Balanced Search Trees (cont):

B-Trees

Red-Black Trees

Balanced Search Trees

- Balanced Multiway Trees
  - B-Trees
  - Storing trees in files. Applications of B-Trees
- Balanced Binary Search Trees
  - Red-Black Trees

Multiway Balanced Search Trees

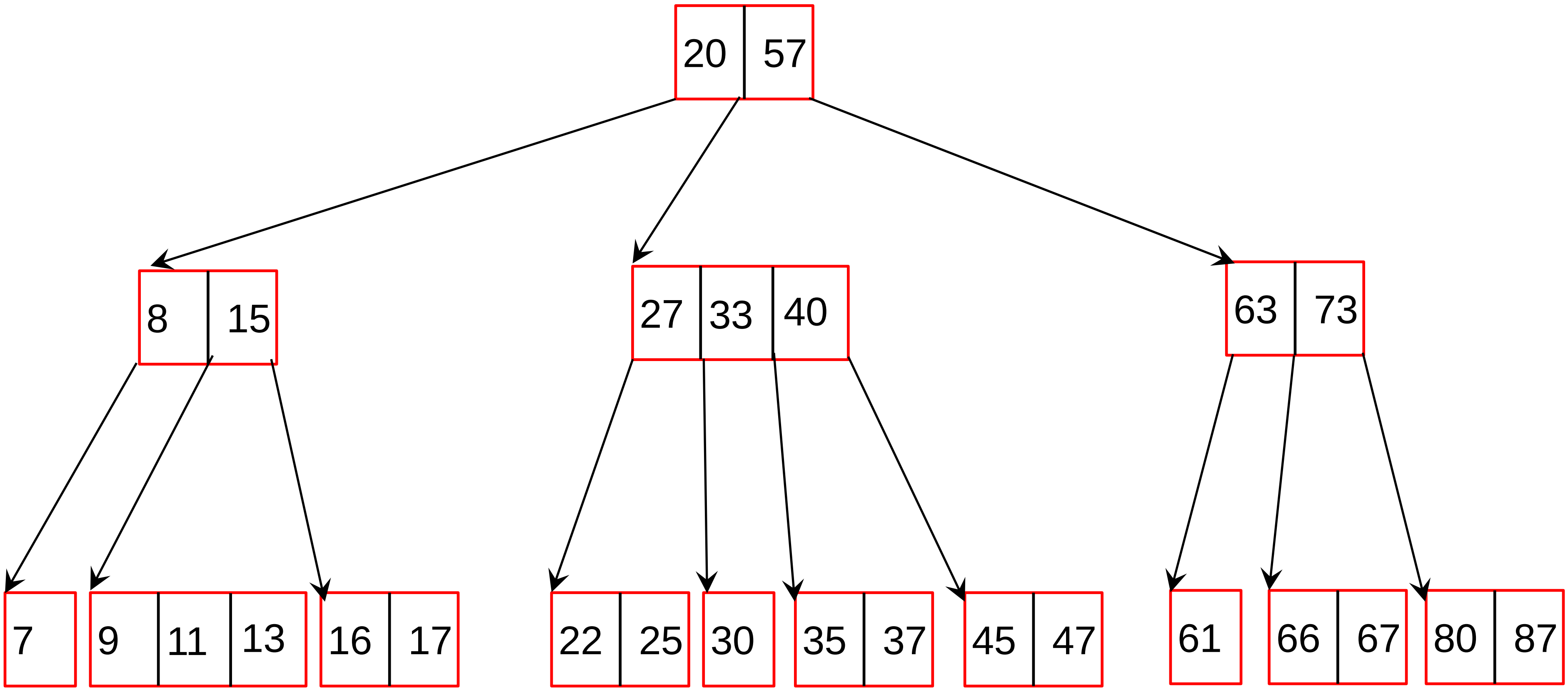
B-Trees *(Bayer & McCreigh, 1971)*

B-Trees

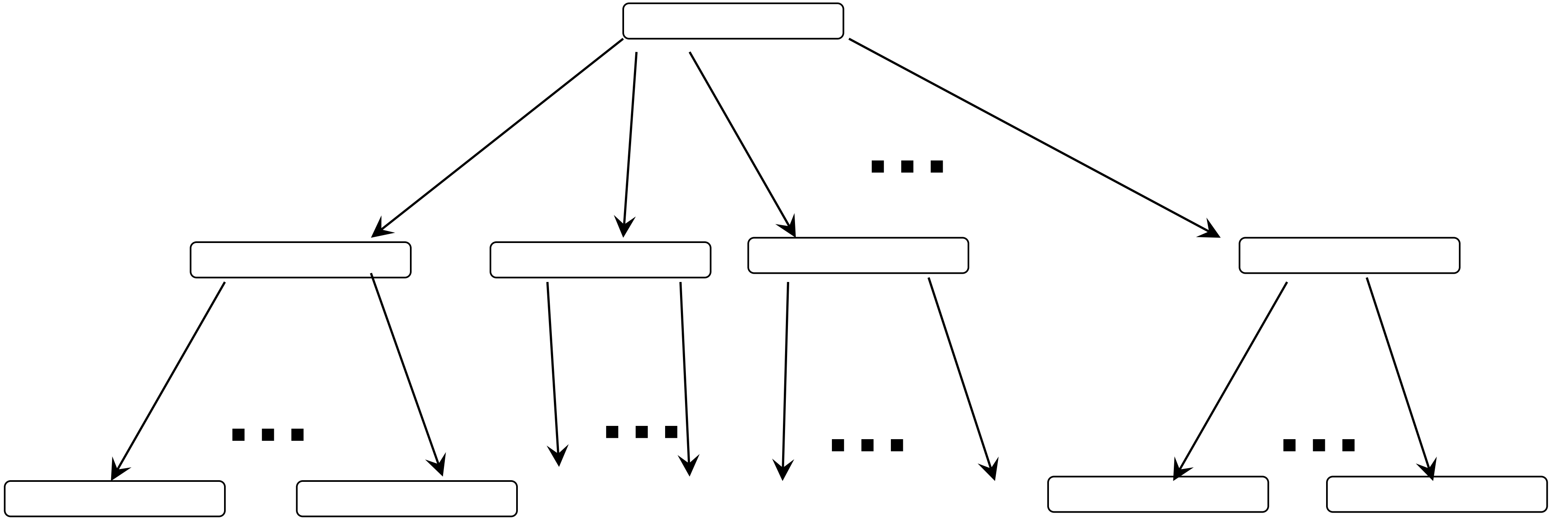
- A generalization of Balanced Binary Search Trees:
  - It is a *multiway* (*multipath*) search tree: each node stores several keys in sorted order and has several children
    - A node contains ***n* keys** and has ***n+1* children** (nodes are also called *pages)*
  - All leaf nodes are on the same level

- B-Trees appeared as a form of Balanced Search Trees that can be stored in the external storage (disks)

Example: B-Tree as a Generalized Search Tree



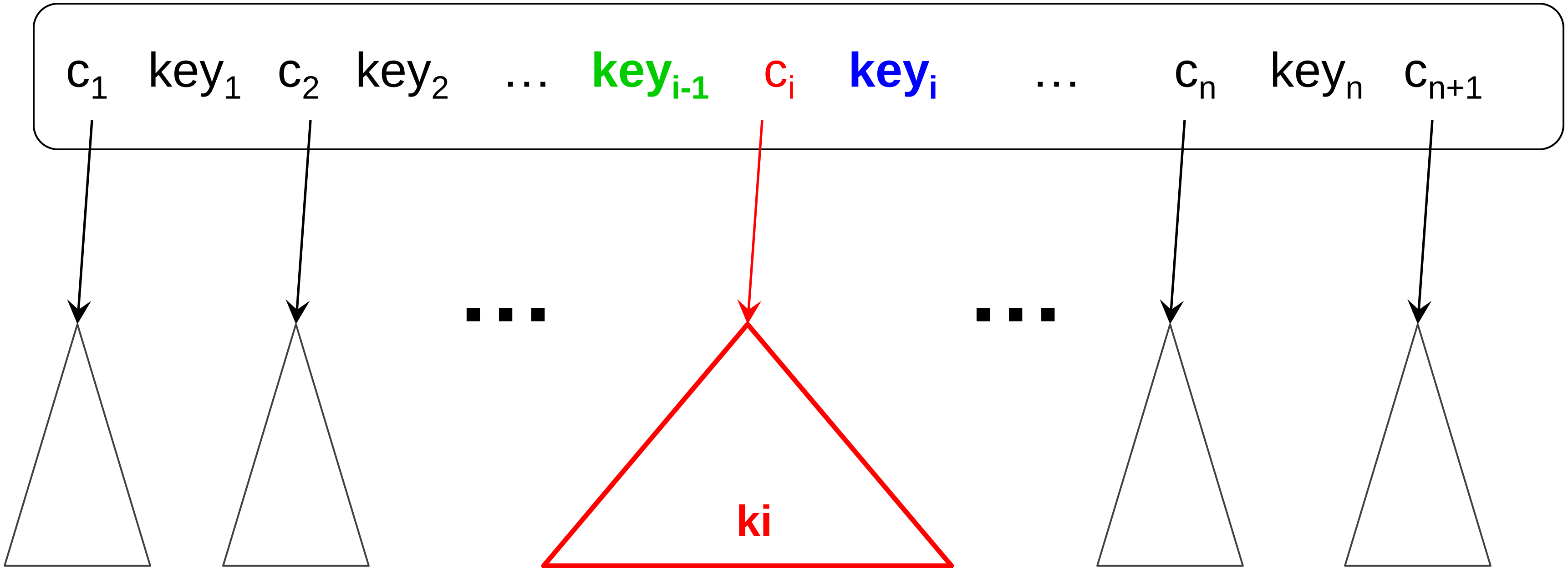
B-Tree General Structure



B-Tree

- Every node has n keys
- The number of children of a node is number of keys plus one (n+1)
- The **number (n)** of keys / (n+1) of children in a node must obey some restrictions:
  - We define **t = the minimum degree of a B-tree**
  - **Restriction:** The **number of children** of any internal node is allowed to vary between **t<=n+1<=2t**
    - The **number of keys** in any internal node can be **t-1<=n<=2t-1**
    - **Exception: root** can have less than t children (root is allowed to have at least 1 key and 2 children)
  - Note: different textbooks may use the concept of branching factor m instead of min degree t
- All **leafs must be at the same depth** (B-tree is always balanced)
- The **values of the keys** in B-tree nodes:
  - **Restrictions** in order to be a ***Search Tree*** -> see next slide details

B-Tree as Search Tree



The keys in a node (page) are sorted:

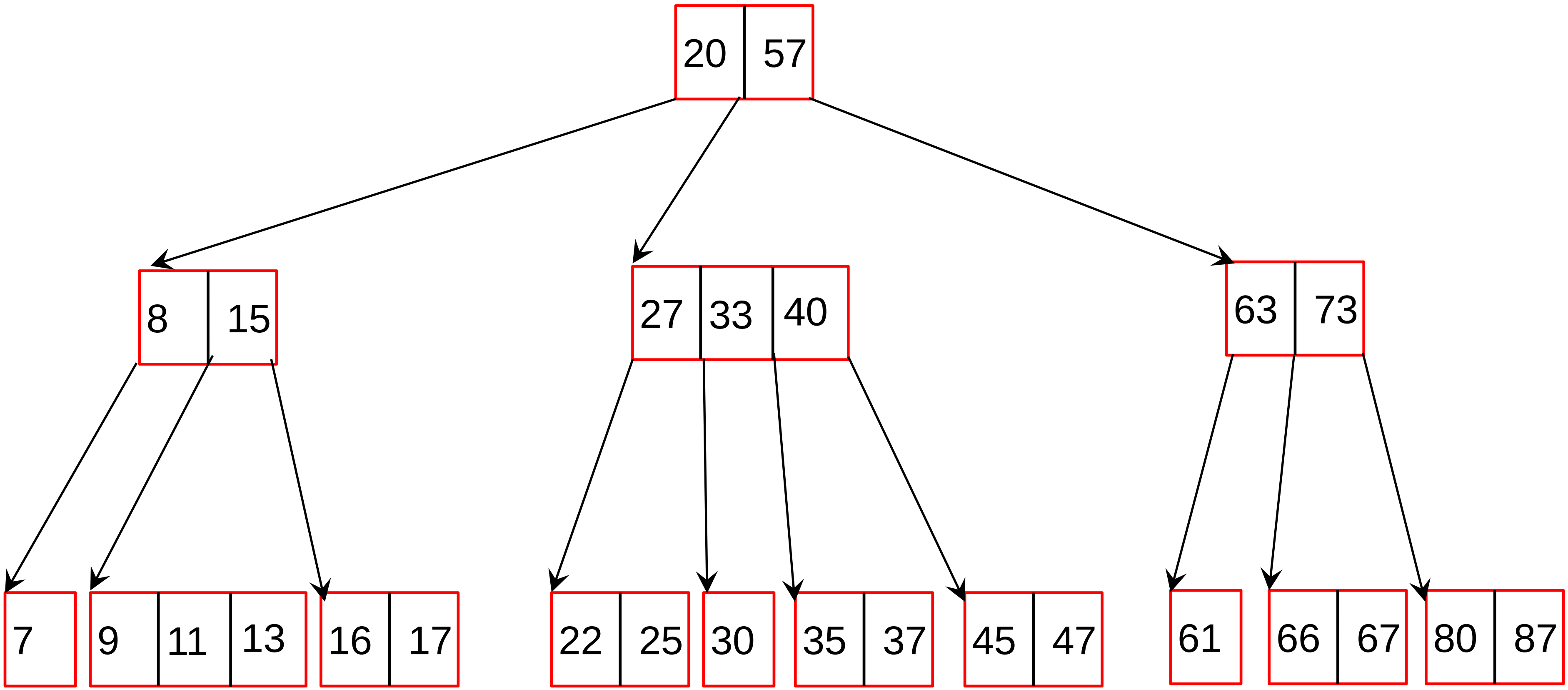
$key_1 < key_2 < \ldots < key_{i-1} < key_i < \ldots < key_n$

For **any key k_i** stored in the subtree with root c_i we have

$key_{i-1} < ki < key_i$

Example: B-Tree with t=2

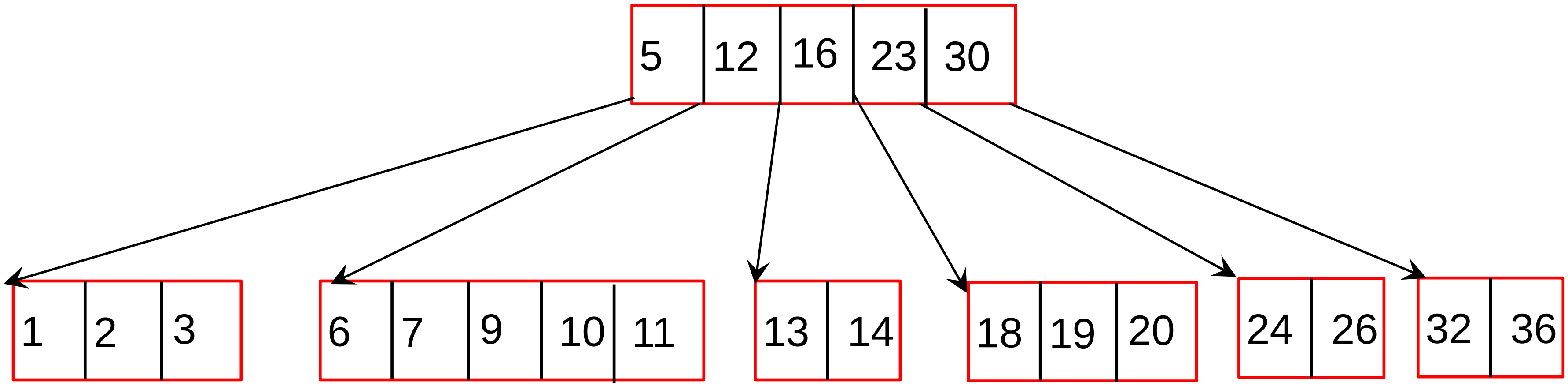
Number of keys in a node: t-1<=n<=2*t-1 => between 1 and 3 keys
Number of children of a node: t<=n+1<=2*t => between 2 and 4 children



Example: B-Tree with t=3

Number of keys in a node: t-1<=n<=2*t-1 => between 2 and 5 keys
(except the root, which can have less keys (1) if needed)
Number of children of a node: t<=n+1<=2*t => between 3 and 6 children

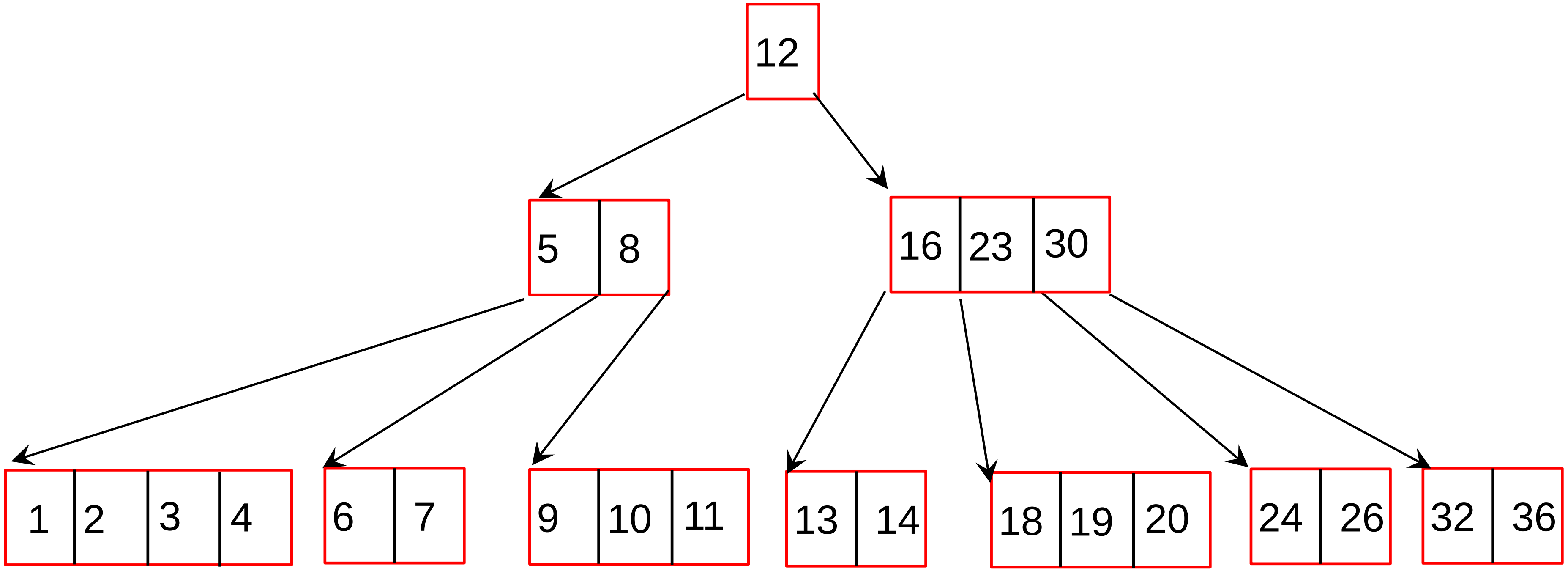
In this example, the tree has only 2 levels (height 2) but it could have any number of levels.



Example: B-Tree with t=3

Number of keys in a node: t-1<=n<=2*t-1 => between 2 and 5 keys
(except the root, which can have less keys (1) if needed)
Number of children of a node: t<=n+1<=2*t => between 3 and 6 children

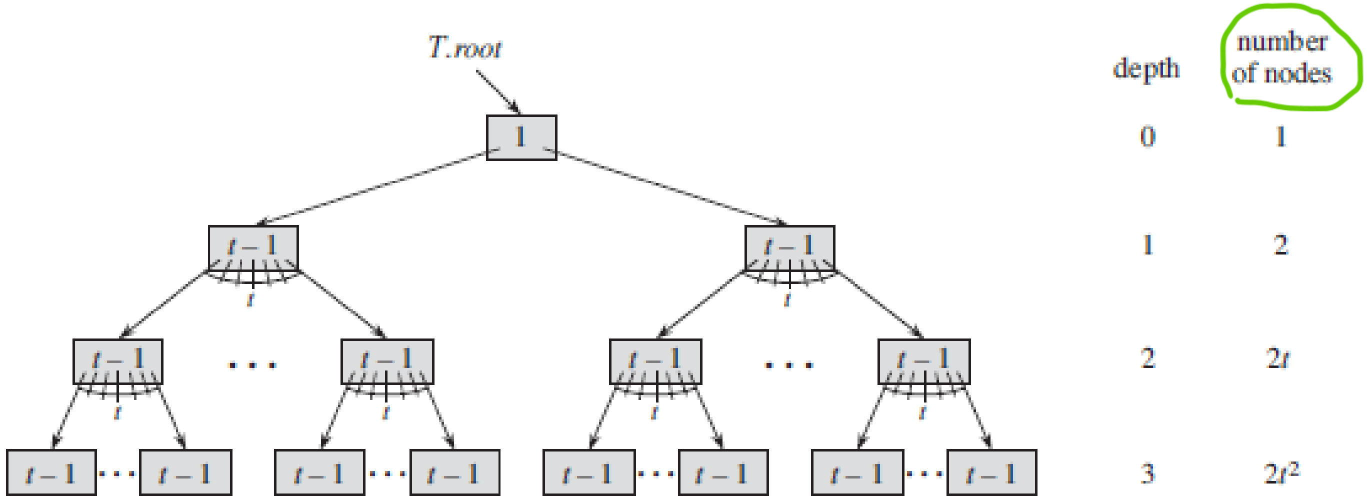
In this example, the root has only 1 key (allowed exception for root)



Height of B-Trees

- B-Tree of mindegree t and height h and ***n****= the minimum possible number of* ***keys***: 1 key in root and every other node has t-1 keys

$$n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2 \cdot t^{i-1} = 2 \cdot t^h - 1 \quad \rightarrow \quad \boxed{h \leq \log_t \frac{n+1}{2}}$$

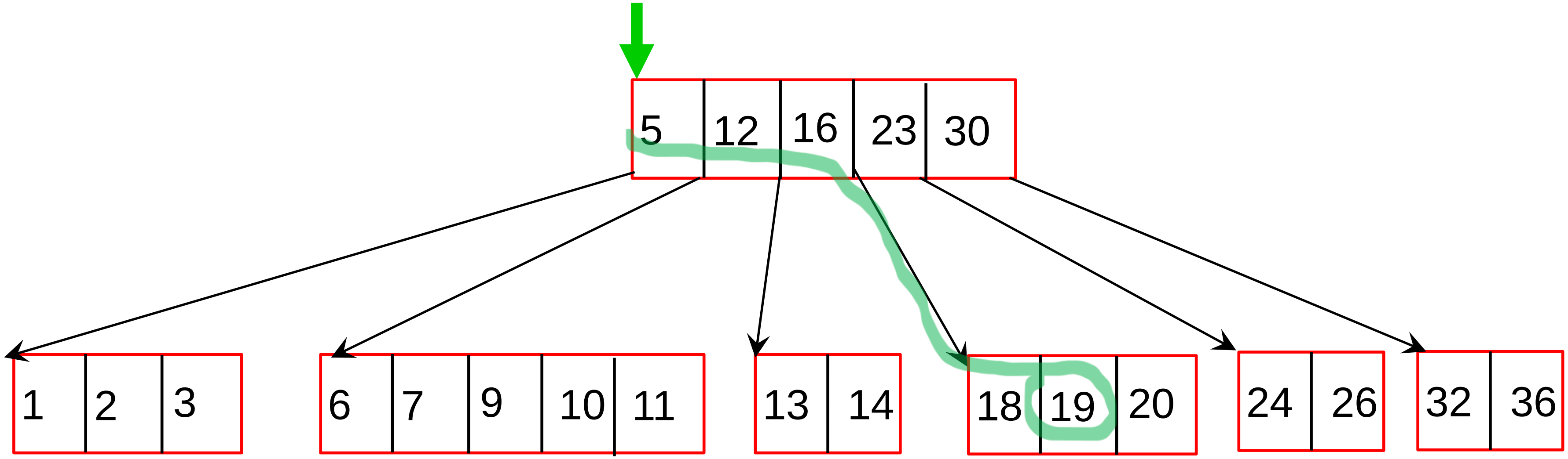


B-Tree Datastructure

```
public class IntegerBTree {

    private int T; // the mindegree of the B-Tree

    private class BTreeNode {
        int n;    // current number of keys contained in node
        Integer key[] = new Integer[2 * T - 1];   //maximum 2T-1 keys
        BTreeNode child[] = new BTreeNode[2 * T]; // maximum 2T children
        boolean leaf = true;

}

public IntegerBTree(int t) {
        T = t;
        root = new BTreeNode();
        root.n = 0;
        root.leaf = true;
    }

    private BTreeNode root; // root of tree

//…
}
```

Simple example of Btree with Integer keys.
Keys could be any comparable type.
Values could also be associated.

Example: B-Tree Search



The tree is traversed from top to bottom, starting at the root. At each level, the search chooses the child pointer (subtree) which is between two key values that frame the searched value. Binary search can be used within each node.

Example: if we search for key=19, starting at the root 16<19<23, thus we continue the search in the subtree rooted in child 4.

```java
public boolean contains(Integer key){
    BTreeNode found= search(root, key);
    if (found == null)
        return false;
    else return true;
}
```

$$O(t \cdot \log_t n)$$

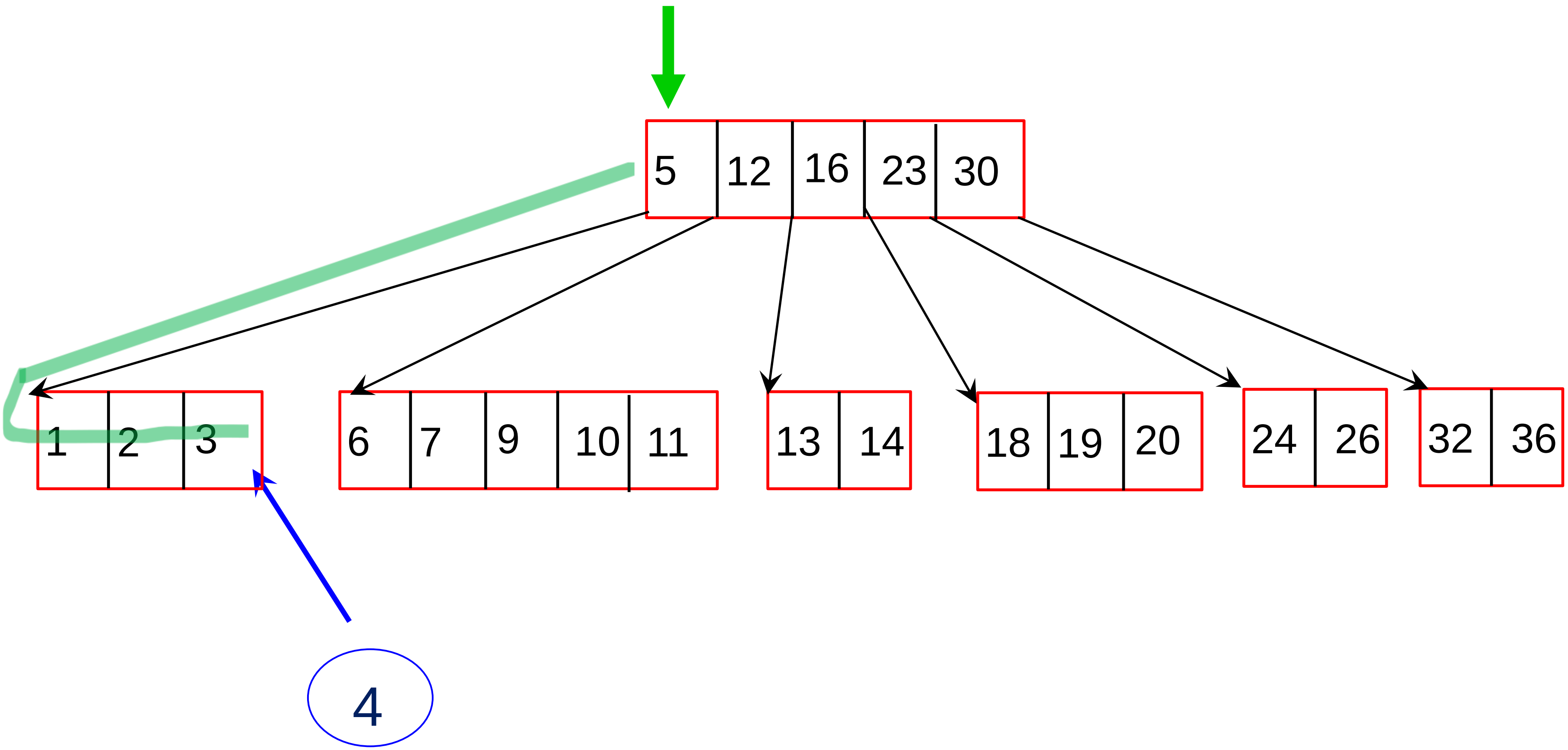
```java
private BTreeNode search(BTreeNode x, Integer key) {
    int i = 0;
    if (x == null)
        return x;   // tree is null

    for (i = 0; (i < x.n) && (key > x.key[i]); i++) ;

    if ((i < x.n) && (key == x.key[i])) {
        return x;   //  found key in root node x
    }
    if (x.leaf)
        return null;  // not found, and current node has no children

    return search(x.child[i], key);  // recursively search in coresponding child
}
```

B-Tree Insert

- To insert new key K:
- New keys are always inserted into leafs
- Perform a search for key K, until a leaf node y is encountered (this is the insertion place)
- Insert key K into node y
- If y is full:
  - Split y around its median key into two nodes
  - Move median key to y’s parent
  - If parent is full, recursively split, all the way to root if necessary
  - If root is full, split root and increase height of tree

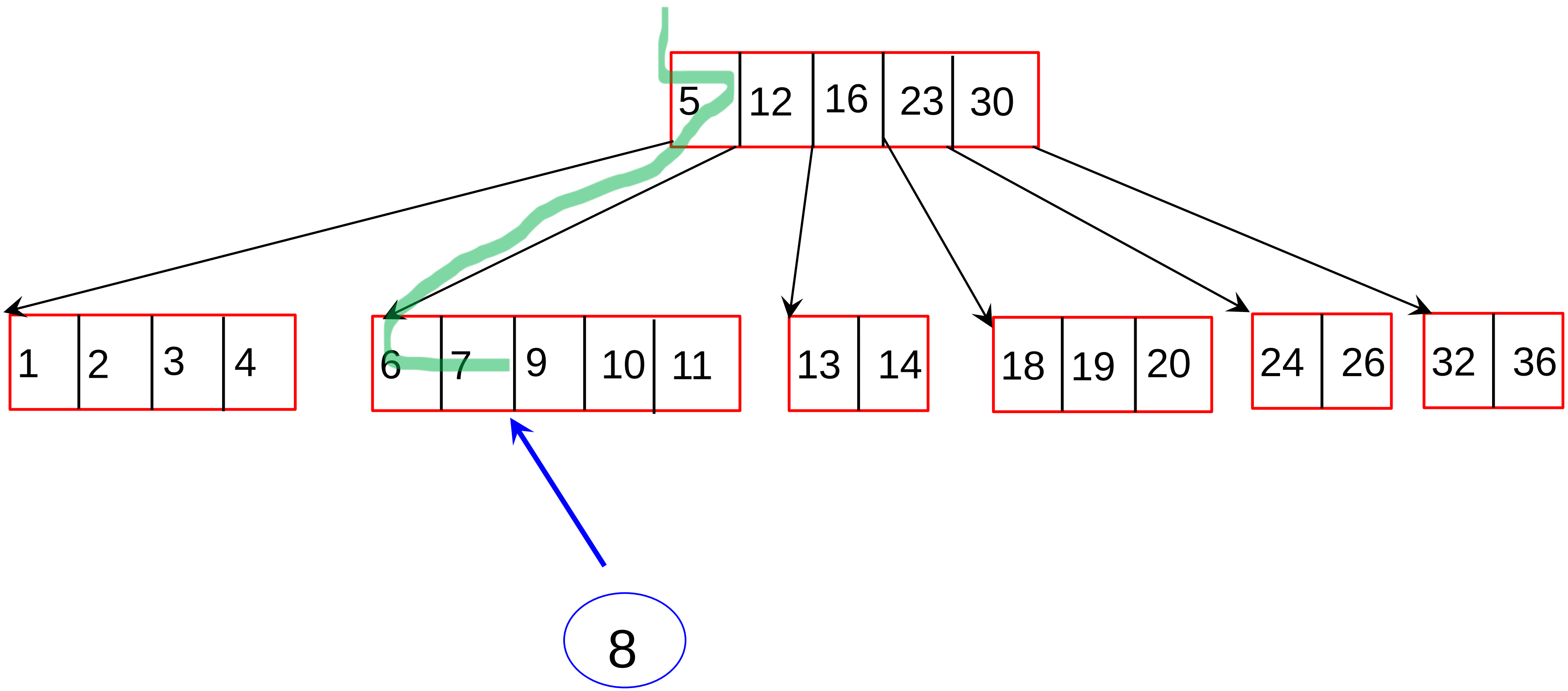
Example: B-Tree Insert

Case 1: the leaf node is not full (less than 2*t-1 keys)

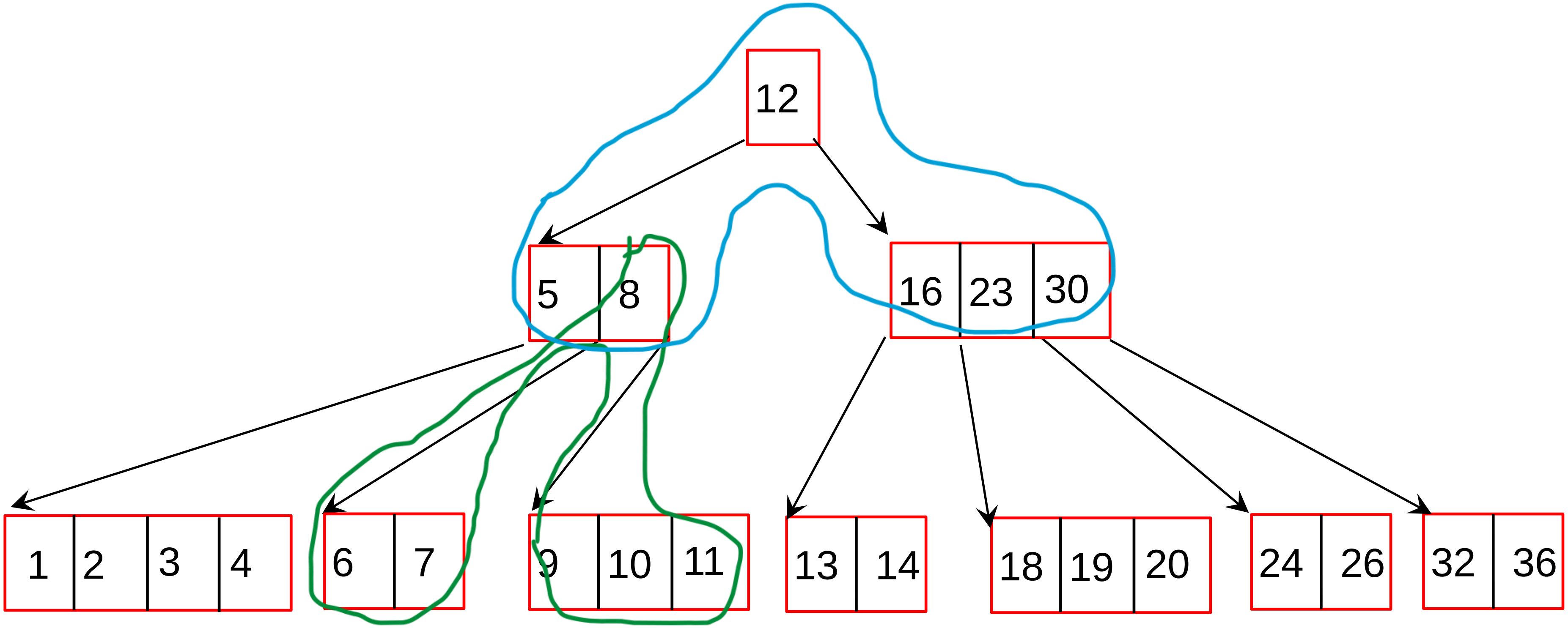


Example: B-Tree Insert

Case 2: the leaf node found is full (has 2*t-1 keys) and must be SPLIT



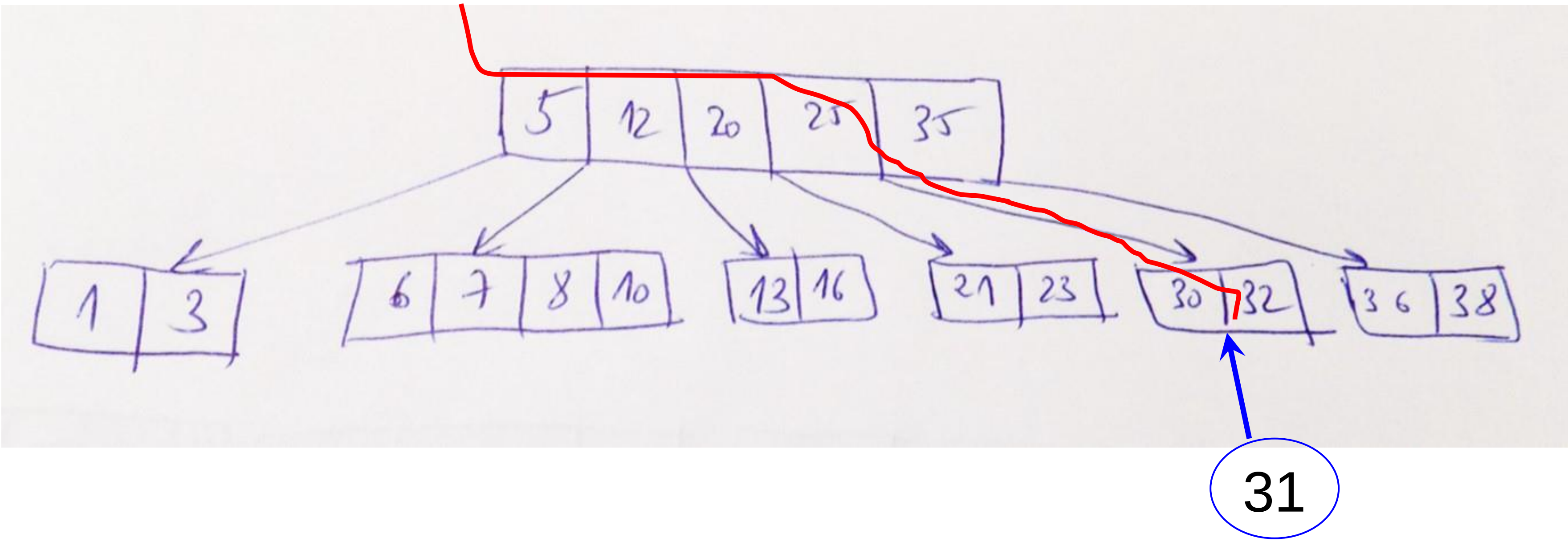
Example: Split nodes after insert



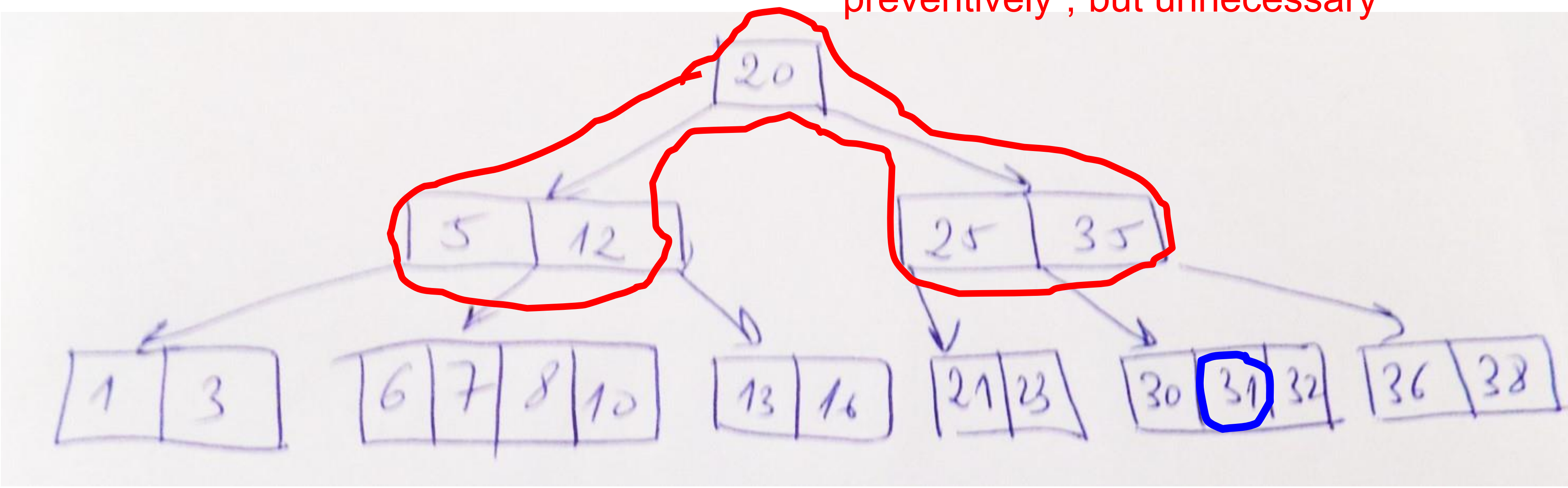
A full node is split into 2 nodes around its median key. The median key moves up to its parent node. If the parent is also full, it will be split as well. **In the worst case we have to split full nodes all the way up to the root** of tree and the tree increases Its height, getting a new root.

B-Tree Efficient Insert

- Basic idea: avoid having to “climb up the tree” in order to do splits propagated up to the root
- Insertion must happen in a single pass down the tree from the root to a leaf.
- To insert new key K:
- Perform a search for key K, until a leaf node y is encountered (this is the insertion place), and every time a full node is encountered on the search path, it is split
  - Thus whenever we want to split a full node y, we are assured that its parent is not full. It is possible that in this way we are doing “unnecessary” root splits.



Efficient Insert: Root node has been split “preventively”, but unnecessary



New key inserted in Non-full leaf

Simple BTree Implementation

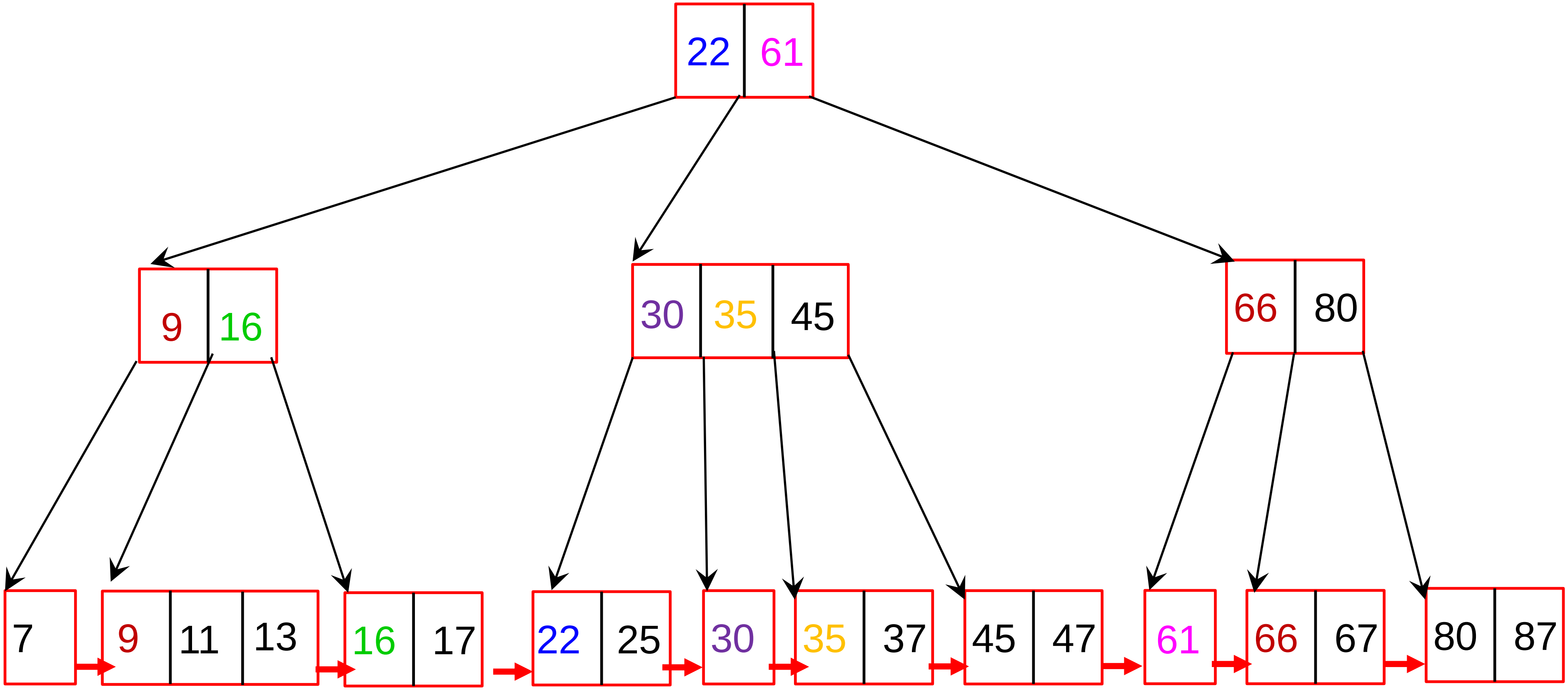
- Source code: IntegerBTree.java

- You do not have to memorize *the implementation* (the code) for the insert operation
- But: operations such as search, inorder, min, max, etc are very reasonable programming exercises!

B-Tree Variations: B+ Trees

- B+ (B plus) Trees:
  - similar with B-Trees but only the leaf nodes hold information (key-value mappings). The internal nodes repeat the key values in order to guide the search only
  - the leaf nodes can be linked in a list (all the keys in sorted order)

Example: B+ Tree



Usage of B Trees

- B-Trees can be used as in-memory datastructures for SortedMap.
  - Their performance is comparable with balanced binary search trees, which are usually preferred due to simplicity
  - **Advantage of B-Trees over BST: *cache friendliness***
- BTreeMap in Rust:
- https://doc.rust-lang.org/std/collections/struct.BTreeMap.html

- The strength of B-Trees relies in using them as **persistent datastructures (tree implemented on disk, not in memory).** *In this form, they are widely used in practice for file systems and database indexes!*

Balanced Search Trees

- Balanced Multiway Trees
  - B-Trees
  - Storing trees in files. Applications of B-Trees.
- Balanced Binary Search Trees
  - Red-Black Trees

Search trees on external storage

- Problem: we need a data structure having features similar to the search trees (efficient *search*, *insert*, *delete, sorted order relations*) but *on external storage (disk, web).*
- *Why needed?*
  - *The amount of data handled is so large that it does not fit into the internal memory at once*

  or

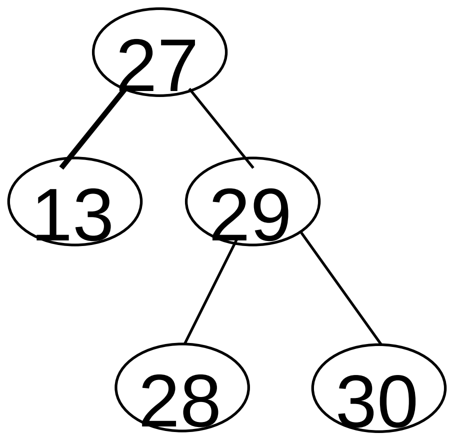
  - *A Persistent data structure is needed (databases implementation, indexing, file systems)*

  - *First idea: store a search tree in a file on disk:*
    - *Generalizes the representation of trees using arrays in memory*

Storing a tree in a file

- A binary file is a sequence of fixed-size records, similar to an array
- One Tree Node is a fixed-size record, containing “key”, “value” and links to its children.
- In-memory representations of trees: links to children can be:
  - References to Node objects (Java)
  - Pointers to Node structures (C)
- Representation of trees in binary files:
  - Each record of the file is the equivalent of a tree node
  - One single file stores many tree nodes
  - Links to children are here offset values from the beginning of the file. (address of a record inside the file)

BST as a sequence of records



rootoffset=4

offset	key	val	left	right	free
0					1
1	13		-1	-1	0
2	29		3	6	0
3	28		-1	-1	0
4	27		1	2	0
5					1
6	30		-1	-1	0
7					1
					1
					1
					1

Search key 30:

1. Fseek offset 4 (go to root)
2. Read node
3. Fseek offset 2
4. Read node
5. Fseek offset 6
6. Read node - found

Storing trees in files

- Binary trees are not suitable for representation on disk because:
  - going from a current node to a child node implies executing a random access positioning operation in the file (fseek)
    - **Random access positioning(fseek)** is time consuming, searching a key should require a minimum number of random access operations -> we must **reduce height** from log2(n) to something smaller
  - every access reads only a small block of data (node containing one key):
    - efficient disk access must **read bigger blocks of data** (pages) in bulk to benefit from **buffering**
- For representation on disk we need trees with a small height and several keys per node (page) -> B-Trees

B-Trees stored in files

- Operations on B-Trees on disk implement the same algorithms as in memory, with one particularity:
  - *following a child link means doing a random access (fseek) operation (the equivalent for following a pointer or a reference to a node) and then loading the whole node in memory in a single read operation*

Real-life applications of B-Trees

- B-Trees (most often their version B+Trees) are used in database systems either for:
  - for ***table data*** sorted according to ***primary key***
  - for ***index tables*** sorted according to some specific columns

- **Examples:**
- **MySQL+InnoDB storage engine** https://dev.mysql.com/doc/refman/8.4/en/innodb-physical-structure.html
- **Oracle databases** https://docs.oracle.com/en/database/oracle/oracle-database/23/cncpt/indexes-and-index-organized-tables.html
- **PostgreSQL** https://www.postgresql.org/docs/current/btree.html
- **MongoDB** https://www.mongodb.com/docs/manual/indexes/

DB Indexing

- Indexes allow to find all rows with a given attribute value without reading and checking every row.
- A typical index in a DBMS: A sorted list of all values for a specific column together with references to the rows that contain the respective value
- The index is a Sorted Map (key = column data, value = row number in table data)
- Table data is not sorted, only index is sorted

Example: Customer Table (CustID is primary key)

- CREATE INDEX customer_idx1 ON customer (name)
- CREATE INDEX customer_idx2 ON customer (card)

Customer_idx1

Name	**Ref**
Ann	2
John	1
Peter	4
Sue	3

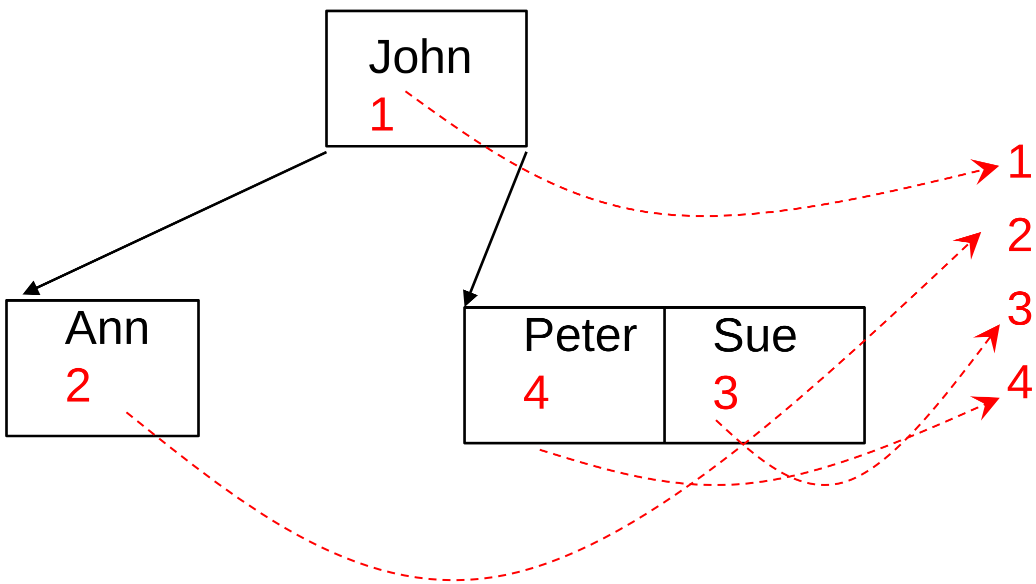
Customer Table

	CustID [PK]	**Name**	**Addres**	**Age**	**Card**
1	CID1	John	TM	19	246
2	CID2	Ann	NY	50	192
3	CID3	Sue	AR	21	145
4	CID4	Peter	CJ	44	234

DB Indexing with BTrees

- The index can be in memory, but it can be made persistent
- B trees or B+trees are used for persistent indexing

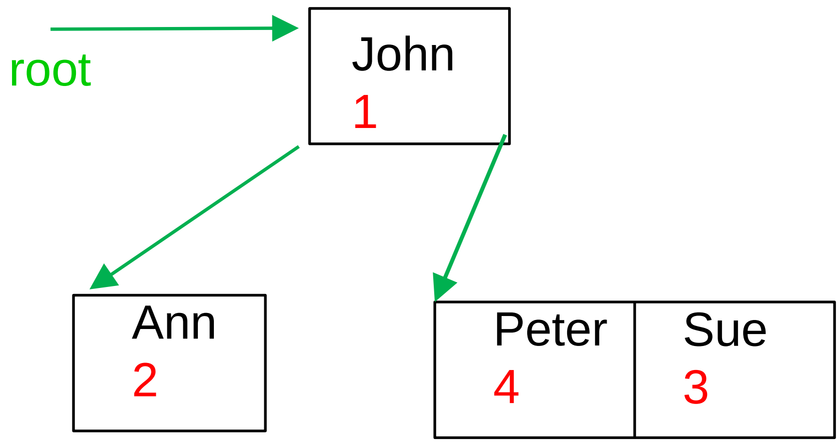
Example:
Index with B Tree with t=2



Customer Table

CustID [PK]	Name	Addres	Age	Car
CID1	John	TM	19	246
CID2	Ann	NY	50	192
CID3	Sue	AR	21	145
CID4	Peter	CJ	44	234

Example:
Index with B Tree with t=2 stored on disk



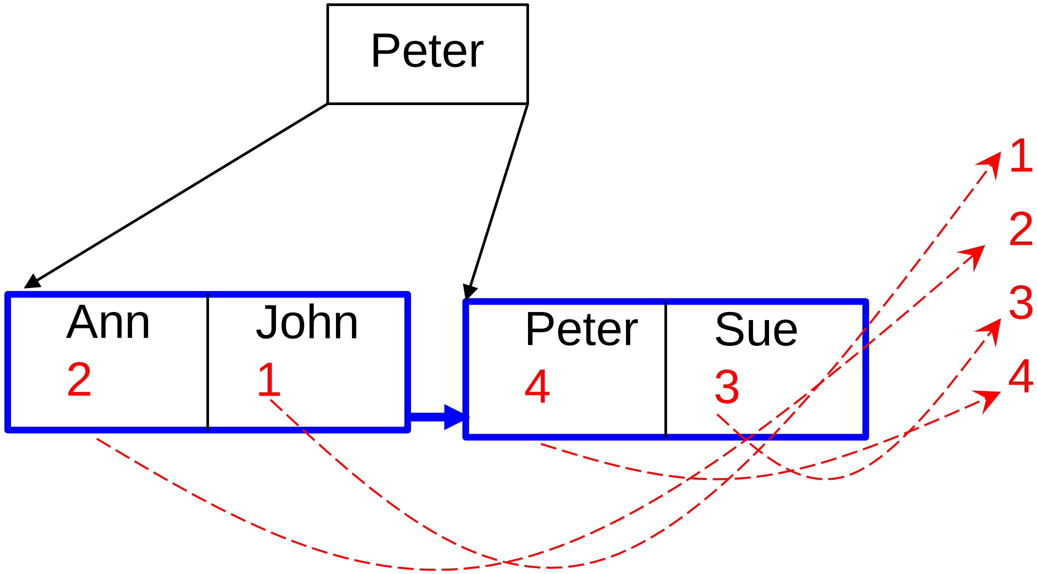
Mindegree t=2
Each node can have:
Keys: 1-3
Children: 2-4

Root = 3

	n	le af	c1	k1	v1	c2	k2	v2	c3	k3	v3	c4
1												
2	2	T	-1	Peter	4	-1	Sue	3	-1			
3	1	F	6	John	1	2						
4												
5												
6	1	T	-1	Ann	2	-1						
7												

DB Indexing with B+ Trees

Index with B+ Tree



Customer Table

CustID [PK]	**Name**	**Addres**	**Age**	**Card**
CID1	John	TM	19	246
CID2	Ann	NY	50	192
CID3	Sue	AR	21	145
CID4	Peter	CJ	44	234

Balanced Search Trees

- Balanced Multiway Trees
  - B-Trees
  - Storing trees in files
- Balanced Binary Search Trees
  - Red-Black Trees

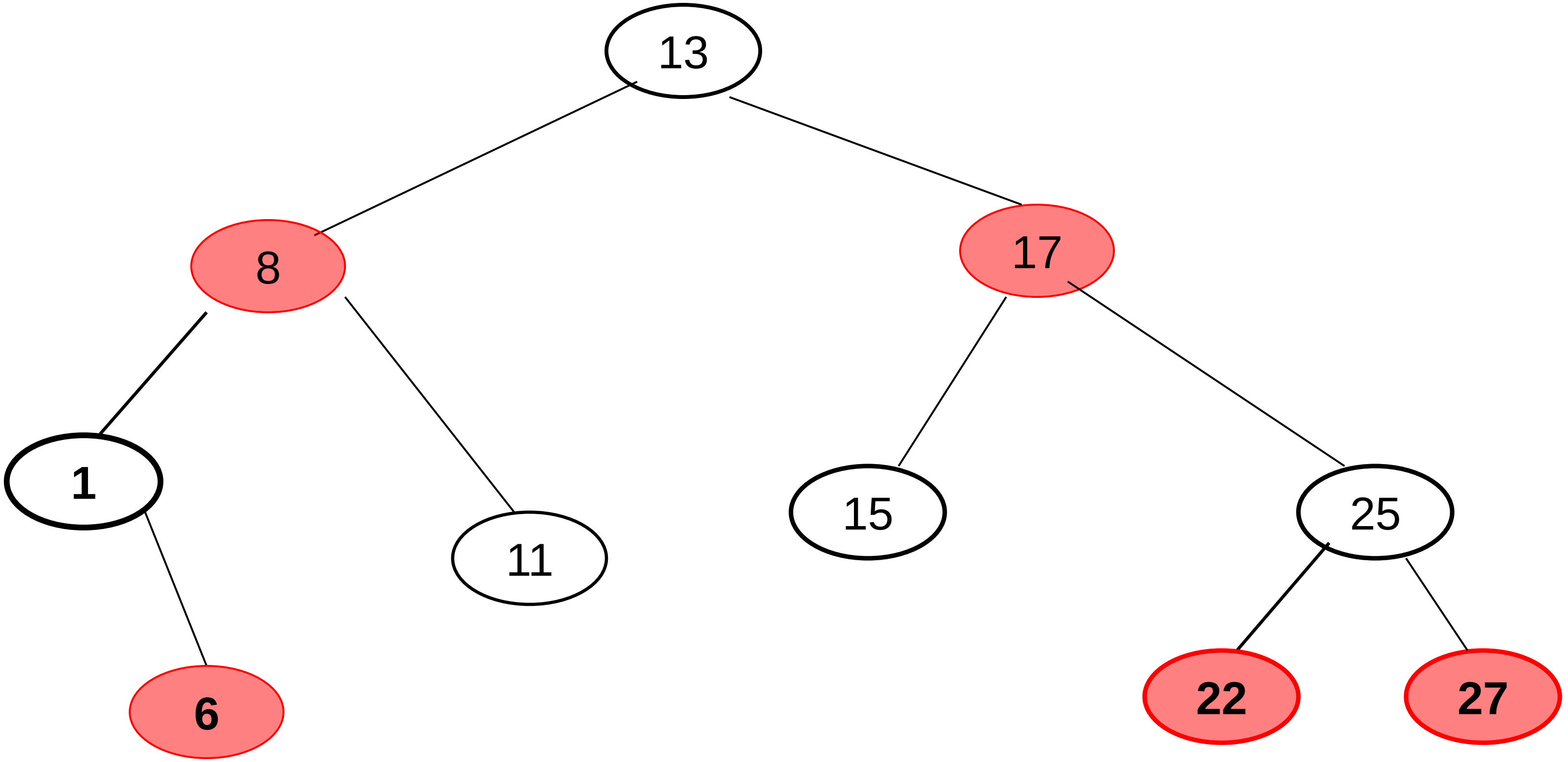
Red-Black Trees

- Another type of BST
- Doing less rotations at delete (compared to AVL)
- Conceptually derived from a case of B-trees

Red-black Trees

- A ***red-black tree*** is a binary search tree with one additional attribute per node: its ***color***, which can be either red or black. It is defined by following rules:
  1. Every node is either red or black.
  2. The root is black.
  3. T.*nil* is black.
  4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

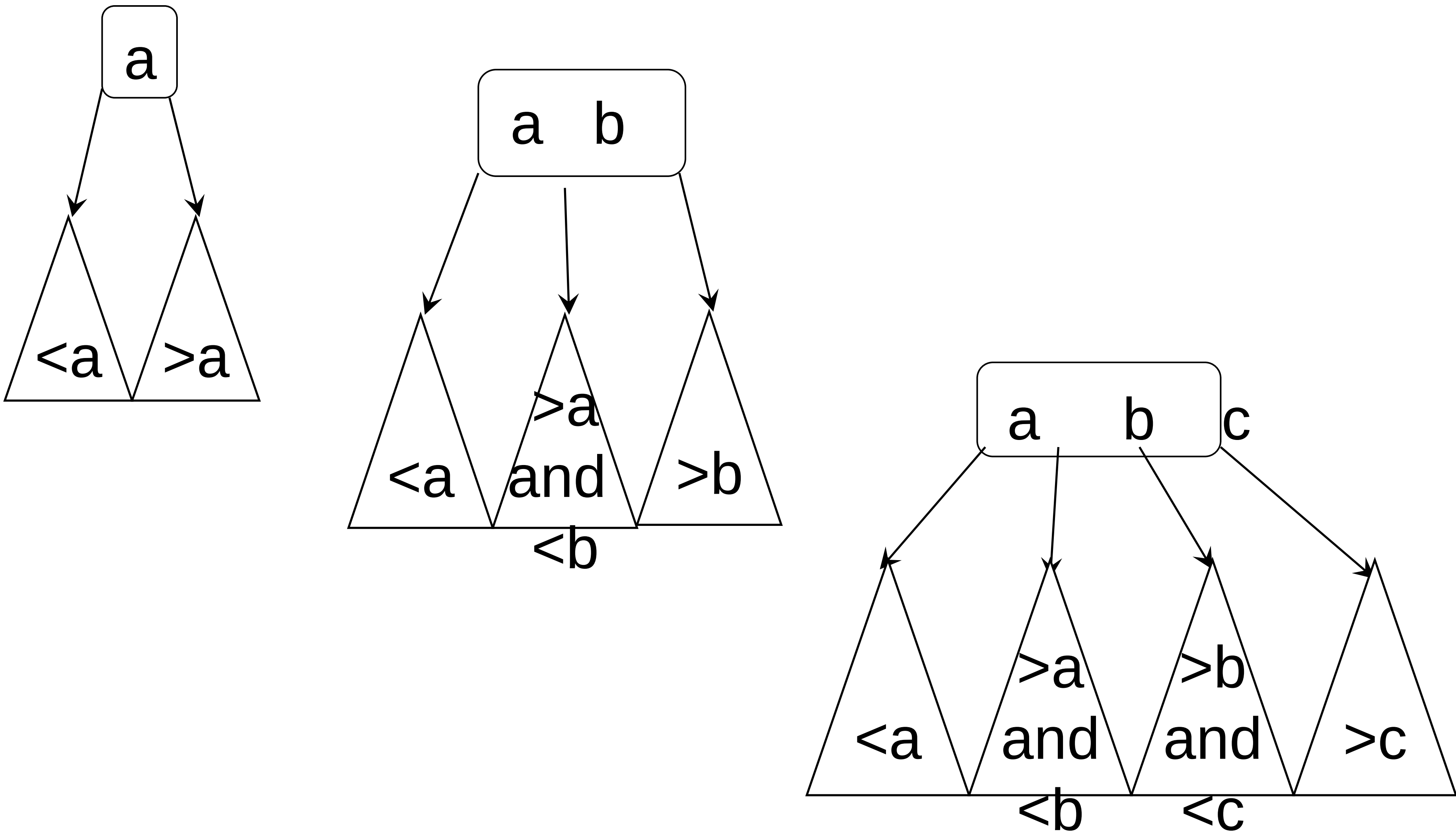
Red-Black Tree



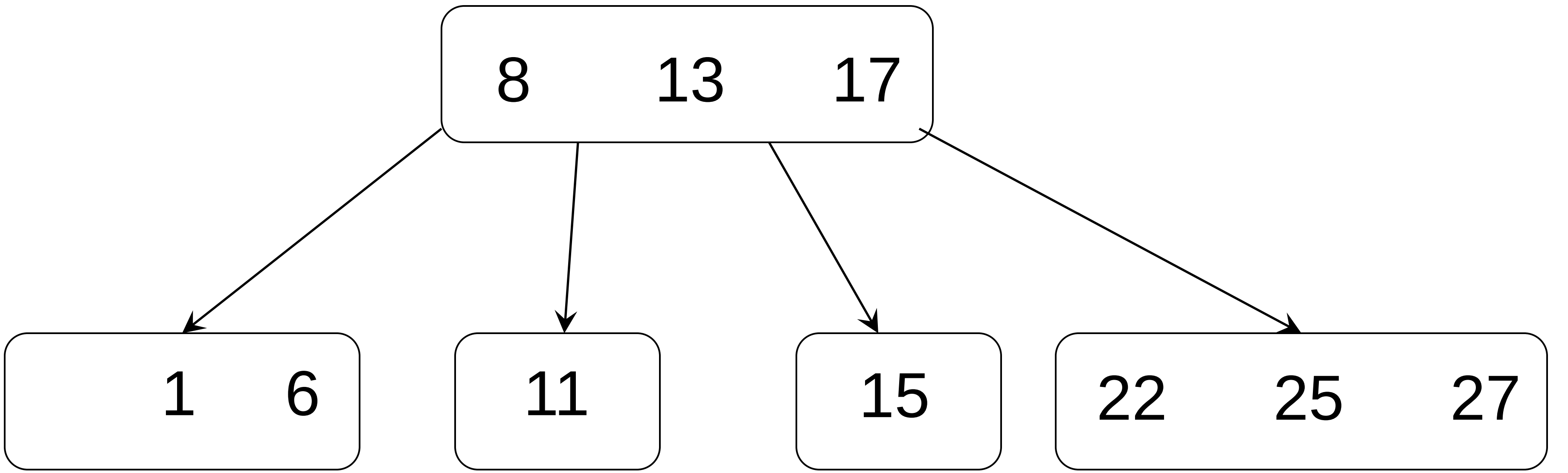
Red-Black Trees are 2-3-4 Trees

- Idea: Construct binary trees as a particular case of B-trees
- 2-3-4 Trees: actually B-trees of min degree 2
  - Nodes may contain 1, 2 or 3 keys
  - Nodes will have, accordingly, 2, 3 or 4 children
  - All leaves are at the same level

2-3-4 Trees Nodes



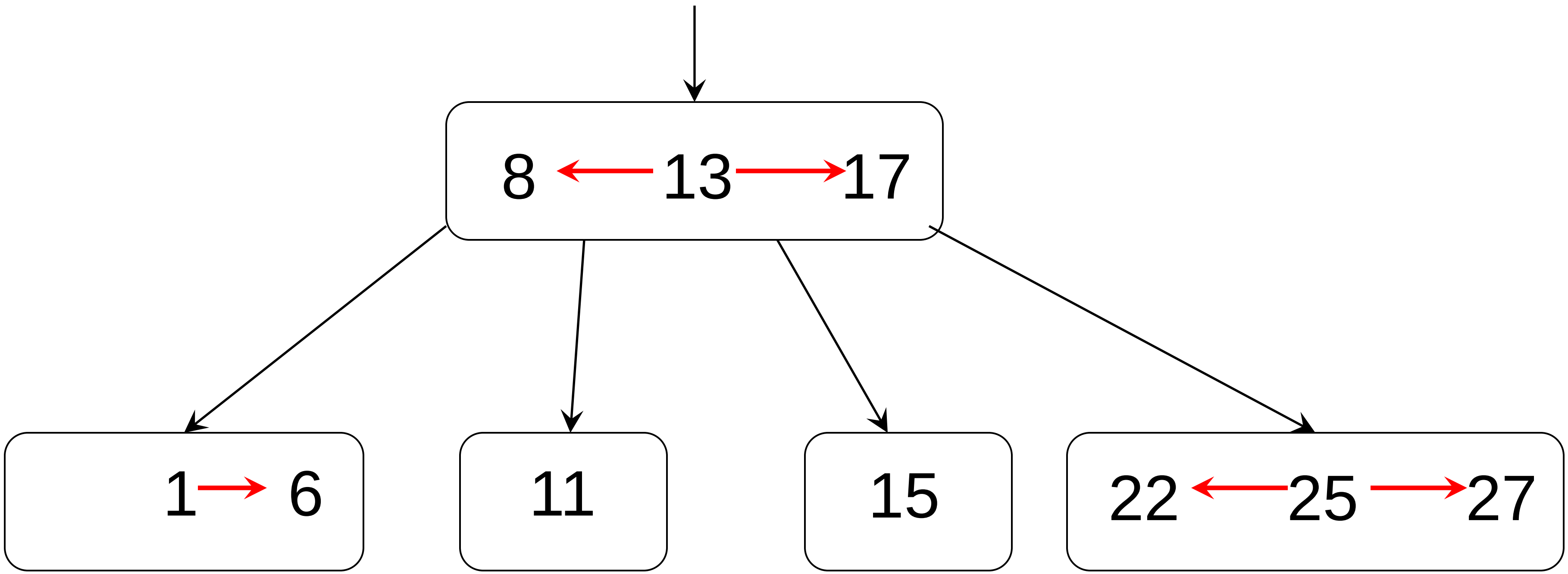
Example: 2-3-4 Tree



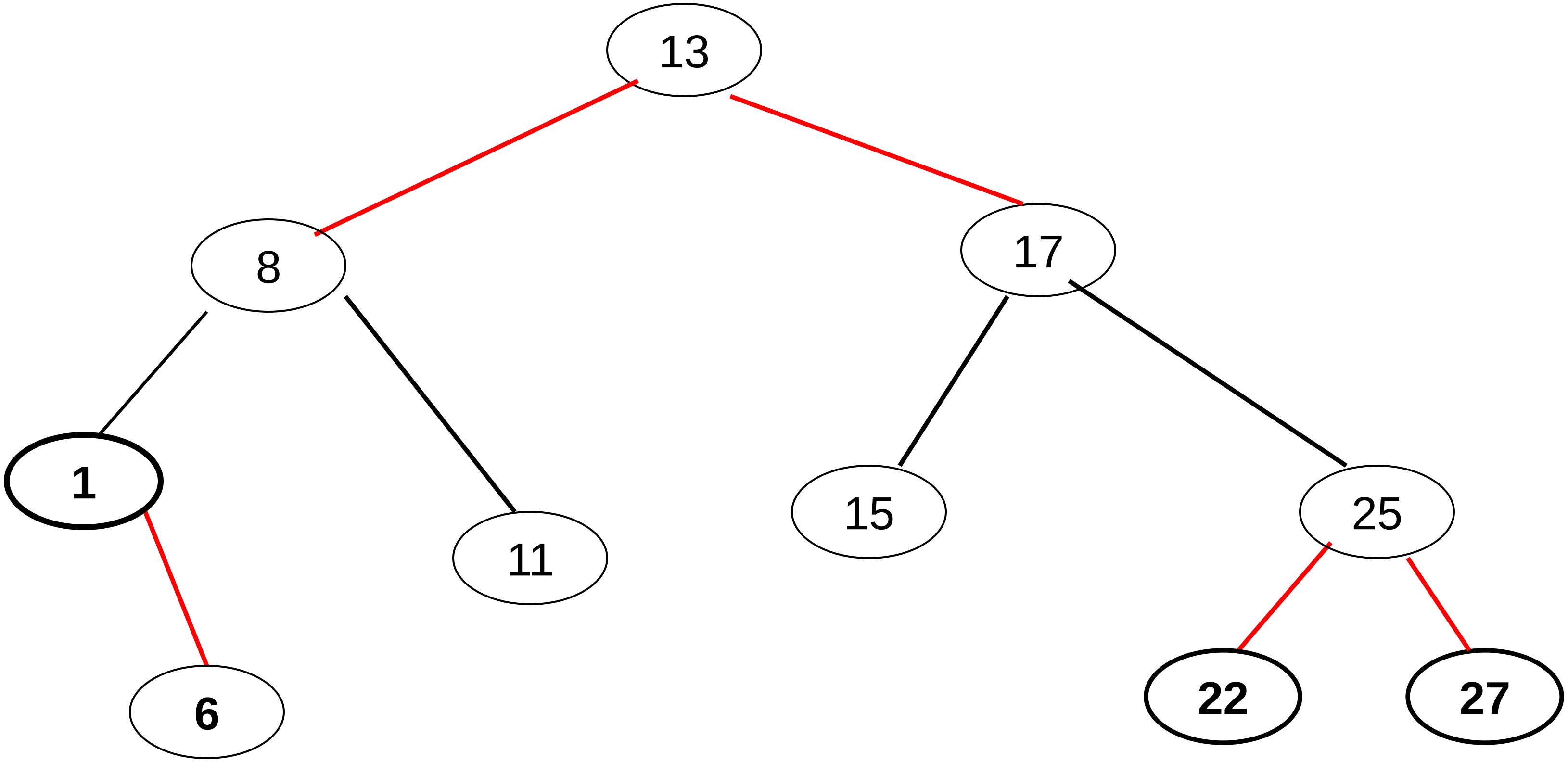
Transforming a 2-3-4 Tree into a Binary Search Tree

- A 2-3-4 tree can be transformed into a Binary Search tree (called also a Red-Black Tree):
  - *Nodes containing 2 keys will be transformed in 2 BST nodes, by adding a red ("horizontal") link between the 2 keys*
  - *Nodes containing 3 keys will be transformed in 3 BST nodes, by adding two red ("horizontal") links originating at the middle keys*

Example: 2-3-4 Tree into Red-Black Tree

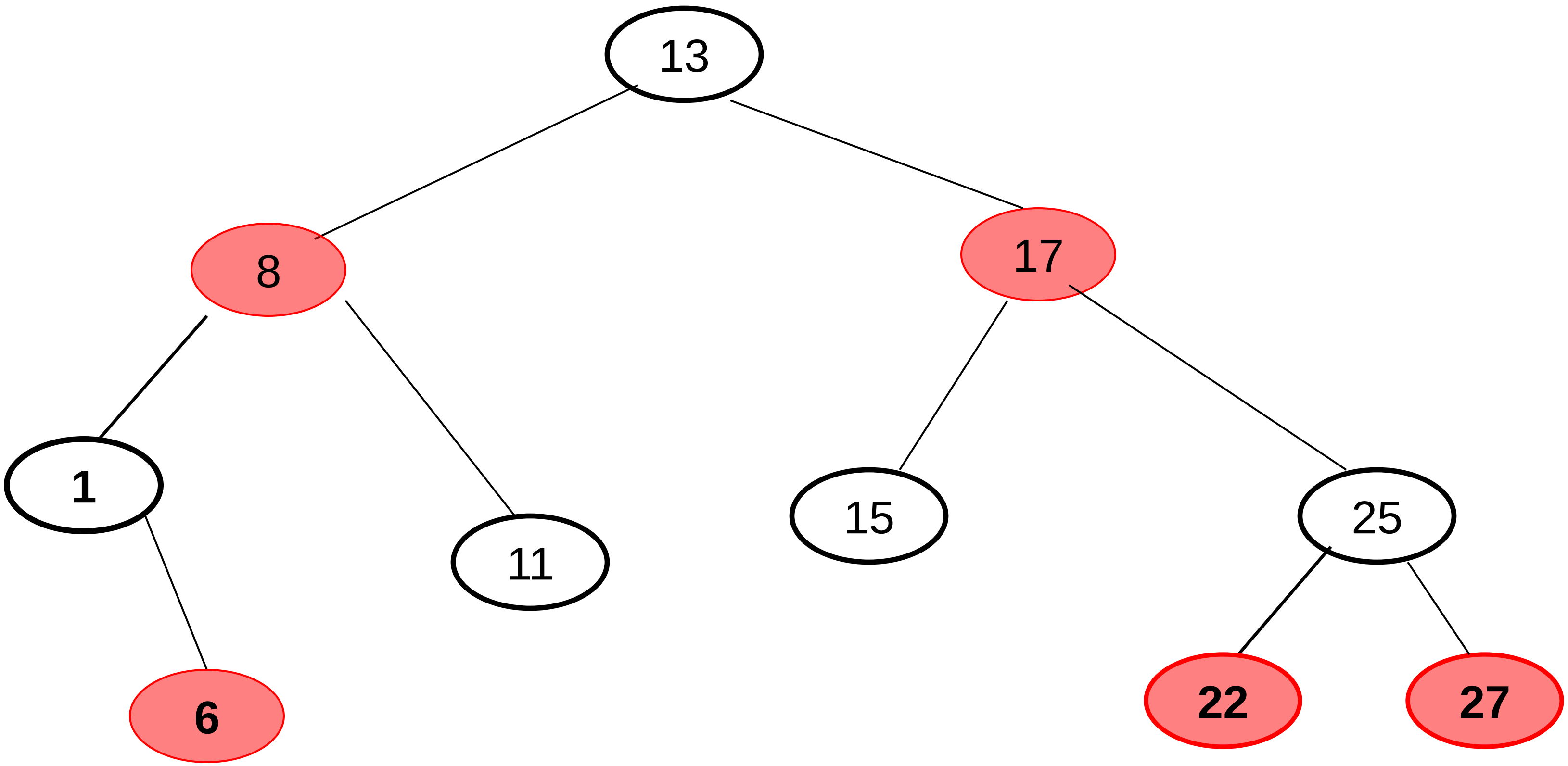


Example: 2-3-4 Tree into Red-Black Tree



Colors can be moved from the links to the nodes pointed by these links

Red-Black Tree



Red-Black Trees

- A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that *no such path is more than twice as long as any other*, so that the tree is ***approximately balanced.***

Red-black Tree Properties

1. Every node is either red or black.
2. The root is black.
3. T.*nil* is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Heights of Red-Black Trees

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: bh(x) is the number of black nodes (including T.*nil*) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

Height of Red-Black Trees

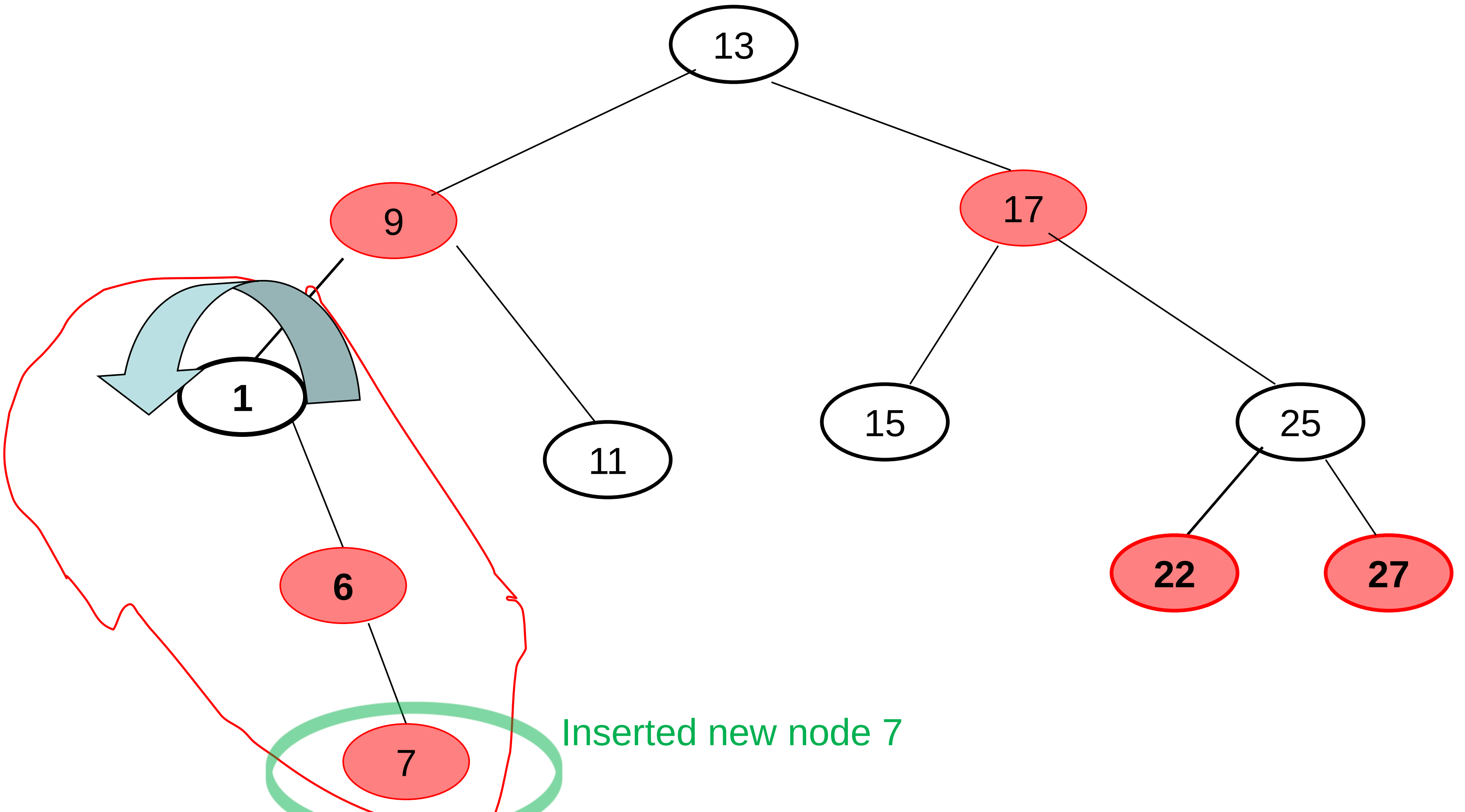
- ***Theorem***
- A red-black tree with n internal nodes has height $h <= 2 \log (n+1)$.

  – ***Proof (optional only) : see [CLRS] – chap 13.1*** .

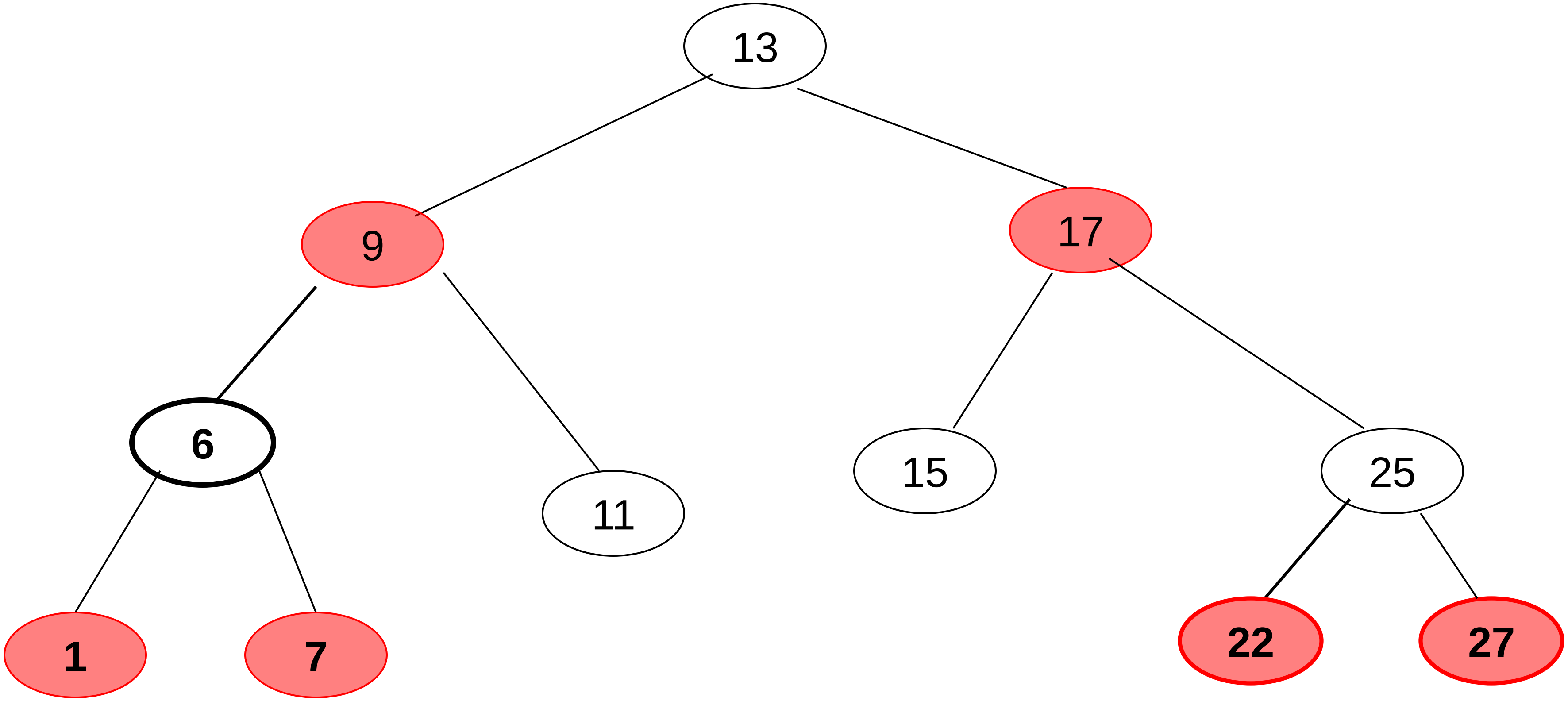
Insert in Red-Black Trees

1. Insert node z into the tree T as if it were an ordinary binary search tree
2. Color z red.
3. To guarantee that the red-black properties are preserved, we then *recolor nodes* and *perform rotations*.
   - The only RB properties that might be violated are:
     - property 2, which requires the root to be black. This property is violated if z is the root
     - property 4, which says that a red node cannot have a red child. This property is violated if z's parent is red.
       - There are 6 cases (3+3) for restoring the RB property by recoloring only or by rotations and recoloring
       - (Further reading – optional only – [CLRS]-chap 13 or [Sedgewick]-chap 3.3)

Example : RB-INSERT with rotation and recoloring



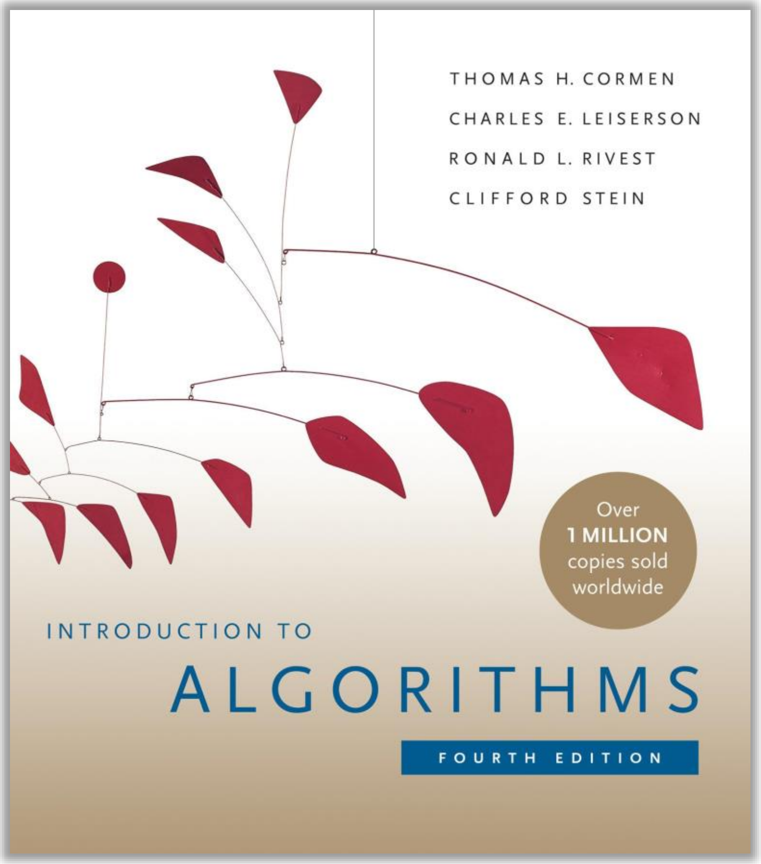
Example : RB-INSERT



AVL vs RB

	AVL	RB
Max Height	1.44 log n	2 log n
INSERT	O(log n)	O(log(n)
Rotations at Insert	O(1)	O(1)
DELETE	O(log n)	O(log n)
Rotations at Delete	O(log n)	O(1)
Used in collection libraries		Java’s TreeSet, TreeMap C++ STL set, map .NET SortedSet

Bibliography and Additional Info



- [CLRS]:
- Chap. 18 - B Trees
- Chap. 13 - Red-Black Trees