CS1332 Fall 2018 - Week 09

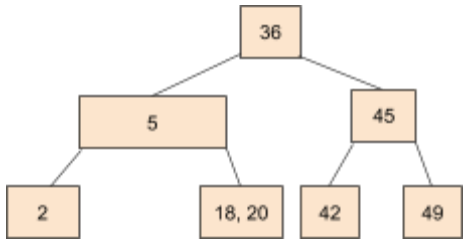
10/15/18

Topics

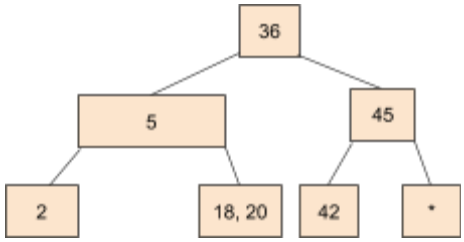
I. 2-4 removals continued

A. remove(49) - push the empty node up to the root so we can remove it

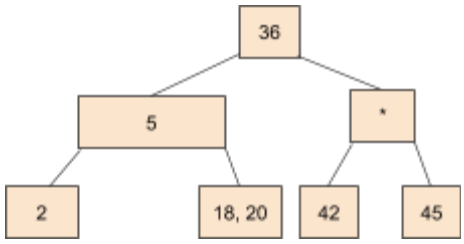
Initial tree →



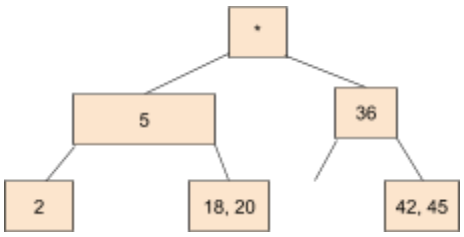
(1) Remove data →



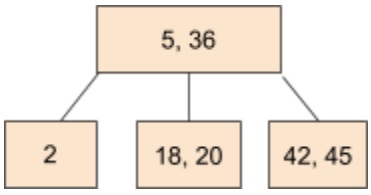
(2) Take from parent →



(3) Fuse & take from parent →

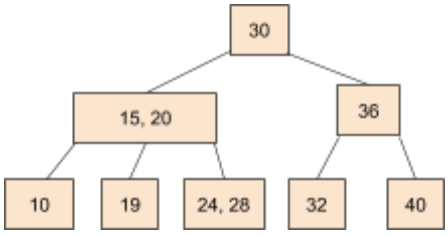


(4) Fuse →

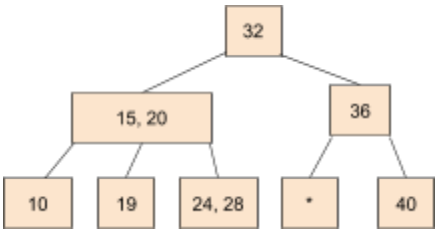


B. remove(30) - a very complex example that can be resolved in many ways

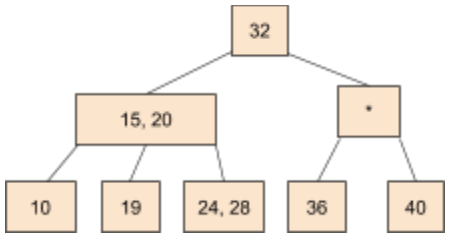
Initial tree →



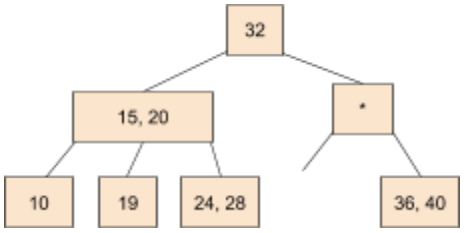
(1) Remove data, replace with successor →



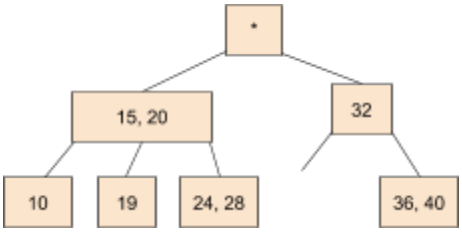
(2) Take from parent →



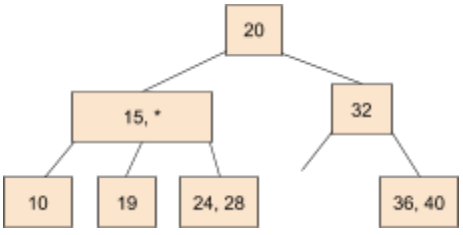
(3) Fuse →



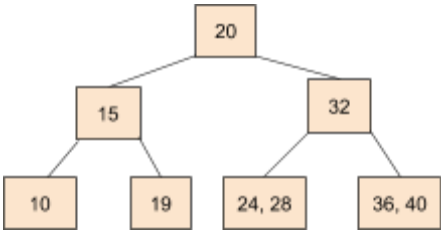
(4) Take from parent →



(5) Take from child →



(6) Move child →



1. We could've solved this by fusing 15, 20, and 32 at step 4, but the way we handled it above handles cases where fusion may cause overflow.
2. There are many, many individual cases you can check for and write specific implementations to handle; these are just a few.
3. The 2-4 tree implementation is extremely complicated.

Activities

I. Big-O tree reality check

Exam 2 - 10/17/18
10/19/18

Topics

I. Sorting Intro
   A. Two categories of sorts:
      1. Iterative - bubble, insertion, selection, cocktail shaker; worst case $O(n^2)$
      2. Divide & conquer - merge, least significant digit (LSD) radix, in-place quicksort (+ quickselect)
   B. Qualities of sorts:
      1. Stability - if duplicates always retain their relative order throughout the sort, the sort is stable; if not, the sort is unstable
         a) Stability is useful if we want to apply several sorts, eg. sort by first name and then by last name such that people with the same last name are sorted by first name
      2. Adaptable - if the algorithm can stop early when the data is sorted, the sort is adaptable; if not, the sort is not adaptable
      3. In-place - if the sort can be done within the same array that was passed in, the sort is in-place; if the sort requires the data to be moved into other structures (eg. more arrays, etc.), the sort is out-of-place
      4. Time complexity
   C. Swap Method

```
swap(arr, index1, index2)
      int temp = arr[index1]
      arr[index1] = arr[index2]
      arr[index2] = temp
```

II. Iterative Sorts - all done in-place
   A. Bubble Sort
      1. Premise - *bubbles* largest element to the end of the array
      2. Algorithm:
         a) k: loop from end to 1, decrement
            (1) i: loop from 0 to end - 1
               (a) compare arr[i] to arr[i +1] → swap if not in order
      3. Optimization 1: if you do no swaps for one iteration, the array is sorted
      4. Optimization 2: keep track of the last place you swapped, this can now be the new "end" where you can stop comparing on the next iteration

5. Example (1 iteration) - compared & swapped, compared & not swapped

idx	0	1	2	3	4	5	6	7	8	9	
	6A	4	2	5	3	7	6B	8	9	10	k=9
	4	6A	2	5	3	7	6B	8	9	10	
	4	2	6A	5	3	7	6B	8	9	10	
	4	2	5	6A	3	7	6B	8	9	10	
	4	2	5	3	6A	7	6B	8	9	10	
	4	2	5	3	6A	7	6B	8	9	10	
	4	2	5	3	6A	6B	7	8	9	10	
	4	2	5	3	6A	6B	7	8	9	10	
	4	2	5	3	6A	6B	7	8	9	10	
	4	2	5	3	6A	6B	7	8	9	10	
	2	4	5	3	6A	6B	7	8	9	10	k=6
	...										

   a) On the next iteration, we would only have to increment i to 5 because we've set k (end) to the location of the last swap (6).

6. Qualities: stable, adaptable (with optimizations), in-place

B. Insertion Sort

1. Premise - *inserts* data into the presorted elements before it
2. Algorithm:
   a) k: loop from 1 to end
      (1) set n = k (index of the element we are inserting)
      (2) loop from n to 1, or until a swap is not needed
         (a) compare arr[n] to arr[n-1] → swap if not in order
3. Example (full) - compared & swapped, compared & not swapped, **sorted**

idx	0	1	2	3	4	5	6	7	8	9	
	6A	4	2	5	3	7	6B	8	9	10	
	4	**6A**	2	5	3	7	6B	8	9	10	n=1
	4	2	**6A**	5	3	7	6B	8	9	10	n=2
	2	**4**	**6A**	5	3	7	6B	8	9	10	
	2	**4**	5	**6A**	3	7	6B	8	9	10	n=3
	2	**4**	**5**	**6A**	3	7	6B	8	9	10	
	2	**4**	**5**	3	**6A**	7	6B	8	9	10	n=4
	2	**4**	3	**5**	**6A**	7	6B	8	9	10	
	2	3	**4**	**5**	**6A**	7	6B	8	9	10	
	2	**3**	**4**	**5**	**6A**	7	6B	8	9	10	
	2	**3**	**4**	**5**	**6A**	**7**	6B	8	9	10	n=5
	2	**3**	**4**	**5**	**6A**	6B	**7**	8	9	10	n=6
	2	**3**	**4**	**5**	**6A**	**6B**	**7**	8	9	10	
	2	**3**	**4**	**5**	**6A**	**6B**	**7**	**8**	9	10	n=7
	2	**3**	**4**	**5**	**6A**	**6B**	**7**	**8**	**9**	10	n=8
	2	**3**	**4**	**5**	**6A**	**6B**	**7**	**8**	**9**	**10**	n=9

4. Qualities: stable, adaptable (by design), in-place