSort

- Input: array of numbers (n)
- Step 1: Build heap: O(n)
- Step 2: for k in 1 to n, deleteMin(): Complexity O(n log(n))
- Heapsort complexity: O(n log(n)): as small as the fastest comparison based sort algorithms.
- Only drawbacks: Not stable, and average case is O(n log(n)): not opportunistic

Priority Queue Applications

- Operating Systems: task scheduling
- Graph algorithms: Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, A* search, Branch and Bound, …
- Discrete-event simulation: Have fast access to next event
- Heapsort complexity: O(n log(n)): as small as the fastest comparison based sort algorithms.
- Only drawbacks: Not stable, and average case is O(n log(n)): not opportunistic