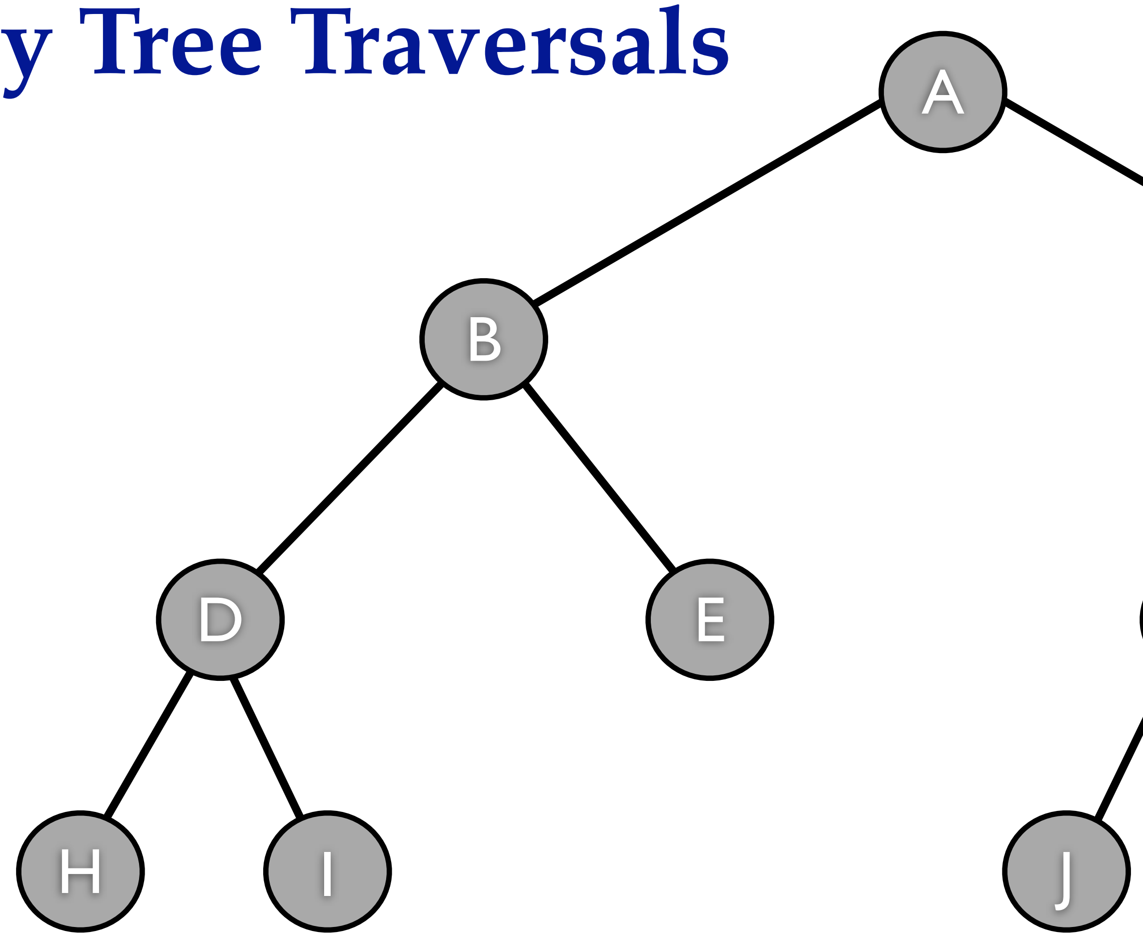
Binary Search Trees

CMSC 420: Lecture 6

Binary Tree Traversals



inorder:	HDIBEAJFCG
preorder:	ABEHIECFJG
postorder:	HIDEBJFGCA

```
void traverse(BinNode *T) {
  if(T != NULL) {
      PREORDER(T);
      traverse(T->left());
      INORDER(T);
      traverse(T->right());
      POSTORDER(T);
   }
}
```

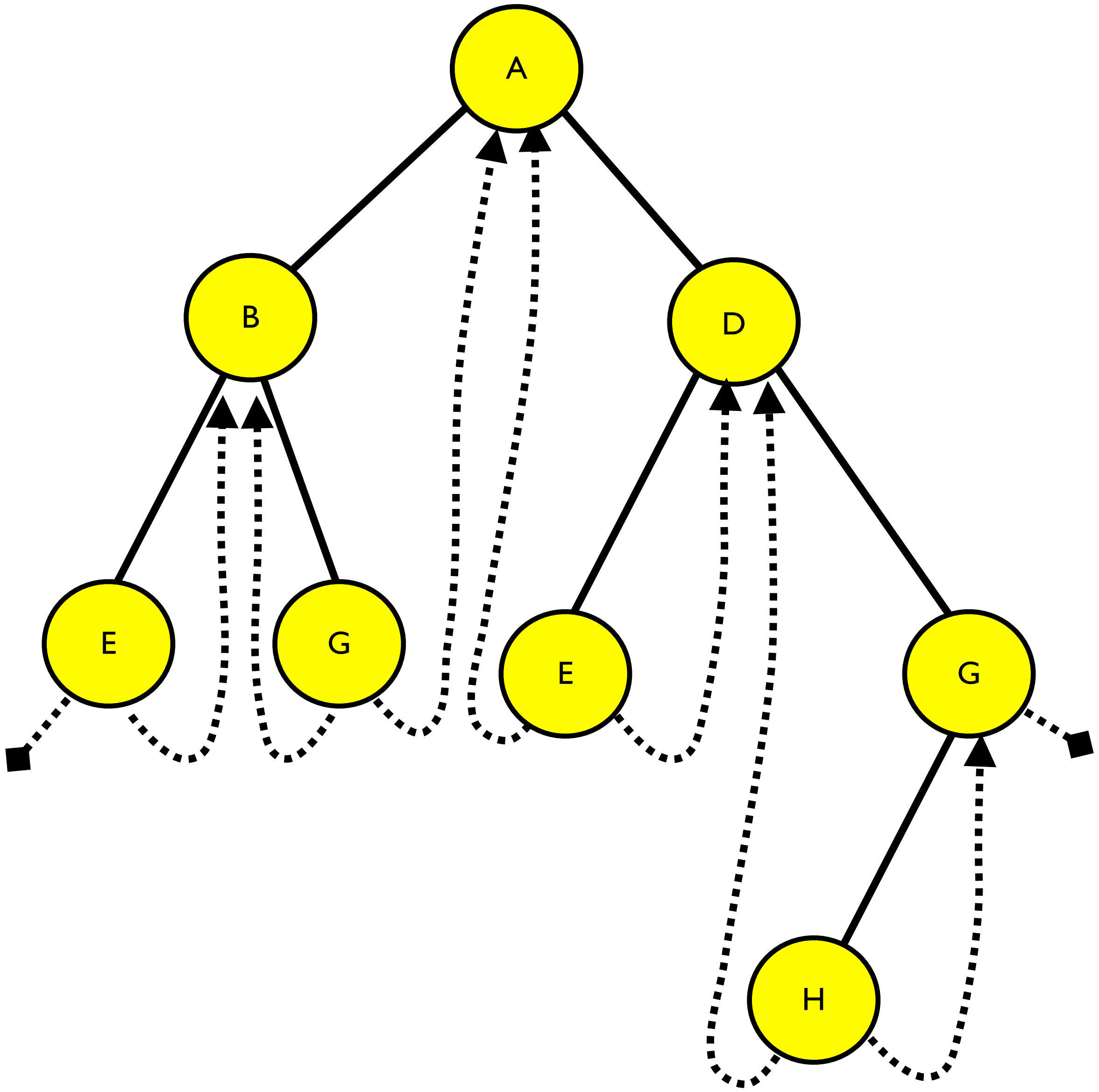
How much space is used?

Threaded Trees

- Traversals:
  - Require extra memory, and
  - Must be started from the root

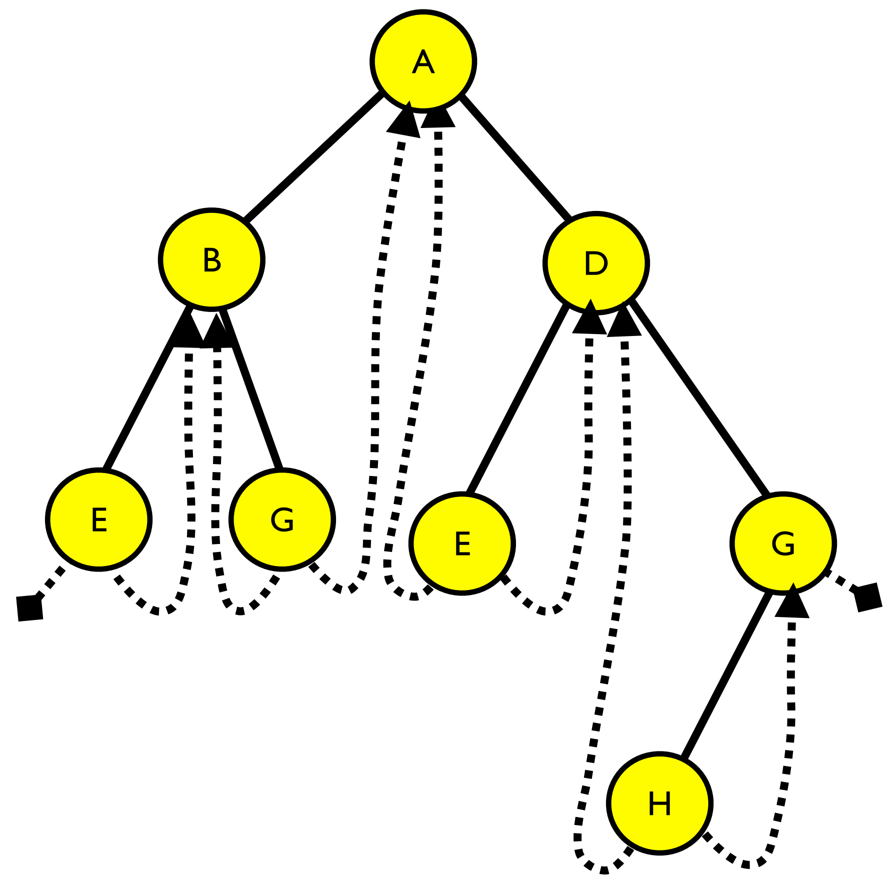
- Use NULL pointers to store in-order predecessors and successors.

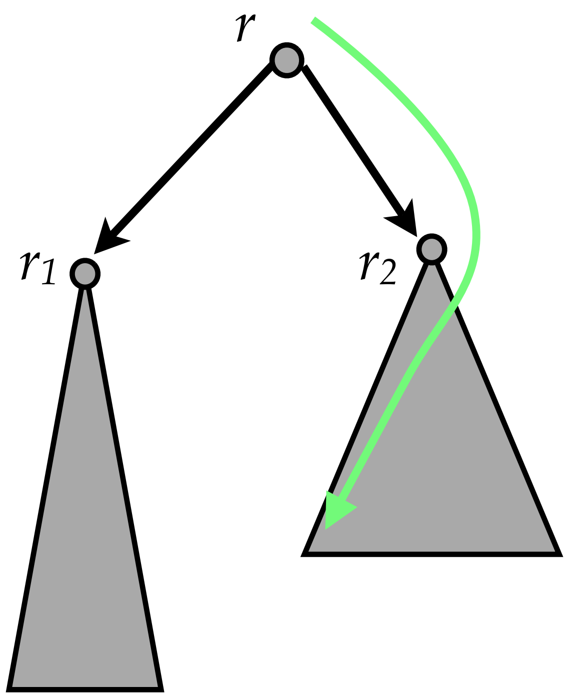
- Extra bit associated with each pointer marks whether it is a thread.



Threaded Trees

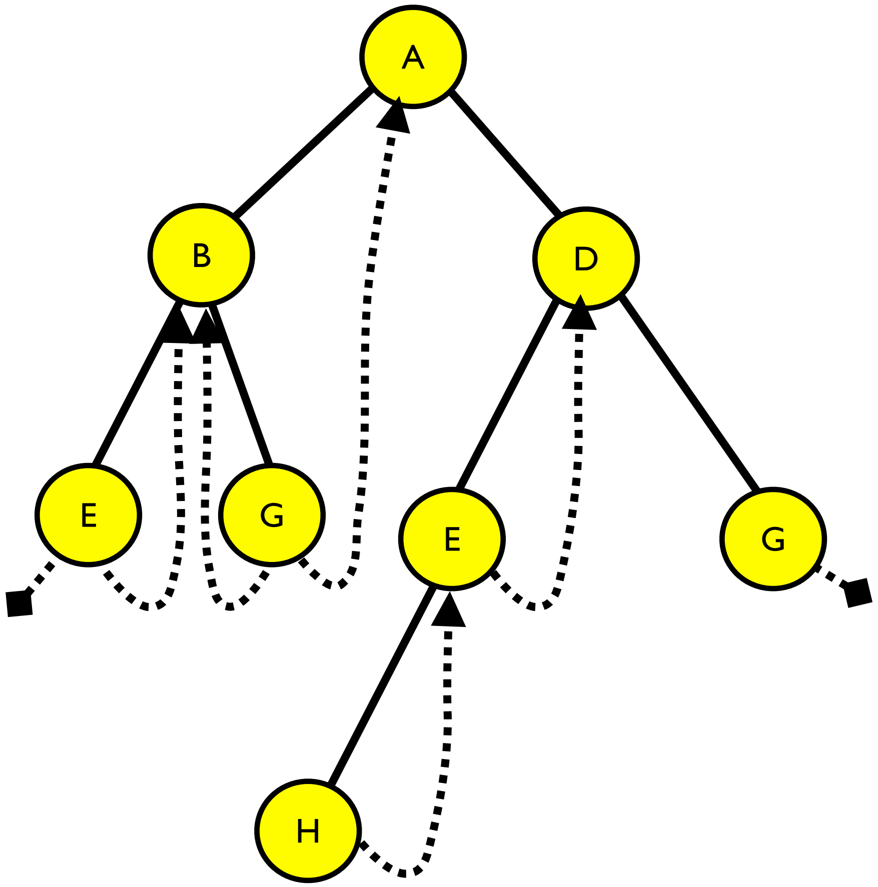
```
void inorder_succ(BinNode *T)
{
   BinNode * next = T->right();

   if(!next) return NULL;

   if(!is_thread(T->right))
   {
      while(next->left() &&
         is_thread(next->left)
      ) next = next->left();
   }

  return next;
}
```





In general, in order successor = *leftmost item in right subtree*

Using Threads for Preorder



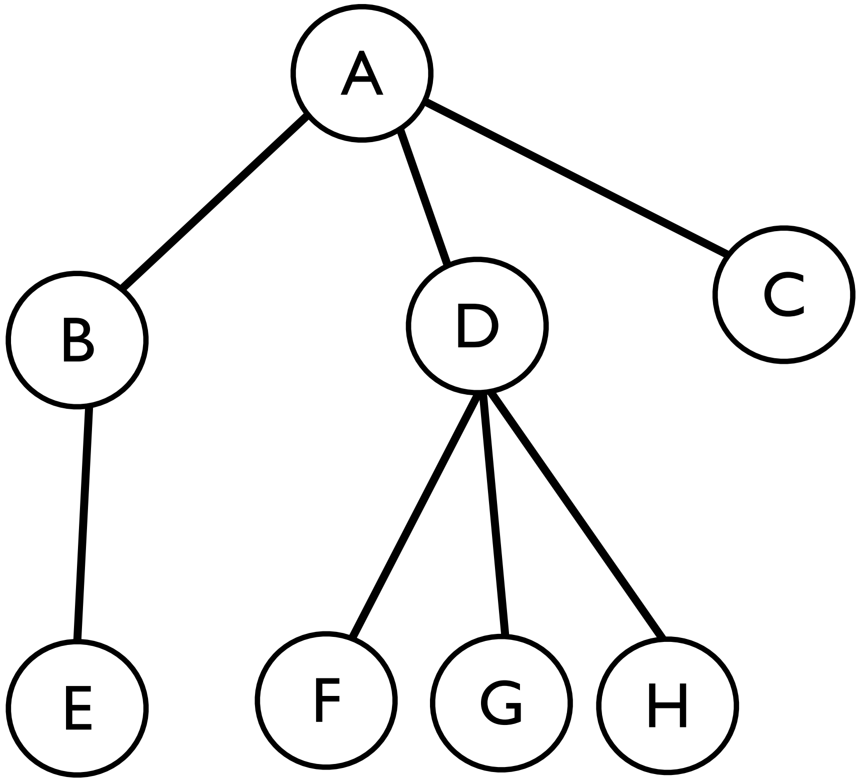
preorder_succ(H) = right child of the lowest ancestor of H that both has H in its left subtree and has a right child.

```
void preorder_succ(BinNode *T)
{
  if(T->left() &&
     !is_thread(T->left) return T->left();

  for(BinNode* next = T->right();
      is_thread(next->right);
      next = next->right()) {}

  return RC(P);
}
```

Walk up right threads

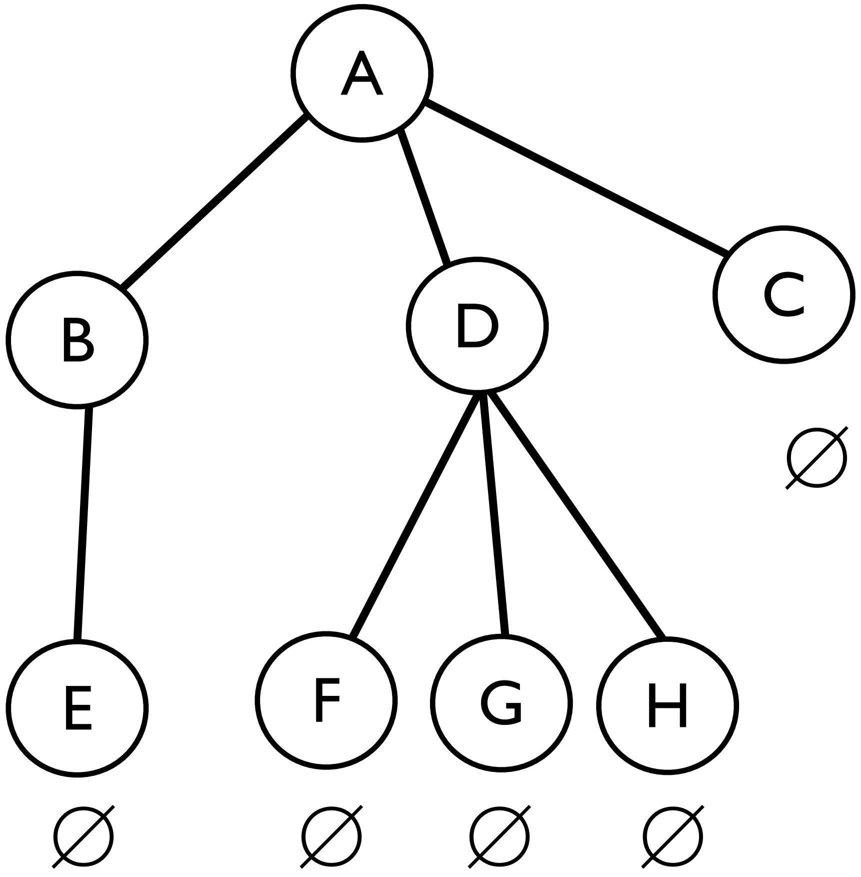
Return right child

Serializing Trees



- Often want to write trees out to disk in a space efficient way.
- Preorder traversal will let you store the nodes.
  - What's the preorder traversal of this tree?
- Need to encode the structure somehow.

Serializing Trees



- In preorder traversal, output a mark when you finish processing a node’s children.

- Leaves = empty children lists: ∅

$$A\ B\ E\ \varnothing\ \underset{e}{)}\ \underset{b}{)}\ D\ F\ \varnothing\ \underset{f}{)}\ G\ \varnothing\ \underset{g}{)}\ H\ \varnothing\ \underset{h}{)}\ \underset{d}{)}\ C\ \varnothing\ \underset{c}{)}\ \underset{a}{)}$$

- ∅ symbols are redundant:
  - ABE))C)D)F)G)H))

- “)” means “go up one level”.

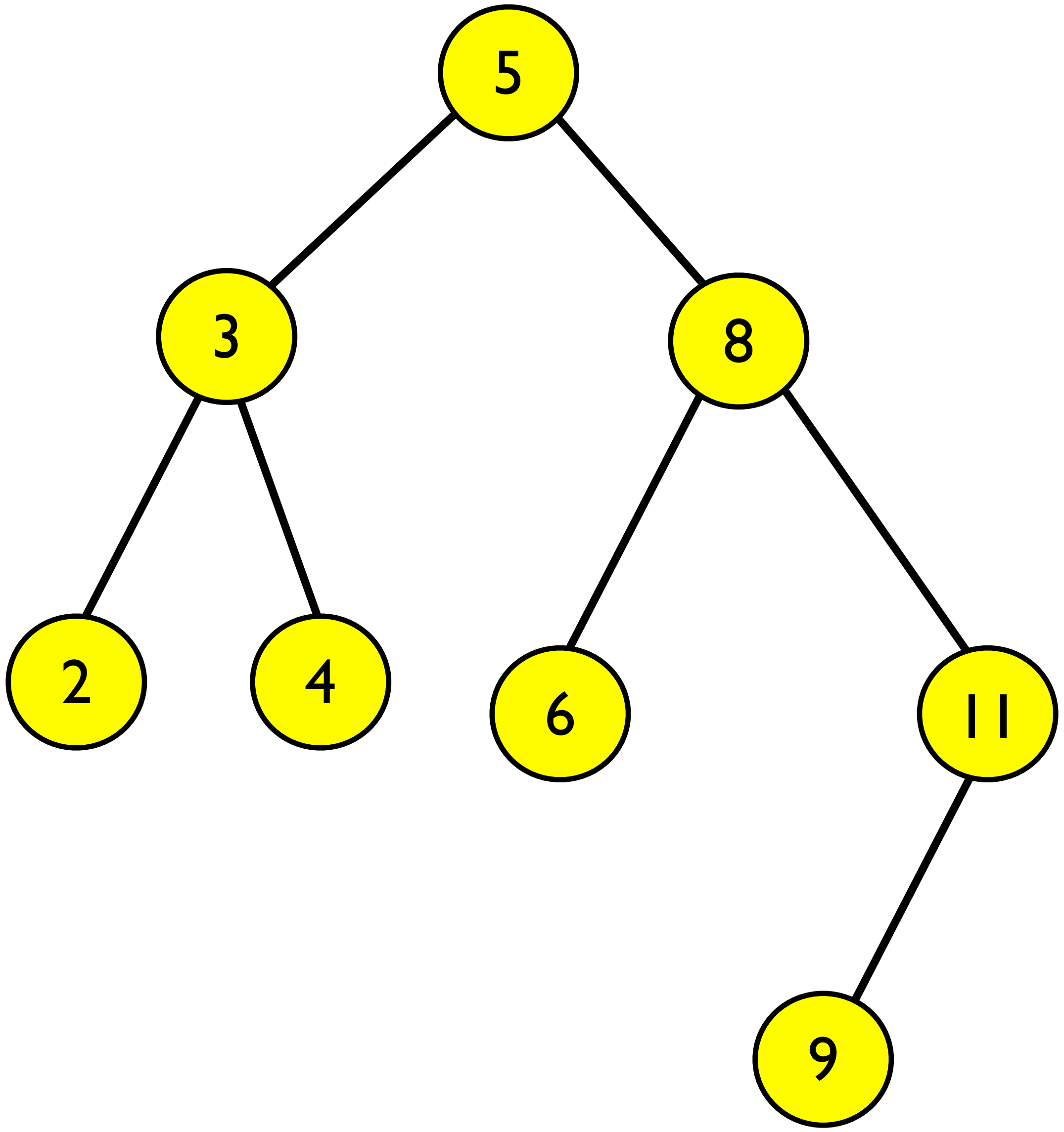
Binary Search Trees

- **BST Property:** If a node has key k then keys in the **left** subtree are $< k$ and keys in the **right** subtree are $> k$.

- We disallow duplicate keys.

- Generalization of the binary search process we saw before:
  - ordering
  - partitioning
  - linking

- Good for implementing the *dictionary ADT* we've already seen: insert, delete, find.



Sorted Set Problem

- If keys are ***totally ordered***, dictionary ADT is sometimes extended to a "sorted set ADT."
  - Totally ordered means: for every pair (a,b) either a < b or b < a).
  - Operations:
    - s = make_sorted_set
    - *find*(*s*, *k*)
    - *insert*(*s*, *k*)
    - *delete*(*s*, *k*)
    - *join*(s_1, *k*, s_2): make a new sorted set from s_1, {k}, s_2; destroy s_1 and s_2. Assumes every item in $s_1 < k$ and $k <$ every time in s_1.
    - *split*(*s*, *k*): return 3 new sorted sets: s_1 with items $< k$, {*k*}, and s_2 with items $> k$.

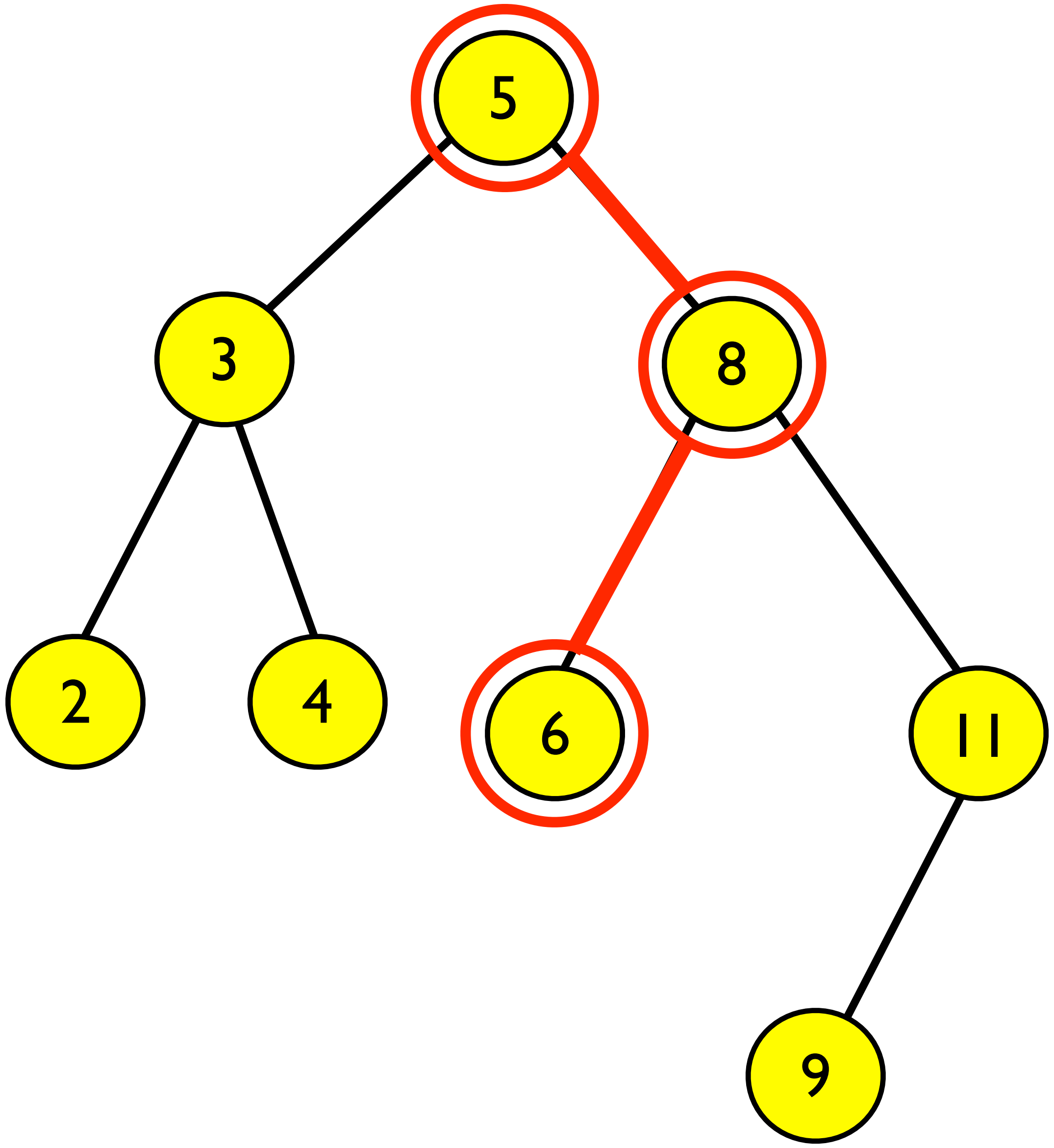
Dictionary operations

BST Find

Find $k = 6$:

Is $k < 5$? No, go right

Is $k < 8$? Yes, go left



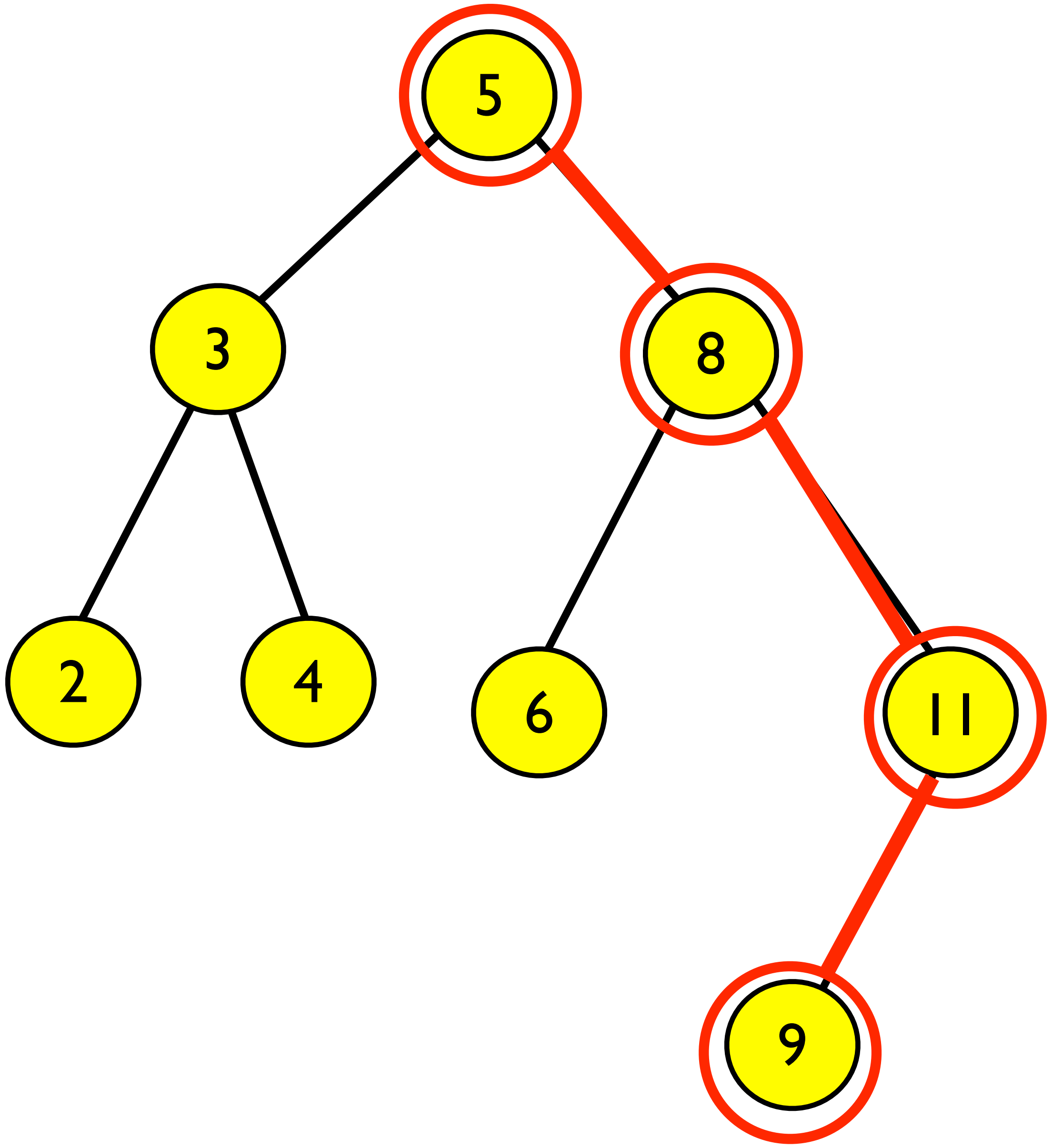
BST Find

Find $k = 9$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

Is $k < 11$? Yes, go left



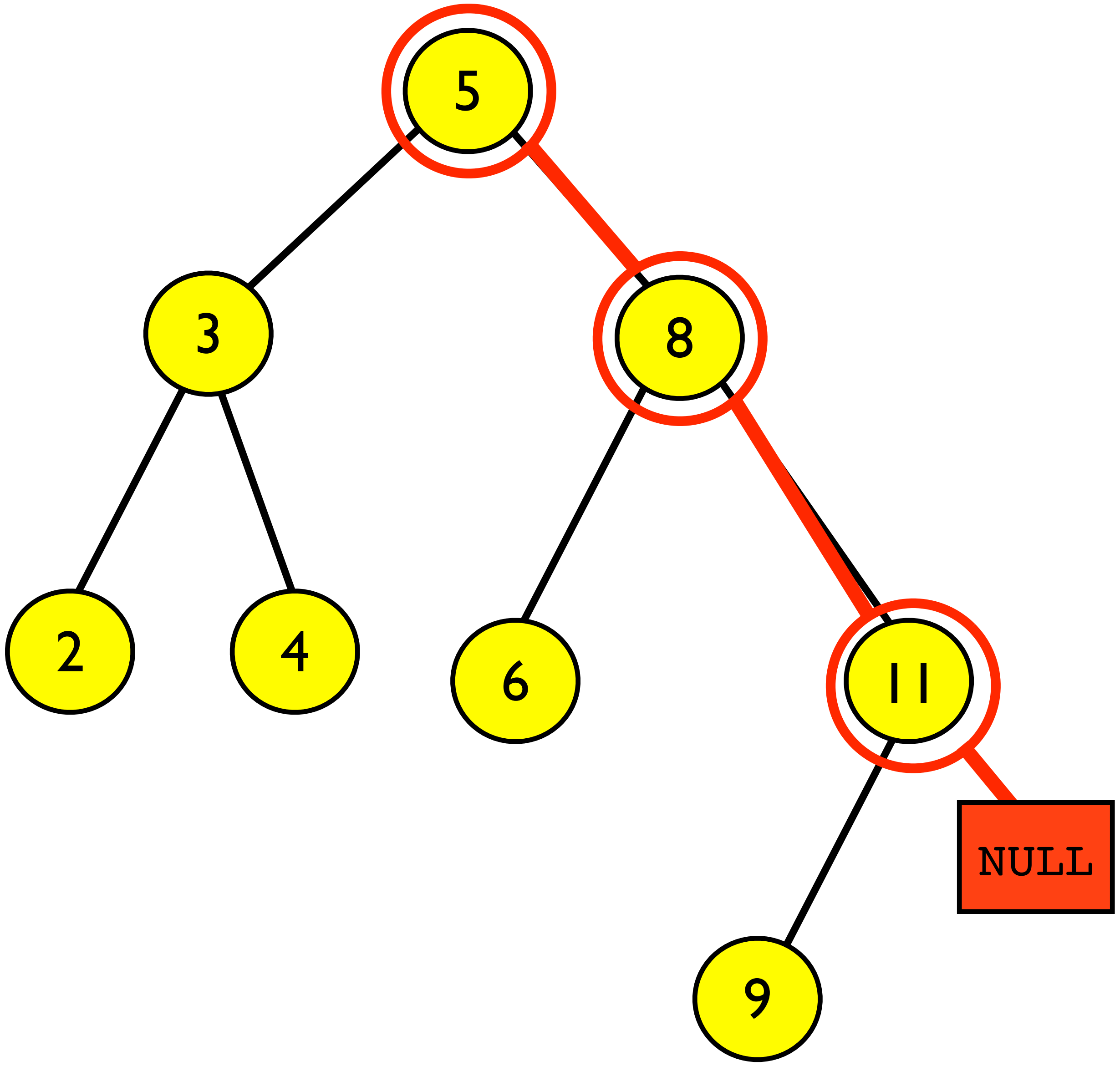
BST Find

Find $k = 13$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

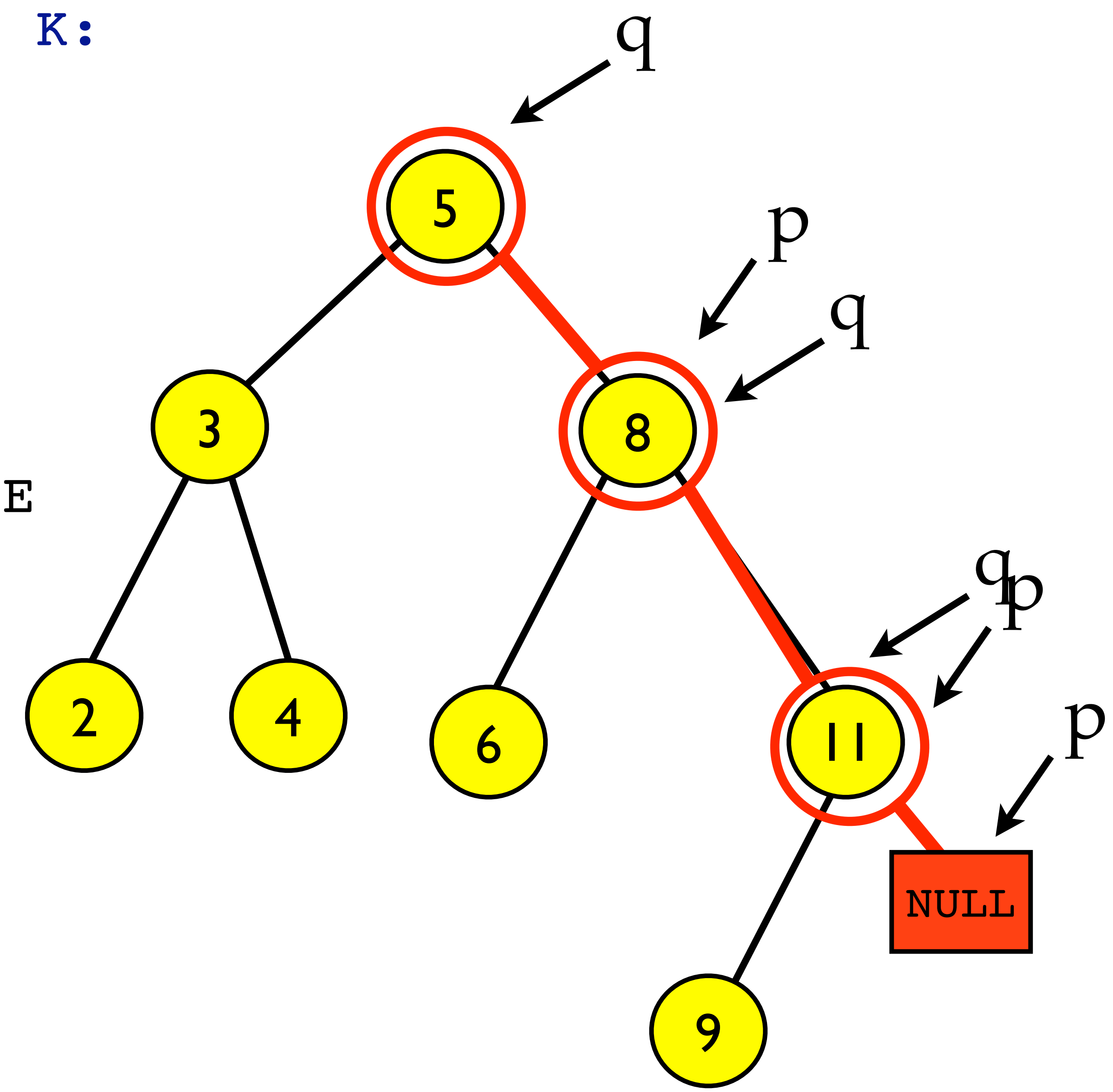
Is $k < 11$? No, go right



BST Insert

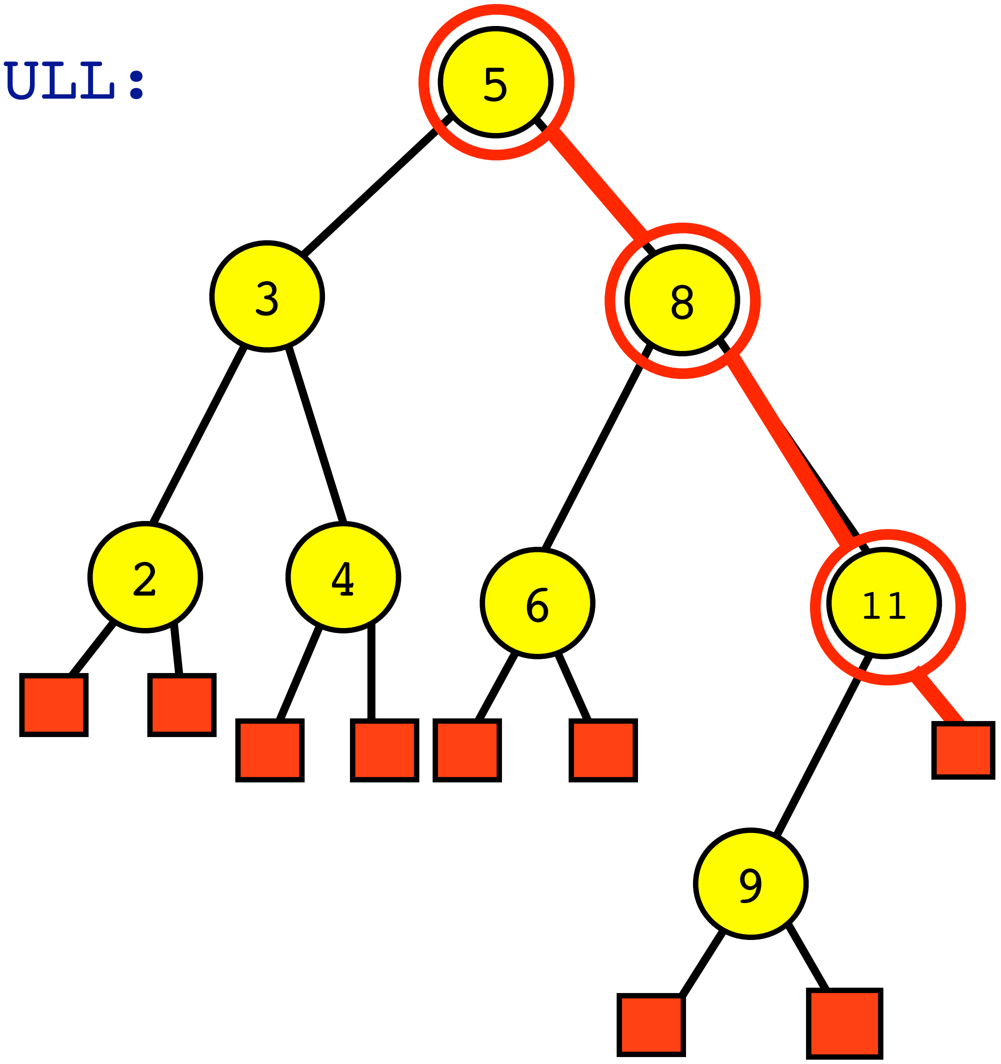
```
insert(T, K):
   q = NULL
   p = T
   while p != NULL and p.key != K:
      q = p
      if p.key < K:
        p = p.left
      else if p.key > K:
        p = p.right

   if p != NULL: error DUPLICATE

   N = new Node(K)
   if q.data > K:
     q.left = N
   else:
     q.right = N
```

Same idea as BST Find

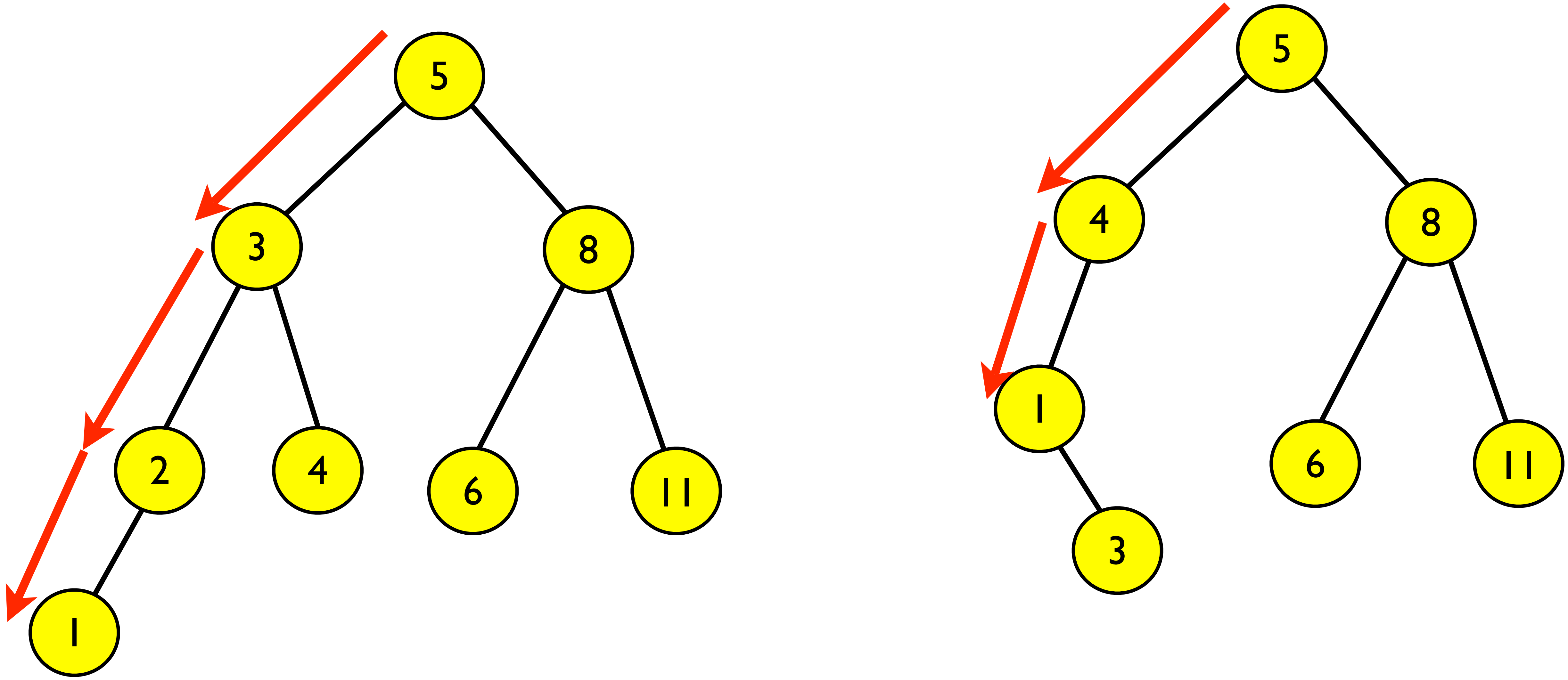


BST Insert with Extended Binary Trees

```
insert(T, K):
   if T == NULL:
      T = new_node(K)
      T.left = new_external_node()
      T.right = new_external_node()
   else
      p = T
      while p.key != K and p.left != NULL:
         if p.key < K:
           p = p.left
         else if p.key > K:
           p = p.right

      if p.left != NULL:
        error DUPLICATE

   p.key = K
   p.left = new_external_node()
   p.right = new_external_node()
```



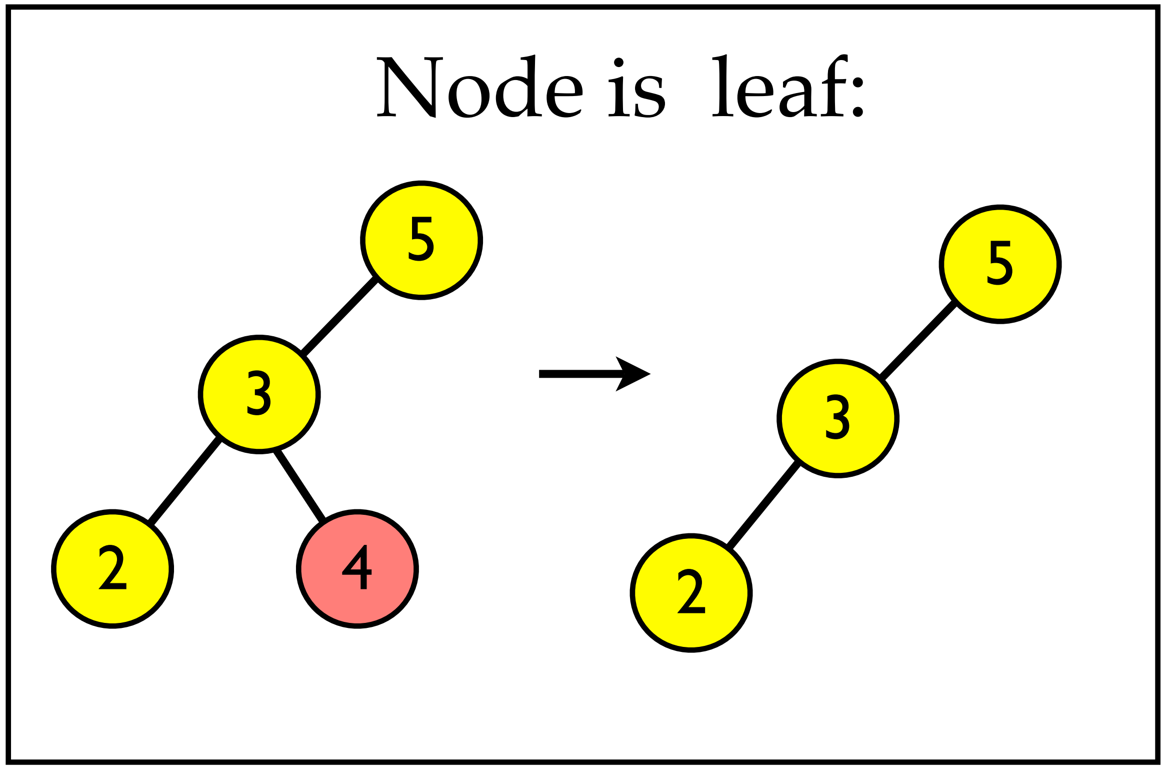
BST FindMin



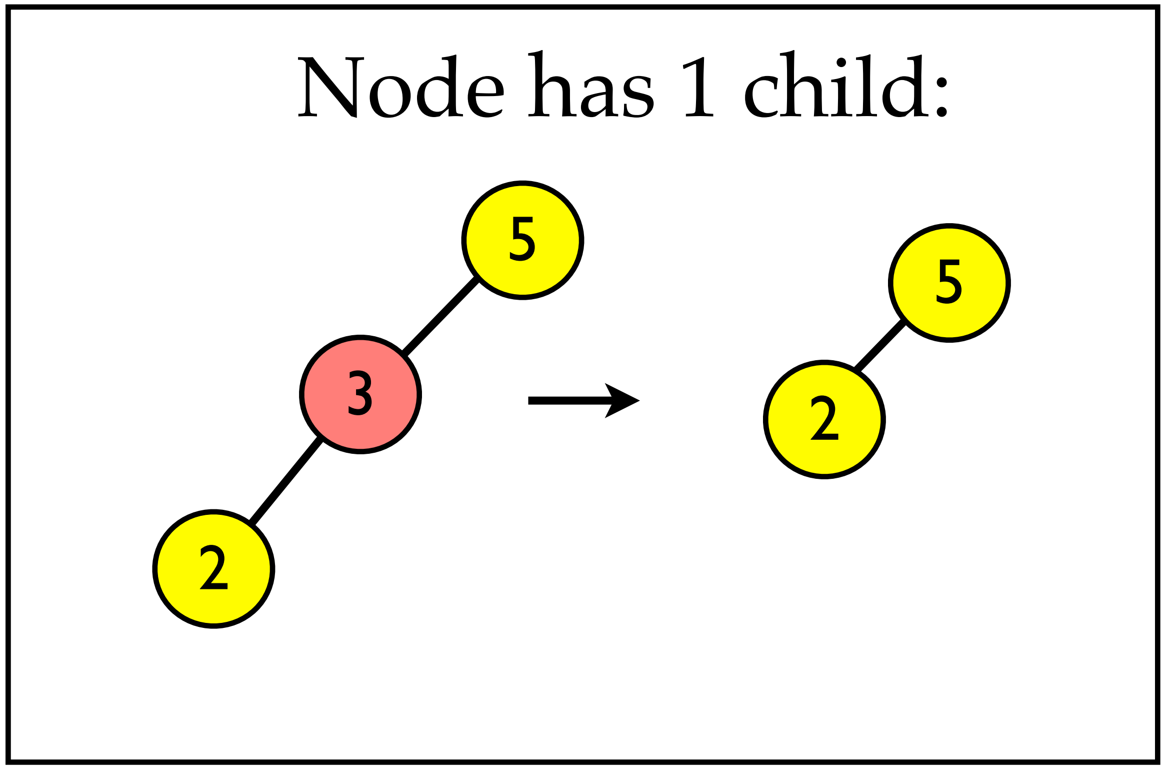
Walk left until you can't go left any more

Can you express inorder_successor using find_min?

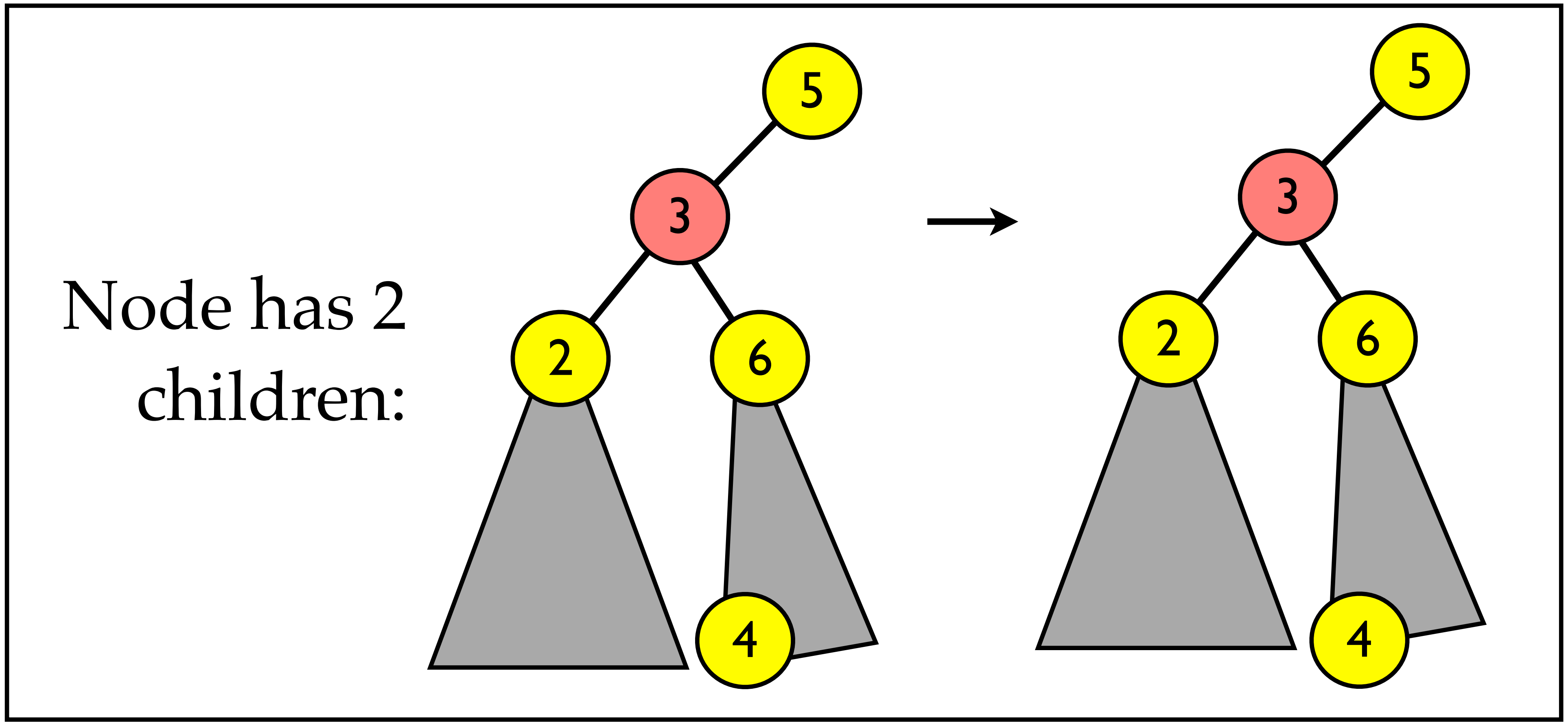
BST Delete



Node is leaf:



Node has 1 child:

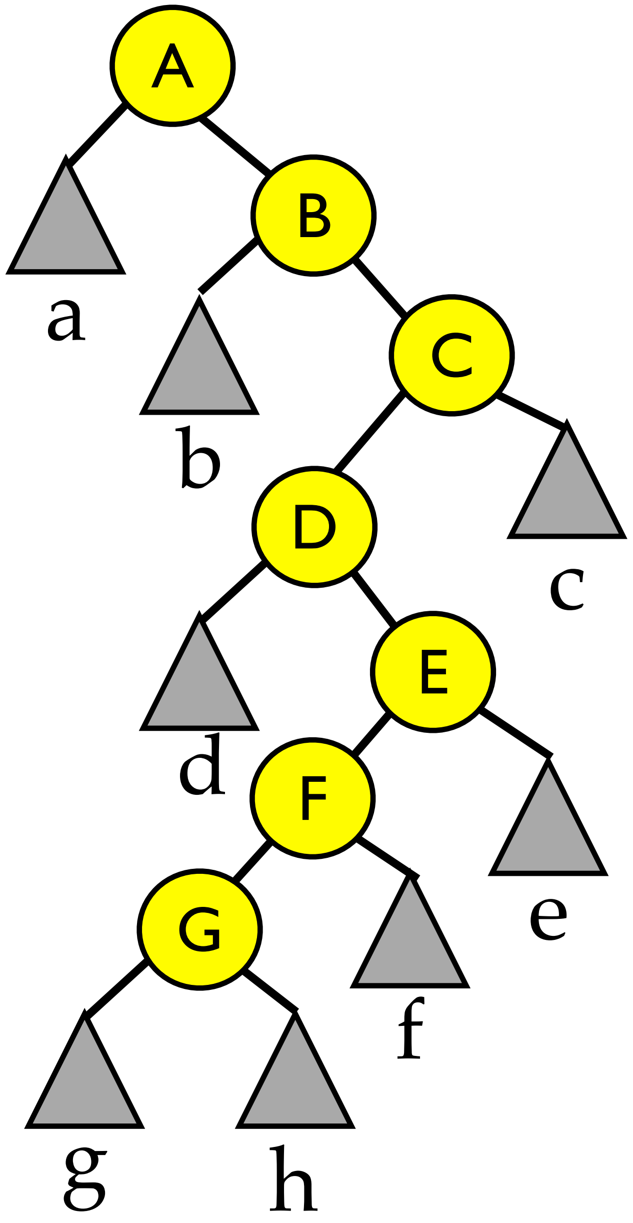


Node has 2 children:

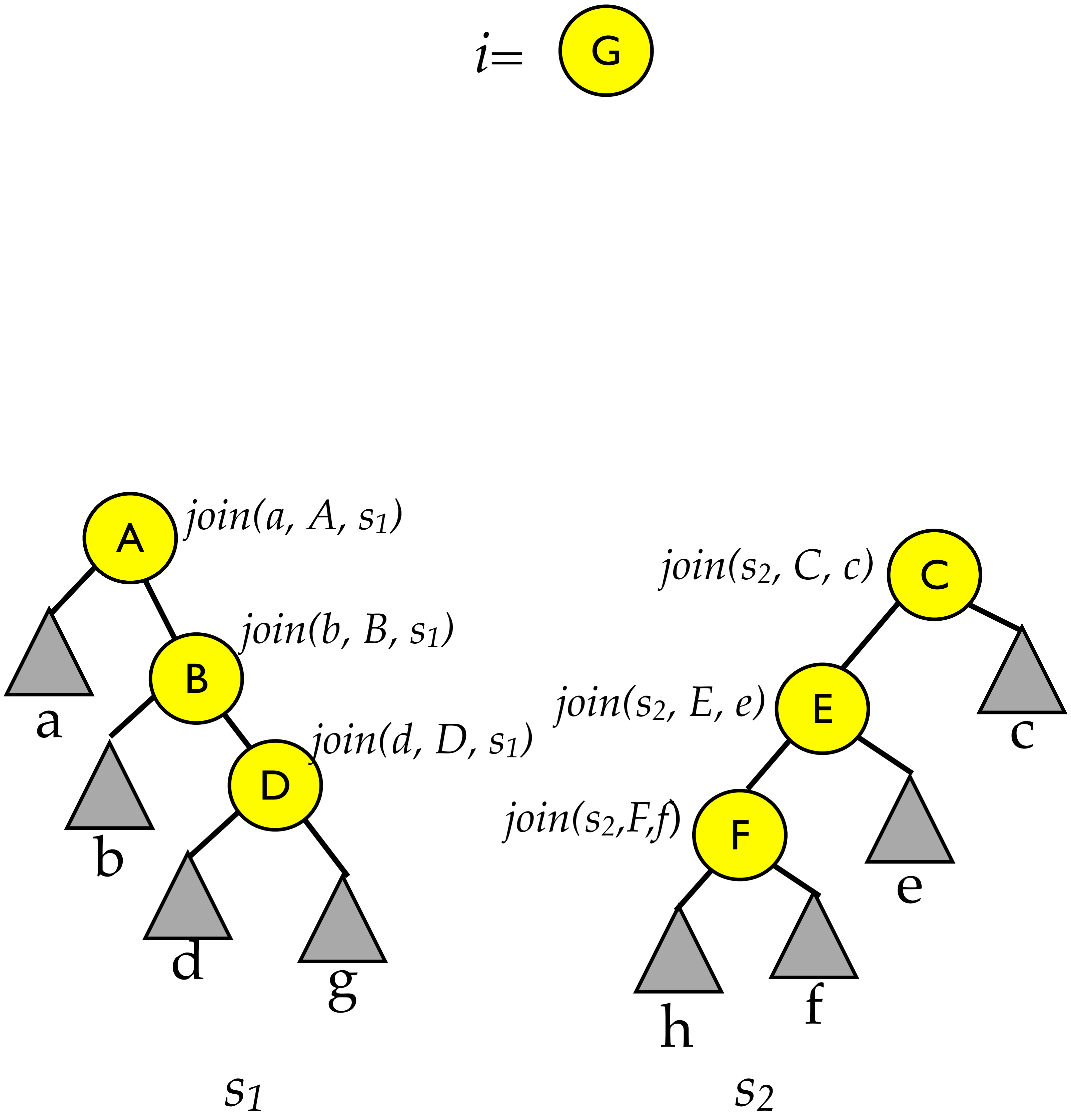
Python Implementation of BST

How would you implement join and split?

Split(G)



Letters represent *labels* not keys



- What’s the worst possible insertion order?

- What’s the best possible insertion order?

Expected Path Length of Random BST

- Suppose the keys k_1, k_2, k_3, ..., k_n are inserted in a random order (every permutation equally likely).
- What is the expected path length to a node in the BST built by inserting these keys?

- **Idea**: consider the leftmost path as a representative path.
- New key k_i added to left-most path when it is the smallest encountered so far ($k_i < k_j$ for $j < i$).
- In a random permutation, how often does the minimum change?
  [This is the length of the leftmost path]

Expected Path Length of Random BST

$$k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8$$

- What's the probability that k_i is the smallest so far?
- $\Pr[k_i \text{ is smallest among } k_1,\ldots k_i] = 1/i$
- Why?
- In a random permutation of $k_1,\ldots k_i$, the minimum is equally likely to be in any one of the i positions.
- Probability it is in the last position = $1/i$.

Expected Path Length of Random BST

keys = $k_1,\ k_2,\ k_3,\ k_4,\ k_5,\ k_6,\ k_7,\ k_8$

random variables = $x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7,\ x_8$

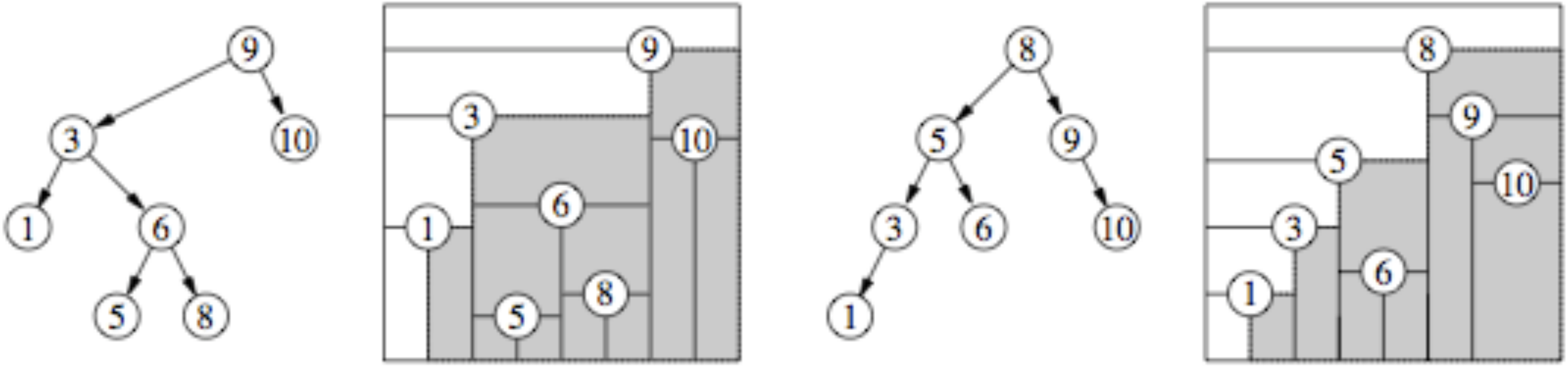
x_i = 1 if k_i is smallest among $k_1,\ldots,k_i$

0 otherwise

- sum of x_i = length of leftmost path.

- Expected length = $E[\sum x_i] = \sum E[x_i]$
  $= \sum[(1/i)1 + 0(1-1/i)]$
  $= \sum(1/i) = H_n = O(\log n)$

Expected Path Length of Random BST


Insertion order: 9 3 10 6 1 8 5

Left chain has 3 nodes

Minimum changes 3 times

Insertion order: 8 9 5 10 3 6 1

Left chain has 4 nodes

Minimum changes 4 times

(Dave Mount's Figure)

Optimal Static BSTs – Cost of trees

$$k_1, \quad k_2, \quad k_3, \quad k_4, \quad k_5, \quad k_6, \quad k_7, \quad k_8$$

$$p_1, \quad p_2, \quad p_3, \quad p_4, \quad p_5, \quad p_6, \quad p_7, \quad p_8$$

- Define the cost of a tree built on keys $k_j,\ldots,k_m$:

$$\mathrm{C(T)} = \sum_{i=j}^{m} p_i(\mathrm{Depth}(T, k_i) + 1)$$

Why is it Depth + 1?

- T is **optimal** if C(T) is smallest among any possible T containing the same keys.

- C(T) = expected cost of searching for a key in T.

Subtrees of optimal tree are optimal trees

- **Goal**: find tree that minimizes C(T).
- **Claim**: every subtree of optimal tree is optimal.
- **Proof:** Let T be an optimal tree on $k_j,\ldots,k_m$, with root = k_r ($j \le r \le m$)

$$\mathrm{C(T)} = p_r + \sum_{i=j}^{r\text{-}1} p_i(\mathrm{Depth}(\mathrm{T}, k_i) + 1) + \sum_{i=r+1}^{m} p_i(\mathrm{Depth}(\mathrm{T}, k_i) + 1)$$

$$\sum_{i=j}^{r\text{-}1} p_i + \sum_{i=j}^{r\text{-}1} p_i\mathrm{Depth}(\mathrm{T}, k_i) \qquad \sum_{i=j}^{r\text{-}1} p_i + \sum_{i=j}^{r\text{-}1} p_i\mathrm{Depth}(\mathrm{T}, k_i)$$

$$C(T_{left}) \qquad C(T_{right})$$

$$\mathrm{C(T)} = \sum_{i=j}^{m} p_i + C(T_{left}) + C(T_{right})$$

So,

$$\mathrm{C(T)} = \sum_{i=j}^{m} p_i + C(T_{left}) + C(T_{right})$$

- If there were a lower cost $\mathrm{T_{left}}$ or $\mathrm{T_{right}}$ we could reduce the total cost of T, contradicting that T is optimal.

- Hence, $\mathrm{T_{left}}$ and $\mathrm{T_{right}}$ must be optimal.

Dynamic Programming to Find OPT Tree

$$C[j, m] = \begin{cases} 0 & \text{if } m < j \text{ (tree is empty)} \\ p_j & \text{if } j = m \text{ (tree is single node)} \\ \sum_{i=j}^{m} p_i + \min_{r} \{ C[j, r\text{-}1] + C[r+1, m]\} & \text{if } j < m \end{cases}$$

$$\underset{\underset{j}{\uparrow}}{k_2},\quad \underset{\underset{r}{\uparrow}}{k_3},\quad k_4,\quad k_5,\quad k_6,\quad k_7,\quad \underset{\underset{m}{\uparrow}}{k_8}$$

So: if you fill in the C[*j*, *m*] table from in order of increasing *m*-*j*, you'll always have the value of C[*j*,*m*] computed when you need it.

Dynamic Programming

The chosen values of *r* partition the nodes and give you the optimal tree structure.