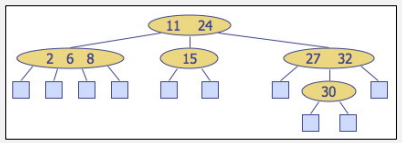
## Multiway Search Trees

A *multiway search tree* allows more than one value per node.

- generalization of binary tree
  - *order m* allows each node to have up to *m*-1 values and *m* children and *m*-1 values has up to m-1 values, in sorted order
- a node with *k* values has *k*+1 children (which may be empty)
- *i*th subtree of a node $[v_1, \ldots, v_k]$ only contains values in the range $v_i \le v < v_{i+1}$
  - $0 \le i \le k$
  - $v_0 = -\infty$, $v_{k+1} = \infty$

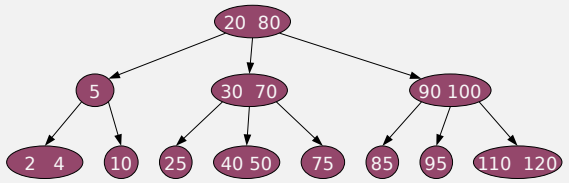





## 2-4 Trees

A *2-4 tree* is a multiway search tree where

- all leaves are at the same depth
- each node has 1, 2, or 3 keys and (# keys)+1 children






## Height of 2-4 Trees

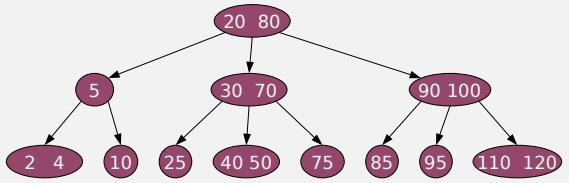
Does this ensure logarithmic height?
→ Yes!

Observe.

- the 2-4 tree with the fewest keys for its height has 1 key per node (complete binary tree)
  - level i has $2^i$ keys and the whole tree has $n = 2^h$-1 keys
    → $h = O(\log n)$
- the 2-4 tree with the most keys for its height has 3 keys per node
  - level i has $3 \times 4^i$ keys and the whole tree has $n = 4^h$-1 keys
    → $h = O(\log n)$



## Operations on 2-4 Trees




Searching in a multiway tree is similar to searching in a binary tree –
if the target element is not one of the keys in the current node, continue the search with the appropriate child.

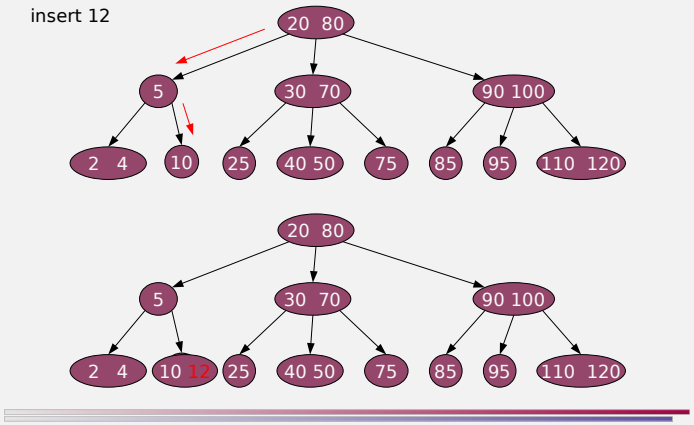
## Operations on 2-4 Trees

For insert and remove, we use the same approach as with AVL trees:

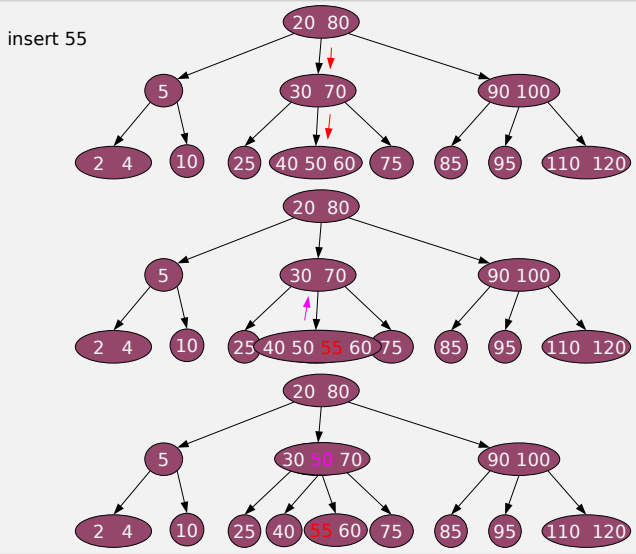
- insert/remove as dictated by the structural and ordering rules
  - new elements are always inserted at a leaf
  - elements can only be removed from a leaf – first swap with next larger (or smaller) as needed
- fix up the broken node size property as needed
  - if insertion creates an overflow (too many keys) –
    - split the node and promote a middle item to the proper place in the parent
    - repeat until there are no more overflows, creating a new root if necessary
  - if removal creates an underflow (not enough keys) –
    - if there's a sibling with at least two keys, transfer one (via the parent)
    - otherwise, merge – move a key from the parent, merging the node with a sibling
    - repeat until there are no more underflows, removing the root if necessary

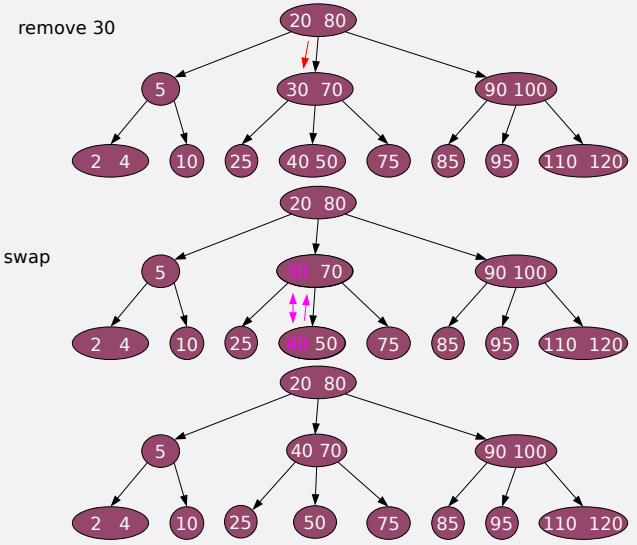


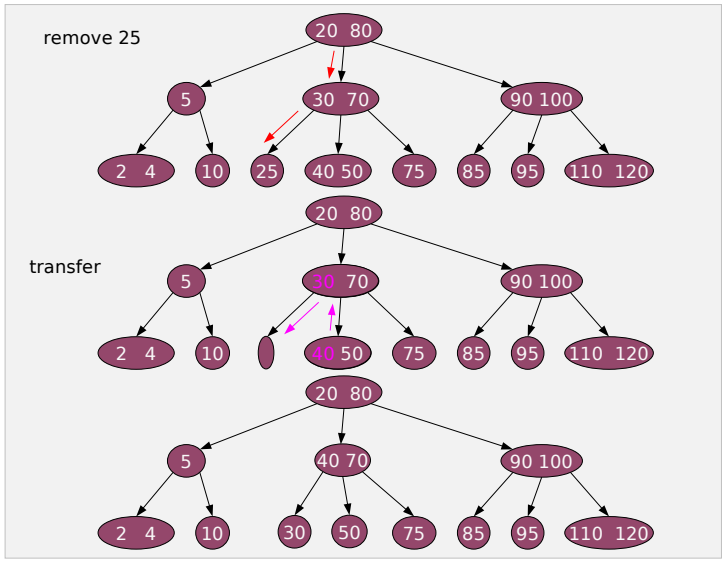
## Insert

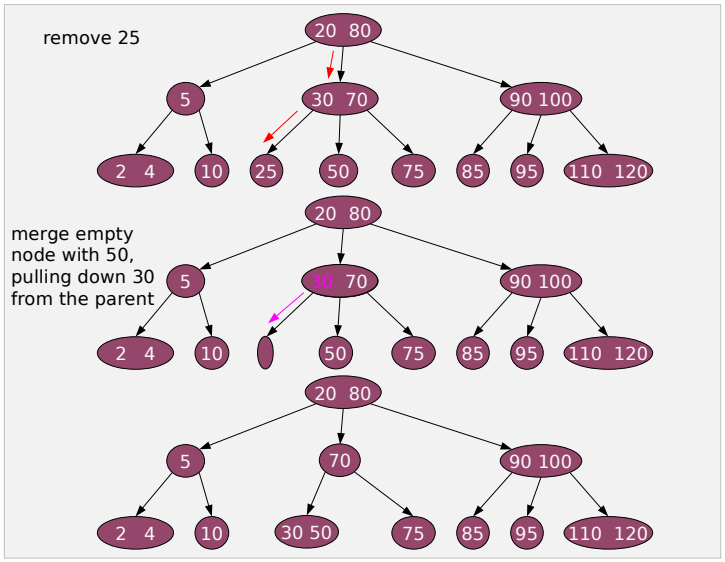


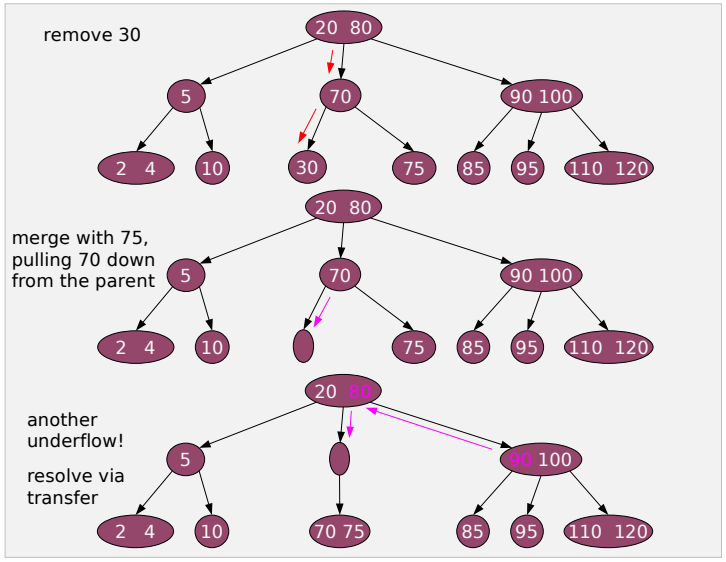


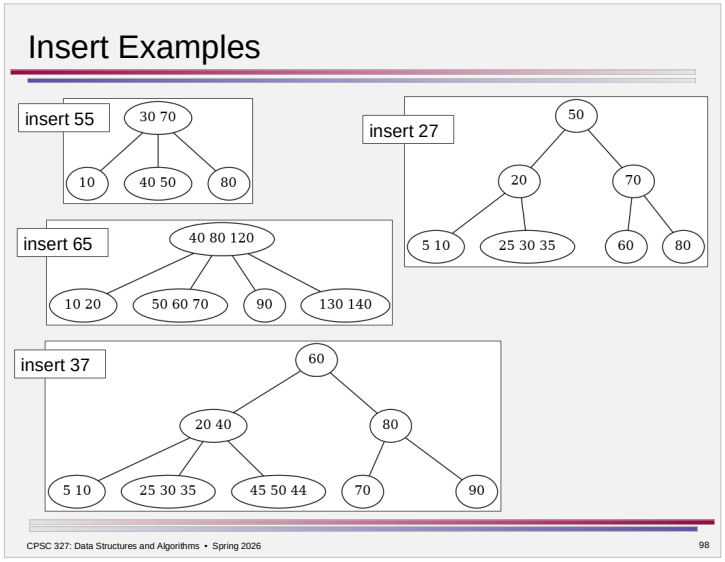



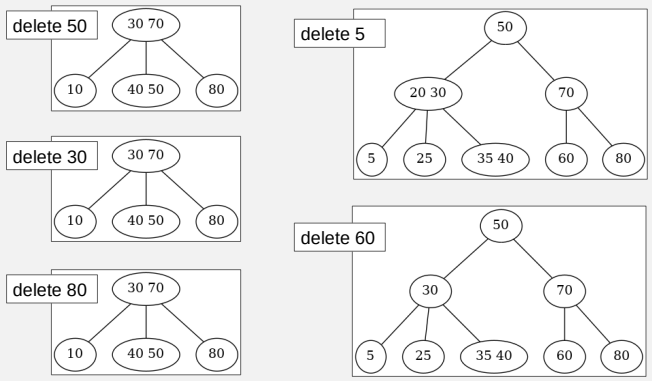

remove 25
20 80
5
30 70
90 100
2 4
10
25
40 50
75
85
95
110 120
transfer
20 80
5
30 70
90 100
2 4
10
40 50
75
85
95
110 120
20 80
5
40 70
90 100
2 4
10
30
50
75
85
95
110 120


remove 25
20 80
5
30 70
90 100
2 4
10
25
50
75
85
95
110 120
merge empty
node with 50,
pulling down 30
from the parent
20 80
5
30 70
90 100
2 4
10
50
75
85
95
110 120
20 80
5
70
90 100
2 4
10
30 50
75
85
95
110 120


remove 30
20 80
5
70
90 100
2 4
10
30
75
85
95
110 120
merge with 75,
pulling 70 down
from the parent
20 80
5
70
90 100
2 4
10
75
85
95
110 120
another
underflow!
resolve via
transfer
20 80
5
90 100
2 4
10
70 75
85
95
110 120


Insert Examples
insert 55
30 70
10
40 50
80
insert 27
50
20
70
5 10
25 30 35
60
80
insert 65
40 80 120
10 20
50 60 70
90
130 140
insert 37
60
20 40
80
5 10
25 30 35
45 50 44
70
90
CPSC 327: Data Structures and Algorithms • Spring 2026
98

# Delete Examples






# 2-4 Trees Running Time

- time for initial insert – O(log n)
- time to fix up one overflow – O(1)
- number of overflows to fix – O(log n)

  → total time for insert – O(log n)

- time for initial remove – O(log n)
- time to fix up one underflow – O(1)
- number of underflows to fix – O(log n)

  → total time for remove – O(log n)