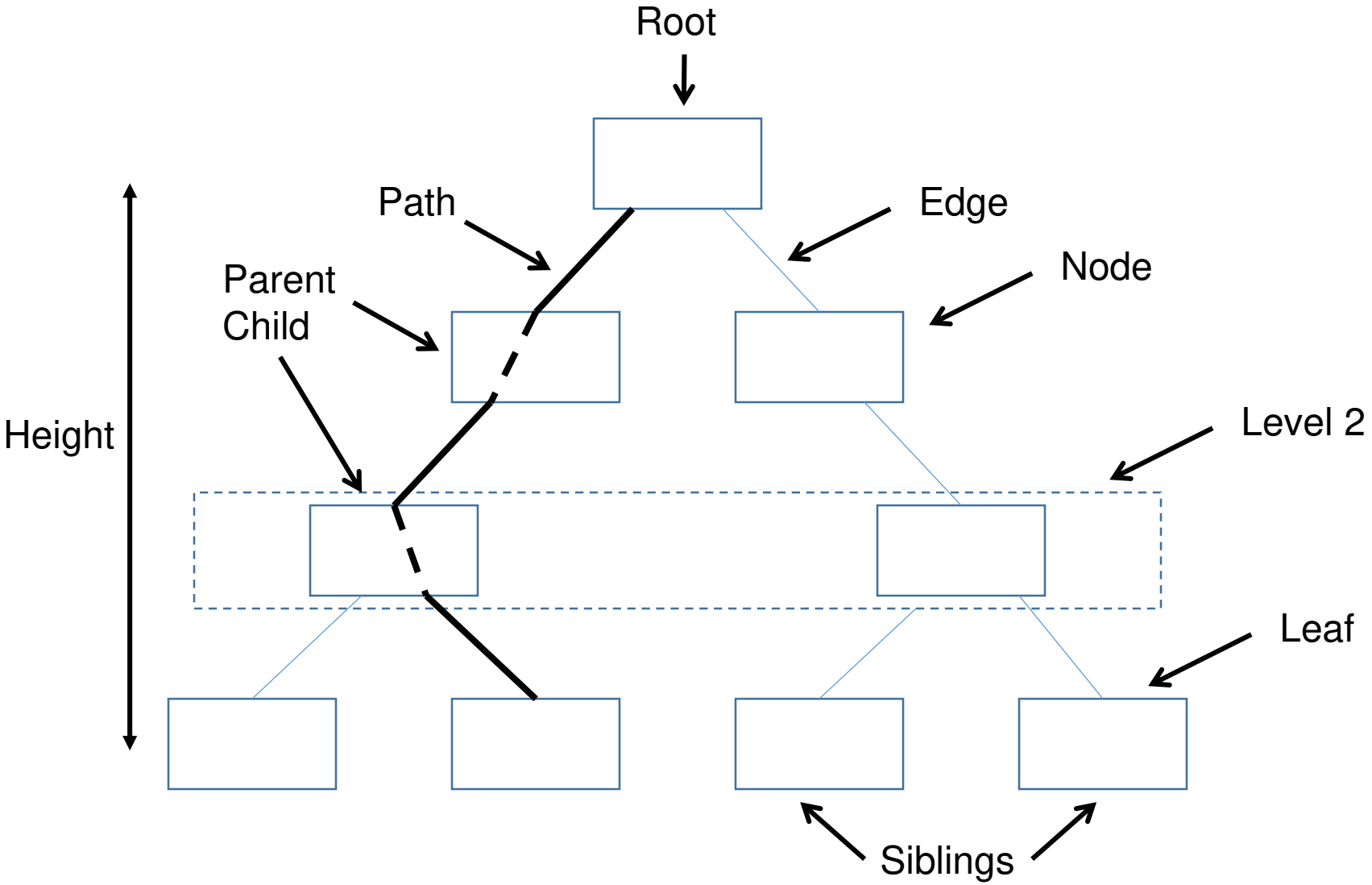
Trees

- Tree nomenclature
- Implementation strategies
- Traversals
  - Depth-first
  - Breadth-first
- Implementing binary trees
- Reading: L&C 10.1 – 10.7

DRAFT

Tree Nomenclature



Tree Nomenclature

- A *tree* is a non-linear structure in which elements are organized into a hierarchy
- A tree has levels of *nodes* connected by *edges*
- Each node is at a *level* in the tree
- The *root* node is the one node at the top level
- Nodes are *children* of nodes at higher levels
- Nodes with the same parent node are *siblings*
- A *leaf* is a node with no children

Tree Nomenclature

- A *path* exists from the root to any node or leaf
- A node is the *ancestor* of another node if it is on the path between the root and the other node
- A node that can be reached along a path away from the root is a descendant
- The level of a node is the length of the path (number of edges) from the root to the node
- The height of a tree is the length of the longest path from the root to a leaf

Tree Nomenclature

- A tree is considered to be *balanced* if all of the leaves are at the same level or within one level of each other
- A tree is considered to be *complete* if it is balanced and all the leaves on the bottom level are on the left
- A tree is considered *full* if all leaves of the tree are at the same level and every node is either a leaf or has exactly n children
- The height of a balanced, complete, or full tree that contains N elements is $\log_n N$

Tree Nomenclature

- The *order* of a tree is an important characteristic
- It is based on the maximum number of children a node can have
  - There is no limit in a *general tree*
  - An *n-ary tree* has a limit of n children per node
  - A *binary tree* has exactly two children per node
- The maze in Project 4 will be a "tri-nary" tree
- Binary trees are often useful, so we'll concentrate on them in the rest of this course

Implementation Strategies

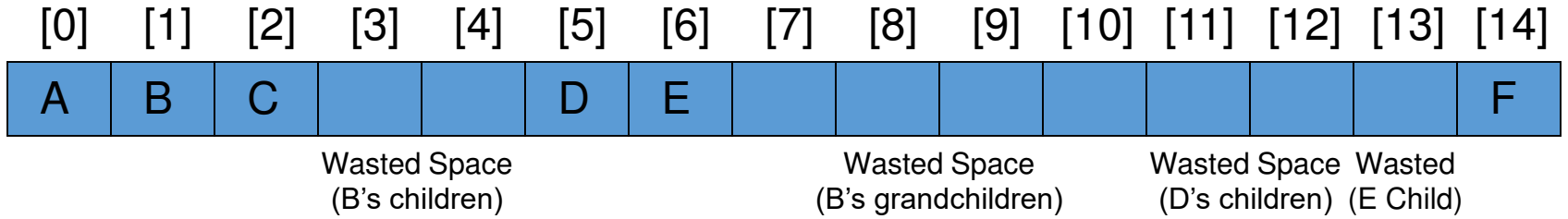
- The *computational strategy* for an array
- In a binary tree, for any element stored in the array in position n, we consider:
  - its left child to be stored in position: 2*n + 1
  - its right child to be stored in position: 2*(n + 1)
- This is a simple numerical index mapping and can be managed by adding capacity as needed
- Its disadvantage is that it may waste memory
- If the tree is not complete or nearly complete, the array may have many empty elements

Implementation Strategies

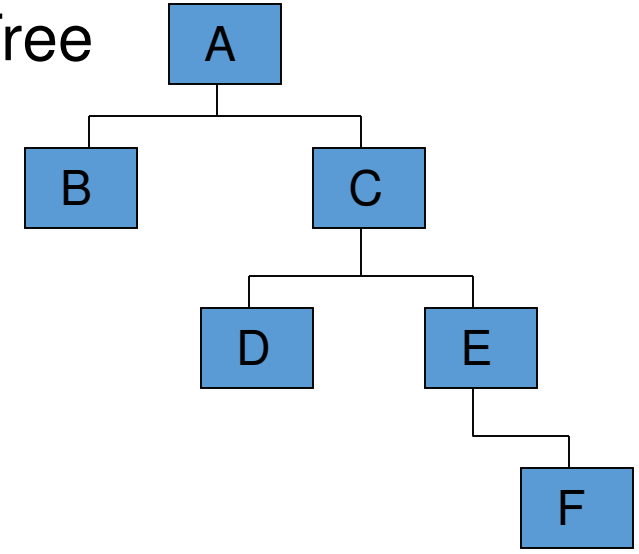
- The *simulated link strategy* for an array
- In a binary tree, each element of the array is an object with a reference to a data element and an int index for each of its two children
- A new child is always added to the end of the contiguous storage area in the array to avoid wasting space
- However, there is increased overhead to remove an element from the array (to shift the remaining elements and alter index values as required)

Implementation Strategies

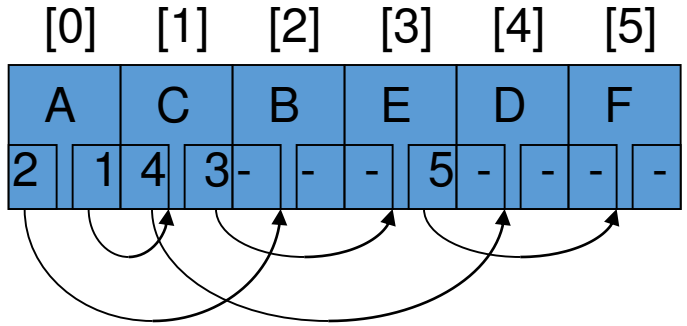
Computational Strategy in an Array



Tree

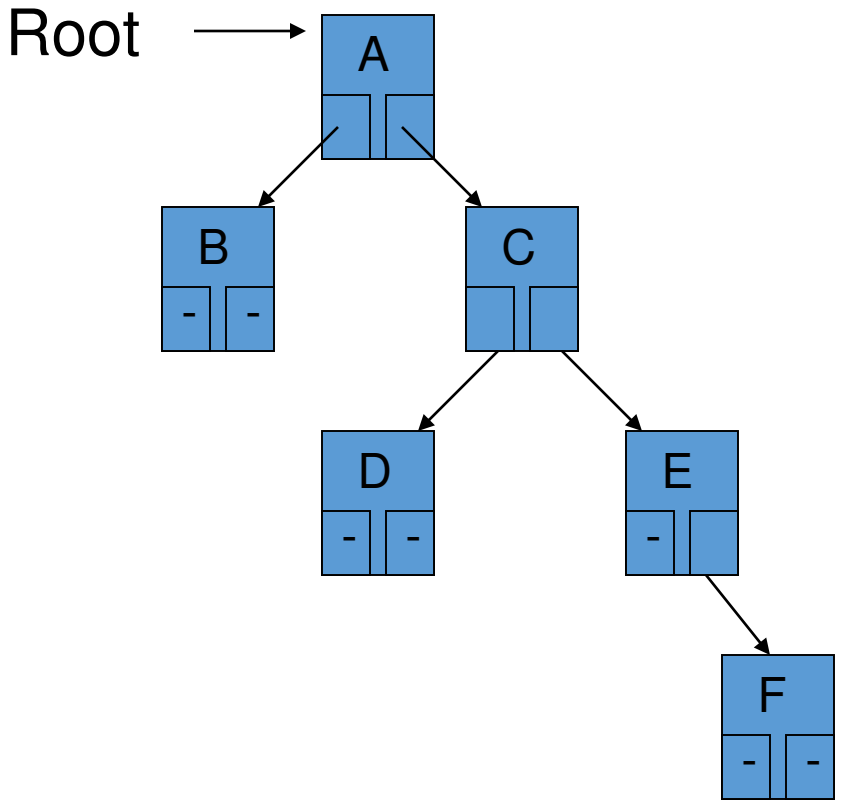


Simulated Link Strategy in an Array (added in the order A, C, B, E, D, F)



Implementation Strategies

- For a binary tree, the *linked strategy* uses a node class containing a reference to the data and a left and a right reference to two child nodes

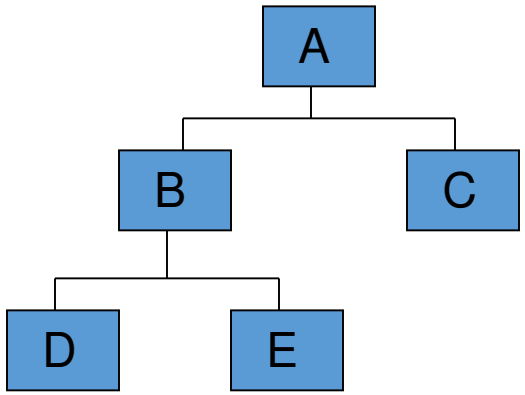


Traversals

- Types of traversals
  - Pre-order (Depth first)
    - Visit node, traverse left child, traverse right child
  - In-order (Depth first)
    - Traverse left child, visit node, traverse right child
  - Post-Order (Depth first)
    - Traverse left child, traverse right child, visit node
  - Level-order (Breadth first)
    - Visit all the nodes at each level, one level at a time

Traversals

Tree



Pre-order traversal would give:
A, B, D, E, C

In-order traversal would give:
D, B, E, A, C

Post-order traversal would give:
D, E, B, C, A

Level-order Traversal would give:
A, B, C, D, E

Level-order Traversal

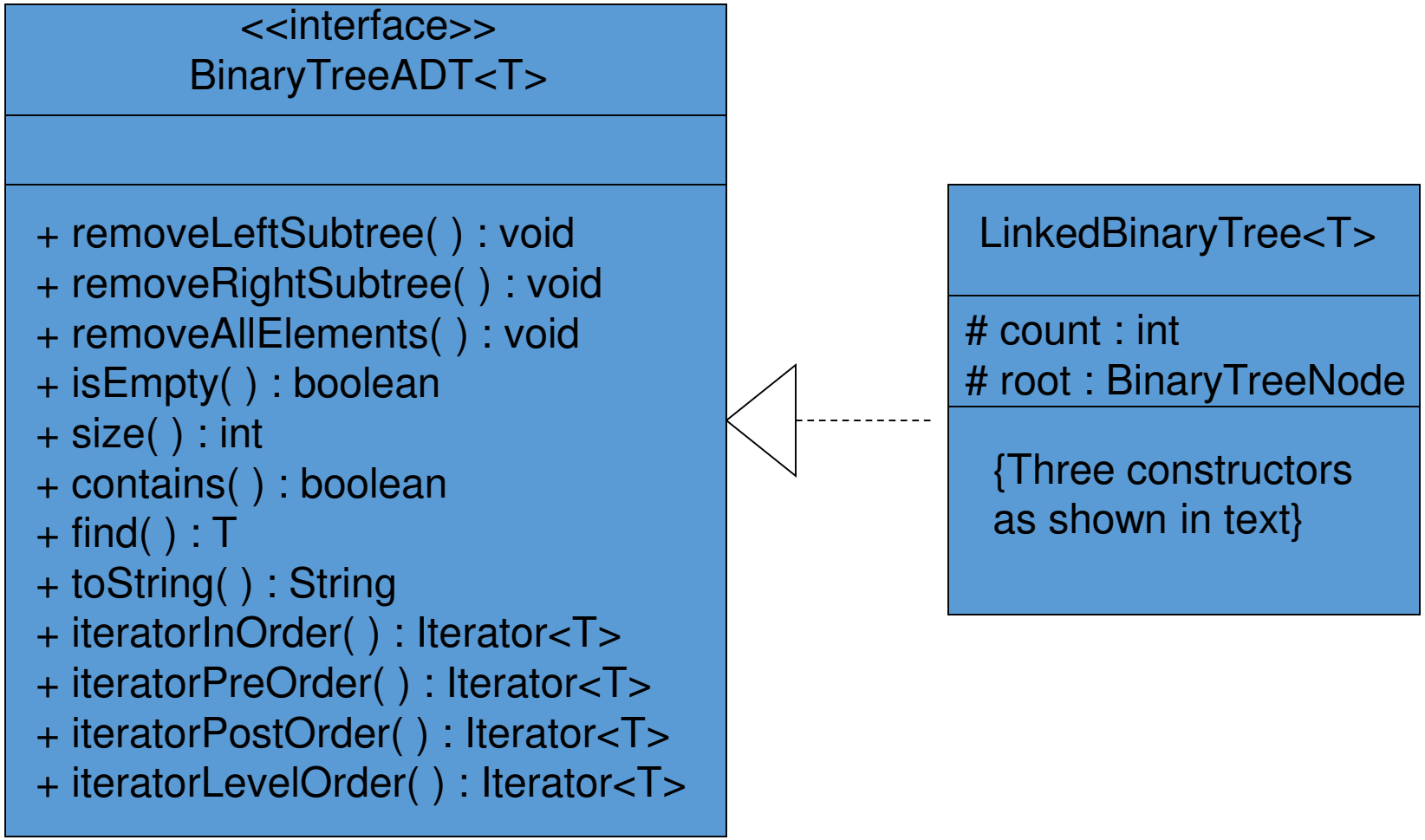
- A possible algorithm for level-order traversal

```
Create a queue called nodes
Create an unordered list called results
Enqueue a reference to the root node onto the nodes queue
While the nodes queue is not empty
  Dequeue the first element
  If it is not null
     add it to the rear of the results list
     Enqueue the children of the element on the nodes queue
  Else
     Add null to the rear of the results list
Return an iterator for the results list
```

Implementing Binary Trees

- A possible interface definition is provided that can be used on any binary tree regardless of the purpose of the tree
- Note: There is no method for adding an element or removing an element yet
- Until we know more about the purpose of the tree, those operations can't be defined
- We will use a less general (child) interface for a *binary search tree* and a *heap* later

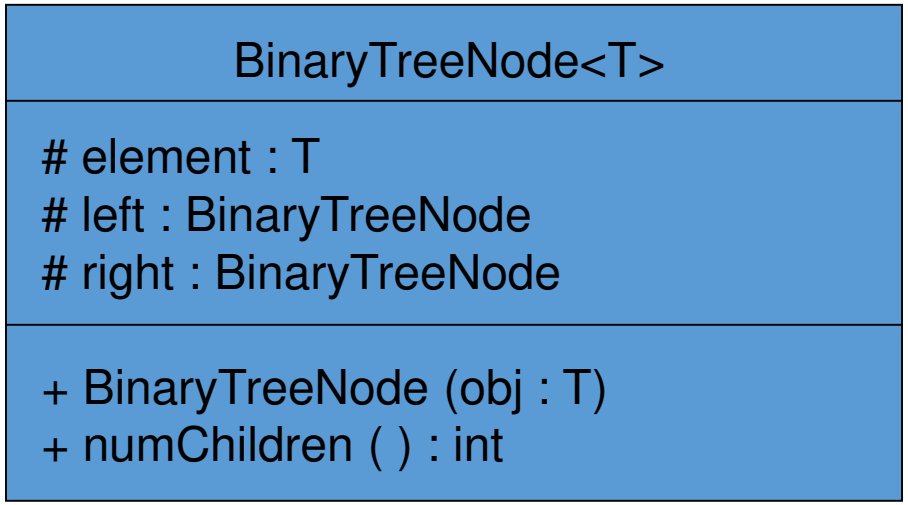
Implementing Binary Trees



Note: toString is missing in L&C Fig 9.9

Implementing Binary Trees

- Here we use a BinaryTreeNode in a linked strategy for our implementation



- Note: L&C code allows "package" access to the BinaryTreeNode attributes - not accessor methods as would be better O-O practice

Implementing Binary Trees

- Three constructors for convenience:
  - One to instantiate an empty tree
  - One to instantiate a tree with one root node
  - One to instantiate a tree with a root node and left and right child nodes from existing trees
- In normal methods for processing a binary tree, it is useful to use recursive algorithms

Implementing Binary Trees

- BinaryTreeNode method to get number of children

```
public int numChildren()
{
  int children = 0;
  if (left != null)
    children = 1 + left.numChildren();
  if (right != null)
    children += 1 + right.numChildren();
  return children;
}
```

- Note: Usual strategy of keeping a count attribute doesn't work well since if we add a child to a node, we need to go back to all parent nodes to update the count attribute in each of them

Implementing Binary Trees

- LinkedBinaryTree remove left sub-tree method

```
public void removeLeftSubtree()
{ // Note: uses methods instead of package access
  if (root.getLeft() != null)
    count = count - root.getLeft().numChildren() - 1;
  root.setLeft(null);    // creates garbage!
}
```

- The Java garbage collection approach saves coding effort here
- In languages like C++, the last line would be a “memory leak”
- This method would be much more complex to implement - needing to release objects’ memory

Implementing Binary Trees

- LinkedBinaryTree method to find target

```
private BinaryTreeNode<T> findagain (T target,
   BinaryTreeNode<T> next)
{ // Note: uses methods instead of package access
  if (next == null)
    return null;
  if (next.getElement().equals(target))
    return next;
  BinaryTreeNode<T> temp = findagain(target,
        next.getLeft());
  if (temp == null)
    temp = findagain(target, next.getRight());
  return temp;
}
```

Implementing Binary Trees

- LinkedBinaryTree method iteratorInOrder()

```
public Iterator<T> iteratorInOrder() {
  ArrayList<T> list = new ArrayList<T>();
  inOrder (root, list);
  return list.iterator();
}
private void inorder(BinaryTreeNode<T> node,
   ArrayList<T> list)
{  // Note: uses methods instead of package access
  if (node != null) {
    inorder(node.getLeft(), list);
    list.add(node.getElement());
    inorder(node.getRight(), list);
}
```