Searching and simple dictionary

Data Structures and Algorithms

Searching and simple dictionary

Searching and simple dictionary

- linear searching
- binary searching
- binary search tree (BST)
- hashing

Linear searching

- unordered collection
- linear access, random access

```
// return position or -1 if not found
int search(int arr[], int n, int value)
{
  for(int i=0;i<n;i++)
    if(arr[i]==value)
      return i;
  return -1;
}
```

Linear searching

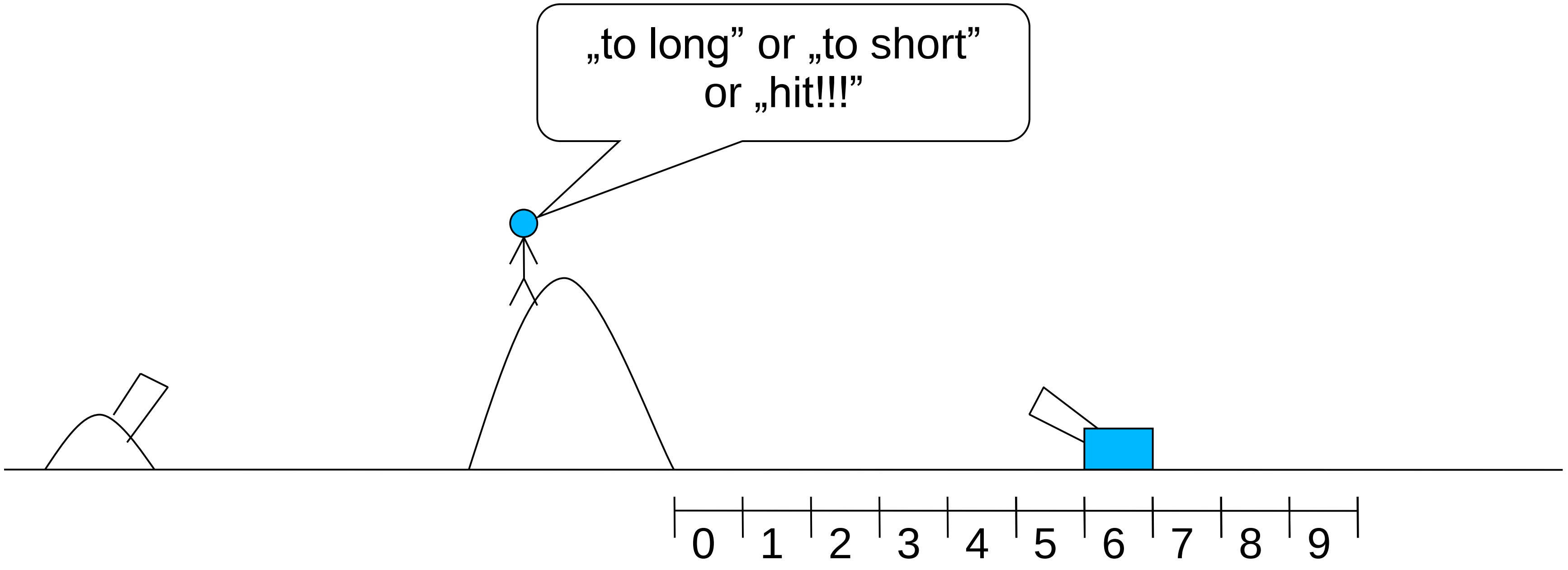
```
// straight linked-list
ElementTyp* search(ElementTyp *head, int value)
{
  ElementTyp *p=head;
  while(p!=null && p->value!=value)
    p=p->next;
  return p;
}
```

```
static Integer search(Collection<Integer> collection, Integer value)
{
  for(Integer elem:collection) // e.a. LinkedList
    if(elem.equals(value))
      return elem;
  return null;
}
```

- worse-case complexity: O(n)
- average-case complexity: O(n)

Binary searching

- ordered collection
- random access



Binary search

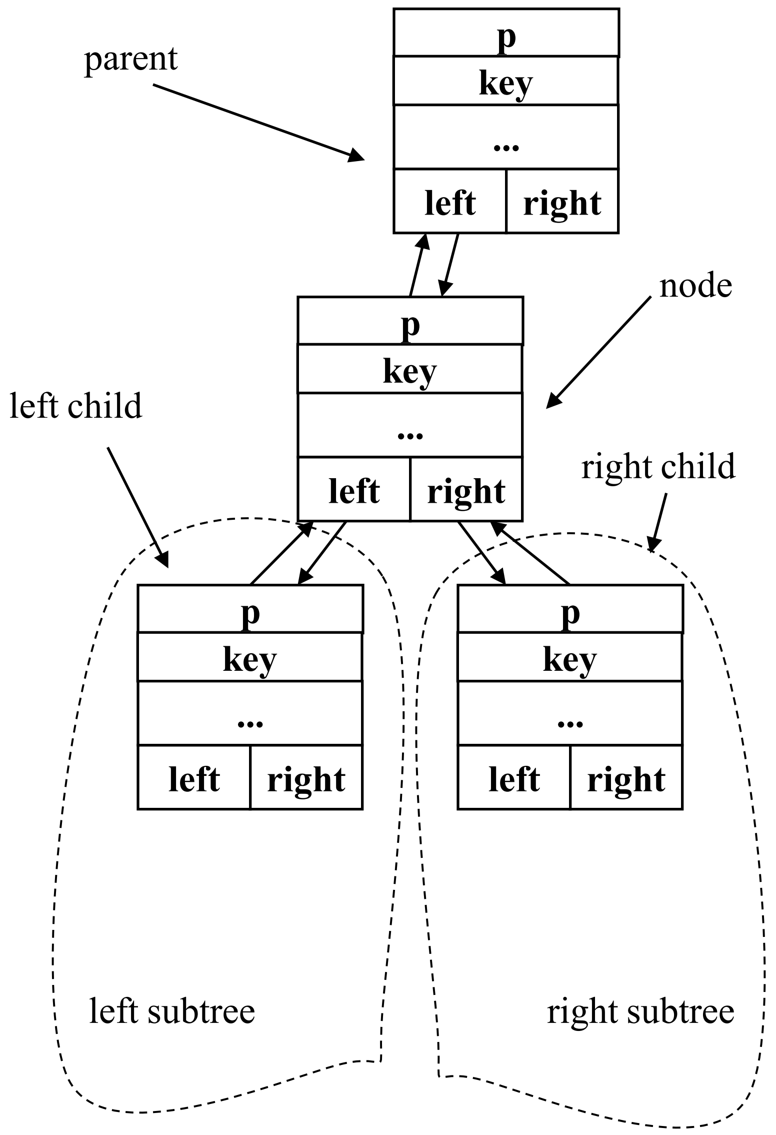
```
// return position or -1 if not found
int binSearch(int arr[], int n, int value)
{
  int left=0,right=n-1;
  int middle;
  while(left<=right)
  {
    middle=(left+right)/2;
    if(arr[middle]==value)
      return middle;
    if(value<arr[middle])
      right=middle-1;
    else
      left=middle+1;
  }
  return -1;
}
```

- worse-case complexity: O(log*n*)
- average-case complexity: O(log*n*)

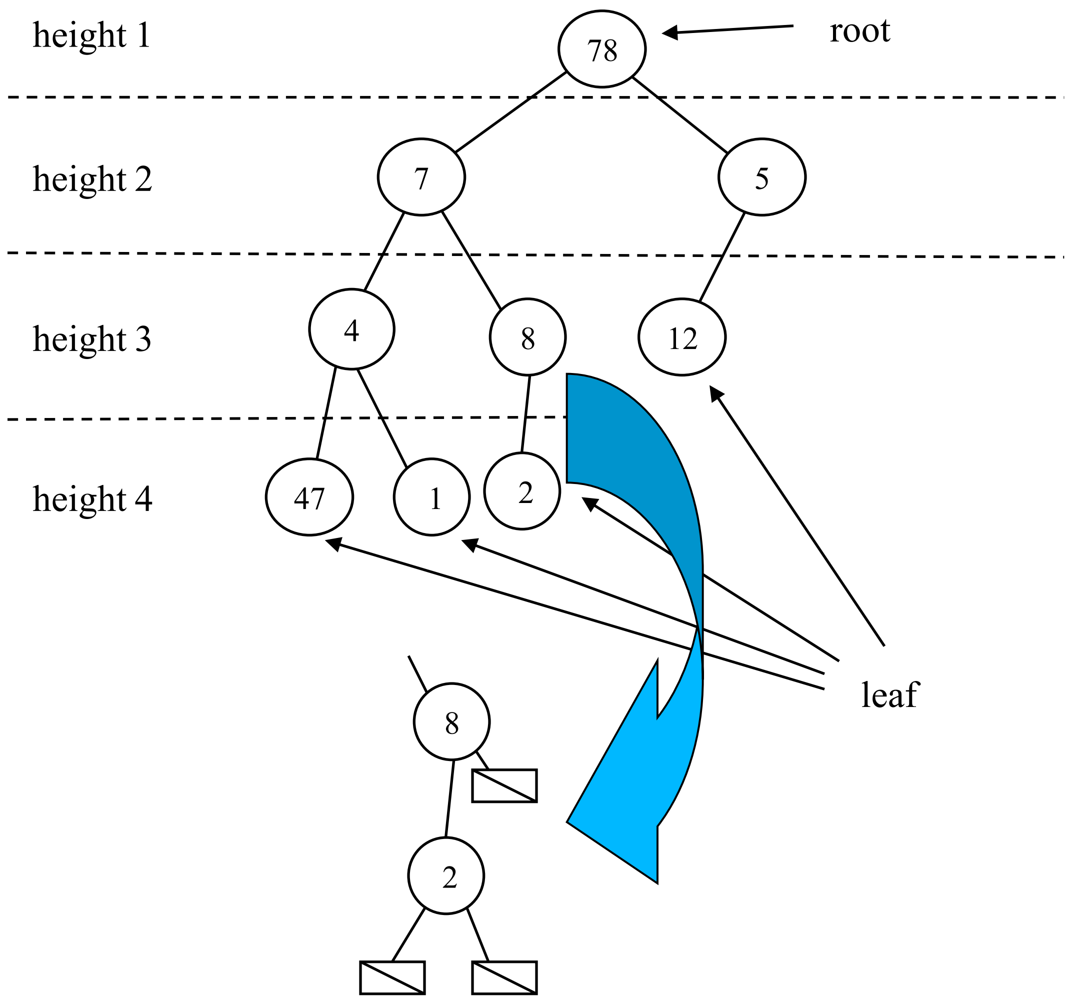
Binary tree

- Binary tree is a tree in which each node has at most two under-nodes (children)
- Binary tree can be represented by a linked data structure in which each node is an object. In addition to a ***key*** field and satellite data, each node contains fields ***left***, ***right***, and ***p*** that point to the nodes corresponding to its *left child*, its *right child*, and its *parent*, respectively.
- If a child or the parent is missing, the appropriate field contains the value **null**. The **root node** is the only node in the tree whose parent field is **null**.

Binary tree



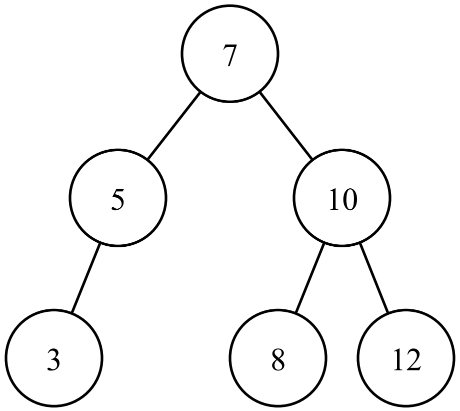
A simplified representation of binary tree:

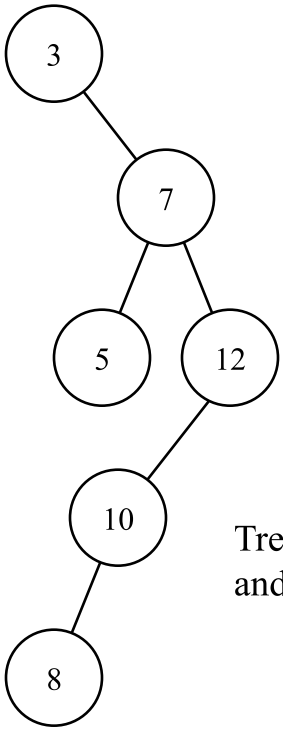


Binary search tree - definition

Binary-search-tree (BST) property:

Let *x* be a node in a binary search tree. If *y* is a node in the left subtree of *x*, then *key*[*y*] ≤ *key*[*x*]. If *y* is a node in the right subtree of *x*, then *key*[*y*] ≥ *key*[x].





Tree with $n = 6$ nodes
and height $h = 3$

$\boxed{\log n <= h <= n}$

Tree with $n = 6$ nodes
and height $h = 5$

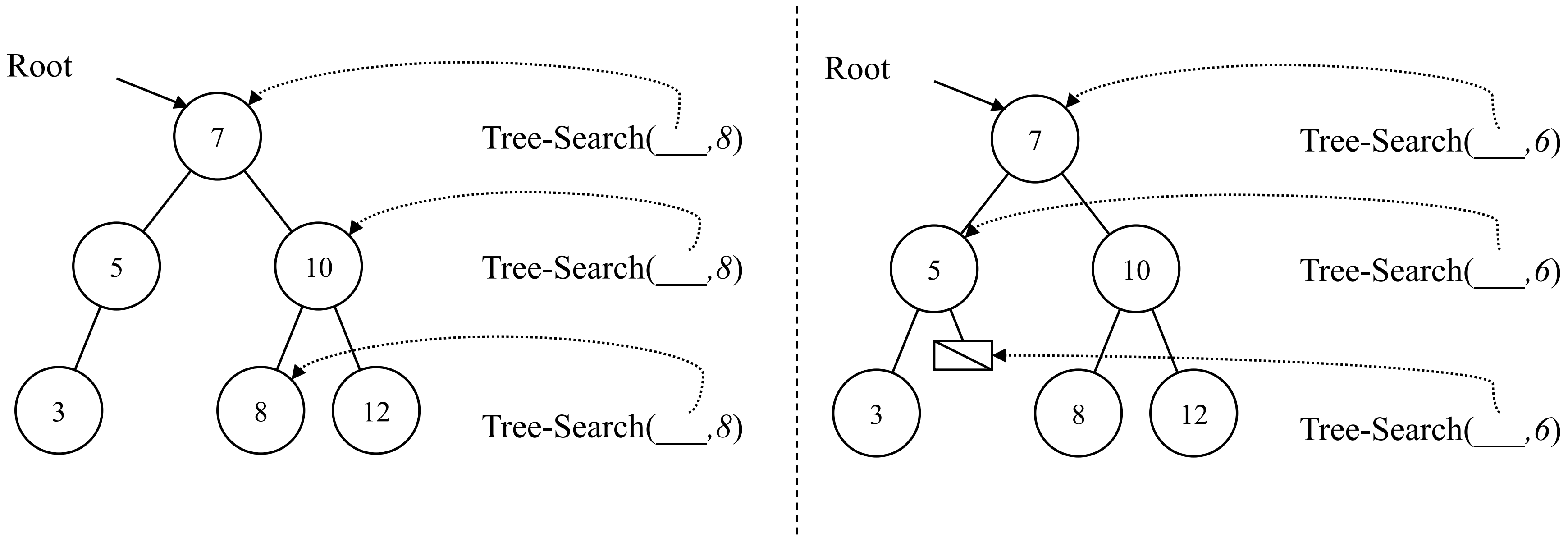
$\boxed{\text{average height } h = O(\log n)}$

Searching in BST

- Searching for any value, we choose proper subtree using the BST property

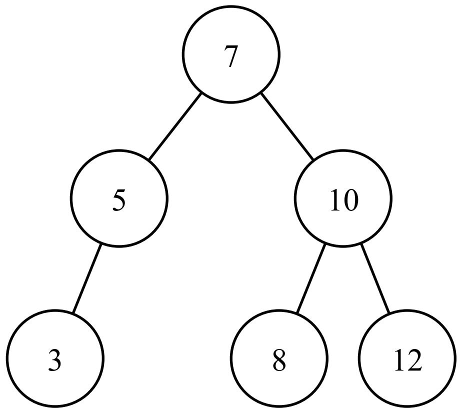
```
     Tree-Search(x,k)
{1}  if (x = null) or (k = key[x])
{2}      then return x
{3}  if k<key[x]
{4}      then return Tree-Search(left[x],k)
{5}      else return Tree-Search(right[x],k)
```

Complexity: O(h)



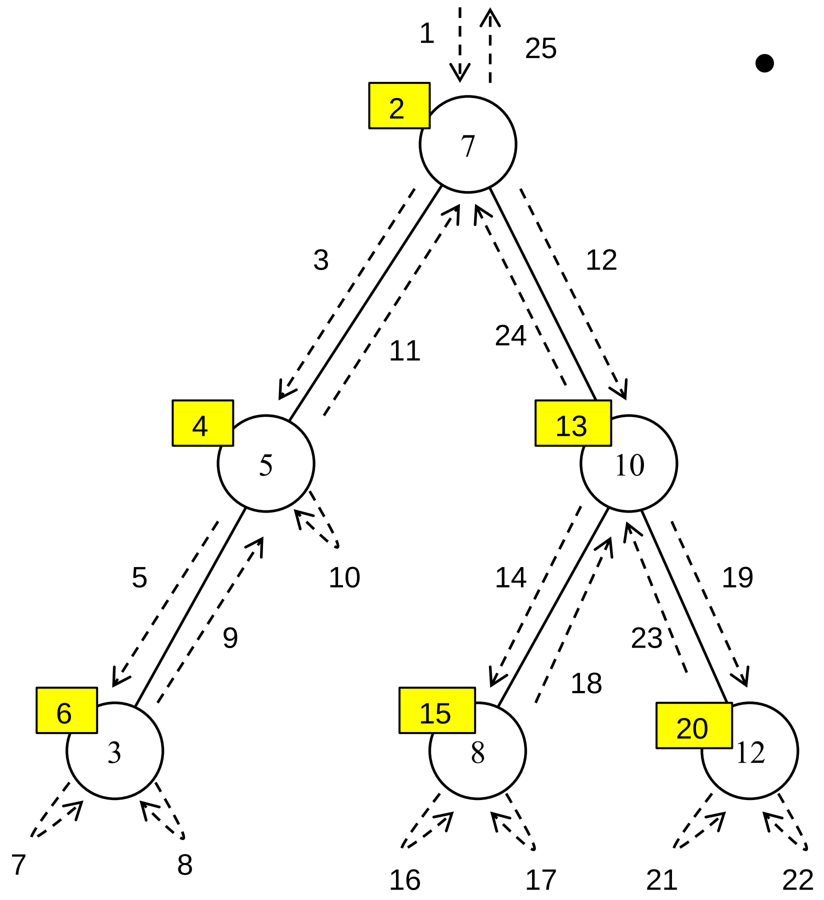
Tree walk in BST

- Tree-walk consist in visitting all the nodes exactly once.
- The visit (a.e. a print) can be done:
  - before walking into subtrees (preorder-walk)
  - after walking into subtrees (postorder-walk)
  - between walking into subtrees (inorder-walk).



Preorder-Walk:	7,5,3,10,8,12
Postorder-Walk:	3,5,8,12,10,7
Inorder-Walk:	3,5,7,8,10,12

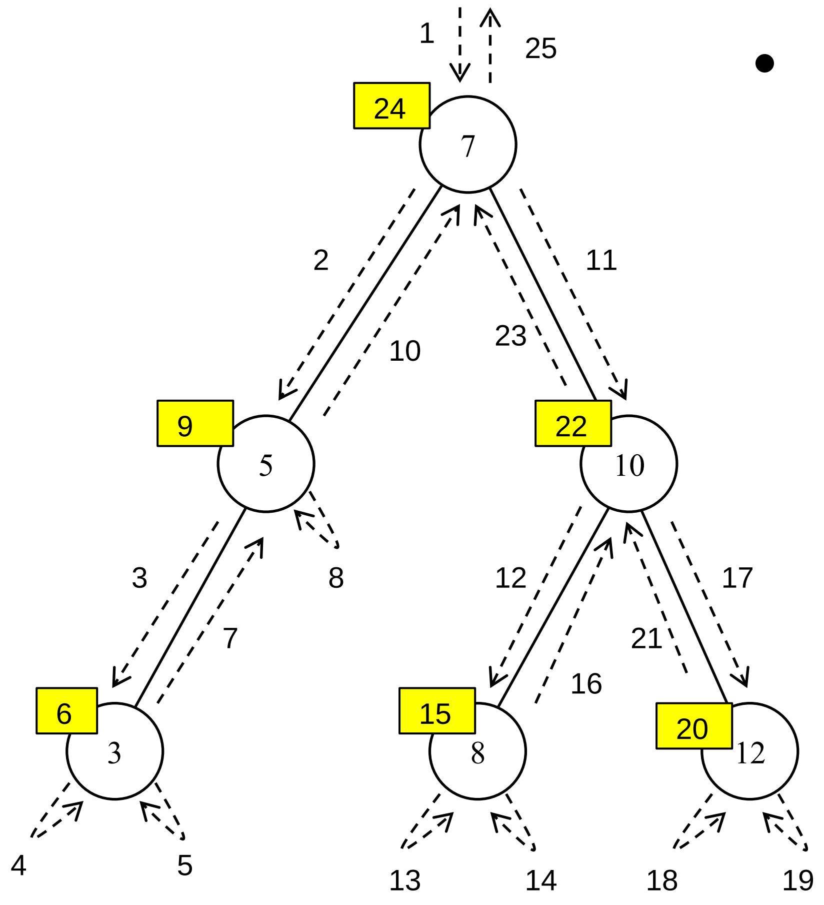
Preorder-walk



- The visit can be done before walking into subtrees

Preorder-Walk: 7,5,3,10,8,12

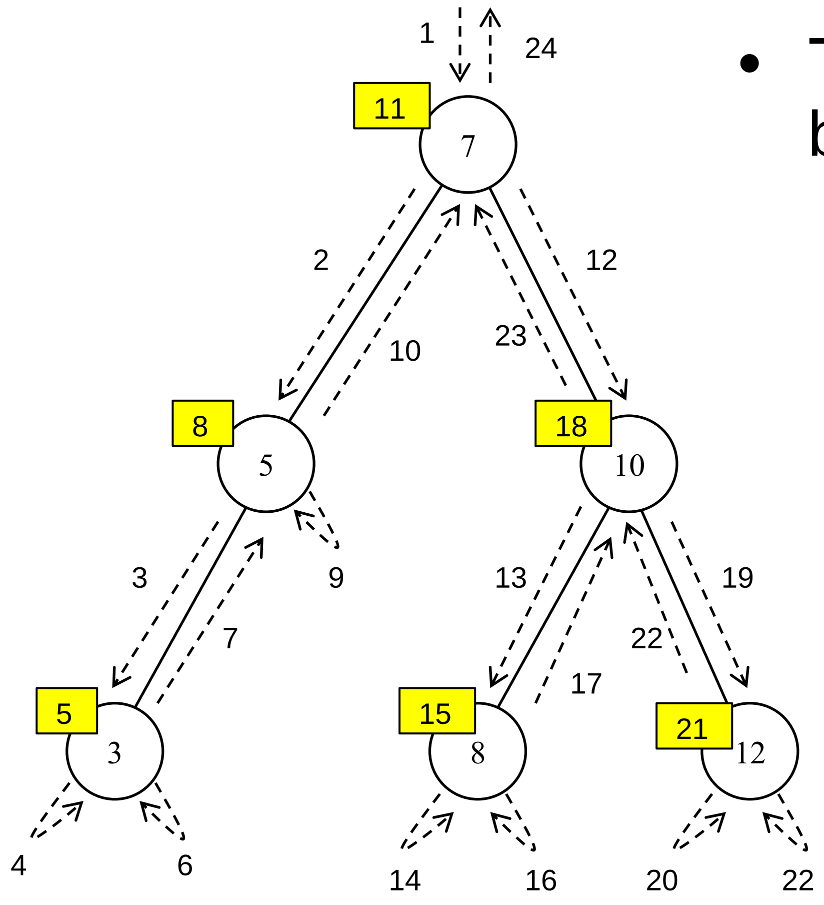
Postorder-walk



- The visit can be done after walking into subtrees

Postorder-Walk: 3,5,8,12,10,7

Inorder-walk



- The visit can be done between walking into subtrees

Inorder-Walk: 3,5,7,8,10,12

Inorder-walk

```
    Tree-Inorder-Walk(x)
{1} if x <> null then
{2}   Tree-Inorder-Walk(left[x])
{3}   show key[x]
{4}   Tree-Inorder-Walk(right[x])
```

- Call: Tree-Inorder-Walk(*root*)

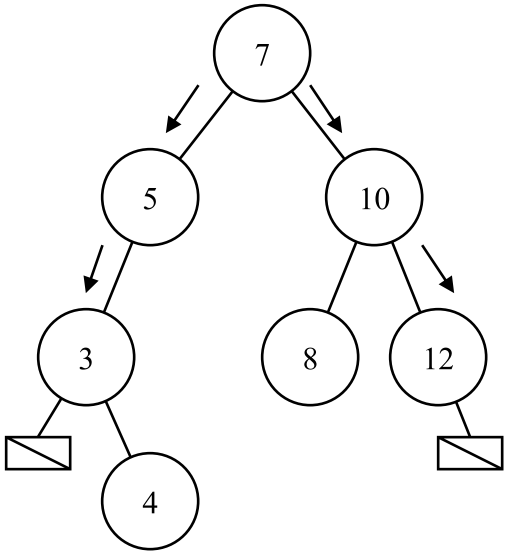
- complexity: $\Theta(n)$
- depth of recurency: $\Theta(h)$

Searching for min and max in BST

Searching for minimum in BST consist in proceeding throw left children, till the node, which has no left child.

Operation of searching maximum proceeds analogous using right children.

```
      Tree-Minimum(x)
{ 1}  while left[x] <> null do
{ 2}     x := left[x]
{ 3}  return x
```



```
      Tree-Maksimum(x)
{ 1}  while right[x] <> null do
{ 2}    x := right[x]
{ 3}  return x
```

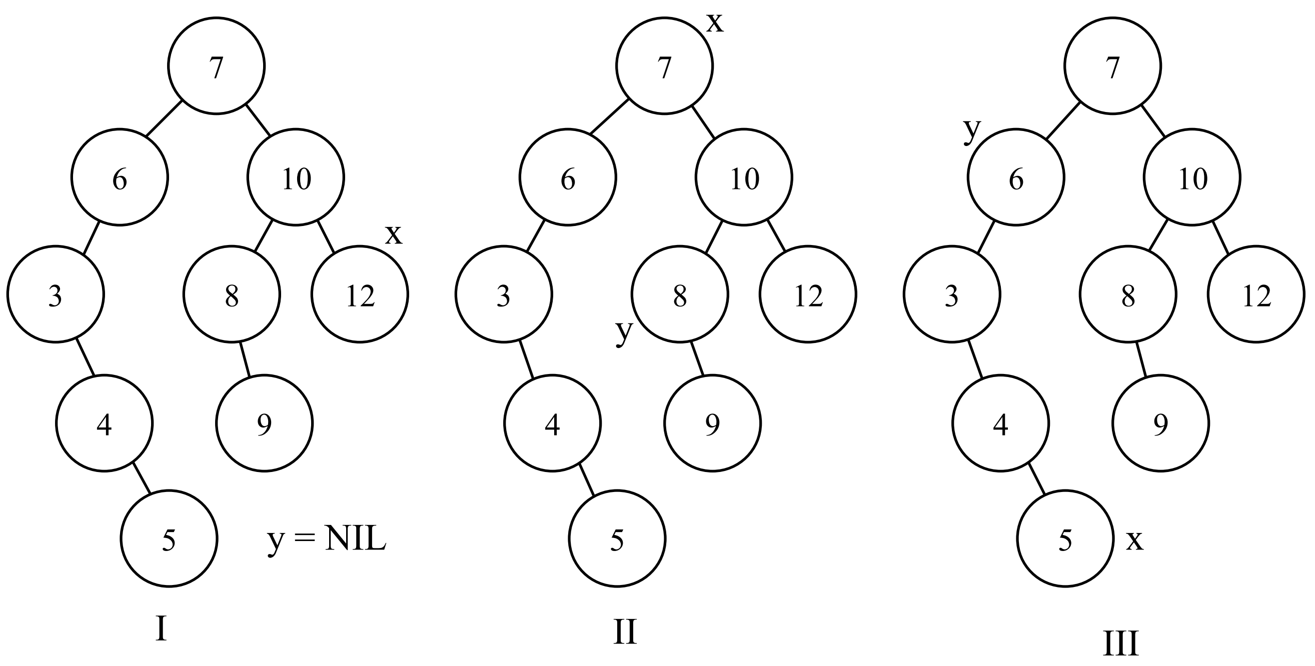
complexity: O(*h*)

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk.

There are 3 cases:

- I) node has no successor
- II) successor of node x is in its right subtree
- III) successor of node x is placed higher



Searching successor in BST

- In case II) it is to search minimum in its right subtree.
- In I) and III) cases it is to search a parent, which has to be on path from left child. If such node does not exist – successor also does not exist.

```
Tree-Successor(x,k)
if right[x] <> NIL then
  return Tree-Minimum(right[x])
y := p[x]
while (y <> NIL) and (x = right[y]) do
  x := y
  y := p[y]
return y
```

Complexity: O(*h*)

Insertion in BST

Insertion new node with key *v* in BST consist in finding new position for this node as a leaf. It is similar to normal search.
We assume that for new node *z* we have *key*[*z*]=*v*, *left*[*z*]=**null**, *right*[*z*]=**null**

```
Tree-Insert(root, z)
y := null
x := root
while x <> null do
  y := x
  if key[z] < key[x]
    then x := left[x]
    else x := right[x]
p[z] := y
if y = null
  then root := z
  else if key[z] < key [y]
    then left[y] := z
    else right[y] := z
```

Tree after insertion values in sequence:
5, 3, 7, 11, 4, 2, 12, 10

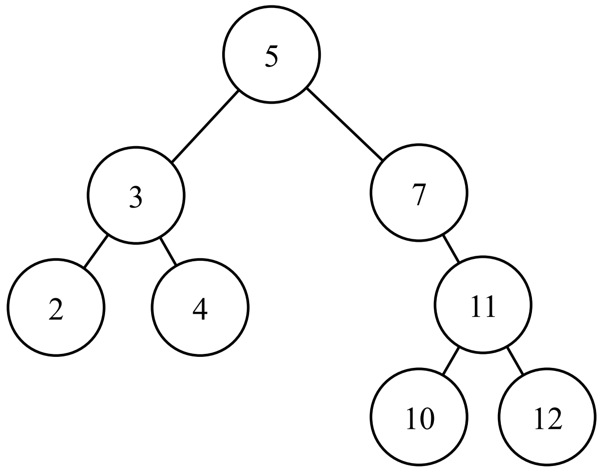
Insertion in BST

Insertion new node with key v in BST consist in finding new position for this node as a leaf. It is similar to normal search.

We assume that for new node z we have $key[z]=v$, $left[z]=$**null**, $right[z]=$**null**

```
Tree-Insert(root, z)
y := null
x := root
while x <> null do
  y := x
  if key[z] < key[x]
    then x := left[x]
    else x := right[x]
p[z] := y
if y = null
  then root := z
  else if key[z] < key [y]
    then left[y] := z
    else right[y] := z
```

Tree after insertion values in sequence:
5, 3, 7, 11, 4, 2, 12, 10



Deletion in BST

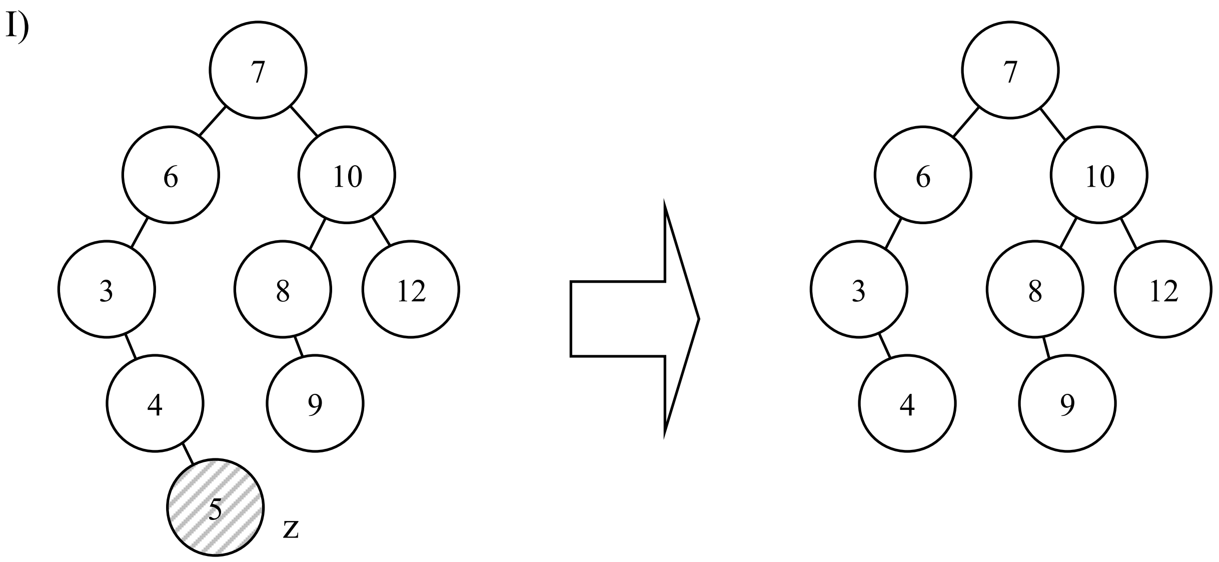
Deletion in BST consist in repairing tree in minimal step to make proper BST.

There are 3 cases during deletion a node ***z***:

- I) node ***z*** has no children.
- II) node ***z*** has exactly one child.
- III) node ***z*** has two children.

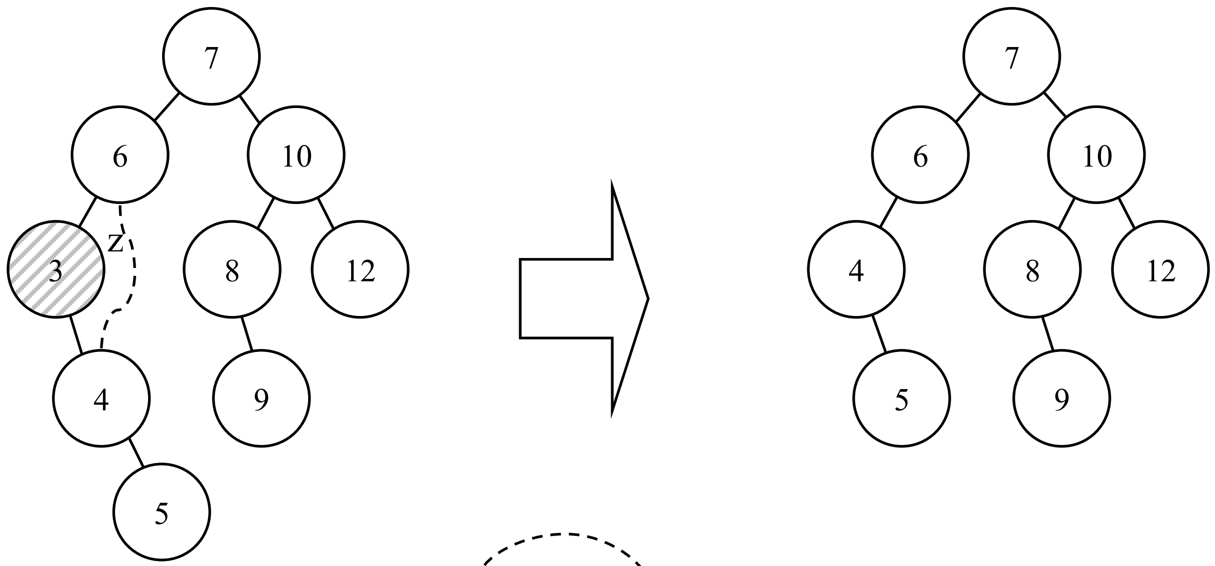
It has to be done:

- I) in parent of node ***z*** modify proper child field to **null**.
- II) in parent of node ***z*** modify proper child field, pointing to ***z,*** to value equal child of node ***z.***
- III) Find the succesor of ***z*** (let's mark it ***y***). This successor has no two children for sure. Swap data fields and key field in ***y*** and ***z***, and then delete node ***y*** (now it is case I or II).

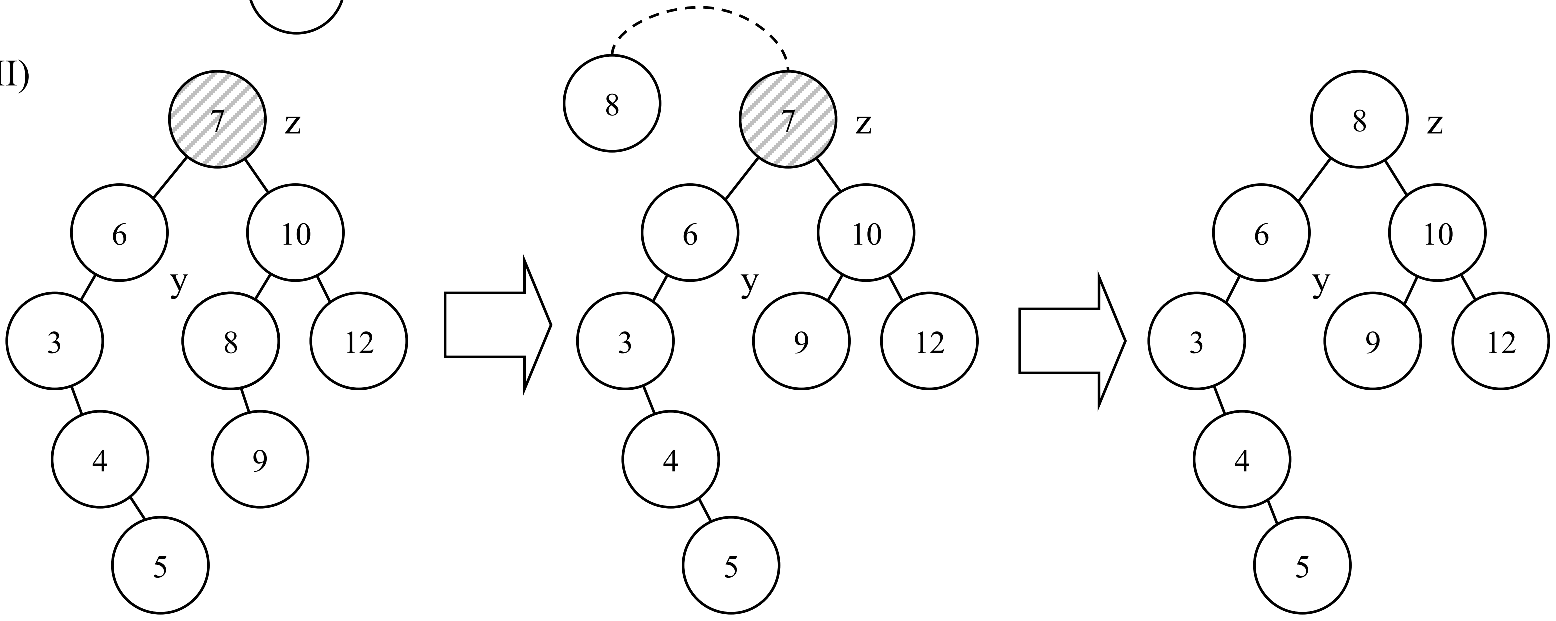


Deletion in BST

II)



III)



Deletion - code

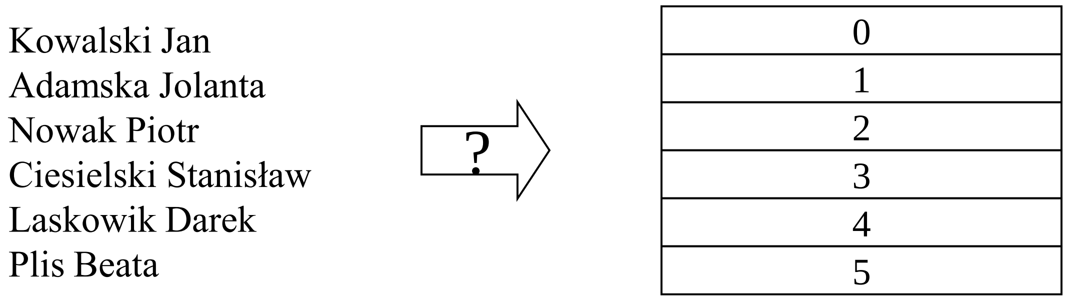
```
Tree-Delete(root, z)
if (left[z] = null) or (right[z] = null)
  then y: = z
  else y := Tree-Successor(z)
if left[y] <> null
  then x := left[y]
  else x := right[y]
if x <> null
  then p[x] = p[y]
if p[y] = null
  then root := x
  else if y = left[p[y]]
    then left[p[y]] := x
    else right[p[y]] := x
if y <> z
  then key[z] := key[y]
  { if node y has another fields, copy them here}
return y
```

complexity: O(*h*)

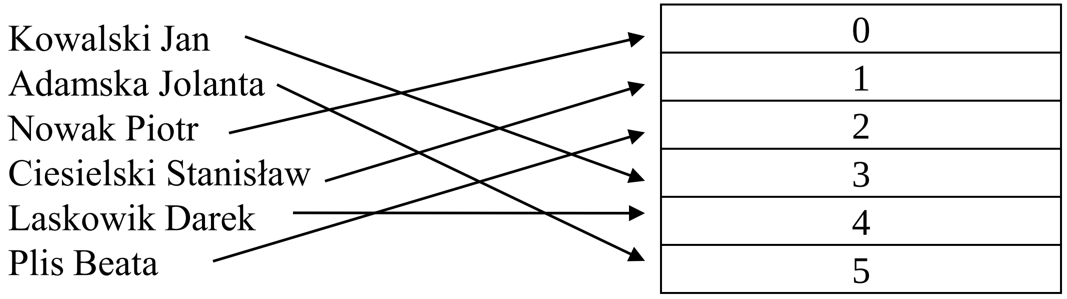
Hashing

insertion and search:

- in a list: O(*n*)
- in a table: O(lg *n*) + O(*n*)
- in a BST: O(lg *n*)
- in ??? : O(1)



Hashing



K - key (here *name*)
h(K) = K[0] mod 6 - generate an index using first letter of a name.

h(„Kowalski”) = („K”) mod 6 = 75 mod 6 = 3
h(„Adamska”) = („A”) mod 6 = 65 mod 6 = 5
h(„Nowak”) = („N”) mod 6 = 78 mod 6 = 0
h(„Ciesielski”) = („C”) mod 6 = 67 mod 6 = 1
h(„Laskowik”) = („L”) mod 6 = 76 mod 6 = 4
h(„Plis”) = („P”) mod 6 = 80 mod 6 = 2

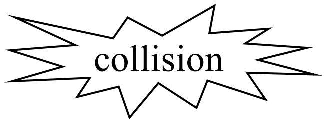
A - 65	G - 71	M - 77
B - 66	H - 72	N - 78
C - 67	I - 73	O - 79
D - 68	J - 74	P - 80
E - 69	K - 75	Q - 81
F - 70	L - 76	R - 82

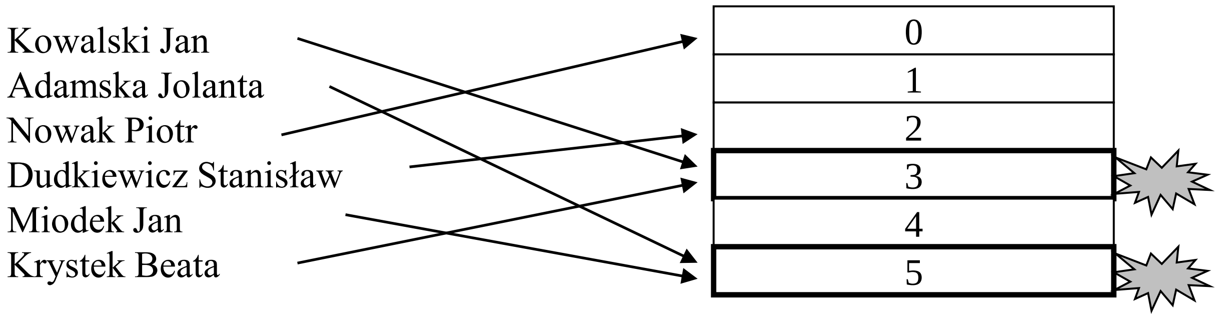
Hashing - Collision

K - key
h(K) - hash function

If h(K) transform different keys into different indexes, it is called **perfect hash function**.

Cases:

- constant set of keys -> search for perfect hash function
- variable set of keys -> 

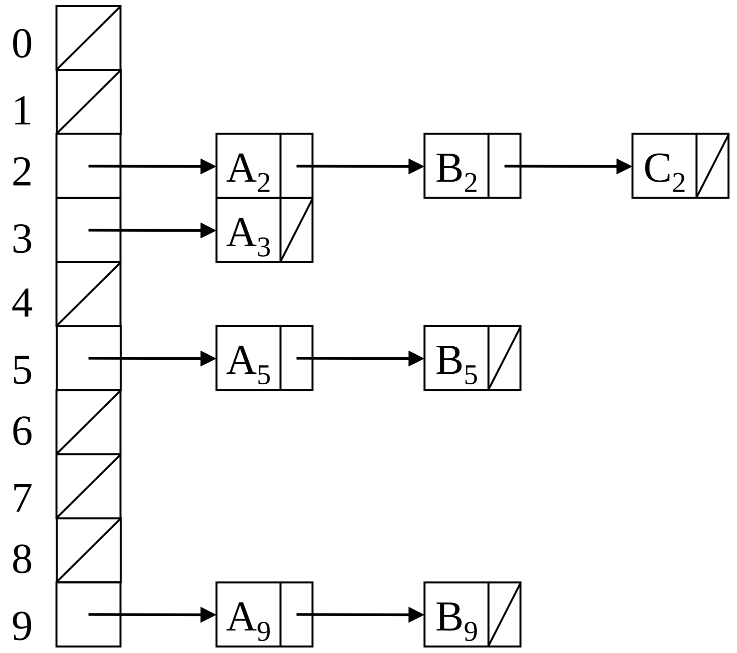


Hash function

- Techniques to achieve good hash function
  - Division method: $h(K) = K \bmod TSize$, *where TSize* – size of array
  - Divide, shift and add:
    for a pesel number: 45071798576 and TSize=1000 divide the number into groups 450-717-985-76 then add achieving 2228 mod 1000 = 228
  - Deletion of identicall part: For database in a publishing house each ISBN number starts with the same number. For hashing it is used only the part which changing.

Collision resolution by chaining

- In chaining, we put all elements that hash to the same slot in a linked list



depends on load factor

Complexity for insertion: O(1)

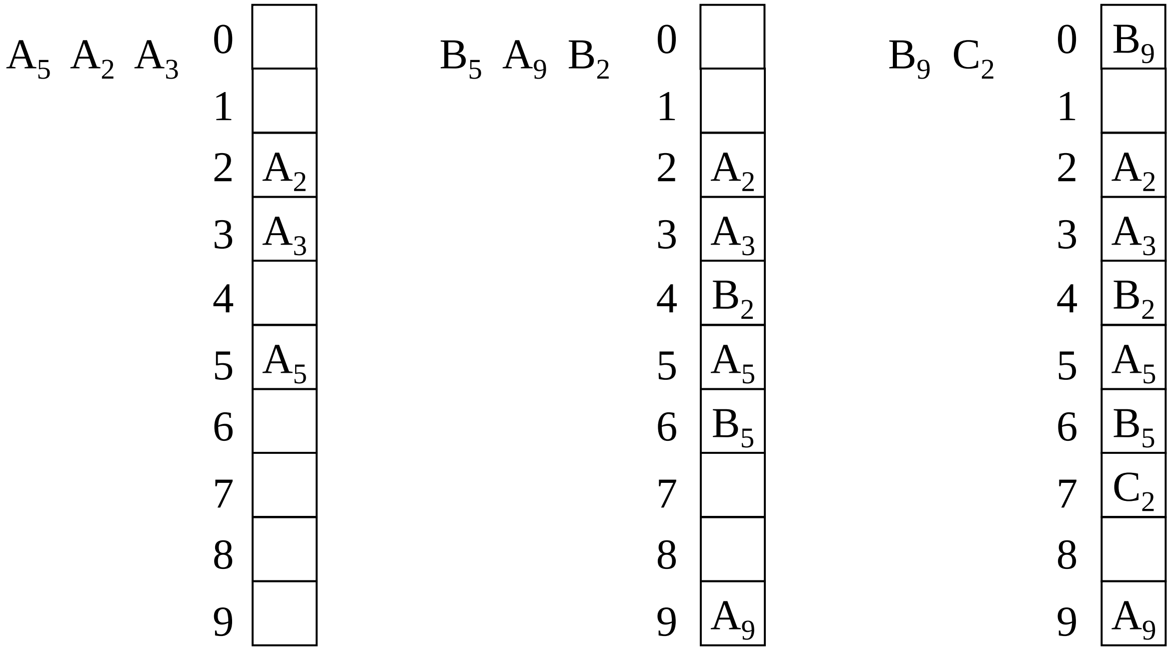
Complexity for search: O(1+a)

Solving collisions in open addressing

- In ***open addressing***, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or **null**.
- in this case we probe to insert new element in following places:
  norm(*h*(*K*)+*p*(1)), *norm*(*h*(*K*)+*p*(2)), ..., *norm*(*h*(*K*)+*p*(*i*)),…
  where: p – increase function,
  norm(…) – normalisation function (a.e modulo)

Linear probing

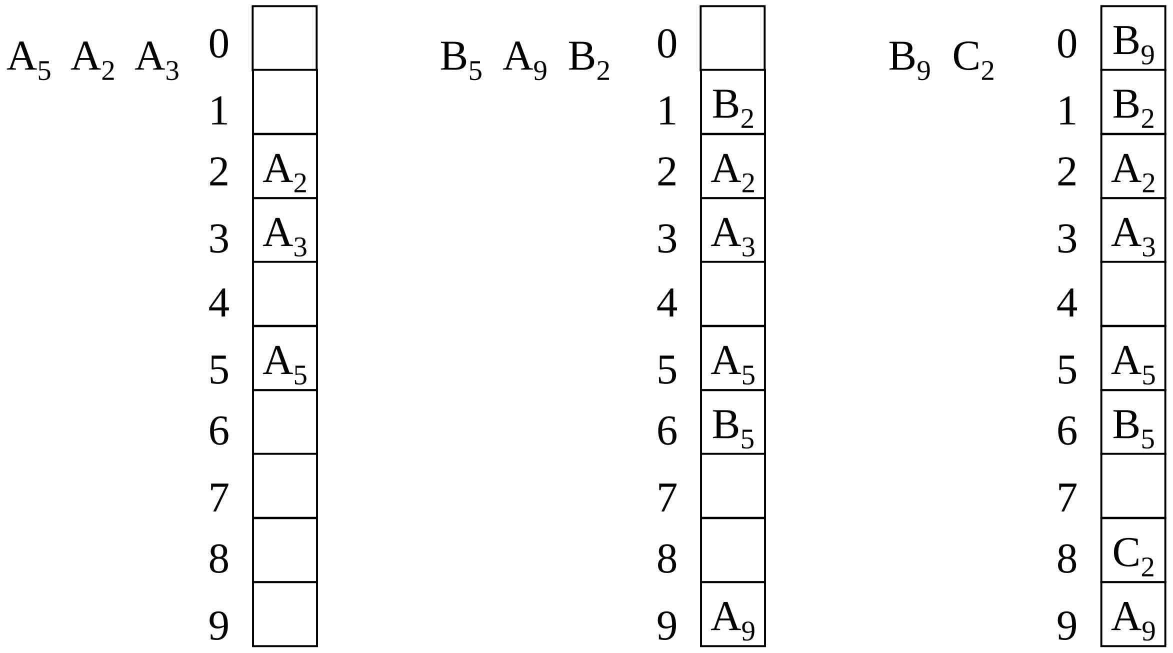
- p(i)=i
- it means, in i-th probe we check (h(K)+i) mod *TSize slot*



Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***.

Quadratic probing

- $p(i)=c_1i+c_2i^2$
- or $p(i)= (-1)^{i-1}((i+1)/2)^2$ (+1, -1, +4, -4, +9, -9, ...)


A_5 A_2 A_3
B_5 A_9 B_2
B_9 C_2
0 1 2 3 4 5 6 7 8 9
A_2 A_3 A_5
B_2 A_2 A_3 A_5 B_5 A_9
B_9 B_2 A_2 A_3 A_5 B_5 C_2 A_9

Quadratic probing leads to ***secondary clustering***.

Statistics of hash tables

Double hashing: $p(i)=i*h_2(K)$

$$\textbf{Load factor LF} = \frac{\text{number of used slots}}{\text{size of table}}$$

	linear probing		quadratic probing		double hashing	
LF	Success	Defeat	Success	Defeat	Success	Defeat
0,05	1,0	1,1	1,0	1,1	1,0	1,1
....						
0,45	1,4	2,2	1,4	2,0	1,3	1,8
...						
0,75	2,5	8,5	2,0	4,6	1,8	4,0
0,80	3,0	13,0	2,2	5,8	2,0	5,0
0,85	3,8	22,7	2,5	7,7	2,2	6,7
0,90	5,5	50,5	2,9	11,4	2,6	10,0
0,95	10,5	200,5	3,5	22,0	3,2	20,0