CS 2213
Advanced Programming
Ch 13 – Trees
Basic definitions and Binary Search Tree (BST)

Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.

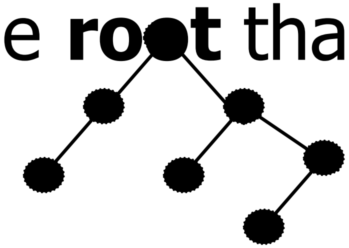
Objectives

- To understand the concept of **trees** and the standard terminology used to describe them.
- To appreciate the **recursive nature of a tree** and how that **recursive structure** is reflected in its underlying representation.
- To become familiar with the data structures and algorithms used to implement **binary search trees**.
- To recognize that it is possible to maintain **balance** in a binary search tree as new keys are inserted. (Part 2)
- To learn how binary search trees can be implemented as a general abstraction.

What is a tree?

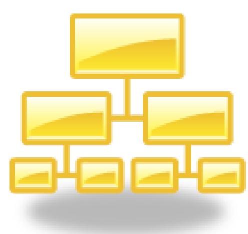
- A **tree** is defined to be a collection of individual entries called **nodes** for which the following properties hold:
  - As long as the tree contains any nodes at all, there is a specific node called the **root** that forms the top of a hierarchy.
  - Every other node is connected

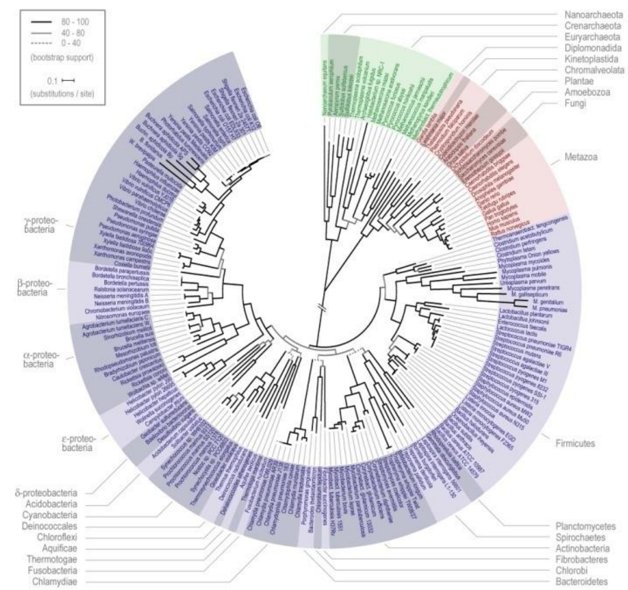
    to the root by a unique line of descent.



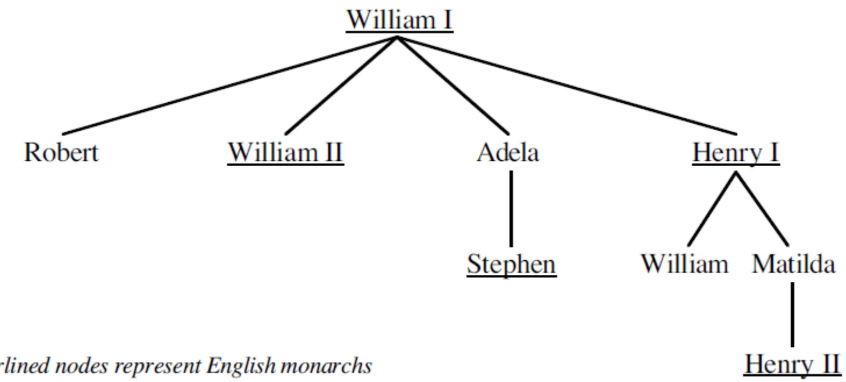
Trees Are Everywhere

- Tree-structured hierarchies occur in many contexts outside of computer science.
  - Game trees
  - Biological classifications
  - Organization charts
  - Directory hierarchies
  - **Family trees**
  - Many more…



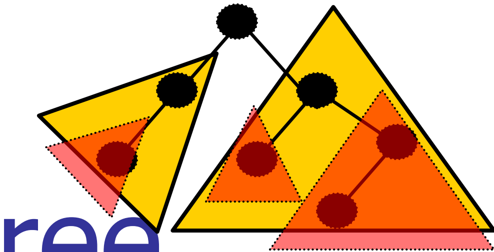


Tree Terminology

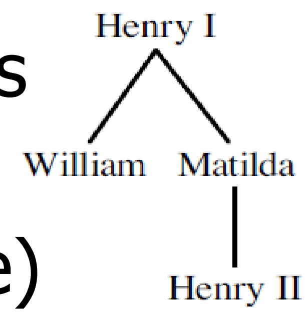


Underlined nodes represent English monarchs

- Most terms come from family tree analogue
  - William I is the **root** of the tree.
  - Adela is a **child** of William I and the **parent** of Stephen.
  - Robert, William II, Adela, and Henry I are **siblings**.
  - Henry II is a **descendant** of William I, Henry I, and Matilda
  - William I is an **ancestor** of everyone else.
- Other terms
  - Nodes that have no children are called **leaves**
  - Nodes that are neither the root nor a leaf are called **interior** nodes
- The **height/depth** of a tree is the length of the longest path from root to a leaf



Recursive nature of a tree



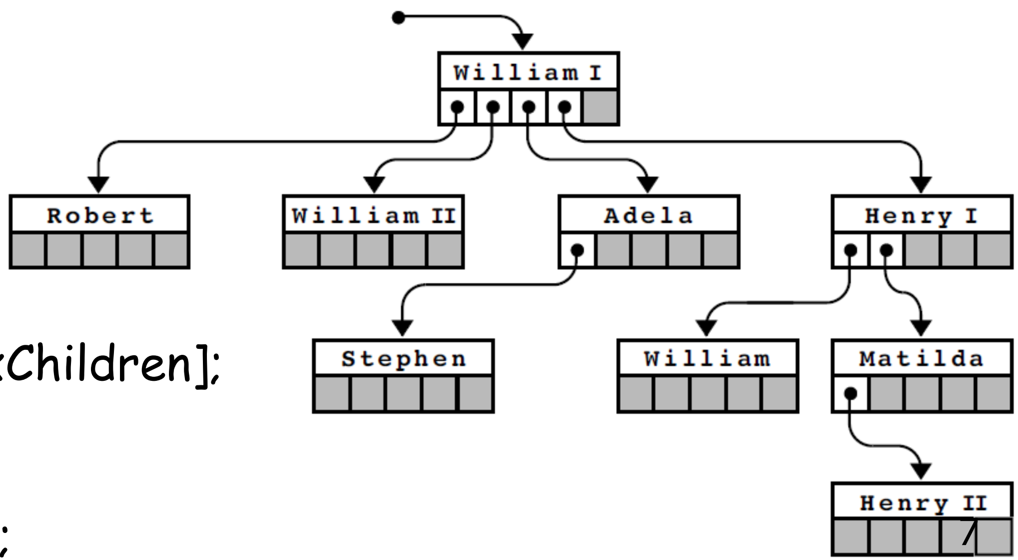
- Take any node in a tree together with all its descendants, the result is also a tree (called a **subtree** of the original one)
- Each node in a tree can be considered the root of its own subtree
- This is the **recursive nature** of tree structures.
  - A tree is simply **a node** and **a set of attached subtrees** — possibly empty set in the case of a leaf node—
  - The recursive character of trees is fundamental to their underlying representation as well as to most algorithms that operate on trees.

Representing family trees in C

- How can we represent the hierarchical (parent/children) relationships among the nodes
  - Include a **pointer** in the parent to point the child
  - A tree is a pointer to a node.
  - A node is a structure that contains some number of trees.

Use index values of an array (Heap)

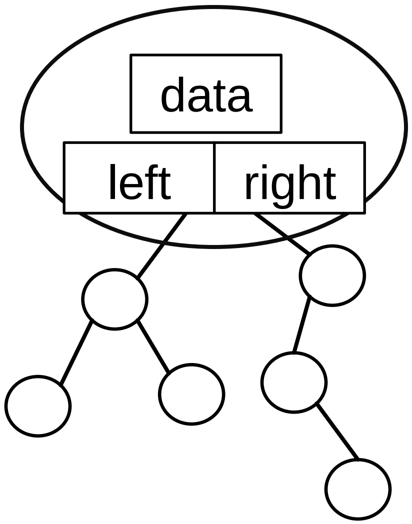
```
#define MaxChildren 5

typedef struct familyNodeT {
   string  name;
   struct  familyNodeT  *children[MaxChildren];
} familyNodeT;

typedef  familyNodeT  *familyTreeT;
```

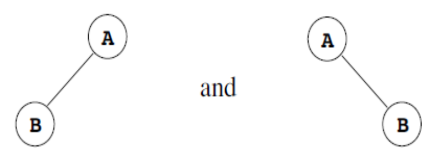


Binary Trees:

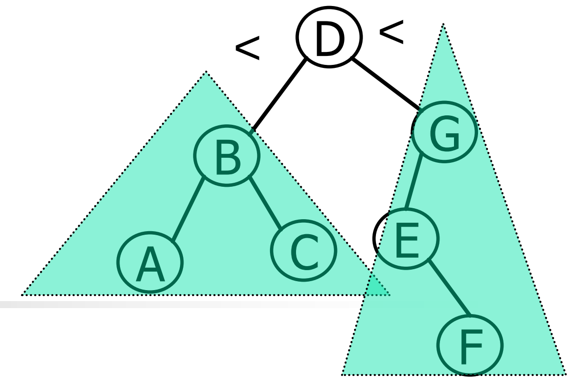
One of the most important subclasses of trees with many practical applications

- A **binary tree** is defined to be a **tree** in which the following additional properties hold:
  - Each node in the tree has at most two children.
  - Every node except the **root** is designated as either a *left child* or a *right child* of its parent.
- This geometrical relationship allows to represent ordered collections of data using binary trees (**binary search tree**)





Binary Search Trees



- A **binary search tree** is defined by the following properties:
  1. Every node contains—possibly in addition to other data—a special value called a ***key*** that defines the order of the nodes.
  2. Key values are ***unique,*** in the sense that no key can appear more than once in the tree.
  3. At every node in the tree, the key value must be
     - **greater than** all the keys in its **left** subtree
     - **less than** all the keys in its **right** subtree.

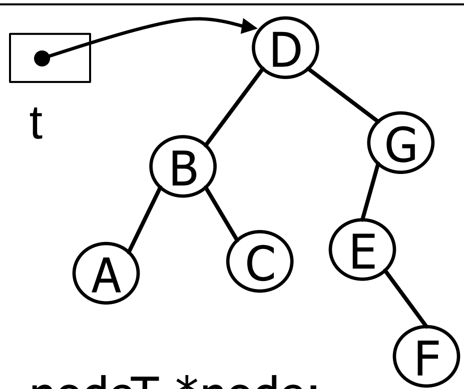
Motivation for Using Binary Search Trees

- Suppose we want to keep keys sorted
- What will be the complexity of lookup and insert if we use an Array
  - **Lookup** can be done in O(log N), how?
  - **Enter/Insert** will be in O(N)
- How about using Linked List
  - Lookup/Enter will be done in O(N). Why?
    - LL cannot find middle element efficiently *(skip list may help)*
- Can both Lookup and Enter be done in O(log N)?
  - Yes, by using Binary search trees

Finding nodes in a binary search tree: Recursive

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

```
nodeT *FindNode(nodeT *t, char key)
{
   if (t == NULL) return NULL;
   if (key == t->key) return t;
   if (key < t->key) {
     return FindNode(t->left, key);
   } else {
     return FindNode(t->right, key);
  }
}
```



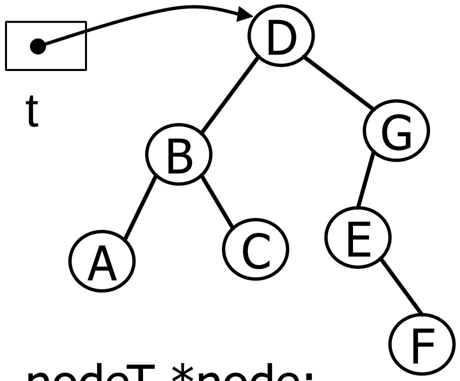

```
nodeT *node;
nodeT *t;
...
node=FindNode(t, 'F');
```

!!! Note !!!: Textbook uses **treeT.** Instead, I use **nodeT ***
Is there any difference?

Exercise: Iterative version of
Finding nodes in a binary search tree

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

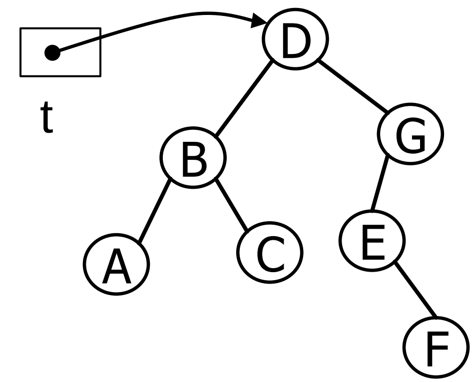
```
nodeT *FindNode(nodeT *t, char key)
{
    while(t !=NULL) {
      if (key == t->key) return t;
      if (key < t->key) {
          t = t->left;
      } else {
          t = t->right;
      }
    }
    return NULL;
}
```




```
nodeT *node;
nodeT *t;
…
node=FindNode(t, 'F');
```

Tree Traversal (inorder)

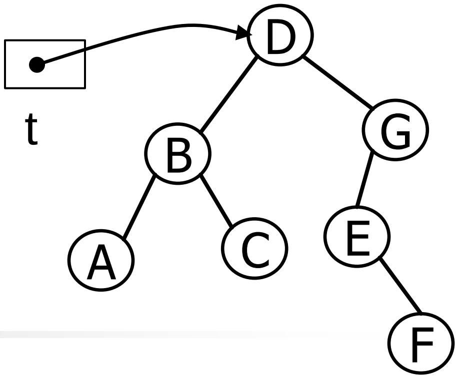
```
void DisplayTree(nodeT *t)
{
  if (t != NULL) {
    DisplayTree(t->left);
    printf(“%c “, t->key);
    DisplayTree(t->right);
  }
}
```




```
nodeT *node;
nodeT *t;
…
DisplayTree(t);
```

A B C **D** E F G

Preorder and Postorder Tree Traversal



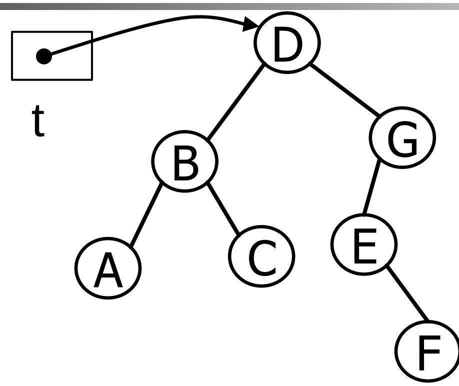

```
void PreOrderWalk(nodeT *t)
{
  if (t != NULL) {
    printf("%c ", t->key);
    DisplayTree(t->left);
    DisplayTree(t->right);
  }
}
```

```
void PostOrderWalk(nodeT *t)
{
  if (t != NULL) {
    DisplayTree(t->left);
    DisplayTree(t->right);
    printf("%c ", t->key);
  }
}
```

D B A C G E F

A C B F E G **D**

Exercise: Modify one of the traversal functions to print the tree as follow




```
D
+---B
|   +---A
|   +---C
+---G
|   +---E
|   |   +---F
```

```
void PrintTree(nodeT *t)
{    ModifyPreOrderWalk(t, 1);  }
void ModifyPreOrderWalk(nodeT *t, int h)
{
   int i;
   if (t == NULL)  return;
   for(i=0; i < h-1; i++) {
        printf(“|   “);
   }
   if (h>1) printf(“+---“);
   printf(“%c\n“, t->key);
   ModifyPreOrderWalk(t->left, h+1);
   ModifyPreOrderWalk(t->right, h+1);
 }
}
```

Exercise: Height

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

- Suppose we want to find the height of a Binary Search Tree t
- Write a function that returns the height of the tree
- Recursive Idea

  1 + Maximum of

    (height of left subtree, height of right subtree).

```
int height(nodeT *t)
{
  if (t == NULL)
      return 0;
  else
     return (1 + maximumof(
                    height(t->left),
                    height(t->right)) );
}
```

Exercise: Sum, Min, Max

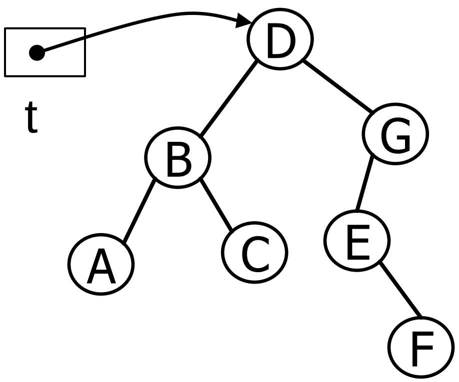
```
struct  tree_node {
   int data;
   struct tree_node *left, *right;
}
```

- Suppose we store integer values in a Binary Search Tree t
- Write a function that returns the sum of values in the tree
- Recursive Idea
  - Value of the current node
  - + the sum of values of all nodes of left subtree
  - + the sum of values of all nodes in right subtree.
- How about max, min in bst or just in a binary tree where values may not be sorted?

```
int add(struct tree_node *p)
{
  if (p == NULL)
      return 0;
  else
     return (p->data +
               add(p->left) +
               add(p->right) );
}
```

Exercise: Tree Traversal level order

```
void DisplayTreeLevelOrder(nodeT *t)
{

  if (t != NULL) {
   ??????

  }
}
```




```
nodeT *node;
nodeT *t;
...
DisplayTreeLevelOrder(t);
```

D B G A C E F

A little bit harder version

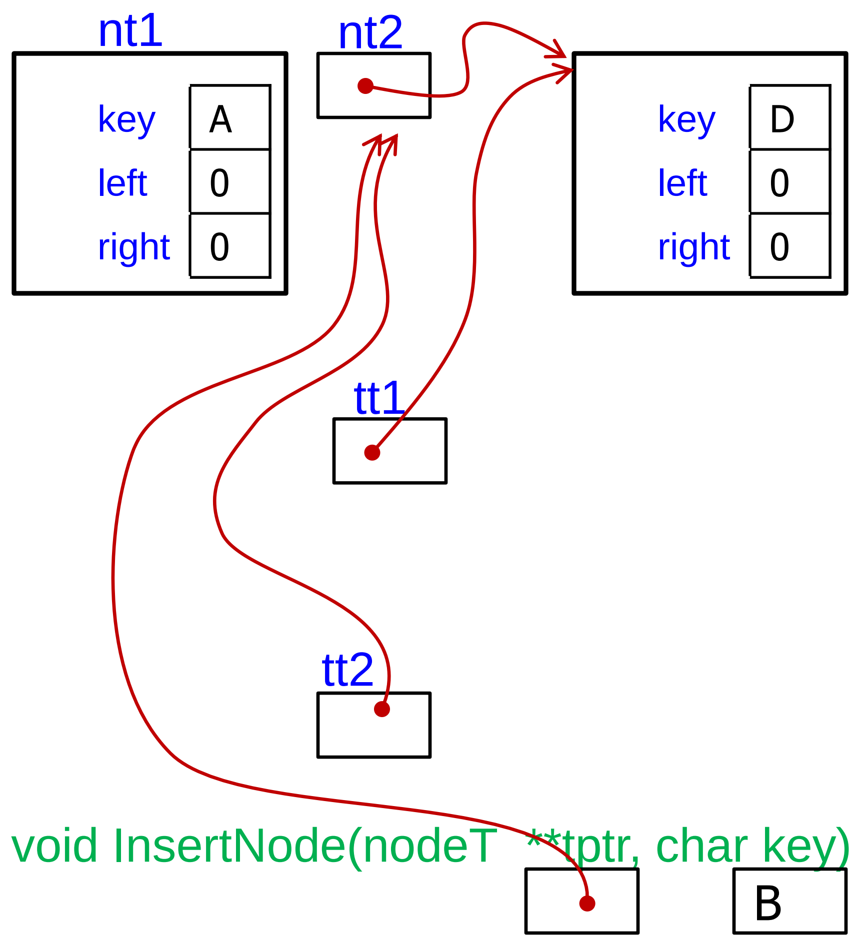
D
B G
A C E
F

Insert - Delete

Creating a tree in a client program

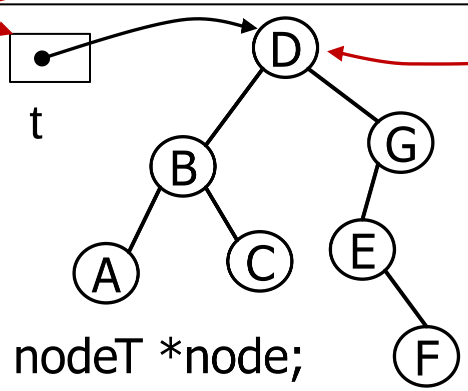
```c
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

```c
main()
{
  nodeT   nt1, *nt2;
  treeT    tt1,  *tt2;          // nodeT   *tt1, **tt2

  nt2 = (nodeT *) malloc(sizeof(nodeT));
  if(nt2==NULL) {
    printf(″no memory″);
    exit(-1);
  }                  // nt2 = New(nodeT   *) ;
  nt1.key = ‘A’; nt1.left = nt1.right= NULL;
  nt2->key = ‘D’; nt2->left = nt2->right= NULL;
  tt1 = nt2;
  tt2 = &nt2;
  InsertNode(&nt2, ‘B’); //  InsertNode(tt2, ‘B’);
}
```



Inserting new nodes in a binary search tree

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```




```
nodeT *node;
nodeT *t=NULL;
InsertNode(&t, 'D');
InsertNode(&t, 'B');
...
```

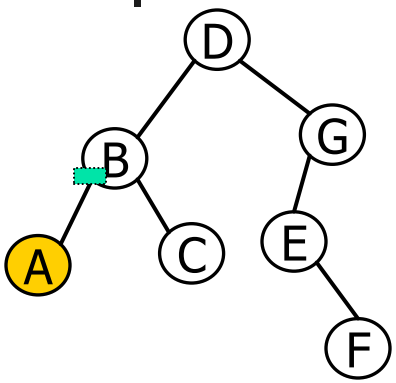
```
void InsertNode(nodeT  **tptr, char key)
{
    nodeT  *tmp;
    tmp=*tptr;
    if (tmp == NULL) {
        tmp=New(nodeT *);
        tmp->key = key;
        tmp->left=tmp->right=NULL;
        *tptr=tmp;
        return;
    }
    if (key < tmp->key) {
            InsertNode(&tmp->left, key);
    } else {
            InsertNode(&tmp->right, key);
    }
}
```

Exercise: modify this such that each node points parent node

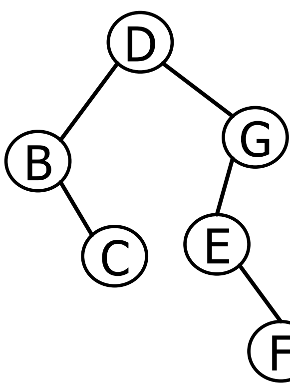
```
typedef struct nodeT {
    char key;
    struct nodeT *parent;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

```
void InsertNode(nodeT  **tptr, char key)
{
    nodeT  *tmp;
    tmp=*tptr;
    if (tmp == NULL) {
        tmp=New(nodeT *);
        tmp->key = key;
        tmp->left=tmp->right=NULL;
        *tptr=tmp;
        return;
    }
    if (key < tmp->key) {
            InsertNode(&tmp->left, key);
    } else {
            InsertNode(&tmp->right, key);
    }
}
```

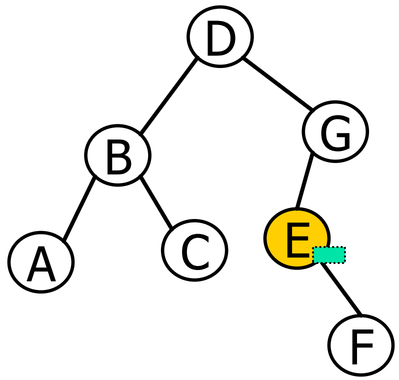
Deleting nodes from a binary search tree



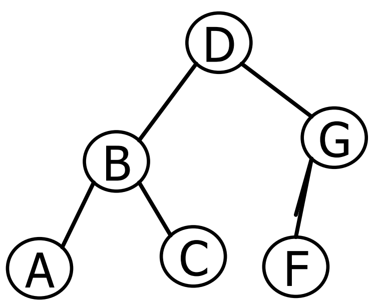
Delete A



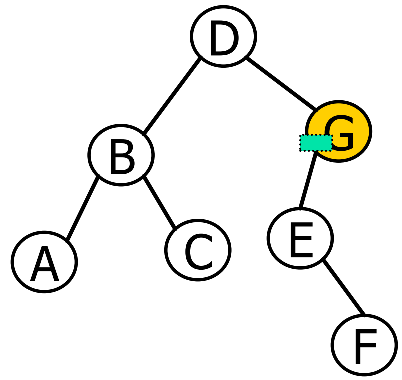
target->left==NULL &&
target->right==NULL



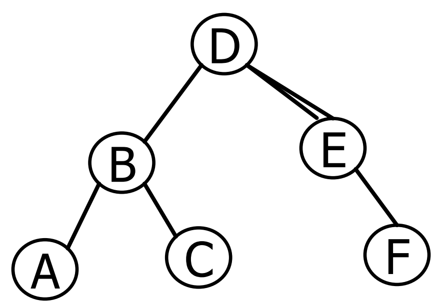
Delete E



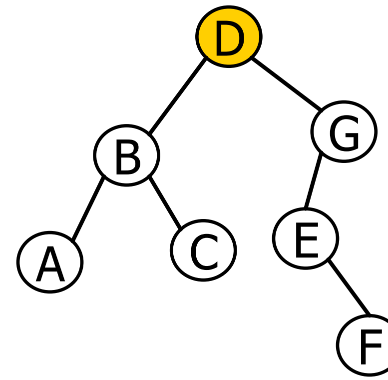
target->left == NULL



Delete G



target->right == NULL

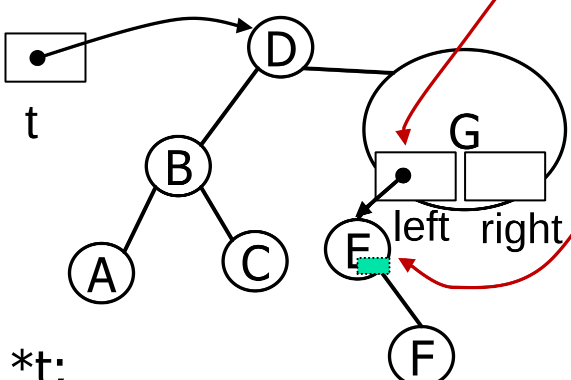


Delete D

???

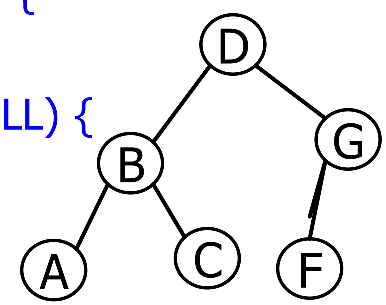
Deleting nodes from a binary search tree: easy cases

```
typedef struct nodeT {
    char key;
    struct nodeT *left, *right;
} nodeT, *treeT;
```

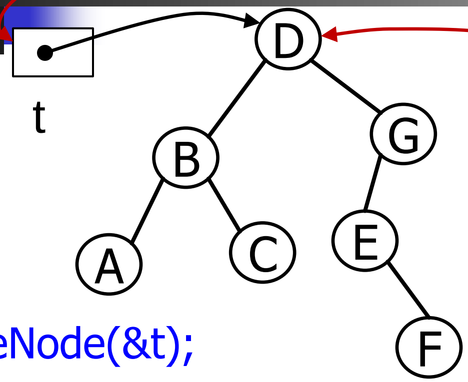



```
nodeT *t;
// Suppose we want to delete 'E',
// find/determine pointer to target node
DeleteNode(&t->right->left);
// if 'A', DeleteNode(&t->left->left);
```

```
void DeleteNode(nodeT  **p)
{
  nodeT *target;
  target=*p;
  if (target->left==NULL && target->right==NULL) {
      *p=NULL;
  } else if (target->left == NULL) {
      *p=target->right;
  } else   if (target->right == NULL) {
      *p=target->left;
  } else  {

      /*   target has two children, see next slide */

  }
  free(target);
}
```

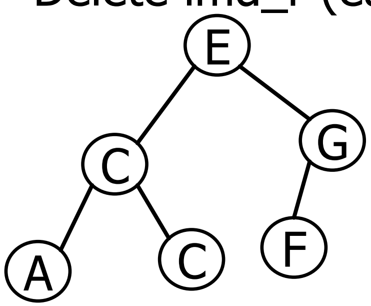


Deleting nodes from a binary search tree: two children



DeleteNode(&t);

1. Replace the target node with its **immediate successor,** which is the smallest value in the **right** subtree (lmd_r -- **l**eft**m**ost **d**escendant in **r**ight subtree)
2. Delete lmd_r (easy case, why?)

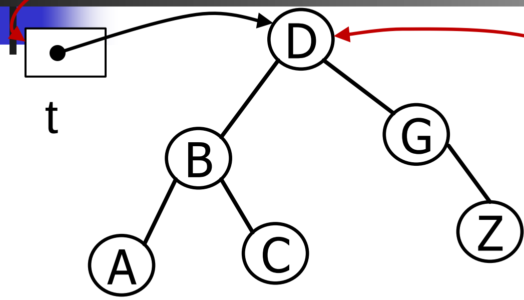


Is there any other case missing!

```
void DeleteNode(nodeT  **p)
{
  nodeT *target, *lmd_r, *plmd_r;
  target=*p;
  ... /* easy cases, see previous slide */
  } else {
    plmd_r = target;
    lmd_r = target->right;
    while( lmd_r->left != NULL){
       plmd_r = lmd_r;
       lmd_r = lmd_r->left;
    }
    plmd_r->left = lmd_r->right;
    lmd_r->left = target->left;
    lmd_r->right = target->right;
    *p = lmd_r;
  }
  free(target);
}
```

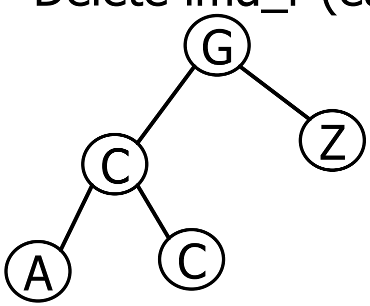
```
target->key =
    lmd_r->key;
target = lmd_r;
```

Deleting nodes from a binary search tree: two children (corrected)



DeleteNode(&t);

1. Replace the target node with its **immediate successor,** which is the smallest value in the **right** subtree (lmd_r -- **l**eft**m**ost **d**escendant in **r**ight subtree)
2. Delete lmd_r (easy case, why?)



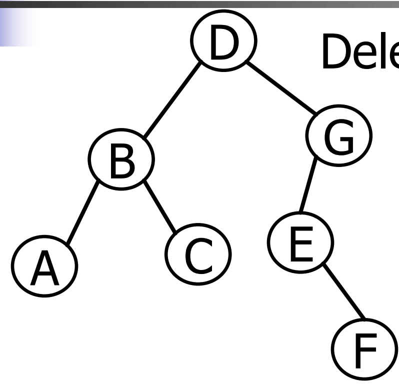
Can you think of another strategy (see the exercise in the next slide)

```
void DeleteNode(nodeT  **p)
{
   nodeT *target, *lmd_r, *plmd_r;
   target=*p;
   ... /* easy cases, see previous slide */
   } else {
     plmd_r = target;
     lmd_r = target->right;
     while( lmd_r->left != NULL){
        plmd_r = lmd_r;
        lmd_r = lmd_r->left;
     }
     if(plmd_r == target)
        plmd_r->right = lmd_r->right;
     else
        plmd_r->left = lmd_r->right;
     lmd_r->left = target->left;
     lmd_r->right = target->right;
     *p = lmd_r;
   }
   free(target);
}
```

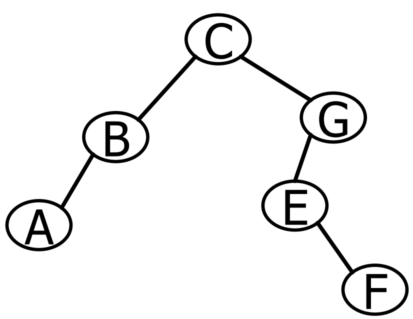
```
target->key =
    lmd_r->key;
target = lmd_r;
```

Exercise: **new strategy** for deleting nodes from a binary search tree: two children

Delete D



1. Replace the target node with its **immediate predecessor,** which is the largest value in the **left** subtree (rmd_l -- **r**ight**m**ost **d**escendant in **l**eft subtree)
2. Delete rmd_l (easy case, why?)




```
void DeleteNode(nodeT  **p) /* Modify this one*/
{
   nodeT *target, *lmd_r, *plmd_r;
   target=*p;
   ... /* easy cases see previous slide */
   } else {
      plmd_r = target;
      lmd_r = target->right;
      while( lmd_r->left != NULL){
          plmd_r = lmd_r;
          lmd_r = lmd_r->left;
      }
    if(plmd_r == target)
       plmd_r->right = lmd_r->right;
    else
       plmd_r->left = lmd_r->right;
      lmd_r->left = target->left;
      lmd_r->right = target->right;
      *p = lmd_r;
   }
```

```
target->key =
    lmd_r->key;
target = lmd_r;
```

Exercise: Deleting a node with two children

- Randomly select one of the previous strategies
- Which one will give better balanced tree

Final word: Importance of Recursion in Binary Trees

- It's very difficult to think about how to iteratively go through all the nodes in a binary tree unless you think **recursively**???
- With recursion, the code is reasonably concise and simple.
- But in some cases **iterative** approaches might be possible and more efficient