# Data Structures: Trees

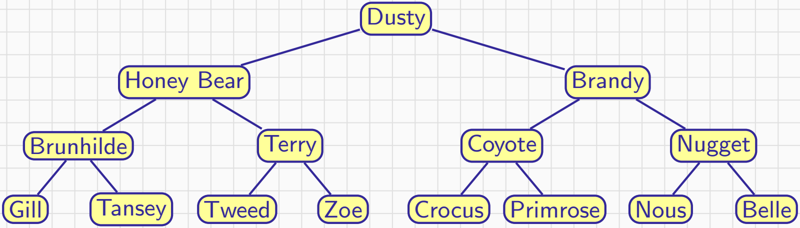
**Wei-Mei Chen**

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

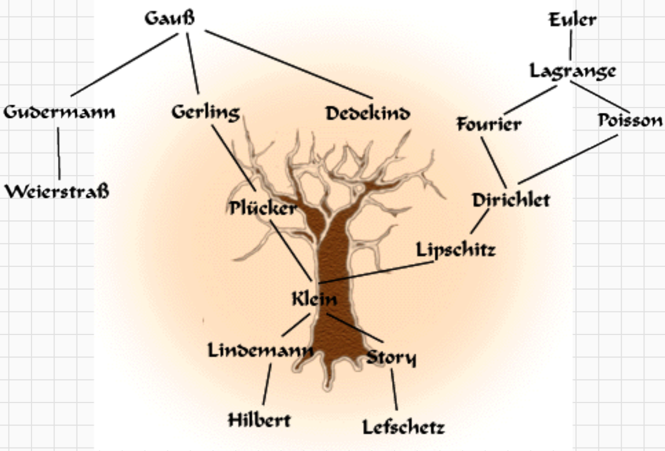
# Introduction

# Introduction

- A tree structure means that the data are organized so that items of information are related by branches
- Example:



# The Mathematics Genealogy Project

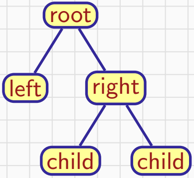


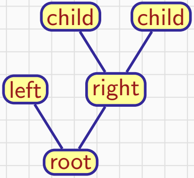
# Definition of Trees

**Definition (recursively):**

A tree is a finite set of one or more nodes such that:

- There is a specially designated node called **root**.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called the subtrees of the root.



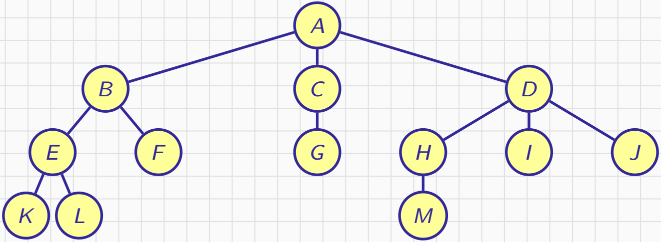




**node**

# Tree Terminology

- **degree:** the number of subtrees of a node
  **degree of a tree:** the maximum of the degree of the nodes in the tree.
- **terminal nodes (or leaf):** nodes that have degree zero
- **nonterminal nodes:** nodes that don't belong to terminal nodes.
- A node that has subtrees is the **parent** of the roots of the subtrees.
- The roots of these subtrees are the **children** of the node.
- Children of the same parent are **siblings**.
- The **ancestors** of a node are all the nodes along the path from the root to the node.

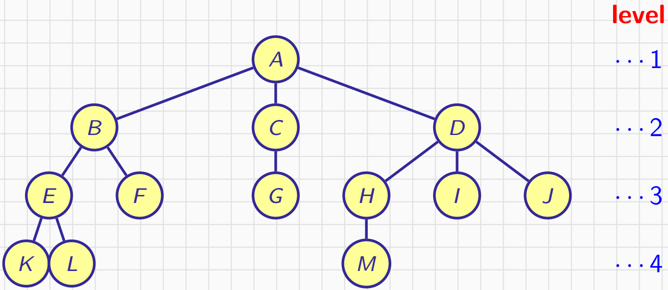
# An Example of Trees




- $A$ is the root node
- $B$ is the parent of $E$ and $F$
- $K$ is the sibling of $L$
- $F, G, I, J, K, L, M$ are external nodes, or leaves
- $A, B, C, D, E, H$ are internal nodes
- The ancestors of node $L$ are $E, B, A$
- The degree of node $B$ is 2 and the degree of the tree is 3

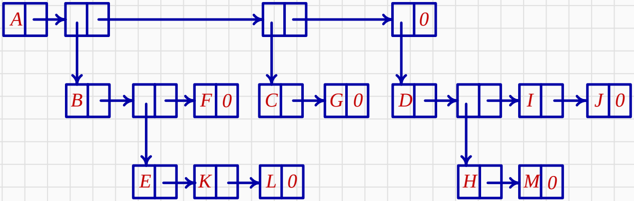
# Level and Depth

- The **level** of a node: defined by letting the root be at level one. If a node is at level $l$, then it children are at level $l + 1$.
- **Height** (or **depth**): the maximum level of any node in the tree



# Representation of Trees

- Parenthetical notation : $(A(B(E(K, L), F), C(G), D(H(M), I, J)))$
- List Representation




- Degree $k$

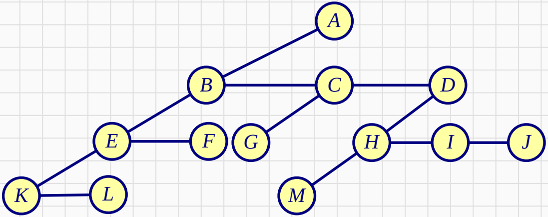
| | | | | |
|---|---|---|---|---|
| *data* | $child_1$ | $child_2$ | $\cdots$ | $child_k$ |

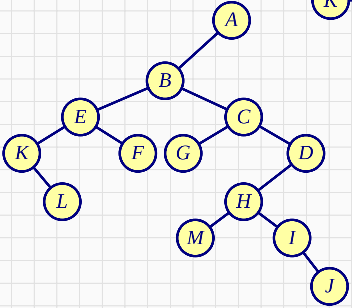
**Lemma 5.1**

If $T$ is a $k$-ary tree with $n$ nodes, each having a fixed size, then $n(k-1)+1$ of the $nk$ child fields are 0, $n \geq 1$.

# Left Child-Right Sibling Representation

| data | |
|---|---|
| *left child* | *right sibling* |



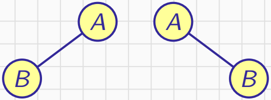
# Binary Trees

# Binary Trees

**Definition:**

A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree

- Binary trees are characterized by the fact that any node can have at most two branches.
- Any tree can be transformed into a binary tree.
  ➠ by left child-right sibling representation
- The left subtree and the right subtree are distinguished.



# ADT Binary_Tree

**structure** *Binary_Tree* (abbreviated *BinTree*) is
**objects**: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.
**functions**:
for all *bt*,*bt*1,*bt*2 ∈ *BinTree*, *item* ∈ *element*

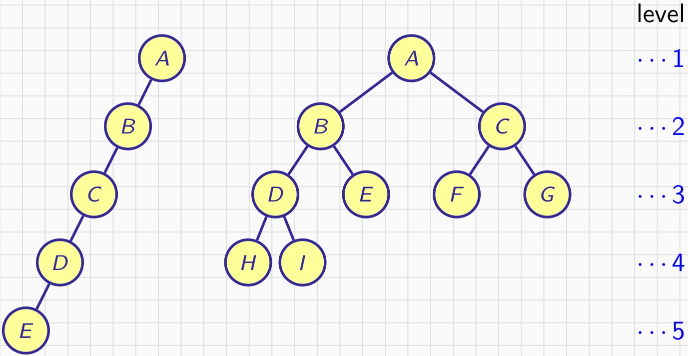
| | | |
|---|---|---|
| *BinTree* Create() | ::= | creates an empty binary tree |
| *Boolean* IsEmpty(*bt*) | ::= | **if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE* |
| *BinTree* MakeBT(*bt*1, *item*, *bt*2) | ::= | **return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*. |
| *BinTree* Lchild(*bt*) | ::= | **if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*. |
| *element* Data(*bt*) | ::= | **if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*. |
| *BinTree* Rchild(*bt*) | ::= | **if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*. |

**Structure 5.1**: Abstract data type *Binary_Tree*

## Samples of Binary Trees

Two special kinds of binary trees:

- skewed trees
- The all leaf nodes of these trees are on two adjacent levels



# Properties of Binary Trees

**Lemma 5.2 [Maximum # of nodes]**

1. The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$.
2. The maximum number of nodes in a binary tree of depth $k$ is $2^k - 1$, $k \geq 1$.

**Lemma 5.3 [Relation between # of leaves and degree-2 nodes]**

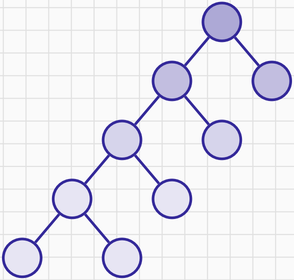
For any nonempty binary tree, $T$, if $n_0$ is the number of leaf nodes and $n_2$ is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

These lemmas allow us to define full and complete binary trees

# Full Binary Trees

**Definition**

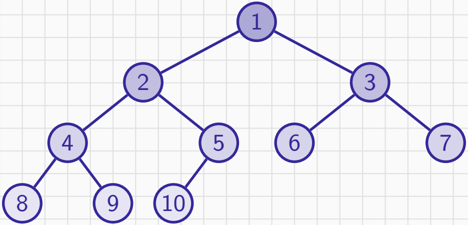
A binary tree is full if every node other than the leaves has two children.



# Complete Binary Trees

**Definition**

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible



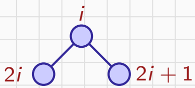

☞ Height of a complete binary tree with $n$ nodes : $\lceil \log_2(n+1) \rceil$

# Binary Tree Representations

## Lemma 5.4

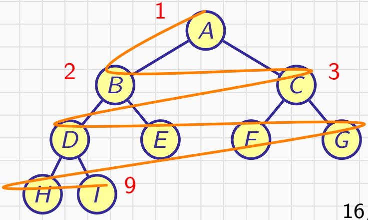
If a complete binary tree with $n$ nodes is represented sequentially, then for any node with index $i$, $1 \leq i \leq n$, we have

1. *parent*$(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, $i$ is at the root and has no parent.
2. *leftChild*$(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then $i$ has no left child.
3. *rightChild*$(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then $i$ has no right child.



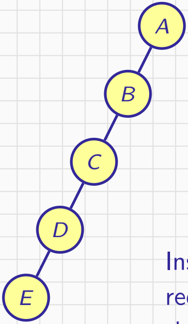

**ARRAY**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| — | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* |



# Sequential Representation

Waste spaces: in the worst case, a skewed tree of depth $k$ requires $2^k - 1$ spaces and only $k$ spaces will be occupied.



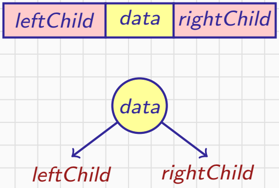

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | $\cdots$ | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| — | $A$ | $B$ | — | $C$ | — | — | — | $D$ | — | — | $E$ |

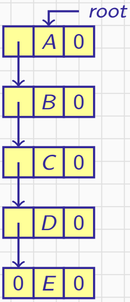
Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes.

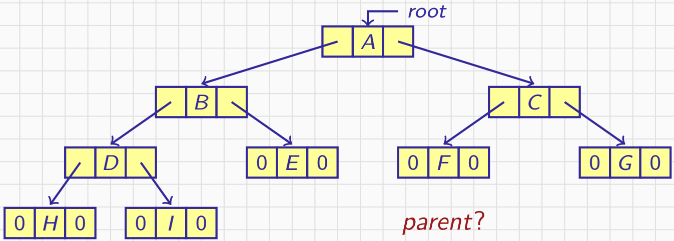
# Linked Representation

## Declaration

```
typedef struct node *treePointer;
typedef struct node {
    int data;
    treePointer leftChild, rightChild;
} node;
```



# Traversals

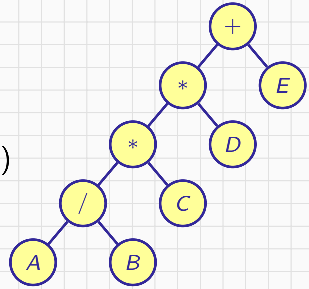
## Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  1. **inorder**: L**V**R,
  2. **postorder**: LR**V**, and
  3. **preorder**: **V**LR

| L | V | R |
|---|---|---|
| *leftChild* | *data* | *rightChild* |

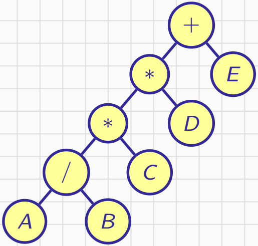
# Arithmetic Expressions

- inorder traversal (infix expression):
  LVR ➠ $A/B * C * D + E$
- preorder traversal (prefix expression)
  VLR ➠ $+ * * / ABCDE$
- postorder traversal (postfix expression)
  LRV ➠ $AB/C * D * E+$
- level order traversal
  ➠ $+ * E * D / CAB$



# Inorder Traveral

```
void inorder(treePointer ptr)
{
    if (ptr) {
        inorder(ptr->leftChild);
        printf("%d",ptr->data);
        inorder(ptr->rightChild);
    }
}
```

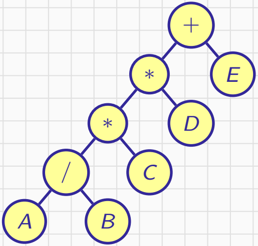



$A/B * C * D + E$

| Call of inorder | Value in *root* | Action |
|---|---|---|
| 1 | + | |
| 2 | * | |
| 3 | * | |
| 4 | / | |
| 5 | A | |
| 6 | NULL | |
| 5 | A | printf |
| 7 | NULL | |
| 4 | / | printf |
| 8 | B | |
| 9 | NULL | |
| 8 | B | printf |
| 10 | NULL | |
| 3 | * | printf |
| 11 | C | |
| 12 | NULL | |
| 11 | C | printf |
| 13 | NULL | |
| 2 | * | printf |
| 14 | D | |
| 15 | NULL | |
| 14 | D | printf |
| 16 | NULL | |
| 1 | + | printf |
| 17 | E | |
| 18 | NULL | |
| 17 | E | printf |
| 19 | NULL | |

# Preorder Traveral

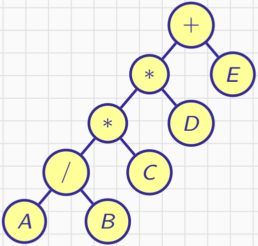
```
void preorder(treePointer ptr)
{
    if (ptr) {
        printf("%d",ptr->data);
        preorder(ptr->leftChild)
        preorder(ptr->rightChild);
    }
}
```




$\Rightarrow$ $+**/ABCDE$

# Postorder Traveral

```
void postorder(treePointer ptr)
{
    if (ptr) {
        postorder(ptr->leftChild)
        postorder(ptr->rightChild);
        printf("%d",ptr->data);
    }
}
```




➠ $AB/C*D*E+$

# Iterative Inorder Traversal

```
void iterInorder(treePointer node)
{
    int top = -1;
    treePointer stack [MAX_STACK_SIZE];
    for (;;) {
        for(; node; node = node->leftChild)
            push(node);
        node = pop();
        if (!node) break;
        printf("%d", node->data);
        node = node->rightChild;
    }
}
```

Using a stack

- No action ➠ the node is added to the stack
- "printf" action ➠ the node is removed from the stack

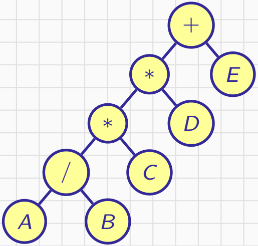
## Analysis of `iterInorder`

- The left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node's right child is stacked.
- Every node is placed on and removed from the stack exactly once.
- Let $n$ be the number of nodes in the tree.
  - Time complexity: $O(n)$
  - Space complexity: $O(n)$ ($\approx$ the depth of the tree )

# Level Order Traversal

```
void levelOrder(treePointer ptr)
{
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return;
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        if (ptr) {
            printf("%d",ptr->data);
            if (ptr->leftChild)
                addq(ptr->leftChild);
            if (ptr->rightChild)
                addq(pt->rightChild);
        }
        else break;
    }
}
```

Using a queue




$+ * E * D / CAB$

# Traversal without a Stack

- Q: Is binary tree traversal possible without the use of extra space for a stack?
- Some possible solutions
  - Add a *parent* field to each node
  - Threaded binary trees in §5.5

# BT Operations

# Copying Binary Trees

```
itreePointer copy(treePointer original)
{
    treePointer temp;
    if (original) {
        MALLOC(temp, sizeof(*temp));
        temp->leftChild = copy(original->leftChild);
        temp->rightChild = copy(original->rightChild);
        temp->data = original->data;
        return temp;
    }
    return NULL;
}
```

postorder

# Equality of Binary Trees

```
int equal(treePointer first, treePointer second)
{
   return ((!first && !second) ||
       (first && second && (first->data == second->data) &&
       equal(first->leftChild,second->leftChild) &&
       equal(first->rightChild,second->rightChild))
}
```

The same structure and data

# The Satisfiability Problem

- variable: $x_1$, $x_2$,..., $x_n$
  operator: $\wedge$(and), $\vee$(or), and $\neg$(not)
  - A variable is an expression.
  - If $x$ and $y$ are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.
  - Parentheses can be used to alter the normal order of evaluation, which is $\neg$ before $\wedge$ before $\vee$).
- Example: $x_1 \vee (x_2 \wedge \neg x_3)$, $x_1, x_3$: false, $x_2$: truth
  $\Rightarrow F \vee (T \wedge \neg F)$
  $= F \vee (T \wedge T)$
  $= F \vee T = T$
- The satisfiability problem [Newell, Shaw, and Simon (1950s)]:
  Q: Is there an assignment to make an expression true?

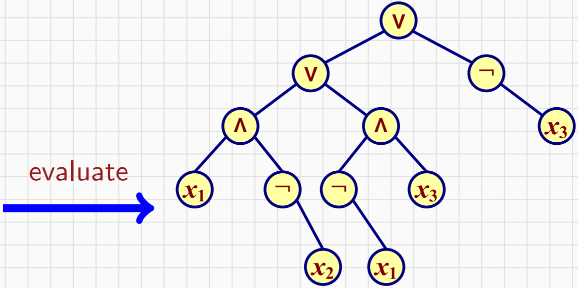
# Propositional Calculus Expression

Consider

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

$n$ variables $\Rightarrow$ $2^n$ possible combination $\Rightarrow O(E \cdot 2^n)$

| $x_1$ | $x_2$ | $x_3$ |
|---|---|---|
| $T$ | $T$ | $T$ |
| $T$ | $T$ | $F$ |
| $T$ | $F$ | $T$ |
| $T$ | $F$ | $F$ |
| $F$ | $T$ | $T$ |
| $F$ | $T$ | $F$ |
| $F$ | $F$ | $T$ |
| $F$ | $F$ | $F$ |




**postorder traversal (postfix evaluation)**

# Node Structure for the Satisfiability Problem

```
typedef emun { not, and, or, true, false } logical;
typedef struct node *treePointer;
typedef struct node {
    treePointer leftChild;
    logical data;
    short int value;
    treePointer rightChild;
} node;
```

| | | | |
|---|---|---|---|
| *leftChild* | *data* | *value* | *rightChild* |

```
void postOrderEval(treePointer node)
{   if (node) {
        postOrderEval(node->leftChild);
        postOrderEval(node->rightChild);
        switch(node->data) {
            case not:
                node->value = !node->rightChild->value;
                break;
            case and:
                node->value = node->rightChild->value &&
                    node->leftChild->value;
                break;
            case or:
                node->value = node->rightChild->value ||
                    node->leftChild->value;
                break;
            case true:
                 node->value = TRUE;
                break;
            case false:
                node->value = FALSE;
        }
    }
}
```

**Time Complexity?**

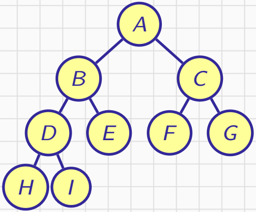
# Threaded BTs

# Threads

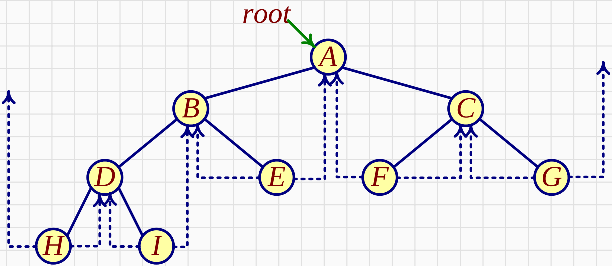
- There are more null links than actual pointers.
- $n$: number of nodes
  number of non-null links: $n - 1$
  total links: $2n$
  null links: $2n - (n - 1) = n + 1$
- Solution: replace these null pointers with some useful "threads"

## Rules for Constructing the Threads

- If $ptr->leftChild$ is null, replace it with a pointer to the node that would be visited before $ptr$ in an inorder traversal
  ➠ **the inorder predecessor of** $ptr$
- If $ptr->rightChild$ is null, replace it with a pointer to the node that would be visited after $ptr$ in an inorder traversal
  ➠ **the inorder successor of** $ptr$

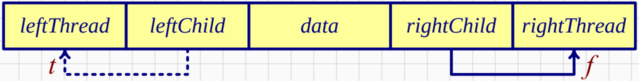


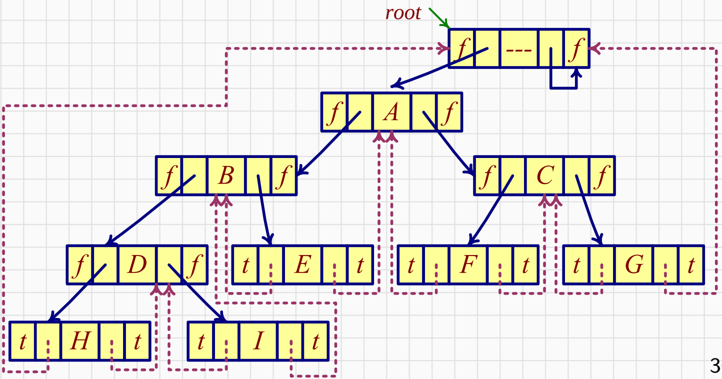

*HDIBEAFCG*



# A Threaded Binary Tree

An empty threaded binary tree



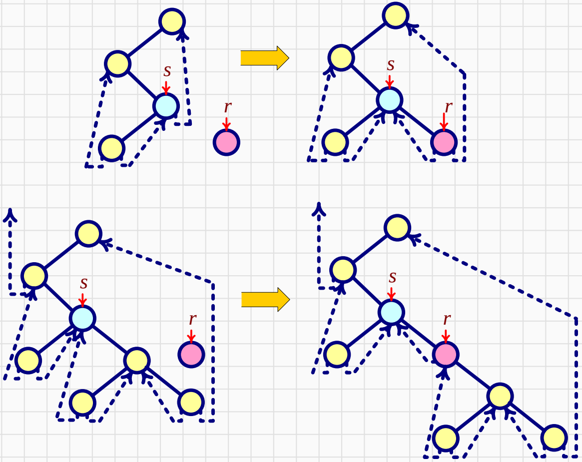
# Inorder Traversal of a Threaded Binary Tree

```
threadedPointer insucc(threadedPointer tree)
{   /*Finding the Inorder Sucessor of a Node
    threadedPointer temp;
    temp = tree->rightChild;
    if (!tree->rightThread)
        while (!temp->leftThread)
            temp = temp->leftChild;
    return temp;
}
```

```
void tinorder(threadedPointer tree)
{  threadedPointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp == tree) break;
        printf("%3c", temp->data);
    }
}
```

$O(n)$

# Right Insertion in a Threaded Binary Trees

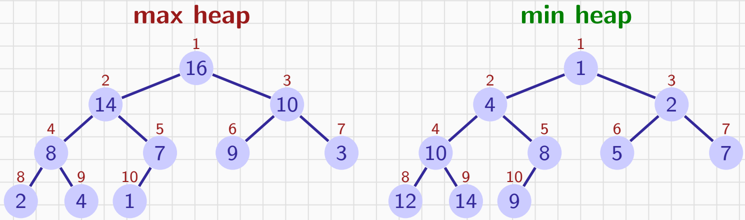


# Heaps

## Definition:

A max(min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children. A max (min) heap is a complete binary tree that is also a max (min) tree.

- Operations on heaps: creation, insertion and deletion




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 2 | 10 | 8 | 5 | 7 | 12 | 14 | 9 |

# ADT MaxPriorityQueue

ADT *MaxPriorityQueue* is
**object:** a collection of $n > 0$ elements, each element has a key
**function:**

| | | |
|---|---|---|
| *MaxPriorityQueue* create(max_size) | ::= | create an empty priority |
| Boolean isEmpty($q, n$) | ::= | **if** ($n > 0$) **return** *TRUE* |
| | | **else return** *FALSE* |
| Element top($q, n$) | ::= | **if** (!isEmpty($q, n$)) **return** an instance |
| | | of the largest element in $q$ |
| | | **else return** error |
| Element pop($q, n$) | ::= | **if**(!isEmpty($q, n$)) **return** an instance |
| | | of the largest element in $q$ and |
| | | remove it from the heap **else return** error |
| *MaxPriorityQueue* push($q$, *item*, $n$) | ::= | insert *item* into *pq* and return the |
| | | resulting priority queue |

## Priority Queues

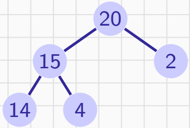
- Queue in Chapter 3: FIFO
- Priority queues
    - Heaps are frequently used to implement priority queues
        - Delete the element with highest (lowest) priority
        - Insert the element with arbitrary priority
    - A heap is an efficient way to implement priority queue
- Application: machine service
    - amount of time (min heap)
    - amount of payment (max heap)

## Priority Queue Representations

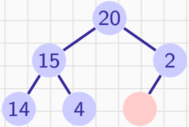
| Representation | Insertion | Deletion |
|---|---|---|
| Unordered array | $O(1)$ | $O(n)$ |
| Unordered linked list | $O(1)$ | $O(n)$ |
| Sorted array | $O(n)$ | $O(1)$ |
| Sorted linked list | $O(n)$ | $O(1)$ |
| Max heap | $O(\log n)$ | $O(\log n)$ |

Note: A heap is a complete binary tree with $n$ elements, it has a height of $\lceil \log_2(n+1) \rceil$.

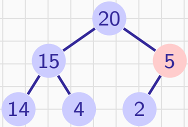
# Insertion into a Max Heap



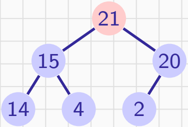

```c
void push(element item, int *n)
{   int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full.  \n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```




The last place

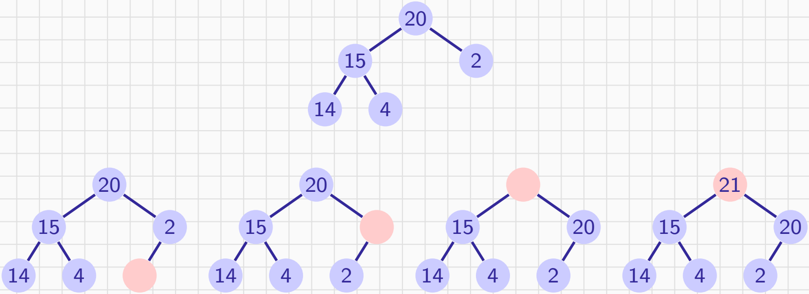



Insert 5




Insert 21

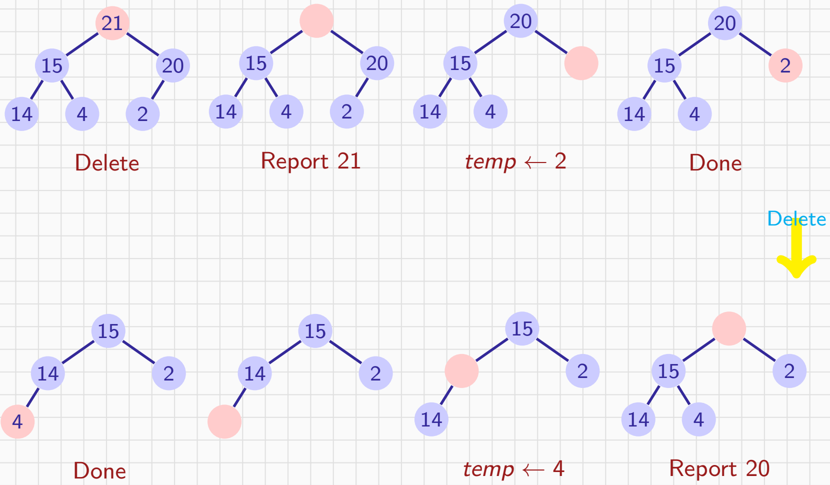
# Inserting 21 into a Max Heap



# Deletion from a Max Heap

```
element pop(int *n)
{   int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    item = heap[1];     // Report the first node
    temp = heap[(*n)--];     // temp is the last value and n=n-1
    parent = 1;
    child = 2;
    while (child <= *n) {
        if ((child < *n) && (heap[child].key < heap[child+1].key))
            child++;
        if (temp.key >= heap[child].key) break;
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return item;
}
```

# Deleting from a Max Heap



# BST

# Dictionaries

- A Dictionary is a collection of pairs has a key and an associated item.
- Assume that no two pairs have the same key.

ADT *Dictionary* is
**object:** a collection of $n$ pairs, each pair has a key and an associated item
**function:** for all $d \in Dictionary,\ item \in Item,\ k \in Key,\ n \in$ integer

| | | |
|---|---|---|
| *Dictionary* create(max_size) | ::= | create an empty dictionary |
| Boolean isEmpty($d, n$) | ::= | **if** ($n > 0$) **return** *TRUE* |
| | | **else return** *FALSE* |
| Element search($d, k$) | ::= | **return** item with key $k$, |
| | | **return** NULL if no such element |
| Element delete($d, k$) | ::= | delete and return item (it any) with key $k$ |
| *void* insert ($d$, *item*, $k$) | ::= | insert *item* with key $k$ into $d$ |

# An Example for Binary Search

Find 45 in $\{23, 78, 45, 8, 32, 56\}$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 8 | 23 | 32 | 45 | 56 | 78 |
| 8 | 23 | 32 | 45 | 56 | 78 |
| 8 | 23 | 32 | 45 | 56 | 78 |

# Binary Search Trees

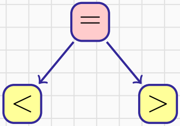
- A binary search tree has a good performance for dictionaries

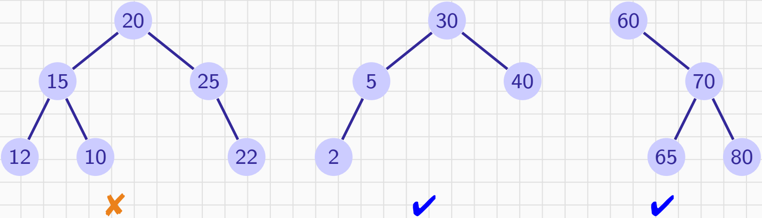
**Definition:**

A **binary search tree** is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

1. Each node has exactly one key and the keys in the tree are distinct.
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are greater than the key in the root.
4. The left and right subtrees are also binary search trees.

# Examples of Binary Search Trees



# Searching a Binary Search Tree

```
element* search(treePointer root, int k)
{  if (!root) return NULL;
   if (k == root->data.key) return &(root->data);
   if (k < root->data.key) return search(root->leftChild, k);
   return search(root->rightChild, k);
}
```

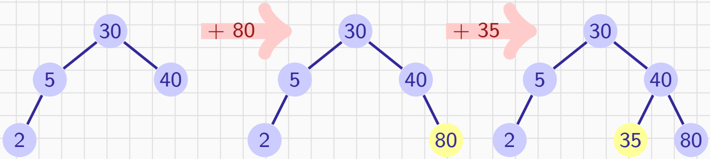
```
element* iterSearch(treePointer tree, int k)
{  while (tree) {
        if (k == tree->data.key) return &(tree->data);
        if (k < tree->data.key)
            tree = tree->leftChild;
        else
            tree = tree->rightChild;
   }
}
```

$O(h)$

# Insertion into a BST

```
void insert(treePointer *node, int k, iType theItem)
{  treePointer ptr, temp = modifiedSearch(*node, k);
   if (temp || !(*node)) { /* k is not in the tree */
      MALLOC(ptr, sizeof(*ptr));
      ptr->data.key = k;
      ptr->data.item = theItem;
      ptr->leftChild = ptr->rightChild = NULL;
      if (*node)   /* as temp's child */
         if(k < temp->data.key) temp->leftChild = ptr;
         else temp->rightChild = ptr;
      else *node = ptr;
   }
}
```

$O(h)$



# Deletion from a BST

- Three cases should be considered

  **case 1.** leaf

  ➠ delete

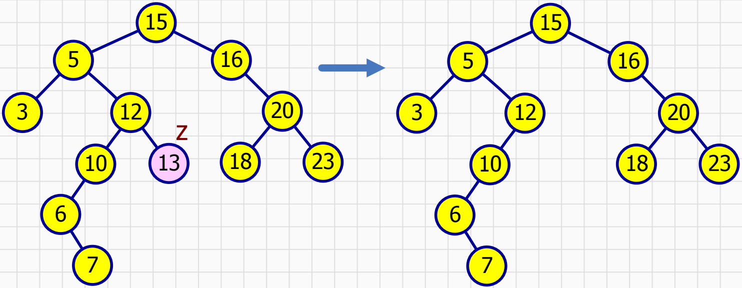
  **case 2.** one child

  ➠ delete and change the pointer to this child

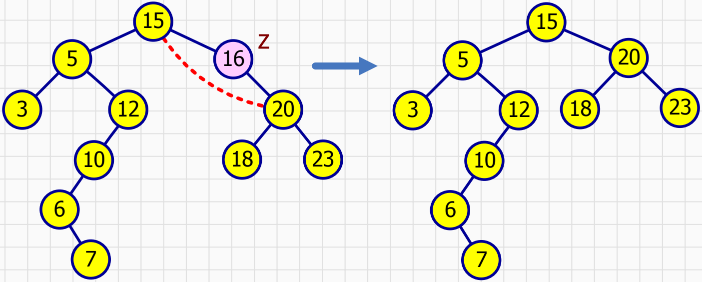
  **case 3.** two children

  1. **the smallest element in the right subtree** or
  2. **the largest element in the left subtree**

# Deleting a Node - with No Children

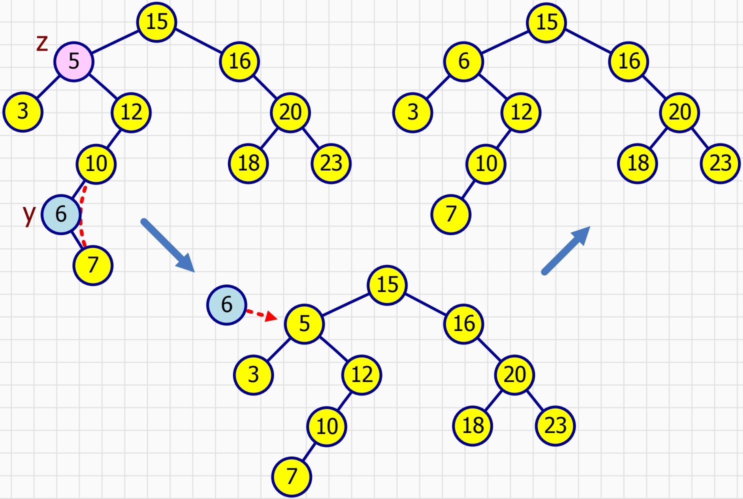


# Deleting a Node - with Only One Child



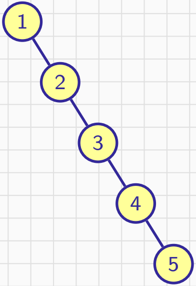

**inorder:** $3, 5, 6, 7, 10, 12, 15, 18, 20, 23$

# Deleting a Node - with Two Children



## Height of a BST

- The height of a binary search tree with $n$ elements can become as large as $n$ (in the worst case).
  ➠ Insert the keys $\{1, 2, 3, \ldots, n\}$
- It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log n)$ on the average.
- Search trees with a worst-case height of $O(\log n)$ are called **balanced search trees**.
  ➠ AVL, 2-3, $B$, $B^+$, and red-black trees

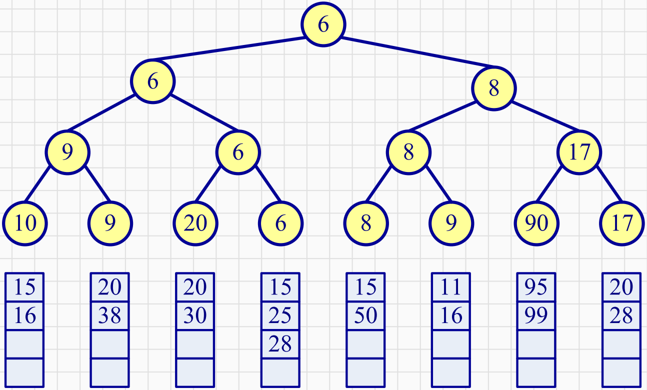


# Selection Trees

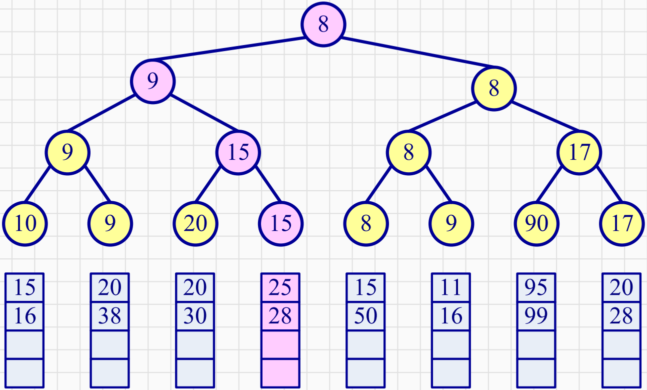
# Selection Trees

- Problem:
  Suppose we have $k$ ordered sequences, called runs, that are to be merged into a single ordered sequence.
- Solution :
  1. straightforward : $k-1$ comparisons for a number
  2. selection tree : $\lceil \log_2(k+1) \rceil$ for a number
- A selection tree is a binary tree where each node represents the smaller of its two children.
- There are two kinds of selection trees:
  1. winner trees
  2. loser trees

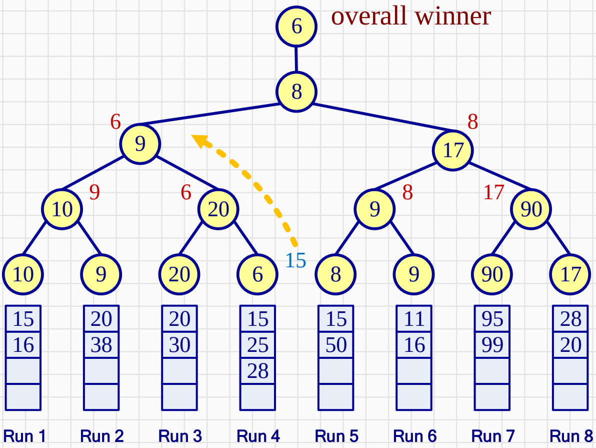
# Winner Trees (1/2)



# Winner Trees (2/2)

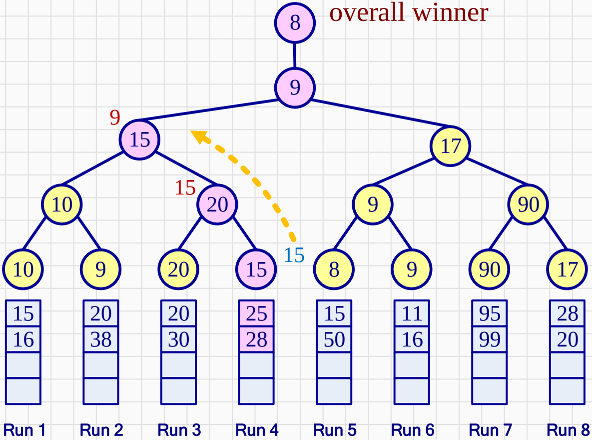


# Loser Trees (1/2)




Each match node stores the match loser (not the winner).

# Loser Trees (2/2)



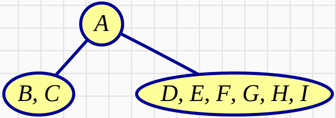

Don't access the sibling node.

# Analysis of Winner Trees

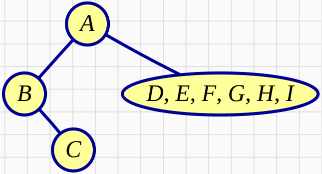
- The time required to re-construct the tree is $O(\log k)$.
- The time required to merge all $n$ records is $O(n \log k)$.
- The total time: $O(n \log k)$

# Construction of a Tree

1 preorder: A B C D E F G H I
inorder: B C A E D G H F I




2 preorder: A B C (D E F G H I)
inorder: B C A (E D G H F I)




3 preorder: A B C (D E F G H I)
inorder: B C A (E D G H F I)