505 22240 / ESOE 2012 Data Structures: Lecture 12
2-3-4 Trees and Sorting

§ 2-3-4 Trees

・Perfectly balanced tree.

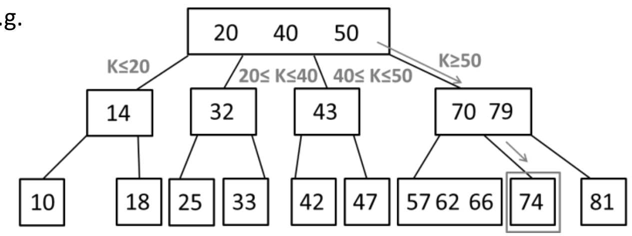
・find(), insert(), and remove() take worst-case O(log n) time.

・Each node has <u>2</u>, <u>3</u>, or <u>4</u> children, except leaves, which are all at bottom level.

・Each node stores 1, 2, or 3 entries.

・<u># of children</u> is <u># of</u> <u>entries + 1</u> or <u>zero</u>

・e.g.



★ "Bottom-up" 2-3-4 trees: the effects of node splits at the bottom of the tree can work their way back up toward the root.

★ We'll discuss "top-down" 2-3-4 trees (faster), in which insertion and deletion finish at the leaves.

◎Operations

① Entry find(const K& k);

・Start at root. Check for k at each node. If it's not present, move down to appropriate child. Continue until k is found, or not found at leaf.

・e.g. find (74)

・You can define an inorder traversal on 2-3-4 trees analogous to binary trees and visit keys in sorted order.

② void insert(cons K& k, const V& v);

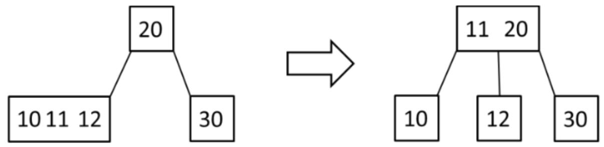
・Walks down tree in search of k, like find( ).

・If it finds k, it proceeds to k's "left child" and continues.

・Whenever insert( ) encounters a <u>3-key</u> node, middle key is placed in the parent node (parent has at most 2 keys; has room for third).

・The other two keys in the 3-key node are split into two separate 1-key nodes.

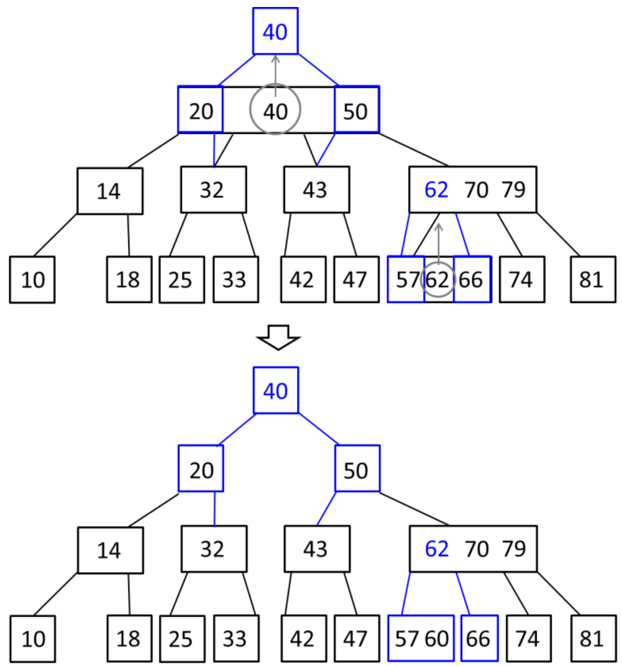
・e.g. (portion trees)



・e.g. insert(60)

Kick middle key (40) upstairs.

Create new root to hold 40.



★Why we split 3-key nodes: (and move its middle key up one level)

・To make sure there's room for new key in leaf.

・To make room for any key that's kicked upstairs.

★Sometimes insertion increases depth of tree by creating a new root.

③ void remove (const K& k); similar to remove( ) on binary trees.

・Find key k.

If it's in leaf, remove it.

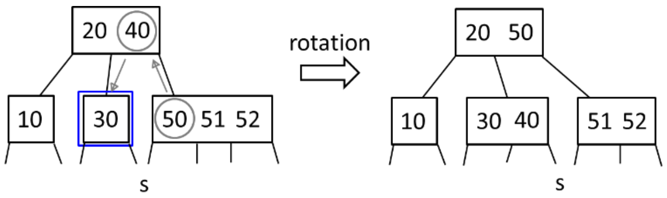
If in internal node, replace it with entry with next <u>higher</u> key. That entry is always in a leaf. In either case, you remove an entry from a leaf in the end.

⇨Eliminates 1-key nodes (except the root) so key can be removed from leaf without emptying it.

<u>Rule1</u>: remove( ) encounters 1-key node (except root)

⇨ Tries to steal key from an adjacent sibling.

・e.g. remove (30)

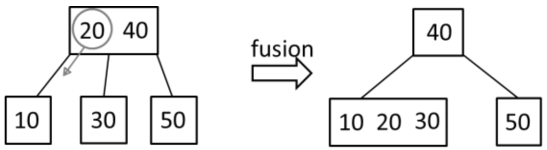


<u>Rule2</u>:

・If no adjacent sibling has > 1 key, steal a key from parent.

⇨ Parent (unless it's root) has ≧ 2 keys.

・e.g. remove(10)



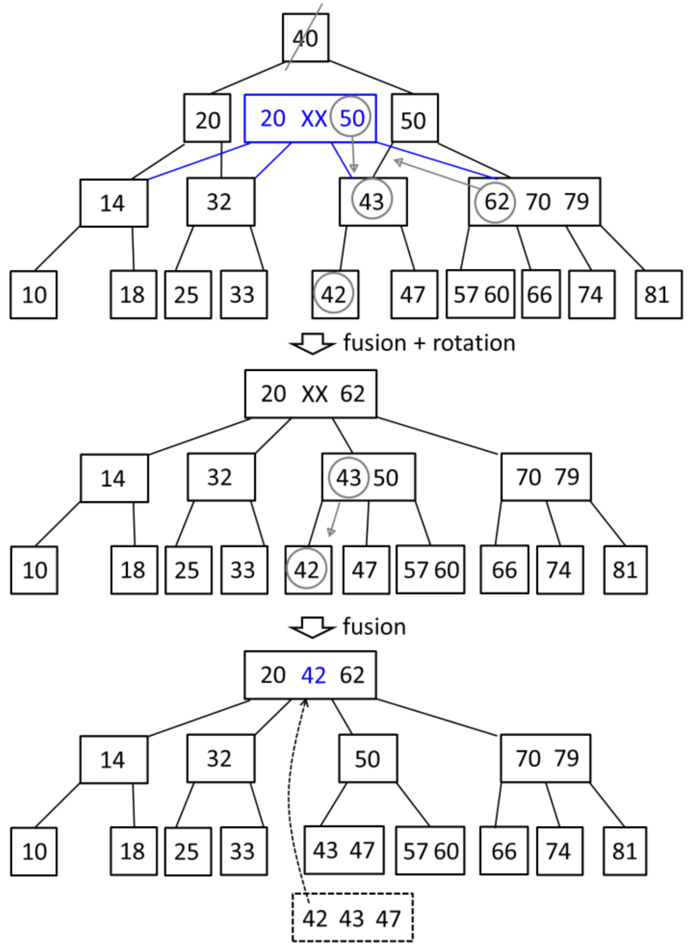
・The sibling is also absorbed, 1-key node becomes 3-key node.

<u>Rule3</u>:

・If parent is root & contains only one key, and sibling has only one key.

⇨ Fuse into 3-key node —> the new root. Depth of tree <u>decreases</u> by one.

・e.g. remove(40)



◎Running times

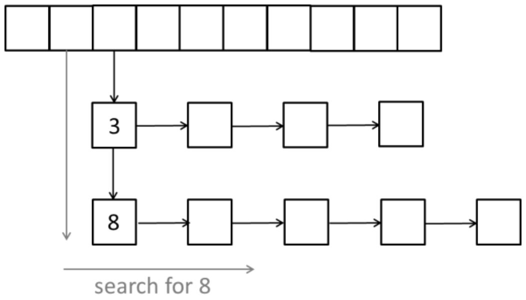
- A 2-3-4 tree with depth d has between 2^d and 4^d leaves.
- Total # of nodes is $n \geq 2^{d+1} - 1$

  $d \in O(\log n)$

- Time spent visiting node $\in$ O(1), per node.
- find( ), insert( ), remove( ): worst-case times: O(d) = O(log n).
- Compare with binary search tree: $\underline{\Theta(n)}$ worst-case time.

◎Another approach to duplicate keys

- Collect all entries that share a common key in one node.
- Each node's entry is list of entries.



- Simplifies implementation of findAll(), which finds all entries with a specified key.

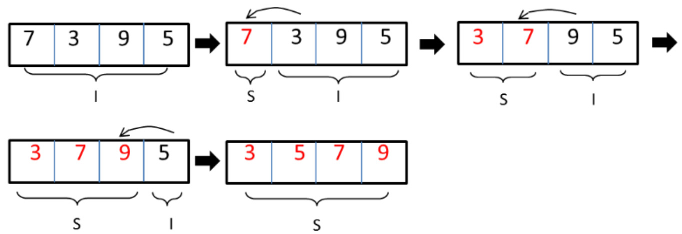
§ Sorting

◎Insertion Sort

- Runs in O(n^2)time, employ a list S.
- Invariant: S is sorted.
- Algorithm:

```
Start with empty list S & unsorted list I of n items
for(each item x in I) {
    Insert x into S, in sorted order
}
```

- If S is linked list, Θ(n) worst-case time to find right position.
- If S is array, Θ(n) worst-case time to shift higher items over.
- If S is array, insertion sort is in-place.
- *<u>In-place sort</u> is a sorting algorithm that keeps the sorted items in the same array.*
- If S is a <u>balanced search tree</u> (e.g., 2-3-4 tree), running time ∈ O(n log n).
- e.g.



◎Selection Sort

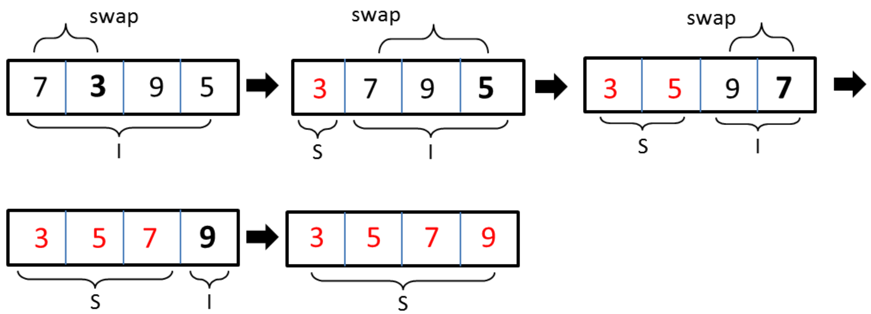
- Invariant : S is sorted.
- Algorithm:

```
Start with empty list S & unsorted list I of n items
for(i=0; i<n; i++) {
    x = item in I with smallest key.
    Remove x from I.
    Append x to end of S.
}
```

• Whether S is array or linked list, $\Theta(n^2)$ time even in <u>best</u> case.

• In-place selection sort.

• e.g.



◎Heapsort

• Selection sort where I is a heap.

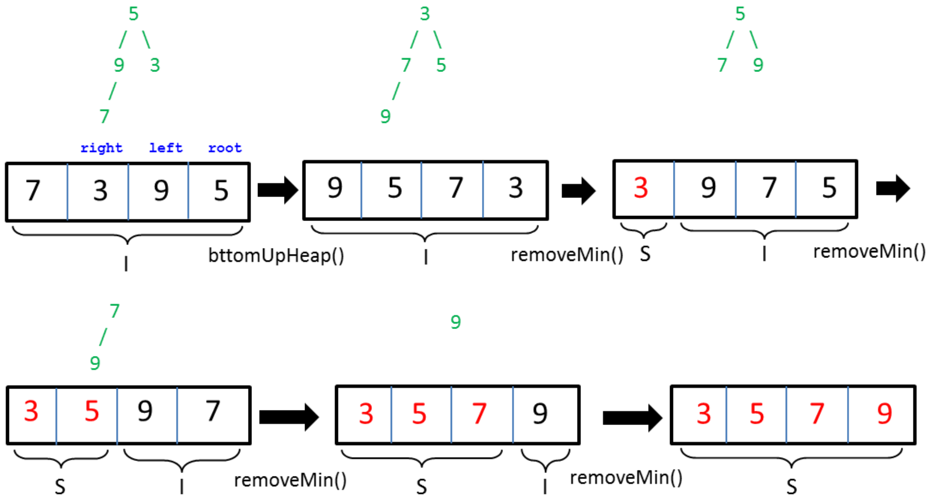
• Algorithm:

```
Start with empty list S and unsorted list I of n items
Toss all items in I onto heap h (ignoring heap-order)
h.bottomUpHeap(); ← O(n) time
// Enforces the heap-order property
for (i=0; i<n; i++) {
    x = h.removeMin();     ← O(log n) time
    Append x to the end of S.
}
```

• Heapsort runs in O(n log n) time.

• In-place: maintain heap <u>backward</u> at the end of the array (in reverse order).

• e.g.



• Excellent for arrays, clumsy for linked lists.

◎Mergesort

• Mergesort is based on the observation that it's possible to merge 2 sorted lists into one sorted list in linear time.

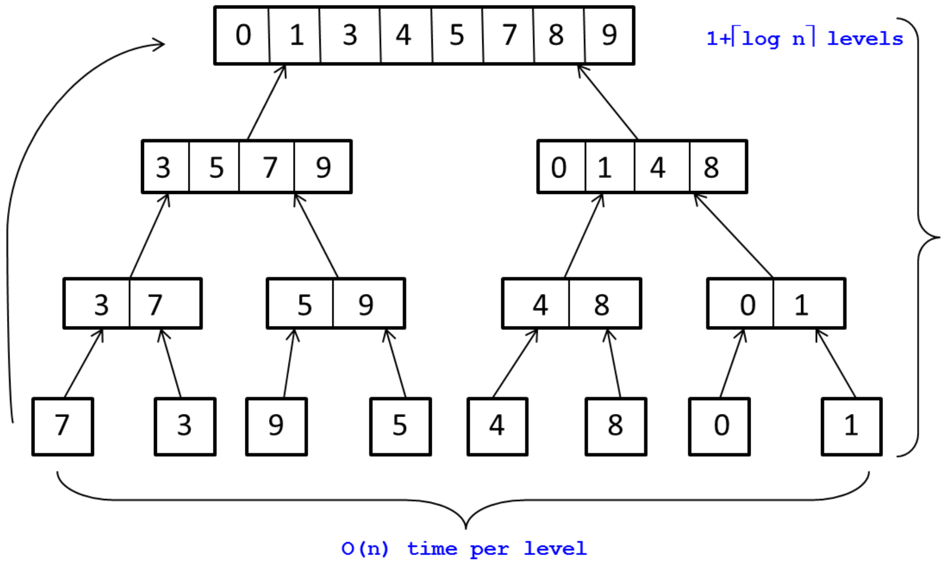
• In fact, we can do it with queues:

```
Let Q1 & Q2 be 2 sorted queues.
Let Q be empty queue.
while (neither Q1 nor Q2 is empty) {
   item1 = Q1.front();
   item2 = Q2.front();
   Move smaller of item1 & item2 from present queue to end of Q.
}
Concatenate remaining non-empty queue (Q1 or Q2) to end of Q.
```

· Mergesort is a recursive divide-and-conquer algorithm:

```
Start with unsorted list I of n items.
Break I into halves I1 & I2, having ⌈n/2⌉ & ⌊n/2⌋ items.
Sort I1 recursively, yielding S1.
Sort I2 recursively, yielding S2.
Merge S1 & S2 into one sorted list S.
```

· e.g. the diagram (tree) shown below is not a data structure, but a sequence of recursive calls.



· O(n log n) time.

· Natural for linked lists.

· Not in-place sort for arrays.