Search trees (AVL, B, B+), Skip list

- Data structures supporting Find, Insert and Delete operations

TODO list z minula:

- Prezentace na CW před přednáškou ✓
- Kvízy ✓
- Kotlin ?

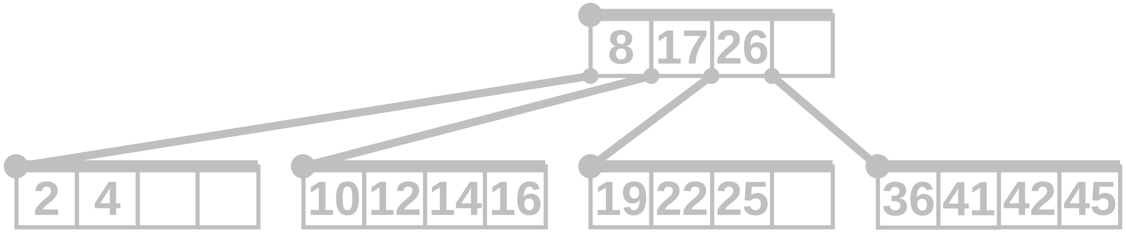
Znáte pojmy AVL strom a B-strom?

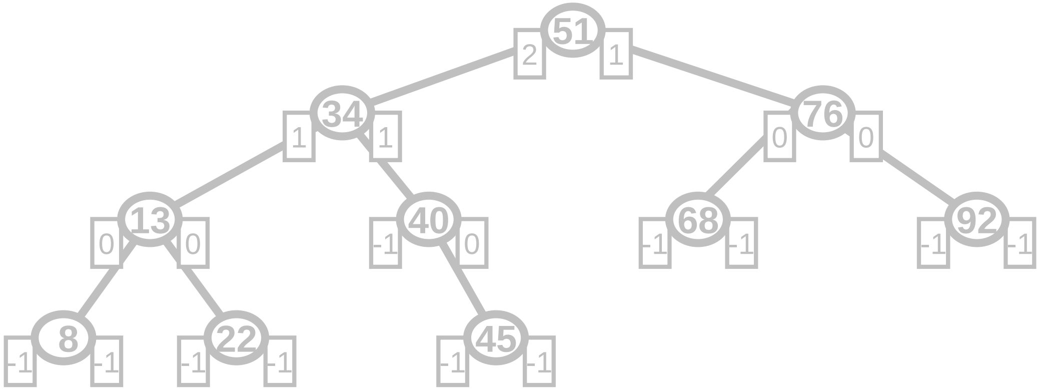
A. Znám AVL strom i B-strom.

B. Znám pouze AVL strom.

C. Znám pouze B-strom.

D. Neznám AVL ani B-strom.





BST and AVL

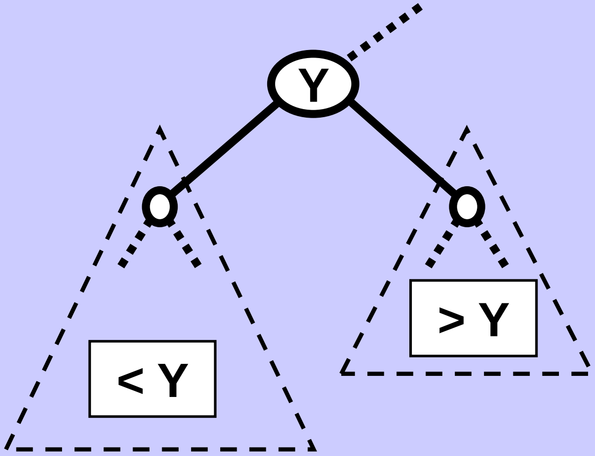
short illustrative repetition

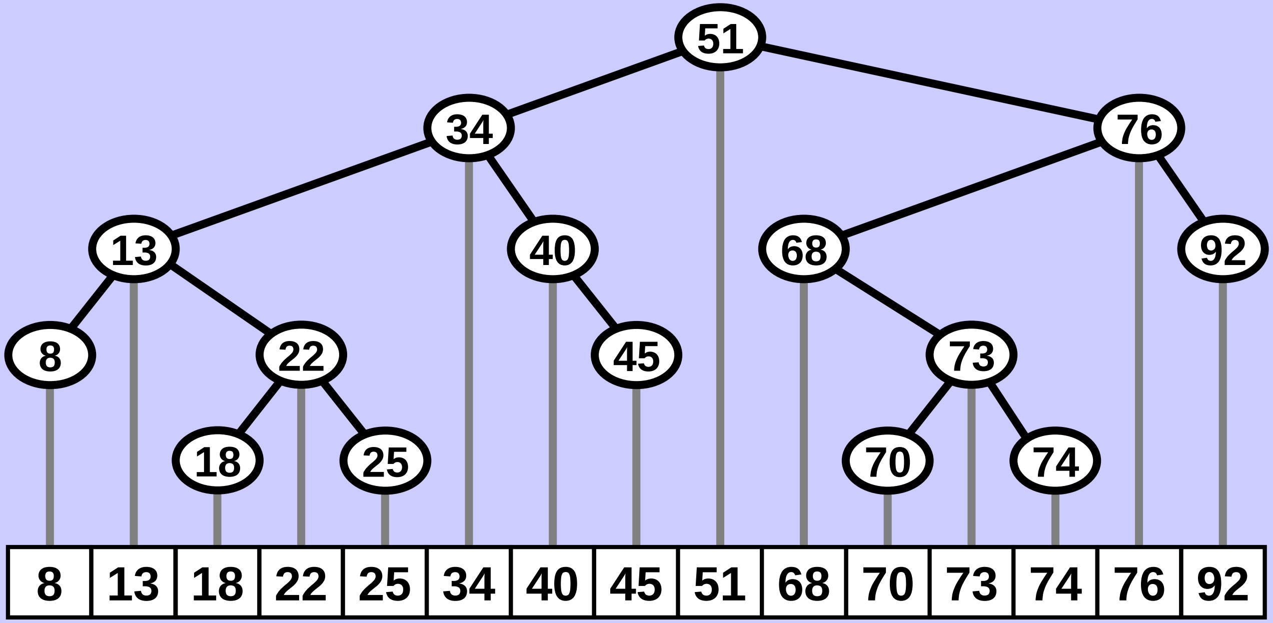
Binary search tree

For each node Y it holds:

Keys in the left subtree of Y are smaller than the key of Y.

Keys in the right subtree of Y are bigger than the key of Y.





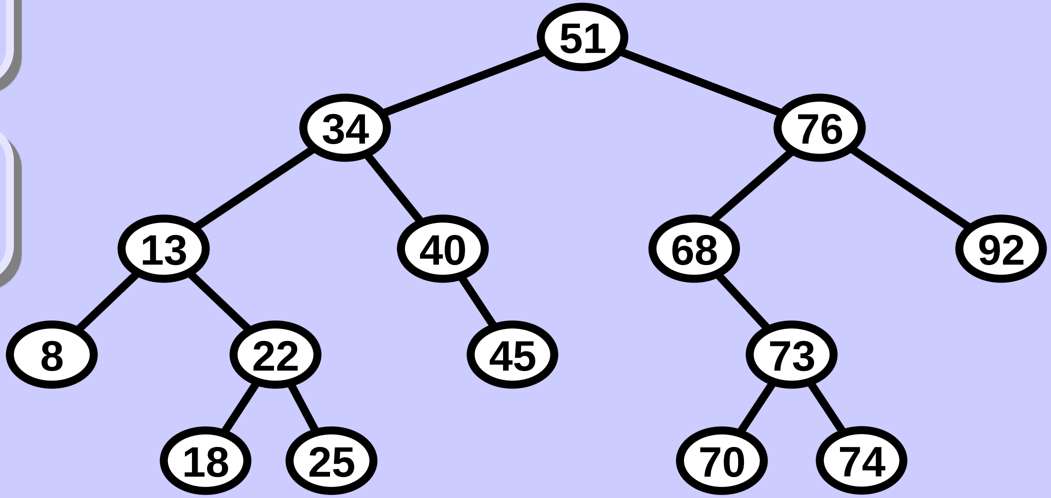
8	13	18	22	25	34	40	45	51	68	70	73	74	76	92

Binary search tree

BST may not be balanced and usually it is not.

BST may not be regular and usually it is not.

Apply the INORDER traversal to obtain sorted list of the keys of BST.

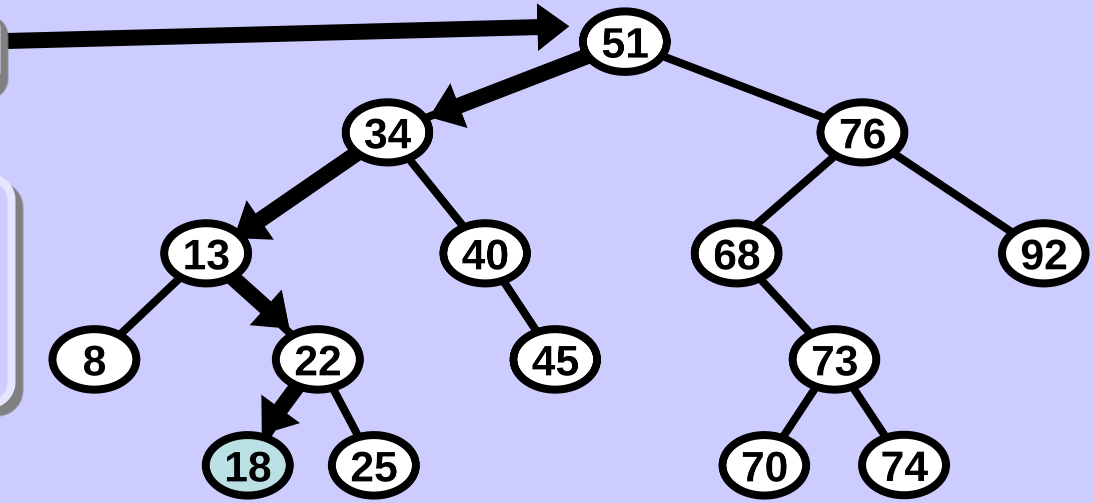


BST is flexible due to operations:

Find – return the pointer to the node with the given key (or null).
Insert – insert a node with the given key.
Delete – (find and) remove the node with the given key.

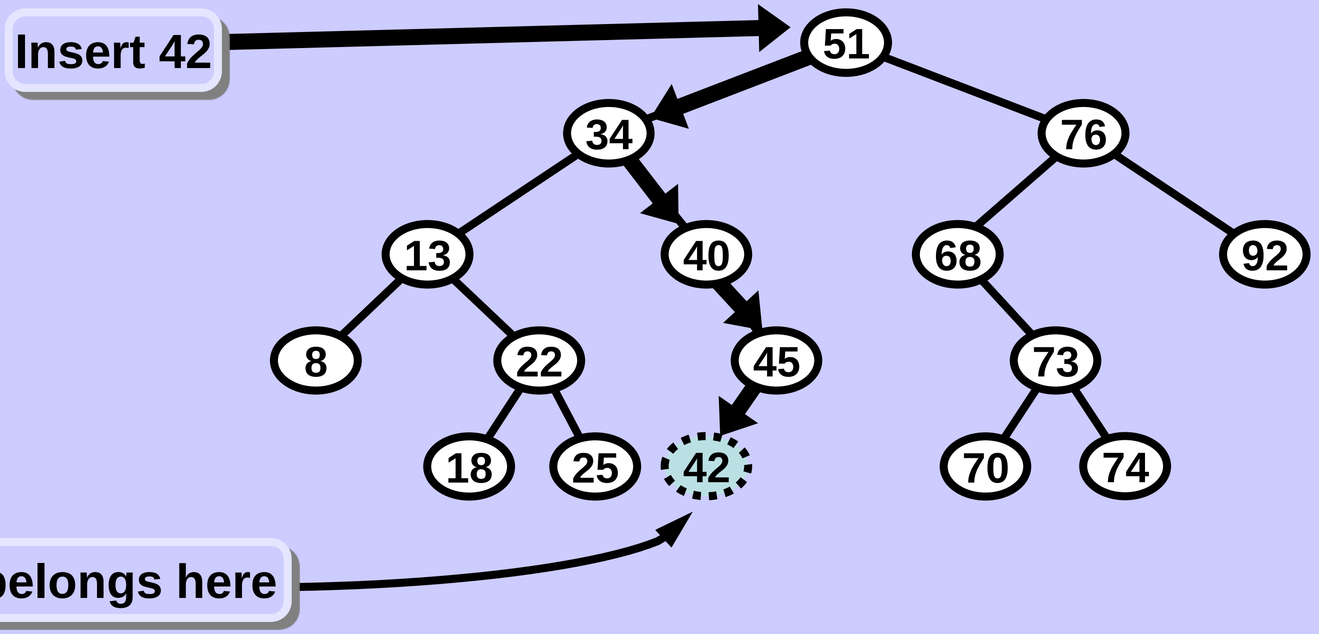
Operation Find in BST

Find 18



Each BST operation starts in the root.

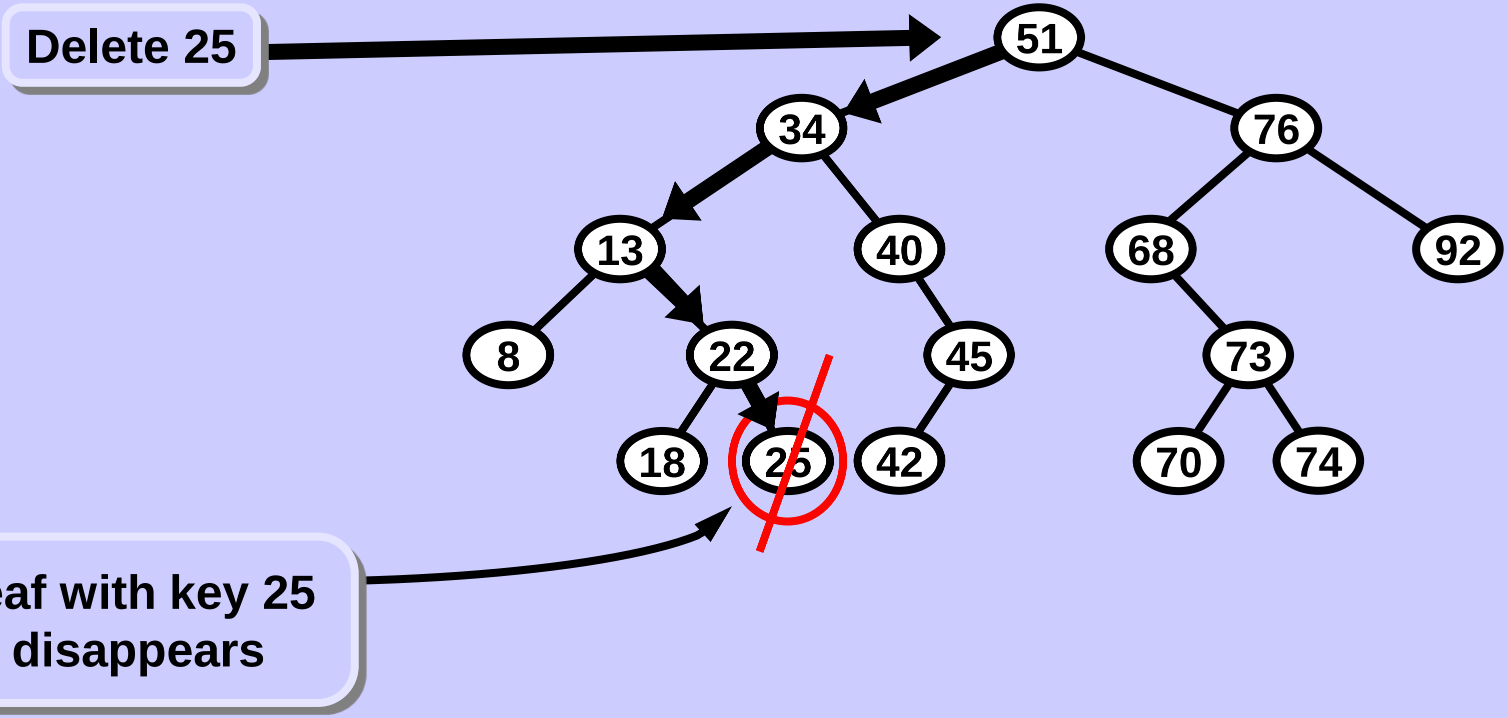
Operation Insert in BST



Insert
1. Find the place (like in Find) for the leaf where the key belongs.
2. Create this leaf and connect it to the tree.

Operation Delete in BST (I.)

Delete a node with 0 children (= leaf)

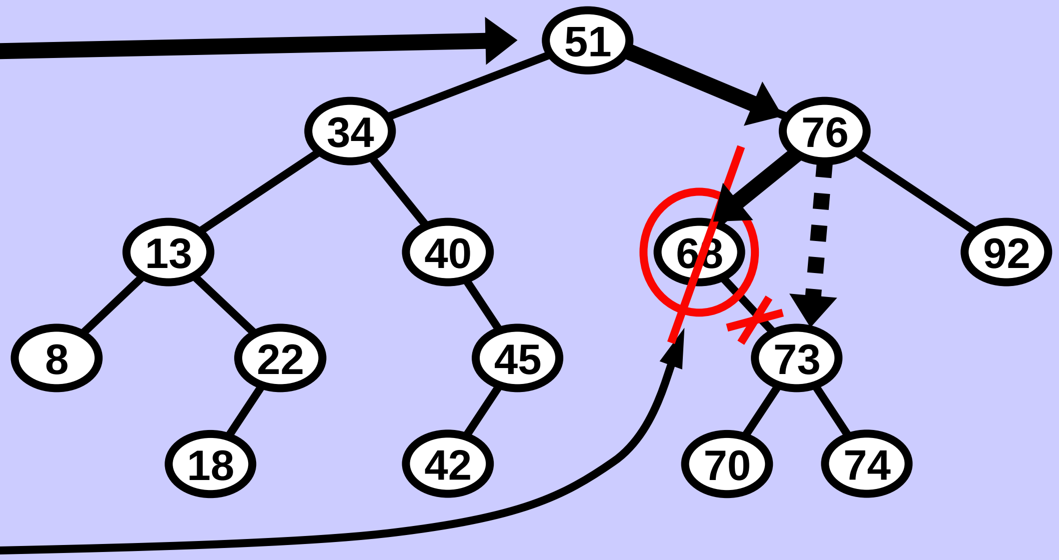


Delete I. Find the node (like in Find operation) with the given key and set the reference to it from its parent to null.

Operation Delete in BST (II.)

Delete a node with 1 child.

Delete 68



Node with key 68 disappears

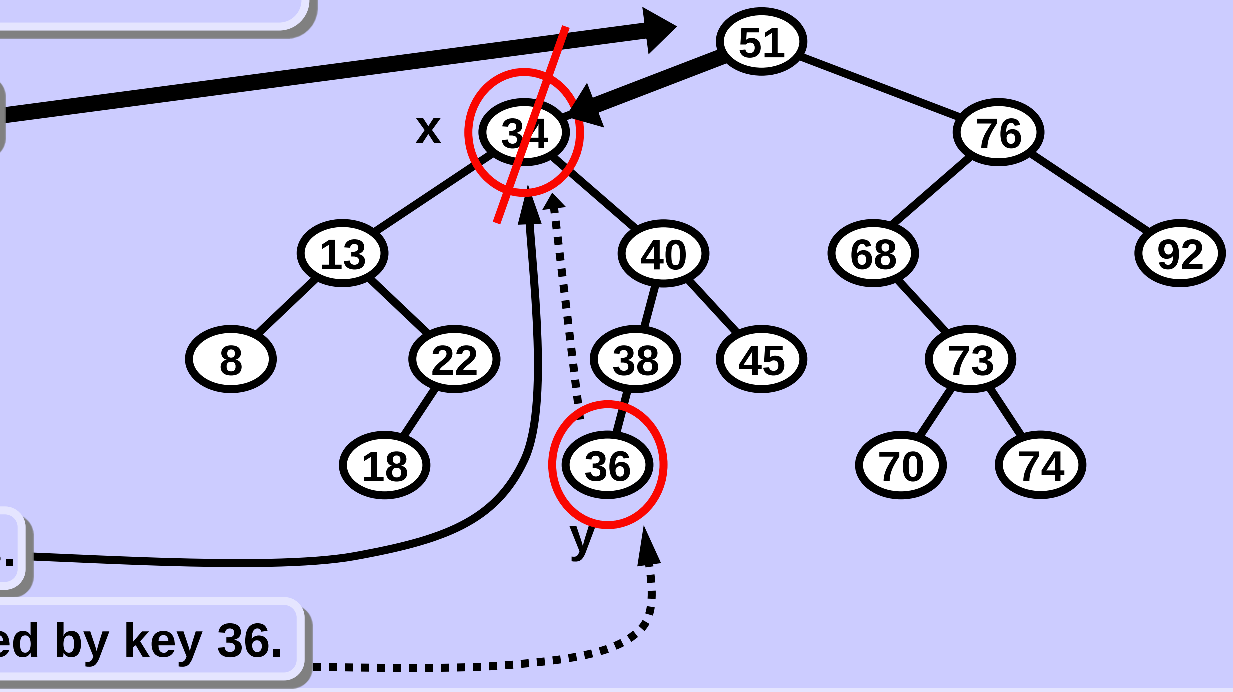
Change the 76 --> 68 reference to 76 --> 73 reference.

Delete II. Find the node (like in Find operation) with the given key and set the reference to it from its parent to its (single) child.

Operation Delete in BST (IIIa.)

Delete a node with 2 children.

Delete 34



Key 34 disappears.

And it is substituted by key 36.

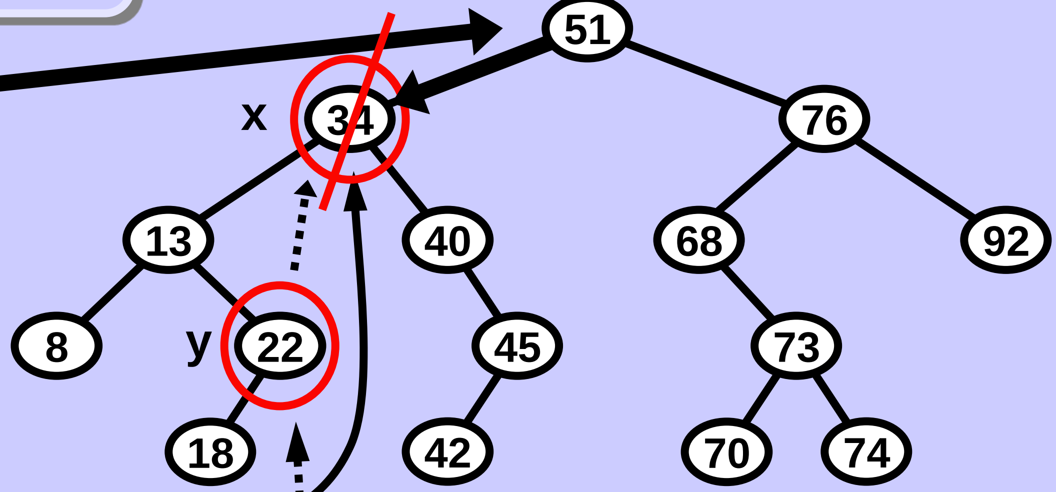
Delete IIIa.

1. Find the node (like in Find operation) with the given key and then find the *__leftmost__* (= smallest key) node y in the *__right__* subtree of x.
2. Point from y to children of x,
   __from parent of y point to the child of y instead of y__,
   from parent of x point to y.

Operation Delete in BST (IIIb.) is equivalent to Delete IIIa.

Delete a node with 2 children.

Delete 34



Key 34 disappears.

And it is substituted by key 22.

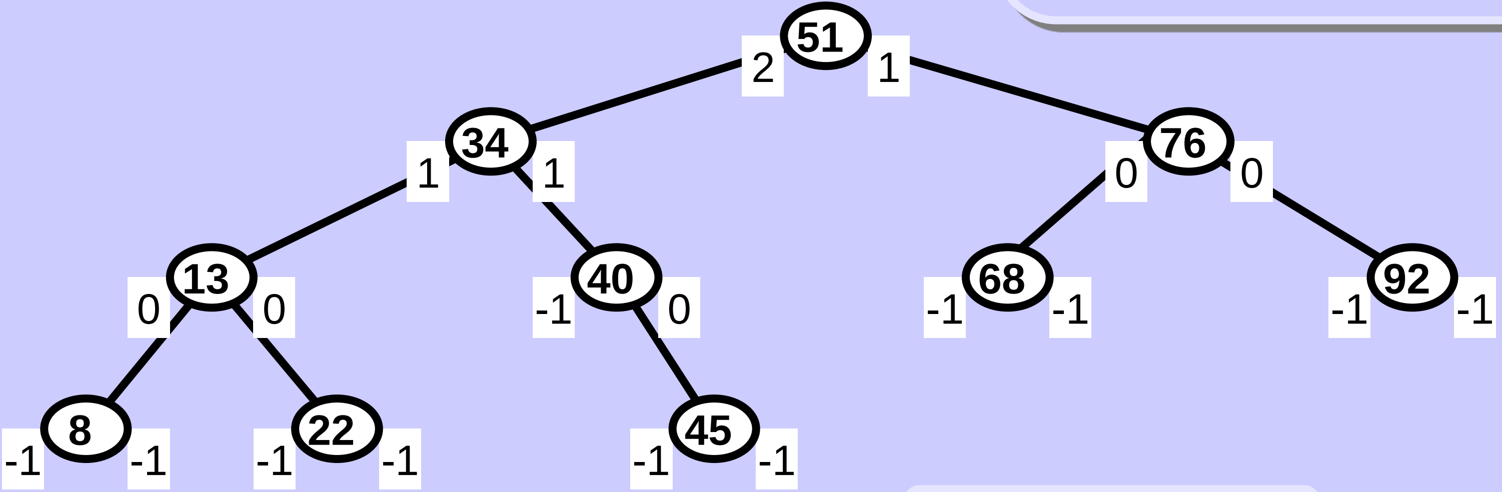
Delete IIIb.

1. Find the node (like in Find operation) with the given key and then find the ***rightmost*** (= smallest key) node y in the ***left*** subtree of x.
2. Point from y to children of x,
   from parent of y point to the child of y instead of y,
   from parent of x point to y.

AVL tree -- G.M. Adelson-Velskij & E.M. Landis, 1962

AVL tree is a BST with additional properties which keep it acceptably balanced.

Operations
Find, Insert, Delete
also apply to AVL tree.



There are two integers associated with each node:
Depth of the left and depth of the right subtree of the node.
Note: Depth of an empty tree is -1.

AVL rule:

The difference of the heights of the left and the right subtree may be only -1 or 0 or 1 in each node of the tree.

AVL tree -- G.M. Adelson-Velskij & E.M. Landis, 1962

Find -- same as in a BST

Insert -- first, insert as in a BST,
next, travel from the inserted node upwards
and update the node depths.
If disbalance occurs in any node along the path then
apply an appropriate rotation and stop.

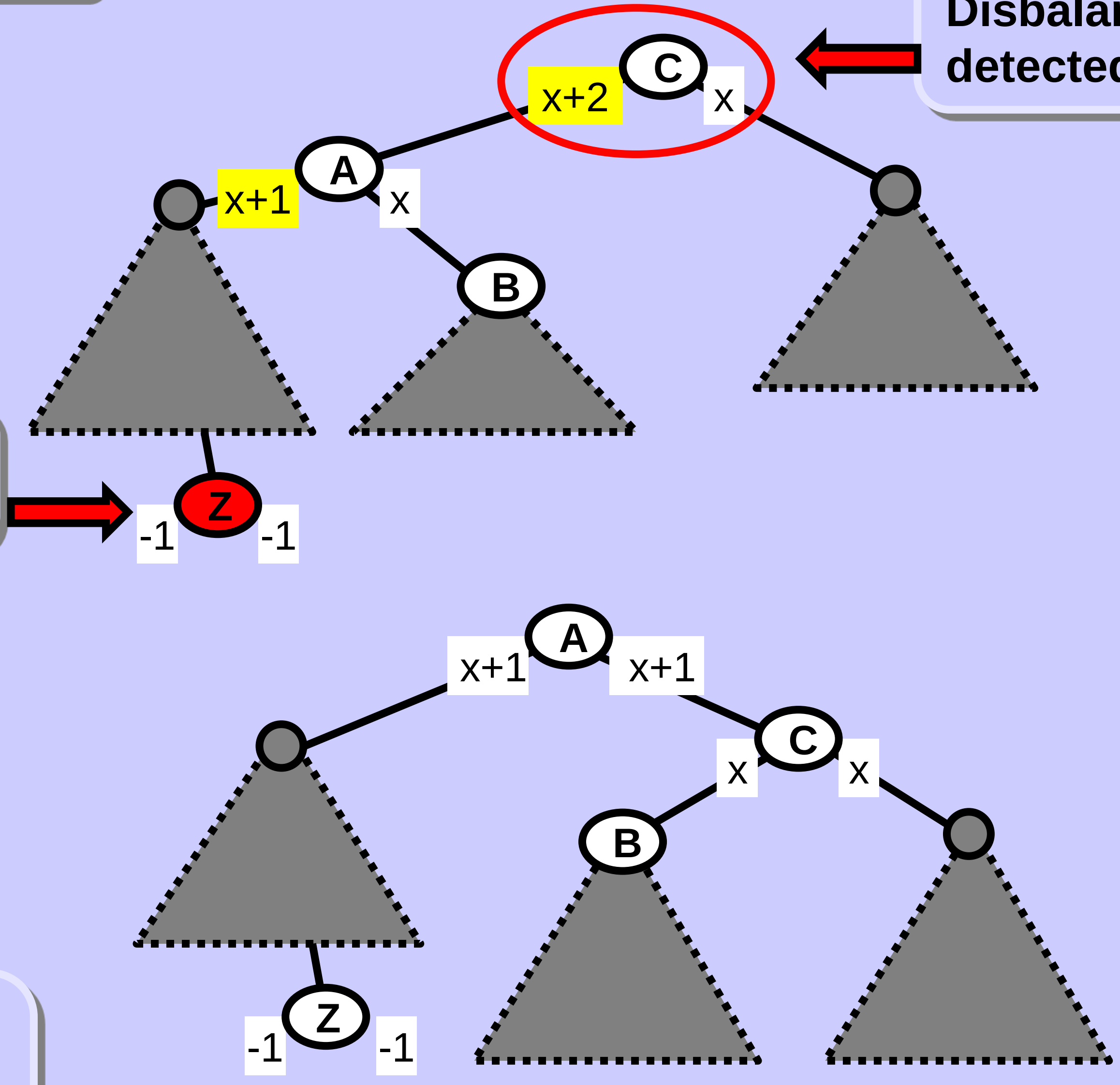
Delete -- first, delete as in a BST,
next, travel from the deleted position upwards
and update the node depths.
If disbalance occurs in any node along the path then
apply an appropriate rotation.
Continue travelling along the path up to the root.

Rotation R in general

Before

Disbalance detected

Disbalancing node

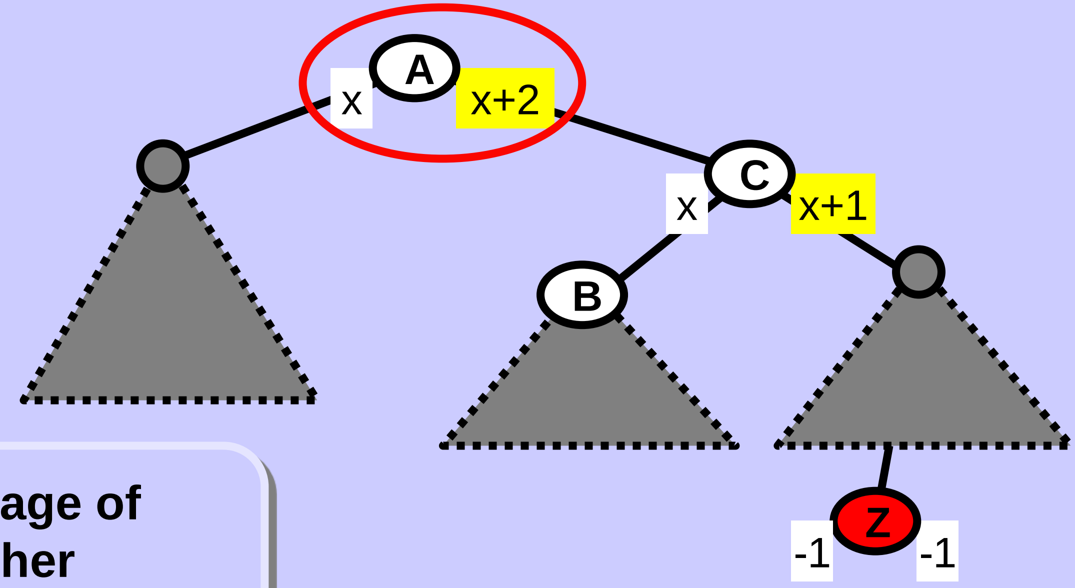


After

Unaffected subtrees ▲

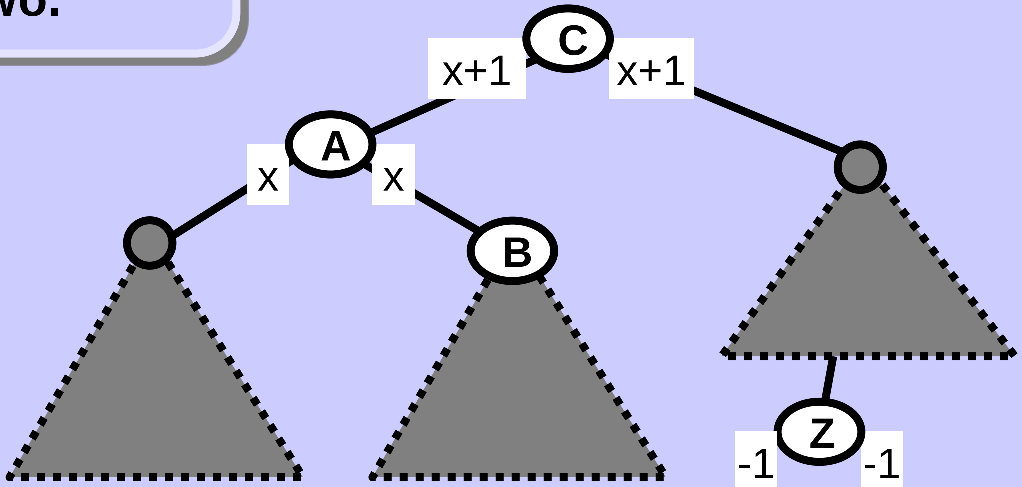
Rotation L in general

Before



Rotation L is a mirror image of rotation R, there is no other difference between the two.

After

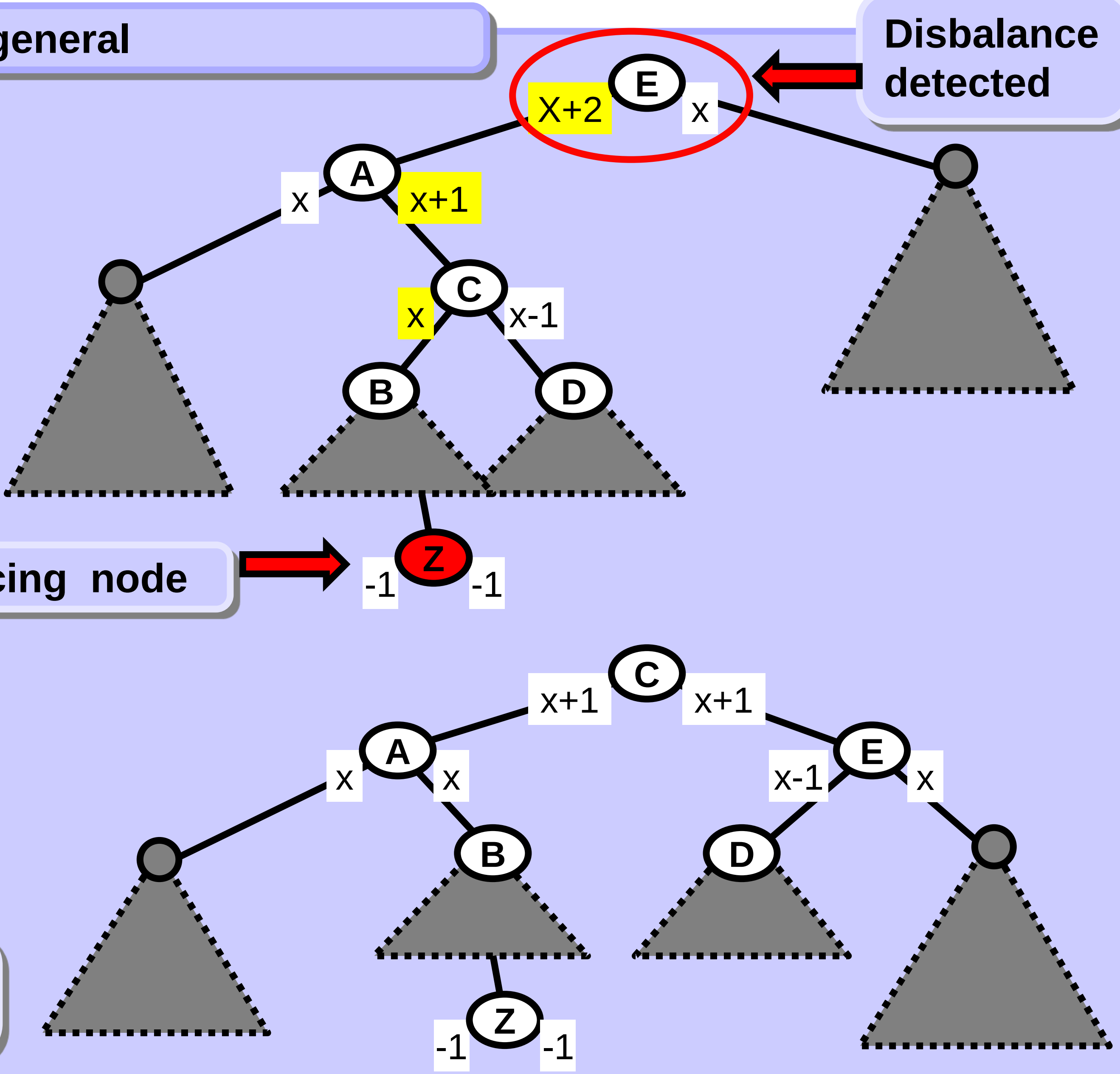


Unaffected subtrees ▲

Rotation LR in general

Before

Disbalance detected



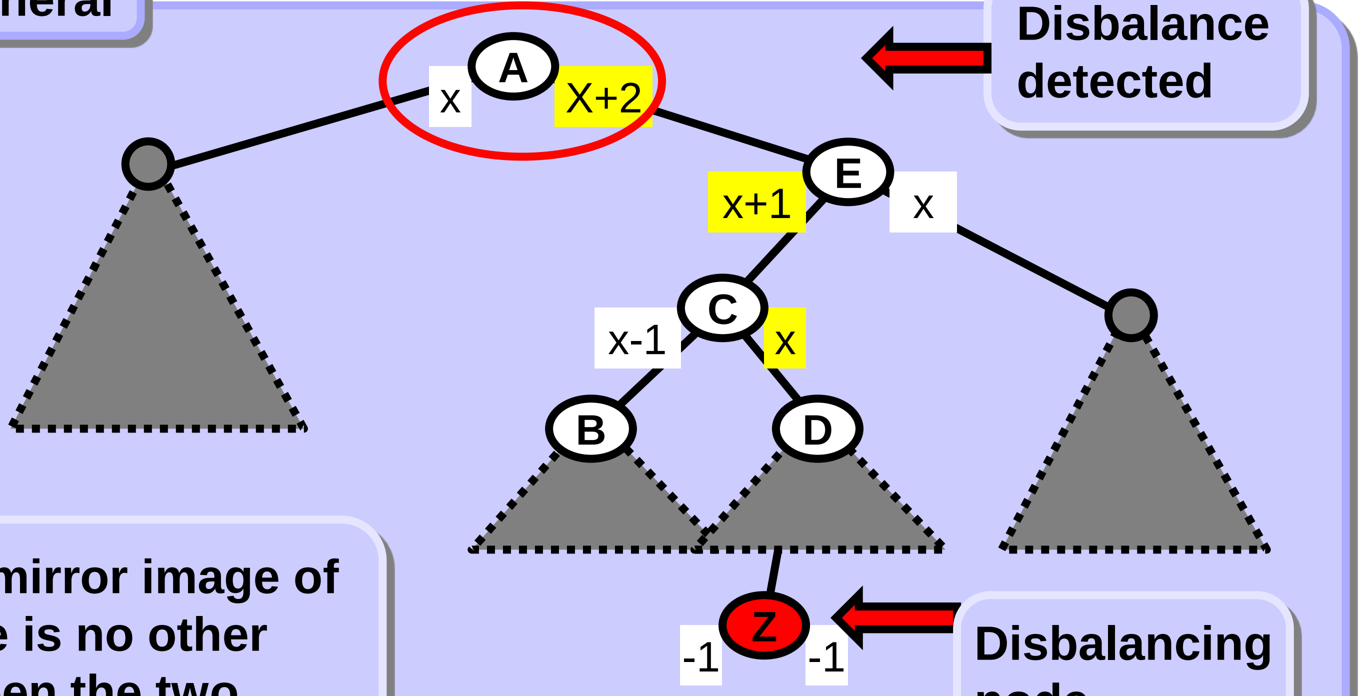
Disbalancing node

After

Unaffected subtrees

Rotation RL in general

Before

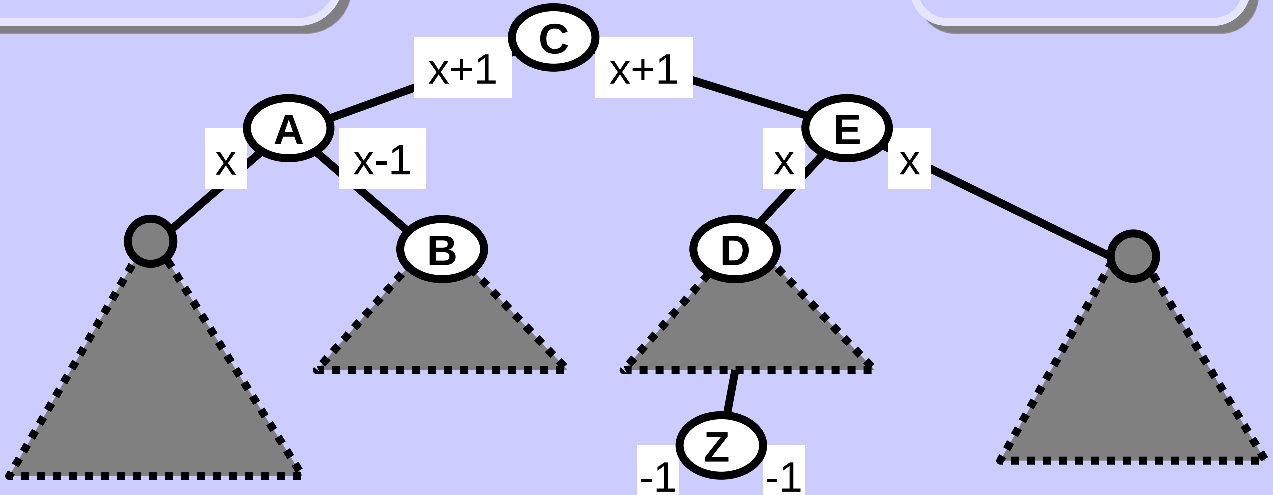


Disbalance detected

Disbalancing node

Rotation RL is a mirror image of rotation LR, there is no other difference between the two.

After



Unaffected subtrees ▲

Rules for aplying rotations L, R, LR, RL in Insert operation

Travel from the inserted node up to the root
and update the subtree depths in each node along the path.

If a node is disbalanced and you came to it along two consecutive edges

***   in the up and *right* direction**
perform rotation R in this node,

***   in the up and *left* direction**
perform rotation L in this node,

***   first in the in the up and *left* and then in the up and *right* direction**
perform rotation LR in this node,

***   first in the in the up and *right* and then in the up and *left* direction**
perform rotation RL in this node,

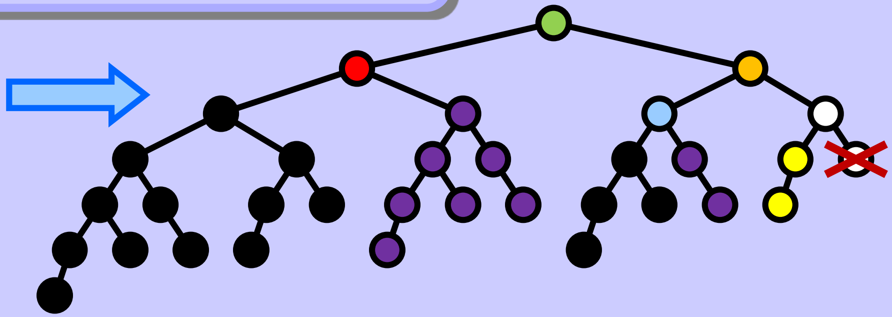
After one rotation in the Insert operation the AVL tree is balanced.

After one rotation in the Delete operation the AVL tree might still
not be balanced, all nodes on the path to the root have to be checked.

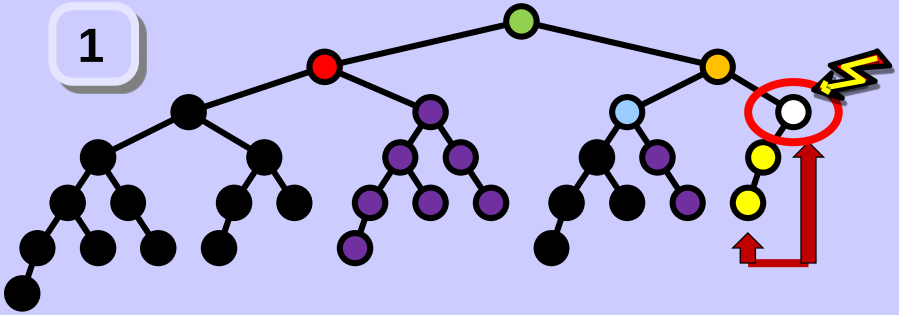
Necessity of multiple rotations in operation Delete.

Example.
The AVL tree
is originally
balanced.

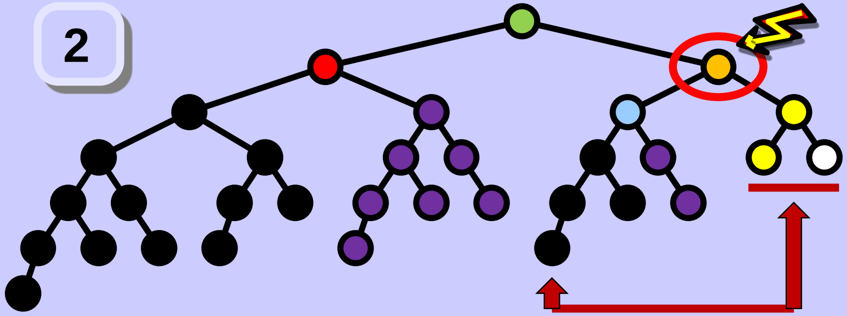
Delete the
rightmost key.



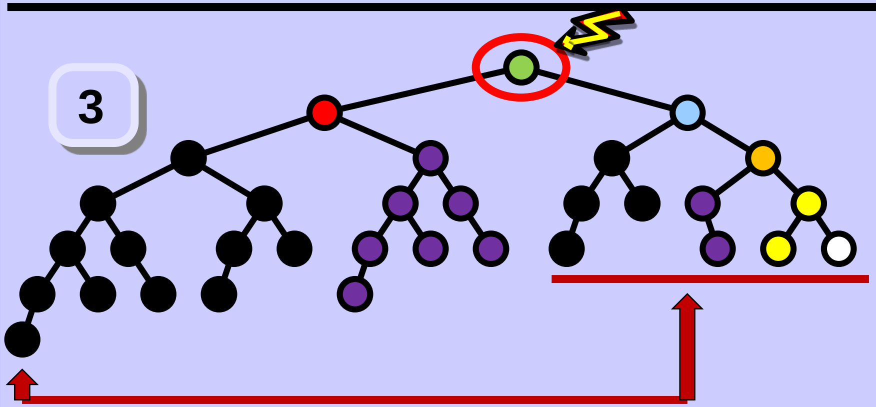
1



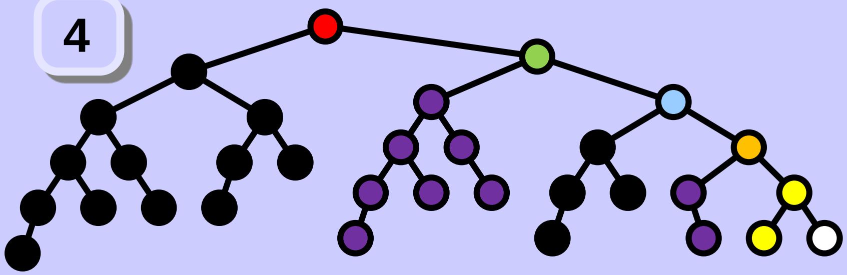
2



3



4



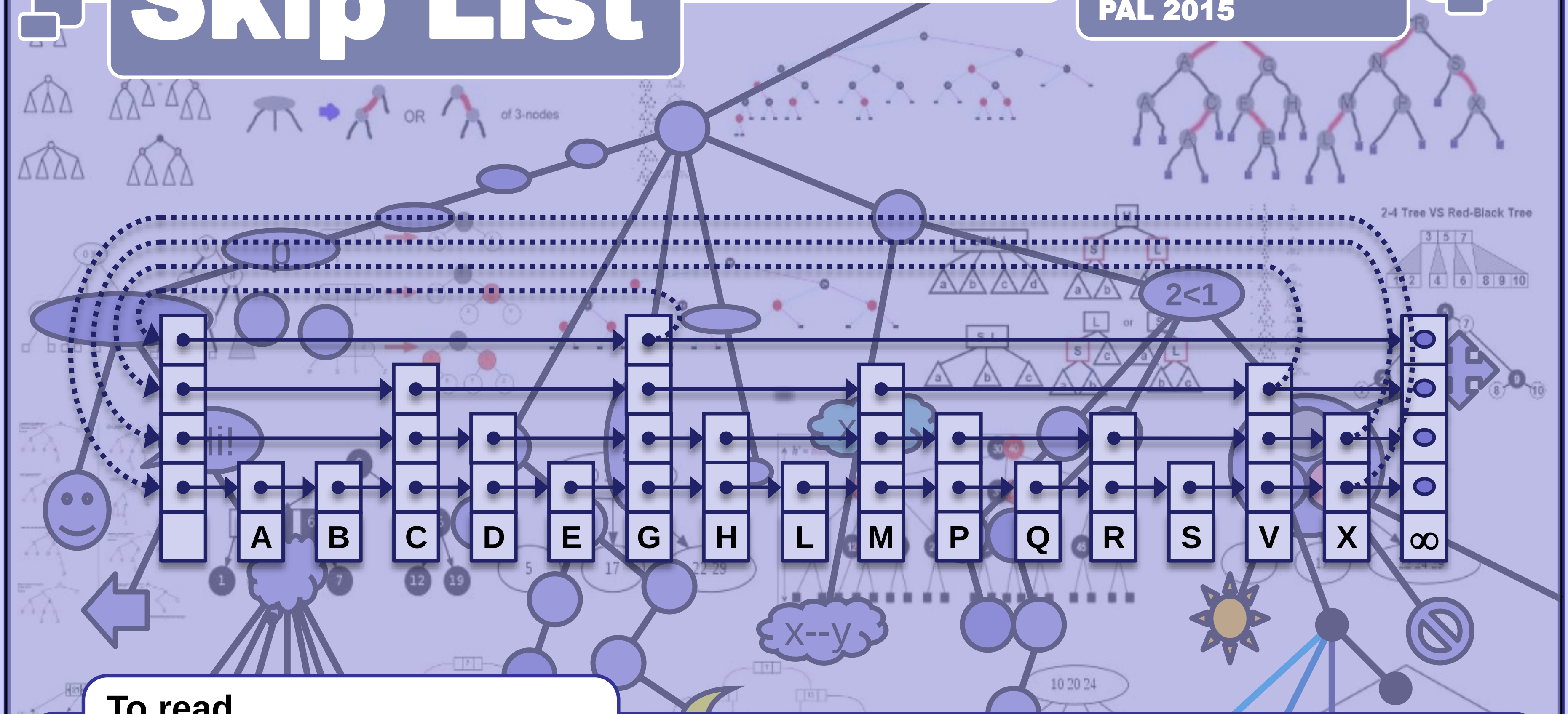
Balanced.

Asymptotic complexities of Find, Insert, Delete in BST and AVL

	BST with n nodes		AVL tree with n nodes
Operation	**Balanced**	**Maybe not balanced**	**Balanced**
Find	$O(\log(n))$	$O(n)$	$O(\log(n))$
Insert	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$
Delete	$O(\log(n))$	$O(n)$	$\Theta(\log(n))$

Skip List

Marko Berezovský
PAL 2015



To read

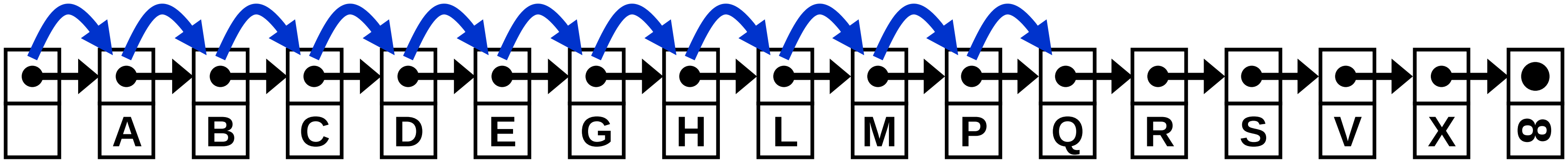
- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998
- William Pugh. *Skip lists: A probabilistic alternative to balanced trees*. Communications of the ACM, 33(6):668–676, 1990.
- William Pugh: *A Skip List Cookbook* [http://cglab.ca/~morin/teaching/5408/refs/p90b.pdf]
- Bradley T. Vander Zanden: [http://web.eecs.utk.edu/~huangj/CS302S04/notes/skip-lists.html]

Skip list

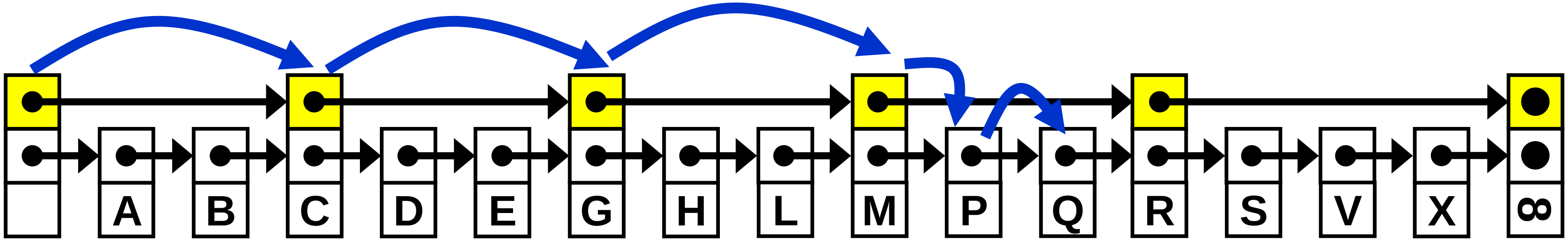
Motivation

Problem: **Find(Q)** in your list.

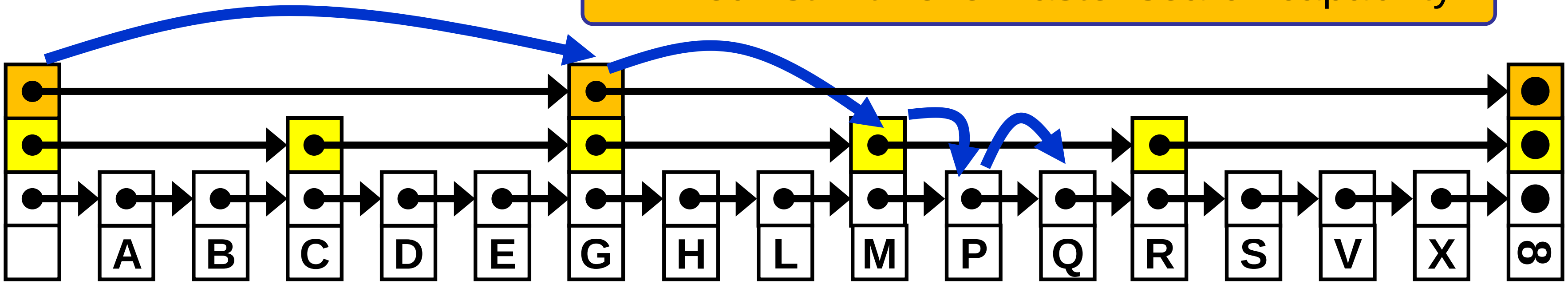
A regular linked list



A linked list with faster search capability

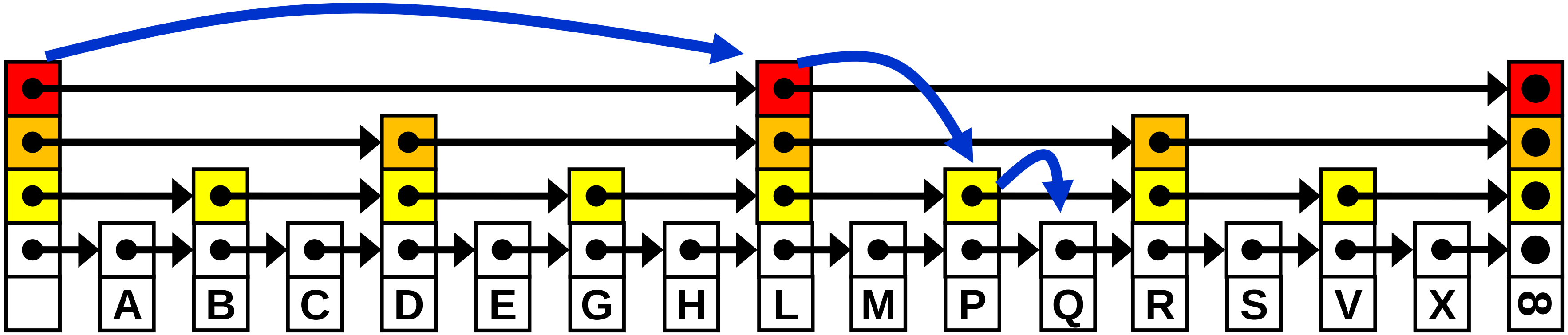


A linked list with even faster search capability



Skip list

A linked list with log(N) search capability.



Note the shape **similarity** to a **balanced binary search tree**.

Difficulty:
Subsequent Insert/Delete operations would destroy this favourable list shape.
The cost of restauration is huge -- Θ(N).

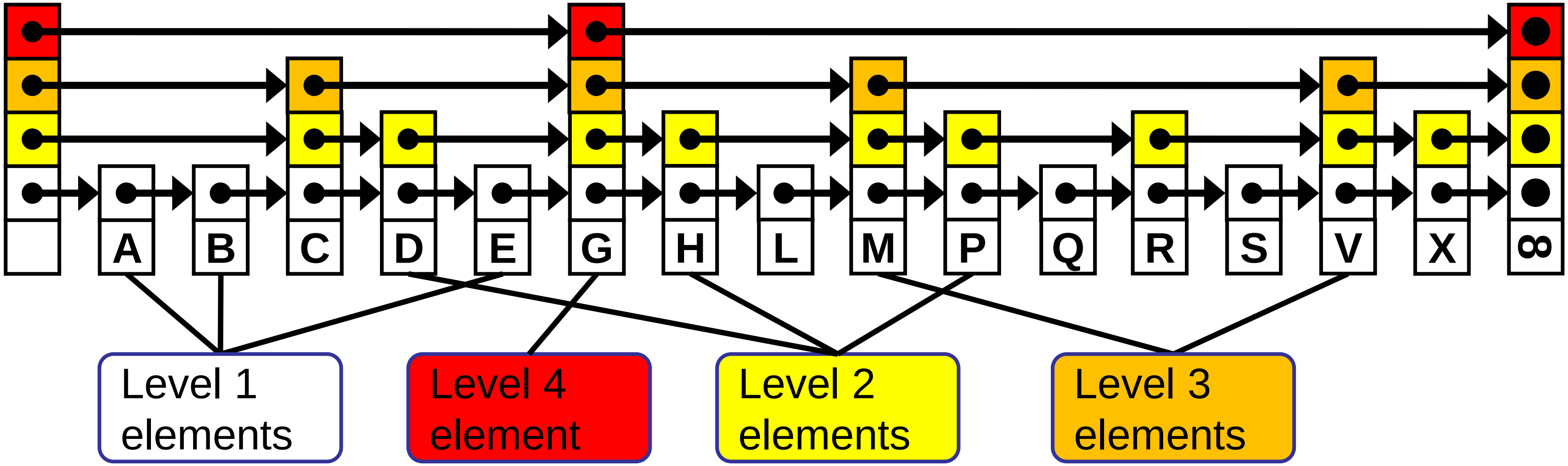
Solution:
Create a randomized shape, roughly similar to the optimal shape.
Random deviations from the nice shape in the long run nearly cancel each other.
The result is a list shape with properties relatively close to the optimal properties.

A skip list is an ordered linked list where each node contains a variable number of links, with the k-th link in the node implementing singly linked list that skips (forward) the nodes with less than k links.

[Sedgewick]

Each element points to its immediate successor (= next element).
Some elements also point to one or more elements further down the list.

A **level *k*** element has ***k*** forward pointers.
the ***j-th*** pointer points to the next element in level ***j*** .



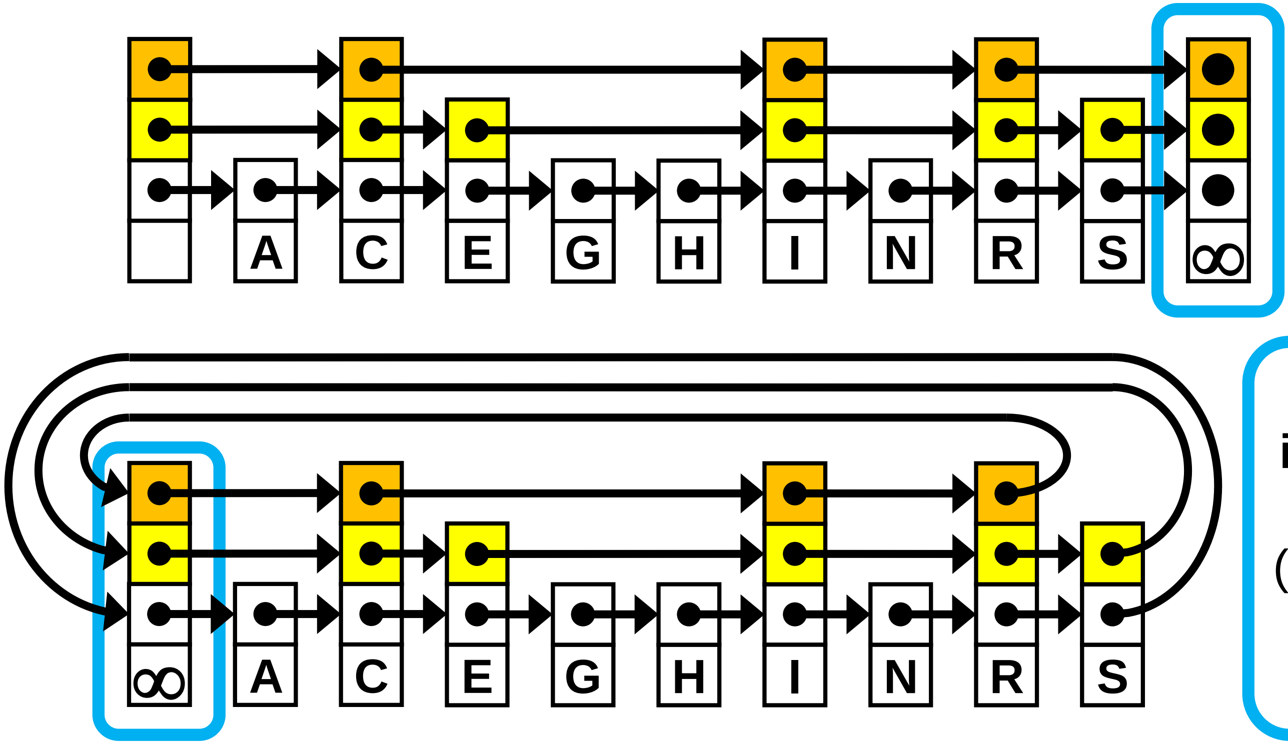
Skip list - Structure

Keeping the code simple

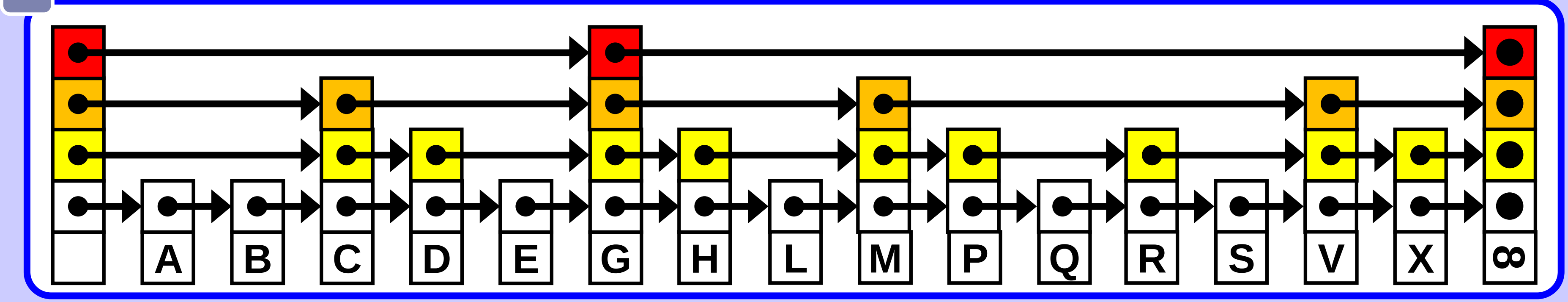
```
while(x.forward[i].key < searchKey) // x.forward[i] != null
```

There is a **sentinel** with infinite key value at the tail of the list.

The level of the sentinel is the same as the whole list level.
The list may be implemented as circular with the header serving as the sentinel.



Note that the skiplist **is displayed as linear in this presentation** (with separate sentinel) to keep pictures less cluttered.



A skip list element contains:

-- **Key** Search key.
-- **Value** (Optional, not discussed here, allowing associative structure.)
-- **Forward[ ]** Array of pointers to the following skip list elements.

The header and the sentinel are of the same type.

A skip list data structure contains also:

-- **Header** A node with the initial set of forward pointers.
-- **Sentinel** Optional last node with ∞ value, it is the header in circular list.
-- **Level** The current number of levels in the skip list.
-- **MaxLevel** The maximum number of levels to which a skip list can grow.
-- **Update[ ]** Auxiliary array with predecessors of an inserted/deleted element see Insert and Delete operations.

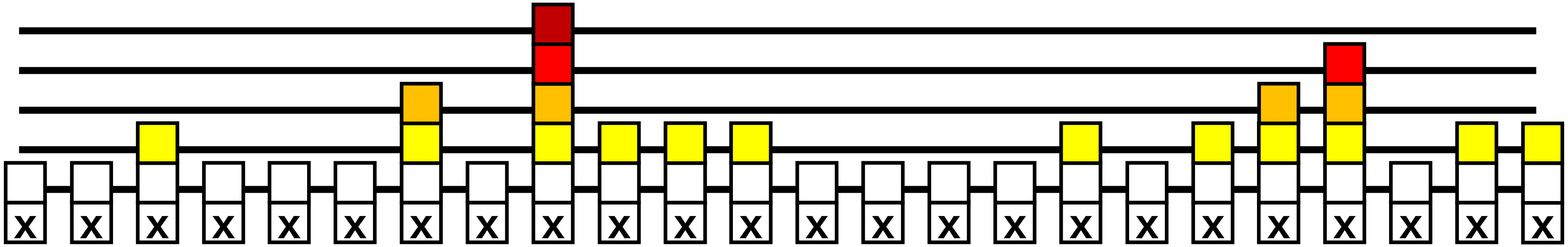
Basic randomness

The level of an element is chosen by **flipping a coin**.

Flip a coin until it comes up tails. Count
one plus the **number of times**
the coin came up heads
before it comes up tails.

This result represents the level of the element.

Sixpence of Queen Elizabeth I,
struck in 1593 at the Tower Mint.
[wikipedia.org]

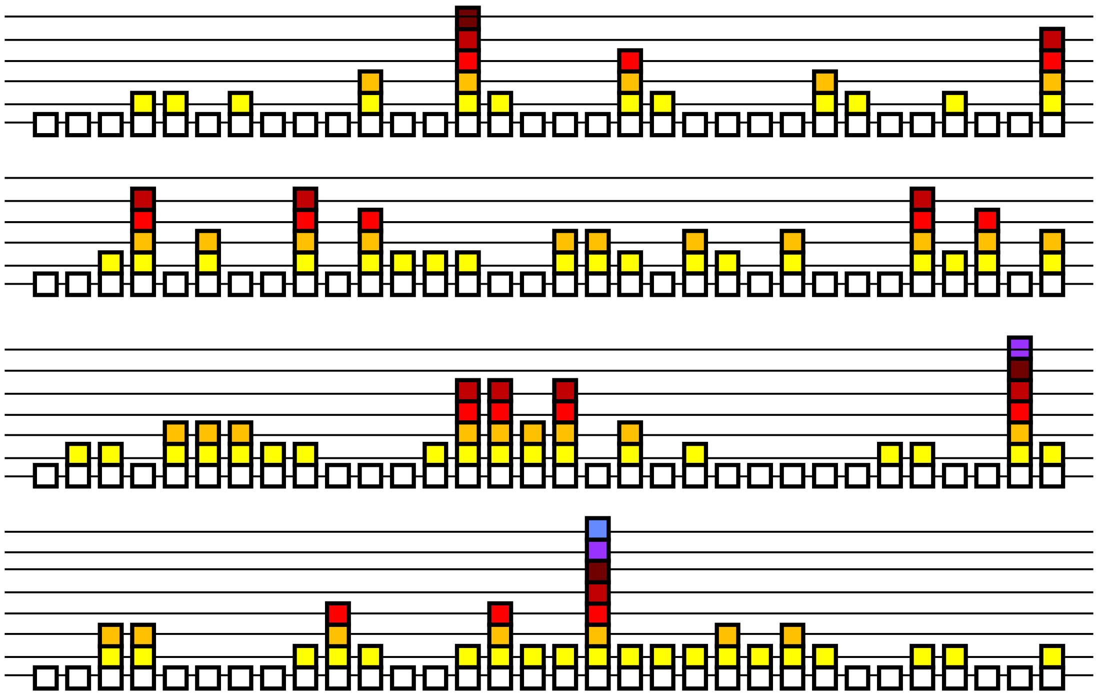


Example of an experimental independent levels calculation (p = 0.5, see below) .

Experiment with Lehmer generator

$X_{n+1} = 16807\ X_n \bmod 2^{31}–1$

seed = 23021905 // birth date of Derrick Henry Lehmer



Coin flipping:

(Xn >> 16) & 1

Head = 1

128 nodes

	Level	1	2	3	4	5	6	7	8	9	...
Number of nodes	Expected	64	32	16	8	4	2	1	1/2	1/4	...
	Actual	60	36	17	5	7	1	1	1	0	...

Skip list

Random level

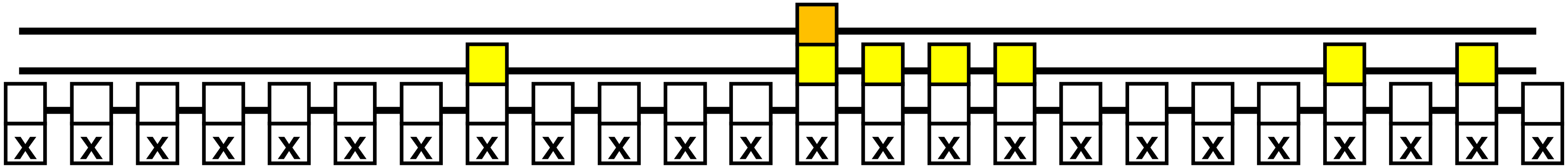
This scheme corresponds to flipping a coin that has
p chance of coming up heads,
(**1–*p***) chance of coming up tails.

More general randomness

Choose a fraction *p* between 0 and 1.
Rule: Fraction *p* of elements with level k pointers
will have level k+1 elements as well.

On average:

□ (1–*p*) elements will be level 1 elements,
🟨 (1–*p*)^2 elements will be level 2 elements,
🟧 (1–*p*)^3 elements will be level 3 elements, etc.



Example of an experimental independent levels calculation with p = 0.33.

Random level

Choosing a Random Level

A level is chosen for an element in effect by flipping a coin that has probablility *p* of coming up heads. We keep flipping until we get "tails" or until the maximum number of levels is reached.

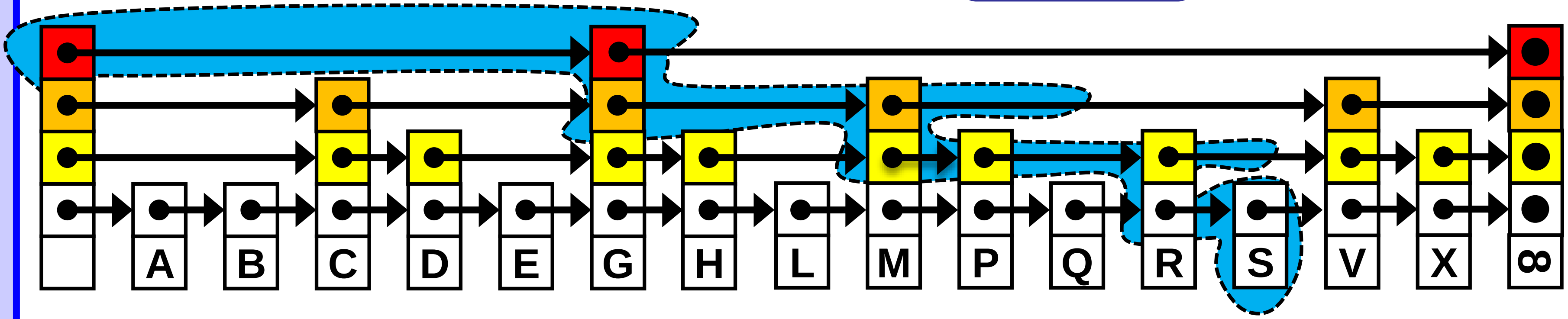
```
int randomLevel( List list ) {
  //random() returns a random value in [0..1)
  int newLevel = 1;
  while( random() < list.p )    // no MaxLevel check!
    newLevel++;
  return min( newLevel, list.MaxLevel );// efficiency!
}
```

Search

Scan through the top list until the current node either contains the search key or it contains a smaller key and a link to a node with a larger key.

Then, move to the second-from-top list and iterate the procedure, continuing forward and downward until the search key is found or a search mismatch happens at the bottom level.

Find S



Skip list - Search

Search

Start with the coarsest grain list and find where in that list the key resides, then drop down to the next less coarse grain list and repeat the search.

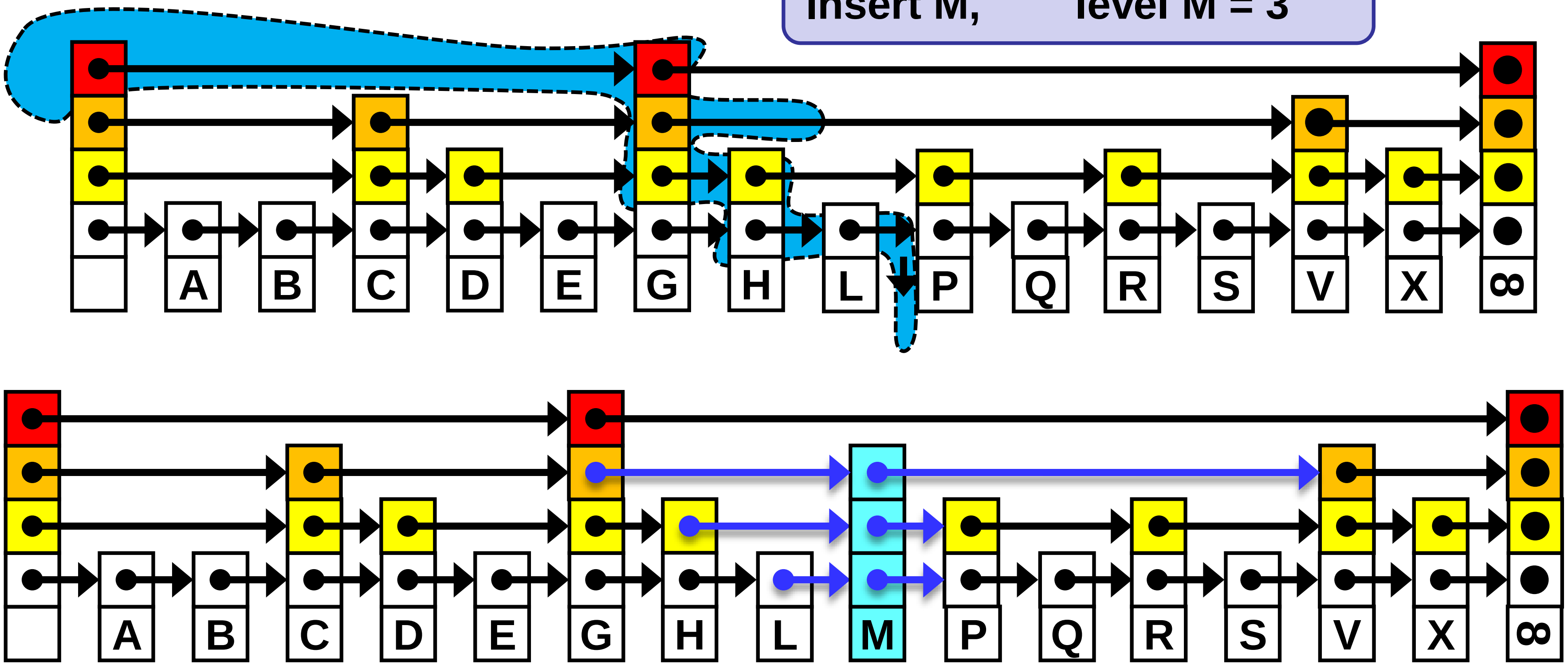
```
Node search( List list, int searchKey ) {
  Node x = list.header;

  // loop invariant: x.key < searchKey, strict ineq!!
  for( int i = list.level; i >= 1; i-- )
     while( x.forward[i].key < searchKey )
       x = x.forward[i];

  // x.key < searchKey <= x.forward[1].key
  x = x.forward[1];
  if( x.key == searchKey ) return x;
  else return null;  // not found
}
```

Skip list - Insert

Insert

Find the place for the new element.
Compute its level ***k*** by flipping the coin.
Insert the element into first ***k*** lists, starting at the bottom list.

Insert M, level M = 3



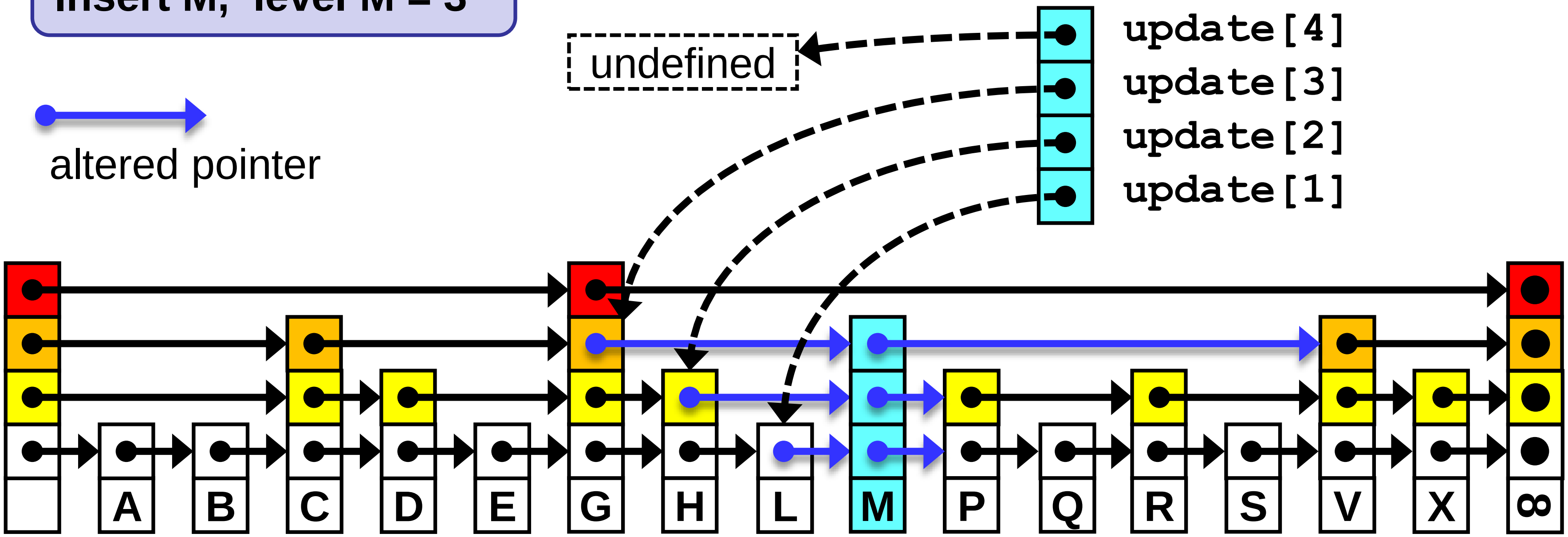
Skip list - Insert

The array **update [ ]** is an auxiliary array supporting Insert / Delete operations.

update[**k**] points to that element in the list
whose level **k** pointer points to the inserted (or deleted) element,
(= predecessor in the **k**-th level).

Note that in many cases, when the level of the inserted/deleted element is 1, only update[**1**] will be used.

Insert M, level M = 3



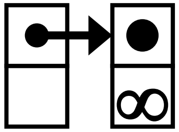
Skip list - Insert

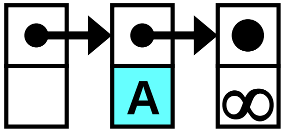
Example

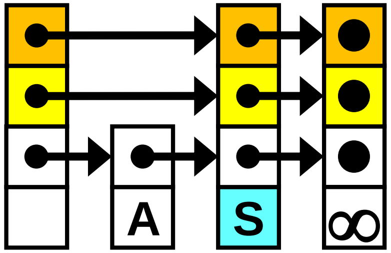
Insert A, S, E, R, C, H, I, N, G.

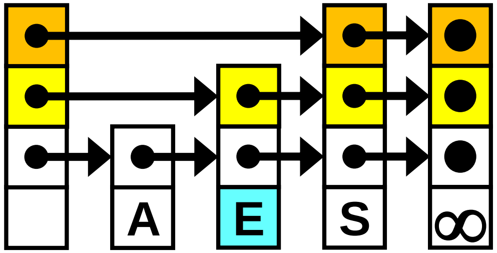
The nodes, in the order of insertion, **(A,S,E,R,C,H,I,N,G)**
were assigned levels **1,3,2,1,1,3,1,3,2.**

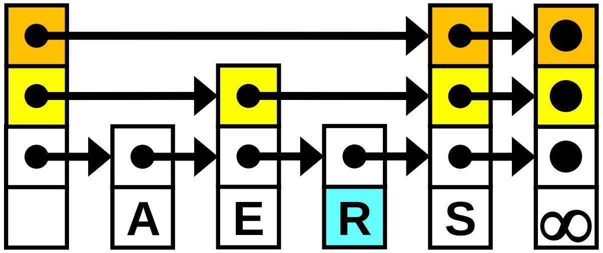
[Sedgewick]

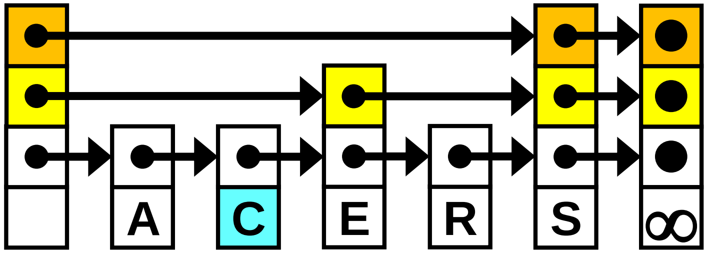


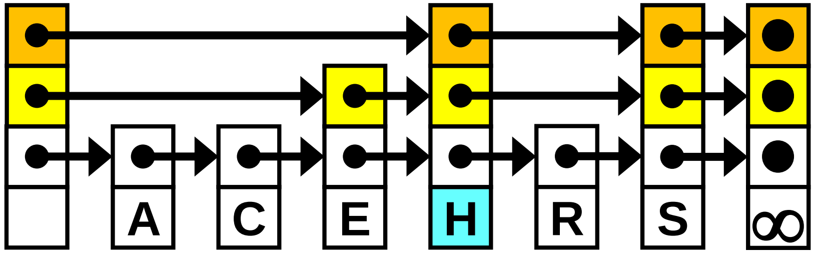


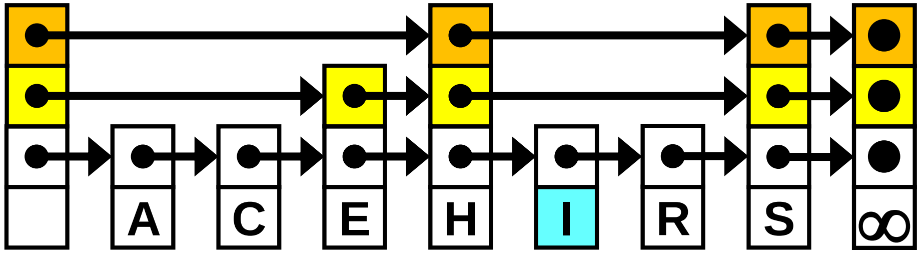










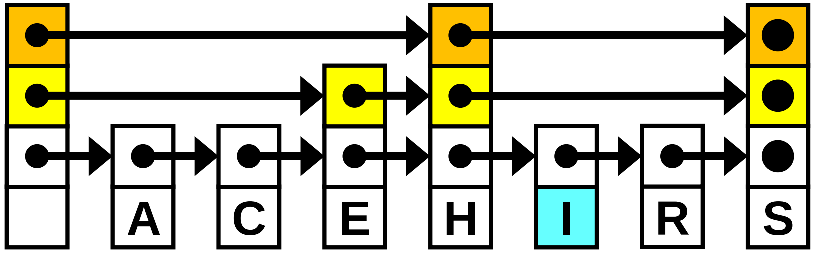


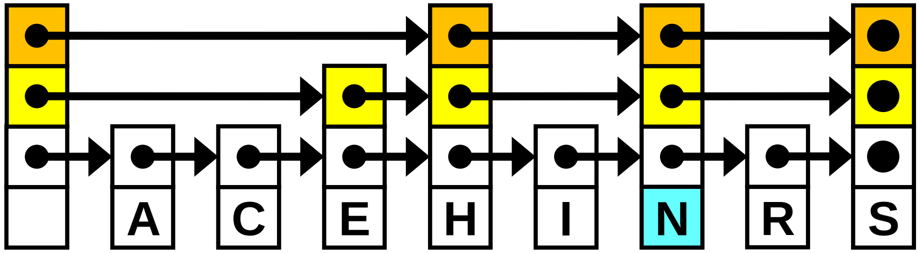
continue...

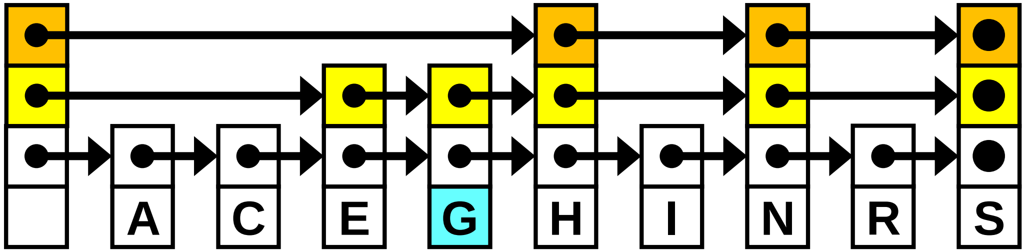
Skip list - Insert

.. continued

Insert A, S, E, R, C, H, I, N, G.







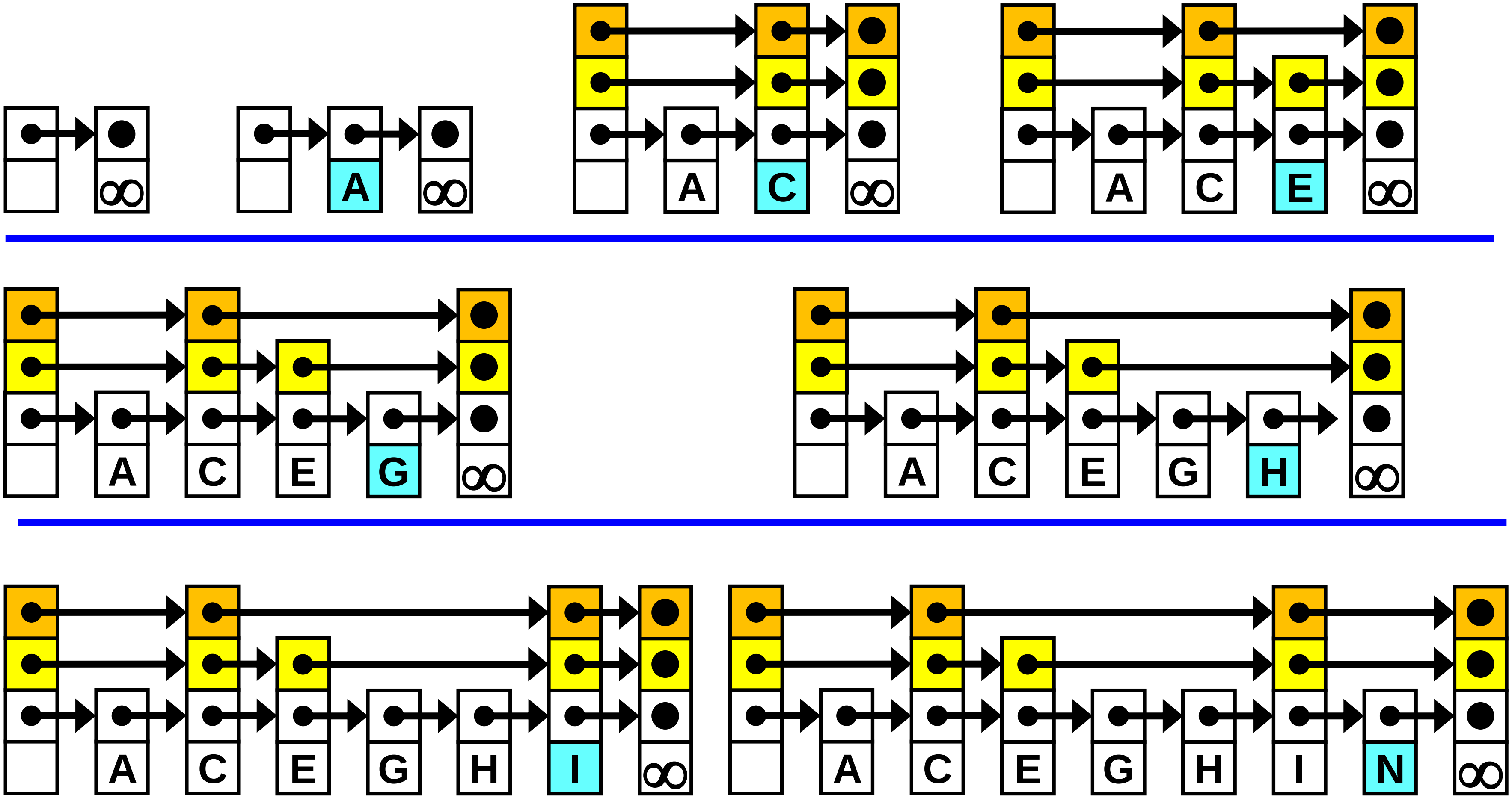
The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,S,E,R,C,H,I,N,G)

Skip list - Insert

Example

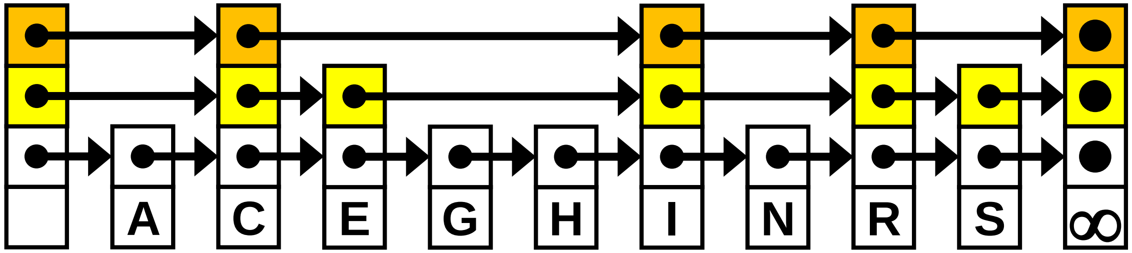
Insert A, C, E, G, H, I, N, R, S.
(Same values, different order)

The nodes `(A,C,E,G,H,I,N,R,S)`
were assigned levels `1,3,2,1,1,3,1,3,2.`



etc...

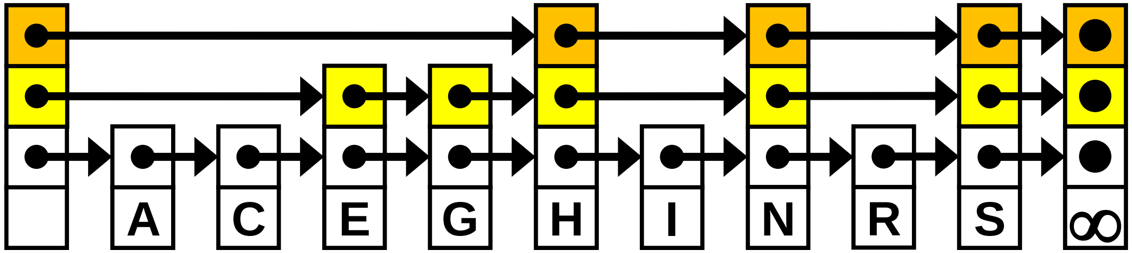
Skip list - Insert



The nodes were inserted in sorted order.

The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,C,E,G,H,I,N,R,S)

The result of the previous example



The nodes were inserted in random order.

The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,S,E,R,C,H,I,N,G)

The shapes of the lists are different, the probabilistic properties are the same.

Skip list - Insert

```
            // update[k] .. predecessor at level k

void insert(List list, int searchKey, Data newValue){

  Node x = list.header;
  for( int i = list.level; i >= 1; i-- ){
    //invariant: x.key < searchKey <= x.forward[i].key
    while( x.forward[i].key < searchKey )
       x = x.forward[i];
    update[i] = x;
  }

  x = x.forward[1];      // expected position

  if( x.key == searchKey )
    x.value = newValue;  // associative structure
  else {         // key not found, do insertion:
```

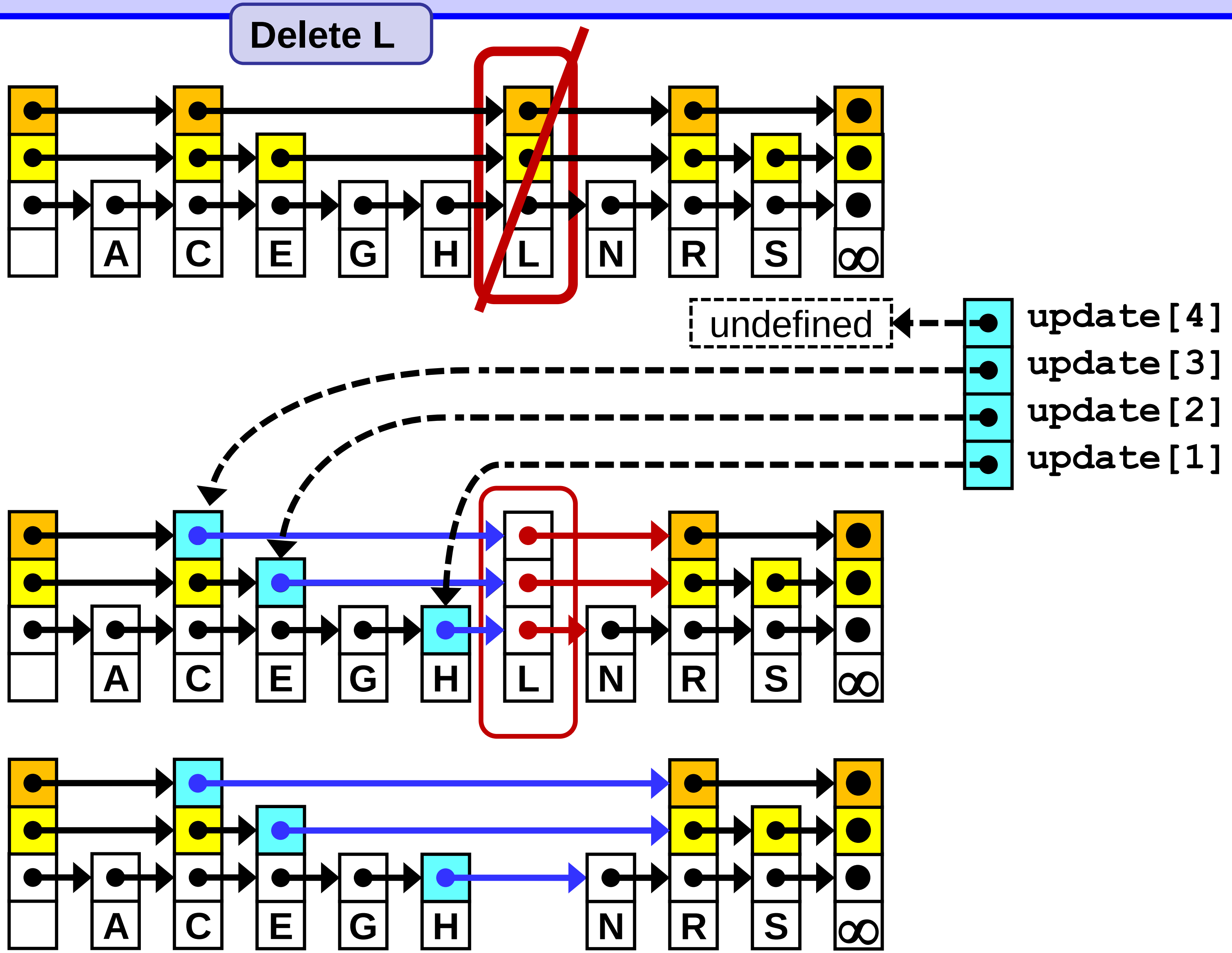
continue...

Skip list - Insert

.. continued

```
... else { // key not found, do insertion here:
  int newLevel = randomLevel( list );
  /* If newLevel is greater than the current level
  of the list, knock newLevel down so that it is only
  one level more than the current level of the list.
  In other words, increase the level of the list
  by at most 1 in each insert operation. */
  if( newLevel > list.level ) {
    if( list.level < list.MaxLevel ) list.level++;
    newLevel = list.level;
    update[newLevel] = list.header; // sentinel
  }
  // finally, physical insertion:
  Node x = makeNode( newLevel, searchKey, newValue );
  for( int i = 1; i <= newLevel; i++ ) {
    x.forward[i] = update[i].forward[i];
    update[i].forward[i] = x; }
  }
}} // of insert
```

Skip list - Delete



Skip list - Delete

Pseudocode

Deleting in a skip list is like deleting the same value independently from each list in which forward pointers of the deleted element are involved.

The algorithm registers the element's predecessor in the list,
makes the predecessor point to the element that the deleted element points to,
and finally deletes the element. It is a regular list delete operation.

```
   // update is an array of pointers to the
   // predecessors of the element to be deleted
void delete(List list, int searchKey) {
 Node x = list.header;
 for (int i = list.level; i >= 1; i--) {
  while (x.forward[i].key < searchKey)
   x = x.forward[i];
  update[i] = x;
 }
 x = x.forward[1];
 if (x.key == searchkey) {  //  go delete ...
```

continue...

Skip list - Delete

(**) If the element to be deleted is a level ***k*** node, break out of the loop when level (***k***+1) is reached. Since the code does not store the level of an element, we determine that we have exhausted the levels of an element when a predecessor element points past it, rather than to it.

.. continued

```
   for (int i = 1; i <= list.level; i++) {
     if (update[i].forward[i] != x) break; //(**)
     update[i].forward[i] = x.forward[i];
   }
   destroy_remove(x);

   /* if deleting the element causes some of the
   highest level list to become empty, decrease the
   list level until a non-empty list is encountered.*/
   while ((list.level > 1) &&
     (list.header.forward[list.level] == list.header))
        list.level--;
}} // deleted
```

Skip list - Properties

Choosing p

One might think that p should be chosen to be 0.5.
If p is chosen to be 0.5, then roughly half our elements will be level 1 nodes, 0.25 will be level 2 nodes, 0.125 will be level 3 nodes, and so on.
This will give us
  -- on average log(N) search time and
  -- on average 2 pointers per node.

However, empirical tests show that choosing p to be 0.25
results in
  -- roughly the same search time
  -- but only an average of 1.33 pointers per node,
  -- somewhat more variability in the search times.

There is a greater chance of a search taking longer than expected, but the decrease in storage overhead seems to be worth it sometimes.

Skip list - Properties

Notes on size and compexity

The average number of links in a randomized skip list with parameter p is
N · 1/(1 – p)

The average number of key comparisons in **search** and **insert**
in a randomized skip list with parameter p is on average

$$-\log_p (N) / 2p = \log(N) * (-1) * (2p * \log (p))^{-1} = \log(N) / (2p * \log (1/p))$$

Experimental time comparisons:

	Search		Insert		Delete	
Skip list	0.051	**(1.0)**	0.065	**(1.0)**	0.059	**(1.0)**
AVL tree	0.046	(0.91)	0.100	(1.55)	0.085	(1.46)
2-3 tree	0.054	(1.05)	0.210	(3.2)	0. 21	(3.65)
Splay tree	0.490	(9.6)	0.510	(7.8)	0.53	(9.0)

Times in ms on some antiquated HW [Pugh, 1990]

Notes on compexity

The probabilistic analysis of skip lists is rather advanced.
However, it can be shown that the expected times of
search, **insert**, **delete** are all

$\mathbf{O((1/p)\log_{1/p} n)=O(\log n).}$

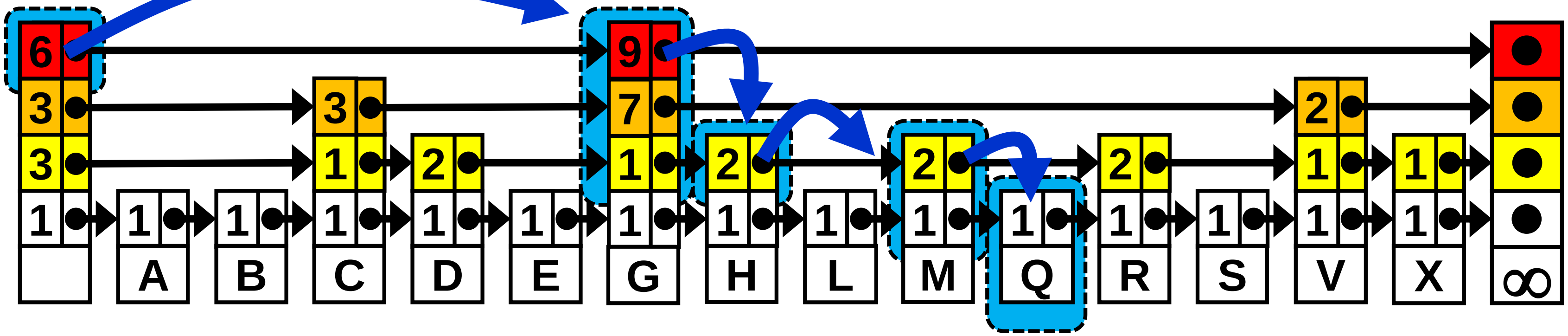
The choice of p determines the variability of these search times.

Intuitively, decreasing p will increase the variability since it will decrease the number of higher-level elements (i.e., the number of "skip" nodes in the list).

The Pugh paper contains a number of graphs that show the probability of a search taking significantly longer than expected for given values of p. For example, if p is 0.5 and there are more than 256 elements in the list, the chances of a search taking 3 times longer than expected are less than 1 in a million. If p is decreased to 0.25, the chances rise to about 1 in a thousand.

Skip list - Index access

list[10]== 'Q' (10 = 6 + 1 + 2 + 1)



Array-like property -- random element access

Supplement each forward pointer with its
"length" = 1 + number of the list elements it skips.

A k-th list element can be acessed in expected O(log n) time.

Search, Insert, Delete are analogous to the "plain" variant. The length of the affected pointers has to be updated after each Insert or Delete. Asymptotic complexity remains the same in all cases -- O(log n).

Skip list - References

Rich sources

erikdemaine.org/

- Erik Demaine's presentation at MIT
  http://videolectures.net/mit6046jf05_demaine_lec12/

- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998

- William Pugh: *Skip lists: A probabilistic alternative to balanced trees*. Communications of the ACM, 33(6):668–676, 1990.

- William Pugh: *A Skip List Cookbook* [http://cglab.ca/~morin/teaching/5408/refs/p90b.pdf]

- Bradley T. Vander Zanden: [http://web.eecs.utk.edu/~huangj/CS302S04/notes/skip-lists.html]

Also: java.util.concurrent.ConcurrentSkipListSet<E>

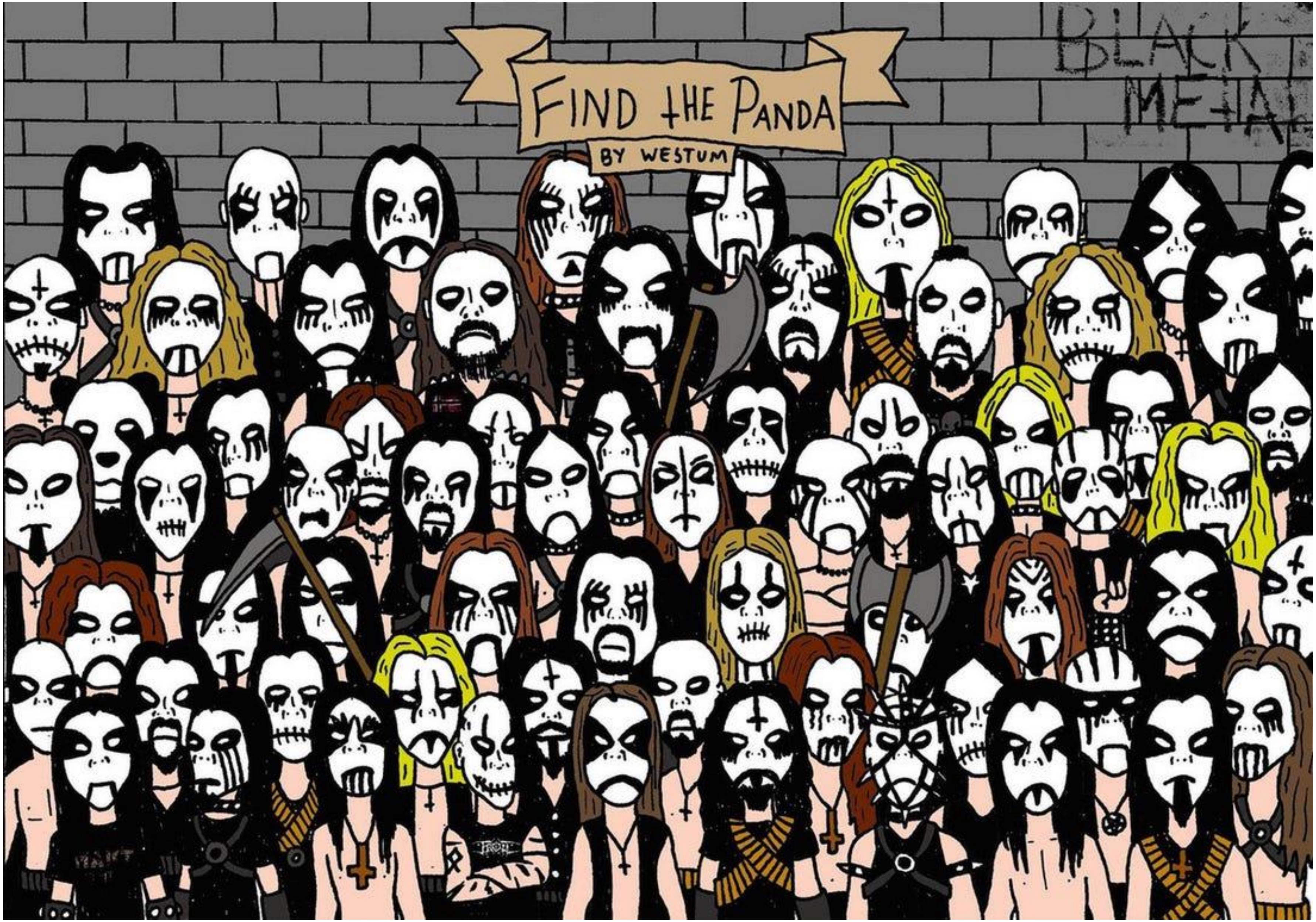
Kvízová pauza

1
11
21
1112
3112
211213
?

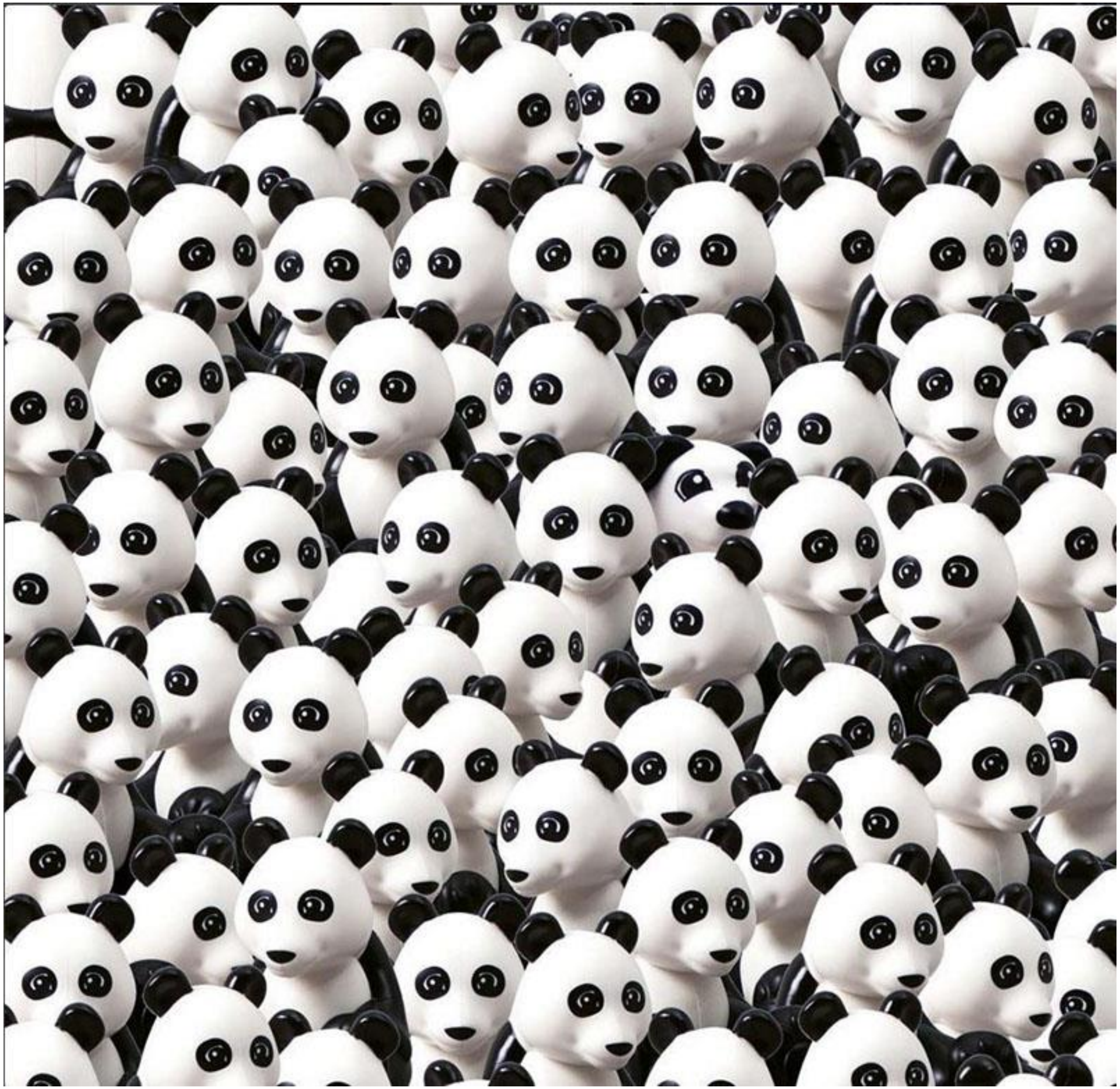
22	13	9	7
	9	?	2
		5	2
			3

1111 = Č
2222 = O
3333 = D
4444 = Š
5555 = ?

Najděte pandu

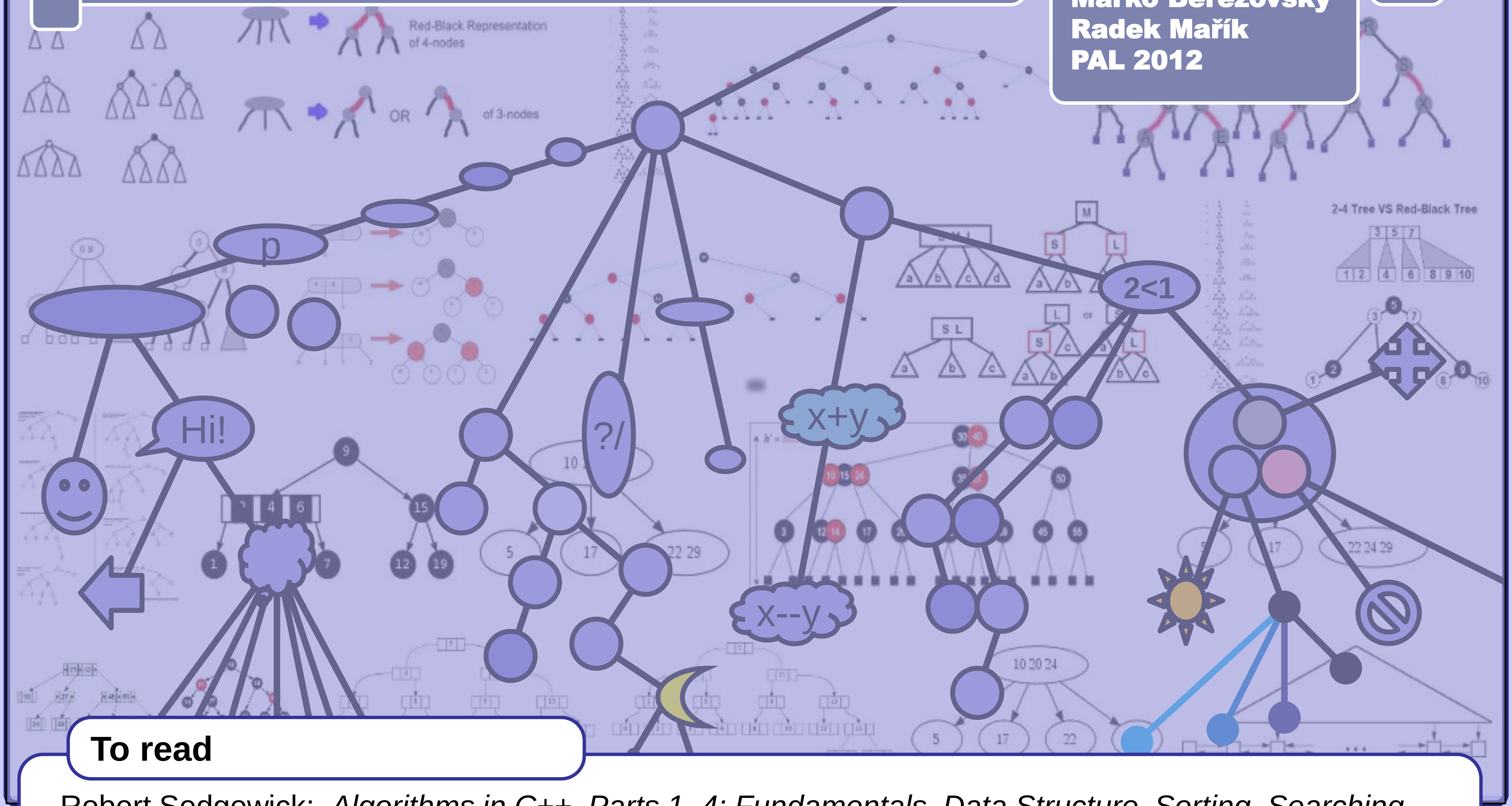


Najděte psa



B and B+ search tree

Marko Berezovský
Radek Mařík
PAL 2012



To read

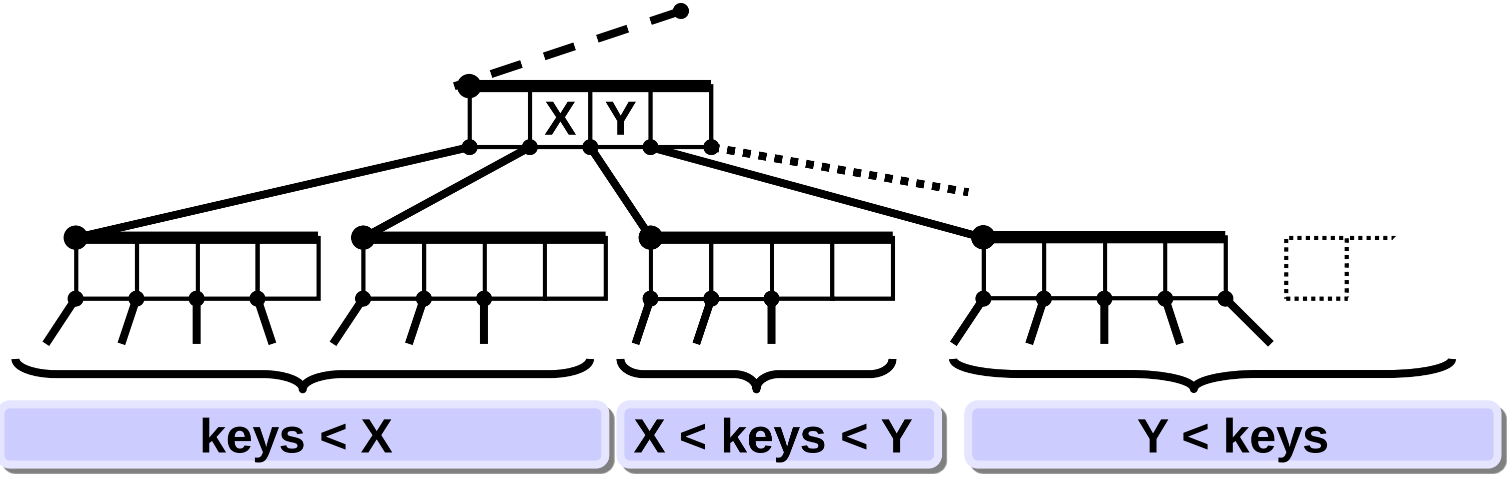
- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998
- http://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf
- (CLRS) Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*, 3rd ed., MIT Press, 2009

See PAL webpage for references

B tree

B-tree -- Rudolf Bayer, Edward M. McCreight, 1972

- All lengths of paths from the root to the leaves are equal.
- B-tree is perfectly balanced. Keys in the nodes are kept sorted.
- Fixed parameter k > 1 dictates the same size of all nodes.
- Each node except for the root contains at least k and at most 2k keys and if it is not a leaf it has at least k+1 and at most 2k+1 children.
- The root may contain any number of keys from 1 to 2k. If it is not simultaneously a leaf it has at least 2 and at most 2k+1children.



B tree

Alternate specification

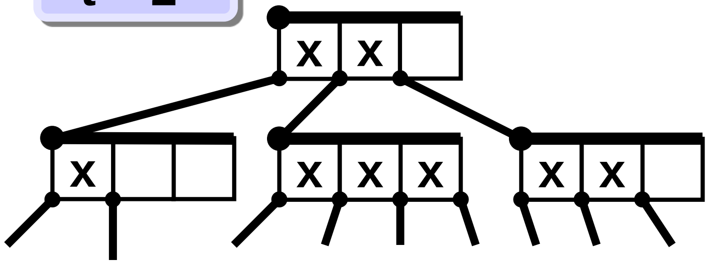
Cormen et al. 1990: B-tree degree:

Nodes have lower and upper bounds on the number of keys they can contain.
We express these bounds in terms of a fixed integer t ≥ 2 called
the minimum degree of the B-tree:

a. Every node other than the root must have at least t–1 keys.
   Every internal node other than the root thus has at least t children.
   If the tree is nonempty, the root must have at least one key.

b. Every node may contain at most 2t–1 keys.
   Therefore, an internal node may have at most 2t children.

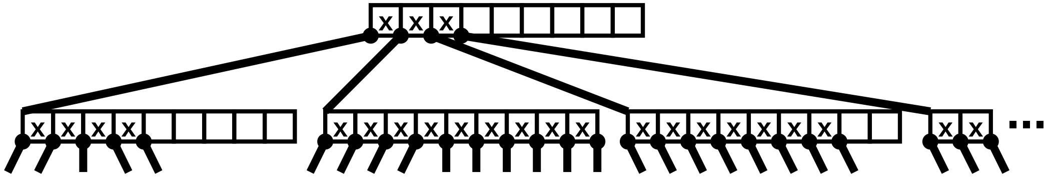
t = 2



min keys = 1
children = 2

max keys = 3
children = 4

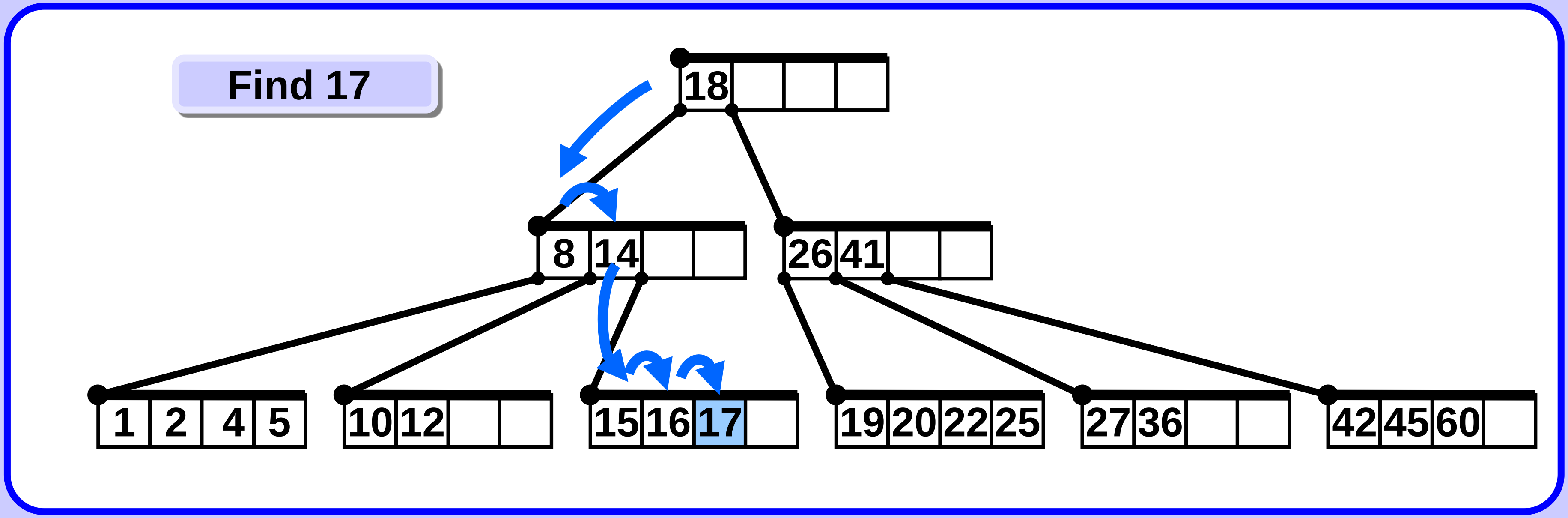
t = 5



min keys = 4
children = 5

max keys = 9
children = 10

B tree - Find



Search in the node is sequential (or binary or other...).

If the node is not a leaf and the key is not in the node
then the search continues in the appropriate child node.

If the node is a leaf and the key is not in the node
then the key is not in the tree.

B tree - Update strategies

Update strategies:

1. **Multi phase strategy**: “Solve the problem when it appears”.
   First insert or delete the item and only then rearrange the tree if necessary.
   This may require additional traversing up to the root.

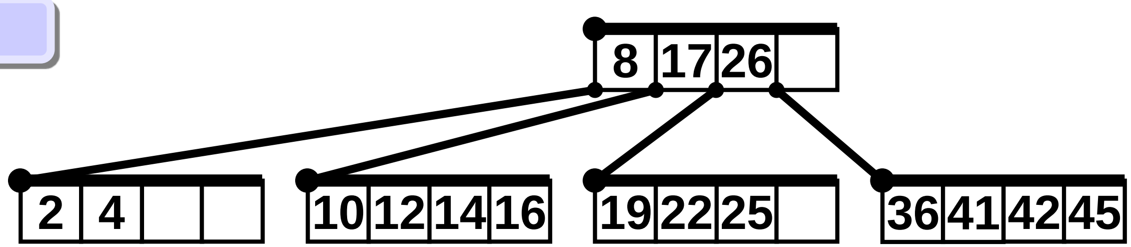
2. **Single phase strategy**: “Avoid future problems”.
   Travel from the root to the node/key which is to be inserted or deleted and during the travel rearrange the tree to prevent the additional traversing up to the root.

B tree - Insert

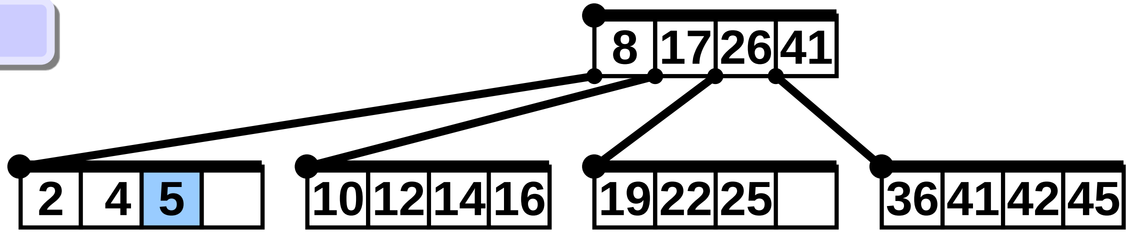
Multi phase strategy

Insert rules I

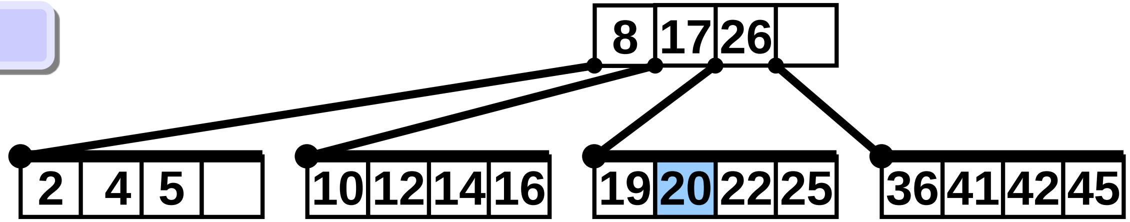
B-tree



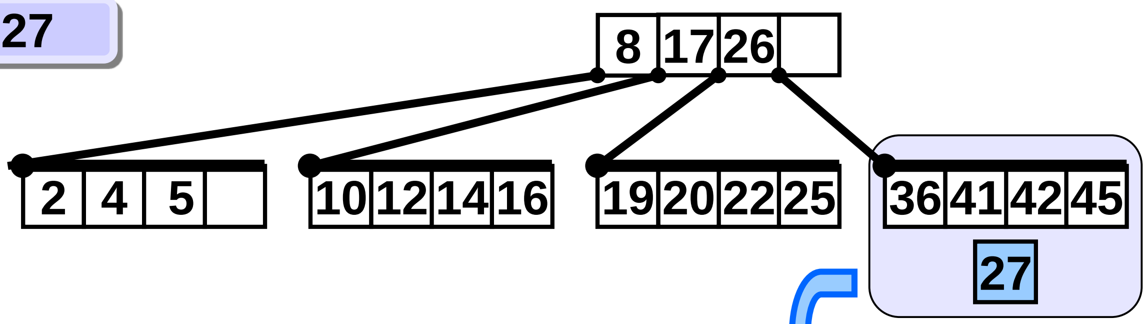
Insert 5



Insert 20



Insert 27

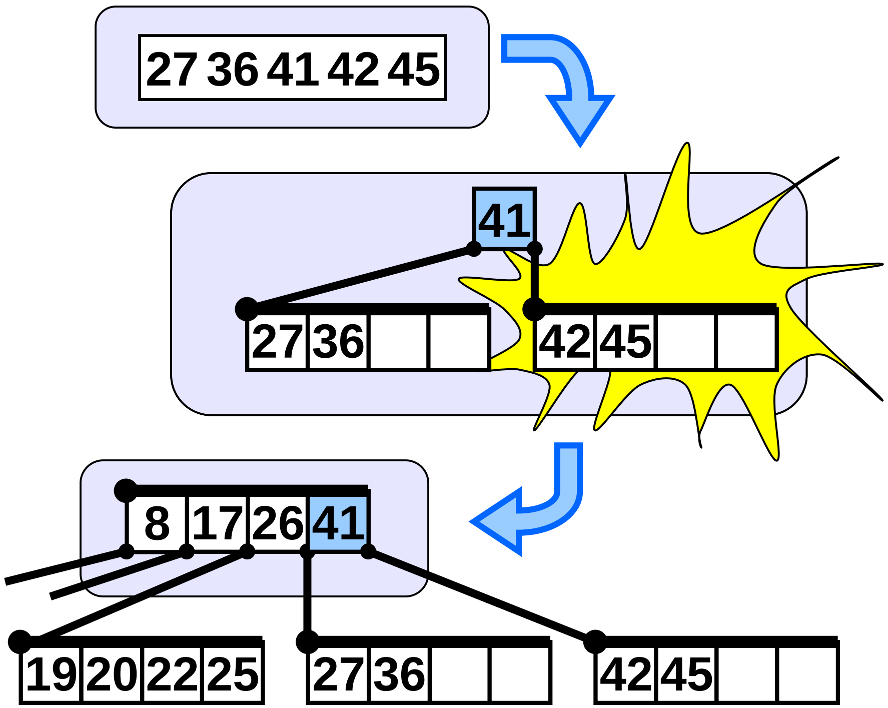


Sort keys outside the tree.

Select median,
create new node,
move to it the values
bigger than the median.

Try to insert the median
into the parent node.

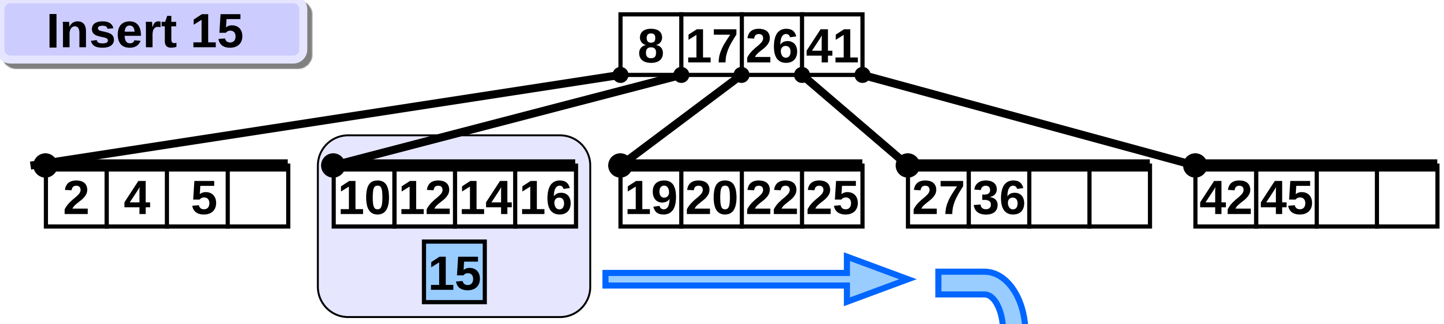
Success.



B tree - Insert

Multi phase strategy

Insert 15

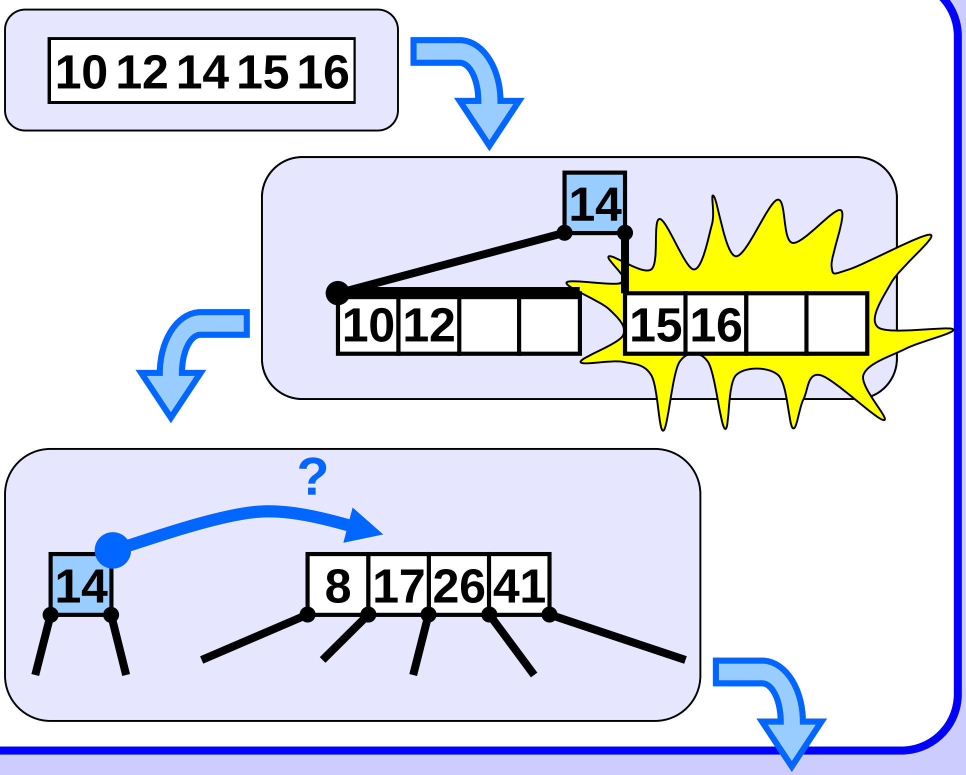


Sort keys outside the tree.

Select median,
create new node,
move to it the values
bigger than the median.

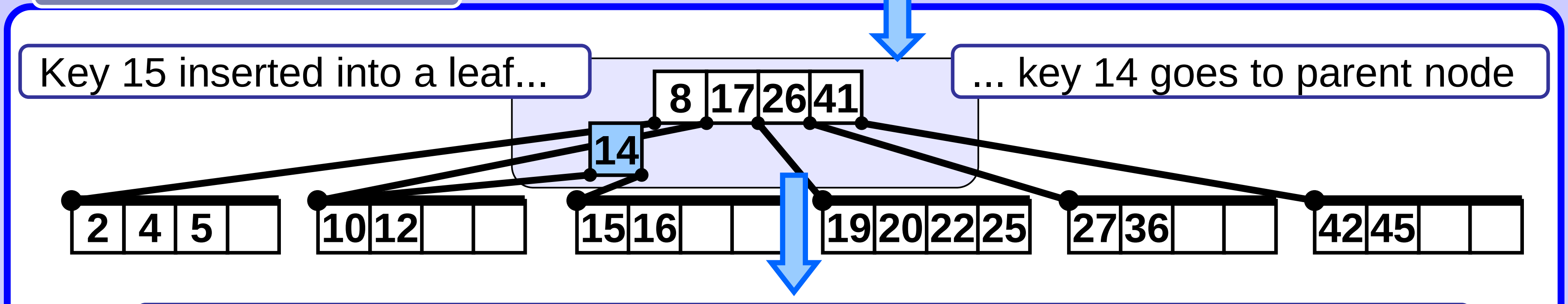
Try to insert the median
into the parent node.

Success?



B tree - Insert

Multi phase strategy



Key 15 inserted into a leaf...

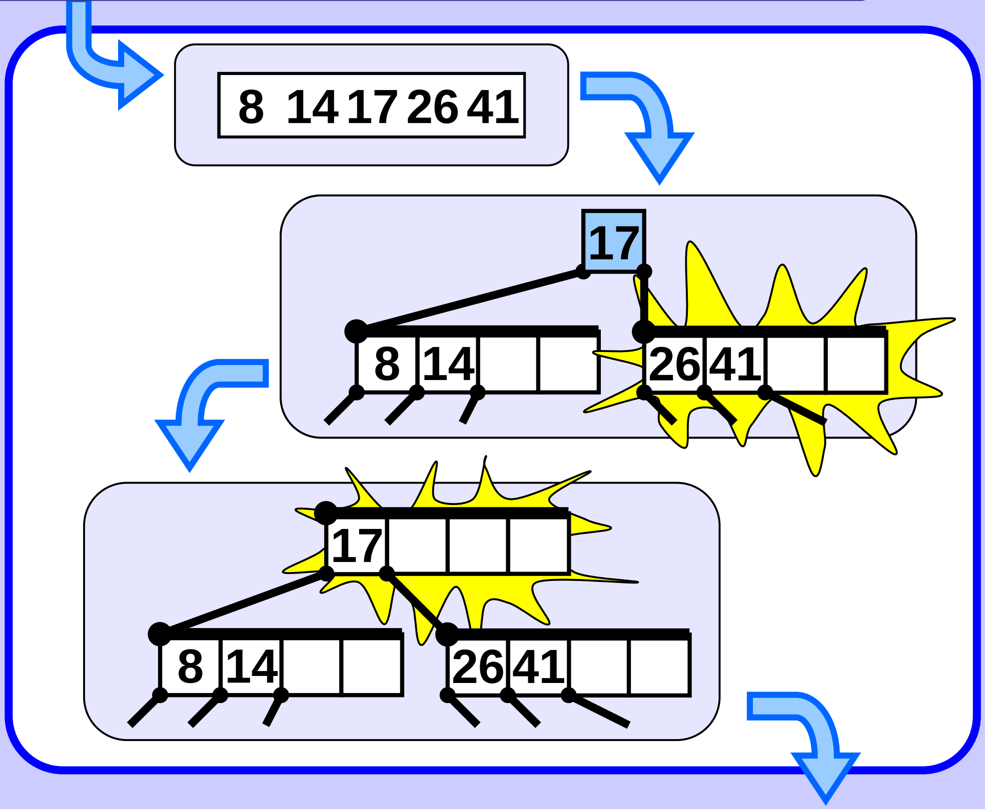
... key 14 goes to parent node

The parent node is full – repeat the process analogously.

Sort values

Select median, create new node, move to it the values bigger than the median together with the corresponding references.

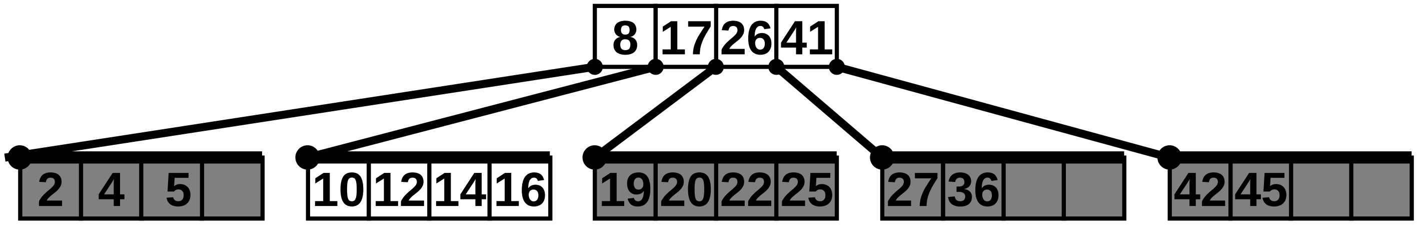
Cannot propagate the median into the parent (there is no parent), create a new root and store the median there.



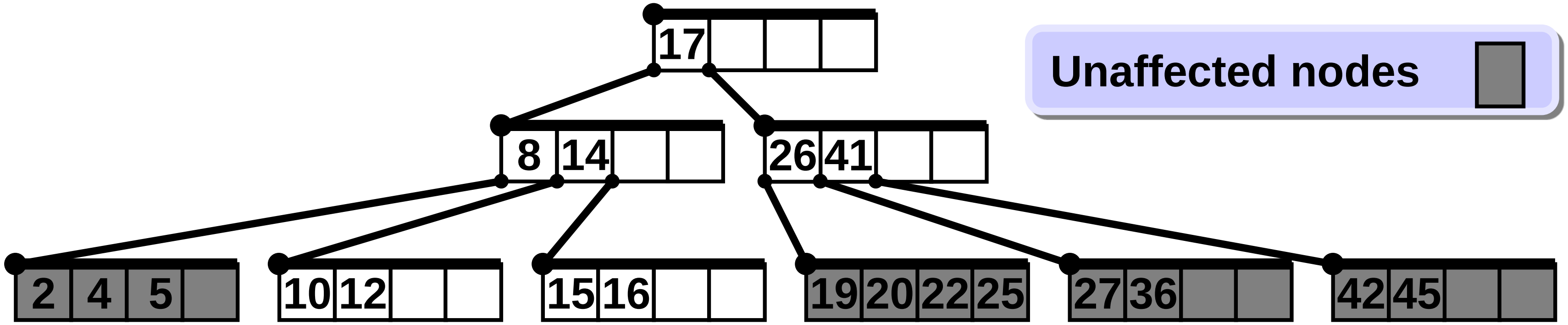
B tree - Insert

Multi phase strategy

Recapitulation - insert 15



Insert 15



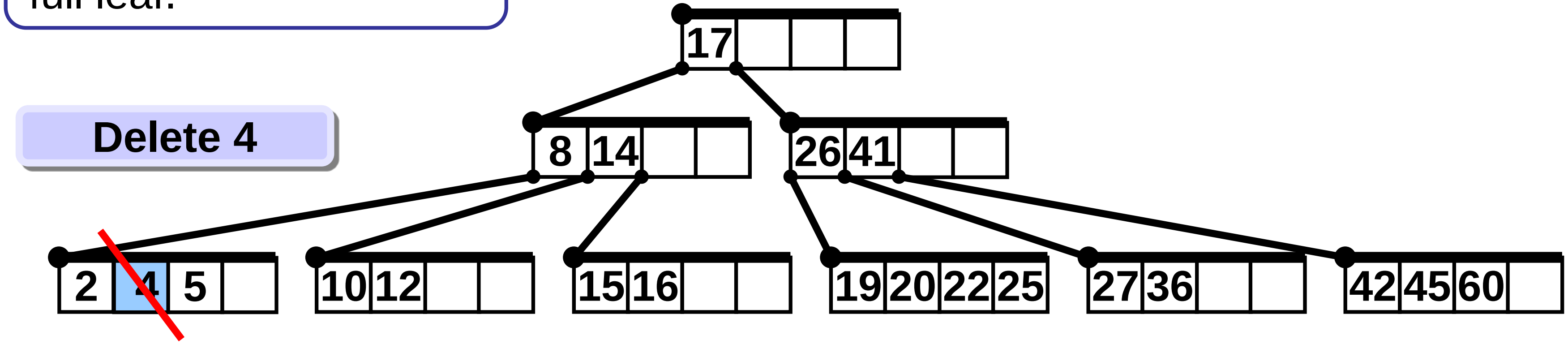
Each level acquired one new node, a new root was created too, the tree **grows upwards** and remains perfectly balanced.

B tree - Delete

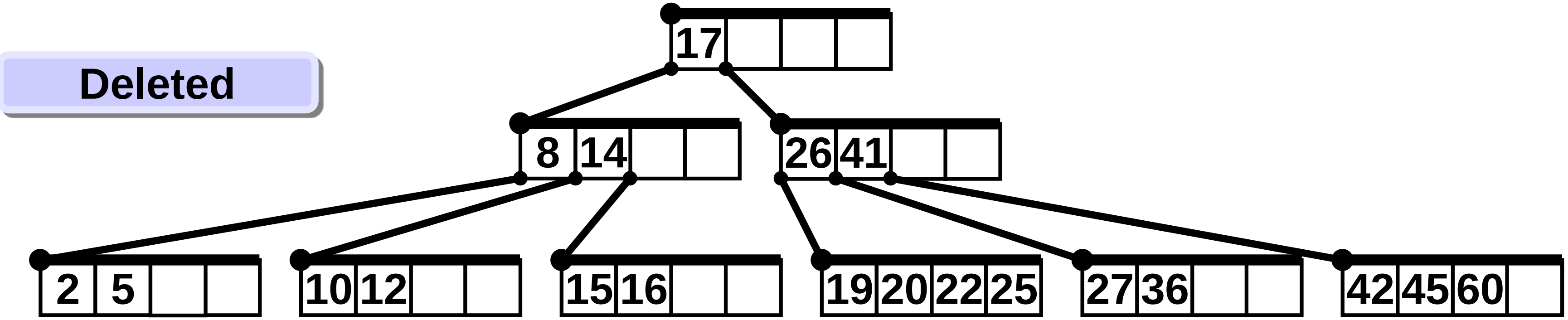
Multi phase strategy

Delete in a sufficiently full leaf.

Delete 4



Deleted



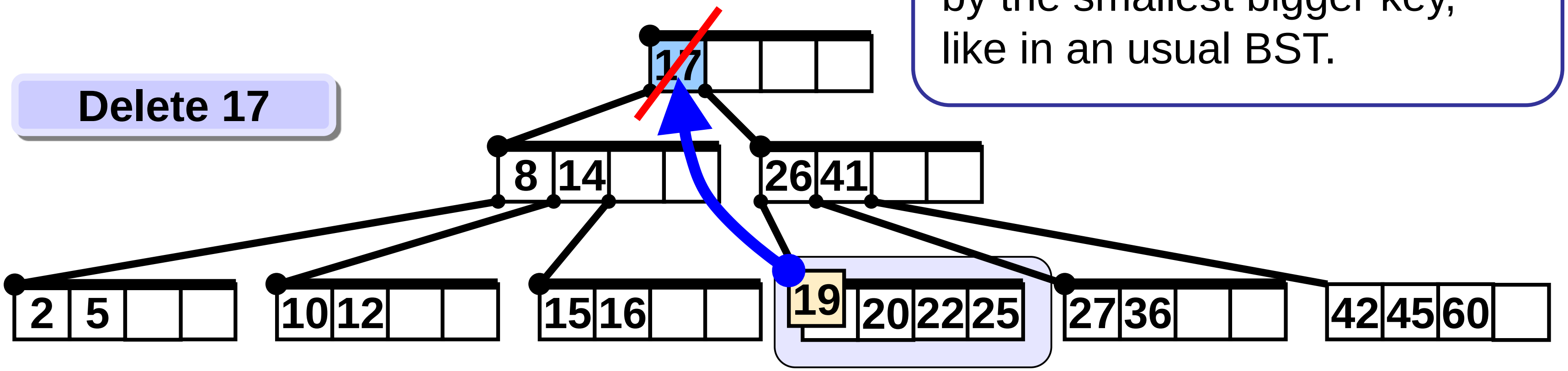
B tree - Delete

Multi phase strategy

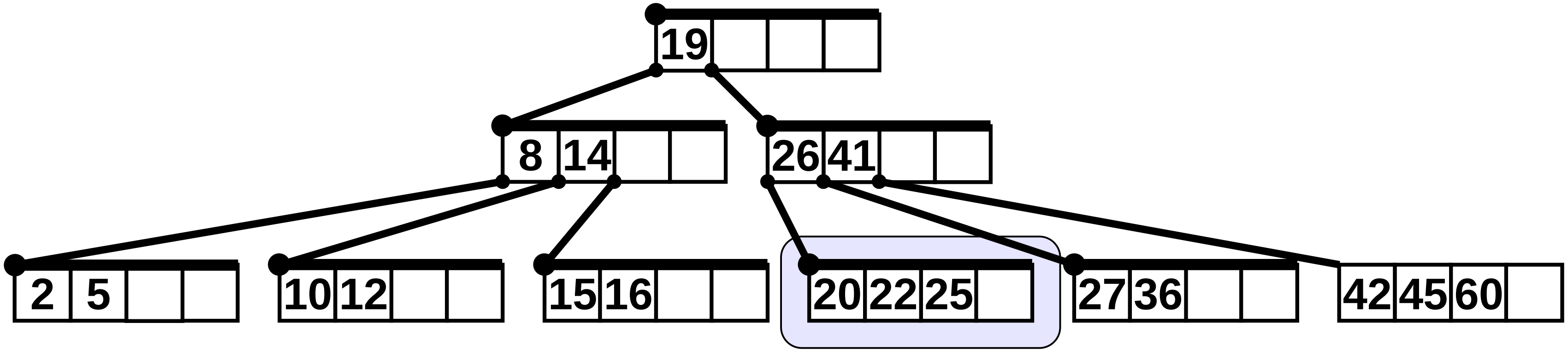
Delete in an internal node

The deleted key is substituted by the smallest bigger key, like in an usual BST.

Delete 17



The smallest bigger key is always in a leaf in a B-tree.
If the leaf is sufficiently full the delete operation is complete.



B tree - Delete

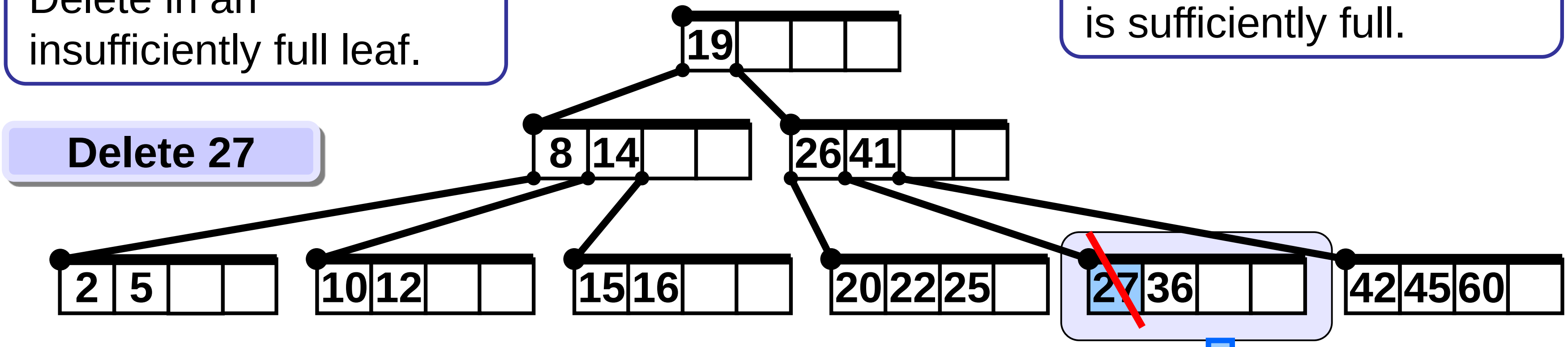
Multi phase strategy

Delete rules III

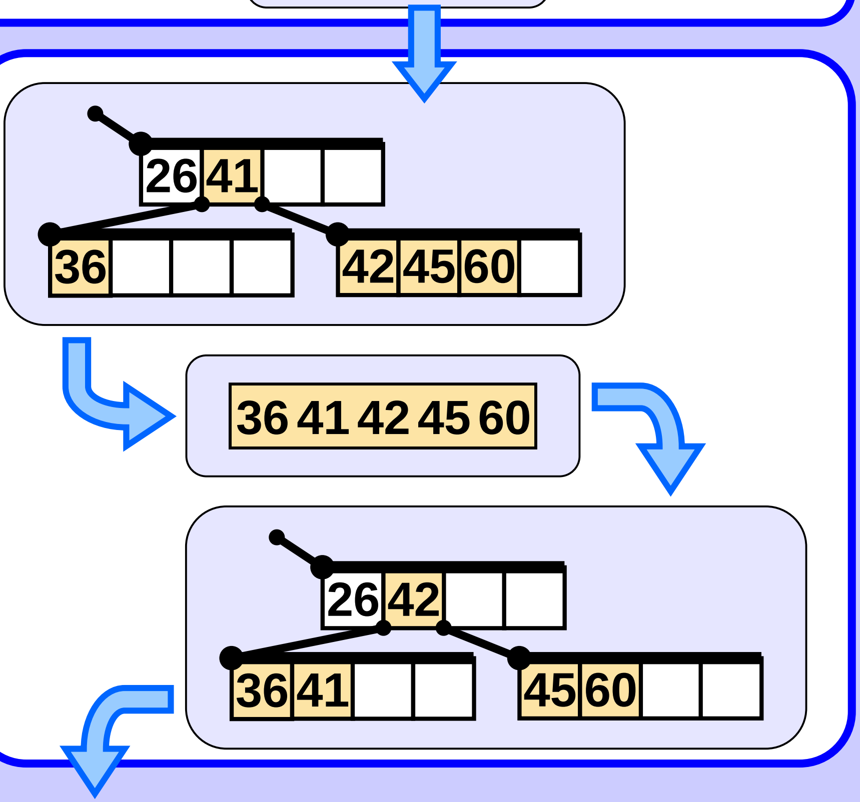
Delete in an insufficiently full leaf.

The neighbour leaf is sufficiently full.

Delete 27

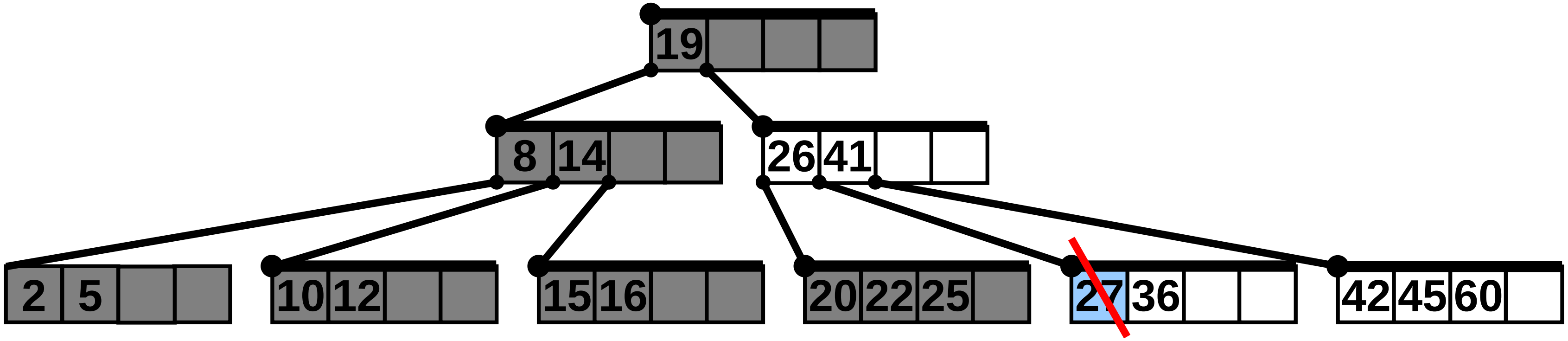


Merge the keys of the two leaves with the dividing key in the parent into one sorted list.

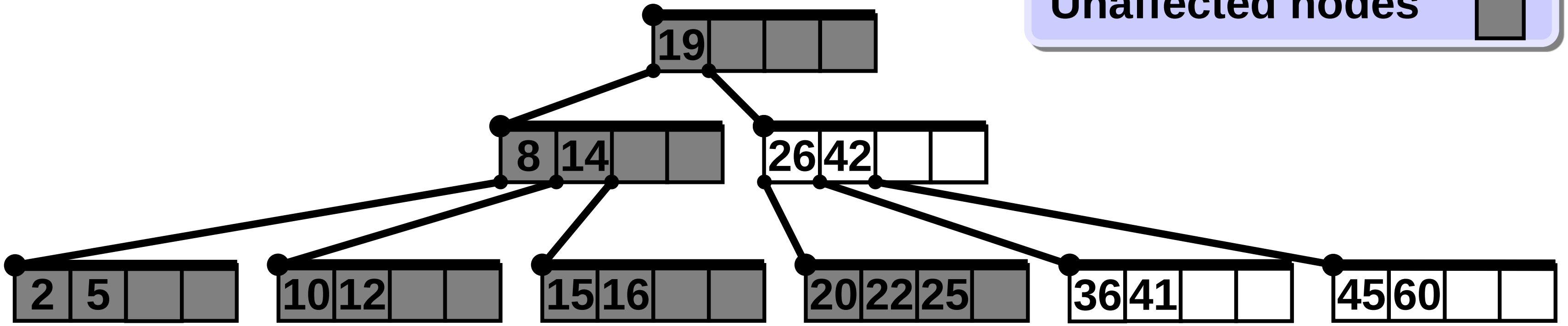


Insert the median of the sorted list into the parent and distribute the remainig keys into the left and right children of the median.

Recapitulation - delete 27



27 correctly deleted



Unaffected nodes ▨

B tree - Delete

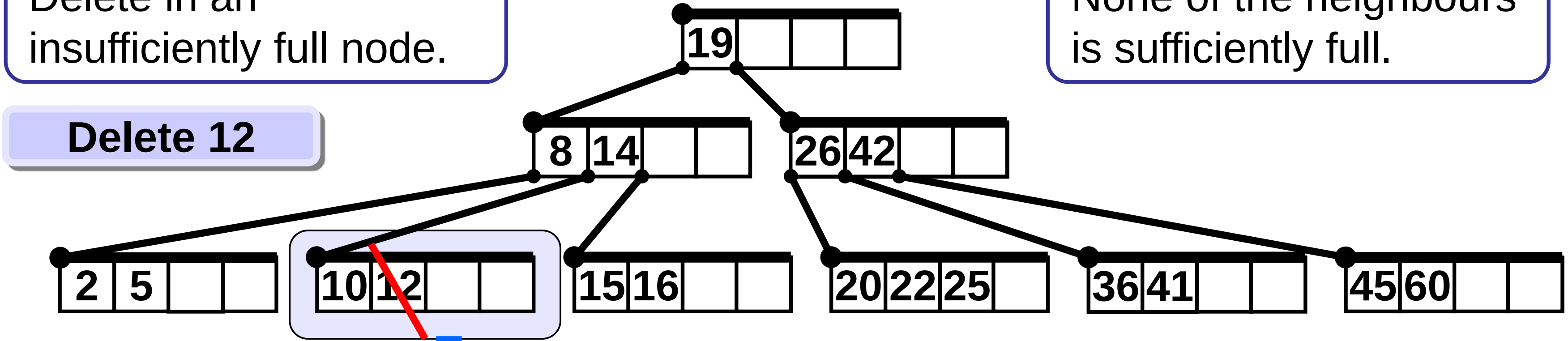
Multi phase strategy

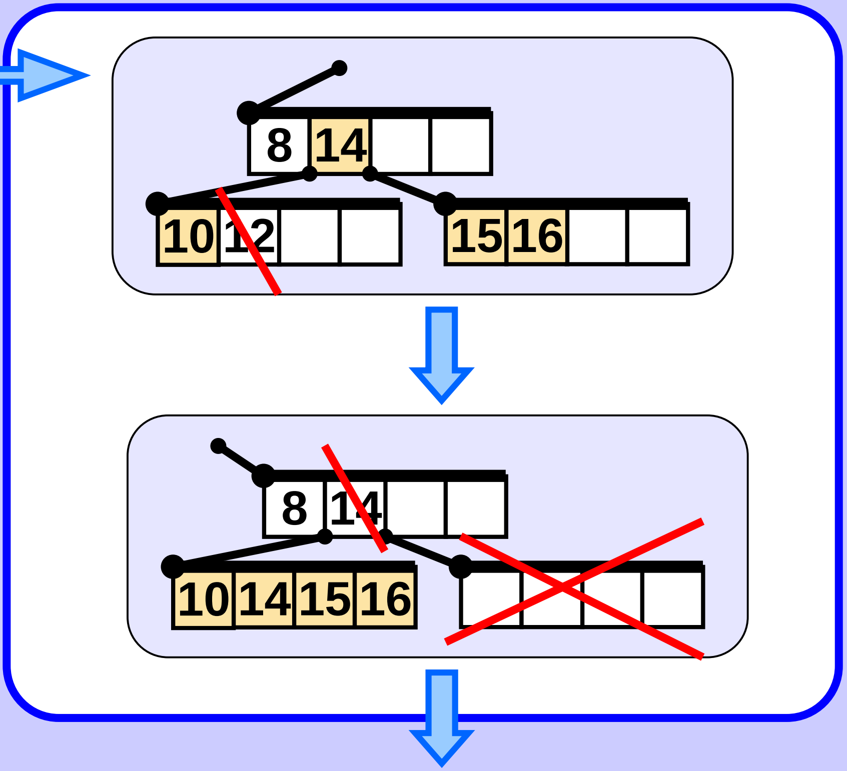
Delete rules IV

Delete in an insufficiently full node.

None of the neighbours is sufficiently full.

Delete 12



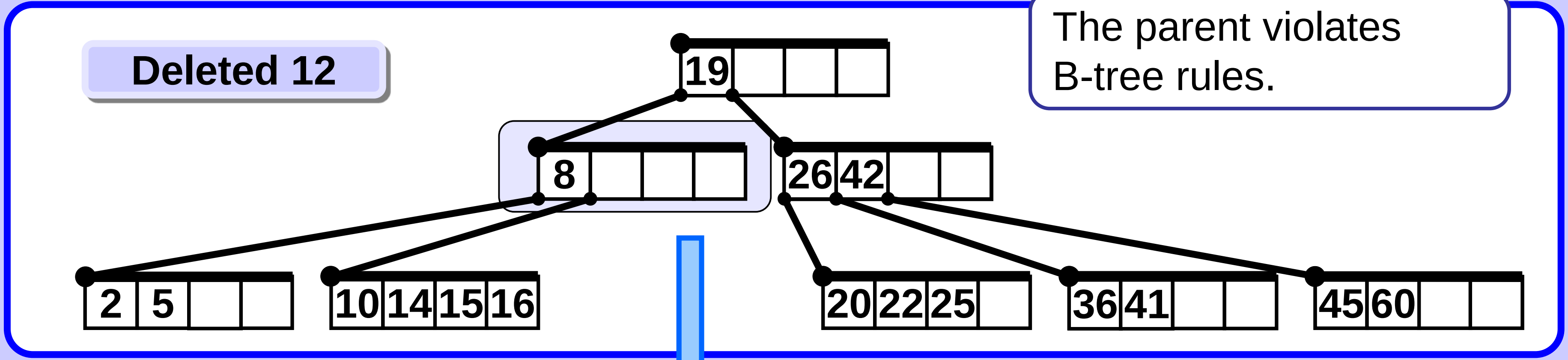


Merge the keys
  of the node
  and of one of the neighbours
  and the median in the parent
into one sorted list.
Move all these keys to the original node,
delete the neighbour, remove the original
median and associated reference
from the parent.

B tree - Delete

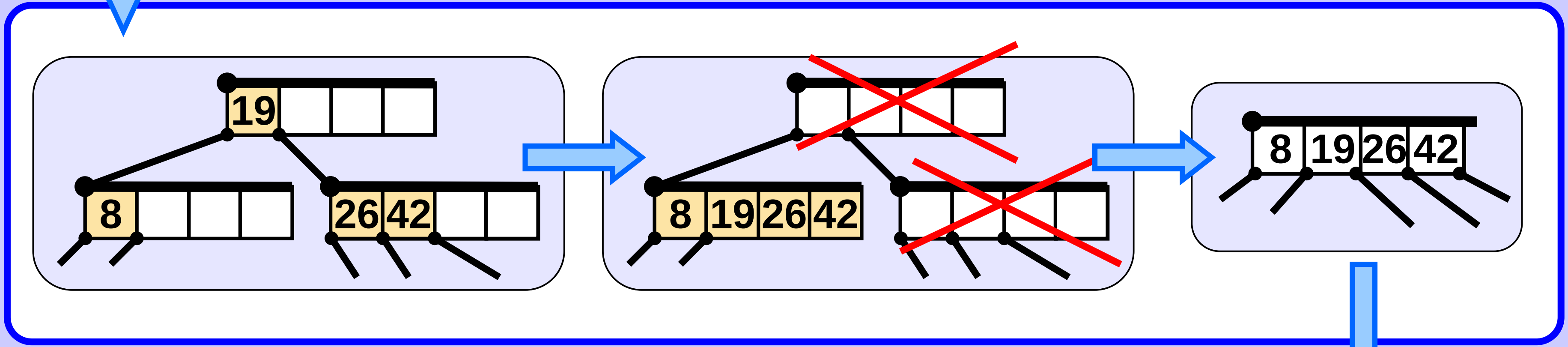
Multi phase strategy

Delete rules IV



The parent violates B-tree rules.

If the parent of the deleted node is not sufficiently full apply the same deleting strategy to the parent and continue the process towards the root until the rules of B-tree are satisfied.

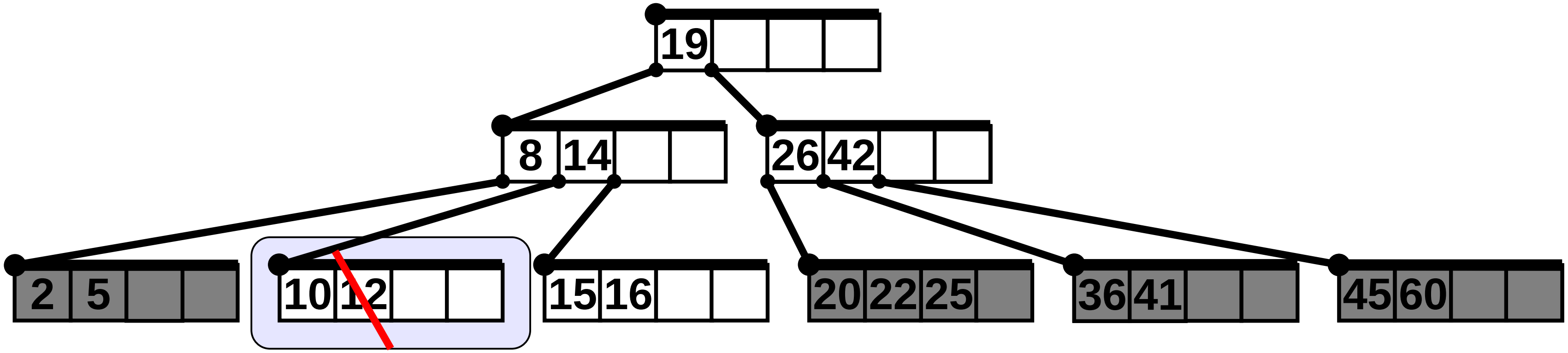


B tree

Multi phase strategy

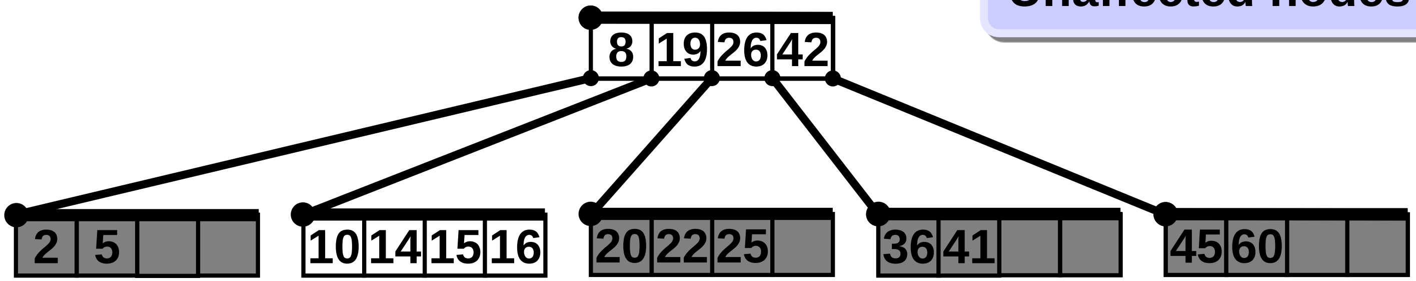
Delete rules IV

Recapitulation - delete 12



Key 12 was deleted and the tree was reconstructed accordingly.

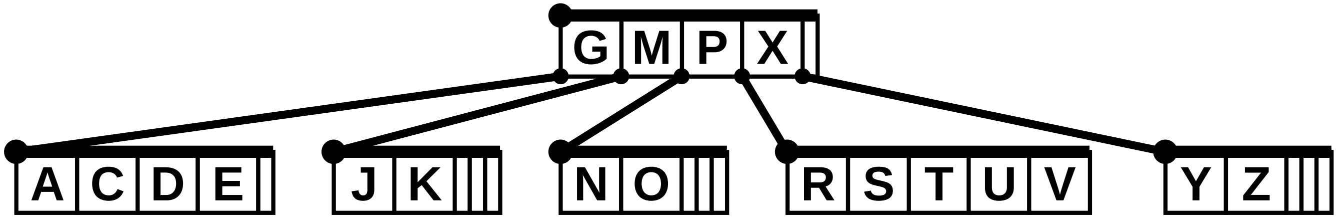
Unaffected nodes ▨



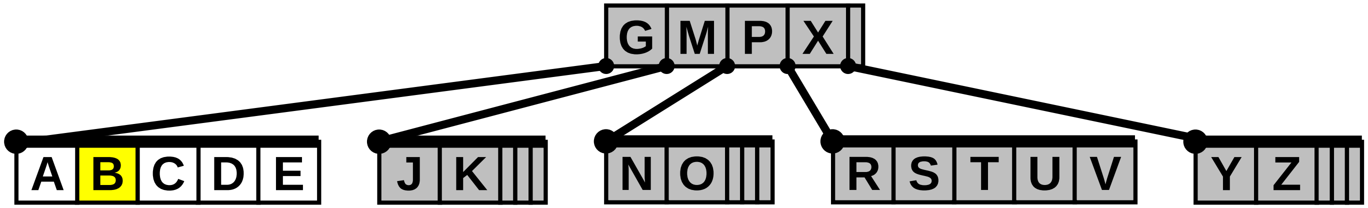
B tree - Insert

Single phase strategy

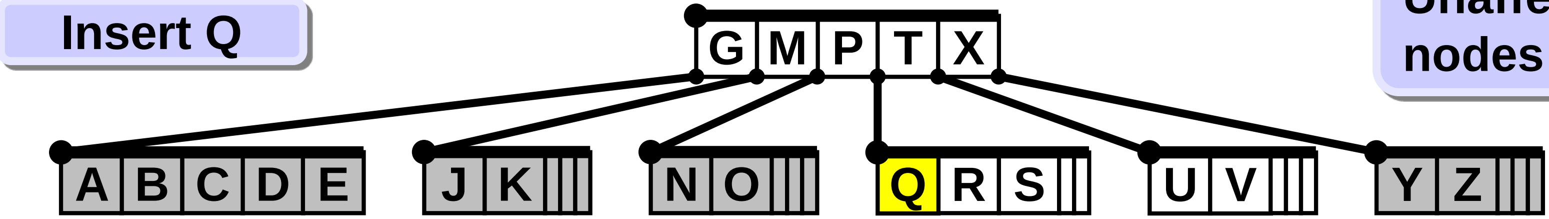
Cormen et al. 1990, t = 3, minimum degree 3, max degree = 6, minimum keys in node = 2, maximum keys in node = 5.



Insert B



Insert Q

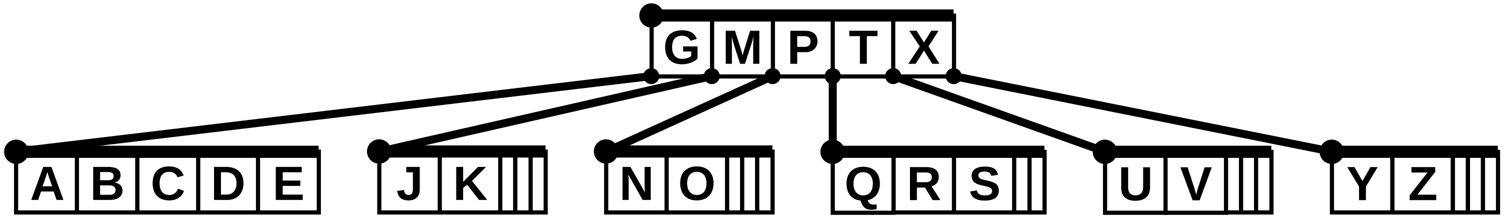


Unaffected nodes

B tree - Insert

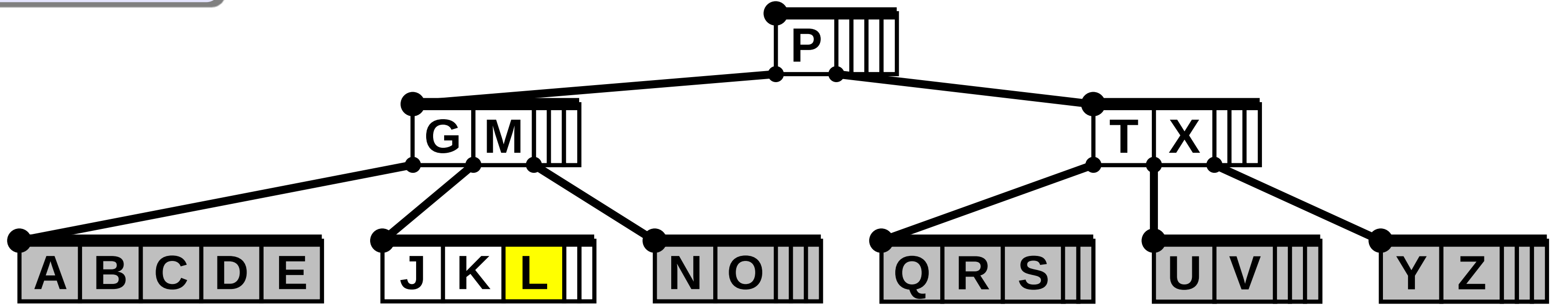
Single phase strategy

Example



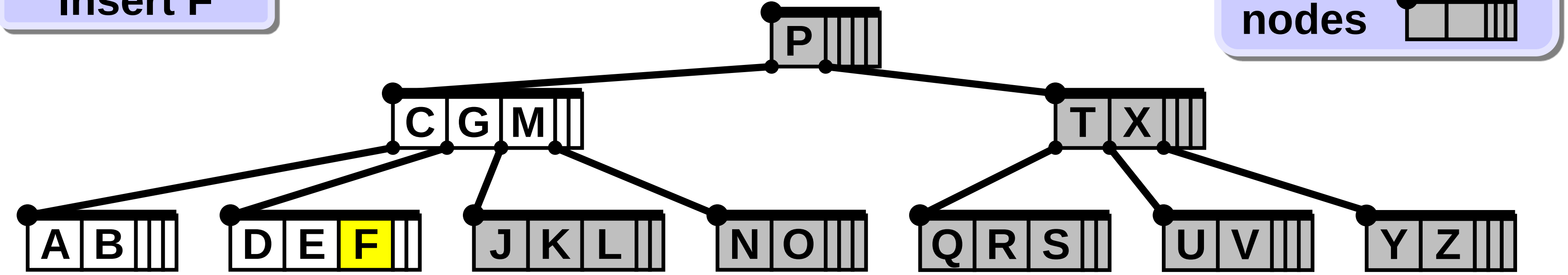
Single phase: Split the root, because it is full, and then continue downwards inserting L

Insert L



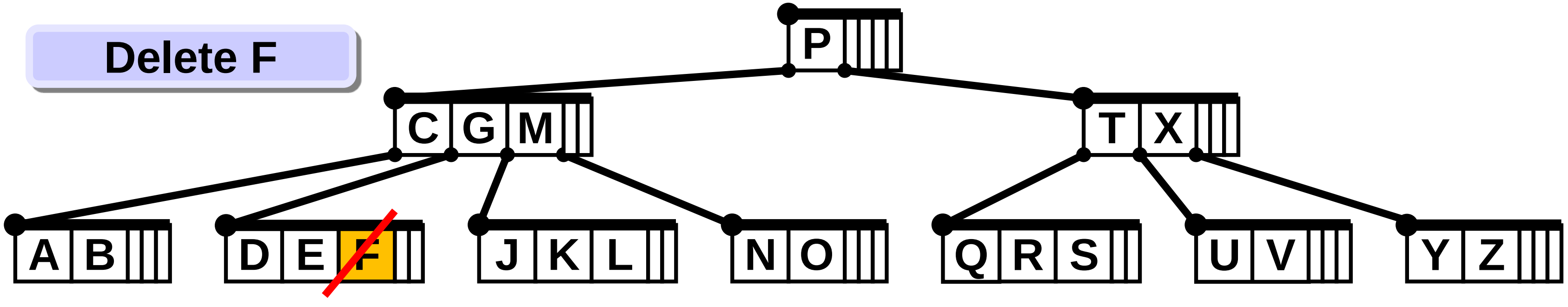
Insert F

Unaffected nodes



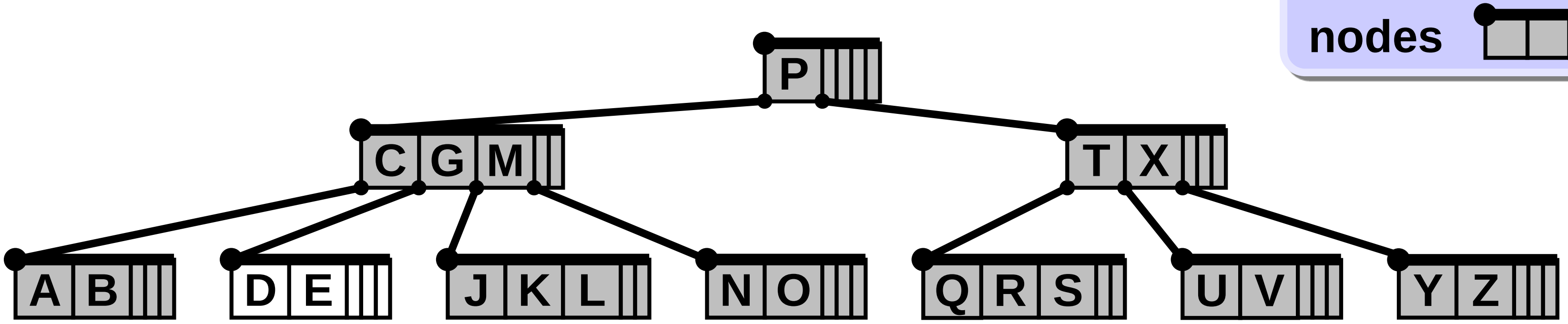
Single phase strategy

Delete F



1. If the key **k** is in node **X** and **X** is a leaf, delete the key **k** from **X**.

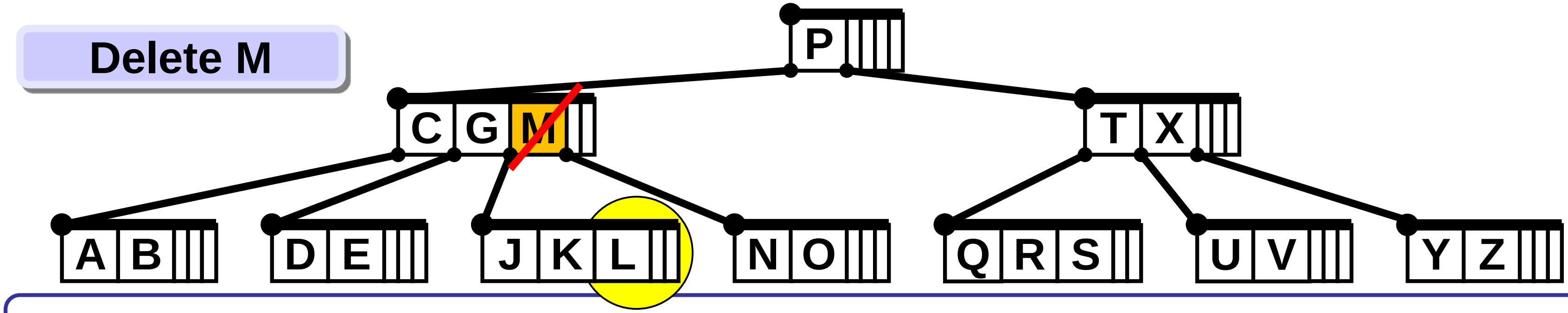
Unaffected nodes ▨



B tree - Delete

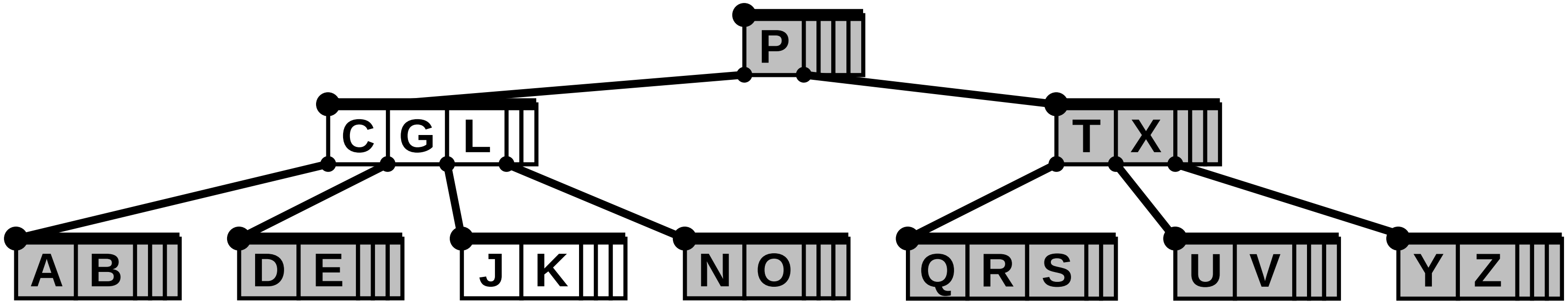
Single phase strategy

Delete M

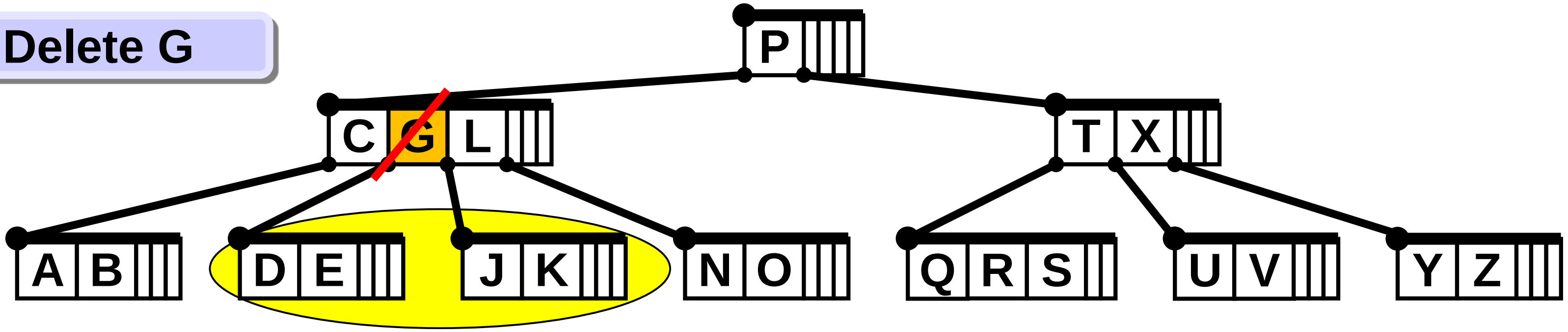


2. If the key **k** is in node **X** and **X** is an internal node, do the following:

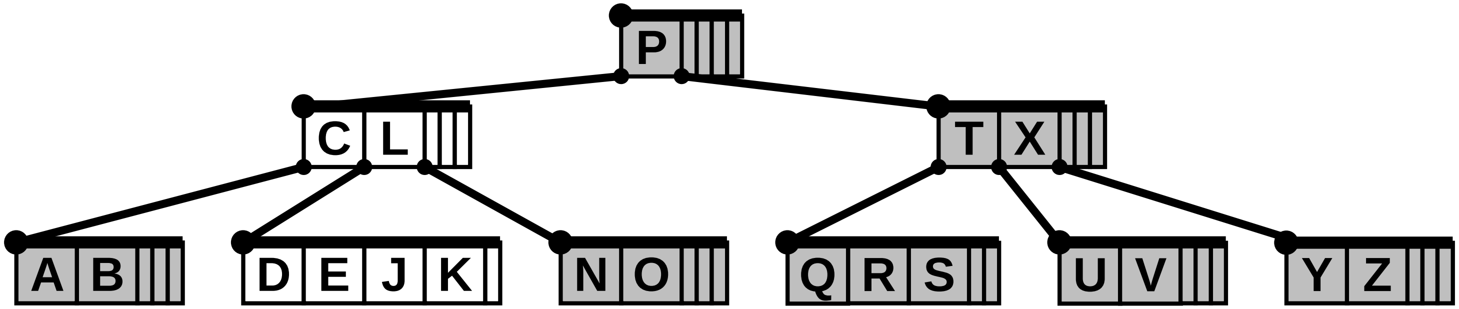
2a. If the child **Y** that precedes **k** in node **X** has at least t keys, then find the predecessor $\mathbf{k_p}$ of **k** in the subtree rooted at **Y**. Recursively delete $\mathbf{k_p}$, and replace **k** by $\mathbf{k_p}$ in **X**. (We can find $\mathbf{k_p}$ and delete it in a single downward pass.)
2b. If **Y** has fewer than t keys, then, symmetrically, examine the child **Z** that follows **k** in node **X** and continue as in 2a.



Delete G

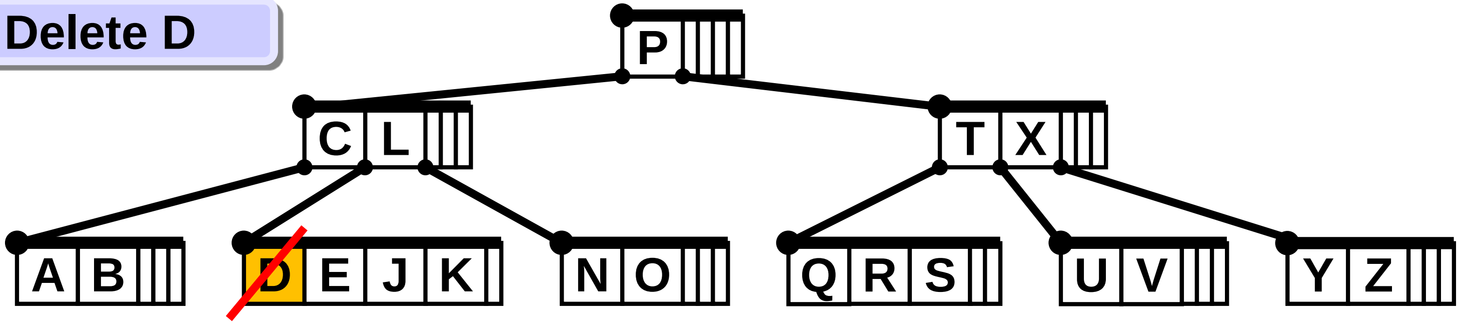


2c. Otherwise, i.e. if both **Y** and **Z** have only t–1 keys, merge **k** and all of **Z** into **Y**, so that **X** loses both **k** and the pointer to **Z**, and **Y** now contains 2t–1 keys. Then free **Z** and recursively delete **k** from **Y**.

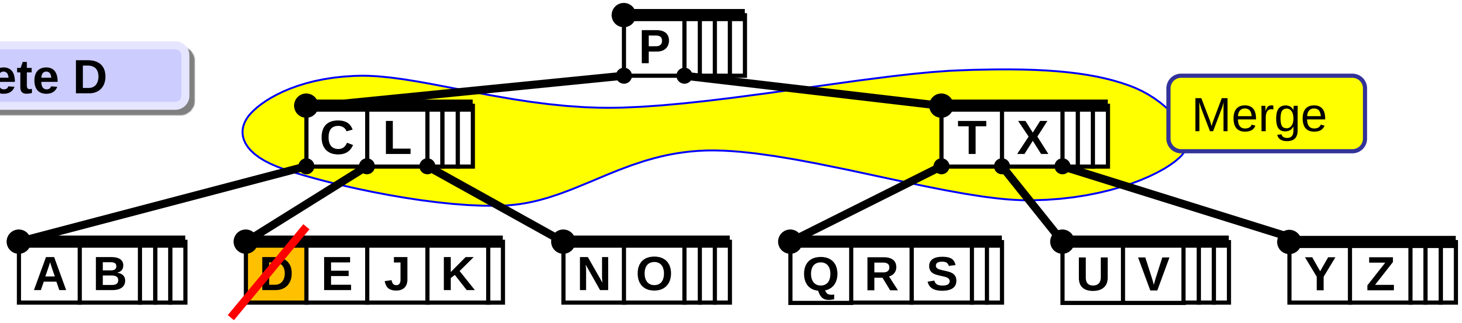


3. If the key **k** is not present in internal node **X**, determine the child **X.c** of **X**.
X.c is a root of such subtree that contains **k**, if **k** is in the tree at all.
If **X.c** has only t–1 keys, execute step 3a or 3b as necessary
to guarantee that we descend to a node containing at least t keys.
Then continue by recursing on the appropriate child of **X**.

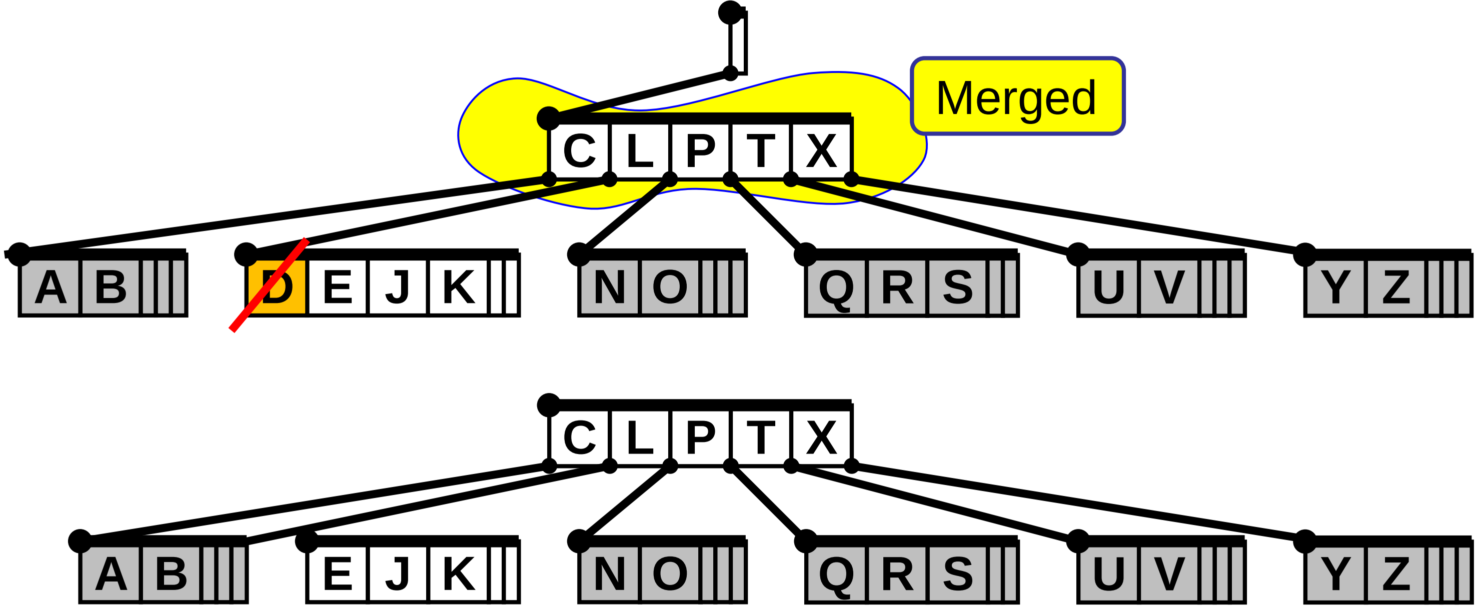
Delete D



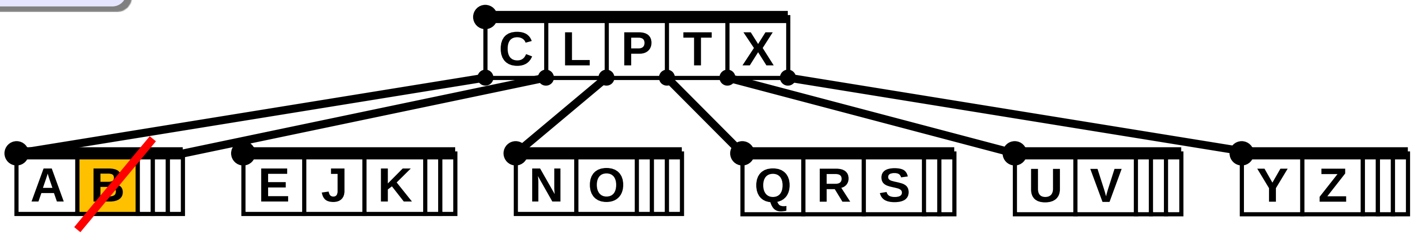
Delete D



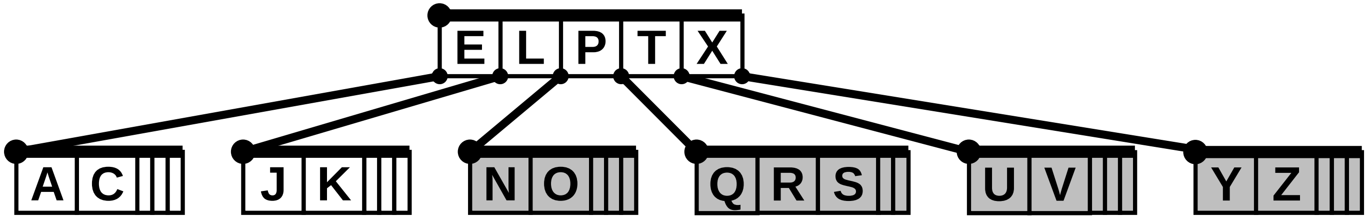
3a. If **X.c** and both of **X.c** 's immediate siblings have t–1 keys, merge **X.c** with one sibling, which involves moving a key from **X** down into the new merged node to become the median key for that node.



Delete B



3b. If **X.c** has only t–1 keys but has an immediate sibling with at least t keys, give **X.*c*** an extra key by moving a key from **X** down into **X.c**, moving a key from**X.c** 's immediate left or right sibling up into **X**, and moving the appropriate child pointer from the sibling into **X.c**.



B+ tree

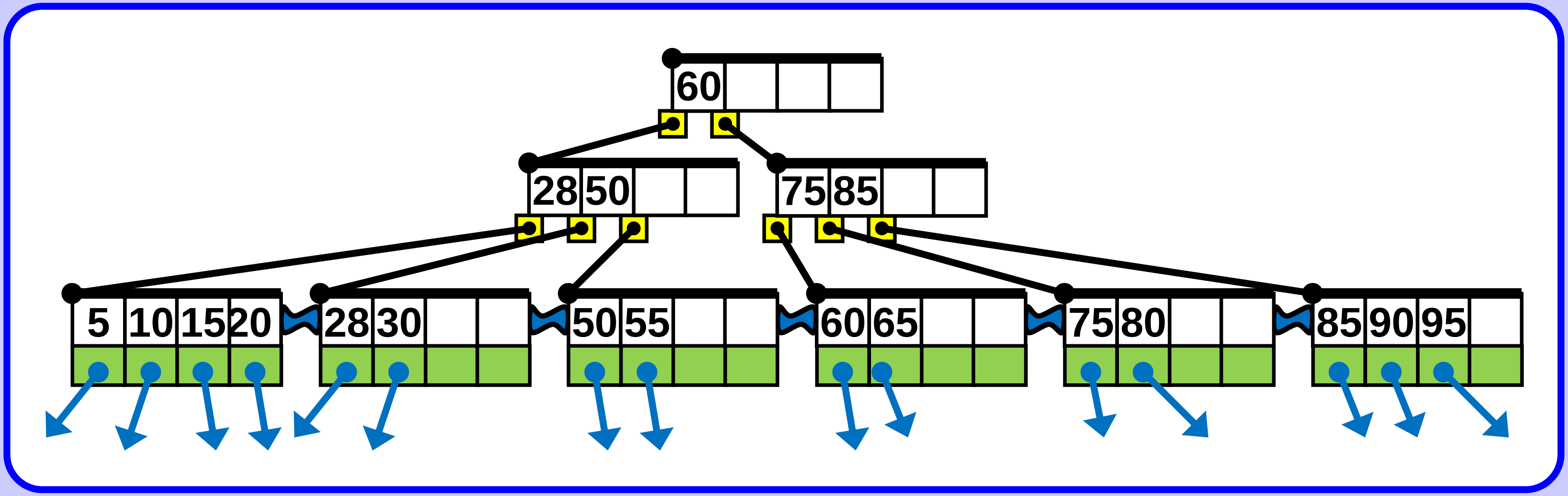
B+ tree is analogous to B-tree, namely in:
-- Being perfectly balanced all the time,
-- that nodes cannot be less than half full,
-- operational complexity.

The differences are:
-- Records (or pointers to actual records) are stored only in the leaf nodes,
-- internal nodes store only search key values which are used only as routers to guide the search.

The leaf nodes of a B^+-tree are linked together to form a linked list. This is done so that the records can be retrieved sequentially without accessing the B^+-tree index. This also supports fast processing of range-search queries.

B+ tree

Description/example



Routers and keys 75

Data records or pointers to them 🟩

Leaves links

Values in internal nodes are routers, originally each of them was a key when a record was inserted. Insert and Delete operations split and merge the nodes and thus move the keys and routers around. A router may remain in the tree even after the corresponding record and its key was deleted.

Values in the leaves are actual keys associated with the records and must be deleted when a record is deleted (their router copies may live on).

Inserting key K (and its associated data record) into B+ tree

Find, as in B tree, correct leaf to insert K. Then there are 3 cases:

Case 1

Free slot in a leaf? YES

Place the key and its associated record in the leaf.

Case 2

Free slot in a leaf? NO. Free slot in the parent node? YES.

1. Consider all keys in the leaf, including K, to be sorted.
2. Insert middle (median) key M in the parent node in the appropriate slot Y.
   (If parent does not exist, first create an empty one = new root.)
3. Split the leaf into two new leaves L1 and L2.
4. Left leaf (L1) from Y contains records with keys smaller than M.
5. Right leaf (L2) from Y contains records with keys **equal to or greater than** M.

Note: Splitting leaves and inner nodes works in the same way as in B-trees.

Inserting key K (and its associated data record) into B+ tree

Find, as in B tree, correct leaf to insert K. Then there are 3 cases:

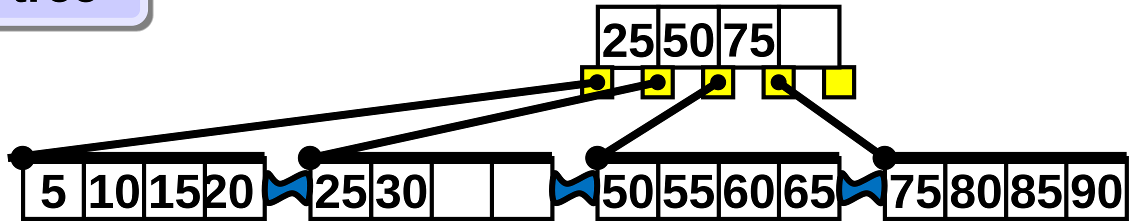
Case 3

Free slot in a leaf? NO. Free slot in the parent node? NO.

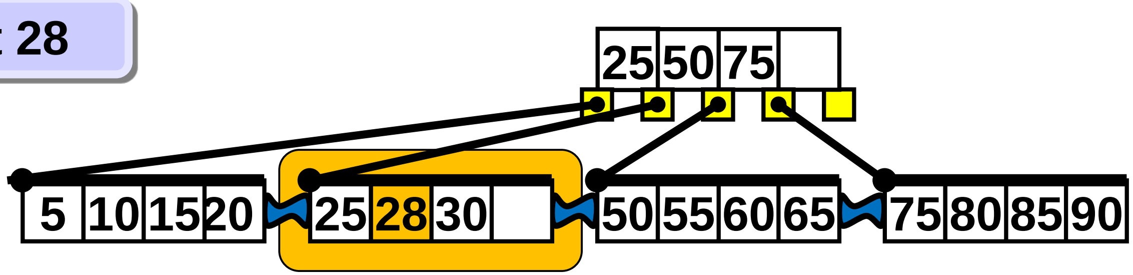
1. Split the leaf into two leaves L1 and L2, consider all its keys including K sorted, denote M median of these keys.
2. Records with keys < M go to the left leaf L1.
3. Records with keys **>=** M go to the right leaf L2.

4. Split the parent node P to nodes P1 and P2, consider all its keys including M sorted, denote M1 median of these keys.
5. Keys < M1 key go to P1.
6. Keys > M1 key go to P2.
7. If parent PP of P is not full, insert M1 to PP and stop.
   (If PP does not exist, first create an empty one = new root.)
   Else set M := M1, P := PP and continue splitting parent nodes recursively up the tree, repeating from step 4.

Initial tree



Insert 28



Changes 🟧

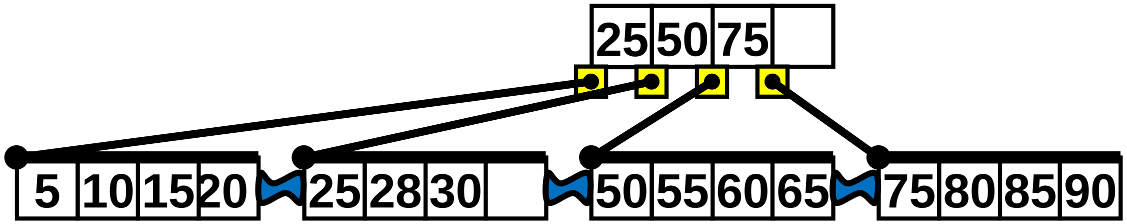
Leaves links ⋈

Data records and pointers to them are not drawn here for simplicity's sake.

B+ tree - Insert

Insert example II

Initial tree



25 50 75

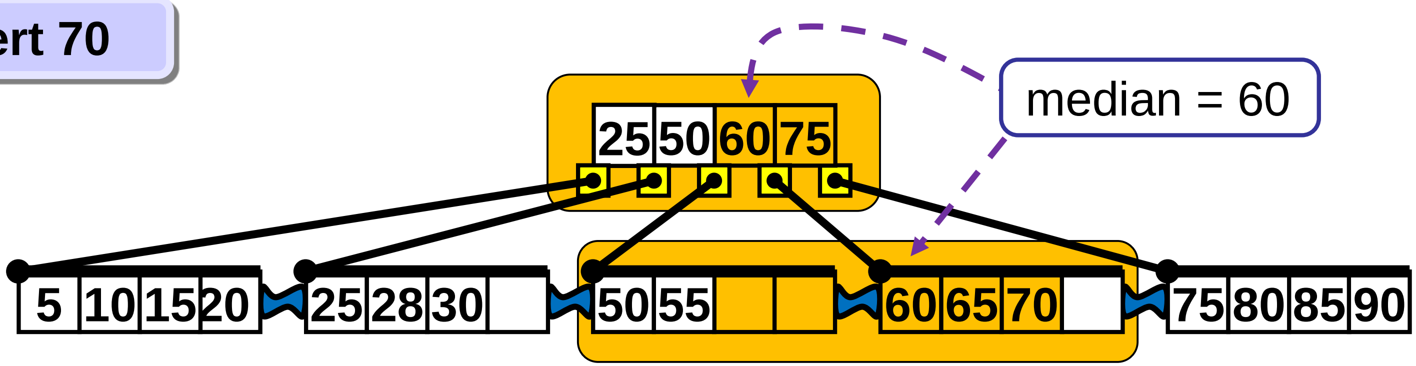
5 10 15 20

25 28 30

50 55 60 65

75 80 85 90

Insert 70



median = 60

25 50 60 75

5 10 15 20

25 28 30

50 55

60 65 70

75 80 85 90

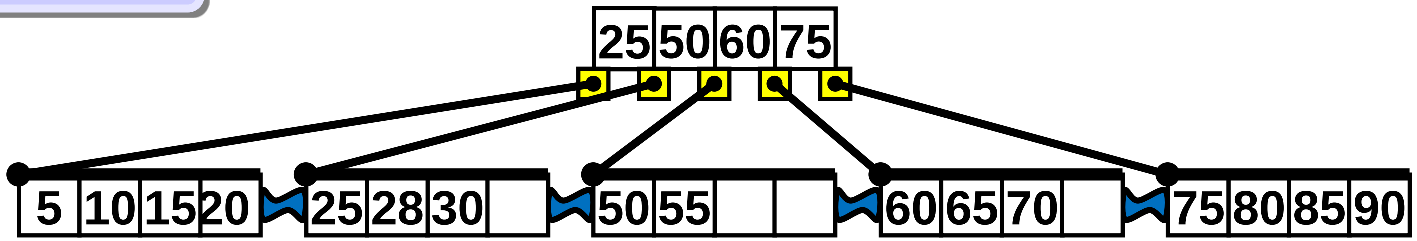
Changes 🟧

Leaves links

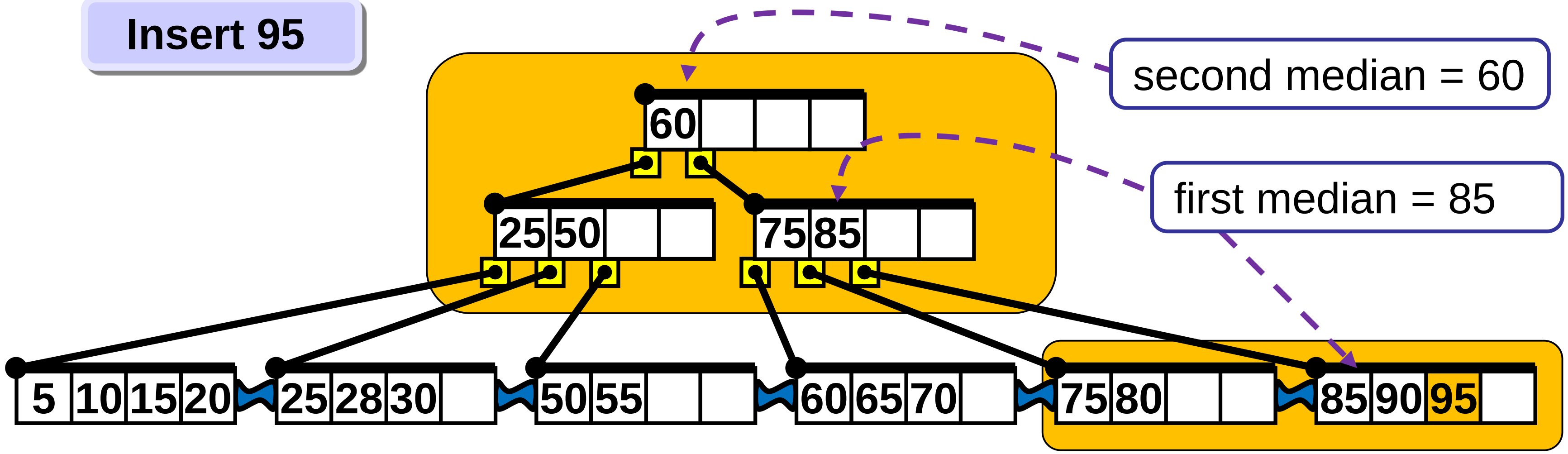
B+ tree - Insert

Insert example III

Initial tree



Insert 95



Changes ▨

Leaves links ⋈

Note the router 60 in the root, detached from its original position in the leaf.

Deleting key K (and its associated data record) in B+ tree

Find, as in B tree, key K in a leaf. Then there are 3 cases:

Case 1

Leaf more than half full or leaf == root?   YES.

Delete the key and its record from the leaf L. Arrange the keys in the leaf in ascending order to fill the void. If the deleted key K appears also in the parent node P replace it by the next bigger key K1 from L (explain why it exists) and leave K1 in L as well.

Case 2

Leaf more than half full? NO.   Left or right sibling more than half full? YES.

Move one (or more if you wish and rules permit) key(s) from sibling S to the leaf L, reflect the changes in the parent P of L and parent P2 of sibling S.
(If S does not exist then L is the root, which may contain any number of keys).

Deleting key K (and its associated data record) in B+ tree

Find, as in B tree, key K in a leaf. Then there are 3 cases:

Case 3

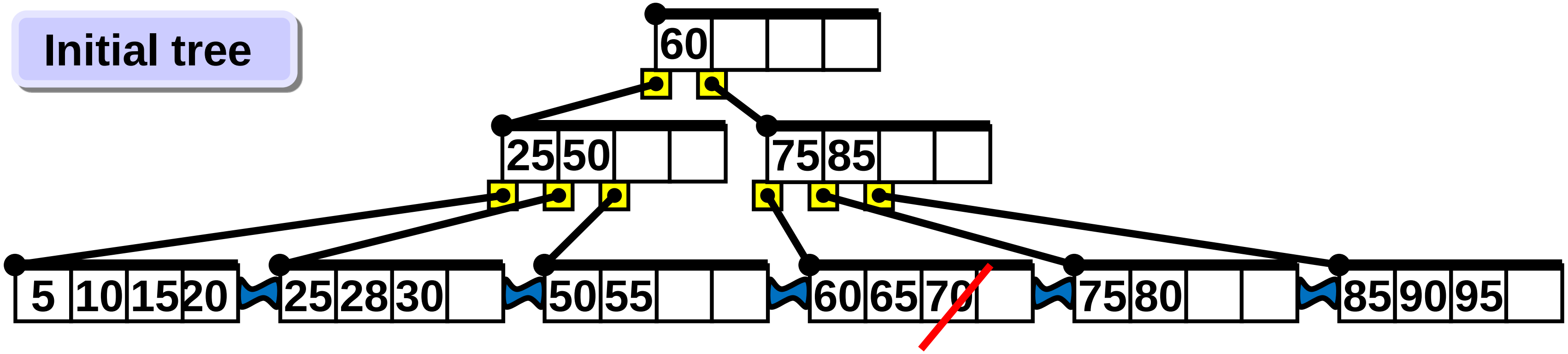
Leaf more than half full? NO. Left or right sibling more than half full? NO.

1. Consider sibling S of L which has the same parent P as L.
2. Consider set M of ordered keys of L and S without K but together with key K1 in P which separates L and S.
3. Merge: Store M in L, connect L to the other sibling of S (if exists), destroy S.
4. Set the reference left to K1 to point to L. Delete K1 from P. If P contains K delete it also from P. If P is still at least half full stop, else continue with 5.
5. If any sibling SP of P is more then half full, move necessary number of keys from SP to P and adjust links in P, SP and their parents accordingly and stop.
   Else set L := P and continue recursively up the tree (like in B-tree), repeating from step 1.

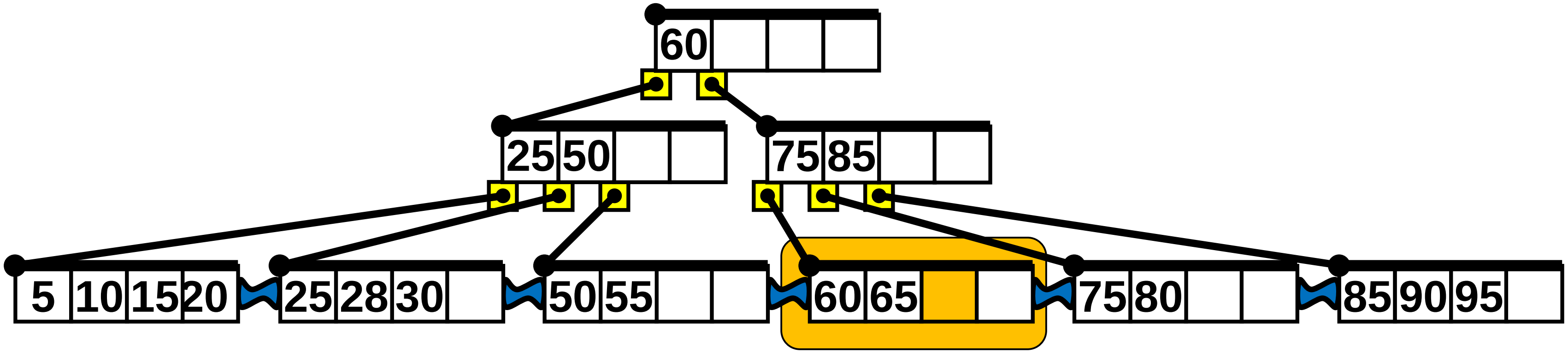
Note: Merging leaves and inner nodes works the same way as in B-trees.

B+ tree - Delete

Initial tree



Delete 70



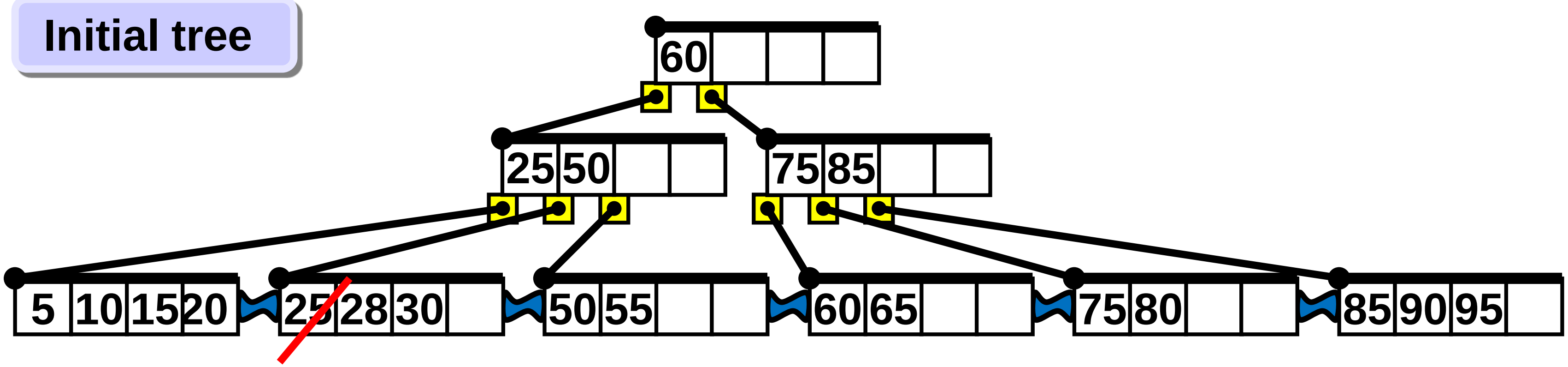
Changes 🟨

Leaves links ⋈

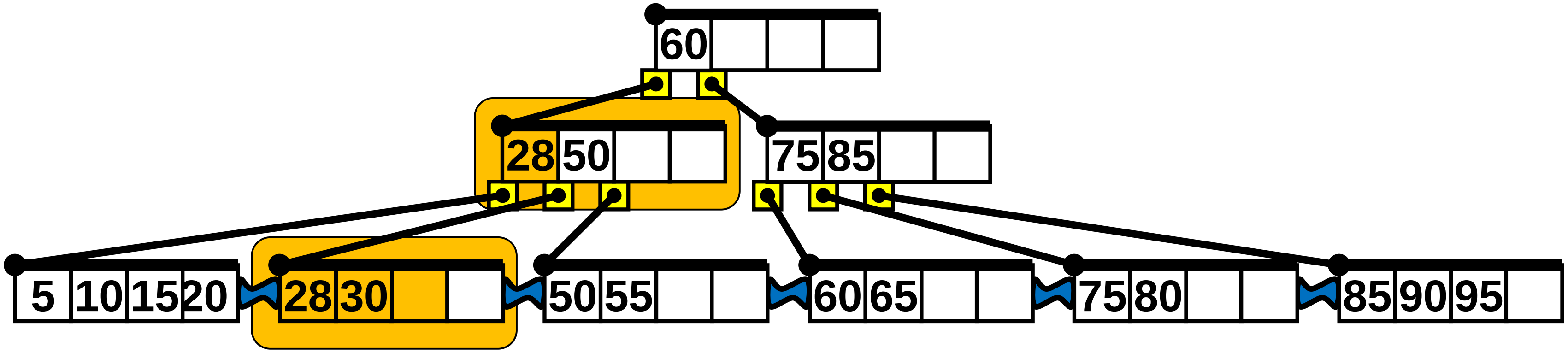
B+ tree - Delete

Delete example II

Initial tree



Delete 25



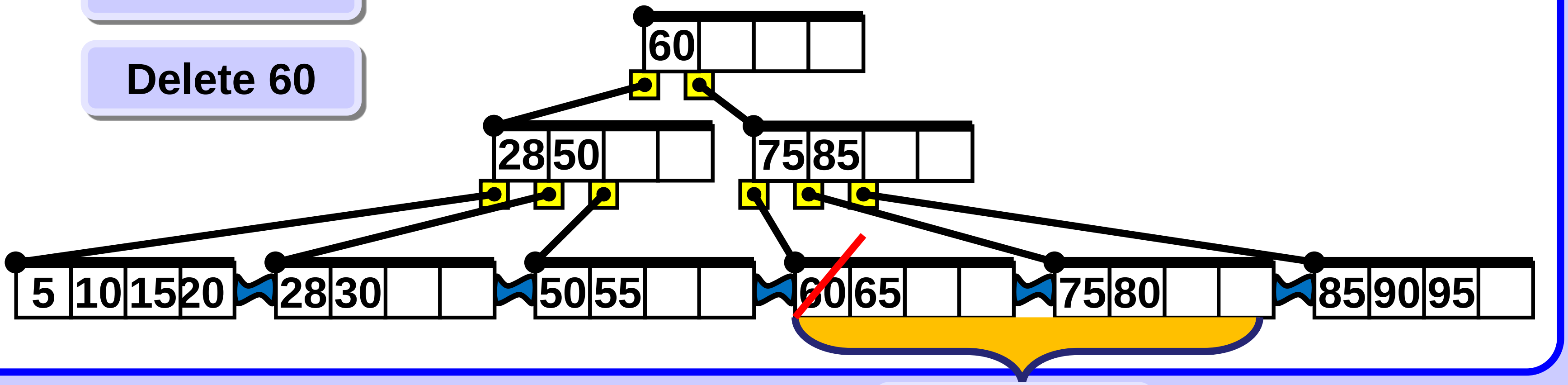
Changes 🟧

Leaves links ⋈

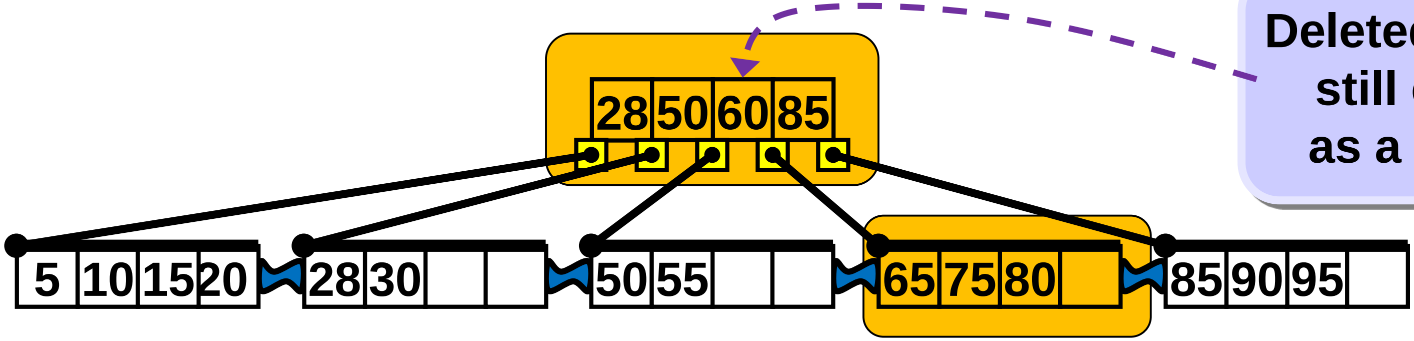
Delete example III

Initial tree

Delete 60



Merge



Deleted key 60 still exists as a router

Changes 🟧

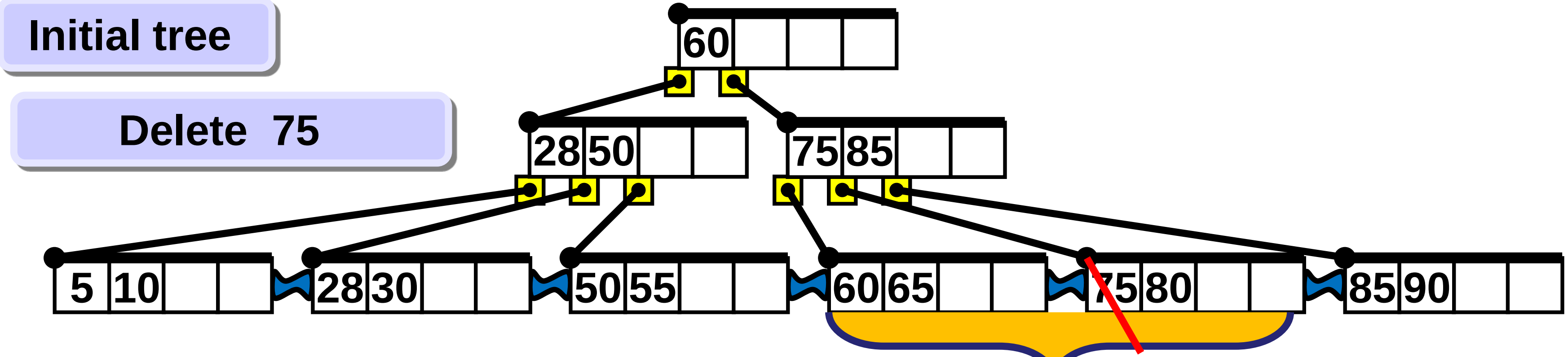
Leaves links ⋈

B+ tree - Delete

Delete example IV

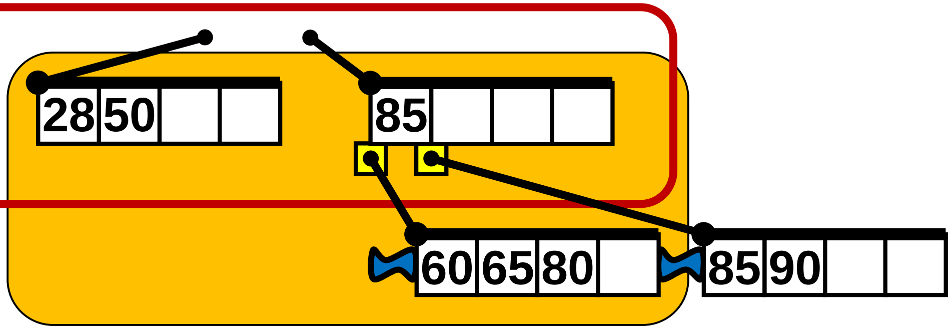
Initial tree

Delete 75



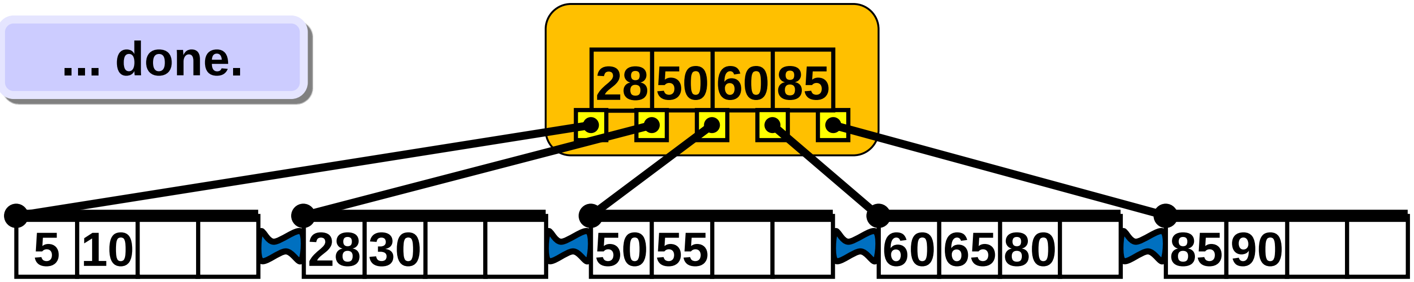
Merge

Too few keys, merge these two nodes and bring a key from parent (recursively).



Progress...

... done.



Complexities

Find, Insert, Delete,
all need Θ(b $\log_b n$) operations, where n is number of records in the tree,
and b is the branching factor or, as it is often understood, the order of the tree.

Note: Be careful, some authors (e.g CLRS) define degree/order of B-tree as [b/2], there is no unified precise common terminology.

Range search thanks to the linked leaves is performed in time
Θ(b $\log_b(n) + k/b$)
where k is the range (number of elements) of the query.