Data Structures and Algorithms

Lecture 10: Indexing & Advanced Trees

Outline

- Indexing
  - Linear index
  - Tree-based index
    - B-trees, B^+-trees
- Advanced Trees
  - Tries
  - Balanced trees
    - AVL tree, Red-black tree, BB(α) tree, Splay tree
  - Spatial data structures
    - K-D tree, PR quadtree

Application limitations of Hash

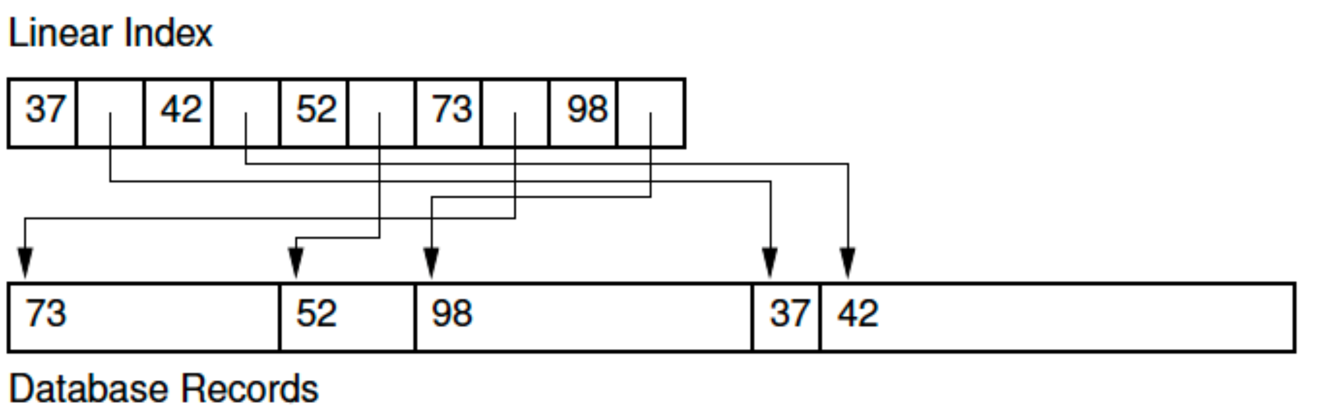
- Hash provides excellent performance for insert, search, and delete, i.e.,
  - Time complexity $\Theta(1)$ on average
- But hash has some application limitations:
  1. Do not support duplicate keys
  2. Only provide exact-search, but not range search
     - E.g., search the students with their height between 1.7m and 1.75m
  3. Do not support efficient searching the record with the minimum or maximum key

What is Index ?

- Index provides following operations:
  - efficient Insert (with duplicate keys): $\Theta(\log n)$
  - efficient exact-search: $\Theta(\log n)$
  - efficient range-search, time is related to the range, but usually is much shorter than $\Theta(n)$
  - Efficient minimum / maximum search: $\Theta(\log n)$
  - Efficient delete: $\Theta(\log n)$
- Index is designed for a large collection of records stored on disks, where the disk access time is much slower than memory access time.

Linear indexing

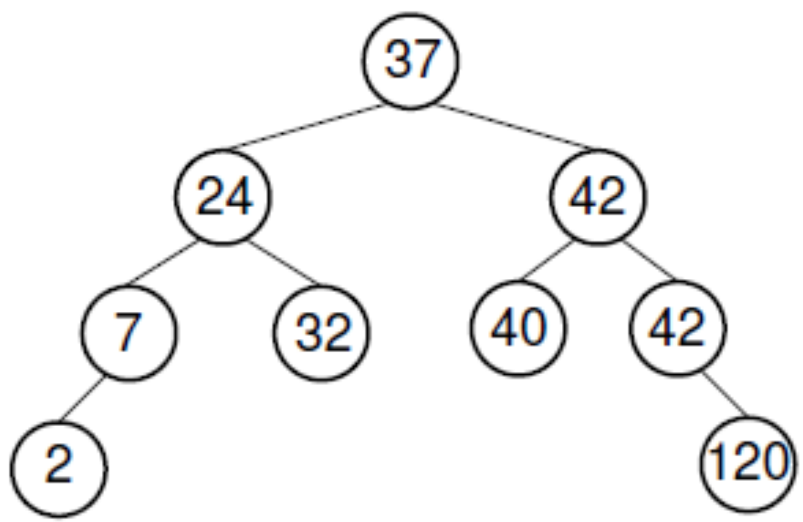
- A linear index is an index file organized as a sequence of key-value pairs where the keys are in sorted order and the pointers either
  - (1) point to the position of the complete record on disk,
  - (2) point to the position of the primary key in the primary index,
  - or (3) are actually the value of the primary key.



➢ If the database contains enough records, the linear index might be *too large* to store in main memory. -> *expensive*!

Index techniques has many similarities with BST

- A binary search tree (BST) is a special binary tree, iff
  - For each node, assume the node value is *K*
  - The values of the nodes in its left subtree are < K
  - The values of the nodes in its right subtree are ≥ K



Index techniques

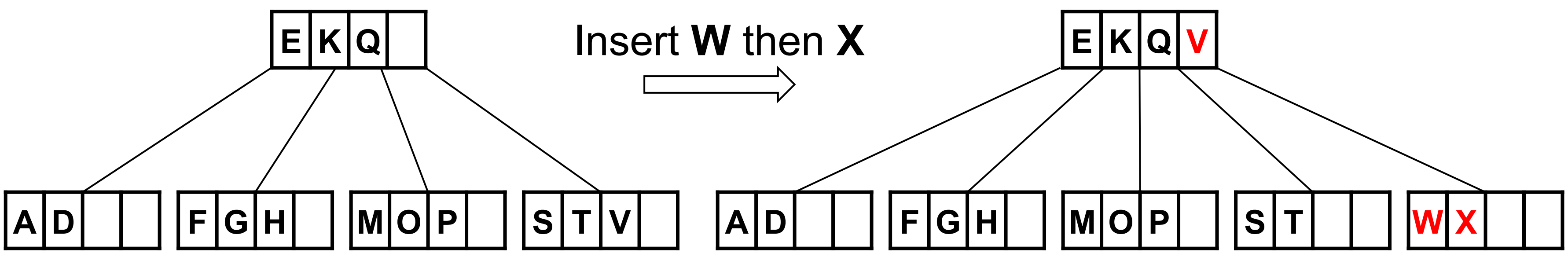
- Two Tree-based indexing techniques:
  - B trees
  - B+ trees
- Why not adopt a binary search tree (BST) for index?
  - A BST may not be balanced
    - E.g., One subtree has many nodes, while the other has a few nodes, poor performance
  - The depth of a balanced BST is still large
    - Need about $log_2\ n$ searches, and possible $log_2\ n$ times of disk accesses, while a disk access is very time-consuming. This is unacceptable.

B tree

- B tree is a height Balanced tree.
- A B tree of order m is defined to have the following properties:
  - The root is has at least two children or either a leaf.
  - Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children.
    - Typically, m will be fairly large, e.g., m=100
  - All leaves are at the same level in the tree, so the tree is always height balanced.
  - The data values in each node are in ascending order.

Node insertion in a B tree

- Insertion follows similar logic to the BST.
  - Basic idea: search for the appropriate leaf, add the new value, then split and promote as necessary.



1. Insert **W** (just fills the leaf),
2. Then insert **X** (would cause the right-most leaf to split, and **V** to be promoted to the root),

Node deletion in a B tree

- The process of node deletion is similar to that in BST
- Deletion from a Leaf node
  - May drop the number of data values in the node below the mandatory floor ("underflow").
    - In this case, the leaf must borrow a value from an adjacent sibling node if node have a value to spare, or be merged with an adjacent sibling node.
- Deletion from an Internal node
  - Accomplished by reducing it to the former case.

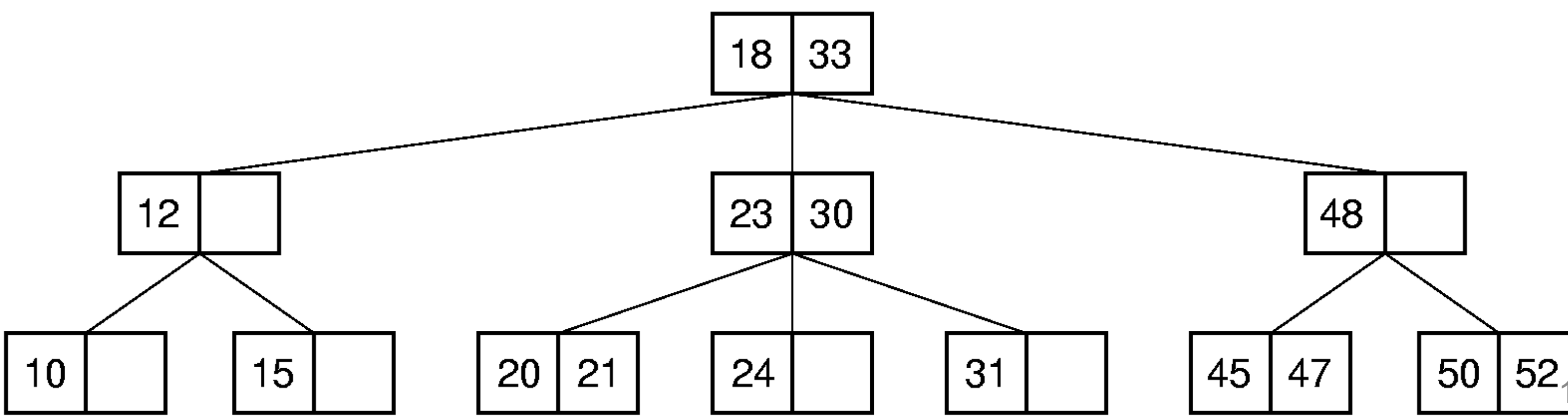
Search in a B tree

- Main steps:

1. Perform a binary search on the records in the current node.
   - If a record with the search key is found, then return that record.
   - If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

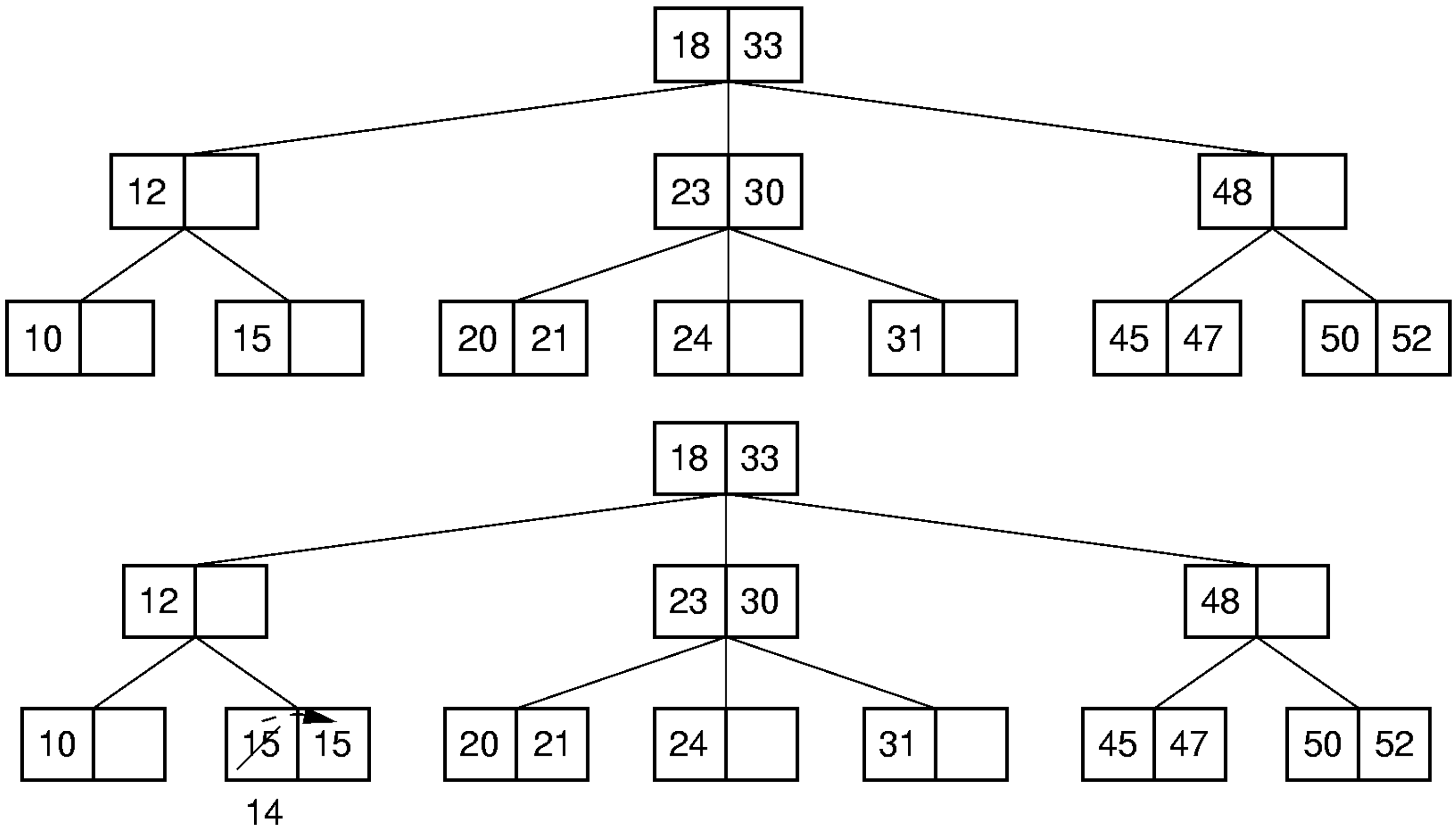
B tree example: 2-3 tree, i.e., m = 3

- Each internal nodes in a 2-3 tree has 2 or 3 children
  - A node contains one or two keys
  - All leaves are at the same level
- The 2-3 Tree has a property analogous to the BST:
  - left subtree < 1st key;
  - 1st key ≤ mid subtree < 2nd key;
  - right subtree ≥ 2nd key



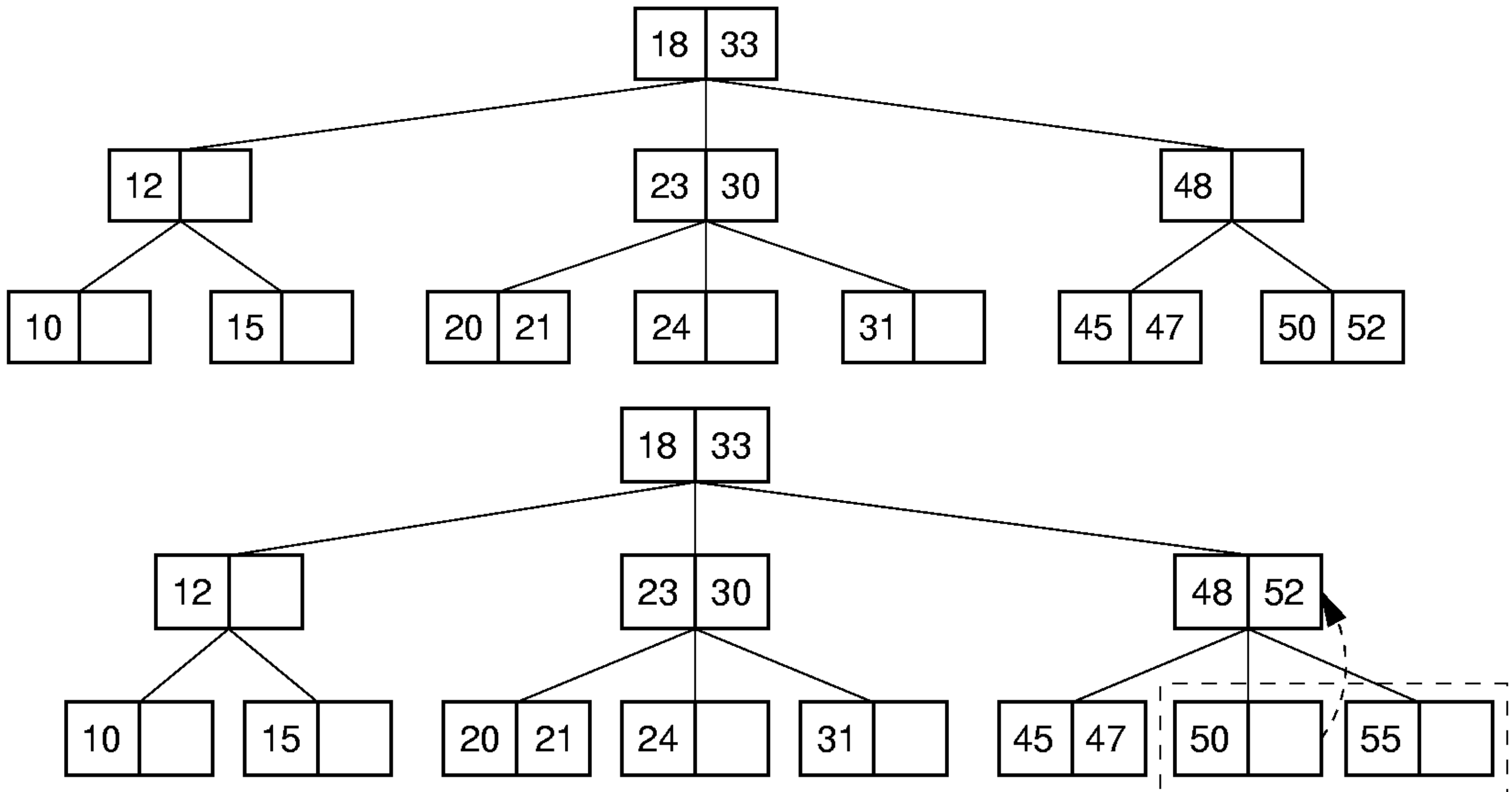
2-3 Tree

- The advantage of the 2-3 Tree over the BST is that it can be updated at low cost, e.g., insert 14

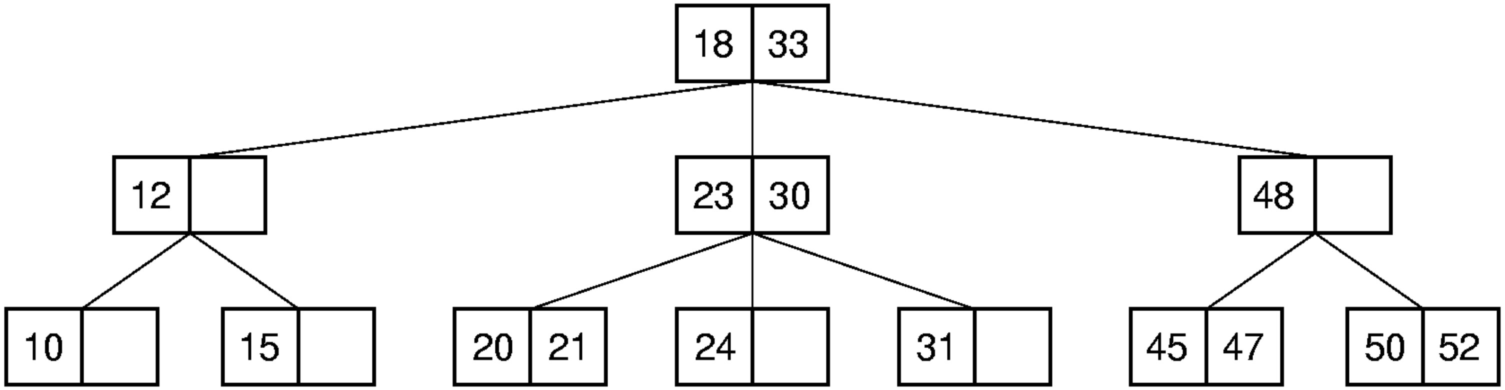


2-3 Tree Insertion, insert 55

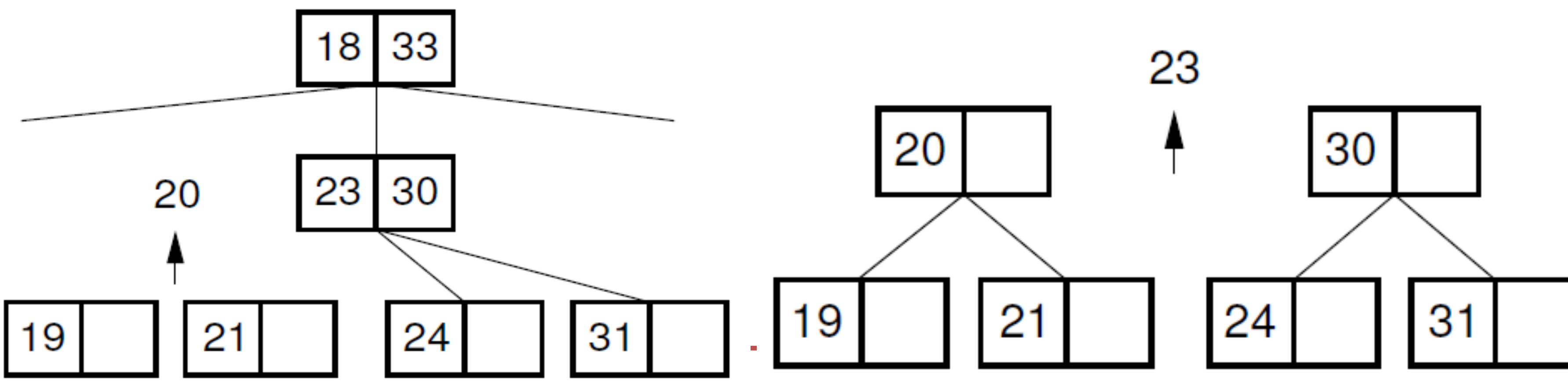
- Split the node has keys 50 and 52, and
- Promote the median of 50, 52, 55 to its parent



2-3 Tree Insertion, insert 19



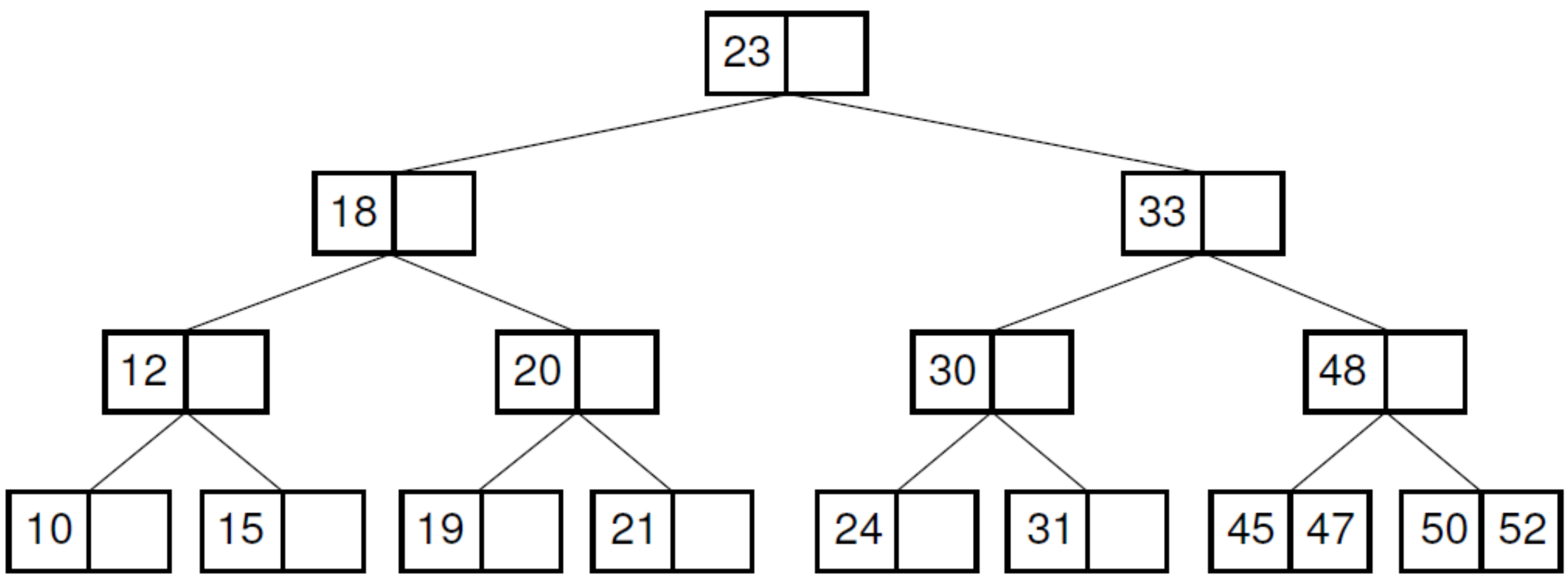
- Split the node has 20,21, promote 20, to node has 23, 30
- Then split node has 23,30, promote 23 to root has 18, 33



(a) (b)

2-3 Tree Insertion, insert 19

- Split the root has18, 33 due to the insertion 23, and
- promote 23, by creating a new root
- The tree height increase by 1
- But all leaves at the same level



(c)

B^+ Trees

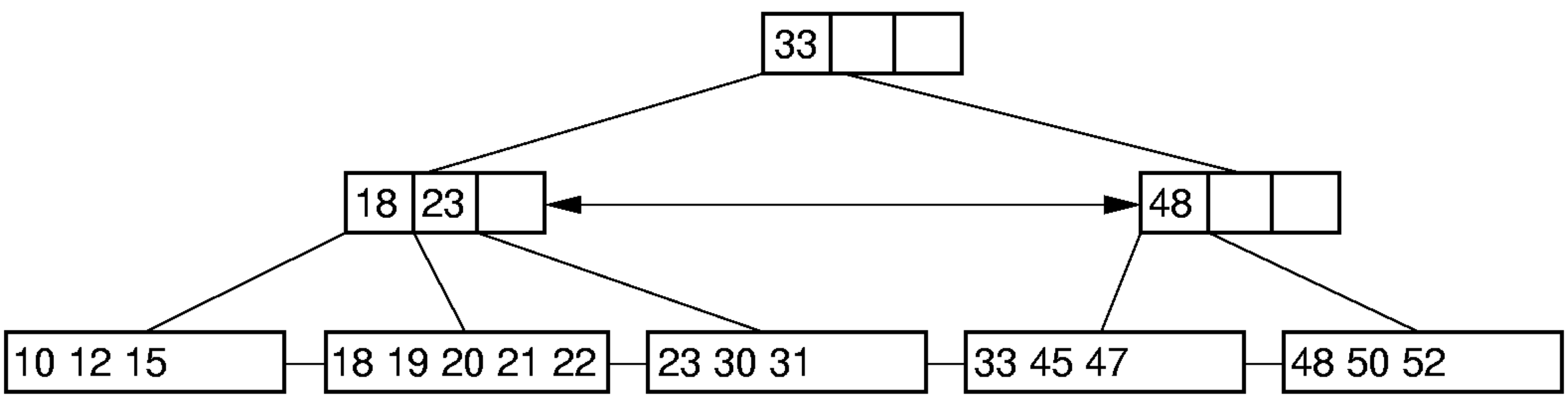
- The most commonly implemented form of the B-Tree is the B^+ Tree.
- B^+ tree stores records ONLY at the leaf nodes.
- Internal nodes store keys to guild the search.
- Leaf nodes store actual records, or else keys and pointers to actual records.
- A leaf node can store no more than m+1 records
- B+ tree supports $\Theta(1)$ time to search the previous or next record, of a given record.

B^+ Trees (cont'd)

- Define a B+ tree of order m as follows:
  - All data is stored at the leaf blocks
  - The root nodes is either:
    - A leaf block, or
    - An m-way tree with between 2 and m children
  - All other internal blocks are m-way trees with $\lceil m/2 \rceil$ to m children
    - The internal blocks store up to $m - 1$ keys to guide the searching where key k denotes the smallest key in sub-tree k.

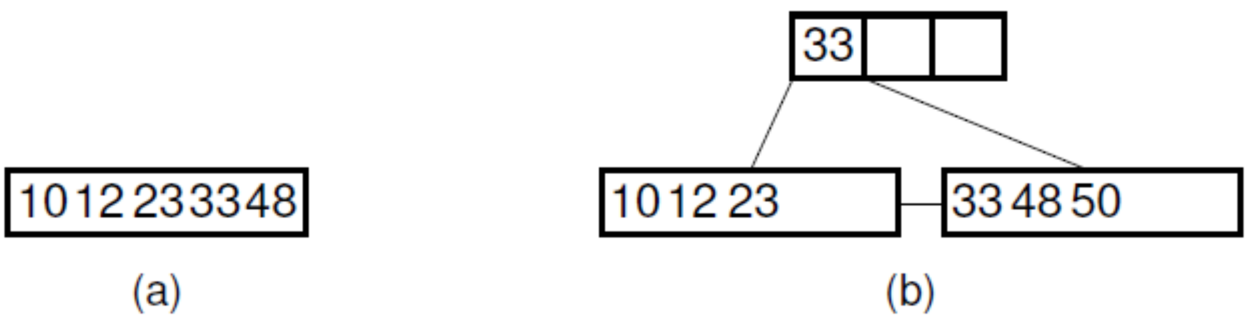
B$^+$-Tree Example with order m=4

- Each internal node should have from $\lceil m/2 \rceil = 2$ to m=4 children
- A leaf has no more than m+1= 5 records, but at least $\lceil (m+1)/2 \rceil = 3$ records
- Nodes in the same level are linked in order



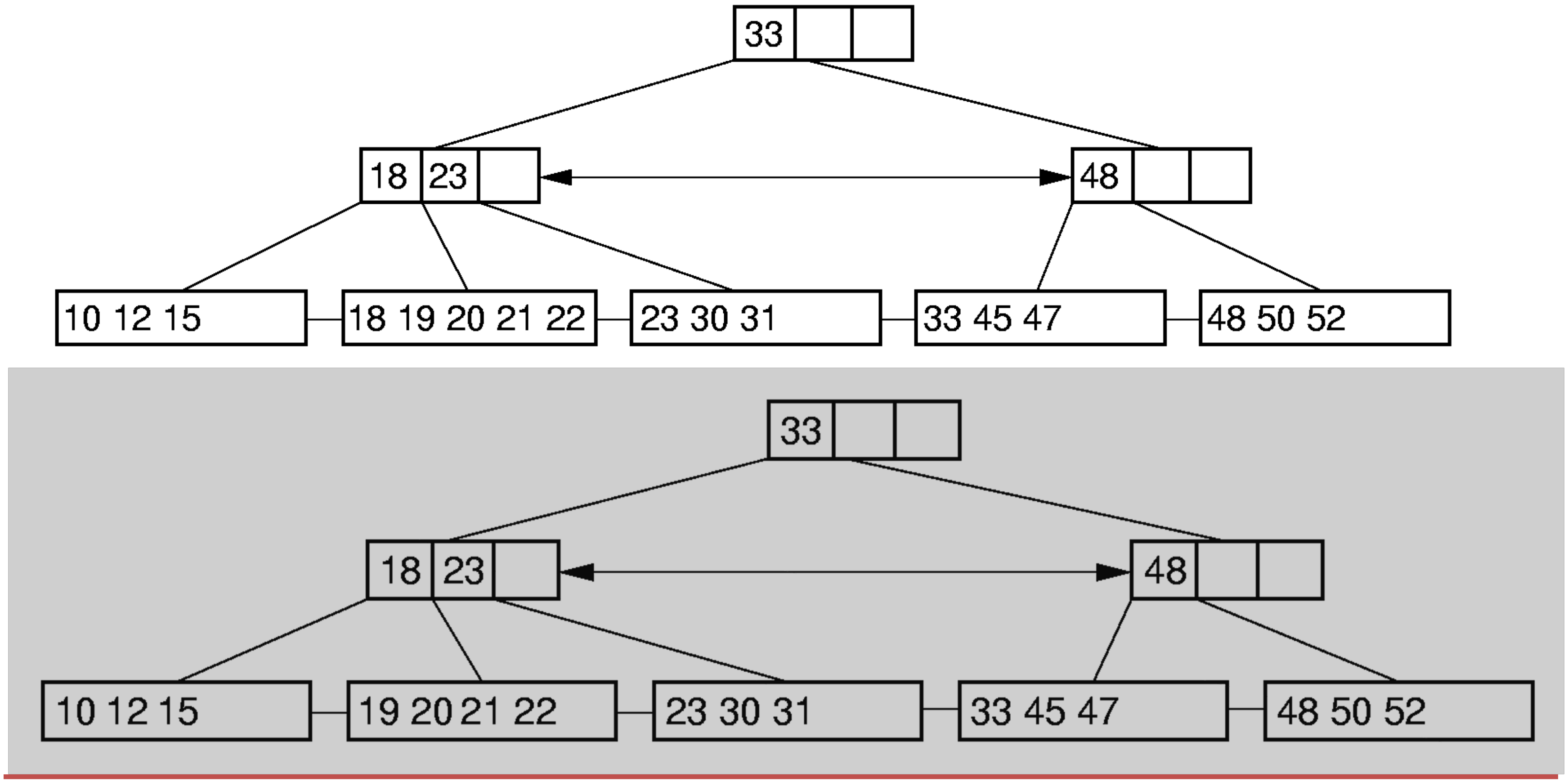
B⁺-Tree Insertion

- Insert 55
- Similar the insertion in B tree



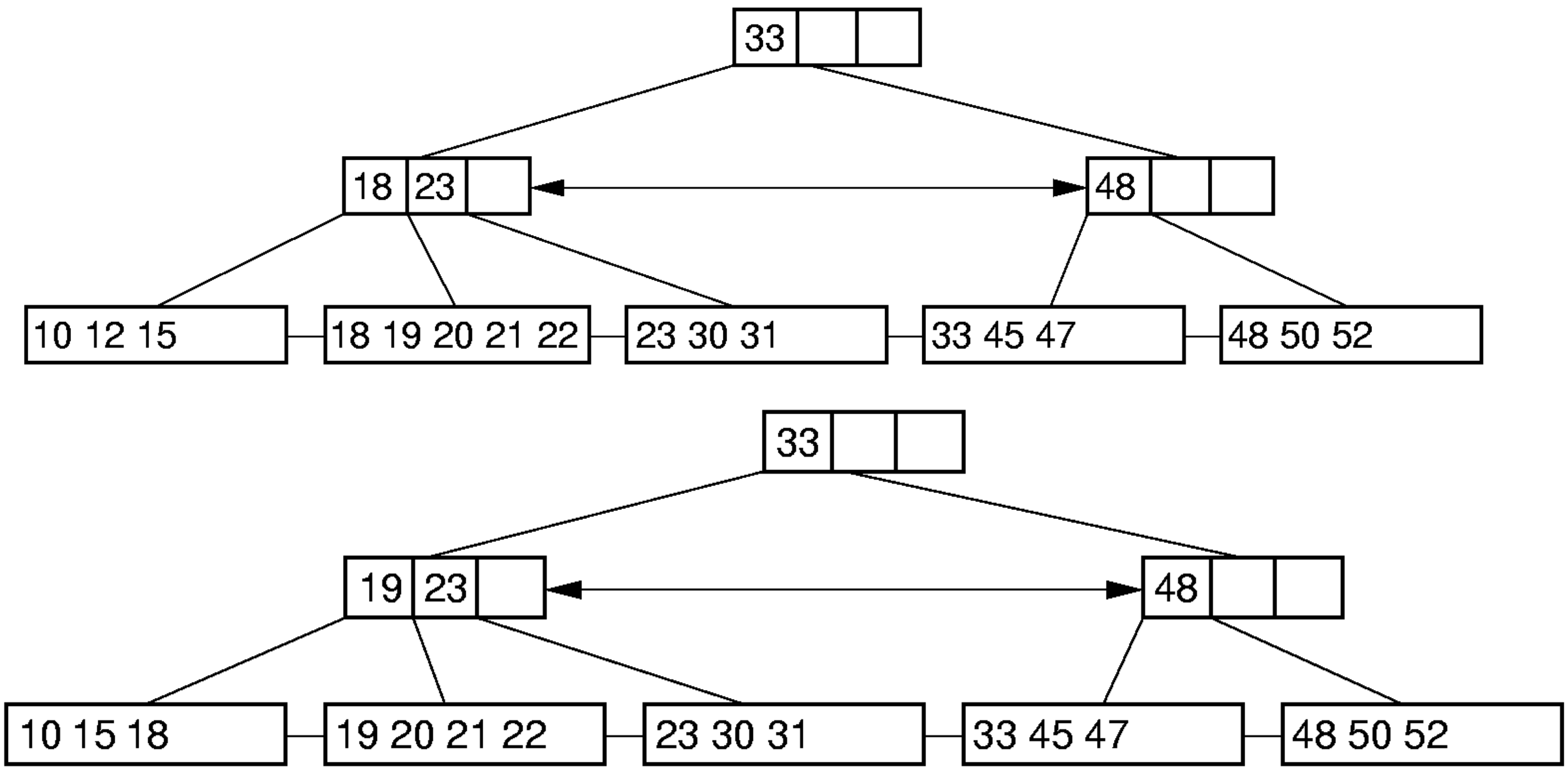
B$^+$-Tree Deletion (1) - delete 18

- Just remove key 18 from its leaf node



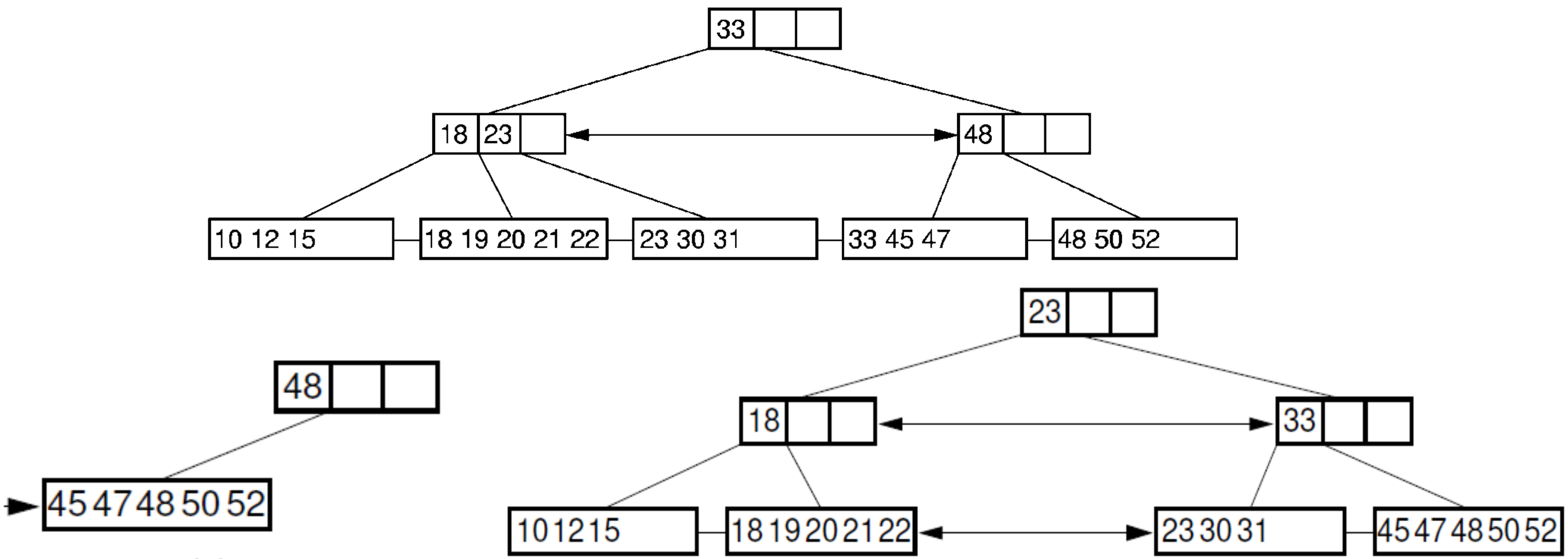
B$^+$-Tree Deletion (2) - delete 12

- Borrow one node 18 from its sibling to make it at least 3 nodes



B^+-Tree Deletion - delete 33

- Node having 33,45,47 cannot borrow from its siblings, merge with its one sibling node 48,50,52
- Node 48 has one less child, borrow one child from its sibling node having 18,23, modify guide keys



(a)
(b)

B* Trees

- A variant of B^+ trees
- All nodes except the root are required to be at least 2/3 full rather than 1/2 full.
- Splitting transforms 2 nodes into 3, rather than 1 node into 2.
- Can be generalized to specify a fill factor of (n+1)/(n+2); a B^n tree.

B-Tree Analysis

- Asymptotic cost of search, insertion, and deletion of nodes from B-Trees is $\Theta(\log n)$.
  - Base of the log is the (average) branching factor of the tree.
- Example: Consider a B+-Tree of order 100 with leaf nodes containing m=100 records.
  - 1 level B+-tree: Min 0, Max 100
  - 2 level: Min: 2 leaves of 50 (100 records). Max: 100 leaves with 100 (10,000 records).
  - 3 level: Min 2 x 50 nodes of leaves, for 5000 records. Max: 100^3 = 1,000,000 records.
  - 4 level: Min: 250,000 records (2 * 50 * 50 * 50). Max: 100^4 = 100 million records.

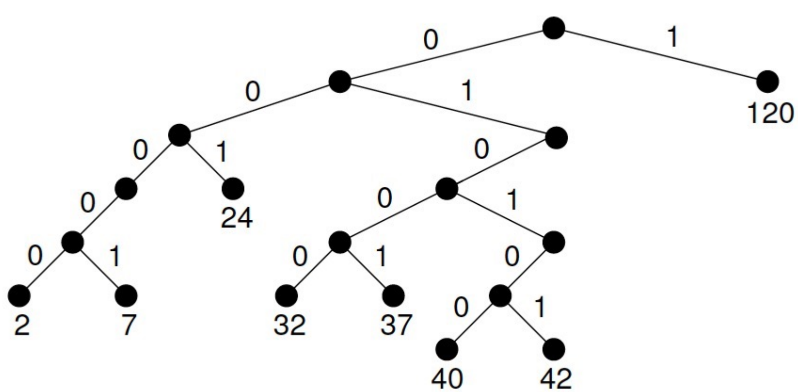
Advanced Tree Structures

Advanced Tree Structures

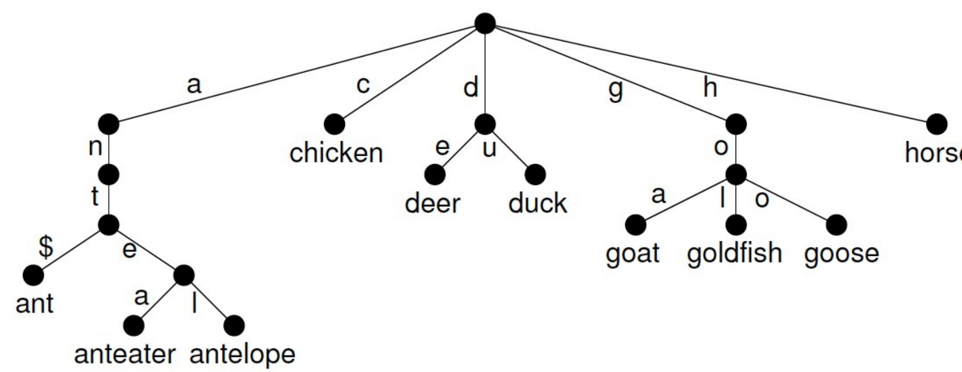
1. Tries
2. Balanced trees
   - AVL tree, Red-black tree, BB(α) tree, Splay tree
3. Spatial data structures
   - K-D tree
   - PR quadtree

1. Tries

- Binary Search Tree (BST) is a data structure based on *object space decmposition*.
- *Trie* is a data structure based on *key space decomposition*.



(a) Binary trie



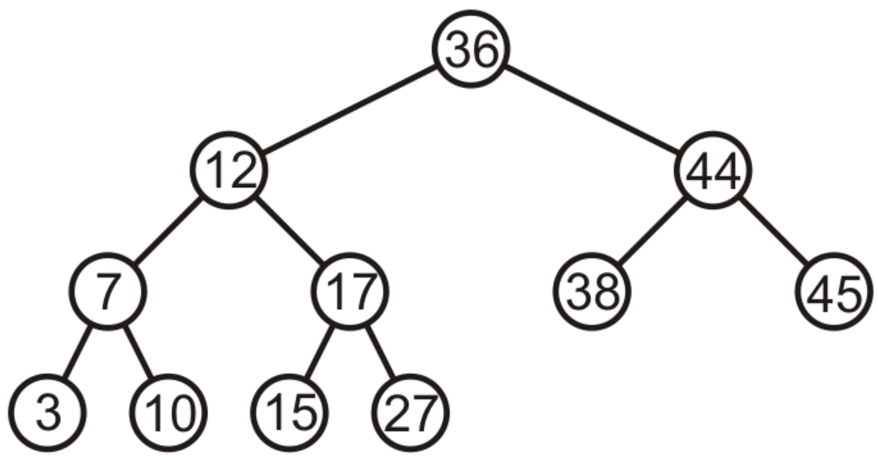
(b) alphabet trie

2. Balanced Trees

- **Balanced** may be defined by
  - *Height balanceing*: comparing the heights of the two sub trees
  - *Null-path-length balancing*: comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree/empty node)
  - *Weight balancing*: comparing the number of null sub-trees in each of the two sub trees
- Balance will ensure the height is $\Theta(\log n)$

Balanced trees - AVL tree

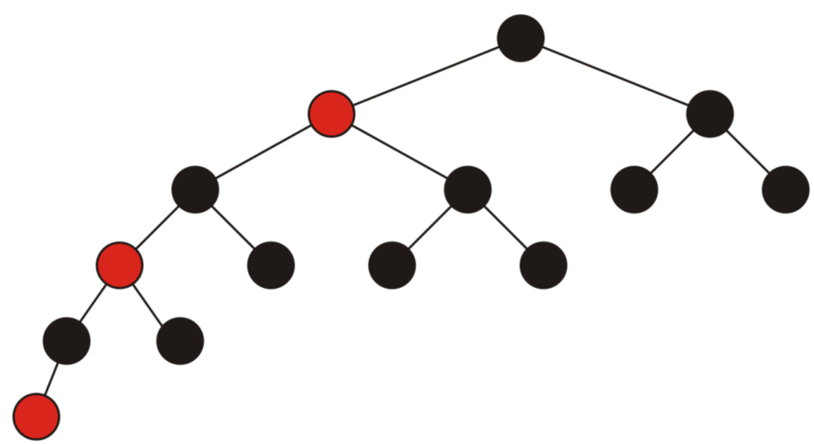
- AVL trees use height balancing
  - For every node, the heights of its left and right subtrees differ by at most 1.



AVL trees with the height of 4

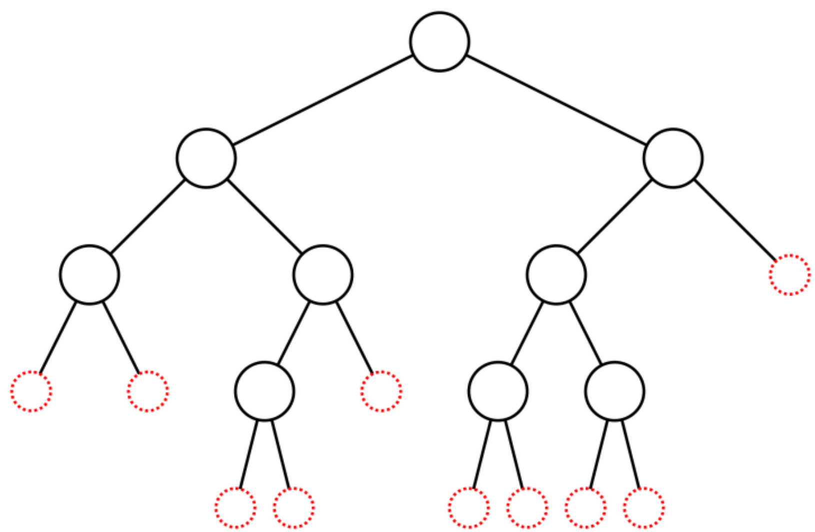
Balanced trees - Red-black tree

- Red-balack trees use null-path-length balancing
  - All nodes are colored red or black (0 or 1)
  - The root must be black
  - All children of a red mode must be black
  - Any path from the root to an empty node must have the same number of black nodes
  - Length: One sub-tree must not be greater than twice the other.



Balanced trees - BB(α) tree

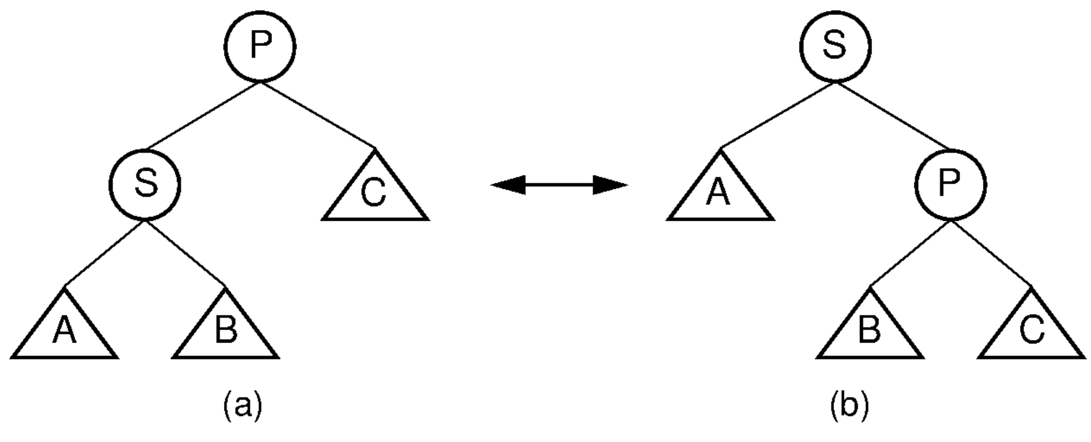
- BB(α) trees ($0 < \alpha \leq 1/3$) use weight balancing
  - Neither side has less than a proportion α of the empty nodes, i.e., both proportions fall in [α,1- α]



BB(5/10) trees

Balanced trees - Splay tree

- Splay tree falls into an average cost $\Theta(\log n)$ of per access operation.
  - Access nodes could be rotated or *splayed* to the root of the tree.



Splay tree single rotation

3. Spatial data structures

- Searching on a one-dimensional key
  - BST, AVL tree, splay tree, 2-3 tree, B-tree, tries
- Searching on multi-dimensional key
  - Requires the use of spatial data structure
- Spatial data structure store data objects in two or more dimensions
  - widely used in geographic information systems, computer graphics, robitics, etc.

3. Spatial data struectures (cont'd)

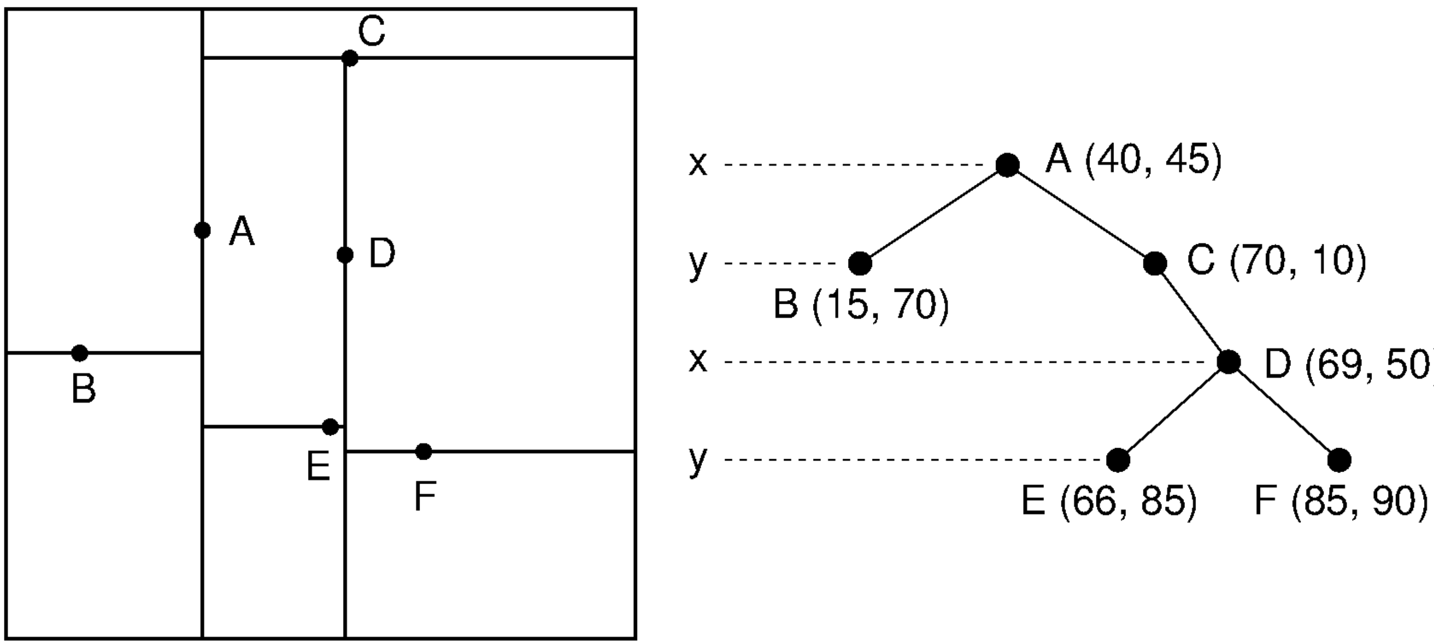
- Two typical Spatial Data Structures
  - K-D tree
  - PR quadtree

K-D tree

- K-D tree is an extension of the BST to multiple dimensions.
  - It is a binary tree whose spliting decisions alternate among the key dimensions.
  - Like the BST, the K-D tree uses object space decomposition.

K-D tree (cont'd)

- In a K-D tree, each level makes branching decisions based on a particular search key associated with that level.



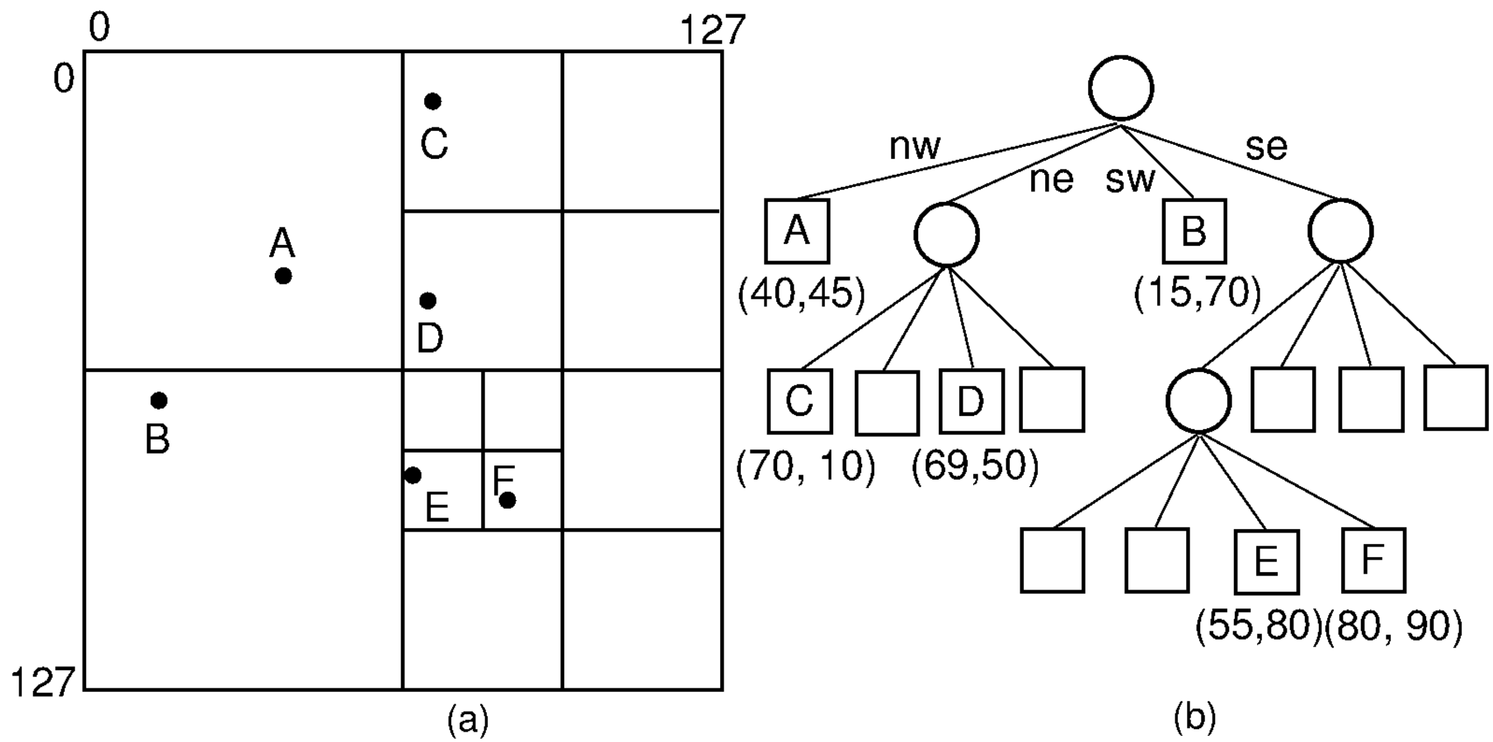
(a) (b)

PR quadtree

- PR (Point-Region) quadtree is a form of trie.
  - It uses key space decomposition.
  - It is a binary tree only for one-dimensional keys (in which case it is a trie with a binary alphabet).
  - For d dimensions it has 2^d branches. Thus, in two dimensions, the PR quadtree has four branches (hence the name "quadtree"), spliting space into four equal-sized quadtrants at each branch.

PR quadtree (cont'd)

- In a PR quadtree, each node either has exactly four children or is a leaf.
  - A full four-way branching (4-ary) tree in shape.



Summary

- Indexing
  - Tree-based index
    - B trees, B^+ trees,
- Advanced Trees
  - Tries
  - Balanced trees
    - AVL tree, Red-black tree, BB(α) tree, Splay tree
  - Spatial data structures
    - K-D tree, PR quadtree