Stack. Representations

initEmpty
push **O(1)**
pop
isEmpty

- over array (/ vector)
- over linked-list

Queue. Representations

initEmpty
enqueue **O(1)**
dequeue
isEmpty

- over array (/ vector)
- over linked-list

Deque. Representations

initEmpty
push_back **O(1)**
push_front
pop_back
pop_front
isEmpty

- over array (/ vector)
- over linked-list

STL: Stack, Queue. Issues

std::stack - container adaptor

template < class T, class Container = deque<T> > class queue;

- the underlying container
  - back()
  - push_back()
  - pop_back()

std::queue - container adaptor

template < class T, class Container = deque<T> > class queue;

- the underlying container
  - front()
  - back()
  - push_back()
  - pop_front()

STL: deque

Deque:

- Specific libraries may implement deque in different ways

    generally as some form of dynamic array

- with efficient insertion and deletion of elements at the beginning and at its end.

Vector vs. **deque**

provide a very similar interface and can be used for similar purposes

internally can be quite different

Vector: use a single array

Deque: deques are not guaranteed to store all its elements in contiguous storage locations

can be scattered in different chunks of storage

ex.: implemented as a vector of vectors

Java: Stack, Queue, Deque. Issues

Java™ Platform
Standard Ed. 7

Interface :	Queue
Subinterface:	Deque
Implementing Classes:	ArrayDeque
	LinkedList

Java: Stack, Queue, Deque. Issues

Java™ Platform
Standard Ed. 7

public class Stack<E>
extends Vector<E>

- Use Deque instead of Stack

  Deque<Integer> stack = new ArrayDeque<Integer>();

  A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations,

  which ***should be used in preference*** to this class.

Lists. Variations

- Multiple Values Per Node

```
ULNode = record
            next: Position
            elemCount: integer
            elemData: array[0..MAX-1] of TElement
        end
Position = ^ULNode
```

Terminology: **unrolled linked list**

Lists. Variations

- with multiple links from nodes

Node = ***record:***

info: **TElement**

links: List<Position> ***//array, record, linked***

end

Terminology: General Linked Lists

Multiply Linked List

Multi-Linked Lists

Multidimensional arrays

For example:

C++ : described as "arrays of arrays".

```
int matrix [3][5];
```

```
matrix[1][3]
```

the second element vertically and fourth horizontally

Using STL : Vector based multi-dim. array

```
vector<vector<double> > array2D
```

// Put some values in

```
array2D[1][2] = 6.0
```

Jagged arrays

many rows of varying lengths

For example:

```
int jagged_row0[] = {0,1};
int jagged_row1[] = {1,2,3};
int *jagged[] = { jagged_row0,
                            jagged_row1 };
```

Lists. Variations. Application

Sparse Matrices

Sparse Matrix

- sparse … many elements are zero
- dense … few elements are zero

Structured sparse

- diagonal
- tridiagonal
- upper/lower triangular (?)

Unstructured sparse matrix. Example

- Web page matrix.
  - web pages are numbered 1 through n
  - web(i,j) = number of links from page i to page j
- n ~ 10^9
- n x n array => 10^{18} consecutive position

  (linear representation)
- each page links to 10 (say) other pages on average

  on average there are 10 nonzero entries per row
- space needed for non-*empty* elements is approximately

  1 billion x 10 = 10^{10} consecutive elements

- **sparse matrix** is a matrix populated primarily with zeros
- How to store it will depend at least partly on exactly how sparse it is, and what you want to do.
  - For some applications, just treating it as a regular matrix is just fine (especially if the dimensions aren't very big). A 15x15 sparse matrix isn't a big memory hog.
  - suppose the sparse matrix has 10240x10240 elements, and you're using 8-byte floating point numbers: how much memory do you need?
- **Representation of sparse matrix (idea)**
  - list: keep information about non-zero cells
  - links: for fast access from one non-zero cell to the next, on the same raw/column ← general linked list

sparse matrix

Unstructured Sparse Matrices. Representations

linear list in row-major order.

- nonzero elements of the sparse matrix
- each nonzero element is represented by a triple

(row, column, value)

➔the list of triples (linked , …)

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

List:

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

One Linear List Per Row

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

row1 = [(3, 3), (5,4)]

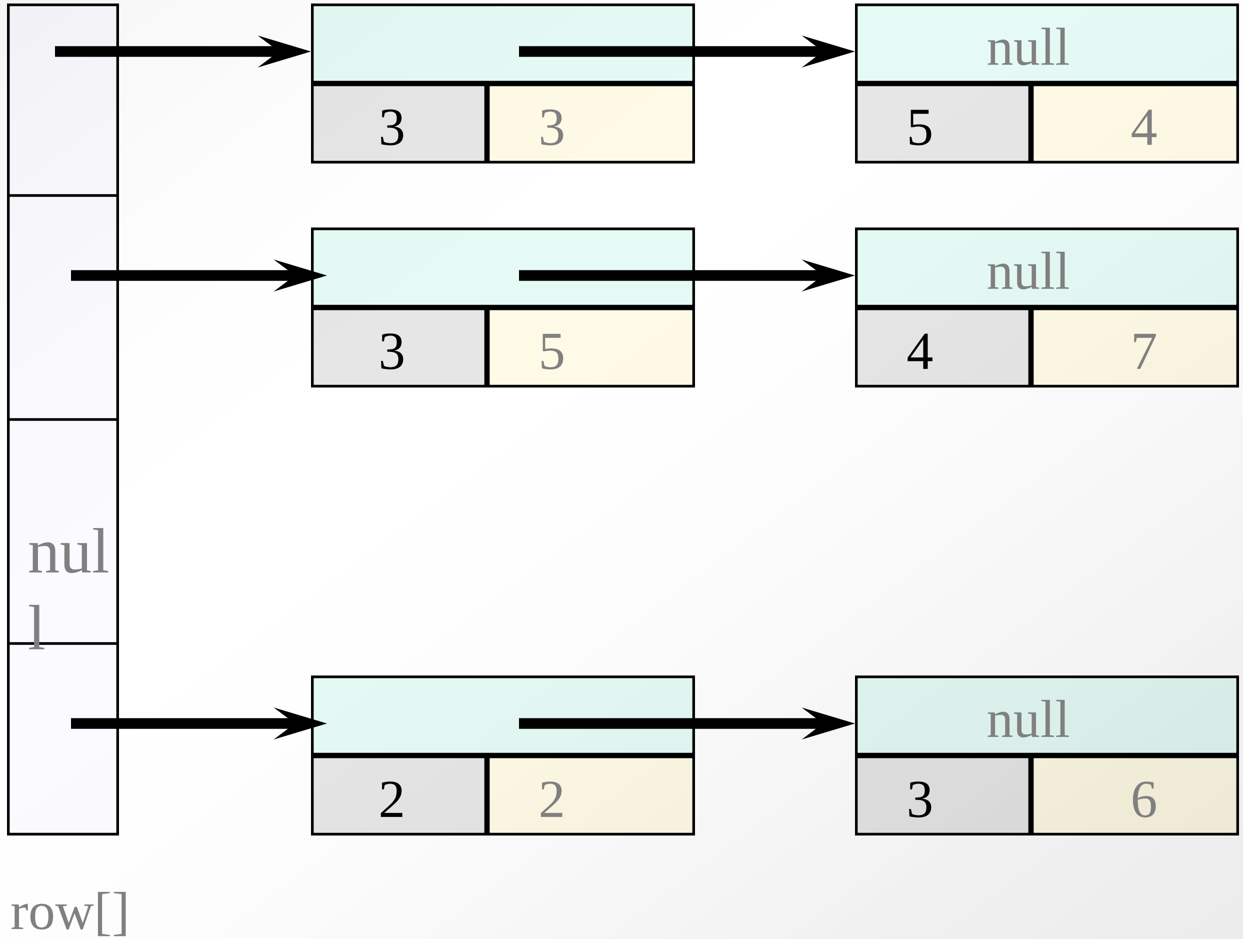
row2 = [(3,5), (4,7)]

row3 = []

row4 = [(2,2), (3,6)]

One Linear List Per Row (Linked)

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



Similar: one list per column

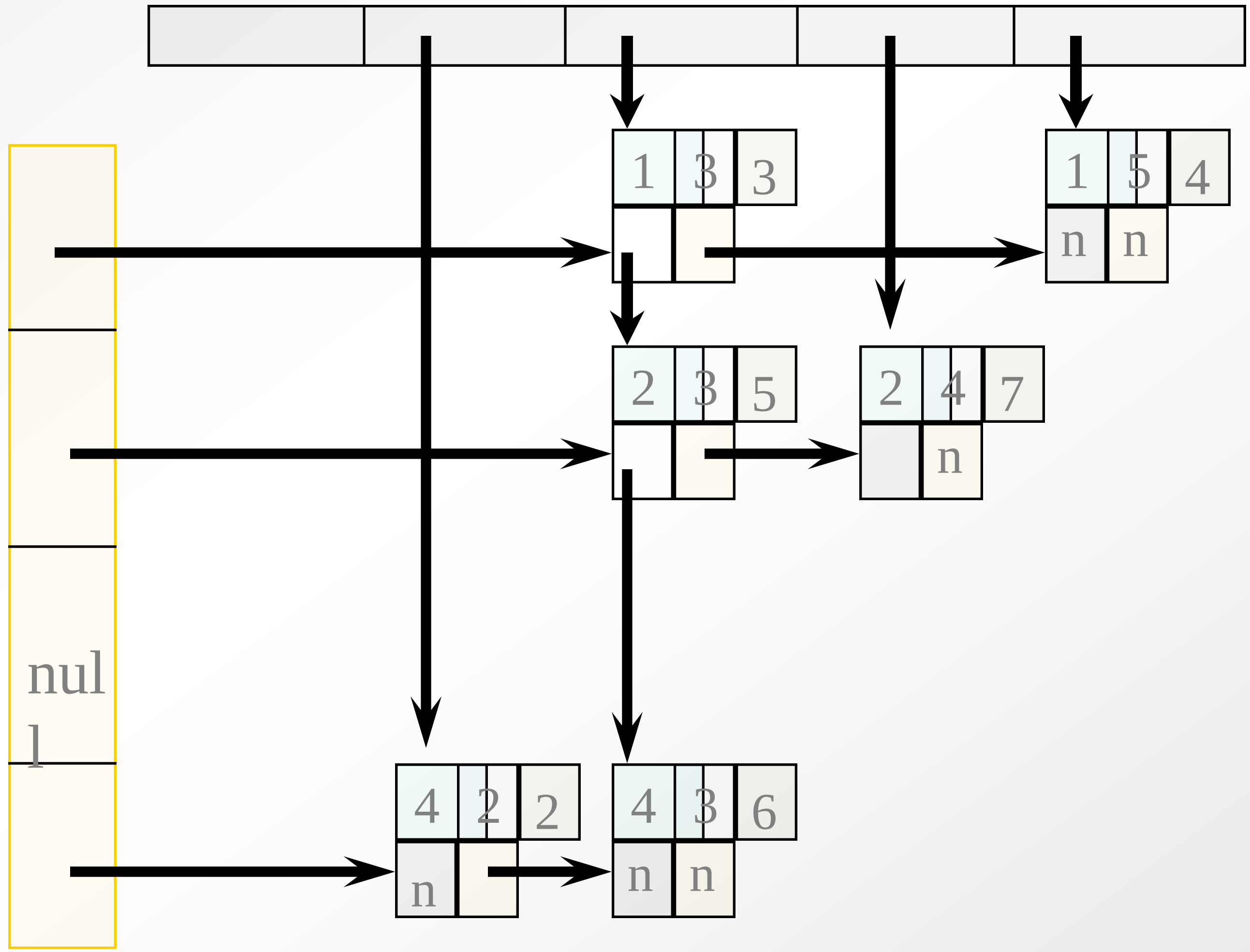
Orthogonal Lists

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



Variation: use circular lists

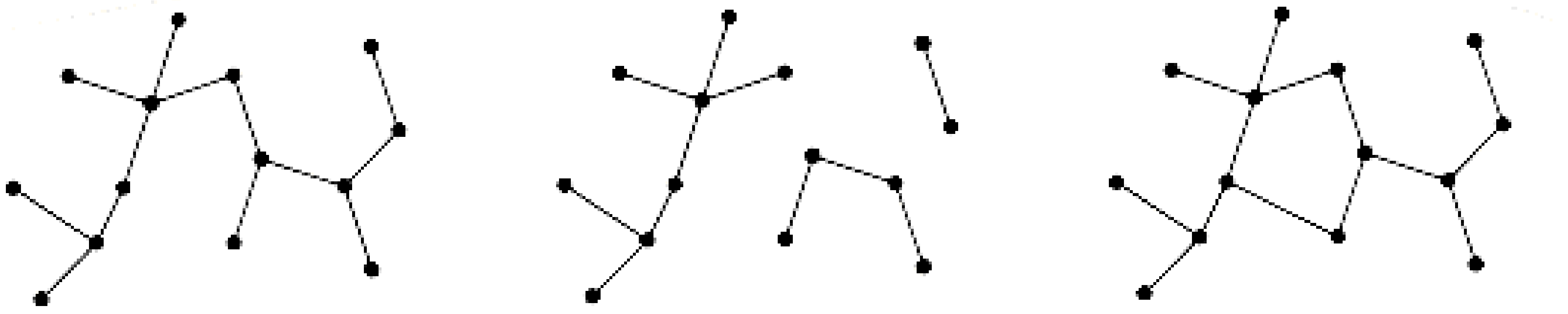
```
#include <iostream>
#include <string>

typedef  char * CString;
typedef CString* TJaggedArray;
//typedef char** TJaggedArray;

void readStrings(TJaggedArray& strs)
{
    int n;
    char myStr[200];
    std::cin >> n;
    strs = new CString[n+1];
    strs[n]=NULL;
    for(int i=0; i<n;i++){
        std::cin >> myStr;
        strs[i]=new char[strlen(myStr)+1];
        strcpy(strs[i],myStr);
    }
}

void printStrings(const TJaggedArray& strs)
{
    for(int i = 0; strs[i] != NULL ; i++) {
        std::cout << strlen(strs[i]) <<" : " << strs[i] << std::endl;
    }
}
```

Tree



(a) (b) (c)

Tree

Free tree **(graph theory)**

- any two vertices are connected
- no cycles

Rooted tree

- ***+ root*** : one of the nodes is distinguished from the others

Ordered tree **(most used in computer science)**

- is a rooted tree in which the children of each node are ordered

  if a node has k children, then there is a first child, a second child, . . . , and a k-th child

Data Structure → rooted, ordered tree
(for us, by default)

Tree

recursive definition

Tree:

is empty

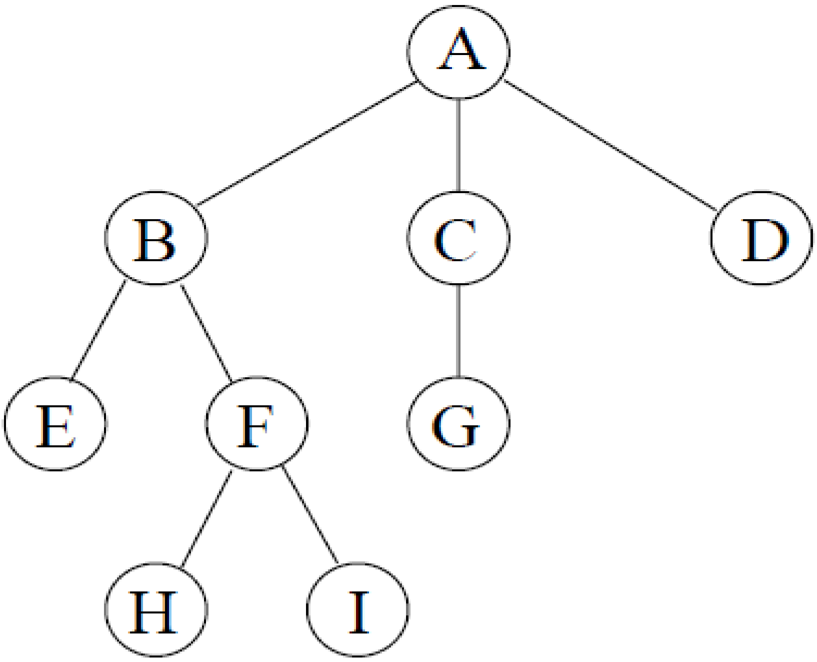
or it has a root **r** and 0 or more sub-trees

Properties:

- Each node has exactly one “*predecessor*” – its parent
- has exactly zero, one or more “*successors*” – its children

Trees

- root
- parent , children, sibling
- ancestor , descendants
- leaves , internal
- depth (level) , height
- degree



Tree

Node degree – the number of descendants

Node depth ***(level)***

- the length of the path to the root
- root – depth 0

Node height:

- the longest path from that node to a leaf (of the tree)
- (equivalent) the height of the subtree having that node as root

If the last edge on the path from the root *r* of a tree *T* to a node *x* is (*y, x*), then *y* is the ***parent*** of *x*, and *x* is a ***child*** of *y*.

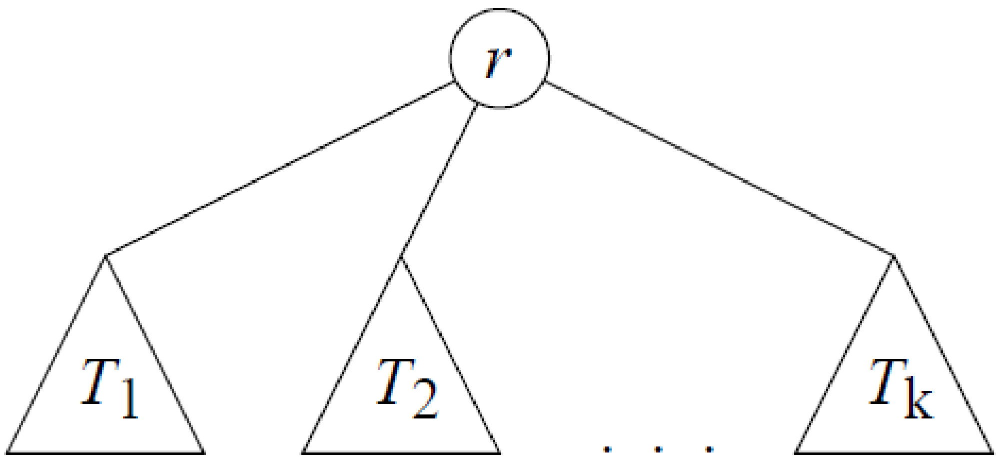
If two nodes have the same parent, they are ***siblings***.

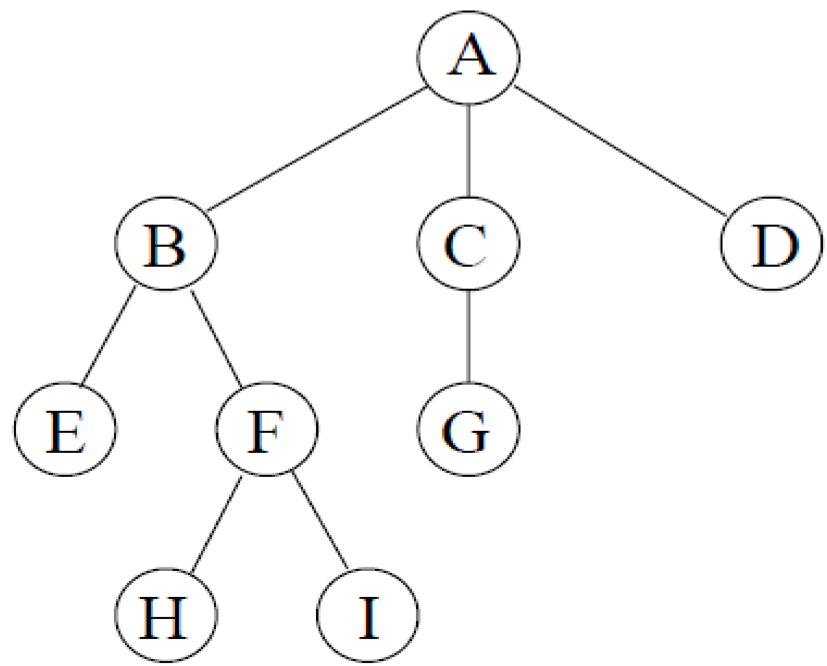
The root is the only node in *T* with no parent.

A node with no children is a ***leaf***. A non-leaf node is an ***internal node.***

k-ary tree

- A ***k-ary tree*** – each node have at most ***k*** descendants





Binary trees

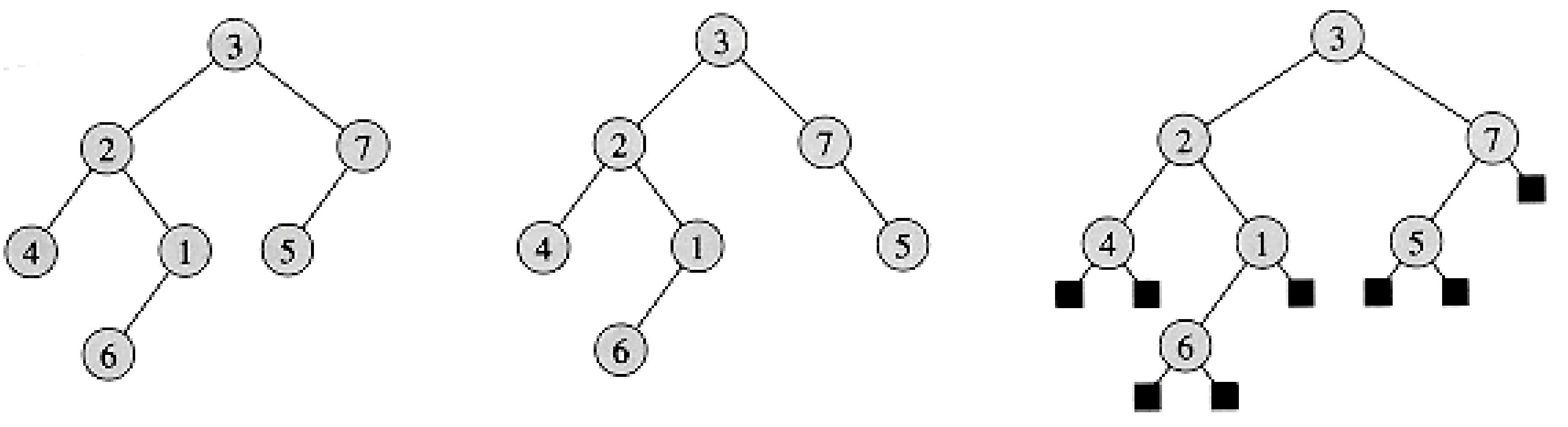
Rooted trees

each node have at most two descendants.

– first descendant is the left descendant

– second descendant is the right descendant

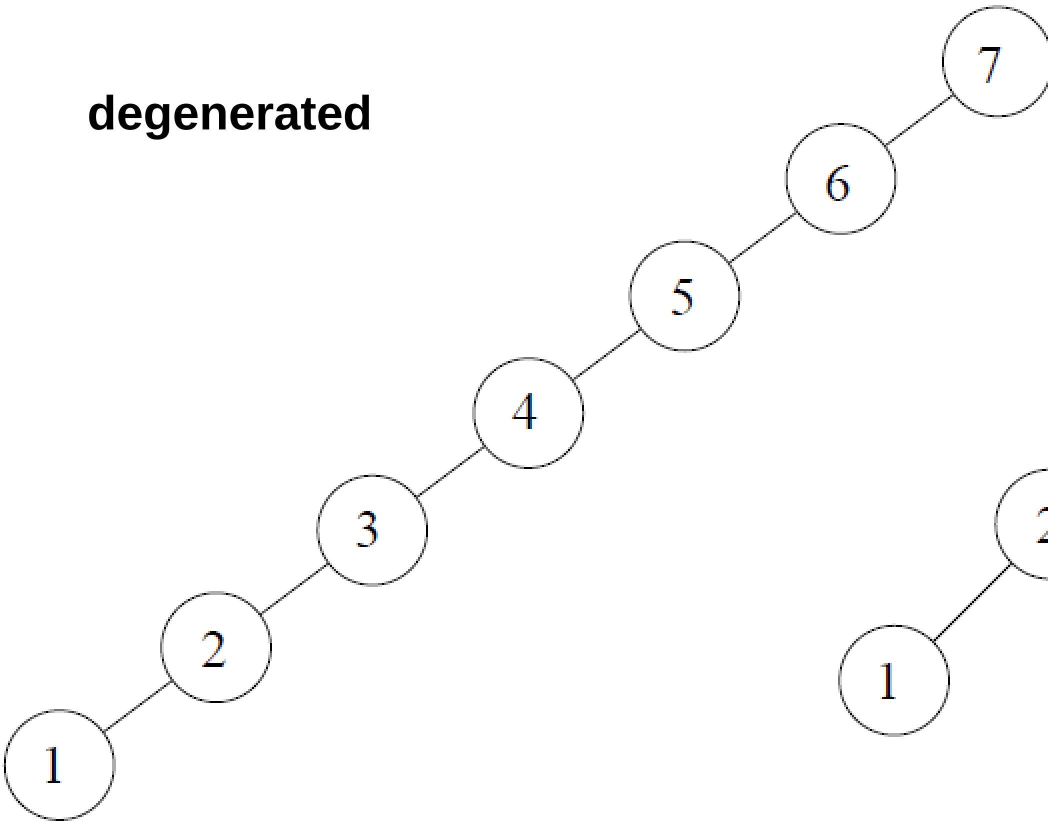
A tree with N nodes has N-1 edges



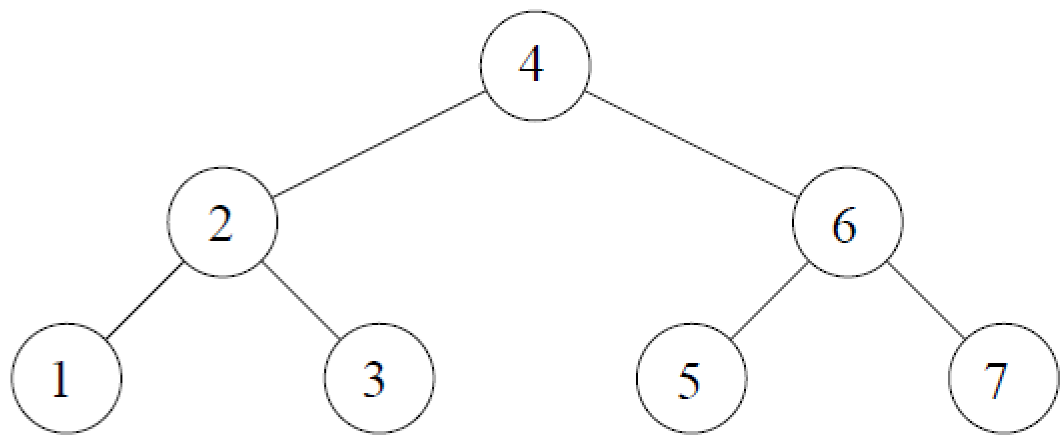
(a) (b) (c)

Binary trees

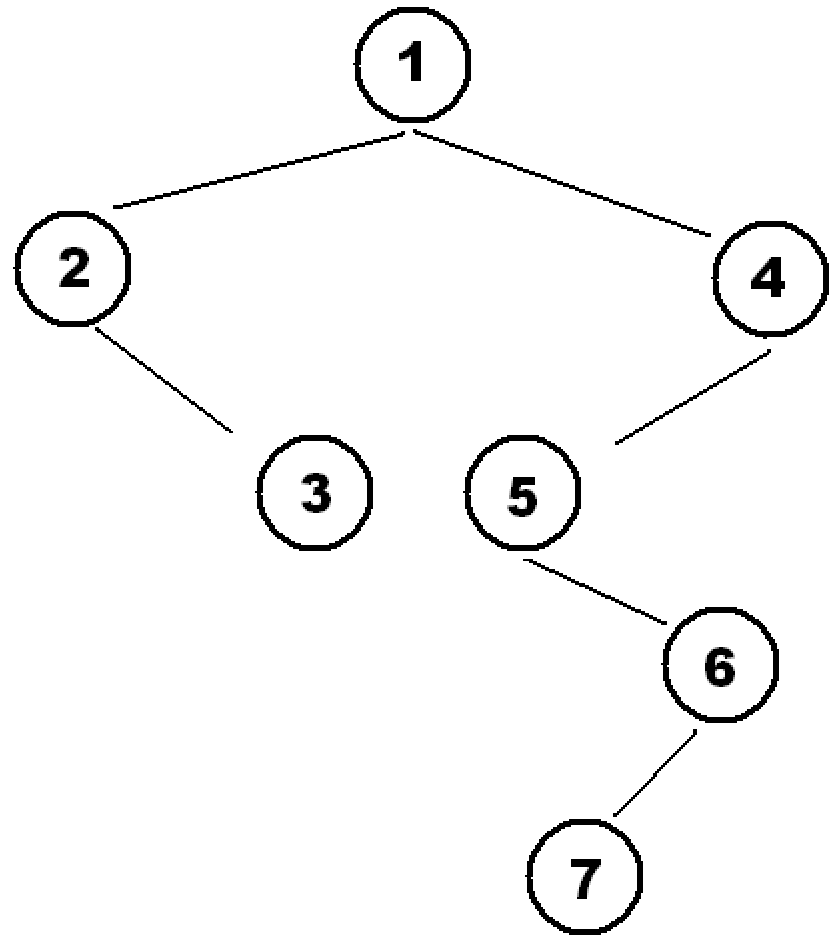
degenerated

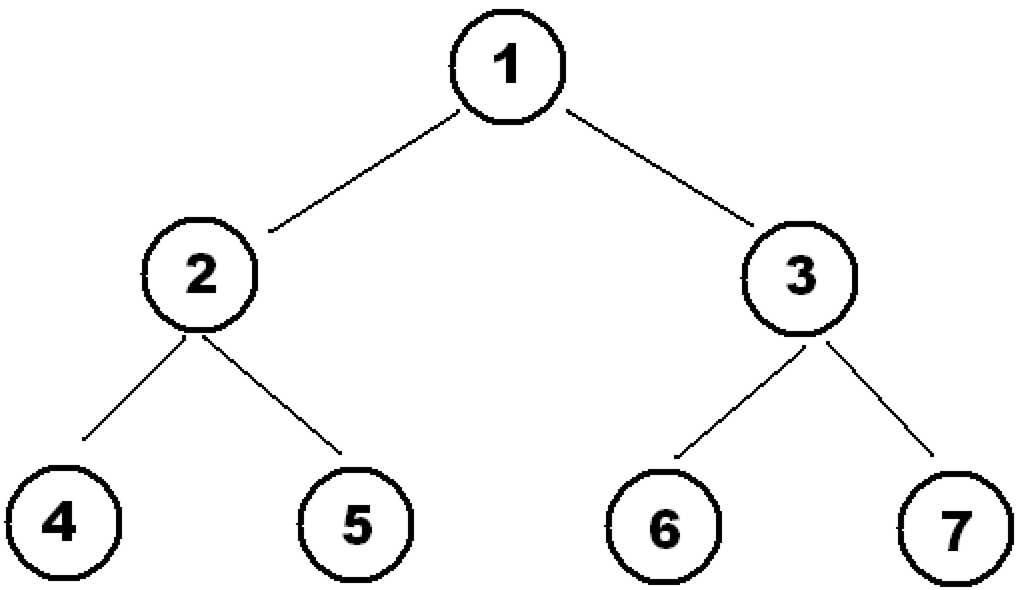


perfect



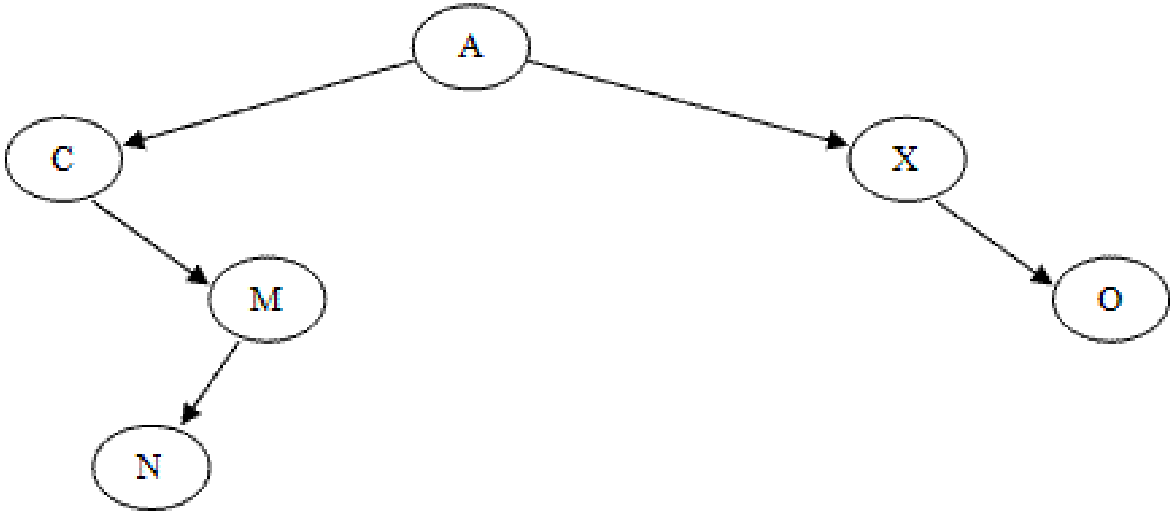
Binary trees



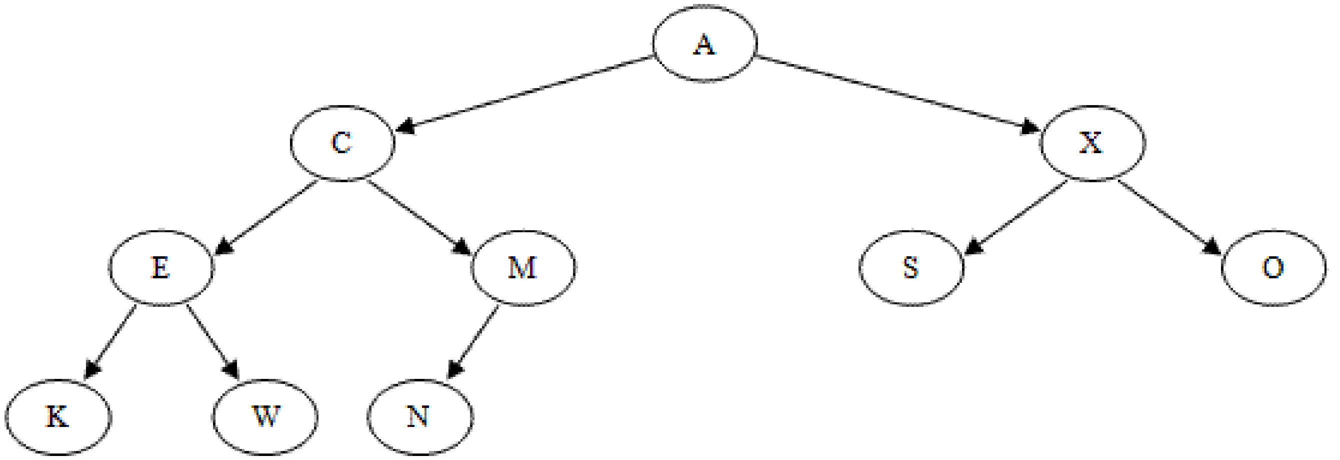


Binary trees

degenerated



(almost) complete



Binary tree types

Perfect tree:

- all leaves have the same depth
- and all internal nodes have two children

(Almost) **complete** tree:

- for each level , except possibly the deepest, the nodes have 2 children
- in the deepest level, all nodes are as far left as possible

A **degenerate** tree

- each parent node has only one child

➔ the tree will essentially behave like a linked list data structure

A **balanced binary tree**

- no leaf is much farther away from the root than any other leaf
  - different balancing schemes allow different definitions of "much farther

Binary tree types

(true or false ?)

Perfect tree:

A binary tree with all leaf nodes at the same depth.

All internal nodes have degree 2.

(Almost) **complete** tree:

A binary tree in which every level, except possibly the deepest, is completely filled.

At depth n, the height of the tree, all nodes must be as far left as possible.

http://xlinux.nist.gov/dads

(equivalent definitions)

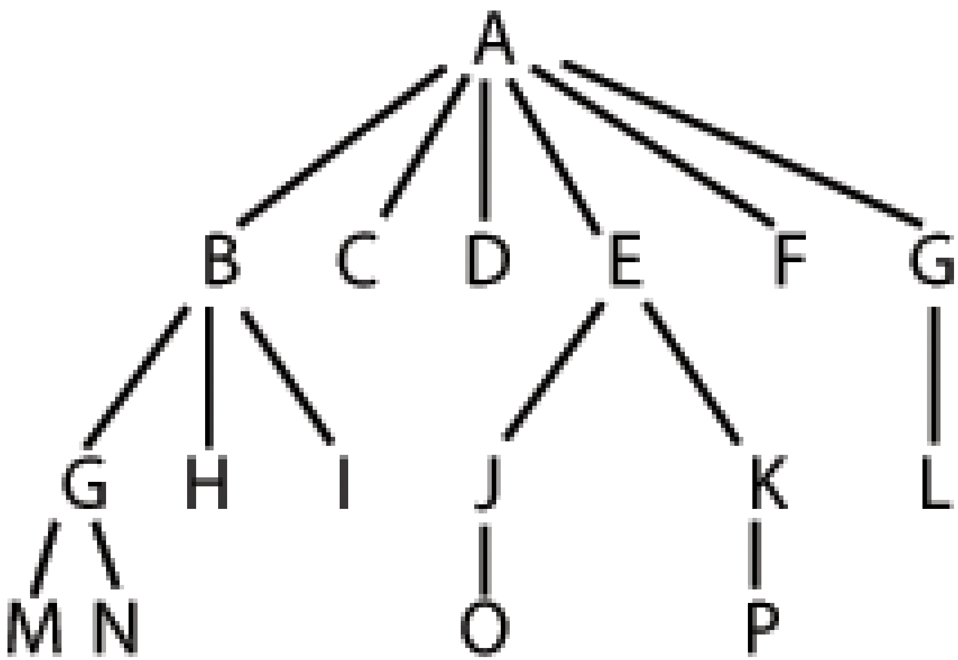
Binary tree properties

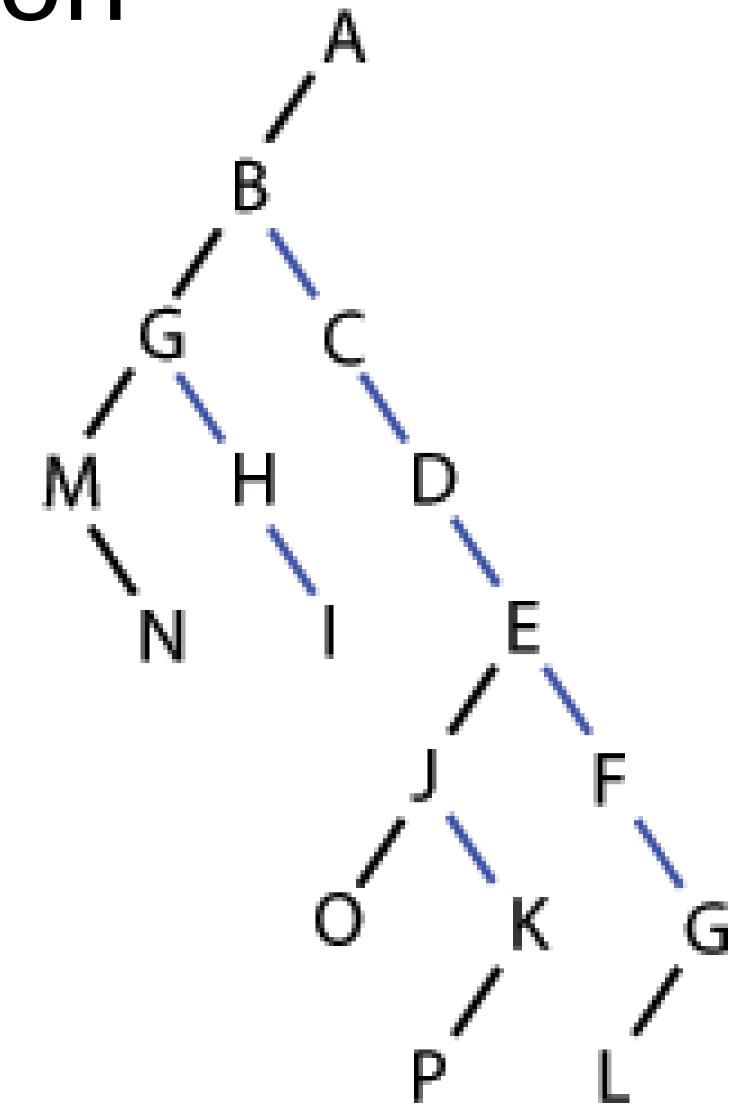
1. A tree with N nodes has N-1 edges

   (true for any tree)

2. No of nodes in a perfect binary tree with height h is $2^{h+1}-1$
3. Maximum no of nodes in a binary tree with height h is $2^{h+1}-1$
4. A binary tree with n nodes has height at least $[\log_2 n]$

Tree representation





Tree representation

(A (B (E (K, L) , F) , C (G) , D (H, I, J)))

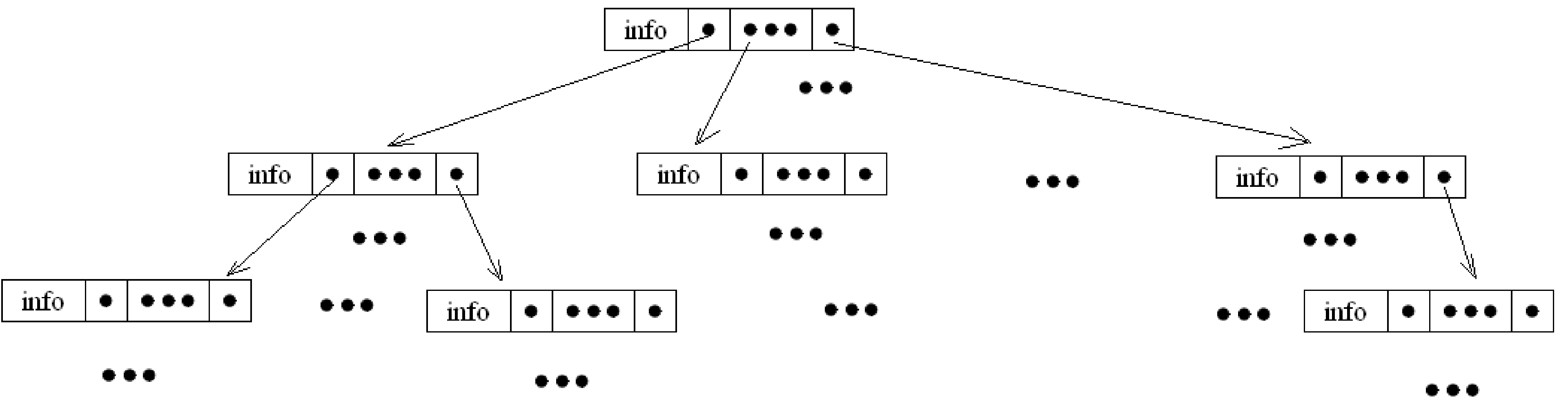
Tree representation (1)

Based on recursive definition

- Node root information
  list of subTrees

<u>remark</u>: a tree is known by knowing its root (links to subtrees)

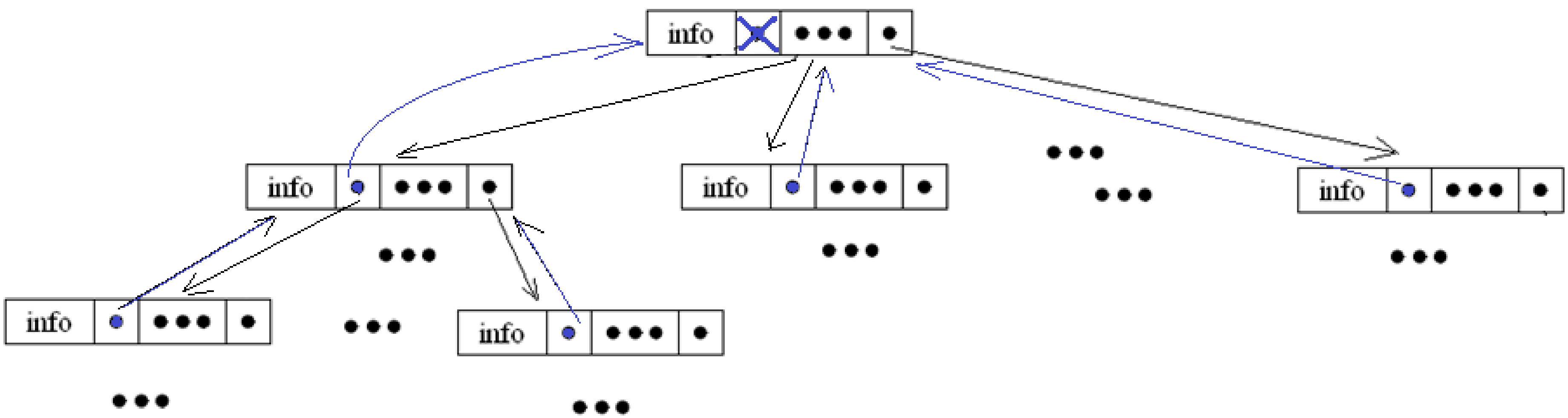
collection?



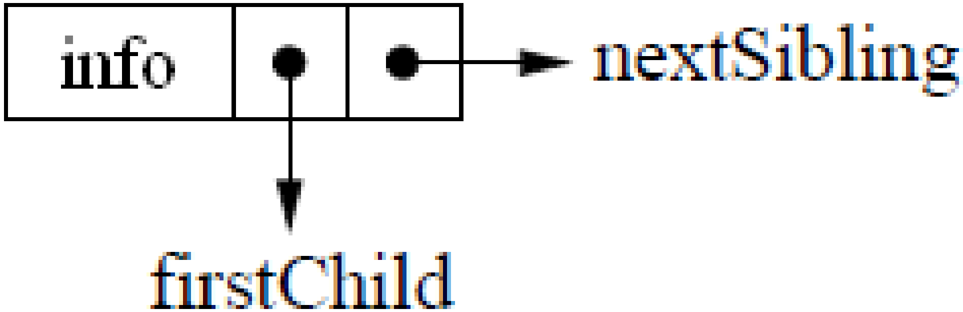
Linked representation (1)

Tree representation (1b)

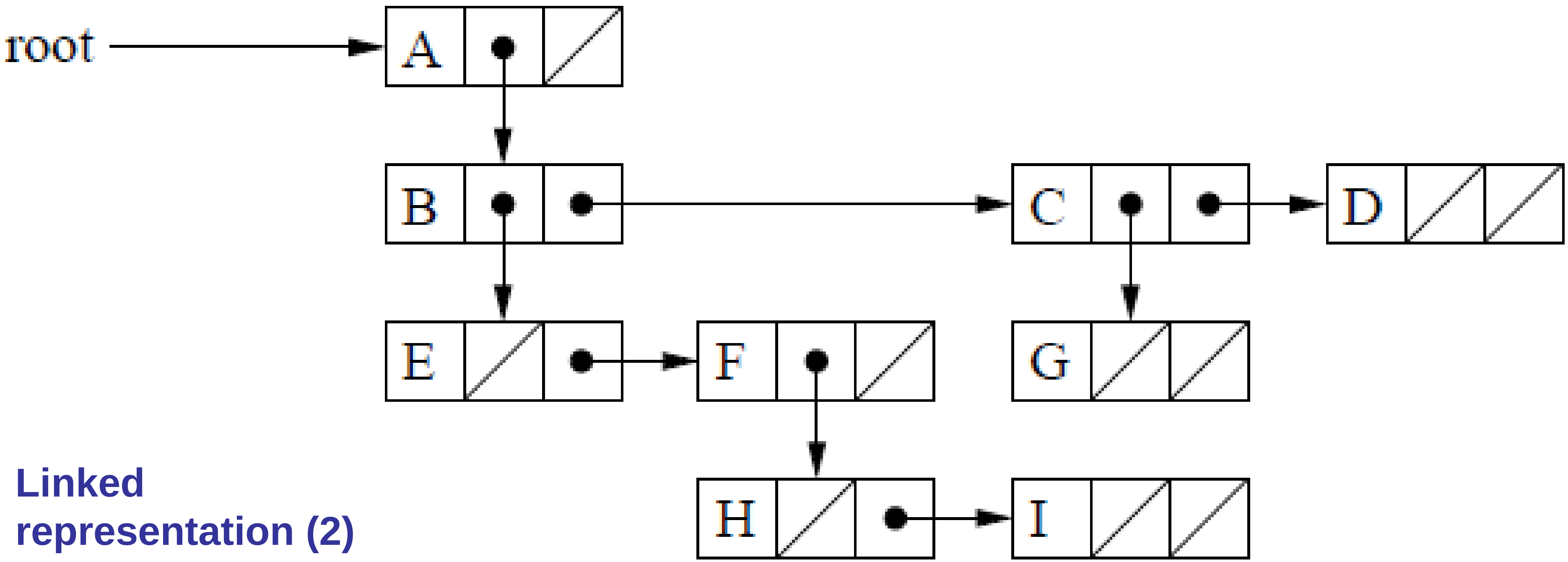
- Sometimes, a link to the parent node is also kept



Tree Representation (2)



=> sibling list



Linked representation (2)

Tree traversal

can be traversed in many ways

- depth-first traversal
- breadth-first traversal

  on levels

Representation (1) & dynamic allocation

```
TreeNode: record
            info: TElement
            left: Position
            right:Position
      end
Position: ^TreeNode
Tree: record
            root: Position
      end
```

Representation (1) & dynamic allocation

```
TreeNode: record
        info: TElement
        left: ^TreeNode
        right: ^TreeNode
    end

Tree: record
        root: ^TreeNode
    end
```

```
class TreeNode {
private:
    TElement info;
    TreeNode* left;
    TreeNode* right;
public:
    TreeNode(TElement value) {
        this->info = value;
        left = NULL;
        right = NULL;
    }
        …
};
class BinaryTree {
private:
    TreeNode* root;
public:
        …
```

Representation (1b) & dynamic allocation

TreeNode: record

info: TElement

left: ^TreeNode

right:^TreeNode

end

Tree: ^TreeNode

This representation fits the recursive definition of binary tree.

For some recursive algorithms, we are going to use this representation.

Representation (1) & over arrays

```
TreeNode: record
        info: TElement
        left: Integer
        right: Integer
    end

Tree: record
      root: Integer
      nodes: array [1..MAX] of TreeNode
        // …  information needed for freespace management
    end
```

Variations:

- using 3 arrays: Infos, Lefts, Rights
- over a dynamic vector