Tree data structure
(part 2)

Binary Search Trees - (**BST**)
Implementation of Trees
Application Examples

Prof. Hidouci W.K. / ALSDD / 1CP / ESI

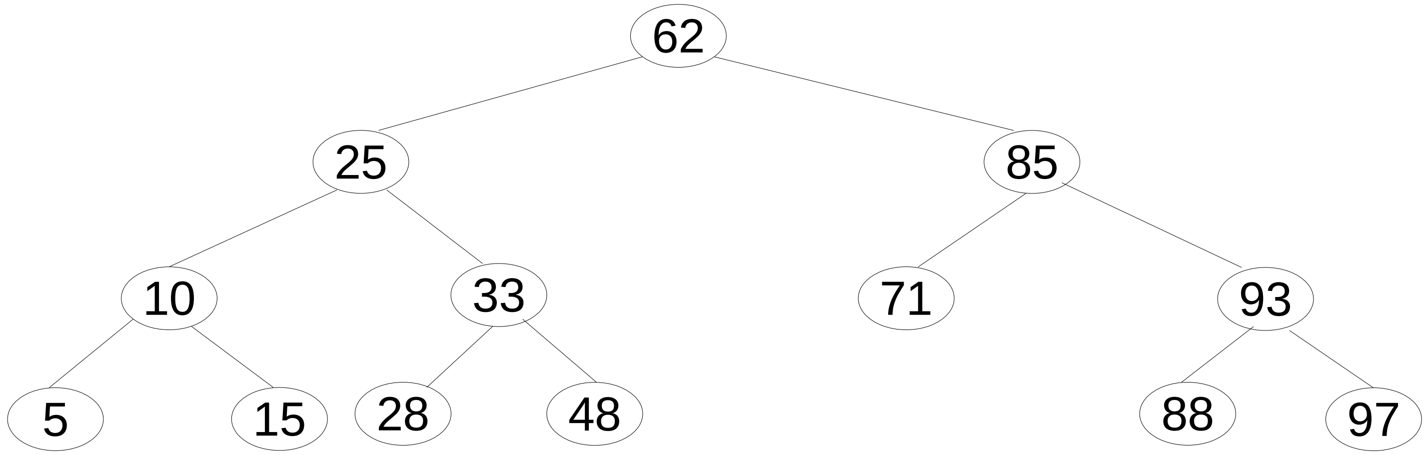
Binary Search Trees : BST

*used to <u>speed up</u> the operations of **searching**, **inserting** and **deleting** values in sets for which a <u>total ordering relationship</u> exists*

***R** is a **binary search tree** if:*

*   * all values in the <u>left subtree of **R**</u> are <u>less than info(**R**)</u>*
*   * all values in the <u>right subtree of **R**</u> are <u>greater than info(**R**)</u>*
*   * the <u>left subtree of **R**</u> is a **binary search tree***
*   * the <u>right subtree of **R**</u> is a **binary search tree***

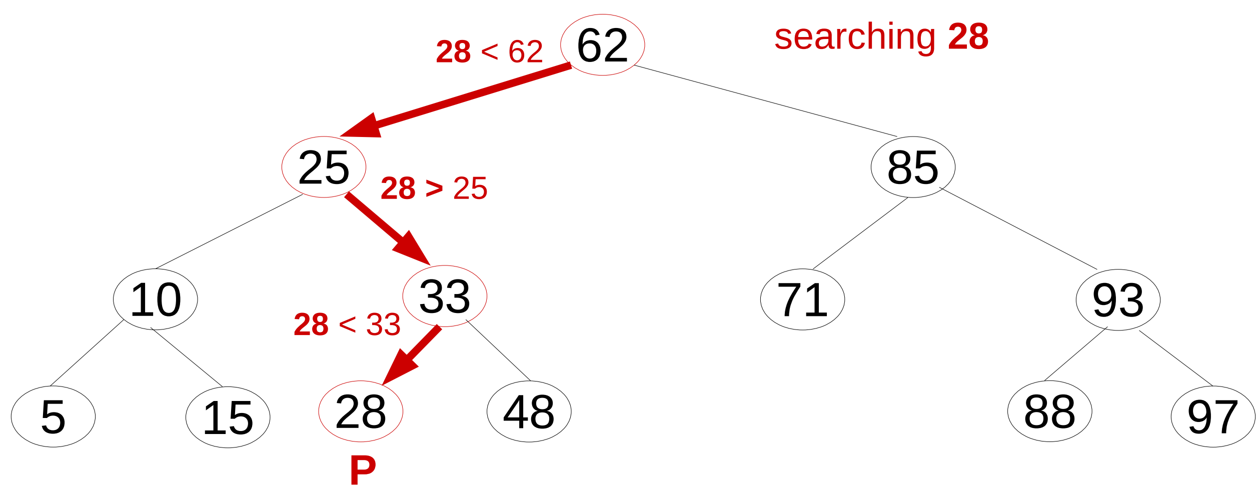
Base case :
*   * **NIL** (the empty tree) is a **binary search tree***



BST : Search Algorithm

The search for a value ***v*** consists in visiting, at most, all the nodes of <u>a single branch</u> of the tree (the algorithm is efficient if the branch is not too long)

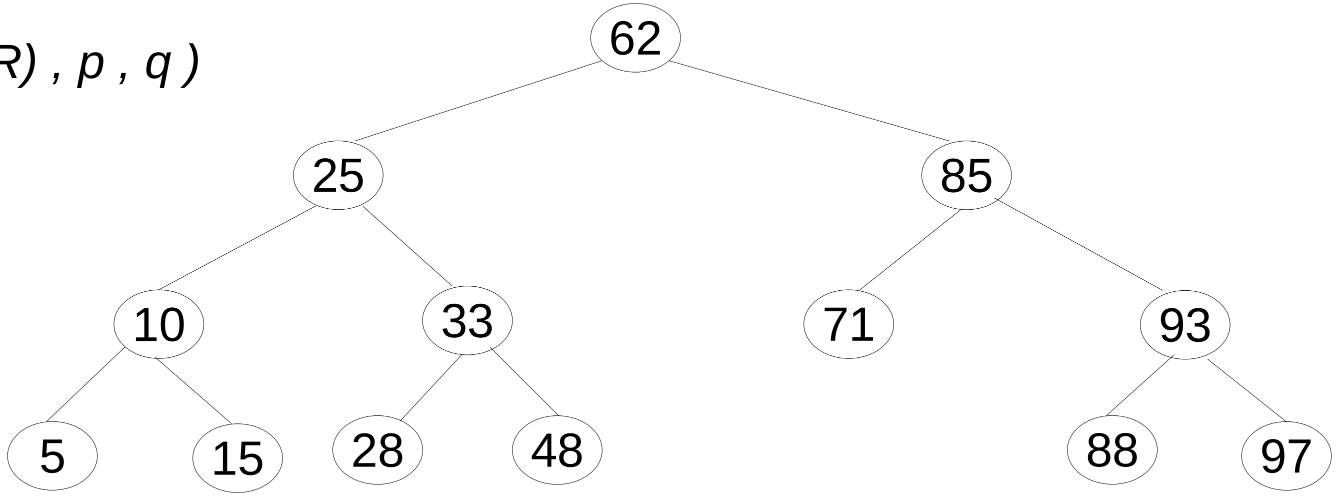
```
// return in p the node containing v (or NIL if not found)
Search( in : v , R ; out : p )
IF ( R == NIL )
    p ← NIL                 // in case of an empty tree (R==NIL), v does not exist
ELSE
    IF ( v == info(R) )     // in case v exists in root node (v == info(R))
        p ← R
    ELSE                    // in all other cases,
        IF ( v < info(R) )          // the search continues either
            Search( v , lc(R) , p ) // in the left subtree
        ELSE                        // or
            Search( v , rc(R) , p ) // in the right subtree
        EndIf
    EndIf
EndIf
```



BST : Search Algorithm (extended version)

if ***v*** exists, we return the node containing ***v*** (in ***p***) and its parent (in ***q***)
else we return NIL (in ***p***) and the last visited node (in ***q***)

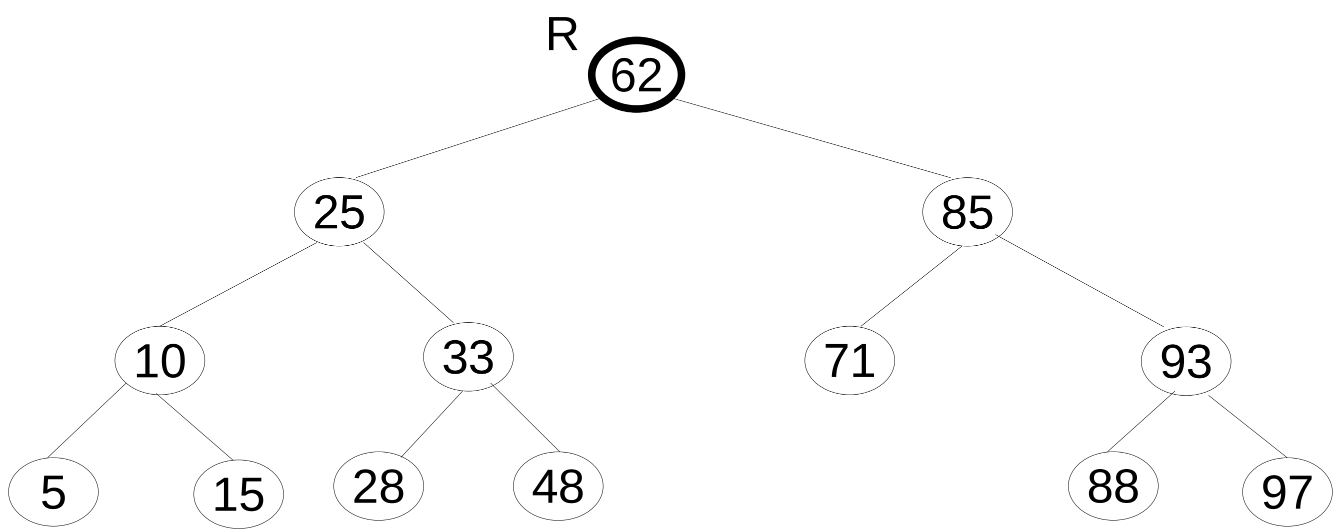
```
Search( in : v , R ; out : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL                // base case 1 : v cannot exist in an empty tree
ELSE
    IF ( v == info(R) )
        p ← R ; q ← NIL              // base case 2 : v exists in the root of the tree

    ELSE                             // general case : search for v in one of R's subtrees
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:<u>the last visited node</u>

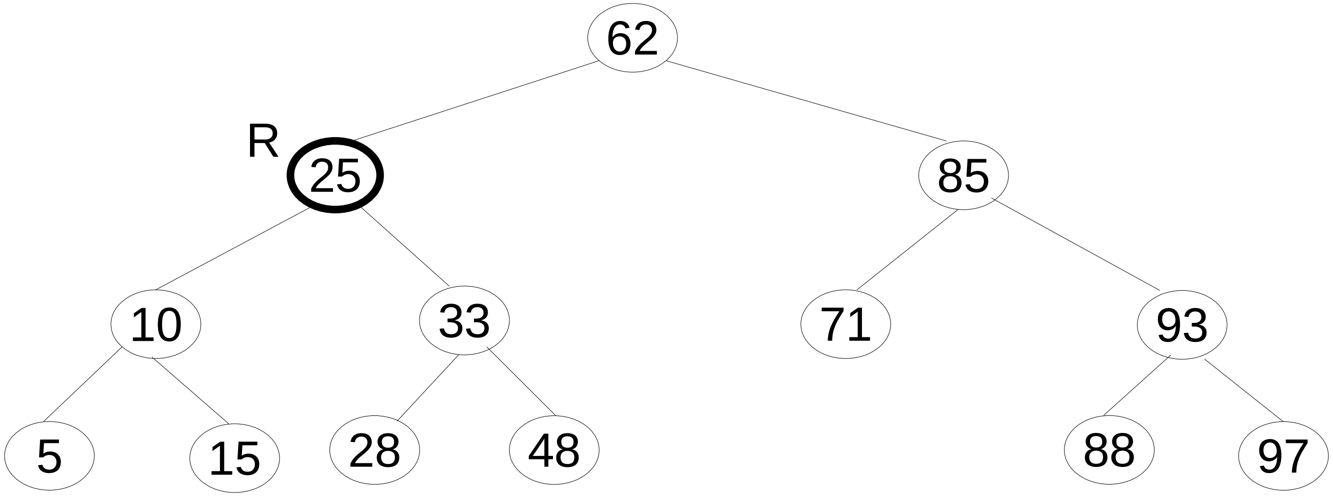
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:<u>the last visited node</u>

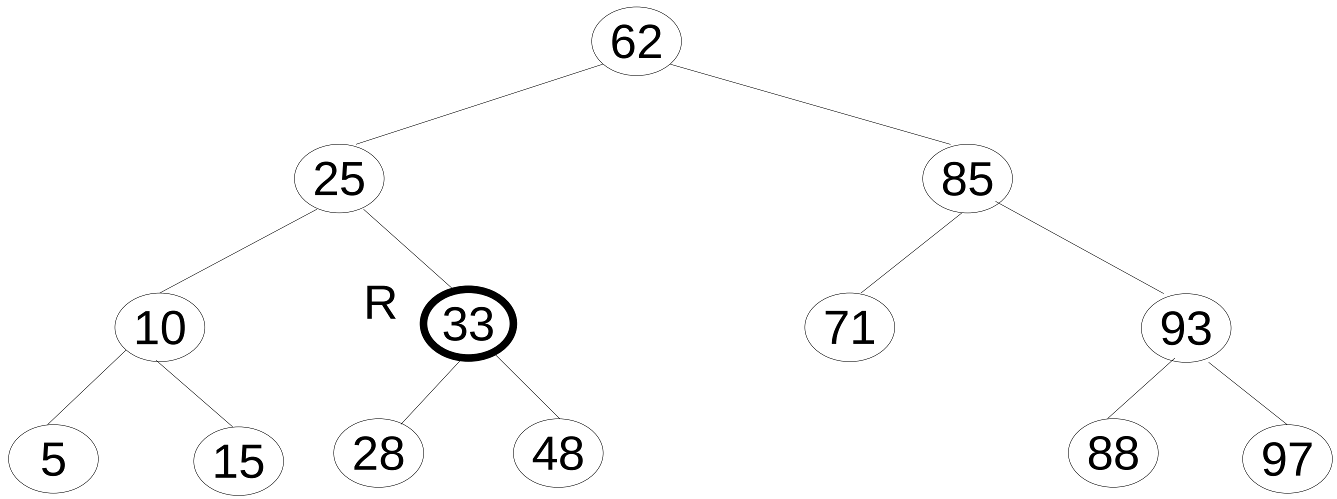
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
      p ← NIL ; q ← NIL
ELSE
      IF ( v == info(R) )
            p ← R ; q ← NIL
      ELSE
            IF ( v < info(R) )
                        Search( v , lc(R) , p , q )
            ELSE
                        Search( v , rc(R) , p , q )
            EndIf

            IF ( q == NIL )
                  q ← R
            EndIf

      EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q ← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

Search*(entrées : v , R ; sorties : p , q)*

IF *(R == NIL)*

    p ← NIL ; q ← NIL

ELSE

    IF *(v == info(R))*

        p ← R ; q ← NIL

    ELSE

        IF *(v < info(R))*

                Search*(v , lc(R) , p , q)*

        ELSE

                Search*(v , rc(R) , p , q)*

        EndIf

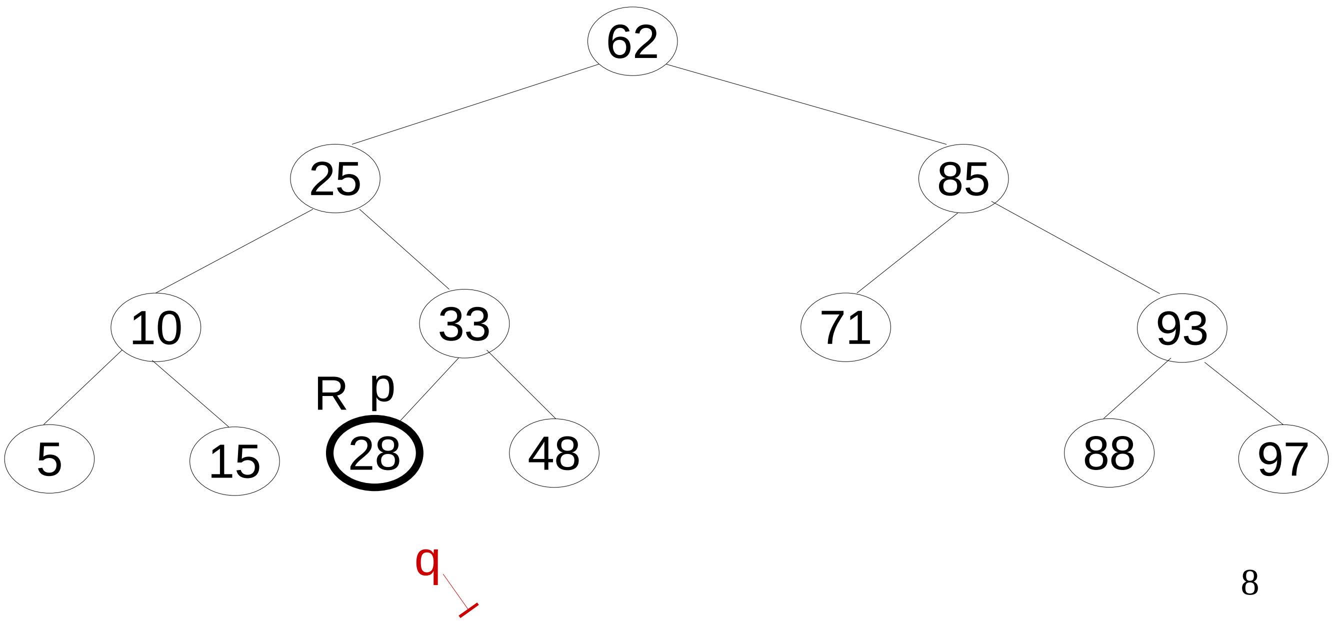
        IF *(q == NIL)*

            q ← R

        EndIf

    EndIf

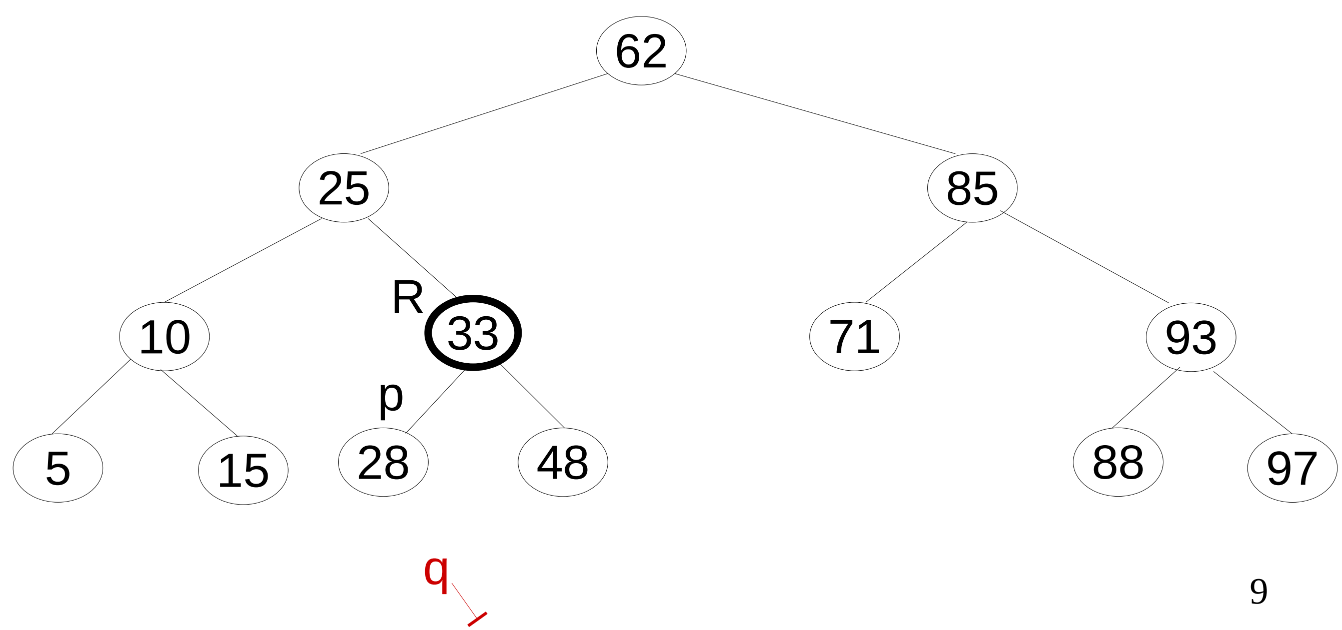
EndIf



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

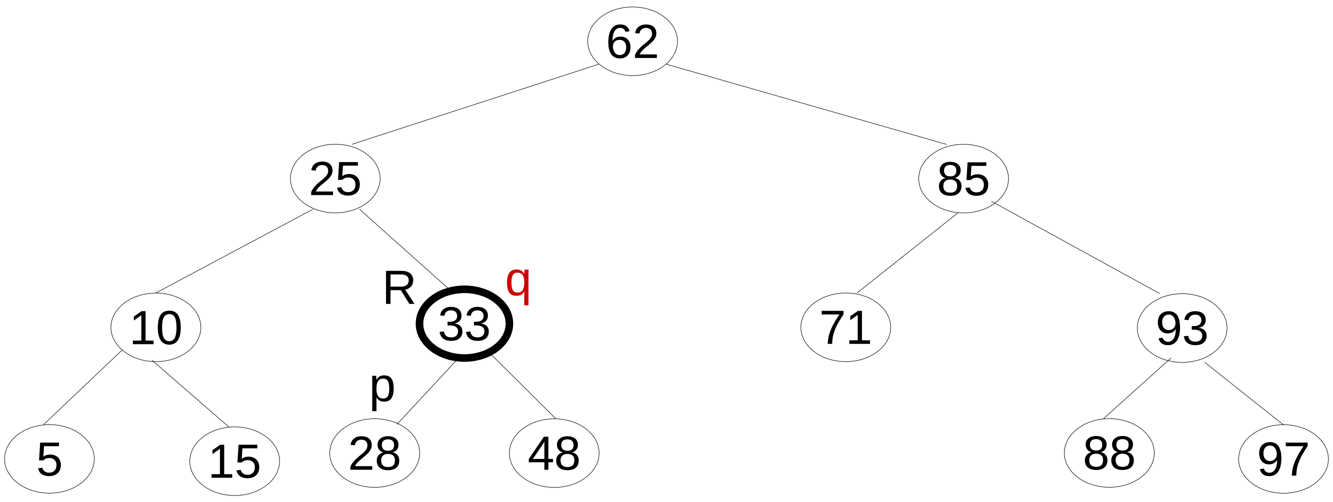
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
      p ← NIL ; q ← NIL
ELSE
      IF ( v == info(R) )
            p ← R ; q← NIL
      ELSE
            IF ( v < info(R) )
                        Search( v , lc(R) , p , q )
            ELSE
                        Search( v , rc(R) , p , q )
            EndIf

            IF ( q == NIL )
                  q ← R
            EndIf

      EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

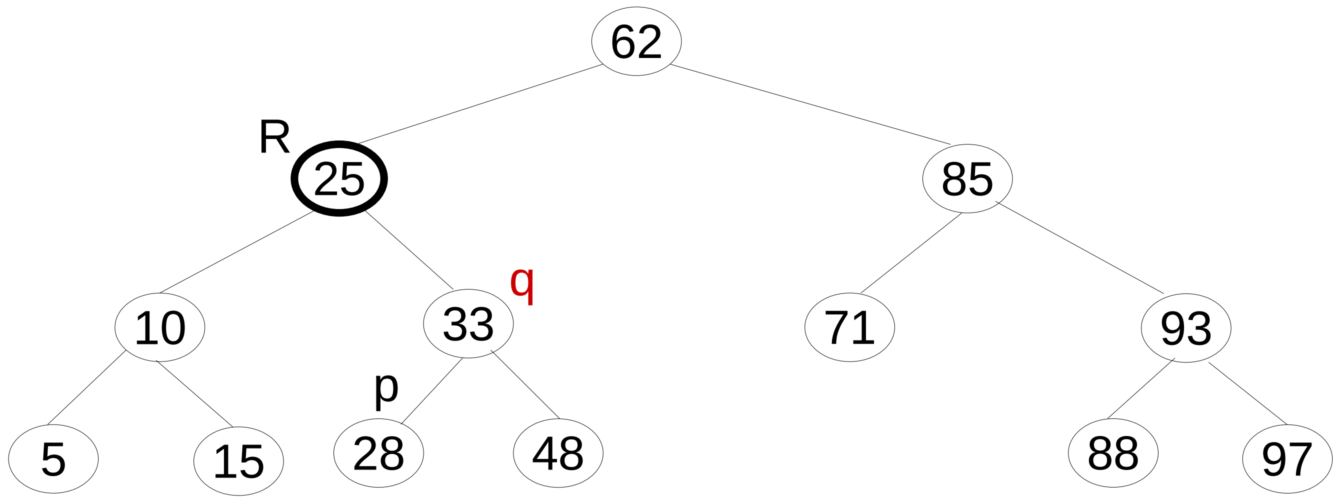
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
      p ← NIL ; q ← NIL
ELSE
      IF ( v == info(R) )
            p ← R ; q← NIL
      ELSE
            IF ( v < info(R) )
                        Search( v , lc(R) , p , q )
            ELSE
                        Search( v , rc(R) , p , q )
            EndIf

            IF ( q == NIL )
                  q ← R
            EndIf

      EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

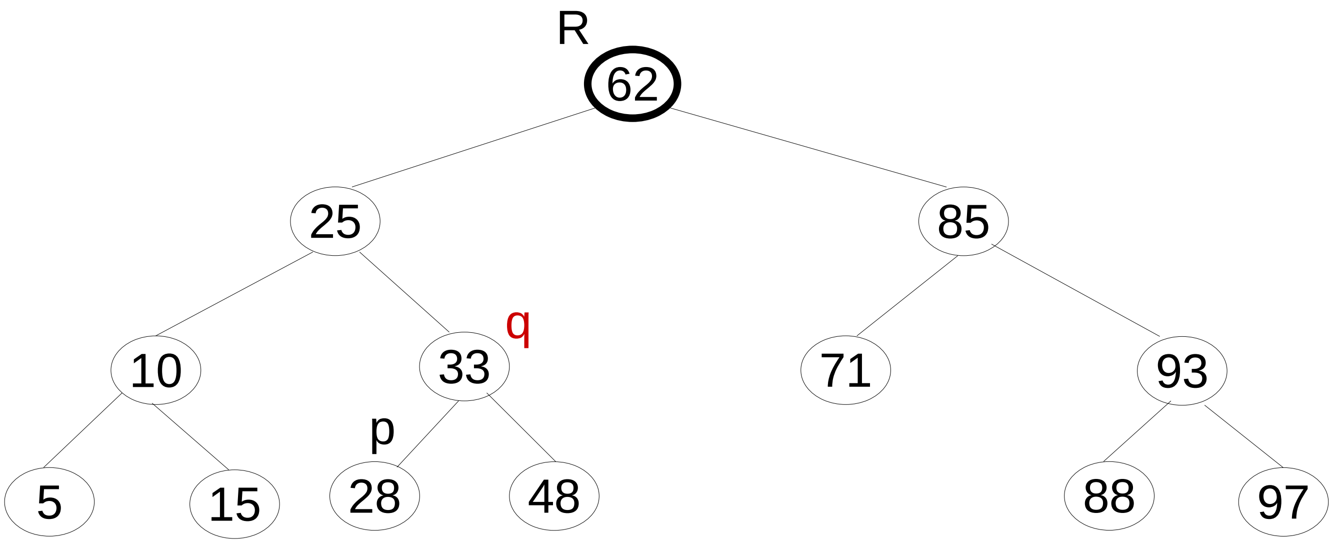
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q ← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

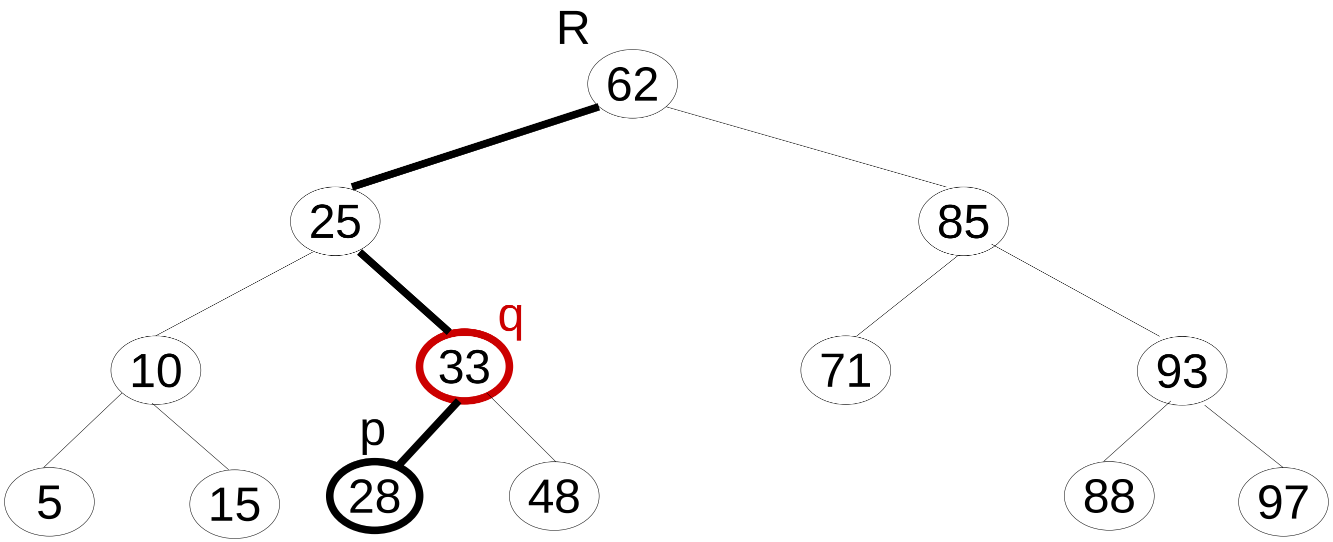
```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q ← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



BST : Unfolding of the extended search algorithm (v = 28)

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```



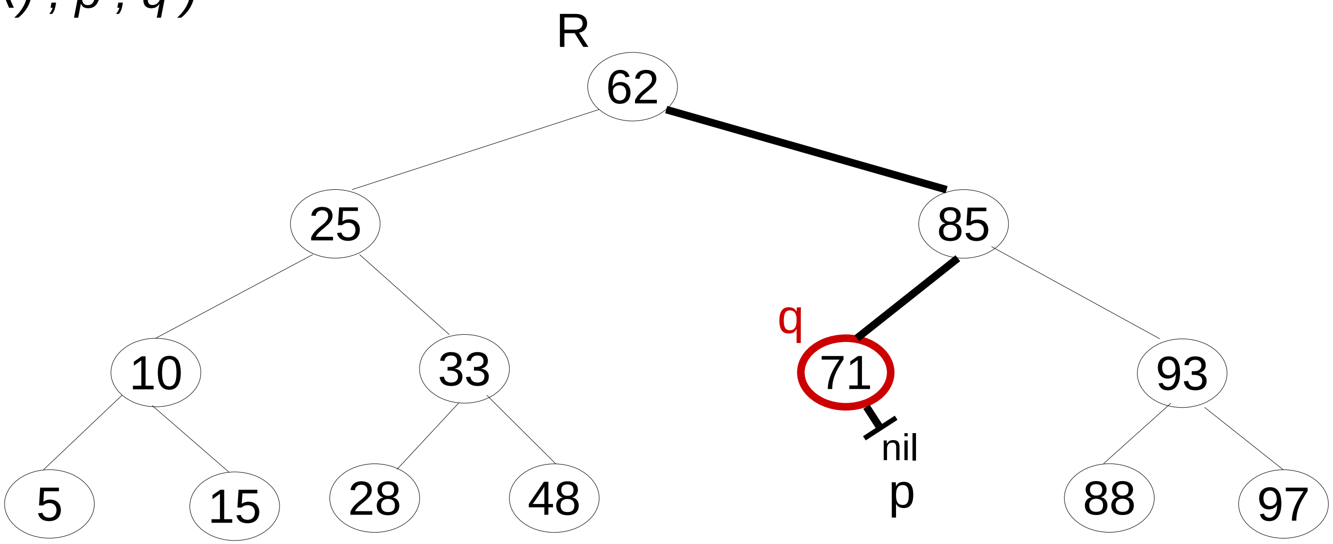
Searching 28 → (p and q)

Searching 80 …

⇒ the traversed branch is : 62 → 85 → 71 → NIL

// if v exists, return node p and its parent q, otherwise p:NIL and q:the last visited node

```
Search( entrées : v , R ; sorties : p , q )
IF ( R == NIL )
    p ← NIL ; q ← NIL
ELSE
    IF ( v == info(R) )
        p ← R ; q ← NIL
    ELSE
        IF ( v < info(R) )
                Search( v , lc(R) , p , q )
        ELSE
                Search( v , rc(R) , p , q )
        EndIf

        IF ( q == NIL )
            q ← R
        EndIf

    EndIf
EndIf
```

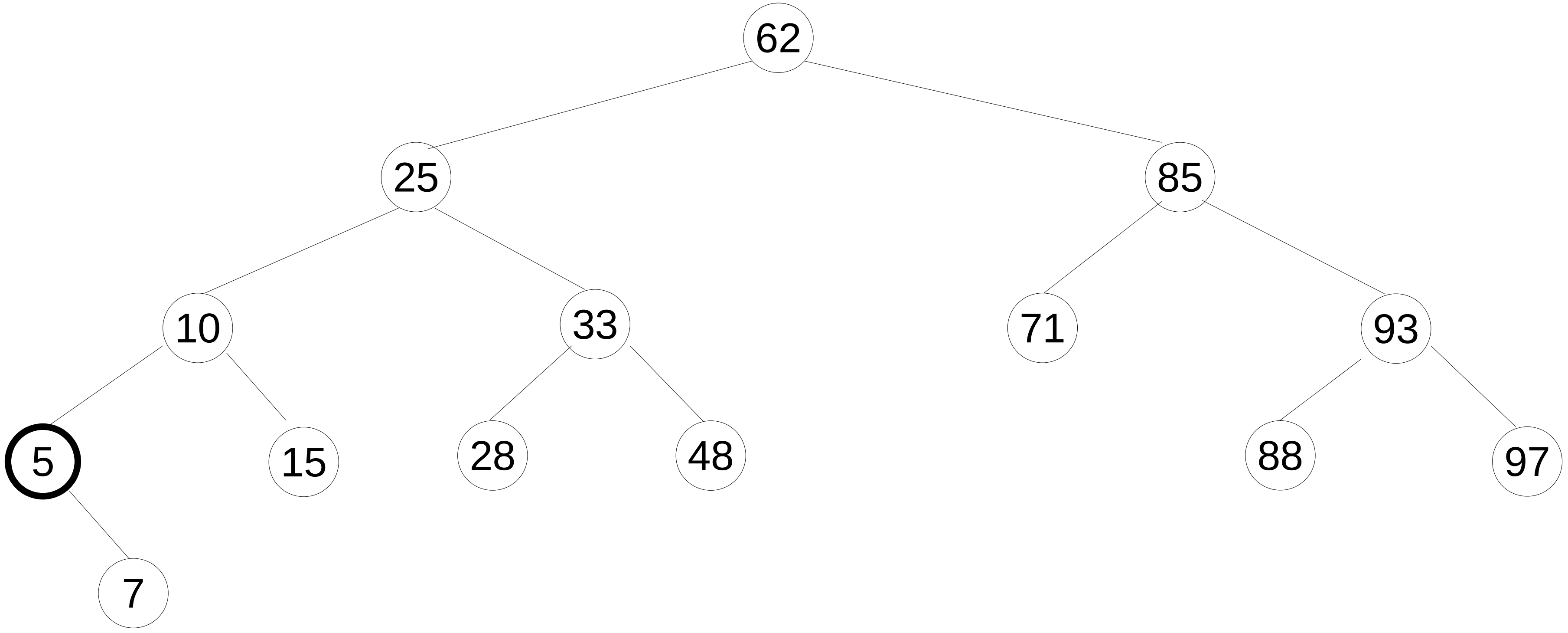


Searching 80 → (p=nil and q=node(71))

BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

In this example nodes will be visited as follows:
5, 7, 10, 15, 25, 28, 33, 48, 62, 71, 85, 88, 93, 97



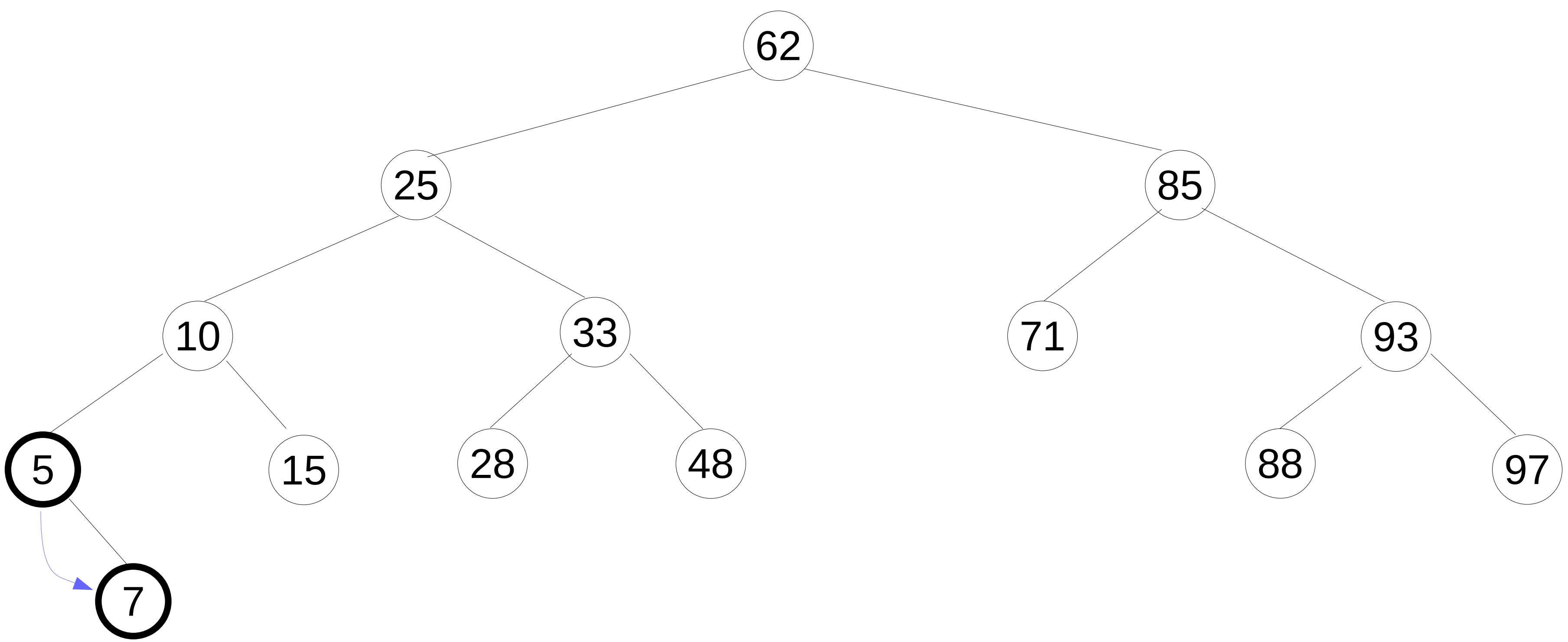
BST : Inorder Traversal

Property :

The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

In this example nodes will be visited as follows:

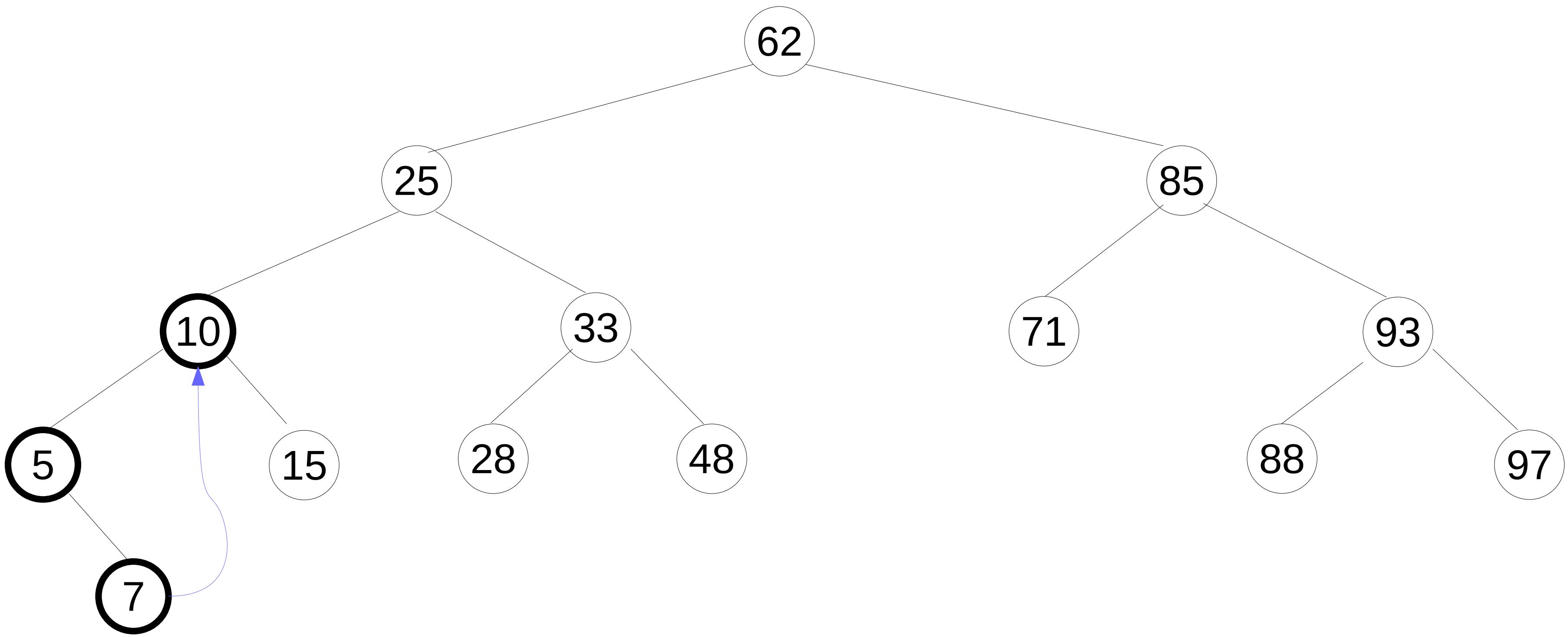
5, **7**, 10, 15, 25, 28, 33, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

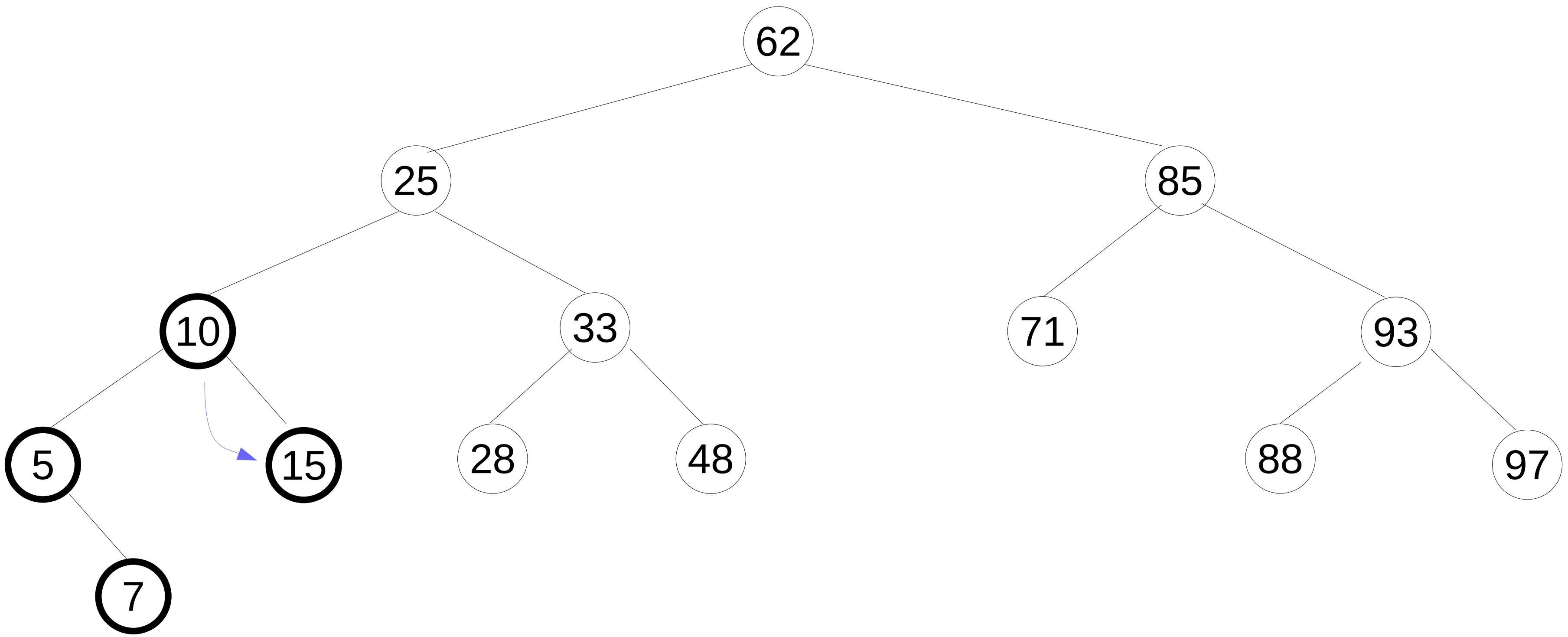
In this example nodes will be visited as follows:
5, **7**, **10**, 15, 25, 28, 33, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

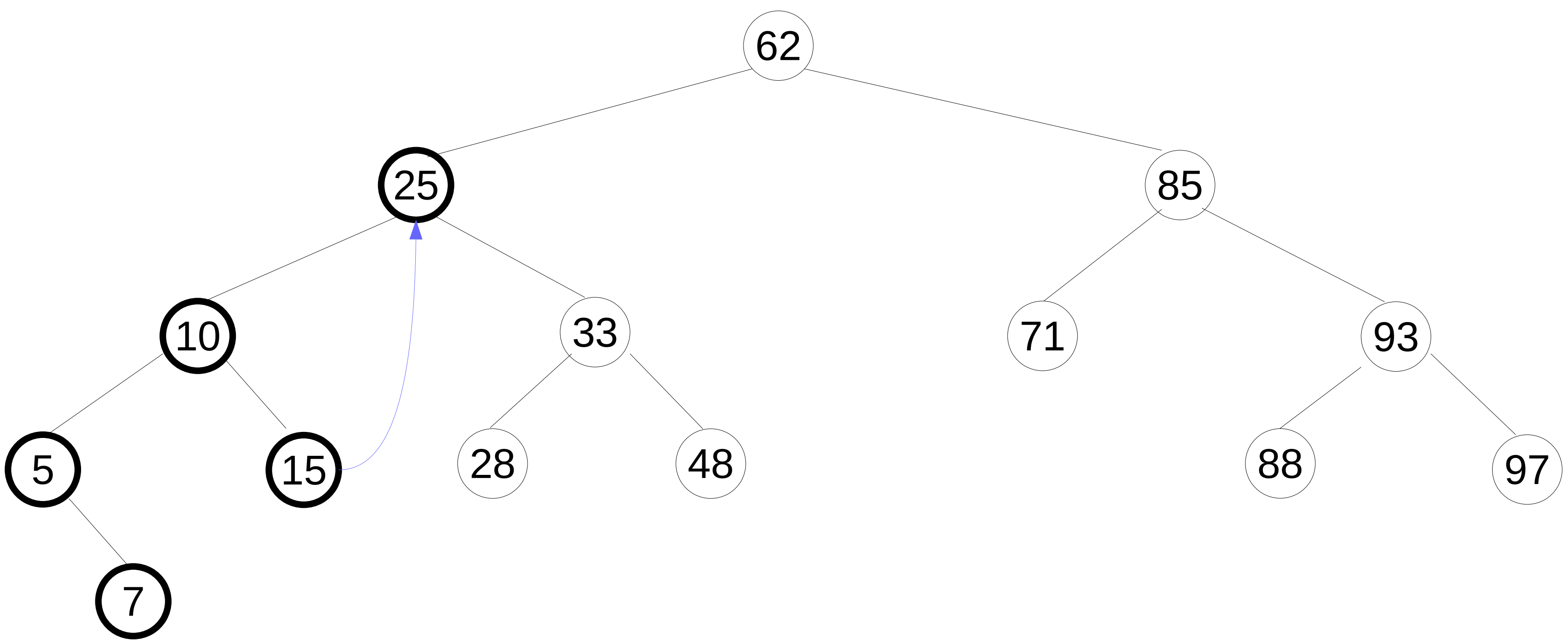
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, 25, 28, 33, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

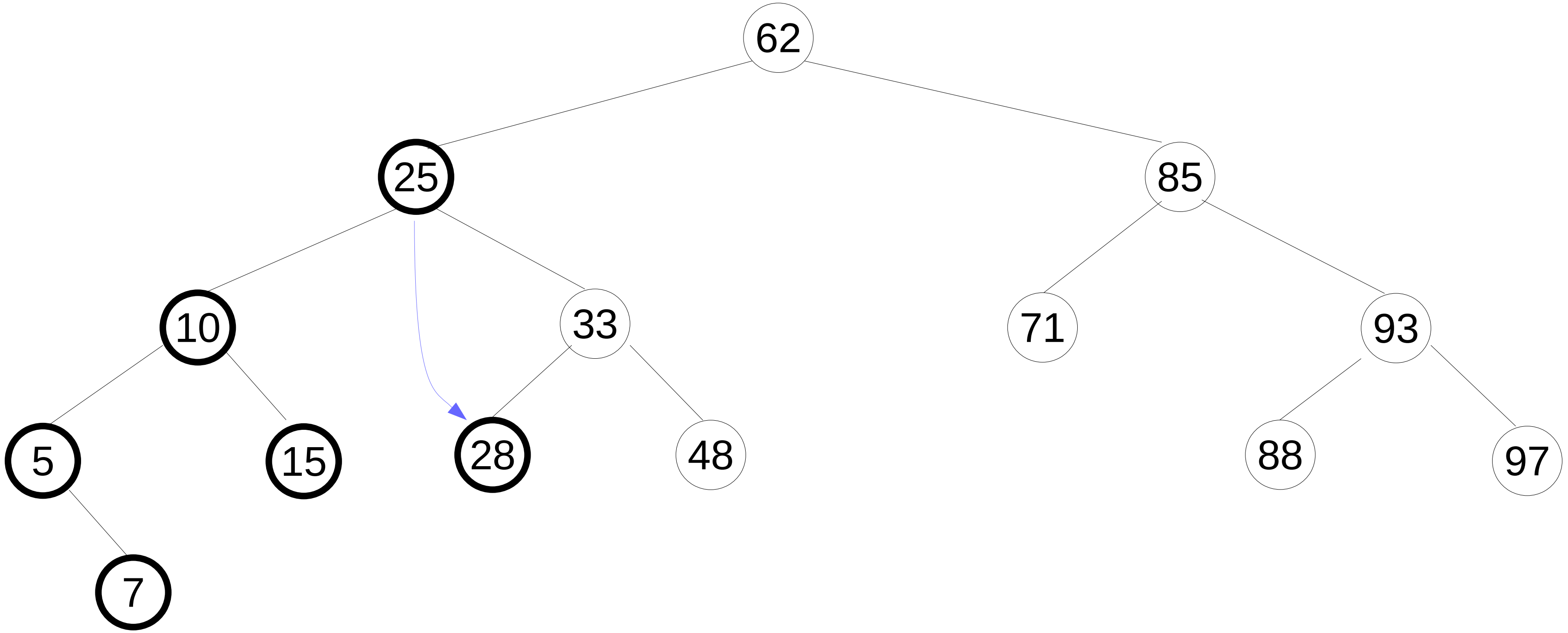
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, 28, 33, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

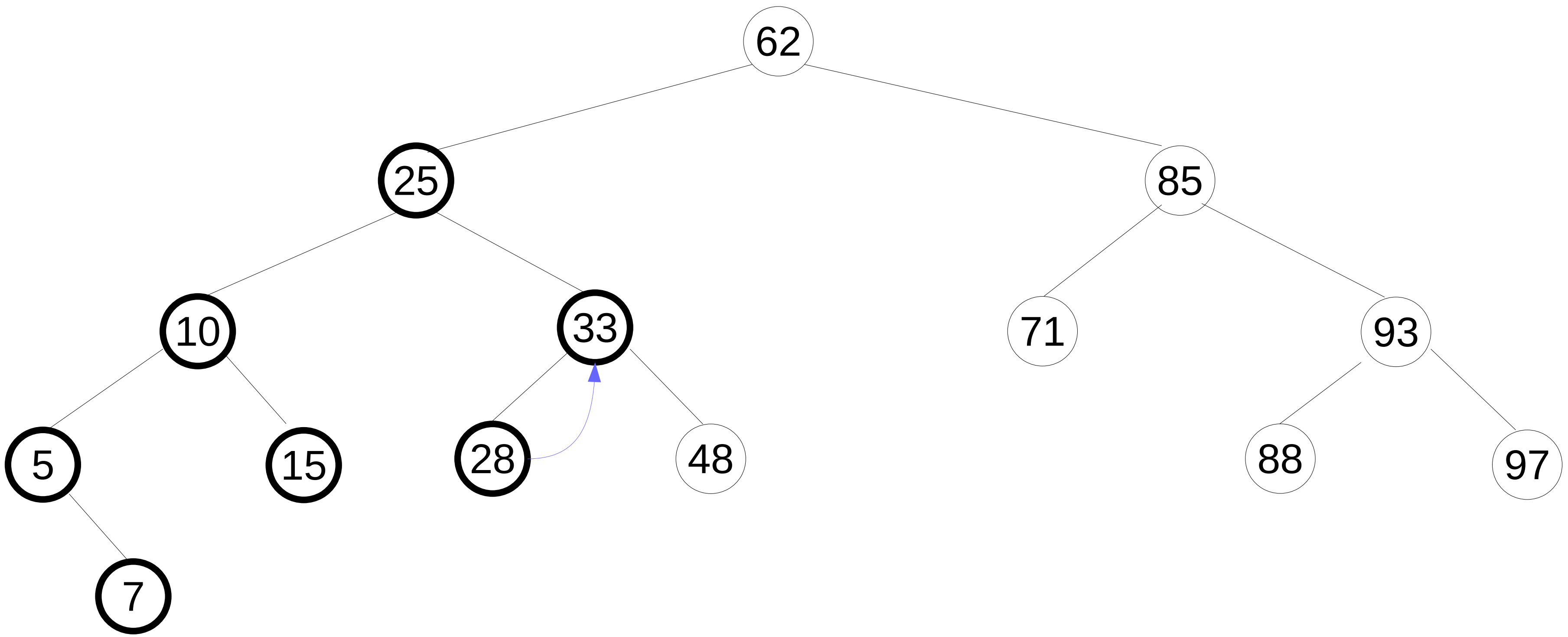
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, 33, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

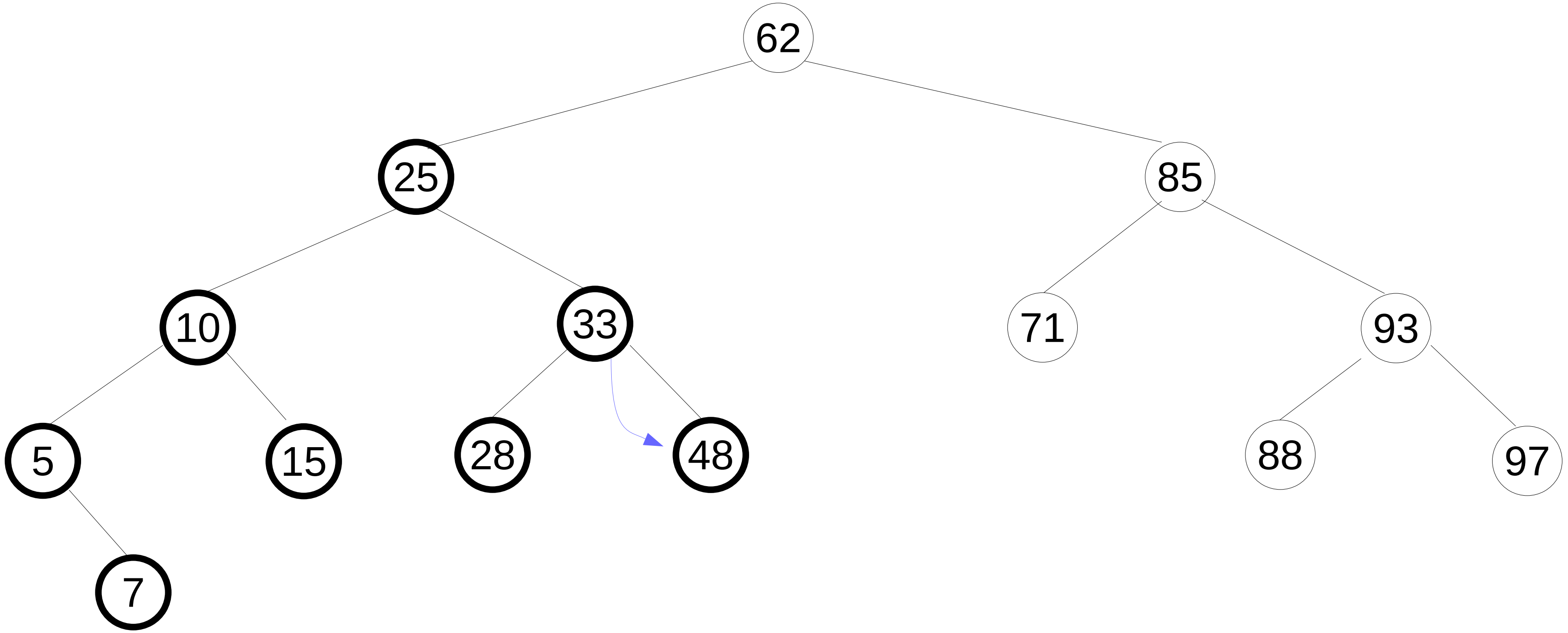
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, 48, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

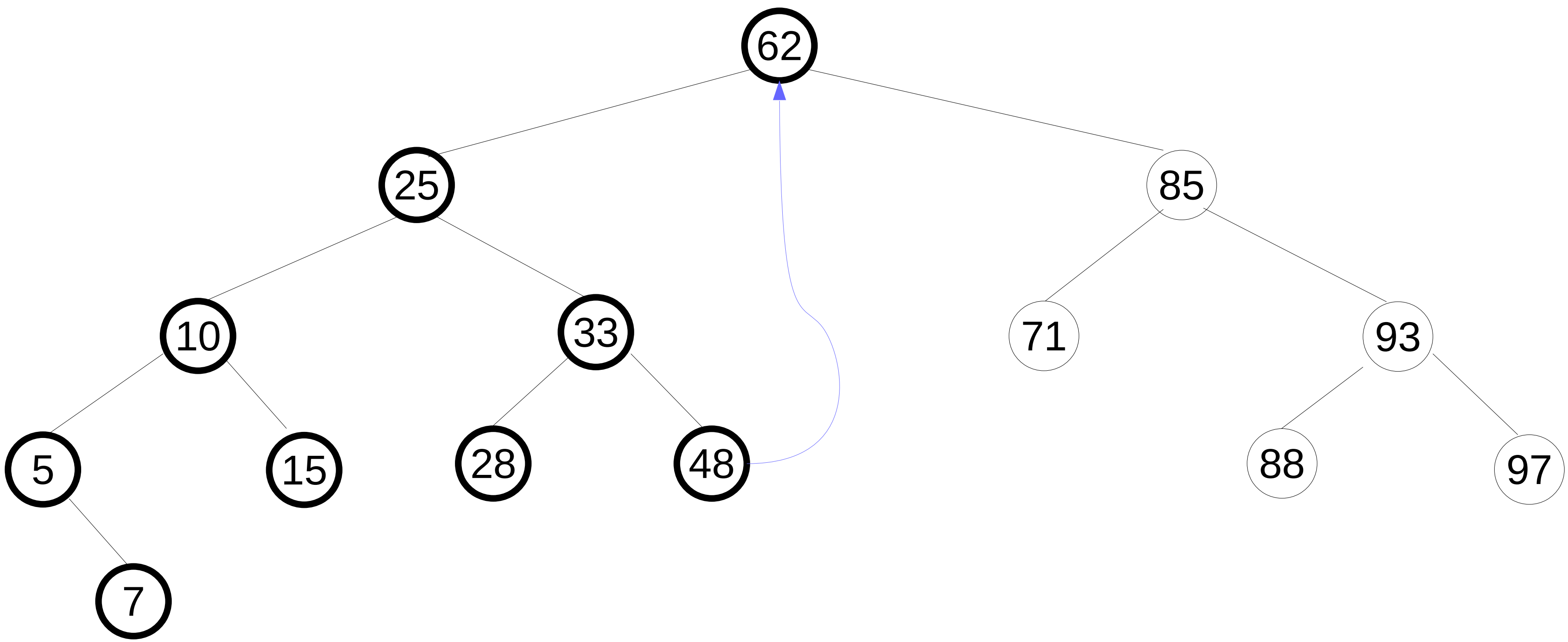
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, 62, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

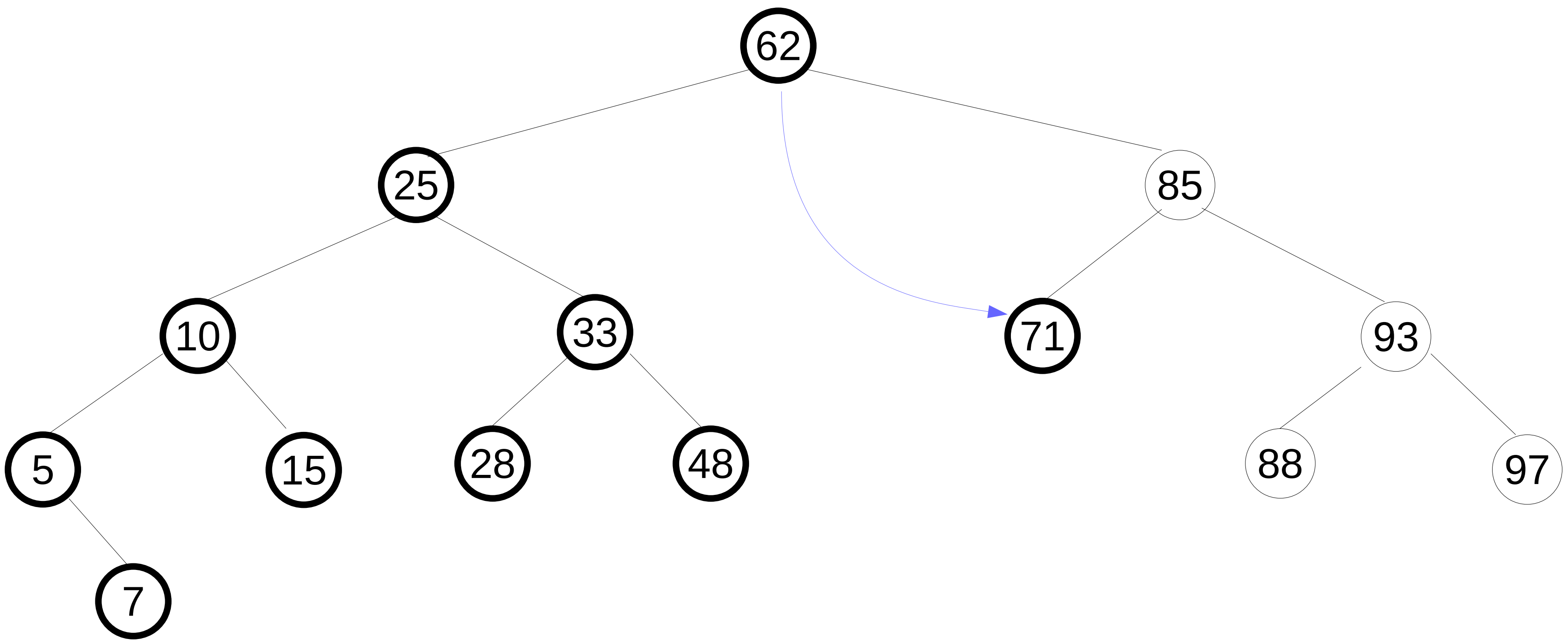
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, 71, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

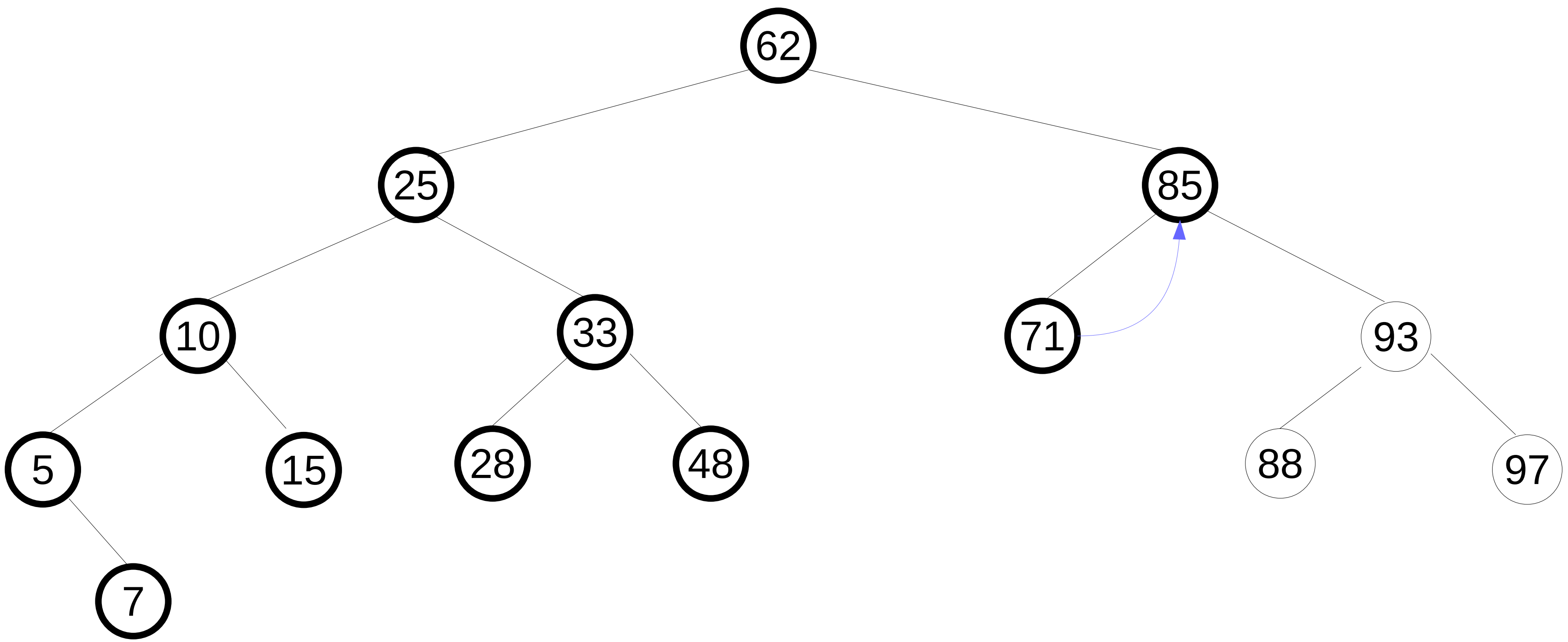
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, **71**, 85, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

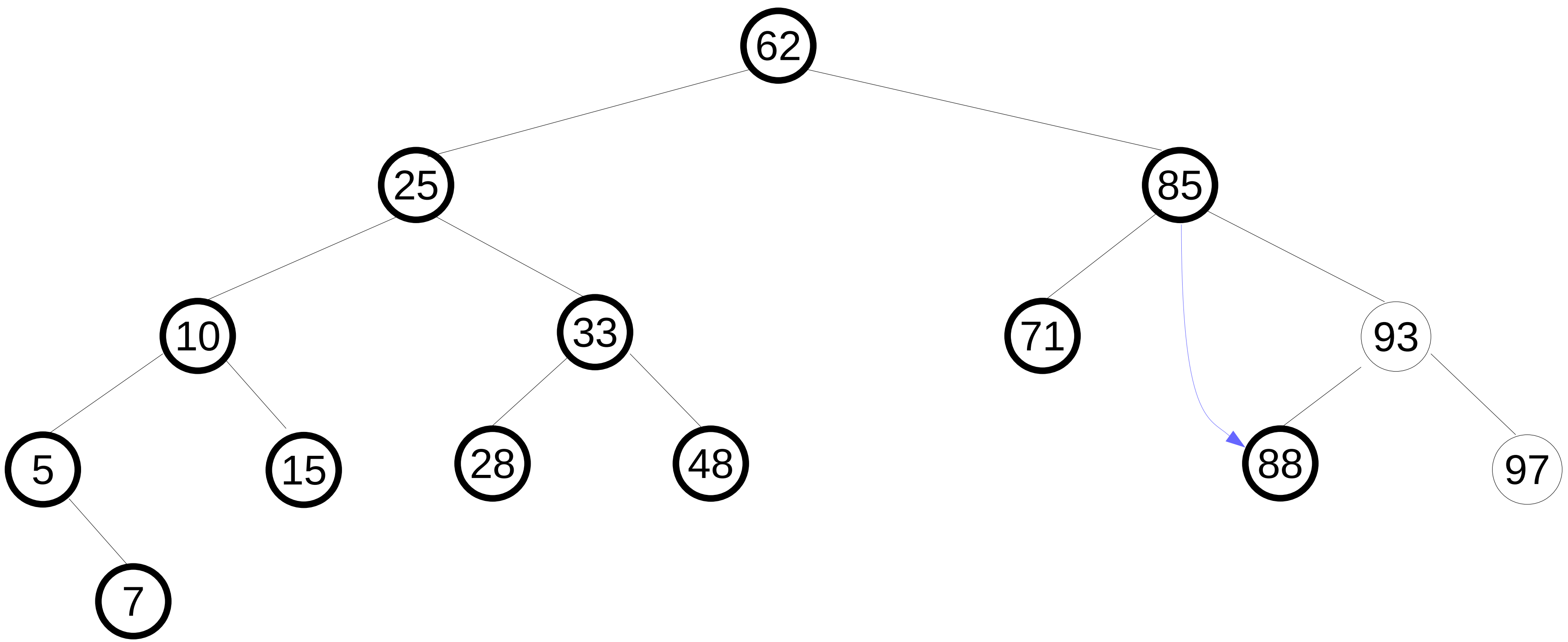
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, **71**, **85**, 88, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

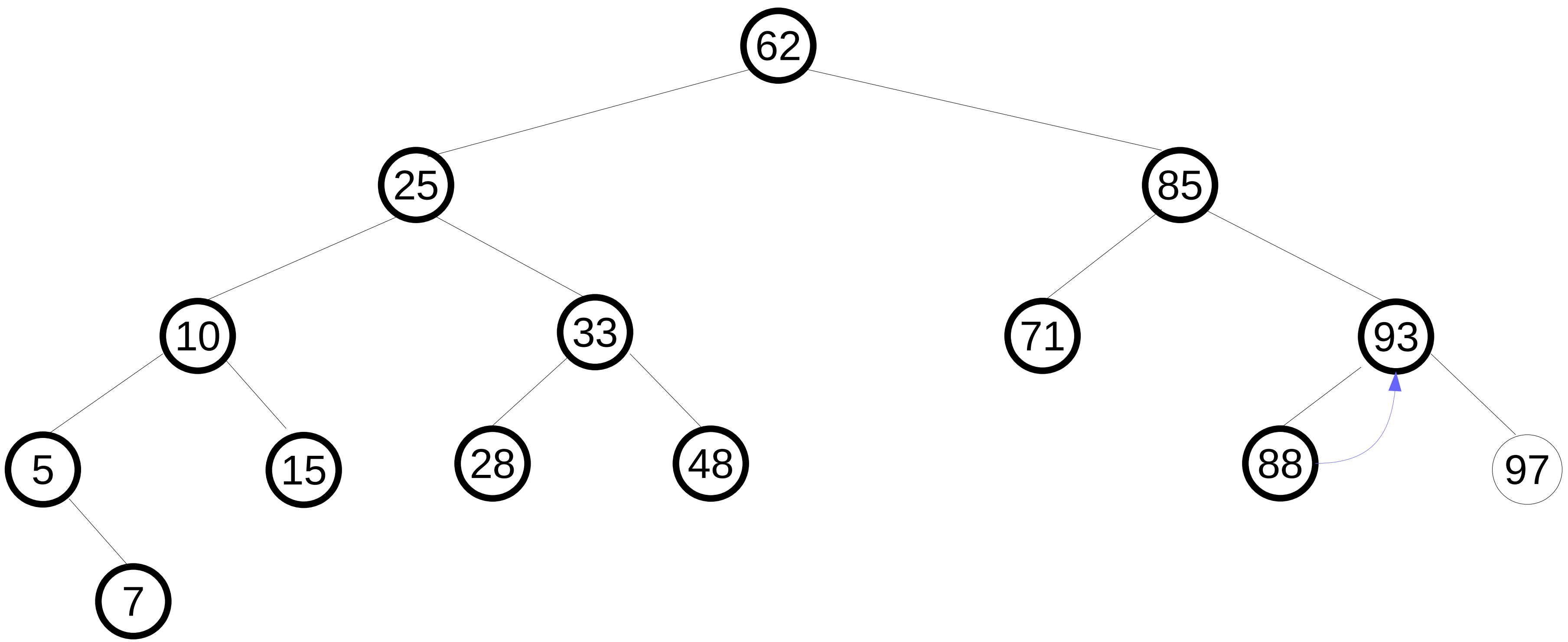
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, **71**, **85**, **88**, 93, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

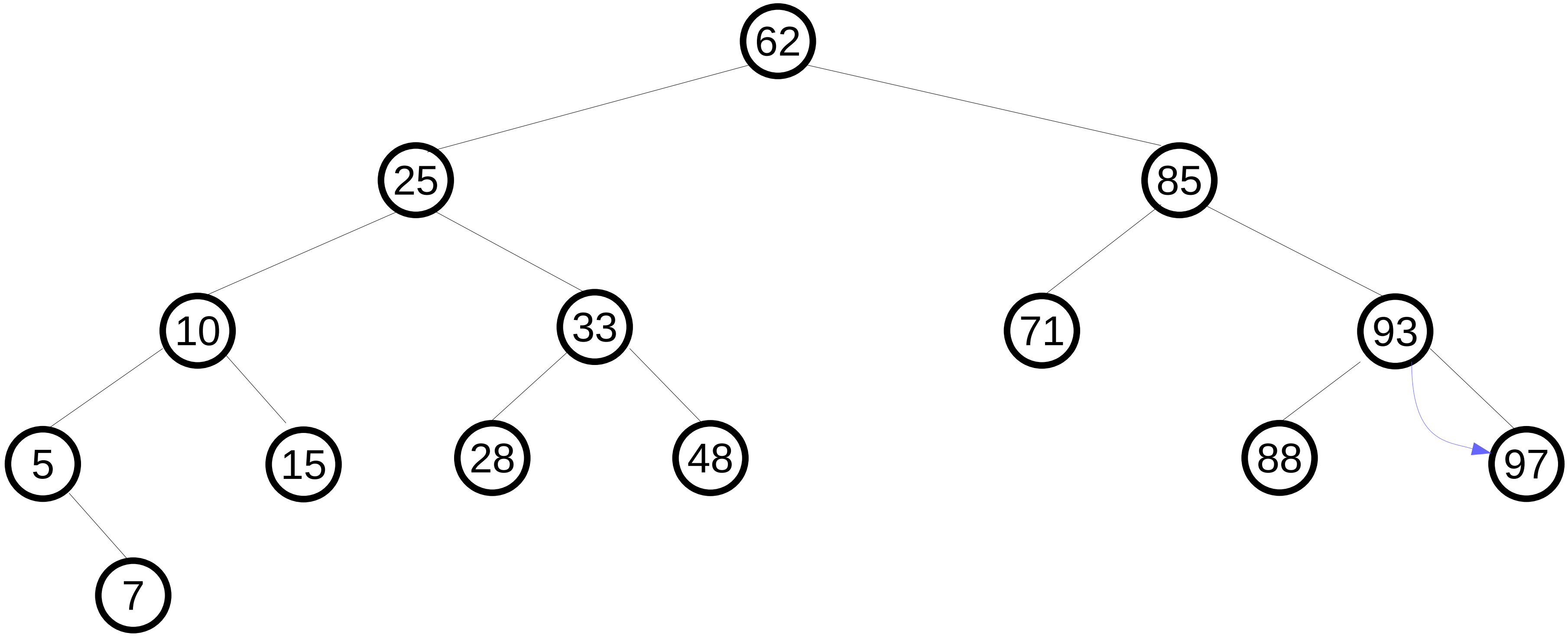
In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, **71**, **85**, **88**, **93**, 97



BST : Inorder Traversal

Property :
The **inorder traversal** of a binary search tree makes it possible to visit its values in **ascending order**.

In this example nodes will be visited as follows:
5, **7**, **10**, **15**, **25**, **28**, **33**, **48**, **62**, **71**, **85**, **88**, **93**, **97**



BST : Range Query

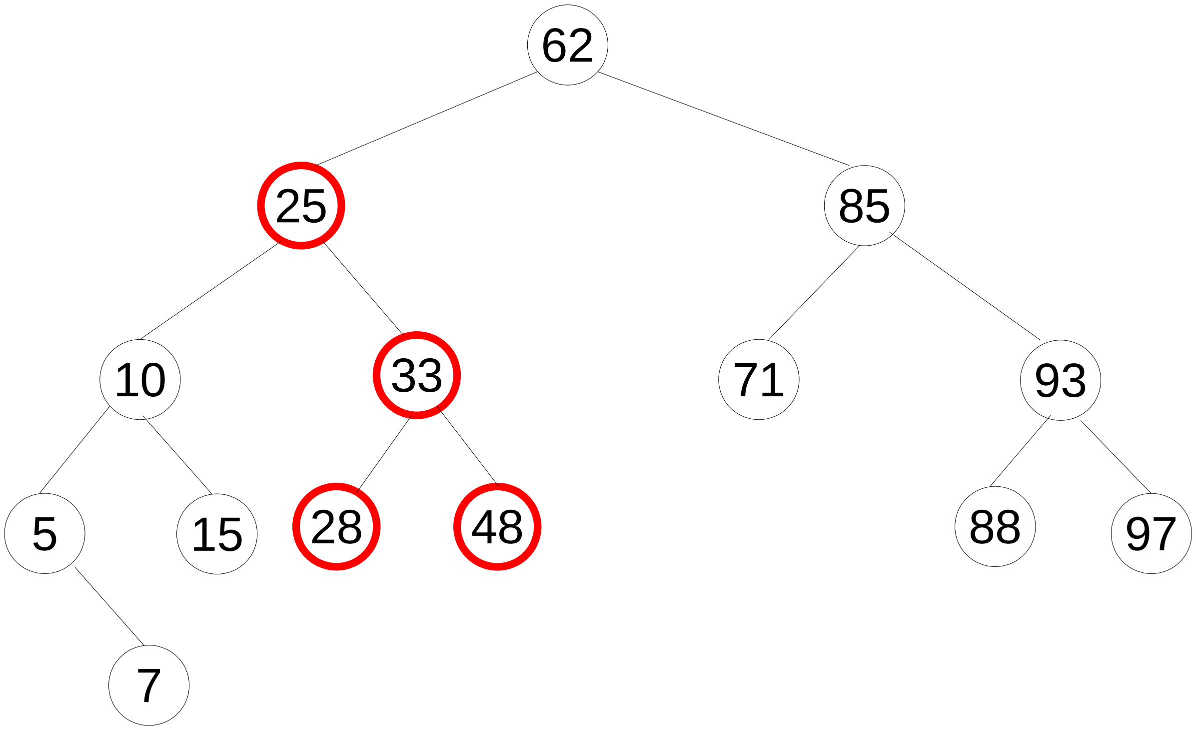
This is one of the <u>important applications</u> of **inorder traversal** in a Binary Search Tree

General form of a **range query**

→ given two bounds ***a*** and ***b***,
find <u>all the values</u> of the tree belonging to the interval ***[a,b]***

Example: find all values of the tree belonging to the interval ***[20 , 50]***

Result → 25 , 28 , 33 and 48



BST : Range Query

find <u>all the values</u> of the tree belonging to the interval ***[a,b]***

A *naive solution* (inefficient) → requires total traversal of the tree

inefficient_rangeQuery(R , a , b)

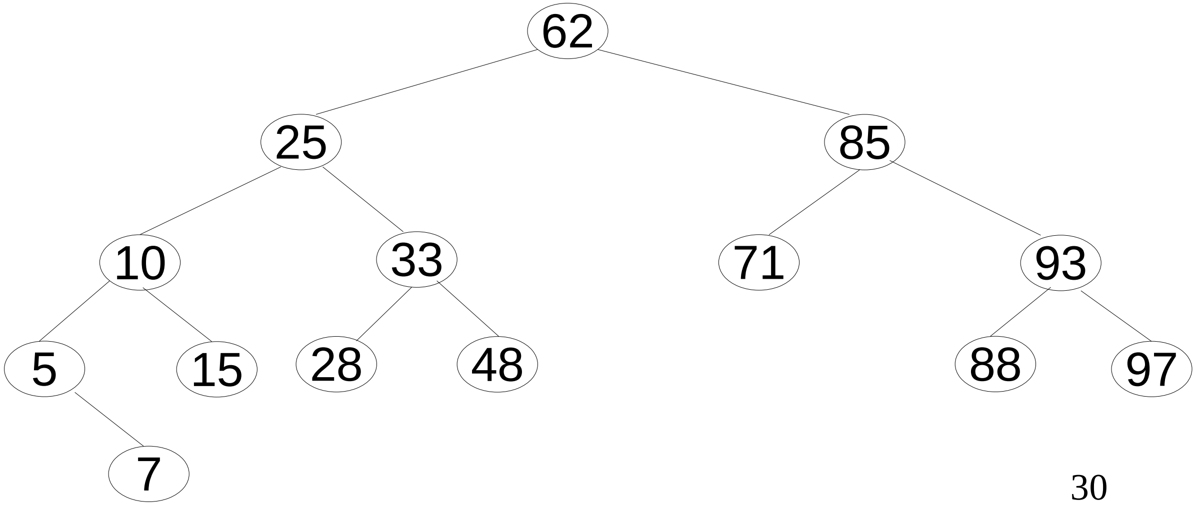
IF *(R ≠ NIL)*

***inefficient*_rangeQuery**(*lc(R) , a , b)*

IF *(info(R) ∈ [a,b])* ***process(R)*** ***EndIf***

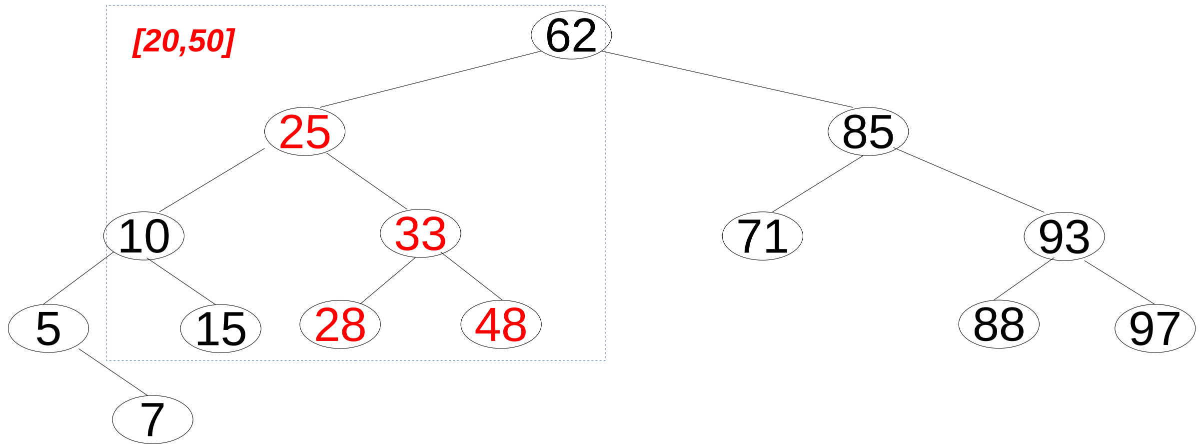
***inefficient*_rangeQuery**(*rc(R) , a , b)*

EndIf



BST : Range Query

A more efficient solution → an inorder traversal **limited** to the **region of interest**



Range query : find all the values of the tree belonging to the interval ***[a,b]***

rangeQuery(R , ***a*** , ***b***)

*1. search in R the smallest value greater than or equal to **a** → node n*
*2. **WHILE** (info(n) ≤ **b**)*
3. process(n)
4. n ← next_inorder(n)
*5. **EndWhile***

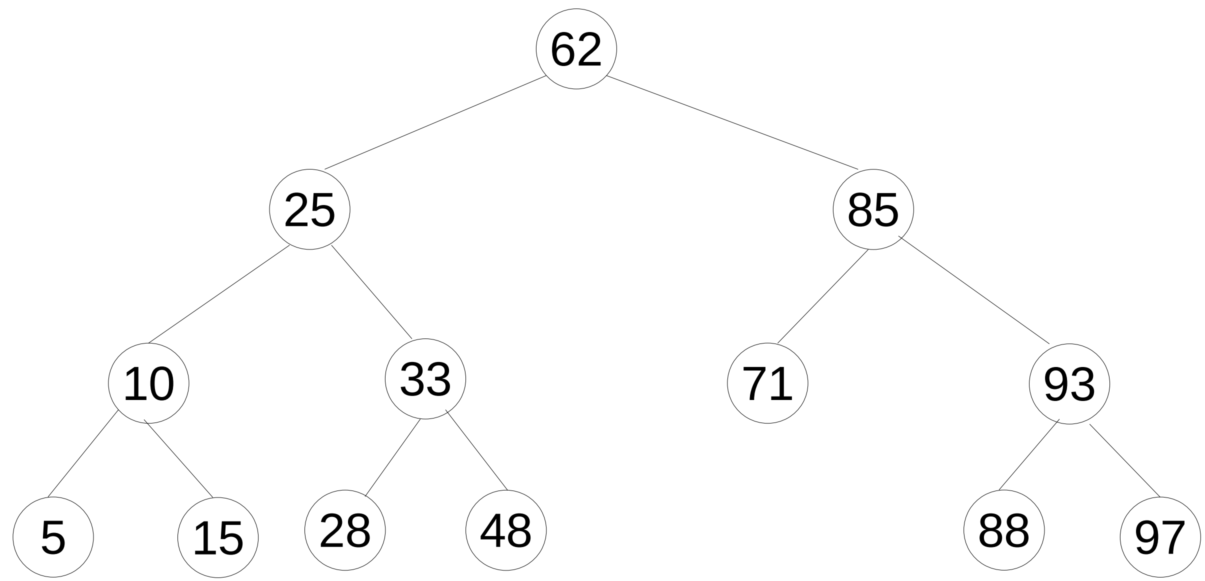
Line 1. ⇒ easy (using the BST search algorithm, slightly modified)
Line 4. ⇒ requires efficient implementation of the function : *next_inorder(n)*

BST : Insertion algorithm

Inserting a new value ***v***, consists of adding a **new leaf** at the end of the branch traversed by the search algorithm

Ex. insert 30 :

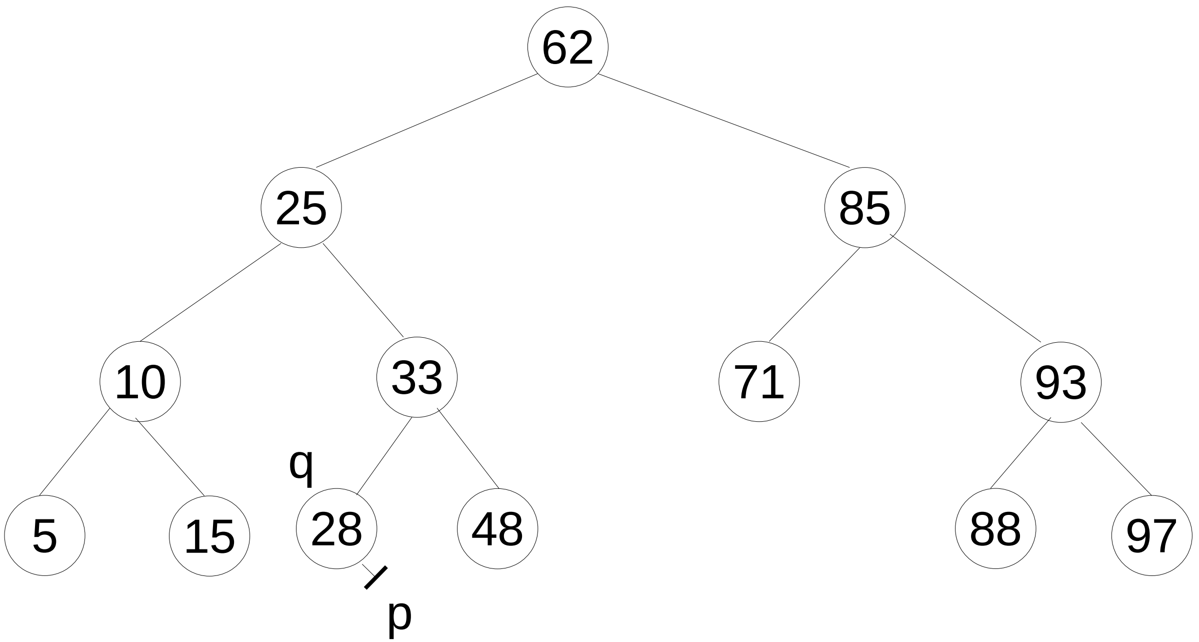
…



Ex. insert 30 :

1- Search(30) → (p , q)
p = **NIL** and q points node 28

…

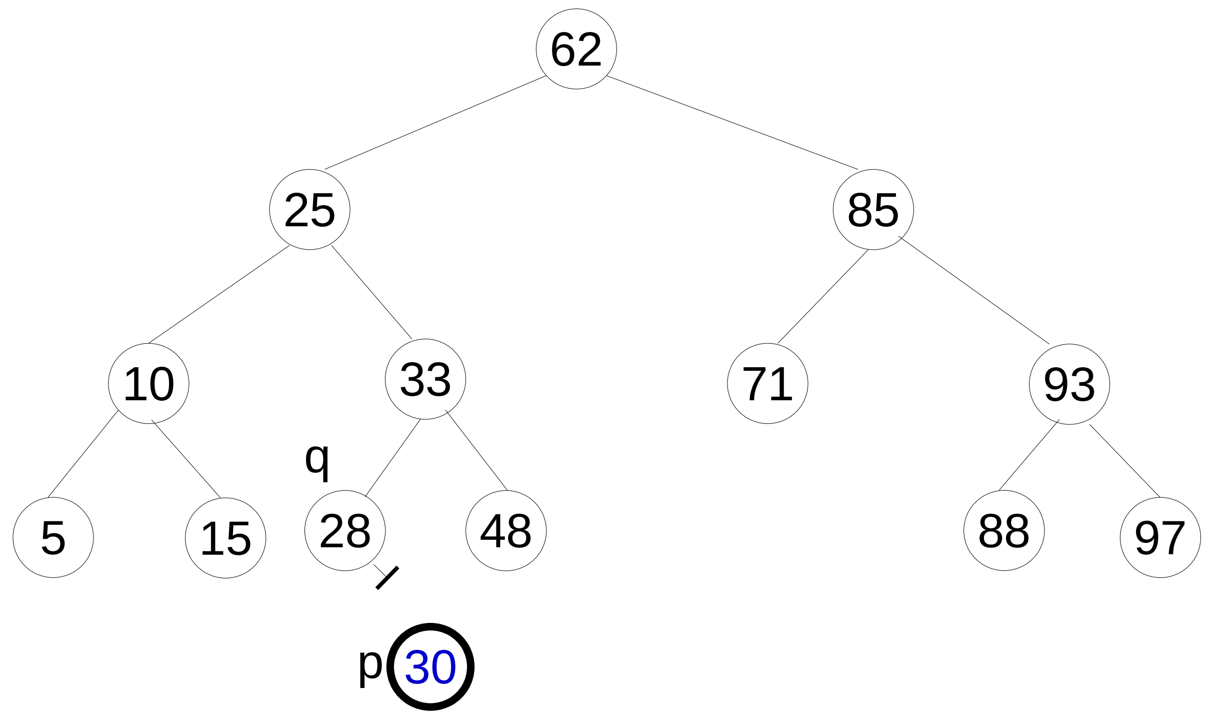


Ex. insert 30 :

1- Search(30) → (p , q)
    p = NIL and q points node 28

2- p ← **createTNode**(30)

…

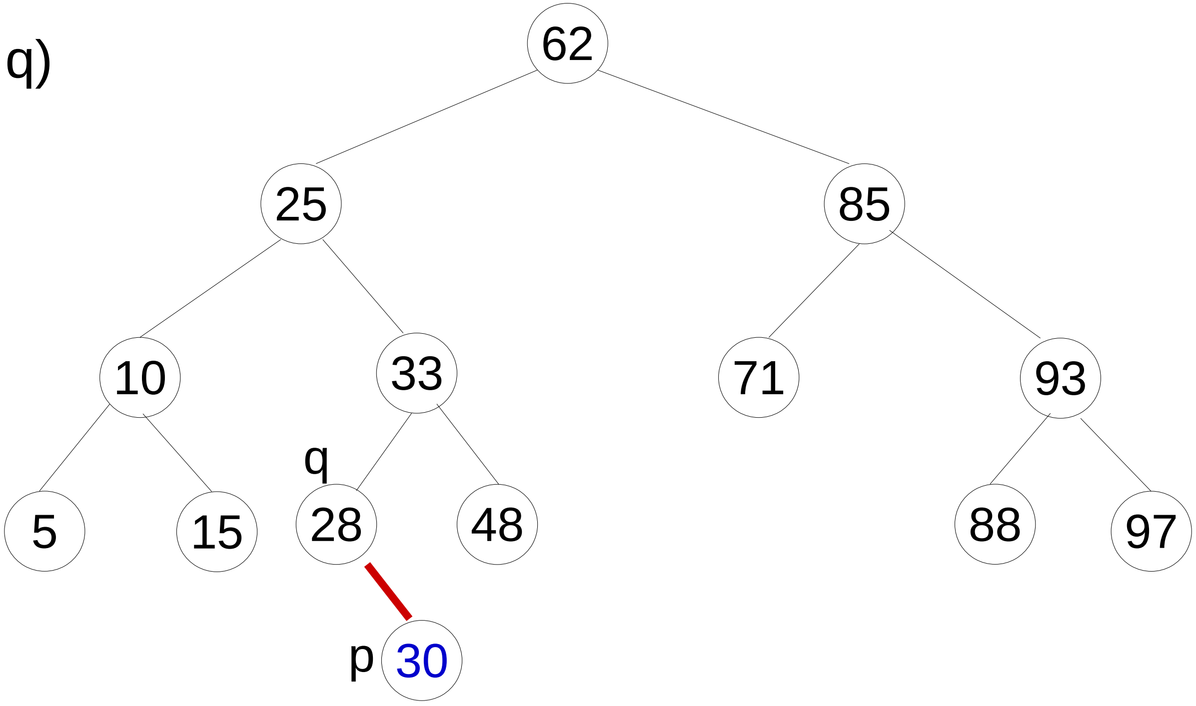


Ex. insert 30 :

1- Search(30) → (p , q)
p = NIL and q points node 28

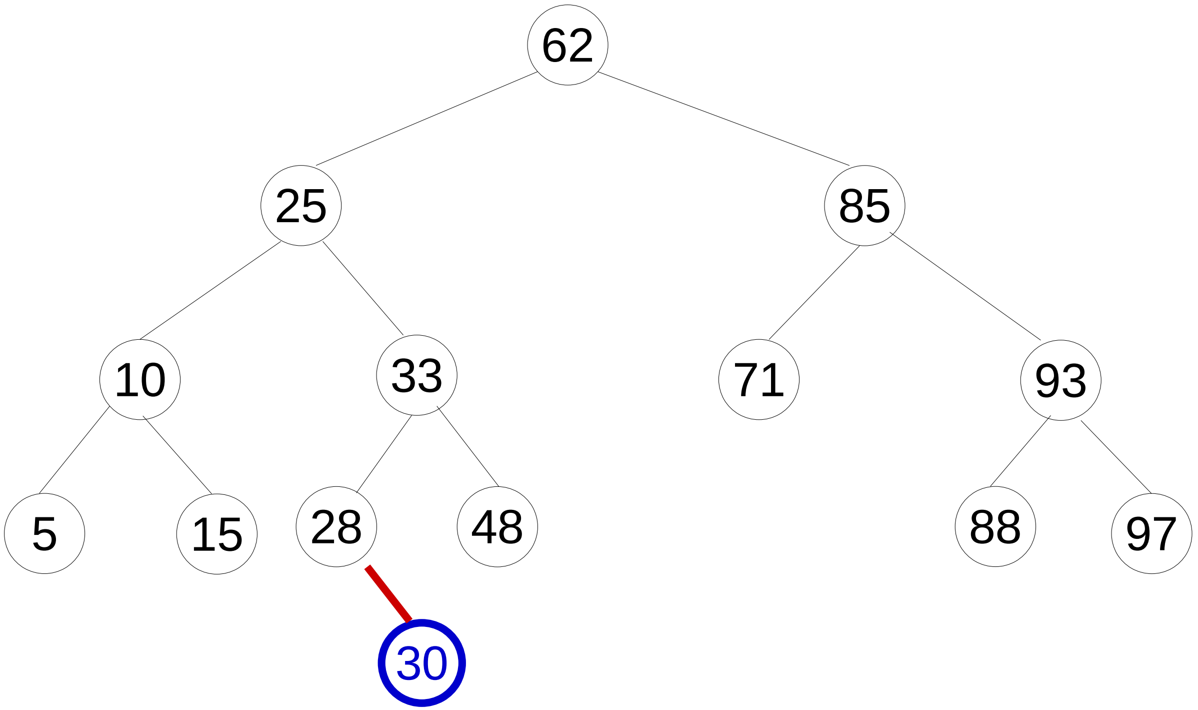
2- p ← createTNode(30)

3- connect the new node p to the tree (through the last visited node q)
set_rc(q , p)



BST : Insertion algorithm (recursive version)

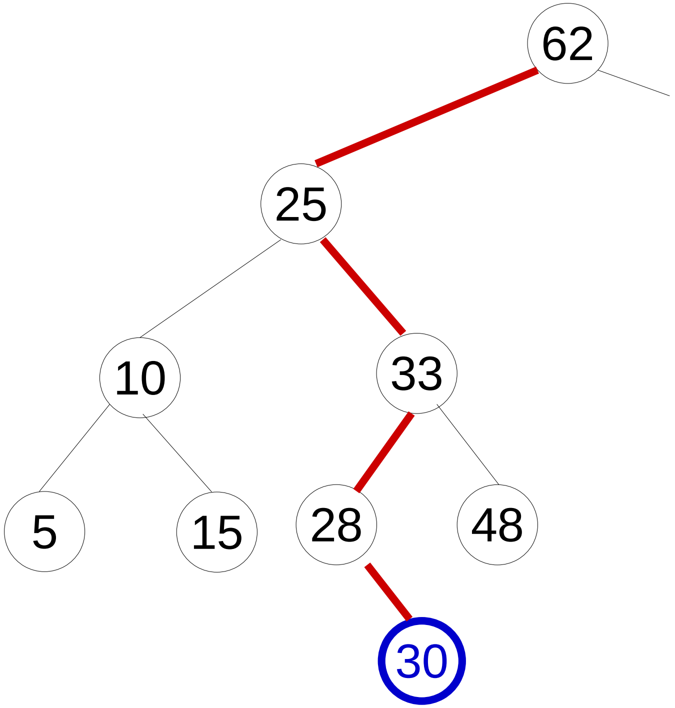
```
Ins( in : v , out : R ) : bool
  IF ( R == NIL )
        R ← createTNode(v)
        return true

  ELSE
        IF ( v == Info( R ) )
            return false
        EndIf

        IF ( v < Info(R) )
            p ← lc(R)
            Res ←  Ins( v , p )
            set_lc( R  p )
        ELSE
            p ← rc( R )
            Res ← Ins( v , p )
            set_rc( R , p )
        EndIf
        return Res
  EndIf
```



```
Ins( in : v , out : R ) : bool
   IF ( R == NIL )   R ← createTNode(v)
                     print("New node → " , v)
                     rerturn true
   ELSE
          IF ( v == Info( R )   return false  EndIf
          IF ( v < Info(R) )
                p ← lc(R)
                Res ← Ins( v , p )
                set_lc( R  p )
                print("set_lc : ",Info(R) , "→" , Info(p))
          ELSE
                p ← rc( R )
                Res ← Ins( v , p )
                set_rc( R , p )
                print("set_rc : ",Info(R) , "→" , Info(p))
          EndIf
          return Res
   EndIf
------------------------------
```

Execution trace :

New node → 30
set_rc : 28 → 30
set_lc : 33 → 28
set_rc : 25 → 33
set_lc : 62 → 25



BST : Deletion algorithm

Removing a <u>value ***v***</u> from the tree ⇒ freeing a **node**
Tree nodes should stay connected and the order of remaining values should not be disturbed

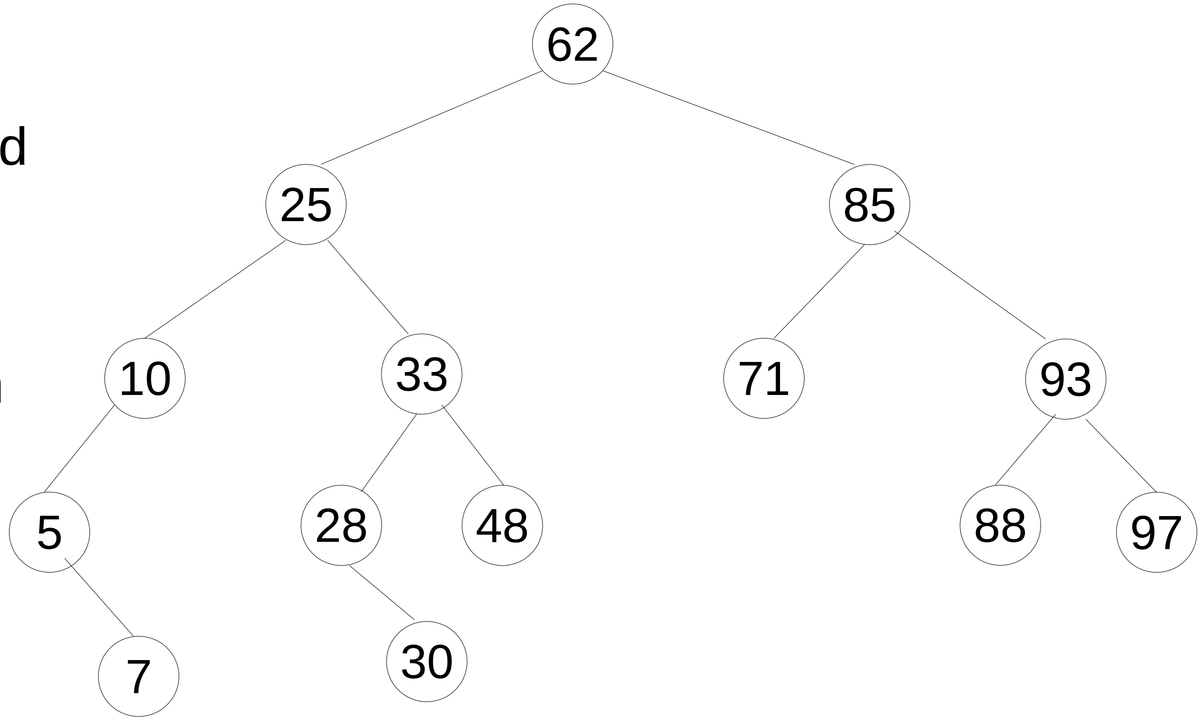
to **delete *v*** :

1) Search ***v*** → *p* and *q* (the node containing ***v*** and its parent)

2) Then there are **2 cases** to be considered:

2a) If *p* has at least one NIL child
→ update *q* (the parent of *p*) and
→ free *p*

2b) Else // *p* has no NIL children
→ Replace the value ***v*** in *p* with
that of its <u>**next inorder**</u>: *p'*
→ Free the node p'



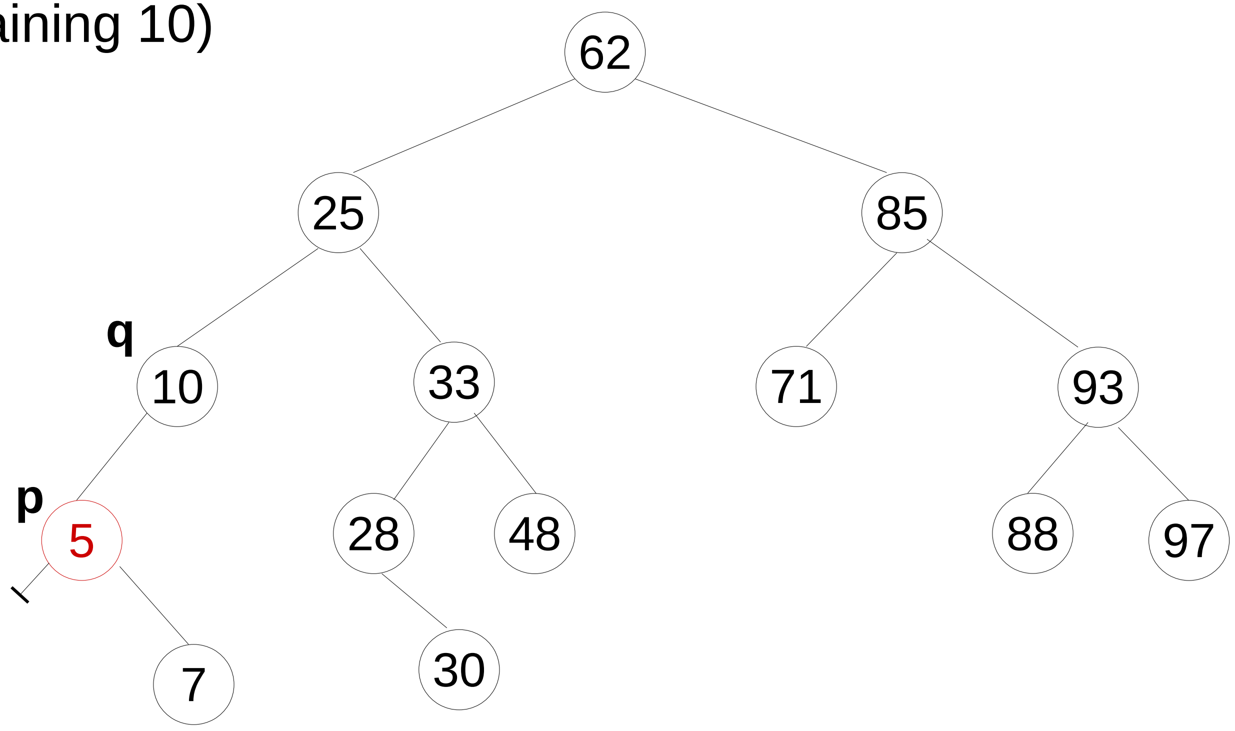
Exemples :
Suppression de 5
Suppression de 25

Example 1 : delete 5

1) Search the value 5

→ **p** : the node containing 5

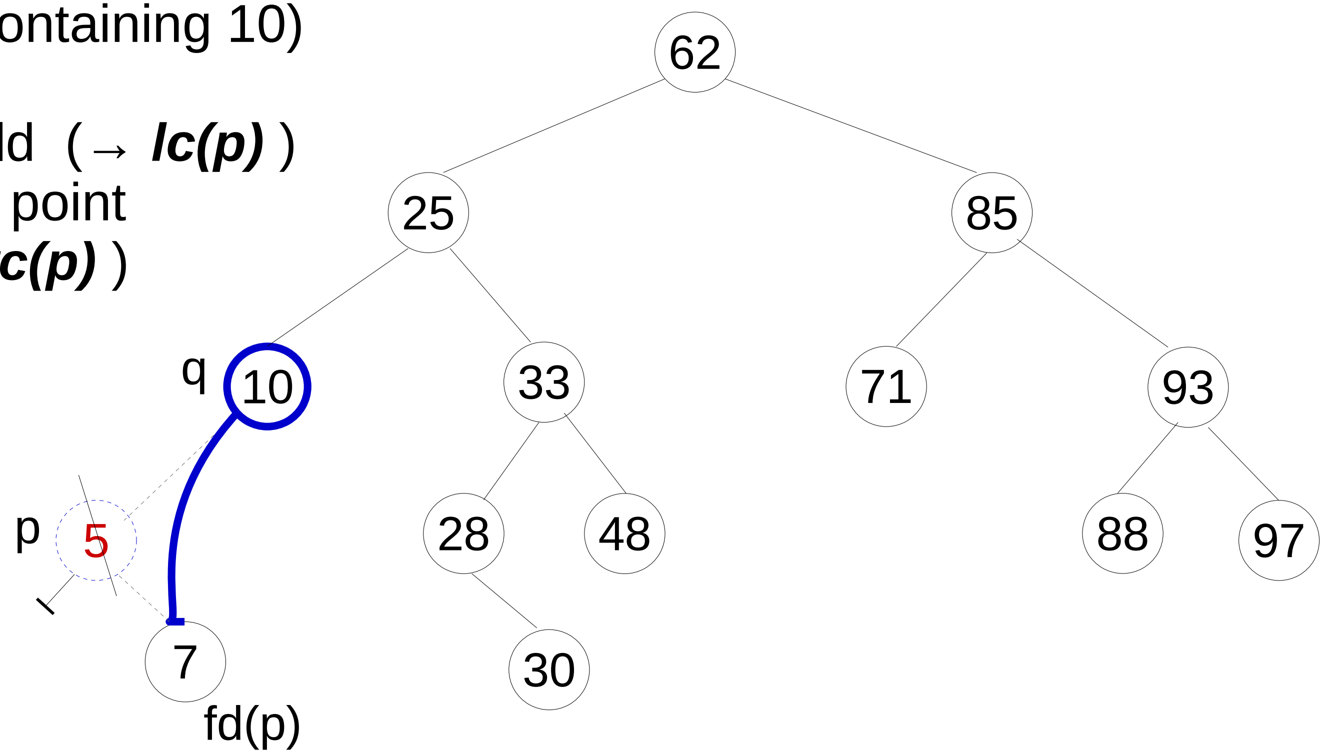
→ **q** : the parent node (containing 10)



Example 1 : delete 5

1) Search the value 5
→ **p** : the node containing 5
→ **q** : the parent node (containing 10)

2a) **p** has at least one NIL child (→ ***lc(p)***)
// update the parent (**q**) to point
// the other child of **p** (→ ***rc(p)***)
set_lc(q , rc(p))
// and free **p**
freeTNode(p)



Example 2 : delete 25

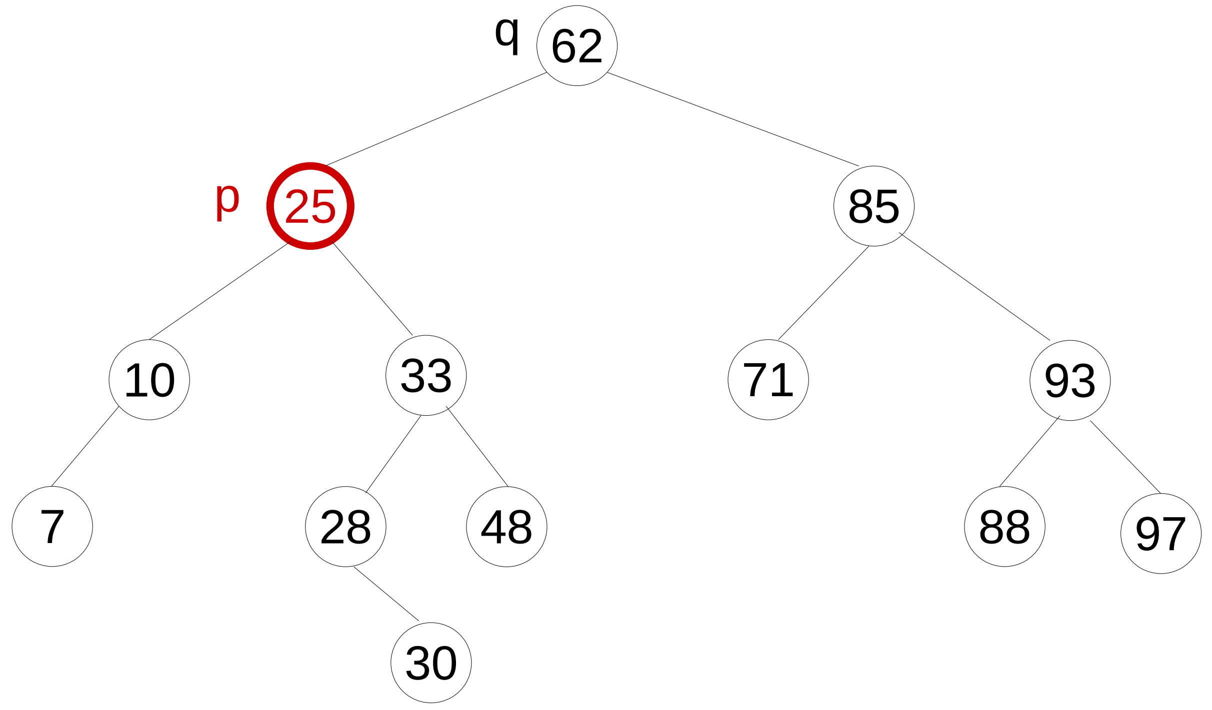
1) Search 25

→ **p** : the node containing 25

→ **q** : the parent node

2b) **p** has 2 non-empty sub-trees

(i.e. lc(p)≠NIL and rc(p)≠NIL)



Example 2 : delete 25

1) Search 25

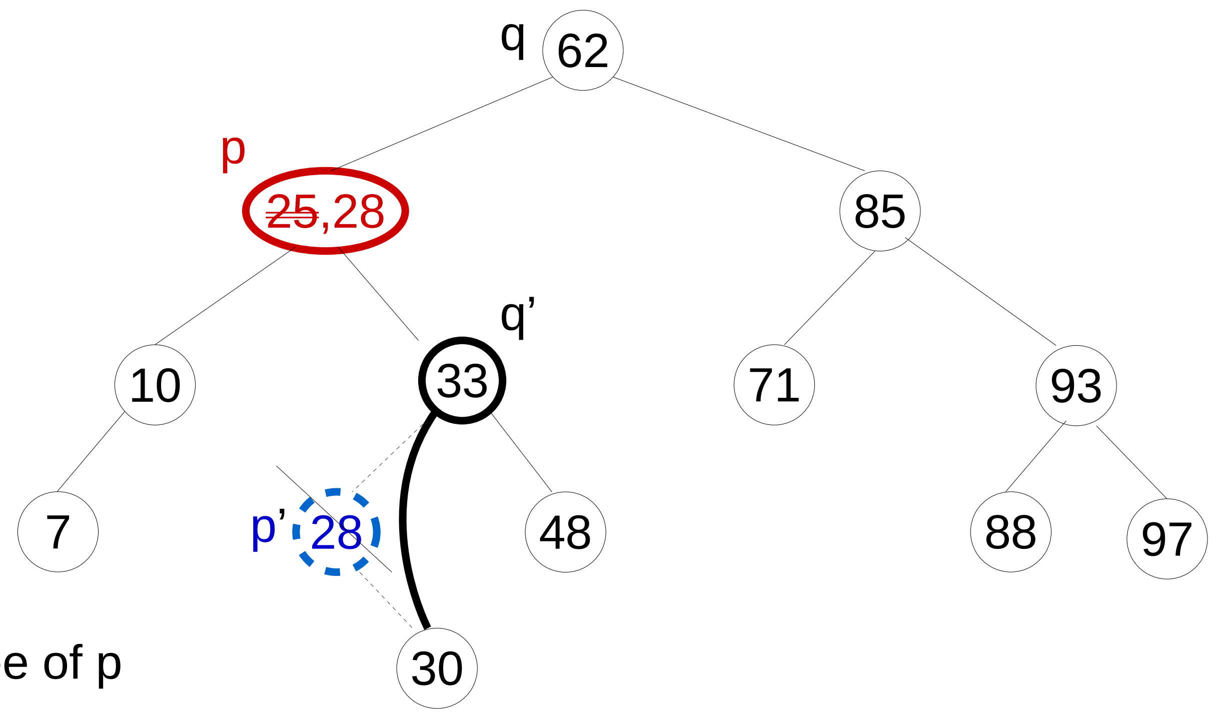
→ **p** : the node containing 25

→ **q** : the parent node

2b) **p** has 2 non-empty sub-trees

Replace 25 in **p** with **28**
(p' the **next inorder** of **p**)

The next inorder of a node p
with a right-child ≠ NIL is:
The leftmost node of the right subtree of p



Free **p'** and update its parent **q'** to point rc(**p'**)

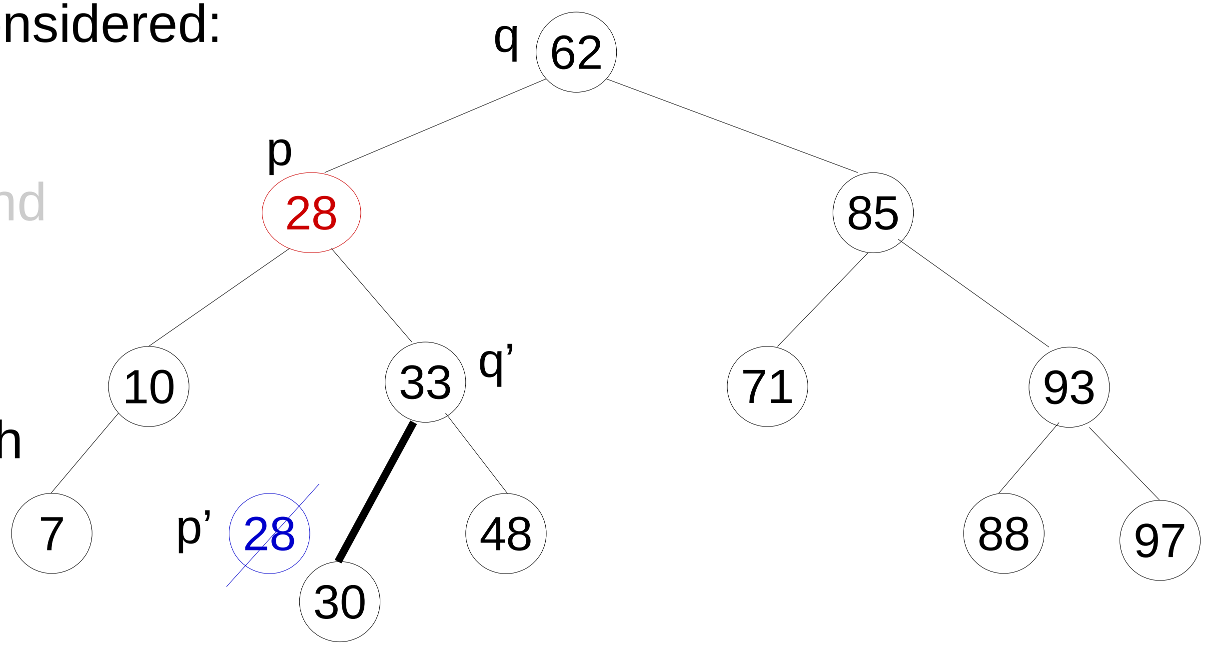
Example 2 : delete 25

1) Search ***v*** → *p* and *q* (the node containing ***v*** and its parent)

2) Then there are **2 cases** to be considered:

2a) If *p* has at least one NIL child
→ update *q* (the parent of *p*) and
→ free *p*

2b) Else // *p* has no NIL children
→ Replace the value ***v*** in *p* with
that of its **<u>next inorder</u>**: *p'*
→ Free the node p'



Implementing Trees in Contiguous Memory Areas (arrays)

1) full representation

Nodes are represented inside a table where each cell contains at least 3 fields: the value of the node (information), the left child and the right child (integers)

The pointers are therefore the indices of the table (-1 for NIL)

createTNode(v) : retrieves an empty cell and returns its index :
Efficient list management of empty cells :
→ get the head of the empty cells list *ECL* O(1)
p ← *ECL* ; *ECL* ← T[p].lc ; T[p] ← (-1 , v , -1)
return p

T

	lc	info	rc
1	-1		
2	-1	c	-1
3	8	d	-1
4	6	a	2
5	1		
6	-1	b	3
7	5		
8	-1	e	-1

ECL

freeTNode(p) : make cell p empty
→ insert cell p at the beginning of the list *ECL* O(1)
T[p].lc ← *ECL* ; *ECL* ← p

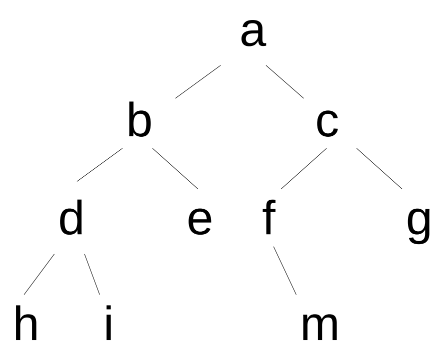
set_info(p,v) : T[p].info ← v
set_lc(p,q) : T[p].lc ← q
set_rc(p,q) : T[p].rc ← q
Info(p) *:* returne T[p].info
lc(p) : return T[p].lc
rc(p) : return T[p].rc

2) Sequential Représentation

Nodes are represented inside a table, but positions (indexes in the tables) are fixed and reserved once and for all.

- the root of the tree is at position 1 (index 1) (indice 1)
- the left child of a node at index i, is always at index 2i
- the right child of a node at index i, is always at index 2i+1
- the parent of a node at index i, is always at index i div 2
  (for i > 1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	c	d	e	f	g	h	i				m			

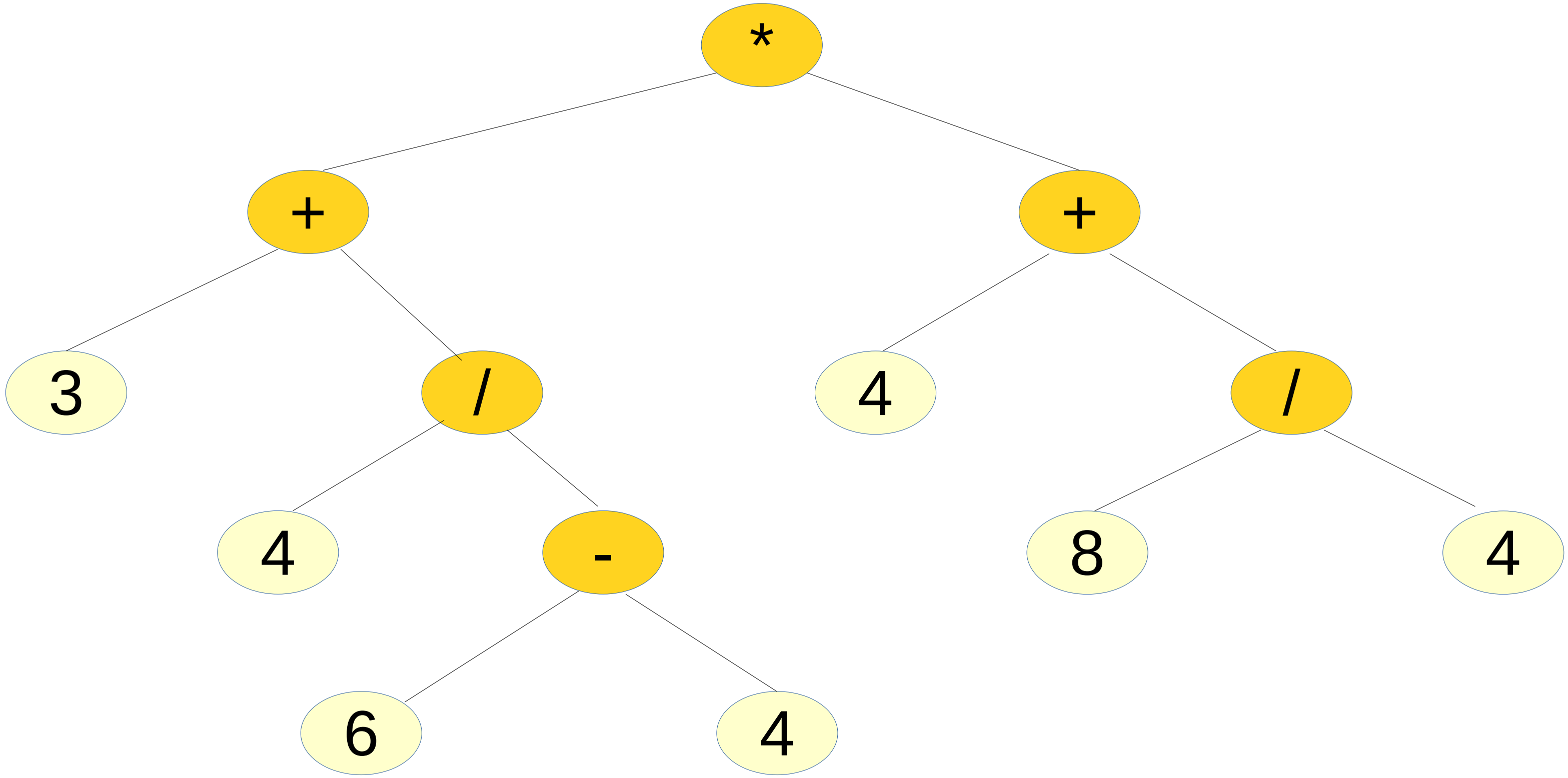


Application Examples

1) Representation of arithmetic expressions

internal nodes ⇒ operators

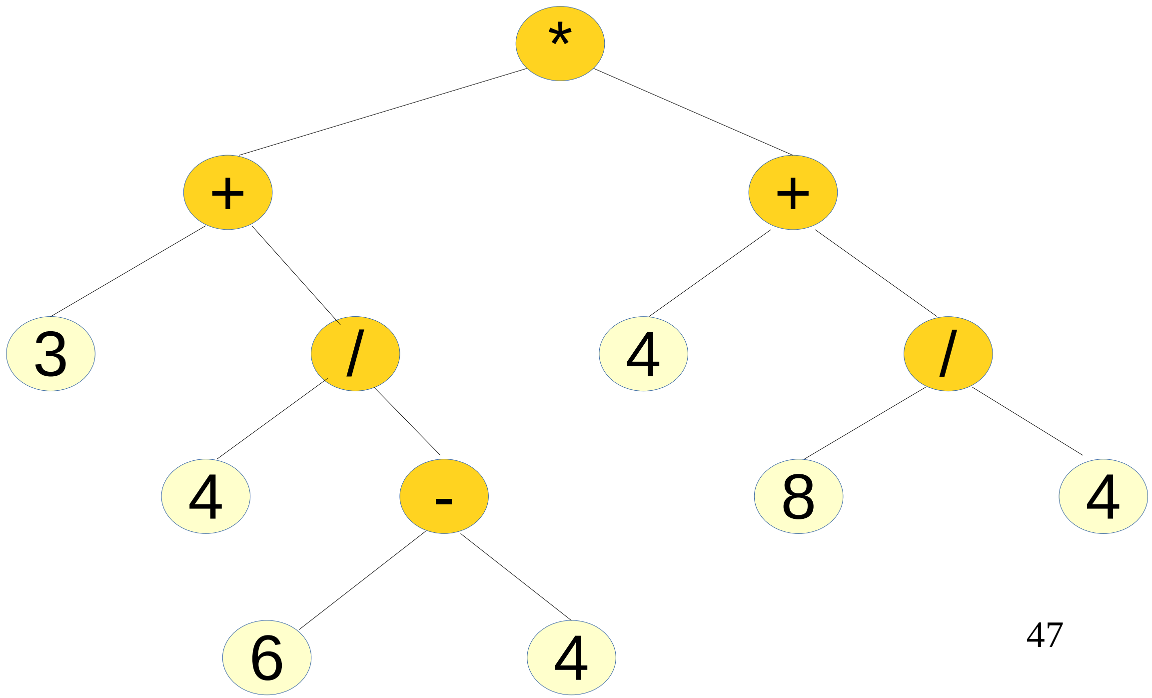
Example : (3 + 4 / (6 – 4)) * (4 + 8 / 4)



Algorithm evaluating expressions represented as a binary tree

```
Eval( r:ptr ) : tval /* int, float, … */
IF ( r == NIL )
    return 0
ELSE
    IF ( lc( r ) == NIL  et  rc( r ) == NIL )
        // leaf node...
        return Info(r)
    ELSE
        // internal node ...
        return oper( Info(r) , Eval( lc(r) ) , Eval( rc(r) ) )
    EndIf
EndIf
```

```
oper( op:char, g , d:tval ) : tval
switch(op)
    case '+' : return g+d
    case '-'  : return g-d
    case '*'  : return g*d
    case '/'  : return g/d
    …
```



2) using a binary tree to speed up finding positions in a list

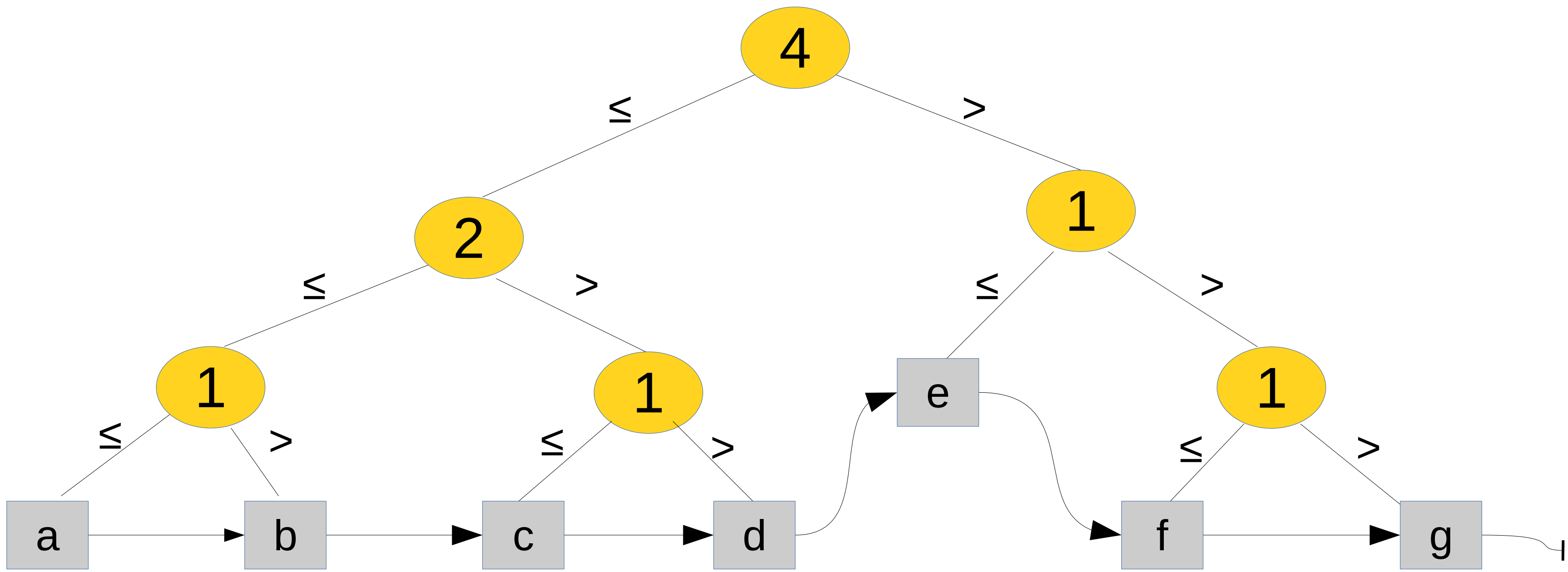
Leaves level :

→ the given linked list

Internal nodes :

→ a kind of BST ordered by the positions of the linked-list nodes

one can also consider only the **number of leaves in each left subtree** as in the figure below



```
search_pos( in : pos:int,   r:ptr, out : p,q:ptr ) : bool

p ← r ; q ← NIL ; leaf ← false

WHILE ( not leaf  and  p ≠ NIL )
        IF ( lc(p) == NIL and rc(p) == NIL )
            leaf ← true
        ELSE
                q ← p
                IF ( pos ≤ Info(p) )
                        p ← lc(p)
                ELSE
                        pos ← pos - Info(p)
                        p ← rc(p)
                EndIf
        EndIf
EndWhile

IF ( leaf and pos == 1 ) return true ELSE return false  EndIf
```

3) Huffman Coding (a compression method)

Construction of a **variable-length binary code** to compress a message (or file) :

Input: a sequence of symbols (a,b,c, …) to encode

Calculate the frequency (number of occurrences) of each symbol and associate a node (initially isolated). The nodes are inserted in a priority queue (key = frequency)

WHILE the priority queue contains more than one element

Dequeue 2 nodes ***x*** and ***y*** (therefore having the lowest frequency in the queue)

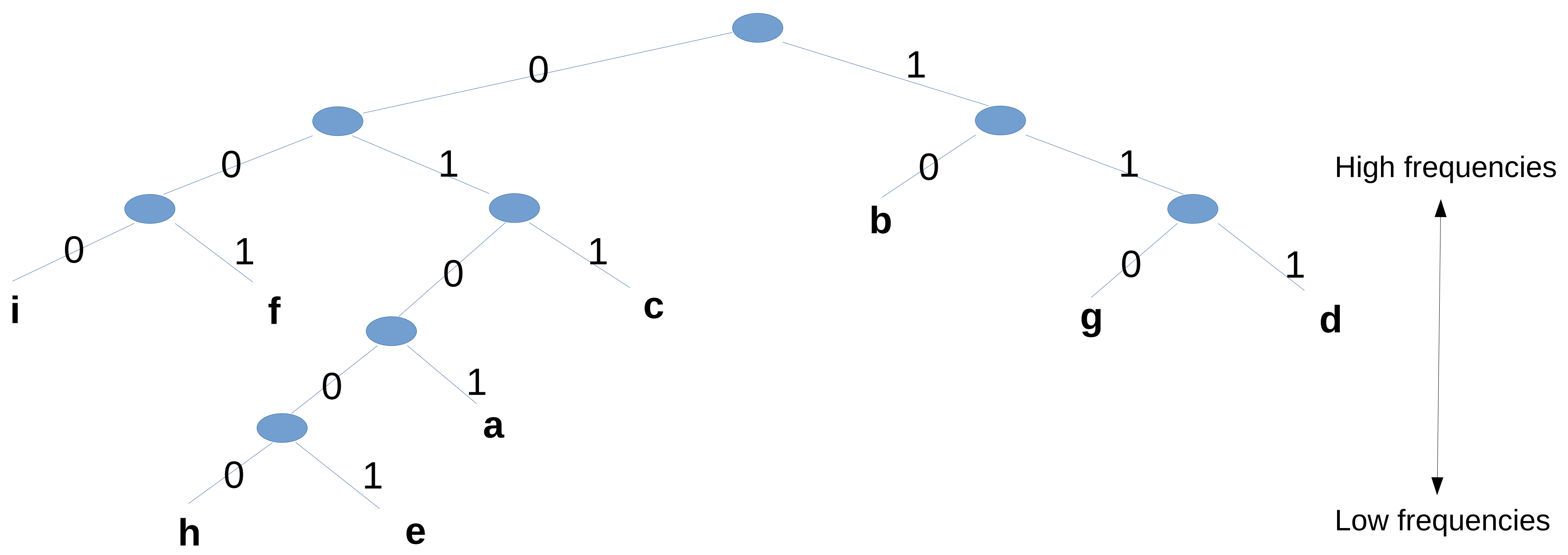
Create a new node ***n*** and connect ***x*** and ***y*** to it as right and left children
The frequency of ***n*** will be the sum of that of its children

Enqueue ***n***

EndWhile

Output: A ***binary tree*** where the leaves represent the symbols forming the input message → this is the ***Huffman tree***

By associating bits 0 and 1 to represent the left and right directions respectively (or the reverse), we obtain a **coding of the different leaves** of the tree



code of a is 0101, code of b is 10, code of c is 011 …

Huffman codes are **prefix-free** (no code is prefix of any other)
→ simplify decoding

Example : the following bit string can be decoded in only one possible way
1 0 0 1 0 1 0 1 0 1 0 1 1 ⇒ baac

Example of a message to compress :

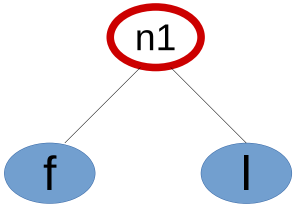
ceci_est_un_message_pour_montrer_le_fonctionnement_du_codage

The different symbols appearing in the message (with their frequencies) are :
f(1) l(1) p(1) a(2) d(2) i(2) g(2) m(3) r(3) s(3) u(3) c(4) t(4) o(5) n(6) e(9) _(9)

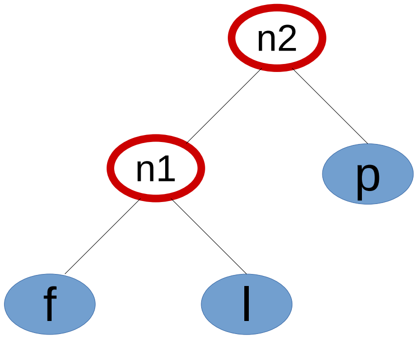
This is the initial content of the Priority Queue
(symbol nodes have already been created and queued)

Content of the priority queue :

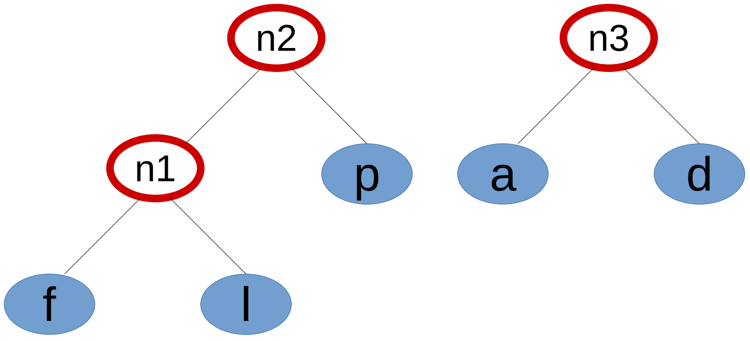
p(1) n1(2) a(2) d(2) i(2) g(2) m(3) r(3) s(3) u(3) c(4) t(4) o(5) n(6) e(9) _(9)



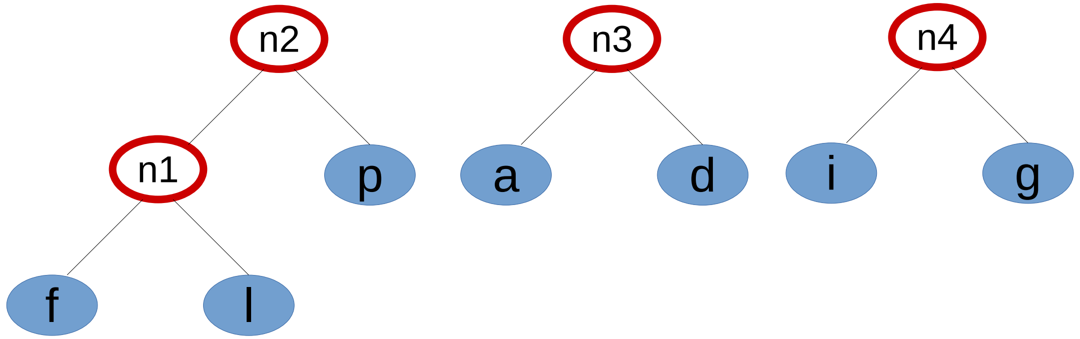
Content of the priority queue :
a(2) d(2) i(2) g(2) n2(3) m(3) r(3) s(3) u(3) c(4) t(4) o(5) n(6) e(9) _(9)



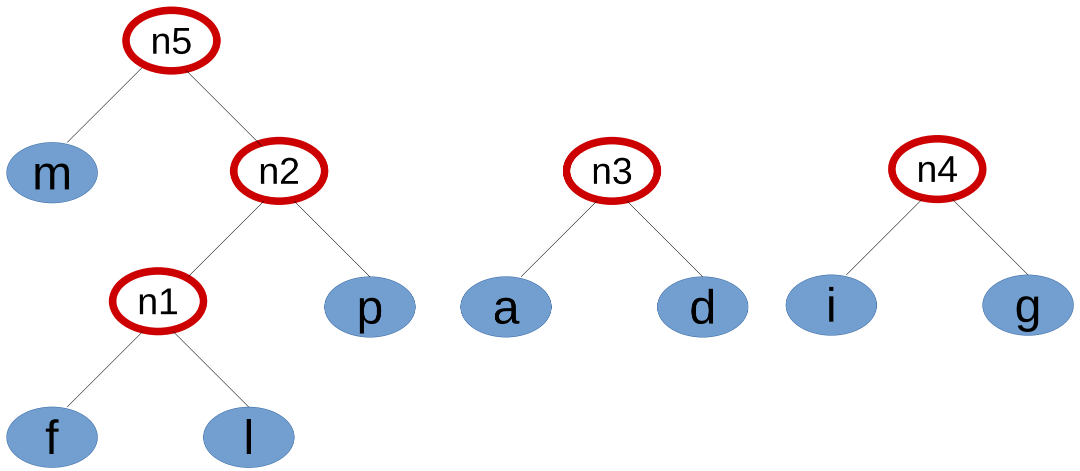
Content of the priority queue :
i(2) g(2) n2(3) m(3) r(3) s(3) u(3) n3(4) c(4) t(4) o(5) n(6) e(9) _(9)



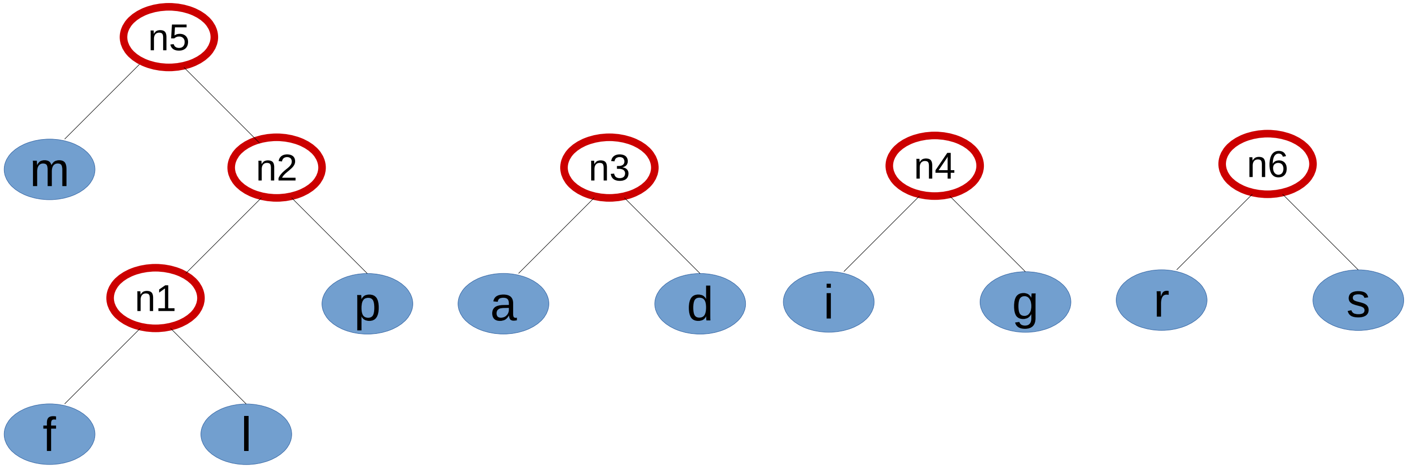
Content of the priority queue :
n2(3) m(3) r(3) s(3) u(3) n4(4) n3(4) c(4) t(4) o(5) n(6) e(9) _(9)



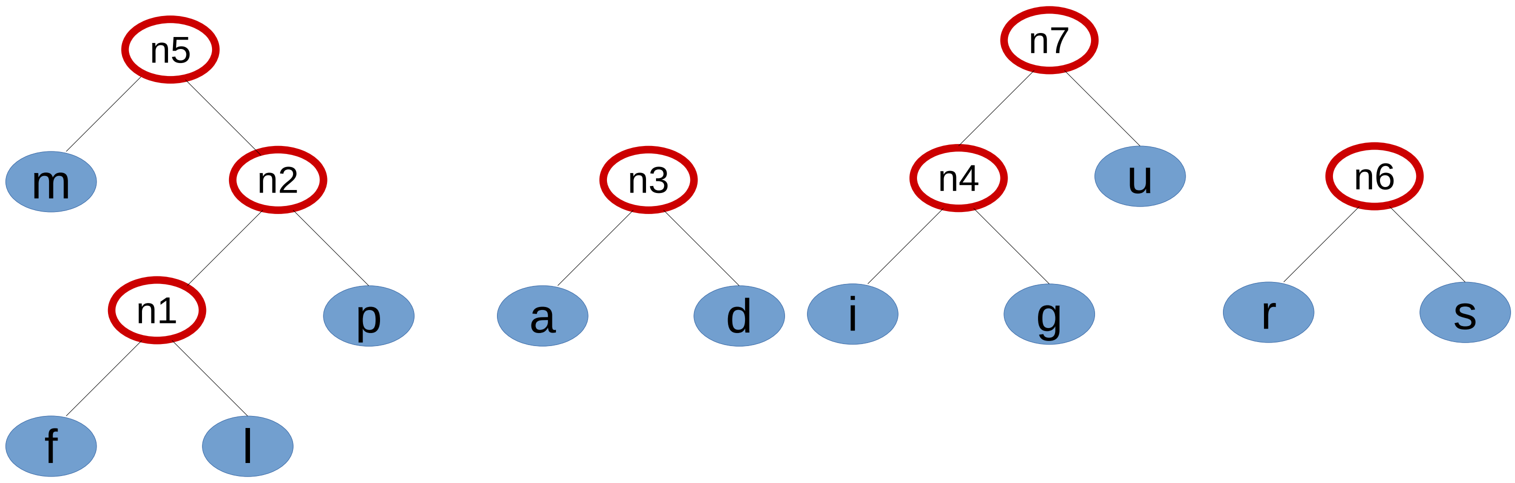
- Content of the priority queue :
  r(3) s(3) u(3) n4(4) n3(4) c(4) t(4) o(5) n5(6) n(6) e(9) _(9)



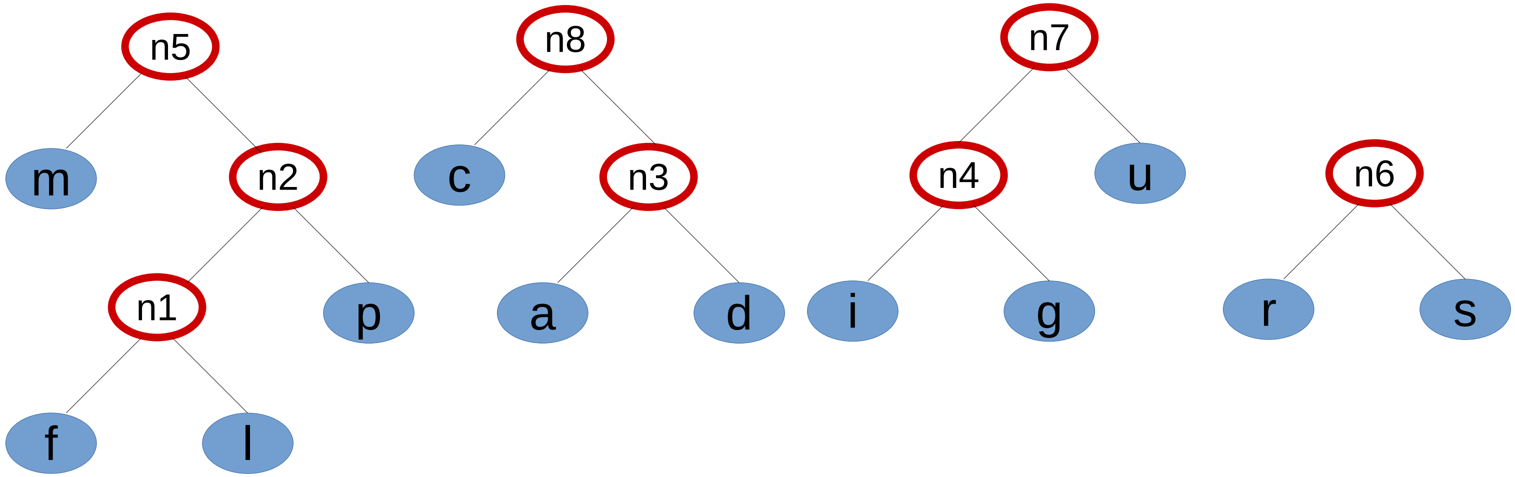
- Content of the priority queue :
  u(3) n4(4) n3(4) c(4) t(4) o(5) n6(6) n5(6) n(6) e(9) _(9)



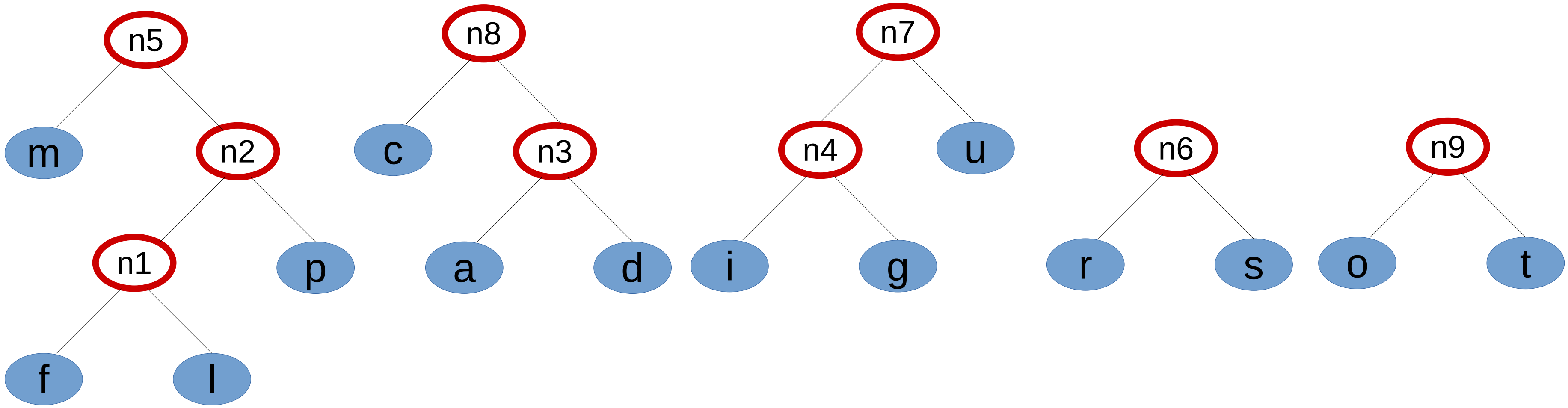
- Content of the priority queue :
  n3(4) c(4) t(4) o(5) n6(6) n5(6) n(6) n7(7) e(9) _(9)



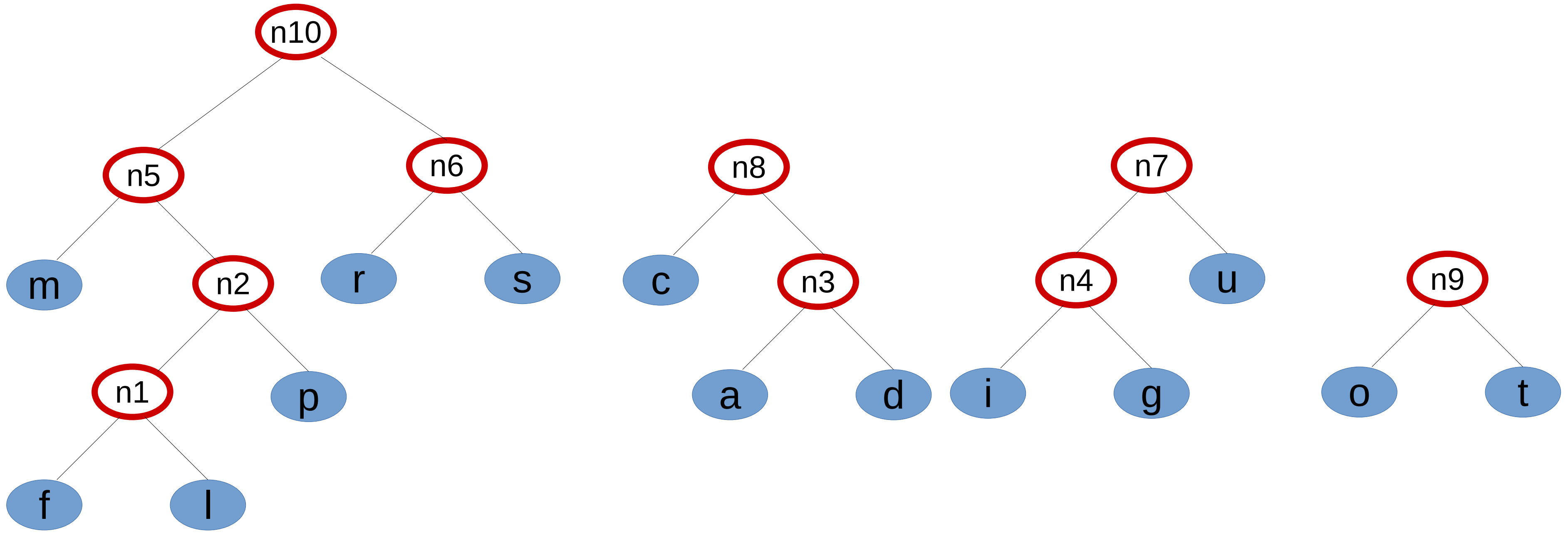
- Content of the priority queue :
  t(4) o(5) n6(6) n5(6) n(6) n7(7) n8(8) e(9) _(9)



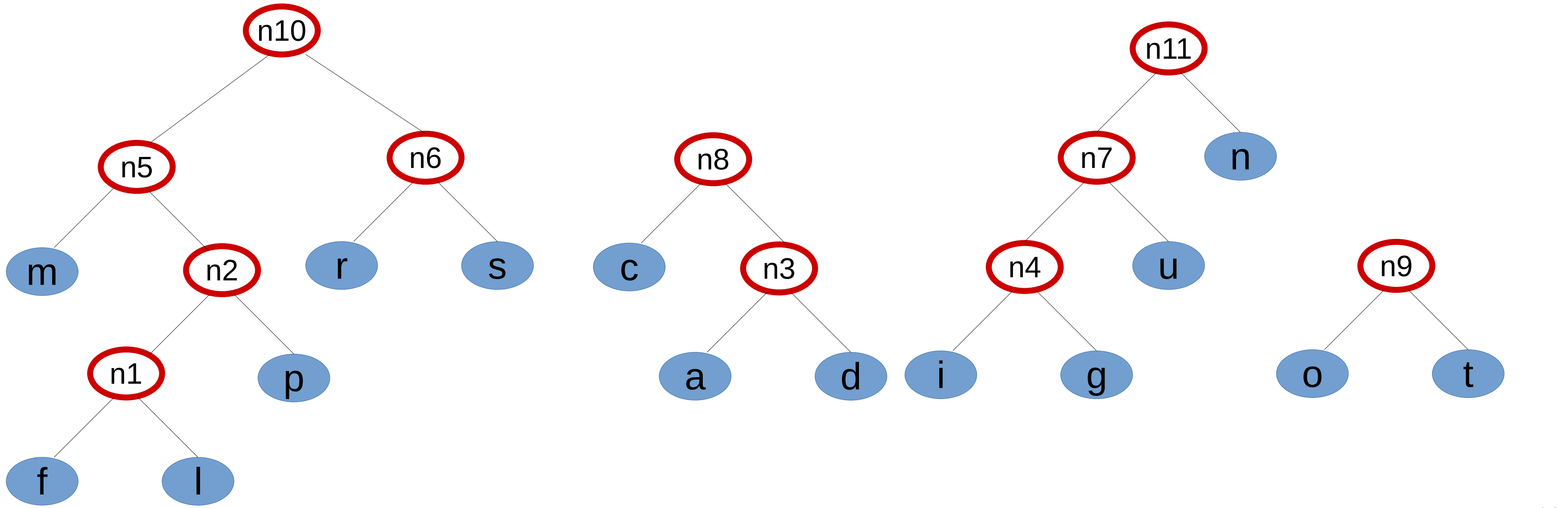
- Content of the priority queue :
  n6(6) n5(6) n(6) n7(7) n8(8) n9(9) e(9) _(9)



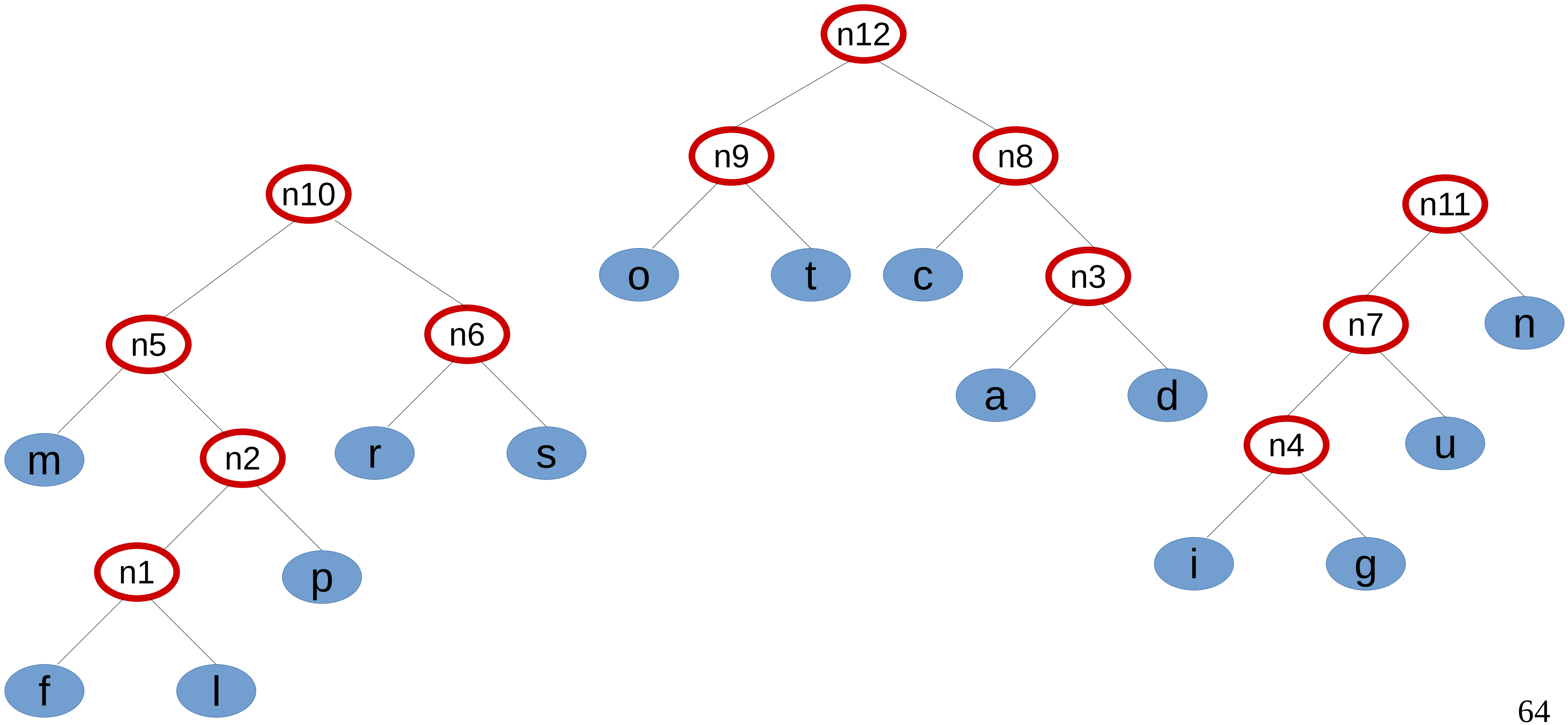
- Content of the priority queue :
  n(6) n7(7) n8(8) n9(9) e(9) _(9) n10(12)



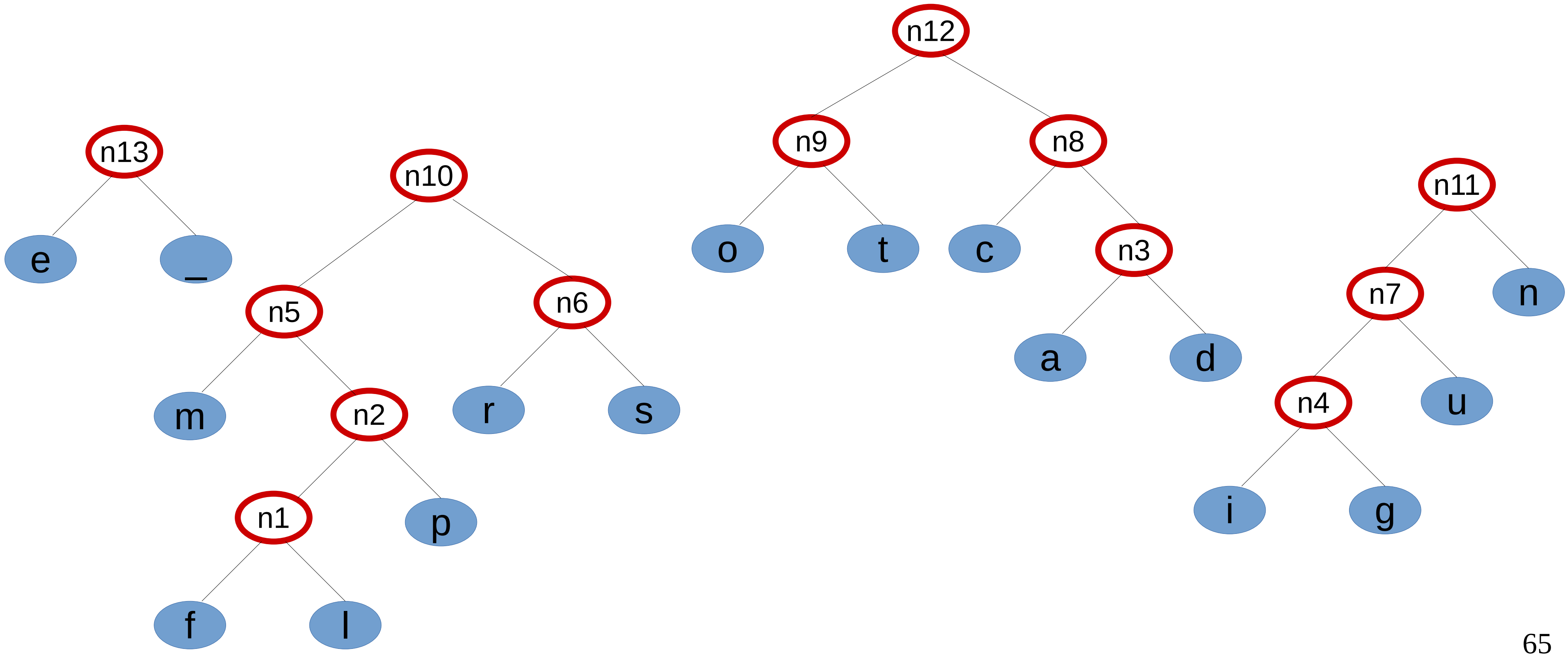
- Content of the priority queue :
  n8(8) n9(9) e(9) _(9) n10(12) n11(13)



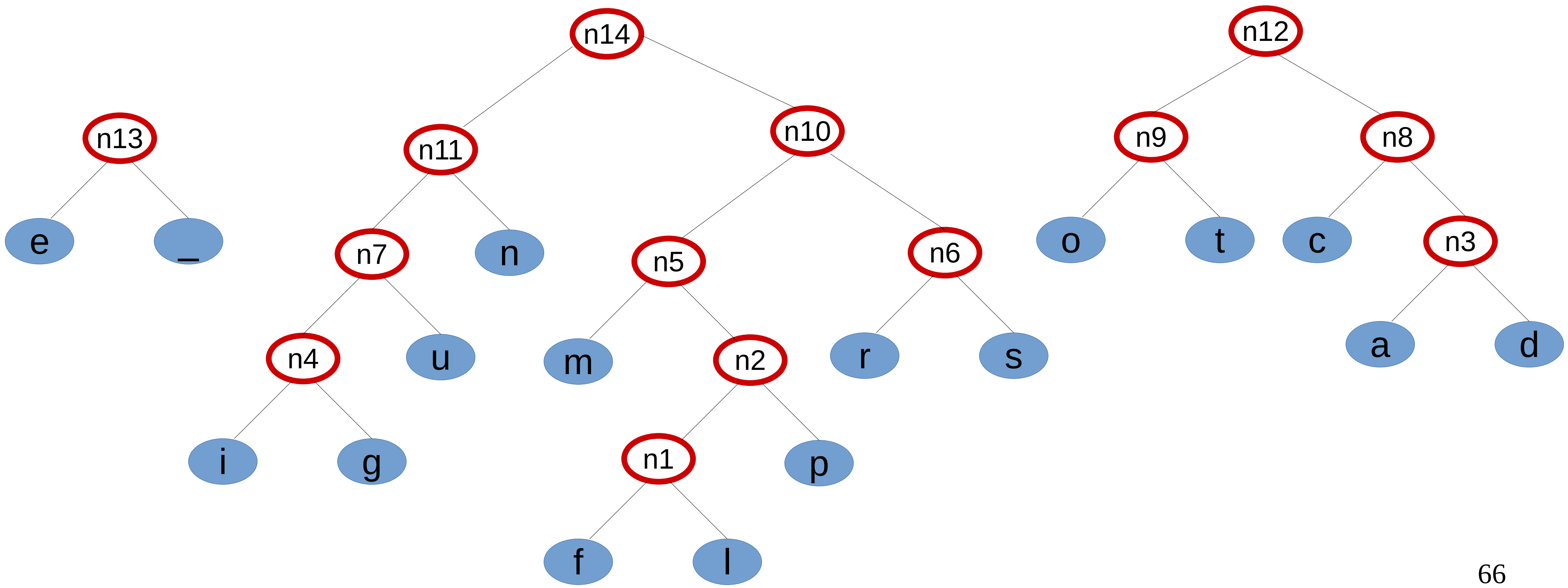
- Content of the priority queue :
  e(9) _(9) n10(12) n11(13) n12(17)



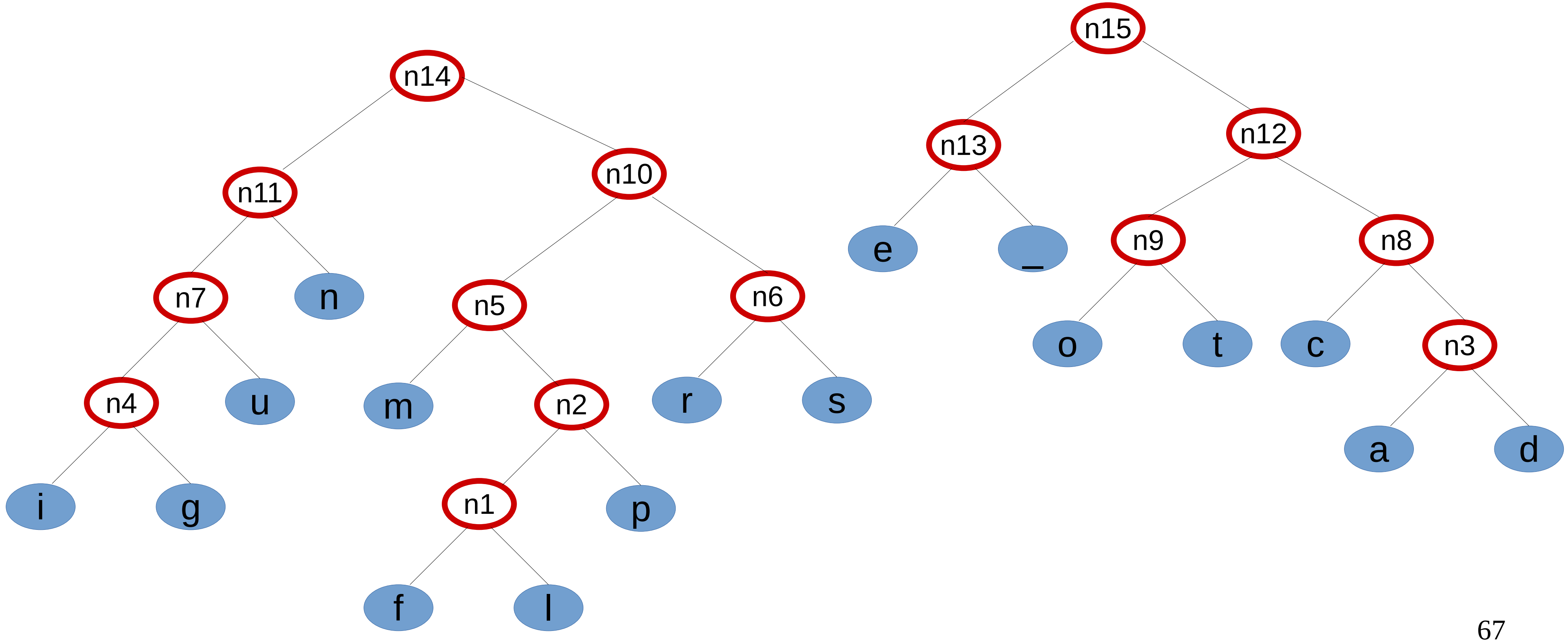
- Content of the priority queue :
  n10(12) n11(13) n12(17) n13(18)



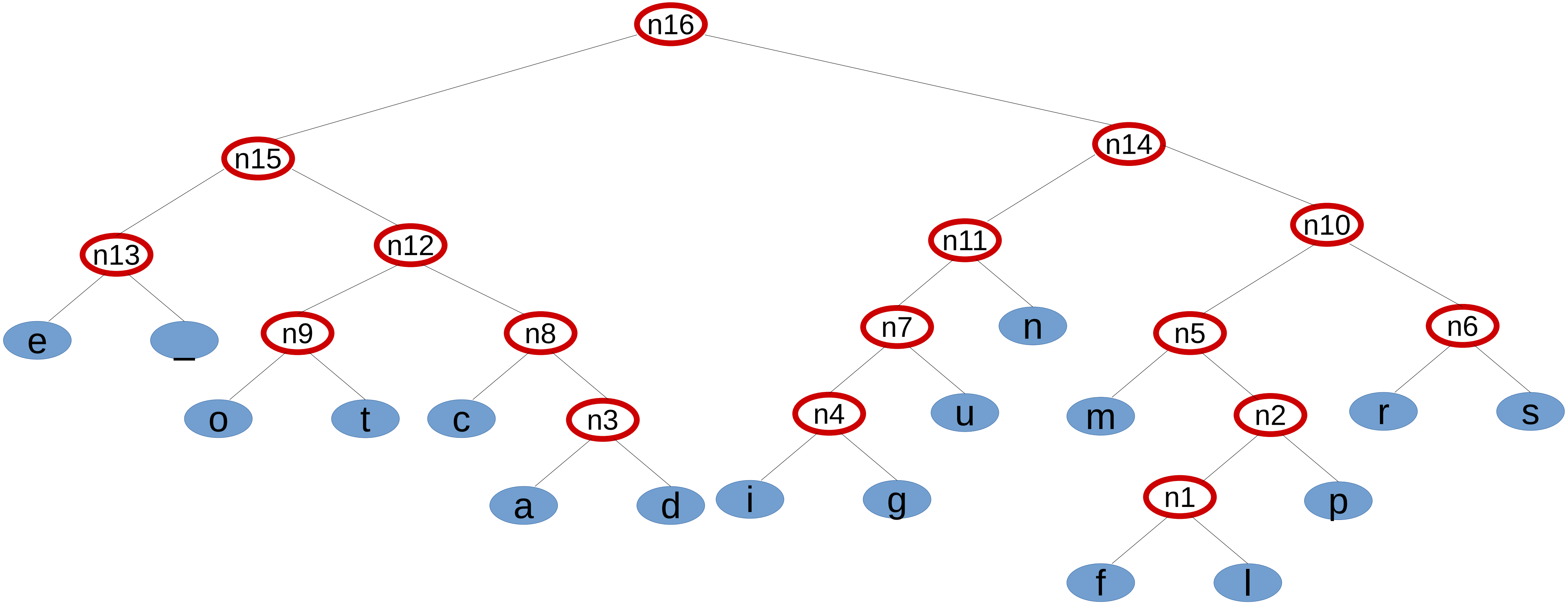
- Content of the priority queue :
n12(17) n13(18) n14(25)



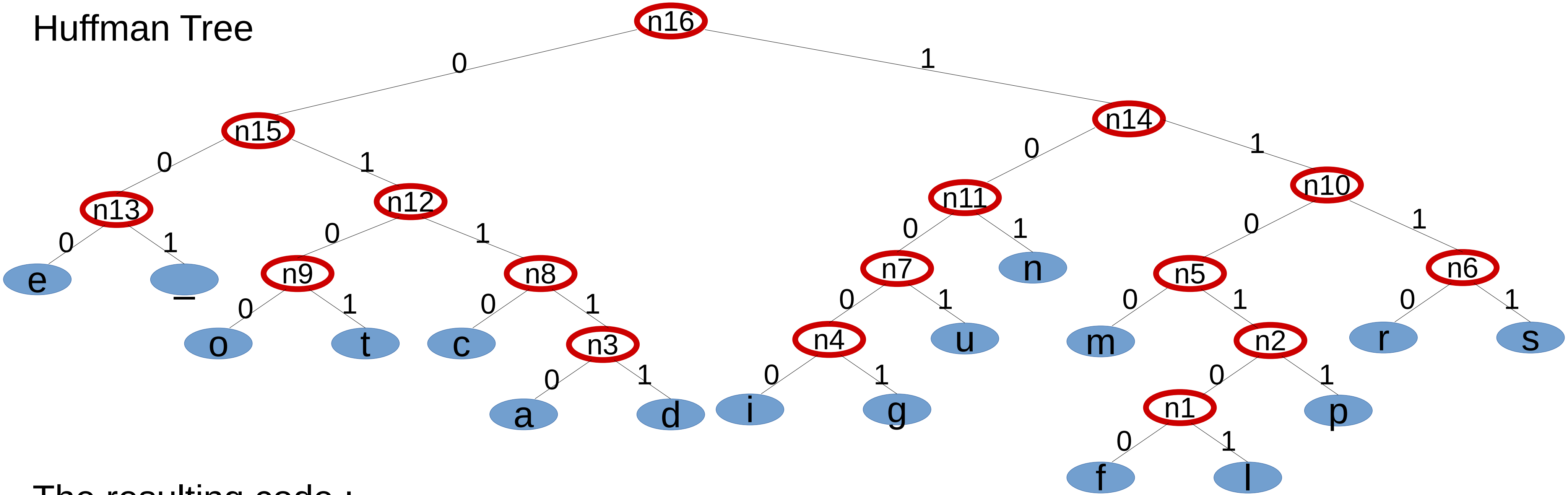
- Content of the priority queue :
  n14(25) n15(35)



Huffman Tree



Huffman Tree



The resulting code :

a : 01110	c : 0110	d : 01111	e : 000	f : 110100	g : 10001	i : 10000
l : 110101	m: 1100	n : 101	o : 0100	p : 11011	r : 1110	s :1111
t : 0101	u : 1001	_ : 001				

The input message : *ceci_est_un_message_pour_montrer_le_fonctionnement_du_codage*
size = *60 char* = ***60 bytes*** = *60 * 8* = ***480 bits***

The encoded message (in bits):
011000001101000000100011110101001100110111000001111111101110100010000011101101001001111000111000100
10101011110000111000111010100000111010001001010110010110000010010110100011000001010101001011111001
0010110010001111011101000100

The input message (size = *60 char* = ***60 bytes*** = *60 * 8* = ***480 bits***) :

ceci_est_un_message_pour_montrer_le_fonctionnement_du_codage

01100011 01100101 01100011 01101001 01011111 01100101 01110011 01110100 01011111 01110101
01101110 01011111 01101101 01100101 01110011 01110011 01100001 01100111 01100101 01011111
01110000 01101111 01110101 01110010 01011111 01101101 01101111 01101110 01110100 01110010
01100101 01110010 01011111 01101100 01100101 01011111 01100110 01101111 01101110 01100011
01110100 01101001 01101111 01101110 01101110 01100101 01101101 01100101 01101110 01110100
01011111 01100100 01110101 01011111 01100011 01101111 01100100 01100001 01100111 01100101

The resulting code :

a:01110	*c:0110*	*d:01111*	*e:000*	*f:110100*	*g:10001*	*i:10000*
l:110101	*m:1100*	*n:101*	*o:0100*	*p:11011*	*r:1110*	*s :1111*
t:0101	*u:1001*	*_:001*				

The encoded message (size = **229 bits**):

0110000011010000001000111101010011001101110000011111111011101000100000111011010010011110001110001001010 10
11110000111000111010100000111010001001010110010110000010010110100011000000101010100101111100100101100 10001
1110111010001000

The encoded message in bytes (size = **29 bytes**) :

01100000 11010000 00100011 11010100 11001101 11000001 11111110 11101000 10000011 10110100
10011110 00111000 10010101 01111000 01110001 11010100 00011101 00010010 10110010 11000001
00101101 00011000 00101010 10010111 11001001 01100100 01111011 10100010 00******

4) Efficient Priority Queue Implementation

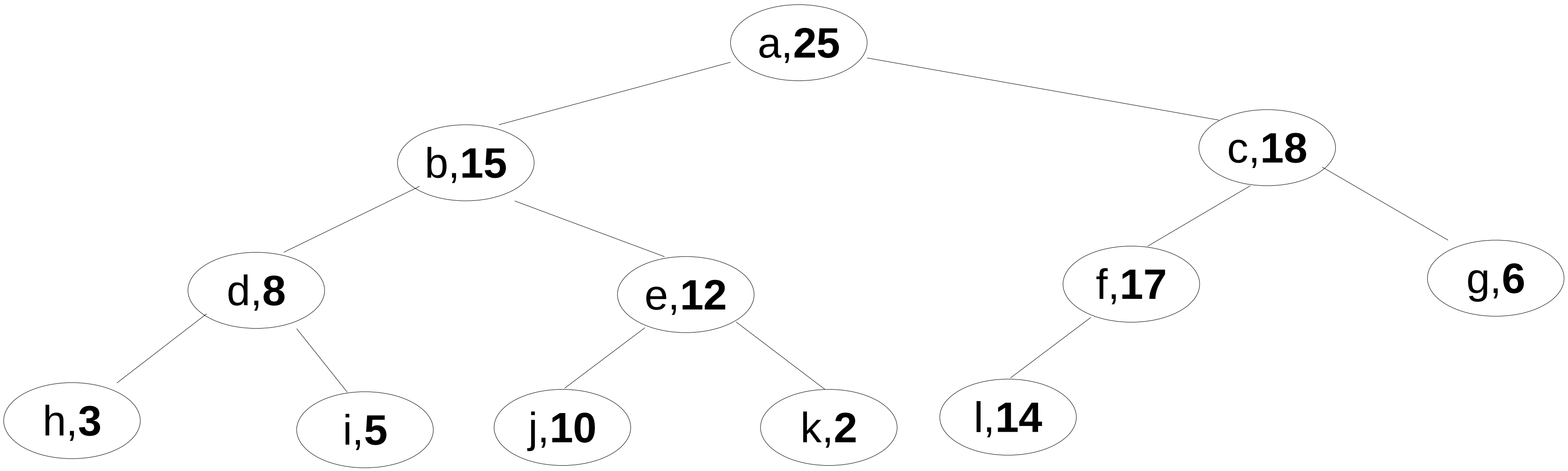
Elements are {value, priority} pairs. The value with the highest priority is dequeued first

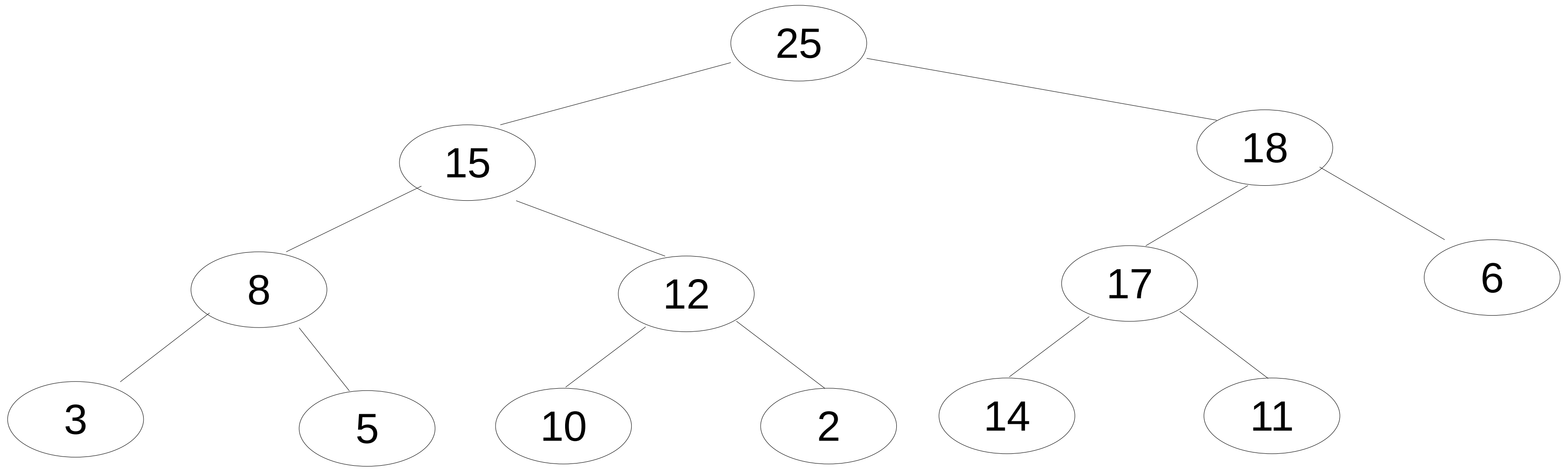
Naive approach

→ inefficient (enqeue or dequeue in *O*(n))

Heap approach

The queue is an **almost complete binary tree** (called **Heap**)
All levels of the tree are full (except possibly the last one which fills from left to right)
The root contains the element with the highest priority
Each <u>internal node has a priority</u> **greater than or equal to** that of <u>its children</u>



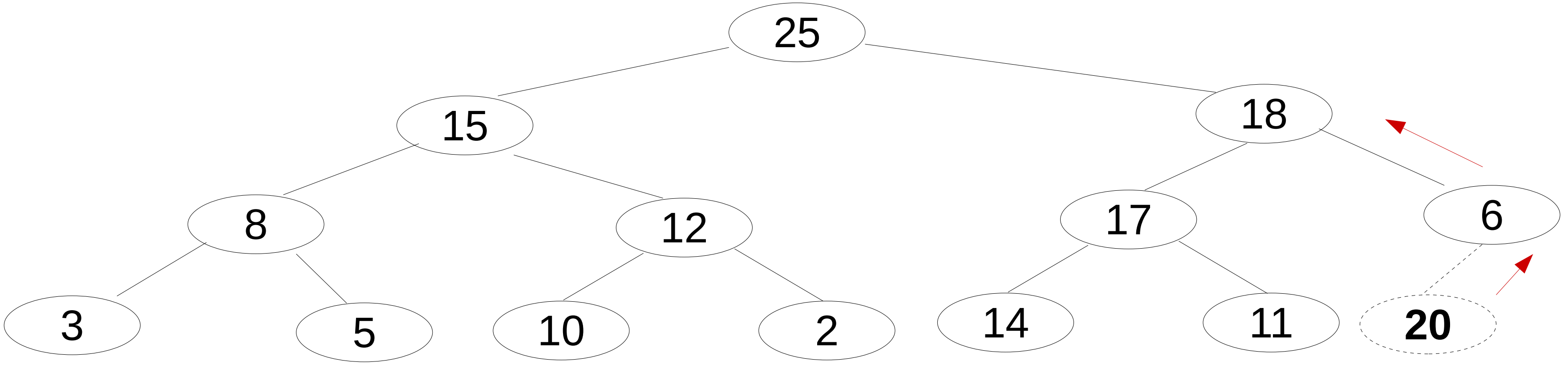


enqueue({v,p}) = Add a node in the last level and swap with its ancestors until its priority is less than or equal to that of its parent → ***O*(log n)**

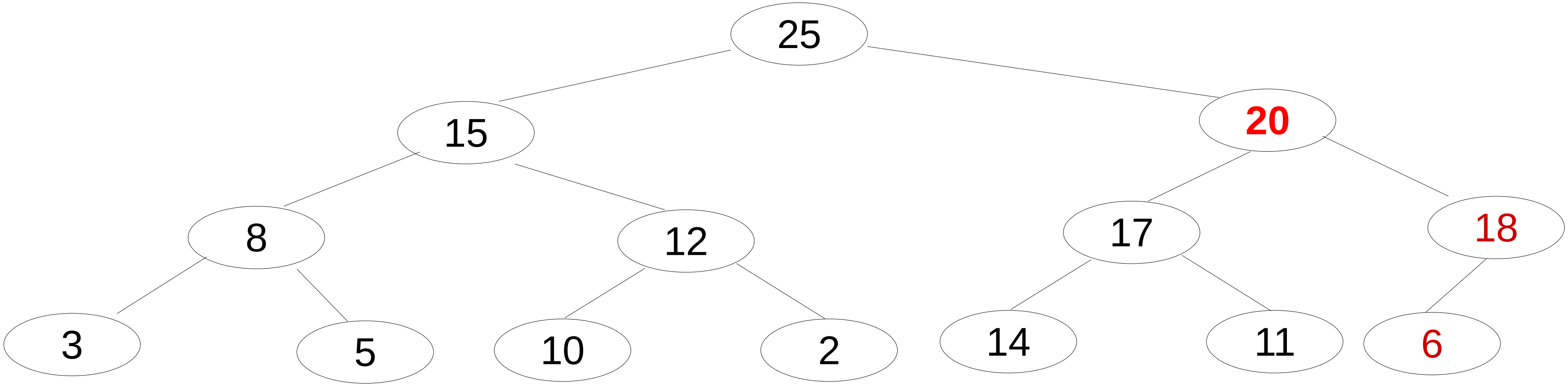
dequeue(v) = Remove the root elmt and replace it with the rightmost node of the last level. Then swap with its highest priority child until its priority is greater than or equal to those of its 2 children → ***O*(log n)**

Example : **enqueue({…,20})**

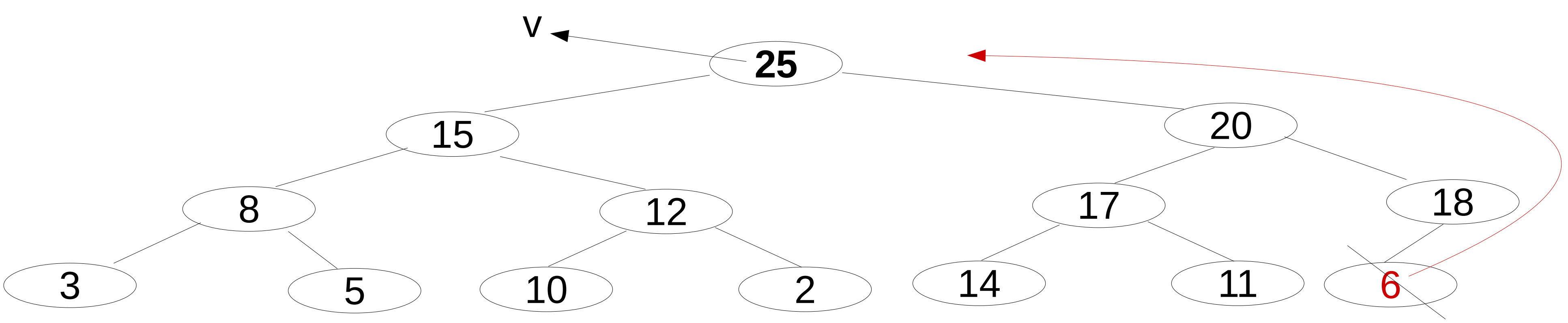
Add a new node (20) temporarily in the last level of the tree (filling the level from left to right)



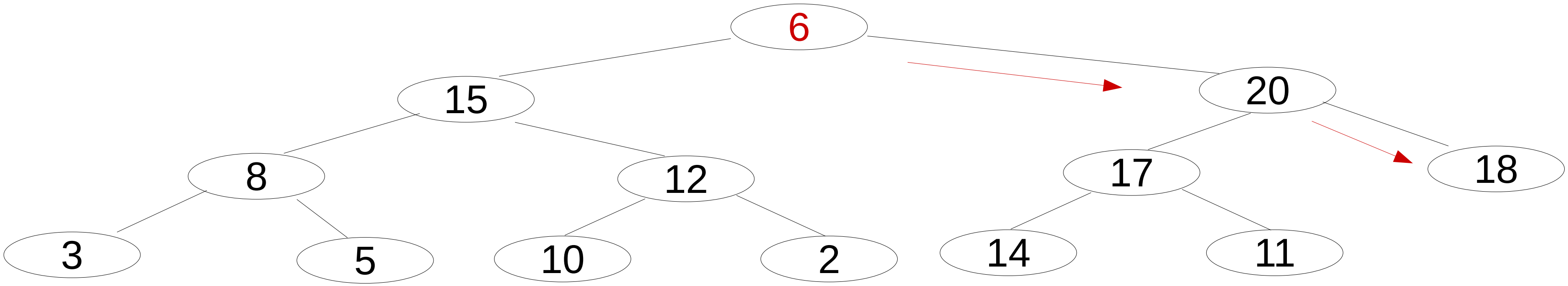
swap with parent nodes, until child has lower or equal priority than parent



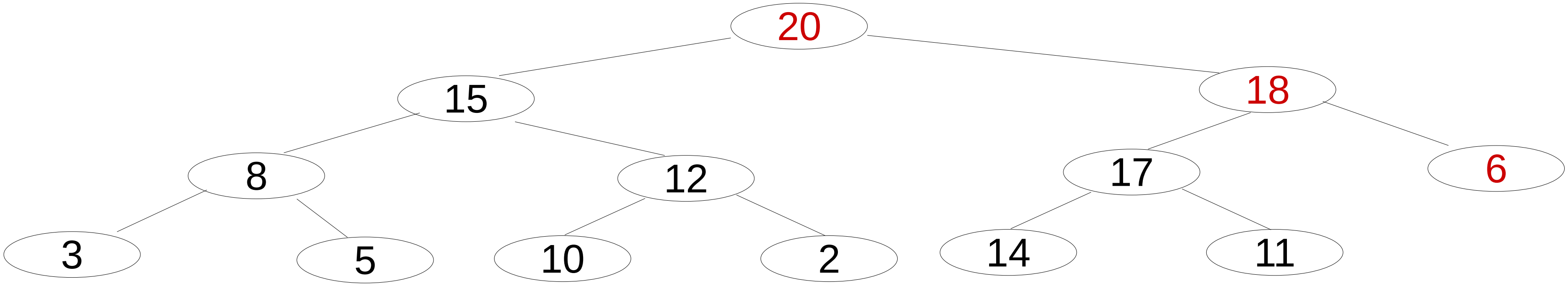
Example : **dequeue(v)** → return in v, the root value



temporarily consider node 6 as the new root (instead of 25)



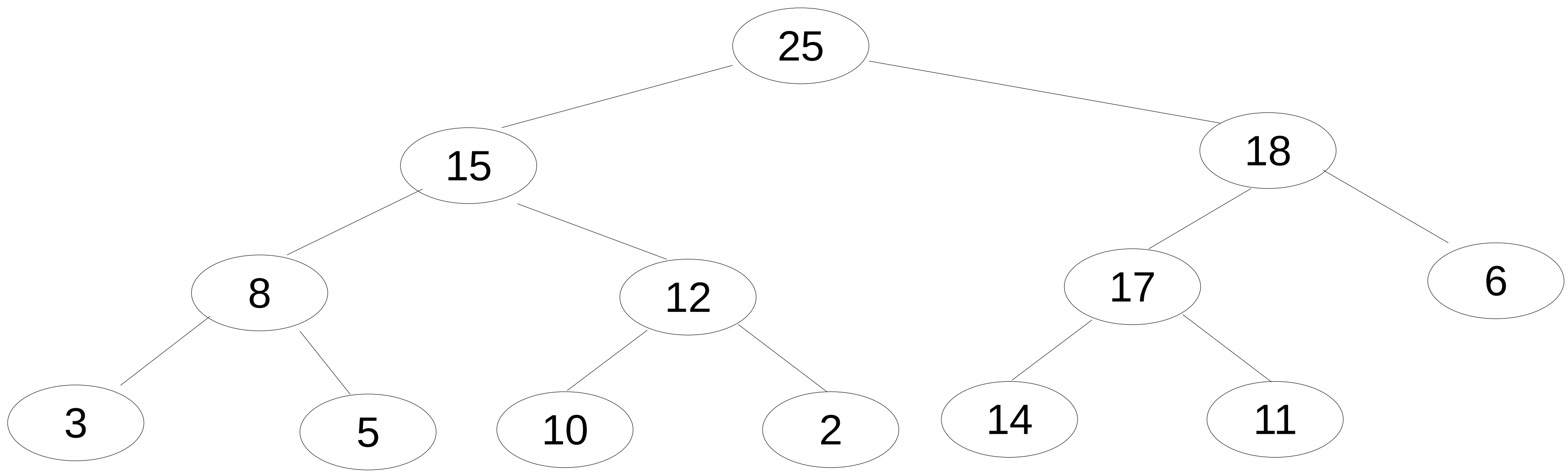
swap with the highest priority child, until all children have lower or equal priority than parent node



In practice, Heap is a sequential representation of the tree

Heap = **array** T[1..N] of elmt {value, priority}

root node is T[1] / left_child(i) is 2i / right_child(i) is 2i+1 / parent(i) is i div 2



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	25	15	18	8	12	17	6	3	5	10	2	14	11			

tail