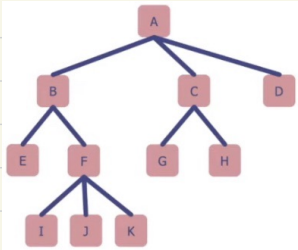
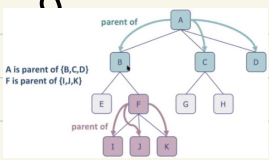
Video 1. Terminology

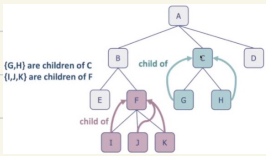
tree – abstract model of hierarchical structure



nodes + edges

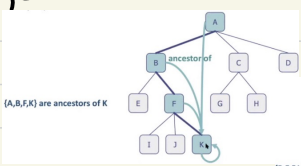
edge = hierarchical, parent-child relation

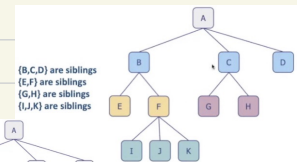


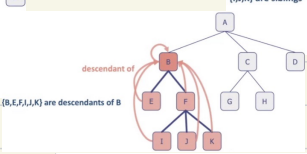


Relationships between nodes:

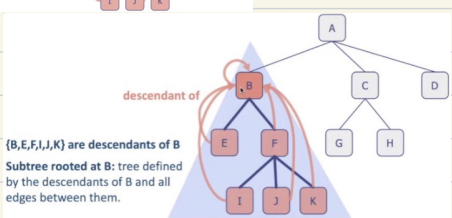
- Parent
- Child
- Sibling
- Ancestor
- Descendant

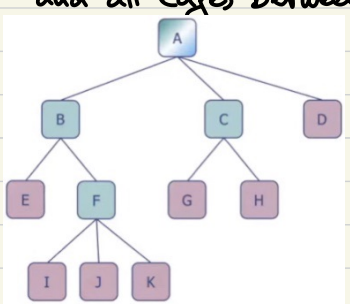






subtree rooted at a node
- tree defined by the descendants of the node and all edges between them

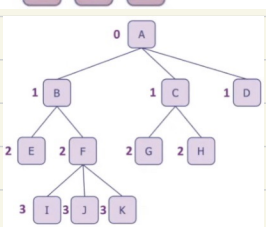




root – node without parent

internal nodes – nodes with child

external nodes (leaves) – without child



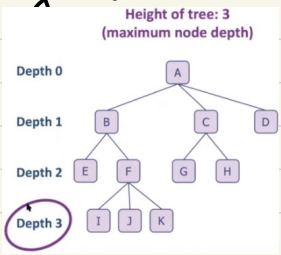

```
/** Position ensures that nodes are not accessible from the outside. **/
public interface Position<E> {
  E getElement() throws IllegalStateException;
}

/** Interface for a tree where nodes can have an arbitrary number of children. **/
public interface Tree<E> extends Iterable<E> {
  Position<E> root();                                          // Get root position.
  Position<E> parent(Position<E> p) throws IllegalArgumentException; // Get parent of position p.
  Iterable<Position<E>> children(Position<E> p)
        throws IllegalArgumentException;                       // Get children of p.
  int numChildren(Position<E> p) throws IllegalArgumentException;    // Get number children of p.
  boolean isInternal(Position<E> p) throws IllegalArgumentException; // Check if p is internal.
  boolean isExternal(Position<E> p) throws IllegalArgumentException; // Check if p is external.
  boolean isRoot(Position<E> p) throws IllegalArgumentException;     // Check if p is root.
  int size();                                                  // Get size.
  boolean isEmpty();                                           // Check if tree is empty.
  Iterator<E> iterator();                                      // Get element iterator.
  Iterable<Position<E>> positions();                           // Get position iterable.
}
```

depth of node:
ancestors – the node

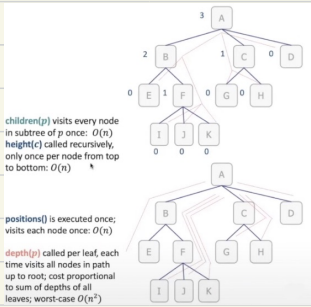
```
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

height of a tree – maximum depth



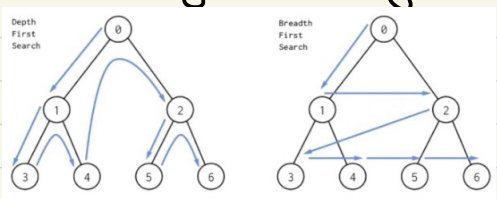

```
public int height(Position<E> p) {
  int h = 0; // base case (p is leaf)
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;
}

public int height() {
  int h = 0;
  for (Position<E> p : positions())
    if (isExternal(p))
      h = Math.max(h,depth(p));
  return h;
}
```



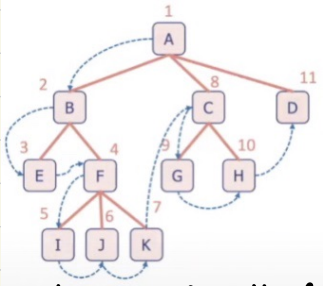
Video 2. Traversals

traversals – systematic ways of visiting all nodes in a tree



depth-first – first traverse all the way down to the leaves before return

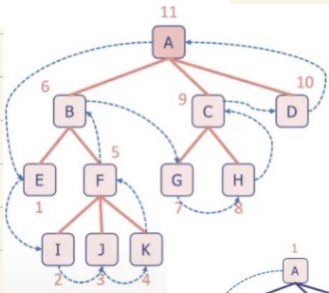
breadth-first – traverse the nodes of a tree level-wise



pre-order (depth-first)
node is visited before its children

```
public void preorder(Position<E> p) {
  visit(p);
  for (Position<E> c : children(p))
    preorder(c);
}
```

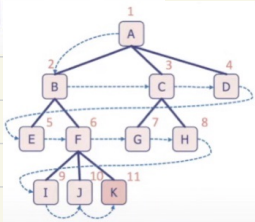
post-order (depth-first)
node is visited after its children



```
public void postorder(Position<E> p) {
  for (Position<E> c : children(p))
    postorder(c);
  visit(p);
}
```

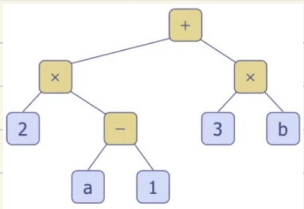
breadth-first
visit nodes per level

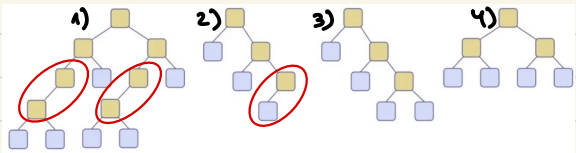
```
public void breadthfirst(Position<E> p) {
  Queue<Position<E>> q = new Queue();
  q.enqueue(p);
  while (!q.isEmpty()) {
    Position<E> p = q.dequeue();
    visit(p);
    for (Position<E> c : children(p))
      q.enqueue(c);
  }
}
```



Video 3. Binary trees

- each internal node has at most 2 children
- children of a node – ordered pair (left child, right child)



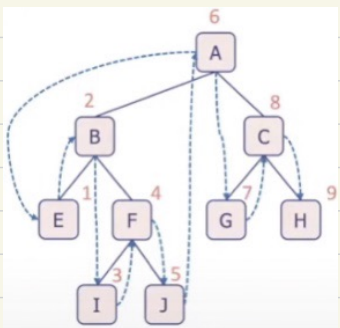


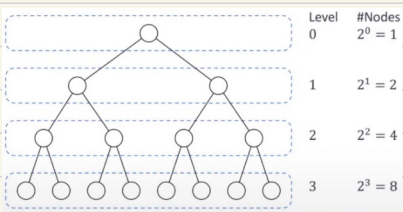
3,4 – proper 1,2-improper either 0 or 2 children

in-order (depth-first) - binary tree traversal

left child, node, right child

```
public void inorder(Position<E> p) {
  if(left(p) != null) inorder(left(p));
  visit(p);
  if(right(p) != null) inorder(right(p));
}
```





max number of nodes in a binary tree with height h:

$n = \sum_{d=0}^{h} 2^d = 2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$