6. Search Trees

Goals

- Understand search, insert, and delete in binary search trees
- Learn search, insert, and delete in red-black trees
- Learn search, insert, and delete in B-trees
- Understand multidimensional search trees

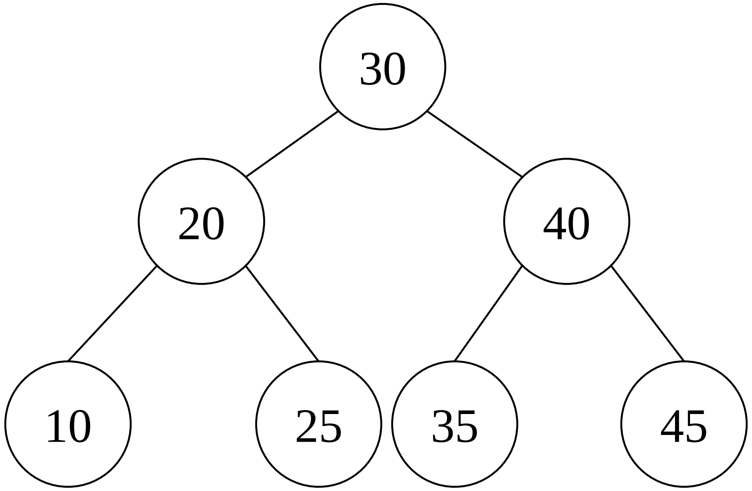
Record, Key, Search Tree

- Record
  - Storage unit including all information about an object
  - e.g., record for a person
    - Name, national ID, home address, mobile phone number, etc.
- Field
  - Each item in the record
  - e.g., name
- Key
  - Field(s) that represent records uniquely
  - Key may consist of one field or multiple fields.
- Search tree
  - Contains keys and record pointers
  - Provides searching mechanism by keys

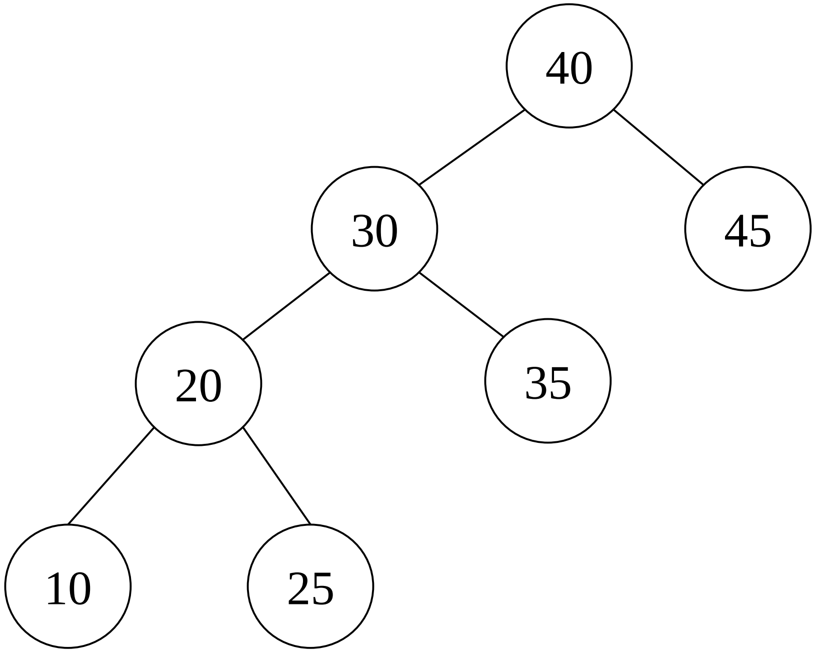
Binary Search Tree

- Each node of a binary search tree contains one key.
- Each node has at most two children. (binary tree)
- For any node x, key of x > any key in x's left subtree, and key of x < any key in x's right subtree. (search tree)

Binary Search Tree



(a)

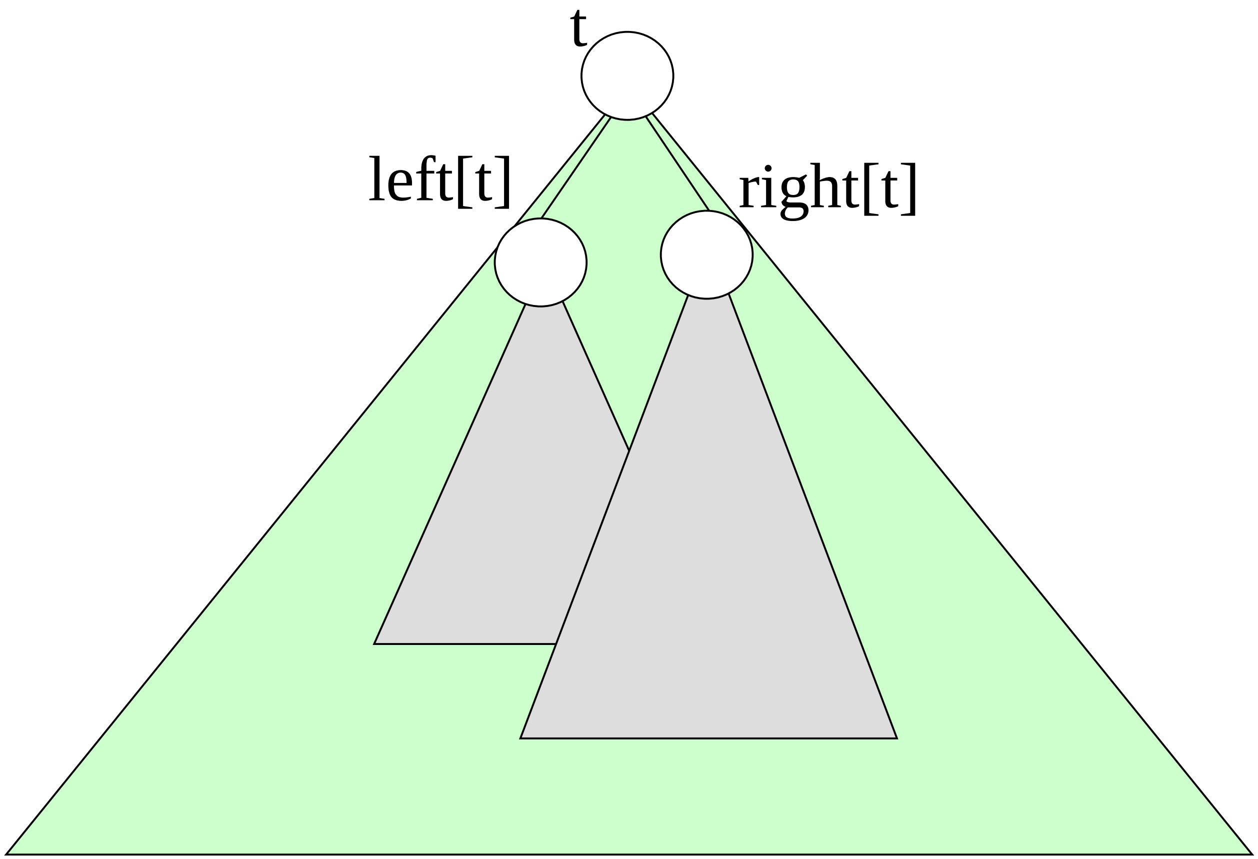


(b)

Search in Binary Search Tree

```
treeSearch(t, x)
▷ t: tree node
▷ x: search key
{
        if (t=NIL or key[t]=x) then return t;
        if (x < key[t])
                then return treeSearch(left[t], x);
                else return treeSearch(right[t], x);
}
```

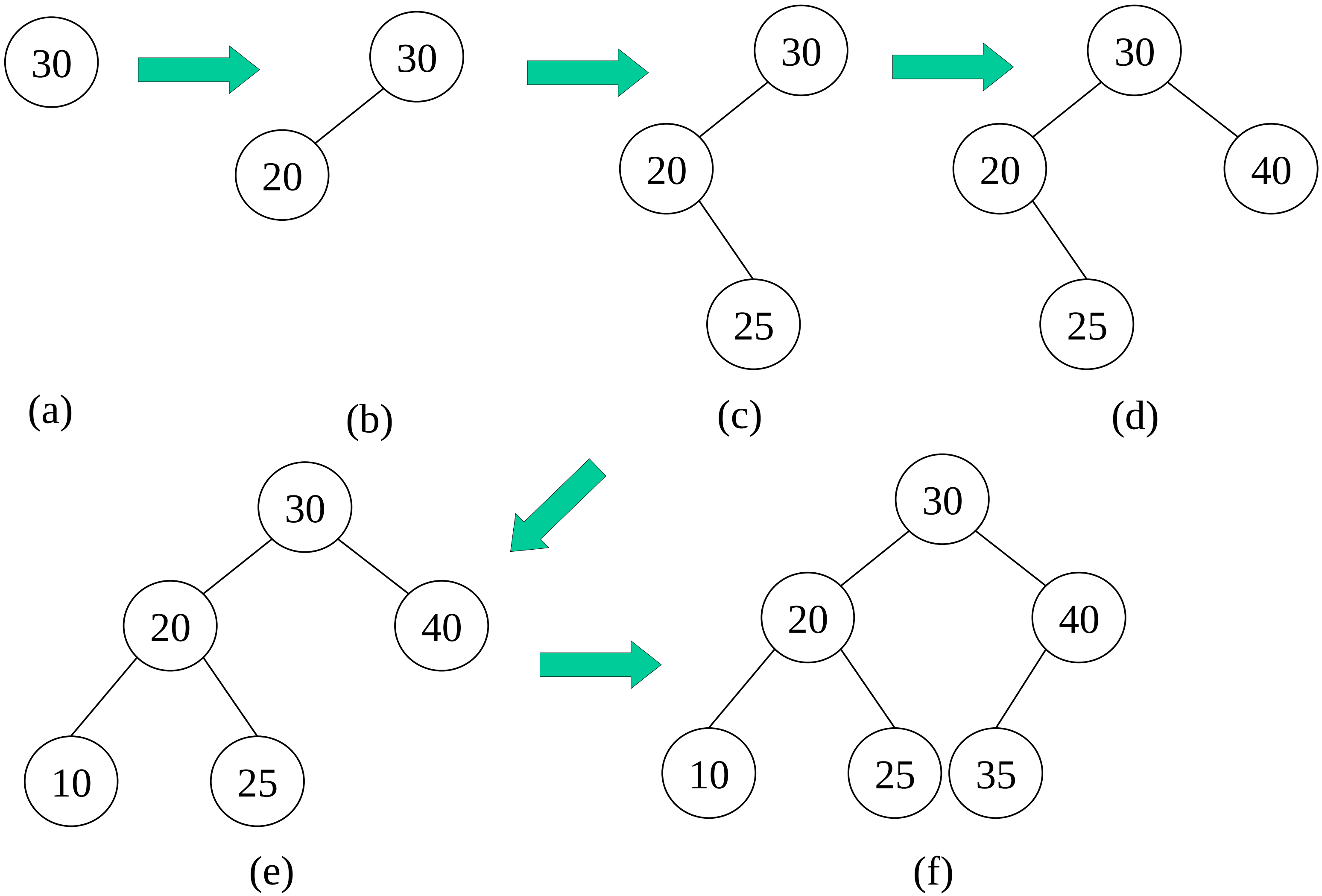
Recursion in Search



Insert in Binary Search Tree

```
treeInsert(t, x)
▷ t: tree node
▷ x: insert key
▷ return root after insertion
{
     if (t=NIL) then {
          key[r] ← x;  left[r] ← NIL;  right[r] ← NIL;    ▷ r : new node
          return r ;
     }
     if (x < key(t))
          then {left[t] ← treeInsert(left[t], x); return t;}
          else {right[t] ← treeInsert(right[t], x); return t;}
}
```

Insert



Delete in Binary Search Tree

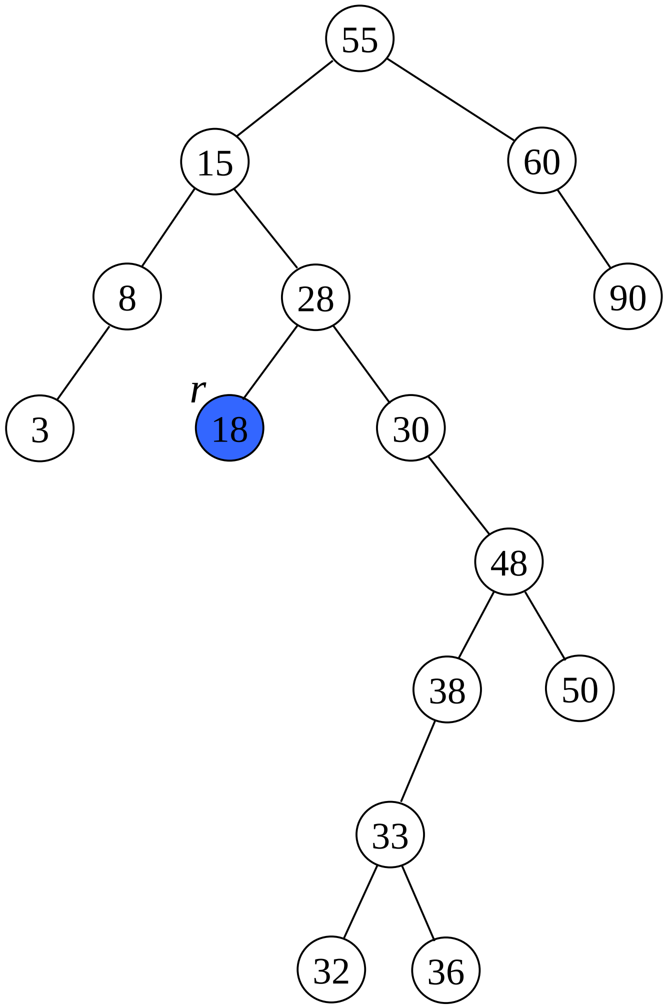
r: node to be deleted

- There are three cases
  - Case 1 : *r* is a leaf
  - Case 2 : *r* has one child
  - Case 3 : *r* has two children

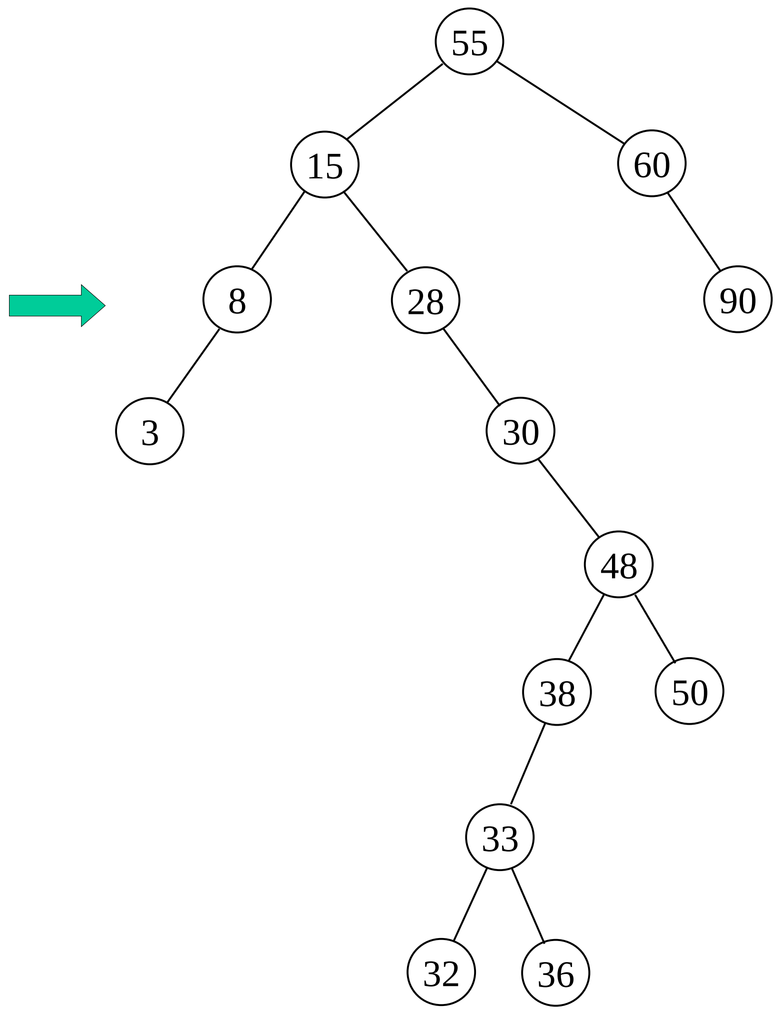
Delete

```
treeDelete(t, r)
▷ t: tree root
▷ r: node to be deleted
{
    if (r is leaf) then                                  ▷ Case 1
        remove r
    else if (r has one child) then                       ▷ Case 2
        replace r by r's child
    else                                                 ▷ Case 3
        replace r by r's successor s (smallest element in r's right subtree)
        delete s (case 1 or 2)
}
```

Delete: Case 1

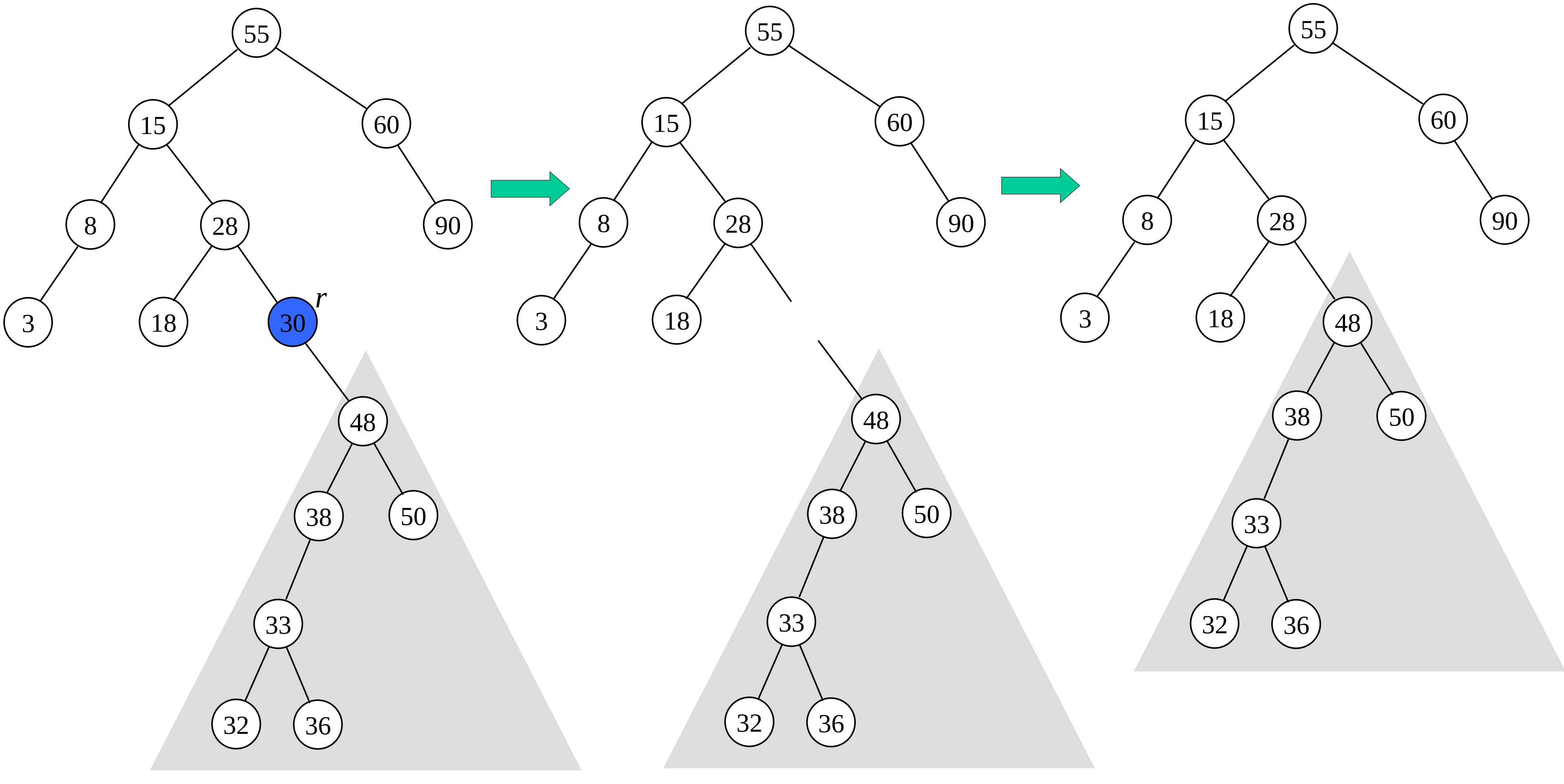


(a) *r* is a leaf



(b) Remove *r*

Delete: Case 2

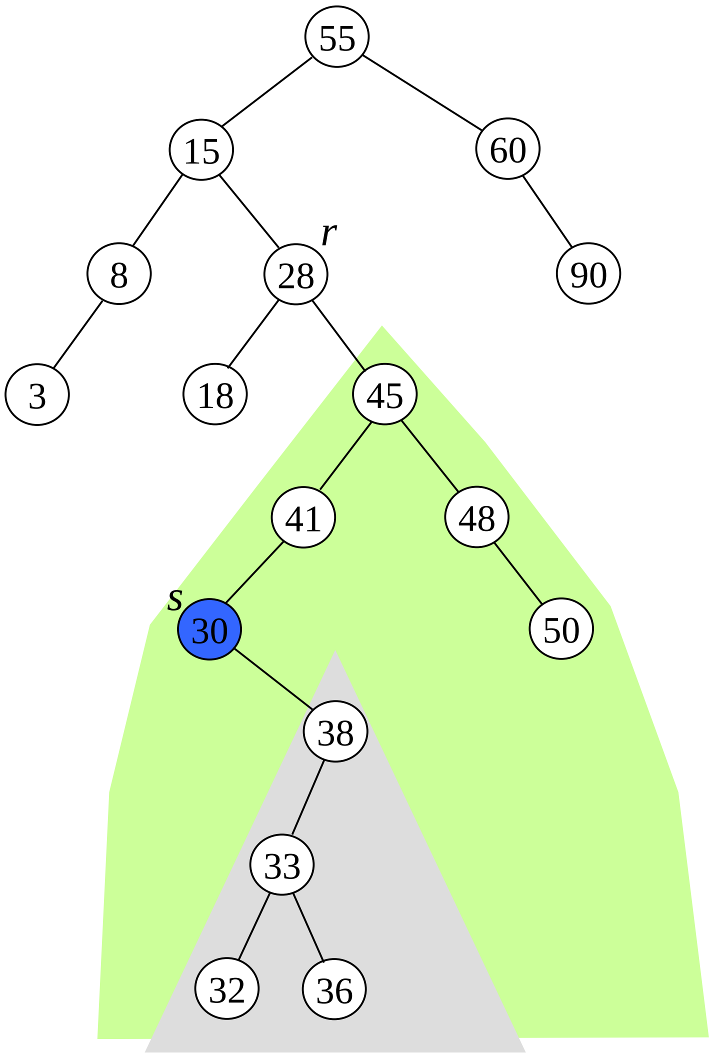


(a) *r* has one child

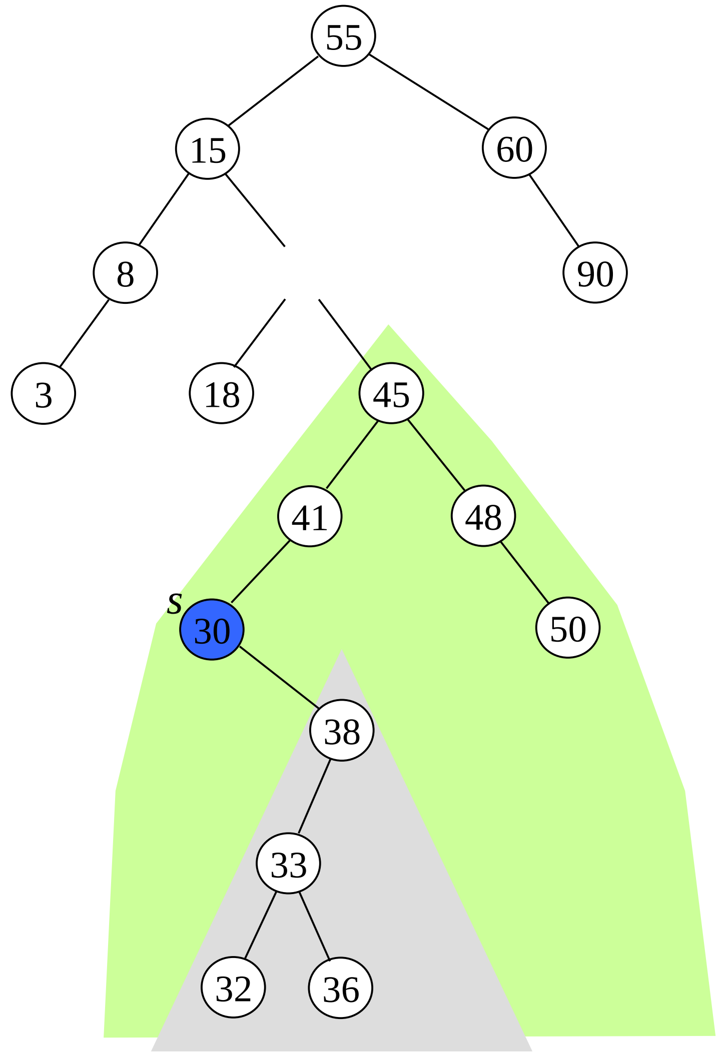
(b) Remove *r*

(c) Replace it by *r*'s child

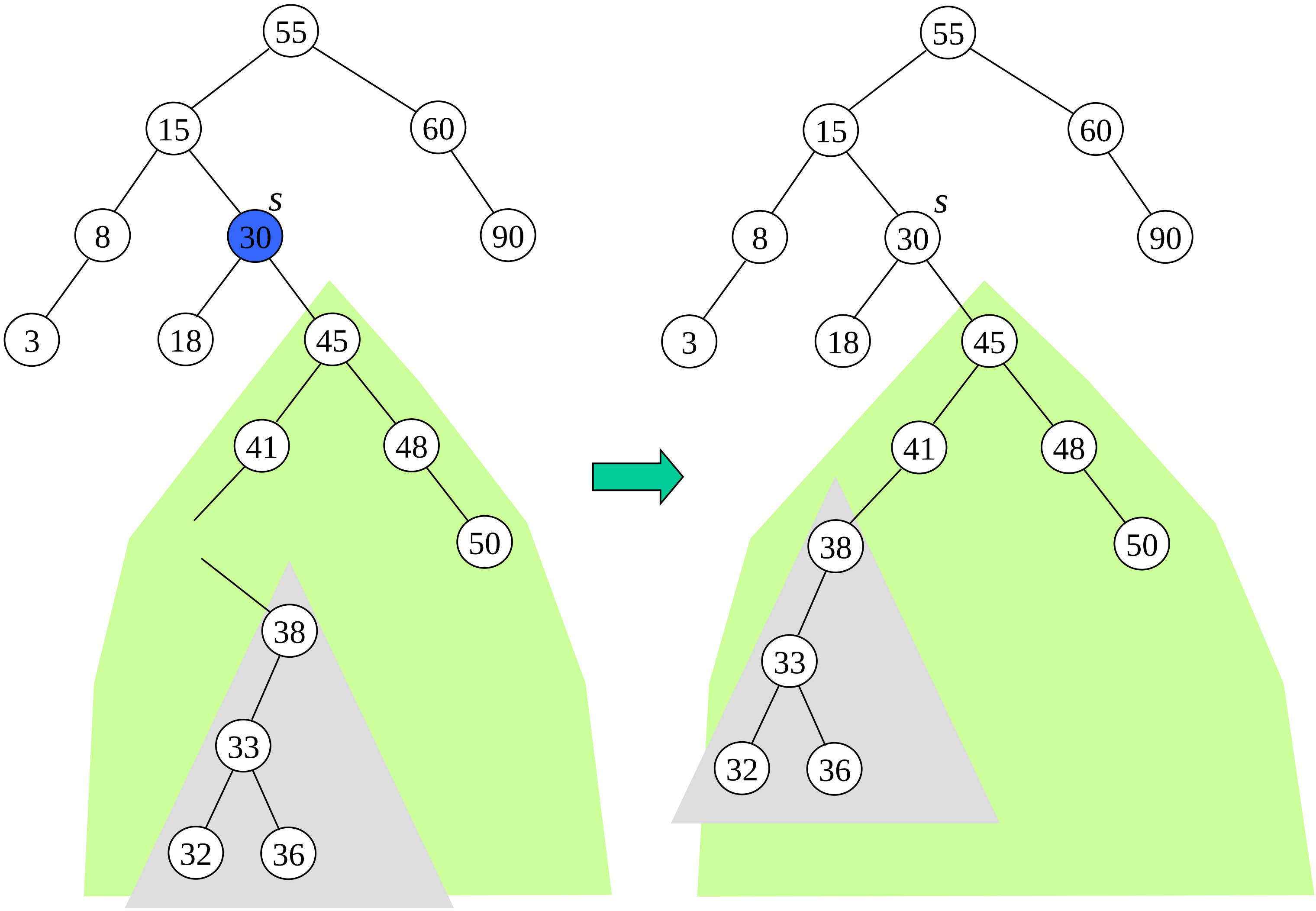
Delete: Case 3



(a) Find *r*'s successor *s*



(b) Remove *r*



(c) Replace it by *s*

(d) Case 2

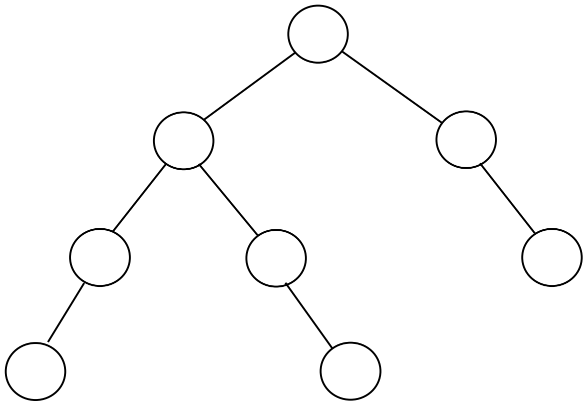
Red-Black Tree

- Red-black tree is a binary search tree with 1 bit information per node (color: red or black) satisfying the following red-black properties.

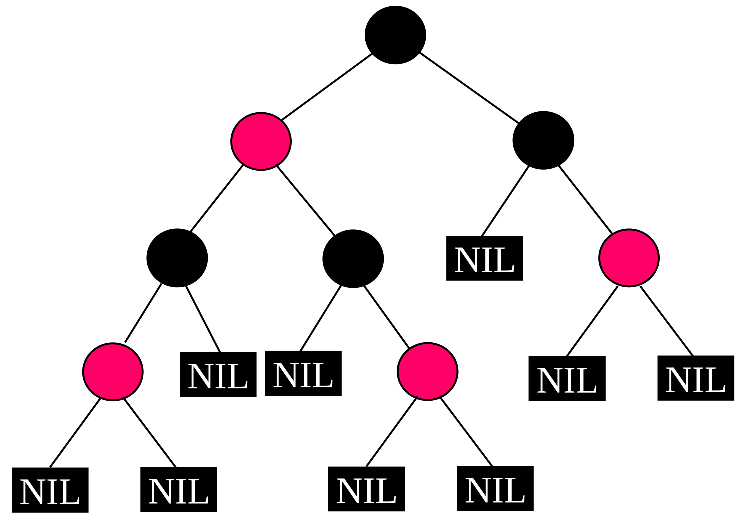
  ① Root is black
  ② Every leaf (NIL) is black
  ③ If a node is red, both its children must be black
  ④ All paths from root to leaf contain the same number of black nodes

✓ where each leaf has value NIL (actually each leaf has a pointer to a special node which has value NIL

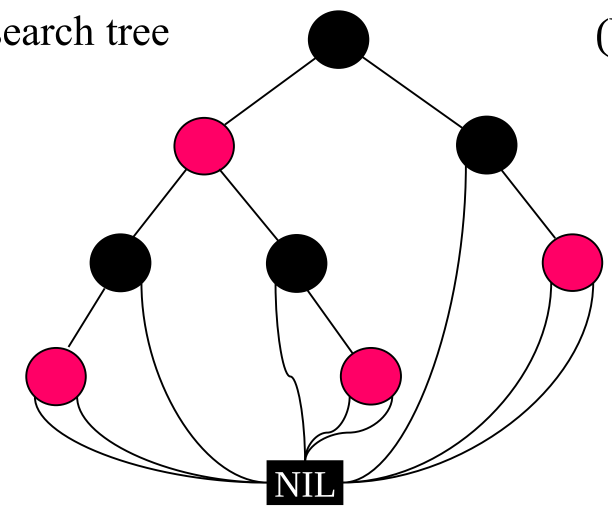
Red-Black Tree



(a) Binary search tree



(b) Red-black tree



(c) Actual implementation

Red-Black Tree

- Black height bh(x): number of black nodes in the path from x to a leaf, excluding x
- A red-black tree has at least $2^{bh(r)} - 1$ internal nodes.
- A red-black tree with n internal nodes has height $h \leq 2\log_2(n+1)$.
  - $n \geq 2^{bh(r)} - 1$
  - $bh(r) \leq \log(n+1)$
  - $h \leq 2bh(r) \leq 2\log_2(n+1)$

Search in Red-Black Tree

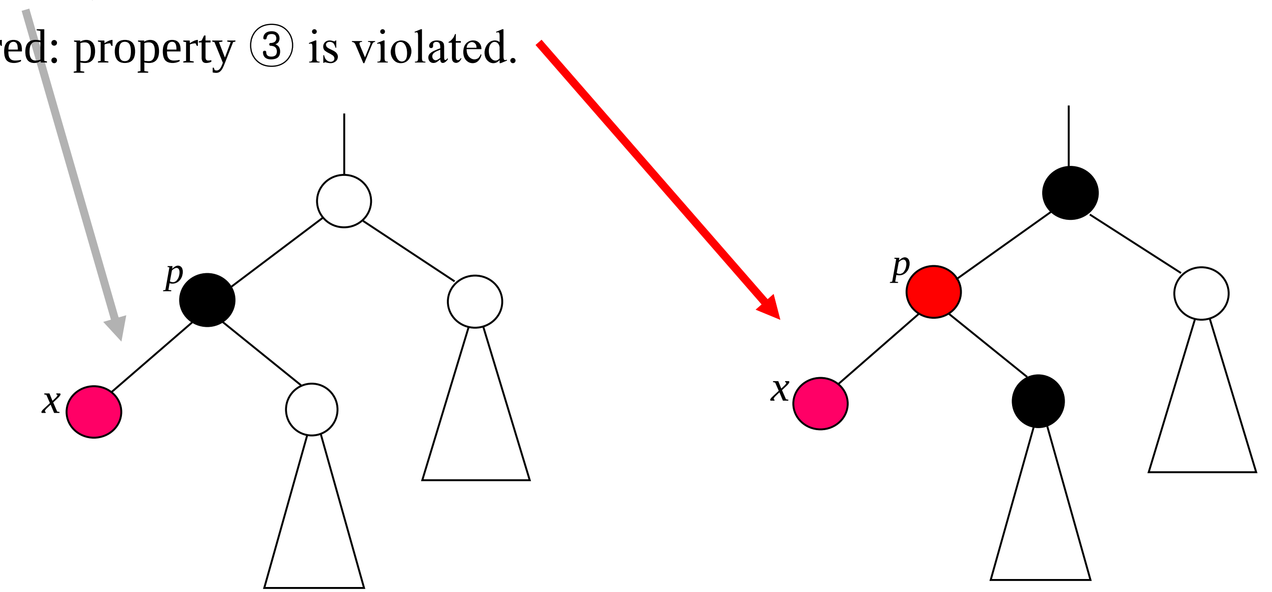
- does not change the tree
- procedure: same as that of binary search tree
- but in worst-case $O(\log n)$ time

Insert and Delete in Red-Black Tree

- modify the tree structure
- Basic operations: rotations
  - Left rotate
  - Right rotate
- need to change colors of some nodes to satisfy the red-black properties

Insert in Red-Black Tree

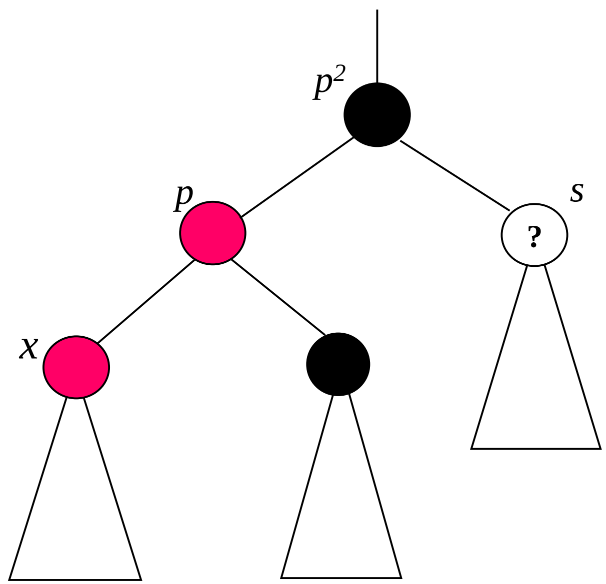
- Insert *x*: same as insert in binary search tree.
- Color *x* red.
- If the color of *x*'s parent *p* is
  - black, done.
  - red: property ③ is violated.



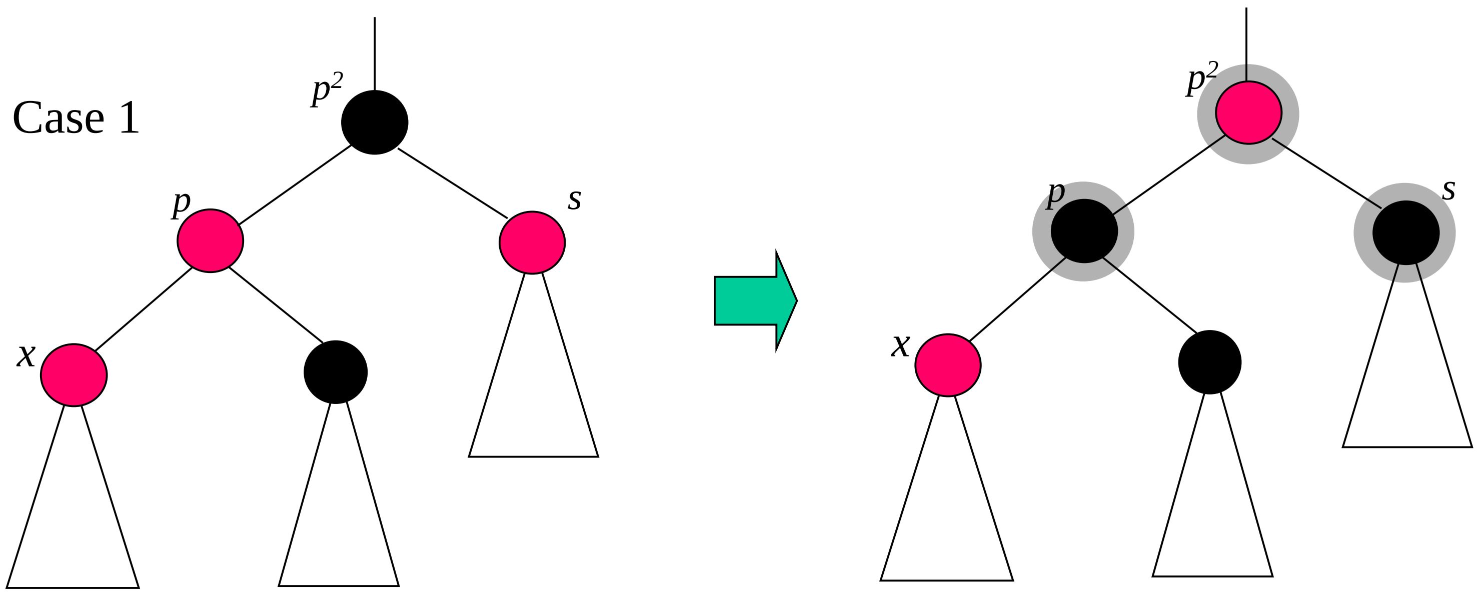
✓ Hence, consider the case *p* is red

Insert in Red-Black Tree

- p^2 and x's sibling must be black
- There are two cases depending on color of s (p's sibling)
  - Case 1: s is red
  - Case 2: s is black



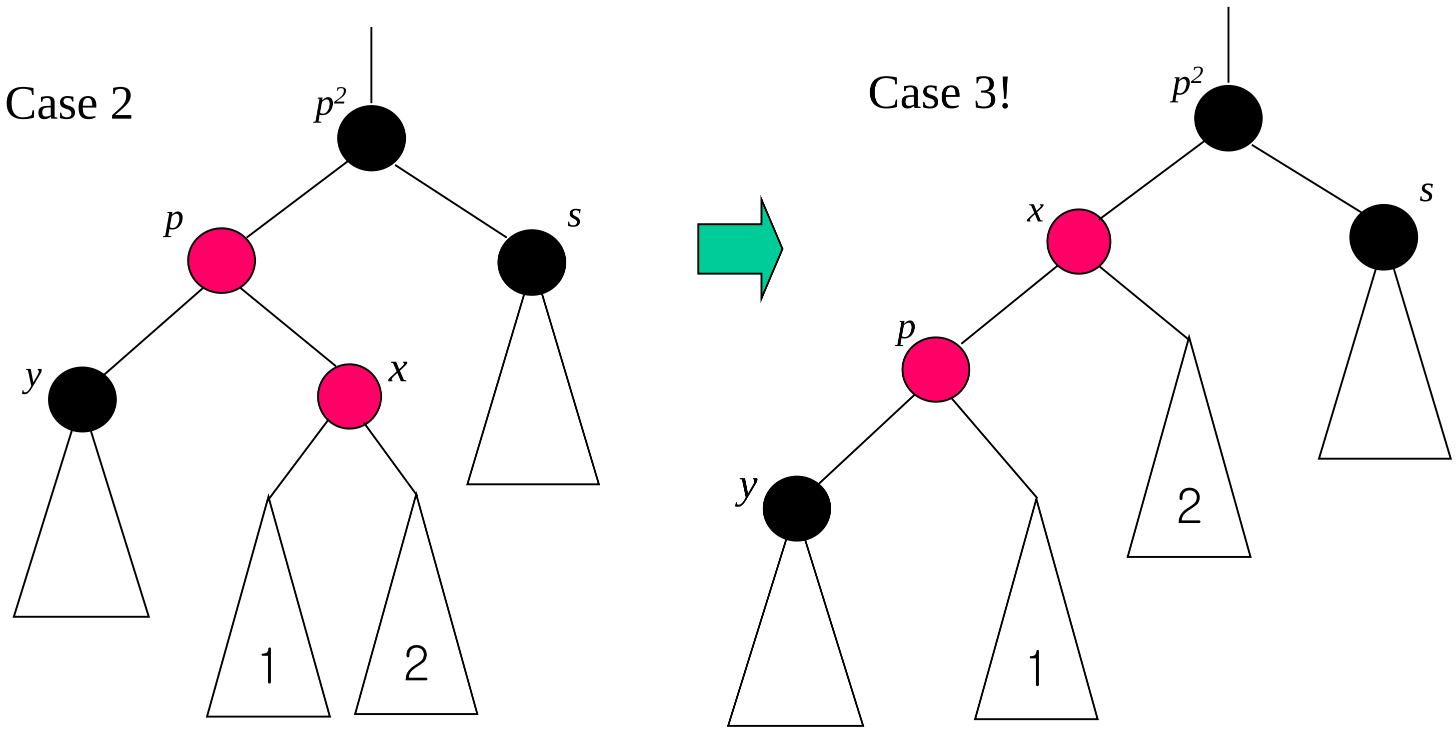
Case 1: *s* is red



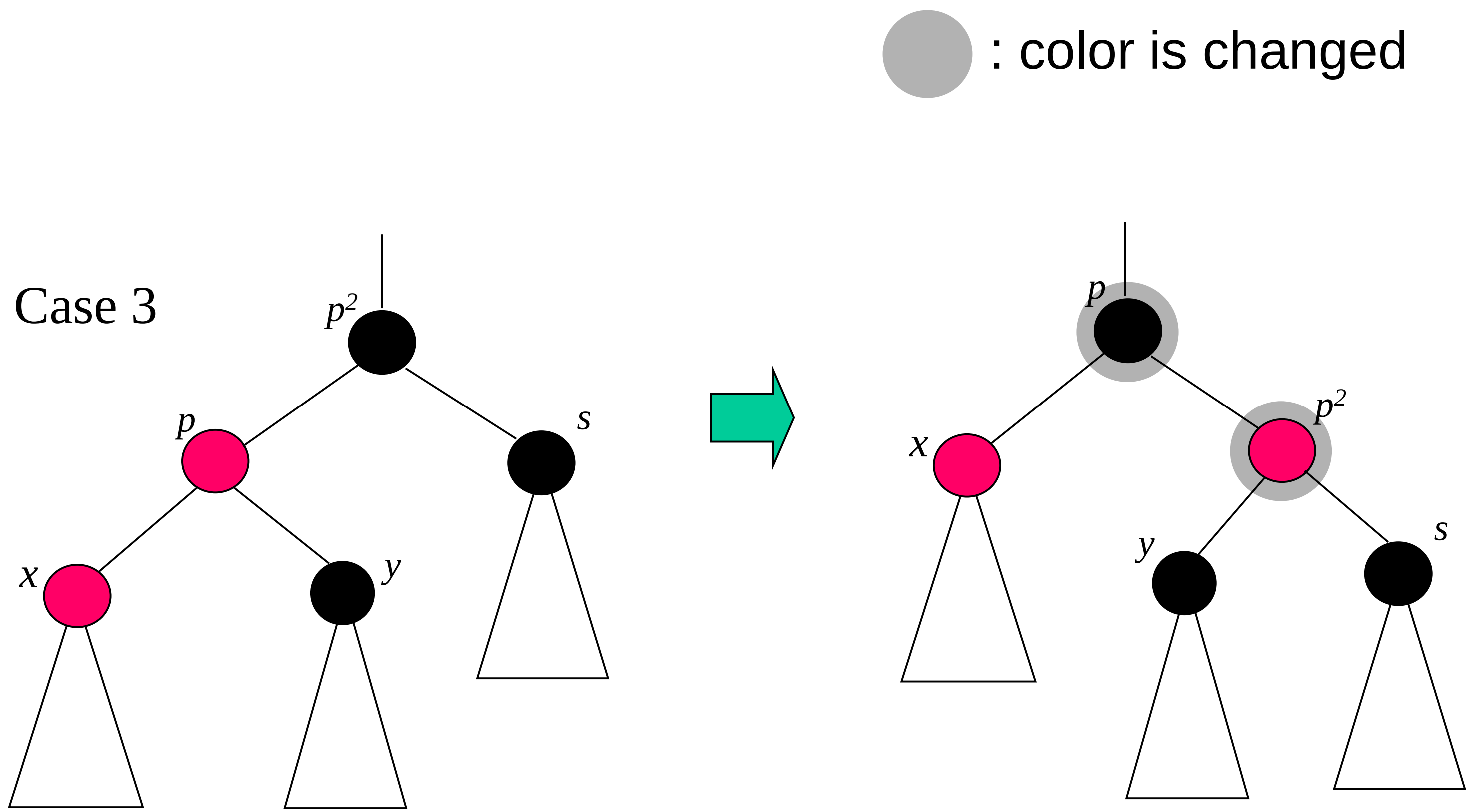
: color is changed

✓ p^2 may have the same problem

Case 2: *s* is black, *x* is *p*'s right child



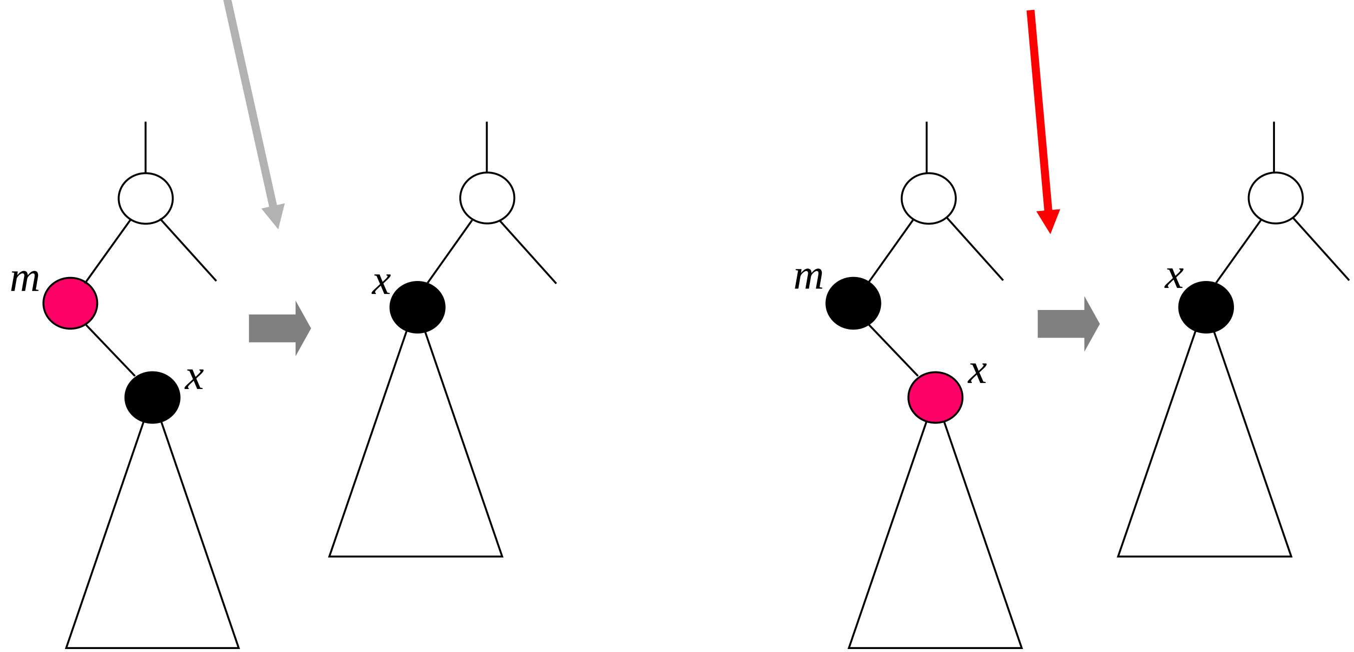
Case 3: s is black, x is p's left child



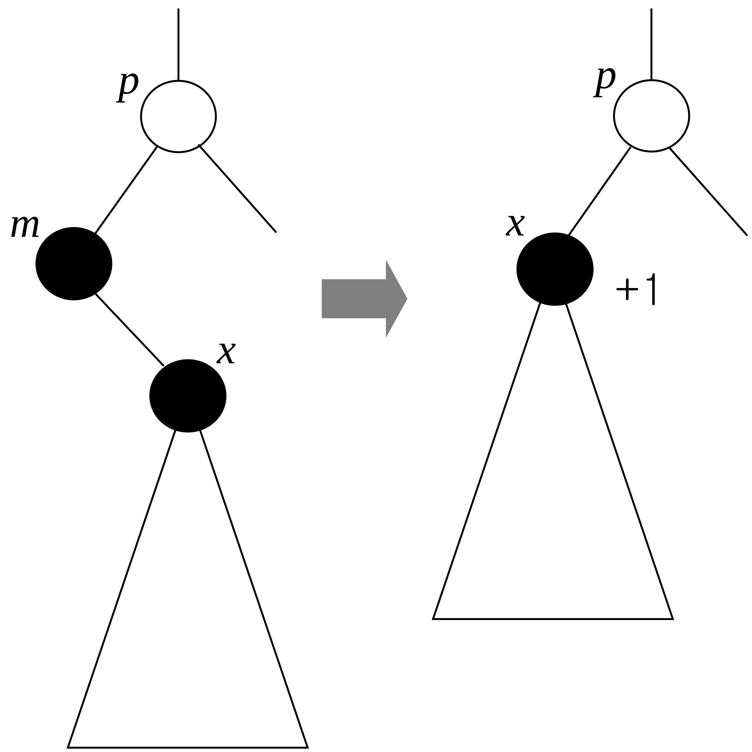
✓ insert done! $O(\log n)$ time

Delete in Red-Black Tree

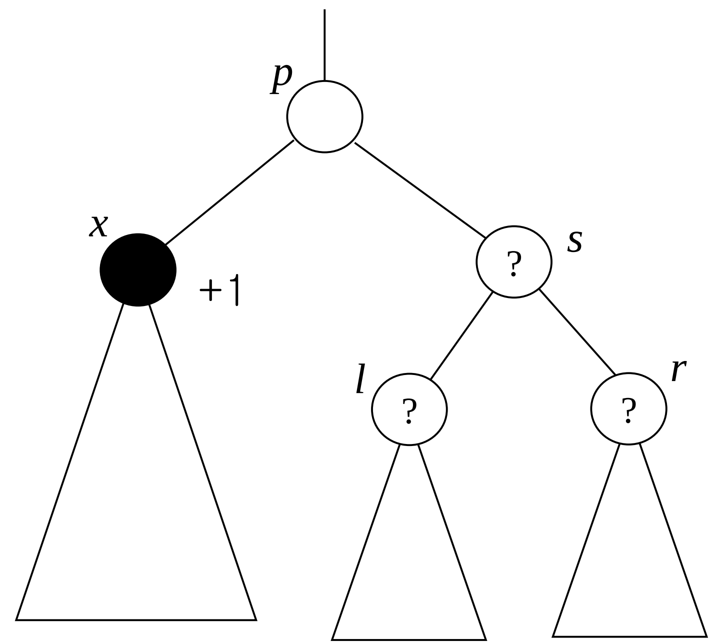
- Delete *m*: same as delete in binary search tree.
- Node *m* had at most one internal child.
- If *m* was red, we are done.
- If *m* was black and internal child was red, done.



✓ +1 means that x has an extra black



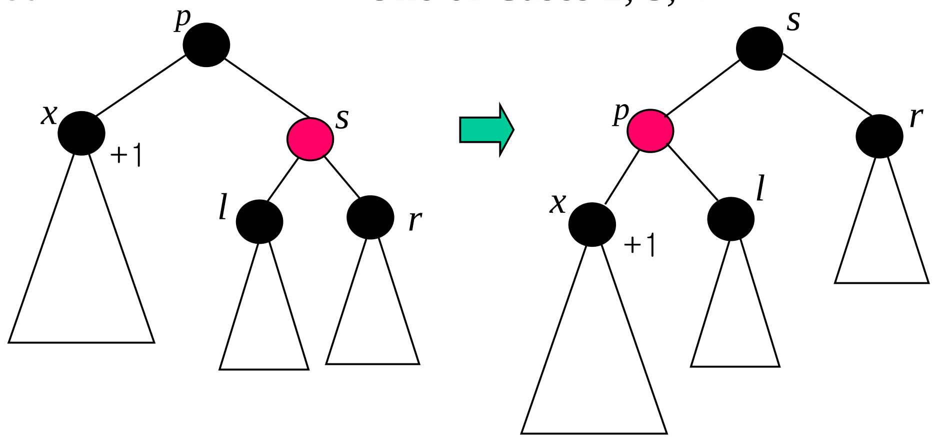
After *m* is deleted,
property ④ is violated



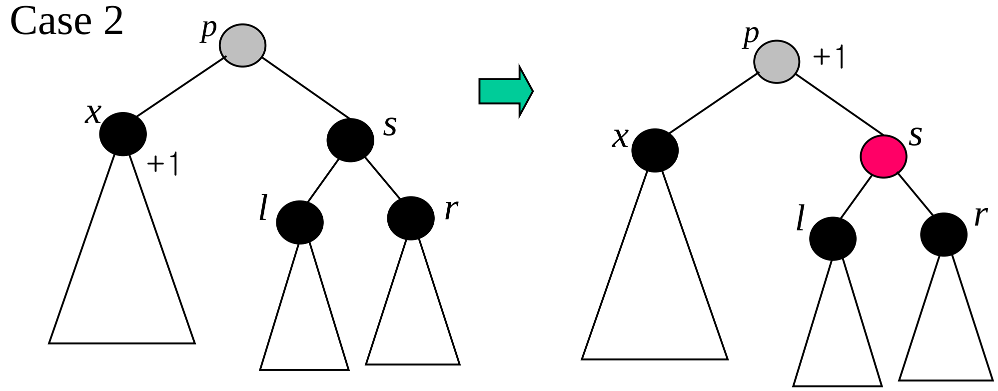
Cases by colors of *s*, *l*, *r*

Case 1: *s* is red

Case 1 One of Cases 2, 3, 4



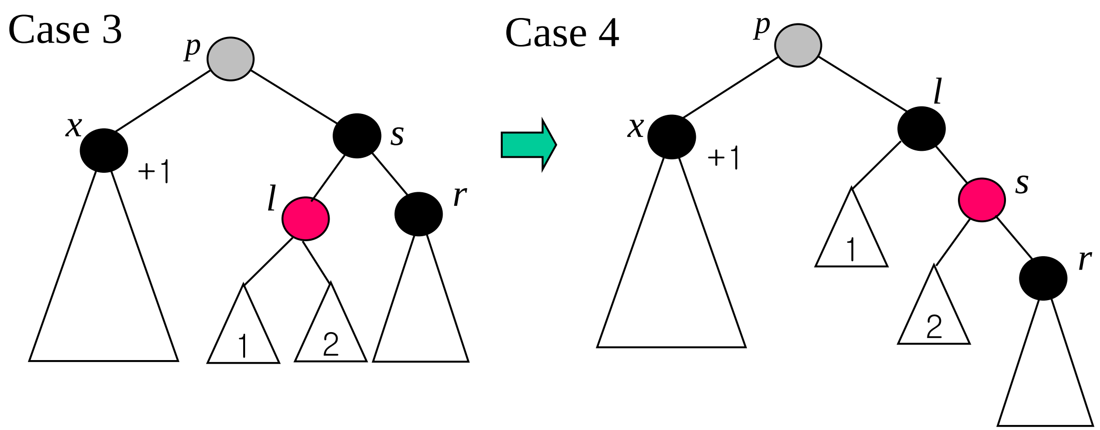
Case 2: s is black, and both of s's children are black



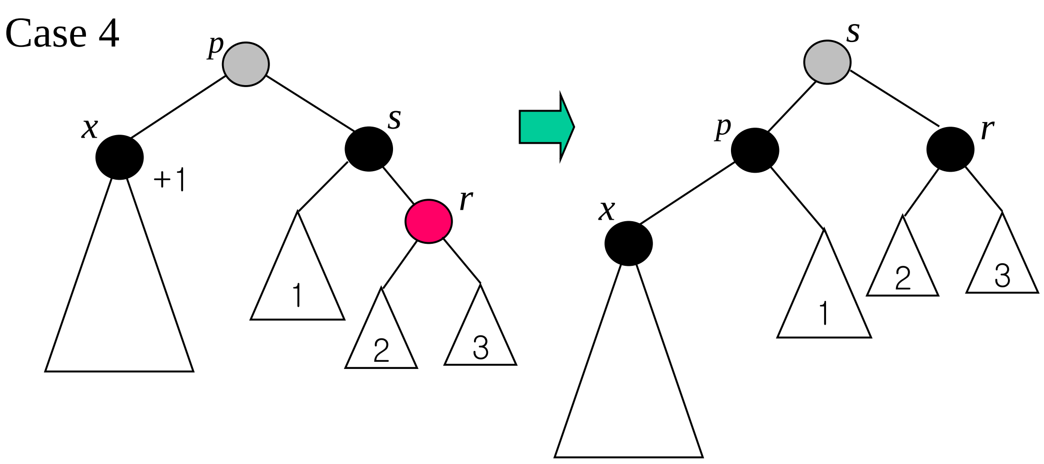
✓ p may have the same problem

$O(\log n)$ time

Case 3: *s* is black, *s*'s left child is red, and *s*'s right child is black



Case 4: *s* is black, and *s*'s right child is red

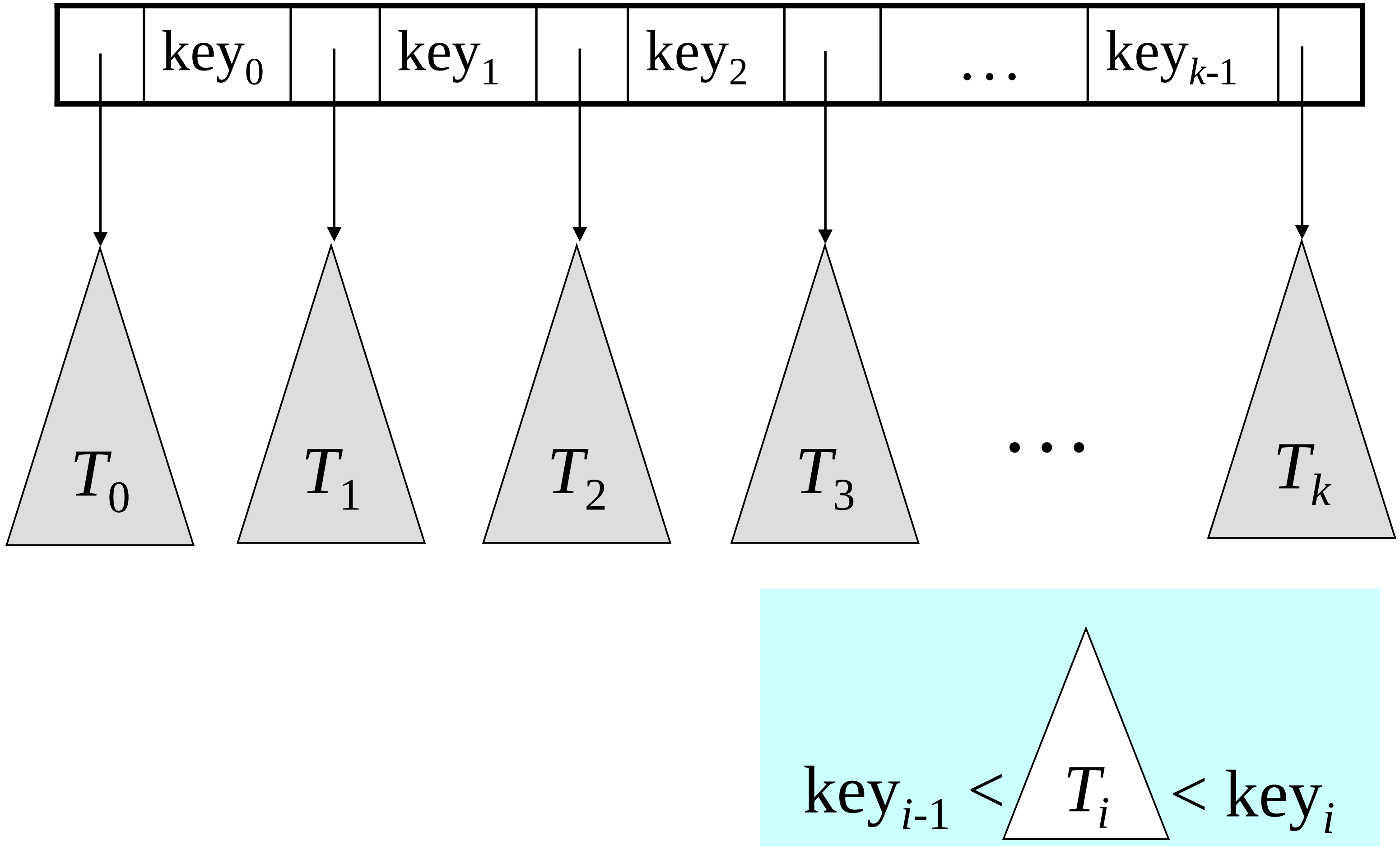


✓ Delete is done!

B-Trees

- Access unit
  - Memory to disk: page (4KB)
  - Cache to memory: cache line (64B)
- Disk access is costly (equivalent to processing time of several hundred thousand instructions).
- If a search tree is stored in a disk, minimize the height of search tree
- B-tree is a balanced multi-way search tree that minimizes disk accesses in the worst case.

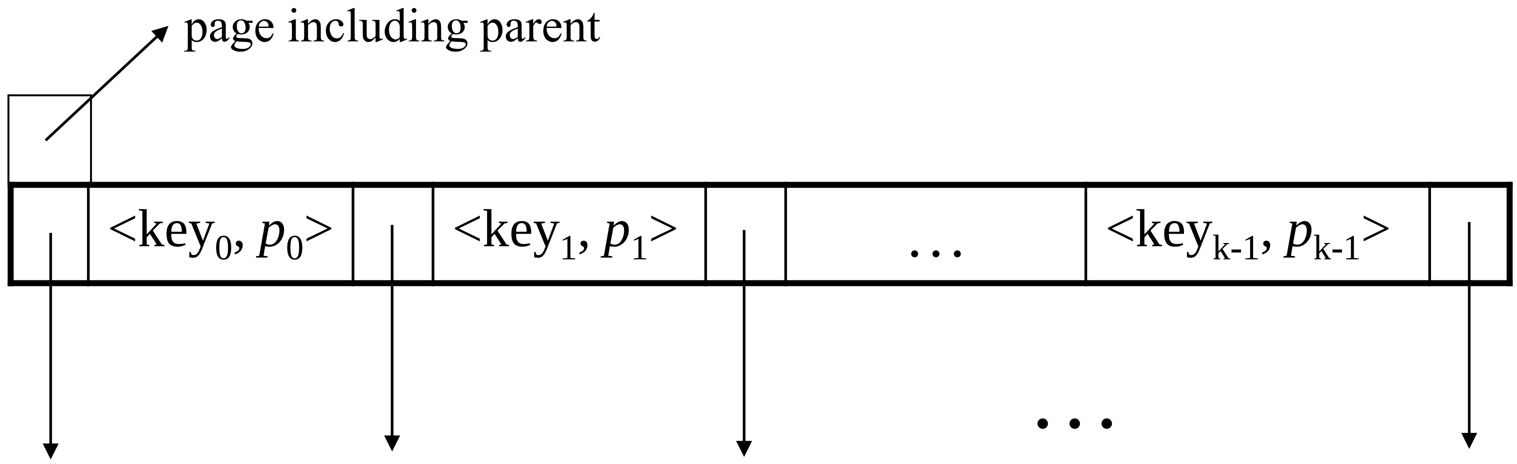
Multi-Way Search Tree



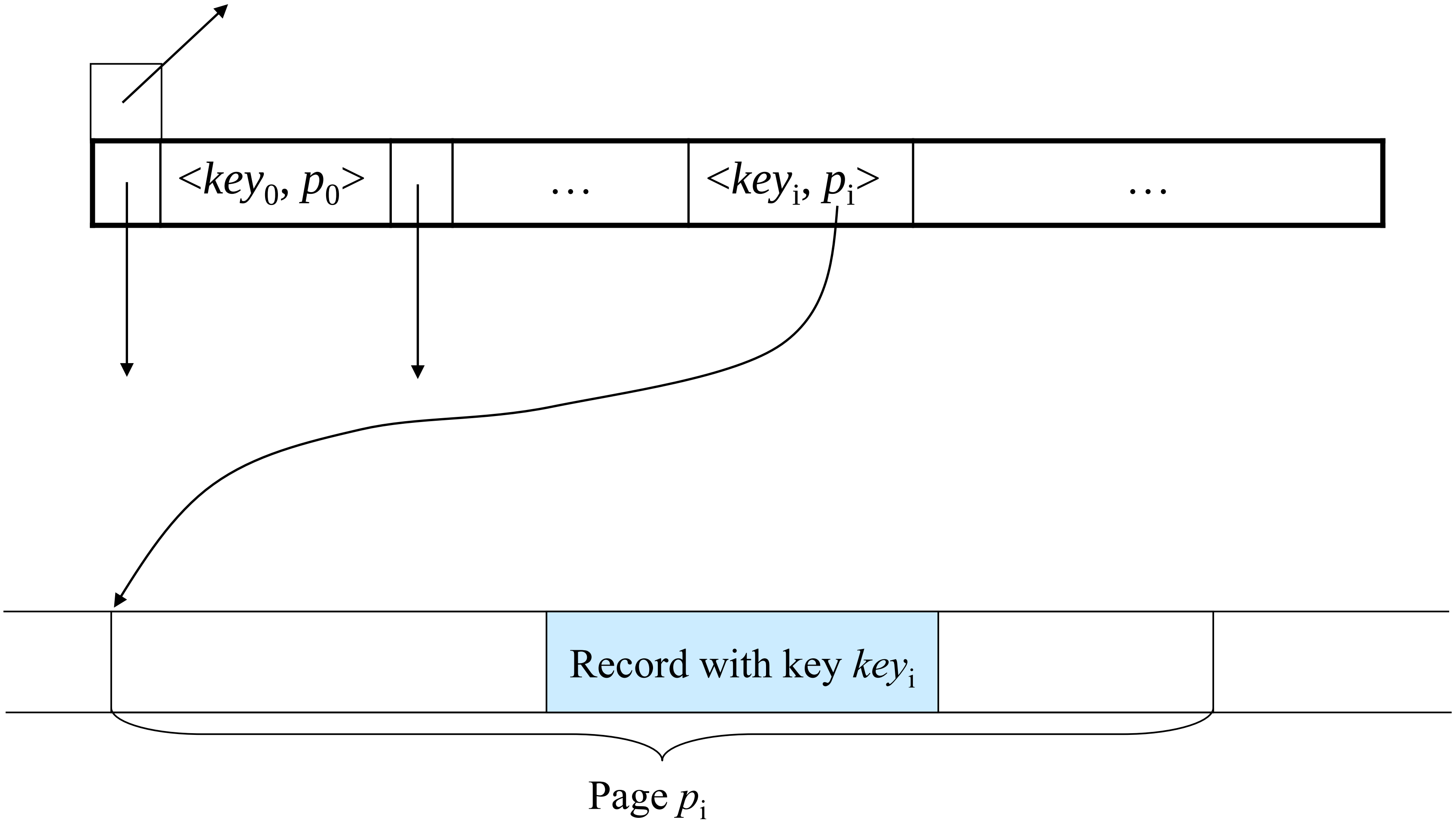
B-Tree

- B-tree is a balanced multi-way search tree that satisfies the following properties.
  - Every node except root has $\lfloor k/2 \rfloor \sim k$ keys.
  - All leaves have the same depths.

Node Structure of B-Tree



Access to Record through B-Tree

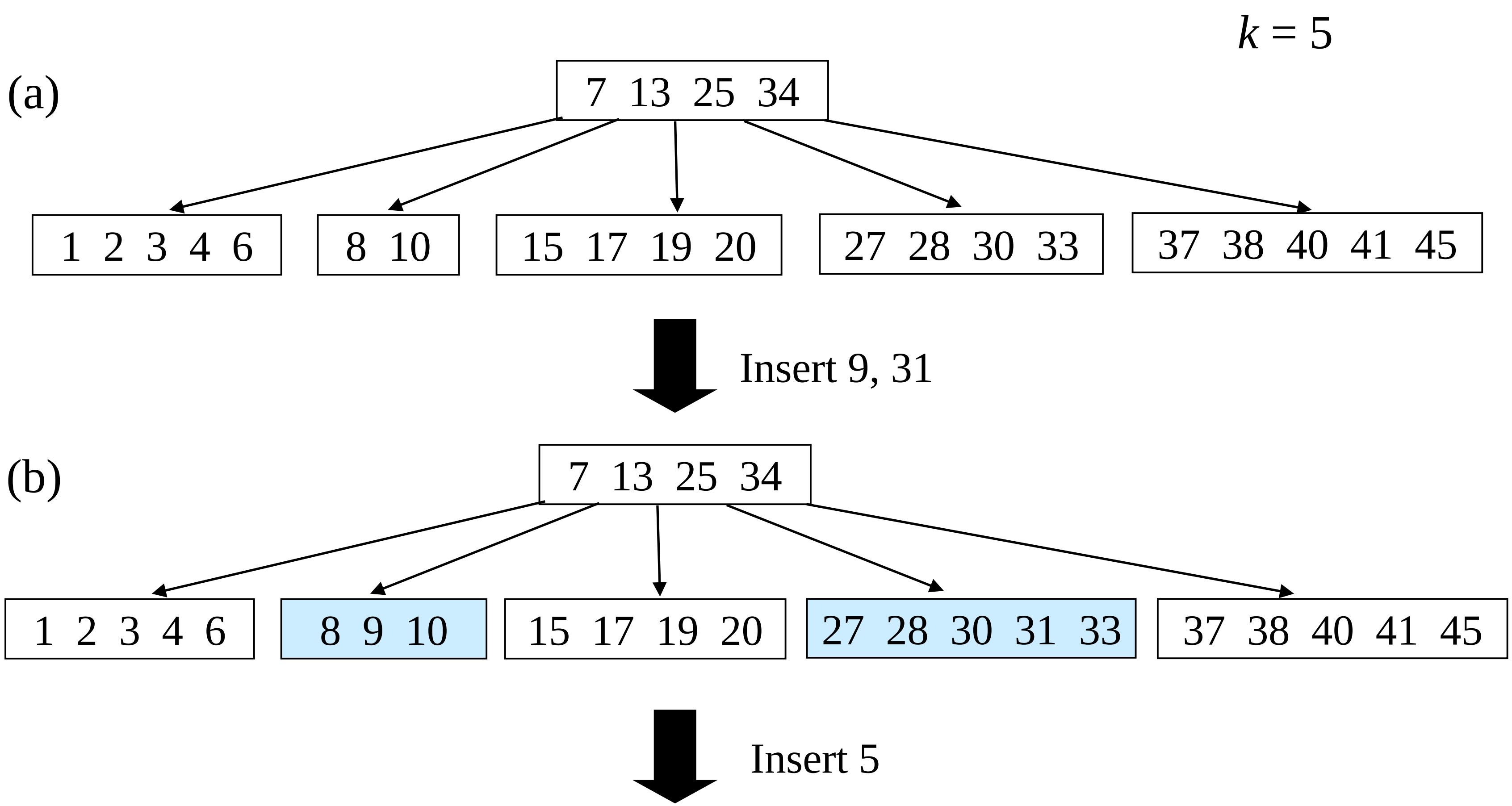


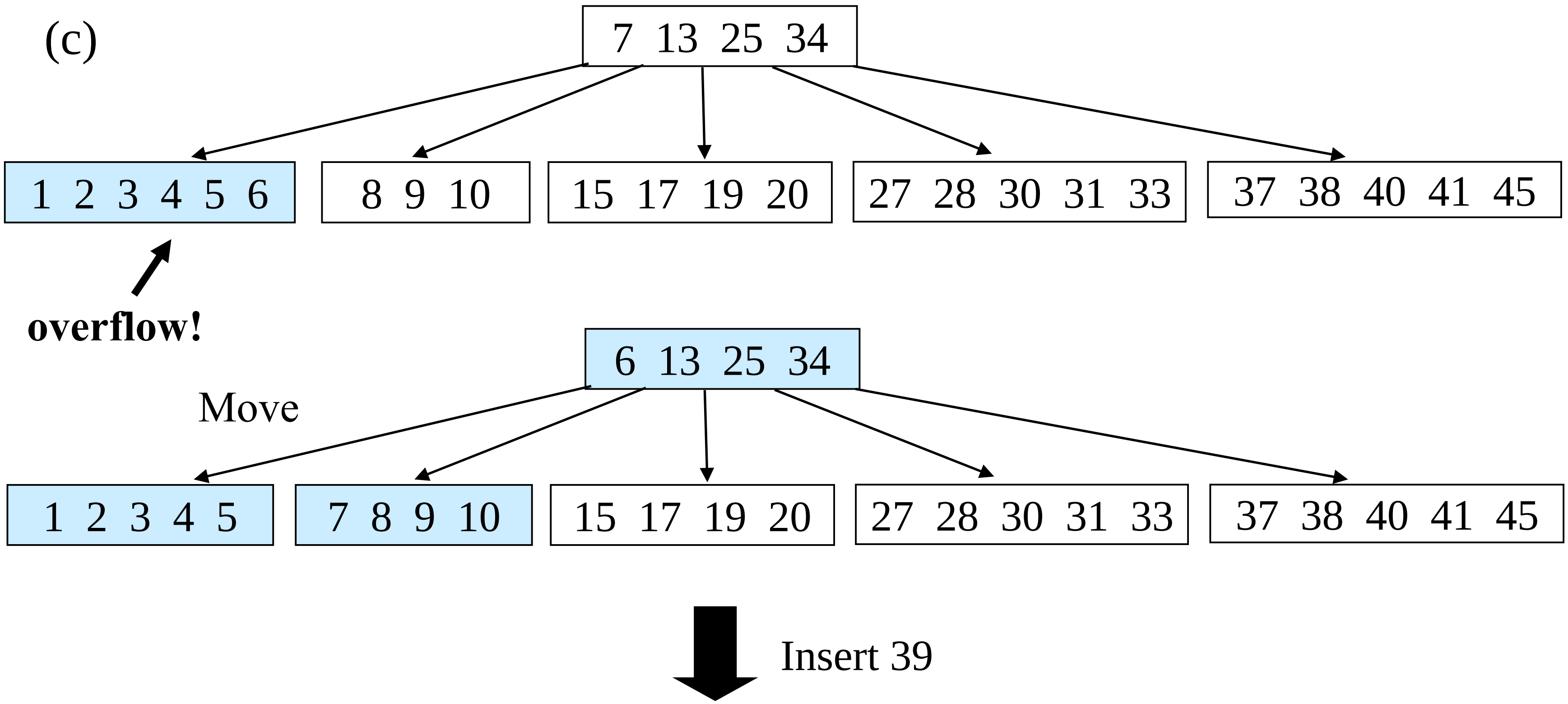
Insert in B-Tree

▷ t : root node
▷ x : key to be inserted

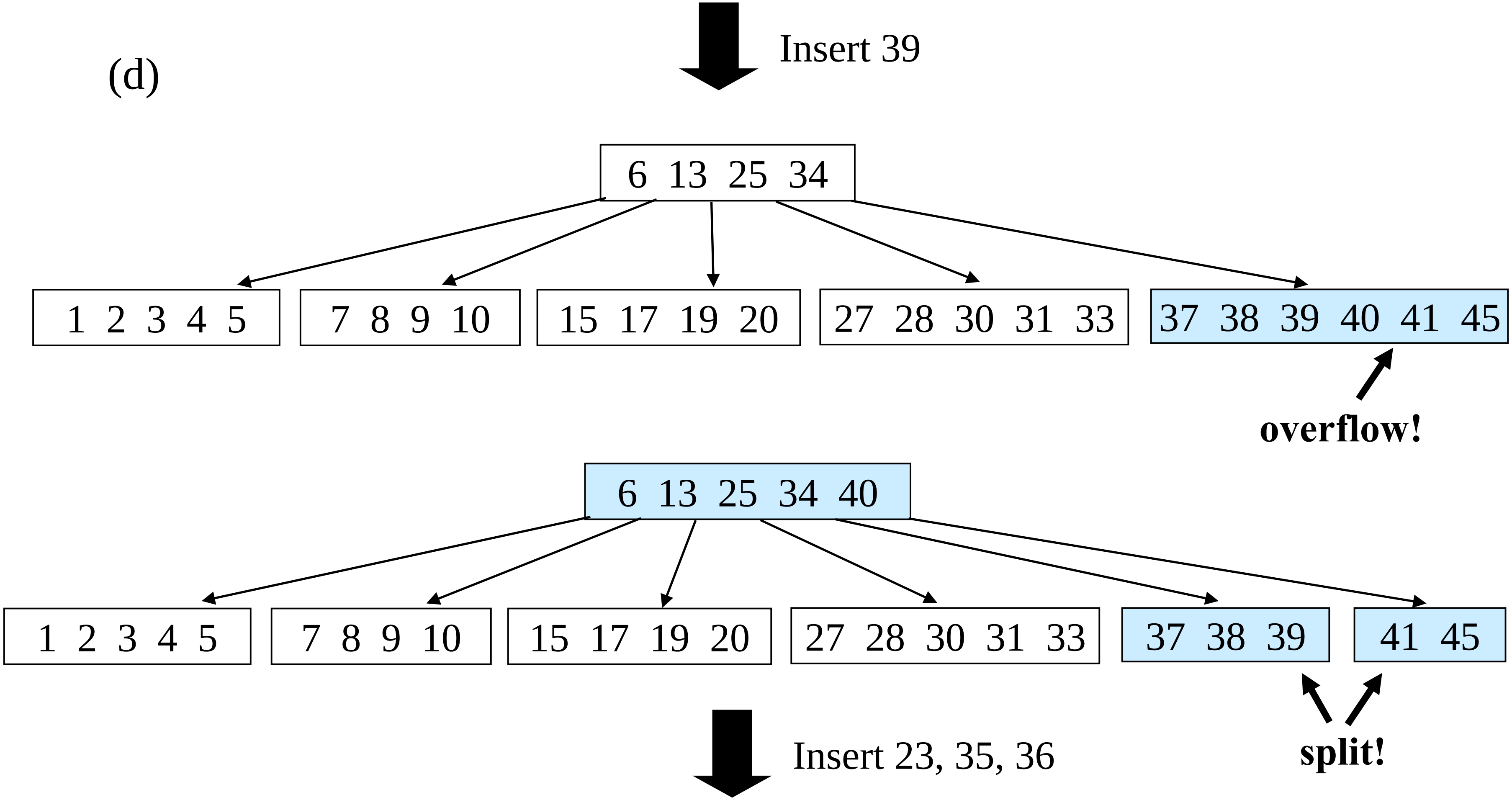
```
BTreeInsert(t, x)
{
        find leaf r where x should be inserted;
        insert x into r;
        if (overflow in r) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r's sibling s has room) then {move key in r to s};
    else {
          split r into two, and move middle key to parent p;
          if (overflow in parent p) then clearOverflow(p);
    }
}
```

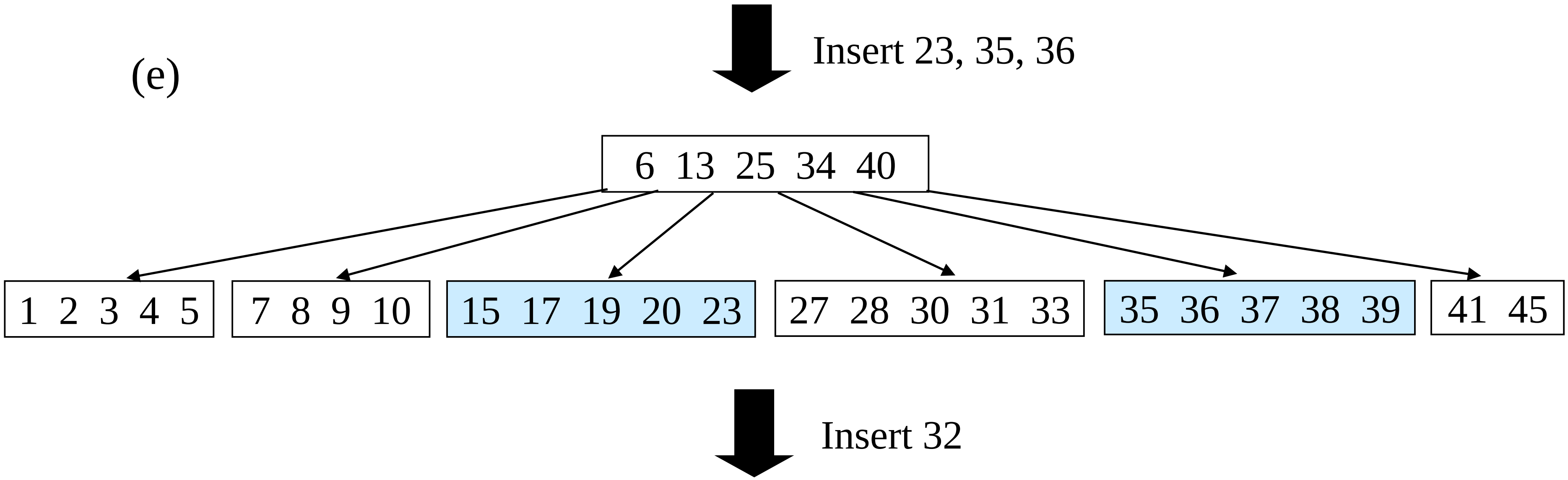
Insert in B-Tree



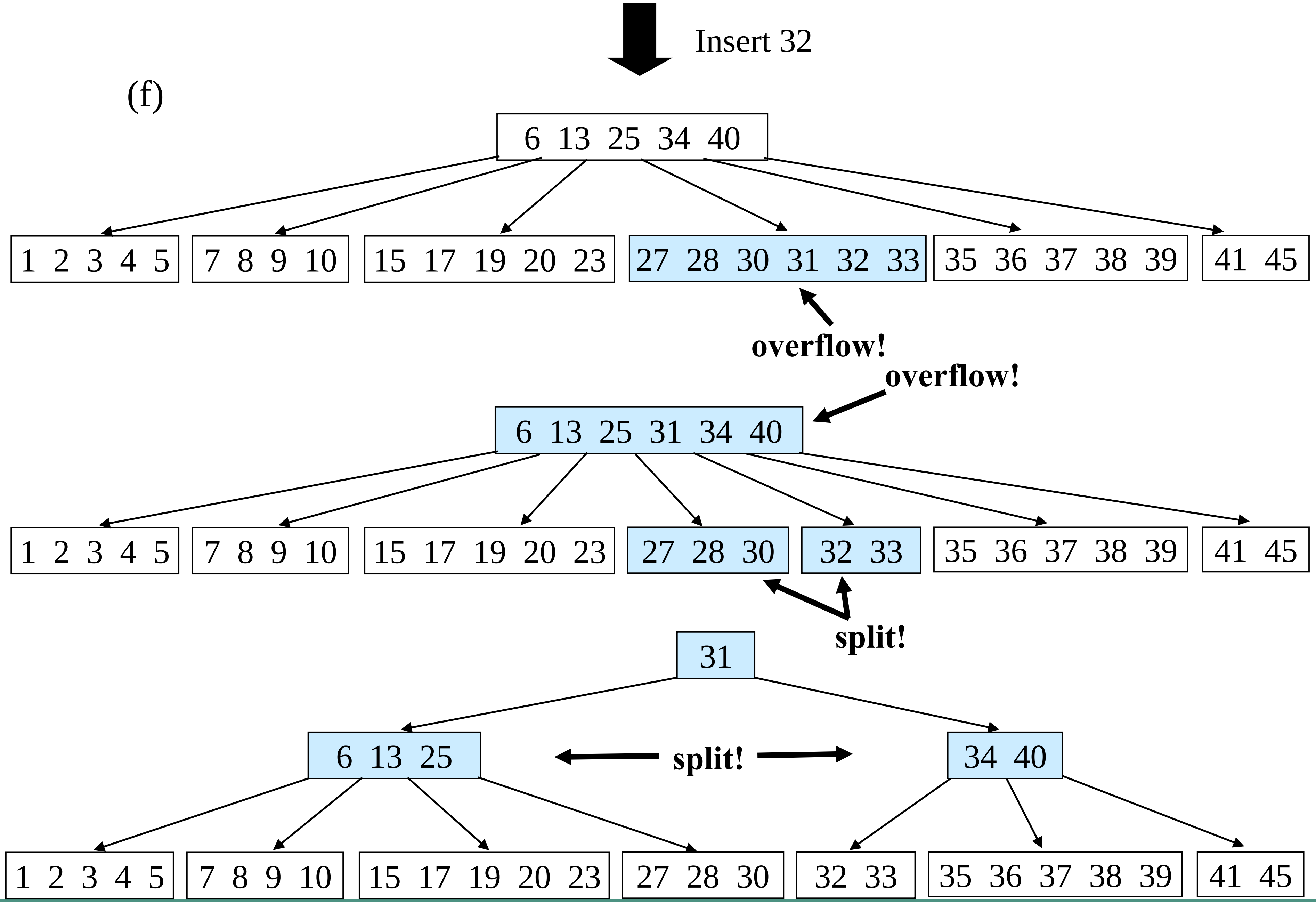

(c)
7 13 25 34
1 2 3 4 5 6
8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
overflow!
6 13 25 34
Move
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
Insert 39

(d)


Insert 39
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39 40 41 45
overflow!
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39
41 45
split!
Insert 23, 35, 36


Insert 23, 35, 36
(e)
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30 31 33
35 36 37 38 39
41 45
Insert 32

(f)



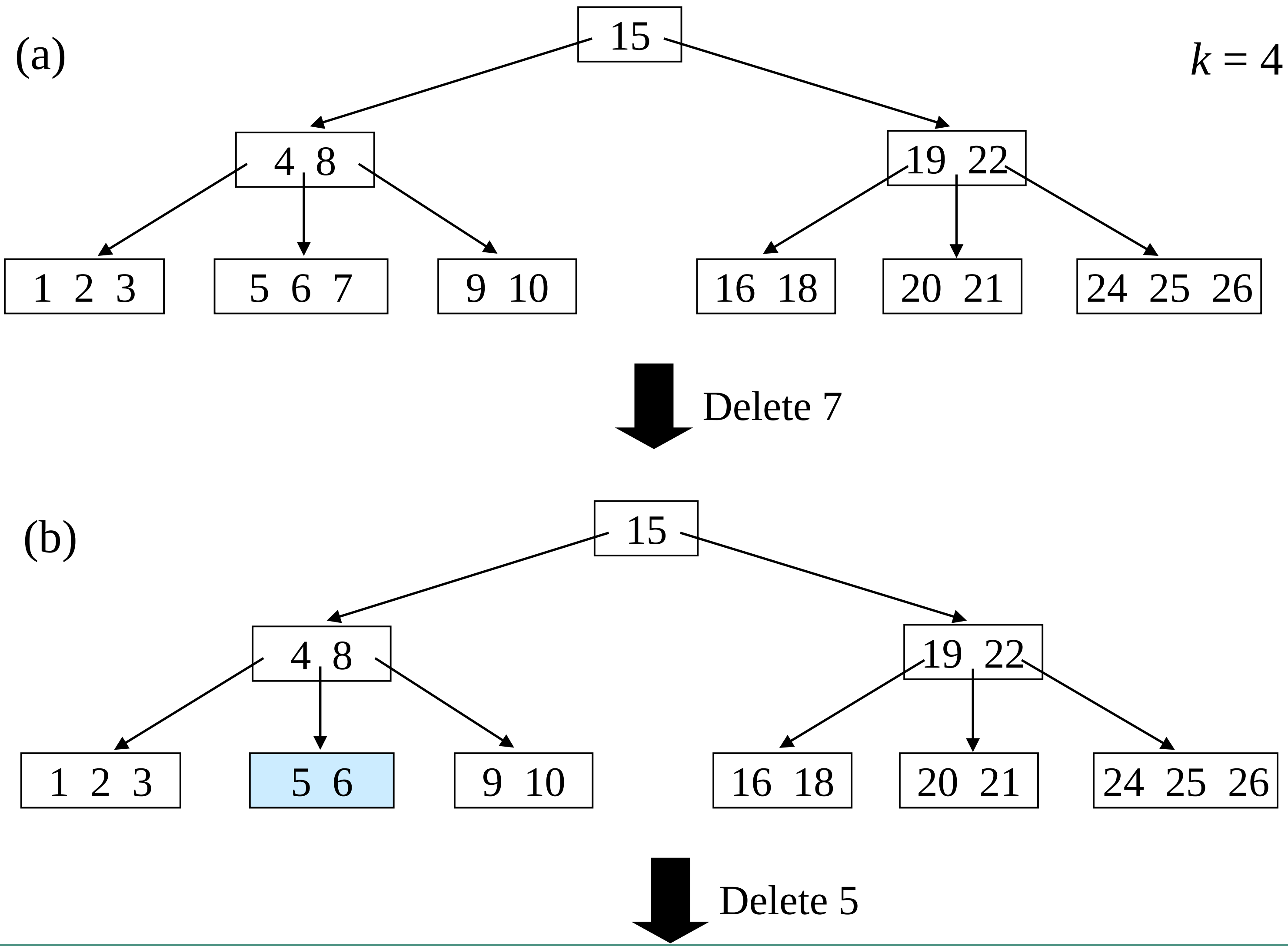
Insert 32
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30 31 32 33
35 36 37 38 39
41 45
overflow!
overflow!
6 13 25 31 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45
split!
31
6 13 25
split!
34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45

Delete in B-Tree

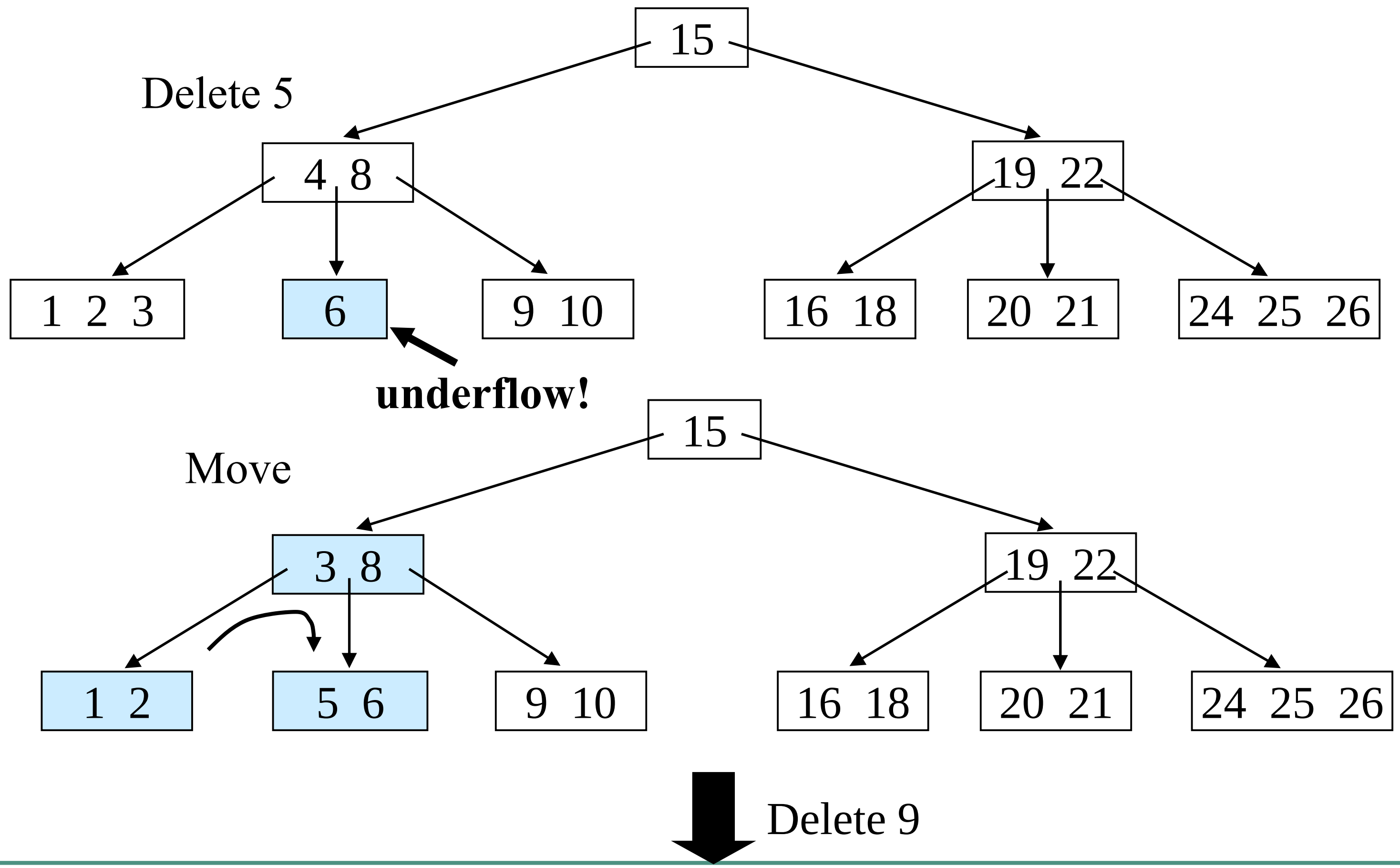
- t : root node
- x : key to be deleted
- v : node containing x

```
BTreeDelete(t, x, v)
{
    if (v not leaf) then {
        find leaf r that contains x's successor  y;
        swap x and  y;
    }
    delete x in  r;
    if (undeflow in r) then clearUnderflow(r);
}
clearUnderflow(r)
{
    if (r's sibling s has keys to move)
        then {move key in s to r;}
        else {
                merge r and s;
                 if (underflow in parent  p) then clearUnderflow(p);
        }
}
```

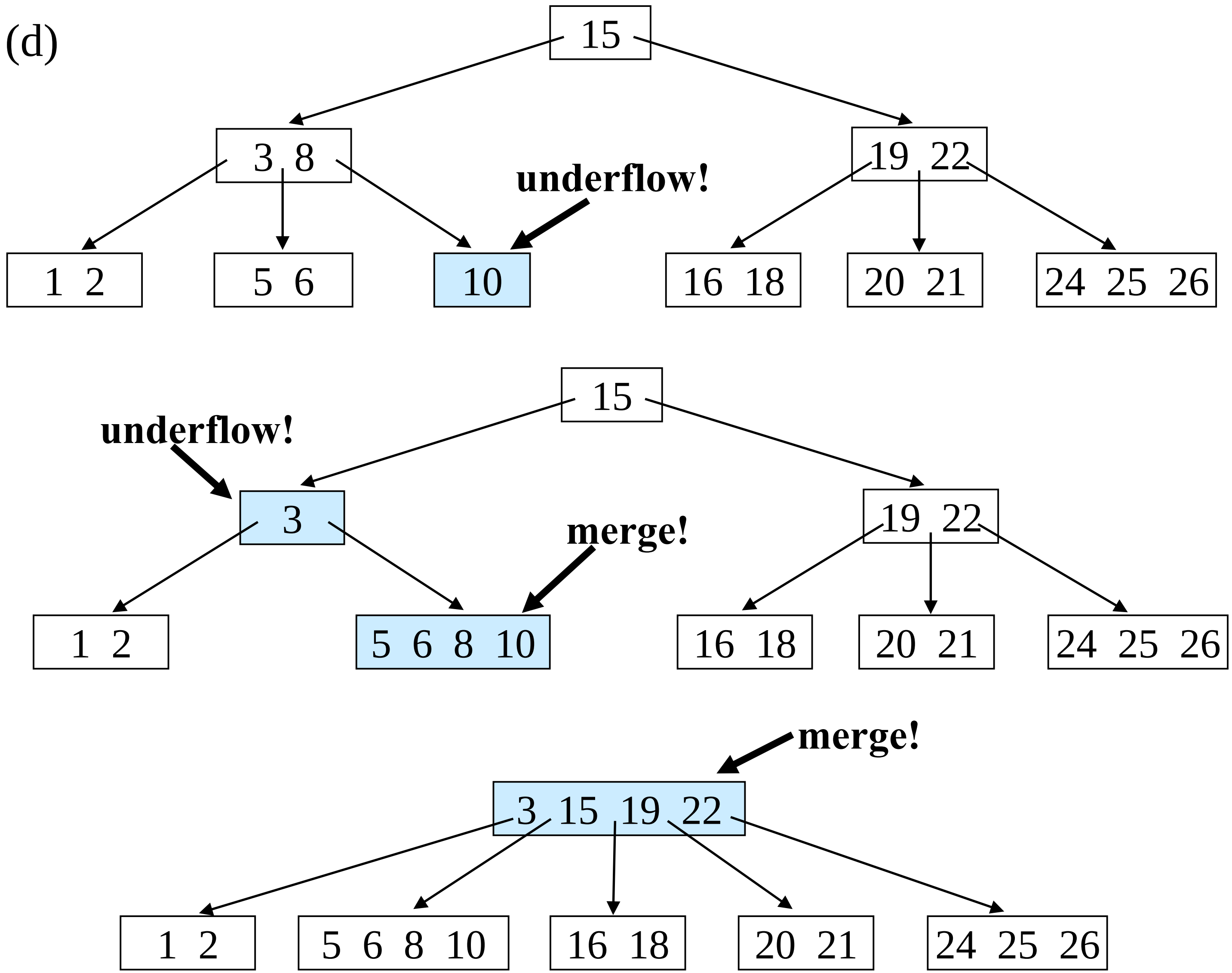
Delete in B-Tree



(c)



Delete 5
15
4 8
19 22
1 2 3
6
9 10
16 18
20 21
24 25 26
underflow!
Move
15
3 8
19 22
1 2
5 6
9 10
16 18
20 21
24 25 26
Delete 9


(d)
15
3 8
19 22
underflow!
1 2
5 6
10
16 18
20 21
24 25 26
15
underflow!
3
merge!
19 22
1 2
5 6 8 10
16 18
20 21
24 25 26
merge!
3 15 19 22
1 2
5 6 8 10
16 18
20 21
24 25 26