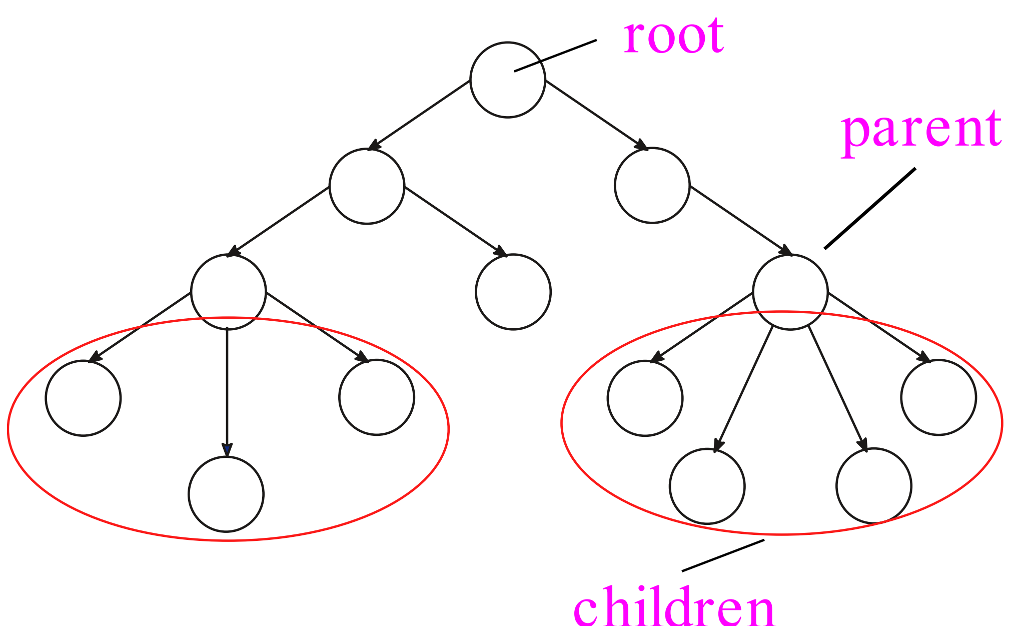
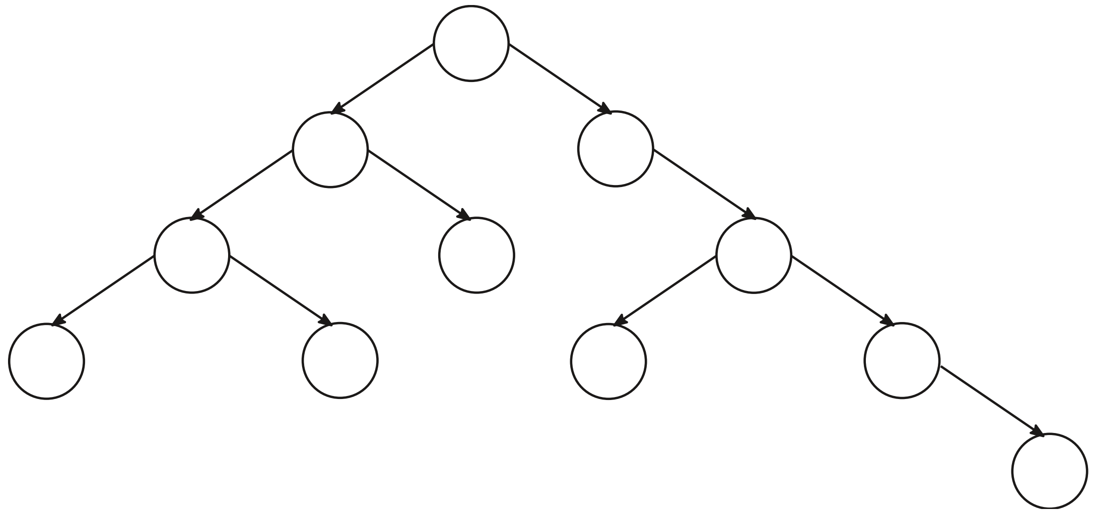
Tree and binary tree



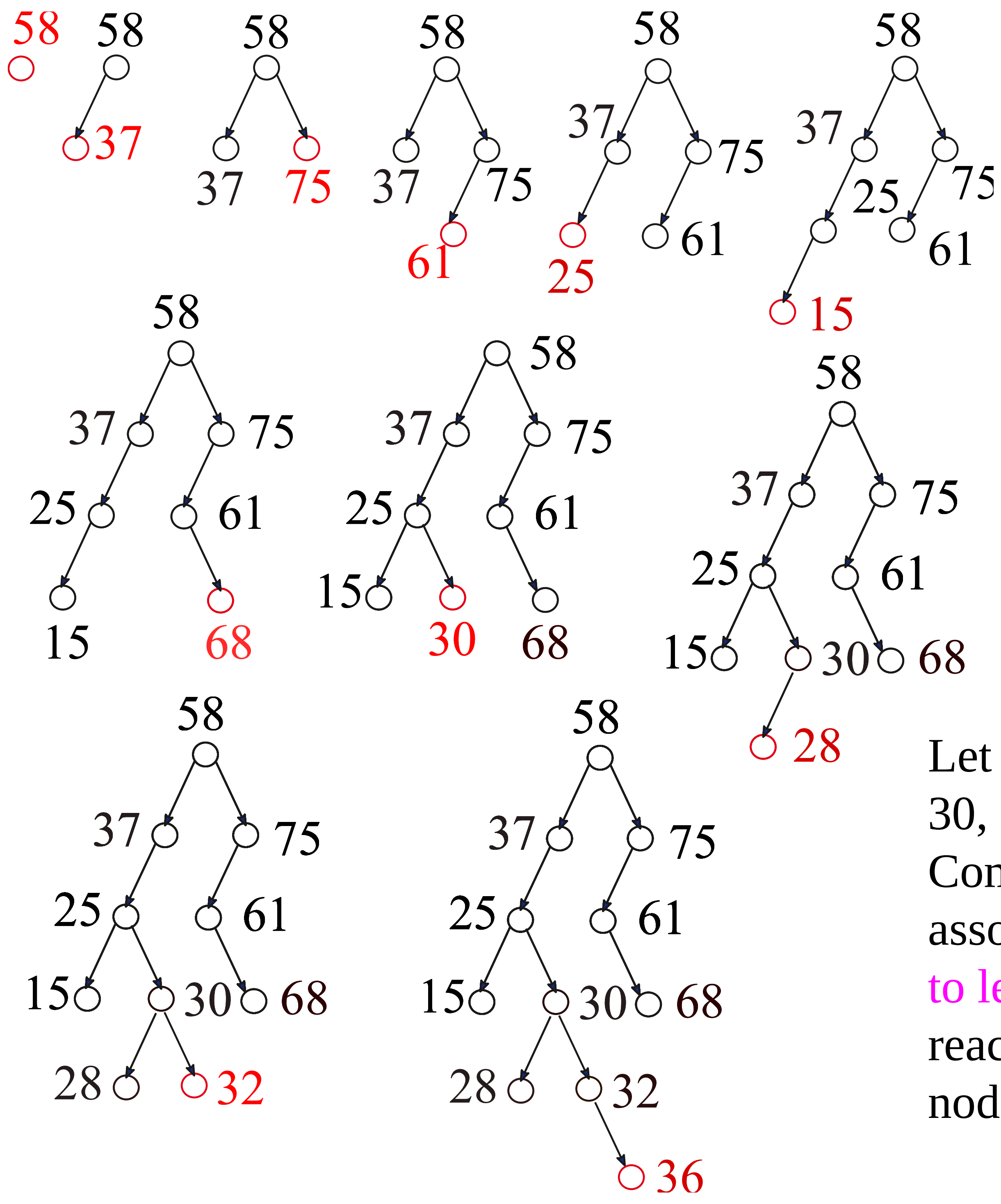
Leaf is a node which does not have children.
Siblings are children of the same parent.
The tree is ordered if the order of siblings (from left to right) has set by a special rule.
The root is on level 0, its children on level 1, etc.
Number of edges in the path from a leaf located on the lowest level to the root is the height of tree.



In binary tree the number of children can be 0, 1 or 2.

A tree is a collection of nodes. This collection may be empty, contain only one (root) node or root node with connected to it branches. But the branches are also trees.
This is the recursive definition of tree.

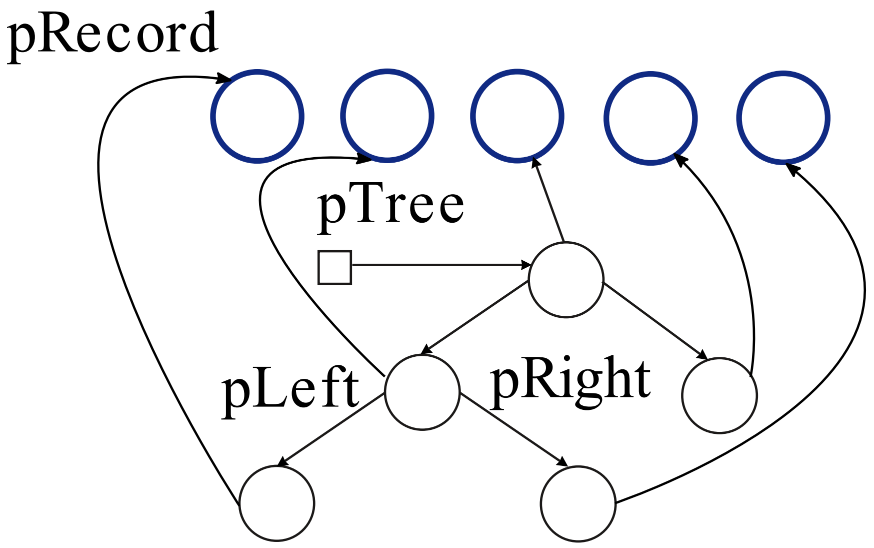
Binary search tree (1)



Let us have keys 58, 37, 75, 61, 25, 15, 68, 30, 28, 32, 36.
Compare the new key with the key associated with the current node. Less – go to left. Greater – go to right. If you have reached an empty place, build the new node.

Binary search tree (2)

```
struct node
{
      void *pRecord = nullptr;  // pointer to the associated record
      node *pLeft = nullptr,       // pointer to the left child
               *pRight = nullptr;     // pointer to the right child
 };
node *pTree = nullptr;  // create empty tree
```



Building: compare the new key with the key associated with the current node. Less – go to left. Greater – go to right. If p*Left* or *pRight* is *nullptr*, create the new node.
Searching: compare the searched key with the key associated with the current node. Identical – you have got it. Less – go to left. Greater – go to right. If p*Left* or *pRight* is *nullptr*, the needed record does not exist.

Binary search tree (3)

```
void *TreeSearch(node *pTree, void *pKey, int (*pCompare)(const void *, const void *))
{  // recursive function for searching from binary tree
      int i;
      if (!pTree || !pKey)
          return nullptr;
      if (!(i = (pCompare) (pKey, pTree->pRecord)))
          return pTree->pRecord;  // found
       else if (i < 0)
          return TreeSearch(pTree->pLeft, pKey, pCompare); // continue in left branch
       else
          return TreeSearch(pTree->pRight, pKey, pCompare); // continue in right branch
}
```

pCompare is a pointer to function (will be discussed later in this course). It may point to the following function:

```
int CompareKeys(const void *pKey, const void *pRecord)
{
       return strcmp((const char *)pKey, ((const PERSON *)pRecord)->pName);
}
```

Example:

```
Person *pStudent = (Person *)TreeSearch(pTree, "John Smith", CompareKeys);
```

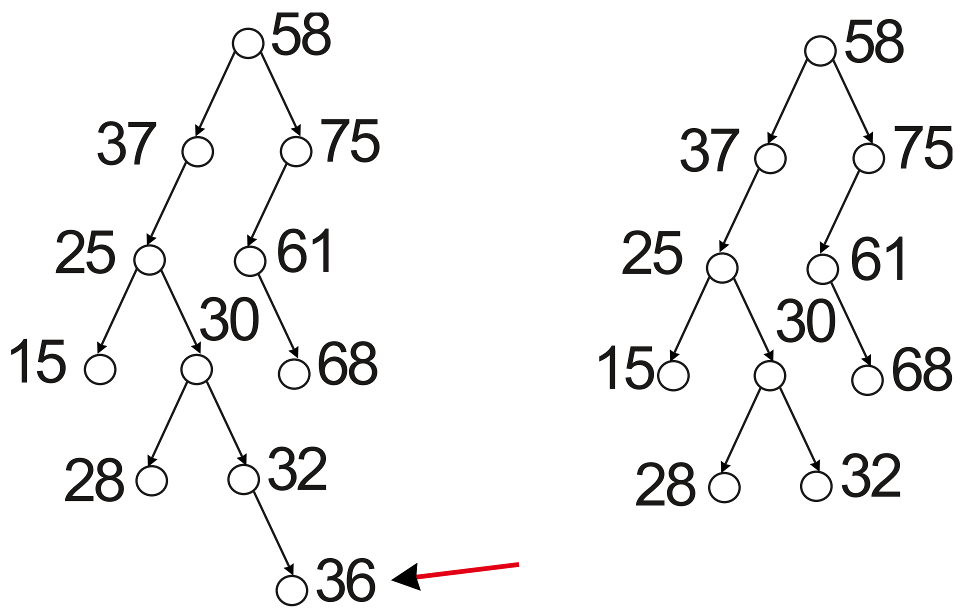
Binary search tree (4)

```
void *TreeSearch(node *pTree, void *pKey, int (*pCompare)(const void *, const void *))
{  // searching from binary tree without recursion
      int i;
      node *p = pTree;
      if (!pTree || !pKey)
           return nullptr;
      for (; p; )
      {
             if (!(i = (pCompare) (pKey, p->pRecord)) <  0)  // call by pointer, discussed later
                   p= p->pLeft;  // go to left
              else if (i > 0)
                    p = p->pRight; // go to right
               else
                    return p->pRecord;  // got it

         }
         return nullptr;  // not found
}
```

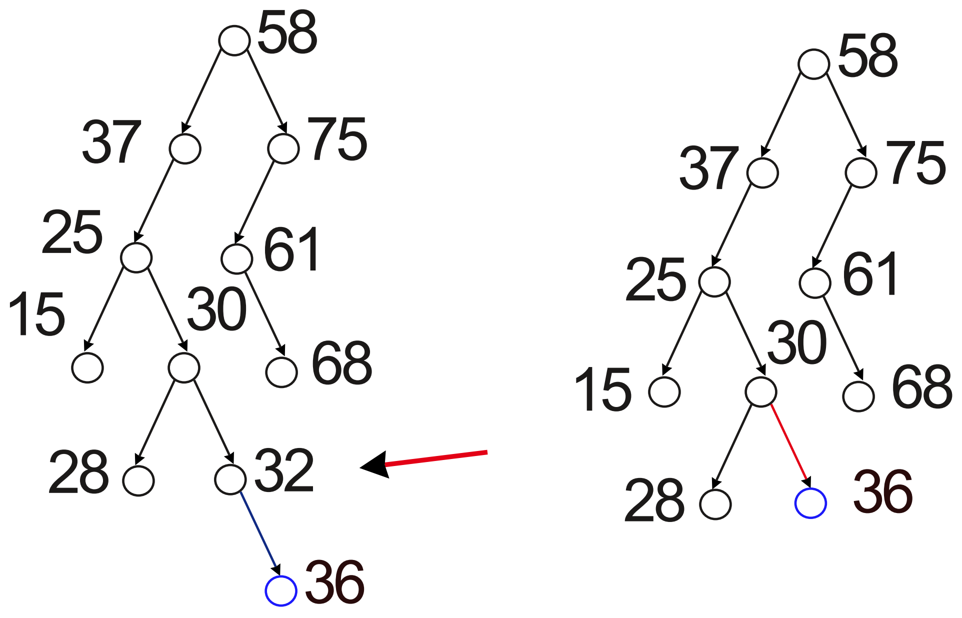
Binary search tree (5)

```
node *InsertNode(node *pTree, void *pNewRecord,
                              int (*pCompare)(const void *, const void *))
{  // inserting a new node without recursion
     node *pNew = new node; // create new node
     pNew->pRecord = pNewRecord;
     if (!pTree)
          return pNew;  // the tree was empty, the new node is the root
     for (node *p = pTree; true; ) {
            if (!((pCompare) (pNewRecord, p->pRecord) <  0) {
                  if (!p->pLeft) {  // found an empty place
                      p->pLeft = pNew;  // connect to the tree
                      return pTree;  // ready
                   }
                   else
                      p = p->pLeft;  // move left
             }
             else  {
                  // similar code for pRight
             }
        }
 }
```

Binary search tree (6)

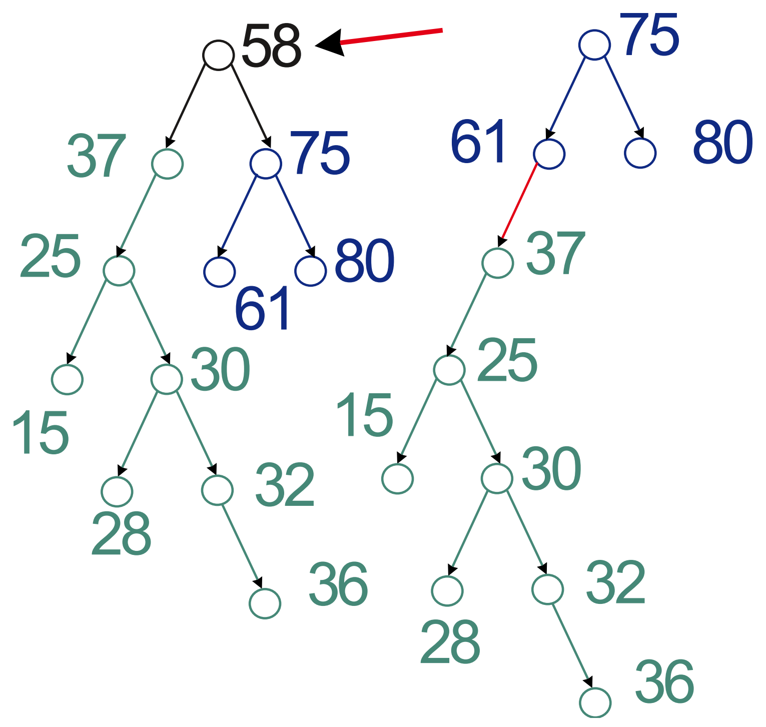


There are no problems to remove a leaf node.

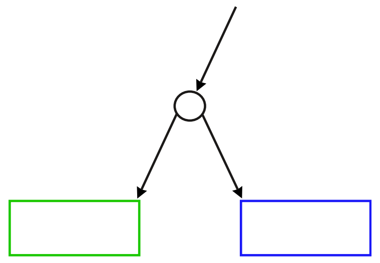


Also, it is easy to remove a parent node with one child.

Binary search tree (7)



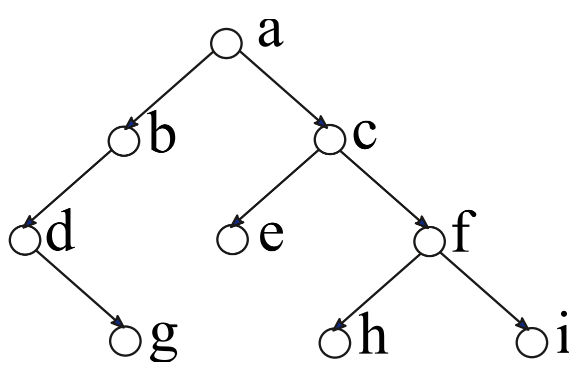
When a parent with two children has been removed, we get two branches not connected to eeach other. Any key in the right branch is greater than keys from the left branch:



To find the node associated with the smallest key move left until possible. To find the node associated with the biggest key move right until possible.

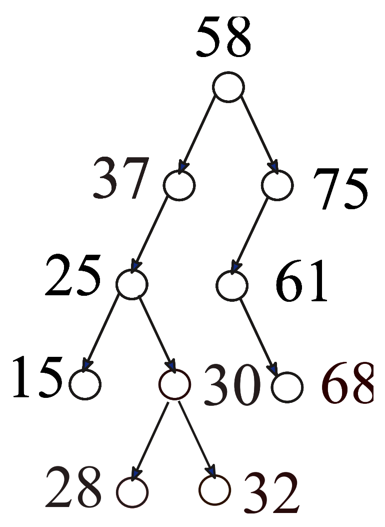
Find the minimum of the right branch. Then connect the root of left branch to it as its left child.

Binary search tree (8)



The tree traversal is the process of visiting each node exactly once. Actually, the traversal is the linearization of tree. The traversal is a recursive process.

Postorder travesal (left – right - root): g, b, d, e, h, i, f, c, a
Preorder traversal (root – left- right): a, b, d, g, c, e, f, h, i
Inorder traversal (left – root – right): d, g, b, a, e, c, h, f, i
Level-order traversal: a, b, c, d, e, f, g, h, i



Postorder travesal (left – right - root): 15, 28, 32, 30, 25, 37, 68, 61, 75, 58
Preorder traversal (root – left- right): 58, 37, 25, 15, 30, 28, 32, 75, 61, 68
Inorder traversal (left – root – right): 15, 25, 28, 30, 32, 37, 58, 61, 68, 75. The list is sorted.
Level-order traversal: 58, 37, 75, 25, 61, 15, 30, 68, 28, 32

See also https://builtin.com/software-engineering-perspectives/tree-traversal#4

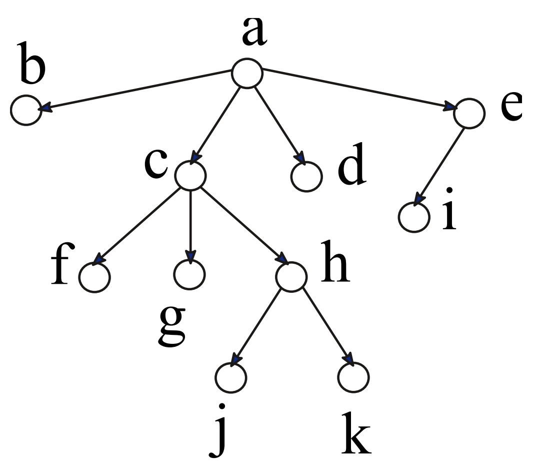
Binary search tree (9)

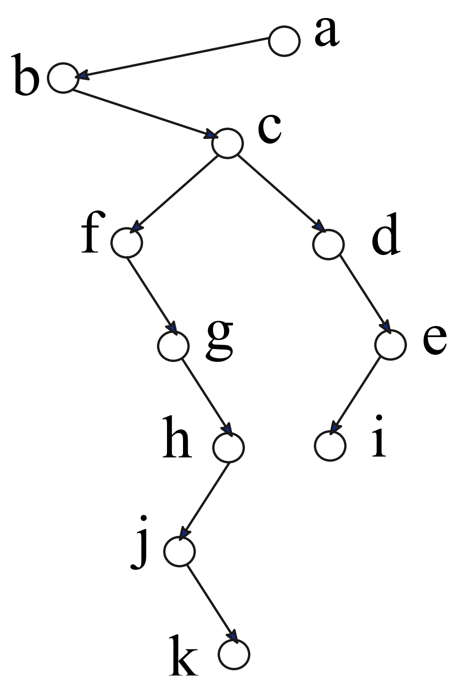
```
void InorderTraversal(node *pTree, void (*pProcess(node *))
{  // recursive inorder traversal, pProcess points to function that does something with node
   // (for example prints the contents of associated record)
     if (pTree)
     {
          InorderTraversal(pTree->pLeft, pProcess)
          (pProcess)(pTree);
          InorderTraversal(pTree->pRight, pProcess);
      }
}

void DestroyTree(node *pTree)
{   // recursive postorder traversal for erasing the tree
     if (pTree)
     {
           DestroyTree(pTree->pLeft);
           DestroyTree(pTree->pRight);
           free(pTree);
      }
}
```

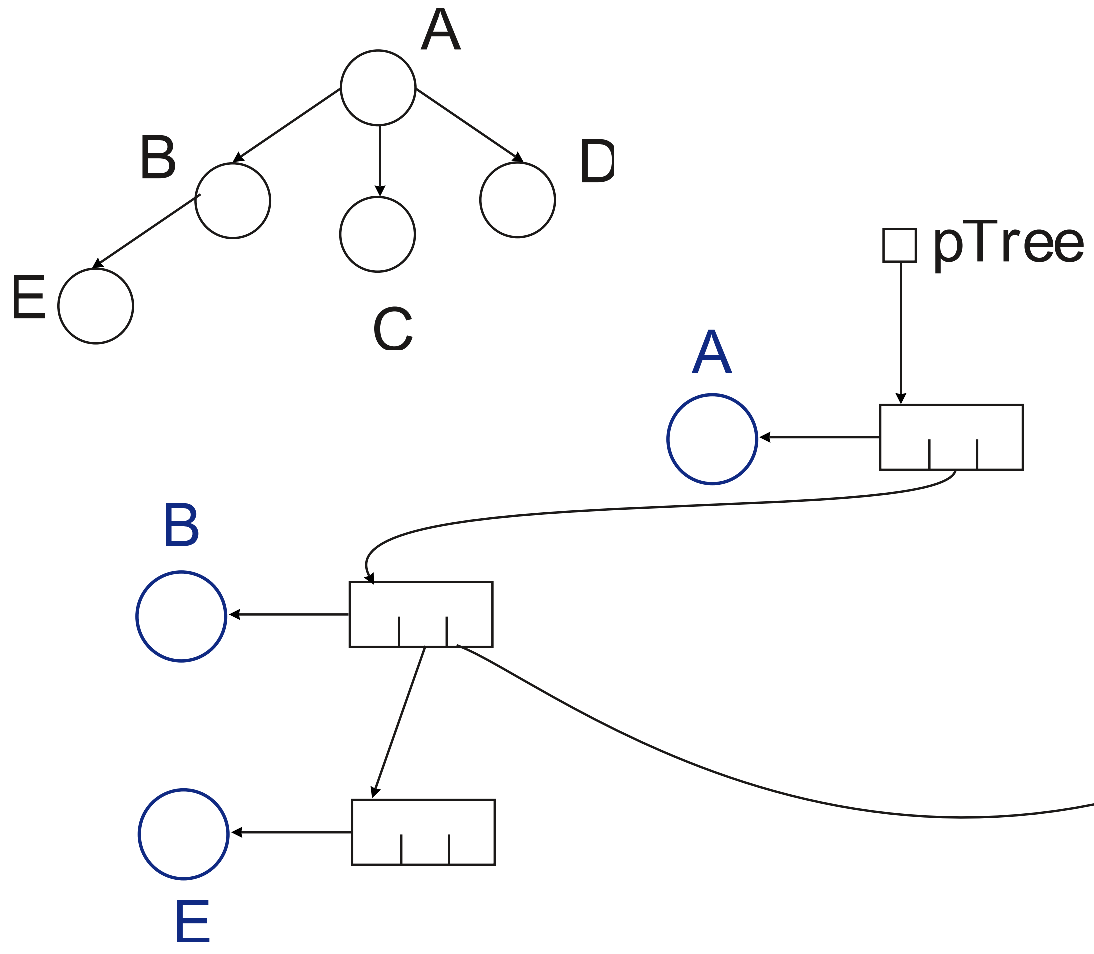
Binary search tree (10)

A multiway tree can be reduced to binary tree:

```
struct node
{
  void *pRecord = nullptr;
  node *pChild = nullptr,  // pointer to leftmost child
          *pSibling = nullptr;  // pointer to next right sibling
};
```

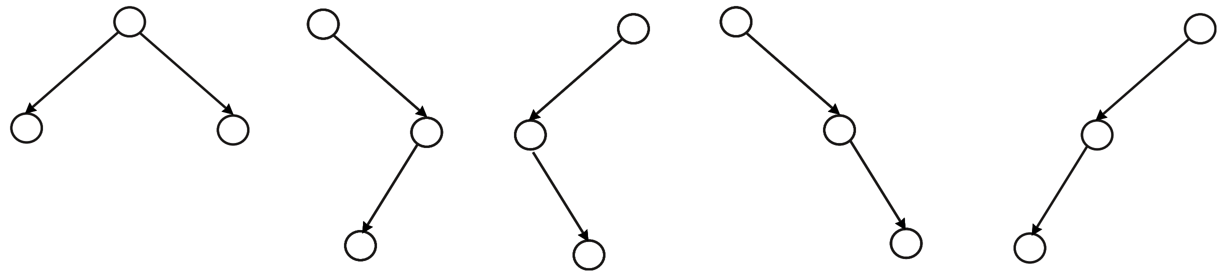




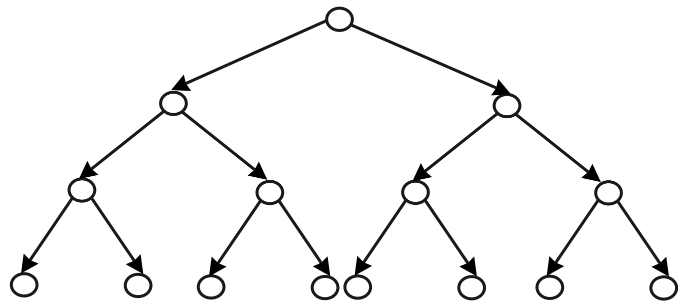


Binary search tree (11)

Suppose we have 3 records. Depending on their arrival order we may get one of the following trees:



The look like of a tree is occasional. In the worst case the tree is actually a linear list. The best case is the perfectly balanced tree in which the heights of the both subtrees proceeding from any parent node is equal:



However, the perfectly balanced tree can be built only if the number of nodes $n = 2^k - 1$ (k is an integer). For this tree $h(n) \leq \log_2 (n + 1)$.

For random trees:

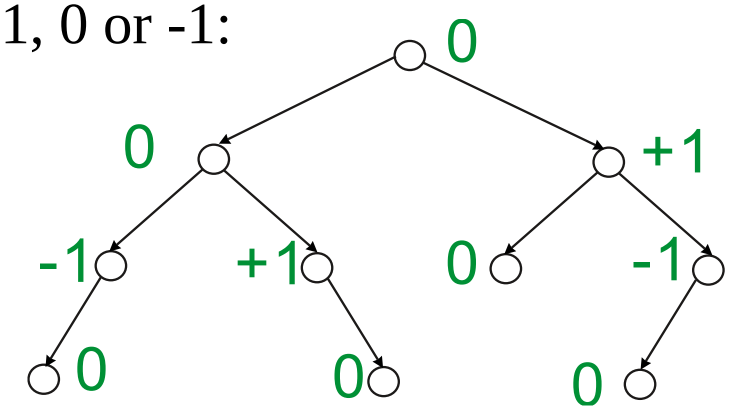
$\log_2 (n + 1) < h(n) \leq n$ if $n \neq 2^k - 1$

$\log_2 (n + 1) \leq h(n) \leq n$ if $n = 2^k - 1$

$h(n)$ is the random height. The number of tests in searching is $T(n) \leq h(n)$.

AVL tree (1)

An AVL tree (Adelson-Velski & Landis, Kyiv 1962) is a binary tree in which the difference of heights of subtrees proceeding from any parent node is zero or one. The balance coefficient of a node is calculated as the height of right subtree minus the height of left subtree and it can be +1, 0 or -1:



In AVL tree:

$T(n) \leq 1.4404 \log_2 (n+2) - 0.328.$

If $n > 500$:

$T(n) \approx \log_2(n) - 0.86$

To build an AVL tree, after insertion of a new node we must check the balance coeffcients. If some of them is +2 or -2, we need to modify the tree using rotation algorithms. See:

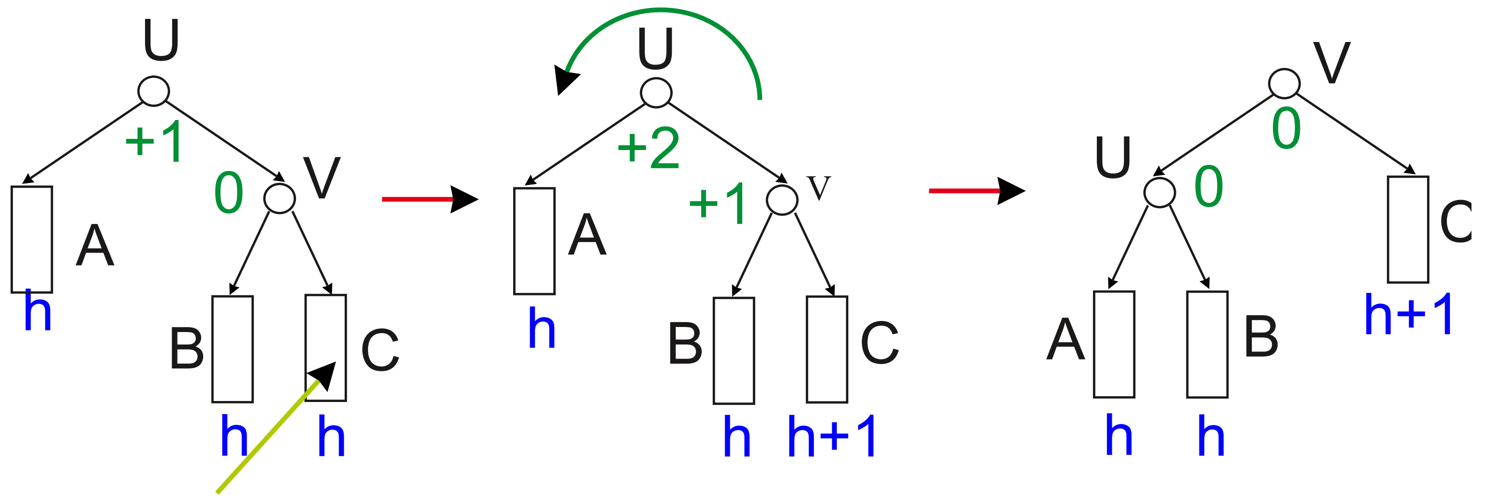
http://www.geeksforgeeks.org/avl-tree-set-1-insertion

http://www.geeksforgeeks.org/avl-tree-set-1-deletion

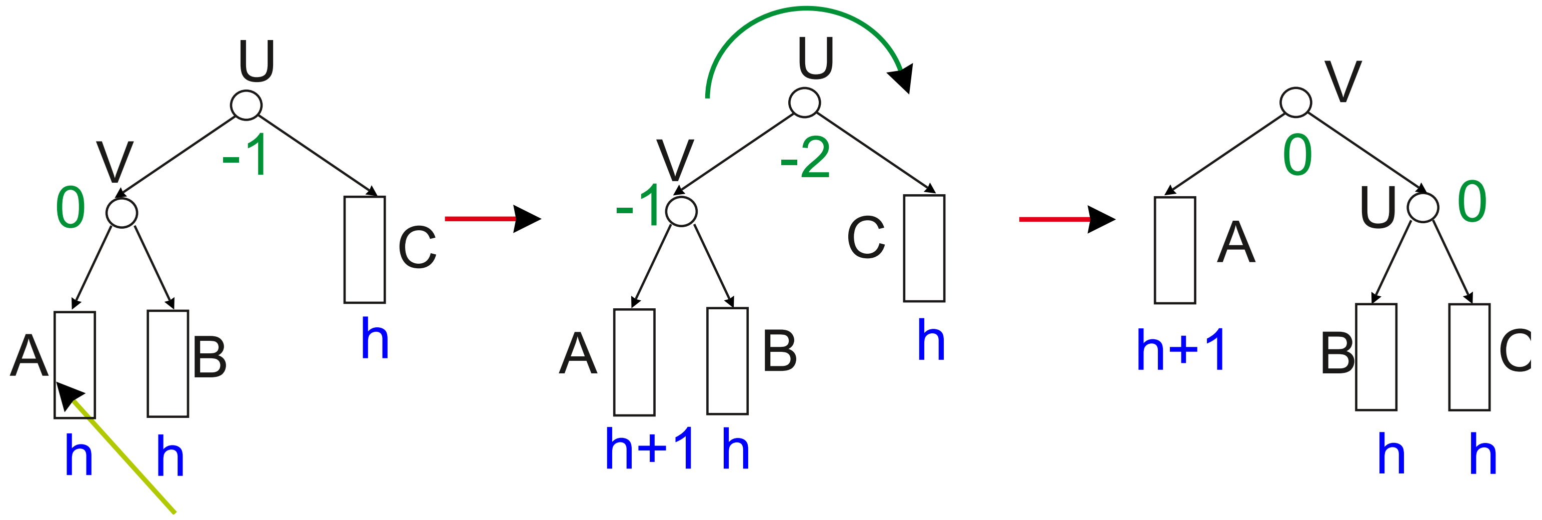
Those pages present also the corresponding code for C/C++functions.

AVL tree (2)

A, B and C are branches, h is the height of branch.

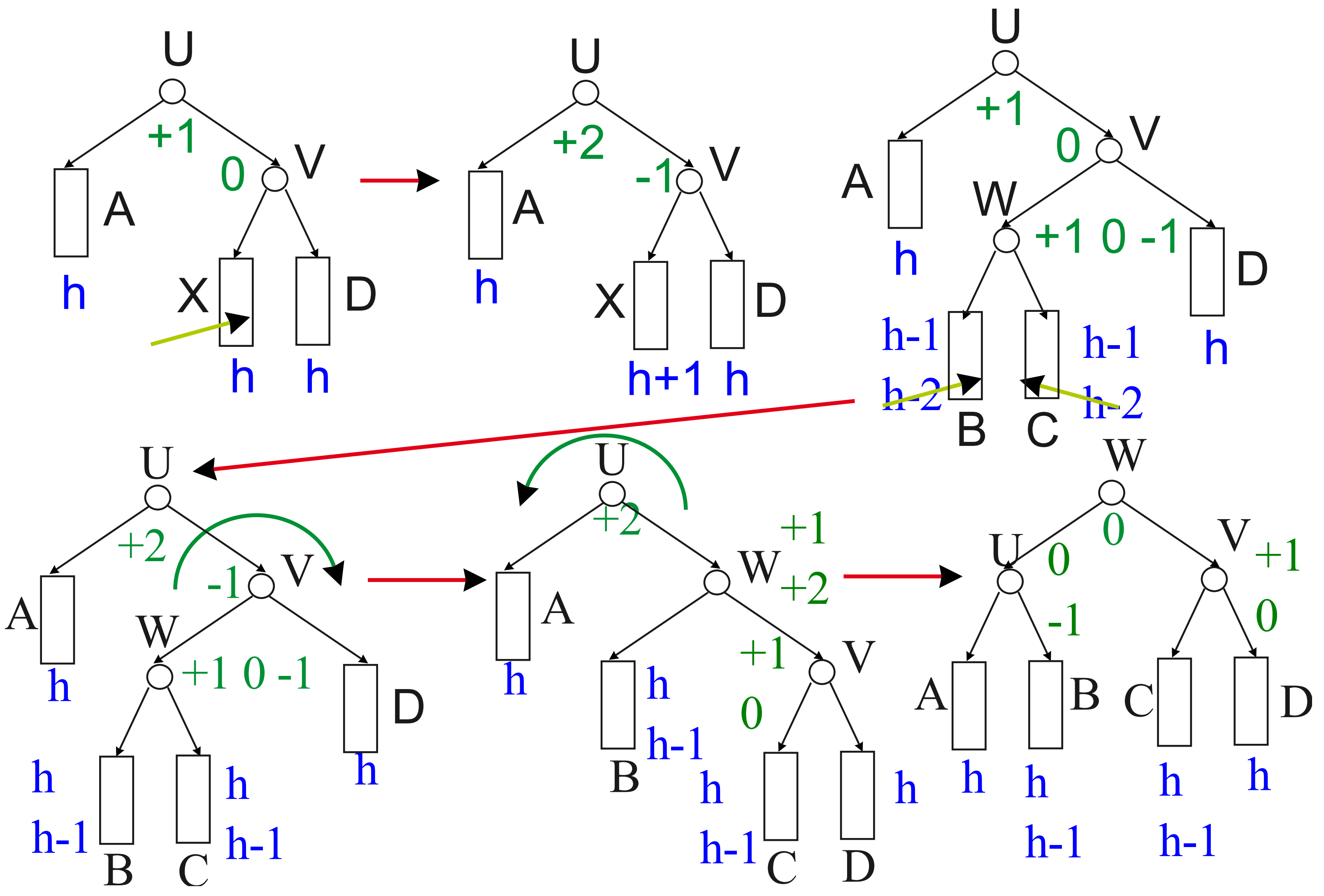


New node into branch C



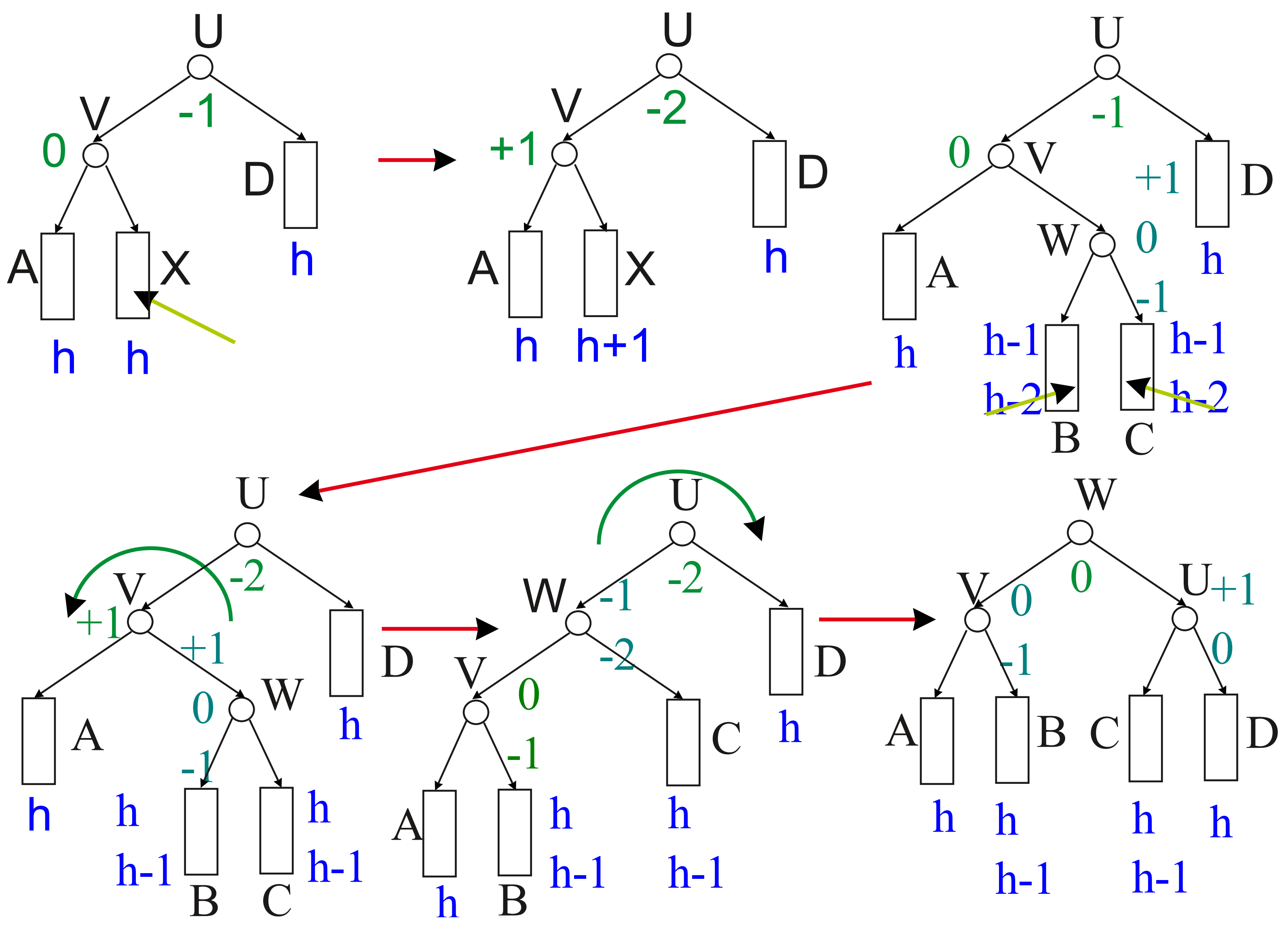
New node into branch A

AVL tree (3)



New node into branch X, after expanding into branch B or C.

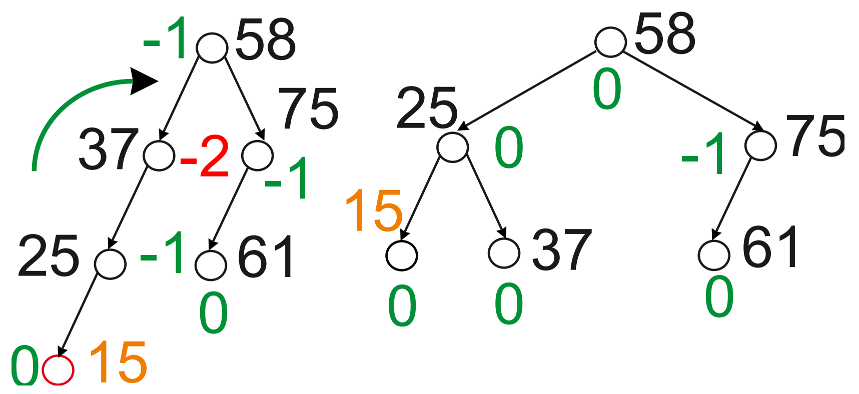
AVL tree (4)



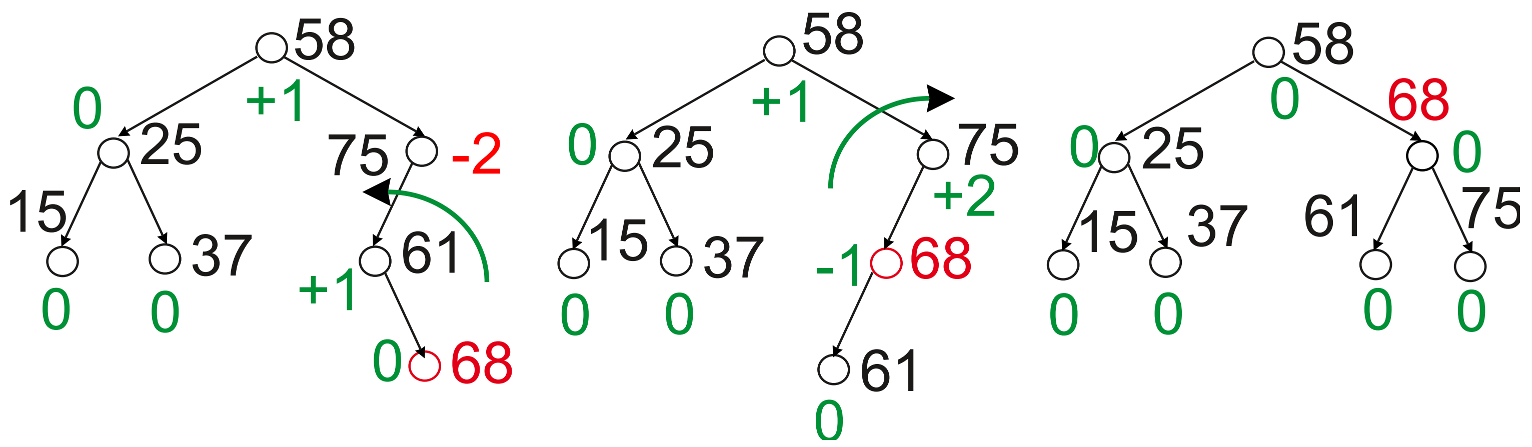
New node into branch X, after expanding into branch B or C.

AVL tree (5)

Example: let us have keys 58, 37, 75, 61, 25, 15, 68. The first 5 nodes are inserted without problems. After adding the node for key 15 our tree is not an AVL tree and we need to use rotation presented on slide *AVL tree (2)***:**

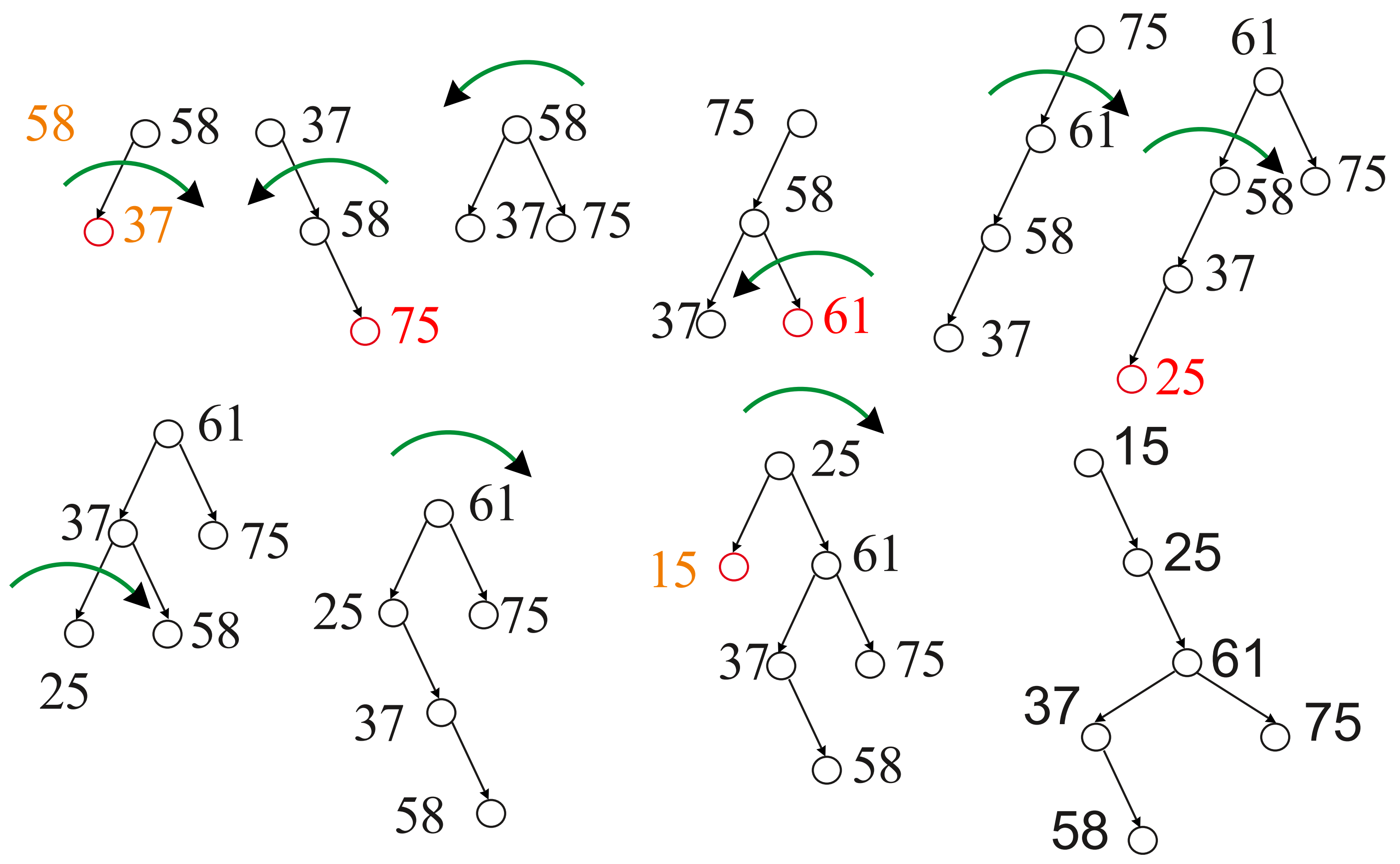


To restore the balance after adding node for key 68 we need to use two rotations presented on slide *AVL tree(4)***:**



Splay tree

In splay trees if a node is found, it is pushed to the root by a series of rotations. A new node is inserted into tree as usual but right after that it is also pushed to the root. Thus, after some time the nodes that are queried more frequently will be automatically concentrated on the upper levels. For example, let us have keys 58, 37, 75, 61, 25, 15:



See:
https://www.geeksforgeeks.org/spaly-tee-set-1-insert
https://www.geeksforgeeks.org/spaly-tree-2-insert-delete

Bitwise operations (1)

Operations like addition, comparison, logical *AND*, etc. operate with bytes. C/C++ has also operations for handling bits. The operands of bitwise operations must be integers (*char*, *int*, *unsigned int*, etc.).

Bitwise negation ~ converts each bit 1 to bit 0 and each bit 0 to bit 1. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = ~c1; // get 0101 1010
printf("%u\n", (unsigned int)c2); // prints 90
```

Remember that there is also negation ! (logical *NOT*) that converts zero (*FALSE*) to 1 (*TRUE*) and any non-zero (*TRUE*) to 0 (*FALSE*).

Bitwise AND & performs bit-by-bit comparison of bits. If the both bits are 1, the resulting bit is also 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 & c2; // gets 0010 0000
printf("%u\n", (unsigned int)c3); // prints 32
```

Remember that there is also logical AND && in which *TRUE && TRUE = TRUE* and all the other combinations produce *FALSE*.

Bitwise operations (2)

Bitwise OR | performs bit-by-bit comparison of bits. If the both bits are not 0, the resulting bit is 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 | c2; // gets 1010 0101
printf("%u\n", (unsigned int)c3); // prints 165
```

Remember that there is also logical OR || in which *FALSE* || *FALSE* = *FALSE* and all the other combinations produce *TRUE*.

Bitwise exclusive OR ^ (XOR) performs bit-by-bit comparison of bits. If the both bits are different, the resulting bit is 1, otherwise 0. Example:

```
unsigned char c1 = 0xA5, c2 = 0x20; // bits are 1010 0101 and 0010 0000
printf("%u %u\n", (unsigned int) c1, (unsigned int) c2); // prints 165 32
unsigned char c3 = c1 ^ c2; // gets 1000 0101
printf("%u\n", (unsigned int)c3); // prints 133
```

Bitwise operations (3)

Applying bits instead of bytes we can compress data. Suppose we need to describe properties of a file:

- reading allowed or not alllowed
- writing allowed or not allowed
- on open, if not found, create; if found inform about error
- on open, if found, delete the existing contents or keep it
- …………………………

Suppose there is no more that 8 properties. Then we may pack this information into one byte:

- If bit 7 is 1, reading is allowed; if 0, not allowed
- If bit 6 is 1, writing is allowed; if 0 not, not allowed
- If bit 5 is 1, create the file if not found; if 0 consider that file open operation failed
- If bit 4 is 1, destroy the contents of existing files; if 0 keep it
- ………………………………………………………….

Here bit 7 is the highest (leftmost) bit.

So, the function opening file does not need 9 parameters (filename and properties). 2 is enough – the name of file and properties packed into a variable of type *unsigned char*.

Bitwise operations (4)

Now suppose we have

```
unsigned char properties = 0;
```

and we want to open file both for reading and writing. For that we need to set bits 7 and 6 to 1:

```
properties = properties | 0xC0; // we may write also properties |= 0xC0;
// 0000 0000 | 1100 0000 gives us 1100 0000
```

Next we want to set that if the file exists, its contents must be destroyed:

```
properties |= 0x10; // 1100 0000 | 0001 0000 = 1101 0000
```

So, if we want to set a bit in the target variable to 1, we must bitwise *OR* the target with a constant in which this bit is 1 and all the others are 0. If the bit in the target variable already was 1, it keeps its value. If it was 0, it becomes 1.

The function opening the file must analyse the properties, i.e. to clarify which bits are 0 and which are 1. It can be done with bitwise AND, for example:

```
if (properties & 0x10)
{ // we get 0001 0000 that is TRUE or 0000 0000 that is FALSE
  ………………… // destroy file contents
}
```

So, if we need to know is a bit in the target variable 0 or 1, we must bitwise *AND* the target with a constant in which this bit is 1 and all the others are 0.

Bitwise operations (5)

If we want to set a bit in the target variable to 0, we must bitwise *AND* the target with a constant in which this bit is 0 and all the others are 1. If the bit in the target variable already was 0, it keeps its value. If it was 1, it becomes 0. Example:

```
unsigned char target = 0xD0; // 1101 0000
unsigned char mask = 0xEF; // 1110 1111
target = target & mask; // 1100 0000
```

or

```
target &= mask;
```

Toggling a bit means that if it was 1, it must be converted to 0 and if it was 0, it must be converted to 1. For that we have to bitwise *XOR* the target with a constant in which this bit is 1 and all the others are 0. Example:

```
unsigned char target = 0xD0; // 1101 0000
```

Toggle bit 4:

```
unsigned char mask = 0x10; // 0001 0000
target = target ^ mask; // 1100 0000
```

Toggle once more:

```
target = target ^ mask; // 1101 0000
```

or

```
target ^= mask;
```

Bitwise operations (6)

Binary bitwise shifting left << operation shifts all the bits of the value of left operand to the left by the number of places given by the right operand. The vacated places are filled with zeroes. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = c1 << 5; // gets 1010 0000, the higher bits were lost
printf("%u\n", (unsigned int)c2); // prints 160
unsigned char c3 = c1 << 8; // gets 0000 00000
printf("%u\n", (unsigned int)c3); // prints 0
```

Binary bitwise shifting right >> operation shifts all the bits of the value of left operand to the right by the number of places given by the right operand. The vacated places are filled with zeroes. Example:

```
unsigned char c1 = 0xA5; // bits are 1010 0101
printf("%u\n", (unsigned int) c1); // prints 165
unsigned char c2 = c1 >> 5; // gets 0000 0101, the lower bits were lost
printf("%u\n", (unsigned int)c2); // prints 5
unsigned char c3 = c1 >> 8; // gets 0000 00000
printf("%u\n", (unsigned int)c3); // prints 0
```

Shifting of negative signed values leads to unpredictable results.

Bitwise operations (7)

Example:

```
unsigned int color; // the higher byte is not used, the following bytes present the intensity
                    // of red, green and blue components. For example 0x00FF0000
                    // presents the most intensive red, 0x00FF00FF the most intensive
                    // magenta, 0x00007F00 dark green
unsigned char mask = 0xFF;
unsigned int red = (color >> 16) & mask;
unsigned int green = (color >> 8) & mask;
unsigned int blue = color & mask;
```

If the color is 0x00AA0000 (dark red) or 0000 0000 1010 1010 0000 0000 0000 0000, then shifting right 16 positions gives us 0000 0000 0000 0000 0000 0000 1010 1010. Before bitwise *AND* mask is automatically converted to unsigned int, so we get

```
  0000 0000 0000 0000 0000 0000 1010 1010
&
  0000 0000 0000 0000 0000 0000 1111 1111
   ---------------------------------------
  0000 0000 0000 0000 0000 0000 1010 1010 // intensity of red
```

Bitwise operations (8)

If the color is 0x00AA8000 (light brown) or 0000 0000 1010 1010 1000 0000 0000 0000, then shifting right 8 positions gives us 0000 0000 0000 0000 1010 1010 1000 0000.
Before bitwise *AND* mask is automatically converted to unsigned int, so we get

```
 0000 0000 0000 0000 1010 1010 1000 0000
```

&

```
 0000 0000 0000 0000 0000 0000 1111 1111
  ----------------------------------
0000 0000 0000 0000 0000 0000 1000 0000 // intensity of green
```

Bit fields (1)

Bit fields is the alternative way to handle separate bits. Suppose we want to store the parameters of font for a section of text. The font may bold, italic, underlined or double underlined or any combination of them. We may use a variable of type *unsigned char* and agree that bit 3 (7 is the highest) is 1 if the text is bold and 0, if not. Similarly bit 2 is 1 if the text is in italic and 0 if not, etc. But it is unpleasant to remember the meaning of each bit. The corresponding bit field may be as follows:

```
struct {
   unsigned char bold: 1;
   unsigned char italics: 1;
   unsigned char single_underlined: 1;
   unsigned char double_underlined: 1;
} font_par;
```

Now we can handle each bit as a member of struct, for example:

```
font_par.bold = 0;
font_par.italics = 1;
font_par. single_underlined = 1;
font_par.double_underlined = 0;
```

Description *unsigned char italics : 1* tells that the type of member *italics* is *unsigned char* but one bit is enough for storing its value.

Bit fields (2)

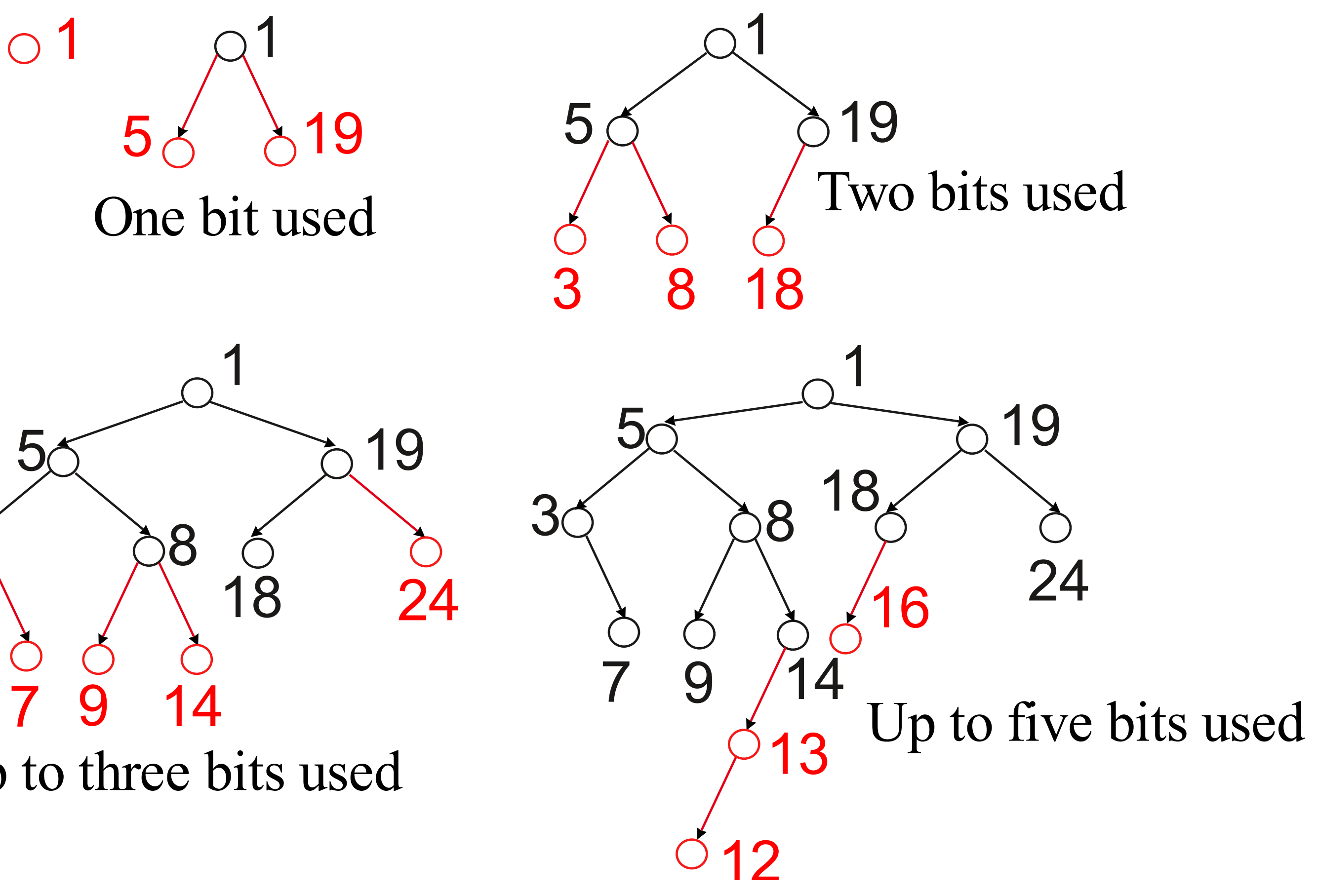
Number of bits for a bit field member may be greater than 1. For example in a date day cannot exceed 31 (0001 1111), month cannot exceed 12 (0000 1100) and the year cannot exceed 2020 (0111 1110 0100). So to economize the memory usage we can define

```
struct Date {
unsigned char day : 5;
unsigned char month : 4;
unsigned short int year : 11;
};
```

Bit field members can be integers, but not arrays or pointers.

Digital search tree

1	00001
19	10011
5	00101
18	10010
3	00011
8	01000
9	01001
14	01110
7	00111
24	11000
13	01101
16	10000
12	01100



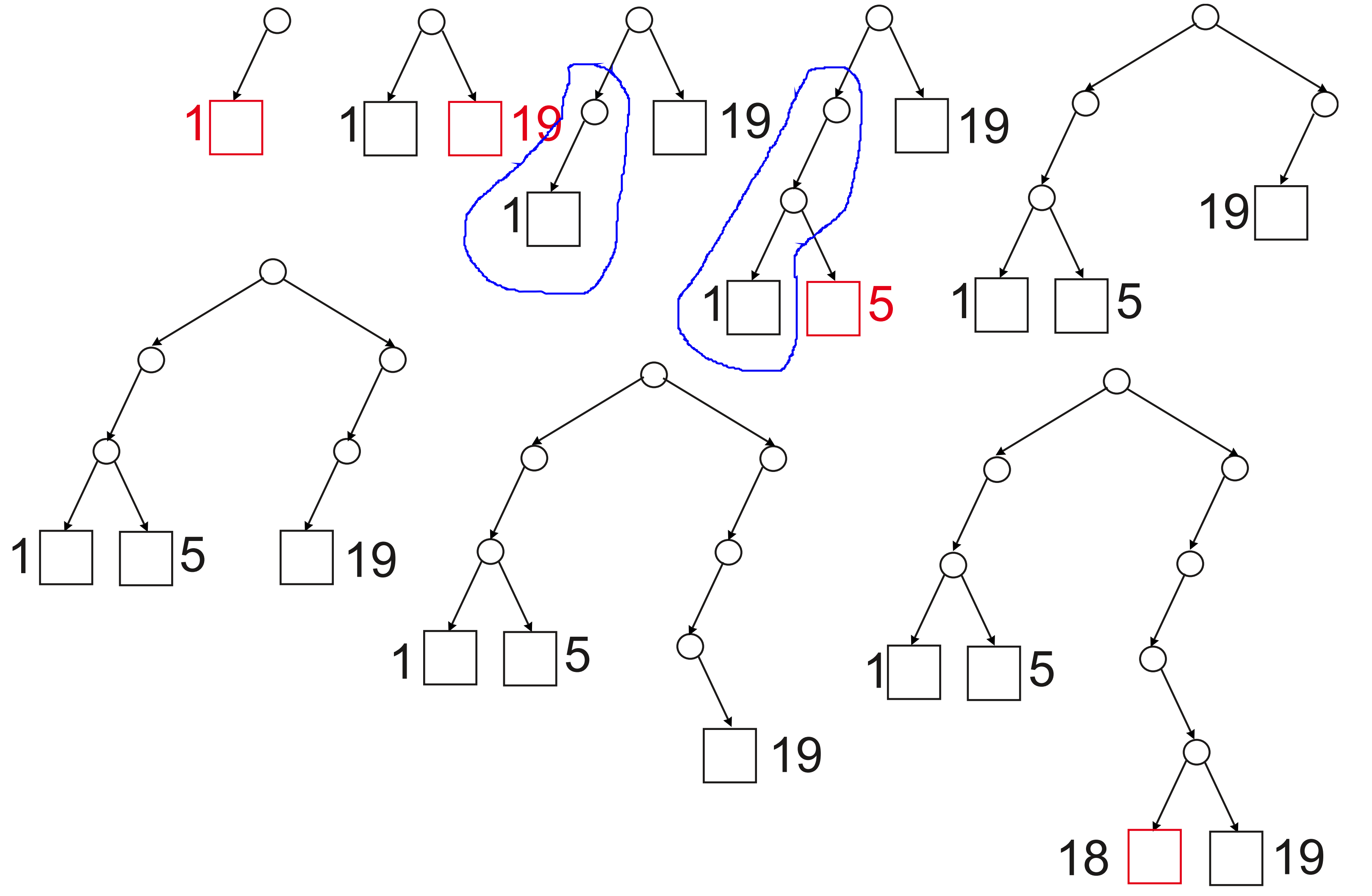
0-bit: go to left, 1-bit: goto right. To locate a node read bits from left to right. When you reach an empty place, put the node there.

If the keys (integers) do not exceed 2^n, the height of tree cannot exceed n +1. Thus, if the keys are *shot int*, the tree can contain max 65536 records and its height cannot exceed 17, i.e. the digital search tree is rather well balanced.

On searching you must move according to the bits but on each node the complete test is necessary.

Binary trie (1)

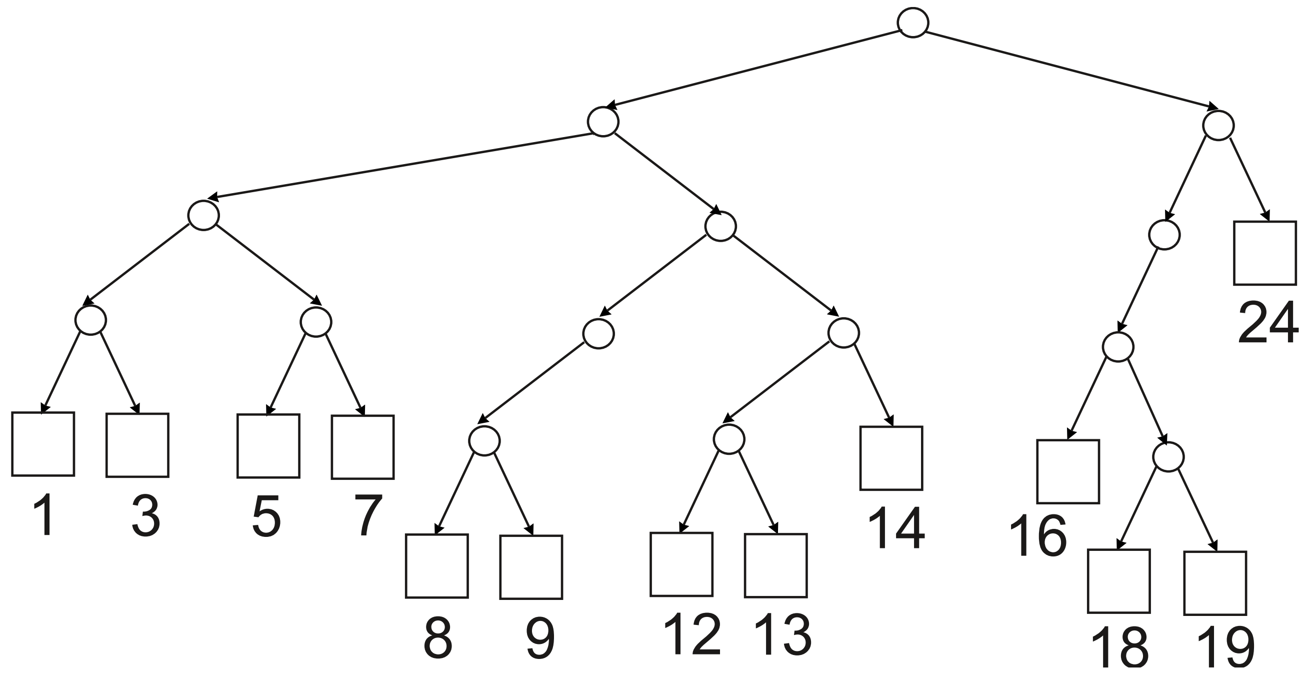
1	00001
19	10011
5	00101
18	10010
3	00011
8	01000
9	01001
14	01110
7	00111
24	11000
13	01101
16	10000
12	01100



Retrieve.
The records are in leaves (rectangles). 0-bit: go to left, 1-bit: goto right. To locate a leaf read bits from left to right. When you have reached an already existing leaf, take the next bit associated with this leaf and push the leaf down to the next level.

Binary trie (2)

1	00001
19	10011
5	00101
18	10010
3	00011
8	01000
9	01001
14	01110
7	00111
24	11000
13	01101
16	10000
12	01100

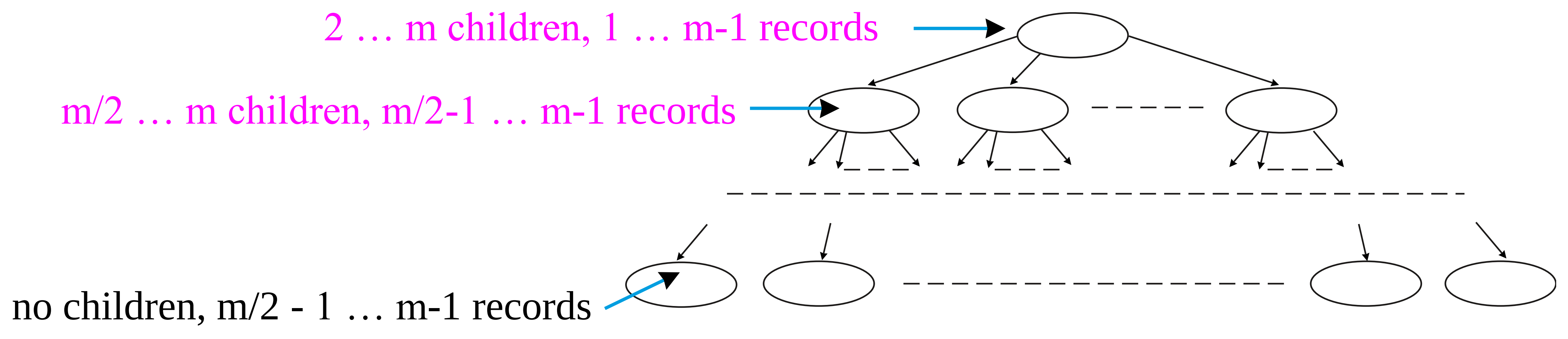


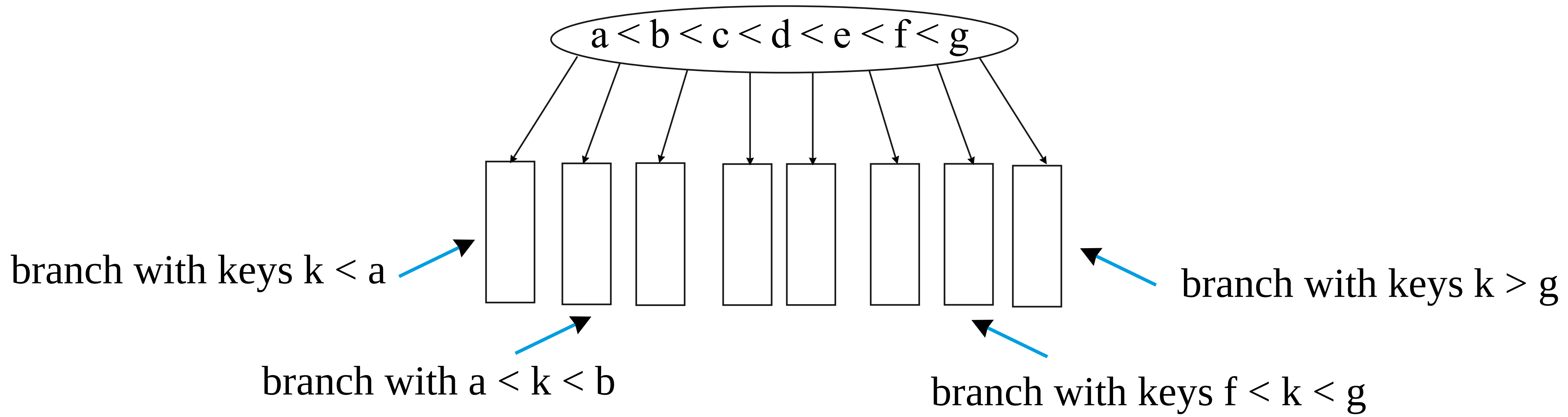
If the keys (integers) do not exceed 2^n, the height of trie (without leaves) cannot exceed n.

On searching you must move according to the bits but on the leaf the complete test is necessary.

Here we deal with tries for storing records in which the keys are integers. Tries for for storing records in which the keys are strings are out of scope of this course. Read https://www.geeksforgeeks.org/trie-insert-and-search.

B-tree (1)

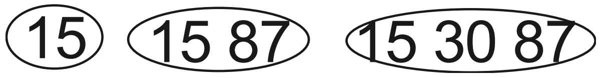

2 … m children, 1 … m-1 records
m/2 … m children, m/2-1 … m-1 records
no children, m/2 - 1 … m-1 records



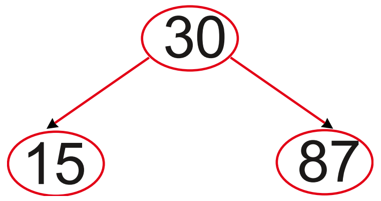
B-tree is always perfectly balanced. Records in a node are sorted. Mostly on disks. The size of node should be as large as the block on disk.

B-tree (2)

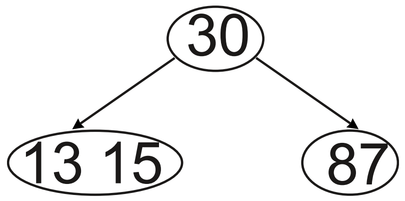
B-tree of order 4 is called as 2-3-4 tree and can be comfortably used in memory.
Let us have keys 15, 87, 30, 13, 72, 20, 39, 41, 60, 38, 32, 90, 10, 51, 67, 42.
First fill the root (keys are in sorted order):



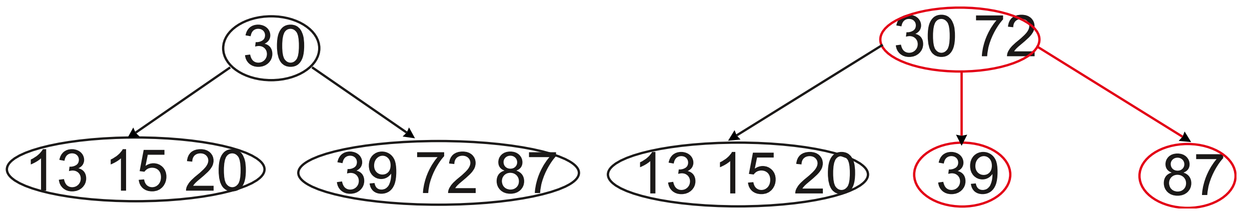
The root is full, split it:



New records must be inserted only into leaves:

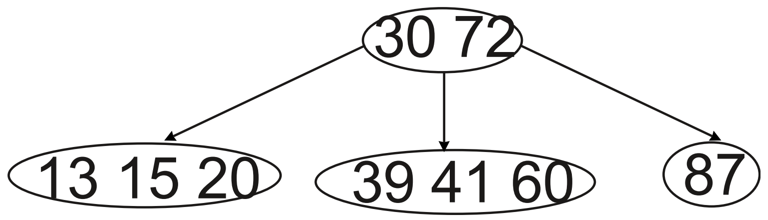


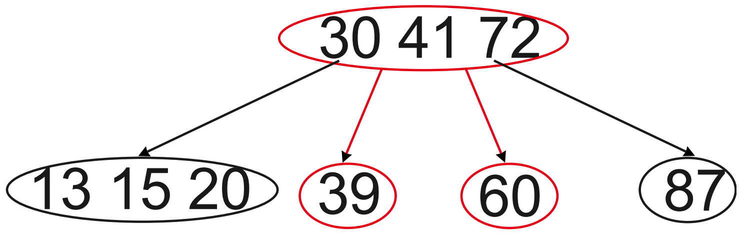
If the node into which you want to insert a new record is full, split it. The record in the middle moves into the parent node:

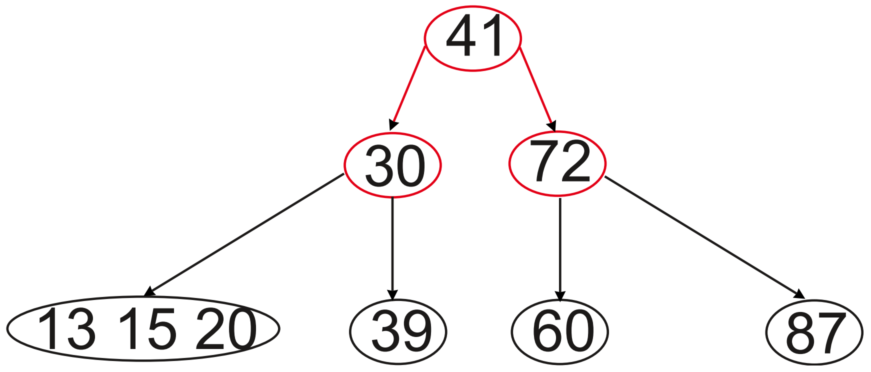


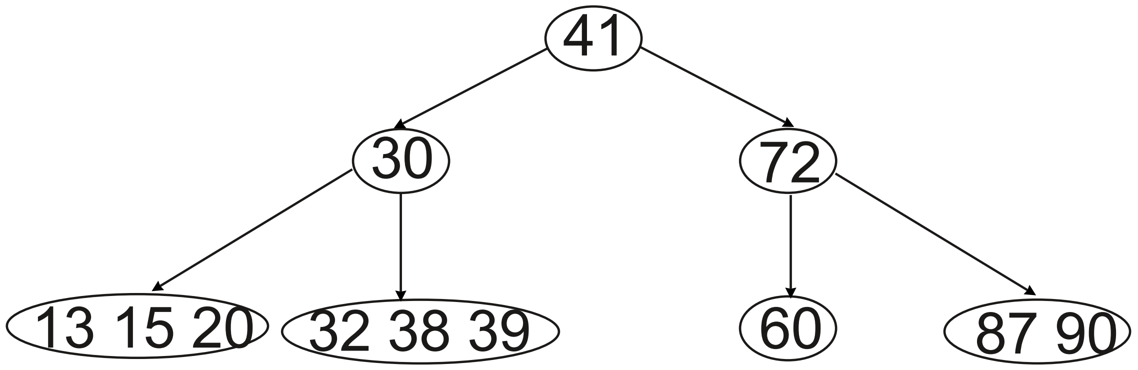
B-tree (3)

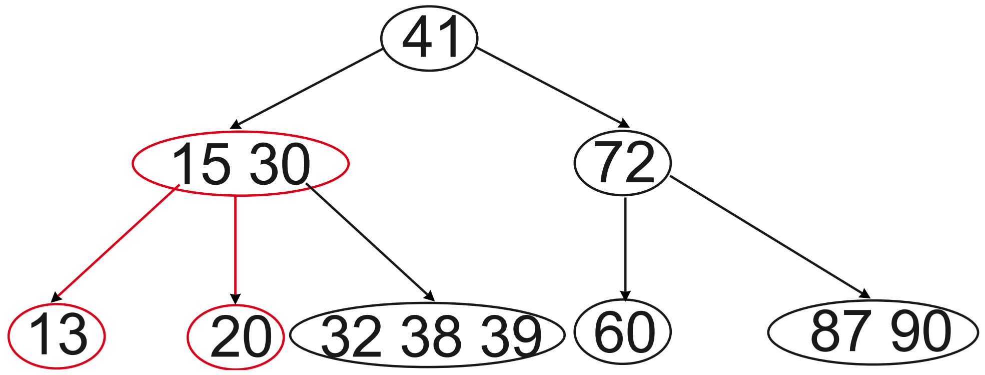
15, 87, 30, 13, 72, 20, 39, 41, 60, 38, 32, 90, 10, 51, 67, 42





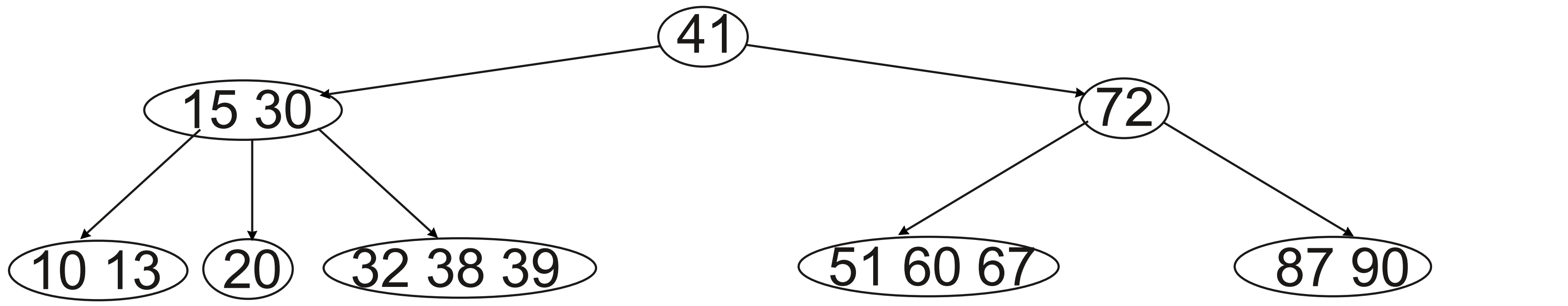


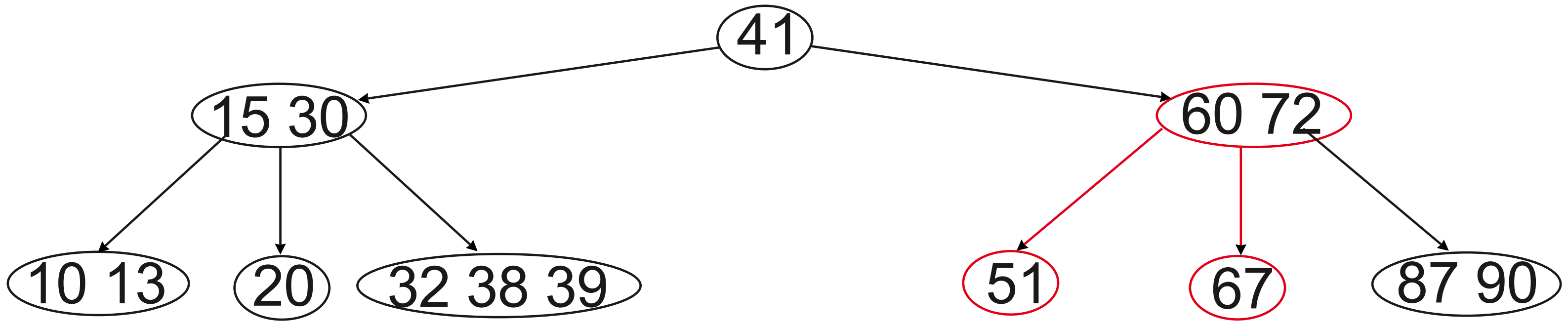


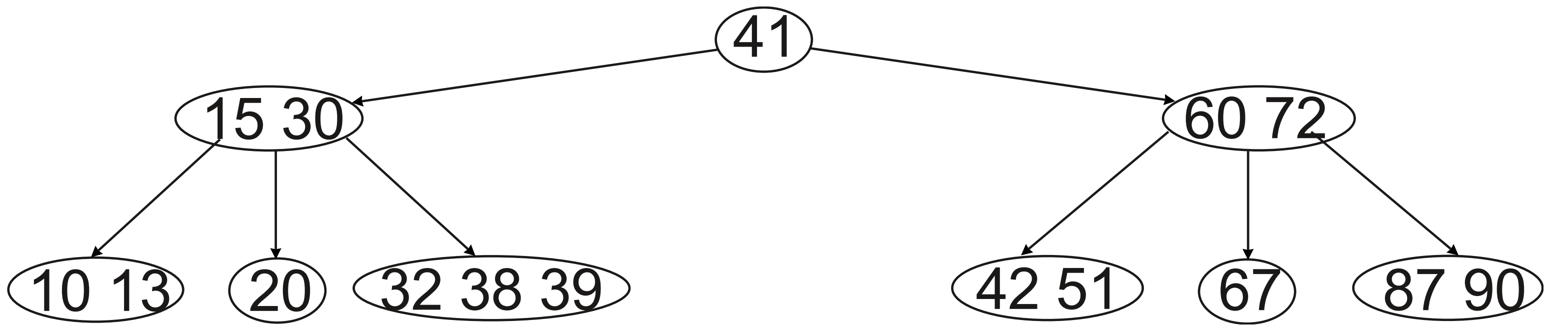


B-tree (4)

15, 87, 30, 13, 72, 20, 39, 41, 60, 38, 32, 90, 10, 51, 67, 42

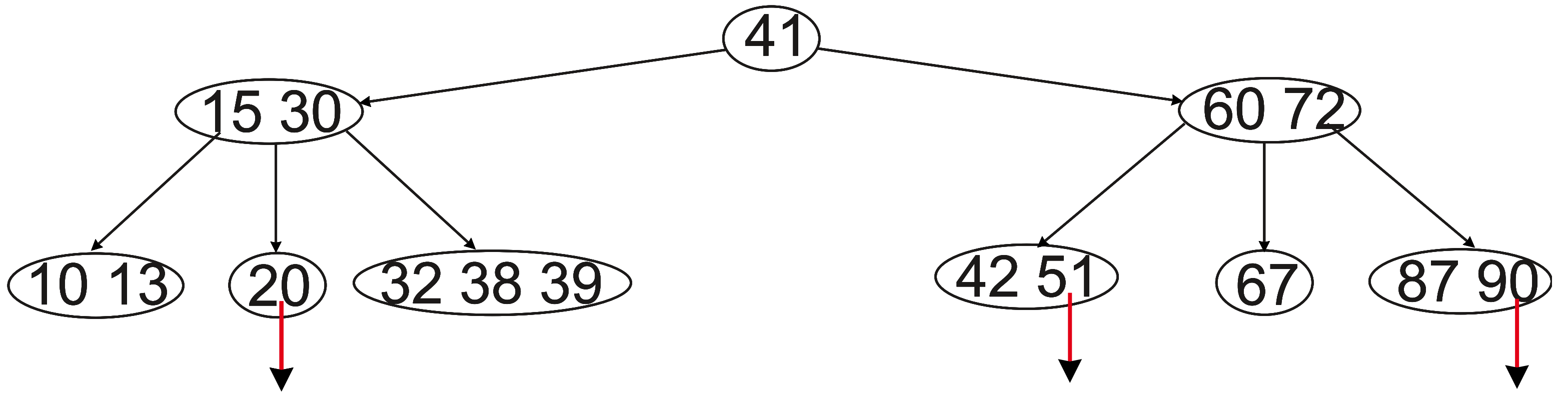




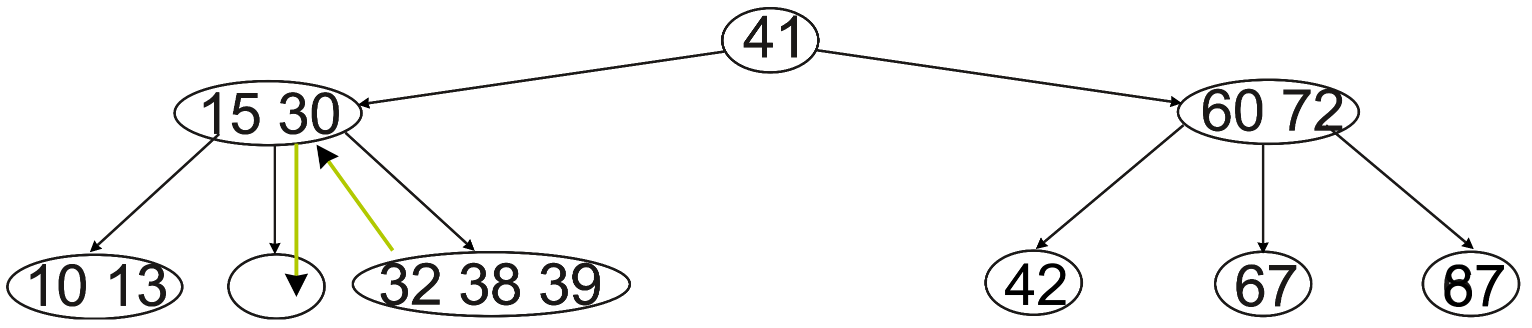


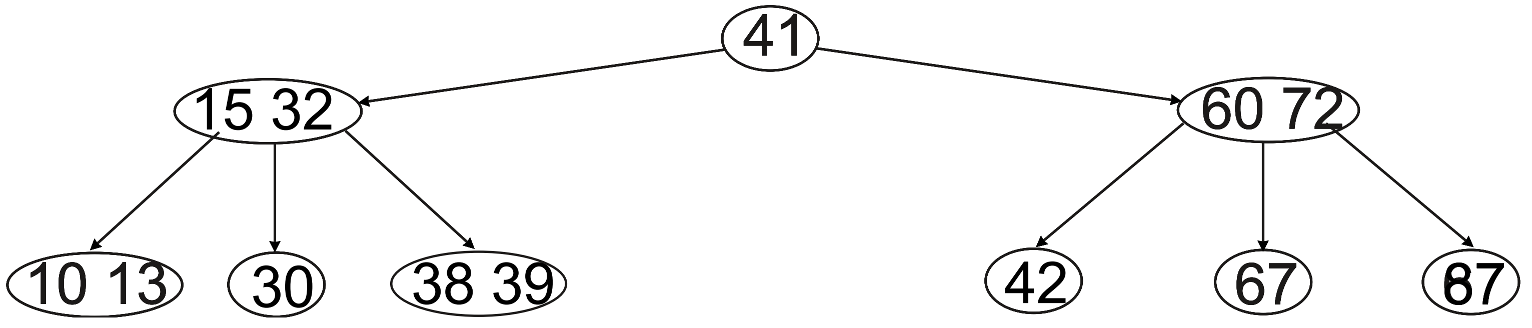
B-tree (5)

We can without problems remove records from leaves containing two or three objects. But empty nodes are not allowed.

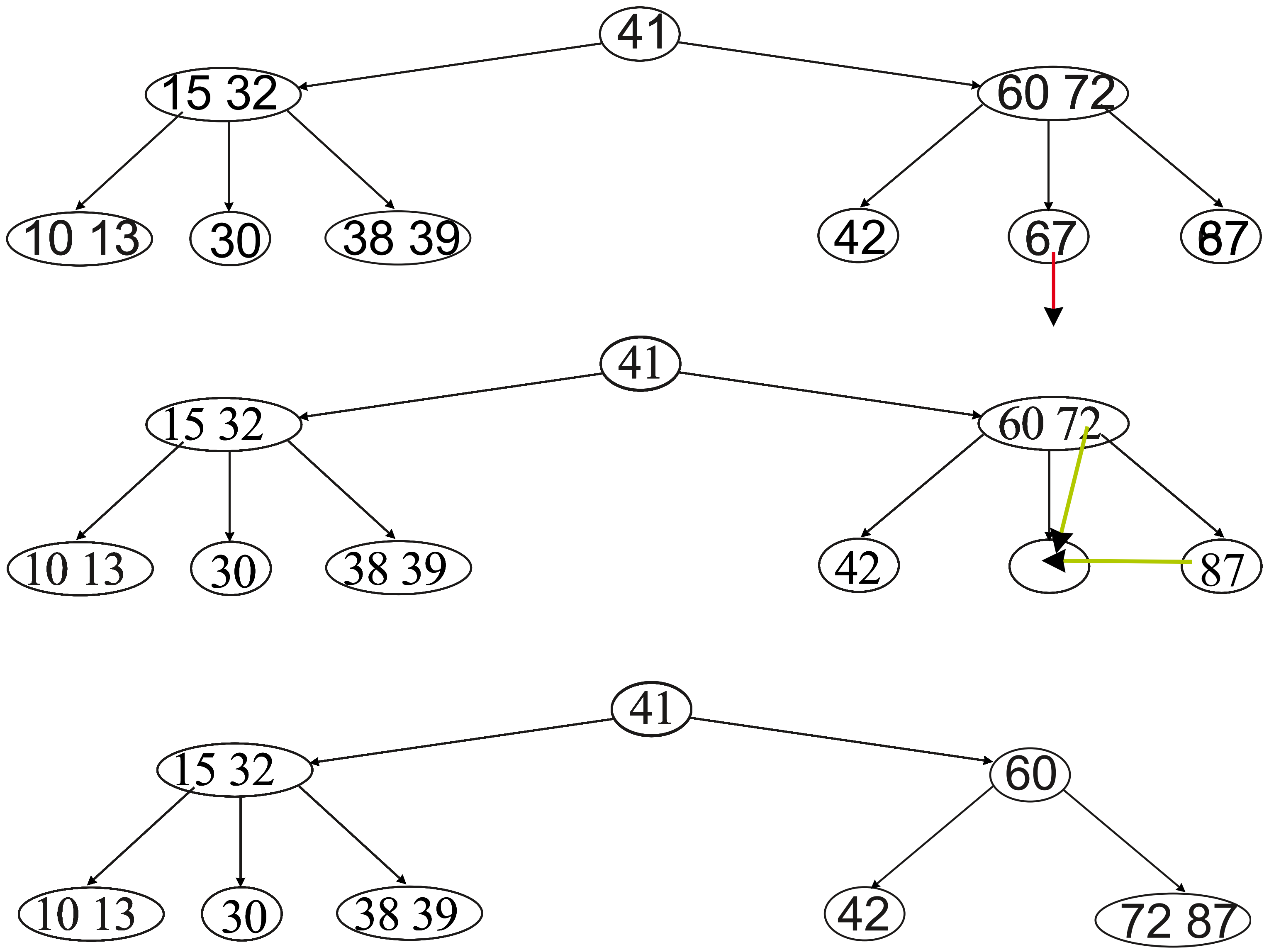


If the empty node has siblings containing two or three records, move the smallest from the right sibling (or biggest from the left sibling) to the parent. And move the record in the parent separating the empty node and the sibling that lost one of its records into the empty node.



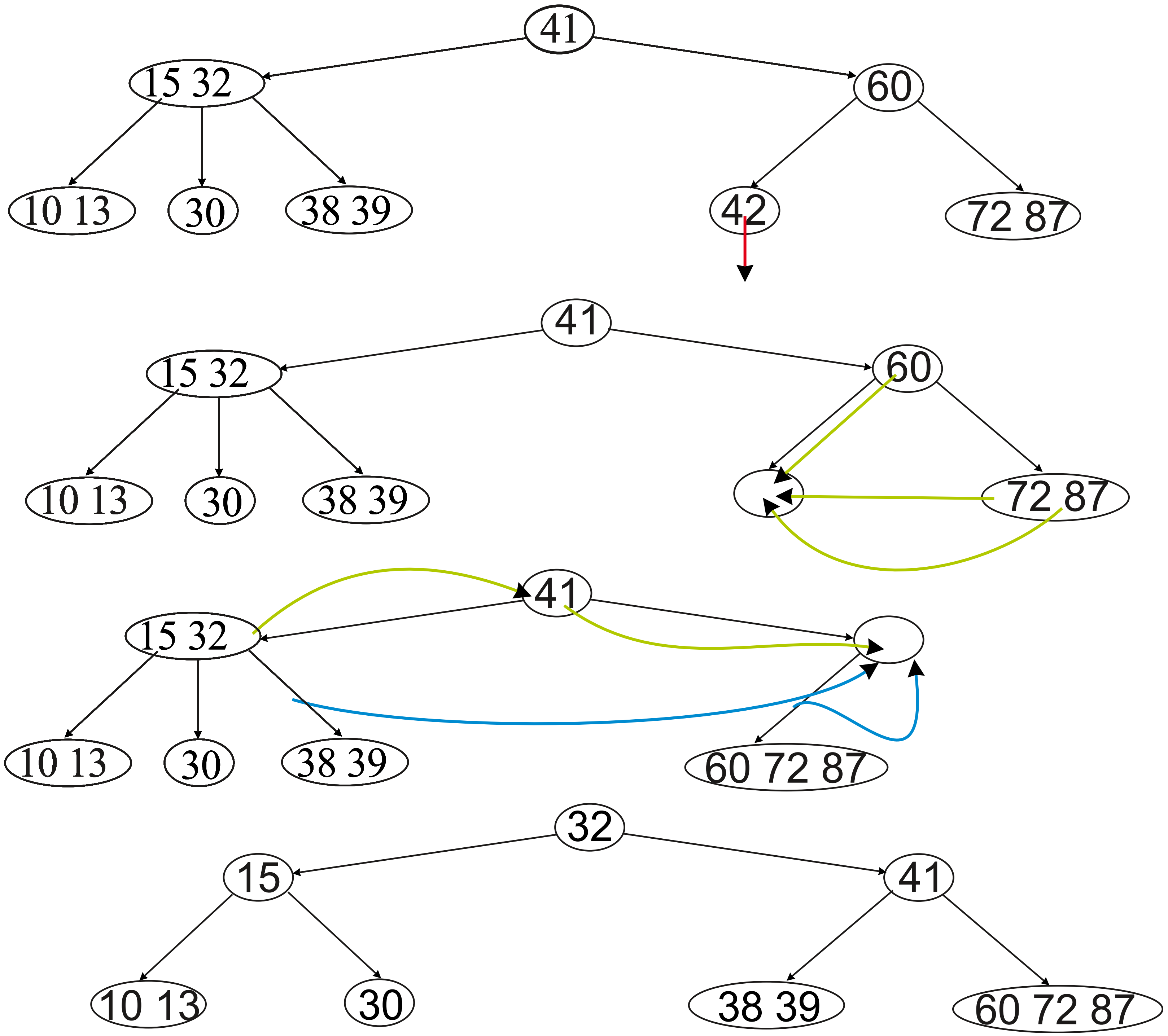


B-tree (6)



If the both esiblings of empty node contain only one record, the empty node must be removed. From its parent the biggest or the smallesr record moves down to the child.

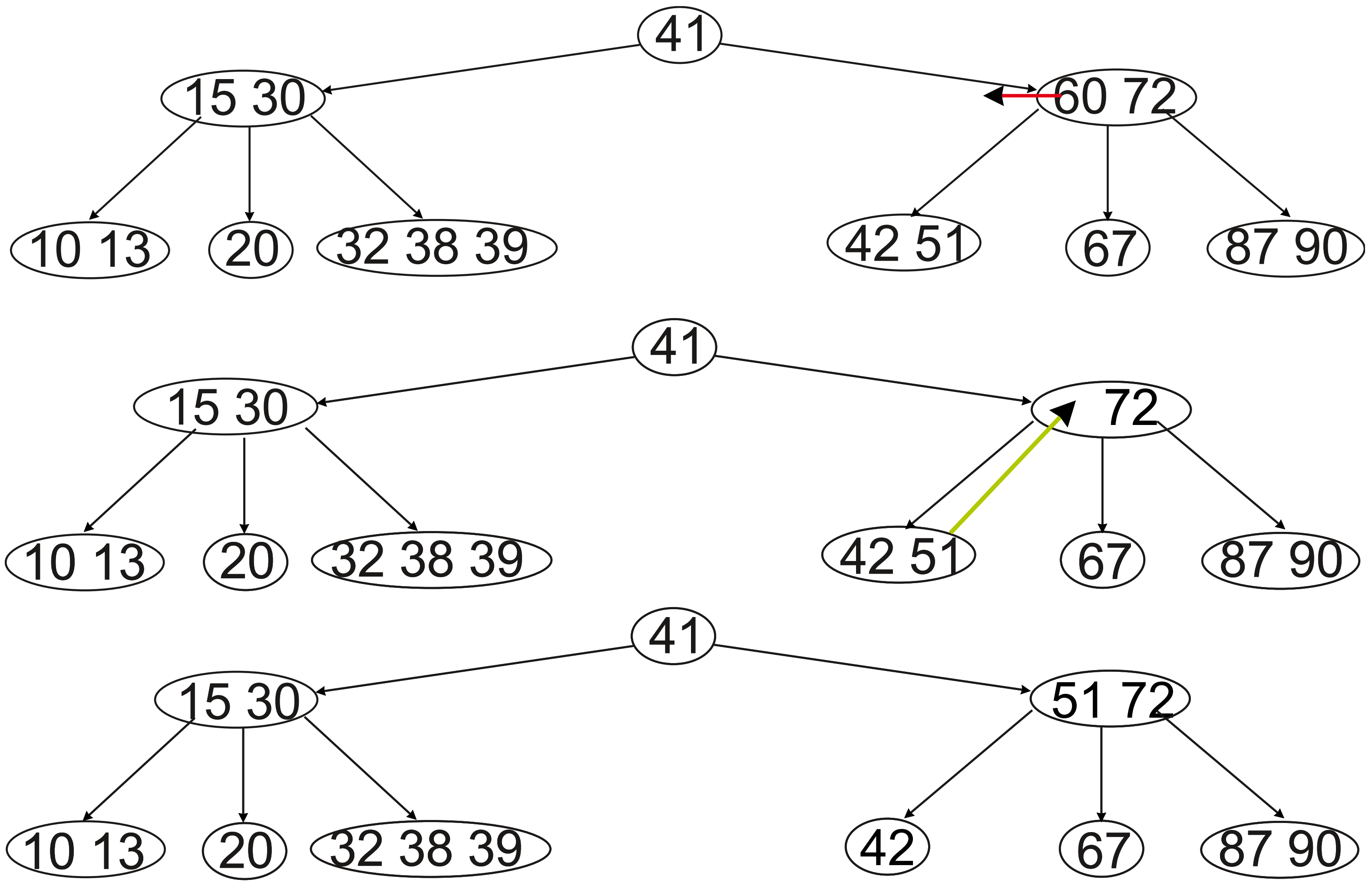
B-tree (7)



41
15 32
60
10 13
30
38 39
42
72 87
41
15 32
60
10 13
30
38 39
72 87
41
15 32
10 13
30
38 39
60 72 87
32
15
41
10 13
30
38 39
60 72 87

B-tree (8)

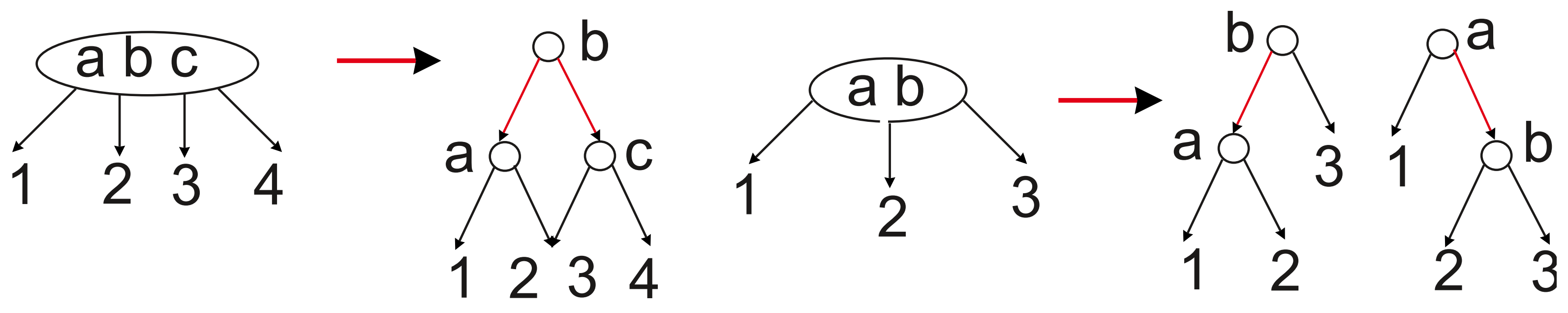
To remove from node that is not a leaf, find the record that in linear sorted list follows or precedes the record we want to remove. For that go down keeping all time to left or to right until you reach the leaf. Then move the found record to the place that has become empty. Actually you have provided the removing from leaf.

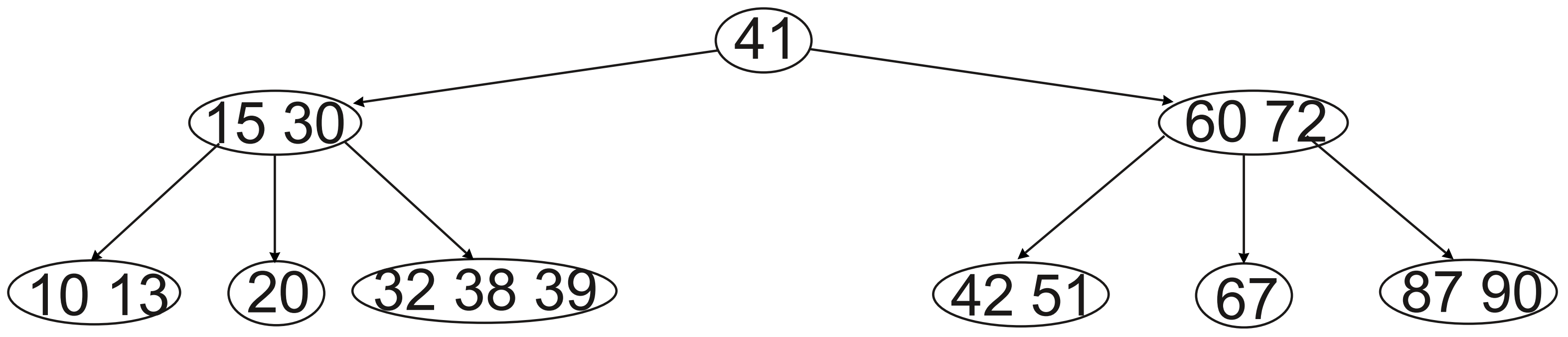


More about B-trees read:
https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
https://www.geeksforgeeks.org/b-tree-set-1-insert-2/
https://www.geeksforgeeks.org/b-tree-set-3delete/
Those pages present also the corresponding code for C/C++functions.

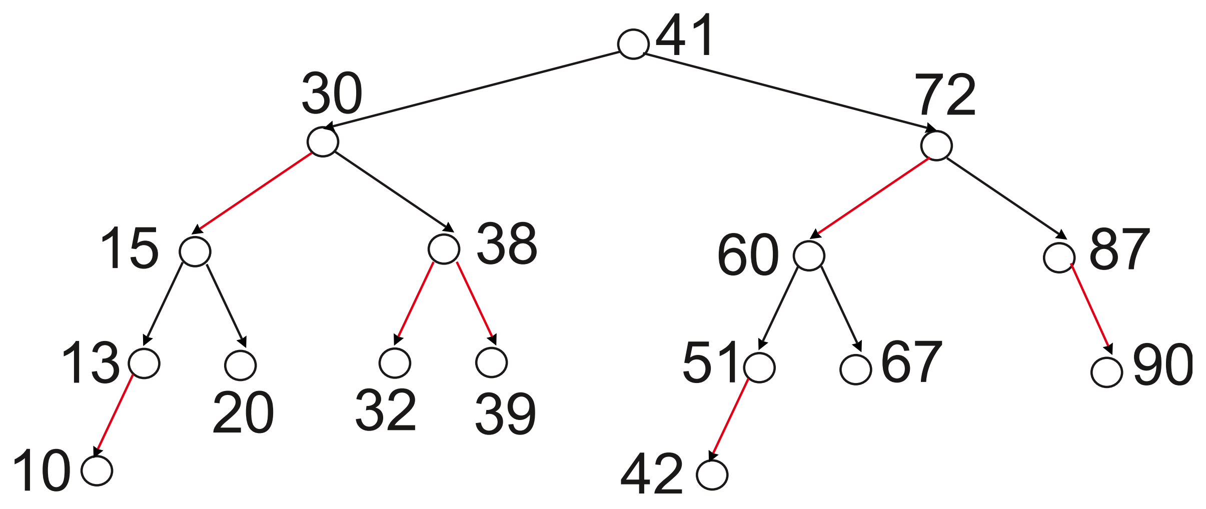
Red-black tree (1)

Nodes of 2-3-4 tree can be replaced with fragments of binary trees.





The result is well-balanced binary tree called as red-black tree.



Red-black tree (2)

In red-black tree:
$T(n) \leq 2*\log_2 n+2$.

If *n* is rather large:
$T(n) \approx 1.002*\log_2 n$

The red-black trees are as good as AVL trees.

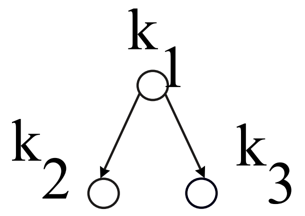
Of course, to build a red-black tree we must not first build the 2-3-4 tree. Algortihms for building and handling a red-black tree from scratch are on pages:
https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/
https://www.geeksforgeeks.org/red-black-tree-set-2-insert/
https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/
https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/
Those pages present also the corresponding code for C/C++functions.

Priority queue

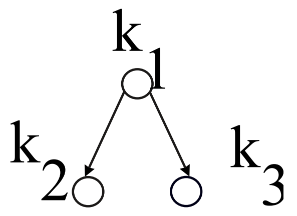
In simple queue (FIFO) we can remove only the first record. The new records are appended to the end of queue.

In priority queue (called also as heap) each record has its priority. Similarly to the keys, the priority is something we can retrieve from a record directly or after some calculations. Also, there must be algorithms to determine are the two priorities equal and if not, which of them is larger. The records in priority queue can be located randomly (i.e. without ordering), but only the record with highest priority can be removed.

Consequently, data structures implementing the priority queue must ensure that the highest priority record can be found as quickly as possible.

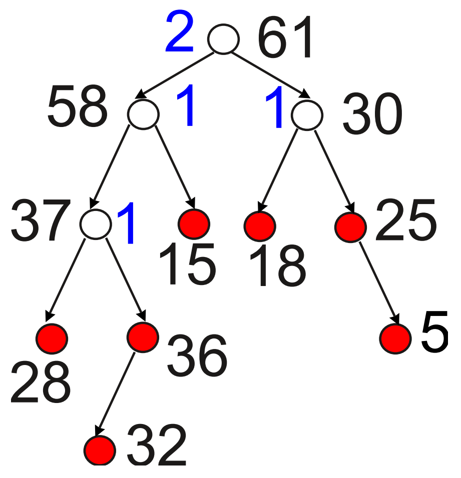


In ordered binary tree $k_2 < k_1$ and $k_3 > k_1$, consequently $k_3 > k_2$



In heap-ordered binary tree $k_1 > k_2$ and $k_1 > k_3$, relation between k_2 and k_3 may be any. Such trees are the most suitable for implementing priority queues. The record with highest priority is associated with the root node.

Leftist tree (1)



A leftist tree has two types of nodes:

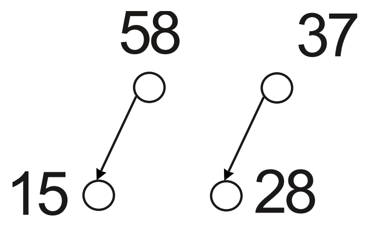
- the inner nodes have two children (here black).
- the outer node has one child or no children at all (here red).

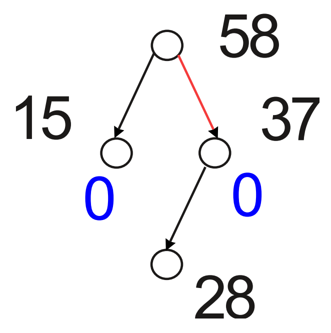
The rank of an inner node (here blue) is the length of shortest path from this node to an outer node. The rank of an outer node is zero.

The leftist tree is:

- heap-ordered.
- for each inner node the rank of left child is greater or equal with the rank of right child.

Let us have priorities 58, 15, 37, 28, 36, 32, 61, 25, 30, 18, 5.

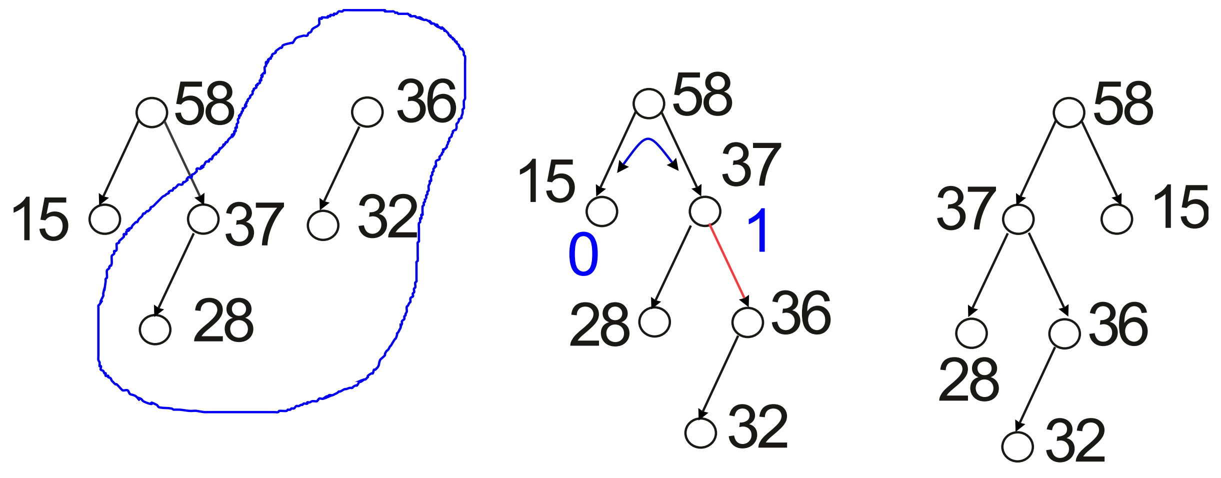




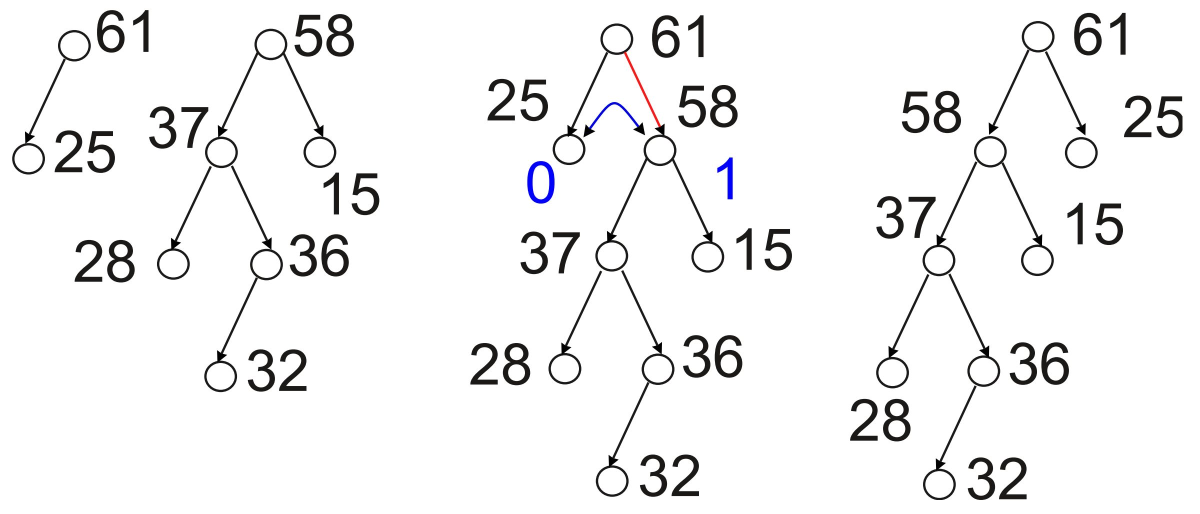
First create two subtrees. Let us call them A and B. The priority of root in A must be greater than the priority of root B. Then merge them, setting the B as the right branch of A.

Leftist tree (2)

58, 15, 37, 28, 36, 32, 61, 25, 30, 18, 5.



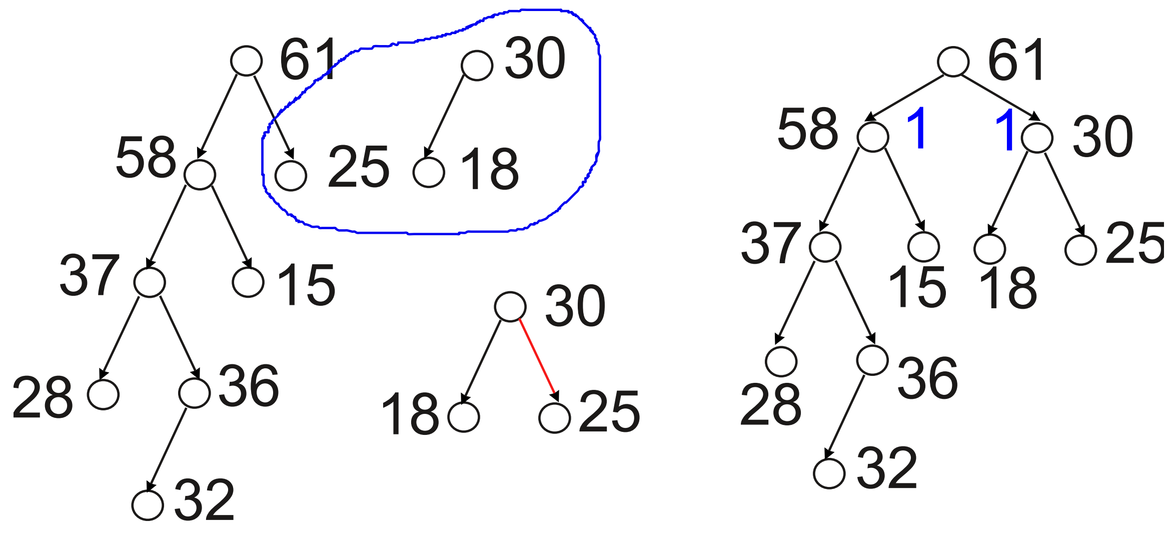
Here subtree A already has right branch. In that case we must merge the right branch with B (i.e. the A is now the right branch). After merging recalculate the ranks. As now the rank of left child is less than the rank of right child, exchange the branches.



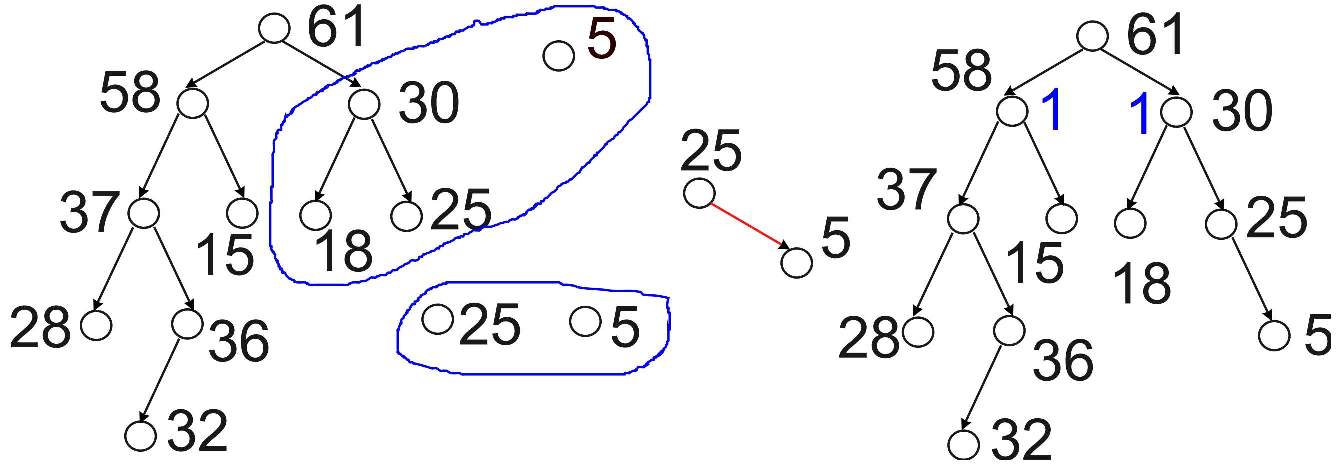
Now subtree with root 61 is the A and the complete B becomes its right branch. The rule of ranks is violated, so we need to exchange the branches.

Leftist tree (3)

58, 15, 37, 28, 36, 32, 61, 25, 30, 18, 5.

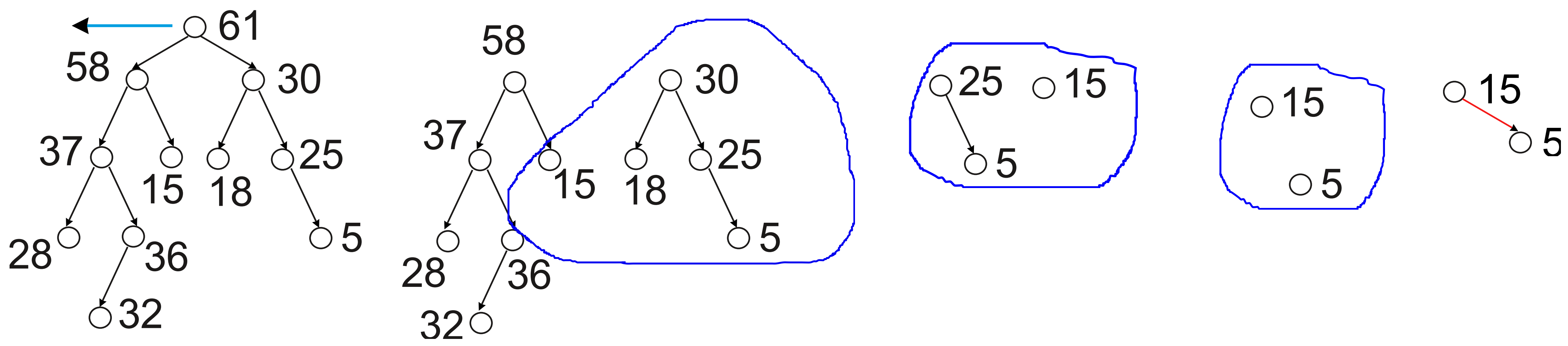


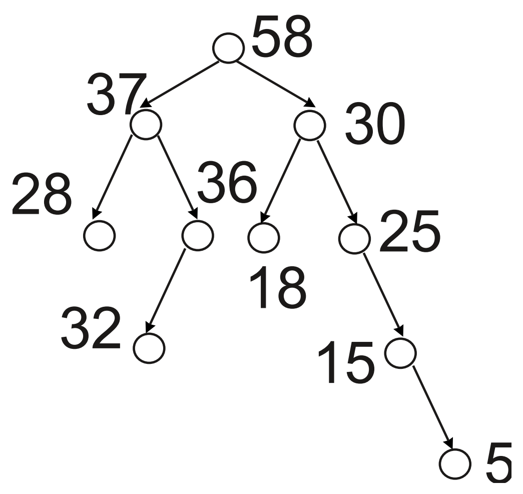
A already has right branch. We must merge B with the right branch of A, but as 30 > 25, B becomes to A and A is the single node 25.



Subtree B is the single node 5. The right branch of A has also its right branch: single node 25. So we need to merge two single nodes: 25 as A and 5 as B.

Leftist tree (4)





The node with highest priority is the root. After removing the tree breaks into two subtrees that we need to merge together.

More about leftist tree see on page:
https://www.geeksforgeeks.org/leftist-tree-leftist-heap/
This page presents also the corresponding code for C/C++functions.

Hashing (1)

The general idea of hashing is as follows:

1. Let us take an array with length m and fill it with zeroes (or some other objects signalling that all the positions in the array are empty). This is the hash table.
2. Next take a function h(k). Its arguments must be the keys of records, its output value is an integer from interval 0 … m-1. Principally, there are no other requirements on h(k) named as hash function.
3. To store a record into the table calculate the index – output of hash function.
4. To find a record from the table calculate the index and if the got position is not empty, retrieve the record.

However, there is a serious problem: hash function may produce non-unique output and therefore several records with different keys may claim the same location. This is the collision. Consequently, after retrieving the record we must carry on an additional testing.

The main problems of hashing are:

1. How to select the length of table and hash function so that the probability of a collision is minimal.
2. What to do if the collision has occurred.

The weak point in hashing is that we need to estimate the possible number of records. If the table is too short, we need also to select a new hash function, i.e. start all the work from scratch.

Hashing (2)

```
int k;  // key
int m; // length of table
int h = k % m;  // the simplest hash function (hashing by division method)
```

Here $m = 2^n$ is a very bad idea. Example:

Let n = 10.

k_1 = 2837 = 0xB15 = 1011 0001 0101

k_1 % m = 789 = 0x315 = 0011 0001 0101

k_2 = 1813 = 0x715 = 0111 0001 0101

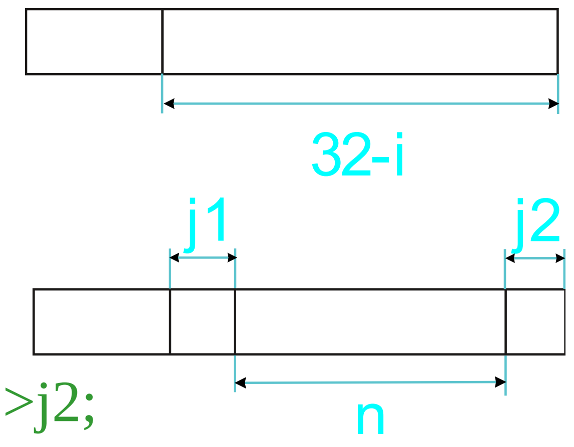
k_2 % m = 789 = 0x315 = 0011 0001 0101 – collision!

Reason: if we delete with 2^n, the remainder is the last n bits of key. The higher bits are not used.

Recommended: the length of table should be a prime number exceeding the supposed number of records. About prime numbers see https://www.mathsisfun.com/prime_numbers.html.

Hashing (3)

```
int k;  // key
int m; // length of table, must be 2^n
int h; // result
int i, j1, j2; // auxiliary values
unsiged int kk = k * k, bit = 0x80000000; // auxiliary values
unsigned int mask; // in this constant the lower n bits are ones, the others are 0, for example
                   // if n = 10, the mask is 0x3FF = 0000 0000 0000 0000 0000 0011 1111 1111
for (i = 0;  !(bit & kk);  i++, bit >>1); // find the highest bit that is not zero
```



```
j1 = (32 –i–n) / 2;
j2 = 32 – i –n –j1;

h = ((mask<<j2) & kk) >>j2;
```

Let k = 2837, kk = 8048569 = 0x7ACFB9, n = 8.

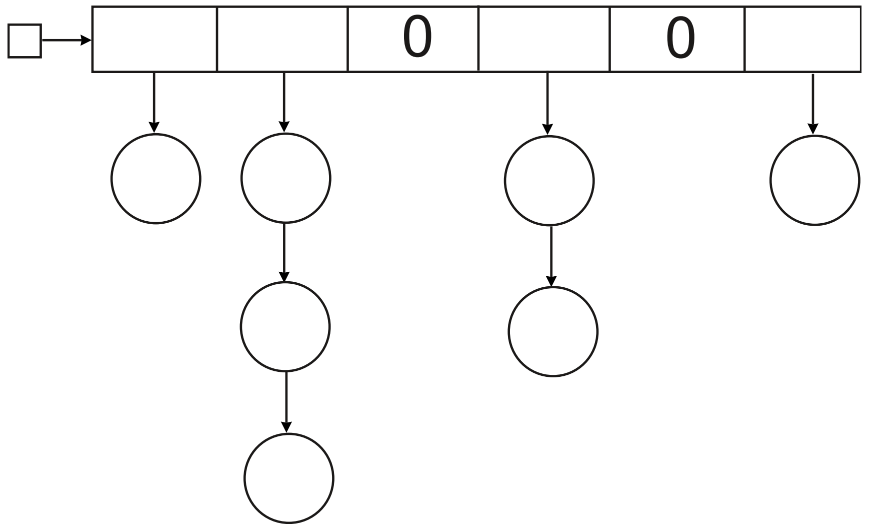
```
0000 0000 0111 1010 1100 1111 1011 1001 // kk, cut out the n middle bits
0000 0000 0000 0000 1111 1111 0000 0000 // mask<<j2
0000 0000 0000 0000 1100 1111 0000 0000 // (mask<<j2) & kk
0000 0000 0000 0000 0000 0000 1100 1111 // h = 207
```

This is hashing by middle square method.

Hashing (4)

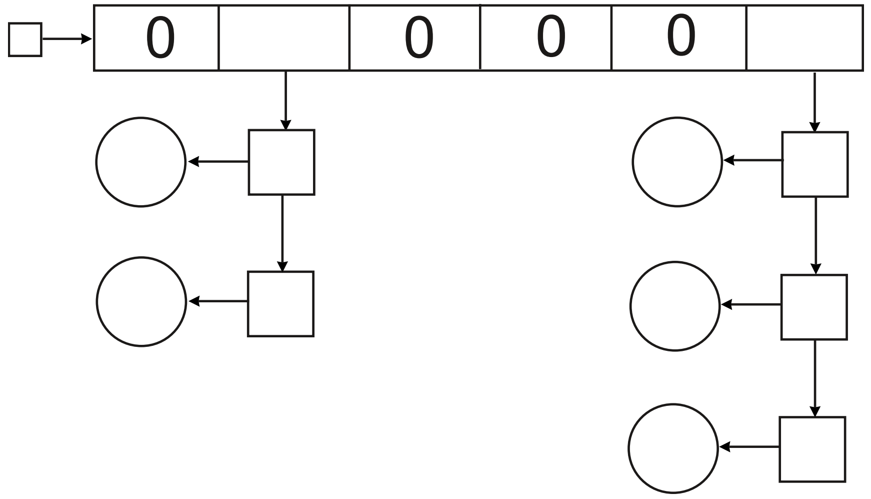
Collided records are inserted into chains. Advantage: the length of table is not critically important. Example:

```
int hash_fun(char *pKey)
{  // key is a string, lenght of table is 26
      return *pKey – 'A';
 }
```



If the records do not have *pNext* pointers, introduce headers:

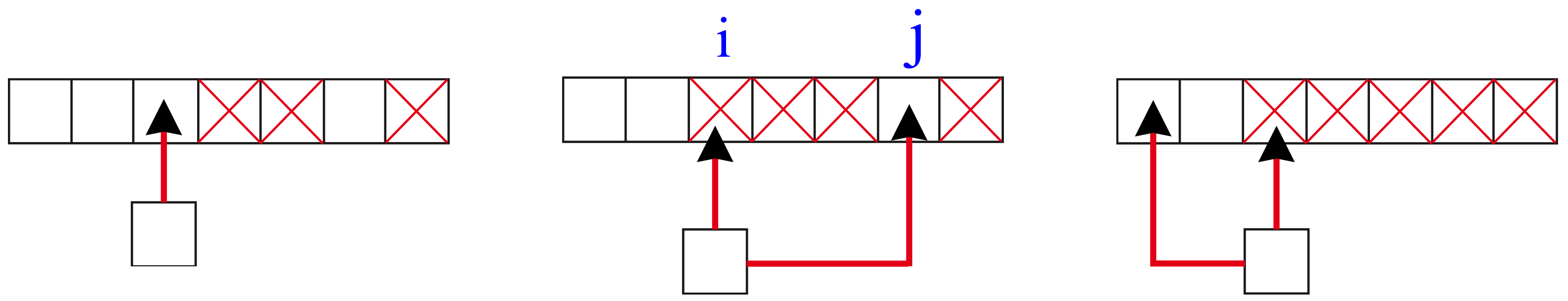
```
struct Header
{
     void *pRecord;
     Header *pNext;
};
```



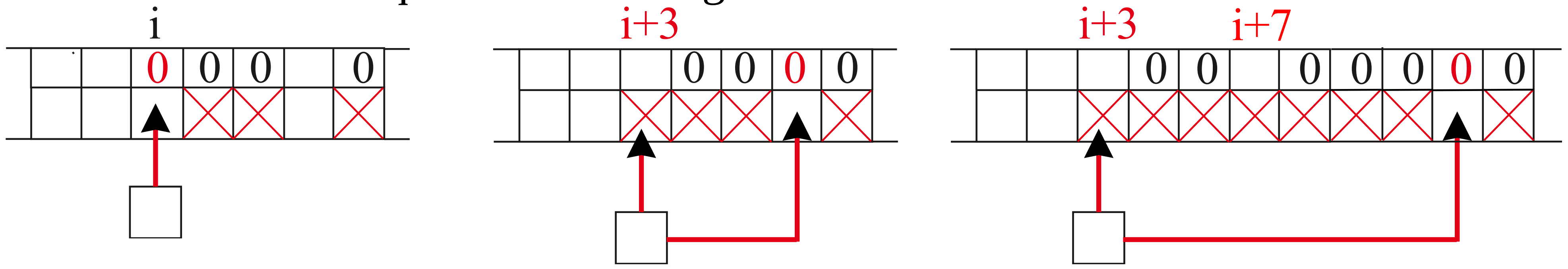
Resolution of collisions with cellar: the cellar is just a linked list. Insert the record pretending to cell that is already occupied into cellar. Searching: if the cell is empty, the record is missing. If not, carry on the complete testing of keys. If the result is positive, you have found your record. If negative, the record may be in the cellar: apply sequential search.

Hashing (5)

Open addressing: if a cell is already occupied, apply some rules to find an empty cell. The simplest algorithm of open addressing is the linear probing: just move step by step until the first empty cell. If the end of table is reached, jump to the beginning and continue.



Suppose that $h(k_1) = i$ and $h(k_2) = i$. Due to collision the record with key k_2 is inserted into cell j. Suppose now that $h(k_3) = j$. As cell j is occupied, we have also to find a new cell for record with key k_3. Theoretically it may happen that only the first record is on its right place. Mostly, once confusion has arisen, it grows rapidly and searching from hash table becomes similar to sequential searching.

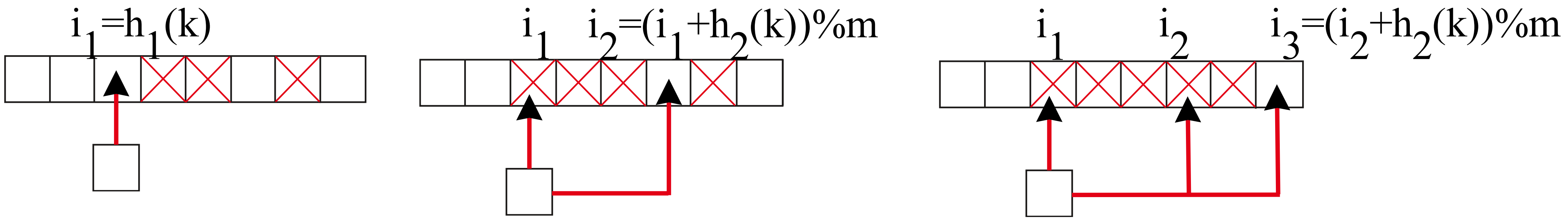


In coalesced hashing each cell has additional field for index. If a cell is inserted into not relevant cell, the index shows its location. Due to it the searching is faster.

Hashing (6)

In double hashing two hash functions are applied:

```
int k; // key
int m; // length of the table
int h1, h2; // hash indeces
h1 = hash1(k);
h2= hash2(k);
while (!empty(h1)) h1 = (h1 + h2) % m;
```



$h_2(k)$ can never return zero and $h_2(k) \% m$ can never be zero, otherwise we get an endless loop. This is because:

$(a + b) \% c = (a \% c + b \% c) \% c$ and if $a < c$ then $a \% c = a$.

Recommended:

$h_2(k) = k \% (m - 1) + 1$

$h_2(k) = k \% (m - 2) + 1$ (if m is a prime number)