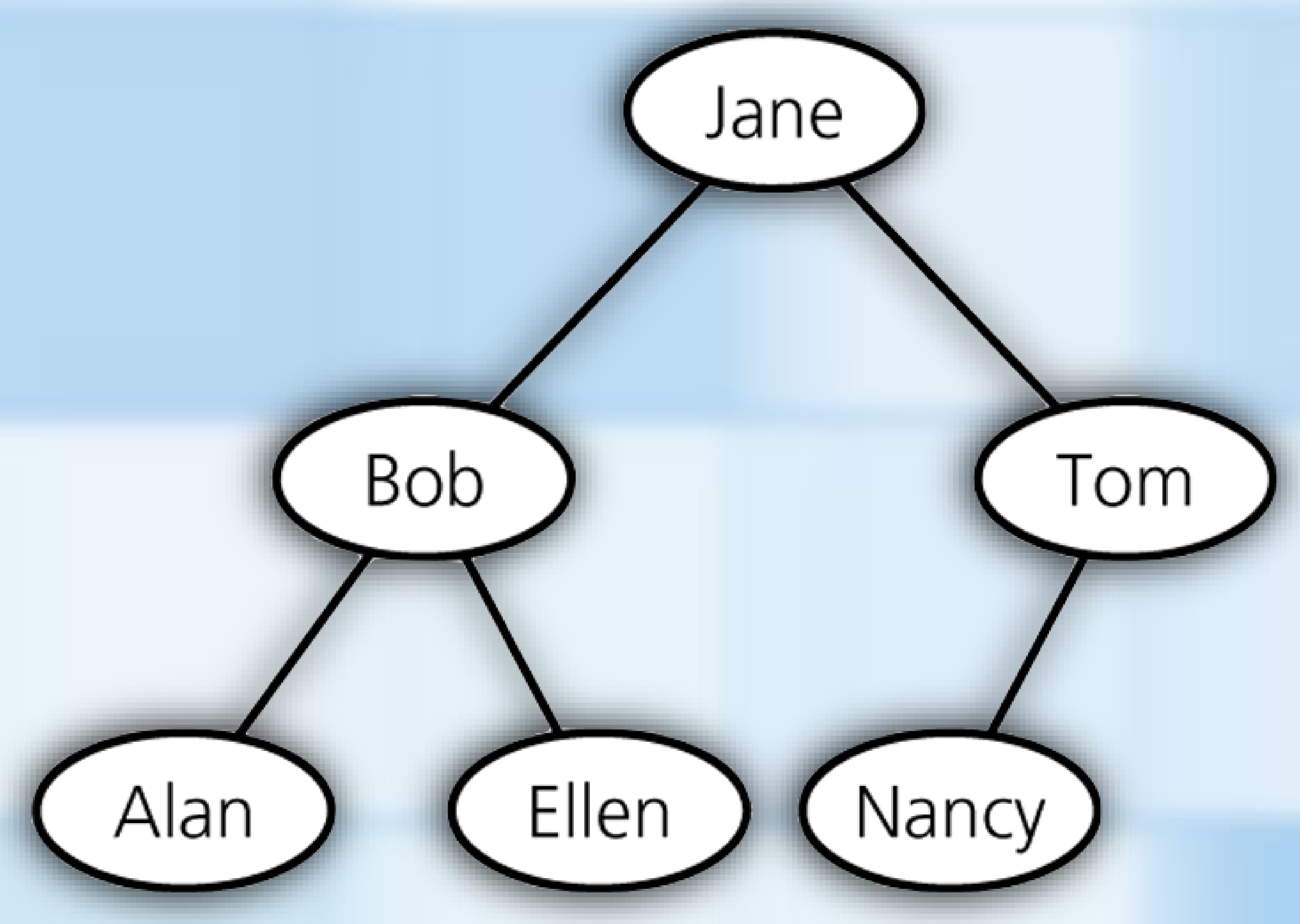
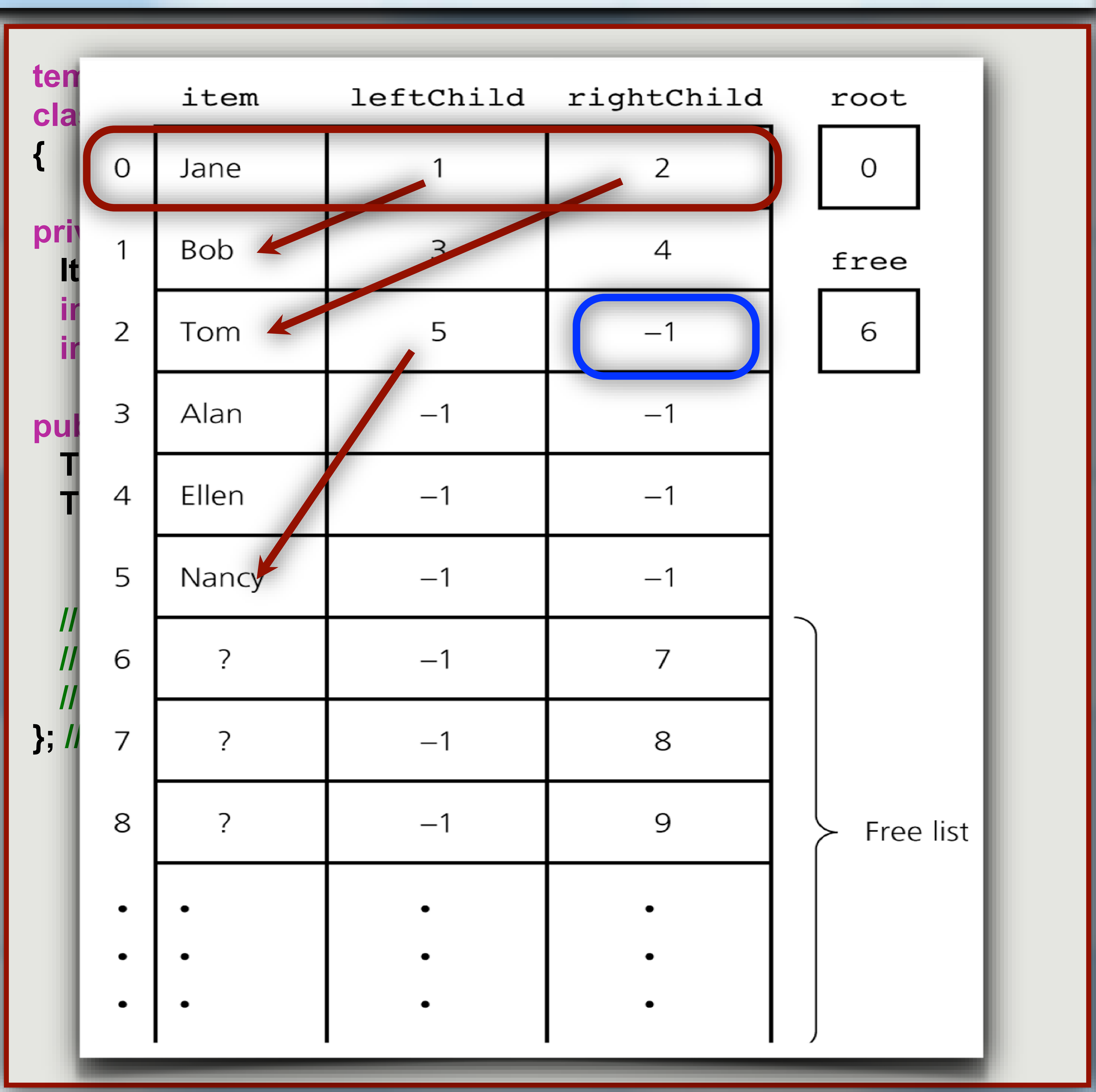
Implementing Binary Trees

Pearson

Array-based Binary Trees

- **Array-based Tree Representation**
  - Each array element contains a **`TreeNode`**
  - Child nodes indicated by index number
    - If there is no left or right child, use **`-1`**





```
tem
cla
{

pri
  It
  in
  in

pub
  T
  T

  //
  //
  //
}; //
```

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

root: 0

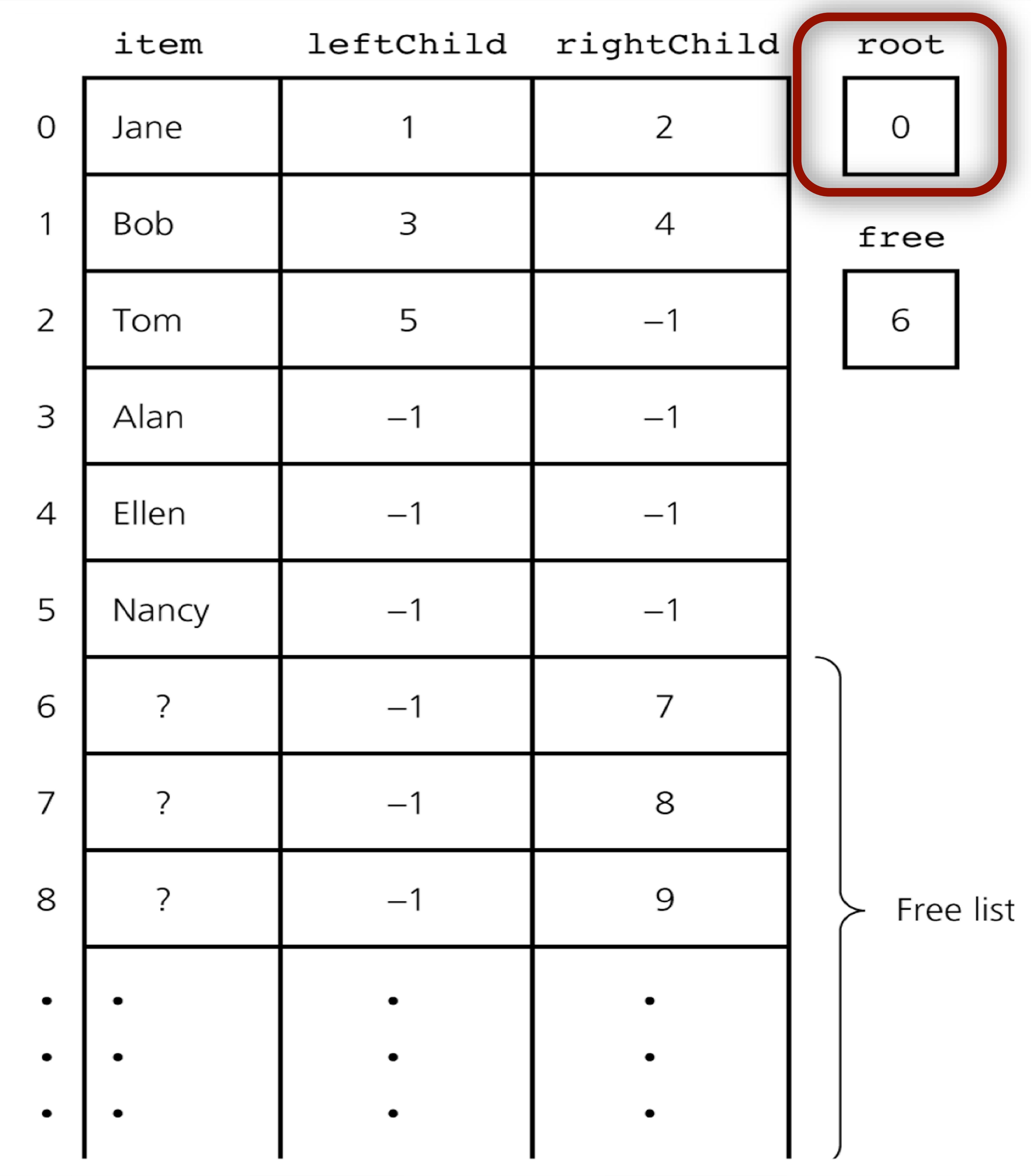
free: 6

Free list

Array-based Binary Trees

- **Requires three member variables**
  - **`items`** - the array of **`TreeNodes`**
  - **`root`** - index of array element containing root node
  - **`free`** - index to a "free space" list
    - Keeps track of available nodes
    - Implemented as a linked list using array indexes

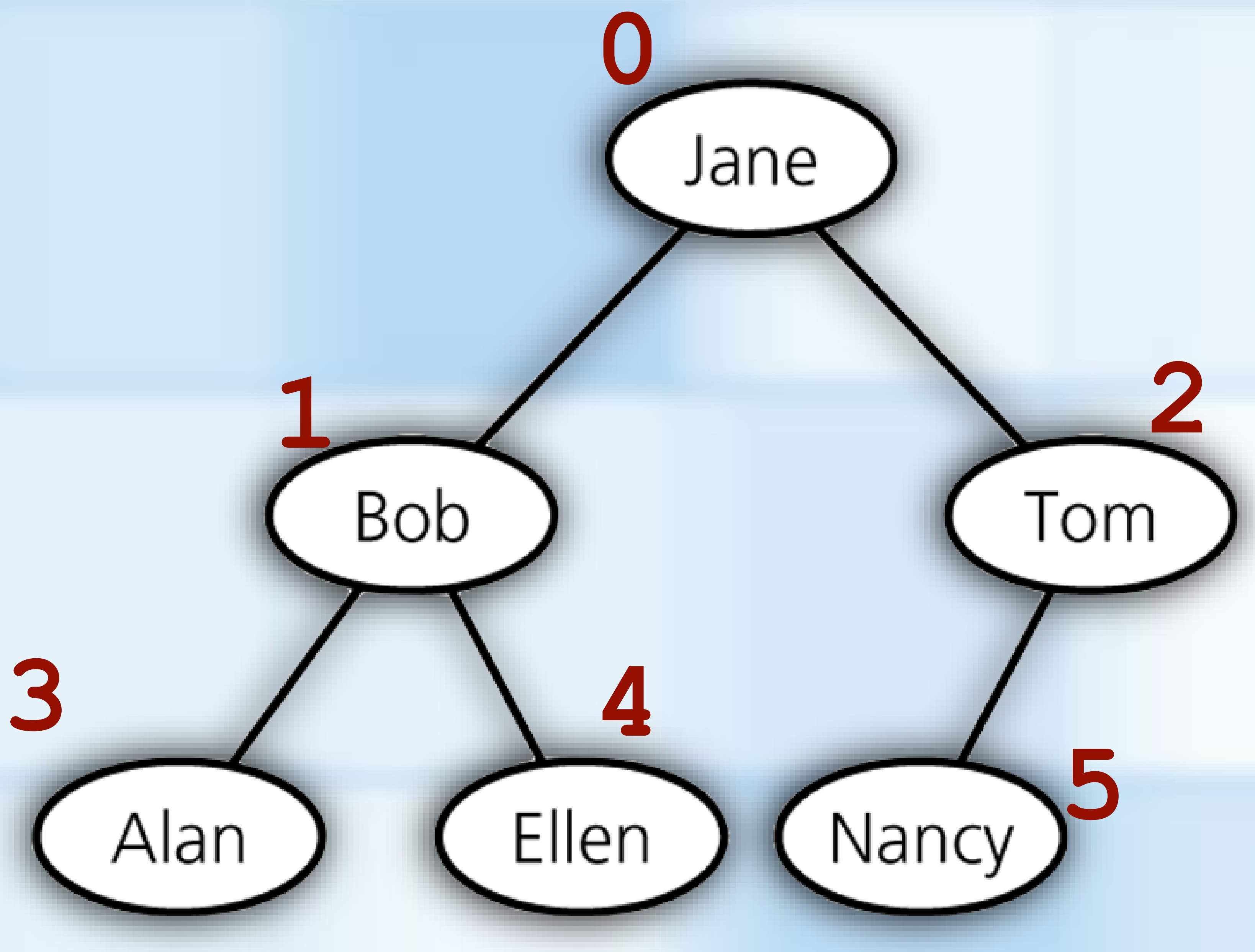
```
template<class ItemType>
class BinaryTree
{
private:
  TreeNode<ItemType> items[MAX_TREE_SIZE];
  int root;
  int free;
           . . .
}
```

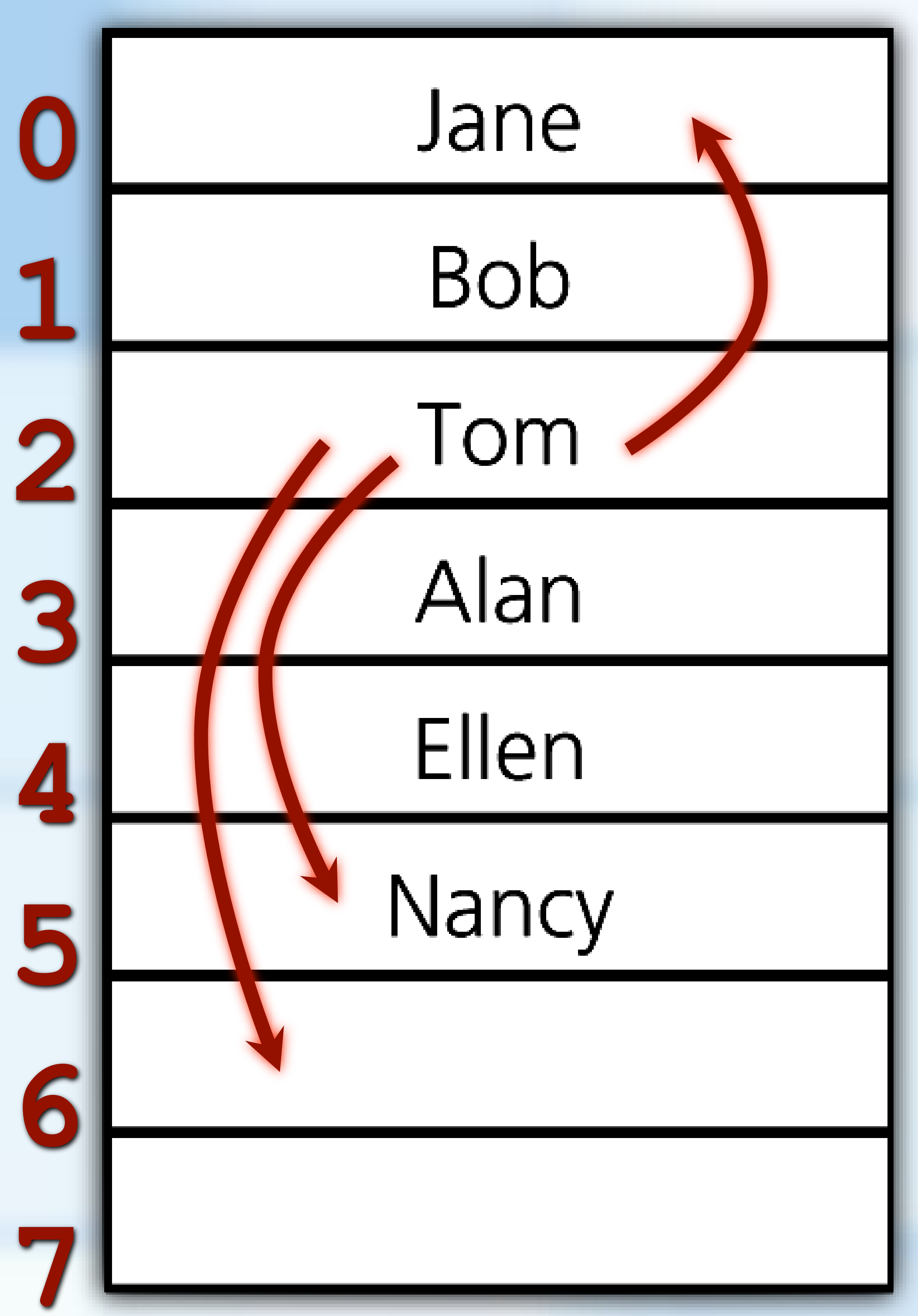


	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

Array-Based Binary Trees

- **If a binary tree is complete**
  - and will remain complete during tree use (Heap)
- **Use an memory-efficient array-based implementation**
  - Use a level-order traversal of tree to store items
  - `root` is at `0`
  - For node at index *`n`*:
    - index of `leftChild` is `2 * n + 1`
    - index `rightChild` is `2 * (n + 1)`
    - index of parent is `(n - 1) / 2`





Reference-Based Binary Trees

Binary Nodes

H
J
K
L
M
N
P
Q
R
S
leftChil
d
dat
a
rightChil
d
K
leftChil
d
dat
a
rightChil
d
/
/
N
leftChil
d
dat
a
rightChil
d
/
/
P

The Class BinaryNode

BinaryNode.h

```
template<class ItemType>
class BinaryNode
{

private:
   ItemType item;                      // Data portion
   std::shared_ptr<BinaryNode<ItemType>> leftChildPtr;   // Pointer to left child
   std::shared_ptr<BinaryNode<ItemType>> rightChildPtr;  // Pointer to right child

public:
   BinaryNode();
   BinaryNode(const ItemType& anItem);
   BinaryNode(const ItemType& anItem,
            std::shared_ptr<BinaryNode<ItemType>> leftPtr,
            std::shared_ptr<BinaryNode<ItemType>> rightPtr);

   void setItem(const ItemType& anItem);
   ItemType getItem() const;

   auto getLeftChildPtr() const;
   auto getRightChildPtr() const;

   void setLeftChildPtr(std::shared_ptr<BinaryNode<ItemType>> leftPtr);
   void setRightChildPtr(std::shared_ptr<BinaryNode<ItemType>> rightPtr);

   bool isLeaf() const;

}; // end BinaryNode
```

BinaryNode.cpp

```
template<class ItemType>
bool BinaryNode<ItemType>::isLeaf() const
{
   return ((leftChildPtr == nullptr)
                              && (rightChildPtr == nullptr));
}
```

Reference-Based Binary Trees

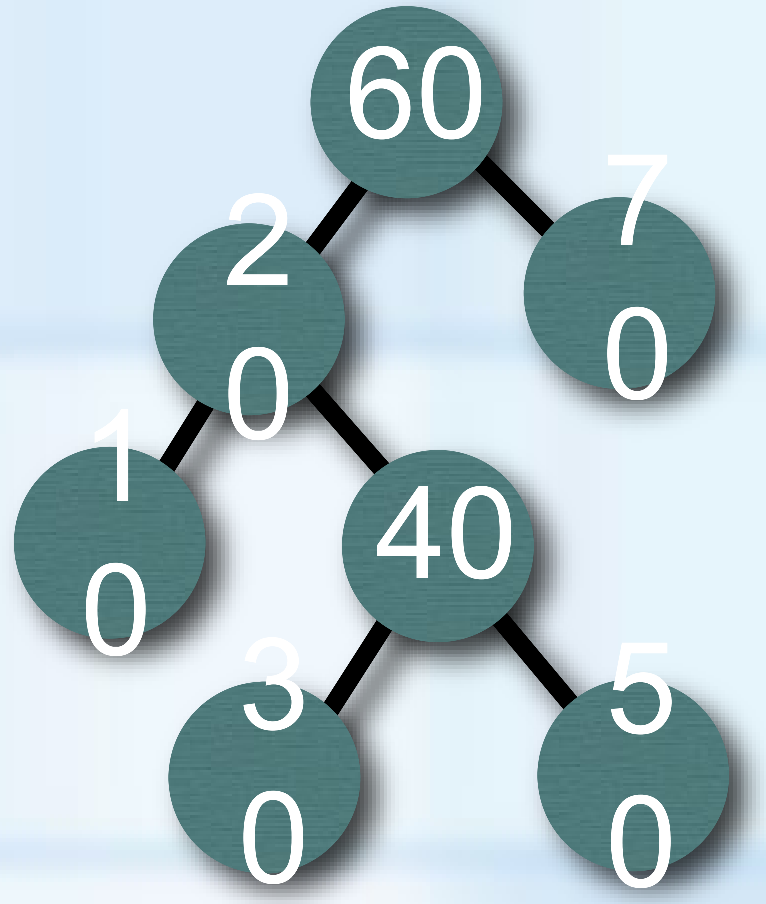
- In-Order traversal -- Visit root after visiting it's left subtree

```
template<class ItemType>
void BinaryNodeTree<ItemType>::inorderTraverse(std::function<void (ItemType&)> visit) const
{
   inorder(visit, rootPtr);
}  // end inorderTraverse
```

```
template<class ItemType>
void BinaryNodeTree<ItemType>::inorder(std::function<void (ItemType&)> visit,
                                        std::shared_ptr<BinaryNode<ItemType>> treePtr) const
{
   if (treePtr != nullPtr)
   {
      ItemType theItem = treePtr->getItem();
      if (treePtr.isLeaf())
      {
         visit(theItem);
      }
      else
      {
         inorder(visit, treePtr->getLeftChildPtr());
         visit(theItem);
         inorder(visit, treePtr->getRightChildPtr());
      }  // end if
   }  // end if
}  // end inorder
```

Base Case: empty subtree

Base Case: binary node is a leaf



BinaryNodeTree.cpp