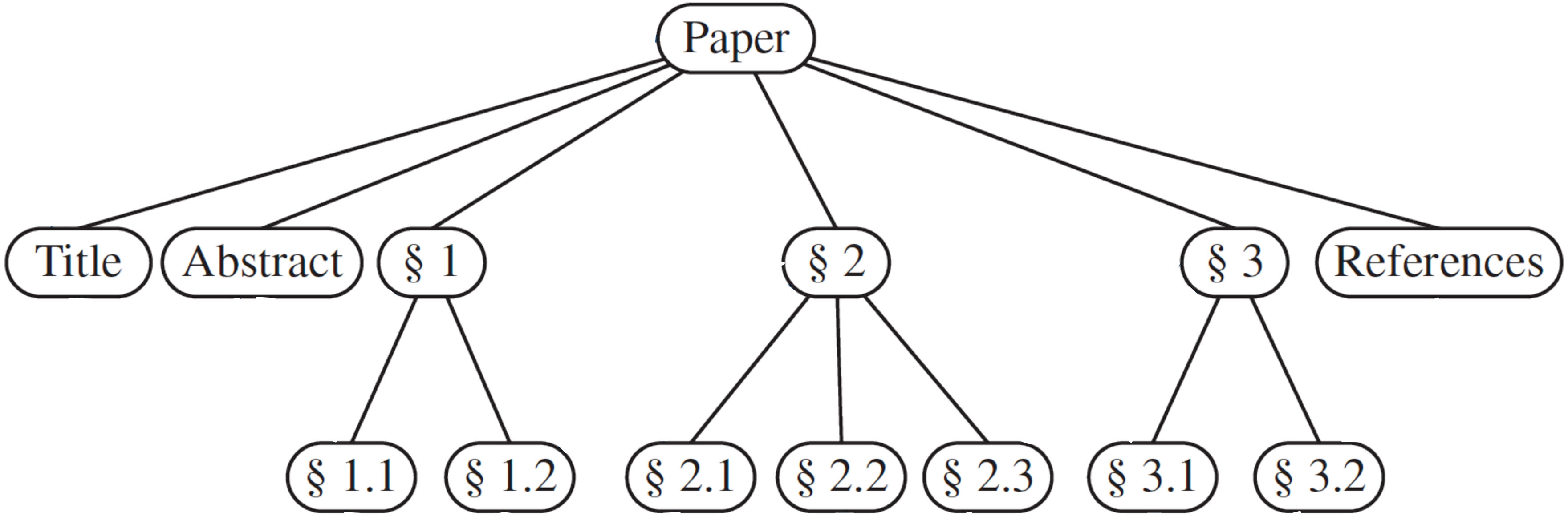
23

TREE TRAVERSAL ALGORITHMS

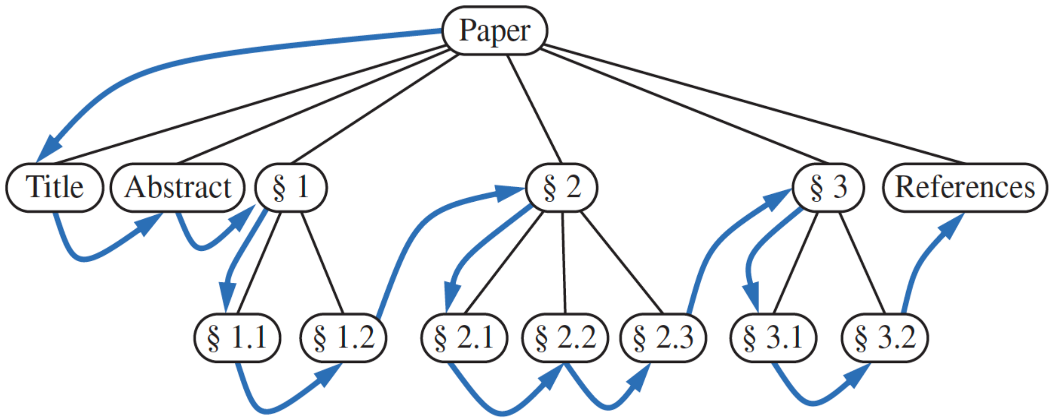
PREORDER TRAVERSAL

- In a **preorder traversal** of a tree, the root is visited first and then the subtrees rooted at its children are traversed recursively.



- We never move to a sibling of node p before traversing all descendants of p

PREORDER TRAVERSAL



Algorithm preorder(T, p):
  perform the "visit" action for node p
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling preorder(T,q)

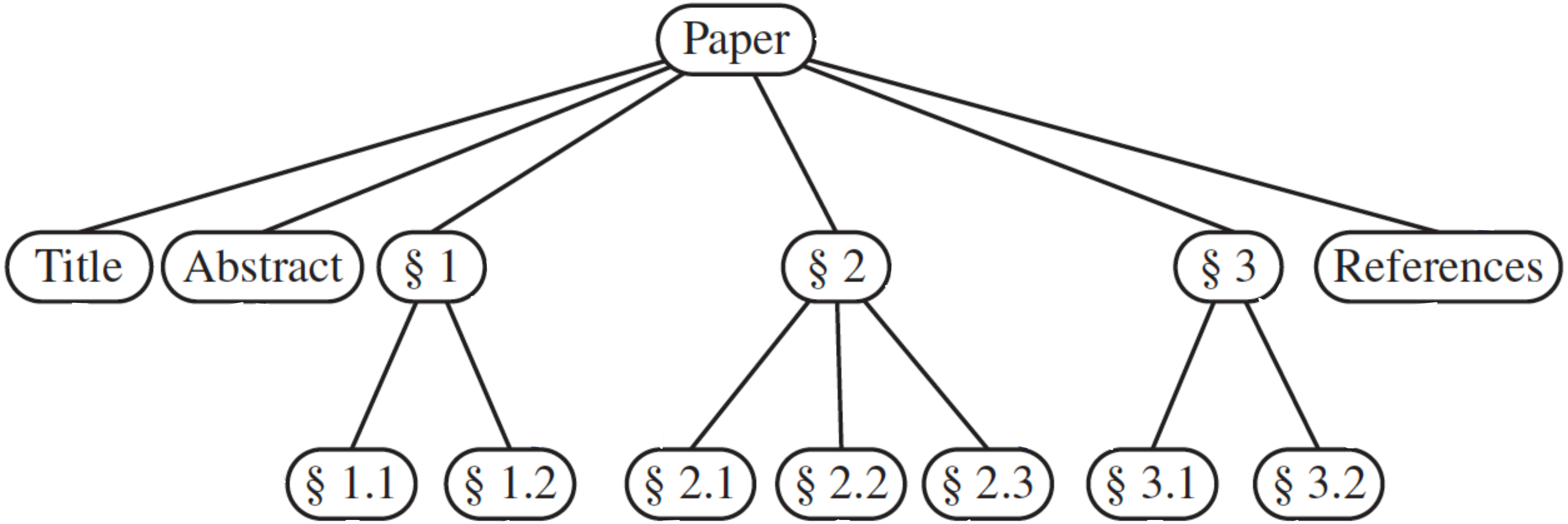
Here is a C++ implementation where the "visit" action is to print the element in the node

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p; // print element
  PositionList ch = p.children(); // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

At each node p, the non-recursive part takes $O(c_p)$ time, and we showed earlier that $\sum_p c_p = n - 1$. Thus, the overall running time is $O(n)$.

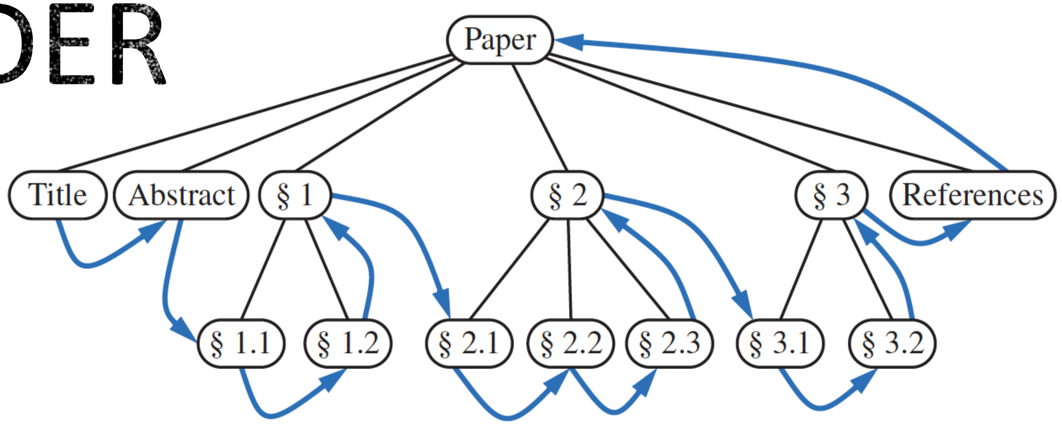
POSTORDER TRAVERSAL

- **Postorder traversal** can be viewed as the opposite of the preorder traversal; it recursively traverses the subtrees rooted at the children of the root first, and then visits the root.



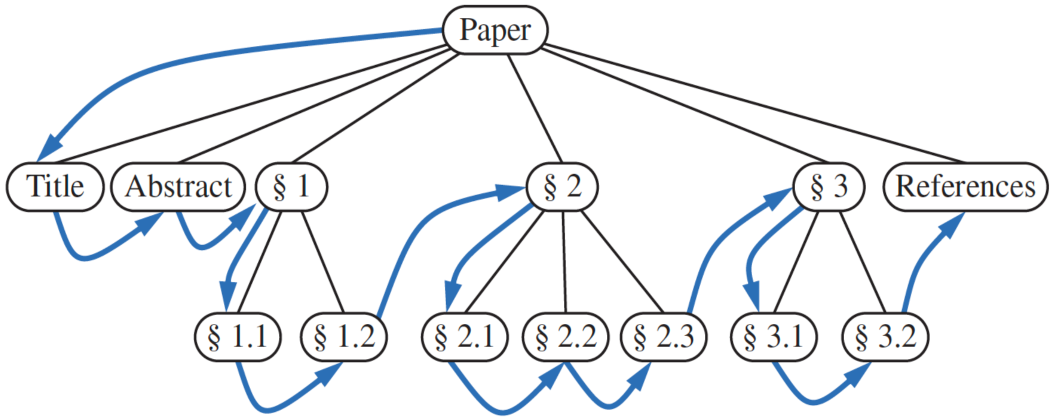
- We never move to a sibling, s, of a node p before traversing all descendants of s

PREORDER VS POSTORDER



Algorithm **postorder**(T, p):
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling postorder(T,q)
  perform the "visit" action for node p

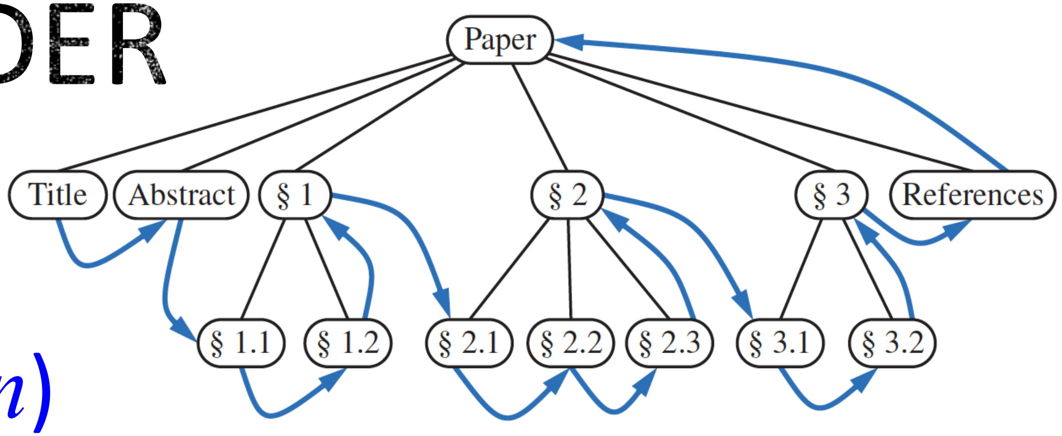
The "visit" action is performed **after** the loop, while preorder performs it **before** the loop!



Algorithm **preorder**(T, p):
  perform the "visit" action for node p
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling preorder(T,q)

PREORDER VS POSTORDER

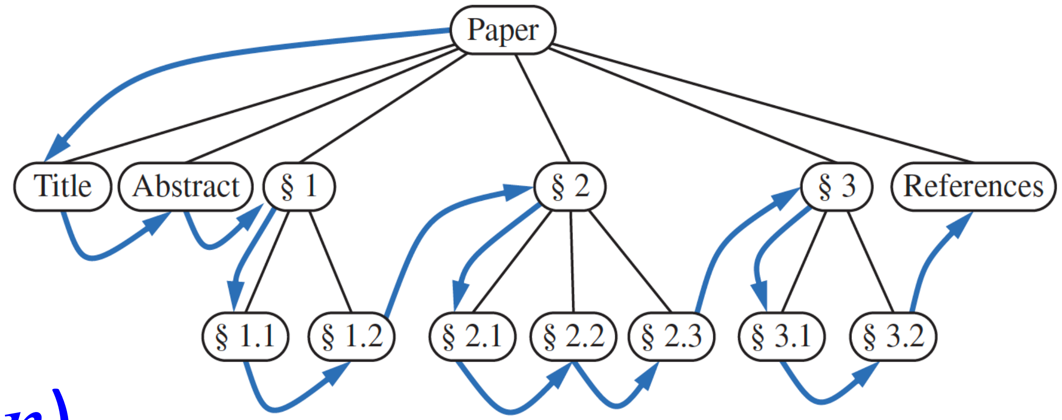
```
void postorderPrint(const Tree& T, const Position& p){
    PositionList ch = p.children(); // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
        postorderPrint(T, *q);
        cout << " ";
    }
    cout << *p; // print element
}
```



$O(n)$

The "visit" action is performed **after** the loop, while preorder performs it **before** the loop!

```
void preorderPrint(const Tree& T, const Position& p){
    cout << *p; // print element
    PositionList ch = p.children(); // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
        cout << " ";
        preorderPrint(T, *q);
    }
}
```

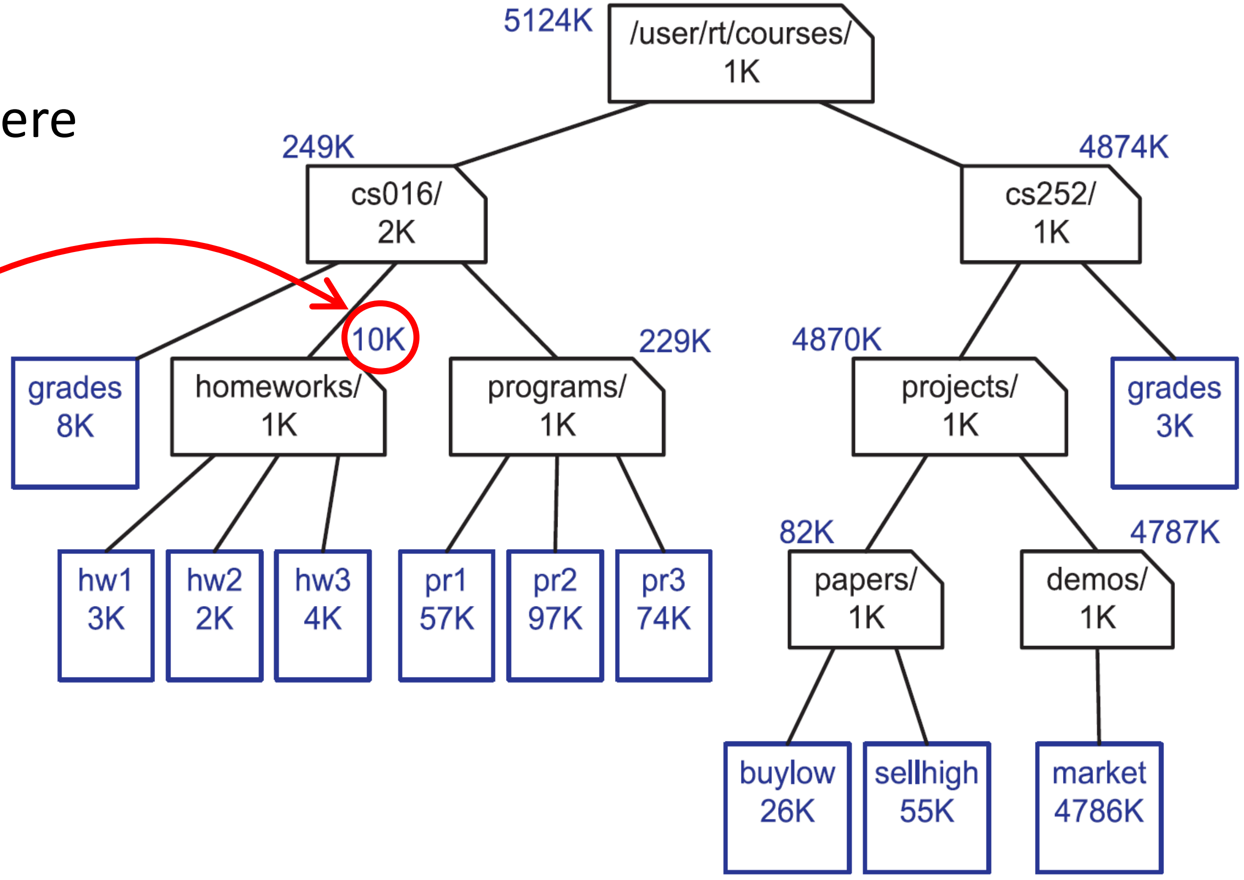


$O(n)$

POSTORDER TRAVERSAL

Postorder traversal is useful for solving problems where we wish to compute some property for each node p in a tree, but computing the property for p requires that we have already computed that same property for p's children

Example: Consider a file-system tree *T*, where external nodes represent files and internal nodes represent directories, and we want to **compute the disk space used by each directory**

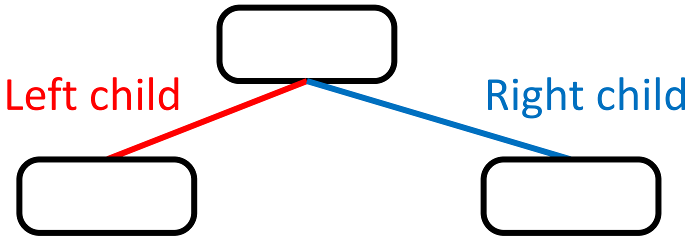


E.g., for homeworks, we added it size (which is 1K) to the size of its children (which is 3K+4K+4K), resulting in 10K

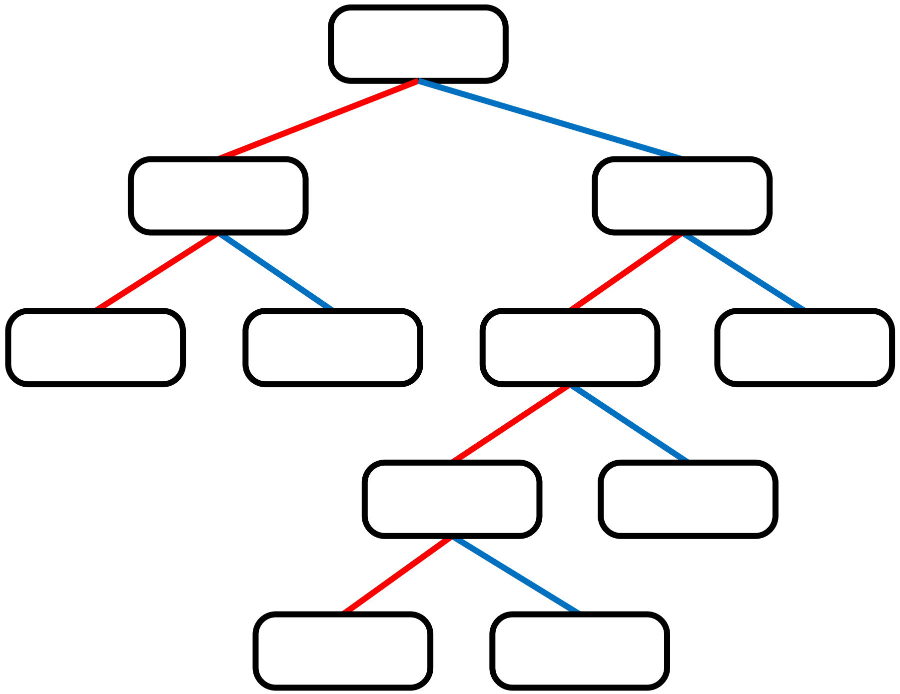
30

BINARY TREES

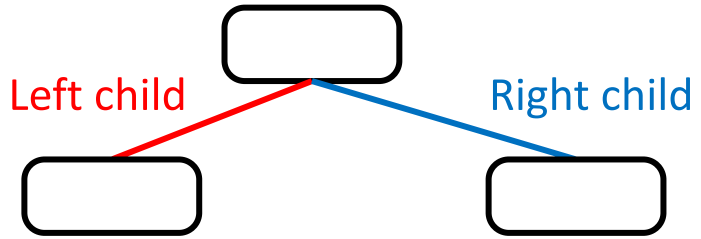
BINARY TREES



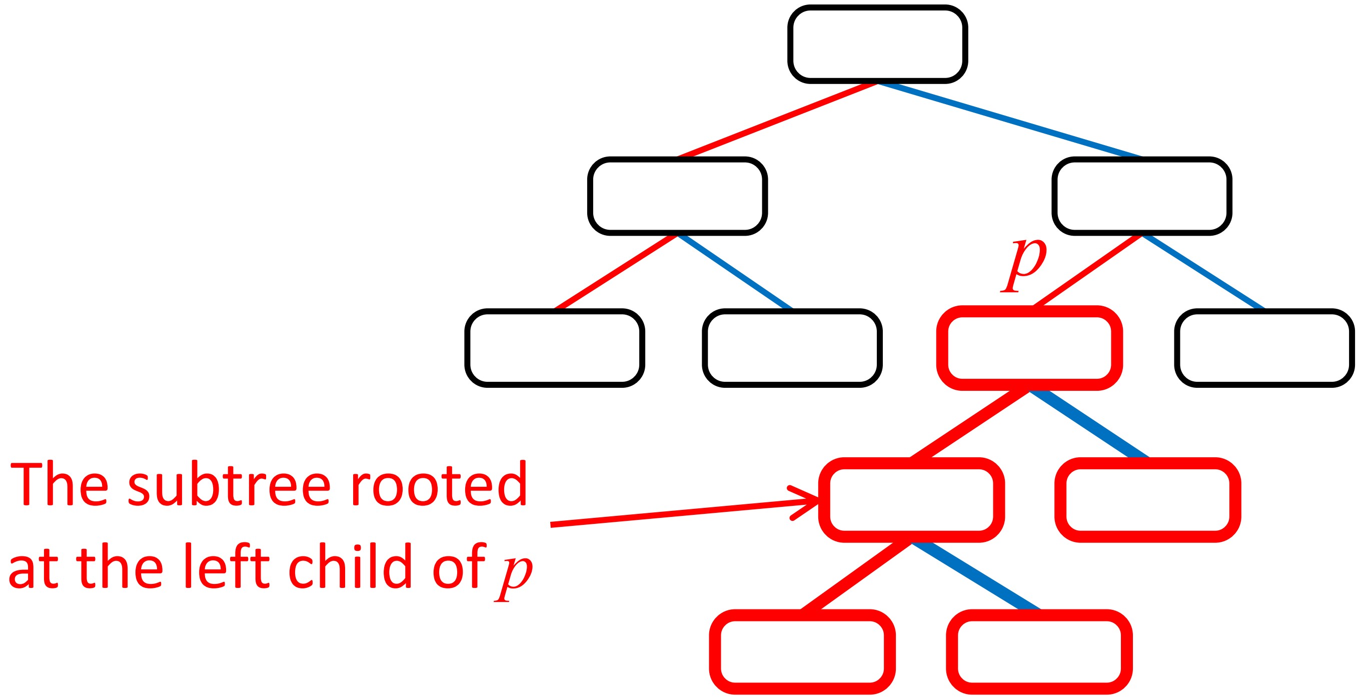
- A binary tree is a tree in which:
  - Each node has **at most two children**, called the “**left**” child and “**right**” child
  - If the binary tree is ordered, the **left child precedes the right** child



BINARY TREES

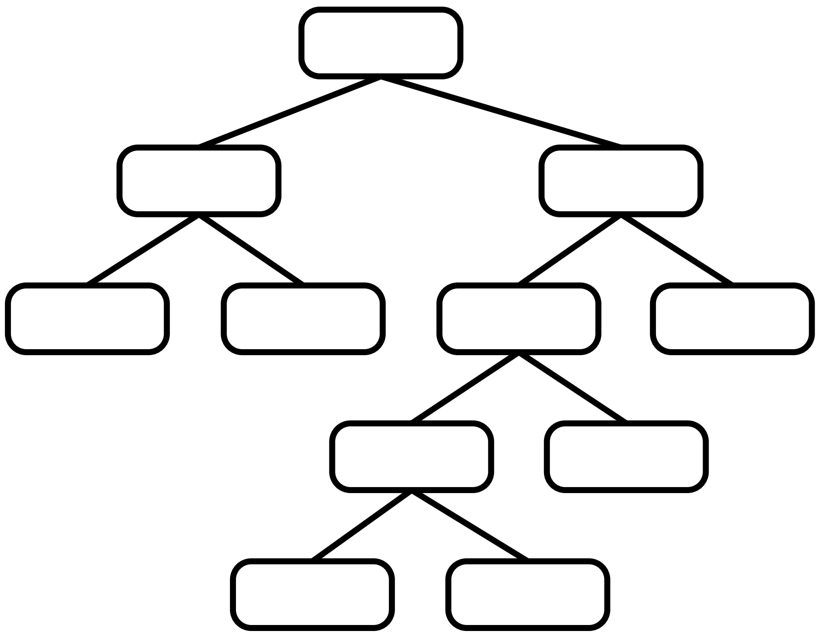


- A binary tree is a tree in which:
  - Each node has **at most two children**, called the “**left**” child and “**right**” child
  - If the binary tree is ordered, the **left child precedes the right child**

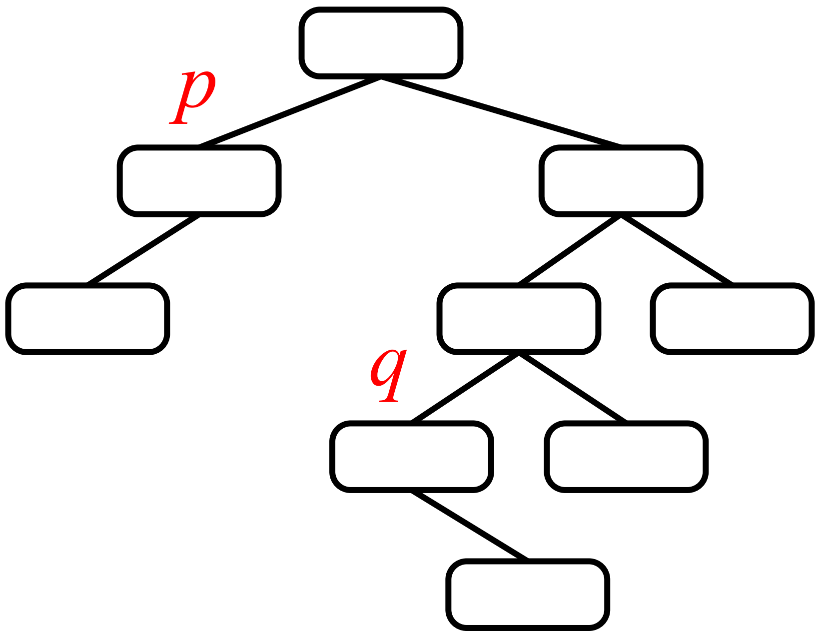


BINARY TREES

- A proper binary tree → each node has either **Zero children or Two children**.
- Otherwise, the tree is improper!



A **proper** tree



An **improper** tree
(nodes p and q have 1 child each)

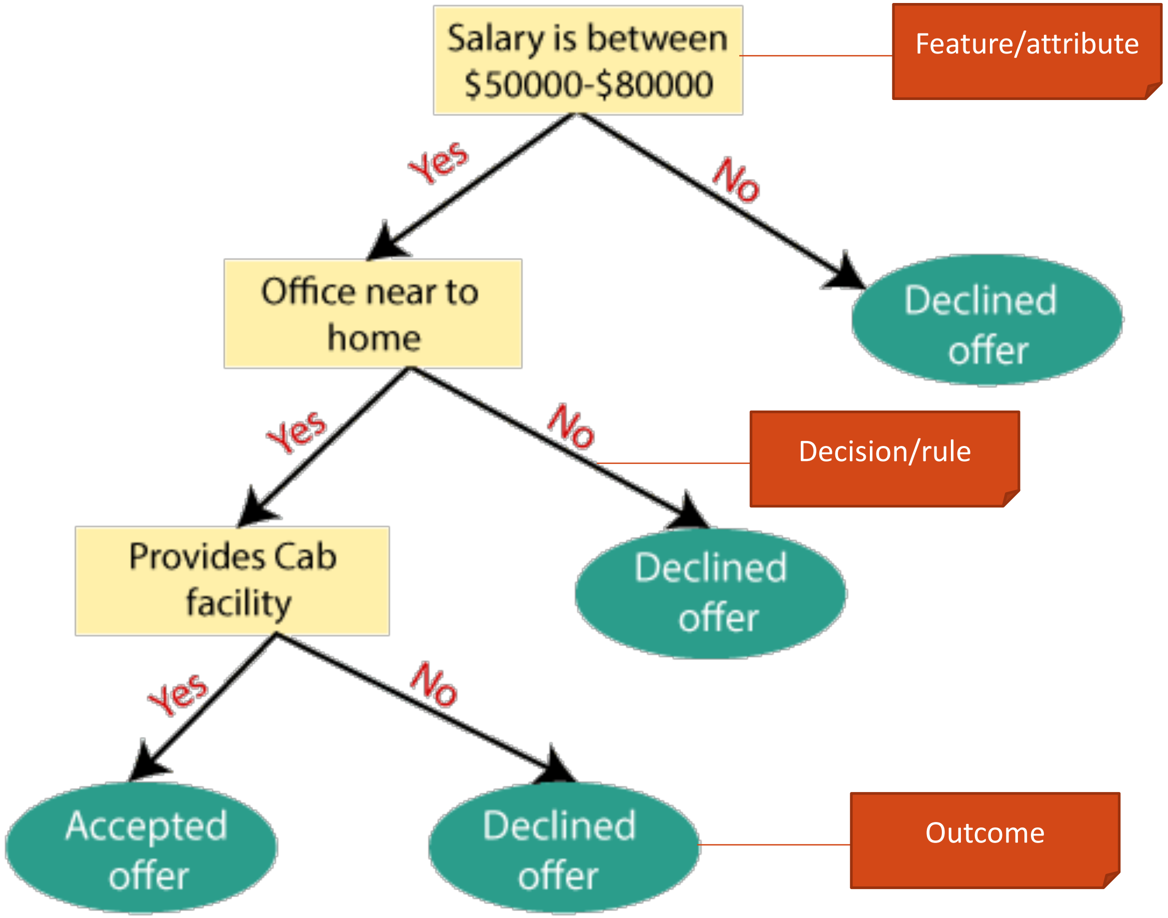
BINARY TREES – EXAMPLE (1)

- Binary trees can represent different outcomes that can result from answering a series of yes-or-no questions.

What are such binary trees called?

- Such binary trees are known as **decision trees**, because each external node p represents an outcome of what to do if the questions associated with p's ancestors are answered in a way that leads to p.

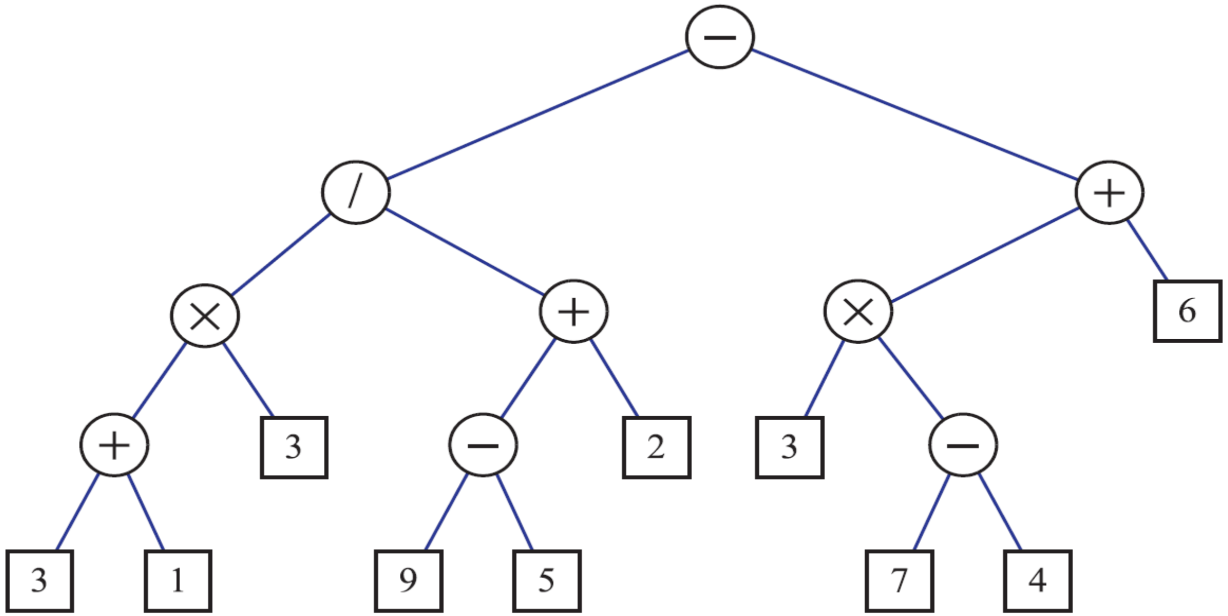
- Note that a decision tree is a **proper** binary tree



BINARY TREES – EXAMPLE (2)

- An arithmetic expression can be represented by a tree whose **external** nodes are associated with variables or constants, and whose **internal** nodes are associated with one of the operators +, −, ×, and /

- Such an arithmetic expression tree is a **proper** binary tree, since each of the operators +, −, ×, and / take exactly two operands.



Example: A binary tree representing the expression

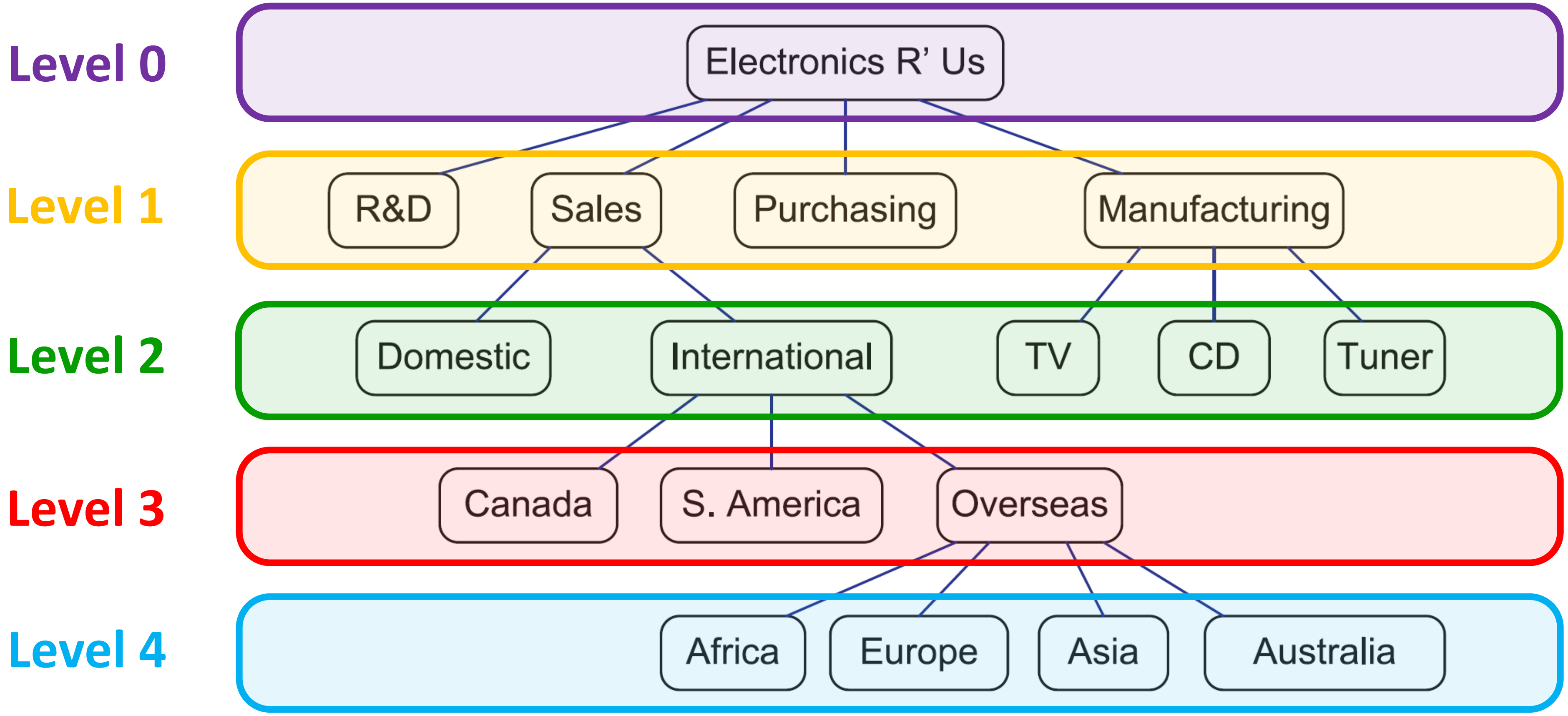
$$(((3+1)\times 3)/((9-5)+2)) - ((3\times(7-4))+6))$$

BINARY TREES ADT

- Let's associate each node with a **position**, p, that supports these operations:
  - **p*: Provides **access** to the node associated with *p*
  - *p*.left(): Return the position of the **left child** of *p* (<u>Error if *p* is external</u>).
  - *p*.right(): Return the position of **right child** of *p* (<u>Error if *p* is external</u>).
  - *p*.parent(): Return the position of **parent** of *p* (<u>Error if *p* is the root</u>).
  - *p*.isRoot(): Return **true if *p* is the root** and false otherwise.
  - *p*.isExternal(): Return **true if *p* is external** and false otherwise.
- The tree itself provides the same operations as the standard **tree ADT**, which are:
  - size(): Return the **number of nodes** in the tree.
  - empty(): Return **true if the tree is empty** and false otherwise.
  - root(): Return the **position for the root** (<u>Error if tree is empty</u>).
  - positions(): Return a **list of positions of all nodes** in the tree.

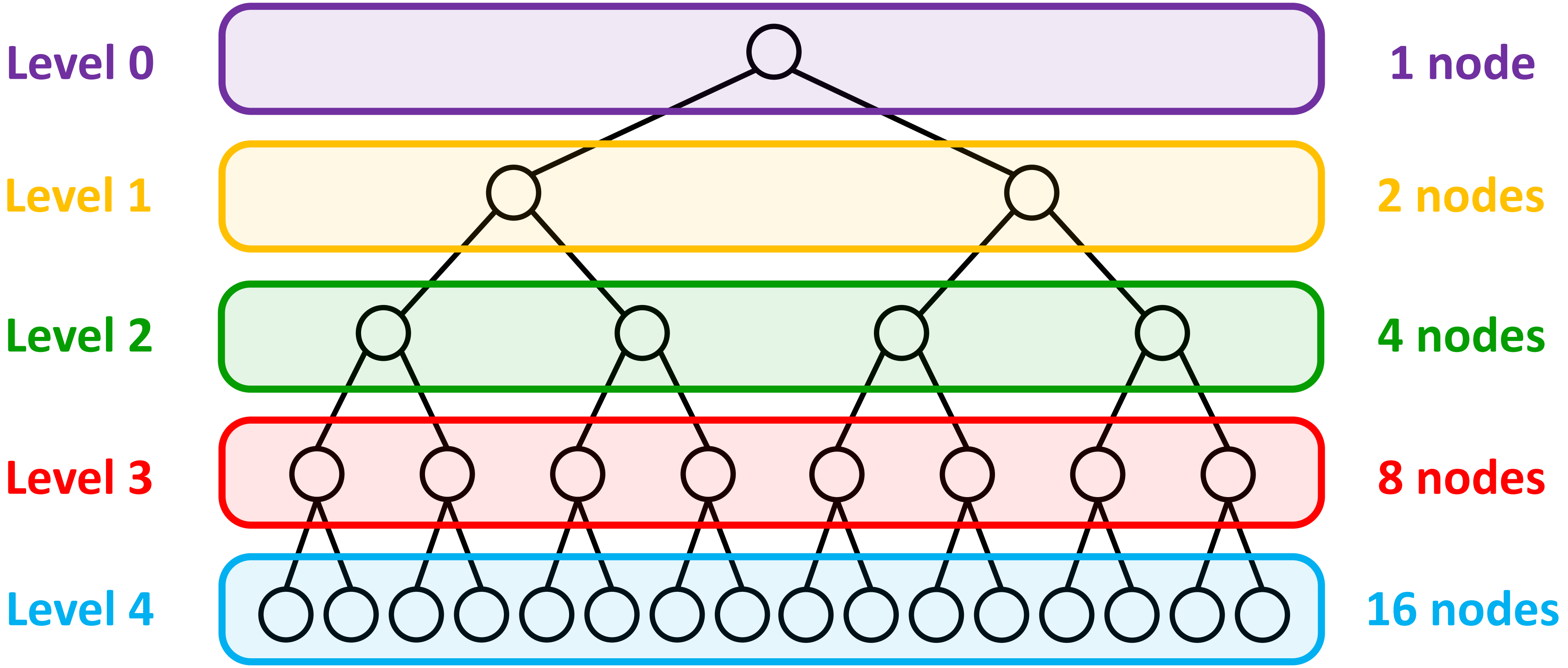
LEVELS OF A TREE

- For any given tree, the set of nodes at **depth x** is called "**Level x**"



LEVELS OF A BINARY TREE

- What is the **maximum number of nodes in Level x** of a **binary tree**?



- In a binary tree, the maximum number of nodes in Level x is 2^x

BINARY TREE PROPERTIES

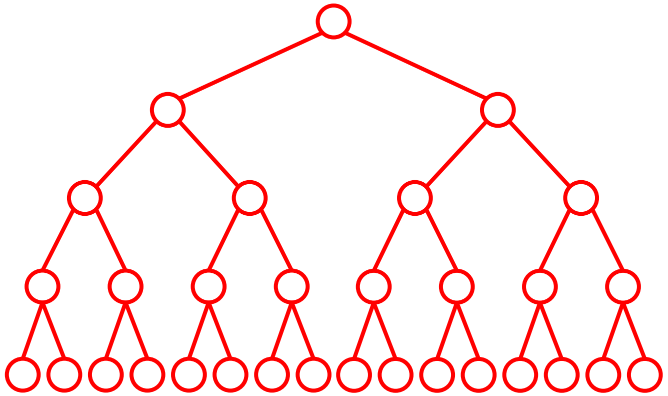
In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

T has the following properties:

- $h+1 \leq n \leq 2^{h+1}-1$

What is the **largest** possible number of **nodes** in a binary tree of **height h**?



h = 4
n = 31

What is the **smallest** possible number of **nodes** in a binary tree of **height h**?

h = 4
n = 5

BINARY TREE PROPERTIES

In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

T has the following properties:

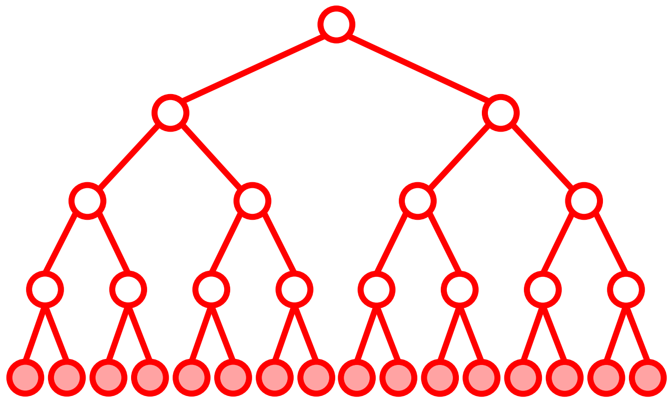
- $h + 1 \le n \le 2^{h+1} - 1$
- $1 \le n_E \le 2^h$

What's the **largest** No. of **external nodes** in a binary tree of **height** h?

What's the **smallest** No. of **external nodes** in a binary tree of **height** h?

$h = 4$

$n_E = 1$



$h = 4$

$n_E = 16$

BINARY TREE PROPERTIES

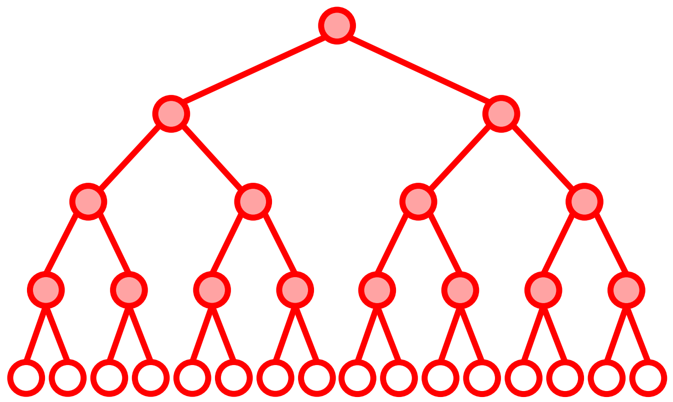
In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

T has the following properties:

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$

What's the **largest** No. of **internal nodes** in a binary tree of **height h**?



$h = 4$

$n_I = 15$

What's the **smallest** No. of **internal nodes** in a binary tree of **height h**?

$h = 4$

$n_I = 4$

BINARY TREE PROPERTIES

In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

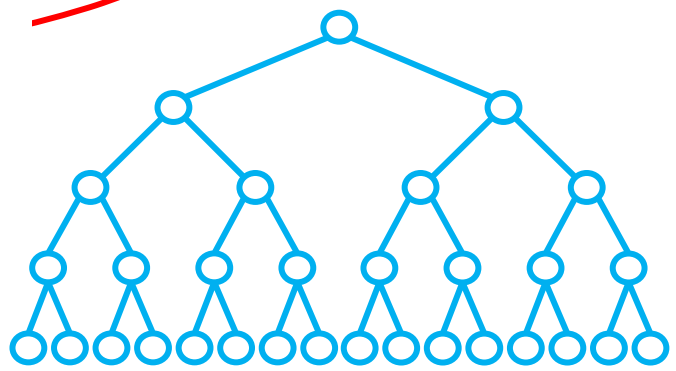
T has the following properties:

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n+1) - 1 \leq h \leq n - 1$

What's the <u>largest</u> **height** of a binary tree containing n nodes?

$n = 5$
$h = 4$

What's the <u>smallest</u> **height** of a binary tree containing n nodes?



$n = 31$
$h = \log(32) - 1$

BINARY TREE PROPERTIES

In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

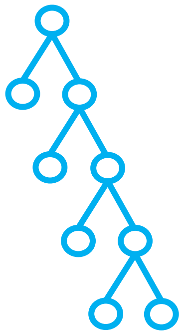
T has the following properties:

- $\boxed{h+1} \leq n \leq 2^{h+1}-1$
- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n+1) - 1 \leq h \leq n-1$

$n = 5$
$h = 4$

What if T is proper?

- $2h+1 \leq n \leq 2^{h+1}-1$
- $??? \leq n_E \leq 2^h$
- $??? \leq n_I \leq 2^h - 1$
- $\log(n+1) - 1 \leq h \leq ???$



$n = 9$
$h = 4$

BINARY TREE PROPERTIES

In a binary tree T:

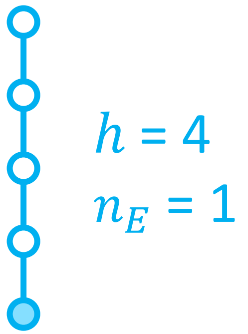
- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

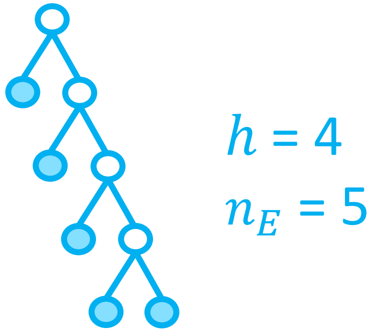
T has the following properties:

- $h+1 \le n \le 2^{h+1}-1$
- $\boxed{1} \le n_E \le 2^h$
- $h \le n_I \le 2^h - 1$
- $\log(n+1)-1 \le h \le n-1$

What if T is proper?

- $2h+1 \le n \le 2^{h+1}-1$
- $h+1 \le n_E \le 2^h$
- $??? \le n_I \le 2^h - 1$
- $\log(n+1)-1 \le h \le ???$





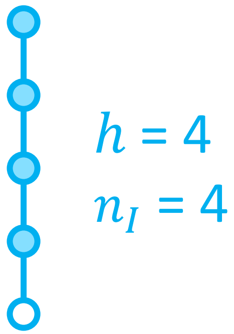
BINARY TREE PROPERTIES

In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

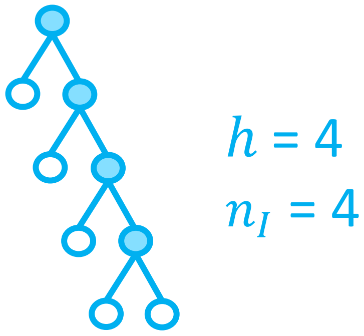
T **has the following properties:**

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $1 \leq n_E \leq 2^h$
- $\boxed{h} \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq n - 1$



What if T is proper?

- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq ???$



BINARY TREE PROPERTIES

In a binary tree T:

- n = No. of nodes
- h = tree height
- n_E = No. of external nodes
- n_I = No. of internal nodes

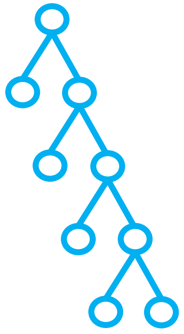
T has the following properties:

- $h + 1 \le n \le 2^{h+1} - 1$
- $1 \le n_E \le 2^h$
- $h \le n_I \le 2^h - 1$
- $\log(n + 1) - 1 \le h \le \boxed{n - 1}$

n = 5
h = 4

What if T is proper?

- $2h + 1 \le n \le 2^{h+1} - 1$
- $h + 1 \le n_E \le 2^h$
- $h \le n_I \le 2^h - 1$
- $\log(n + 1) - 1 \le h \le (n - 1)/2$



n = 9
h = 4

BINARY TREE PROPERTIES

Proposition: In a nonempty proper binary tree T, the number of external nodes is one more than the number of internal nodes, i.e., $n_E = n_I + 1$.

Justification: by induction:

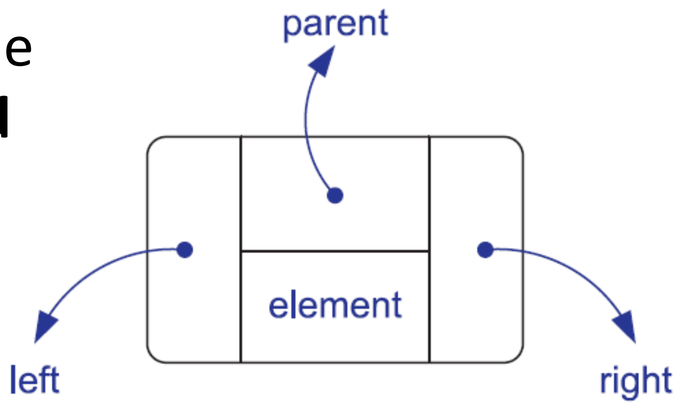
- If n = 1, we have a single node (the root) which is external. Thus n_E = 1; n_I = 0.
- The root of T may have two subtrees: T' and T''. Since these are smaller than T, we may assume that they satisfy the proposition as well, i.e.: $n'_E = (n'_I + 1)$ and $n''_E = (n''_I + 1)$
- Note that:

$$n_I = n'_I + n''_I + 1 \text{ (i.e., plus the root)}$$

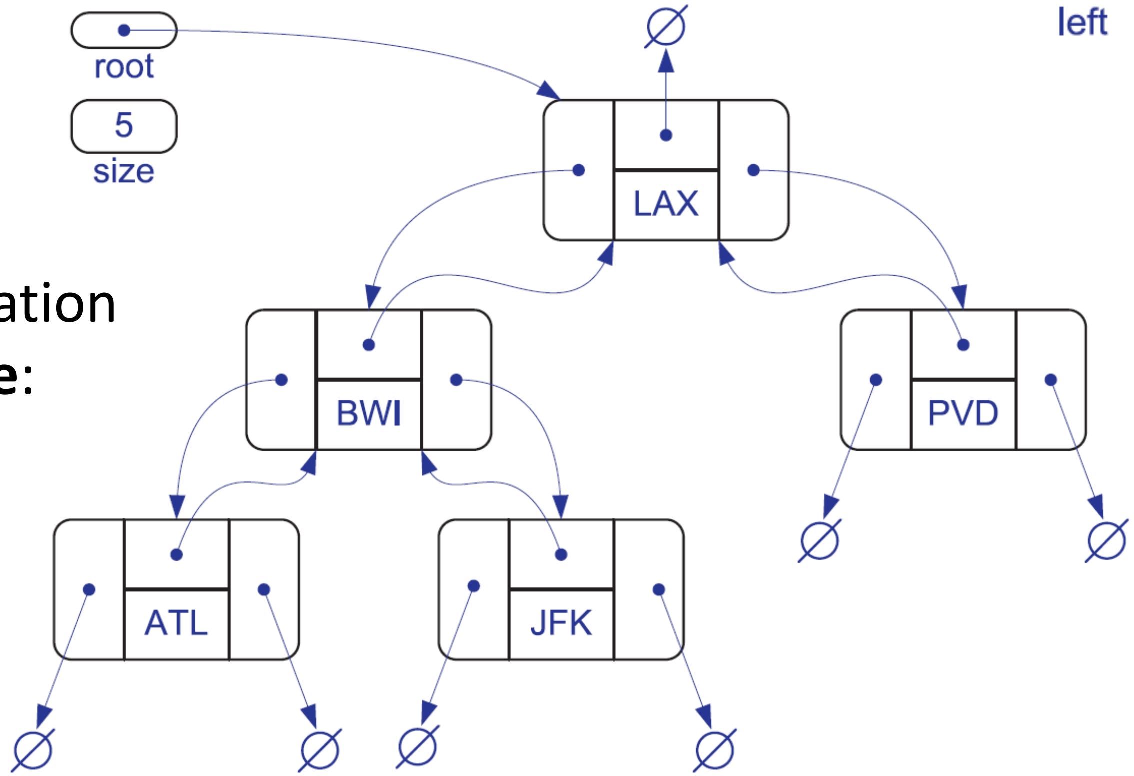
$$n_E = n'_E + n''_E = (n'_I + 1) + (n''_I + 1) = n'_I + n''_I + 2 = n_I + 1$$

BINARY TREE – LINKED LIST IMPLEMENTATION

- Here is an illustration of a node, which has the **element**, the **parent** (which is NULL if the node is the root), the **left child** and the **right child**



- Here is an illustration of the entire **tree**:



BINARY TREE – LINKED LIST IMPLEMENTATION

- Following is a possible structure/implementation of a position class in a binary tree:

```
class Position {
public:
    Elem& operator*();
    Position left() const;
    Position right() const; /*you can add children() for traversal purposes */
    Position parent() const;
    bool isRoot() const;
    bool isExternal() const;
private:
    Node* v; Position(Node* u);
};
```

- And this is a possible structure/implementation of a node class in a binary tree:

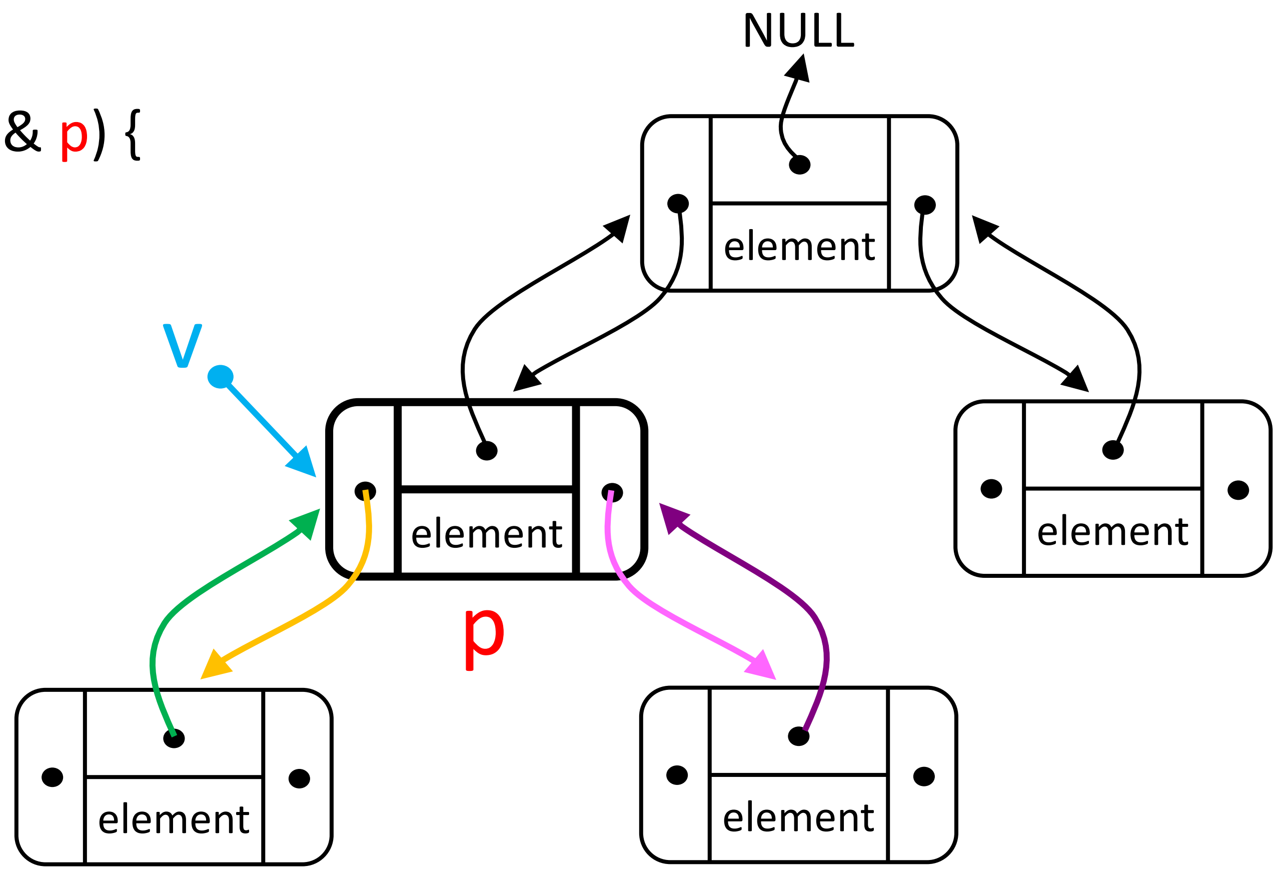
```
class Node {
  private:
     Elem elm; // element value
     Node* left; // left child
     Node* right; // right child
     Node* par; // parent
  public:
     Node() : elm(), par(NULL), left(NULL), right(NULL) { }
};
```

BINARY TREE – LINKED LIST IMPLEMENTATION

expandExternal(): A method that **creates two children** of position p.

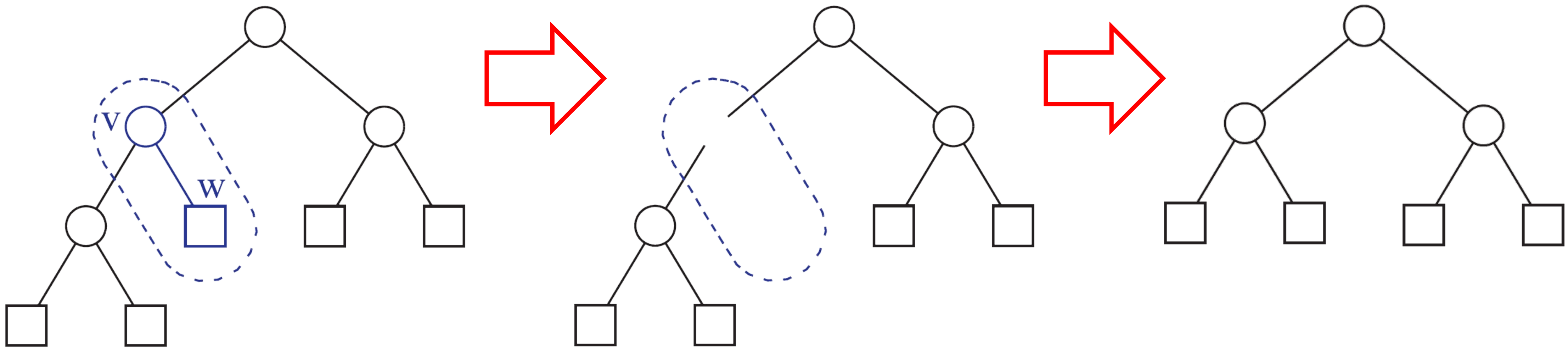
```
void expandExternal(const Position& p) {
    Node* v = p.v;
    v−>left = new Node;
    v−>left−>par = v;
    v−>right = new Node;
    v−>right−>par = v;
    n += 2;  //update No. of nodes

}
```



BINARY TREE – LINKED LIST IMPLEMENTATION

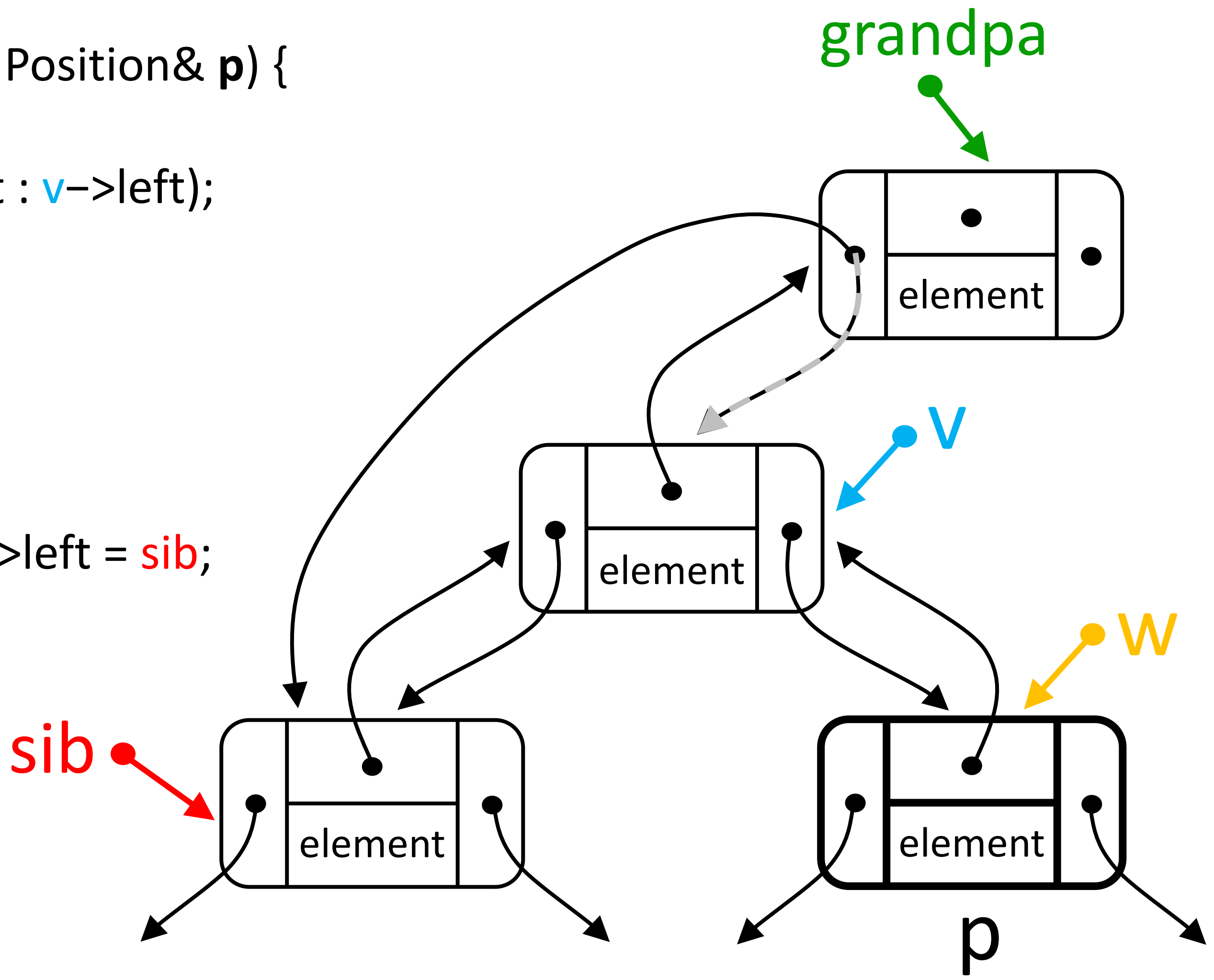
removeAboveExternal(): it **removes an external node along with its parent**.



BINARY TREE – LINKED LIST IMPLEMENTATION

removeAboveExternal(): it **removes an external node along with its parent**.

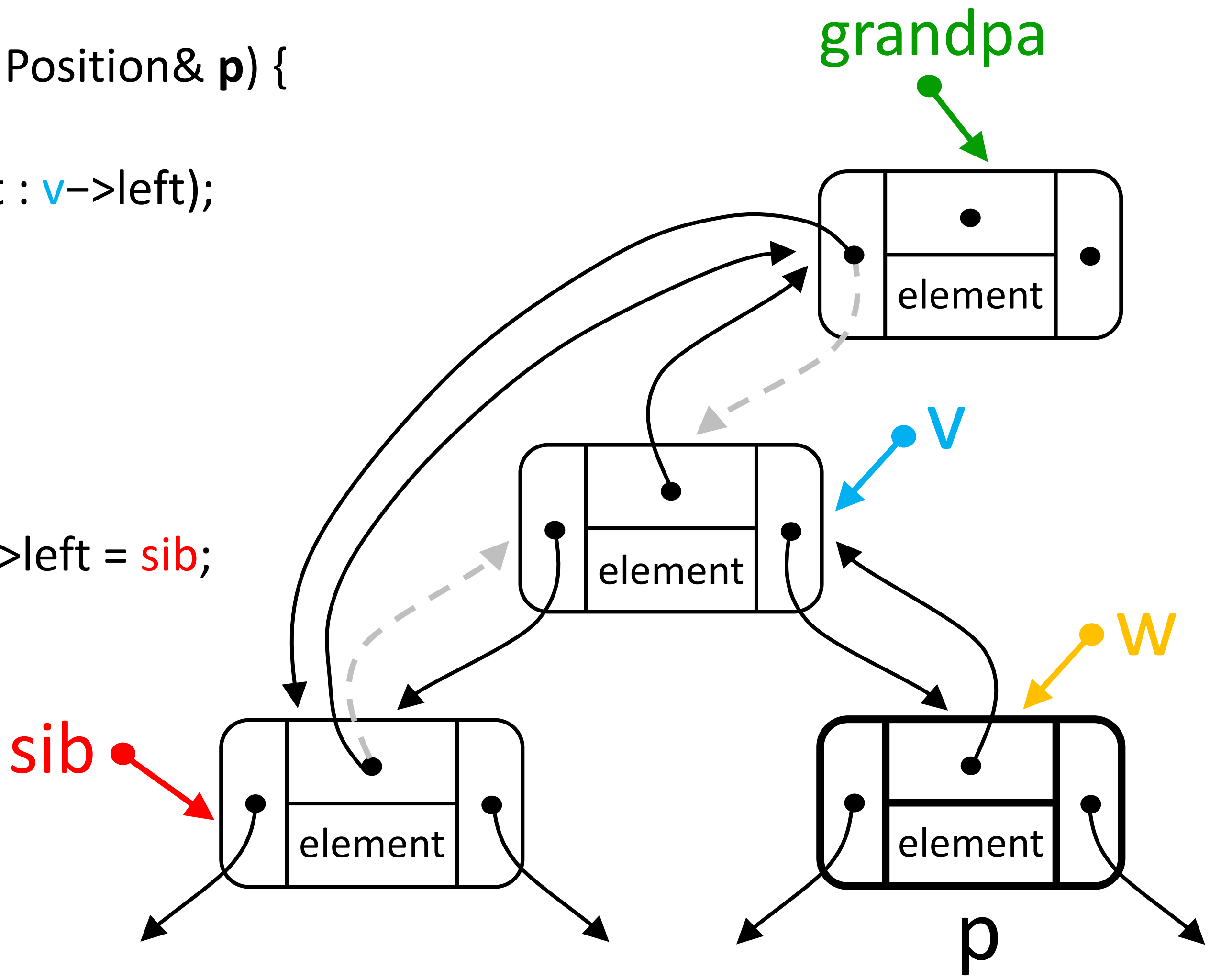
```
Position removeAboveExternal(const Position& p) {
    Node* w = p.v; Node* v = w–>par;
    Node* sib = (w == v–>left ? v–>right : v–>left);
    if (v == root) {
        root = sib;
        sib–>par = NULL;
    }else{
        Node* grandpa = v–>par;
        if (v == grandpa –>left) grandpa –>left = sib;
        else grandpa –>right = sib;
```



BINARY TREE – LINKED LIST IMPLEMENTATION

removeAboveExternal(): it **removes an external node along with its parent**.

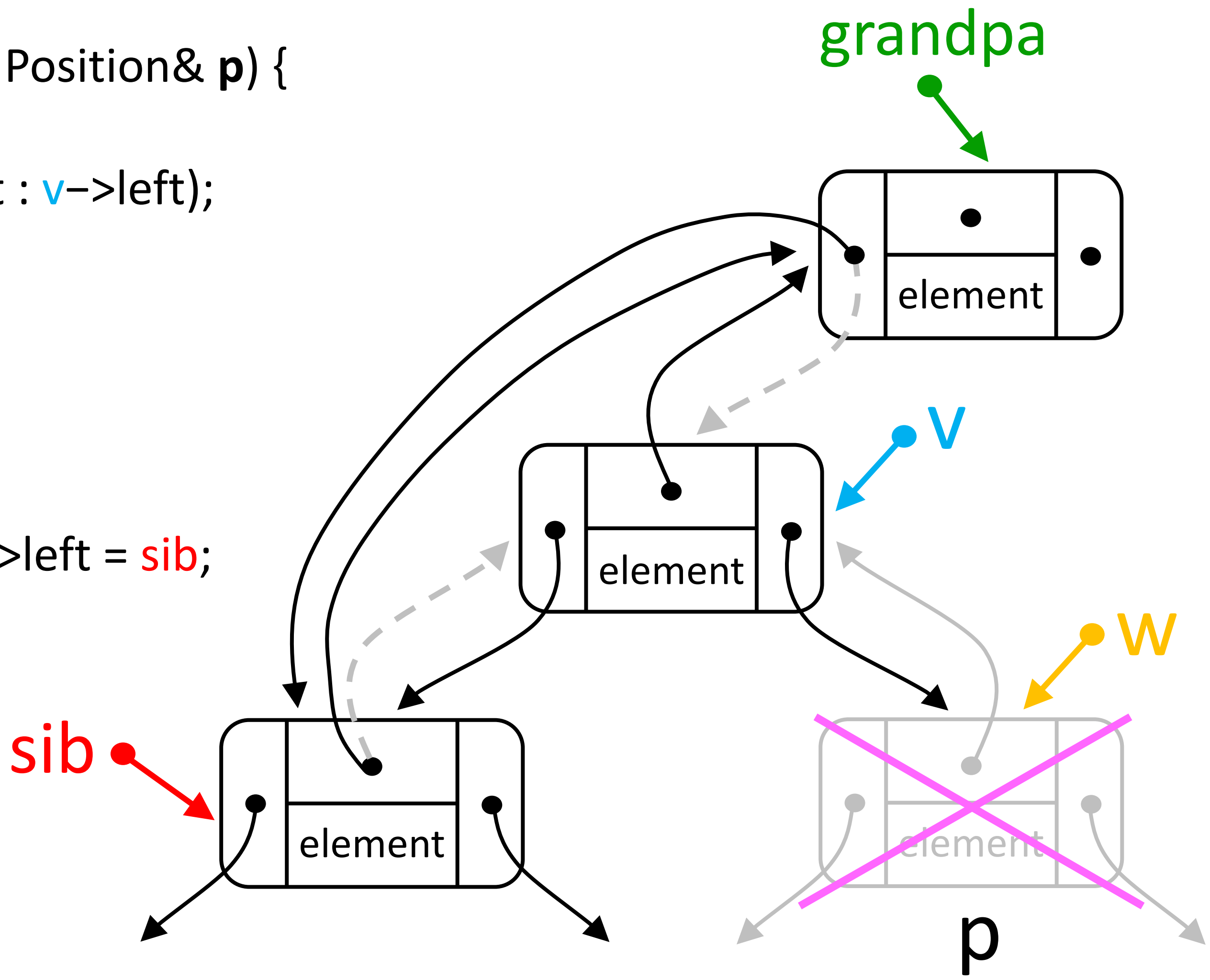
```
Position removeAboveExternal(const Position& p) {
    Node* w = p.v; Node* v = w->par;
    Node* sib = (w == v->left ? v->right : v->left);
    if (v == root) {
        root = sib;
        sib->par = NULL;
    }else{
        Node* grandpa = v->par;
        if (v == grandpa ->left) grandpa ->left = sib;
        else grandpa ->right = sib;
        sib->par = grandpa;
    }
```



BINARY TREE – LINKED LIST IMPLEMENTATION

removeAboveExternal(): it **removes an external node along with its parent**.

```
Position removeAboveExternal(const Position& p) {
   Node* w = p.v; Node* v = w->par;
   Node* sib = (w == v->left ? v->right : v->left);
   if (v == root) {
      root = sib;
      sib->par = NULL;
   }else{
      Node* grandpa = v->par;
      if (v == grandpa ->left) grandpa ->left = sib;
      else grandpa ->right = sib;
      sib->par = grandpa;
   }
   delete w;
```



BINARY TREE – LINKED LIST IMPLEMENTATION

removeAboveExternal(): it **removes an external node along with its parent**.

```
Position removeAboveExternal(const Position& p) {
    Node* w = p.v; Node* v = w−>par;
    Node* sib = (w == v−>left ? v−>right : v−>left);
    if (v == root) {
        root = sib;
        sib−>par = NULL;
    }else{
        Node* grandpa = v−>par;
        if (v == grandpa −>left) grandpa −>left = sib;
        else grandpa −>right = sib;
        sib−>par = grandpa;
    }
    delete w;
    delete v;
    n −= 2; // update No. of nodes
    return Position(sib);
}
```