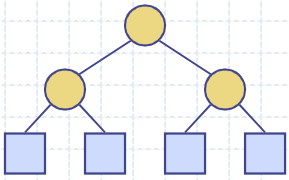
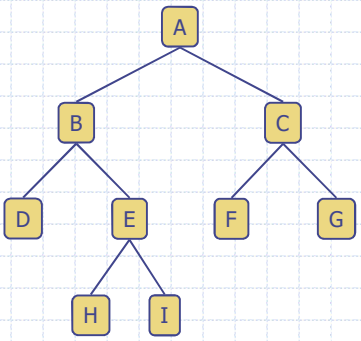
Binary Trees



Binary Trees

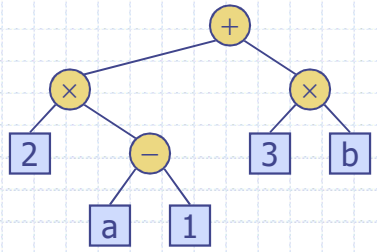
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Arithmetic Expression Tree

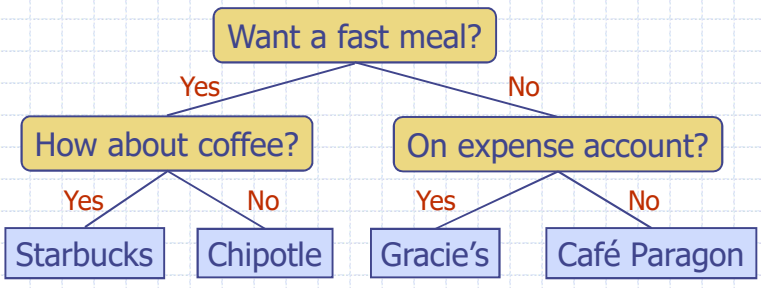
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



3

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

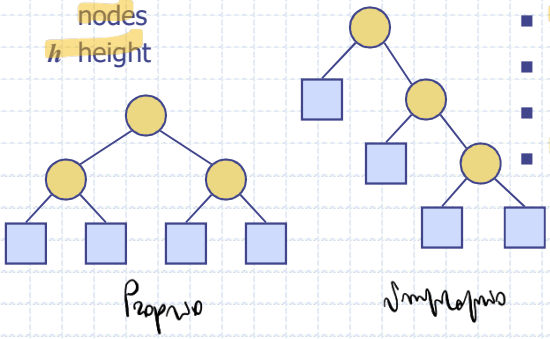


4

e = numero delle foglie

Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height
- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$



Proprio

Improprio

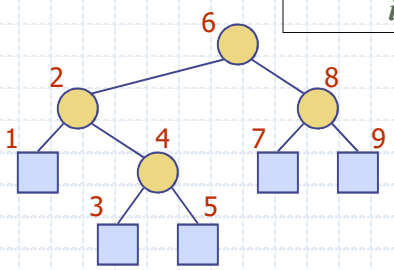
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- The above methods return null when there is no left, right, or sibling of p, respectively

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

Algorithm *inOrder*(*v*)
  if *left* (*v*) ≠ **null**
    inOrder (*left* (*v*))
  visit(*v*)
  if *right*(*v*) ≠ **null**
    inOrder (*right* (*v*))

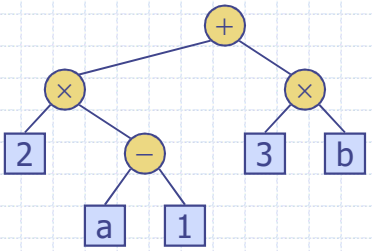


7

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

Algorithm *printExpression*(*v*)
  if *left* (*v*) ≠ **null**
    print("(")
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *right*(*v*) ≠ **null**
    inOrder (*right*(*v*))
    print (")")

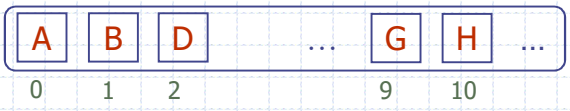


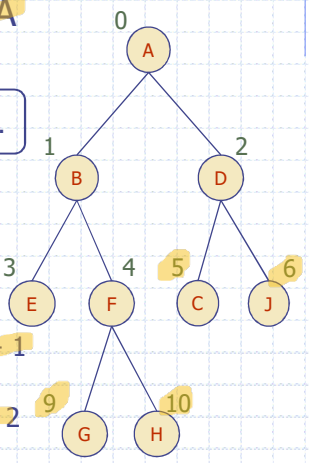
((2 × (a − 1)) + (3 × b))

8

Array-Based Representation of Binary Trees

- Nodes are stored in an array A





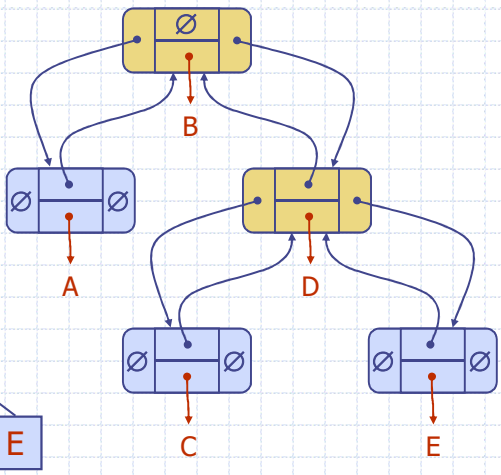
Node v is stored at A[rank(v)]

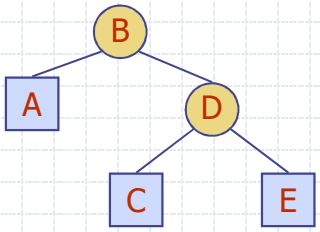
- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 2

9

Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





10

Java Implementation (Binary Tree)

```
public class LinkedBinaryTree_mod<E> implements BinaryTree<E> {

 //---------------- nested Node class ----------------
 /** Nested static class for a binary tree node. */
 protected static class Node<E> implements Position<E> {
   private E element;          // an element stored at this node
   private Node<E> parent;     // a reference to the parent node (if any)
   private Node<E> left;       // a reference to the left child (if any)
   private Node<E> right;      // a reference to the right child (if any)

   /**
    * Constructs a node with the given element and neighbors.
    *
    * @param e  the element to be stored
    * @param above        reference to a parent node
    * @param leftChild    reference to a left child node
    * @param rightChild   reference to a right child node
    */
   public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
     element = e;
     parent = above;
     left = leftChild;
     right = rightChild;
   }

   // accessor methods
   public E getElement() { return element; }
   public Node<E> getParent() { return parent; }
   public Node<E> getLeft() { return left; }
   public Node<E> getRight() { return right; }

   // update methods
   public void setElement(E e) { element = e; }
   public void setParent(Node<E> parentNode) { parent = parentNode; }
   public void setLeft(Node<E> leftChild) { left = leftChild; }
   public void setRight(Node<E> rightChild) { right = rightChild; }
 } //----------- end of nested Node class -----------

 /** Factory function to create a new node storing element e. */
 protected Node<E> createNode(E e, Node<E> parent,
                              Node<E> left, Node<E> right) {
   return new Node<E>(e, parent, left, right);
 }
```

Implementation of the single node of the **Binary Tree** using a nested class **Node**

Java Implementation (Binary Tree)

```
public class LinkedBinaryTree_mod<E> implements BinaryTree<E> {
// LinkedBinaryTree instance variables
 /** The root of the binary tree */
 protected Node<E> root = null;     // root of the tree

 private int size = 0;              // number of nodes in the tree

 /** Constructs an empty binary tree. */
 public LinkedBinaryTree_mod() { }      // constructs an empty binary tree

 /* implements specific methods added by the interface Binary Tree */
 @Override
 public Position<E> left(Position<E> p) throws IllegalArgumentException {
   Node<E> node = validate(p);
   return node.getLeft();
 }
 @Override
 public Position<E> right(Position<E> p) throws IllegalArgumentException {
   Node<E> node = validate(p);
   return node.getRight();
 }
 @Override
 public Position<E> sibling(Position<E> p) {
   Position<E> parent = parent(p);
   if (parent == null) return null;                  // p must be the root
   if (p == left(parent))                            // p is a left child
     return right(parent);                           // (right child might be null)
   else                                              // p is a right child
     return left(parent);                            // (left child might be null)
   }

 public Position<E> addRoot(E e) throws IllegalStateException {
   if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
   root = createNode(e, null, null, null);
   size = 1;
   return root;
 }
 public Position<E> addLeft(Position<E> p, E e)
                            throws IllegalArgumentException {
   Node<E> parent = validate(p);
   if (parent.getLeft() != null)
     throw new IllegalArgumentException("p already has a left child");
   Node<E> child = createNode(e, parent, null, null);
   parent.setLeft(child);
   size++;
   return child;
 }
 public Position<E> addRight(Position<E> p, E e)
                             throws IllegalArgumentException {
   Node<E> parent = validate(p);
   if (parent.getRight() != null)
     throw new IllegalArgumentException("p already has a right child");
   Node<E> child = createNode(e, parent, null, null);
   parent.setRight(child);
   size++;
   return child;
 }
```

Java Implementation (Binary Tree)

```
public Iterable<Position<E>> children(Position<E> p) {
  ArrayList<Position<E>> snapshot = new ArrayList<>(2);    // max capacity of 2
  if (left(p) != null)
    snapshot.add(left(p));
  if (right(p) != null)
    snapshot.add(right(p));
  return snapshot;
}

@Override
public boolean isEmpty() { return size() == 0; }

private void inorderSubtree(Position<E> p, ArrayList<Position<E>> snapshot) {
  if (left(p) != null)
    inorderSubtree(left(p), snapshot);
  snapshot.add(p);
  if (right(p) != null)
    inorderSubtree(right(p), snapshot);
}

public Iterable<Position<E>> inorder() {
  ArrayList<Position<E>> snapshot = new ArrayList<>();
  if (!isEmpty())
    inorderSubtree(root(), snapshot);   // fill the snapshot recursively
  return snapshot;
}

@Override
public Iterable<Position<E>> positions() {
  return inorder();
}

 //---------------- nested ElementIterator class ----------------
/* This class adapts the iteration produced by positions() to return elements. */
private class ElementIterator implements Iterator<E> {
  Iterator<Position<E>> posIterator = positions().iterator();
  public boolean hasNext() { return posIterator.hasNext(); }
  public E next() { return posIterator.next().getElement(); } // return element!
  public void remove() { posIterator.remove(); }
}

/**
 * Returns an iterator of the elements stored in the tree.
 * @return iterator of the tree's elements
 */
@Override
public Iterator<E> iterator() { return new ElementIterator(); }
```

Implementation of a method **iterator**() of **Binary Tree**

Java Implementation (Visit Binary Tree)

```
public  class VisitBinTree<E>  extends LinkedBinaryTree_mod<E> {

 // constructor

 public VisitBinTree()  { super();}      // constructs an empty generic tree

public void  preorder(Position<E> p) {
     System.out.println(p.getElement()); // for preorder, we visit p before exploring subtrees
      if (left(p) != null)
        preorder(left(p));
      if (right(p) != null)
        preorder(right(p));
     return ;
  }

 public void  postorder(Position<E> p) {

      if (left(p) != null)
        postorder(left(p));
      if (right(p) != null)
        postorder(right(p));
      System.out.println(p.getElement()); // for postrder, we exploring subtrees before visit p
     return ;
  }

public void breadthfirst() {
    if (!isEmpty()) {
      Queue<Position<E>> queue = new LinkedQueue<>();
      queue.enqueue(root());                  // start with the root
      while (!queue.isEmpty()) {
        Position<E> p = queue.dequeue();      // remove from front of the queue
        System.out.println(p.getElement());         // visit  this position
        // add childrens to back of queue
        if (left(p) != null)
        queue.enqueue(left(p));
         if (right(p) != null)
        queue.enqueue(right(p));
      }
    }
    return;
    }


} //------------ end of VisitGenTree class ------------
```

Extend the class **Binary Tree** to implement different traversal algorithms

Different visits of a Binary Tree

```
public class TestVisitBinTre

 static VisitBinTree btree=

public static void addSons(
    btree.addLeft(parent,k);
    btree.addRight(parent,k+
}

 public static void main(St
{
 int i;


System.out.println("Avvio r

btree.addRoot(0);
Position root=btree.root();
addSons(root,1);
addSons(btree.left(root),3);
addSons(btree.right(root),5
System.out.println("Dimensio

System.out.println("Stampa nodi albero in ordine");
Iterable<Position<Integer>> it1=btree.inorder();
for (Position p: it1)
 System.out.println(p.getElement());

System.out.println("Stampa nodi albero in preordine");
btree.preorder(root);

System.out.println("Stampa nodi albero in postordine");
btree.postorder(root);

System.out.println("Stampa nodi albero in ampiezza");
btree.breadthfirst();

}
}
```

```
algoritmi miei — bash — 78×37
Last login: Mon Apr  1 14:55:25 on ttys000
iMac-di-Rocco:algoritmi miei roccoaversa$ java TestVisitBinTree
Avvio riempimento albero
Dimensione btree= 7
Stampa nodi albero in ordine
3
1
4
0
5
2
6
Stampa nodi albero in preordine
0
1
3
4
2
5
6
Stampa nodi albero in postordine
3
4
1
5
6
2
0
Stampa nodi albero in ampiezza
0
1
2
3
4
5
6
iMac-di-Rocco:algoritmi miei roccoaversa$
```