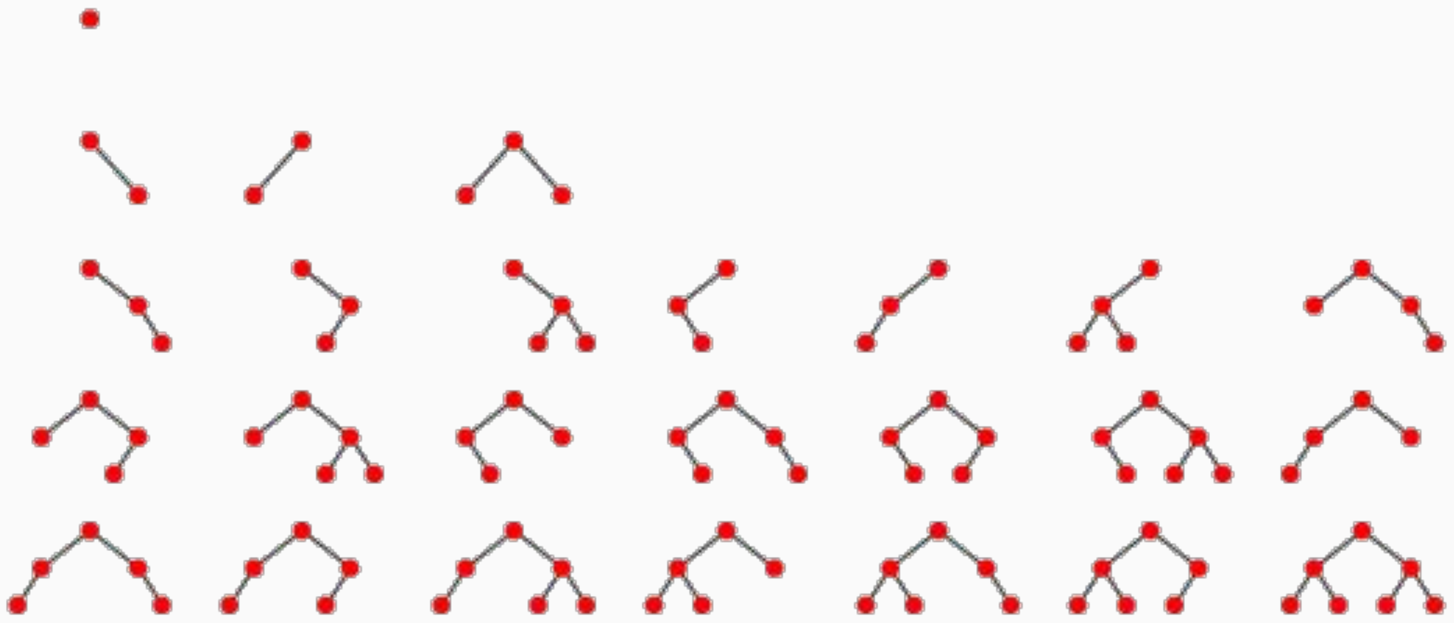
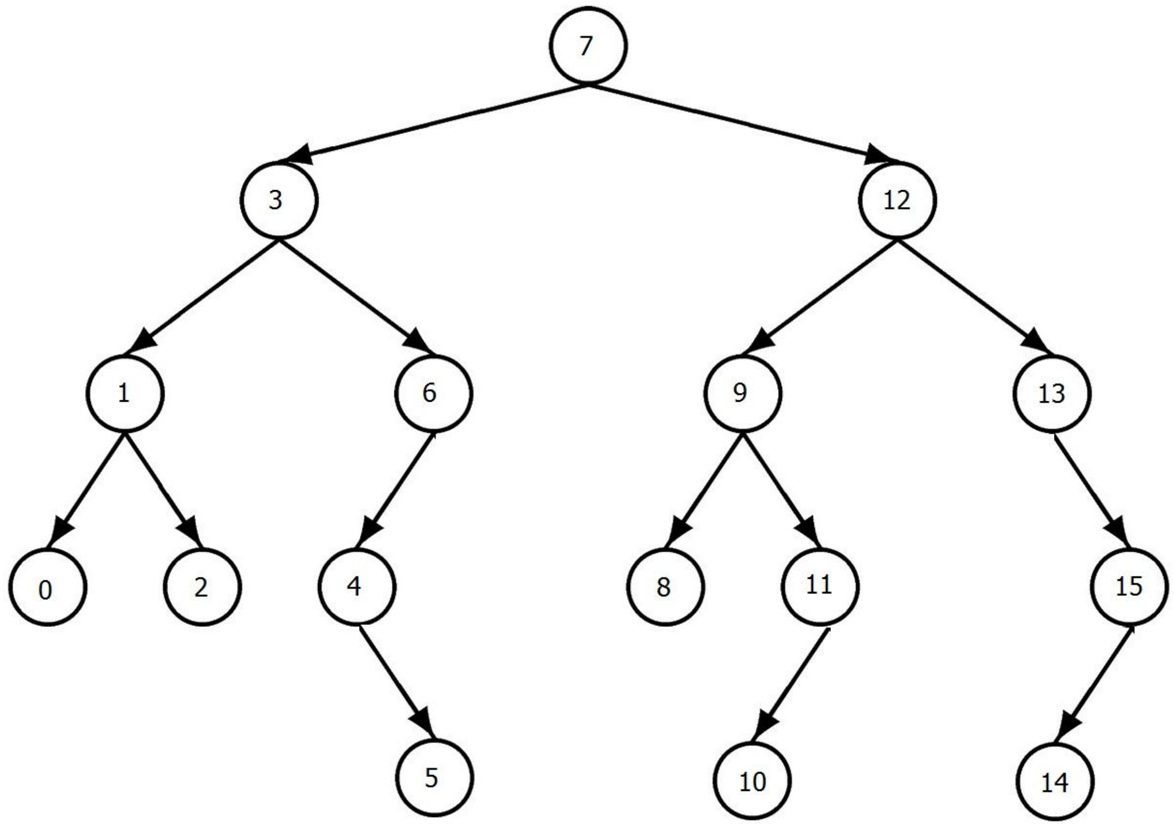
Binary Trees

Binary tree definition

- Nonlinear data structure
- Set of nodes (may be empty)
- Parts
    - Root (single node)
    - Left subtree (binary tree)
    - Right subtree (binary tree)
- Two disjoint binary trees
- Recursive
- Branch



Binary trees



Binary tree

Degree of a node

- Number of **nonempty subtrees** of a node
- Number of **branches** starting from a node
- 0, 1, or 2

Leaf / Terminal node

- Node with **degree 0**
- Node with **empty** left and right **subtrees**
- Node with **no branches** starting from it

Path

- Involves **two nodes** A and B
- **Sequence of branches** starting from A and ending at B
- Path from A to B is **unique**

Length of a path

- Number of **branches** in the **path** from A to B
- Number of nodes in path not including A
- Number of nodes in path not including B
- Number of nodes from A just before reaching B

Level of a node

- **Length** of the path **from the root** to the node
- Level of **root** node is **0**
- Levels of root nodes of its left and right subtrees are 1
- +1 for each subtree root

Height of a binary tree

- **Level** of **bottommost** nodes
- Largest level of its terminal nodes
- Maximum level of all its nodes

Relationships

- **Father** / mother
- **Sons** / daughters / **children**
- Brothers / sisters / **siblings**
- Descendants
- Ancestors

Operations

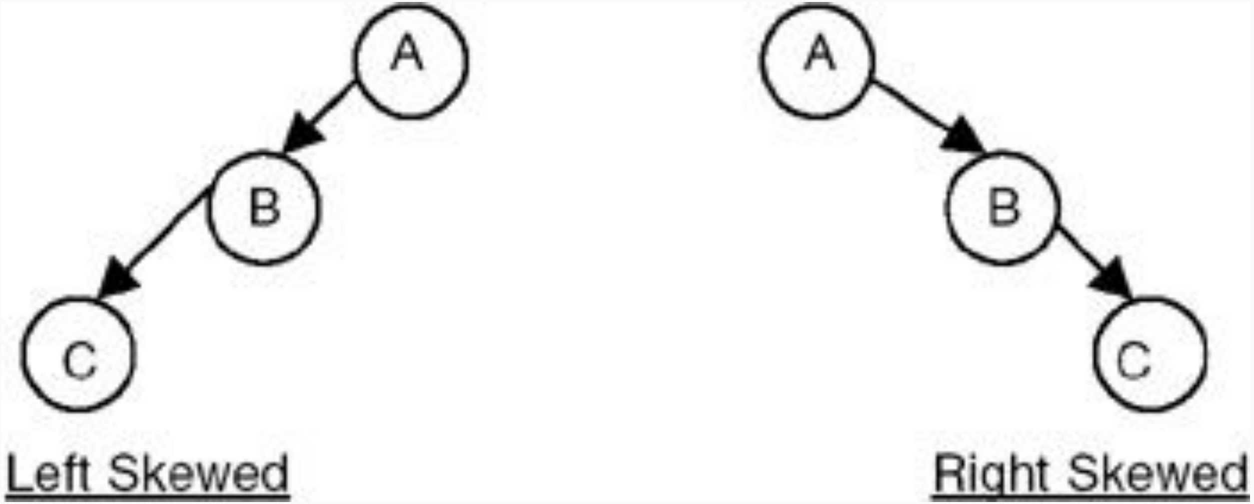
- Check if empty
- Make / grow
- Get number of nodes (size)
- Get height
- Copy
- Test for equivalence
- Traverse
- Find leftmost/rightmost node
- Insert node
- Delete node
- Replace subtree
- Get node with respect to some node
- Etc.

Properties

- Maximum nodes at level $\mathbf{i}$ is $\mathbf{2^{i}}$
- Maximum nodes in binary tree of height $\mathbf{h}$ in $\mathbf{2^{h+1} - 1}$
- Maximum height of binary tree with $\mathbf{n}$ nodes is $\mathbf{n - 1}$
- **Number of terminal nodes = 1 + number of nodes with degree 2**
- Number of distinct binary trees on n unlabeled nodes is
  - `((2n) C n) / (n + 1)`

Skewed binary tree

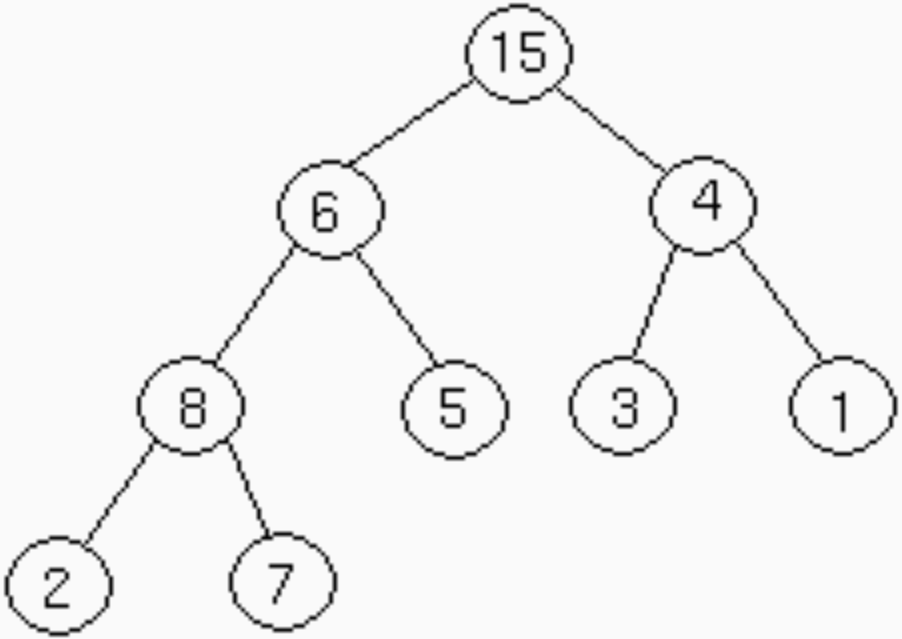
- Left-skewed or right-skewed
- Left-skewed
  - All **right** subtrees are empty
- Right-skewed
  - All **left** subtrees are empty
- Height is equal to number of nodes
- Straight line leaning to a side



Skewed binary trees

Strictly binary tree

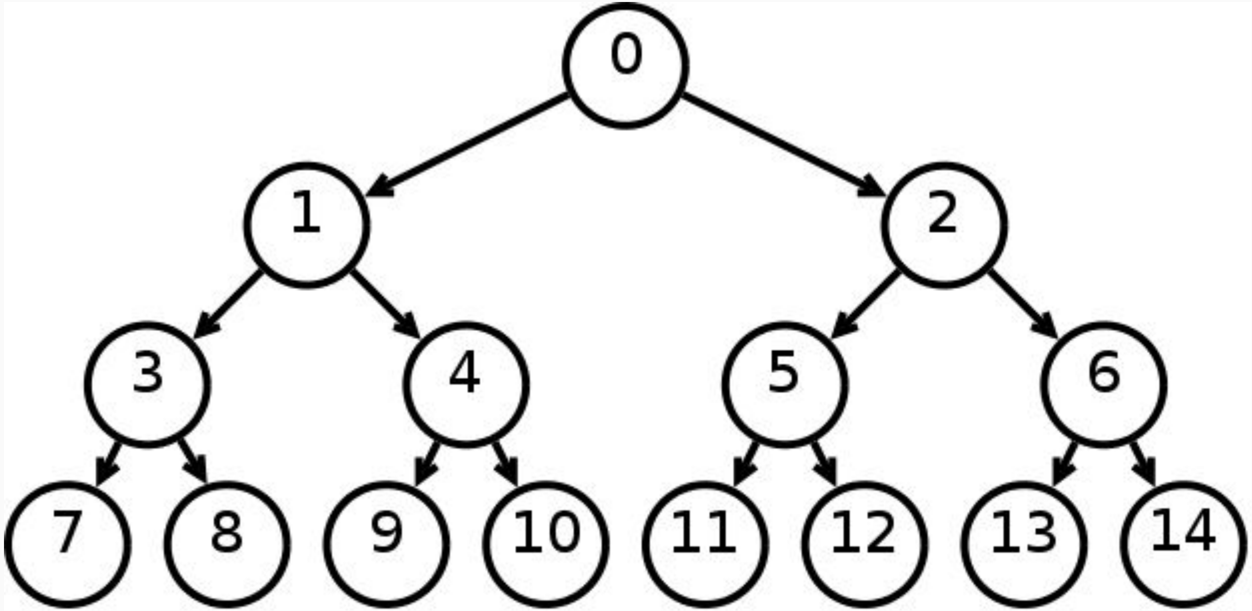
- All nodes are of **degree two** or **zero**
- All nodes have either **two** or **zero children**/non-empty subtrees
- Number of nodes is always **odd**



Strictly binary tree

Full binary tree

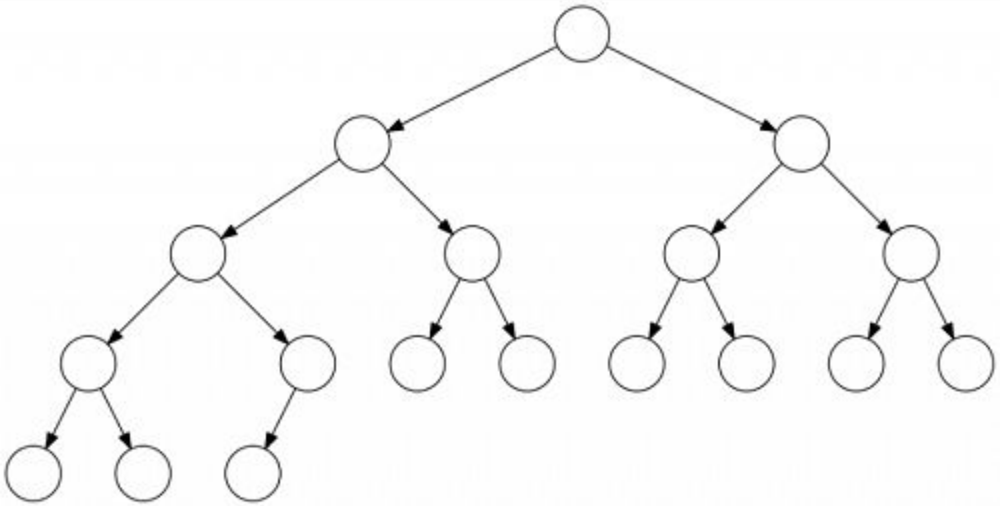
- **Strictly binary tree** (two or zero children)
    - All **terminal nodes** are at the **bottom**
    - **Level** of all terminal nodes are **equal**
- **Maximum number of nodes** for all levels
- No more space for more nodes without affecting the height
- Height is **log_2(n+1) - 1** or **floor(log_2(n))**

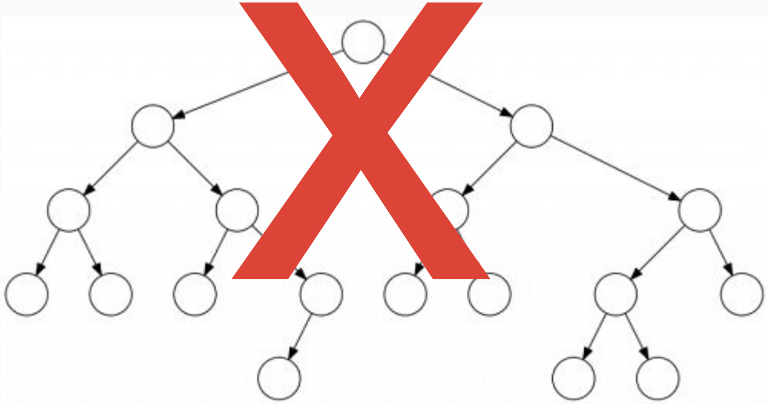


Full binary tree

Complete binary tree

- Every level except bottommost has maximum number of nodes
  - Bottommost level may have maximum number as well (not restricted)
  - Bottommost level must have only rightmost nodes removed if not full
- Full binary tree with rightmost nodes of bottommost level **possibly** removed
  - Implies that full binary trees are complete





Complete binary tree

Traversal

- Go through each node in some order
- Binary tree
  - **Root node**
  - **Left subtree**
  - **Right subtree**
- **LRN**, **LNR**, **RLN**, **RNL**, **NLR**, and **NRL**
- Which part should be processed first? Next? Last?

Left-before-right traversal

- LRN, LNR, NLR
- Preorder (start)
  - Root, left subtree, right subtree
- Inorder (middle)
  - Left subtree, root, right subtree
- Postorder (end)
  - Left subtree, right subtree, root

Preorder traversal

1. Visit the **root**
2. Traverse the **left subtree** in **preorder**
3. Traverse the **right subtree** in **preorder**

Inorder traversal

1. Traverse the **left subtree** in **inorder**
2. Visit the **root**
3. Traverse the **right subtree** in **inorder**

Postorder traversal

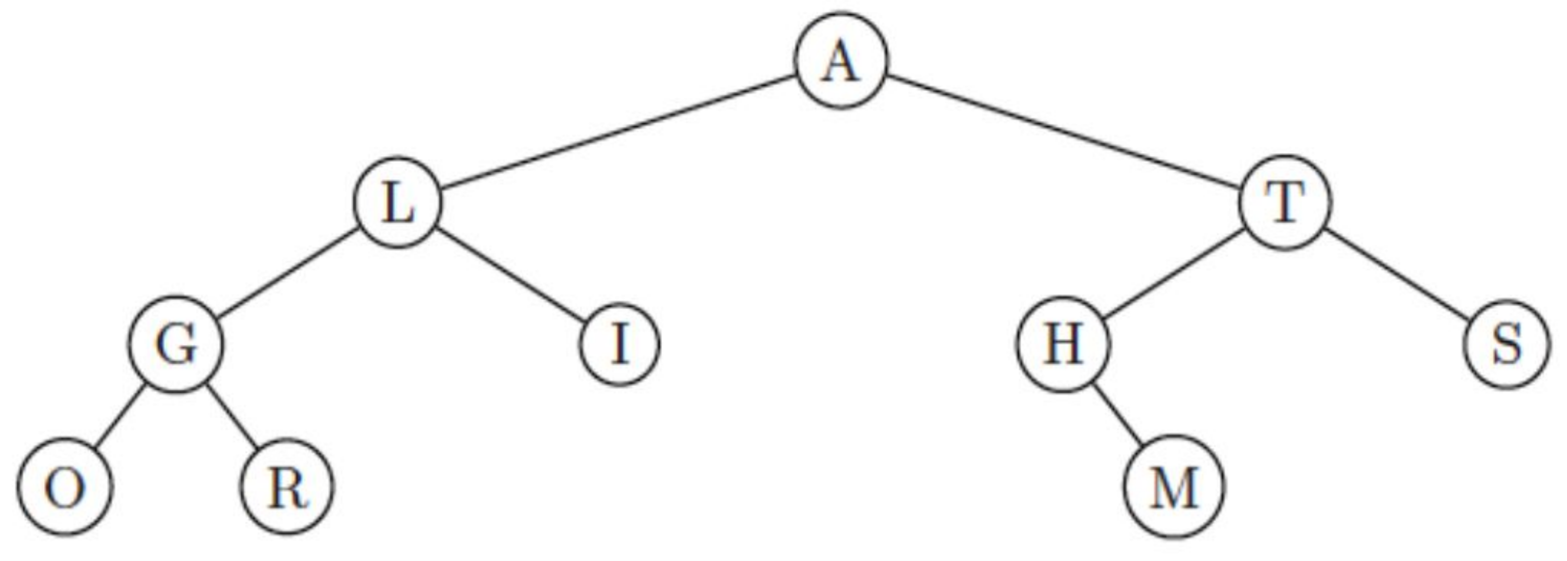
1. Traverse the **left subtree** in **postorder**
2. Traverse the **right subtree** in **postorder**
3. Visit the **root**

Right-before-left traversal

- Interchange order of traversing **left subtree** and **right subtree**
  - Converse preorder
  - Converse inorder
  - Converse postorder

Level order traversal

- Top level first, bottom level last (**top-to-bottom**)
  - Left nodes first, right nodes last (**left-to-right**)
- Reverse level order traversal
  - Bottom-to-top, right-to-left

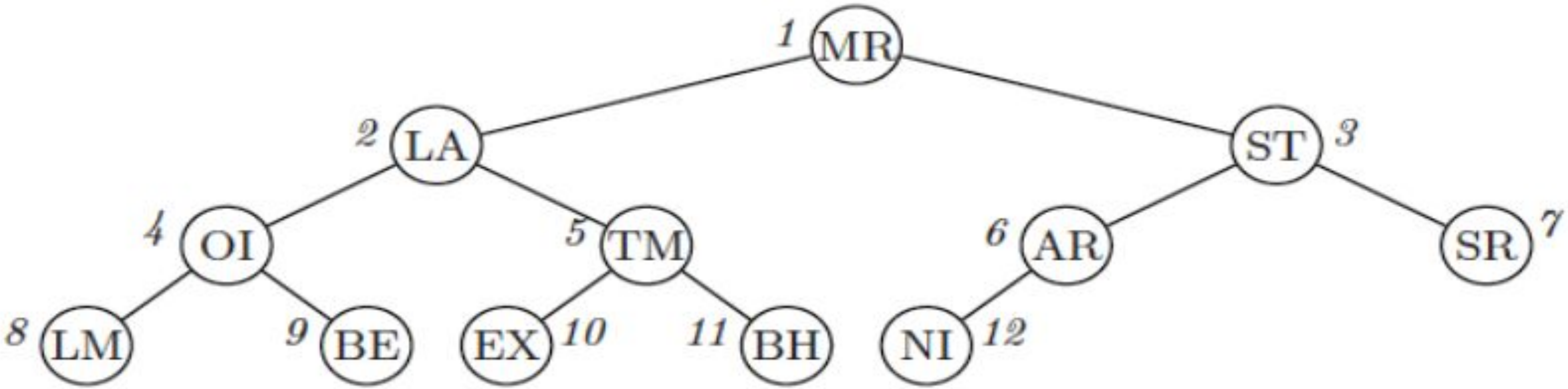


Binary tree

	Preorder	Inorder	Postorder
Preorder	NO	YES	NO
Inorder	YES	NO	YES
Postorder	NO	YES	NO

Representing a binary tree

- Sequential representation
  - Array
- Linked representation
  - Nodes

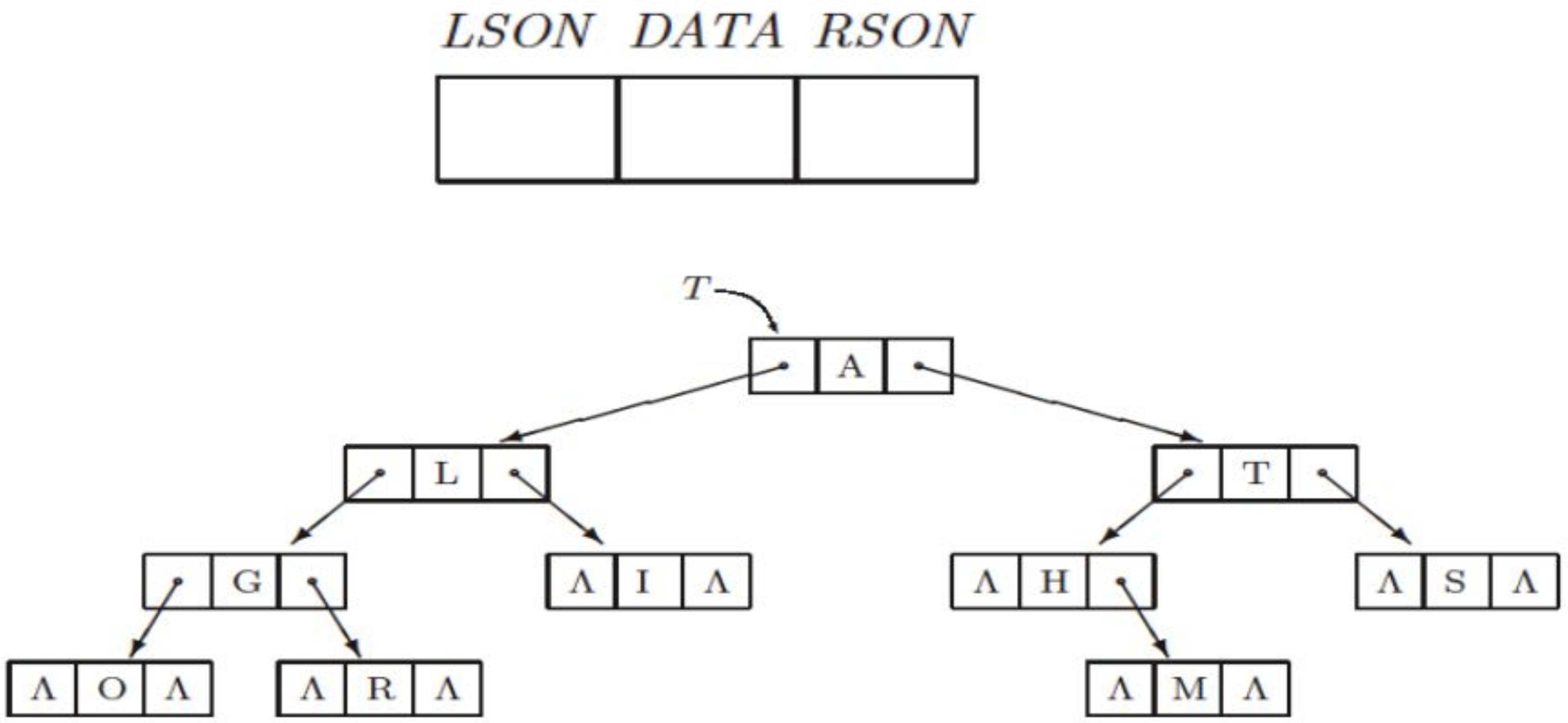


$i \Longrightarrow$	1	2	3	4	5	6	7	8	9	10	11	12	...	m
KEY:	MR	LA	ST	OI	TM	AR	SR	LM	BE	EX	BH	NI	···	

Sequential representation

Sequential representation

- n nodes
- **Array** of size `m >= n`
- **Node `i`** is in index `i`
- **Left son** of **node `i`** is in index `2 * i` if `2 * i <= n`
    - **Left sons** are even
- **Right son** of **node `i`** is in index `2 * i + 1` if `2 * i + 1 < n`
    - **Right sons** are odd
- **Father** of **node `i`** is `floor(i/2)` if `1 < i <= n`



Linked representation of a binary tree

Linked representation

- Pointer to root node
- Node
  - LSON
    - Left son
    - Pointer to root node of left subtree (may be NULL)
  - DATA
    - Actual information
  - RSON
    - Right son
    - Pointer to root node of right subtree (may be NULL)

Traversal implementation

- Recursive
    - Stack is used by the runtime system
- Iterative
    - Stack is managed manually by the programmer

```
procedure PREORDER(T)
if T ≠ Λ then [
        call VISIT(T)
        call PREORDER(LSON(T))
        call PREORDER(RSON(T))
]
end PREORDER
```

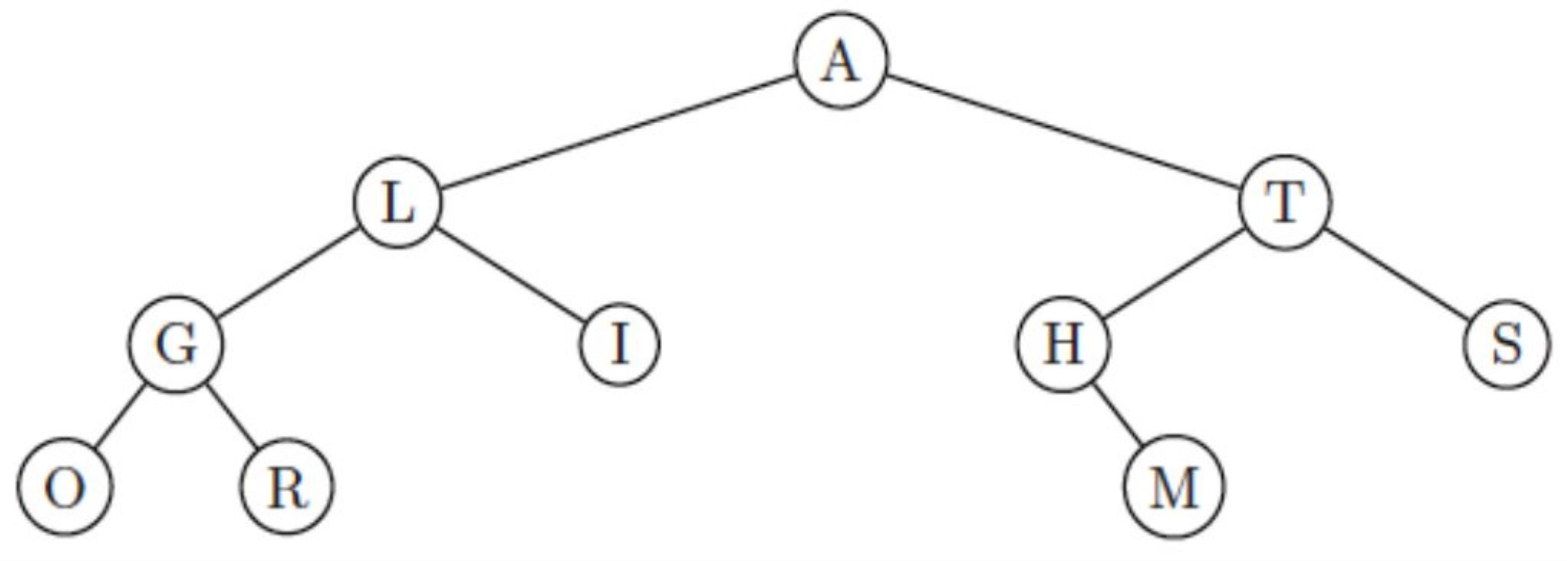
Recursive implementation of preorder traversal (linked implementation of a binary tree)

procedure INORDER(T)
if $T \neq \Lambda$ **then** [
    call INORDER(LSON(T))
    call VISIT(T)
    call INORDER(RSON(T))
]
end INORDER

Recursive implementation of inorder traversal (linked implementation of a binary tree)

```
procedure POSTORDER(T)
if T ≠ Λ then [
        call POSTORDER(LSON(T))
        call POSTORDER(RSON(T))
        call VISIT(T)
]
end POSTORDER
```

Recursive implementation of postorder traversal (linked implementation of a binary tree)



Binary tree

```
procedure PREORDER(T)
call InitStack(S)
α ← T
loop
    while α ≠ Λ do
        call VISIT(α)
        call PUSH(S, α)
        α ← LSON(α)
    endwhile
    if IsEmptyStack(S) then return
    else [ call POP(S, α); α ← RSON(α) ]
forever
end PREORDER
```

Iterative implementation of preorder traversal (linked implementation of a binary tree)

```
procedure INORDER(T)
call InitStack(S)
α ← T
loop
    while α ≠ Λ do
        call PUSH(S, α)
        α ← LSON(α)
    endwhile
    if IsEmptyStack(S) then return
    else [ call POP(S, α); call VISIT(α); α ← RSON(α) ]
forever
end INORDER
```

Iterative implementation of inorder traversal (linked implementation of a binary tree)

```
procedure POSTORDER(T)
call InitStack(S)
α ← T
go to 1
go to 2
end POSTORDER

1:  while α ≠ Λ do
        call PUSH(S, −α)
        α ← LSON(α)
    endwhile

2:  if IsEmptyStack(S) then return
    else [
        call POP(S, α)
        if α < 0 then [
            α ← −α
            call PUSH(S, α)
            α ← RSON(α)
            go to 1
        ] else [
            call VISIT(α)
            go to 2
        ]
    ]
```

Iterative implementation of postorder traversal (linked implementation of a binary tree)

Copy implementation

- Traverse **left subtree** of **node a** in **postorder** and copy
- Traverse **right subtree** of **node a** in **postorder** and copy
- Copy **node a** and attach copies of **left** and **right** subtrees to it

procedure COPY(T)

$\alpha \leftarrow \Lambda$

if $T \neq \Lambda$ **then** [

$\boldsymbol{\delta} \leftarrow$ COPY(LSON(T))

$\boldsymbol{\varepsilon} \leftarrow$ COPY(RSON(T))

call GETNODE(α)

DATA(α) $\leftarrow$ DATA(T)

LSON(α) $\leftarrow \boldsymbol{\delta}$

RSON(α) $\leftarrow \boldsymbol{\varepsilon}$

]

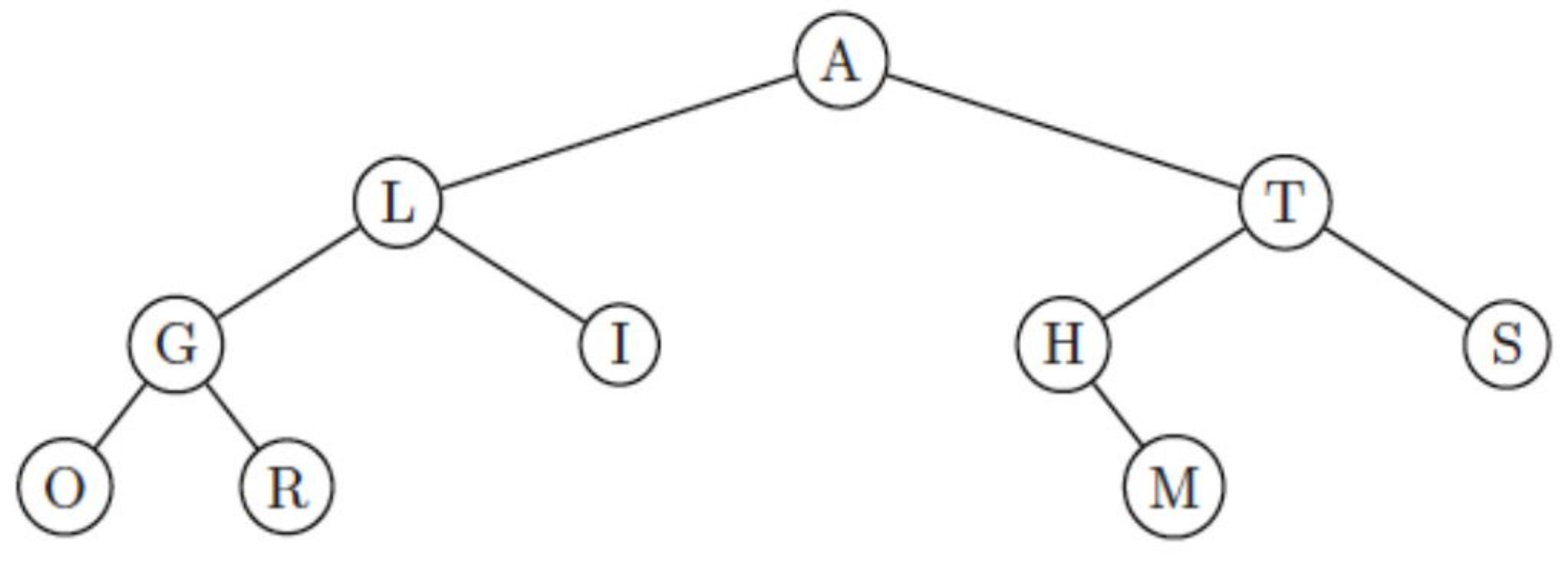
return(α)

end COPY

$\ldots$

$S \leftarrow$ COPY(T)

$\ldots$

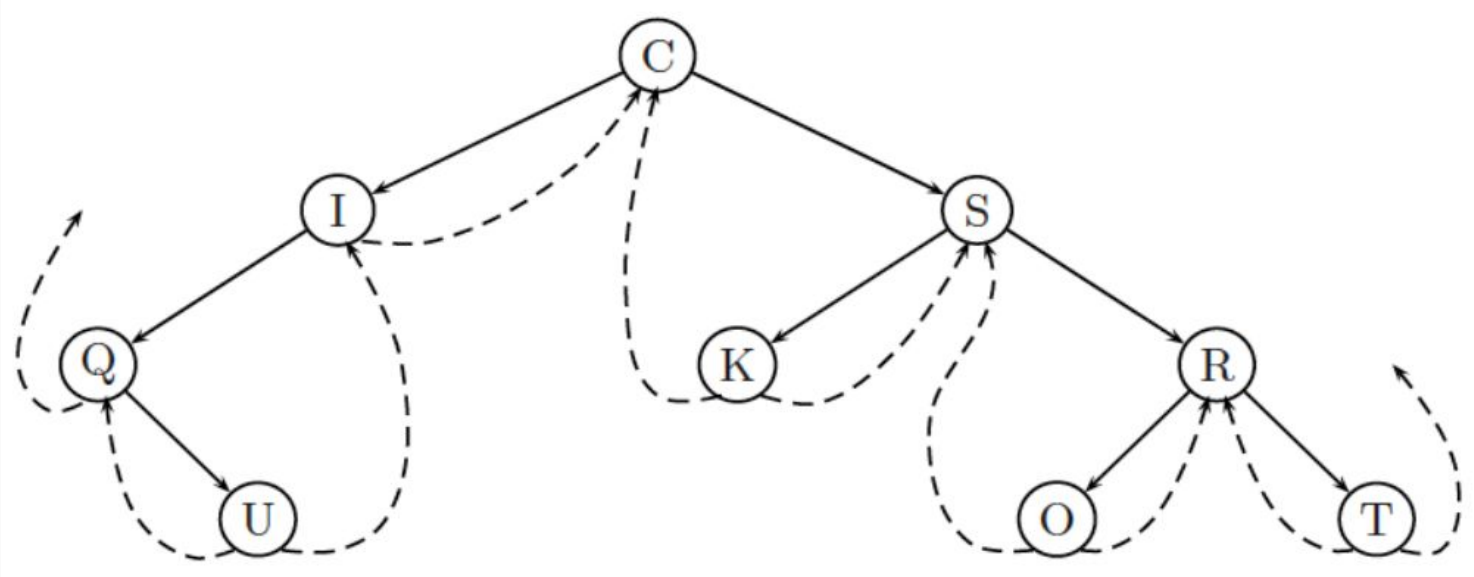


Binary tree

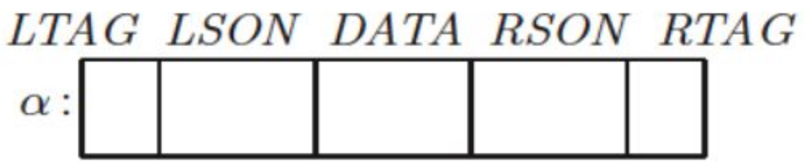
Equivalence implementation

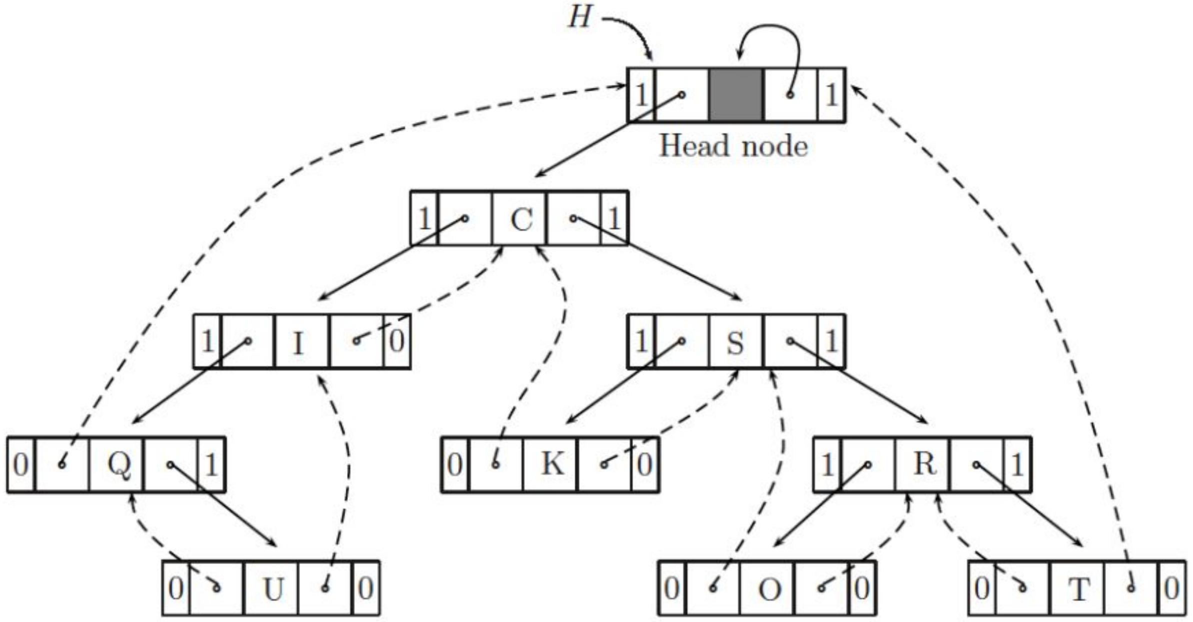
- Check whether **node a** and **node b** contain same data
- Traverse left subtree of **node a** and **node b** in **preorder** and check for equivalence
- Traverse right subtree of **node a** and **node b** in **preorder** and check for equivalence

```
procedure EQUIVALENT(S, T)
ans ← false
case
      : S = Λ and T = Λ : ans ← true
      : S ≠ Λ and T ≠ Λ : [
            ans ← (DATA(S) = DATA(T))
            if ans then ans ← EQUIVALENT(LSON(S), LSON(T))
            if ans then ans ← EQUIVALENT(RSON(S),
RSON(T))
      ]
endcase
return(ans)
end EQUIVALENT
```



Inorder threaded binary tree





Linked representation of an inorder threaded binary tree

Link inversion

Siklossy traversal

Heap

- Complete binary tree
- Total order (keys)
- Max-heap
  - Key of each node is greater than keys of both children
- Min-heap
  - Key of each node is less than keys of both children

Heap

- Subtrees are heaps
- Root of max-heap has largest key
- Root of max-heap has smallest key
- Not necessarily unique
- Height is floor(log_2(n))

Sift-up

- Binary tree to heap
- Smallest subtrees first
- Containing subtrees next
- Bottom to top, right to left
    - Reverse level order
- Terminal nodes are heaps
- floor(n / 2) trees in total

Almost-heap

- Subtrees are already heaps
- Root may not be largest node
- Sifting-up may cause subtrees to lose heap property
    - Reconvert subtrees into heaps

Heapify

- Make root current node
- Do until no more left child
  - Get larger child as current node
  - Exit if root is larger than current node
  - Place larger key in current node
- Place root key in current

Conversion

- Heapify floor(n/2) times
- Start from rightmost almost-heap at bottommost level
    - Continue in reverse level order

Sequential complete binary tree

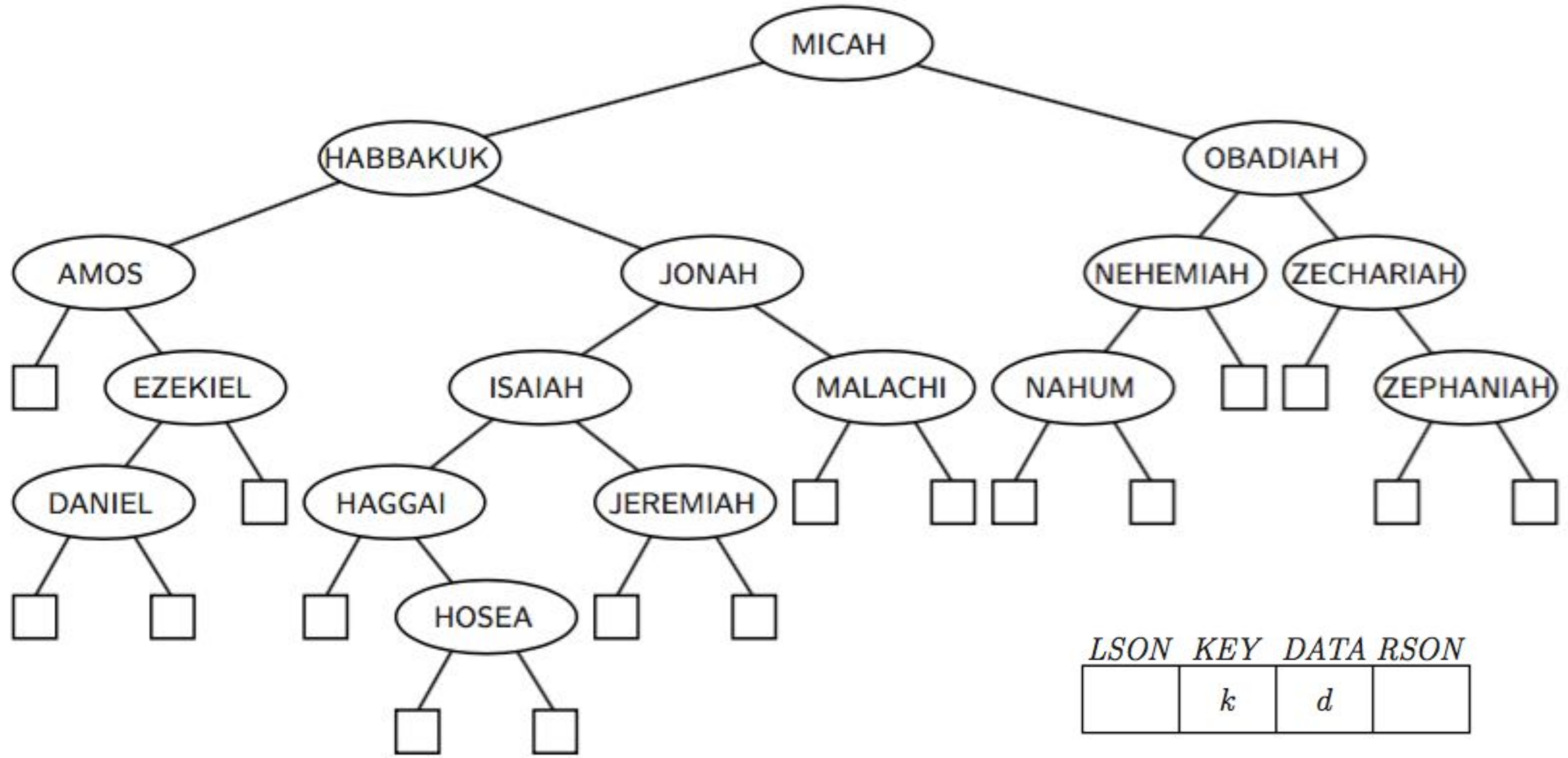
- Easy to traverse roots in reverse level order

Heapsort

- Place keys in complete binary tree [O(n)]
- Convert binary tree into heap via sift-up in reverse level order [O(n)]
- Do until heap is empty: [O(n log n)]
    - Pop root into queue
    - Make rightmost node in bottommost level the new root
    - Apply sift-up

Heapsort

- Can use same array for queue
- O(n log n)
- Priority queue



Binary search tree

```
procedure BST_SEARCH(𝔹, K)
▷ Given a BST 𝔹 and a search argument K, procedure searches for the node in 𝔹
▷ whose key matches K. If found, procedure returns the address of the node found;
▷ otherwise, it returns Λ.
   α ← T
   while α ≠ Λ do
      case
         :K = KEY(α): return(α)          ▷ successful search
         :K < KEY(α): α ← LSON(α)        ▷ go left
         :K > KEY(α): α ← RSON(α)        ▷ go right
      endcase
   endwhile
   return(α)      ▷ unsuccessful search
   end BST_SEARCH
```

LSON	KEY	DATA	RSON
	k	d	

Searching from a BST

procedure BST_MINKEY($\mathbb{B}$)

▷ *Given a BST $\mathbb{B}$, procedure searches for the node in $\mathbb{B}$ with the smallest*

▷ *key and returns the address of the node found.*

$\beta \leftarrow \alpha \leftarrow T$

while $\alpha \neq \Lambda$ **do**

  $\beta \leftarrow \alpha$

  $\alpha \leftarrow LSON(\alpha)$

endwhile

return(β)

end BST_MINKEY

LSON	*KEY*	*DATA*	*RSON*
	k	d	

Smallest key

```
procedure BST_INSERT(𝔹, k, d)
▷ Given a BST 𝔹 and a record (k,d), procedure inserts a new node into the BST
▷ to store the record. If there is already a record in 𝔹 with the same key, procedure
▷ issues an error message and terminates execution.
    α ← T
    while α ≠ Λ do
        case
            : k = KEY(α) : [output 'Duplicate key found.'; stop]
            : k < KEY(α) : [β ← α;  α ← LSON(α)]
            : k > KEY(α) : [β ← α;  α ← RSON(α)]
        endcase
    endwhile
▷ Exit from the loop means unsuccessful search; insert new node where unsuccessful
▷ search ended.
    call GETNODE(τ)
    KEY(τ) ← k; DATA(τ) ← d
    case
        :     T = Λ     : T ← τ
        : k < KEY(β) : LSON(β) ← τ
        : k > KEY(β) : RSON(β) ← τ
    endcase
    return
end BST_INSERT
```

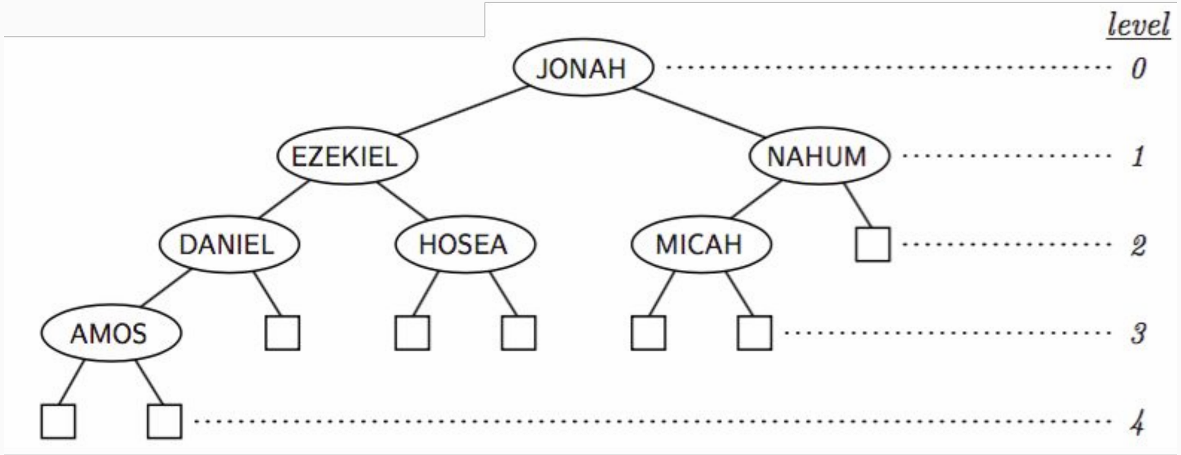
LSON	*KEY*	*DATA*	*RSON*
	k	d	

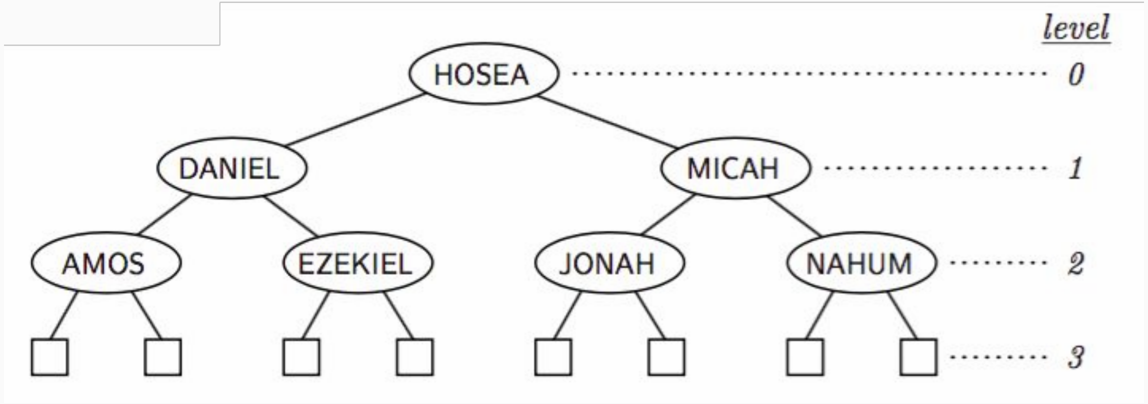
Inserting into a BST

```
procedure BST_DELETE(𝔹, α)
▷ Deletes node α from a BST where α = {T | LSON(β) | RSON(β)}, i.e., node α is the
▷ root of the entire BST or is either the left or the right son of some node β in the BST.
    τ ← α
    case
      :      α = Λ       : return
      : LSON(α) = Λ : α ← RSON(α)
      : RSON(α) = Λ : α ← LSON(α)
      : RSON(α) ≠ Λ : [γ ← RSON(α)
                        σ ← LSON(γ)
                        if σ = Λ then [LSON(γ) ← LSON(α);  α ← γ]
                                 else [while LSON(σ) ≠ Λ do
                                           γ ← σ
                                           σ ← LSON(σ)
                                       endwhile
                                       LSON(γ) ← RSON(σ)
                                       LSON(σ) ← LSON(α)
                                       RSON(σ) ← RSON(α)
                                       α ← σ] ]
    endcase
    call RETNODE(τ)
    return
    end BST_DELETE
```

$LSON$	KEY	$DATA$	$RSON$
	k	d	

Deleting from a BST


level
JONAH
0
EZEKIEL
NAHUM
1
DANIEL
HOSEA
MICAH
2
AMOS
3
4


level
HOSEA
0
DANIEL
MICAH
1
AMOS
EZEKIEL
JONAH
NAHUM
2
3

Trees

Types (species) of trees

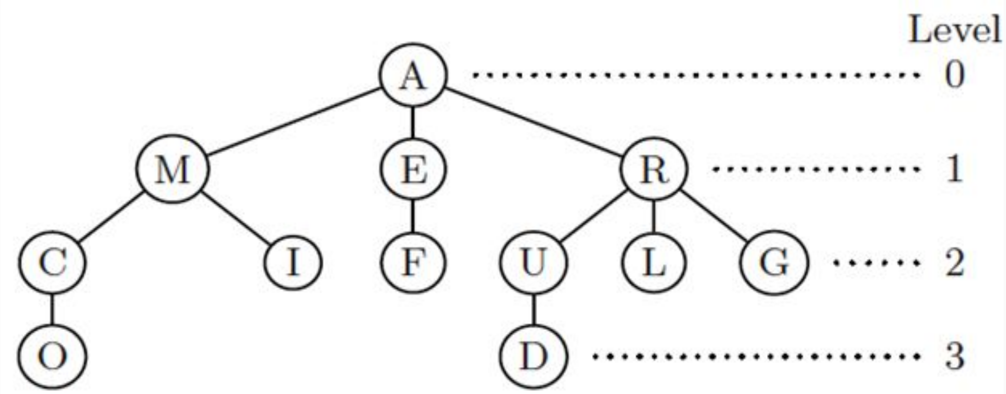
- Ordered
- Oriented
- Free

Ordered tree (tree)

- Set of nodes
  - At least one
  - Cannot be infinite
- Root node
- Subtrees
  - Zero or more
  - Partition of other nodes (disjoint)
  - Ordered trees
  - Order of subtrees is relevant

Properties

- **Degree (tree)**
  - Max node degree
- **Degree (node)**
  - Number of subtrees
- **Level (node)**
  - Branches from root
- **Height (tree)**
  - Max node level
- Terminal nodes / leaves

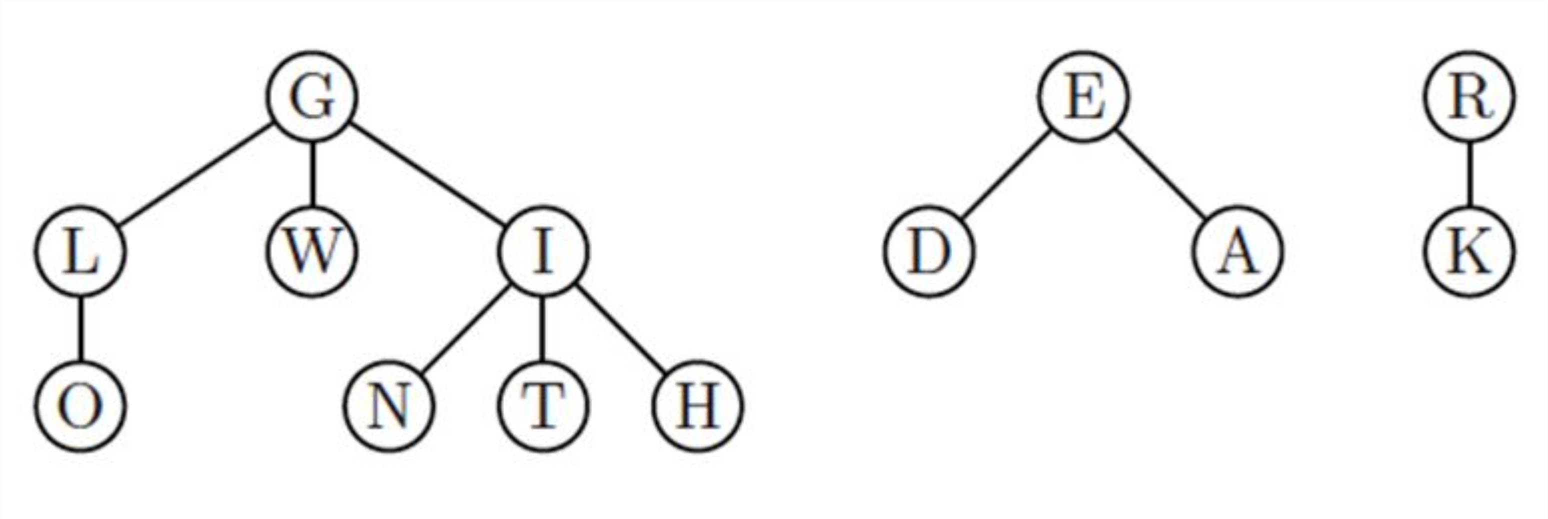


Relationships

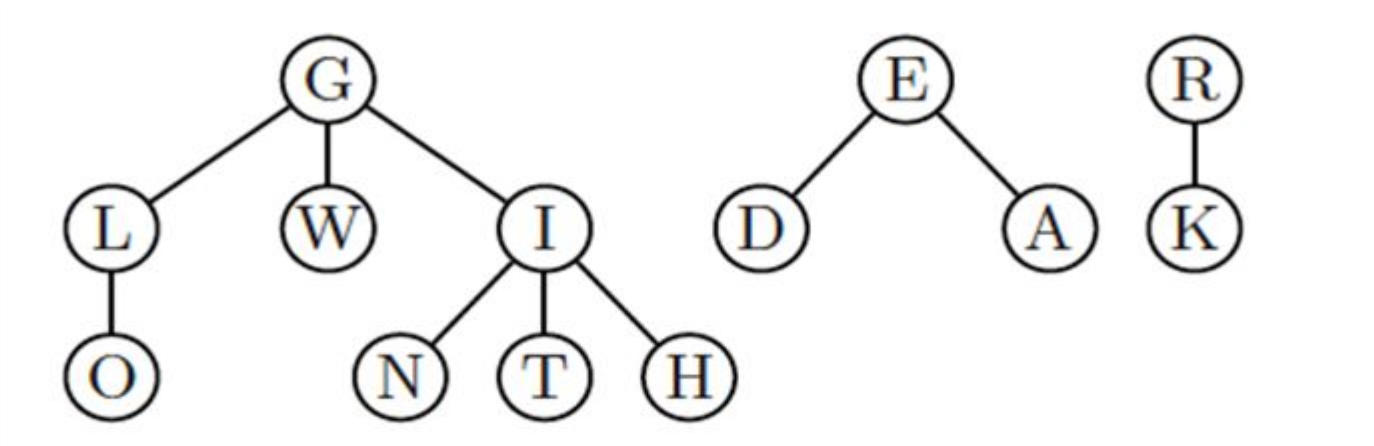
- Father
- Sons
- Brothers
- Age
  - Oldest is leftmost
  - Youngest is rightmost

Forest

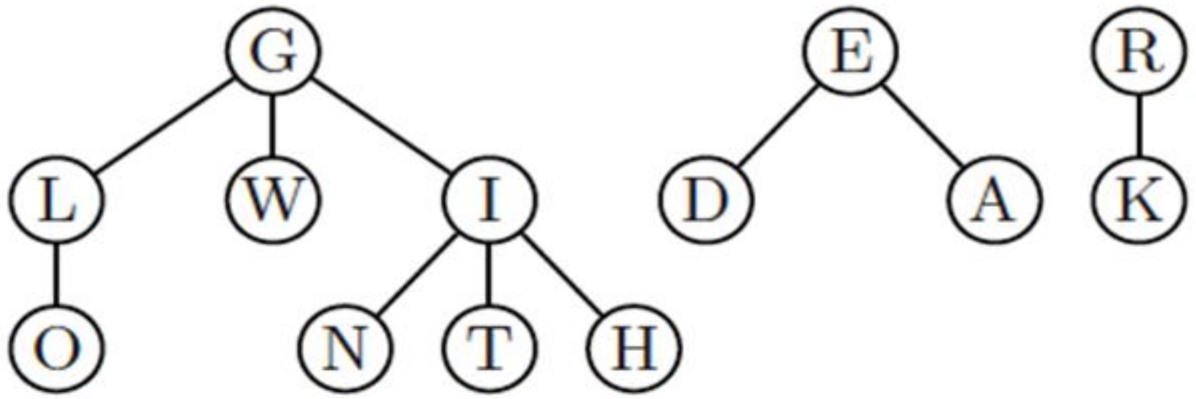
- Set of disjoint trees
- May be empty
- Single tree = single forest
- Assume trees are ordered



Forest with three trees



Forest Preorder	G L O W I N T H E D A R K
Forest Inorder	O L W N T H I G D A E K R
Forest Postorder	O H T N I W L A D K R E G



Preorder: G L O W I N T H E D A R K

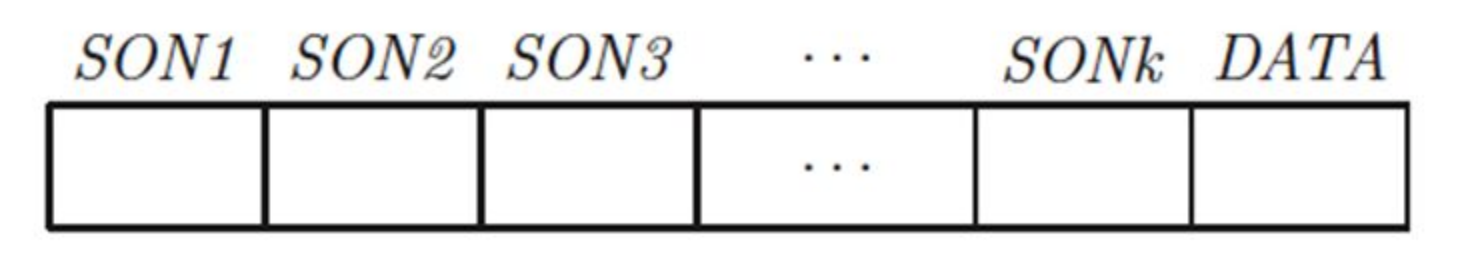
Postorder: O L W N T H I G D A E K R

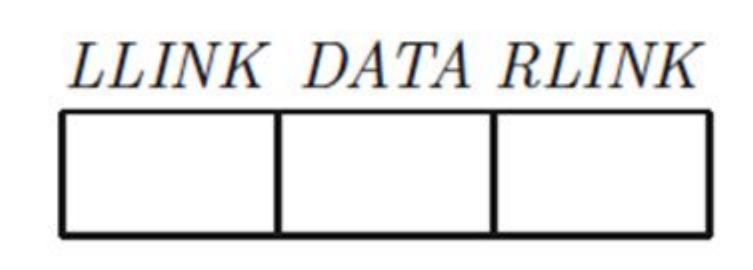
Preorder and postorder traversal on forests

Representation

- Linked
- Sequential

Linked representation





Sequential representation

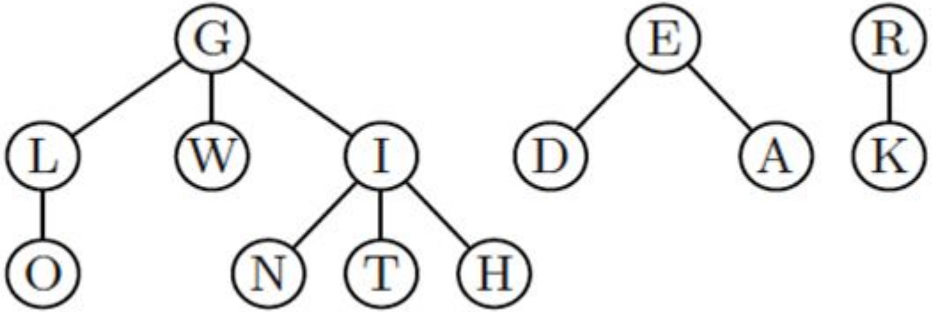
- Use link and tag arrays
  - Link to child
  - Tag for existence
- Arranged based on traversal order
- Types
  - Preorder sequential
  - Family-order sequential
  - Level-order sequential

Preorder

- Order
  - Node
  - All descendants of node
    - Oldest family first
  - Next sibling of node
- One tree at a time

Preorder sequential representation (1)

- RLINK, DATA, LTAG
  - Arrays with **n** elements
- RLINK
  - Index of next younger sibling
- LTAG
  - Boolean
  - Has children?

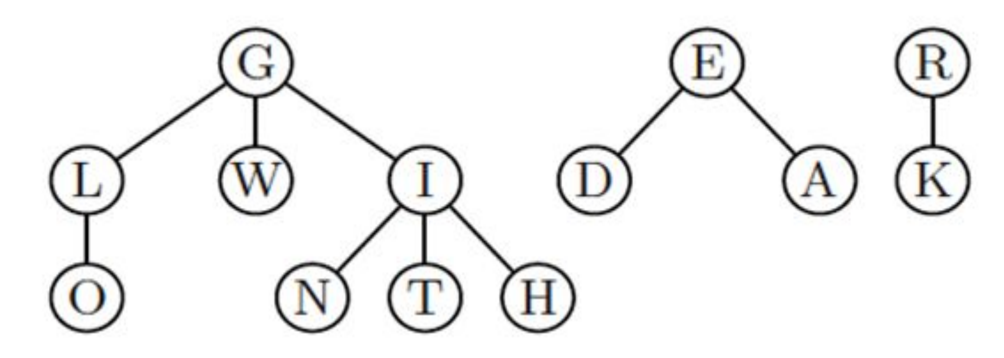


	1	2	3	4	5	6	7	8	9	10	11	12	13
RLINK:	9	4	0	5	0	7	8	0	12	11	0	0	0
DATA:	G	L	O	W	I	N	T	H	E	D	A	R	K
LTAG:	1	1	0	0	1	0	0	0	1	0	0	1	0

Preorder sequential representation (approach 1)

Preorder sequential representation (2)

- RTAG, DATA, LTAG
  - Arrays with **n** elements
- RTAG
  - Boolean
  - Has next sibling?
- LTAG
  - Boolean
  - Has children?



	1	2	3	4	5	6	7	8	9	10	11	12	13
RTAG:	1	1	0	1	0	1	1	0	1	1	0	0	0
DATA:	G	L	O	W	I	N	T	H	E	D	A	R	K
LTAG:	1	1	0	0	1	0	0	0	1	0	0	1	0

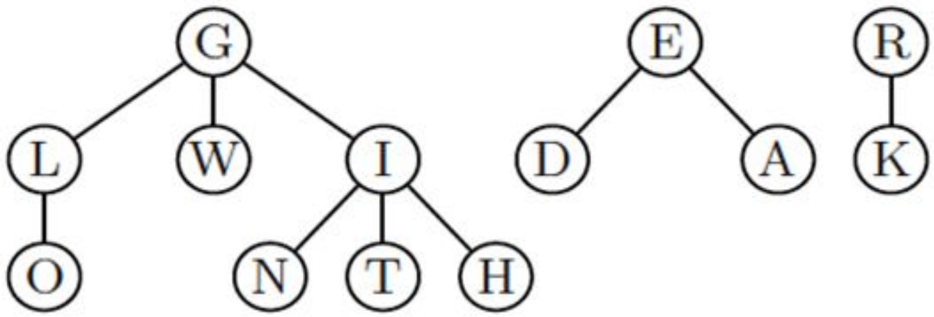
Preorder sequential representation (approach 2)

Family-order

- Order
  - Top-level siblings
  - Family-order of last sibling
  - …
  - Family-order of first sibling

Family-order sequential representation (1)

- LLINK, DATA, RTAG
  - Arrays with **n** elements
- LLINK
  - Index of oldest child
- RTAG
  - Boolean
  - Has next sibling?

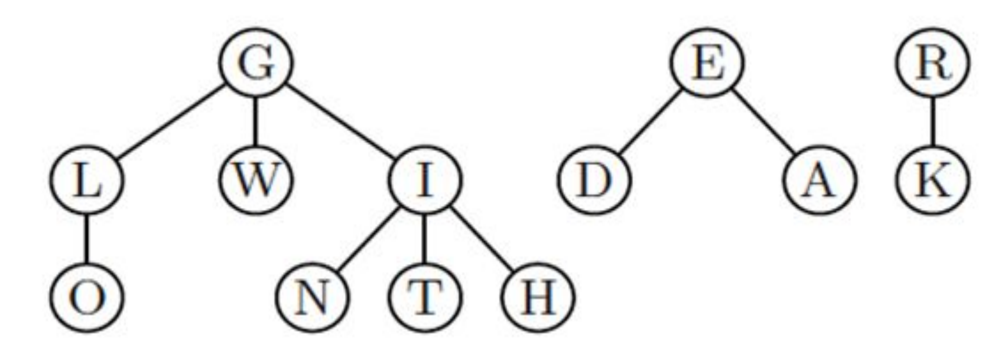


	1	2	3	4	5	6	7	8	9	10	11	12	13
LLINK:	7	5	4	0	0	0	13	0	10	0	0	0	0
DATA:	G	E	R	K	D	A	L	W	I	N	T	H	O
RTAG:	1	1	0	0	1	0	1	1	0	1	1	0	0

Family-order sequential representation (approach 1)

Family-order sequential representation (2)

- LTAG, DATA, RTAG
  - Arrays with **n** elements
- LTAG
  - Boolean
  - Has children?
- RTAG
  - Boolean
  - Has next sibling?



	1	2	3	4	5	6	7	8	9	10	11	12	13
LTAG:	1	1	1	0	0	0	1	0	1	0	0	0	0
DATA:	G	E	R	K	D	A	L	W	I	N	T	H	O
RTAG:	1	1	0	0	1	0	1	1	0	1	1	0	0

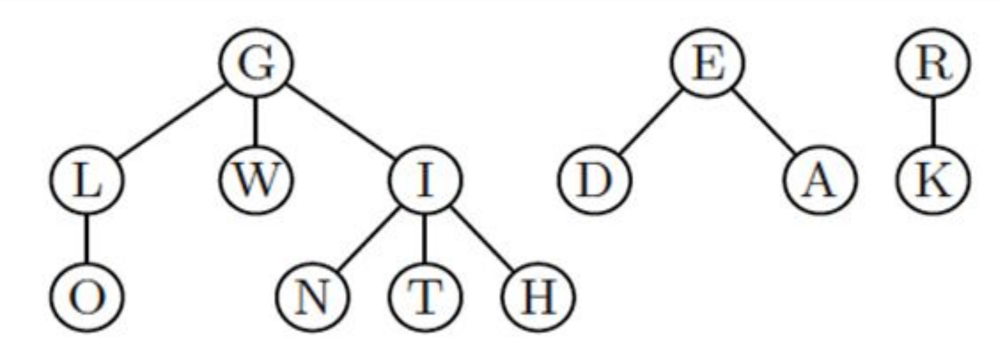
Family-order sequential representation (approach 2)

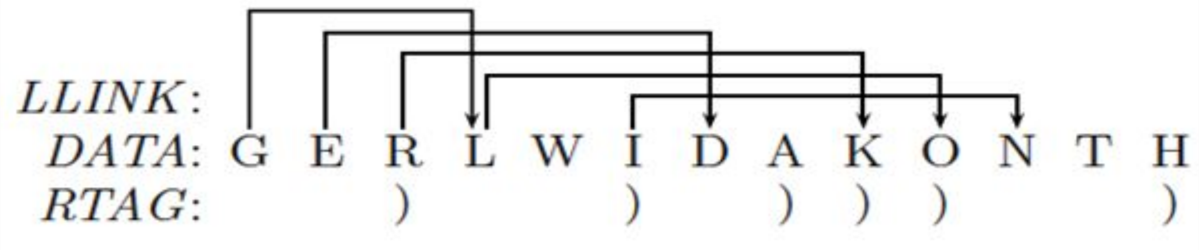
Level-order

- Order
  - Siblings of first level (oldest to youngest)
  - Siblings of second level (oldest to youngest)
  - ...
  - Siblings of last level (oldest to youngest)

Level-order sequential representation (1)

- LLINK, DATA, RTAG
  - Arrays with **n** elements
- LLINK
  - Index of oldest child
- RTAG
  - Boolean
  - Has next sibling
    -) means no

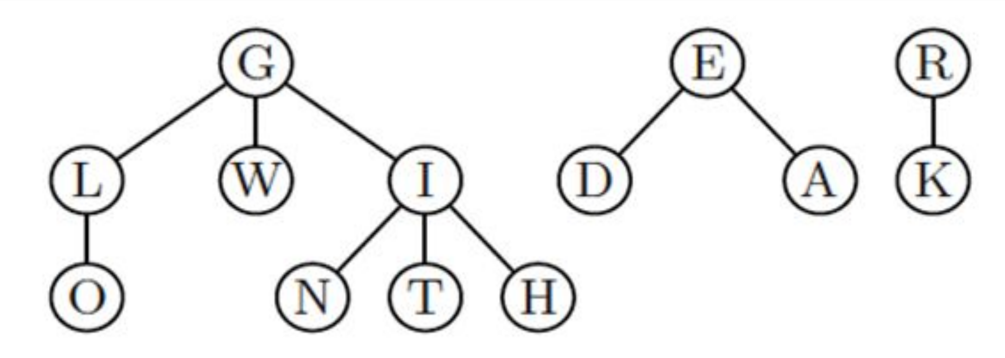




Level-order sequential representation (1)

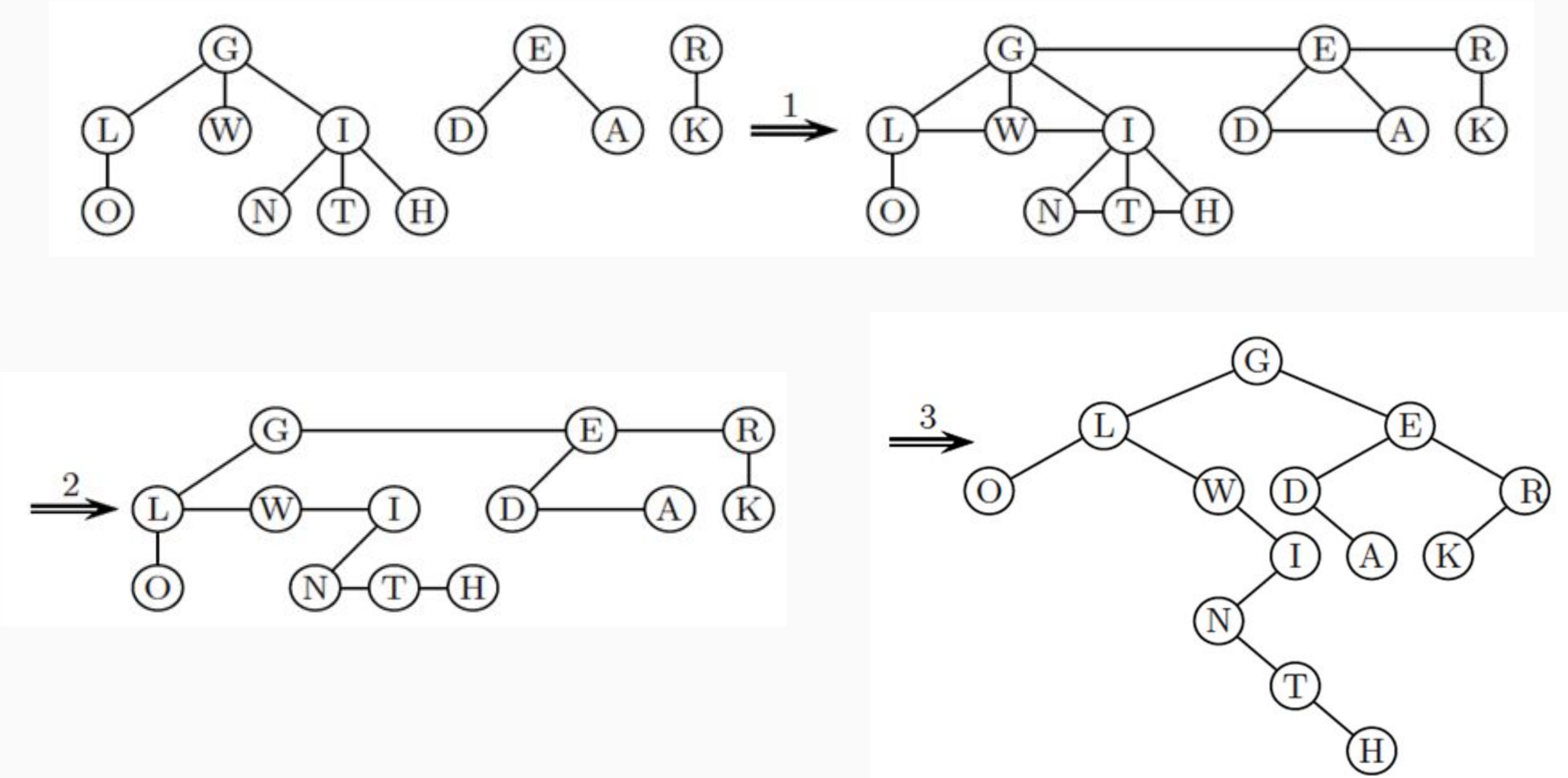
Level-order sequential representation (2)

- LTAG, DATA, RTAG
  - Arrays with **n** elements
- LTAG
  - Boolean
  - Has children?
    - (means yes
- RTAG
  - Boolean
  - Has next sibling?
    -) means no



LTAG:	(	(	(	(		(							
DATA:	G	E	R	L	W	I	D	A	K	O	N	T	H
RTAG:			)			)		)	)	)			)

Level-order sequential representation (2)



Natural correspondence: turning trees/forests into binary trees

Equivalence relation

- Relation (≡)
  - Reflexive
    - x ≡ x
  - Symmetric
    - If x ≡ y, then y ≡ x
  - Transitive
    - If x ≡ y and y ≡ z, then x ≡ z
- Partitions a set into **equivalence classes**
  - All elements in an equivalence class are equivalent

Equivalence problem

- Given a set of equivalence relations under elements in **S**, determine for two elements **x** and **y** in **S** if they are equivalent.

Union-Find algorithm

- Use forest of oriented trees
  - Each tree in the forest is an equivalence class
  - Root of each tree is a **father**
  - Disjoint-set abstract data type
- Checking for equivalence
  - **Equivalent** if two nodes share the **same root** (part of same tree)
  - Not equivalent otherwise

Union-Find algorithm

- Problem
  - Start with forest of **n** trees (one tree for each element)
  - Building the forest from the given equivalence relations
- Two operations
  - Union
    - Combine equivalent trees
    - Make **root** of one tree the **father** of the other
  - Find
    - Get the **father** of a node

Union-Find algorithm

- Representation
  - Array of size **n** (**father**)
  - Array index corresponds to node label
  - Value is the index of another node
  - Initialized to 0 (not pointing anywhere)
    - 0 means the node is a root

Union (initial)

- Input is A and B
- Get root of A as AA
    - Traverse father array starting from A with AA until 0
- Get root of B as BB
    - Traverse father array starting from B with BB until 0
- If roots are not equal
    - Make BB the father of AA
- If roots are equal
    - Do nothing

Find (initial)

- Input is A and B
- Get root of A as AA
  - Traverse father array starting from A with AA until 0
- Get root of B as BB
  - Traverse father array starting from B with BB until 0
- If roots are not equal
  - A and B are not equivalent
- If roots are equal
  - A and B are equivalent

Union improvement

- Performing union for each line of input (say n as well)
  - Single linear tree (height is n - 1)
  - O(n^2)
  - Second root is always new father
- Better solution
  - Choose root with greater number of descendants as father
  - Weighting rule
    - Negative values in **father** mean **number of descendants**
    - **father** is now initialized with -1
    - Max height is floor(log_2(n))

procedure UNION(i, j)

$count \leftarrow FATHER(i) + FATHER(j)$

if $|FATHER(i)| > |FATHER(j)|$ **then** [$FATHER(j) \leftarrow i$; $FATHER(i) \leftarrow count$]

else [$FATHER(i) \leftarrow j$; $FATHER(j) \leftarrow count$]

end UNION

Union in EASY

Find improvement

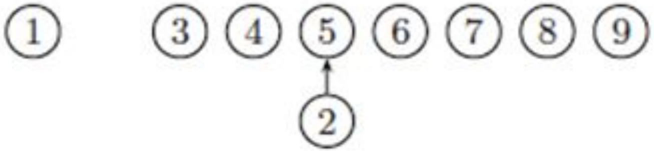
- Finding roots requiring climbing up the entire tree
    - More efficient to shorten tree as we climb so future calls are faster
    - Make the root of all nodes in path to root equal to root
    - Path compression
- Collapsing rule
    - Find and store the root
    - Climb up the tree again
        - Assign **father** of each node traversed (except root) to stored root

```
procedure FIND(i)
Find root
  r ← i
  while FATHER(r) > 0 do
    r ← FATHER(r)
  endwhile
Compress path from node i to the root r
  j ← i
  while j ≠ r do
    k ← FATHER(j)
    FATHER(j) ← r
    j ← k
  endwhile
  return(r)
  end FIND
```

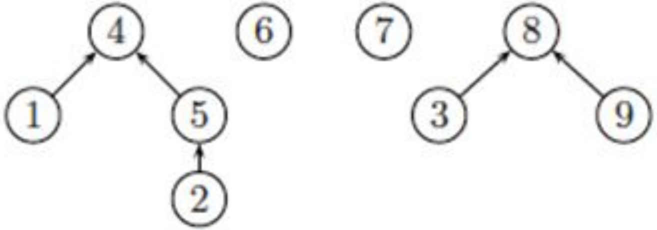
Find in EASY

Complexity after improvement

- Union
  - O(1)
- Find
  - O(n^2) alone
  - Practically linear with path compression

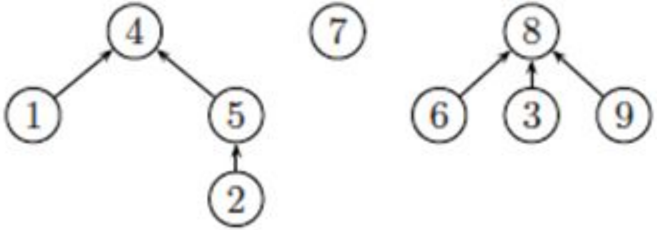
	Forest of oriented trees	The *FATHER* array
		$\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{array}$
$2 \equiv 5$		$\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline -1 & 5 & -1 & -1 & -2 & -1 & -1 & -1 & -1 \end{array}$
$3 \equiv 8$ $1 \equiv 4$		$\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 4 & 5 & 8 & -2 & -2 & -1 & -1 & -2 & -1 \end{array}$
$3 \equiv 9$		$\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 4 & 5 & 8 & -2 & -2 & -1 & -1 & -3 & 8 \end{array}$

$2 \equiv 1$



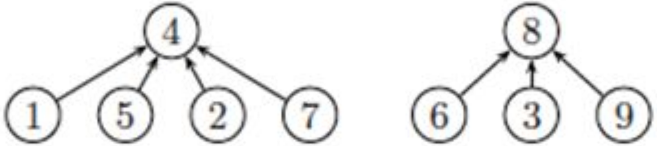
1	2	3	4	5	6	7	8	9
4	5	8	−4	4	−1	−1	−3	8

$3 \equiv 6$



1	2	3	4	5	6	7	8	9
4	5	8	−4	4	8	−1	−4	8

$2 \equiv 7$

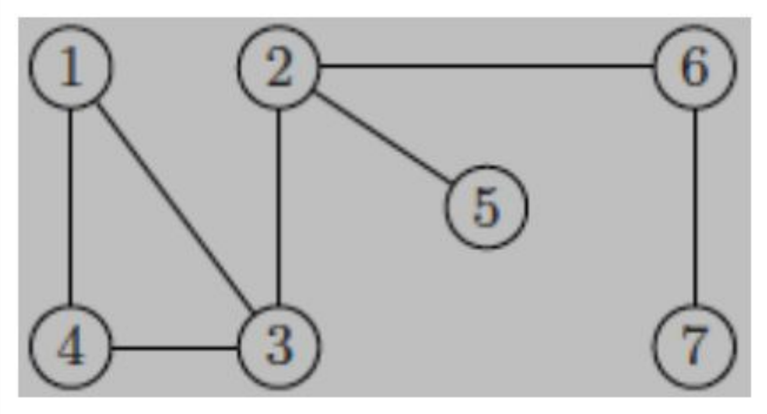


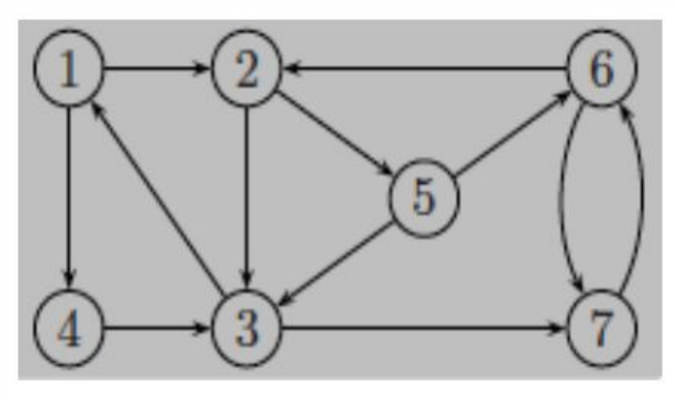
1	2	3	4	5	6	7	8	9
4	4	8	−5	4	8	4	−4	8

Graphs

Graphs

- Set of vertices (V)
  - Nonempty
  - Finite
- Set of edges (E)
  - Pair of vertices
  - Can be empty
  - Finite
- G = (V, E)


1
2
6
5
4
3
7


1
2
6
5
4
3
7

Subgraph

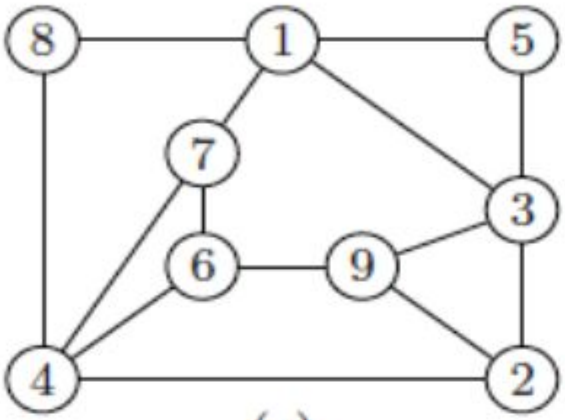
- Given a graph G
- Contains subset of:
  - Set of vertices of G
  - Set of edges of G

Graph types

- Undirected
- Directed

Undirected graph

- Edges are unordered
  - (a, b) and (b, a) are **the same**
- No self-loops
  - (a, a)
- Terms for edge (a, b)
  - Edge (a, b) is **incident on** vertices **a** and **b**
  - Vertices **a** and **b** are **adjacent** to each other

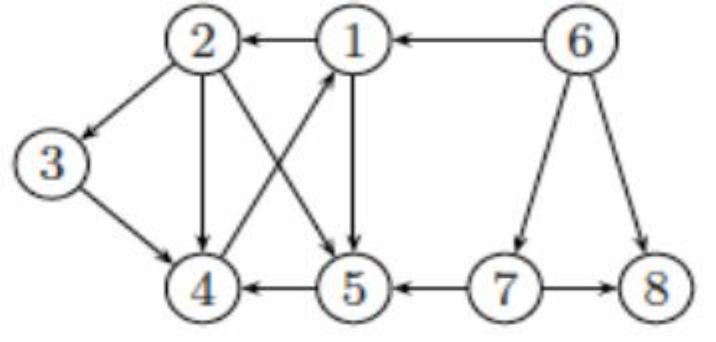


(a)

Undirected graph

Directed graph

- Digraph
- Edges are ordered
    - (a, b) and (b, a) are **distinct**
- Self-loops are allowed
    - Assume there are none for now (**simple** digraph)
- Terms for edge (a, b)
    - Edge (a, b) is **incident from** or **leaves** vertex **a**
    - Edge (a, b) is **incident to** or **enters** vertex **b**
    - Vertices **a** and **b** are **adjacent** to each other



Directed graph

Properties

- Maximum number of edges (undirected graph)
  - n (n - 1) / 2
  - n - 1 edges at most can be incident on a vertex
  - Edges are distinct
  - Complete graph
- Edges vs. vertices
  - Sparce
    - |E| << |V|^2
  - Dense
    - |E| ≅ |V|^2

Properties

- Degree (undirected)
  - Number of edges **incident on** a node
- Degree (directed)
  - Out-degree
    - Number of edges **incident from** a node
  - In-degree
    - Number of edges **incident to** a node
- Total degree
  - 2 |E|
  - |E| = sum of in-degree of all nodes = sum of out-degree of all nodes

Properties

- Path
  - Sequence of vertices
  - Start vertex
  - End vertex
  - Each adjacent pair of vertices in sequence has edge
- Path length
  - Number of edges in path
- Simple path
  - All vertices included (start and end are optional)

Properties

- Reachable
  - There is a path from vertex a to vertex b
- Cycle
  - Path with same start and end vertices
- Acyclic
  - No cycles

Properties

- Equivalence classes
  - Vertices in each pair are reachable from each other
  - Connected component (CC)
- Connected graph
  - SIngle CC
- Articulation point
  - Vertex in connected graph
  - Removing this disconnects the graph

Connected components (directed)

- Equivalence classes
  - Vertices in each pair are reachable from each other
  - Strongly connected component (SCC)
- Strongly connected graph
  - SIngle SCC

Connected components (undirected)

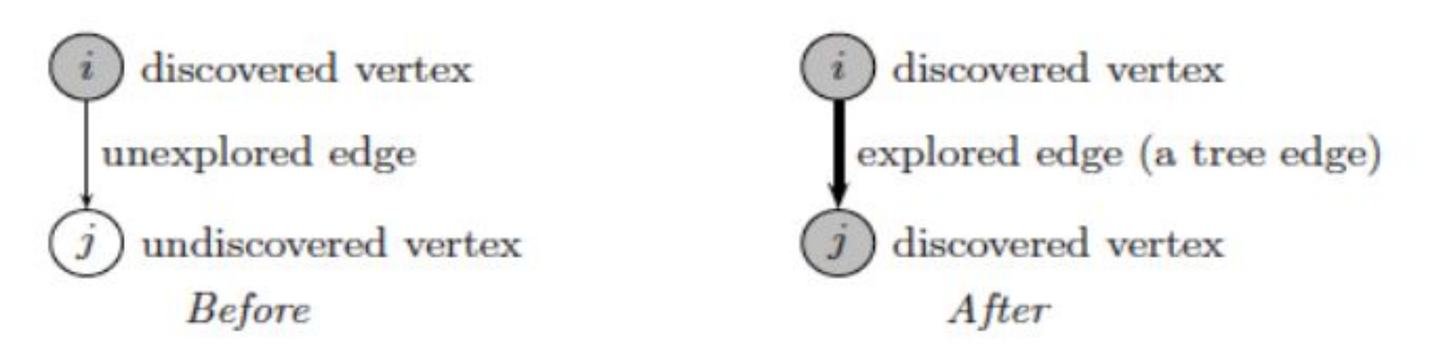
- Equivalence classes
  - Vertices in each pair are reachable from each other
  - Connected component (CC)
- Connected graph
  - SIngle CC
- Articulation point
  - Vertex in connected graph
  - Removing this disconnects the graph

Representation

- Sequential
  - Adjacency matrix
  - Cost adjacency matrix
- Linked
  - Adjacency list

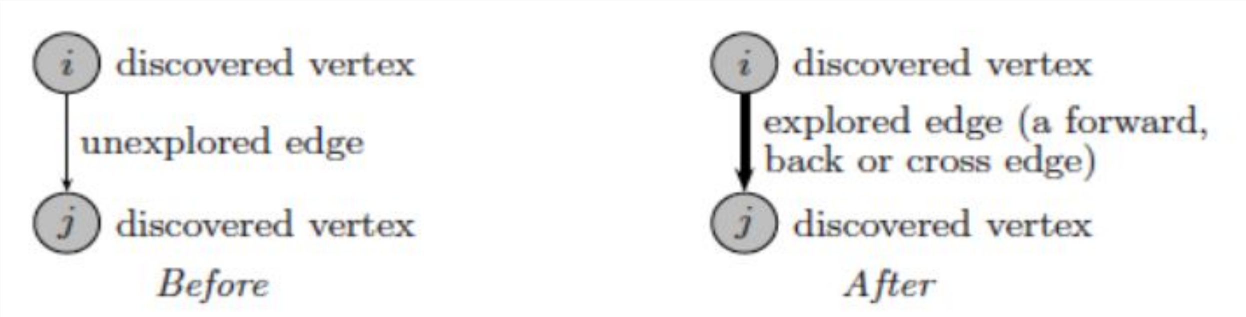
Traversal

- Discovery edge



Traversal

- Non-discovery edge



Traversal

- Depth-first search
- Breadth-first search

Depth-First Search

Color	**State**	**Meaning**
White	Undiscovered	All **entering edges** are unexplored
Gray	Discovered but unfinished	There are still unexplored **leaving edges**
Black	Discovered and finished	All **leaving edges** have been explored

Name	**Description**	**Rule for edge (A,B)**	**Undirected?**	**Directed?**
Tree edge	**Branch** in depth-first tree (DFT)	**B** is **white**	Yes	Yes
Back edge	Edge from **A** to its **ancestor B** in DFT	**B** is **gray**	Yes	Yes
Forward edge	Edge from **A** to its **descendant B** in DFT	**B** is **black** and **A** was discovered **before B** (d(A) < d(B))	No	Yes
Cross edge	Edge from **A** to **B** (**neither descendant nor ancestor** in DFT)	**B** is black and **A** was discovered **after B** (d(A) > d(B))	No	Yes

Properties of DFS

- Descendants of vertex **A** are all undiscovered vertices which are reachable from **A**
- **B** is a descendant of **A** if **B** can be reached from **A** using only undiscovered vertices after DFS discovers **A**
- Complexity
  - O(n + e) for adjacency list
  - O(n^2) for adjacency matrix
- Discovery time (vertex)
  - Time DFS colored vertex **gray**
- Finishing time (vertex)
  - Time DFS colored vertex **black**

Breadth-First Search

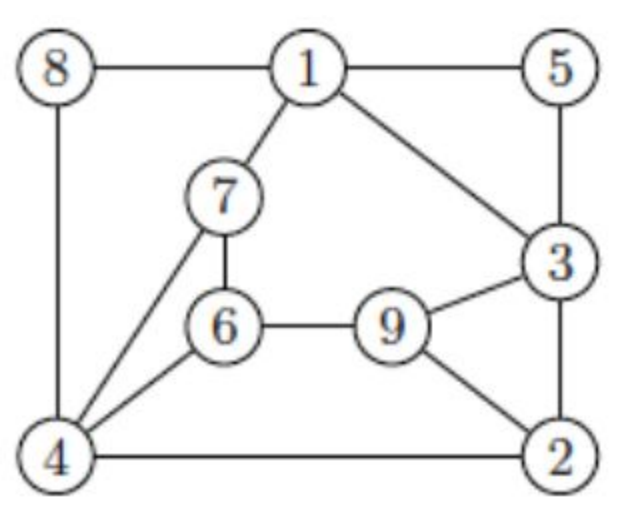
Color	**State**	**Meaning**
White	Undiscovered	All **entering edges** are unexplored
Gray	Discovered, fringe	There are still unexplored **leaving edges**
Black	Discovered, finished	All **leaving edges** have been explored

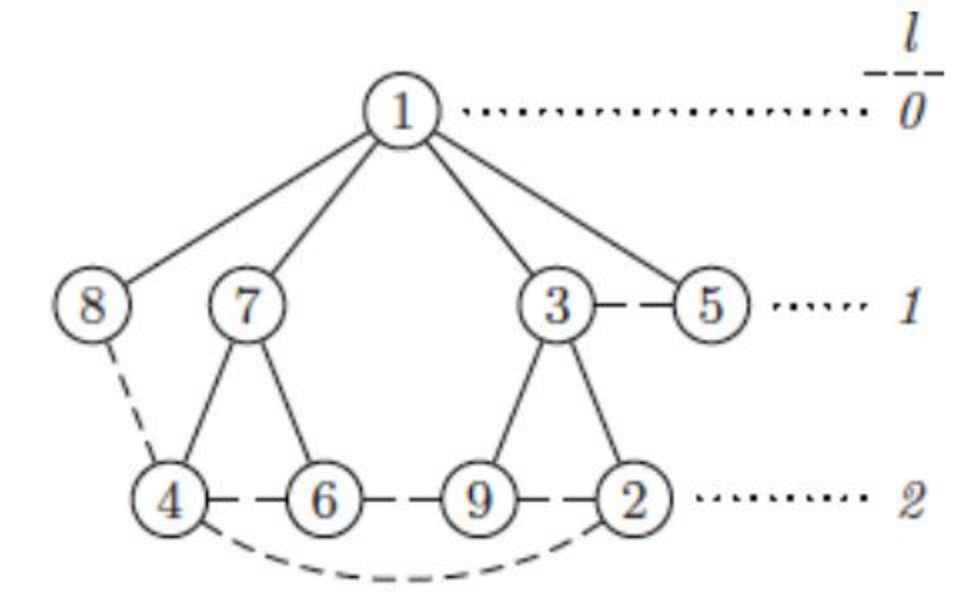
Edges in BFS

- No forward edges
- Undirected
  - Tree edges
  - Cross edges
- Directed
  - Tree edges
  - Cross edges
  - Back edges

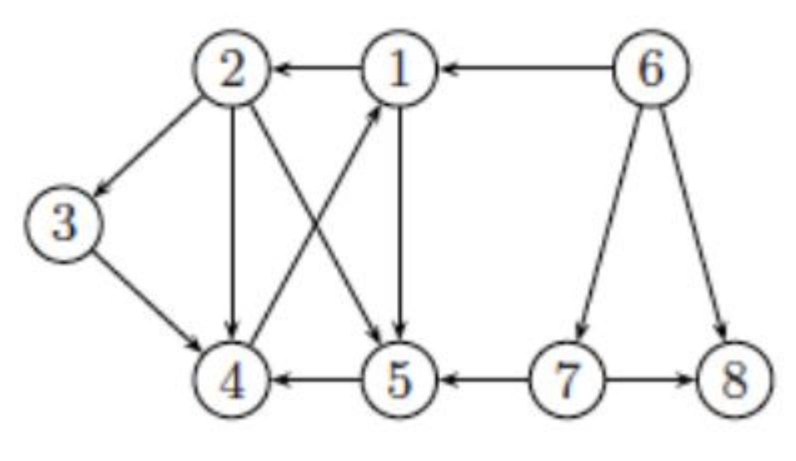
BFS

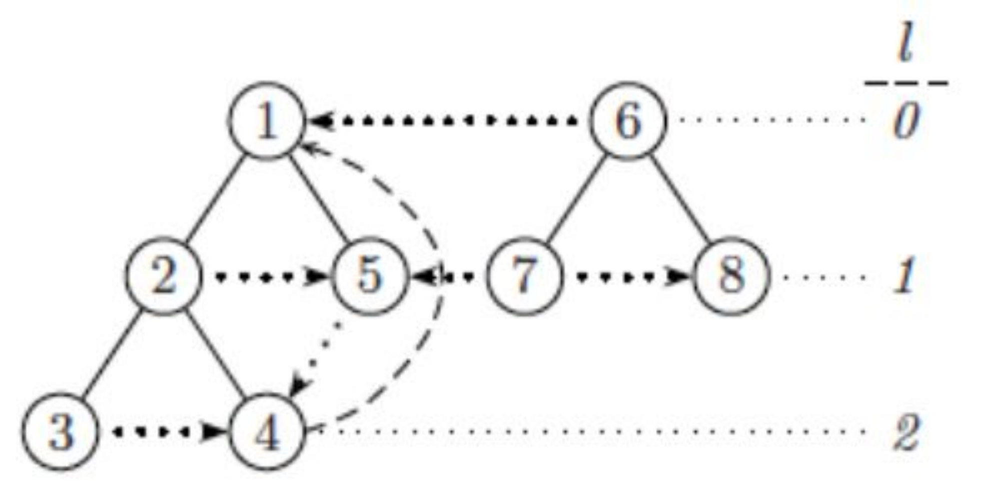
- Make all vertices white
- Get starting vertex and enqueue
- Assign 0 to starting vertex distance
- Do until queue is empty:
  - Dequeue vertex (current)
  - For each adjacent vertex from current
    - If adjacent vertex is white
      - Make gray
      - Make distance = current distance + 1
      - Enqueue
  - Make current vertex black





Undirected graph


2
1
6
3
4
5
7
8


l
1
6
0
2
5
7
8
1
3
4
2

Properties of BFS

- Shortest path from start vertex
- Queue is used to store vertices in fringe
- Finish all vertices with distance **x** before those with distance **x + 1**
- Complexity
  - O(n + e) for adjacency list
  - O(n^2) for adjacency matrix

Identifying directed acyclic graphs

Directed acyclic graphs: cycle detection

- Directed graphs
- Undirected graphs

Cycle detection (directed)

- Directed acyclic graph
  - Dag
  - Directed graph with no cycles
- Traverse using DFS
  - Back edge means there is a cycle
- Key points
  - If digraph is acyclic, DFS will have no back edges
  - If DFS has no back edges, digraph is acyclic

Cycle detection (undirected)

- Free tree
  - Connected acyclic undirected graph
  - n - 1 edges
  - Adding another edge produces a cycle
- Traverse using BFS
  - Cross edge means there is a cycle
- O(n)

Toposort

- Perform DFS then push vertex upon finishing
- Pop all vertices after DFS as output

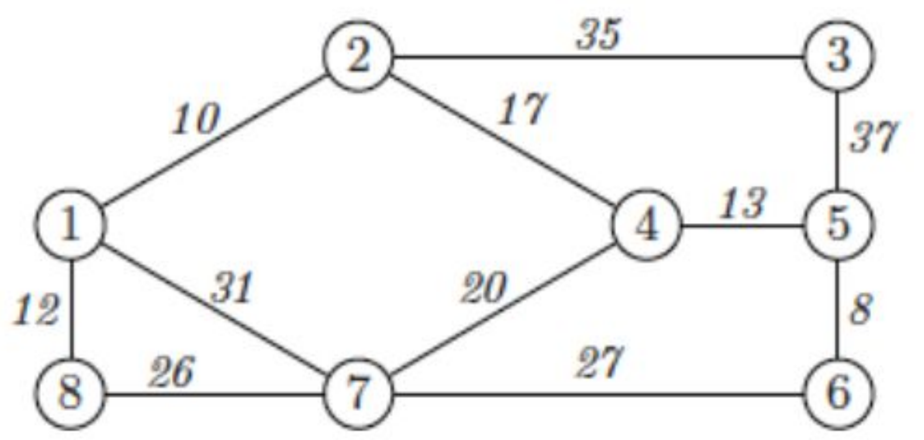
Finding strongly connected components (directed graph)

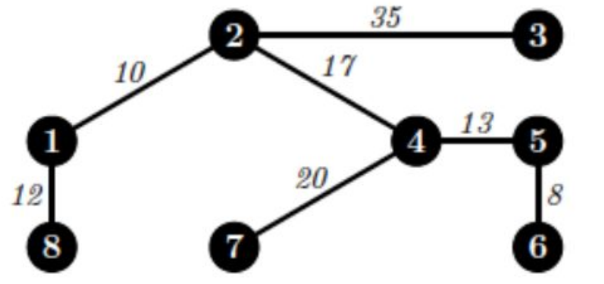
- Perform DFS on the graph G
- Get the transpose of G
  - Reverse all edges
- DFS on the transpose of G
  - Use finishing times of DFS on G
  - Largest to smallest
- Component graph
  - Always a dag
  - Cross-component edges

Articulation points and biconnected components (undirected graph)

- Root (DFS)
  - Two or more children
- Nonroot vertex (**Z**)
  - No backedge from any descendant of **Z** to **Z** itself or any ancestor of **Z**

1. [Start vertex] Choose any vertex in V and place it in U.
2. [Next vertex] From among the vertices in $V - U$ choose that vertex, say j, which is connected to some vertex, say i, in U by an edge of least cost. Add vertex j to U and edge (i, j) to T.
3. [All vertices considered?] Repeat Step 2 until $U = V$. Then, T is a minimum-cost spanning tree for G.

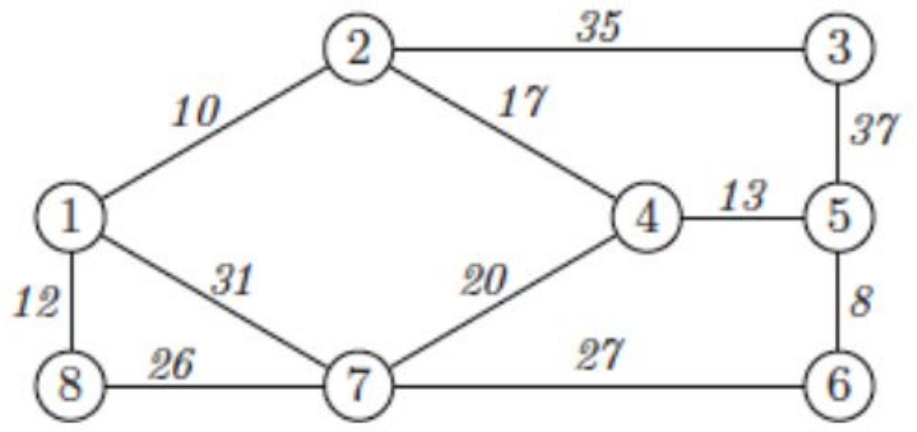


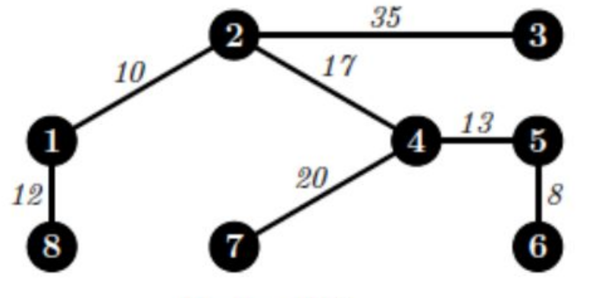


Cost = 115

Prim’s algorithm

1. [Initial edge.] Choose the edge of least cost among all the edges in E and place it in T.
2. [Next edge.] From among the remaining edges in E choose the edge of least cost, say edge (i, j). If including edge (i, j) in T creates a cycle with the edges already in T, discard (i, j); otherwise, include (i, j) in T.
3. [Enough edges in T?] Repeat Step 2 until there are $n - 1$ edges in T. Then T is a minimum-cost spanning tree for G.



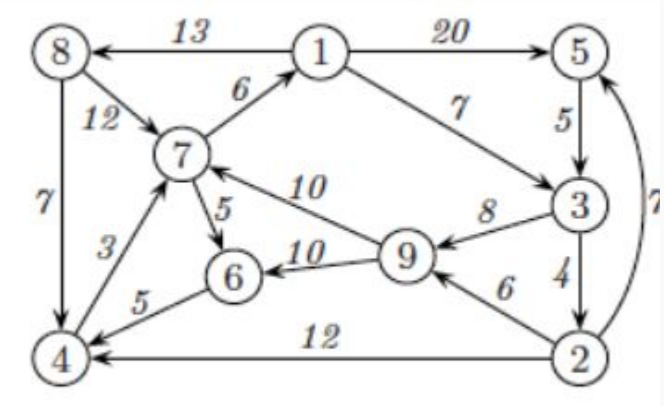


Kruskal's algorithm

1. Place vertex s in class 1 and all other vertices in class 2.
2. Set the value of vertex s to zero and the value of all other vertices to ∞.
3. Do the following until all vertices v in V that are reachable from s are placed in class 1:
   a. Denote by u the vertex most recently placed in class 1.
   b. Adjust all vertices v in class 2 as follows:
      (i) If vertex v is not adjacent to u, retain the current value of $d(v)$.
      (ii) If vertex v is adjacent to u, adjust $d(v)$ as follows:

$$\textbf{if } d(v) > \delta(u)+w((u,v)) \textbf{ then } d(v) \leftarrow \delta(u)+w((u,v)) \tag{10.1}$$

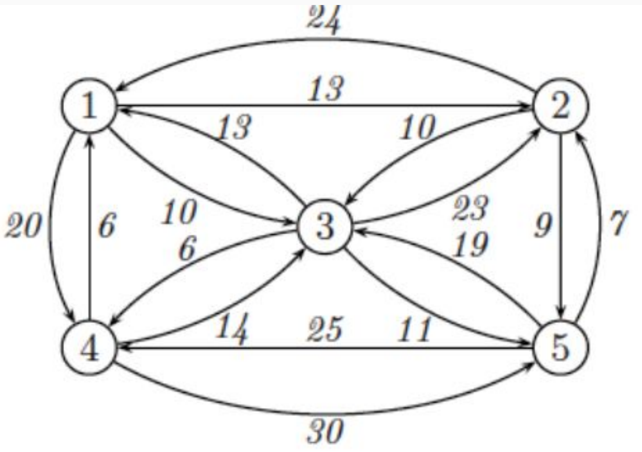
   c. Choose a class 2 vertex with *minimal* value and place it in class 1.



1. [Initialize] $D^{(0)} \leftarrow C$
2. [Iterate] Repeat for $k = 1, 2, 3, \ldots, n$

$$d_{ij}^{(k)} \leftarrow \text{minimum}[\, d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\,], \quad 1 \le i, j \le n$$

Then, $D^{(n)}$ contains the cost of the shortest path between every pair of vertices i and j in G.



$$\begin{bmatrix} 0 & 13 & 10 & 20 & \infty \\ 24 & 0 & 10 & \infty & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & \infty & 14 & 0 & 30 \\ \infty & 7 & 19 & 25 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 13 & 10 & 20 & \infty \\ 24 & 0 & 10 & \mathit{44} & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & \mathit{19} & 14 & 0 & 30 \\ \infty & 7 & 19 & 25 & 0 \end{bmatrix}$$

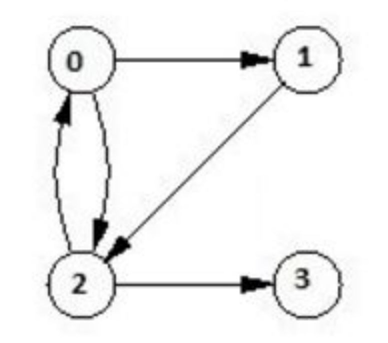
Floyd’s algorithm

$$\begin{bmatrix} 0 & 13 & 10 & 20 & \infty \\ 24 & 0 & 10 & \infty & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & \infty & 14 & 0 & 30 \\ \infty & 7 & 19 & 25 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 2 & 0 & 2 \\ 3 & 3 & 0 & 3 & 3 \\ 4 & 0 & 4 & 0 & 4 \\ 0 & 5 & 5 & 5 & 0 \end{bmatrix}$$

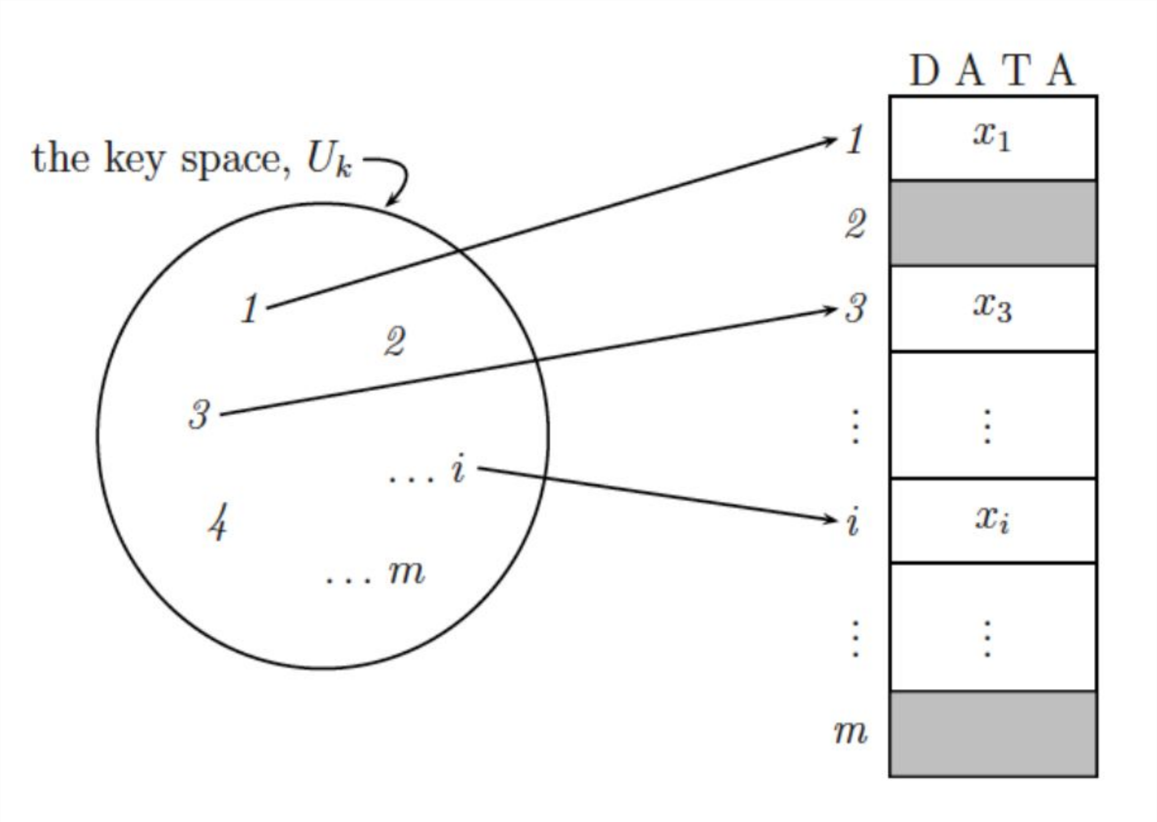
$$D^{(0)} \qquad pred^{(0)}$$

$$\begin{bmatrix} 0 & 13 & 10 & 20 & \infty \\ 24 & 0 & 10 & \mathit{44} & 9 \\ 13 & 23 & 0 & 6 & 11 \\ 6 & \mathit{19} & 14 & 0 & 30 \\ \infty & 7 & 19 & 25 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 2 & \mathit{1} & 2 \\ 3 & 3 & 0 & 3 & 3 \\ 4 & \mathit{1} & 4 & 0 & 4 \\ 0 & 5 & 5 & 5 & 0 \end{bmatrix}$$

$$D^{(1)} \qquad pred^{(1)}$$


0
1
2
3

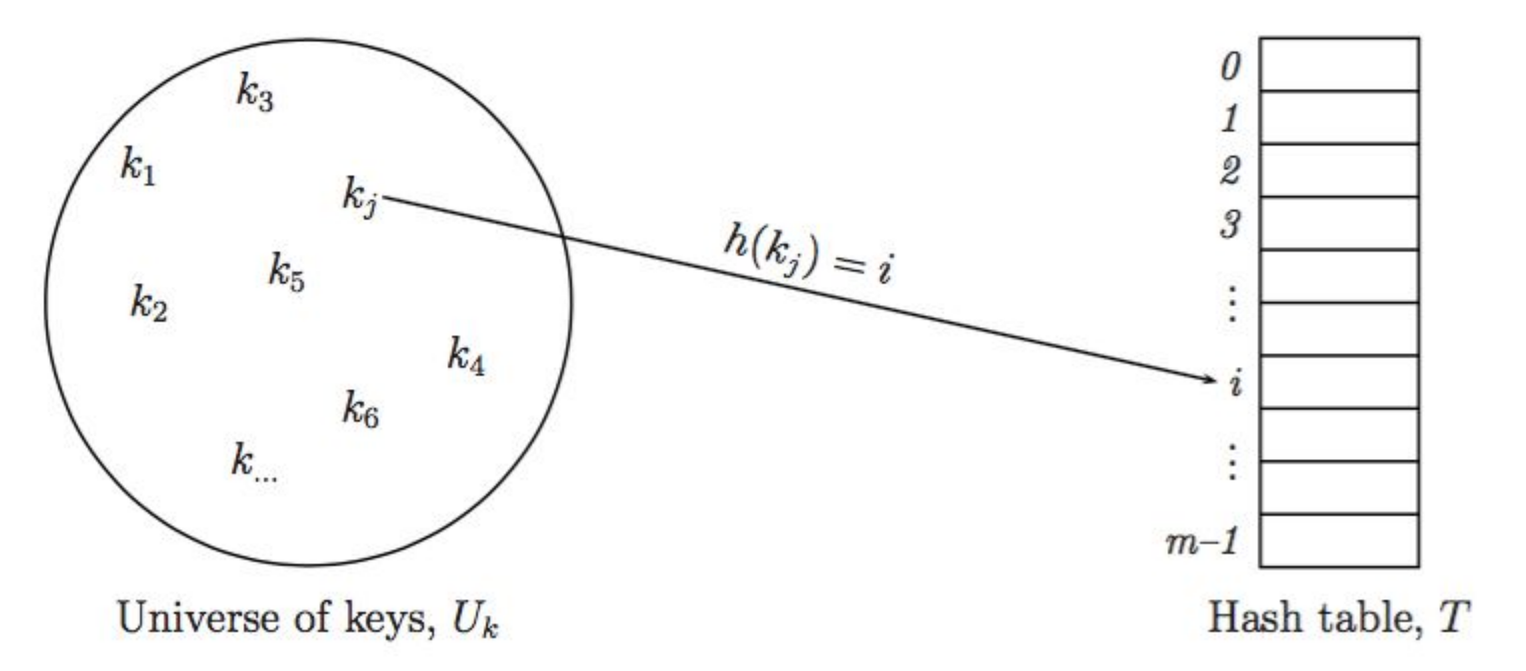
Hash tables



Direct access table

Hash table

- Maps **keys** to **data**
- Fast accessing
  - Array implementation underneath
- Keys are converted to indexes



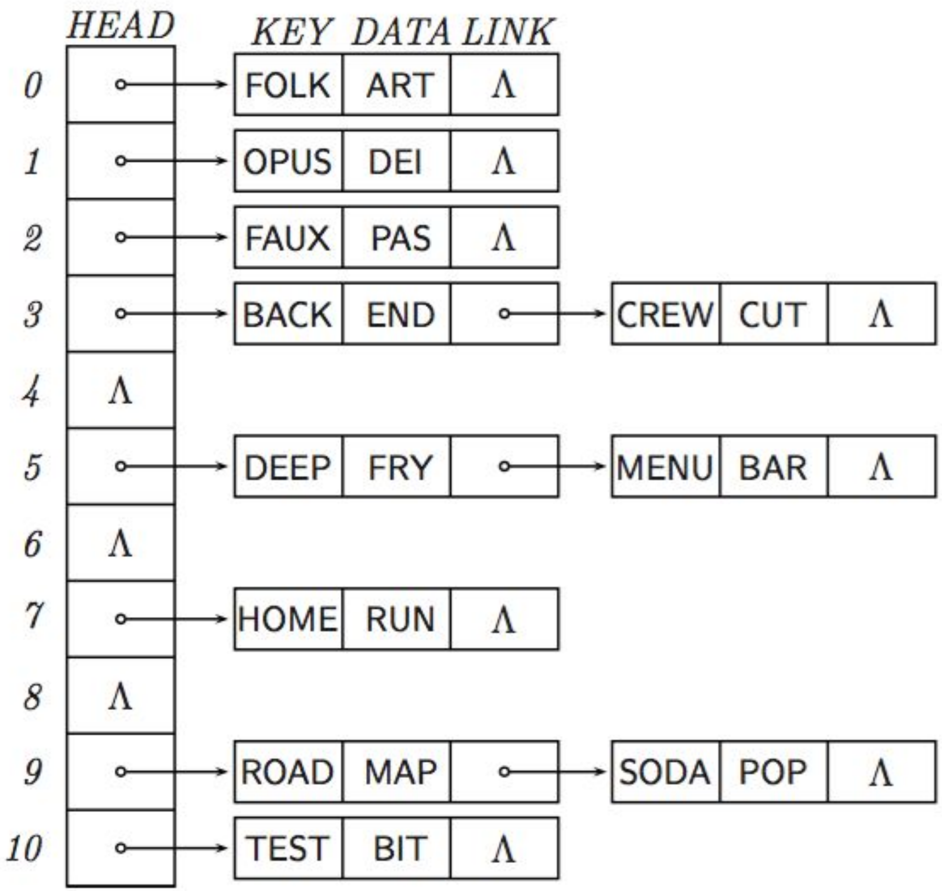
Converting keys to indexes

Hash function h(k)

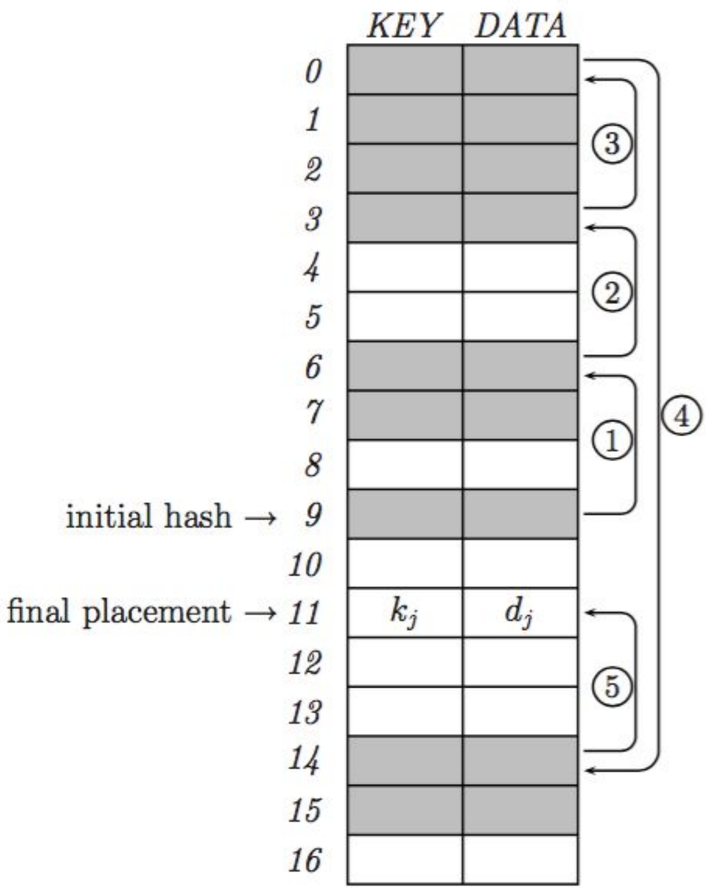
- Must be computed quickly
- Must minimize **collisions**

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

$$h(K) = \lfloor m(K\theta \bmod 1) \rfloor$$



Collision resolution by chaining



Collision resolution by open addressing

$$p(K) = [h(K) - i] \bmod m \qquad i = 0, 1, 2, \ldots, m-1$$

(k, d)		$h(k)$
FOLK	ART	10
TEST	BIT	16
BACK	END	2
ROAD	MAP	7
HOME	RUN	8
CREW	CUT	10
FAUX	PAS	15
OPUS	DEI	15
SODA	POP	0
DEEP	FRY	2
MENU	BAR	18

	KEY	*DATA*
0	SODA	POP
1	DEEP	FRY
2	BACK	END
3		
4		
5		
6		
7	ROAD	MAP
8	HOME	RUN
9	CREW	CUT
10	FOLK	ART
11		
12		
13		
14	OPUS	DEI
15	FAUX	PAS
16	TEST	BIT
17		
18	MENU	BAR

Open addressing: linear probing (i = 1); clustering

(k, d)		$h(k)$	$h'(k)$
FOLK	ART	10	12
TEST	BIT	16	15
BACK	END	2	2
ROAD	MAP	7	6
HOME	RUN	8	9
CREW	CUT	10	17
FAUX	PAS	15	3
OPUS	DEI	15	5
SODA	POP	0	7
DEEP	FRY	2	15
MENU	BAR	18	4

	KEY	*DATA*
0	SODA	POP
1		
2	BACK	END
3		
4		
5	OPUS	DEI
6	DEEP	FRY
7	ROAD	MAP
8	HOME	RUN
9		
10	FOLK	ART
11		
12	CREW	CUT
13		
14		
15	FAUX	PAS
16	TEST	BIT
17		
18	MENU	BAR