Chapter 3. Dictionaries

Data Structures and Algorithms

FIB

Q2 2018–19

Jordi Delgado
(after original slides by Antoni Lozano)

Chapter 3. Dictionaries

Chapter 3. Dictionaries

Abstract data types

Data structures can be formalized through the concept of

Abstract Data Types (ADT)

ADT = VALUES + OPERATIONS

Example

The integer type consists of:

- VALUES: set of integers in an interval [min,max]
- OPERATIONS: $+$, $-$, $*$, $/$, %

Abstract data types

Data structures can be formalized through the concept of

Abstract Data Types (ADT)

ADT = VALUES + OPERATIONS

Example

The integer type consists of:

- VALUES: set of integers in an interval [min,max]
- OPERATIONS: $+$, $-$, $*$, $/$, %

Abstract data types

In order to define the behavior of an ADT, we have to specify the **properties** of its operations:

- values
- operations
- properties of the operations

Example

The integer type is specified as follows:

- values: set of integers in an interval [min,max]
- operations: $+$, $-$, $*$, $/$, %
- properties: $a + 0 = a$, $a * 0 = 0$, $a * 1 = a$, $a * b = b * a$, etc.

Abstract data types

The specification of an ADT describes the behavior of a data structure independently of its implementation.

DAT: knowing what without knowing how

Abstract data types

Example: DAT *booleans*

- VALUES: set of booleans Bool $= \{\text{true}, \text{false}\}$
- OPERATIONS: true, false, and, or, not

  true: $\rightarrow$ Bool
  false: $\rightarrow$ Bool
  and: Bool $\times$ Bool $\rightarrow$ Bool
  or: Bool $\times$ Bool $\rightarrow$ Bool
  not: Bool $\rightarrow$ Bool

- PROPERTIES:

  and (true, x) $= x$
  and (false, x) $=$ false
  or (true, x) $=$ true
  or (false, x) $= x$
  not (true) $=$ false
  not (false) $=$ true

Abstract data types

Example: DAT *naturals*

- VALUES: set $\mathbb{N}$ of natural numbers
- OPERATIONS: zero, suc, sum, equal?

  zero: $\rightarrow \mathbb{N}$ sum: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

  suc: $\mathbb{N} \rightarrow \mathbb{N}$ equal?: $\mathbb{N} \times \mathbb{N} \rightarrow$ Bool
- PROPERTIES:

  sum (zero, x) = x

  sum (suc (x), y) = suc (sum (x, y))

  equal? (zero, zero) = true

  equal? (suc(x), zero) = false

  equal? (zero, suc(x)) = false

  equal? (suc(x), suc(x)) = true

Dictionaries

We call dictionary to a set with the operations:

- **assign**: including a new element
- **delete**: removing an element
- **consult**: checking whether an element belongs to it

Dictionaries

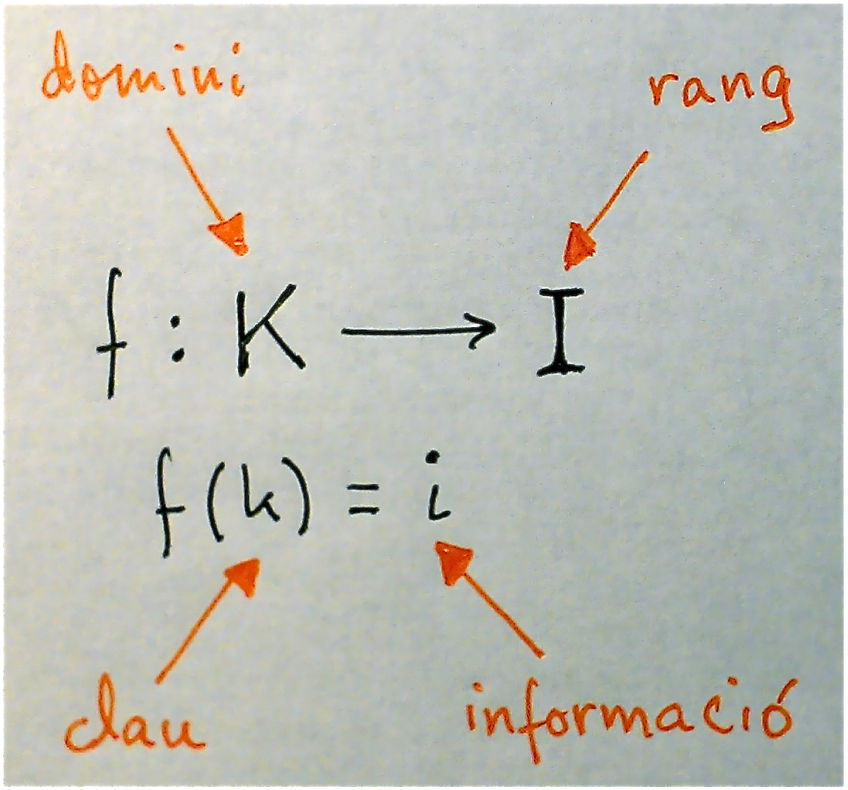
1. They are used to implement symbol tables in compilers and memory tables in operating systems.
2. We will consider variants of **consult** but, in general, if we add more complex operations we obtain new data structures.
    - For example, if we add the operation of **removing the minimum**, we obtain priority queues.

Dictionaries

The elements will have a field called key that will identify them.

element = (key, information)

We can think of dictionaries as functions or generalizations of vectors:



Dictionaries

Example 1

If we have a set of bank accounts,

- the key van be the account number
- the information can be the owner, the account type, the movements, ...

Example 2

A sports club wants to organize the information about members. It chooses:

- the member number as the key
- the key, address, telephone number and email address as part of the information accessible from the key

Dictionaries

Example 1

If we have a set of bank accounts,

- the key van be the account number
- the information can be the owner, the account type, the movements, ...

Example 2

A sports club wants to organize the information about members. It chooses:

- the member number as the key
- the key, address, telephone number and email address as part of the information accessible from the key

Dictionaries

We will assume that:

- there is a bijection between keys and elements
- the keys are a totally ordered set

Dictionaries

We will consider the following operations:

- **assign**: add an element (key, information) to the dictionary. If an element with the same key existed before, information is overwritten.
- **delete**: given a key, the element with that key is removed. If there is no such element, nothing is done.
- **present**: given a key, it returns a boolean indicating whether the dictionary contains an element with that key.
- **search**: given a key, it returns a reference to the element with that key.
- **consult**: given a key, it returns a reference to the information associated with that key.
- **size**: it returns the dictionary size.

Example: DAT *dictionary*

- VALUES: $\mathcal{C} = \mathcal{P}(K \times I)$, where K is a set of keys and I of informations
- OPERATIONS:

  create: $\rightarrow \mathcal{C}$ assign: $K \times I \times \mathcal{C} \rightarrow \mathcal{C}$

  delete: $K \times \mathcal{C} \rightarrow \mathcal{C}$ consult: $K \times \mathcal{C} \rightarrow I$

- PROPERTIES: let us assume $k, k_1, k_2 \in K, i, j \in I, k_1 \neq k_2$

  $\text{delete}\,(k, \text{create}) = \text{create}$
  $\text{delete}\,(k, \text{assign}\,(k, i, D)) = \text{delete}(k, D)$
  $\text{delete}\,(k_1, \text{assign}\,(k_2, i, D)) = \text{assign}\,(k_2, i, \text{delete}\,(k_1, D))$

  $\text{assign}\,(k, i, \text{assign}\,(k, j, D)) = \text{assign}\,(k, i, D)$
  $\text{assign}\,(k_1, i, \text{assign}\,(k_2, j, D)) = \text{assign}\,(k_2, j, \text{assign}\,(k_1, i, D))$

  $\text{consult}\,(k, \text{create}) = \perp$
  $\text{consult}\,(k, \text{delete}\,(k, D)) = \perp$
  $\text{consult}\,(k, \text{assign}\,(k, i, D)) = i$
  $\text{consult}\,(k_1, \text{assign}\,(k_2, i, D)) = \text{consult}\,(k_1, D)$

Dictionaries

Number of elements inspected in the dictionary operations

worst-case/average	assign	delete	consult
non-sorted vector	n, $n/2$	n, $n/2$	n, $n/2$
sorted vector	n, $n/2$	n, $n/2$	$\log n$, $\log n$
non-sorted list	n, n	n, $n/2$	n, $n/2$
sorted list	n, $n/2$	n, $n/2$	n, $n/2$
hash table	n, 1	n, 1	n, 1

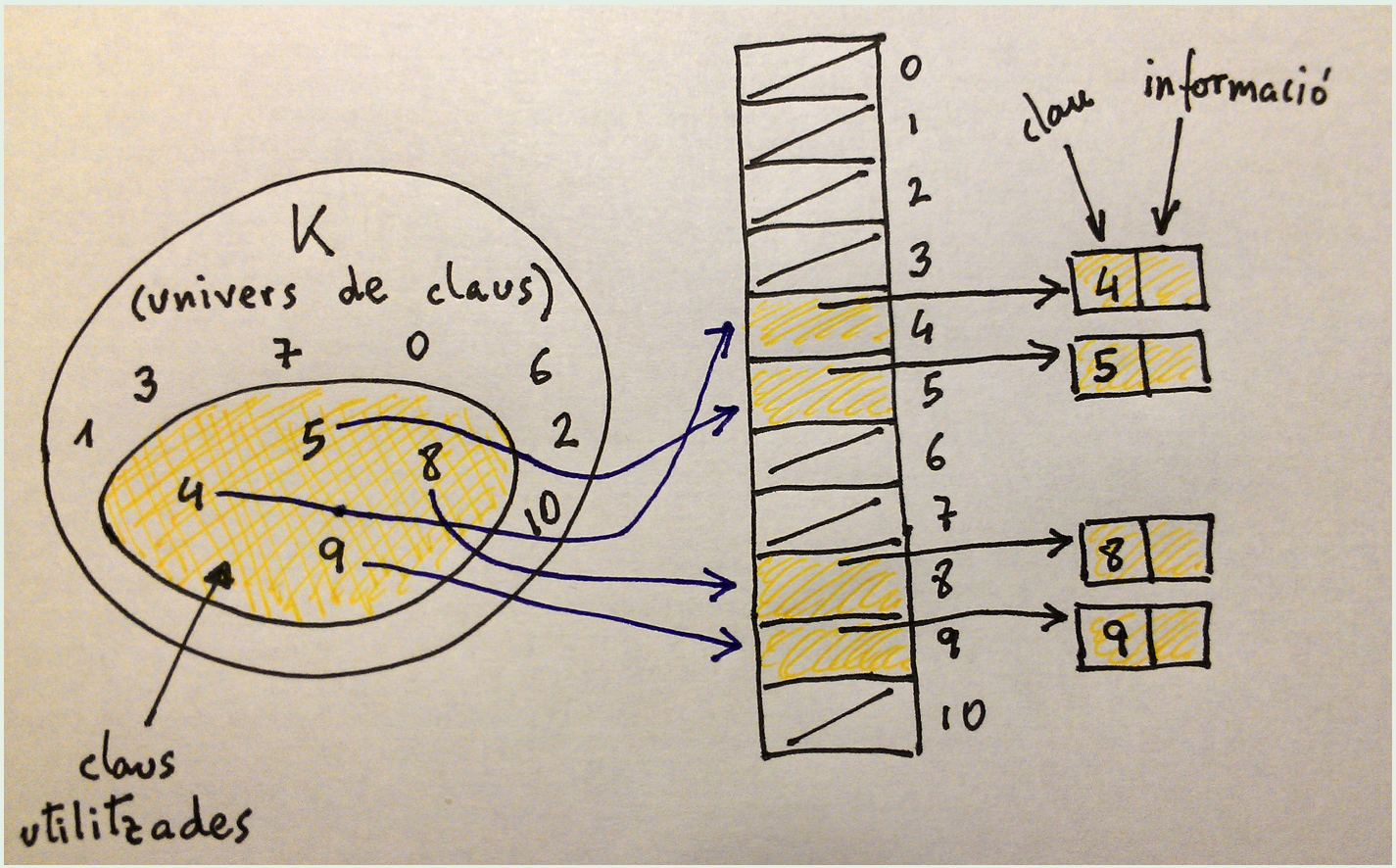
In vectors, we need to shift the elements.
In non-sorted structures, we need to check for repetitions.

Chapter 3. Dictionaries

Direct-access tables

Let us assume that a sports club will never have more than 300 members. To implement a dictionary with the member number as key, it is enough to use a vector $V[0..299]$.

Example of direct access (simplified)



Direct-access tables

In direct access:

- Each position corresponds to a key.
- Operations will be $\Theta(1)$ in the worst case.
- Information can be stored directly in the position of the table corresponding to its key, but we have to mark whether the space is empty.

Direct-access tables

Exercise

We want to implement a dictionary using direct access in a *huge* vector. We have to take into account that:

- initially, entries can contain junk information and
- it is not a good idea to initialize the vector due to its huge size.

Describe a method to implement the operations **consult**, **assign** and **delete** in time $\Theta(1)$.

Direct-access tables

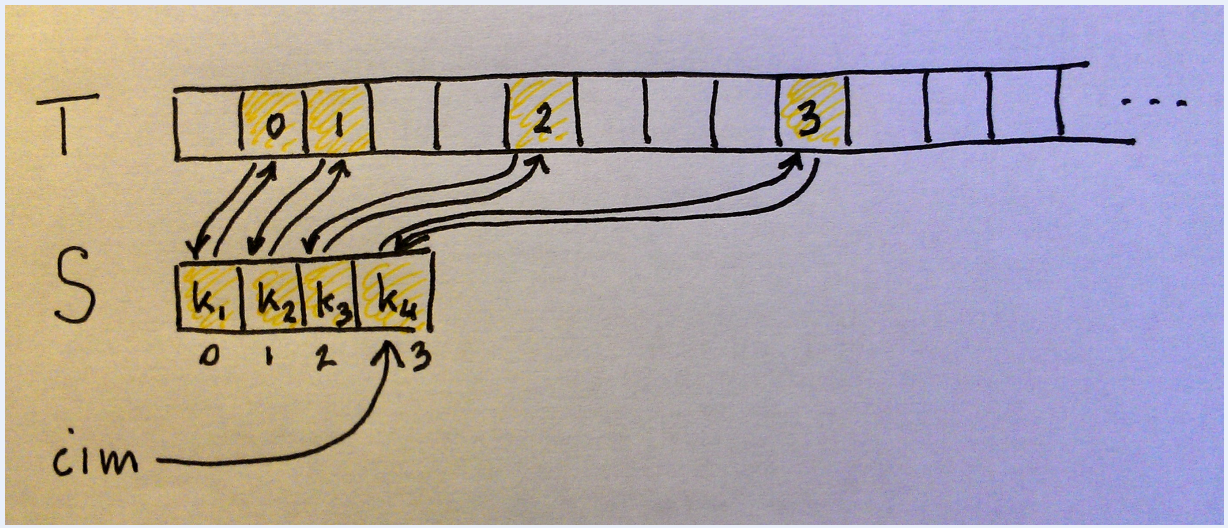
Solution

We define a vector S with as many entries as keys to store ($cim + 1$) such that for a key k:

- $T[k]$ contains the index j of a valid entry of S and
- $S[j]$ contains k.

That is, $S[T[k]] = k$ and $T[S[j]] = j$ (we call this a **validation cycle**).

We also define a vector S' that contains the information. When key k defines a validation cycle, $S'[T[k]]$ contains the information.



Direct-access tables

Solution

Operations:

- **initialize**

```
cim = -1;
```

- **consult**. Given a key `k`,

```
if (S[T[k]] == k)
    return S'[T[k]];
else
    return NULL;
```

Direct-access tables

Solution

Operations:

- **assign**. Given an object `x` with key `k`, if the key is not present

```
++cim;
S[cim] = k;
S'[cim] = x;
T[k] = cim;
```

- **delete**. Given a key `k` (assuming that the key is present), we need to guarantee that no "hole" is left in S.

```
S[T[k]] = S[cim];
S'[T[k]] = S'[cim];
T[S[T[k]]] = T[k];
T[k] = 0;
--cim;
```

Hash tables

Hash tables are efficient data structures to implement dictionaries.

- They are generalizations of vectors.
- Worst-case time can be $\Theta(n)$, but with reasonable assumptions expected time will be $\Theta(1)$ in all operations.

They have been used since the 1950s:

W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1(2), April 1957.

Hash tables

When

- the number of used keys is small comparing with the number of possible keys or
- the number of keys is huge

we should not use direct-access tables. Then,

Instead of using the key as the index to access the table, the index will be computed using the key.

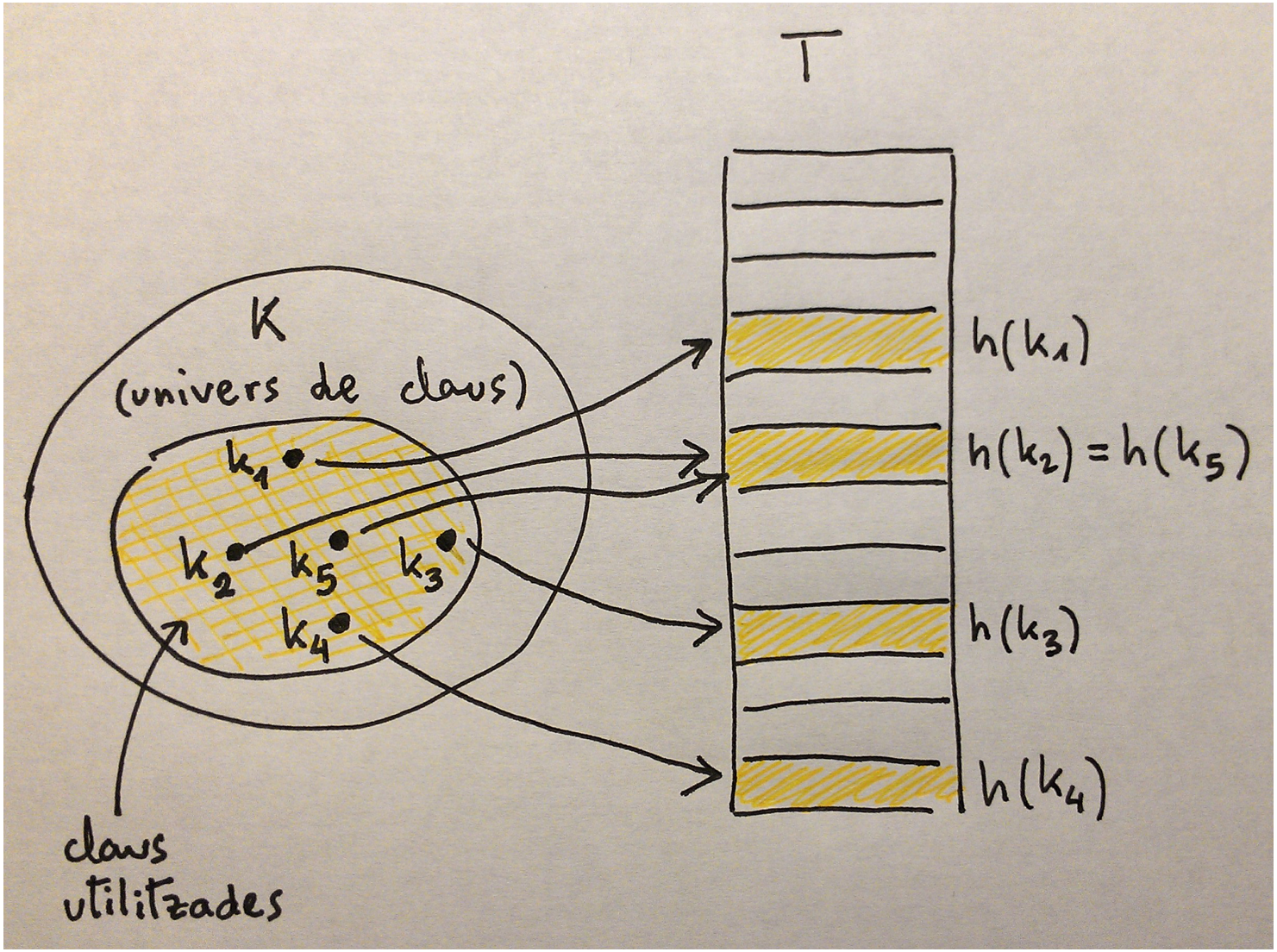
Hash tables

Let us assume that we want to store at most n keys. We declare a table T with $m \geq n$ positions.

- With direct access, $m = n$ and element with key k goes to slot k
- With hash tables, the element goes to the slot $h(k)$, where h is the hash function

$$h : K \to \{0, 1, \ldots, m-1\}$$

Hash tables



The situation where two different keys go to the same slot (like k_2 and k_5) is called collision.

Collisions

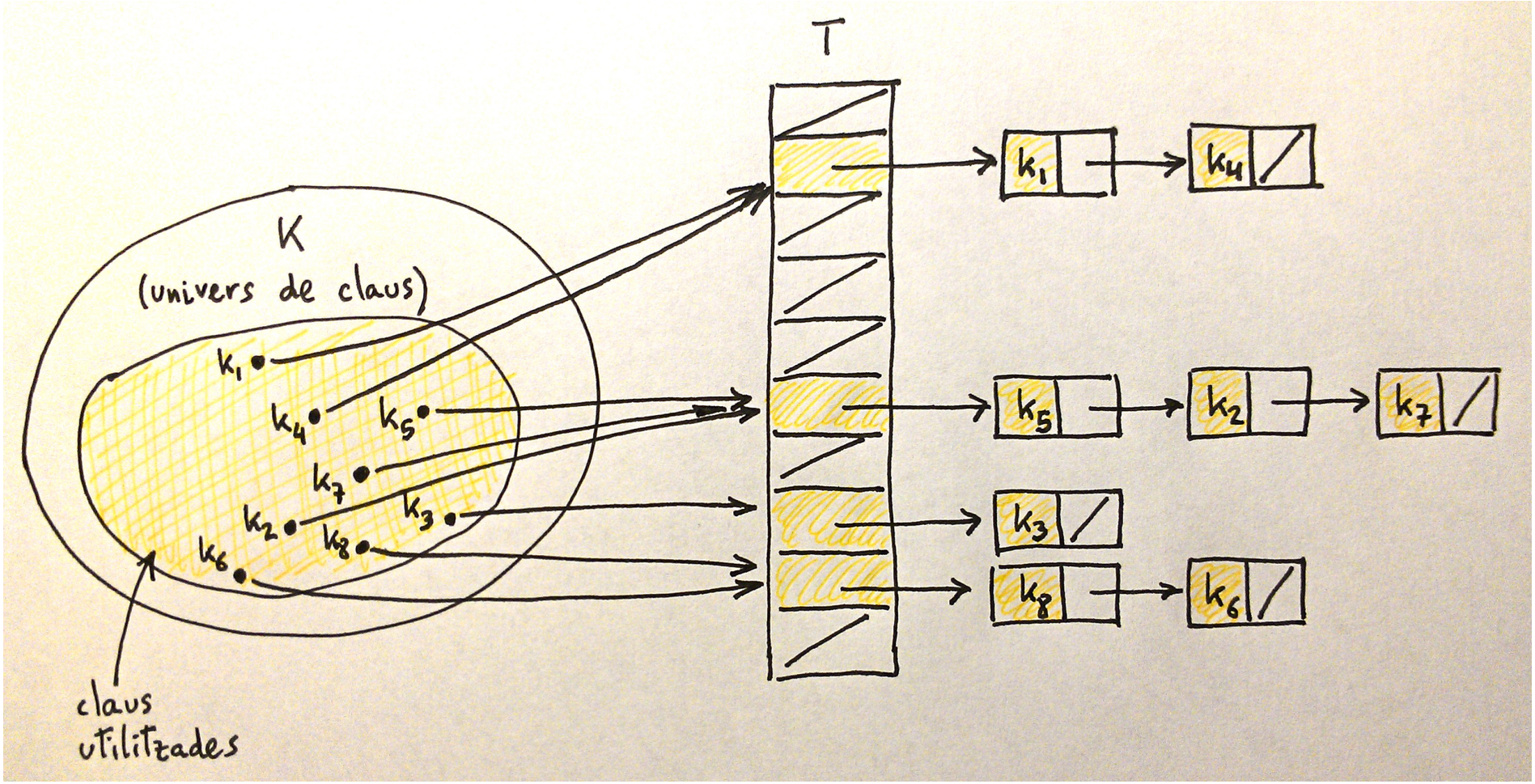
The ideal situation should avoid collisions by choosing a good hash function h.

Some considerations about h:

- It has to be deterministic but look random.
- Since $|K| > m$, collisions cannot be avoided.
- We have to choose a method to resolve collisions.

The simplest method to resolve collisions is separate chaining.

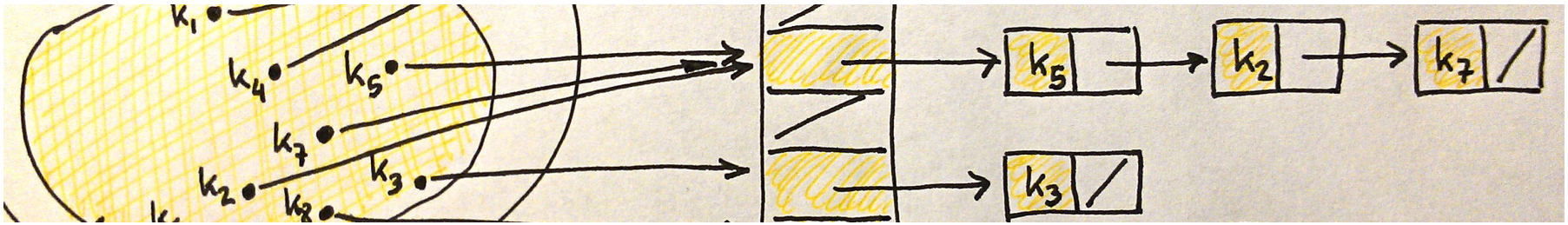
Collisions



Resolving collisions by separate chaining. Each entry $T[j]$ has a linked list with they keys with hash value j.
For example, $h(k_1) = h(k_4)$ and hence $T[h(k_1)]$ points to k_1 followed by k_4.

Separate chaining

In separate chaining, elements with the same hash value are placed in a linked list: $T[i]$ points to the head of the list of elements with hash value i.



The cost of consulting is $\Theta(n)$ worst-case: when all elements have the same hash value and form a unique list.

On average, the cost of a consult can be considered as constant.

Separate chaining

Dictionary class: Implementation (*Algorithms in C++, EDA*)

```
template <typename Key, typename Info>
class Dictionary {

private:

typedef pair<Key, Info> Pair;
typedef list<Pair> List;
typedef typename List::iterator iter;

vector<List> t; // Hash Table
int n;          // Number of Keys
int M;          // Number of Positions
```

Separate chaining

Dictionary class: Implementation

```
public:

Dictionary (int M = 1009)
: t(M), n(0), M(M) { }

void assign (const Key& key, const Info& info) {
     int h = hash(key) % M;
     iter p = find(key, t[h]);
     if (p != t[h].end())
        p->second = info;
     else {
        t[h].push_back(Pair(key, info));
        ++n;
     }
}
```

Separate chaining

Dictionary class: Implementation

```
void delete (const Key& key) {
     int h = hash(key) % M;
     iter p = find(key, t[h]);
     if (p != t[h].end()) {
        t[h].erase(p);
        --n;
     }
}

Info& consult (const Key& key) {
     int h = hash(key) % M;
     iter p = find(key, t[h]);
     if (p != t[h].end())
        return p->second;
     else
        throw ''Key does not exist'';
}
```

Dictionary class: Implementation

```
bool present (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    return p != t[h].end();
}

int size () {
    return n;
}
```

The worst case of operations assign, delete, consult and contains is $\Theta(n)$
Average cost is $\Theta(1 + \frac{n}{M})$

Dictionary class: Implementation

```
bool present (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    return p != t[h].end();
}

int size () {
    return n;
}
```

The worst case of operations assign, delete, consult and contains is $\Theta(n)$
Average cost is $\Theta(1 + \frac{n}{M})$

Dictionary class: Implementation

Costs depend on a search in a list, that are: worst case $\Theta(n)$ and average case $\Theta(1 + \frac{n}{M})$

```
private:

static iter find (const Key& key, list<Pair>& L) {
     iter p = L.begin();
     while (p != L.end() and p->first != key)
          ++p;
     return p;
}
```

Separate chaining: Worst case analysis

Proposition

Worst-case cost of a search in separate chaining is $\Theta(n)$.

Corollary

Worst-case cost of an assignment or deletion (with previous search) in separate chaining is $\Theta(n)$.

Separate chaining: Average case analysis

Let us assume that table T has m positions and stores n elements.

Definition

The load factor $\alpha = n/m$ for T is the average number of elements per position.

Proposition

Average-case cost of a search in separate chaining is $\Theta(1 + \alpha)$.

Proof

Let us assume that we are searching a key k in T.
The expected cost is $\Theta(1 + E[X])$, where X is the size of $T[h(k)]$.
Define the boolean variable $X_e = [h(e) = h(k)]$. Then

$$E[X] = E[\sum_e X_e] = \sum_e E[X_e] = \sum_e prob(X_e = 1) = n \cdot (1/m)$$

Thus, $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$

Separate chaining: Average case analysis

Let us assume that table T has m positions and stores n elements.

Definition

The load factor $\alpha = n/m$ for T is the average number of elements per position.

Proposition

Average-case cost of a search in separate chaining is $\Theta(1 + \alpha)$.

Proof

Let us assume that we are searching a key k in T.
The expected cost is $\Theta(1 + E[X])$, where X is the size of $T[h(k)]$.
Define the boolean variable $X_e = [h(e) = h(k)]$. Then

$$E[X] = E[\sum_e X_e] = \sum_e E[X_e] = \sum_e prob(X_e = 1) = n \cdot (1/m)$$

Thus, $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$

Separate chaining: Average case analysis

Proposition

Average-case cost of a search in separate chaining is $\Theta(1 + \alpha)$.

Corollary

Average-case cost of an assignment, deletion or a consult (with previous search) in separate chaining is $\Theta(1 + \alpha)$.

Chaining

Example

We want to store the information about students enrollment:

- If a name has ≤ 20 characters, the space of possible keys is $\approx 27^{20}$.
- The maximum number of new students is $n = 300$.

We define a vector with $m = 300$ positions and hence each synonym list will have size ≈ 1 on average and operations will have cost $\Theta(1)$.

But how do we choose the hash function?

Hash functions

Keys will always be natural numbers. Otherwise, they are interpreted as such.

Example

Given a string of characters, we interpret it as a natural number.
Given the string “CLRS”:

- values in ASCII: C= 67, L= 76, R= 82, S= 83
- there are 128 characters in ASCII
- hence, CLRS is transformed in the natural number

$$67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0$$

$$= 141.764.947$$

Hash functions

The division method

We hash a key k in one of the m positions using the function

$$h(k) = k \bmod m$$

- Example: if the table has size $m = 12$ and $k = 100$, then $h(k) = 4$.
- Advantage: it is quite fast
- Disadvantage: values of m such as 2^i should be avoided.
- Good choices for m: prime numbers not close to powers of 2.

Example

We want a hash table with separate chaining to store about $n = 2000$ string characters. It is OK for us to examine about 3 elements in a failed search.

Hence, we choose a table with size

$$m = 701.$$

We choose 701 because it is a primer number close to $2000/3$ but it is not a power of 2. The hash function will be

$$h(k) = k \bmod 701.$$

Hash functions

Multiplication method

To hash a key k into one of the m positions:

1. multiply k by a constant A s.t. $0 < A < 1$ and extract its fractional part
2. then, we multiply its value by m and take the floor

$$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor.$$

- Advantage: the value of m is not critical.
- Disadvantage: it is slower than the division method
- Good choices for m : powers of 2 (make implementation easy).
- Good choice for A : Knuth suggests

$$A \approx (\sqrt{5} - 1)/2 = 0,6180339887\ldots$$

Example

In a hash table with $m = 1024$, $k = 123$ i $A = 0,61803399$, we have

$$h(k) = \lfloor 1024 \cdot (123 \cdot A - \lfloor 123 \cdot A \rfloor) \rfloor$$

$$= \lfloor 1024 \cdot (76,0181808 - 76) \rfloor$$

$$= \lfloor 1024 \cdot 0,0181808 \rfloor = 18.$$

Open addressing

An alternative to separate chaining is open addressing:

- all elements are stored in the table
- when we look for an element, we examine the positions in a systematic way until we find it
- there are no pointers to positions in the same table
- hash function has two parameters: the key and the "attempt"

$$h : K \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$

For a key k, the sequence of positions tried is

$$(h(k, 0), h(k, 1), \ldots, h(k, m-1)).$$

Chapter 3. Dictionaries

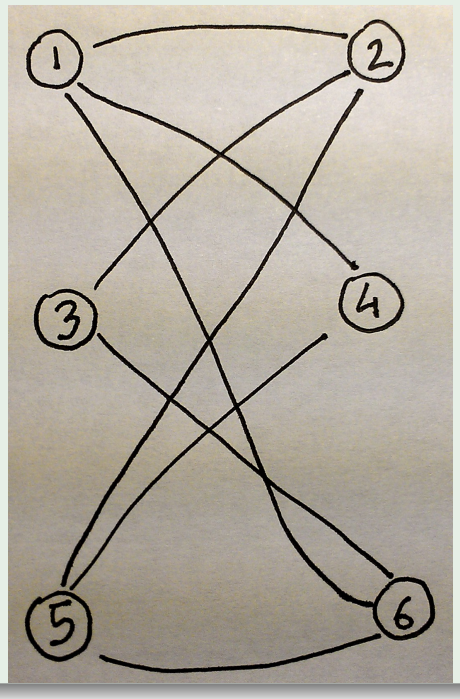
Definitions and terminology

Definition

A graph is a pair (V, E) where:

- V is a finite set (*vertices*)
- E is a finite set of unordered pairs of vertices (*edges*)

Graph example (1)



- Connected: any vertex is reachable from any other ones (through edges).
- Cyclic: there are cycles, such as $1 \to 2 \to 5 \to 4 \to 1$.

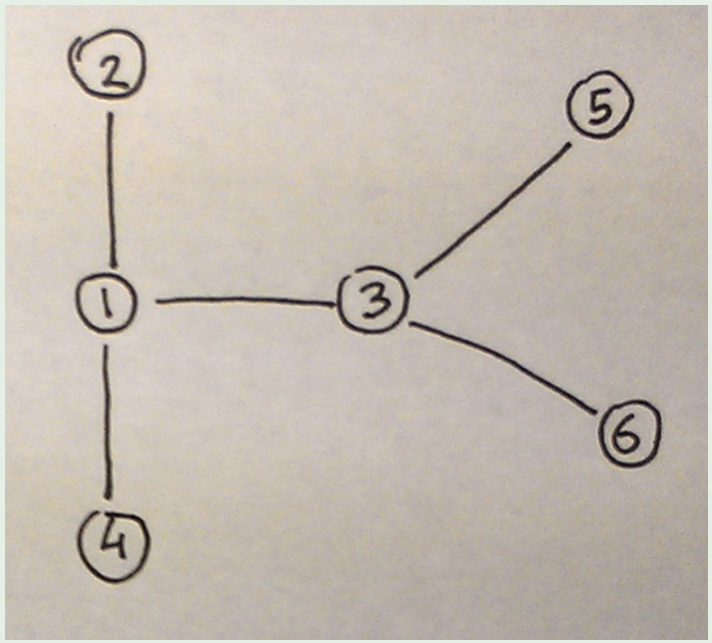
Definitions and terminology

Definition

A graph is a pair (V, E) where:

- V is a finite set (*vertices*)
- E is a finite set of unordered pairs of vertices (*edges*)

Graph example (2)



Connected and acyclic graph.

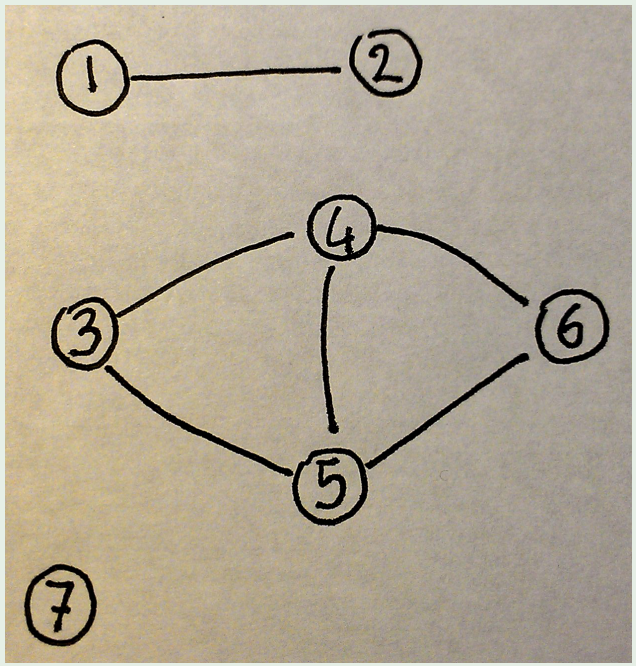
Definitions and terminology

Definition

A graph is a pair (V, E) where:

- V is a finite set (*vertices*)
- E is a finite set of unordered pairs of vertices (*edges*)

Graph example (3)



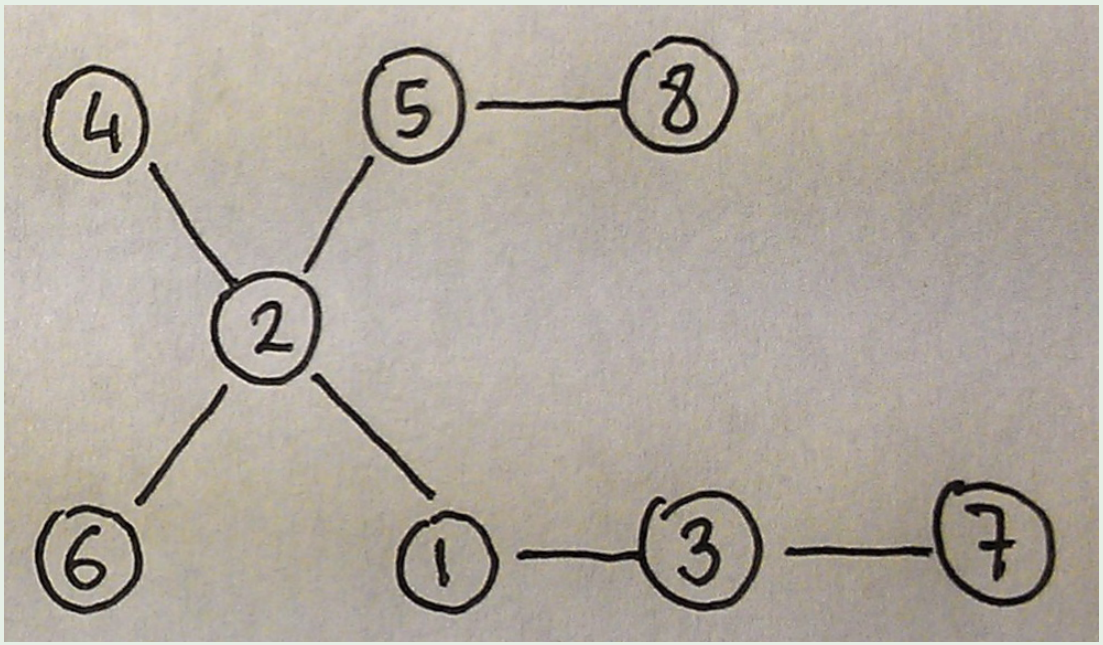
Unconnected and cyclic graph.

Definitions and terminology

Definition

A tree is a connected and acyclic graph.

Tree example (1)



Definitions and terminology

Theorem

Let $G = (V, E)$ be a graph and let $n = |V|$ and $m = |E|$.

The following statements are equivalent:

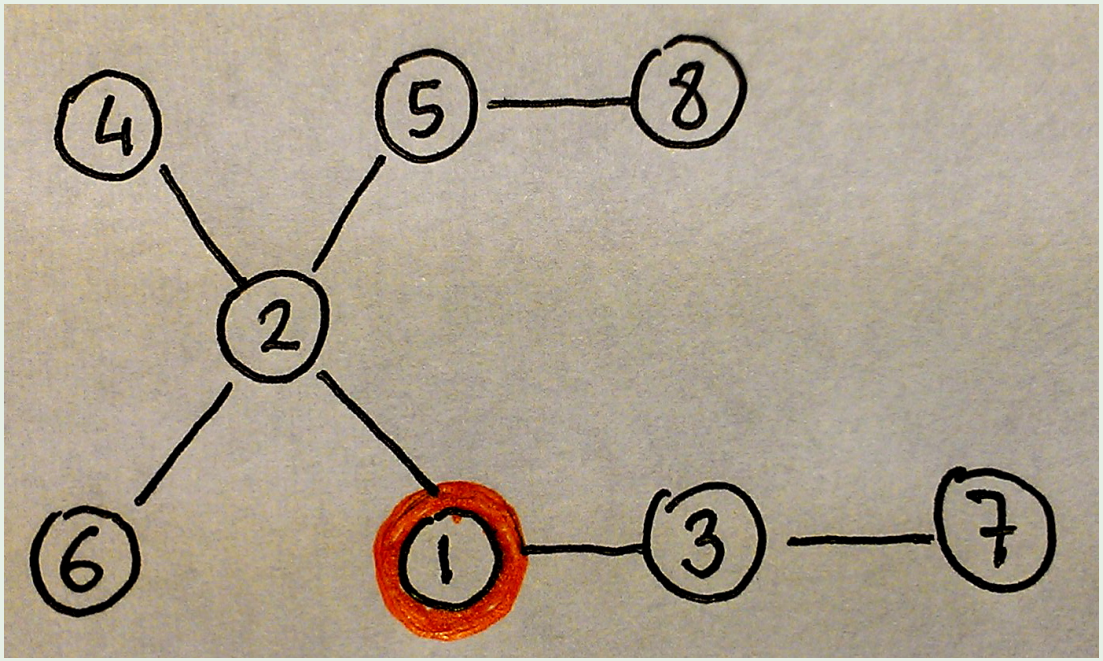
- G is a tree.
- Any pair of vertices of G are connected by a unique path.
- G is connected and $m = n - 1$.
- G is connected, but if an edge is removed, the graph becomes unconnected.
- G is acyclic and $m = n - 1$.
- G is acyclic, but if we add an edge, the graph becomes cyclic.

Definitions and terminology

Definition

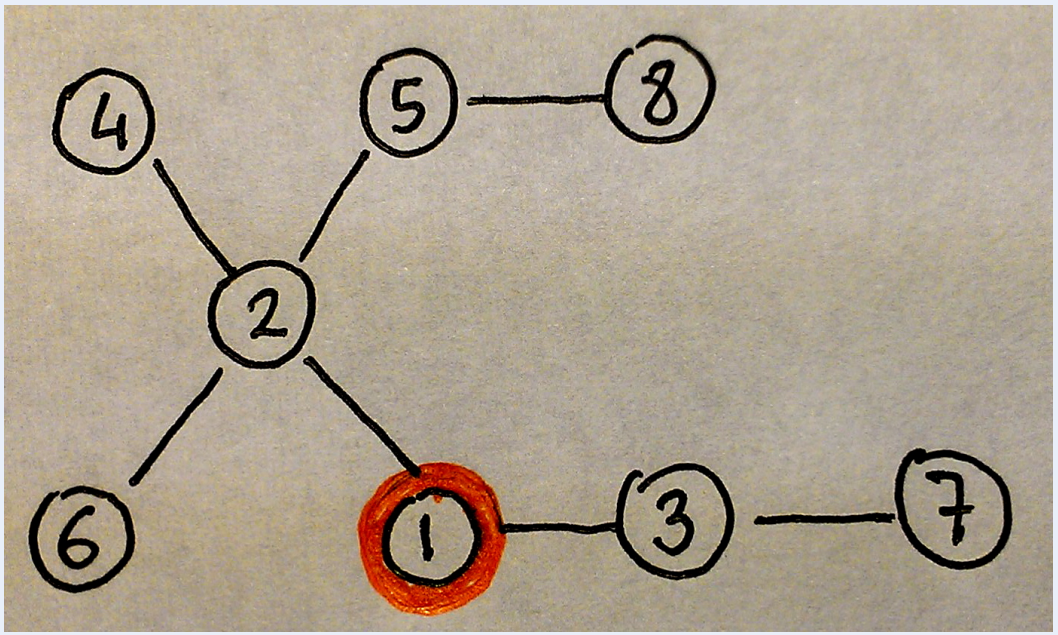
A rooted tree is a connected and acyclic graph with a distinguished vertex.

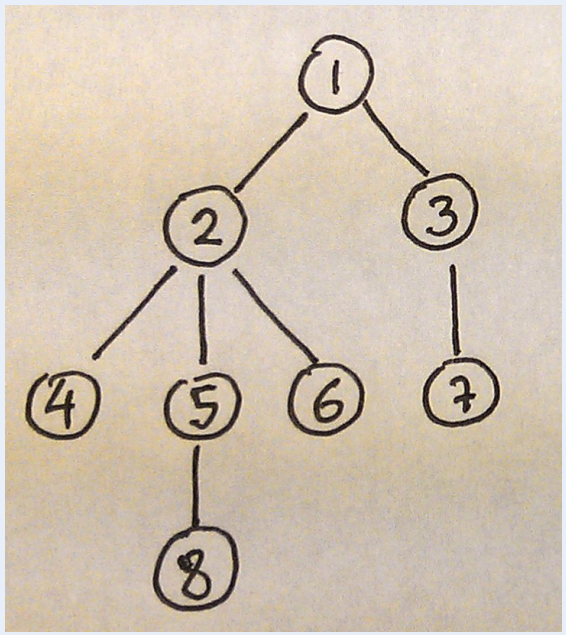
Tree example (2)



Representation

We will represent the rooted trees with the distinguished vertex on top.





Definitions and terminology

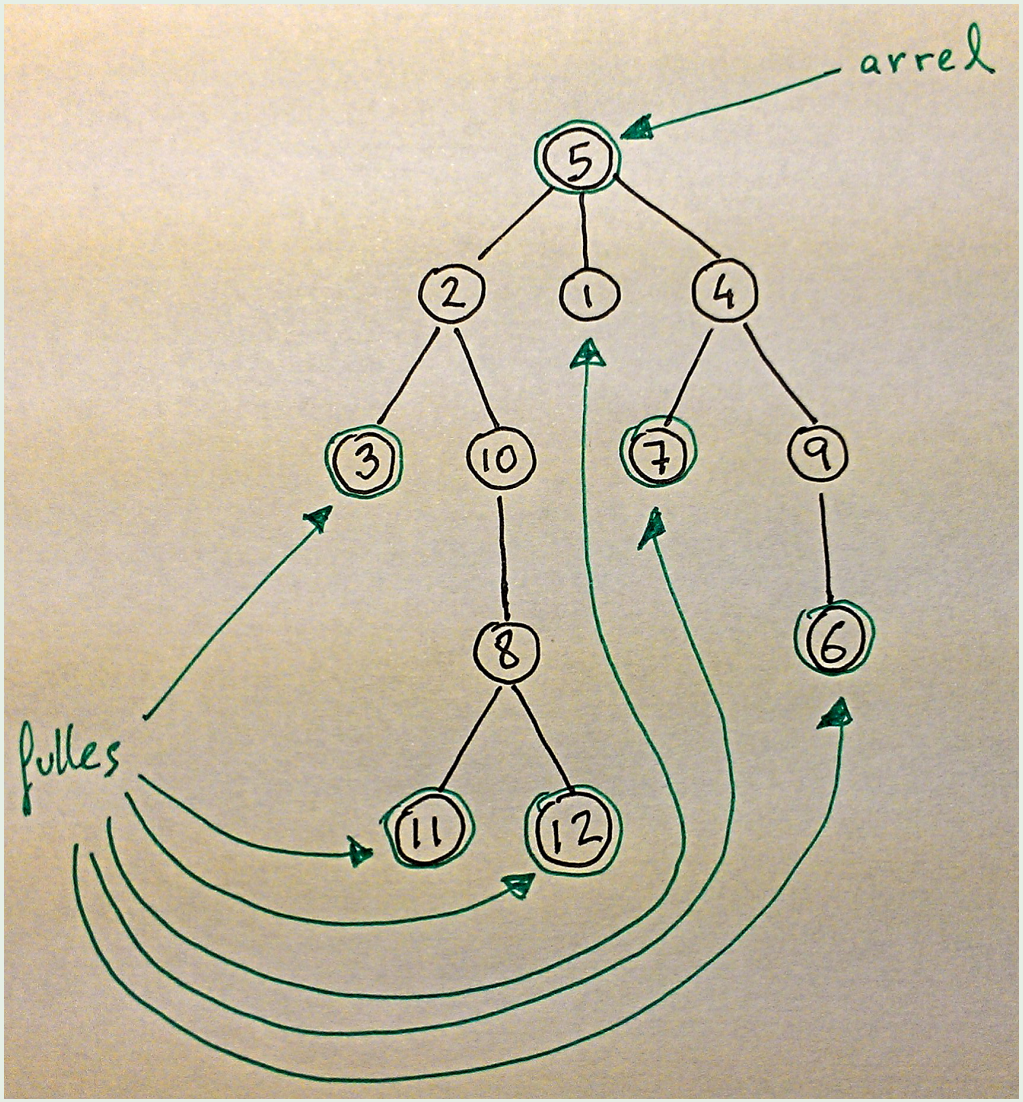
Terminology related to trees

Vertices are called nodes. The distinguished node is called root.

Let x be a node from a tree T with root r:

- every node y in the path from r to x is a predecessor of x
- if y is a predecessor of x, then x is a descendant of y
- if y is a predecessor of x and the edge $\{y, x\}$ belongs to T, then y is the father of x and x is the son of y
- the father's father of x is called grandfather of x; if y is the grandfather of x, then x is the grandson of y
- two nodes with the same father are called siblings
- a node with no children is called a leaf
- a non-leaf node is an internal node
- the height of T is the maximum distance (number of edges in a path) between the root and a leaf

Tree example (3)



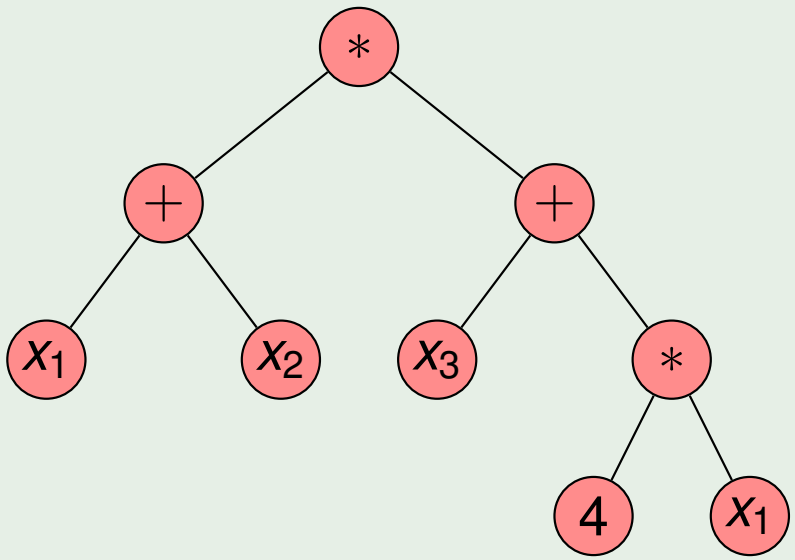
- 4 is the father of 9
- 6 is grandson of 4
- the predecessors of 10 are 5, 2 and 10
- the successors of 10 are 10, 8, 11 and 12
- 3 and 10 are siblings
- 9 is an internal node
- the tree has height 4 (distance between 5 and 11 or between 5 and 12)

Definitions and terminology

In this chapter, by *tree* we will mean *rooted tree*.

Tree example (4)

Tree representing the expression $(x_1 + x_2) * (x_3 + 4 * x_1)$:



In the previous example, each node has at most two children. If operations were not commutative, it would be important to distinguish between left and right child.

Definitions and terminology

Definition

A binary tree is a rooted tree where each node has at most two children, which are referred to as left child and right child.

Binary trees can be defined recursively.

Definition

A binary tree is a structure built over a finite set of nodes such that:

- it does not contain any node, or
- it is built of three sets of nodes: the root, a binary tree called left subtree and another binary tree called right subtree.

Definitions and terminology

Definition

A binary tree is a rooted tree where each node has at most two children, which are referred to as left child and right child.

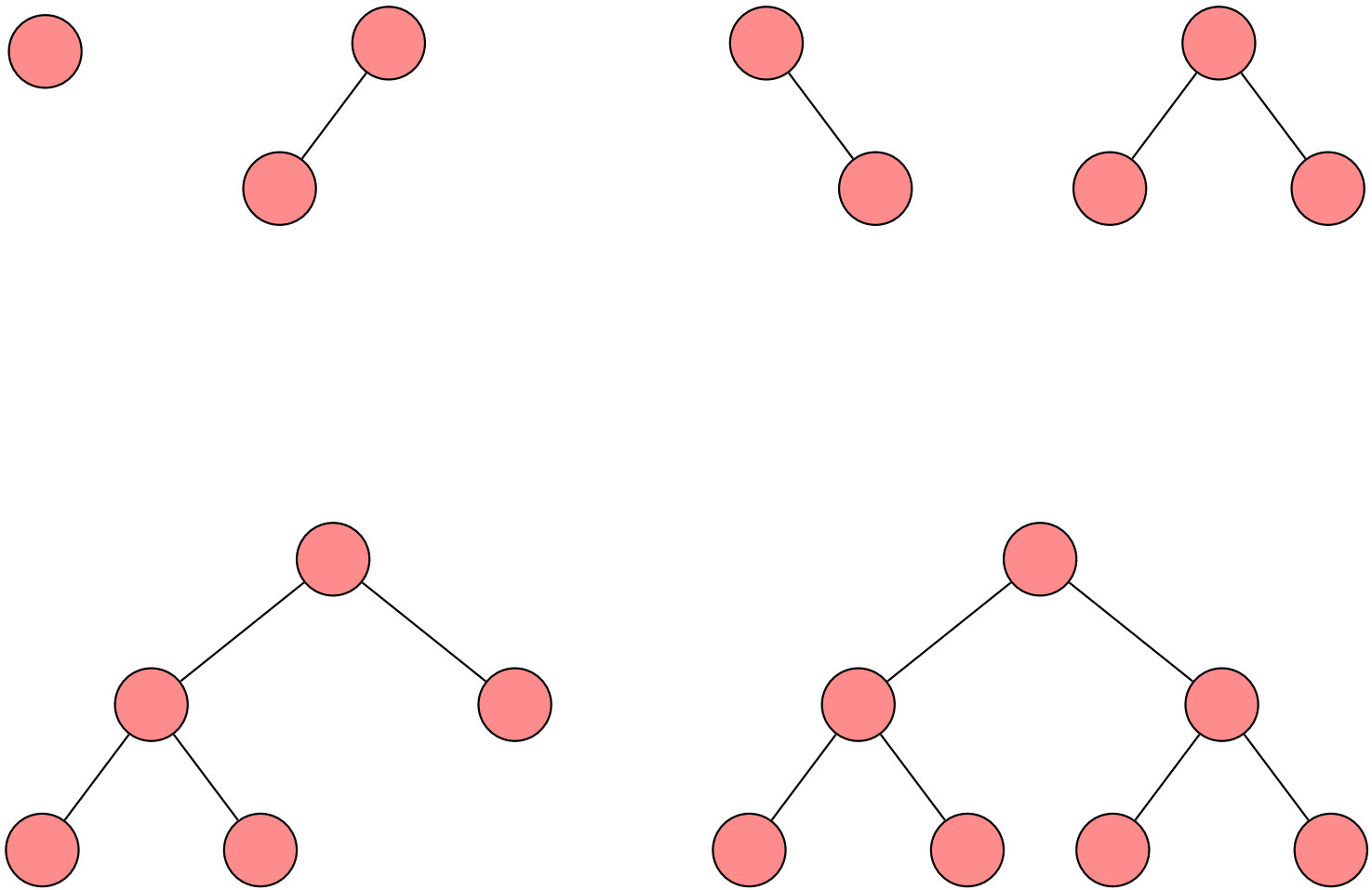
Binary trees can be defined recursively.

Definition

A binary tree is a structure built over a finite set of nodes such that:

- it does not contain any node, or
- it is built of three sets of nodes: the root, a binary tree called left subtree and another binary tree called right subtree.

Binary trees



Chapter 3. Dictionaries

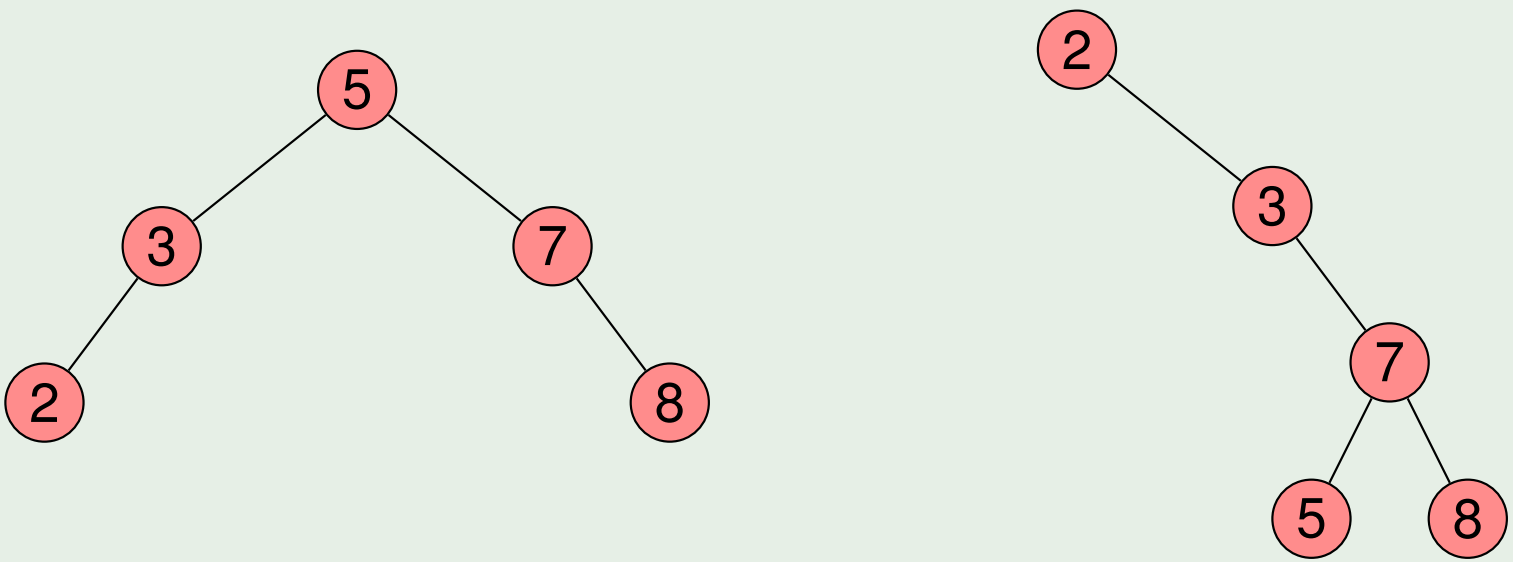
Introduction

Definition

A binary search tree (BST) is a binary tree that has a key associated with each node and that satisfies that the key of every node is

- greater than the key of all nodes in the left subtree
- smaller than the key of all nodes in the right subtree

Example



Implementation (BST)

Exercise

Define binary search trees recursively.

Introduction

Dictionary operations.

- BSTs allow one to perform the basic operations from dictionaries (**consult**, **assign**, **delete**) in time proportional to the height of the tree.
- The expected height of a BST with n nodes is $\Theta(\log n)$.
- Hence, basic operations have an average cost of $\Theta(\log n)$ in a BST.
- In the worst case, the costs of basic operations in a BST are $\Theta(n)$.

Introduction

The property of BSTs allow one to implement other operations such as

- compute the ordered list of its elements in time $\Theta(n)$
- find the maximum or the minimum in average time $\Theta(\log n)$
- find the next or the previous of a given element in average time $\Theta(\log n)$

Ordered list of all elements of a BST

To enumerate the elements of a BST in order, we have to traverse the tree in inorder:

```
INORDER-TRAVERSAL(x)
    if x ≠ NUL then
        INORDER-TRAVERSAL(left(x))
        write key(x)
        INORDER-TRAVERSAL(right(x))
```

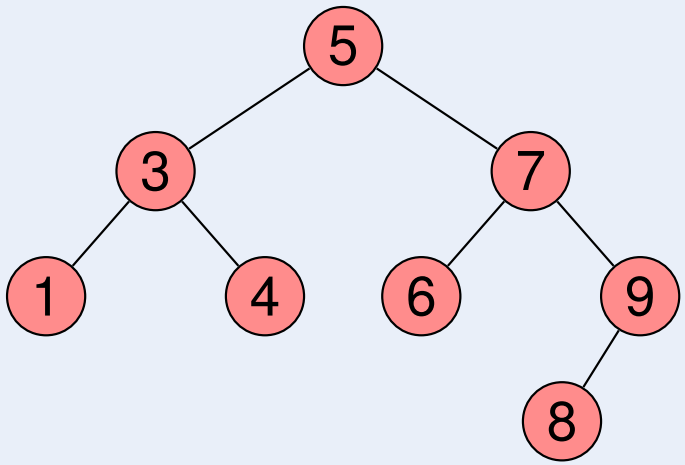
The inorder traversal of the following BST is: 1, 3, 4, 5, 6, 7, 8, 9.

Ordered list of all elements of a BST

To enumerate the elements of a BST in order, we have to traverse the tree in inorder:

```
INORDER-TRAVERSAL(x)
    if x ≠ NUL then
        INORDER-TRAVERSAL(left(x))
        write key(x)
        INORDER-TRAVERSAL(right(x))
```

The inorder traversal of the following BST is: 1, 3, 4, 5, 6, 7, 8, 9.



Implementation (BST)

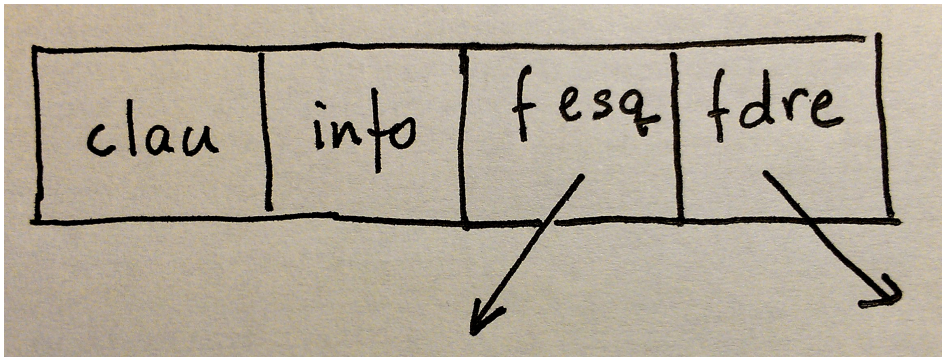
Exercise

Prove that a BST with n nodes can be restructured in such a way that has height $\lceil \log n \rceil$.

Implementation (BST)

Implementation: *Algorismes en C++ (EDA)*, J. Petit, S. Roura, A. Atserias.

Representation:



- A node is:
- A dictionary implemented as a BST has two fields:
  - the number of nodes of the BST
  - a pointer to the root of the BST

The costs given in the following:

- refer to the worst-case
- depend on n, which is the field with the same name in the dictionary (number of elements).

Implementation (BST)

Definition of `Dictionary` and `Node`

A `Node` stores a key, its information and two pointers to two other `Node`.

```
template <typename Key, typename Info>
class Dictionary {
private:
struct Node {
  Key key;
  Info info;
  Node* left; // Punter al fill esquerre
  Node* right; // Punter al fill dret
  Node (const Key& k, const Info& i, Node* l, Node* r)
  : key(k), info(i), left(l), right(r) { }
};

int n;          // Number of elements in the BST
Node* root;     // Pointer to the root of the BST
```

Implementation (BST): public functions

Constructors of creation and copy / Destructor

Create Dictionary: $\Theta(1)$.

```
Dictionary () {
  n = 0;
  root = null;
}
```

Make a copy: $\Theta(n)$.

```
Dictionary (const Dictionary& d) {
  n = d.n;
  root = copy(d.root);
}
```

Destructor: $\Theta(n)$.

```
~Dictionary () {
  free_dic(root);
}
```

Implementation (BST): public functions

Constructor of assignment

Redefinition of assignment: $\Theta(n + d.n)$.

```
Dictionary& operator= (const Dictionary& d) {
  if (&d != this) {
    free_dic(root);
    n = d.n;
    root = copy(d.root);
  }
  return *this;
}
```

Implementation (BST): public functions

Assign and delete

Assign to `key` the value `info`: $\Theta(n)$.

```
void assign (const Key& key, const Info& info) {
  assign(root, key, info);
}
```

Delete `key` and its information (if `key` does not exist, do nothing): $\Theta(n)$.

```
void delete (const Key& key) {
  delete_3(root, key);
}
```

Implementation (BST): public functions

Consulting

Given a `key`, return the reference to its information: $\Theta(n)$.

```
Info& consult (const Key& key) {
  if (Node* p = search(root, key)) {
    return p->info;
  } else {
      throw ErrorPrec("Key was not present");
} }
```

Determine whether `key` is present or not: $\Theta(n)$.

```
bool present (const Key& key) {
  return search(root, key) != null;
}
```

Return the dictionary size: $\Theta(1)$.

```
int size () {
  return n; }
```

Implementation (BST): private functions

Let us assume that the tree to consider (pointed by $\mathtt{p}$) has

- s nodes and
- height h.

Worst-case costs are given by $\Theta(s)$ o $\Theta(h)$.

Implementation (BST): private functions

Delete

Delete the tree pointed by `p`: $\Theta(s)$.

```
static void free_dic (Node* p) {
  if (p) {
    free_dic(p->esq);
    free_dic(p->right);
    delete p;
} }
```

Copy

Return a pointer to a copy of the tree pointed by `p`: $\Theta(s)$.

```
static Node* copy (Node* p) {
  return p ? new Node(p->key, p->info,
                      copy(p->left), copy(p->right))
           : null;
}
```

Implementation (BST): private functions

Search

Return a pointer to the node of the tree pointed by `p` that contains `key` (or `null` if it does not exist): $\Theta(h)$.

```
static Node* search (Node* p, const Key& key) {
  if (p) {
    if (key < p->key) {
      return search(p->left, key);
    } else if (key > p->key) {
      return search(p->right, key);
    }}
  return p;
}
```

Search is done by going down the adequate subtrees using the BST property.

Implementation (BST): private functions

Assign

Assign `info` to `key` if key is in the subtree pointed by `p`; if it is not there, add a new node with `key` and `info`: $\Theta(h)$.

```
void assign
       (Node*& p, const Key& key, const Info& info) {
  if (p) {
    if (key < p->key) {
      assign(p->left, key, info);
    } else if (key > p->key) {
      assign(p->right, key, info);
    } else {
      p->info = info;
    }
  } else {
    p = new Node(key, info, 0, 0);
    ++n;
} }
```

Implementation (BST): private functions

Minimum

Return a pointer to the node containing the minimum value in the subtree pointed by `p` (assuming that `p` is not null): $\Theta(h)$.

```
static Node* minimum (Node* p) {
  return p->left ? minimum(p->left) : p;
}
```

Maximum

Return a pointer to the node containing the maximum value in the subtree pointed by `p` (assuming that `p` is not null): $\Theta(h)$.

```
static Node* maximum (Node* p) {
  while (p->right) p = p->right;
  return p;
}
```

Implementation (BST): private functions

Delete (1)

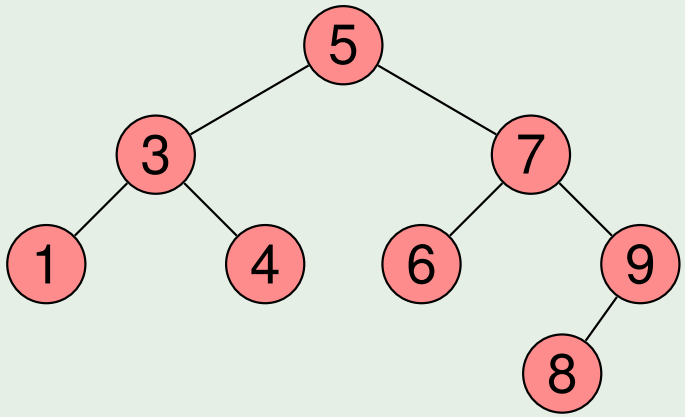
Deletes the node containing `key` in the subtree pointed by `p`: $\Theta(h)$.
The left subtree becomes the new left child of the minimum of the right child.

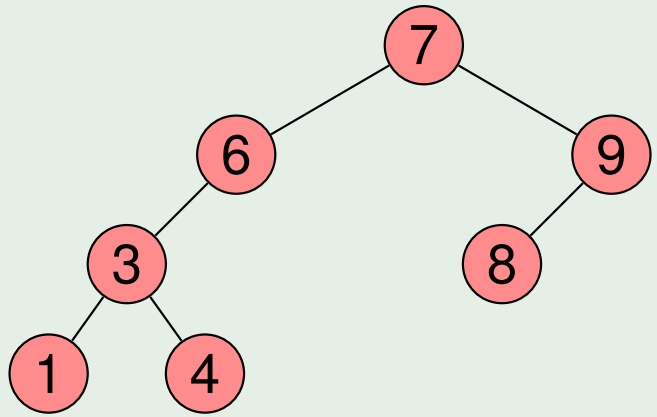
```
void delete_1 (Node*& p, const Key& key) {
  if (p) {
    if (key<p->key) {
      delete_1(p->left, key);
    } else if (key>p->key) {
      delete_1(p->right, key);
    } else {
      Node* q = p;
      if (!p->left) p = p->right;
        else if (!p->right) p = p->left;
      else {
         Node* m = minimum(p->right);
         m->left = p->left;
         p = p->right;}
      delete q; --n;
```

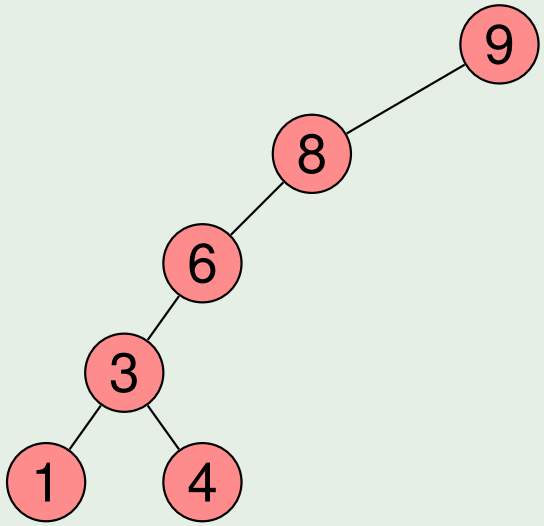
Implementation (BST): private functions

Example

Removal of 5 and 7. Trees get degraded very quickly.







Implementation (BST): private functions

Delete (2)

Deletes the node containing `key` in the subtree pointed by `p`: $\Theta(h)$.
The minimum of the right child is copied to the deleted node and is later removed.

```
void delete_2 (Node*& p, const Key& key) {
  if (p) {
  if (key < p->key) {
    delete_2(p->left, key);
  } else if (key > p->key) {
    delete_2(p->right, key);
```

Implementation (BST): private functions

Delete (2), cont.

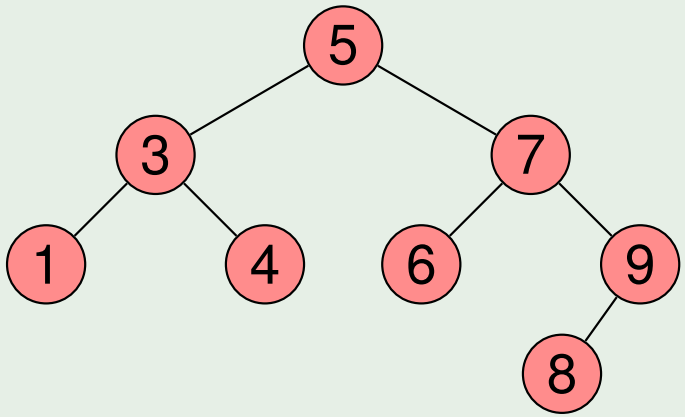
```
  } else if (!p->left) {
    Node* q = p; p = p->right;
    delete q; --n;
  } else if (!p->right) {
    Node* q = p; p = p->left;
    delete q; --n;
  } else {
    Node* m = minimum(p->right);
    p->key = m->key; p->info = m->info;
    delete_2(p->right, m->key);
} } }
```

Disadvantage: keys and informations are copied, more costly than copying pointers.

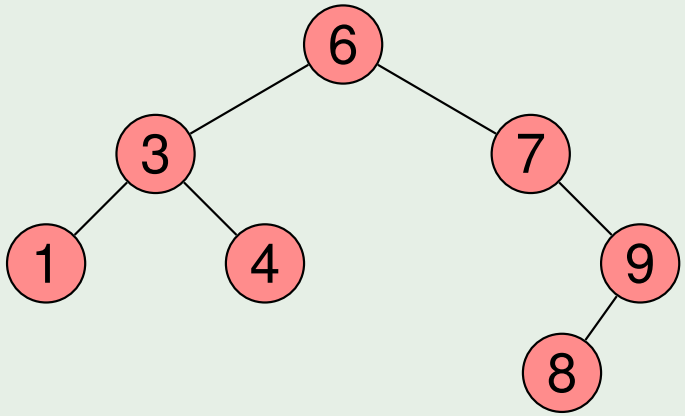
Implementation (BST): private functions

Example

Given the tree



we remove the root



Implementation (BST): private functions

Delete (3)

Deletes the node containing `key` in the subtree pointed by `p`: $\Theta(h)$.
The minimum of the right child will be removed and becomes the new root.

```
void delete_3 (Node*& p, const Key& key) {
  if (p) {
    if (key<p->key) {
      delete_3(p->left, key);
    } else if (key>p->key) {
      delete_3(p->right, key);
    } else {
      Node* q = p;
      if (!p->left) p = p->right;
      else if (!p->right) p = p->left;
           else {Node* m = delete_minimum(p->right);
                 m->left = p->left; m->right = p->right;
                 p = m;}
      delete q; --n; } } }
```

Implementation (BST): private functions

Delete minimum

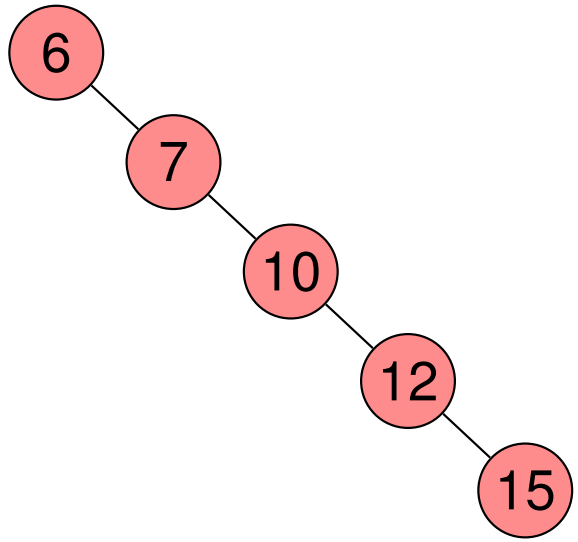
Deletes and returns the node containing the minimum element of `p`: $\Theta(h)$.

```
Node* delete_minimum (Node*& p) {
  if (p->left) {
    return delete_minimum(p->left);
  } else {
    Node* q = p;
    p = p->right;
    return q;
} }
```

Chapter 3. Dictionaries

Introduction

In BST, we have seen that the cost of operations is $\Theta(h)$, where h is the maximum height. But h can be as much as the number of nodes.

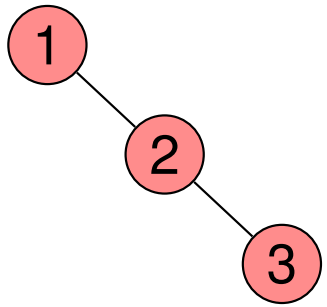


Hence, worst-case cost is $\Theta(n)$, where n is the number of nodes.

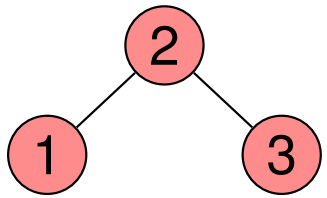
Introduction

In order to improve the cost we can do two things:

- prove that if elements are randomly inserted in a BST, the height is $\approx \log n$.
- perform insertions and deletions in such a way that the height is $\approx \log n$. Instead of having



we would have



Introduction

How do we keep the subtrees balanced?

1. Forcing the BST to be complete.
   A complete tree is one in which all levels are full except, maybe, for the last one, where the nodes are as much to the left as possible.

   But adding an element would be too costly.

Example

In order to add element 1 in the first tree, almost everything has to be changed.

Introduction

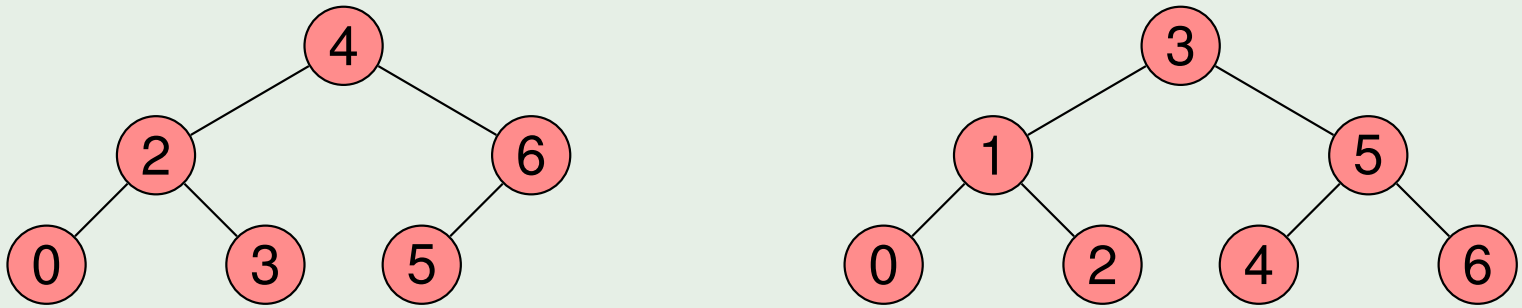
How do we keep the subtrees balanced?

1. Forcing the BST to be complete.
   A complete tree is one in which all levels are full except, maybe, for the last one, where the nodes are as much to the left as possible.

   But adding an element would be too costly.

Example

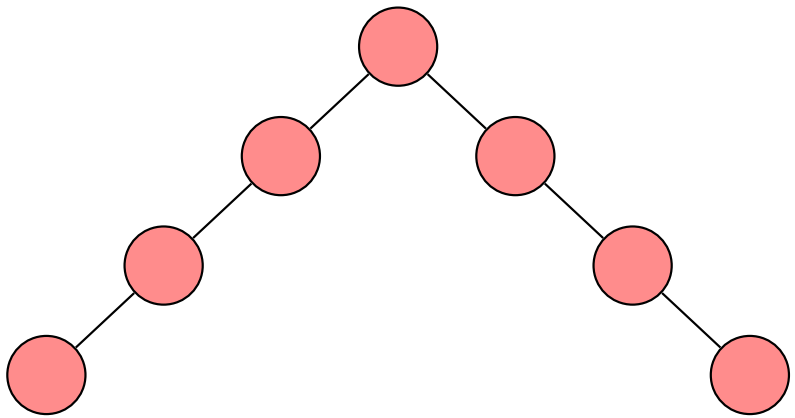
In order to add element 1 in the first tree, almost everything has to be changed.



How to keep the subtrees balanced?

2 Forcing that the two subtrees have almost the same height.

But this is not enough: the cost of basic operations in the following tree is $\approx n/2$.



Introduction

How to keep the subtrees balanced?

3. Allowing a small difference in the heights of the left and right subtrees: a maximum difference of 1.

   But this has to be done for all nodes!

Definition

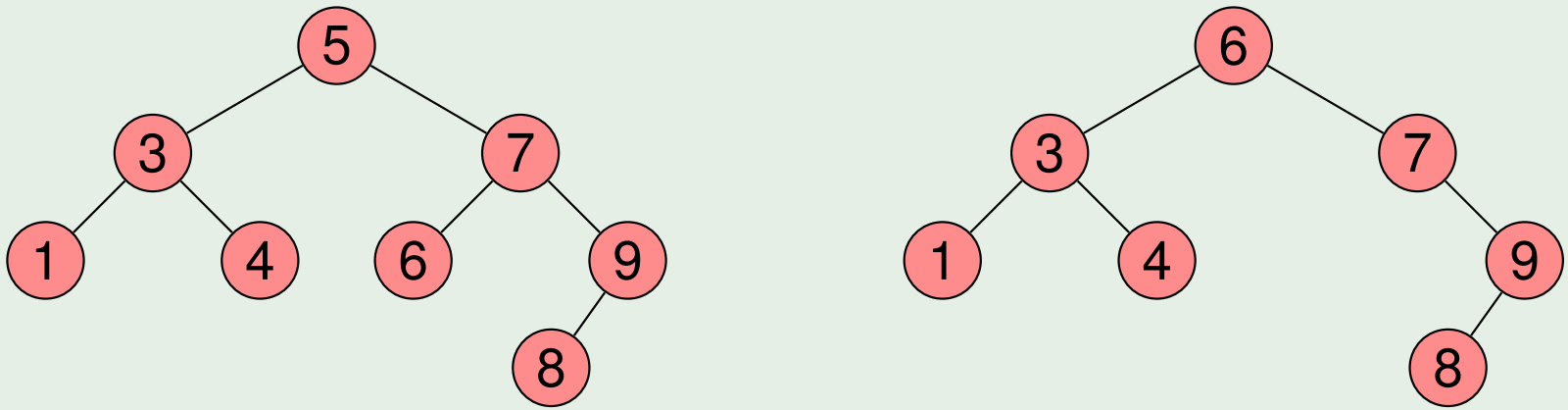
A binary tree is balanced if

- it is empty, or
- the difference of heights between its subtrees is at most 1, and its subtrees are also balanced.

The BST with the balanced condition are called AVL trees (Adelson-Velskii i Landis).

Example

The tree in the left is balanced, but if we remove the root as in BST, the resulting tree is no longer balanced.



Implementation (AVL)

Definition of `Dictionary` and `Node`

Similar to BST but adding the height.

```
template <typename Key, typename Info>
class Dictionary {
private:
struct Node {
  Key key;
  Info info;
  Node* left; // Pointer to the left child
  Node* right; // Pointer to the right child
  int hei; // Height of the tree
  Node (const Key& c, const Info& i,
            Node* l, Node* r, int h)
  : key(c), info(i), left(l), right (r), hei(h) {} };

int n;         // Number of elements in the AVL
Node* root; // Pointer to the AVL root
```

Implementation (AVL)

Functions **free_dic**, **copy** and **search** are as in BSTs.

In AVLs we will need two easy functions referring to size, both with cost $\Theta(1)$.

Return and update the height

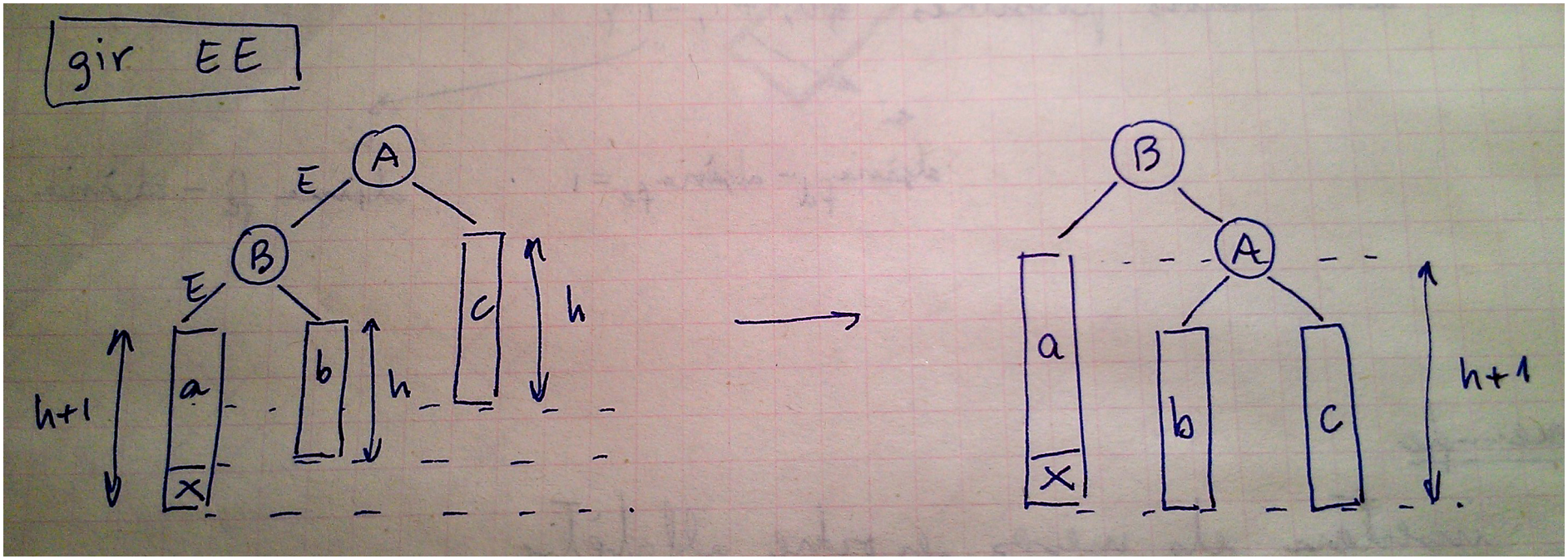
```
static int height (Node* p) {
    return p ? p->hei : -1;
}

static void update_height(Node* p) {
    p->hei = 1 + max(height(p->left), height(p->right));
}
```

Implementation (AVL)

In order to keep a tree balanced after an assignment or deletion, we consider four rotations of the tree: LL, RR, LR and RL.

All four have cost $\Theta(1)$.

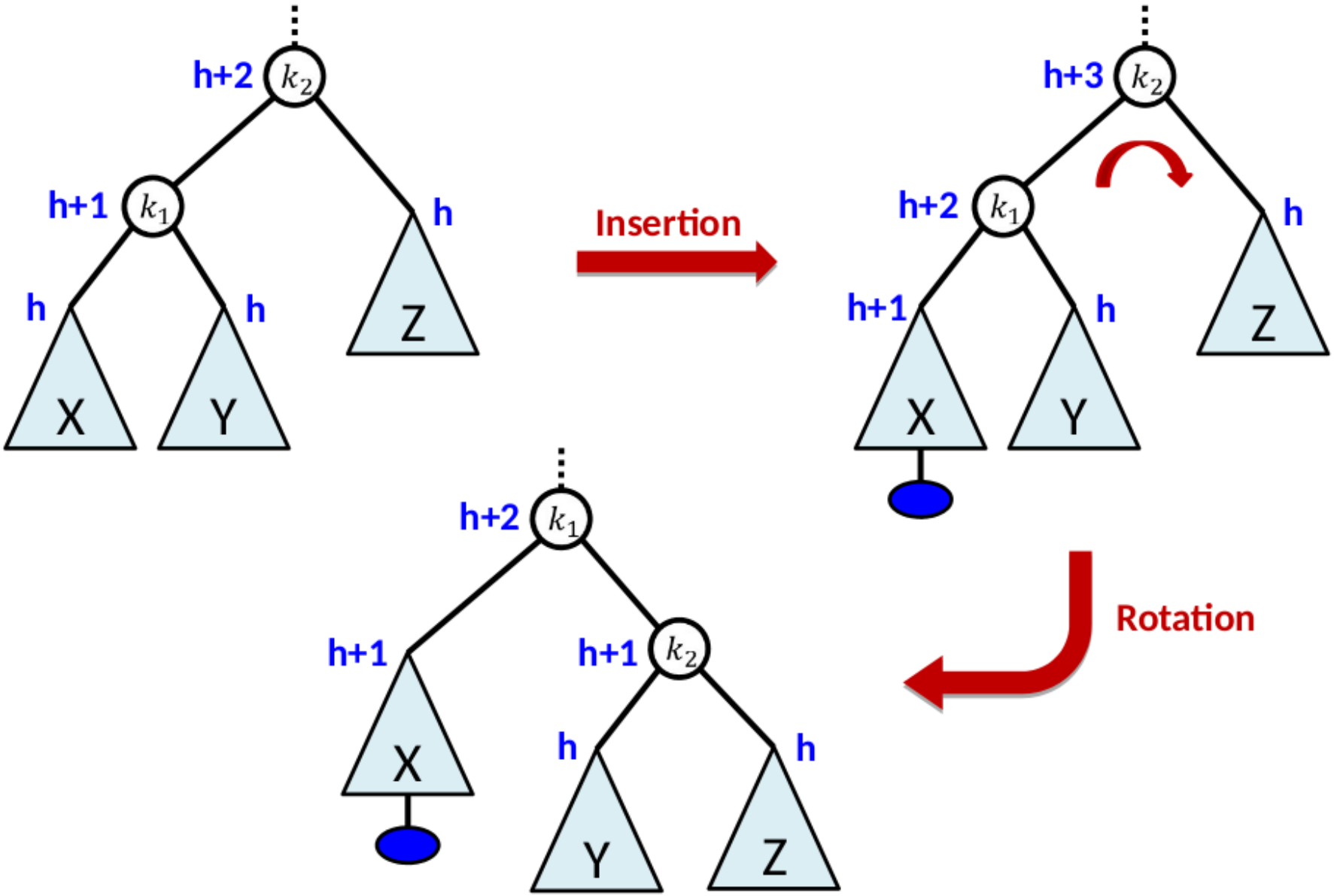
Implementation (AVL)



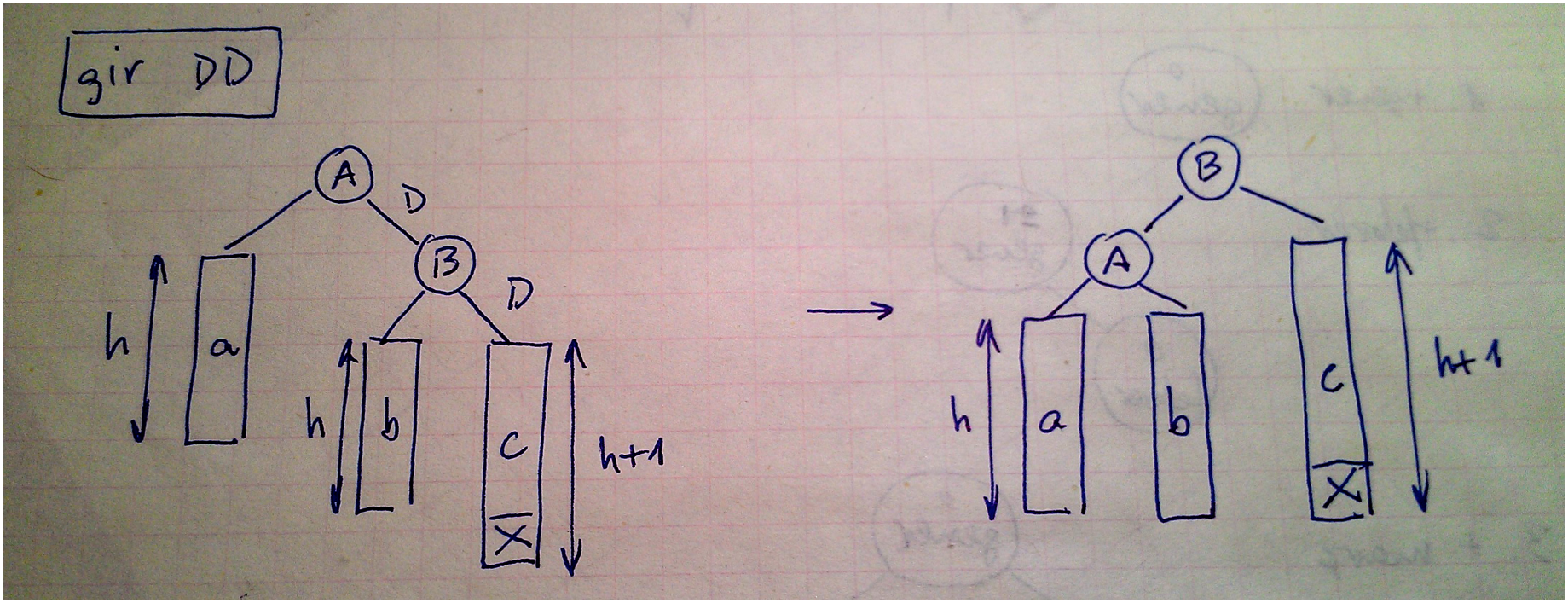
Rotation LL

```
static void LL (Node*& p) {
    Node* q = p;
    p = p->left;
    q->left = p->right;
    p->right = q;
    update_height(q);
    update_height(p); }
```

Implementation (AVL)



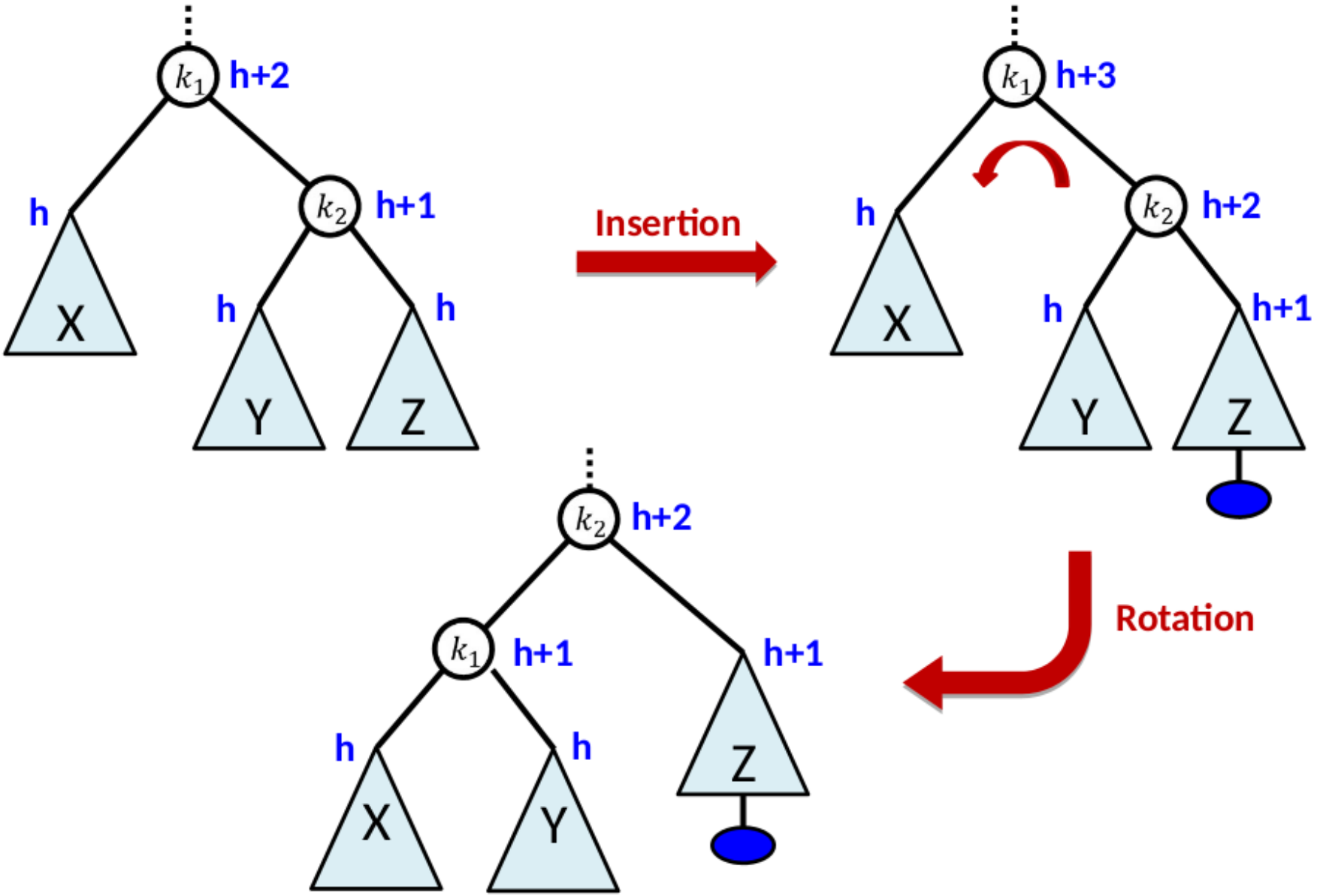
Implementation (AVL)



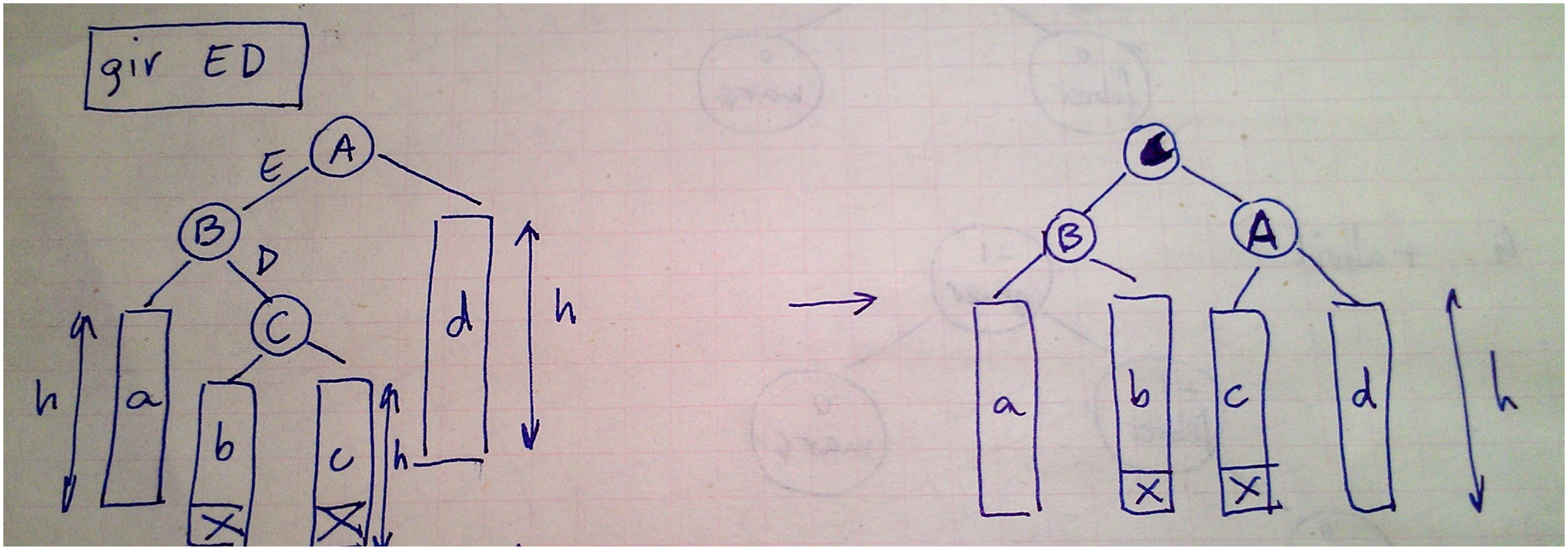
Rotation RR

```
static void RR (Node*& p) {
    Node* q = p;
    p = p->right;
    q->right = p->left;
    p->left = q;
    update_height(q);
    update_height(p); }
```

Implementation (AVL)



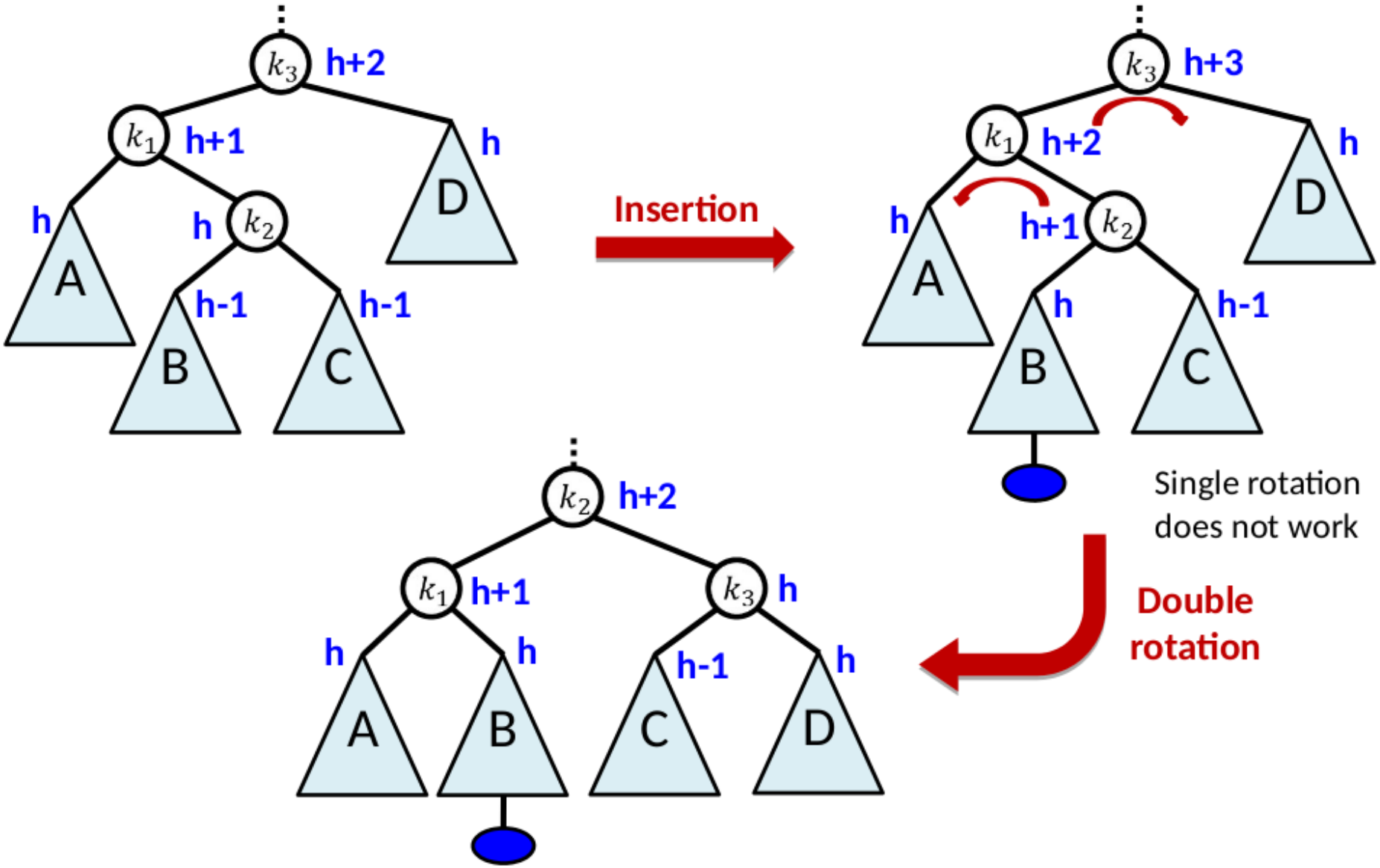
Implementation (AVL)



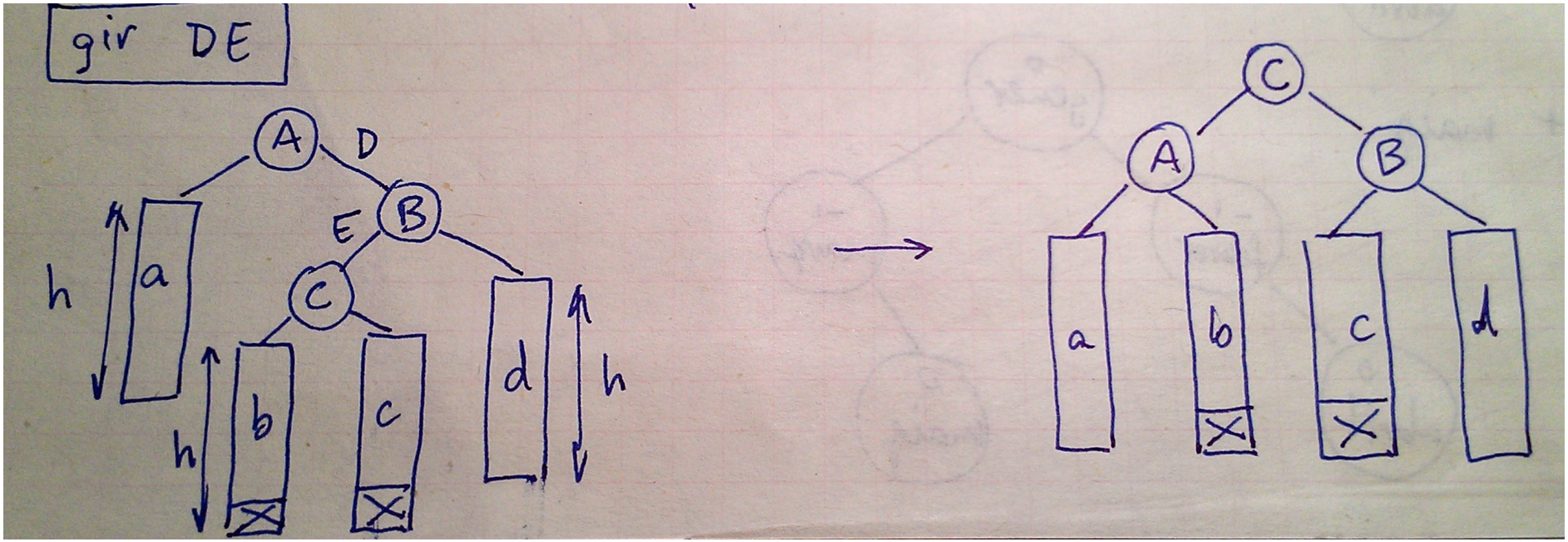
Rotation LR

```
static void LR (Node*& p) {
    RR(p->left);
    LL(p);
}
```

Implementation (AVL)



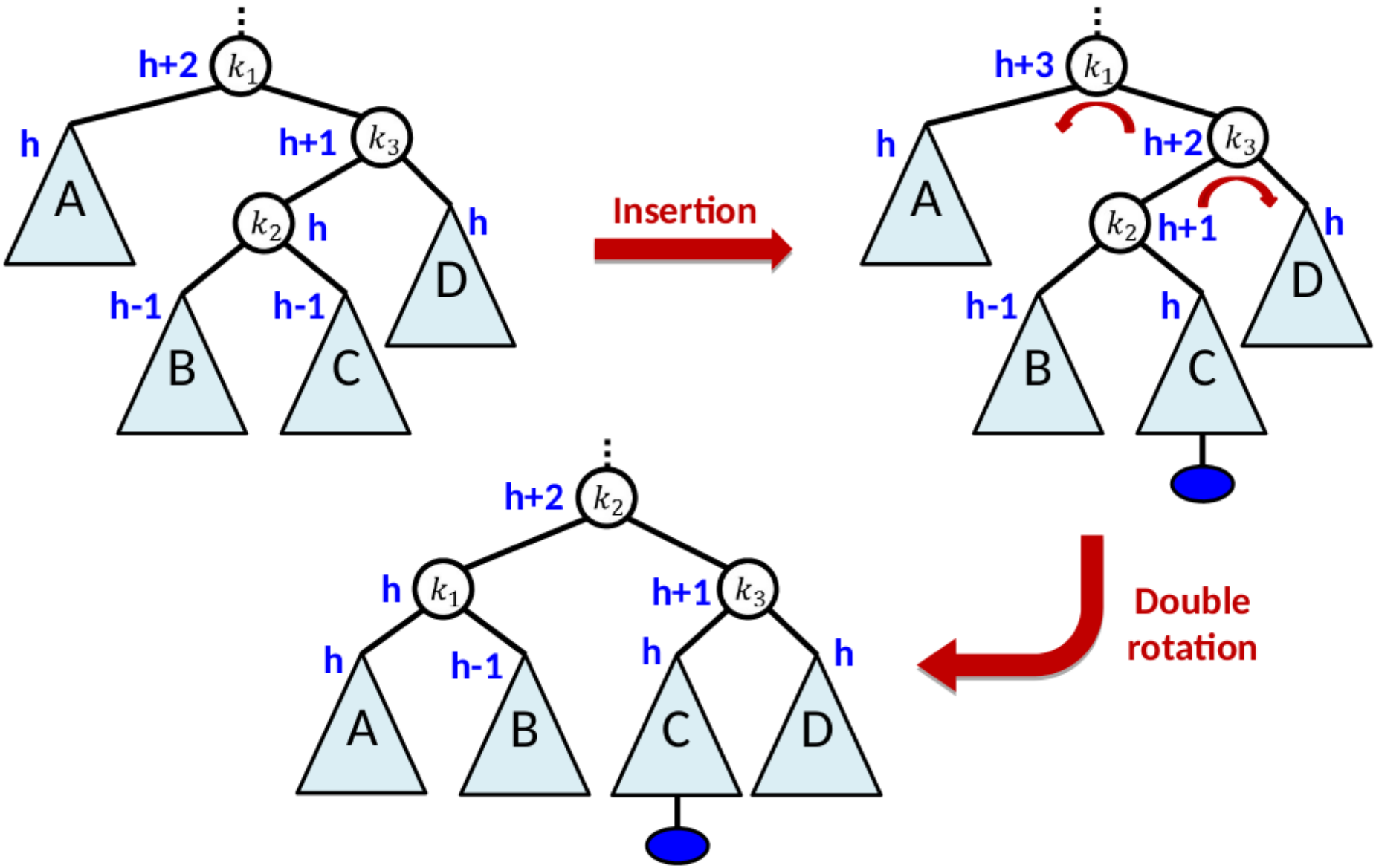
Implementation (AVL)



Rotation RL

```
static void RL (Node*& p) {
    LL(p->right);
    RR(p);
}
```

Implementation (AVL)



Implementation (AVL)

Assign

```
void assign (Node*& p, const Key& key, const Info& info) {
  if (p) {
     if (key < p->key) {
        assign(p->left, key, info);
        if (height(p->left)-height(p->right) == 2) {
           if (key < p->left->key) LL(p);
           else LR(p); }
        update_height(p);
     } else if (key > p->key) {
        assign(p->right, key, info);
        if (height(p->right)-height(p->left) == 2) {
           if (key > p->right->key) RR(p);
           else RL(p); }
        update_height(p);
     } else p->info = info;
  } else {
     p = new Node(key, info, nullptr, nullptr, 0);
     ++n; }}
```

Implementation (AVL)

On average, a rotation is done every two assignments.

Operations **assign**, **delete** use rotations and have worst-case cost $\Theta(\log n)$.

Operation **consult**, as in BST, has worst-case cost $\Theta(\log n)$.