Unit 3: Search Algorithms

3.1 Basic search algorithms

Known results from our previous analysis

- Linear search
  - $W_{LSearch}(N) = N$ with basic operation the KC
  - $A^s_{LSearch}(N) = \sum_{i=1}^{N} n_{LSearch}(k = T[i])\, p(k == T[i]) \sim \frac{S_N}{C_N}$
- Binary search

$$W_{BSearch}(N) = \lceil \lg(N) \rceil = \lg(N) + O(1) = A^u_{BSearch}(N)$$

- We now have to calculate $A^s_{BSearch}(N)$

Average case of a successful Bsearch

- Let us consider an example:

T=[1 2 3 4 5 6 7] N=7=2^3-1

$$A^s_{BSearch}(N)=\frac{1}{7}\sum_{i=1}^{7} n_{BSearch}(k=T[i])=\frac{1}{7}(\underset{4}{1}+\underset{2}{2}+\underset{6}{2}+\underset{1}{3}+\underset{3}{3}+\underset{5}{3}+\underset{7}{3})$$

$$\Rightarrow A^s_{BSearch}(N)=\frac{1}{7}(1+2\cdot 2+3\cdot 4)=\frac{1}{7}(1\cdot 2^0+2\cdot 2^1+3\cdot 2^2)$$

- For N=2^k-1 we have:

Obs: N=2^k-1⇒k≈log(N)

$$A^s_{BSearch}(N)=\frac{1}{N}\sum_{i=1}^{k} i2^{i-1}=\frac{1}{N}[k2^k-2^k+1]\Rightarrow A^s_{BSearch}(N)=\frac{1}{N}[N\lg(N)-N+1]\Rightarrow$$

$$A^s_{BSearch}(N)=\lg(N)-1+\frac{1}{N}\Rightarrow \boxed{A^s_{Bsearch}(N)=\lg(N)+O(1)}$$

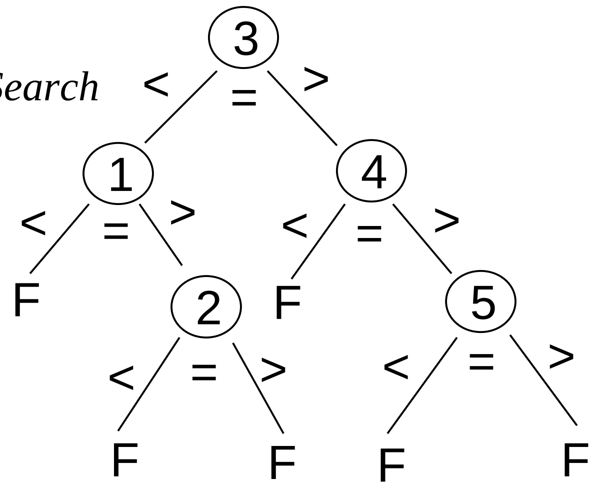
Decision trees for comparison based search algorithms. Definition:

- If **A** is a key comparison based search algorithm and **N** is the size of its input array, we can build its **decision tree T_A^N** for inputs $\sigma \in \Sigma_N$ with the following 5 conditions:

1. The tree contains nodes in the form **i** that indicates the key comparison between the i-th element of the array and a generic key k.
2. If k coincides with the i-th element in the array (T[i]==k) then the search of key k ends in node **i**
3. The left subtree of node **i** in $\mathbf{T_A^N}$ contains the key comparisons that algorithm A performs if $k < T[\mathbf{i}]$.
4. The right subtree of node **i** in $\mathbf{T_A^N}$ contains the key comparisons that algorithm A performs if $k > T[\mathbf{i}]$.
5. The leaves L_σ in $\mathbf{T_A^N}$ represent the evolution of failing searches.
6. The nodes represent those of successful searches.

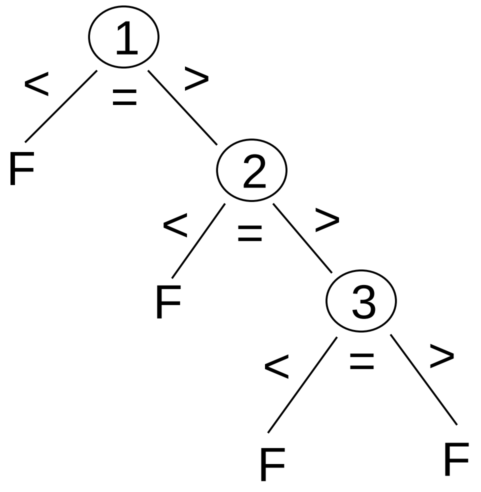
Search decision trees: Examples

$T^5_{BSearch}$



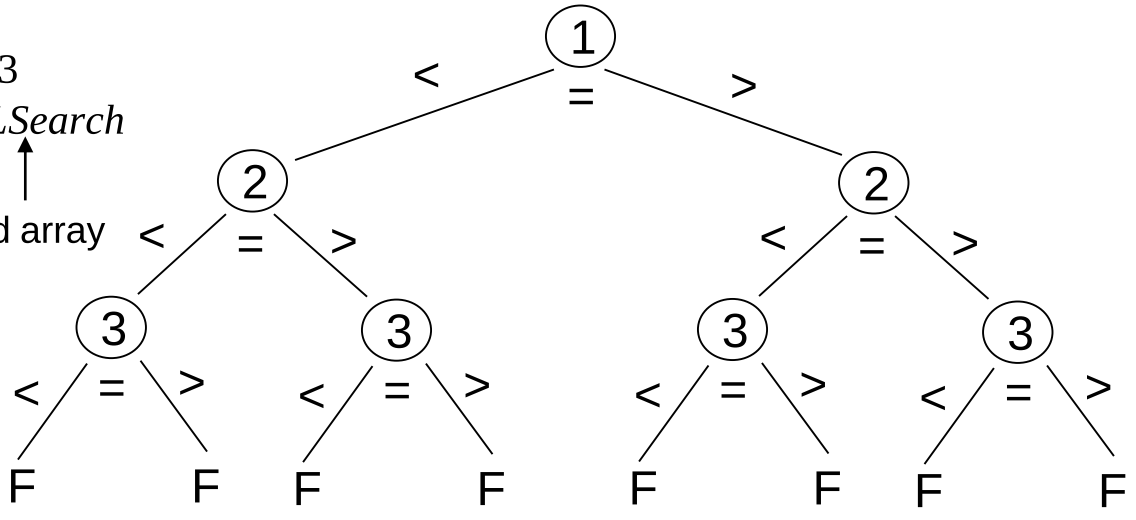
$T^3_{SortedLsearch}$

Sorted array



$T^3_{LSearch}$

Unsorted array



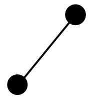
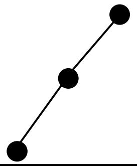
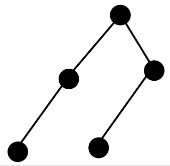
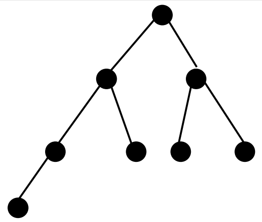
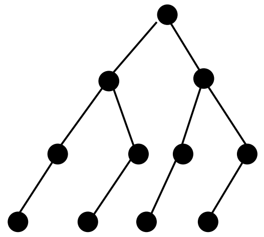
Obs: T_A^N is a binary tree with at least N internal nodes. This yields the lower bound:

$$\boxed{W_A(N) \geq \text{height}_{min}(N)} \longleftarrow$$

Minimum height of a tree with at least N internal nodes

Decision trees: worst case lower bound

- We can estimate $\text{Height}_{min}(N)$

N	T	$\text{Height}_{min}(N)$
1		1
2		2
3		2
4		3
7		3

Comparison search lower bounds

- We have that

  $W_A(N) \geq Height_{min}(N) = \lfloor lg(N) \rfloor + 1 \Rightarrow$

  $\boxed{W_A(N)=\Omega(lg(N))}$ $\forall$ $A \in \mathcal{S}$ with

  $\mathcal{S}$={A: key comparison based search}
- BSearch is **optimal** for the **worst case.**
- We can also demonstrate that

  $\boxed{A_A(N)=\Omega(lg(N))}$ $\forall$ $A \in \mathcal{S}$
- BSearch is **optimal** for the **average case**.

Is that all?

- **Observation:** search processes are not isolated processes.
- Elements are not only searched but also **inserted** or **removed.**
- It is not only important how to search but also where to search.
- Context: **Dictionary Abstract Data Type (ADT).**

In this section we have…

- reminded the worst and average costs of linear and binary searches.
- learnt the concept of decision trees for key comparison based searches.
- learnt to build key comparison based search decision trees.
- analyzed that binary search is optimal in the worst and average cases for key comparison based search algorithms.

3.2 Search on dictionaries

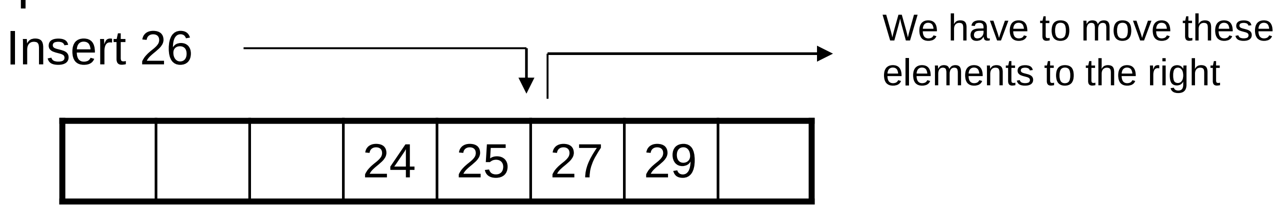
Dictionary Abstract Data Type (ADT)

- **Dictionary:** a sorted set of data that supports the following operations:
  - pos Search(key k, dict D)
    - Returns the position of key **k** in dictionary **D** or an error code **ERR** if **k** is not in **D.**
  - status Insert (key k, dict D)
    - Inserts key **k** in dictionary D and returns **OK** or **ERR** if **k** could not be added in **D.**
  - void Remove (key k, dict D)
    - Deletes key **k** in dictionary D

Data structure for Dictionaries I

- What is the most adequate data structure for a dictionary?
- Option 1: Sorted array (|D|=N)
  - **Search:** Using BSearch => $n_{BSearch}(k,D)=O(\log(N))$ => optimal.
  - **Insert:** We should keep the array sorted => insertion is costly.

    Example:



- If we insert in position 1, we have to move N elements.
- In the average case we move N/2 elements.
- Thus, $n_{Insert}(k,D)=\Theta(N)$: **too bad !!!**

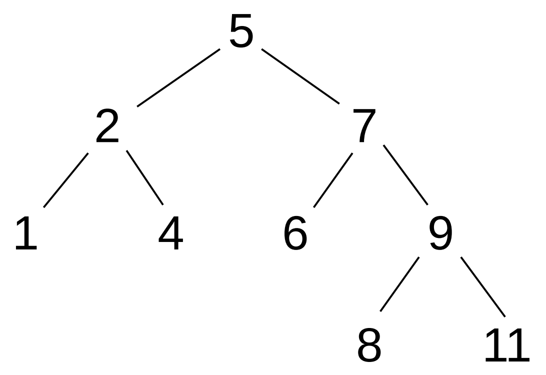
Data structure for Dictionaries II

- Option 2: **Binary Search Tree** (BST)
- **Definition: A BST** is a binary tree **T** in which for all nodes T'∈T, the following relation is met:

  $\boxed{\text{Info(T'')<Info(T')<Info(T''')}}$

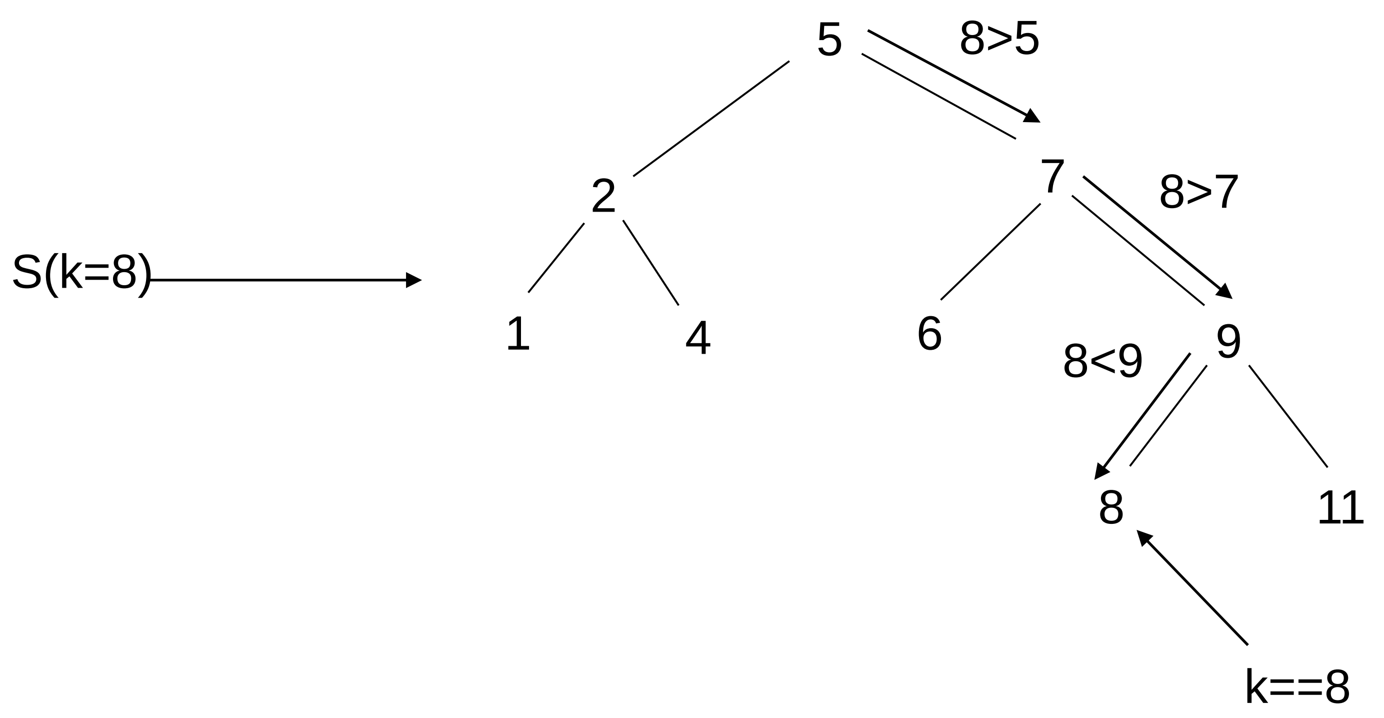
  for all nodes T'' at the left of T' and T''' at the right of T'
- Thus, **all nodes at the left** of T' have **lesser values** than info(T') and **all nodes at the right** of T' have **larger** values than info(T')

Example:



Searches on BST I

- Example:



Searches on BST II

- Pseudocode:

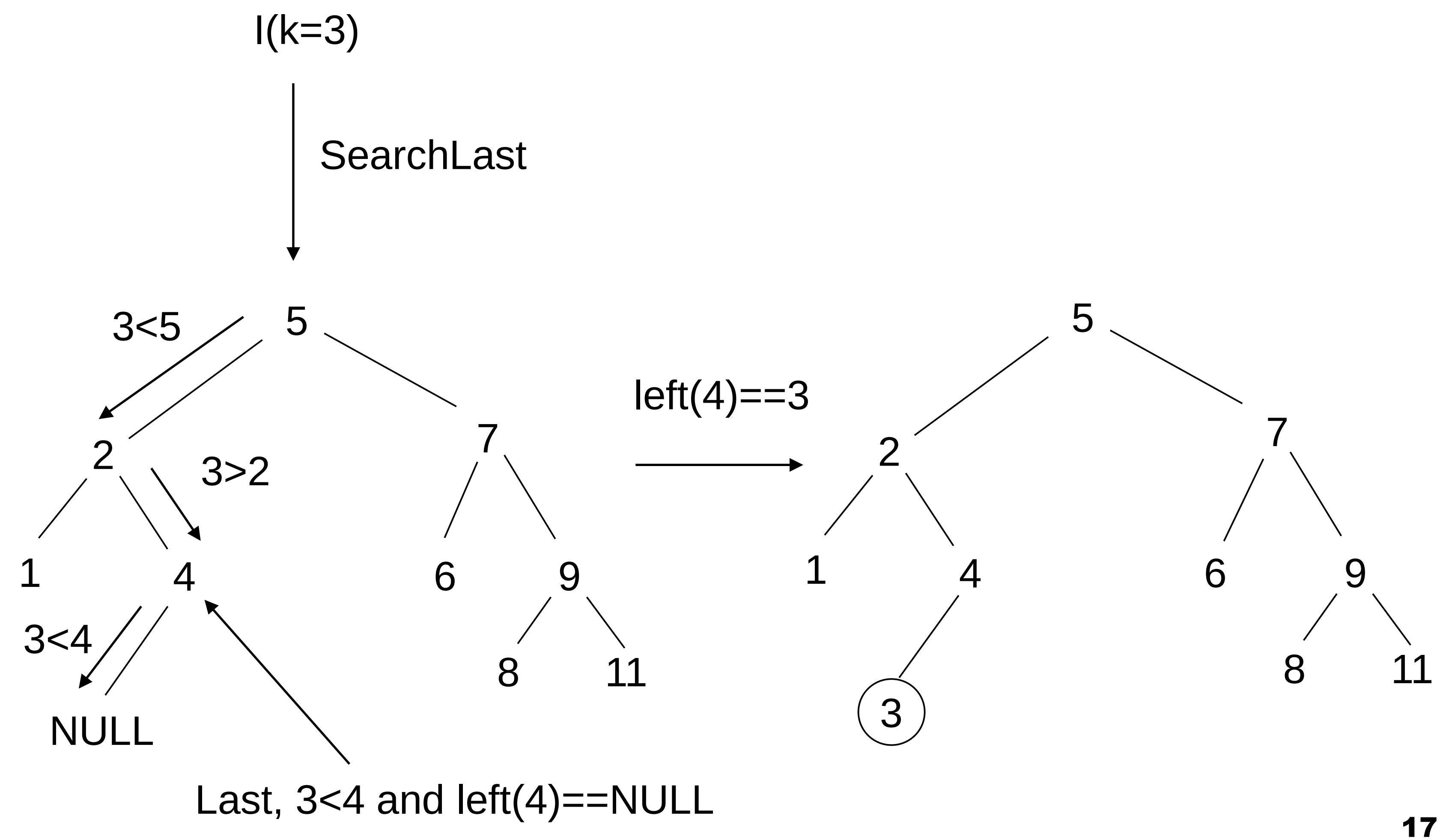
```
BT Search (key k, BT T)
   if T==NULL : return NULL;
   if info(T)==k : return T;
   if k<info(T) :
      return Search(k,left(T));
   if k>info(T) :
      return Search(k,right(T));
```

- Observation:

$$n_{\text{Search}}(k,T)=\text{height}(k, T) + 1 = O(\text{height}(T))$$

Insertion in BSTs I

- Example



Insertion in BSTs II

- Pseudocode

```
status Insert (key k, AB T)
    T'=SearchLast(k,T);
    T''=GetNode();
    if T''==NULL : return ERR;
    info(T'')=k;
    if k<info(T') :
        left(T')=T''
    else :
        right(T')=T'';
    return OK;
```

```
AB SearchLast(key k, AB T)
    if k == info(T): return NULL;
    if (k<info(T) and left(T) ==NULL) or
        (k>info(T) and right(T) ==NULL):
        return T;
    if k<info(T) and left(T) !=NULL :
        return SearchLast(k, left(T));
    si k>info(T) and right(T) !=NULL :
        return SearchLast(k, right(T));
```

- Observation

$$n_{Insert}(k,T)=n_{SearchLast}(k,T)+1 \Rightarrow n_{Insert}(k,T)=O(height(T))$$

Remove in BSTs

- Pseudocode:

```
void Remove (key k, AB T)
  T'=Search(k,T);
   if T'!=NULL :
      Remove&Readjust(T',T);
```

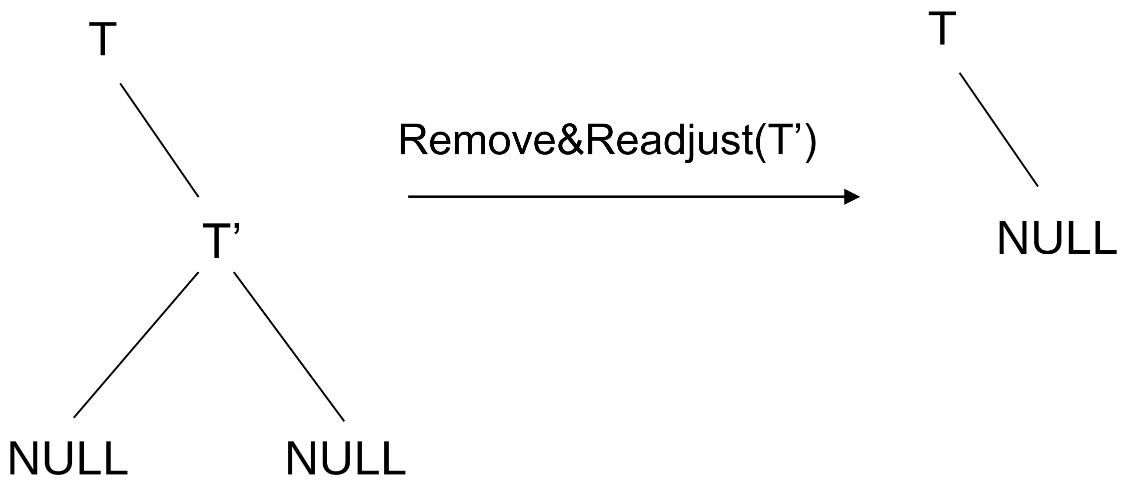
- Thus,

$$n_{Remove}(k,T)=n_{Search}(k,T)+n_{R\&R}(T',T)$$

- In **Remove&Readjust** there are three possible cases, depending on the number of children of the node T' to be removed.

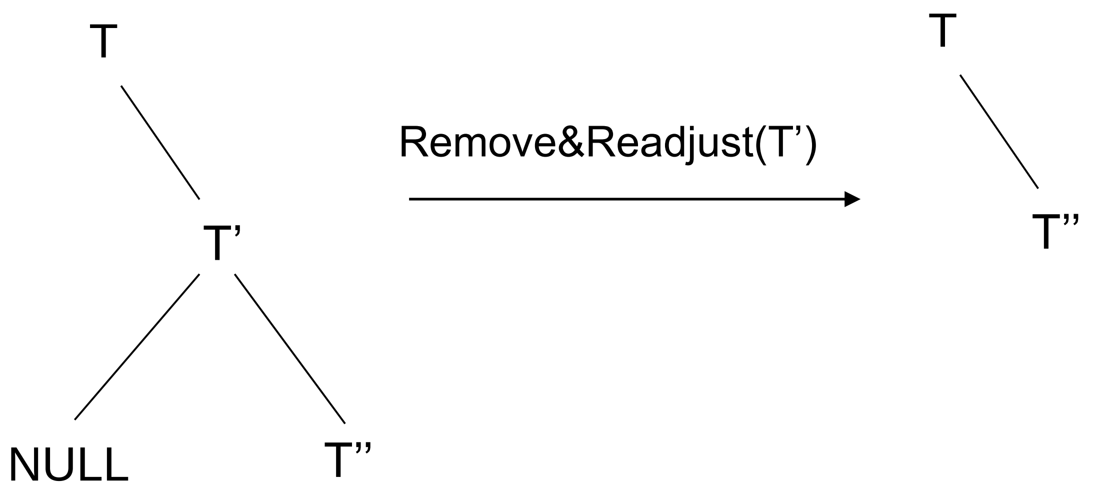
Remove & Readjust I

- **Case 1**: the node to be removed **has no children**
  - We free node T’ (free(T’)), and
  - The pointer of the parent of T’ that pointed to T’ is reassigned to NULL.
- Cost of Remove&Readjust = O(1)



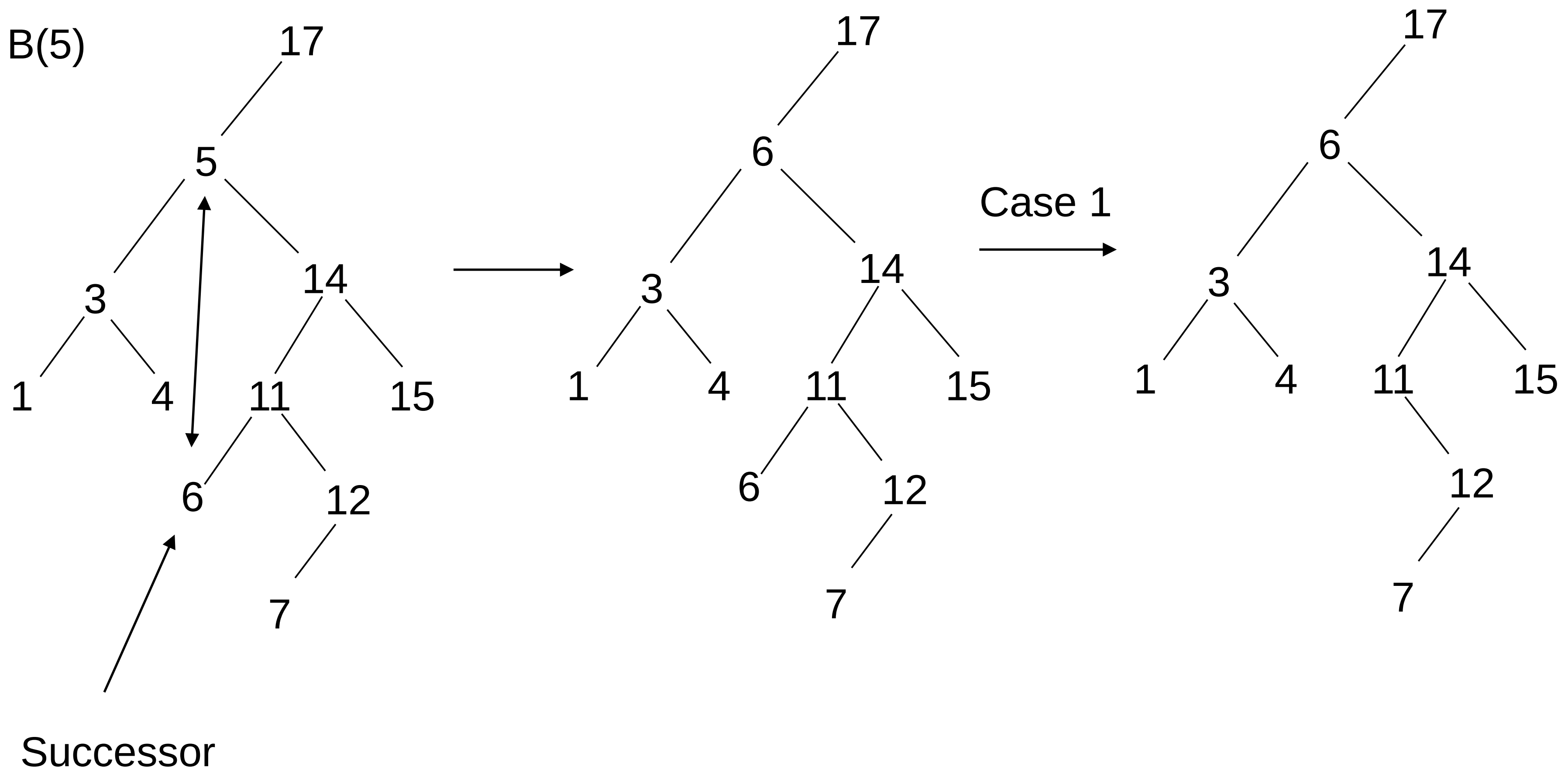
Remove & Readjust II

- Case 2: The node to be remove **has one single child**
  - The pointer of the parent of T' that pointed to T' is reassigned to the only child of T' and
  - we freeT'
- Cost of Remove&Readjust = O(1)



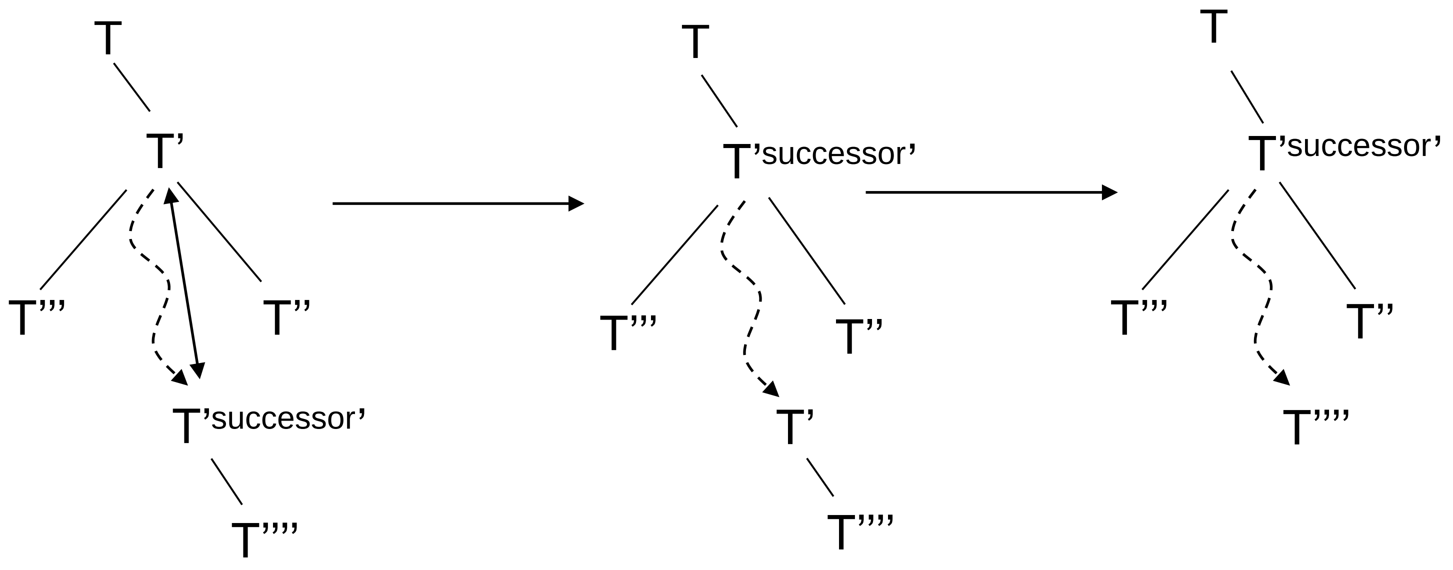
Remove & Readjust III

- Case 3: The node to be removed **has two children**



Remove & Readjust IIV

- When the node to be removed has **two children**:
  - T’ is replaced by the node that contains the **successor** (the next element in the sorted array), and
  - Node T’ is removed.
- Cost of Remove&Readjust ≤ height(T)



Find the successor in a BST

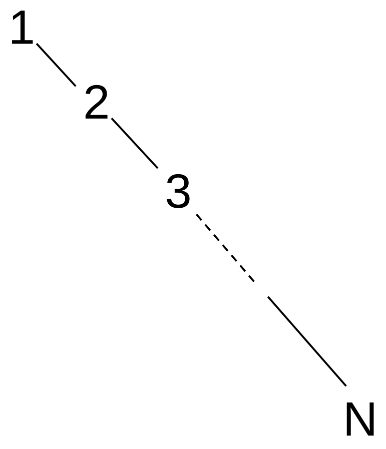
- Pseudocode

```
AB FindSuccessor(AB T')
   T''=right(T');
   while left(T'')!=NULL :
       T''=left(T'');
   return T'' ;
```

- **Observation:** If k' is the successor of k in a BST, then **left(k')==NULL**:
  - If left(k')==k'' then we would have k''<k'
  - but k''>k, since k'' is at the right of k
  - Then we have: k<k''<k' and
  - Thus, k' **cannot be the successor** of k.

Efficacy of the operations associated to searching in BSTs

- $n_{R\&R}(T',T)= n_{FindSuccessor}+n_{ReadjustPointers}=$
  $= O(height(T))+O(1) =O(height(T))$
- Thus,
  $n_{Borrar}(k,T)=\underbrace{O(height(T))}_{\text{Search}}+ \underbrace{O(height(T))}_{\text{R\&R}}= O(height(T))$
- BSTs ere adequate as long as height(T)= $\Theta(\lg(N))$
- But in some BSTs $W_{Search}(N)=N$: too bad !**!!**



Average case of searching in BSTs I

- $A^S_{Search}(N)$= average cost of **(1)** the search of all elements and **(2)** for all Tσ

$$A^S_{Search}(N) = \frac{1}{N!}\sum_{\sigma\in\Sigma_N} A^S_{Search}(T_\sigma) = \frac{1}{N!}\sum_{\sigma\in\Sigma_N}\frac{1}{N}\sum_{i=1}^{N} n_{Search}(\sigma(i), T_\sigma)$$

$$= \frac{1}{N!}\sum_{\sigma\in\Sigma_N}\frac{1}{N}\sum_{i=1}^{N}[height(\sigma(i))+1] = \frac{1}{N!}\sum_{\sigma\in\Sigma_N}[1+\frac{1}{N}\sum_{i=1}^{N} height(\sigma(i))]$$

$$= 1+\frac{1}{N}\times\frac{1}{N!}\sum_{\sigma\in\Sigma_N}\sum_{i=1}^{N} height(\sigma(i))] = 1+\frac{1}{N}\left(\frac{1}{N!}\sum_{\sigma\in\Sigma_N} n_{Create}(T_\sigma)\right)$$

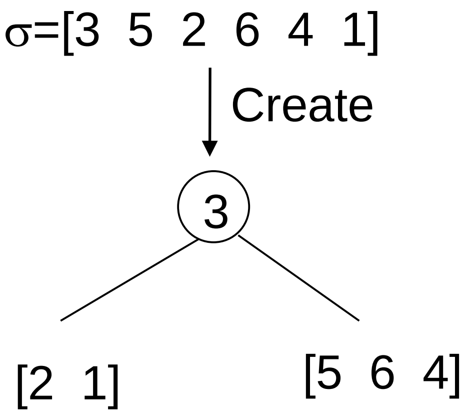
Thus,

$$\boxed{A^S_{Search}(N) = 1+\frac{1}{N}A_{Create}(N)}$$

Average case of searching in BSTs II

- Let us consider an **alternative** pseudocode for Create

```
AB Create (array σ)
   T=IniBT(σ);
   InsBT(σ(1),T);
   Distribute(σ, σ_l, σ_r);
   T_l=Create(σ_l);
   T_r=Create(σ_r);
   left(T)=T_l;
   right(T)=T_r;
   return T;
```



Similar case as QS:

$n_C(\sigma) = N-1+ n_C(\sigma_l)+ n_C(\sigma_r)$

$\Downarrow$

$$A_{Create}(N) = 2N\log(N) + O(N)$$

Thus:

$$A^s_{Search}(N) = 1 + \frac{1}{N} A_{Create}(N) = 1 + \frac{1}{N}\left[2N\log(N) + O(N)\right] = \Theta(\log(N))$$

Summary on search operations on BSTs

- If S is a key comparison based search algorithm:

$$W_S(N)=\Omega(\lg(N))$$

- If the ADT is a BST all search operations are efficient **on average.**
- If we could guarantee that for all $\sigma \in \Sigma_N$ we could build a BST so that height(T_σ)=$\Theta(\lg(N))$ then we would have

$$\mathbf{W_{Search}(N)= \Theta(lg(N))}$$

In this section we have…

- introduced the concept of dictionary and the operations associated with searching
- studied its implementation on BSTs
- shown that its costs is associated with the height of the BSTs
- shown that its implementation is optimal in the average case
- Shown that in the worst case the implementation has a cost of Θ(N)

Tools and techniques to work on

- The creation and use of Binary Search Trees (BSTs).
- Removing nodes in BSTs and finding the successor.
- Problems to solve (at least !!!): those recommended in section 11.

3.3 AVL Trees

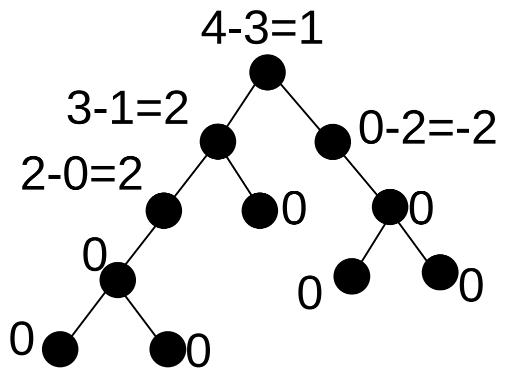
AVL trees (Adelson-Velskii-Landis)

- **Definition:** The balance factor of a node T in a BST is defined as:

$$\boxed{BF(T)=height(T_l)-height(T_r)}$$

T_l left subtree
T_r right subtree

- **Example:**



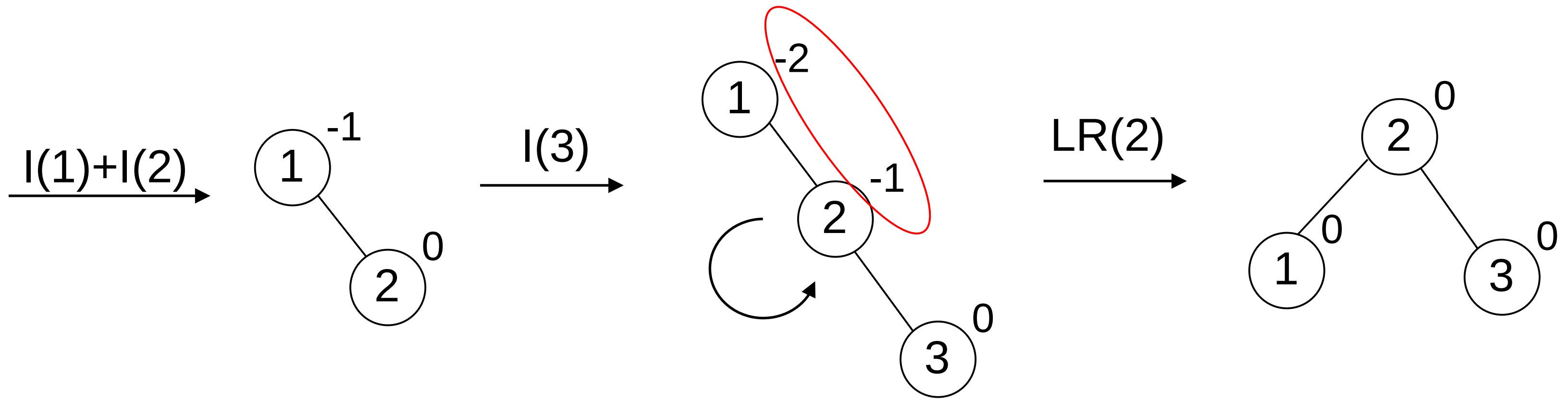
- **Definition:** An AVL tree T is a BST in which ∀ subtreee T' of T it holds that

$$\boxed{BF(T')=\{-1,0,1\}}$$

Construction of AVLs

- To build an AVL we follow these two steps:
  - Step 1: We perform the normal insertion of nodes in a BST.
  - Step 2: If necessary, we arrange the unbalance of the nodes (rebalancing), and we come back to step 1.

Example: T=[1 2 3 4 5 6 7 15 14 13 12 11 10 9 8]

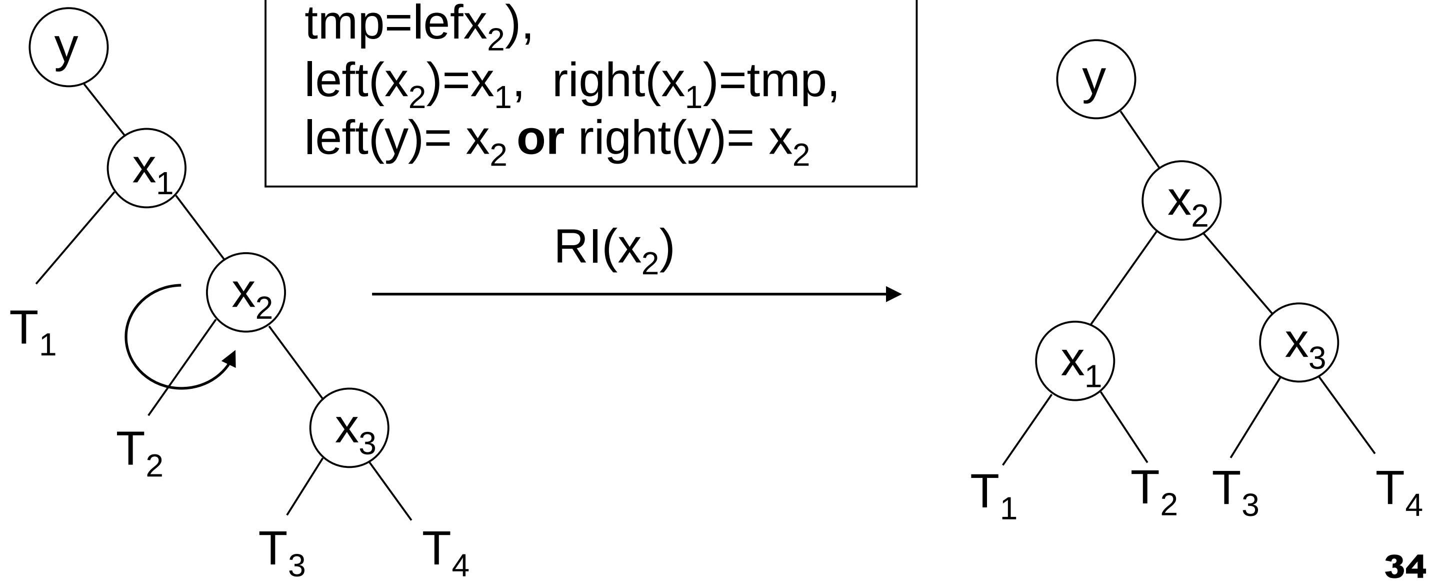


Building AVLs

- The operation that we just made is called **Left Rotation on the node with BF -1**, in this case at element 2.

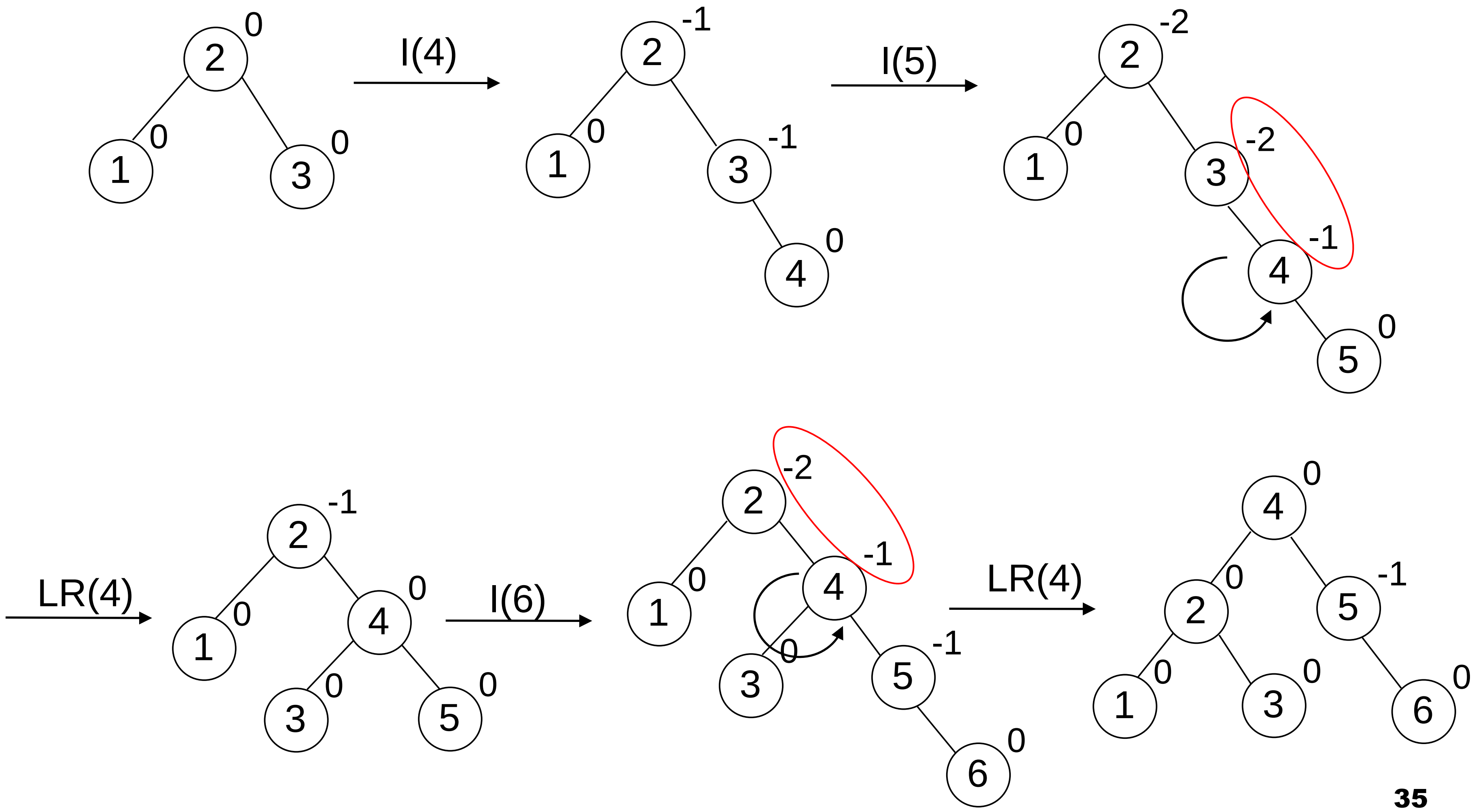
- The left rotation on the node with BF -1 corresponds to the following pointer reassigning:

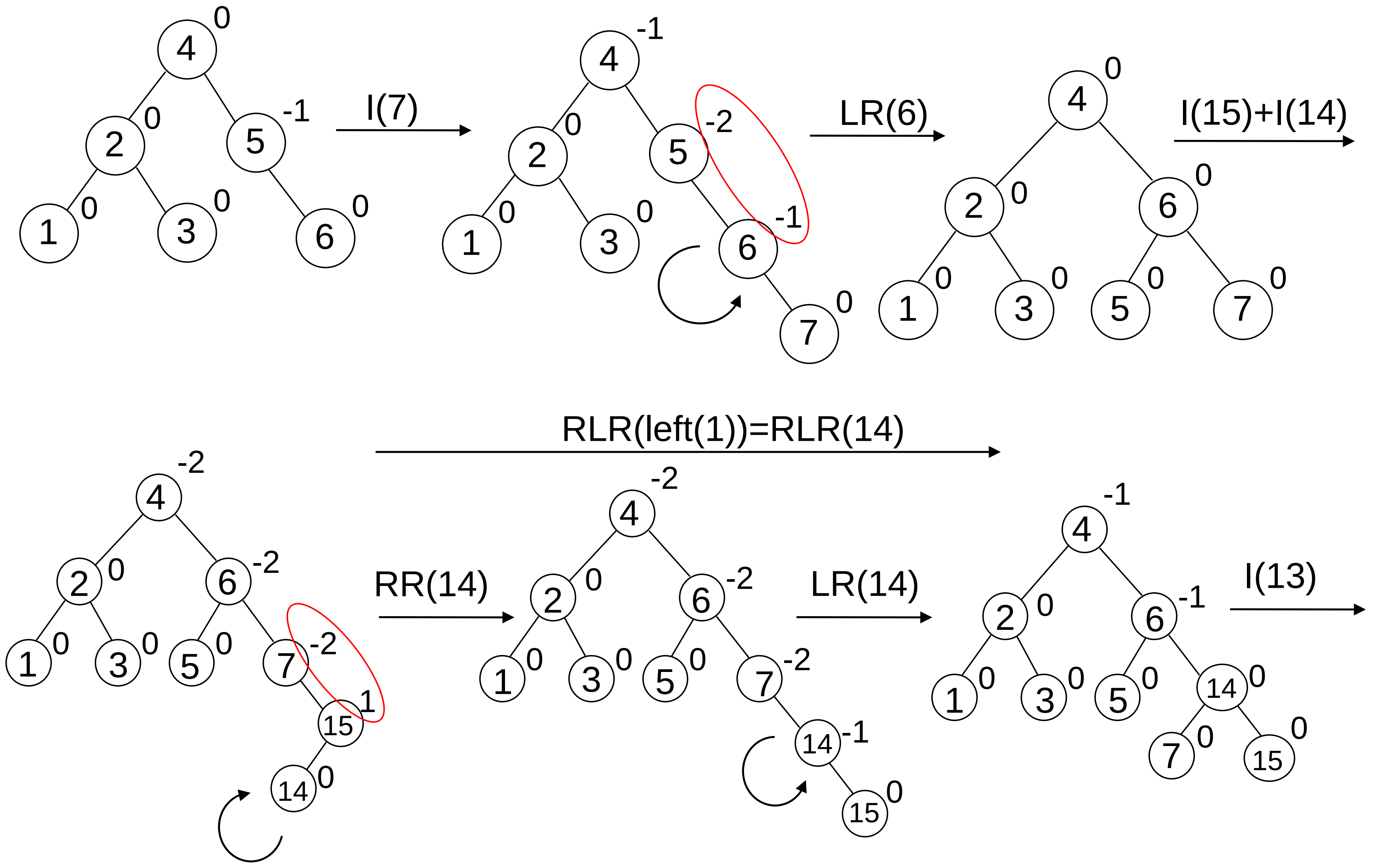
tmp=lefx_2),
left(x_2)=x_1, right(x_1)=tmp,
left(y)= x_2 **or** right(y)= x_2

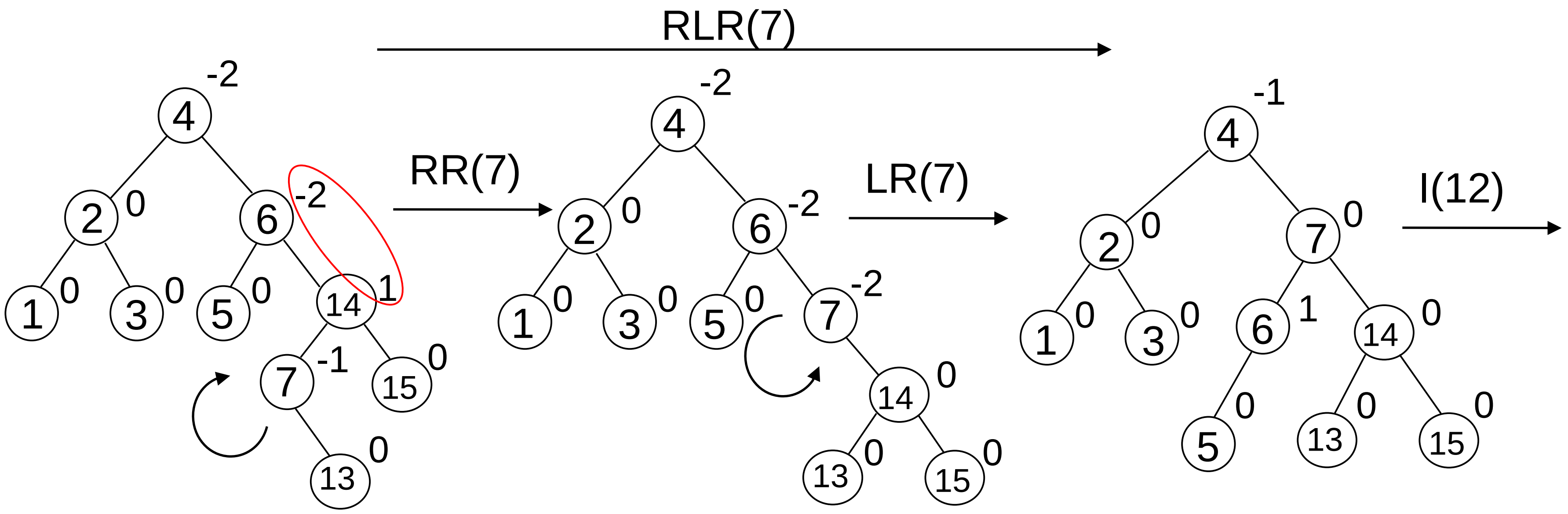


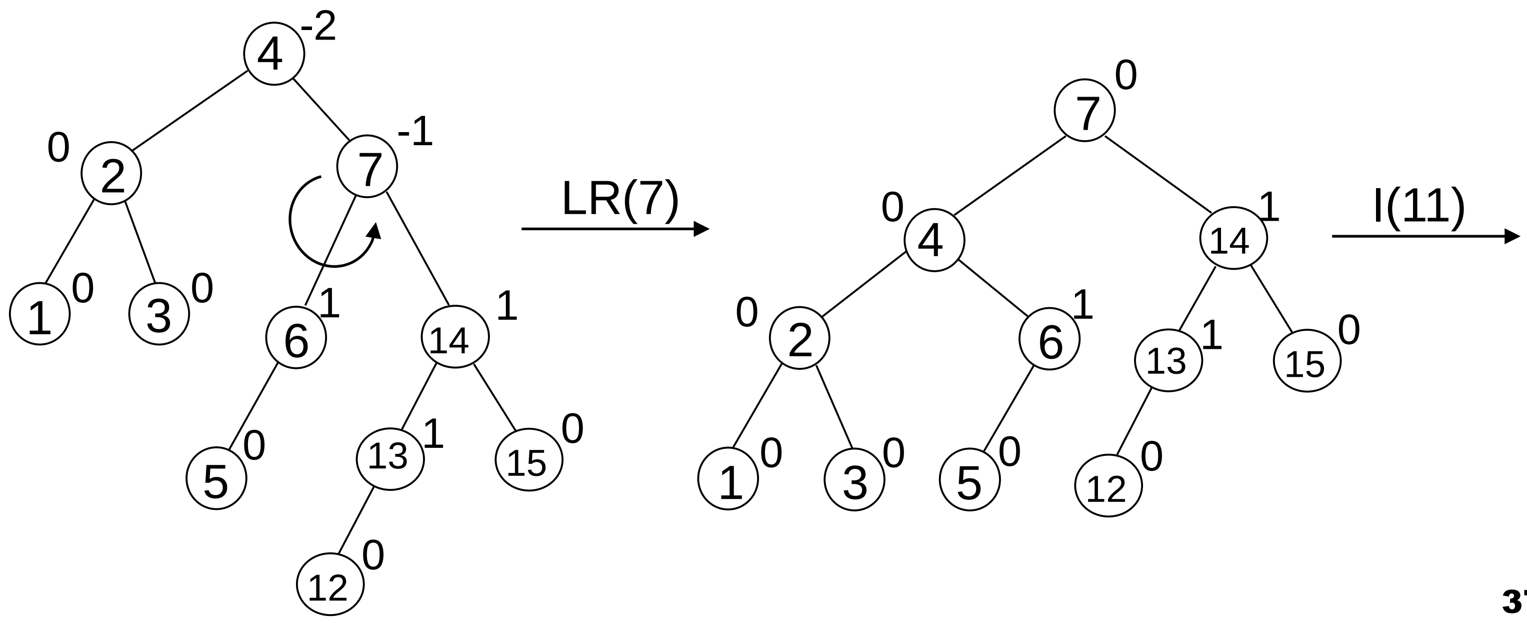
Building AVLs

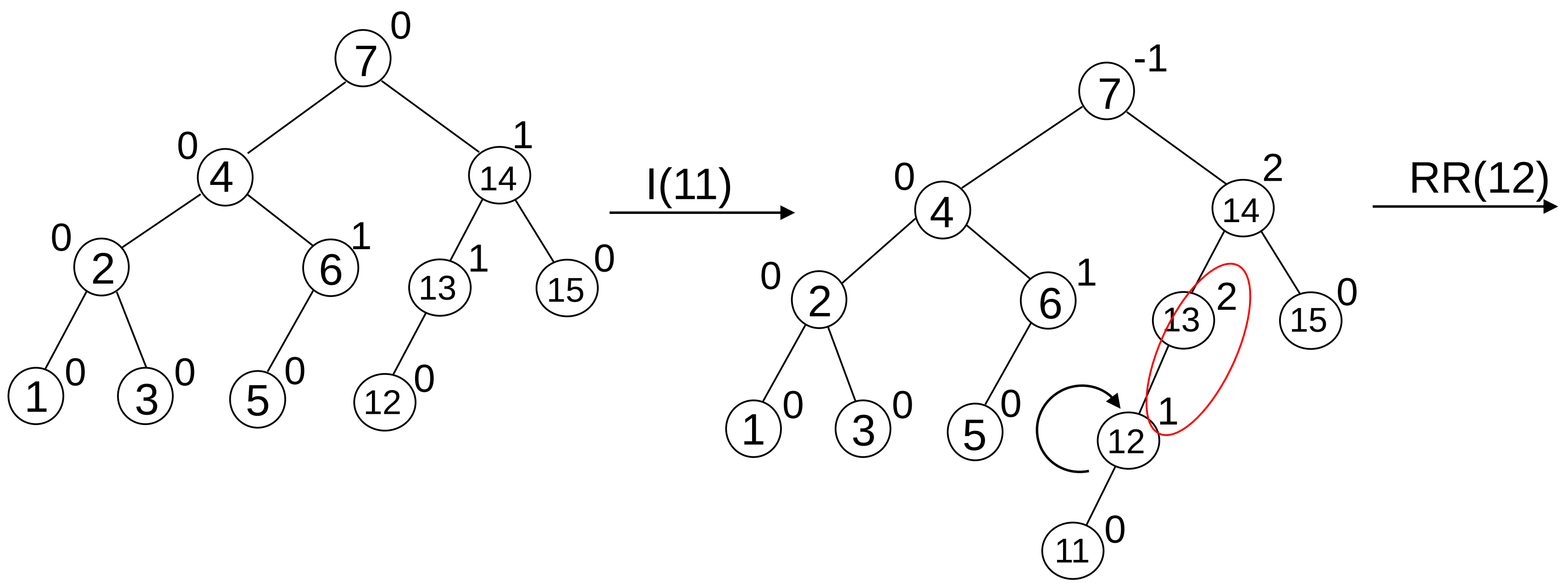
- Continuing with this process

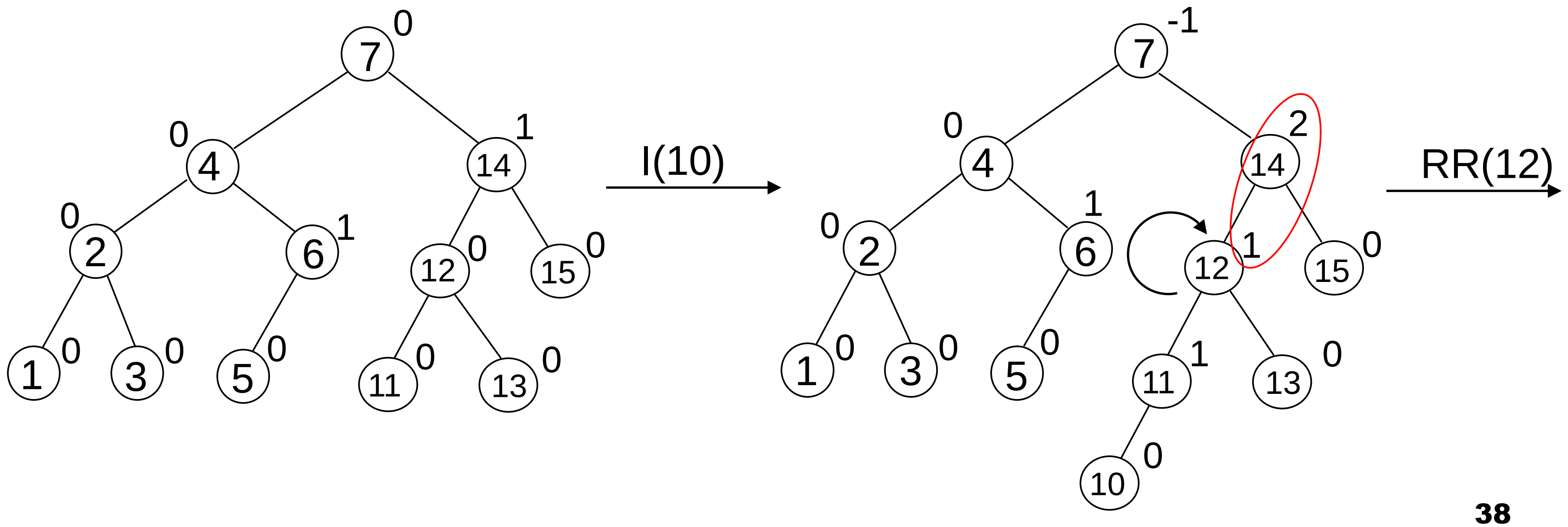


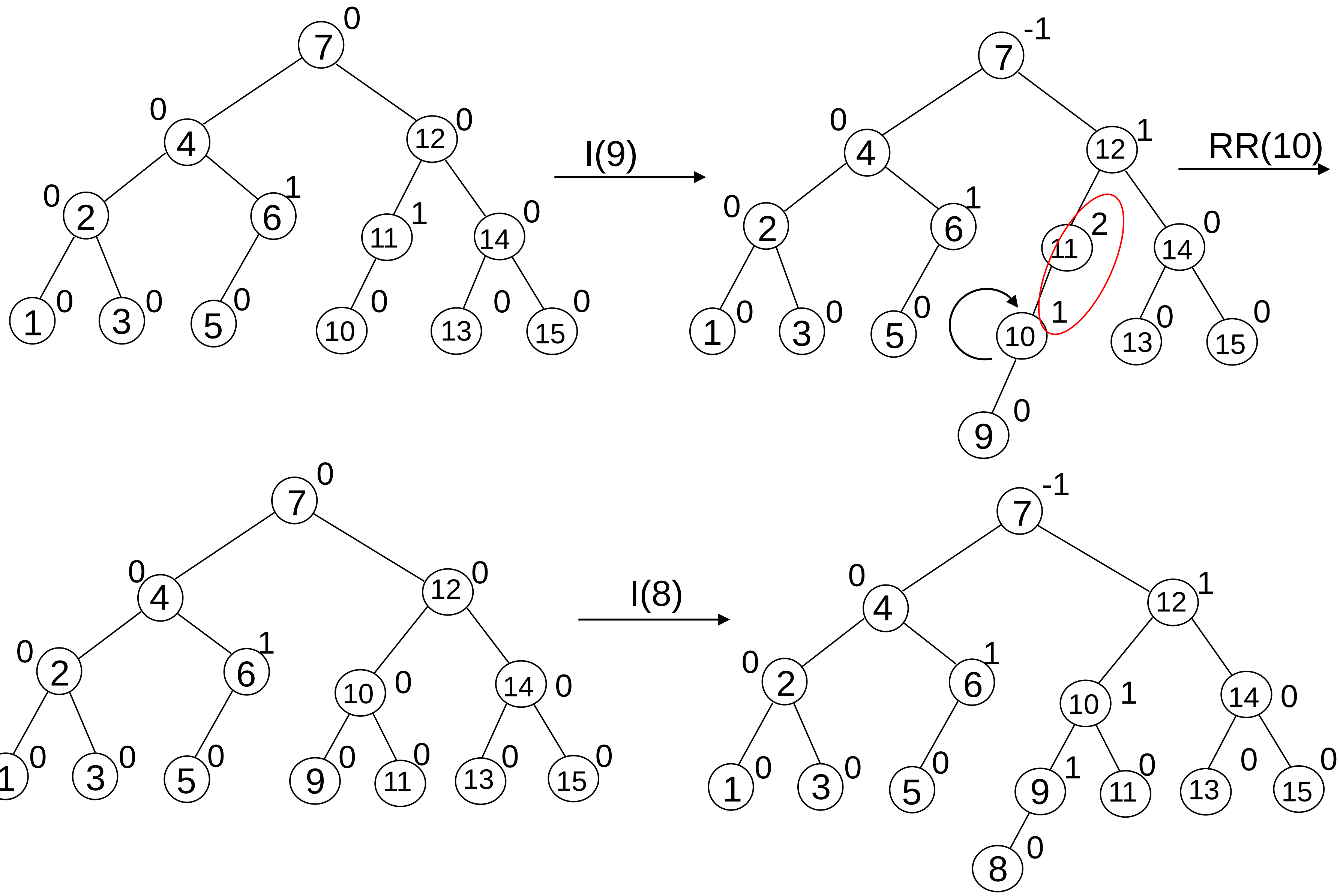

I(7)
LR(6)
I(15)+I(14)
RLR(left(1))=RLR(14)
RR(14)
LR(14)
I(13)


RLR(7)
RR(7)
LR(7)
I(12)
4 -2
2 0
6 -2
14 1
7 -1
15 0
13 0
1 0
3 0
5 0
-1
14 0
7 0
6 1


LR(7)
I(11)
4 -2
7 -1
2 0
6 1
14 1
13 1
15 0
12 0
7 0
4 0
14 1
6 1
13 1


I(11)
RR(12)


I(10)
RR(12)



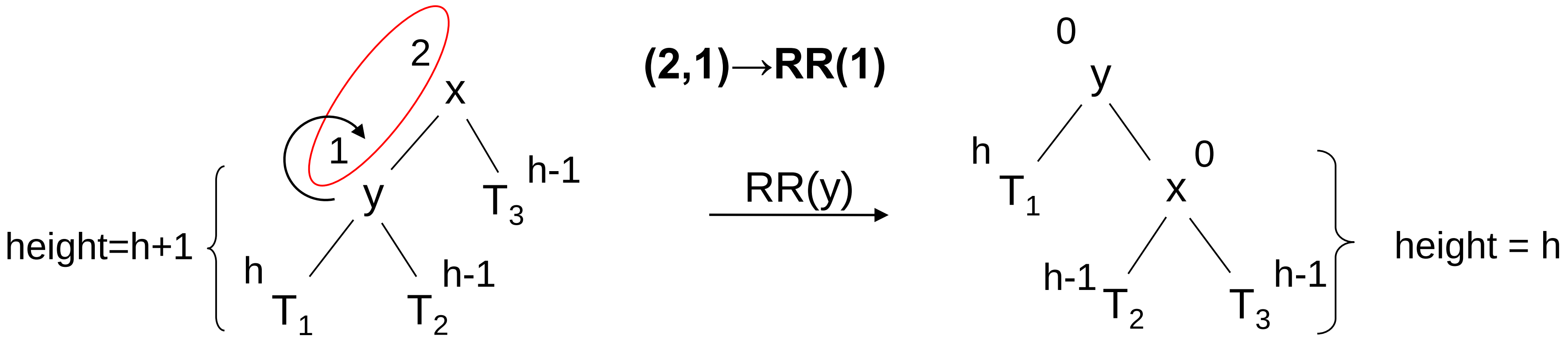
The AVL tree is built

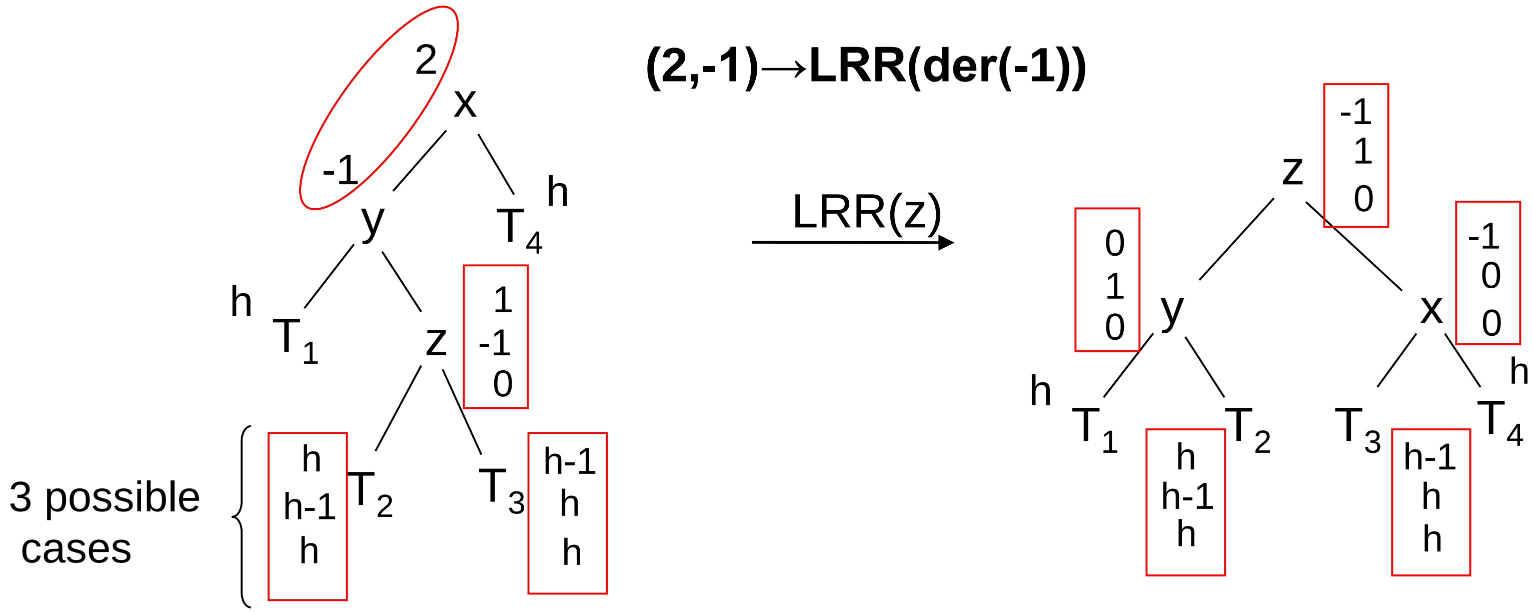
Summary of Rotations

Unbalance	Rotation
(-2,-1)	Left Rotation (LR) at -1 (Left child of -1 turns into right child of -2)
(2,1)	Right rotation (RR) at 1 (Right child of 1 turns into left child of 2)
(-2,1)	Right-left rotation (RLR) at the left of 1 RR(left(1))+ LR(left(1))
(2,-1)	Left-right rotation (LRR) at the right of -1 LR(right(-1))+ RR(right(-1))

Rotation operation I

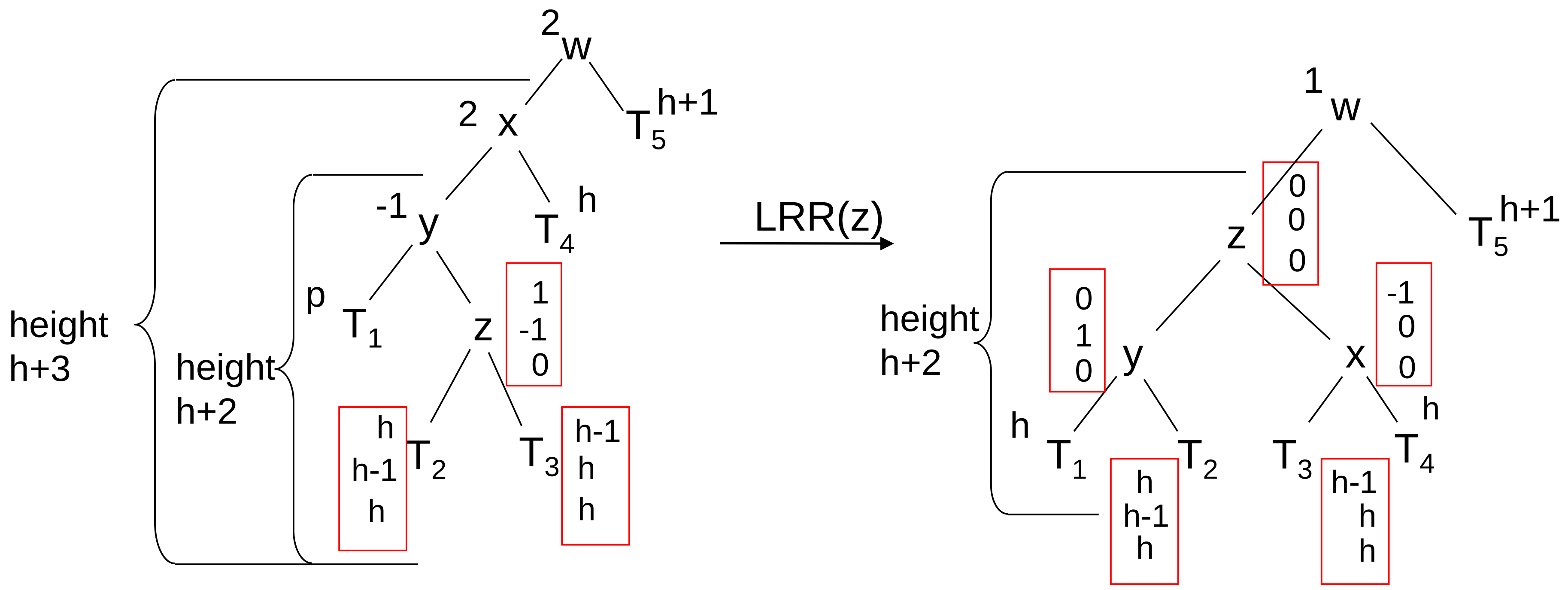
- These four rotations indeed solve the unbalance as we can check in each of the cases, e.g.,





Rotation operation II

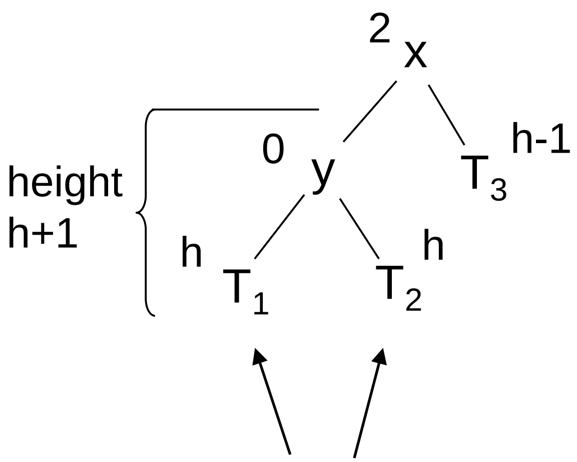
- **Observation:** Rotations solve the unbalancing of type ± 2 located further up the unbalance (±2, ±1)



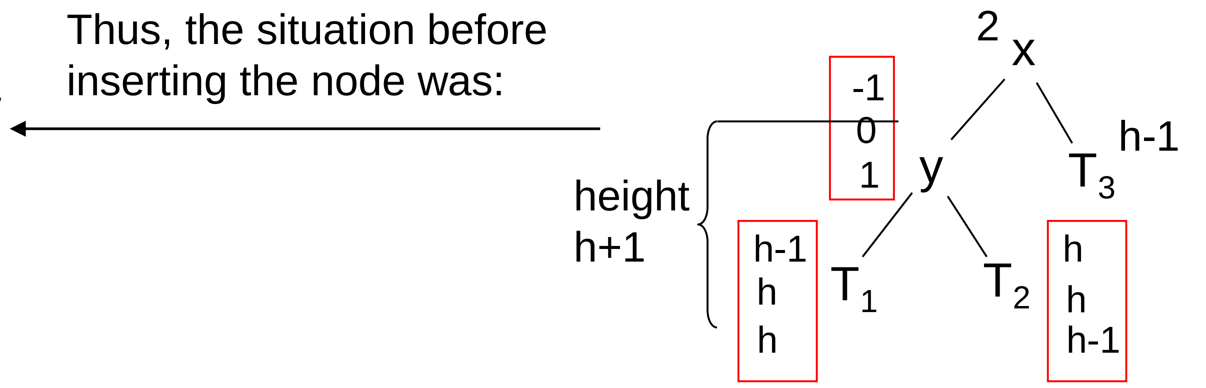
Rotation operation III

- **Observation:** After inserting an element in an AVL, unbalances of type (±2,0) are not possible.

Let us assume that after an insertion we have



Thus, the situation before inserting the node was:



The node was inserted in one of these subtrees to produce the unbalance

It was not an AVL !

Height of AVL trees

- **Proposition:** If T is an AVL with N nodes, then

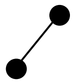
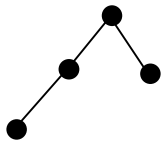
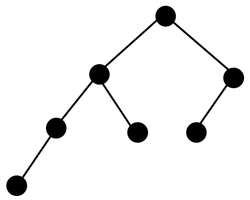
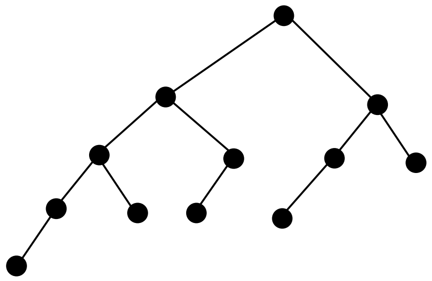
$$\boxed{\text{height}(T)=O(\log(N))}$$

- Because for any binary tree with N nodes it holds that height(T)=Ω(log(N)), then, if T is an AVL then

$$\boxed{\text{height}(T)=\Theta(\log(N))}$$

- To show the above, we are going to estimate the minimum number of nodes $\mathbf{n_h}$ of an AVL $\mathbf{T_h}$ with height **h**.

Mimimum AVLs

h	AVL	n_h	n_h+1	F_{h+2}
0	•	1	2	F_2
1		2	3	F_3
2		4	5	F_4
3		7	8	F_5
4		12	13	F_6
…			….	….

Fibonacci AVL trees I

- F_h is the h-th Fibonacci number.
- Fibonacci numbers verify that:
  - $F_n=F_{n-1}+F_{n-2}$, with $F_0=F_1=1$
- AVL trees T_h are built as

$$\begin{tikzcd}[column sep=small] & T_h \arrow[dl, no head] \arrow[dr, no head] & \\ T_{h-1} & & T_{h-2} \end{tikzcd} \quad \text{or} \quad \begin{tikzcd}[column sep=small] & T_h \arrow[dl, no head] \arrow[dr, no head] & \\ T_{h-2} & & T_{h-1} \end{tikzcd}$$

- $\mathbf{n_h} =1+ \mathbf{n_{h-1}}$**+** $\mathbf{n_{h-2}}$**, y** and thus we have

$$\underbrace{1+\mathbf{n_h}}_{H_h} = \underbrace{1+ \mathbf{n_{h-1}}}_{H_{h-1}} + \underbrace{1+ \mathbf{n_{h-2}}}_{H_{h-2}}$$

Obs:
$H_0=1+n_0=2=F_2$,
$H_1=1+n_0=3=F_3$

- Thus, $n_h+1= H_h= F_{h+2}$

Fibonacci AVL trees II

- It can be shown that the N-th Fibonacci number is

$$F_N = \frac{1}{\sqrt{5}}\left(\underset{\substack{\downarrow N\to\infty \\ \infty}}{\Phi^{N+1}} - \underset{\substack{\downarrow N\to\infty \\ 0}}{\Psi^{N+1}}\right) \text{ where } \Phi = \frac{1+\sqrt{5}}{2} \text{ and } \Psi = \frac{1-\sqrt{5}}{2}$$

since $\Phi>1$ and $|\Psi|<1$

- Thus, we have $F_N \approx (1/\sqrt{5})\Phi^{N+1}$ and since $n_h = F_{h+2}-1$ we get

$$\boxed{n_h \approx \frac{\Phi^3}{\sqrt{5}}\Phi^h = C\Phi^h,}$$

Where h is the tree height and C a constant

Height of an AVL II

- Then, if T is an AVL with N nodes and height h, it follows that

$$N \geq n_h \approx C\Phi^h$$

- Thus, we have that

$$\lg(N) \geq \lg(n_h) = \Omega\ (h \cdot \lg(\Phi)) = \Omega\ (h) = \Omega\ (\text{height}(T))$$

And then

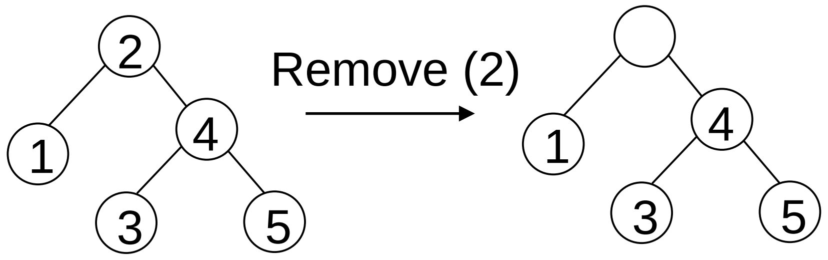
$$\text{height}(T) = h = O(\lg(n_h)) = O(\log(N))$$

- And thus, the cost of searching on an AVL is **O(lg(N)) in the worst case.**

Conclusion

- If we use an AVL as a data structure for a dictionary, both **Search** and **Insert** have a cost O(log(N)) in the worst case.
- How about **Remove**?
  - It is not easy to readjust the nodes of an AVL after deleting a node.
  - The common solution is to perform a **lazy deletion:** instead of removing the node, it s marked as free. Furthermore, if the element is re-inserted, the insertion is fast and easy.

Example:



  - The inconvenience of this method is that storage positions are lost because they are not available for any arbitrary insertion.

In this section we have learnt…

- The concept of AVL tree.
- How to build and AVL tree inserting nodes like in the BSTs and fixing the unbalances with rotations.
- How to estimate the minimum number of nodes of an AVL tree with height H.
- To relate the above to the Fibonacci numbers and to some of their properties.
- The height of an AVL tree of N nodes is O(lg(N)).
- The worst case of searching in an AVL tree is O(lg(N))

Tools and techniques to work on

- Building and properties of AVL.
- Building and properties of Fibonacci trees.
- Problems to solve (at least !!!): those recommended for section 12.

3.4 Hashing

Sorting and searching I

- Grossly, from our analyses in this course, we can see that searching costs are about 1/N times those of sorting:

KC-base methods	Sort	Search
Uneff. methods	$O(N^2)$	$O(N)$
Effic. methods	$O(N\lg N)$	$O(\lg N)$
Lower bound	$O(N)$	$O(1)$

Sorting and searching II

- Is it possible to make searches in a time less than O(log(N))?
  1. Impossible with key comparisons.
  2. But very easy changing our view point !!!

- Scenario:
  1. ADT dictionary with D={data **D**}.
  2. Each data **D** has a unique key **k=k(D)**.
  3. We search **by** keys but **not through** keys (i.e. without key comparisons).

Idea 1

1. We calculate $k^*=\max\{k(D): D\in\mathcal{D}\}$
2. We store each D in an array T of size k* (assuming there are not repeated keys).

   Pseudocode:

```
ind Search(data D, array T)
   if T[k(D)]==D
      return k(D);
  else
      return NULL
```

- Consequence: **$n_{Search}(k,D)=O(1)$!!!**
- **Problem:** if k* is too larch (even though when $|\mathcal{D}|$ is small), the amount of memory to store array T is exessive.

Idea 2

1. We fix $M > |\mathcal{D}|$ and we define an injective function (if, $k \neq k' \Rightarrow \kappa(k) \neq \kappa(k')$) $\kappa : \{k(D)/D \in \mathcal{D}\} \rightarrow \{1,2,3,\ldots,M\}$.
1. We place D at index $\kappa(k(D))$ in array T.
2. Search pseudocode:

```
ind Search2(data D, array T)
  if T[κ(k(D))]==D
    return κ(k(D));
  else
    return NULL
```

Obs: $n_{Search2}(k,D)=O(1)$

- Searching is done in a constant time with a reasonable memory consumption.
- Problem: it is very hard to find such **injective and universal function** (i.e. independent of the key set).

Idea 3

1. We search for a universal k function (valid for any set of keys).
2. We are flexible about k being injective:

   We allow that k is not injective. Thus, two or more distinct data could occupy the same position in array T, but:

   a) We impose that the number of **collisions,** i.e., pairs for which k≠k' but k(k)=k(k') are only a few.

   b) We implement a mechanism to deal with collisions.
3. Open questions:

   a) How to find such function

   b) How to solve collisions

Hash functions

- Goal: low probability for collisions.
- If T has M data, it would be optimal that

  p(collision)=1/M
- Idea: h(D) = value after "rolling" a dice with M sides, but
  - Every time that we are dealing with data D, the dice can send it to different positions !!!
  - Thus, we would like that h(k(D)) has always the same value for each specific k(D).
- This is, we would like that h is a **function** and **random**, like rand() in C.
- Q: How to build random functions?

Hashing: division method

- Given a dictionary D, we fix a number ~**m**>|D|, prime.
- We define **h(k)=k%m**
- With some additional condition over **m**, we can have that for random values k_j, the values of $h(k_j)$ also look random.
  - That is, they would pass randomness tests.

Hashing: multiplication method

- We fix a number **m**>|D|, not necessarily prime (e.g., 2^k or 10^k) and an irrational number Φ (e.g. $(1+\sqrt{5})/2$ or $(\sqrt{5}-1)/2$)
- We define the multiplication hash function as

  $$\mathbf{h(k)=\lfloor m\cdot(k\cdot\Phi)\rfloor},$$

  with (x) meaning the fractional part of x: $\mathbf{(x)=x-\lfloor x\rfloor}$
- Thus we get that, for random values k_j, the value of $h(k_j)$ also looks random.
- Pending issue: **how to solve collisions**

Uniform hash function

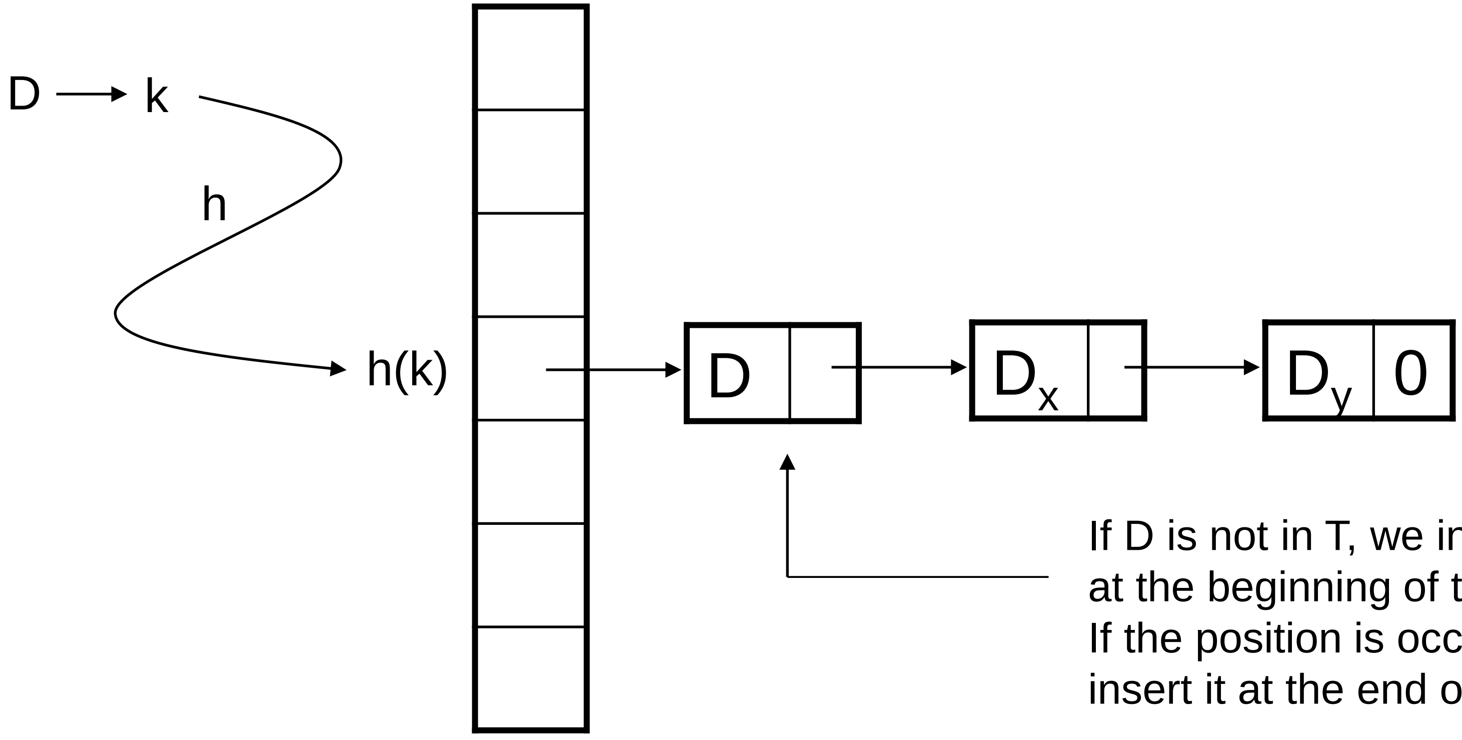
- **Definition:** We say that a hash function h is **uniform** if

  given k,k' with $k \neq k'$, then $p(h(k)=h(k'))=1/m$

- Uniform hash functions are "ideal".
  1. We cannot build them with algorithmic means.
  2. But he performance with this functions is optimal.
- We will use them to simplify the following theoretical analyses.

Collision resolution by chaining

- In hashing with chaining, we use as a hash table an array with pointers to **linked lists**



If D is not in T, we insert data D at the beginning of the linked list. If the position is occupied, we insert it at the end of the list.

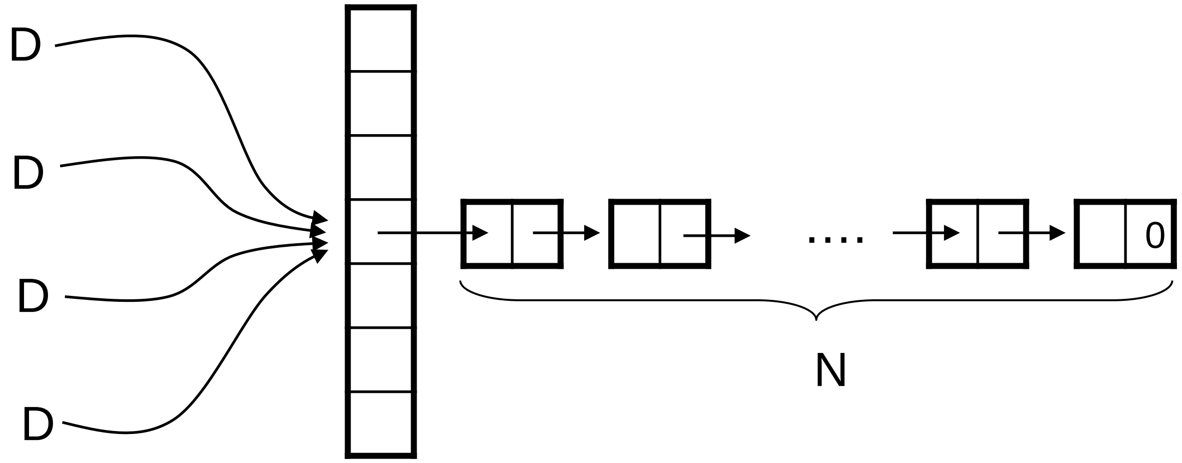
Search with hashing with chaining

- Pseudocode: Linear search in the linked list.

```
ind Search(data D, array T)
    return LSearch(D,T[h(k(D))]);
```

- The cost is not O(1), since **LSearch** has a loop

- Furthermore, $W_{LSearch}$ (N)=N if for every k, h(k) = h_0



Observation: This situation may occur, but it is very unlikely if the hash function is well designed.

Chaining with uniform hashing I

- **Proposition:** Let h be a uniform hashing function in a hash table with chaining and dimension **m** and let **N** be the number of data to insert:

$$(i)\quad A^{f}_{SHC}(N,m)=\frac{N}{m}=\lambda \longleftarrow \text{Load factor}$$

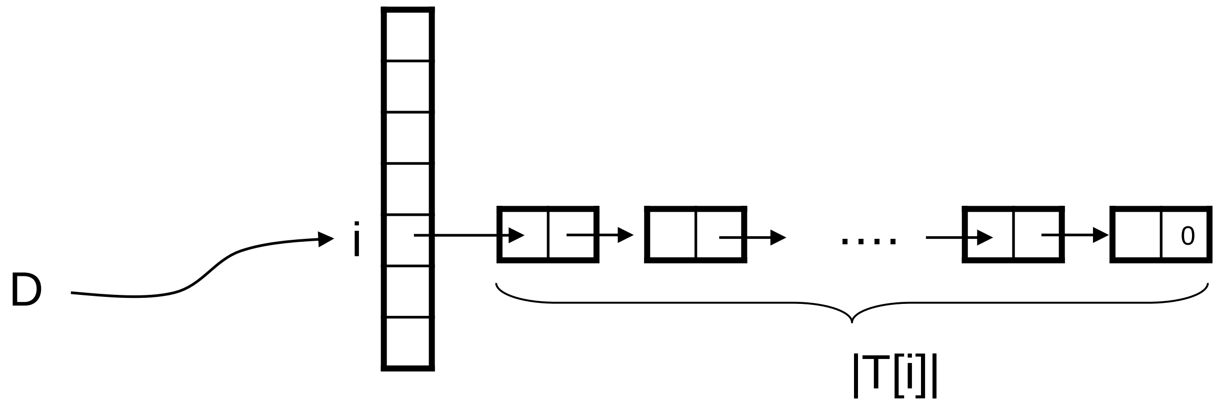
$$(ii)\quad A^{s}_{SHC}(N,m)=1+\frac{\lambda}{2}+O(1)$$

SHC= search using hashing with chaining

- λ is called the **load factor**: the larger this factor, the more costly is the search.

Average cost in an unsuccessful search

- **Demostration (i):** Let **D** be data that is not in the hash table (fail search), let h(k(D))=h(D)=i, and let n_{SHC}(D,T)=|T[i]| (number of elements at index i in the linked list):



$$\Rightarrow A^{f}_{SHC}(N,m)=\sum_{i=1}^{m}\underbrace{p(h(D)=i)}_{\text{uniform h}}|T[i]|=\frac{1}{m}\sum_{i=1}^{m}|T[i]|=\frac{N}{m}=\lambda$$

Average cost in a successful search

- **Demonstration (ii): Let us reduce the successful search to an unsuccessful search in a smaller table.**
- We number the data according to the order in which we introduce them in table T, $\{D_1,D_2,\ldots,D_j,\ldots,D_N\}$
- In addition, we denote by T_i to the state of table T **before** introducing element D_i (i.e., the table T_i has elements $D_1,D_2,\ldots,D_{j-1}$),
- Thus D_i **is not** in T_i, and then

$$\underbrace{n^{s}_{SHC}(D_i, m; T)}_{\text{successful search}} = 1 + \underbrace{n^{f}_{SHC}(D_i, m; T_i)}_{\text{failed search}}$$

successful search =
1 + unsuccessful search

Note: here we assume that each element $\mathbf{D_i}$ is inserted at the end of the linked list.

Average cost in a successful search II

- We assume the following approximation:

$$n^s_{SHC}(D_i, m) \cong 1 + A^f_{SHC}(i-1, m) = 1 + \frac{i-1}{m} \longleftarrow$$

Load factor in T_i

then

$$A^s_{SHC}(N,m) = \frac{1}{N}\sum_{i=1}^{N} n^s_{SJC}(D_i, m) \cong \frac{1}{N}\sum_{i=1}^{N}\left(1 + \frac{i-1}{m}\right) = 1 + \frac{1}{Nm}\sum_{j=1}^{N-1} j =$$

$$= 1 + \frac{1}{Nm}\frac{N(N-1)}{2} = 1 + \frac{1}{2}\frac{N}{m} - \frac{1}{2m} = 1 + \frac{\lambda}{2} + O(1)$$

$$A^f_{SHC}(N,m) = \frac{N}{m} = \lambda$$

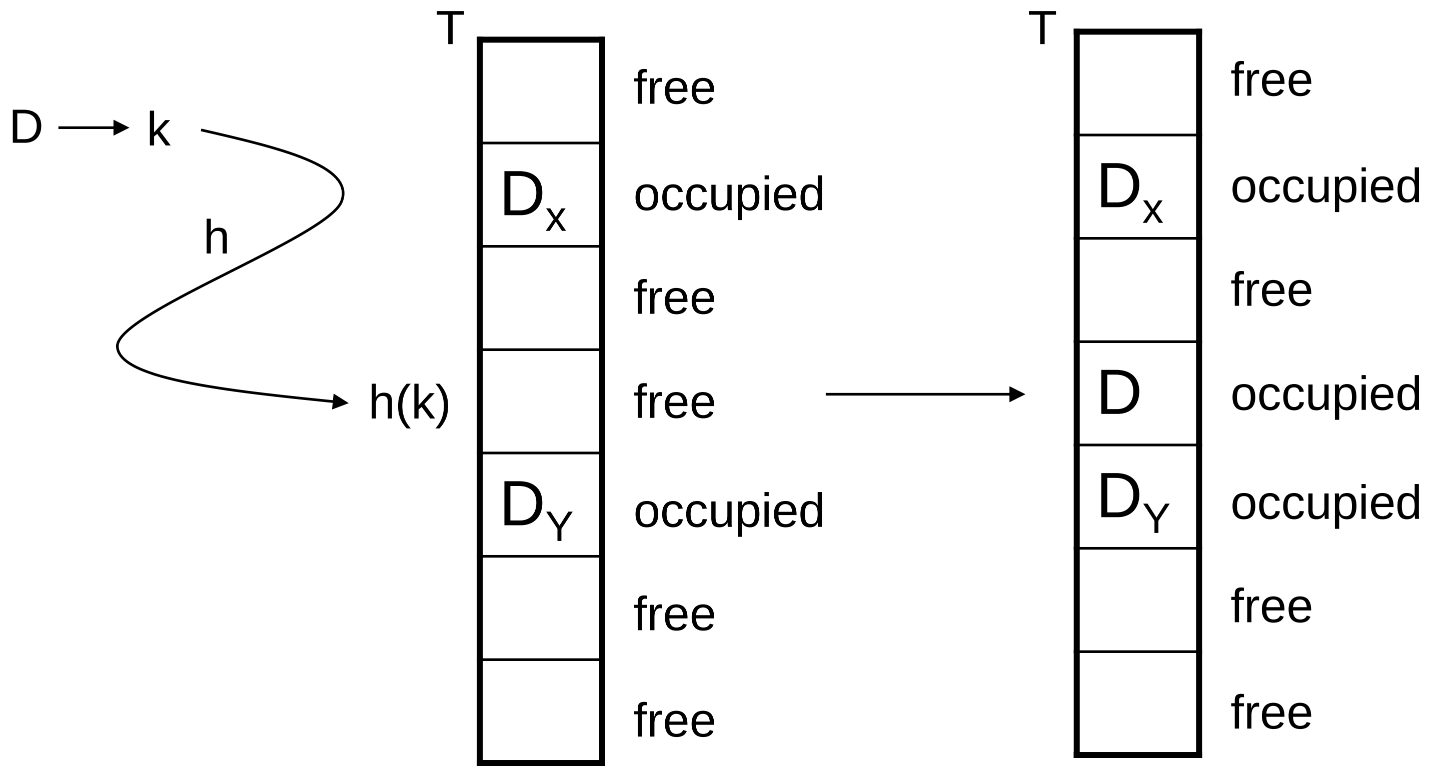
$$A^s_{SHC}(N,m) = 1 + \frac{\lambda}{2} + O(1)$$

Obs: If the hash function is uniform the cost of searching is constant if $\lambda=\Theta(1)$, which occurs if $N \cong m$. For example, if N=200 and m=100.

$A^f \cong 200/100=2$ $\quad$ $A^s \cong 1+2/2=2$

Collision resolution by open addressing

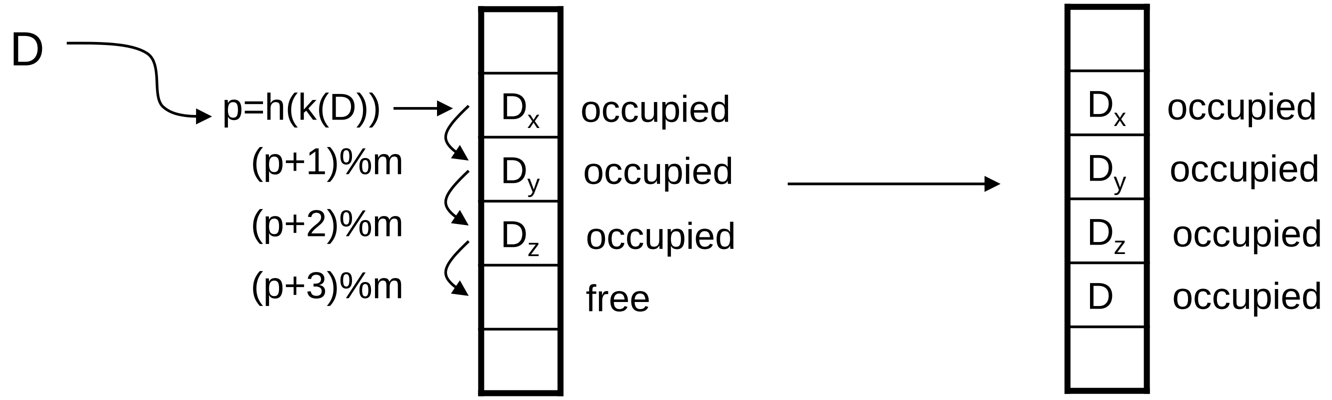
- In open-addressing hashing the table T stores data with the following strategy:



- What do we do when the hash function assigns a location that is already occupied (collision)?

Collision resolution by open addressing

- There are several methods to solve collisions in open addressing by repeated probing.
- **Linear probing:** If position **p=T[h(D)]** is occupied, we try to place **D** successively in positions **(p+1)%m, (p+2)%m,…., ,** until we reach an i **where** position **(p+i)%m** is free.



Collision resolution by open addressing

- **Quadratic probing:** Same as in the linear probing but trying in positions **$p=(p+0^2)\%m$, $(p+1^2)\%m$, $(p+2^2)\%m$,….,** until we find an **i**, were **$(p+i^2)\%m$** is free**.**
- **Random probing:** we try in positions **p_1, p_2, p_3,….,p_i** randomly established.
  - This method is not used in real situations.
  - But it is an "ideal" situation in hashing.
  - It allows us to calculate the cost of searching with open addressing.

Differences in the methods

- **Obs 1:** In the chaining method, the location of a given data D is always a fixed position in the table (h(k(D)). In the open addressing method the position of D will depend on h(k(D) and **the estate of the table** at the time of the insertion.
- **Obs 2:** In chaining hashing the load factor λ (=N/m), can be >1.

  In open addressing hashing we will always have $N \leq m$, and thus $\boldsymbol{\lambda \leq 1}$.

  In practice, we use N< m and λ <1 (for example m=2*N y λ=0.5).

Average cost with random probing I

- **Proposition:** Let h be a uniform function in the context of a hash table with open addressing and random probing. Then:

$$(i)\quad A^{f}_{HRP}(N,m) = \frac{1}{1-\lambda}$$

$$(ii)\quad A^{s}_{HRP}(N,m) = \frac{1}{\lambda}\log\frac{1}{1-\lambda}$$

Obs 1: If $\lambda \to 1$ then $A^{f}_{HRP}(N,m) \to \infty$

Obs 2: If $\lambda \to 1$ then $A^{s}_{HRP}(N,m) \to \infty$

Prove it as an exercise.

Average cost in unsuccessful searches

- Demonstration (i): Let T be a hash table with RA with dimension m and N data. Since h is uniform, given a data **D** we have:

N data in a table of size m

p(T[h(D)] occupied) = N/m = λ

p(T[h(D)] free) = 1-λ

$$\Rightarrow A^{f}_{HRP}(N,m) = \sum_{k=1}^{\infty} k \cdot p(\text{k probes}) = \sum_{k=1}^{\infty} k \cdot \lambda^{k-1}(1-\lambda) =$$

# of probes

For k probes we must have { k-1 occupied positions and 1 free position }

p(k probes) =$\lambda^{k-1}(1-\lambda)^1$

$$= (1-\lambda)\sum_{k=1}^{\infty} k \cdot \lambda^{k-1} = (1-\lambda)\frac{d\left(\sum_{k=0}^{\infty}\lambda^k\right)}{d\lambda} = (1-\lambda)\frac{d\left(\frac{1}{1-\lambda}\right)}{d\lambda} = (1-\lambda)\frac{1}{(1-\lambda)^2} = \frac{1}{1-\lambda}$$

Average cost in successful searches I

- **Proposition (ii):** $A^{s}_{HRP}(N,m) = \frac{1}{\lambda}\log\frac{1}{1-\lambda}$
- **Demonstration:** Again, **we are going to reduce the successful search to an unsuccessful search in a smaller table.**
- As in SHC, we number the data in table T according to the order in which we insert the data $\{D_1,D_2,\ldots,D_j,\ldots,D_N\}$, and we denote T_i the state of the table T **before** introducing the element D_i.

Obs: if $n^{s}_{T}(D_i)$ is the number of probes needed to find (= needed to insert) element D_i in table T_i , we have

$$n^{s}_{T}(D_i) = n^{f}_{T_i}(D_i) \cong A^{f}_{HRP}(i-1,m)$$

Average cost in successful searches II

- Thus we have:

$$A^{s}_{HRP}(N,m) = \frac{1}{N}\sum_{i=1}^{N} n^{s}_{T}(D_i) \cong \frac{1}{N}\sum_{i=1}^{N}\frac{1}{1-\frac{i-1}{m}} = \frac{1}{N}\sum_{j=0}^{N-1}\frac{1}{1-\frac{j}{m}}$$

Approximating by integrals we have:

$$A^{s}_{HRP}(N,m) \cong \frac{1}{N}\int_0^N \frac{1}{1-\frac{x}{m}}dx = \frac{1}{N/m}\int_0^{N/m}\frac{1}{1-u}du = \frac{1}{\lambda}\int_0^{\lambda}\frac{1}{1-u}du$$

Changing variables
u=x/m ⇒ dx=m·du

Thus,

$$A^{s}_{HRP}(N,m) \cong \frac{1}{\lambda}\log\frac{1}{1-\lambda}$$

Average costs for other probing methods I

- In the previous demonstration we can see that if we have the expression for the cost in unsuccessful searches:

$$f(\lambda) = A^{f}_{HRP}(N,m) = \frac{1}{1-\lambda}$$

we can calculate the cost for successful searches by calculating

$$A^{s}_{HRP}(N,m) \cong \frac{1}{\lambda}\int_0^{\lambda} f(u)du = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-u}du$$

- This argument can be repeated for any open addressing probing method P, i.e.:

$$\boxed{\text{If } A^{f}_{P}(N,m) = f(\lambda) \quad \text{then } A^{s}_{P}(N,m) \cong \frac{1}{\lambda}\int_0^{\lambda} f(u)du}$$

Average costs for other probing methods II

- Proposition: If we use linear probing:

$$(i) A_{LP}^{f}(N,m) \cong \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

$$(ii) A_{LP}^{s}(N,m) \cong \frac{1}{\lambda}\int_0^{\lambda}\frac{1}{2}\left(1+\frac{1}{(1-u)^2}\right)du = \frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$$

In this section…

- We have learnt
  - The concept of hash table.
  - The mechanisms to build a hash table and to search on it.
  - The concept of uniform hash function.
  - Some universal types of hash functions (division and multiplication).

In this section…

- And also
  - The main methods for collision resolution in a hash table: **chaining and open addressing**.
  - The main methods for **probing** in a hash table with **open addressing**.
  - To estimate the **average cost** of successful and unsuccessful searches in the case of **random probing**.
  - To reduce the average cost of **successful searches** to the cost of **unsuccessful searches**.

Tools and techniques to work on

- Function and construction of hash tables.
- Hash table design strategies that guaranty a certain performance.
- Estimation of average cost of successful searches from the average cost of unsuccessful searches.
- Problems to solve (at least !!!): those recommended in section 13.