# Binary Search Tree Insertions

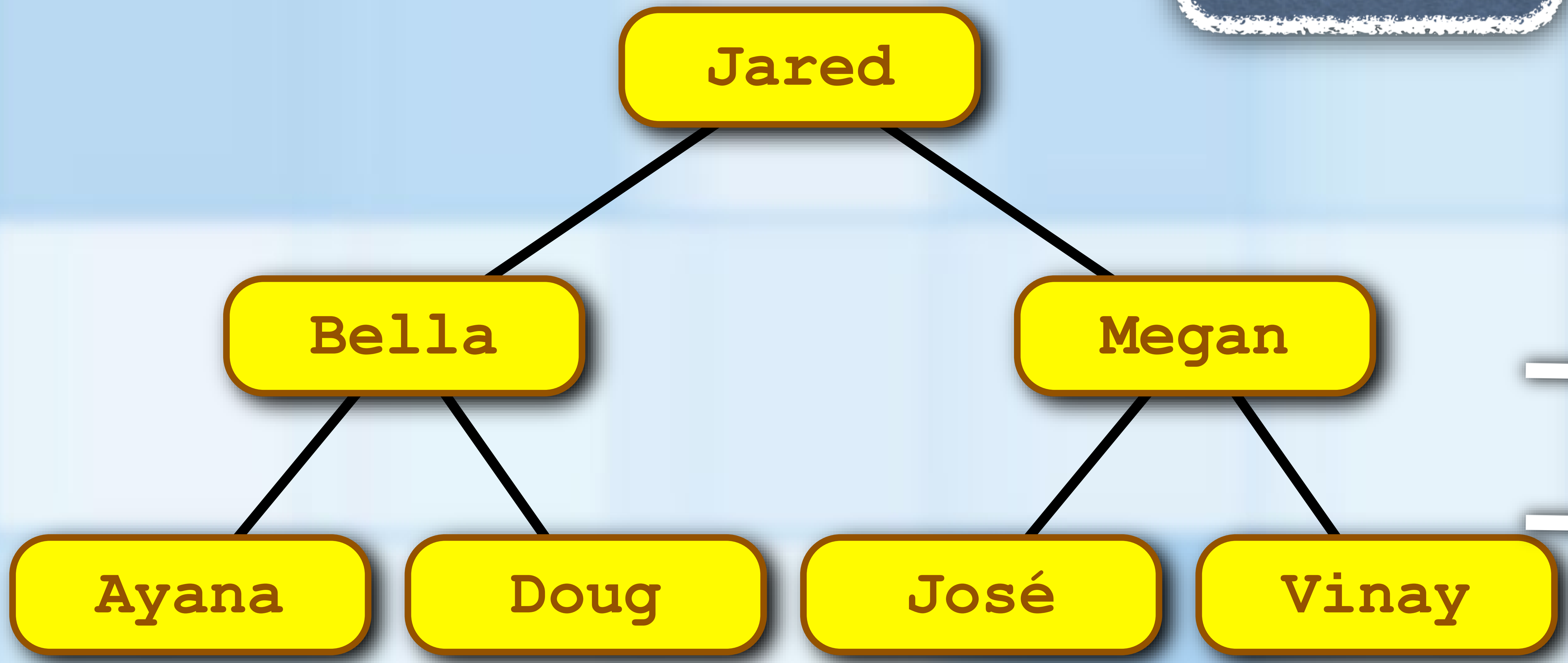
Pearson

# The Class BinarySearchTree

BinarySearchTree.cpp

- **Searching for an entry**
  - Recursive implementation
  - Similar to Binary Search algorithm

findNode( Doug )



Base Case: empty tree

Base Case: target found

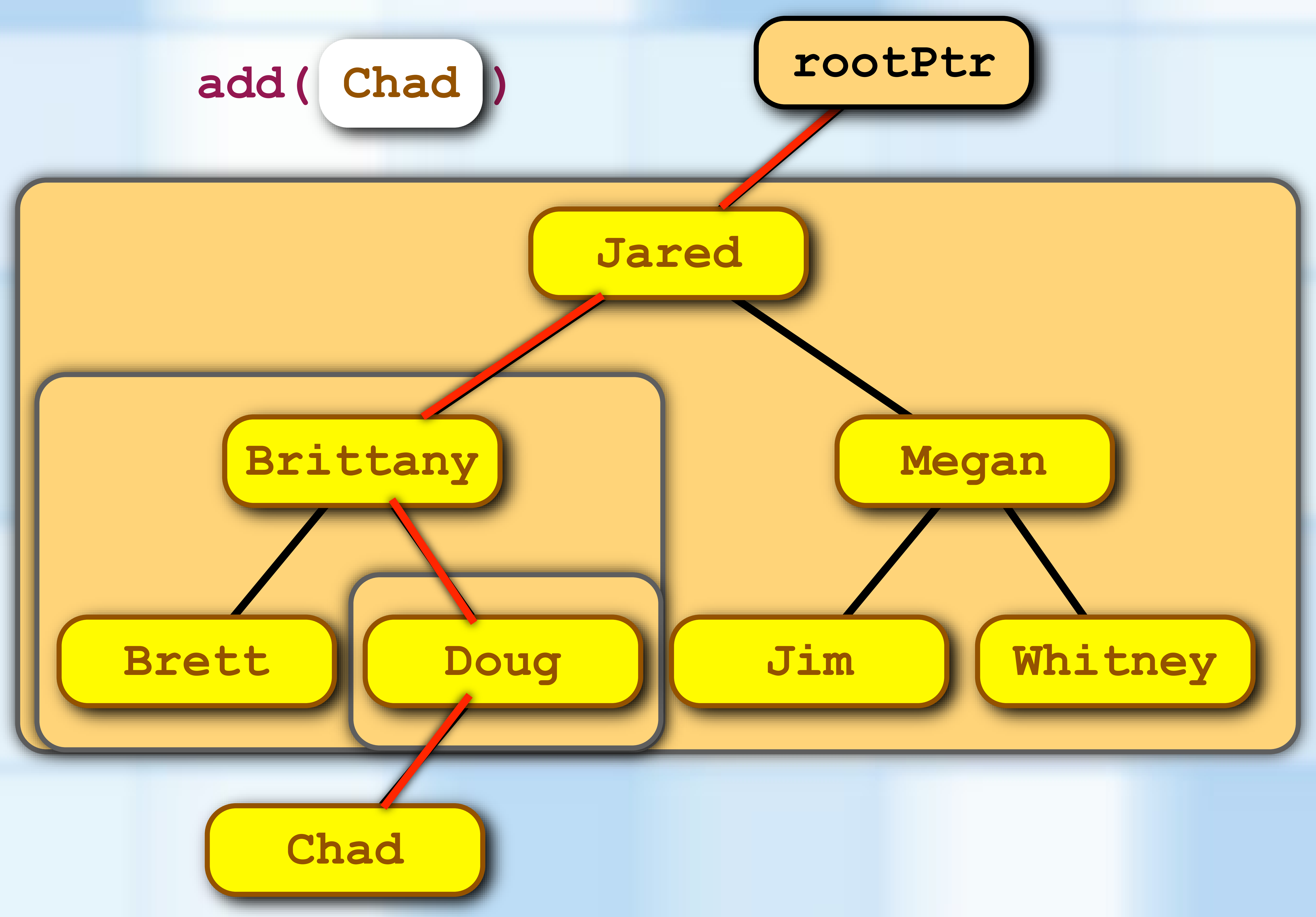
```
template<class ItemType>
bool BinarySearchTree<ItemType>::contains(
                                        const ItemType& anEntry) const
{
  return findNode(rootPtr, anEntry);
}
```

```
template<class ItemType>
auto BinarySearchTree<ItemType>::
   findNode(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                        const ItemType& target) const
{
  if(subTreePtr == nullptr) // not found here
    return nullptr;

  else if (subTreePtr->getItem() == target) // found it
    return subTreePtr;

  else if (subTreePtr->getItem() > target)
    return(findNode(subTreePtr->getLeftChildPtr(), target));

  else
    return(findNode(subTreePtr->getRightChildPtr(), target));
}
```

Pearson

# Adding to a Binary Search Tree

- Must maintain binary search tree structure
- Every addition to a binary search tree adds a new leaf to the tree.



# Adding to a Binary Search Tree

- Must maintain binary search tree structure
- Every addition to a binary search tree adds a new leaf to the tree.

```
template<class ItemType>
bool BinarySearchTree<ItemType>::add(const ItemType& newData)
{
  auto binaryNodePtr = std::make_shared<BinaryNode<ItemType>>(newData);
  rootPtr = insertInorder(rootPtr, binaryNodePtr);
  return true;
}
```

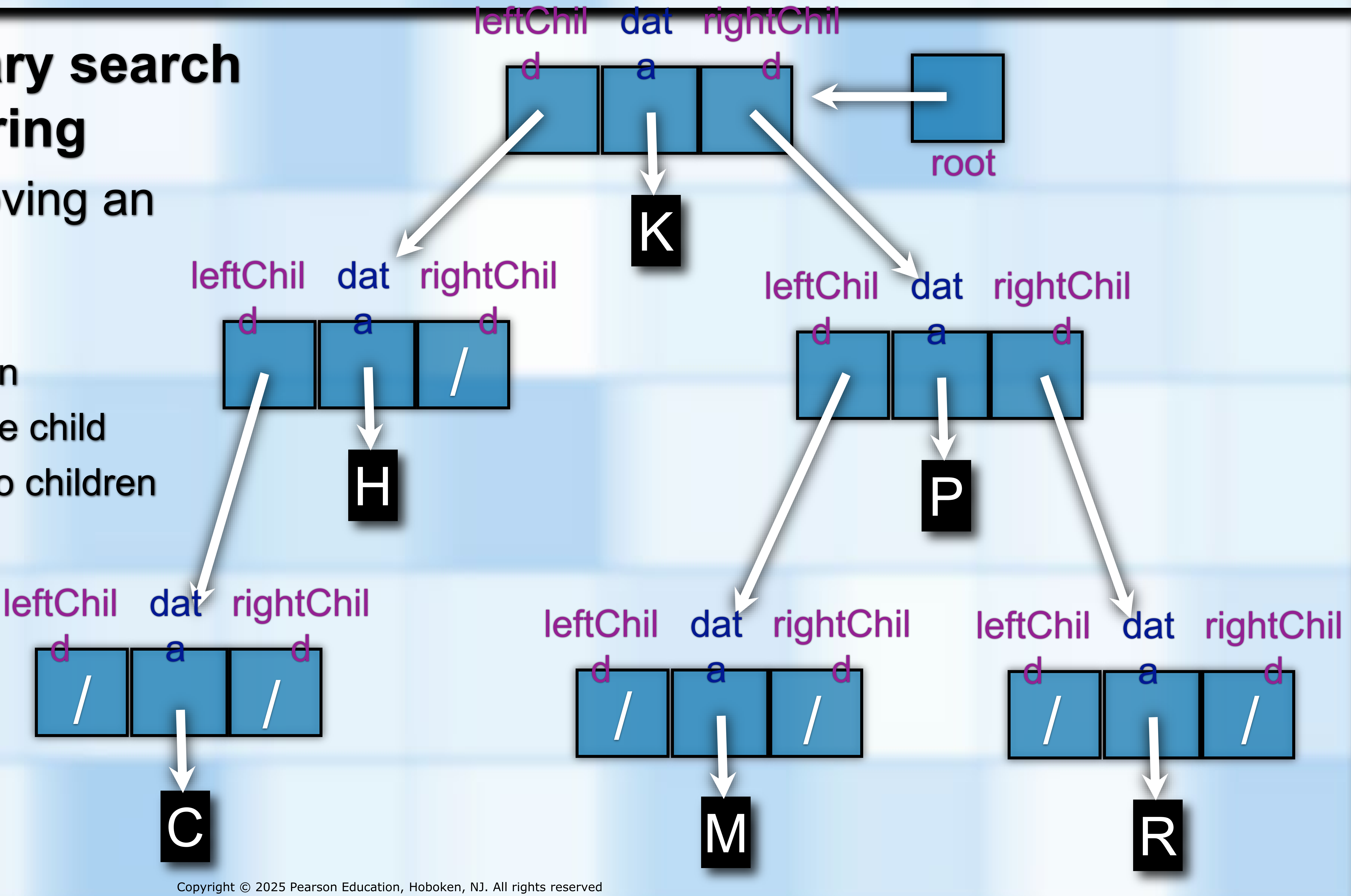
```
template<class ItemType>
auto BinarySearchTree<ItemType>::
                insertInorder(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                  std::shared_ptr<BinaryNode<ItemType>> newNodePtr)
{
  if (subTreePtr == nullptr)
    return newNodePtr;
  else
  {
    if (subTreePtr->getItem() > newNodePtr->getItem())
      subTreePtr->setLeftChildPtr(insertInorder(subTreePtr->getLeftChildPtr(), newNodePtr));
    else
      subTreePtr->setRightChildPtr(insertInorder(subTreePtr->getRightChildPtr(), newNodePtr));
    return subTreePtr;
  }
}
```

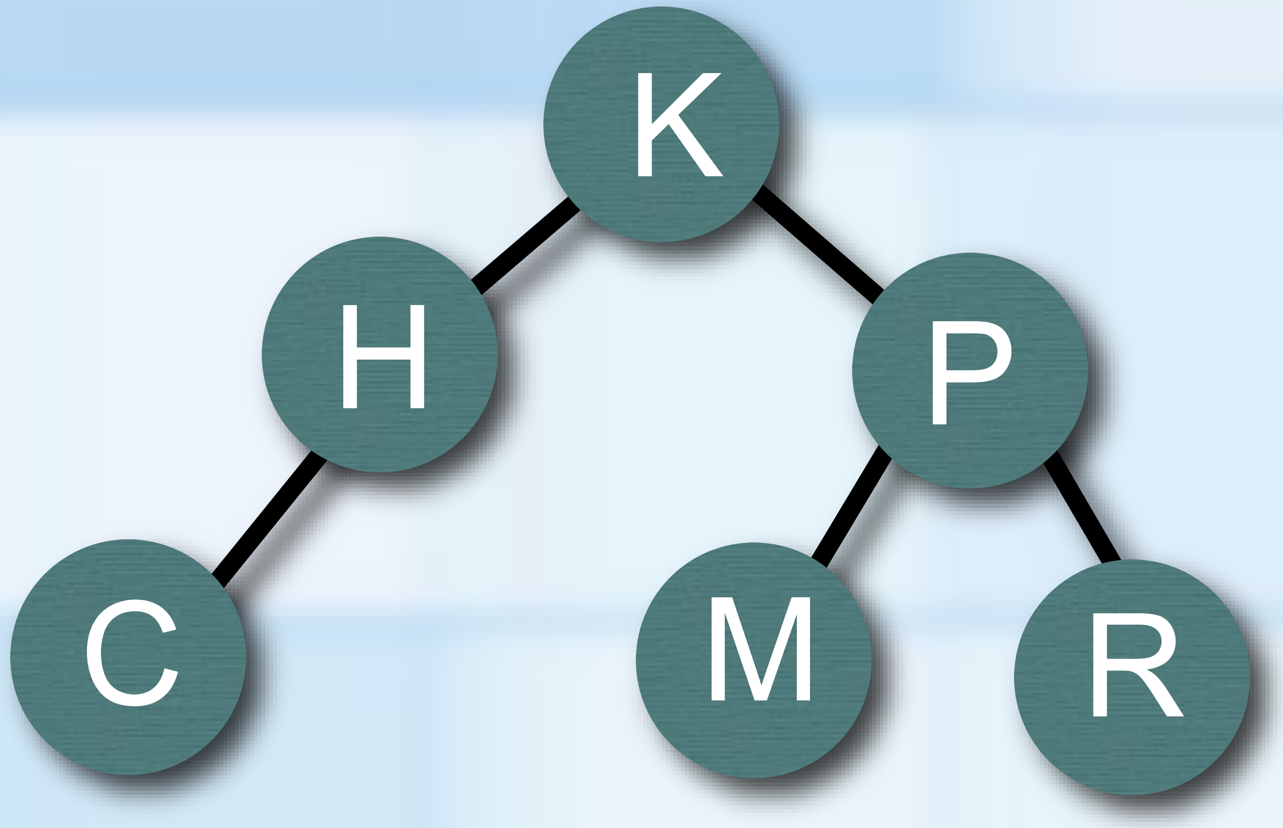
Base Case

# Binary Search Tree Deletions

# Removing an Entry from a BST

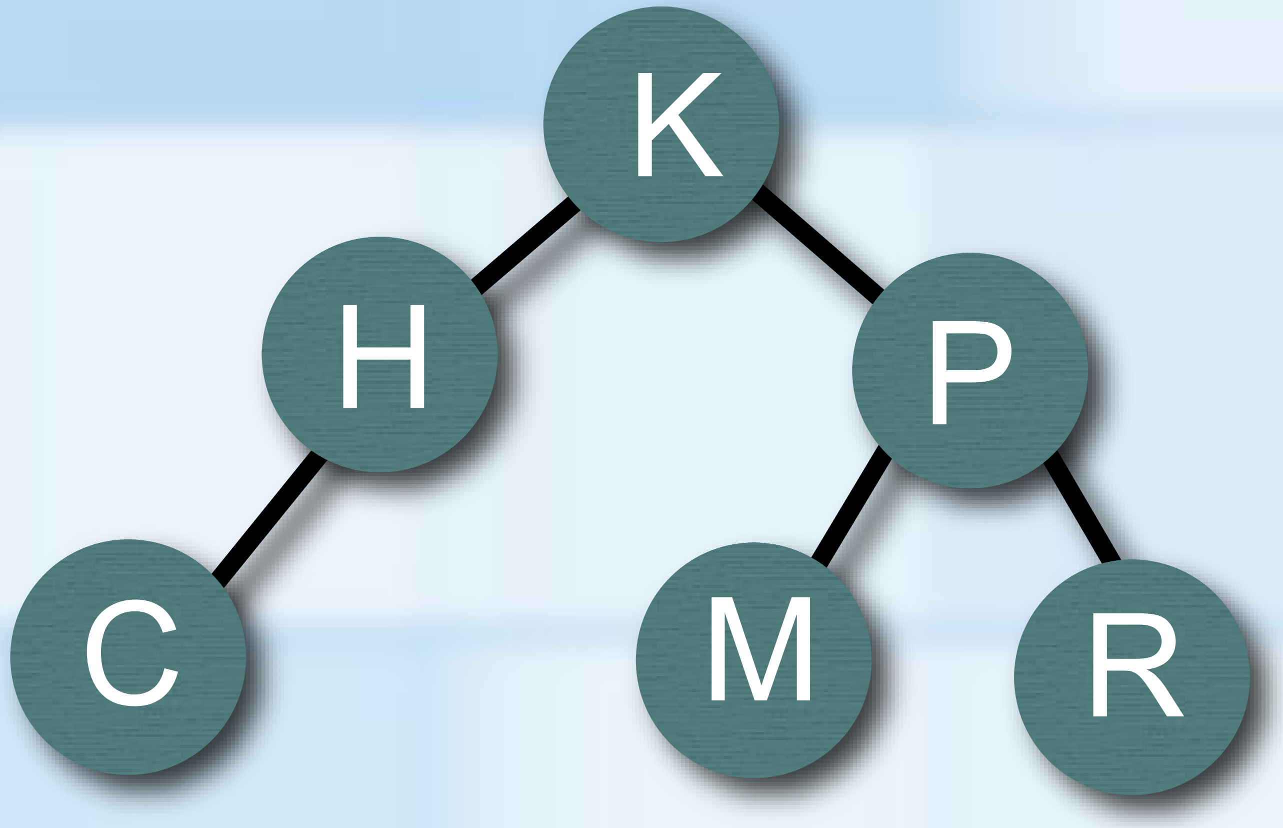
- **Must maintain binary search tree traversal ordering**
  - Three cases for removing an entry
    - Entry is in a leaf
      - the node has no children
    - Entry is in a node with one child
    - Entry is in a node with two children



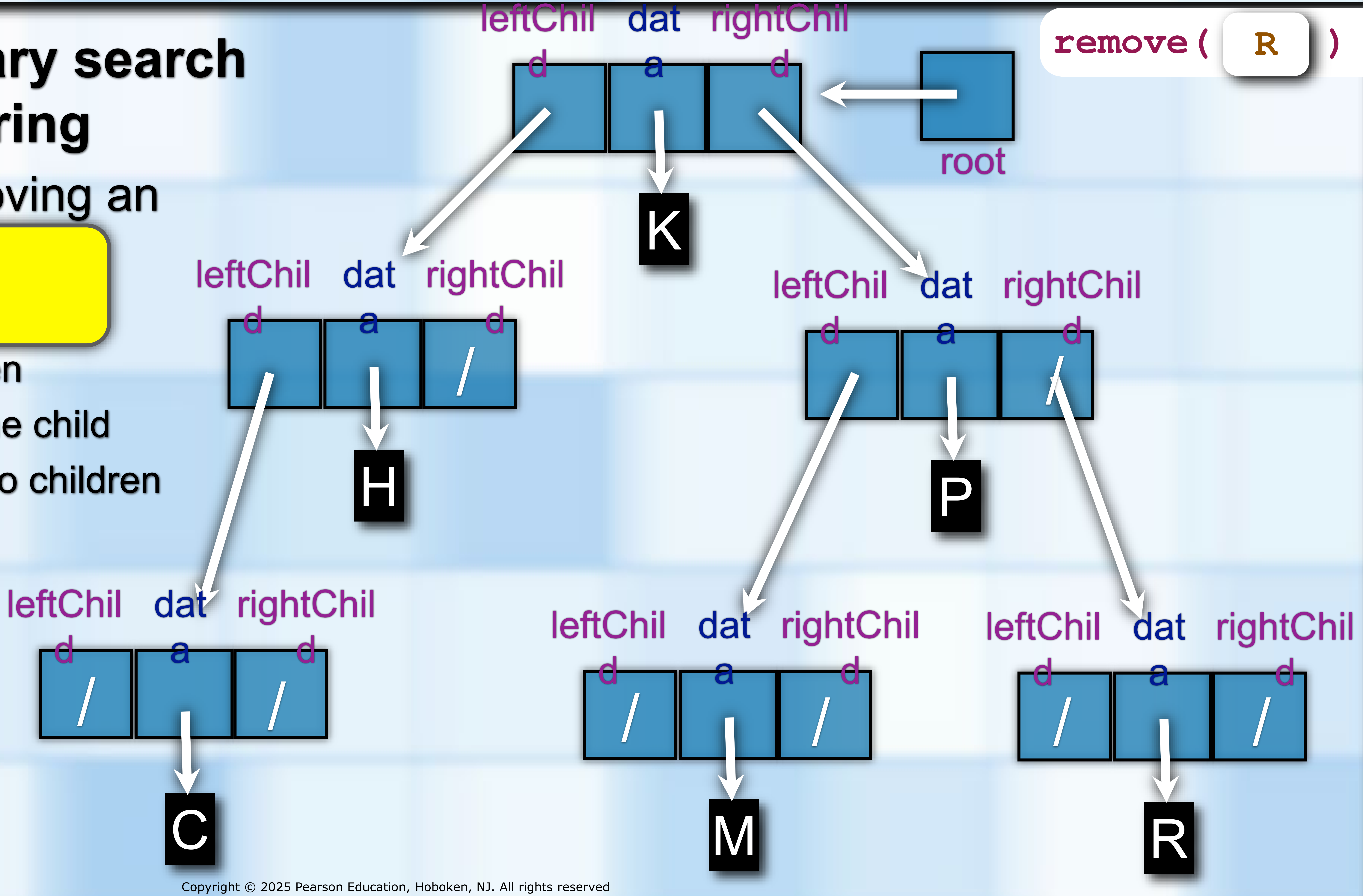


# REMOVING AN ENTRY FROM A BST

- **Must maintain binary search tree traversal ordering**
  - Three cases for removing an entry
    - Entry is in a leaf
      - the node has no children
    - Entry is in a node with one child
    - Entry is in a node with two children

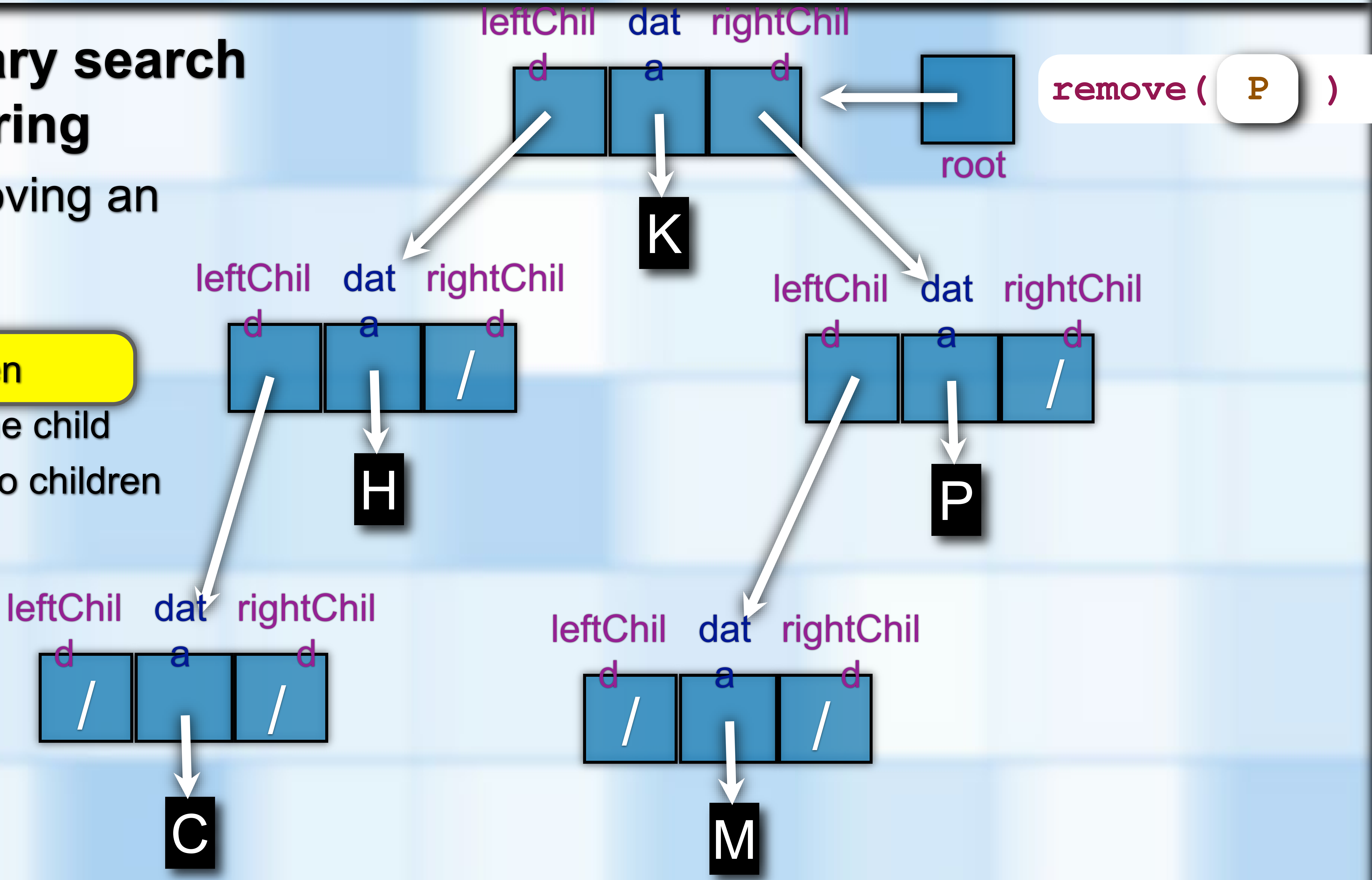


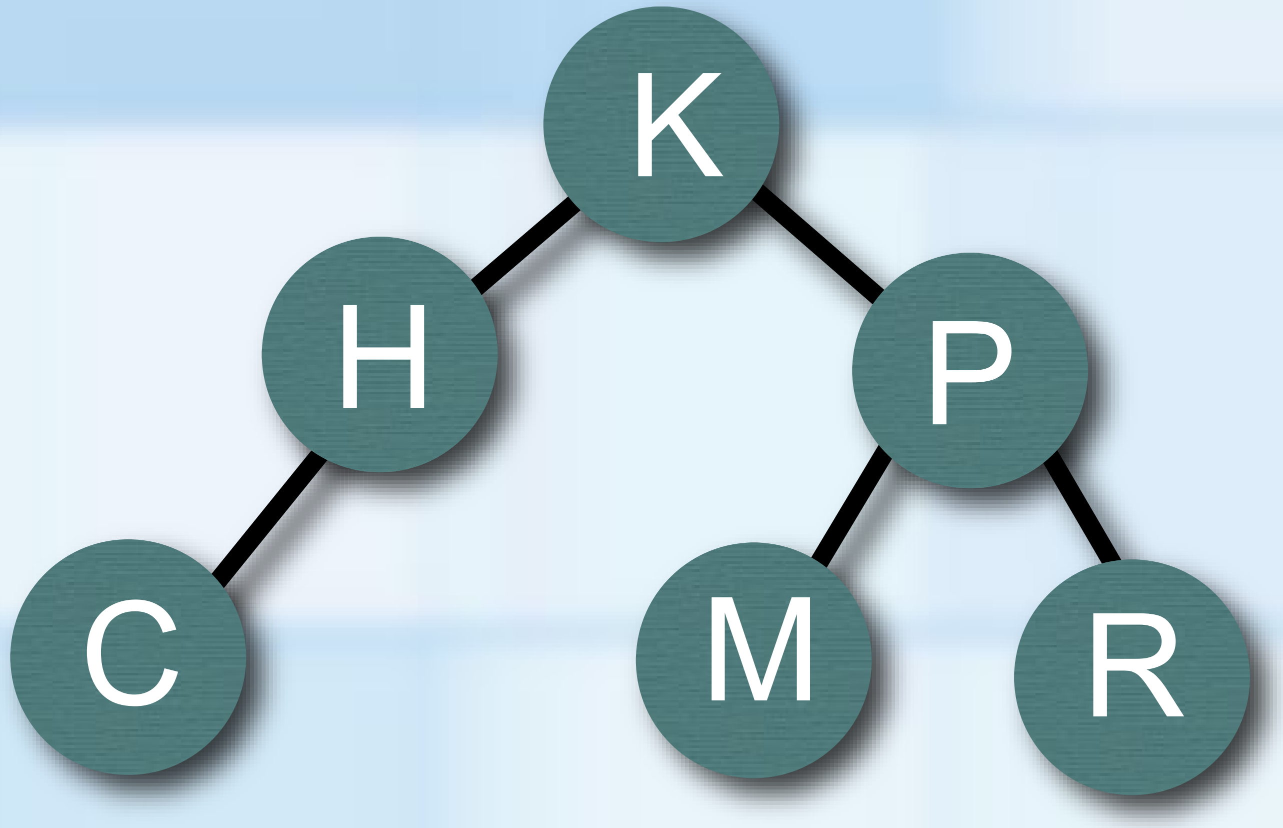
remove( R )



# Removing an Entry from a BST

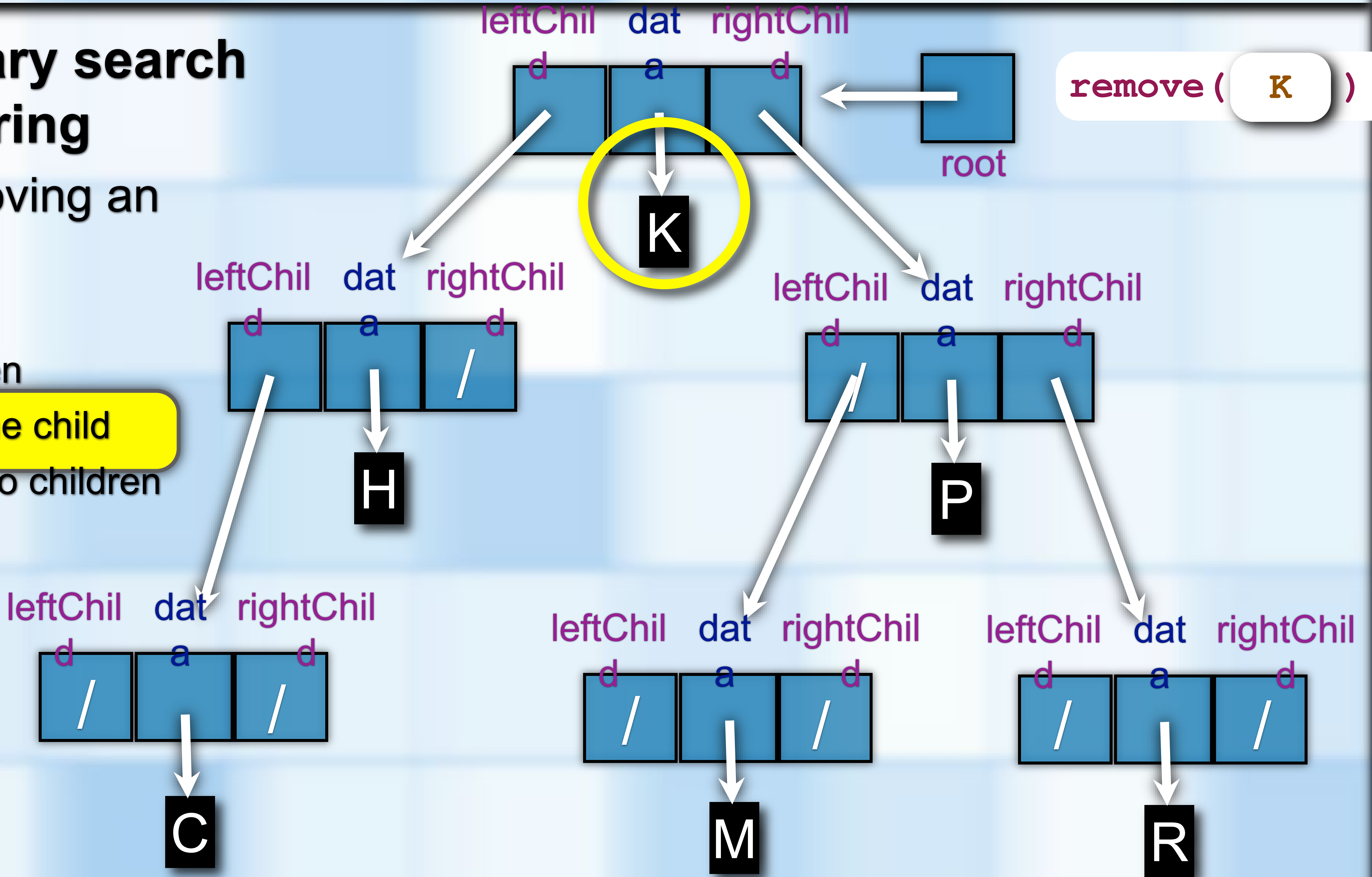
- **Must maintain binary search tree traversal ordering**
  - Three cases for removing an entry
    - Entry is in a leaf
      - the node has no children
    - Entry is in a node with one child
    - Entry is in a node with two children





# Removing an Entry from a BST

- **Must maintain binary search tree traversal ordering**
  - Three cases for removing an entry
    - Entry is in a leaf
      - the node has no children
    - Entry is in a node with one child
    - Entry is in a node with two children



```
remove( K )
```