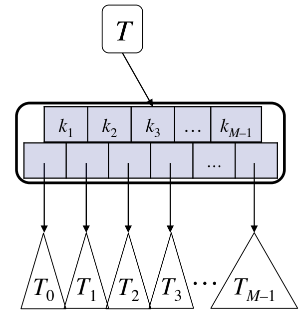
eecs 281 DATA STRUCTURES AND ALGORITHMS

Lecture 10: Multi-way Search Trees:

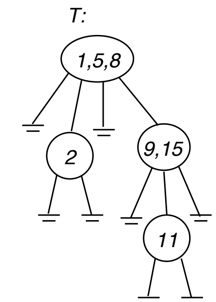
- intro to B-trees
- 2-3 trees
- 2-3-4 trees

# Multi-way Search Trees




A node on an $M$-way search tree with $M$–1 distinct and ordered keys: $k_1 < k_2 < k_3 < \ldots < k_{M-1}$, has $M$ children $\{T_0, T_1, T_2, \ldots, T_{M-1}\}$

Every element in child $T_i$ has a value larger than $k_i$ and smaller than $k_{i+1}$

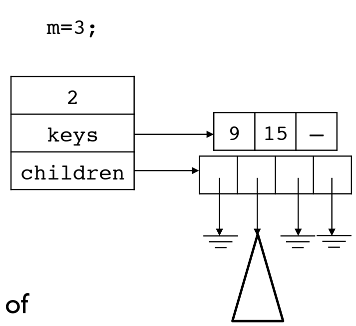



Number of valid keys doesn't have to be the same for every node on the tree

# $M$-Way Search Trees Representation

A node of an $M$-way search tree can be represented as:

```
int m;
struct mnode {
  int in_use;
  Key keys[m-1];
  mnode *children[m];
}
```




`in_use`: how many of the $m$ keys of this node are currently in use

# $M$-Way Search

Search on an $M$-way search tree is similar to that on a BST, with more than 1 compares per node (a BST is just an $M$-way search tree with $M = 2$)

If all nodes have $M$ – 1 keys, with linear search on the nodes, it takes $O(M \log_M N)$ time to search an $M$-way search tree of $N$ internal nodes ($(M – 1) N$ keys)

With binary search on the nodes, it takes $O(\log_2 M \log_M N)$ time

## Advantage 1: External Search

$M$-way search tree can be used as an index for external (disk) searching of large files/databases

Characteristics of disk access:

- orders of magnitude slower than memory access
- for efficiency, data usually transferred in blocks of 512 bytes to 8KB

To speed up external search, put as much data as possible on each disk block, for example, by making each node on an $M$-way search tree the size of a disk block

## B-Trees

Invented by Bayer and McCreight in 1972
A B-tree is a Balanced $M$-way search tree, $M \geq 2$ (usually ~100)

Search takes $O(\log M \log N)$
Insertion and removal each takes $O(M \log N)$ time

B-Trees are covered in detail in EECS 484, here we look at the in-memory versions: 2-3 trees and 2-3-4 trees (a.k.a., 2-4 trees)

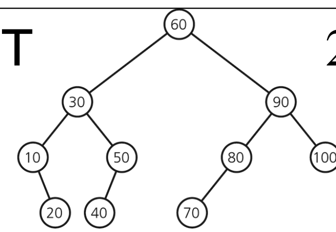
## Advantage 2: Balanced Search Trees

AVL trees keep a BST balanced by limiting how unbalanced a tree can be

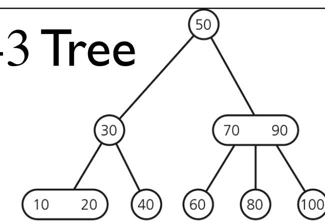
Perfect binary trees are by definition balance, but perfect binary trees of height $h$ have exactly $2^{h+1} - 1$ internal nodes, so only trees with $1, 3, 7, 15, 31, 63, \ldots$ internal nodes can be balanced . . .

Balance $M$-way trees prevent a tree from becoming unbalanced by storing more than 1 keys per node such that the trees are always perfect (but not binary) trees

## BST




## 2-3 Tree



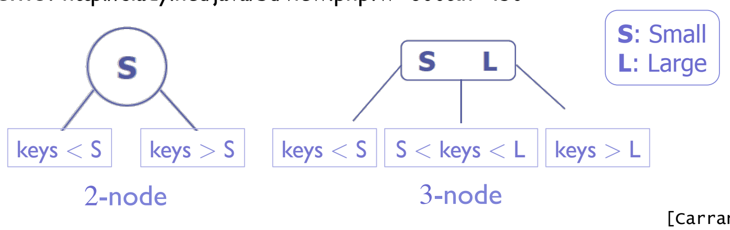

insert 39, 38, ... 32

# 2-3 Trees

Properties (balance condition) of 2-3 trees:

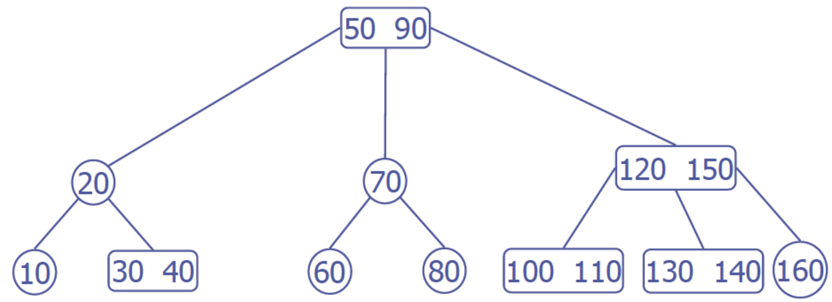
1. all leaf nodes at the same level and contain 1 or 2 keys
2. an internal node has either 1 key and 2 children (a 2-node) or 2 keys and 3 children (a 3-node)
3. a key in an internal node is "between" the keys in its adjacent children

Demo: http://slady.net/java/bt/view.php?w=600&h=450




[Carrano]

# 2-3 Tree Example




[Carrano]

# 2-3 Trees Search Times

A 2-3 tree of height $h$ has least number of nodes when all internal nodes are 2-nodes (a BST)

- since all leaves must be at the same level, the tree is a perfect tree and the number of nodes (and therefore keys) is $n = 2^{h+1}-1 \Rightarrow h = \text{floor}(\log_2 n)$

A 2-3 tree of height $h$ has the largest number of nodes when all internal nodes are 3-nodes

- number of nodes: $N = \sum_{i=0}^{h} 3^i = (3^{h+1}-1)/2$
- number of keys (each node has 2 keys): $n = 3^{h+1}-1 \Rightarrow h = \text{floor}(\log_3 n)$

Search time on 2-3 trees: $O(\log n)$

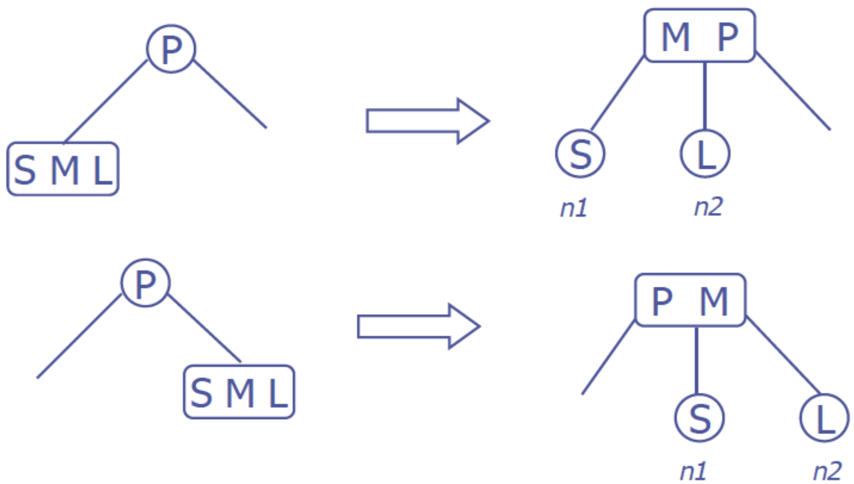
# 2-3 Trees Insert

As with BST, a new node is always inserted as a leaf node

1. Search for leaf where key belongs; remember the search path
2. If leaf is a 2-node, add key to leaf
3. If leaf is a 3-node, adding the new key makes it an invalid node with 3 keys, split the invalid node into two 2-nodes, with the smallest and largest keys, and pass the middle key up to parent
4. If parent is a 2-node, add the child's middle key with the two new children, else split parent by Step 3 above
5. If there's no parent, create a new root (increase tree height by 1)

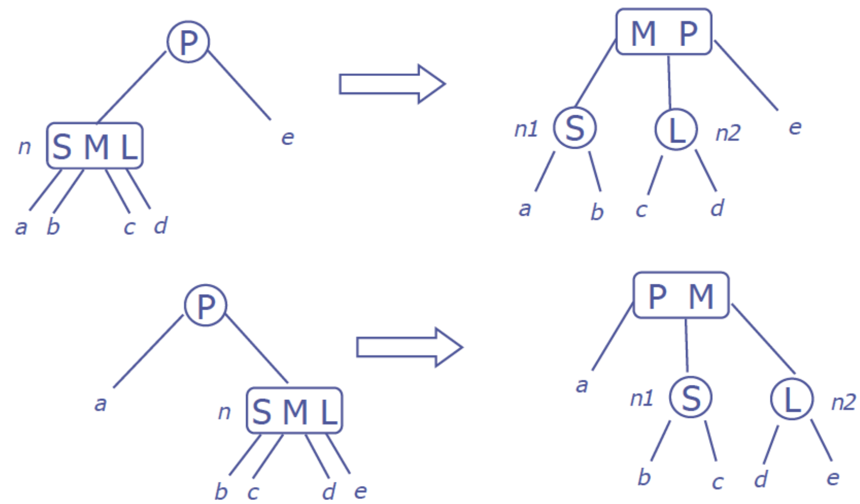
Observation: whereas a BST increases height by extending a single path, a 2-3 tree increases height globally by raising the root, hence it's always balanced

## Splitting a Leaf Node



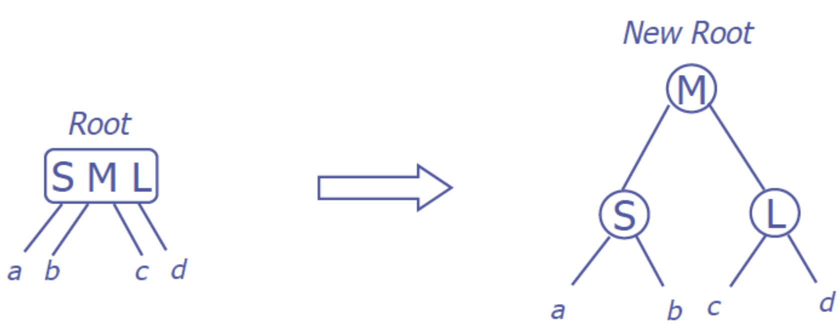

[Gordon-Ross]

## Splitting a Non-Leaf Node



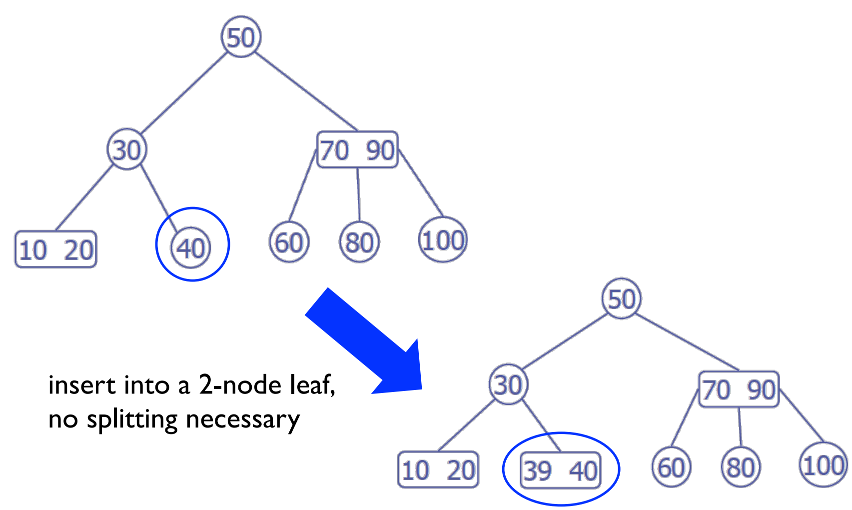

[Gordon-Ross]

## Splitting and Raising the Root



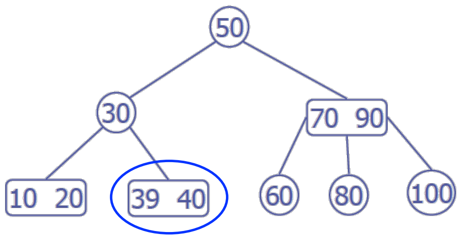

[Gordon-Ross]

## Insert 39

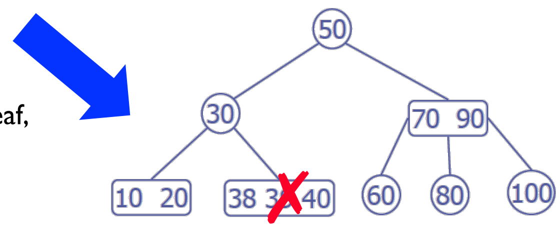



[Gordon-Ross]

## Insert 38

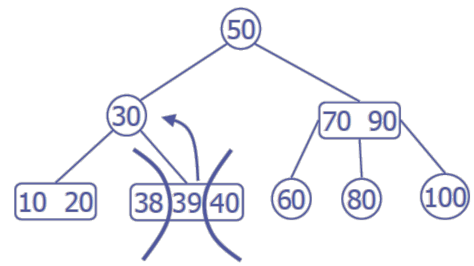



insert to a 3-node leaf,
splitting necessary

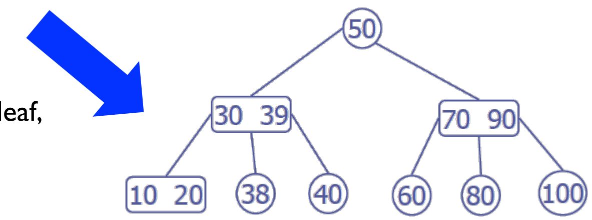



[Gordon-Ross]

## Insert 38: Split Leaf Node

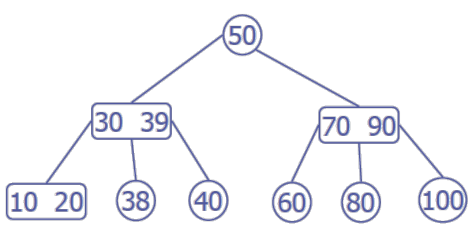


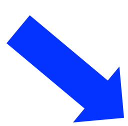

insert to a 3-node leaf,
splitting necessary



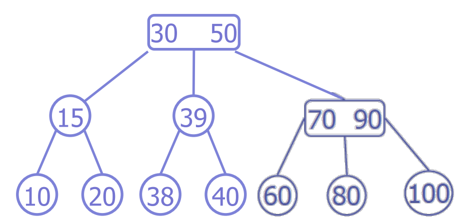

[Gordon-Ross]

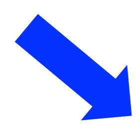
## Exercise: Insert 15



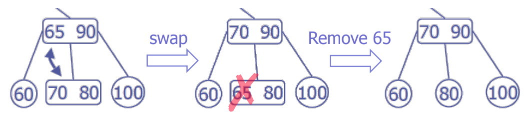
## Exercise: Insert 75 Then 85

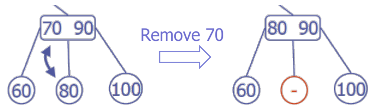


# 2-3 Trees Removal

1. If item to be removed is not in a leaf node, swap with the max of the child to the key's right (the next bigger item, or in-order successor)
2. If $n$ is a 3-node, remove item, done
3. else if $n$ is a 2-node:




next slide

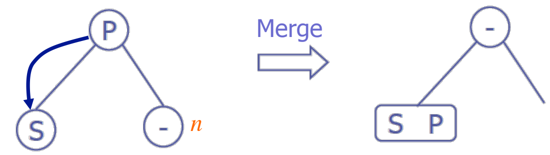
```
while (n has no item &&
       n is not root ) {
   let p be the parent of n;
   let q be the adjacent
        sibling of n (left or right);
   if (q is a 3-node) rotate items;
   else merge nodes;
   n=p;
}
if (n has no item, n must be root)
   the child of n becomes the new root, remove n;
```

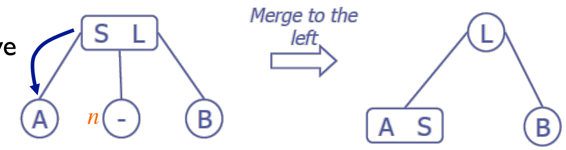
[Carrano]

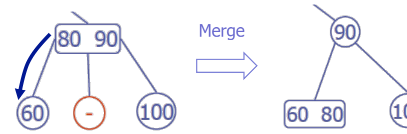
# Removal: Leaf Node

## Merging:

Sibling is not a 3-node:
→ merge nodes
→ move item from parent to sibling
→ merge to left to fill node left to right (think: array)
→ merging could leave parent without any item



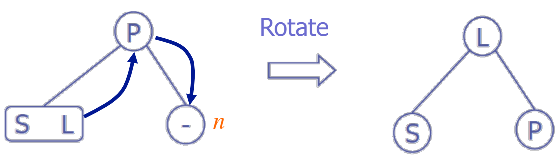

S: Small
L: Large
P: Parent

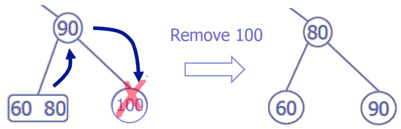
[Singh,Carrano]

# Removal: Leaf Node

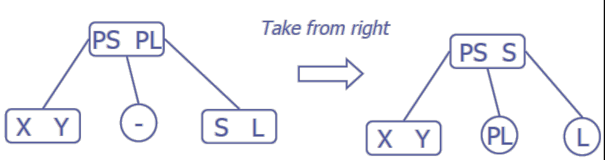
## Rotation:

Sibling is a 3-node:
→ redistribute items between siblings and parent
→ take from right to empty parent right to left (think: array implementation)




S: Small
L: Large
P: Parent

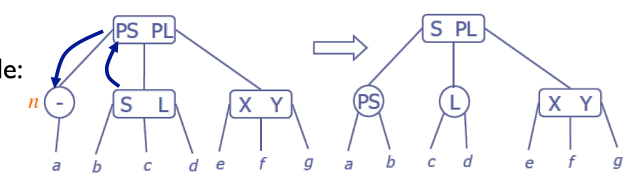



[Singh,Carrano]

# Removal: Non-Leaf Node

## Rotation:

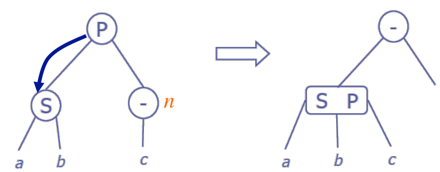
Sibling is a 3-node:
→ redistribute items
→ adopt child




S: Small
L: Large
P: Parent
PS: Parent of S
PL: Parent of L

## Merging:

Sibling is not a 3-node:
→ merge nodes
→ move item from parent to sibling
→ adopt child of $n$

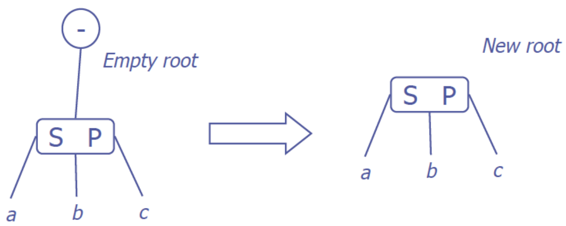



If $n$'s parent ends up without item, apply process on parent

[Singh,Carrano]

## Removal: Root

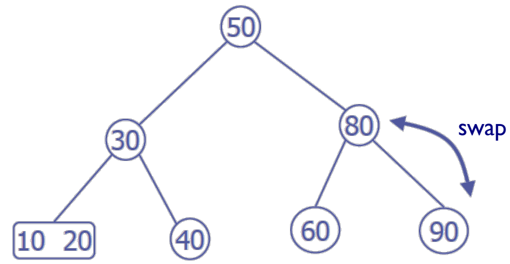
If merging process reaches the root and root is without item → remove root



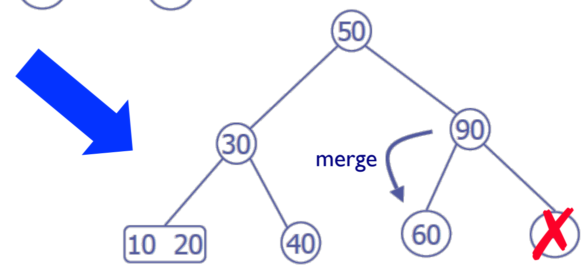

**Observation**: whereas a BST pushes "holes" down to the leaves, a 2-3 tree percolates "holes" up and decreases height globally by lowering the root

[Singh,Carrano]

## Remove 80: Percolated Merging

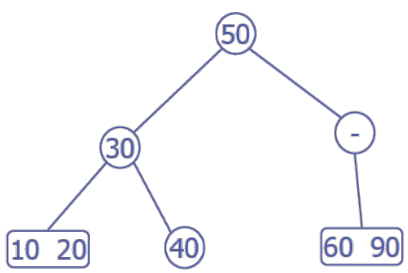



can't redistribute, must merge

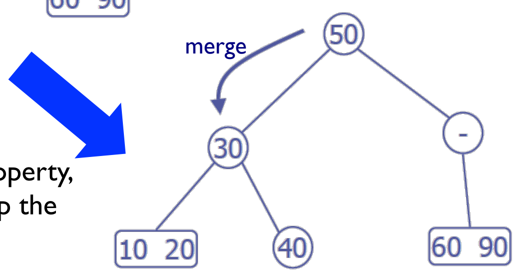



[Carrano]

## Remove 80: Percolated Merging

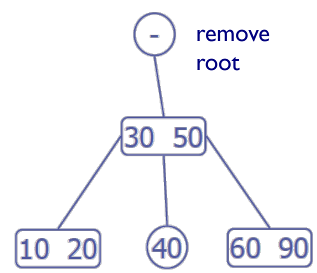



violates 2-3 tree property, merge recursively up the tree

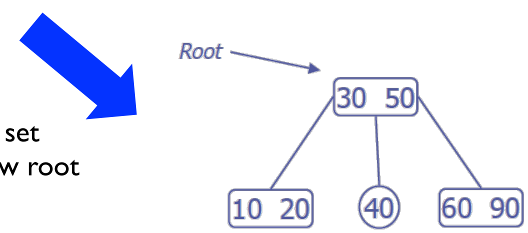



[Carrano]

## Remove 80: Percolated Merging



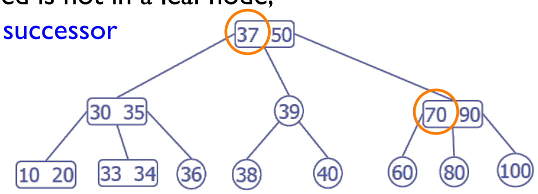

root is now empty, set tree to point to new root




[Carrano]

# Remove 37 Then 70

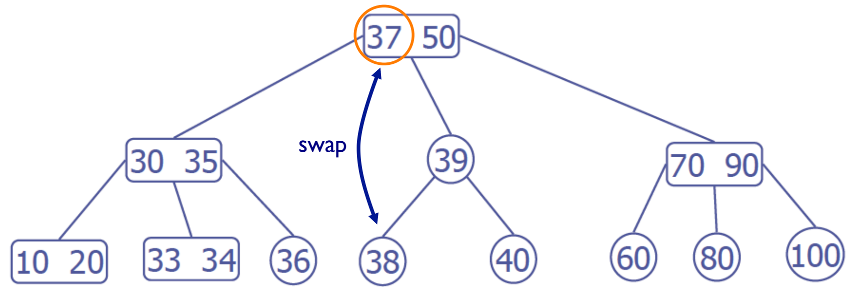
1. Removal always begins at a leaf node
2. If item to be removed is not in a leaf node, swap with in-order successor
3. If $n$ is a 3-node, remove item, done
4. if $n$ is a 2-node:




```
while (n has no item &&
       n is not root ) {
  let p be the parent of n;
  let q be the adjacent sibling of n (left or right);
  if (q is a 3-node) rotate items;
  else merge nodes;
  n=p;
}
if (n has no item, n must be root)
  the child of n becomes the new root, remove n;
```

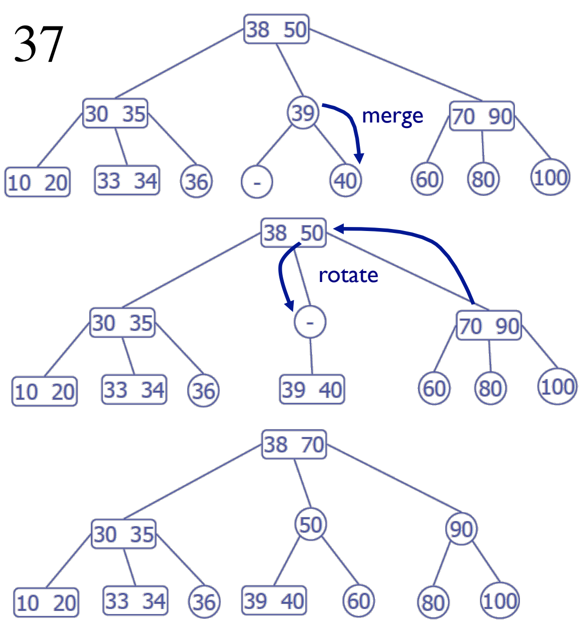
[Carrano]

# Remove 37



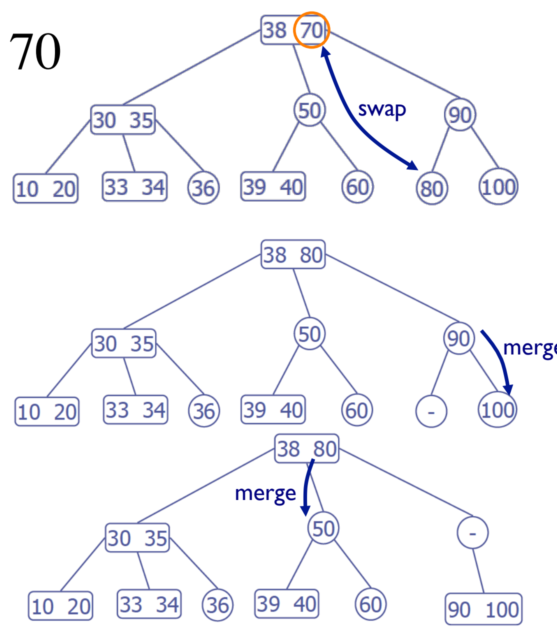

[Carrano]

# Remove 37



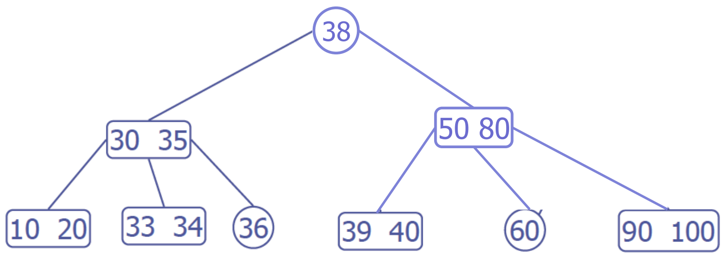

[Carrano]

# Remove 70




[Carrano]

# Remove 70



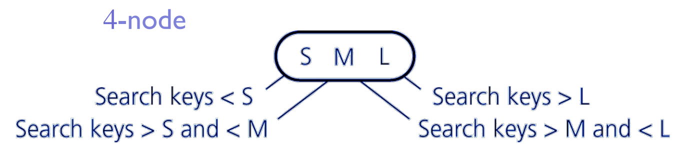

[Carrano]

# 2-3-4 Trees

Similar to 2-3 trees

- also known as 2-4 trees
- demo: http://www.cse.ohio-state.edu/~bondhugu/acads/234-tree/index.shtml

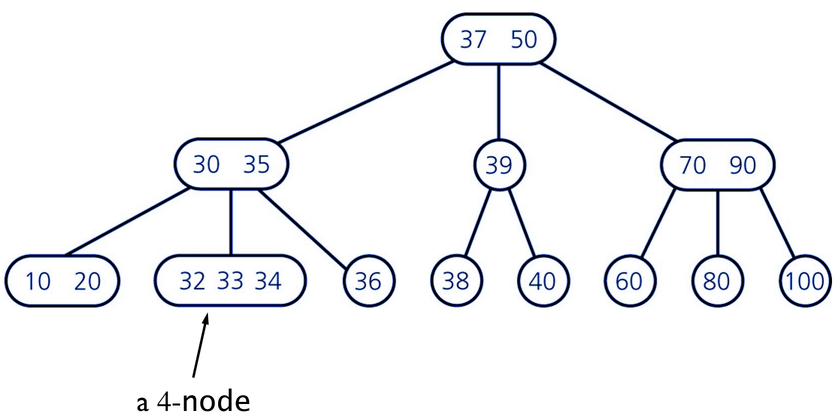
4-nodes can have 3 items and 4 children




Why bother? Unlike with 2-3 trees, insertions and removals in 2-3-4 trees can be done in one pass

[Singh,Carrano]

# 2-3-4 Tree Example




[Carrano]

# 2-3-4 Trees Insert

Items are inserted at leaf nodes

Since a 4-node cannot take on another item, 4-nodes are preemptively split up during the insertion process

On the way from the root down to the leaf: split up all 4-nodes “on the way”

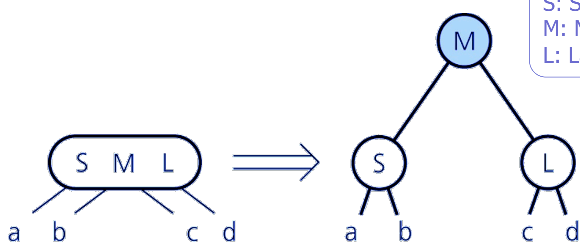
→ insertion can be done in one pass
(in 2-3 trees, a reverse pass is likely necessary)

→ no worrying about overflowing a node when we actually do the insertion–the only kind of node that can overflow (a 4-node) has been made a 2- or 3-node

[Singh,Carrano]

# 2-3-4 Trees Insert

Splitting a 4-node

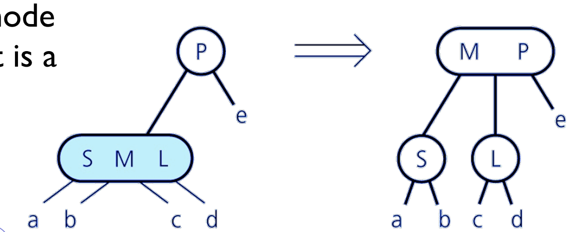



S: Small
M: Medium
L: Large

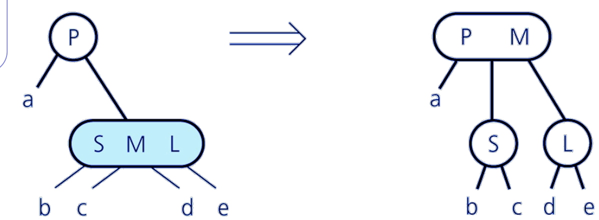
[Carrano]

# 2-3-4 Trees Insert

Splitting a 4-node whose parent is a 2-node

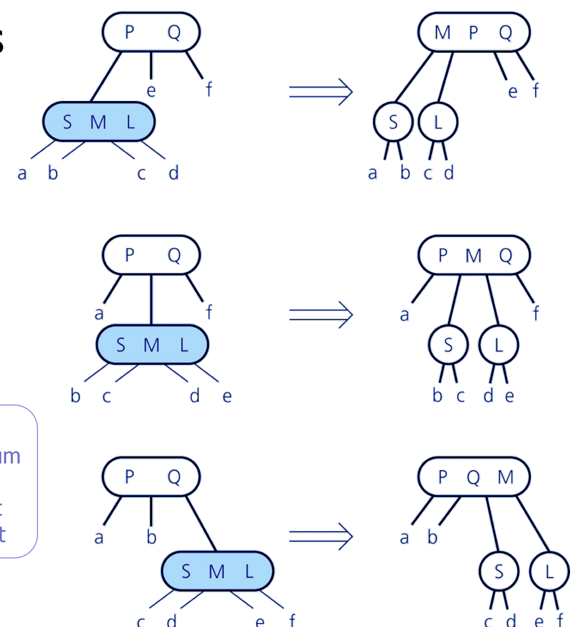



S: Small
M: Medium
L: Large
P: Parent




[Carrano]

# 2-3-4 Trees Insert



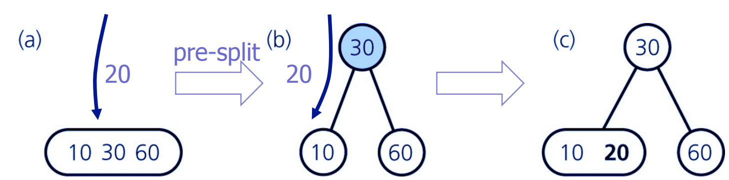

Splitting a 4-node whose parent is a 3-node

S: Small
M: Medium
L: Large
P: Parent
Q: Parent

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 60, 30, 10,

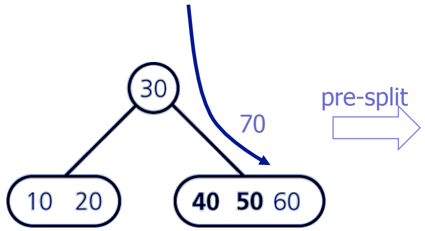



... 50, 40 ...

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 50, 40, ...

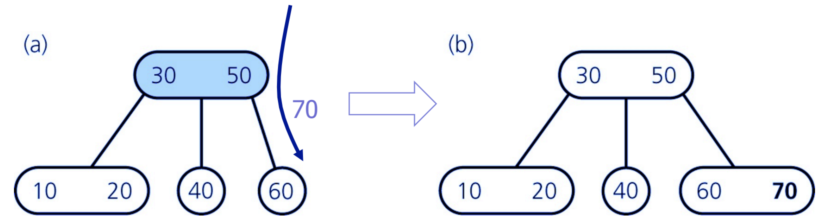



... 70, ...

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 70 ...

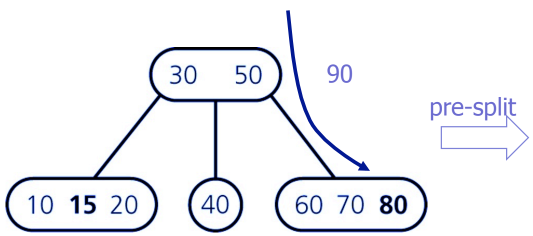



... 80, 15 ...

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 80, 15 ...

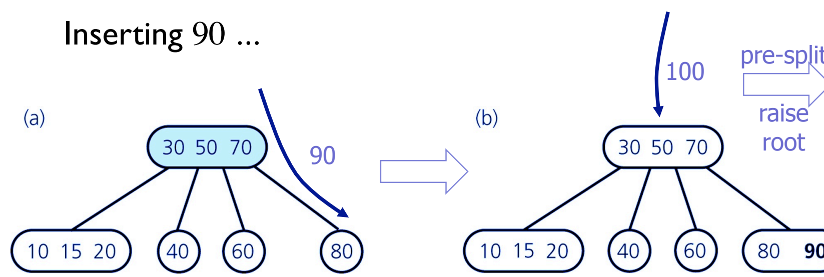



... 90 ...

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 90 ...

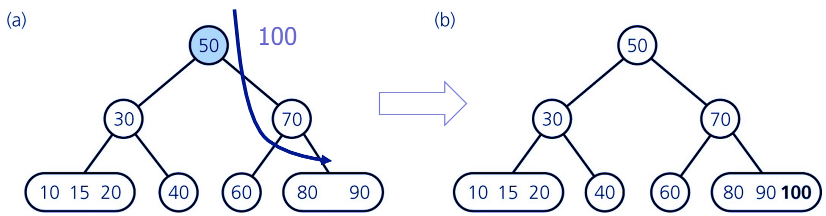



... 100 ...

[Carrano]

# 2-3-4 Trees Insert Example

Inserting 100 ...




[Carrano]

# 2-3-4 Trees Removal

Removal always begins at a leaf node
→ swap item of non-leaf node with in-order successor

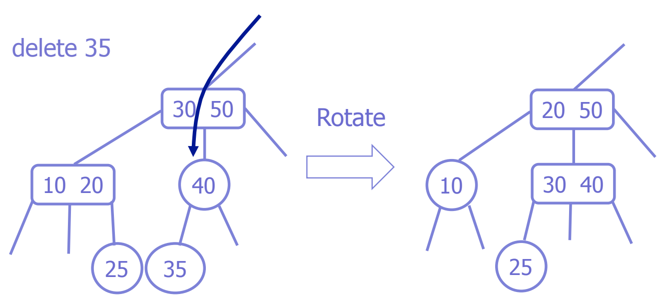
Whereas a 4-node can overflow during insertion,
a 2-node can become empty during removal

On the way from root down to the leaf:
turn 2-nodes (except root) into 3-nodes
→ prevents a 2-node from becoming an empty node
→ deletion can be done in one pass
(in 2-3 trees, a reverse pass is likely necessary)

[Singh,Carrano]

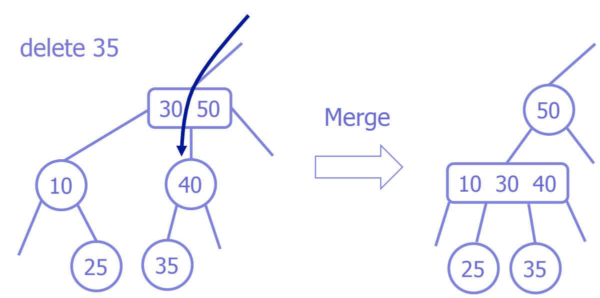
# Removal: 2-Node → 3-Node

Rotate: if adjacent sibling is a 3- or 4-node
→ redistribute items from sibling
→ adopt child



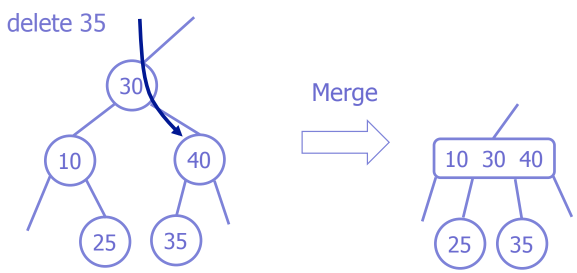

# Removal: 2-Node → 3-Node

Merge: if adjacent sibling is a 2-node
→ redistribute item from parent
parent has at least 2 items, unless it's root
→ merge nodes



# Removal: 2-Node → 3-Node

Root merge: if parent is root and both parent and adjacent sibling are 2-nodes
→ merge with parent and sibling



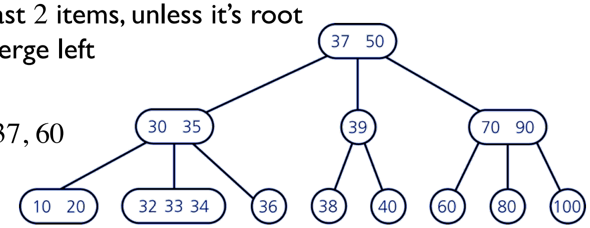

# 2-3-4 Trees Removal Summary

On the way from root down to the leaf: turn 2-nodes (except root) into 3-nodes

Rotate: if adjacent sibling is a 3- or 4-node
→ redistribute items from sibling, take from right
→ adopt child

Merge: adjacent sibling is a 2-node
→ redistribute item from parent; parent has at least 2 items, unless it's root
→ merge nodes, merge left

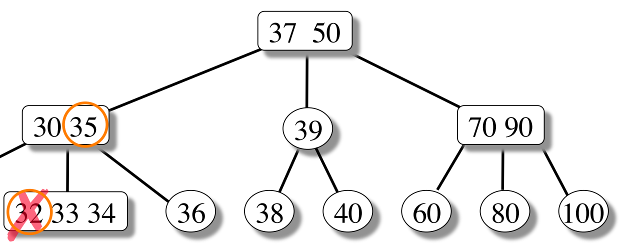
Exercise: remove
32, 35, 40, 38, 39, 37, 60



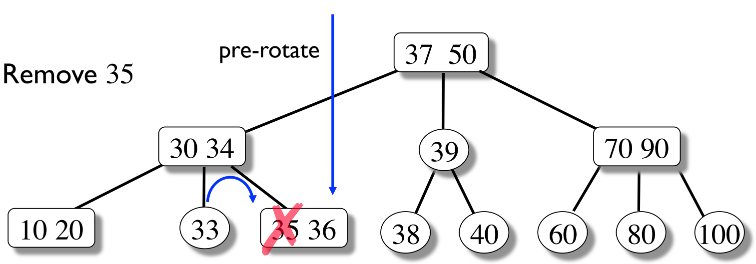

[Carrano]

# Remove 32, 35, 40, 38, 39, 37, 60

Remove 32



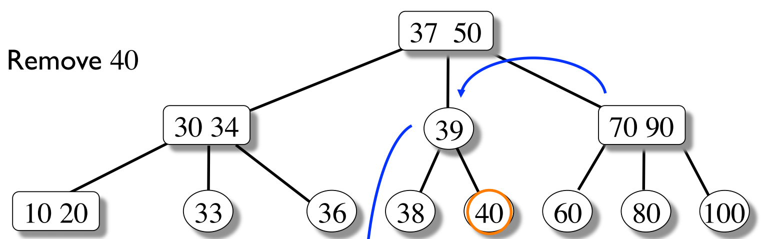
Remove 35

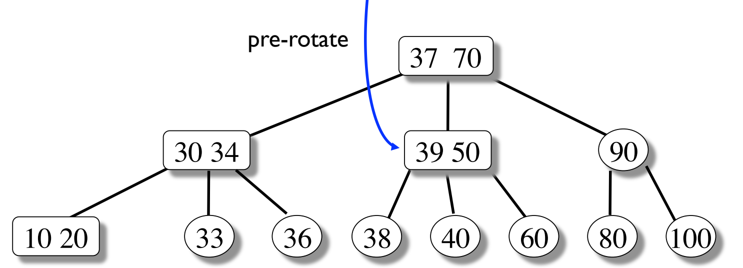



[Carrano]

# Remove 32, 35, 40, 38, 39, 37, 60

Remove 40

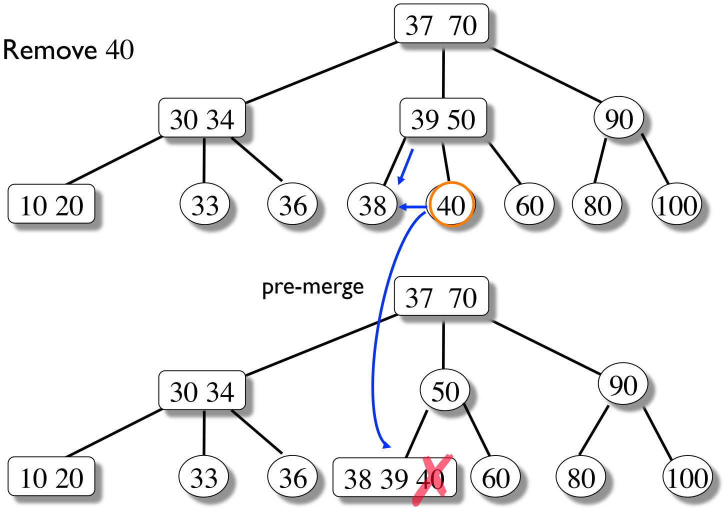





[Carrano]

## Remove 32, 35, 40, 38, 39, 37, 60

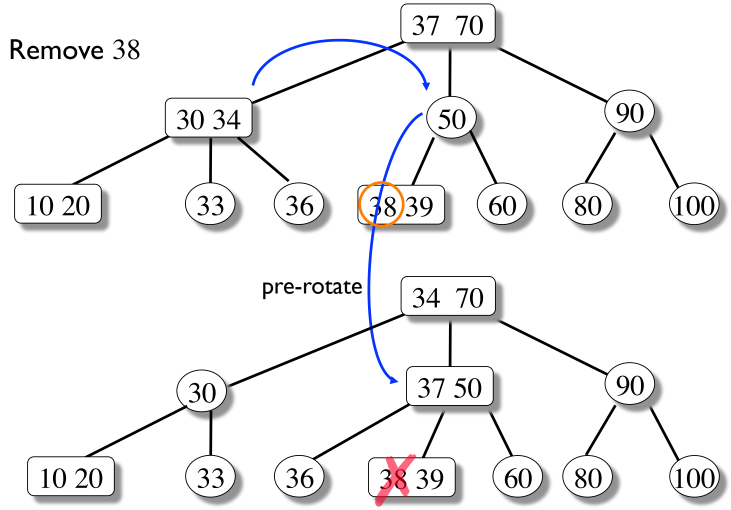
Remove 40




[Carrano]

## Remove 32, 35, 40, 38, 39, 37, 60

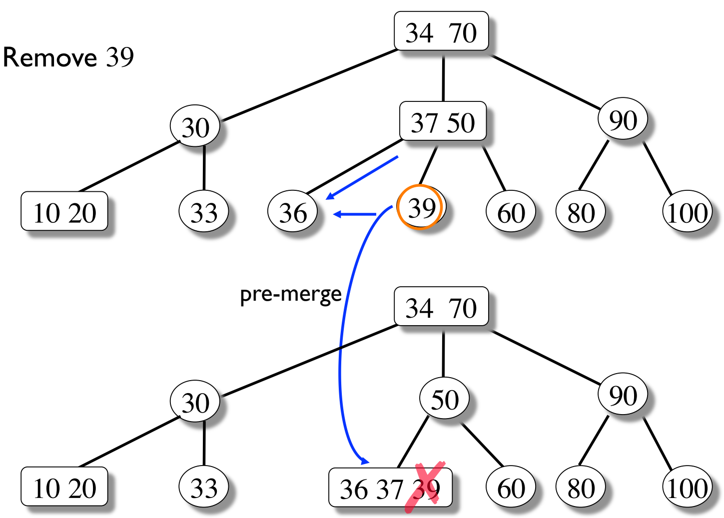
Remove 38




[Carrano]

## Remove 32, 35, 40, 38, 39, 37, 60

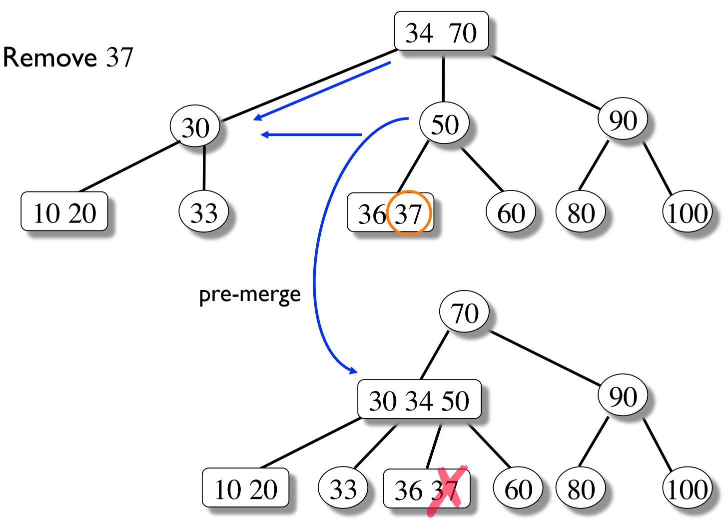
Remove 39




[Carrano]

## Remove 32, 35, 40, 38, 39, 37, 60

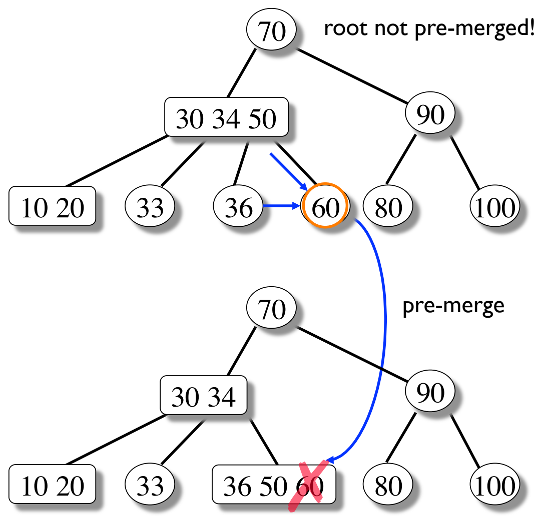
Remove 37




[Carrano]

# Remove 32, 35, 40, 38, 39, 37, 60

Remove 60




[Carrano]

# Compared to 2-3 Trees

Insertion and deletion are easier for 2-4 tree

- one pass
- no need to percolate over/under-flow node all the way back up to root

But at the cost of:

- extra comparison in each node
- wasted space in each node (a 2-node is actually a 4-node with two empty slots and 2 null pointers)
- pre-emptive splitting of 4-nodes pre-allocates space that may not be needed right away → further wasting space
- number of NULL pointers in a tree with $N$ internal nodes is $4N - (N - 1) = 3N+1$

[Rosenfeld,Brinton]

# Implementation

While 2-3 trees and 2-4 trees are conceptually clean, their implementation is complicated because

- we need to maintain multiple node types and
- there are a lot of cases to consider, such as whether we are
  - redistributing from a left sibling or a right sibling
  - merging with a 2-node or a 3-node
  - merging with the small or the large item of the parent
  - passing a node to a 2-node or to a 3-node parent
  - filling the small, middle, or large item slot at the parent
  - adopting a left child or a right child
  - rotating left or right

It would be nice if we could simplify these cases *and* reduce the amount of wasted space by turning 2-3 and 2-4 trees into binary trees …

[Rosenfeld]