

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

→ Hw 1 SW posted (new version)

Due 2/13 8pm

Submit as HW1 folder
in scc, your account.

* Solutions to Quiz + Written
Hw are in folder.

⋮

→ make sort ...

→ make topsortplay ...

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.

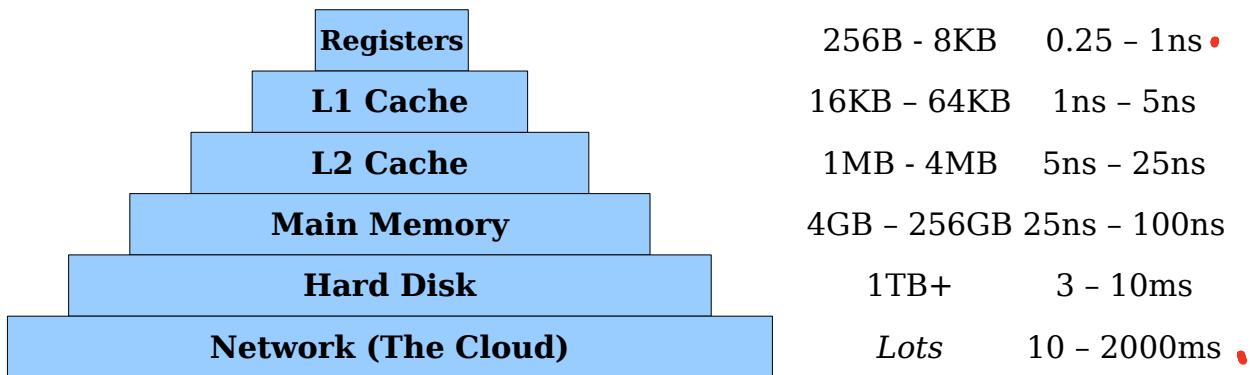
- How are these stored for efficient search? Oracle, SQL? Not in RAM...

Large Databases

- o Use a BST?
 - o Google – 33 trillion items, indexed by IP
 - o Access time
 - o Height = $\log (33 \times 10^{12}) = 44.9$
 - o Assume 120 disk accesses per second: Search takes 0.37 seconds
- o What does Oracle, SQL do?
 - o Use better search trees to reduce disk access
- o How about other solutions (e.g. hash tables, like Python dictionaries?)
 - o Many data base queries imply sorting (hard in hash tables)
 - o 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



Idea: Multiway Search Trees

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

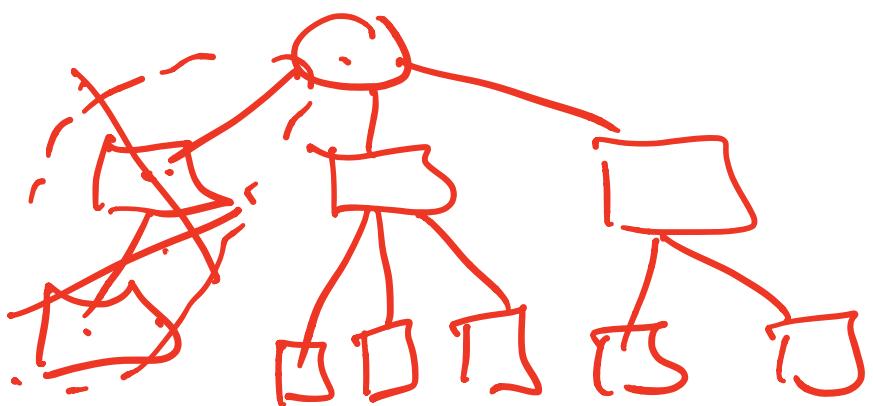
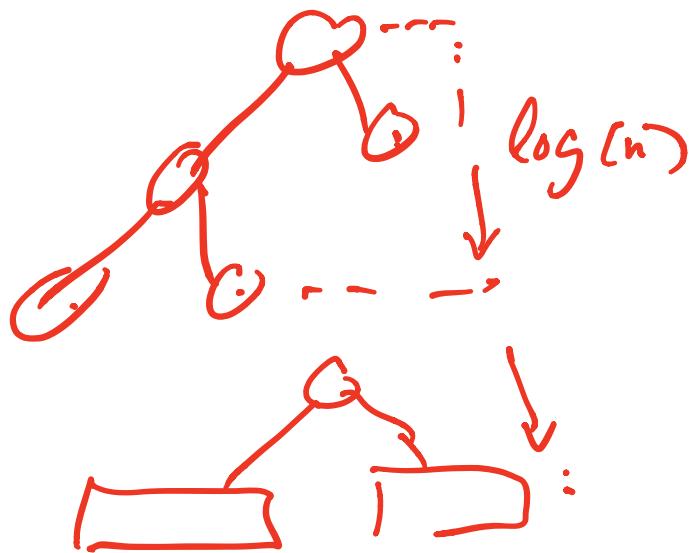
Example

- o Standard disk page size 8192 bytes
- o Assume keys use 32 bytes, pointers use 4 bytes
- o Keys uniquely identify data elements
- o $32*(M-1)+4*M = 8192$
- o M = 228 nodes in a page
- o $\log_{228} 33 \times 10^{12} = \underline{5.7}$ (disk accesses)
- o 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 ✓



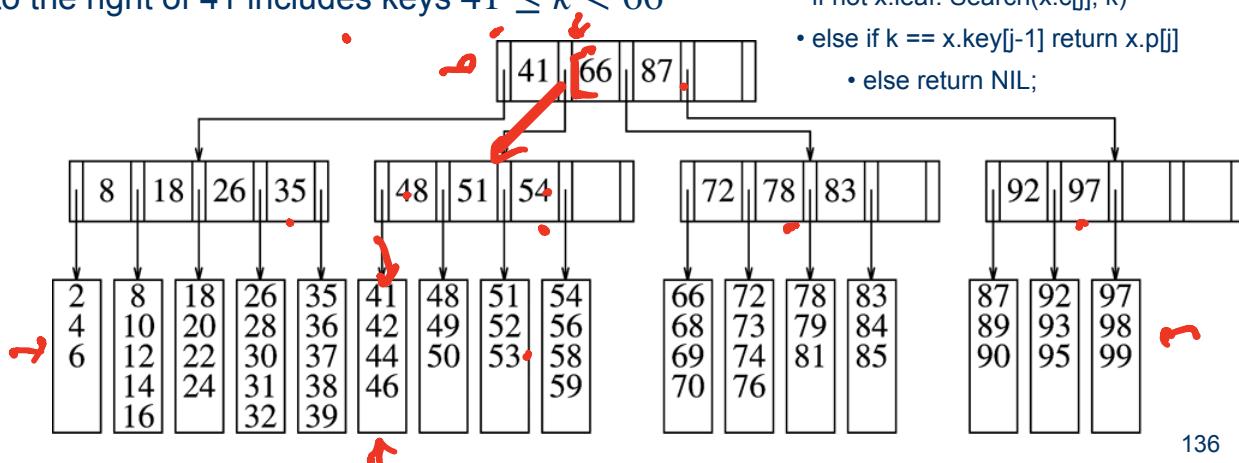


B-Tree vs B⁺-Tree

- 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 ≥ 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;

136

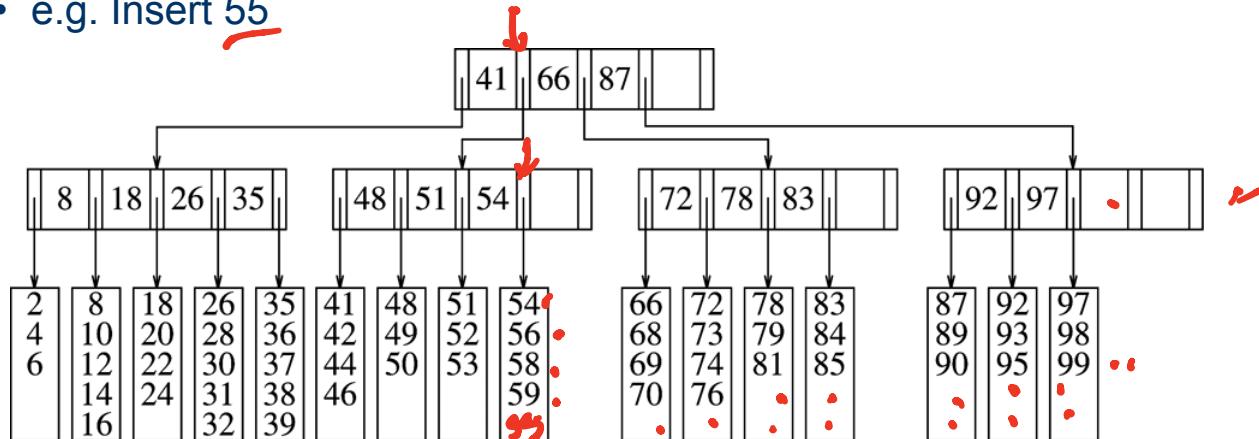
Condition } { ≤ 41 . $41 \leq \dots < 66$
 $66 \leq \dots < 87$ $87 \leq \dots$

1 2 3 4 5 6 7 8 ✓

\downarrow
 3^+ $m=3$

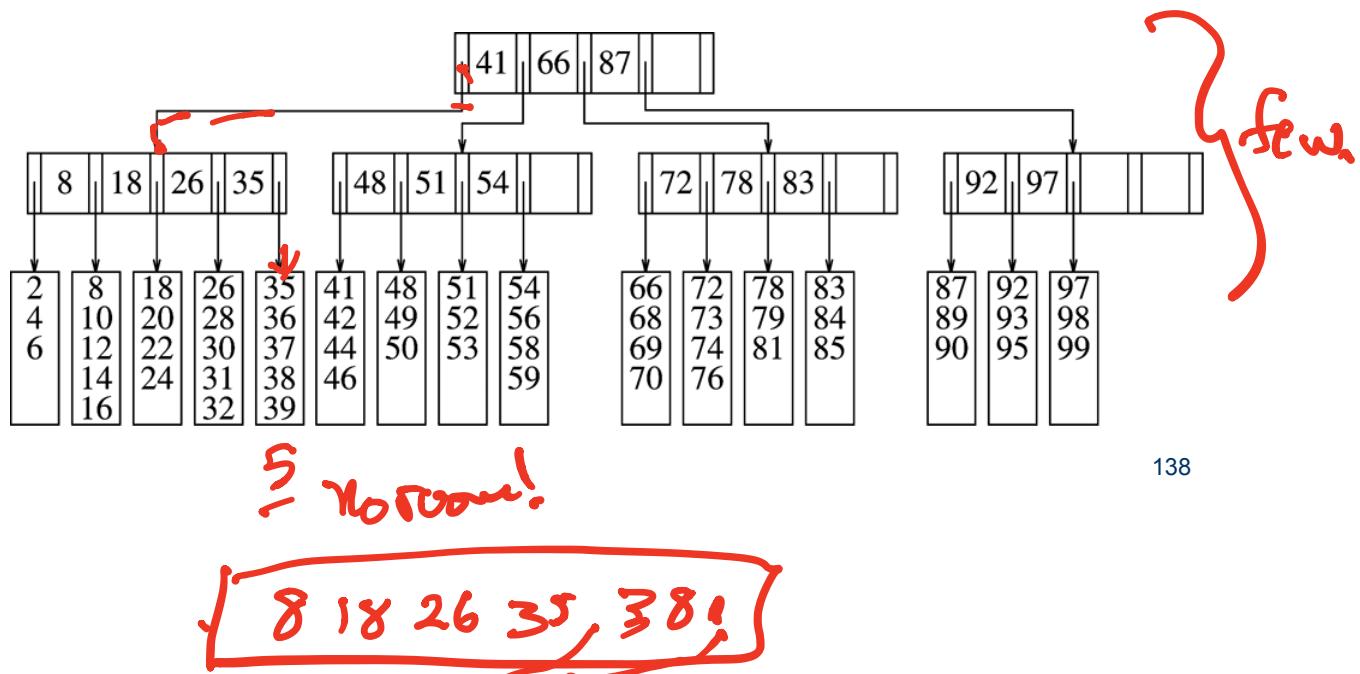
B⁺ Tree Insert

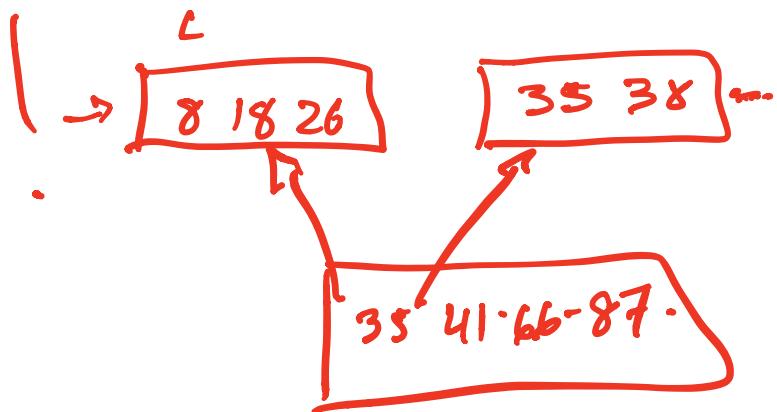
- Insert k
 - Navigate to leaf where key should be
 - **Case 1:** if there is room in that leaf (less than L keys): just add it there, sort key with other keys
 - e.g. Insert 55



B⁺ Tree Insert - 2

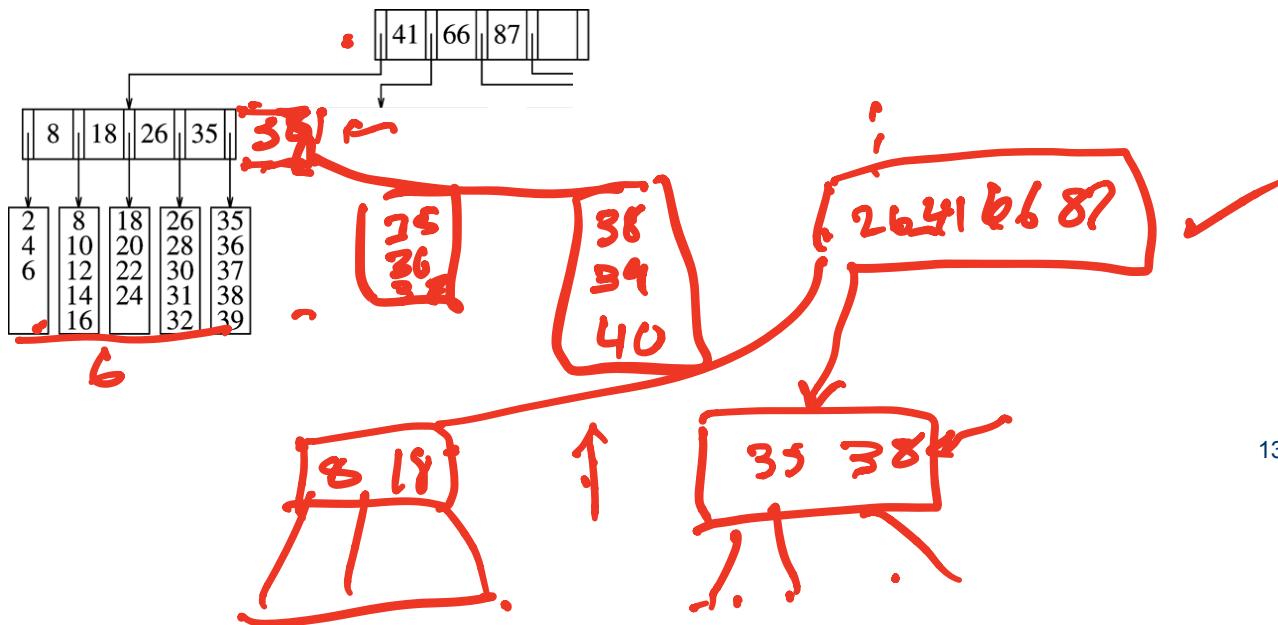
- Insert k
 - Navigate to leaf where key should be
 - **Case 2:** Leaf has L keys already (e.g. insert 40)
 - Need to split the leaf; split the leaf, promote middle key to parent node
 - May need to split parent if no room there...

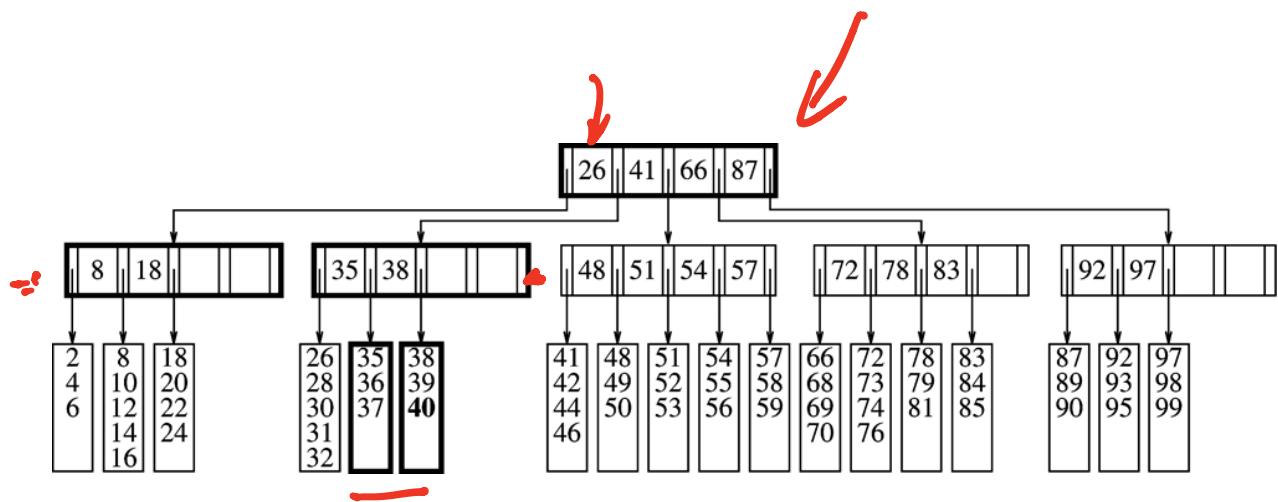




B⁺ Tree Insert: Splitting

- 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

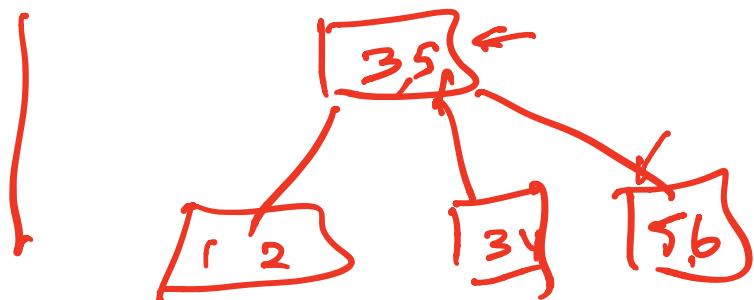
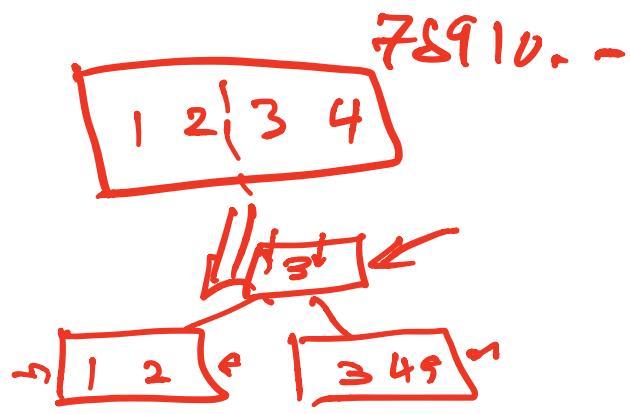




B^+ $M=3$, 2-3 children.

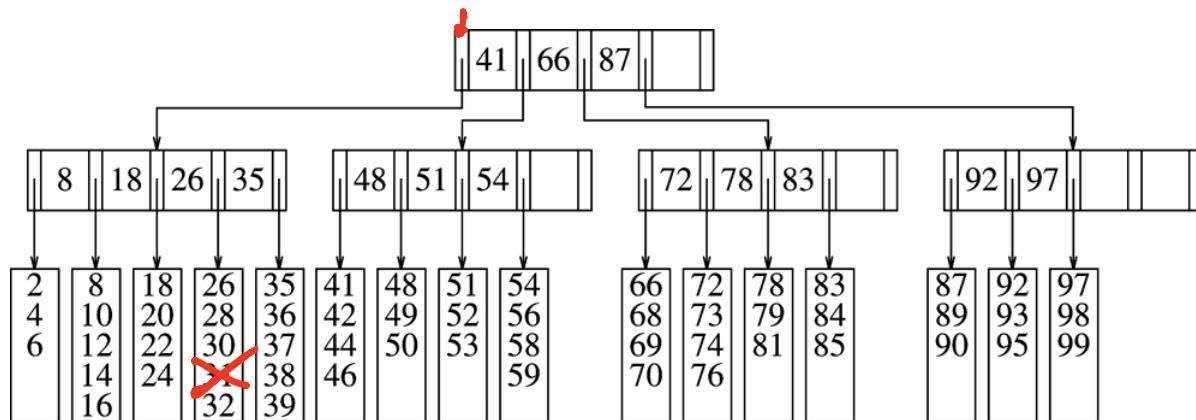
1, 2, 3, 4, 5, 6 $L=3$

B^+
insert.



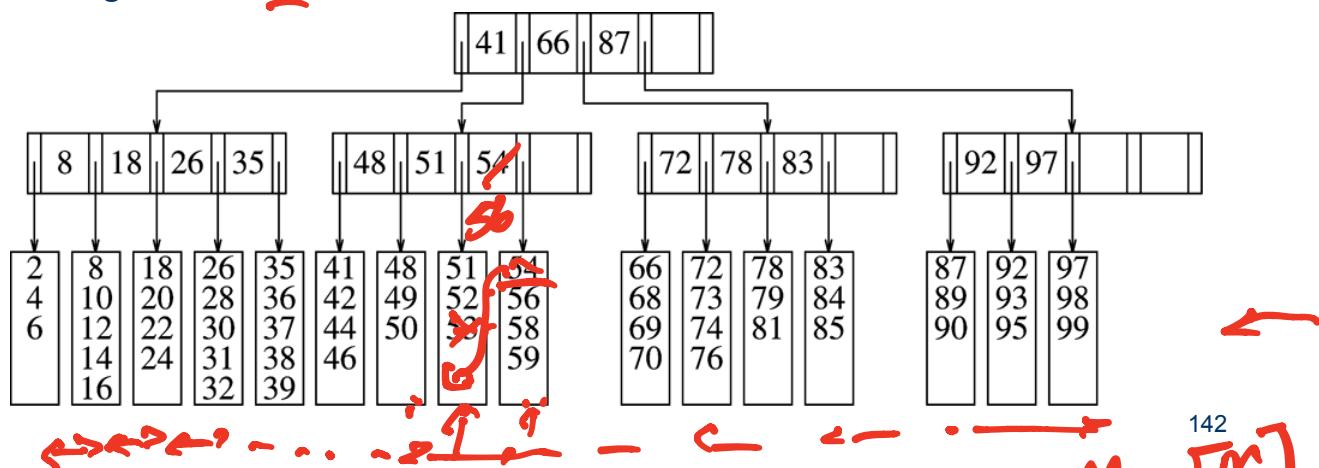
B⁺ Tree Delete

- 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

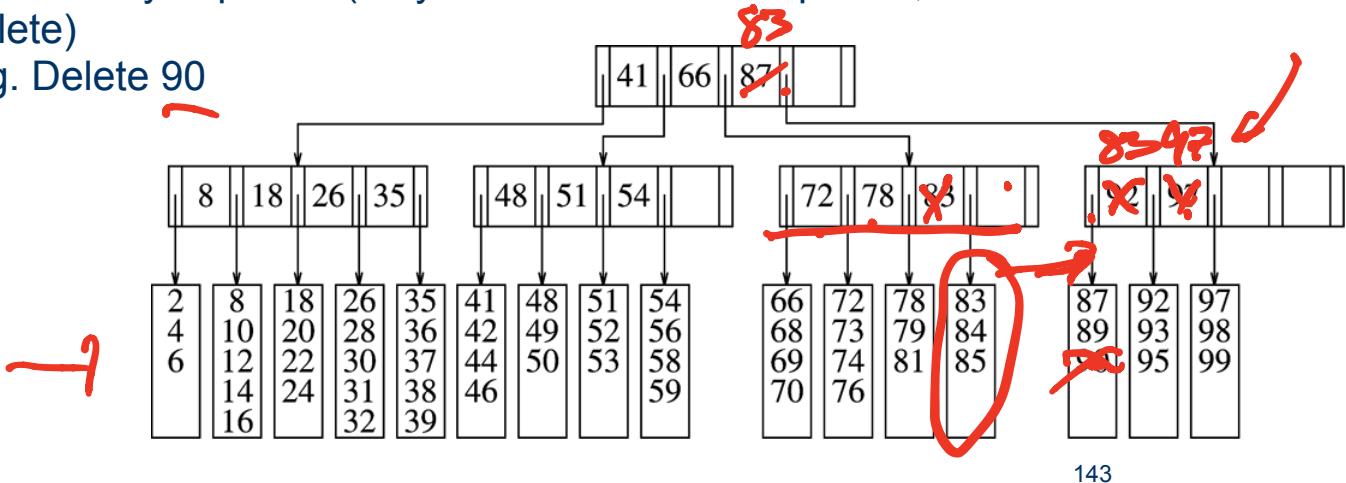


$h. \nearrow (\deg_m n)$

$m=5$
 $m=228 \rightarrow 114$

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



143

Fix

27

Merge

87
89
92
93
95

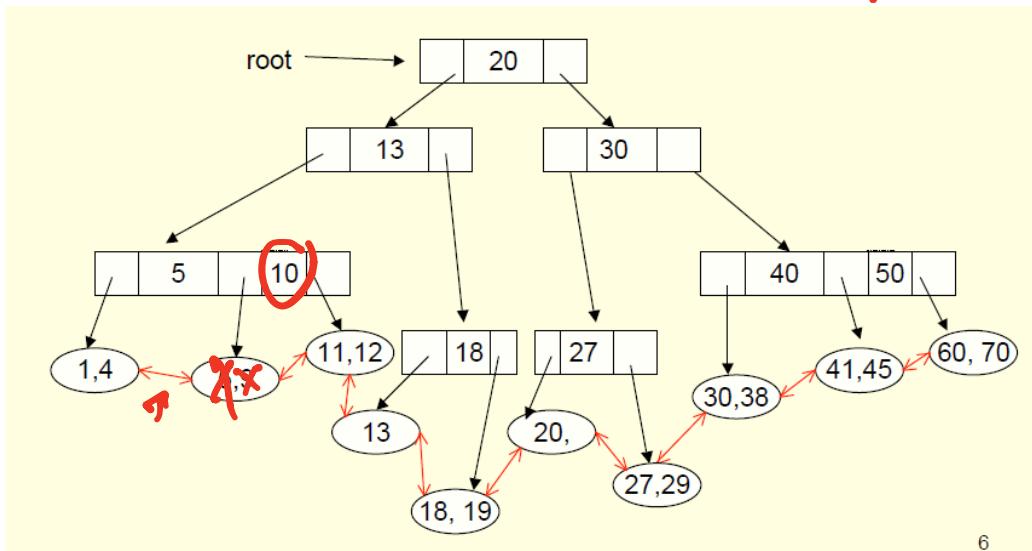
97
98
99

B+ Tree Delete

- Delete 5
 - No problem, leaf has extra key

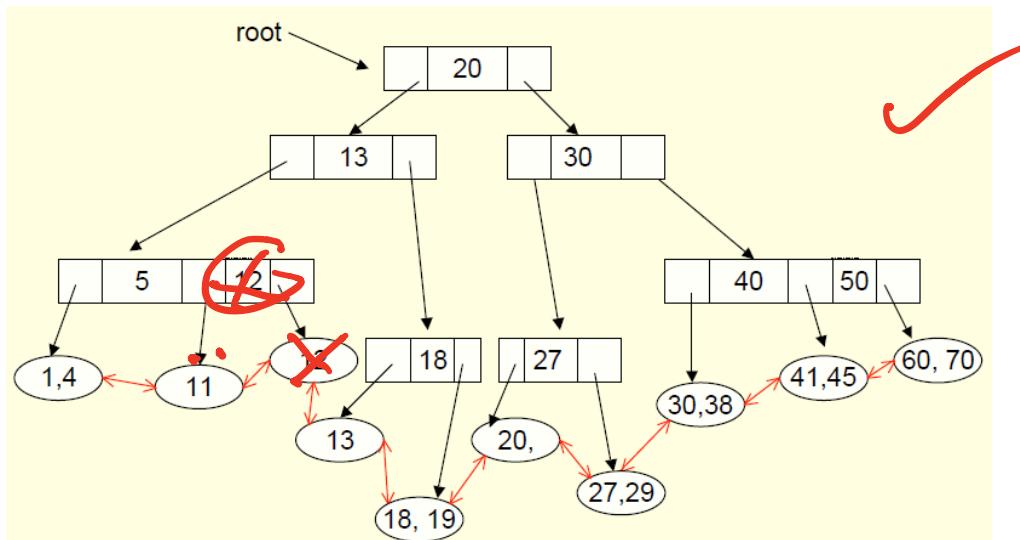
$m = 3$
 $m_{\min} = 2$.

$L=2$
 m
 $= 1$



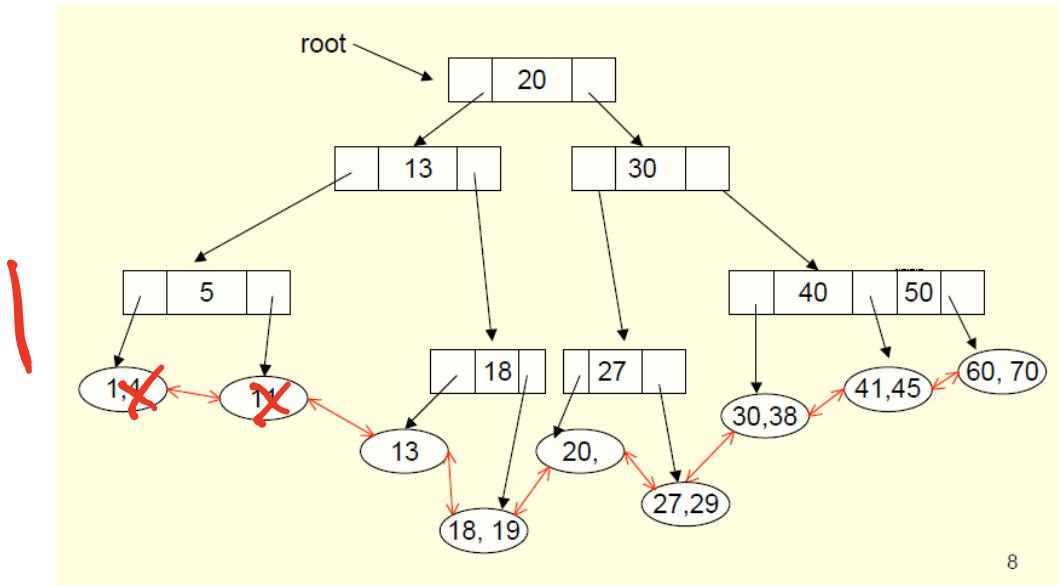
B+ Tree Delete

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



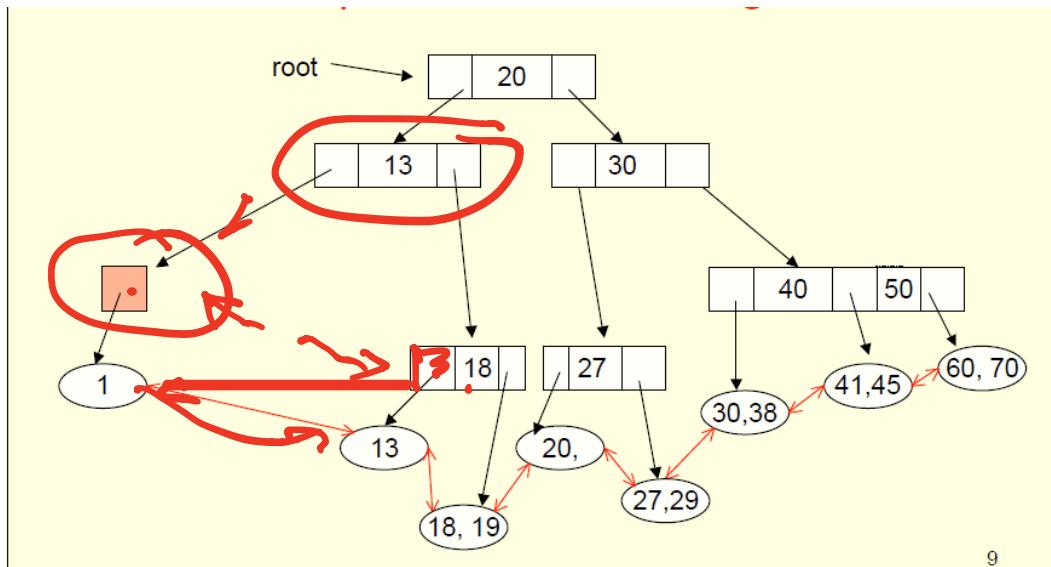
B⁺ Tree Delete

- Delete 12 ✓
 - Underflow, must delete node



B⁺ Tree Delete

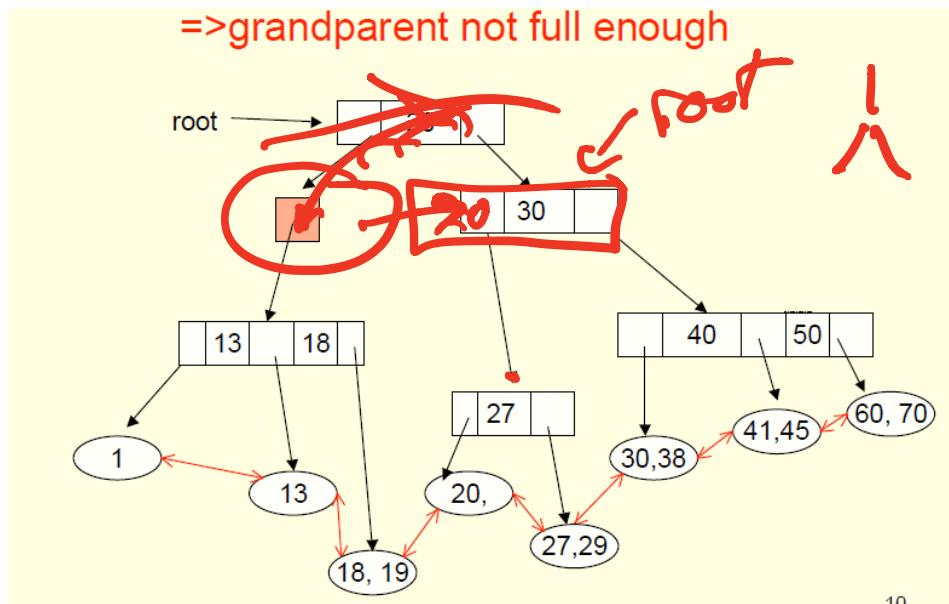
- Delete 4
 - No problem, extra key
- Delete 11
 - Big problem!
 - Underflow parent
 - Parent must merge



9

B+ Tree Delete

- Merge parent
 - Grandparent now has underflow
-

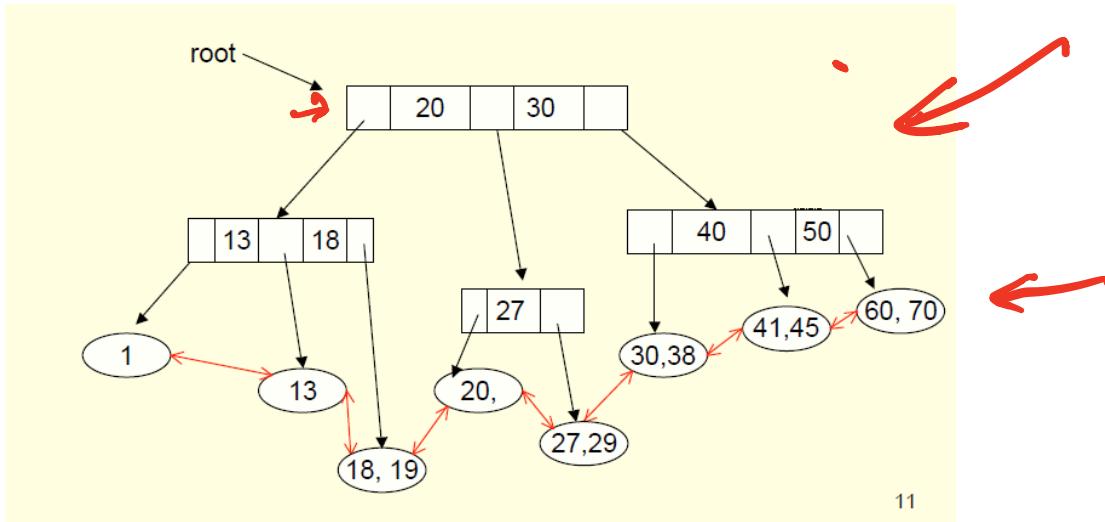


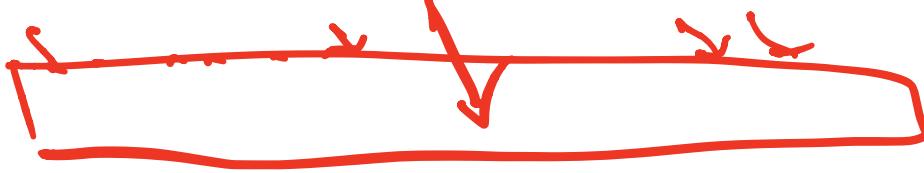
10

148

B⁺ Tree Delete

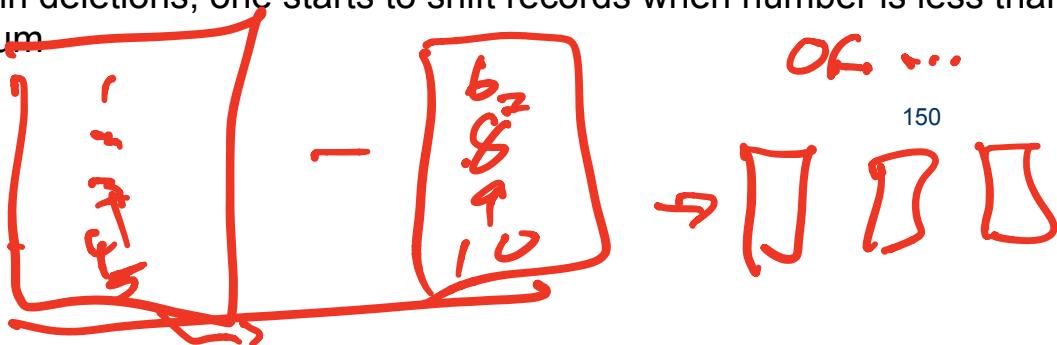
- Merge grandparent
 - Root has underflow, delete root
-





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

Stacks - LIFO
FIFO Q

- o 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)
- o Priority queues
 - Store tasks using a partial ordering based on priority
 - Ensure highest priority task at head of queue
- o Heaps are the underlying data structure of priority queues

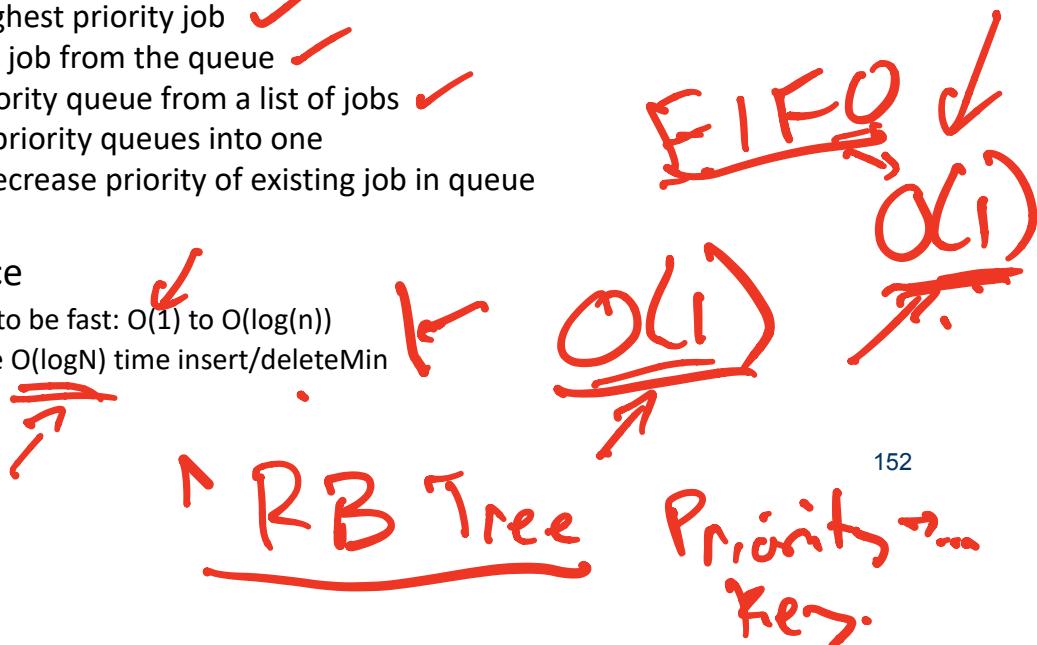
Priority Queue

- 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 $O(\log N)$ 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

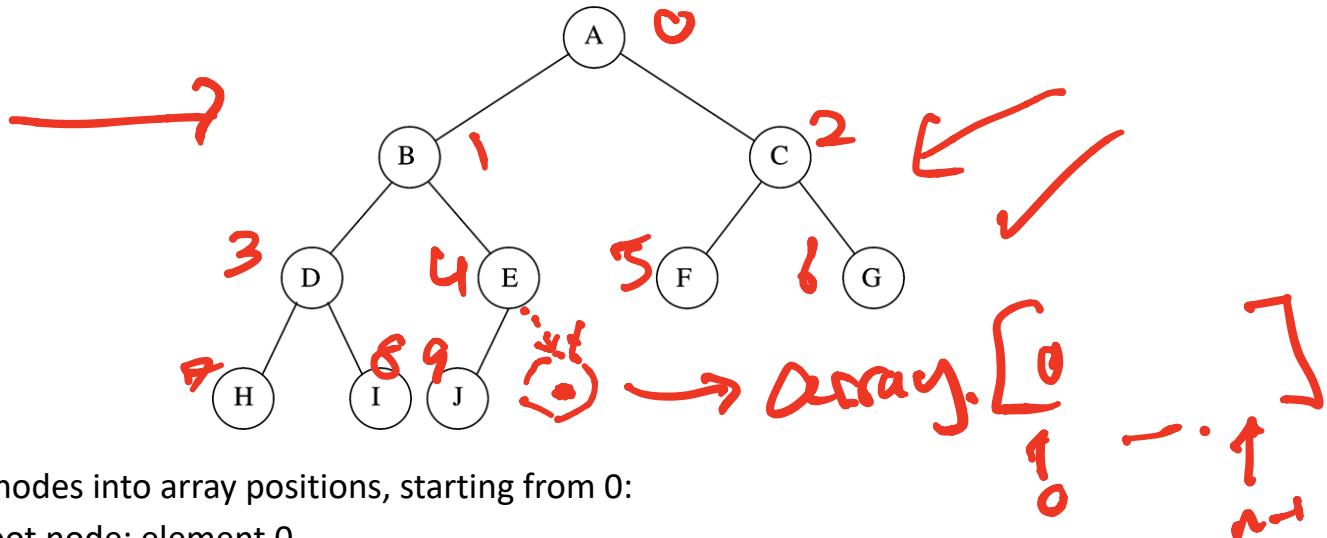
Binary Heap

min 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]

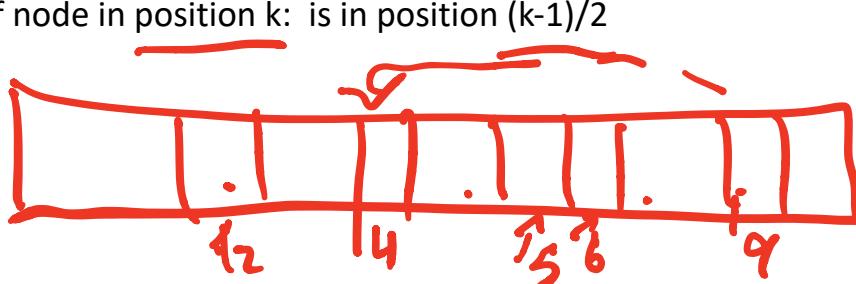


Binary Heap Example



- 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 $(k-1)/2$

155

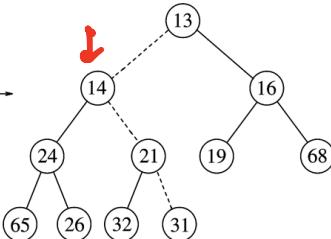
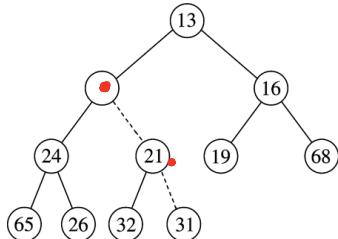
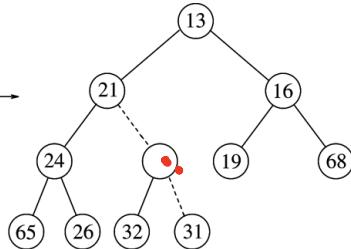
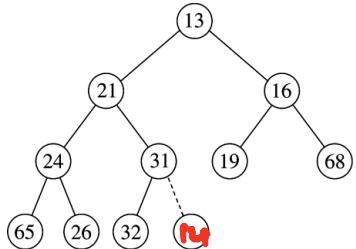


Heap Insert

- 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

$\log(n)$ →

Insert $O(\log(n))$.

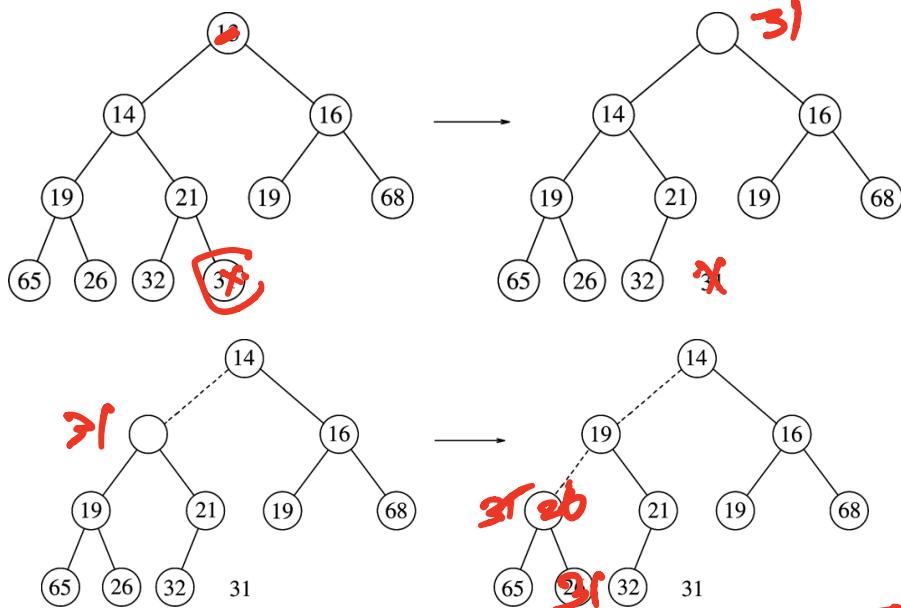
Upheap(k) ✓

```
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;`

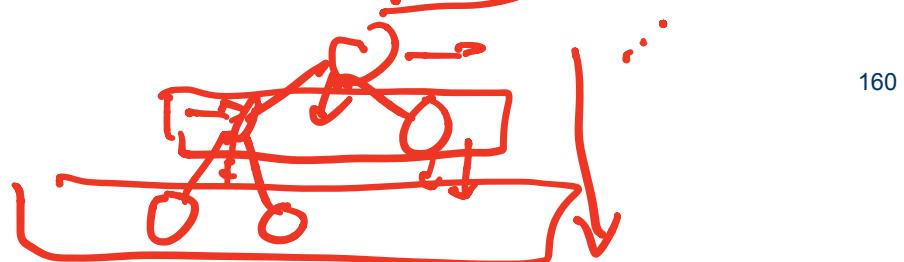
$\log(n) \cdot 2$

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.
 - $\text{Buildheap}(n)$: for ($i = n / 2 - 1$; $i >= 0$; $i--$) $\text{downheap}(i, n)$ ✓
 - $O(N)$ worst case!

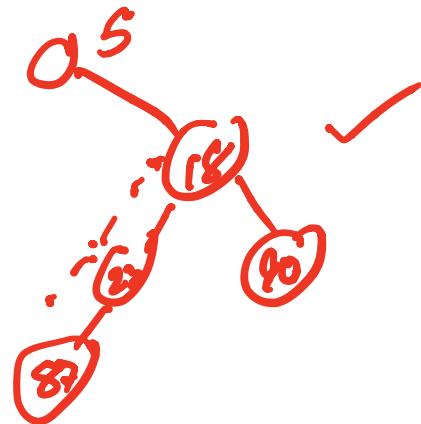
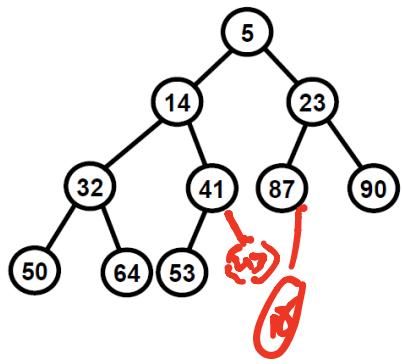
Complexity: $\frac{1}{2}$ elements don't move, $\frac{1}{4}$ move 1, $\frac{1}{8}$ move 2, ...

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



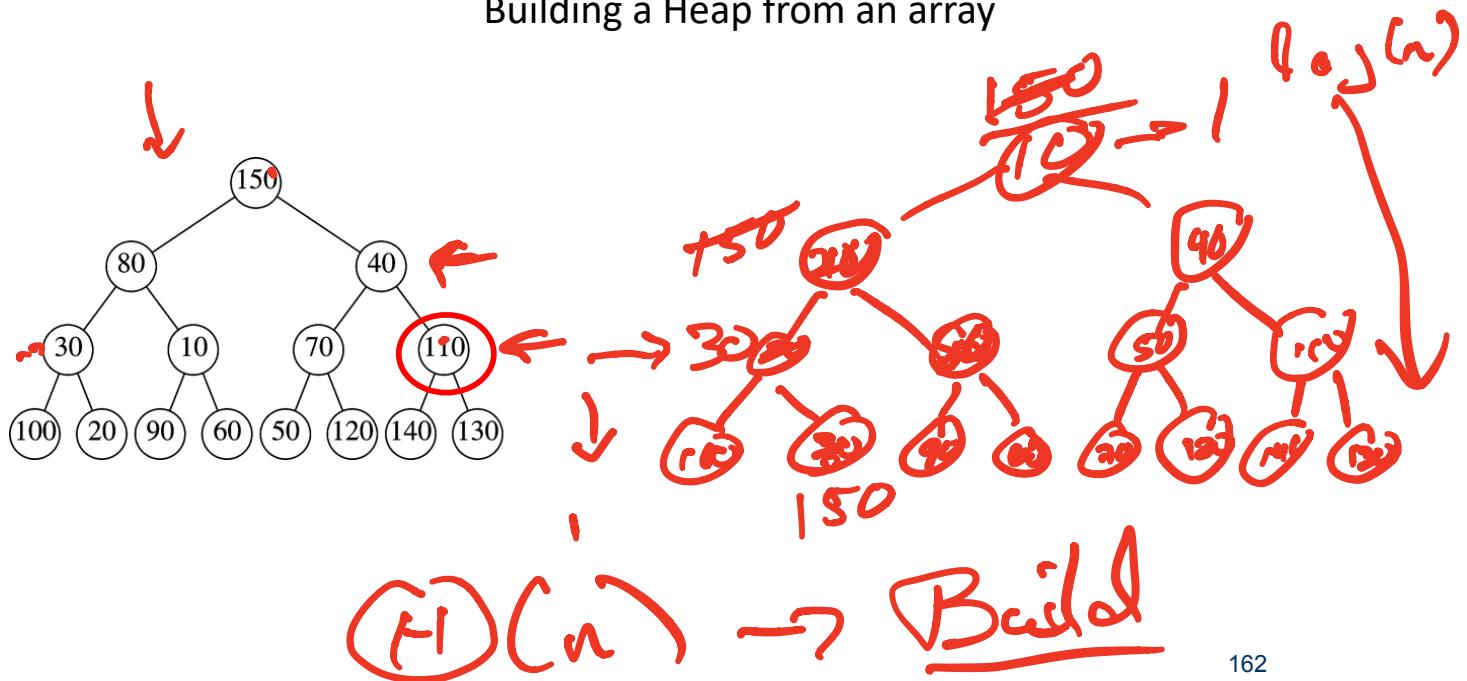
Examples

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: $O(n) \rightarrow$ link two arrays and buildHeap

Wishes: Merge of $O(1)$, decrease key of $O(1) \rightarrow$ useful in future algorithms...

$O(1)$. FIFO Q \rightarrow $O(1)$

Fast Sorting Algorithm: HeapSort

- Input: array of numbers (n) ✓
- Step 1: Build heap: $O(n)$
- Step 2: for k in 1 to n , $\text{deleteMin}()$: Complexity $O(n \log(n))$ ↗
- Heapsort complexity: $O(n \log(n))$: as small as the fastest comparison based sort algorithms.
Matches mergeSort, TimSort, quickSort
- Only drawbacks: Not stable, and average case is $O(n \log(n))$: not opportunistic
Stable ↗



Priority Queue Applications

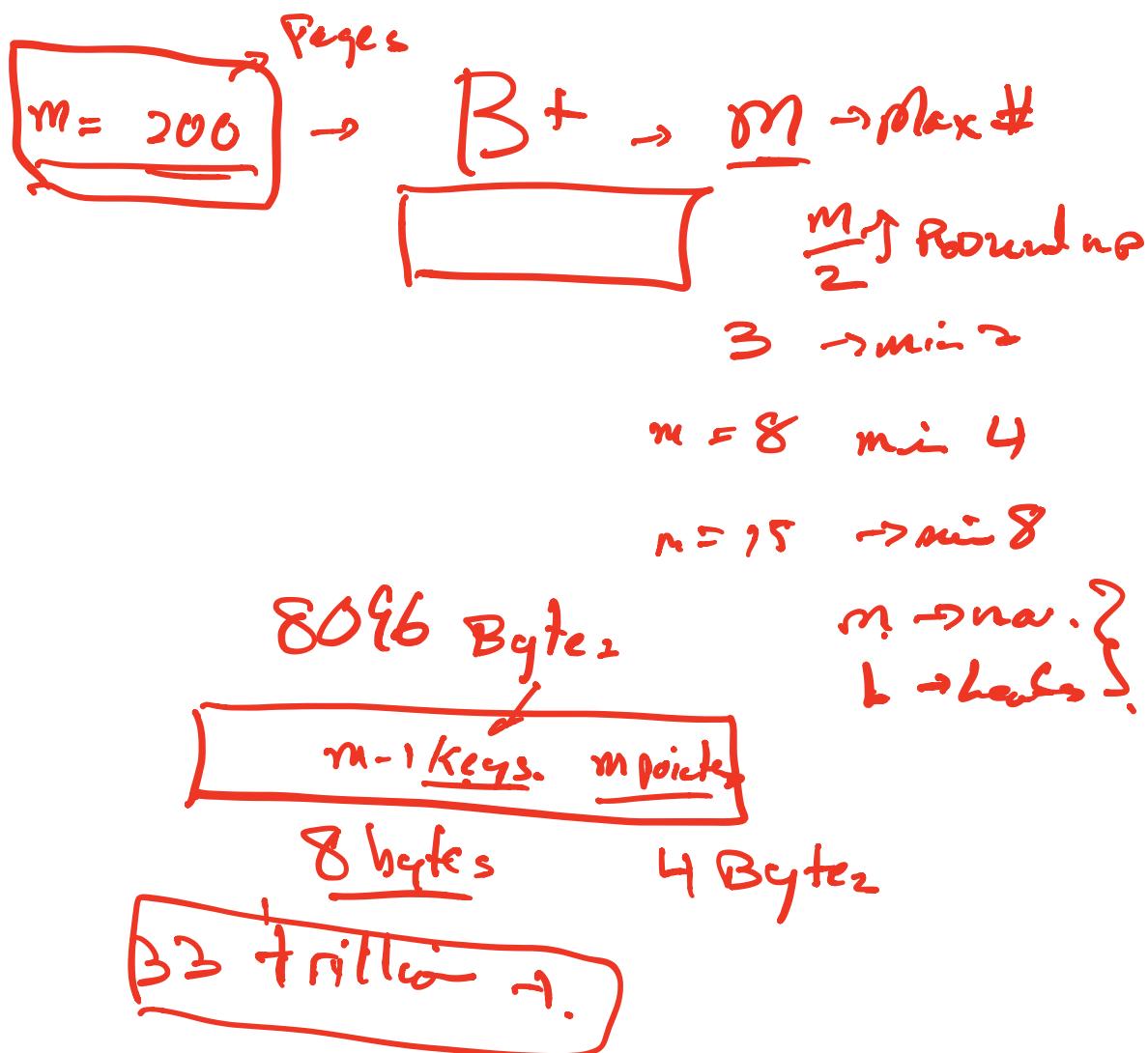
- Operating Systems: task scheduling ✓
- Graph algorithms: Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, A* search, Branch and Bound, ... ↗
- Discrete-event simulation: Have fast access to next event ↙
- Heapsort complexity: $O(n \log(n))$: as small as the fastest comparison based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

Binary heaps? ↗

B^+ trees? \rightarrow

11/20 Midterm tomorrow?

Hw 2 first



$$8(m-1) + 4m \approx 8096$$

$$12M \approx 8096 \cdot 8$$

$$M \approx \frac{8096}{12} \rightarrow \dots$$

