Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Trees: Fundamental

Last updated: **02/03/2011**

Tang Wai Chung, Matthew

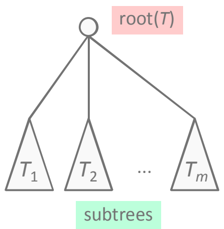
Definition of Tree

- The tree structure means a **branching** relationship between nodes.

> ***Tree***
> A finite set T of one or more nodes such that
> (a) there is one specially designed node called the **root** of the tree, root(T)
> (b) The remaining nodes are partitioned into $m \geq 0$ disjoint sets T_1, ..., T_m and each of these is a tree. The trees T_i are called ***subtrees***.

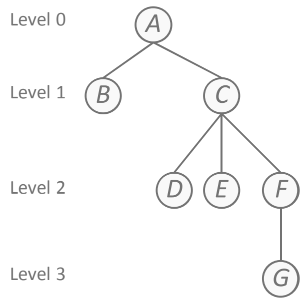
*The definition is clearly **recursive**.*
*Recursion is an **intrinsic** characteristics of tree structures.*

Tree Terminologies



Terminology	Meaning
Degree	the number of subtrees of a node
Terminal node ***leaf***	nodes of degree 0
Non-terminal node ***branch*** node	nodes of non-zero degree
Level	The root has level 0. All other nodes have a one level higher w.r.t. the subtree of the root that contains them.

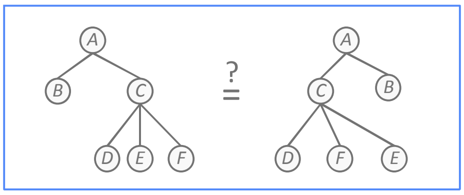
Tree Examples



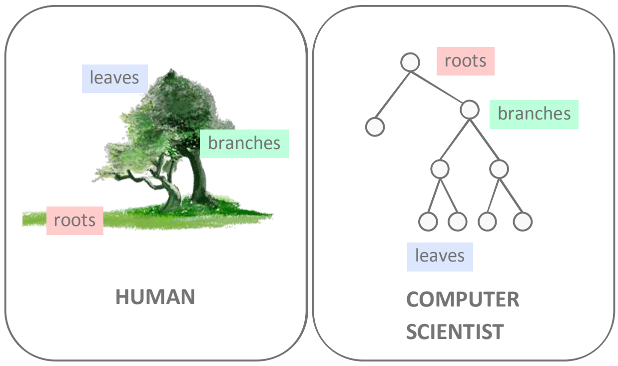
root: A
2 subtrees: {B} and {C, D, E, F, G}
The tree {C, D, E, F, G} has node C as its root.
Node C is on level 1 with respect to the whole tree
Terminal nodes: B, D, E and G
F is the only node with degree 1.
G is the only node with level 3.

Ordered Trees vs Oriented Trees

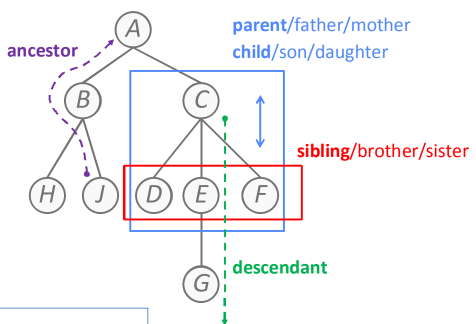
- If the relative order of the subtrees is important, the tree is said to be an **ordered** tree.
- If the ordering is not regarded, the tree is said to be **oriented** since only relative orientation is considered.
- Unless it is explicitly stated otherwise, all trees in our discussions are ***ordered***.



Trees from Different Perspectives



Family-based Tree Terminologies



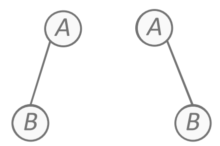
> *Drill:*
> *If a node A has three brothers and B is the father of A, what is degree of B?*

Forests and Binary Trees

> ***Forest***
> A (ordered) set of zero or more disjoint trees.

> ***Binary Tree***
> A finite set of nodes which either is
> **(i)** **empty**, or
> (ii) consists of a root and **two** disjoint binary trees called **left** and **right** subtree of the root.

Two different binary trees



*Note: binary tree is **NOT** a special case of tree.*

Implementation of Binary Trees

- Because a binary tree has exactly two subtrees. We can keep direct pointers(links) to them.
- A node is implemented as a structure consisting of (i) key information (data); (ii) left link; and (iii) right link
- When the subtree is empty, the link points to **NULL**.

```
typedef struct tree_s tree_t;
struct tree_s {
    int e;
    tree_t *l;
    tree_t *r;
};
```


Traversing Binary Trees

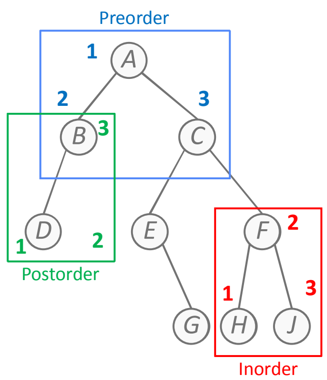
- **Traversal**:
  the algorithms for **walking through** a tree.
  - These are methods to examine the nodes of the tree **systemically** so that each node is visited **exactly once**.

Preorder Traversal
Visit the root
Traverse the left subtree
Traverse the right subtree

Postorder Traversal
Traverse the left subtree
Traverse the right subtree
Visit the root

Inorder Traversal
Traverse the left subtree
Visit the root
Traverse the right subtree

Examples of Traversal



Preorder:
A B D C E G F H J
Inorder:
D B A E G C H F J
Postorder:
D B G E H J F C A

Traversal: Code

```
void tree_preorder(tree_t *t, void (*visit)(tree_t *)){
    if (!t) return;

    visit(t);
    tree_preorder(t->l, visit);
     tree_preorder(t->r, visit);
}
```

```
void visit(tree_t *t){
    printf("%c ", t->e + 'A');
}
```

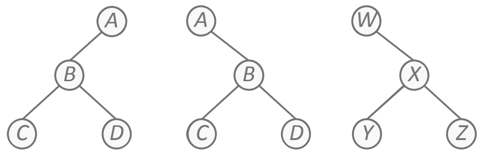
```
void tree_inorder(tree_t *t, void (*visit)(tree_t *)){
    if (!t) return;

    tree_inorder(t->l, visit);
    visit(t);
    tree_inorder(t->r, visit);
}
```

```
void tree_postorder(tree_t *t, void (*visit)(tree_t *)){
    if (!t) return;

    tree_postorder(t->l, visit);
    tree_postorder(t->r, visit);
    visit(t);
}
```


Similarity of Binary Trees



Similarity problem:
How to tell if two binary trees *T* and *T'* are of same shape?

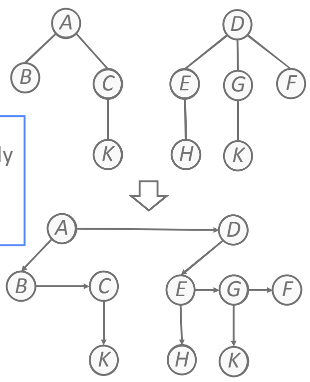
Binary Tree Similarity
Let the preorder of the nodes of binary trees *T* and *T'* be
$$u_1, u_2, \cdots, u_n \text{ and } u'_1, u'_2, \cdots, u'_n$$
respectively. The trees are similar if and only if $n = n'$, and
(i) both left subtrees of u_j and u'_j are non-empty,
(ii) both right subtrees of u_j and u'_j are non-empty, for all $1 \le j \le n$

Binary Tree Representation of Trees

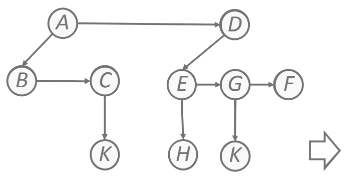
- Recall the basic **difference** between trees and binary trees
  - A tree is **never** empty, i.e. it always has at least one node; and each node can have 0, 1, 2, ... children.
  - A binary tree can be empty, and each of its nodes can have 0, 1, or 2 children; we **distinguish** between a left child and a right child.
- Recall also a forest is an ordered set of zero or more trees.

Forest Represented as Binary Trees

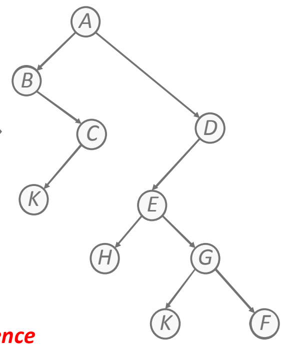
Link together the children of each family and remove vertical links except from father to first son.



Forest Represented as Binary Trees



Tile the diagram 45°
We get a binary tree!



*This is called the **natural correspondence** between forests and binary trees.*

Natural Correspondence: Formal

Let $F = (T_1, T_2, \ldots, T_N)$ be a forest of trees.

The binary tree $B(F)$ corresponding to F can be defined as follows:

(a) If $N = 0$, $B(F)$ is empty.

(b) If $N > 0$, the root of $B(F)$ is root(T_1);
the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \ldots, T_{1M})$ (subtrees of T_1);
and the right subtree of $B(F)$ is $B(T_2, \ldots, T_N)$

Mathematical Properties of Binary Trees

***Property* 06.1**
A binary tree with N internal nodes has $N + 1$ external nodes.

Prove by induction. The property holds for $N = 0$.

For $N > 0$, any binary tree has K internal nodes in the left subtree and $N - 1 - K$ in the right subtree. By hypothesis, there are $K + 1$ and $N - K$ external nodes in the left and right subtree respectively.
This adds to a total of $N + 1$ external nodes.

***Property* 06.2**
A binary tree with N internal nodes has $2N$ links: $N - 1$ to internal nodes and $(N + 1)$ to external nodes.

Follows from 06.1:

Every internal nodes have 1 link to the parents (except root). Similarly for every external nodes have 1 link to parent.

Height and Path Lengths

Terminology	Definition
Height	The maximum of the levels of all the nodes
Path length	The sum of the levels of all nodes
Internal/external path length	The sum of the levels of all internal/external nodes

***Property* 06.3**
The external path length of any binary tree with N internal nodes is $2N$ greater than the internal path length.

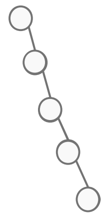
Tree construction: start with a single external node.

Repeat: replace an external node with an internal node connected to 2 new external nodes.

If the addition is done at level K, the internal path length is increased by K while the external one is increased by $K + 2$. After N steps, the difference in the lengths is $2N$.

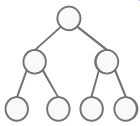
Heights of Binary Trees

***Property* 06.4**
The height of a binary tree with N internal nodes is at least $(\lg N)$ and at most $N - 1$.



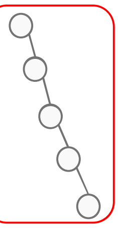
Worse Case: *Degenerate tree*
$N - 1$ links from the root to the leaf

Best Case: *Balanced tree*
There are $2i$ internal nodes at every level i
If height is H, then
$2^{H-1} \le N + 1 \le 2^H$

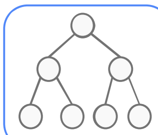


Bounds for Internal Path Length

***Property* 06.5**
The internal path length of a binary tree with N internal nodes is at least $N \lg (N / 4)$ and at most $N(N - 1)/2$.



internal path length
$= 0 + 1 + 2 + \ldots + (N - 1)$
$= N(N - 1)/2$



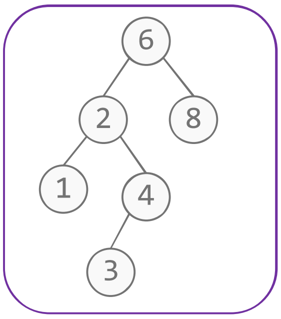
$N + 1$ externals nodes at height $\le$ floor(lg N)
internal path length $= (N + 1)\lg N - 2N > N \lg (N / 4)$

Binary Search Trees

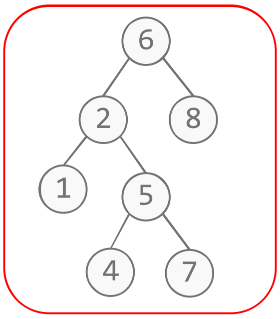
- An important application of binary trees is their use in **searching**.
- Each node in the binary tree is assigned a key value.
  - For simplicity, we assume keys are integers.
- We also assume the keys are **distinct**.

Binary Search Tree (BST)
For every node with key K, the values of all the keys in its **left** subtree are **smaller** than K, and the values of all the keys in its **right** subtree are **larger** than K.

Binary Search Trees: Example



A Binary search Tree



Not a binary search Tree

Binary Search Tree: ADT

Prototype	Operation
make_empty `bst_t  *bst_make_empty(bst_t *t)`	free all nodes in the tree and return the pointer to the emptied tree.
find `bst_t  *bst_find(bst_t *t, int x)`	Search for key x in the tree. Return the pointer to the node, or NULL when x is not found.
find_min / *find_max* `bst_t *bst_find_min/max(bst_t *t)`	Find the minimum/maximum key stored among all nodes in the tree.
insert `bst_t *bst_insert(bst_t *t, int x)`	Insert a new key x into the tree. Return a pointer to the modified tree.
delete `bst_t *bst_delete(bst_t *t, int x)`	Delete the node storing the key x. Return a pointer to the modified tree.

Binary Search Tree: Implm (1)

An empty tree is represented by **NULL** pointers.

```
typedef struct bst_s bst_t;
struct bst_s {
      int e;
      bst_t *l;
      bst_t *r;
};
```

Postorder: recursively free all allocated nodes.

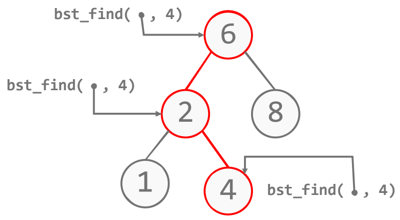
```
bst_t *bst_make_empty(bst_t *t){
    if (t != NULL){
        bst_make_empty(t->l);
        bst_make_empty(t->r);
        free(t);
    }
    return NULL;
}
```


Binary Search Trees: *find*

- ***find*** returns a pointer to the node in root(*T*) that stores a key equal to *x*, or **NULL** if there is no such node.
- If *T* is empty, return **NULL**, otherwise if the key is stored at root(*T*) is *x*, return *T*.
- If not, we continue the search in one of subtrees of *T* based on the result of the comparison:
  - **left** subtree:
    if *x* is **smaller** than the value at root(*T*)
  - **right** subtree:
    if *x* is **larger** than the value at root(*T*).

Binary Search Trees: *find* (Code)

```
bst_t *bst_find(bst_t *t, int x){
    if (t == NULL)
        return NULL;
    if (x < t->e) /* smaller: visit left subtree */
        return bst_find(t->l, x);
    if (x > t->e) /* larger: visit right subtree */
        return bst_find(t->r, x);
    return t; /* match */
}
```



Finding the min. and max.

- ***find_min*** & ***find_max***:
  return the position (a pointer to a tree node) of the smallest and largest entries in the tree respectively.
- To make it consistent with ***find***, the **position** is returned instead of the value.
- ***find_min***: start at the root and go left as long as there is a left child.
  - The stopping point is the smallest entry.
- The ***find_max*** routine is very similar, except that we always branch to the **right** child.

Finding the min. and max.: Code

find_min: a recursive implementation

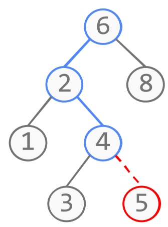
```
bst_t *bst_find_min(bst_t *t){
    if (t == NULL)
        return NULL;
    return (t->l == NULL ? t : bst_find_min(t->l));
}
```

find_max: a non-recursive implementation

```
bst_t *bst_find_max(bst_t *t){
    if (t != NULL)
        for ( ; t->r != NULL; t = t->r) ;
    return t;
}
```


Binary Search Trees: *insert*

- To insert a new key *x* into tree *T*, proceed down the tree as you would in ***find***.
- If *x* is found, do nothing (or update).
- Otherwise, insert *x* at the **last spot** on the path traversed.



Binary Search Trees: *insert* (2)

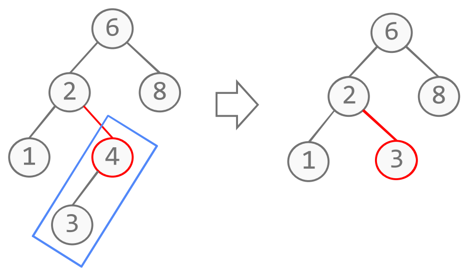
- Duplicates can be handled by keeping an **extra** field in the node indicating the **frequency** of occurrence.
- It returns a pointer to the root of the new tree. This is necessary to make the recursion correct.

```
bst_t *bst_insert(bst_t *t, int x){
    if (t == NULL){ /* create and add a one-node tree */
        t = (bst_t *)malloc(sizeof(bst_t));
        t->e = x;
        t->l = t->r = NULL;
    }
    else if (x < t->e) /* add in left subtree */
        t->l = bst_insert(t->l, x);
    else if (x > t->e) /* add in right subtree */
        t->r = bst_insert(t->r, x);

    return t; /* key x exists, do NOTHING */
}
```

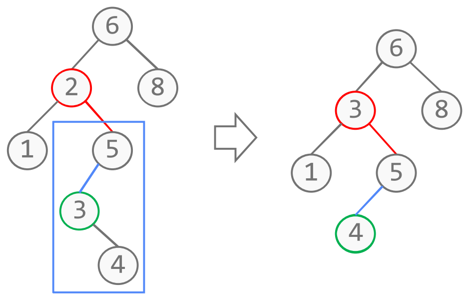

Binary Search Trees: *delete*

- ***delete*** is a bit tricky since we need to consider several possibilities:
  - If the node is **leaf**, it can be deleted immediately.
  - If the node has **one child**, the node can be deleted after its parent adjusts a pointer to bypass the node.



Binary Search Trees: *delete* (2)

- If the node has **two** children:
  replace the key of this node with the smallest key of the right subtree using ***find_min*** and recursively delete that node, which has no left child.



Binary Search Trees: Delete (Code)

```
bst_t *bst_delete(bst_t *t, int x){
    if (t == NULL)
        perror("Key not found!\n");
    else if (x < t->e) /* go left */
        t->l = bst_delete(t->l, x);
    else if (x > t->e) /* go right */
        t->r = bst_delete(t->r, x);
    else { /* key matches */
        if (t->l && t->r){ /* 2 children */
            /* replace smallest in right subtree */
            t->e = bst_find_min(t->r)->e;
            t->r = bst_delete(t->r, t->e);
         }
         else { /* 1 child or no child */
            bst_t *tmp = t;
            t = !t->l ? t->r : (!t->r ? t->l : 0);
            free(tmp);
         }
    }
    return t;
}
```

If the key is not found, print an error.

use find_min. Copy the key and delete that node in the right subtree.

adjust pointers and free the node

Average Height Analysis

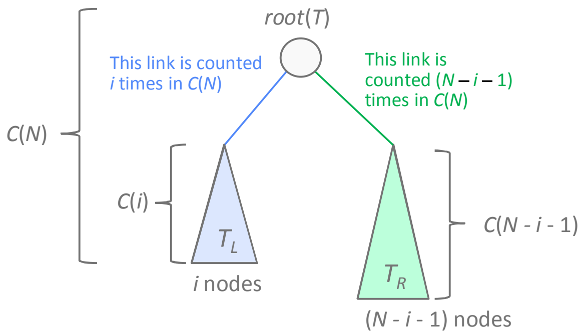
- The running times for ***find***, ***find_min***, ***find_max***, ***insert*** and ***delete*** are $O(H)$, where H is the height of the binary tree.
- We know that the height of a binary tree with N nodes is bounded between ($\lg N$) and ($N - 1$).
- What would be the **average** height over all nodes in a binary search tree?
- If we assume all trees are **equally likely**, then the average height is $O(\lg N)$.

Average Height Analysis (2)

- Let $C(N)$ be the **internal path length** for some tree T of N nodes. $C(1) = 0$
- An N-node binary tree consists of an i-node left subtree and an (N - i - 1)-node right subtree, plus a root at **depth 0 for $0 \le i < N$.**
  - $C(i)$ is the internal path length of the left subtree with respect to its root.
- In the main tree, all these nodes are one level **deeper**. The same holds for the right subtree.

$$C(N) = C(i) + C(N - i - 1) + N - 1$$

Average Height Analysis (3)



$C(N) = C(i) + C(N - i - 1) + N - 1$

Average Height Analysis (4)

- The subtree size of a BST depends on the **relative order** of the keys inserted.
- Let's assume all subtree sizes are **equally likely**
  - The average value of both $C(i)$ and $C(N - i - 1)$ is

$$\frac{1}{N}\sum_{i=0}^{N-1} C(i)$$

- And finally

$$C(N) = C(i) + C(N - i - 1) + N - 1$$
$$C(N) = \frac{2}{N}\left(\sum_{i=0}^{n-1} C(i)\right) + N - 1$$

Average Height Analysis (5)

- To solve this recurrence, we try to get rid of the summation sign.

$$NC(N) - (N-1)C(N-1) = 2C(N-1) + 2(N-1)$$
$$NC(N) \approx (N+1)C(N-1) + 2N$$

- Now we get a recurrence in $C(N)$ only. Divide the whole equation with $N(N + 1)$

$$\frac{C(N)}{N+1} = \frac{C(N-1)}{N} + \frac{2}{N+1}$$
$$\frac{C(N-1)}{N} = \frac{C(N-2)}{N-1} + \frac{2}{N}$$
$$\ldots$$

Average Height Analysis (6)

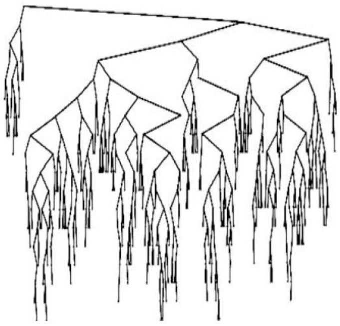
- Adding all equations together (the telescope method), we get:

$$\frac{C(N)}{N+1} = \frac{C(1)}{2} + 2\sum_{i=3}^{N+1}\frac{1}{i}$$
$$\frac{C(N)}{N+1} \approx \frac{T(1)}{2} + 2\left(\ln(N+1) + \gamma - (1 + \frac{1}{2})\right)$$
$$\frac{C(N)}{N+1} = O(\ln N) \Rightarrow C(N) = O(N \ln N)$$

Average height = $C(N)/N$ = $O(\ln N)$

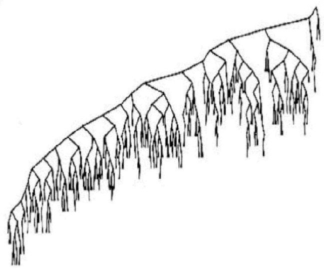
Normal Binary Search Tree

A randomly generated 500-node tree



Is our Assumption Valid?

- ***delete*** favours the **left** subtree.
- After many ***insert*** and ***delete***, we end up with an **unbalanced** binary tree.



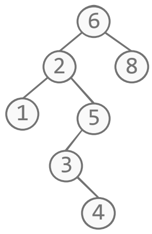
Tree Insertion Sorting

- Construct a binary search tree for all keys
- An inorder traversal will visit the elements in sorted order.

6	2	8	5	1	3	4

inorder traversal

1	2	3	4	5	6	8



Balanced Trees?

- **Balance** of a binary search tree is important to ensure that the tree does not degenerate into an unbalanced tree.
- **Maintenance** of tree balance often requires changes to the structure of the tree
  - Take longer on average for insertion/deletion.
  - Examples: **AVL** trees
- Another solution: **give up** the balance condition and allow the tree to be arbitrarily deep.
  - A tree restructure is carried out occasionally to make future operations efficient. (**self-adjusting**)

Summary

- **Trees**: Definitions, structures and basic properties
- Natural correspondence of forest and binary tree
- Binary Search Tree:
  - Store **key** at every node
  - **Left** subtree contains keys that are smaller than that stored at root.
  - **Right** subtree contains keys that are greater than that stores at root.
- ***find***, ***insert***, ***deletion*** in $O(H)$
- Average-case bound for height H_{avg} of BST → $O(\log N)$
- Application: tree insertion sorting